Spring框架参考文档

作者

杆 约翰逊 , 于尔根· Hoeller , 基思 唐纳德 , 科林 Sampaleanu , 罗伯 Harrop , 托马斯 里斯贝里 , Alef Arendsen , 达伦 戴维森 , 他 Kopylenko , 马克 波拉克 , 蒂埃里 Templier , 欧文 Vervaet , 波西亚 董建华 , 本 硬朗 , 艾德里安 Colyer , 约翰 刘易斯 , 损失 Leau , 马克 费舍尔 , 山姆 Brannen , Ramnivas Laddad , 罗本 Poutsma , 克里斯 梁 , 塔里克· Abedrabbo , 安迪 克莱门特 , 戴夫 Syer , 奥利弗 Gierke , 雷森 Stoyanchev , 菲利普 韦伯

3 2 2释放

版权©2004 - 2013

本文件的复印件可以供你自己使用和 分配给别人,只要你不收取任何费用这样的 册,进一步提供,每个副本包含这个版权 注意,是否分布在打印或电子。

表的内容

即Spring框架的概述
1。 介绍Spring框架
1.1。 依赖注入和控制反转
1.2。 模块
1.2.1。 核心容器
1.2.2。 数据访问/集成
1.2.3。 web
1.2.4。 AOP和仪表
1 2 5。 测试
1.3。 使用场景
1.3.1。 依赖关系管理和命名约定
Spring依赖和依靠弹簧
Maven依赖管理
艾薇依赖管理
1.3.2。 日志
不使用通用日志
使用SLF4J
使用Log4j
二世。 什么是新的在春季3
2。 新特性和增强功能在Spring框架3.0
2.1。 Java 5
2.2。 改进文档
2.3。 新的文章和教程
2.4。 新模块的组织和构建系统
2.5。 概述的新特性
2 5 1。 核心api更新Java 5
2 5 2。 弹簧表达式语言
2 5 3。 这个控制反转(IoC)容器
基于Java bean的元数据
元数据定义bean组件中
专 家 进 阶 入 门 全文翻译
翻译级别
2 5 6。 web层
综合REST支持
@MVC添加
2 5 7。 声明式模型验证
2 5 8。 早期支持Java EE 6
2 5 9。 支持嵌入式数据库
3。 新特性和增强功能在Spring框架3.1
3.1。 缓存抽象
3.2。 Bean定义概要文件
3.3。 环境抽象
3.4。 PropertySource抽象
3.5。 代码等价的Spring的XML名称空间
3.6。 支持Hibernate 4. x
3.7。 还是和TestContext框架支持@ configuration类和bean 定义概要文件
3.8。 c:名称空间更简洁的构造函数注入
3.9。 支持对非标准JavaBeans注射 setter
3.10。 支持基于Servlet的配置Servlet 3 容器
3.11。 3 MultipartResolver支持Servlet

3.12。 JPA会引导没有 persistence . xml
3.13。 新HandlerMethod-based支持类注释的控制器 处理
3.14。 “消费”和“生产”条件 @RequestMapping
3.15。 Flash属性和 RedirectAttributes
3.16。 URI模板变量增强
3.17。 @Valid 在 @RequestBody 控制器方法参数
3.18。 @RequestPart 注释 控制器方法参数
3.19。 UriComponentsBuilder 和 UriComponents
4。 新特性和增强功能在Spring框架3.2
4.1。 支持Servlet 3异步请求处理的基础
4.2。 Spring MVC的测试框架
4.3。 内容协商改进
4.4。 @ControllerAdvice 注释
4.5。 矩阵变量
4.6。 抽象基类基于代码的Servlet 3 +容器 初始化
4.7。 ResponseEntityExceptionHandler 类
4.8。 对泛型的支持的 RestTemplate 在 @RequestBody 参数
4.9。 杰克逊JSON 2和相关的改进
4.10。 瓷砖3
4.11。 @RequestBody 改进
4.12。 HTTP补丁方法
4.13。 排除模式映射的拦截器
4.14。 使用注射点和元注释bean定义方法
4.15。 最初支持JCache 0.5
4.16。 支持 @DateTimeFormat 没有 Joda时间
4.17。 日期和时间格式化全球
4.18。 新的测试功能
4.19。 整个框架的改进并发性
4.20。 新的基于gradle构建和搬到GitHub
4.21。 精制Java SE 7 / OpenJDK 7支持
三世。 核心技术

Search Documentation 所有

5.1。 介绍Spring IoC容器和豆类
5.2。 集装箱概述
5 2 1。 配置元数据
5.2.2。 实例化一个集装箱
组成的基于xml的配置元数据
5 2 3。 使用集装箱
5.3。 Bean概述
5 3 1。 命名bean
混淆一个bean外的bean定义
5 3 2。 实例化bean
与构造函数实例化
与静态工厂方法实例化
使用工厂方法实例化一个实例
5.4。 依赖性
5.4.1之前。 依赖注入
无参依赖注入
setter建立依赖注入
依赖解析过程
依赖注入的例子
5 4 2。 依赖关系和配置细节
直值(原语, 字符串 ,所以)
引用其他豆类(合作者)
内豆
集合
空,空字符串值
XML快捷与p名称空间
XML快捷与c名称空间
复合属性名
第5.4.3。 使用 取决于
5 4 4。 延迟初始化的 bean
5 4 5。 自动装配的合作者
局限性和缺点的自动装配
不包括一个bean从自动装配
5 4 6。 方法注入
查找方法注入
任意方法替代
5.5。 Bean范围
5.5.1。 singleton范围
5 5 2。 原型范围
5 5 3。 单例bean与原型bean的依赖项
5 5 4。 请求、会话和全球会话作用域
初始web配置
请求范围
会话范围

全球会话范围
范围bean作为依赖项
5 5 5。 定制范围
创建一个自定义的范围
使用一个自定义的范围
5.6。 定制一个bean的性质
5.6.1。 生命周期回调
初始化回调
破坏回调
默认的初始化和销毁方法
结合生命周期机制
启动和关闭回调
关闭Spring IoC容器优雅地在非web 应用
5 6 2。 ApplicationContextAware 和 BeanNameAware
5 6 3。 其他 意识到 接口
5.7。 Bean定义继承
5.8。 容器扩展点
5 8 1。 定制bean使用 BeanPostProcessor
例如:你好世界, BeanPostProcessor 风格
示例: RequiredAnnotationBeanPostProcessor
5 8 2。 自定义配置元数据与一个 BeanFactoryPostProcessor
示例: 来完成
示例: PropertyOverrideConfigurer
5 8 3。 自定义实例化逻辑与一个 FactoryBean
5.9。 基于注解的容器配置
5 - 9 - 1。 @ required
5 9 2。 @ autowired
5 9 3。 基于注解的自动装配与限定词微调
5 9 4。 CustomAutowireConfigurer
5 9 5。 @ resource
5 9 6。 @PostConstruct 和 @PreDestroy
5.10。 类路径扫描和管理组件
5 10 1。 component 和进一步的刻板印象 注释
5 10 2。 自动检测和注册的bean类 定义
5 10 3。 使用过滤器来定制扫描
5 10 4。 元数据定义bean组件内
5 10 5。 命名个组件
5 10 6。 一个组件提供的范围
5 10 7。 元数据与注释提供限定符
5.11。 使用JSR 330标准注释
5 11 1。 依赖注入与 @ inject 和 @ named
5 11 2。 @ named :一个标准相当于 component 注释
5 11 3。 的标准方法的局限性
5.12。 java容器配置
5 12 1。 基本概念: @ bean 和 @ configuration
5 12 2。 Spring容器实例化使用 所
结构简单
建筑容器以编程方式使用 寄存器(类< ? >...)
使组件扫描与 扫描(String.....)
支持web应用程序时 AnnotationConfigWebApplicationContext
5 12 3。 使用 @ bean 注释
声明一个bean
接收生命周期回调
bean指定范围
定制bean命名
Bean混淆
5 12 4。 使用 @ configuration 注释
注入国米bean依赖
查找方法注入
进一步的信息关于基于java的配置工作 内部
5 12 5。 构成基于java的配置
使用 @ import 注释
结合Java和XML配置
5.13。 登记 LoadTimeWeaver
5.14。 附加功能的 ApplicationContext
5 14 1。 国际化使用 MessageSource
5 14 2。 标准和定制的事件
5 14 3。 方便的访问底层的资源
5 14 4。 方便 ApplicationContext 为web应用程序的实例化
5 14 5。 部署一个弹簧ApplicationContext作为J2EE RAR文件
5.15。 BeanFactory
5 15 1。 BeanFactory 或 ApplicationContext 吗?
5 15 2。 胶水代码和邪恶的单例
6。 资源
6.1。 介绍
6.2。 这个 资源 接口

6.3。 内置 资源 实现
6 3 1。 UrlResource
6 3 2。 ClassPathResource
6.3.3这时。 FileSystemResource
6.3.4得到。 ServletContextResource
6 3 5。 InputStreamResource
6 3 6。 ByteArrayResource
6.4。 这个 ResourceLoader
6.5。 这个 ResourceLoaderAware 接口
6.6。 资源 作为依赖项
6.7。 应用程序上下文和 资源 路径
6 7 1。 构建应用程序上下文
构建 ClassPathXmlApplicationContext 实例——快捷键
6 7 2。 通配符在应用程序上下文构造函数资源路径
ant是基于模式
这个 classpath *: 前缀
其他笔记有关通配符
6 7 3。 FileSystemResource 警告
7。 验证、数据绑定、类型转换
7.1。 介绍
7.2。 验证使用Spring的 验证器 接口
7.3。 解决代码错误消息
7.4。 Bean操纵和 BeanWrapper
7 4 1。 设置和获取基本和嵌套的属性
7.4.2。 内置 属性编辑器 实现
注册附加自定义 PropertyEditors
7.5。 弹簧3类型转换
7.5.1。 转换器SPI
7.5.2。 ConverterFactory
7 5 3。 GenericConverter
ConditionalGenericConverter
7 5 4。 ConversionService API
7.5.5。 配置一个ConversionService
7 5 6。 使用ConversionService编程
7.6。 弹簧3字段格式
7 6 1。 格式化器SPI
之前。 注解驱动的格式
格式注释API
7 6 3。 FormatterRegistry SPI
7 6 4。 FormatterRegistrar SPI
7 6 5。 配置格式在Spring MVC
7.7。 配置一个全球日期和时间格式
7.8。 弹簧3验证
7 8 1。 概述了jsr - 303 API Bean验证
7 8 2。 配置一个Bean验证实现
注入一个验证器
配置自定义约束
额外的配置选项
7 8 3。 配置DataBinder
7 8 4。 Spring MVC 3验证
触发controller输入验证
配置一个验证器使用Spring MVC
配置一个jsr - 303验证器使用Spring MVC
8。 春天表达式语言(?)
8.1。 介绍
8.2。 特性概述
8.3。 表达式求值的表达式使用Spring接口
夹带了本条件8.3.1。 EvaluationContext接口的
类型转换
8.4。 表达式支持定义bean定义
8 4 1。 基于XML配置
8 4 2。 基于注解的配置
8.5。 语言参考
8.5.1皆是如此。 字面表达式
8.5.2。 属性、数组、列表、地图、索引器
8 5 3。 内联列表
8 5 4。 阵列结构
8 5 5。 方法
8 5 6。 运营商
关系运算符
逻辑运算符
数学运算符
8 5 7。 分配
8 5 8。 类型
8 5 9。 构造函数
8 5 10。 变量

#这和# root变量
8 5 11。 功能
8 5 12。 Bean引用
8 5 13。 三元操作符(是以if - then - else)
8 5 14。 猫王运营商
8 5 15。 安全导航操作符
8 5 16。 集合选择
8 5 17。 收集投影
8 5 18。 表达式模板
8.6。 使用的类的实例
9。 面向方面的编程与弹簧
9.1。 介绍
9 1 1。 AOP概念
9 1 2。 Spring AOP功能和目标
9 1 3。 AOP代理
9.2。 @ aspectj支持
9 2 1。 启用@ aspectj支持
使@ aspectj支持Java配置
使@ aspectj支持使用XML配置
9 2 2。 声明一个方面
9 2 3。 声明一个切入点
支持切入点指示器
结合切入点表达式
分享共同切入点定义
例子
编写好的切入点
9 2 4。 声明建议
建议之前
回国后的建议
在投掷的建议
(最后)后建议
Around通知
建议参数
建议订购
9 2 5。 介绍
9 2 6。 方面的实例化模型
9 2 7。 例子
9.3。 基于AOP支持
如果。 声明一个方面
9 3 2。 声明一个切入点
9 3 3。 声明建议
建议之前
回国后的建议
在投掷的建议
(最后)后建议
Around通知
建议参数
建议订购
9 3 4。 介绍
9 3 5。 方面的实例化模型
9 3 6。 顾问
9 3 7。 例子
9.4。 选择要使用的AOP声明风格
9.4.1。 Spring AOP和AspectJ完整吗?
上装。 @ aspectj或XML为Spring AOP吗?
9.5。 混合方面类型
9.6。 代理机制
9 6 1。 理解AOP代理
9.7。 @ aspectj编程创建代理
9.8。 使用AspectJ和Spring应用程序
9 8 1。 使用AspectJ域对象依赖注入与 春天
单元测试 @Configurable 对象
使用多种应用程序上下文
9 8 2。 其他弹簧方面对AspectJ
9 8 3。 使用Spring IoC配置AspectJ方面
9 8 4。 装入时编织与AspectJ在Spring框架
第一个例子
方面
” meta - inf / aop xml ”
需要的库(jar)
Spring配置
特定于环境的配置
9.9。 进一步的资源
10。 Spring AOP api
10.1。 介绍
10.2。 切入点API在春天

10 2 1。 概念
10 2 2。 操作切入点
10 2 3。 AspectJ切入点表达式
10 2 4。 便利切入点实现
静态切入点
动态切入点
10 2 5。 切入点超类
10 2 6。 定制切入点
10.3。 建议API在春天
10 3 1。 建议生命周期
10 3 2。 建议类型在春天
拦截around通知
建议之前
抛出建议
回国后的建议
介绍的建议
10.4。 顾问API在春天
10.5。 使用ProxyFactoryBean创建AOP代理
10 5 1。 基本
10 5 2。 JavaBean属性
10 5 3。 JDK和cglib建立代理
10 5 4。 代理接口的
10 5 5。 代理类
10 5 6。 使用“全球”顾问
10.6。 简洁代理定义
10.7。 以编程方式创建AOP代理与ProxyFactory
10.8。 操纵建议对象
10.9。 使用“自动代理”设施
10 - 9 - 1。 火狐的一个插件bean定义
BeanNameAutoProxyCreator
DefaultAdvisorAutoProxyCreator
AbstractAdvisorAutoProxyCreator
10 9 2。 使用元数据驱动的汽车代理
10.10。 使用TargetSources
10 10 1。 热可切换目标来源
10 10 2。 池目标来源
10 10 3。 原型目标来源
10 10 4。 ThreadLocal 目标来源
10.11。 定义新的 建议 类型
10.12。 进一步的资源
11。 测试
11.1。 介绍弹簧测试
11.2。 单元测试
11 2 1。 模拟对象
环境
JNDI
Servlet API
Portlet API
11 2 2。 单元测试支持类
通用实用工具
Spring MVC
11.3。 集成测试
11 3 1。 概述
11 3 2。 目标的集成测试
上下文管理和缓存
依赖注入的测试夹具
事务管理
支持类的集成测试
11 3 3。 JDBC测试支持
11 3 4。 注释
弹簧测试注释
标准注释支持
弹簧JUnit测试注释
11 3 5。 春天还是和TestContext框架
关键抽象
上下文管理
依赖注入的测试夹具
测试请求和会话作用域的豆子
事务管理
还是和TestContext框架支持类
11 3 6。 Spring MVC的测试框架
服务器端测试
端REST测试
11 3 7。 宠物诊所的例子
11.4。 进一步的资源
四。数据访问

12。 事务管理
12.1。 介绍Spring框架事务管理
12.2。 优势的Spring框架的事务支持模型
12 2 1。 全局事务
12 2 2。 本地事务
12 2 3。 Spring框架是一致的编程模型
12.3。 了解Spring框架事务抽象
12.4。 同步资源交易
12 4 1。 高级同步方法
12 4 2。 低级同步方法
12 4 3。 TransactionAwareDataSourceProxy
12.5。 声明式事务管理
12 5 1。 了解Spring框架的声明性事务 实现
12 5 2。 声明式事务实现的例子
12 5 3。 一个声明式事务回滚
12 5 4。 配置不同的事务性语义不同 bean
12 5 5。 < tx:建议/ > 设置
12 5 6。 使用 transactional
transactional 设置
多个事务经理与 transactional
自定义快捷注释
12 5 7。 事务传播
需要
RequiresNew
嵌套
12 5 8。 建议事务性操作
12 5 9。 使用 transactional 与 AspectJ
12.6。 编程式事务管理
12 6 1。 使用 TransactionTemplate
指定事务设置
12 6 2。 使用 PlatformTransactionManager
12.7。 选择编程和声明式事务 管理
12.8。 特定于应用服务器的集成
12 8 1。 IBM WebSphere
12 8 2。 BEA WebLogic服务器
12 8 3。 甲骨文OC4J
12.9。 解决常见问题
12 - 9 - 1。 使用错误的事务管理器为一个特定的 数据源
12.10。 进一步的资源
13。 DAO支持
13.1。 介绍
13.2。 一致的异常层次结构
13.3。 注释用于配置刀或存储库类
14。 数据访问与JDBC
14.1。 介绍Spring框架JDBC
14 1 1。 选择一个JDBC数据库访问方法
14 1 2。 包的层次结构
14.2。 使用JDBC核心类来控制基本JDBC加工 错误处理
14 2 1。 JdbcTemplate
JdbcTemplate类的例子使用
JdbcTemplate 最佳实践
14 2 2。 NamedParameterJdbcTemplate
14 2 3。 SQLExceptionTranslator
14 2 4。 执行语句
14 2 5。 运行查询
14 2 6。 更新数据库
14 2 7。 检索自动生成的键
14.3。 控制数据库连接
14 3 1。 数据源
14 3 2。 DataSourceUtils
14 3 3。 SmartDataSource
14 3 4。 AbstractDataSource
14 3 5。 SingleConnectionDataSource
14 3 6。 DriverManagerDataSource
14 3 7。 TransactionAwareDataSourceProxy
14 3 8。 DataSourceTransactionManager
14 3 9。 NativeJdbcExtractor
14.4。 JDBC批处理操作
14 4 1。 基本批处理操作与JdbcTemplate
14 4 2。 批处理操作与对象的列表
14 4 3。 与多个批次的批处理操作
14.5。 简化JDBC操作与SimpleJdbc类
14 5 1。 插入数据使用SimpleJdbcInsert
14 5 2。 自动生成的键SimpleJdbcInsert检索使用
14 5 3。 指定一个SimpleJdbcInsert列
14 5 4。 使用SqlParameterSource提供参数值

14 5 5。 与SimpleJdbcCall调用一个存储过程
14 5 6。 显式地声明参数来使用 SimpleJdbcCall
14 5 7。 如何定义SqlParameters
14 5 8。 使用SimpleJdbcCall调用一个存储功能
14 5 9。 返回结果集/ REF光标从一个SimpleJdbcCall
14.6。 作为Java对象建模JDBC操作
14 6 1。 SqlQuery
14 6 2。 MappingSqlQuery
14 6 3。 SqlUpdate
14 6 4。 StoredProcedure
14.7。 常见问题与参数和数据值处理
14 7 1。 提供SQL类型信息参数
14 7 2。 处理BLOB和CLOB对象
14 7 3。 传入的值列表的条款
14 7 4。 处理复杂类型为存储过程调用
14.8。 嵌入式数据库支持
14 8 1。 为什么使用嵌入式数据库吗?
14 8 2。 创建一个嵌入式数据库实例使用Spring的XML
14 8 3。 以编程方式创建一个嵌入式数据库实例
14 8 4。 扩展嵌入式数据库支持
14 8 5。 使用HSQL
14 8 6。 使用H2
14 8 7。 使用Derby
14 8 8。 测试数据访问逻辑与嵌入式数据库
14.9。 初始化数据源
14 - 9 - 1。 初始化数据库实例使用Spring的XML
初始化的其他组件的依赖 数据库
15。 对象关系映射(ORM)数据访问
15.1。 介绍ORM和春天
15.2。 一般ORM集成考虑
15 2 1。 资源和事务管理
15 2 2。 异常翻译
15.3。 hibernate
15 3 1。 SessionFactory 设置在一个春天 容器
15 3 2。 实现DAOs基于普通Hibernate 3 API
15 3 3。 声明式事务划分
15 3 4。 程序性事务界定
15 3 5。 事务管理策略
15 3 6。 比较本地定义的容器管理和资源
15 3 7。 虚假的应用服务器与Hibernate的警告
15.4。 JDO
15 4 1。 PersistenceManagerFactory 设置
15 4 2。 实现DAOs基于JDO API的平原
15 4 3。 事务管理
15 4 4。 JdoDialect
15.5。 JPA
15 5 1。 三个选项设置在一个春天的JPA环境
LocalEntityManagerFactoryBean
获得一个 会 从 JNDI
LocalContainerEntityManagerFactoryBean
处理多个持久性单元
15 5 2。 实现DAOs基于普通JPA
15 5 3。 事务管理
15 5 4。 JpaDialect
15.6。 iBATIS SQL映射
15 6 1。 设置 SqlMapClient
15.6.2。 使用 SqlMapClientTemplate 和 SqlMapClientDaoSupport
15 6 3。 实现DAOs基于普通iBATIS API
16。 编组XML使用O / X映射器
16.1。 介绍
16.2。 Marshaller和解组程序
16 2 1。 Marshaller
16 2 2。 解组程序
16 2 3。 XmlMappingException
16.3。 使用信号员和解组程序
16.4。 XML的基于配置
16.5。 JAXB
16 5 1。 Jaxb2Marshaller
XML的基于配置
16.6。 蓖麻
16 6 1。 使用CastorMarshaller
16 6 2。 映射
XML的基于配置
16.7。 XMLBeans
16 7 1。 XmlBeansMarshaller
XML的基于配置

16.8。 JiBX
16 8 1。 JibxMarshaller
XML的基于配置
16.9。 XStream
16 9 1。 XStreamMarshaller
诉网络
17。 Web MVC框架
17.1。 介绍Spring Web MVC框架
17 1 1。 Spring Web MVC的特点
17 1 2。 可插入性的MVC实现
17.2。 这个 DispatcherServlet
17 2 1。 特殊的Bean类型 WebApplicationContext
17 2 2。 DispatcherServlet默认配置
17 2 3。 DispatcherServlet处理顺序
17.3。 实现控制器
17 3 1。 定义一个控制器, controller
17 3 2。 映射请求 @RequestMapping
新的支持类 @RequestMapping 方法在Spring MVC 3.1
URI模板模式
URI模板用正则表达式模式
路径模式
模式包含占位符
矩阵变量
消耗品媒体类型
可生产的媒体类型
请求参数和头的值
17 3 3。 定义 @RequestMapping 处理程序 方法
支持方法参数类型
支持方法返回类型
绑定请求参数方法参数 @RequestParam
映射与@RequestBody请求主体 注释
映射响应的身体 @ResponseBody 注释
使用 HttpEntity < ? >
使用 @ModelAttribute 在一个 方法
使用 @ModelAttribute 在一个 方法参数
使用 @SessionAttributes 存储模型 属性在HTTP会话请求之间
重定向和flash属性指定
处理 “应用程序/ x-www-form-urlencoded” 数据
映射cookie的值与@CookieValue注释
映射属性与@RequestHeader请求头 注释
方法参数和类型转换
定制 WebDataBinder 初始化
支持“last - modified”响应头方便 内容缓存
17 3 4。 异步请求处理
异常处理异步请求
拦截异步请求
配置为异步请求处理
17 3 5。 测试控制器
17.4。 处理程序映射
17日4 1。 拦截请求 HandlerInterceptor
17.5。 解决意见
17 5 1。 解决视图和 ViewResolver 接口
17 5 2。 链接ViewResolvers
17 5 3。 重定向到视图
RedirectView
这个 重定向: 前缀
这个 转发: 前缀
17 5 4。 ContentNegotiatingViewResolver
17.6。 使用flash属性
17.7。 建筑 URI 年代
17.8。 使用场所
17 8 1。 AcceptHeaderLocaleResolver
17 8 2。 CookieLocaleResolver
17 8 3。 SessionLocaleResolver
17 8 4。 LocaleChangeInterceptor
17.9。 使用主题
17 9 1。 概述主题
17 9 2。 定义主题
17 9 3。 主题解析器
17.10。 春天的多部分(文件上传)支持
17 10 1。 介绍
17 10 2。 使用 MultipartResolver 与 Commons FileUpload
17 10 3。 使用 MultipartResolver 与 Servlet 3.0
17 10 4。 处理一个文件上传表单中
17 10 5。 处理一个文件上传请求从编程的客户
17.11。 处理异常

17日11 1。 HandlerExceptionResolver
17日11 2。 @ExceptionHandler
17日11 3。 处理标准的Spring MVC例外
17日11 4。 注释业务异常与 @ResponseStatus
17日11 5。 自定义默认Servlet容器的错误页面
17.12。 约定优于配置支持
17 12 1。 控制器 ControllerClassNameHandlerMapping
17 12 2。 该模型 ModelMap (ModelAndView)
17 12 3。 视图- RequestToViewNameTranslator
17.13。 ETag支持
17.14。 基于代码的Servlet容器初始化
17.15。 配置Spring MVC
17日15 1。 启用MVC Java配置或MVC XML名称空间
17日15 2。 定制提供的配置
17日15 3。 配置拦截器
17日15 4。 配置内容协商
17日15 5。 配置视图控制器
17日15 6。 配置服务资源
17日15 7。 mvc:默认servlet处理程序
17日15 8。 更多的Spring Web MVC资源
17日15 9。 高级定制与MVC Java配置
17日15 10。 高级定制与MVC名称空间
18。 视图技术
18.1。 介绍
18.2。 JSP & JSTL
18 2 1。 视图解析器
18 2 2。 “普通的jsp和JSTL
18 2 3。 额外的标签促进发展
18 2 4。 使用Spring标记库的形式
配置
这个 形式 标签
这个 输入 标签
这个 复选框 标签
这个 复选框 标签
这个 radiobutton 标签
这个 radiobuttons 标签
这个 密码 标签
这个 选择 标签
这个 选项 标签
这个 选项 标签
这个 Textarea 标签
这个 隐藏 标签
这个 错误 标签
HTTP方法转换
HTML5标签
18.3。 瓷砖
18 3 1。 依赖性
18 3 2。 如何将瓷砖
UrlBasedViewResolver
ResourceBundleViewResolver
SimpleSpringPreparerFactory 和 SpringBeanPreparerFactory
18.4。 速度& FreeMarker
18 4 1。 依赖性
18 4 2。 上下文配置
18 4 3。 创建模板
18 4 4。 高级配置
速度属性
freemarker
18 4 5。 绑定支持和形式处理
bind宏
简单绑定
表单输入生成宏
HTML转义和XHTML合规
18.5。 XSLT
18 5 1。 我的第一个单词
Bean定义
标准MVC控制器代码
模型数据转换为XML
定义视图属性
文档转换
18 5 2。 总结
18.6。 文档视图(PDF / Excel)
18 6 1。 介绍
18 6 2。 配置和设置
文档视图定义
控制器代码

子类化Excel的观点
子类化对于PDF的观点
18.7。 jasperreports
18 7 1。 依赖性
18 7 2。 配置
配置 ViewResolver
配置 视图 年代
关于报告文件
使用 JasperReportsMultiFormatView
18 7 3。 填充 ModelAndView
18 7 4。 处理子报告
配置子报告文件
配置子报告数据来源
18 7 5。 出口国参数配置
18.8。 饲料的观点
18.9。 XML编组视图
18.10。 JSON映射视图
19。 与其他web框架集成
19.1。 介绍
19.2。 常见的配置
19.3。 JavaServer Faces 1.1和1.2
19 3 1。 DelegatingVariableResolver(JSF 1.1/1.2)
19 3 2。 SpringBeanVariableResolver(JSF 1.1/1.2)
19 3 3。 SpringBeanFacesELResolver(JSF 1.2 +)
19 3 4。 FacesContextUtils
19.4。 Apache Struts 1。 倍和2.倍
19日4 1。 ContextLoaderPlugin
DelegatingRequestProcessor
DelegatingActionProxy
19日4 2。 ActionSupport类
19.5。 网络系统2. x
19.6。 Tapestry 3。 倍和4.倍
19日6 1。 注入spring管理bean
依赖注入Spring bean到Tapestry页面
组件定义文件
添加抽象访问器
依赖注入Spring bean到Tapestry页面- Tapestry 4。 x风格
19.7。 进一步的资源
20。 Portlet MVC框架
20.1。 介绍
20 1 1。 控制器- C在MVC
20 1 2。 视图- V在MVC
20 1 3。 web范围bean
20.2。 这个 DispatcherPortlet
20.3。 这个 ViewRendererServlet
20.4。 控制器
20 4 1。 基类AbstractController 和 PortletContentGenerator
20 4 2。 其他简单的控制器
20 4 3。 命令控制器
20 4 4。 PortletWrappingController
20.5。 处理程序映射
20 5 1。 PortletModeHandlerMapping
20 5 2。 ParameterHandlerMapping
20 5 3。 PortletModeParameterHandlerMapping
20 5 4。 添加 HandlerInterceptor 年代
20 5 5。 HandlerInterceptorAdapter
20 5 6。 ParameterMappingInterceptor
20.6。 观点和解决他们
20.7。 多部分(文件上传)支持
20 7 1。 使用 PortletMultipartResolver
20 7 2。 处理一个文件上传表单中
20.8。 处理异常
20.9。 基于注解的控制器配置
20 - 9 - 1。 设置调度程序注释支持
20 9 2。 定义一个控制器, controller
20 9 3。 映射请求 @RequestMapping
20 9 4。 支持处理程序方法参数
20 9 5。 绑定请求参数方法参数 @RequestParam
20 9 6。 提供一个链接的数据模型 @ModelAttribute
20 9 7。 指定属性存储在会话 @SessionAttributes
20 9 8。 定制 WebDataBinder 初始化
定制数据绑定和 @InitBinder
配置一个自定义 WebBindingInitializer
20.10。 Portlet应用程序部署
VI。集成
21。 使用Spring Remoting和web服务

21.1。 介绍
21.2。 暴露服务使用RMI
21 2 1。 出口服务使用 RmiServiceExporter
21 2 2。 连接的服务在客户端
21.3。 使用麻绳或麻袋远程调用服务通过HTTP
21日3 1。 布线的 DispatcherServlet 对于 黑森和co。
21日3 2。 暴露你的bean使用 HessianServiceExporter
21日3 3。 连接的服务在客户端
21日3 4。 用粗麻布
21日3 5。 应用HTTP基本身份验证服务暴露通过 黑森或麻袋
21.4。 使用HTTP调用程序的公开服务
21 4 1。 公开服务对象
21 4 2。 连接的服务在客户端
21.5。 Web服务
21 5 1。 基于servlet的web服务公开使用jax - rpc
21 5 2。 访问web服务使用jax - rpc
21 5 3。 注册jax - rpc Bean映射
21 5 4。 注册你自己的jax - rpc处理程序
21 5 5。 基于servlet的web服务公开使用jax - ws
21 5 6。 出口独立的web服务使用jax - ws
21 5 7。 导出web服务使用jax - ws国际扶轮的春天 支持
21 5 8。 访问web服务使用jax - ws
21.6。 jms
21日6 1。 服务器端配置
21日6 2。 客户端配置
21.7。 还没有实现自动识别为远程接口
21.8。 考虑当选择技术
21.9。 在客户端访问RESTful服务
21 - 9 - 1。 RestTemplate
使用URI
处理请求和响应头
21日9 2。 HTTP消息转换
StringHttpMessageConverter
FormHttpMessageConverter
ByteArrayHttpMessageConverter
MarshallingHttpMessageConverter
MappingJackson2HttpMessageConverter(或MappingJacksonHttpMessageConverter与杰克逊1.x)
SourceHttpMessageConverter
BufferedImageHttpMessageConverter
22。 Enterprise JavaBeans(EJB)集成
22.1。 介绍
22.2。 ejb访问
22日2 1。 概念
22日2 2。 访问当地SLSBs
22日2 3。 访问远程SLSBs
22日2 4。 访问EJB 2。 x SLSBs与EJB 3 SLSBs
22.3。 使用Spring的EJB实现支持类
22 3 1。 EJB 2。 x基类
22 3 2。 EJB 3注入拦截器
23。 JMS(Java消息服务)
23.1。 介绍
23.2。 使用Spring JMS
23 2 1。 JmsTemplate
23 2 2。 连接
缓存消息传递资源
SingleConnectionFactory
CachingConnectionFactory
23 2 3。 目的地管理
23 2 4。 消息侦听器容器
SimpleMessageListenerContainer
DefaultMessageListenerContainer
23 2 5。 事务管理
23.3。 发送 消息
23日3 1。 使用消息转换器
23日3 2。 SessionCallback 和 ProducerCallback
23.4。 接收消息
23日4 1。 同步接收
23日4 2。 异步接收消息驱动pojo,
23日4 3。 这个 SessionAwareMessageListener 接口
23日4 4。 这个 MessageListenerAdapter
23日4 5。 在事务处理消息
23.5。 支持JCA消息端点
23.6。 JMS名称空间支持
24。 JMX
24.1。 介绍

24.2。 出口你的豆子JMX
24 2 1。 创建一个 MBeanServer
24 2 2。 重用现有的 MBeanServer
24 2 3。 延迟初始化的mbean
24 2 4。 自动注册mbean
24 2 5。 控制登记行为
24.3。 控制管理接口的豆子
24 3 1。 这个 MBeanInfoAssembler 接口
24 3 2。 使用源代码级别的元数据(JDK 5.0注释)
24 3 3。 源代码级别的元数据类型
24 3 4。 这个 AutodetectCapableMBeanInfoAssembler 接口
24 3 5。 使用Java接口定义管理接口
24 3 6。 使用 MethodNameBasedMBeanInfoAssembler
24.4。 控制 ObjectName 年代为你豆
24 4 1。 阅读 ObjectName 年代从 属性
24 4 2。 使用 MetadataNamingStrategy
24 4 3。 配置基于注解的MBean的出口
24.5。 jsr - 160连接器
24日5 1。 端连接器
24日5 2。 端连接器
24日5 3。 JMX在麻袋/黑森/ SOAP
24.6。 通过代理访问mbean
24.7。 通知
24 7 1。 通知监听器注册
24 7 2。 发布通知
24.8。 进一步的资源
25。 JCA CCI
25.1。 介绍
25.2。 配置CCI
25 2 1。 连接器配置
25 2 2。 ConnectionFactory 配置在 春天
25 2 3。 配置CCI连接
25 2 4。 使用单一CCI连接
25.3。 使用Spring的CCI访问支持
25 3 1。 记录转换
25 3 2。 这个 CciTemplate
25 3 3。 DAO支持
25 3 4。 自动输出记录生成
25 3 5。 总结
25 3 6。 使用CCI 连接 和 交互 直接
25 3 7。 例子 CciTemplate 使用
25.4。 作为操作对象建模CCI访问
25 4 1。 MappingRecordOperation
25 4 2。 MappingCommAreaOperation
25 4 3。 自动输出记录生成
25 4 4。 总结
25 4 5。 例子 MappingRecordOperation 使用
25 4 6。 例子 MappingCommAreaOperation 使用
25.5。 交易
26。 邮件
26.1。 介绍
26.2。 使用
26日2 1。 基本 MailSender 和 SimpleMailMessage 使用
26日2 2。 使用 JavaMailSender 和 MimeMessagePreparator
26.3。 使用JavaMail MimeMessageHelper
26日3 1。 发送附件和内联资源
附件
内联资源
26日3 2。 创建电子邮件内容使用模板库
一个速度的基础例子
27。 任务执行和调度
27.1。 介绍
27.2。 春天 TaskExecutor 抽象
27 2 1。 TaskExecutor 类型
27 2 2。 使用 TaskExecutor
27.3。 春天 TaskScheduler 抽象
27日3 1。 这个 触发 接口
27日3 2。 触发 实现
27日3 3。 TaskScheduler 实现
27.4。 注释支持调度和异步 执行
27日4 1。 使调度注释
27日4 2。 @Scheduled注释的
27日4 3。 @Async注释的
27日4 4。 遗嘱执行人资格与@Async
27.5。 任务名称空间
27日5 1。 “调度”元素

27日5 2。 “执行人”元素
27日5 3。 “计划任务”元素
27.6。 使用石英调度器
27 6 1。 使用JobDetailBean
27 6 2。 使用 MethodInvokingJobDetailFactoryBean
27 6 3。 连接使用触发和工作 SchedulerFactoryBean
28。 动态语言支持
28.1。 介绍
28.2。 第一个例子
28.3。 定义bean,动态语言支持
28日3 1。 常见的概念
这个 < lang:语言/ > 元素
可刷新的豆子
内联动态语言源文件
理解构造函数注入上下文中的动态语言支持bean
28日3 2。 JRuby bean
28日3 3。 Groovy bean
通过回调Groovy对象的定制
28日3 4。 BeanShell bean
28.4。 场景
28 4 1。 照本宣科的Spring MVC控制器
28 4 2。 脚本验证器
28.5。 零零碎碎
28 5 1。 AOP——建议脚本bean
28 5 2。 范围
28.6。 进一步的资源
29。 缓存抽象
29.1。 介绍
29.2。 理解抽象的缓存
29.3。 声明基于注解的缓存
29日3 1。 @Cacheable 注释
缺省密钥生成
自定义键生成申报
条件缓存
可用缓存？评估上下文
29日3 2。 @CachePut 注释
29日3 3。 @CacheEvict 注释
29日3 4。 @Caching 注释
29日3 5。 启用缓存注释
29日3 6。 使用自定义注释
29.4。 声明性xml缓存
29.5。 配置缓存存储
29日5 1。 JDK ConcurrentMap 的 缓存
29日5 2。 ehcache基于 缓存
29日5 3。 gemfire基于 缓存
29日5 4。 处理缓存没有后备存储器
29.6。 插入不同的后端缓存
29.7。 我如何设置TTL /创科实业/驱逐政策/ XXX特性?
七世。 附录
a .经典弹簧的使用
背书的。 经典ORM使用
一个1 1。 hibernate
这个 hibernatetemplate
实现基于spring dao没有回调
一个1 2。 JDO
JdoTemplate 和 JdoDaoSupport
一个1 3。 JPA
JpaTemplate 和 JpaDaoSupport
a。 经典Spring MVC
由。 JMS使用
一个3 1。 JmsTemplate
一个3 2。 异步消息接收
一个3 3。 连接
3 4。 事务管理
b .经典Spring AOP使用
责任。 切入点API在春天
b 1 1。 概念
b 1 2。 操作切入点
b 1 3。 AspectJ切入点表达式
实施了。 便利切入点实现
静态切入点
动态切入点
b 1 5。 切入点超类
b 1 6。 定制切入点
b 2。 建议API在春天
b 2 1。 建议生命周期

b 2 2。 建议类型在春天
拦截around通知
建议之前
抛出建议
回国后的建议
介绍的建议
b 3。 顾问API在春天
b 4。 使用ProxyFactoryBean创建AOP代理
b 4 1。 基本
b 4 2。 JavaBean属性
b 4 3。 JDK和cglib建立代理
b 4 4。 代理接口的
b 4 5。 代理类
b 4 6。 使用“全球”顾问
b 5。 简洁代理定义
b 6。 以编程方式创建AOP代理与ProxyFactory
b 7。 操纵建议对象
b 8。 使用“火狐的一个插件”设施
b 8 1。 火狐的一个插件bean定义
BeanNameAutoProxyCreator
DefaultAdvisorAutoProxyCreator
AbstractAdvisorAutoProxyCreator
b 8 2。 使用元数据驱动的汽车代理
b 9。 使用TargetSources
b 9 1。 热可切换目标来源
b 9 2。 池目标来源
b 9 3。 原型目标来源
b 9 4。 ThreadLocal 目标来源
b 10。 定义新的 建议 类型
b 11。 进一步的资源
c .迁移到Spring Framework 3.1
c 1。 组件扫描对“org”基础包
d .迁移到Spring Framework 3.2
d 1。 新可选依赖
d 2。 EHCache支持搬到spring上下文的支持
d 3。 内联的弹簧asm jar
d 4。 明确CGLIB依赖不再需要
d 5。 对OSGi用户
d 6。 MVC Java配置和MVC的名称空间
d 7。 解码的URI变量值
d 8。 HTTP补丁方法
d 9。 瓷砖3
d 10。 Spring MVC测试独立项目
d 11。 弹簧测试依赖关系
d 12。 公共API的变化
d 12 1。 JDiff报告
d 12 2。 随着
e . XML的基于配置
e 1。 介绍
e 2。 XML的基于配置
e 2 1。 引用模式
e 2 2。 这个 util 模式
< util:常数/ >
< util:属性路径/ >
< util:属性/ >
< util:列表/ >
< util:地图/ >
< util:设置/ >
e 2 3。 这个 JEE 模式
< jee:jndi查找/ > (简单的)
< jee:jndi查找/ > (与单JNDI环境设置)
< jee:jndi查找/ > (有多个JNDI环境设置)
< jee:jndi查找/ > (复杂的)
< jee:local-slsb / > (简单的)
< jee:local-slsb / > (复杂的)
< jee:remote-slsb / >
e 2 4。 这个 朗 模式
e 2 5。 这个 jms 模式
e 2 6。 这个 TX (事务)模式
e 2 7。 这个 aop 模式
e 2 8。 这个 上下文 模式
<属性占位符/ >
<注释配置/ >
<组件扫描/ >
<加载时间韦弗/ >
< spring配置/ >

< mbean出口/ >
e 2 9。 这个 工具 模式
e 2 10。 这个 JDBC 模式
e 2 11。 这个 缓存 模式
e 2 12。 这个 bean 模式
f .可扩展的XML编写
f 1。 介绍
f 2。 创作模式
f 3。 编码一个 NamespaceHandler
中。 编码一个 BeanDefinitionParser
四。 注册处理程序和模式
f 5 1。 “meta - inf / spring处理程序”
f 5 2。 “meta - inf / spring模式”
f 6。 使用一个自定义的扩展在Spring的XML配置
有为。 丰满的例子
f 7 1。 自定义标签内嵌套自定义标记
f 7 2。 自定义属性在“正常”的元素
f 8。 进一步的资源
g .弹簧tld
g 1。 介绍
g 2。 这个 绑定 标签
g 3。 这个 escapeBody 标签
g 4。 这个 hasBindErrors 标签
g 5。 这个 htmlEscape 标签
g 6。 这个 消息 标签
g 7。 这个 nestedPath 标签
g 8。 这个 主题 标签
g 9。 这个 变换 标签
g 10。 这个 url 标签
g 11。 这个 eval 标签
h .弹簧形式tld
1。 介绍
h 2。 这个 复选框 标签
h 3。 这个 复选框 标签
h 4。 这个 错误 标签
h 5。 这个 形式 标签
h 6。 这个 隐藏 标签
h 7。 这个 输入 标签
h 8。 这个 标签 标签
h 9。 这个 选项 标签
h 10。 这个 选项 标签
h 11。 这个 密码 标签
h 12。 这个 radiobutton 标签
h 13。 这个 radiobuttons 标签
h 14。 这个 选择 标签
h 15。 这个 Textarea 标签

PartA我。 Spring框架的概况

Spring框架是一个轻量级的解决方案和一个潜在的 一站式为构建企业级应用程序。 然而, 春天是模块化的,允许你只使用那些你需要的部分, 无需引入其余。 您可以使用IoC容器, Struts在上,但是你也可以只使用 Hibernate集成代码 或 JDBC抽象层 。 春天 框架支持声明式事务管理、远程访问 你的逻辑通过RMI或web服务,以及各种选项 坚持你的数据。 它提供了一个功能全面的 MVC框架 ,并允许您 整合 aop 透明地进 你的软件。

春天是设计为非侵入性的,这意味着你的域 逻辑代码通常没有依赖框架本身。 在 你的集成层(比如数据访问层),一些 依赖于数据访问技术和Spring库将 存在。 然而,它应该很容易隔离这些依赖项的 其余的代码库。

这个文档是一个参考指南,Spring框架功能。 如果你有任何请求,评论,或问题在这个文件, 请张贴在用户邮件列表或支持论坛 http://forum.springsource.org/ 。

1。 一个介绍Spring框架

Spring框架是一个Java平台,提供全面的 基础设施支持开发Java应用程序。 弹簧处理 基础设施可以让你专注于你的应用程序。

春天使您能够构建应用程序从一个普通旧式Java objectsa (pojo)和应用企业服务非侵入性的方法来pojo。 这 功能适用于Java SE的编程模型和完全和部分 Java EE。

你的例子,作为应用程序开发人员,可以使用弹簧 平台优势:

- 将一个Java方法中执行数据库事务没有 必须处理事务api。
- 让一个本地Java方法远程过程无需交易 与远程api。
- 让一个本地Java方法无需管理操作 处理JMX api。
- 让一个本地Java方法一个消息处理程序而无需交易 与JMS api。

1.1一个依赖注入和控制反转

Java应用程序——一个松散的术语,历经了从运行 限制到n层端企业应用程序, 通常由对象来协作形式应用 适当的。 因此,在一个应用程序对象有 依赖性 在每个其他。

尽管Java平台提供了一个丰富的应用程序 开发功能,它缺乏意味着组织基础构建块成一个连贯的整体,留下这任务架构师和 开发人员。 真的,你可以使用设计模式如 工厂，抽象工厂，建设者，装饰 ,和 服务定位器 撰写各种类和 对象实例的应用程序。 然而,这些模式 只是简单的说:最佳实践提供了一个名称,描述的是什么 模式确实,应用,问题,地址等等。 模式是形式化的最佳实践 你必须实现 自己 在您的应用程序。

背景

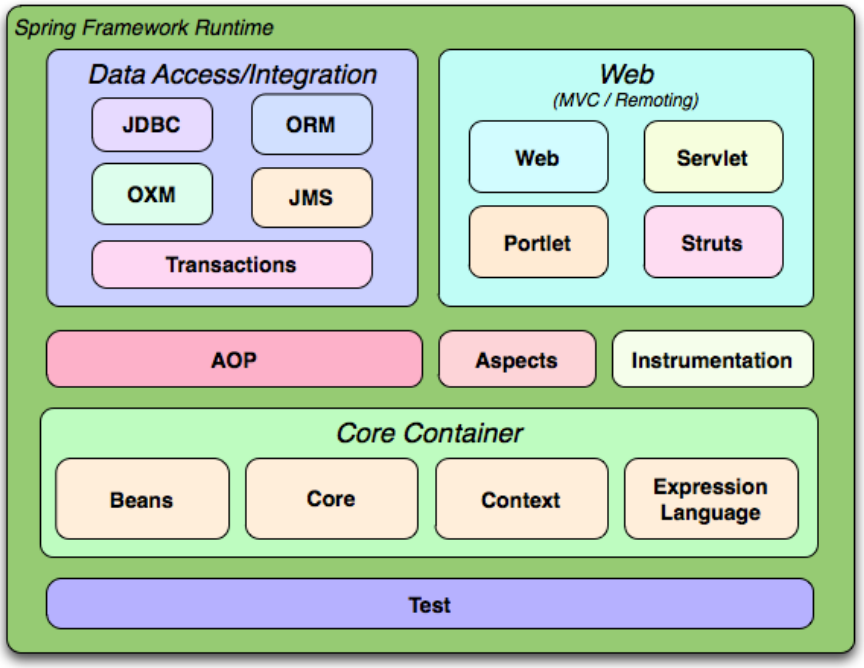
一个 问题是,什么方面的控制 [他们]反相吗? 一个 马丁提出一个问题 关于控制反转(IoC)于2004年在他的网站。 福勒建议 重命名原则,让它更加不言自明,想出了 依赖注入 。

为深入了解国际奥委会和DI,指福勒的文章中 http://martinfowler.com/articles/injection.html 。

Spring框架 控制反转 (IoC) 组件的解决了这个问题通过提供一种形式化的方法 不同的组件组合成一个完整的工作程序准备好使用。 这个 Spring框架概括形式化的设计模式是一流的 对象,可以整合到您自己的应用程序(年代)。 无数 组织和机构使用Spring框架在这种方式 工程师健壮, 维护 应用程序。

1.2一个模块

Spring框架是由功能组织成大约20 模块。 这些模块被分组到核心容器、数据 访问/集成、网络、AOP(面向方面编程), 仪器仪表和测试,见下图。



Spring框架的概述

1.2.1A核心容器

这个 核心 容器 由核心、豆类、上下文和 表达式语言模块。

这个 核心和 bean 模块提供的基本部分 框架,包括国际奥委会和依赖注入特性。 这个 BeanFactory 是一个复杂的实现的 工厂模式。 它使得不需要编程单件和 允许您配置和规范的解耦 从你的实际程序逻辑依赖关系。

这个 上下文 模块构建在提供的坚实的基础 核心和豆类 模块:它是一种手段来访问对象在一个框架风格的方式 这是类似于一个

JNDI注册表。 上下文模块继承它 从bean模块功能并添加国际化支持 (例如,使用资源包)、事件传播, 资源加载,透明,创建上下文 例子,一个servlet容器。 上下文模块还支持Java EE 功能,如EJB,JMX,和基本的远程控制。 这个 ApplicationContext 接口是焦点 上下文的模块。

这个 表达式 语言 模块提供 一个强大的表达式语言查询和操作一个对象 图在运行时。 它是一个扩展的统一表达式语言 (统一EL)按JSP 2.1规范。 语言 支持设置和获取属性值,属性赋值, 方法调用上下文的访问数组、集合和 索引器、逻辑、算术运算符、命名变量,和 检索对象的名字从Spring的IoC容器。 它还 支持列表投影和选择以及常见的列表 聚合。

1.2.2A数据访问/集成

这个 数据访问/集成 层由 JDBC,ORM,OXM、JMS和事务模块。

这个 JDBC 模块提供 一个JDBC抽象层,消除了需要做乏味的JDBC代码 和解析数据库供应商的特定错误代码。

这个 orm 模块 提供集成层为流行的对象-关系映射api, 包括 JPA , JDO , hibernate ,和 iBatis 。 使用ORM包可以使用 所有这些框架的O / r映射结合所有其他的 功能弹簧提供,如简单的声明性事务 前面提到的管理特性。

这个 OXM 模块提供了一个抽象 层,支持对象/ XML映射实现JAXB,Castor, XMLBeans,JiBX和XStream。

Java消息传递服务(jms)模块 包含功能,为生产和消费信息。

这个 事务 模块支持 编程式和声明式事务管理类 实现特定的接口和 你所有的pojo(传统 Java对象) 。

1.2.3A网络

这个 web 层由网络, web servlet、web struts,portlet的web模块。

春天的 web 模块提供了基本 以网络为中心的集成特性,比如多部分文件上传 功能和初始化的IoC容器使用servlet 听众和一个面向web的应用程序上下文。 它还包含了 Spring的web部件远程支持。

这个 web servlet 模块包含弹簧的 模型-视图-控制器(MVC) 实现为web应用程序。 Spring的MVC框架提供了一个 清洁分离域模型代码和web表单,并整合 与所有的其他功能,Spring框架。

这个 web struts 模块包含支持 类为集成一个经典的Struts web层在一个春天 应用程序。 注意,这个支持现在是春天的废弃3.0。 考虑迁移应用程序到Struts 2.0和它的春天 集成或Spring MVC的解决方案。

这个 web portlet 模块提供了MVC 实现用于portlet环境和反映 web servlet模块的功能。

1.2.4A AOP和仪表

春天的 aop 模块 提供了一个 AOP联盟 兼容的面向方面的 编程实现允许您定义,例如, 方法拦截器和切入点来干净代码解耦 实现了功能,应该分开。 使用源代码级别的 元数据功能,您还可以将行为信息 到你的代码,在相似的方式。 净的属性。

单独的 方面 模块提供 整合与AspectJ。

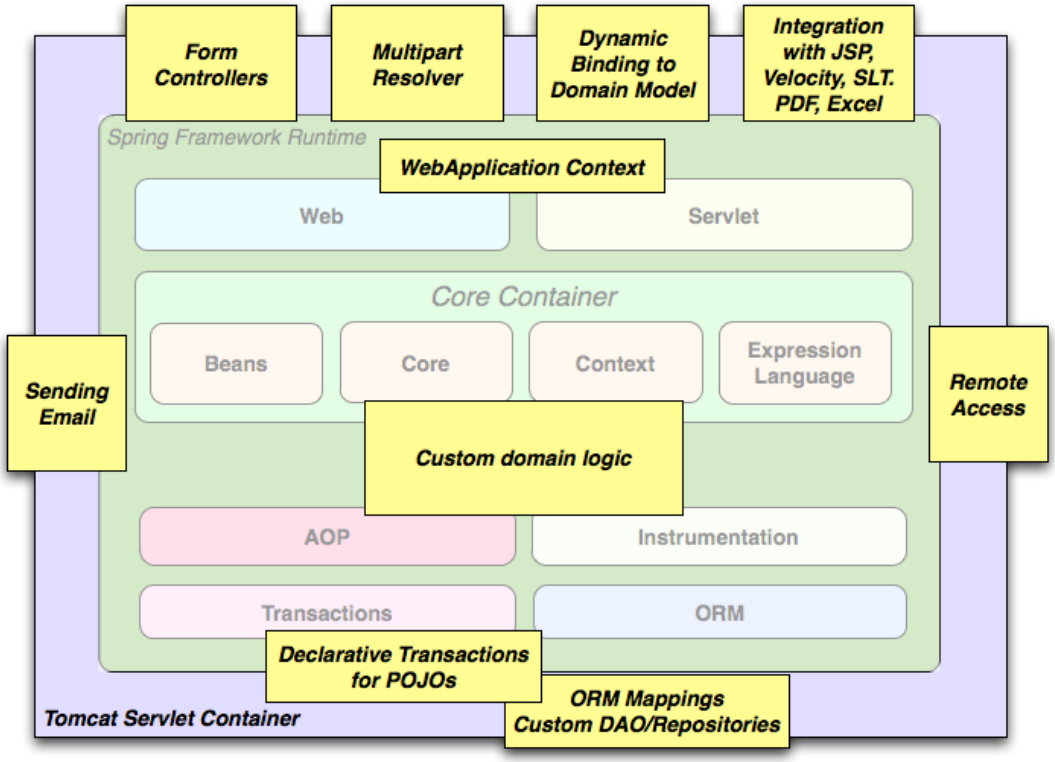
这个 仪表 模块提供类 仪表支持和类加载器的实现中使用 某些应用程序服务器。

1.2.5A测试

这个 测试 模块支持的测试 弹簧组件使用JUnit和TestNG。 它提供了一致的加载 春天ApplicationContexts和缓存的上下文。 它还 为模拟对象,您可以使用它来测试你的代码 隔离。

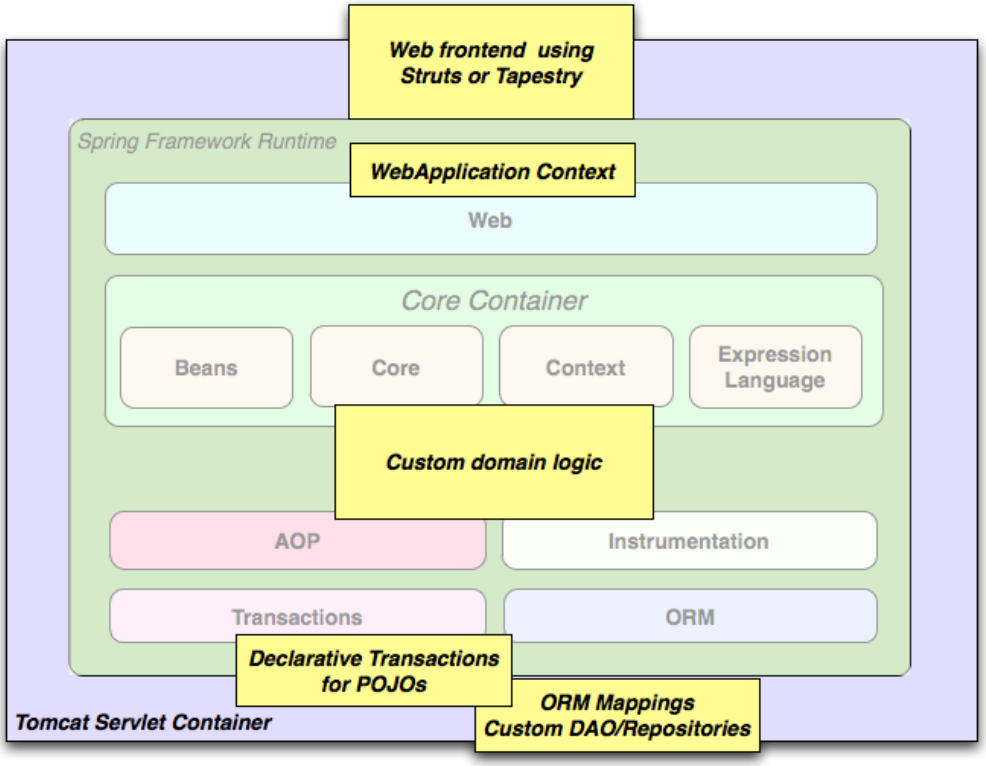
1.3一个使用场景

前面描述的构建块使弹簧一个逻辑 选择在许多场景中,从applet向成熟的企业 应用程序使用Spring的事务管理功能和 web框架集成。



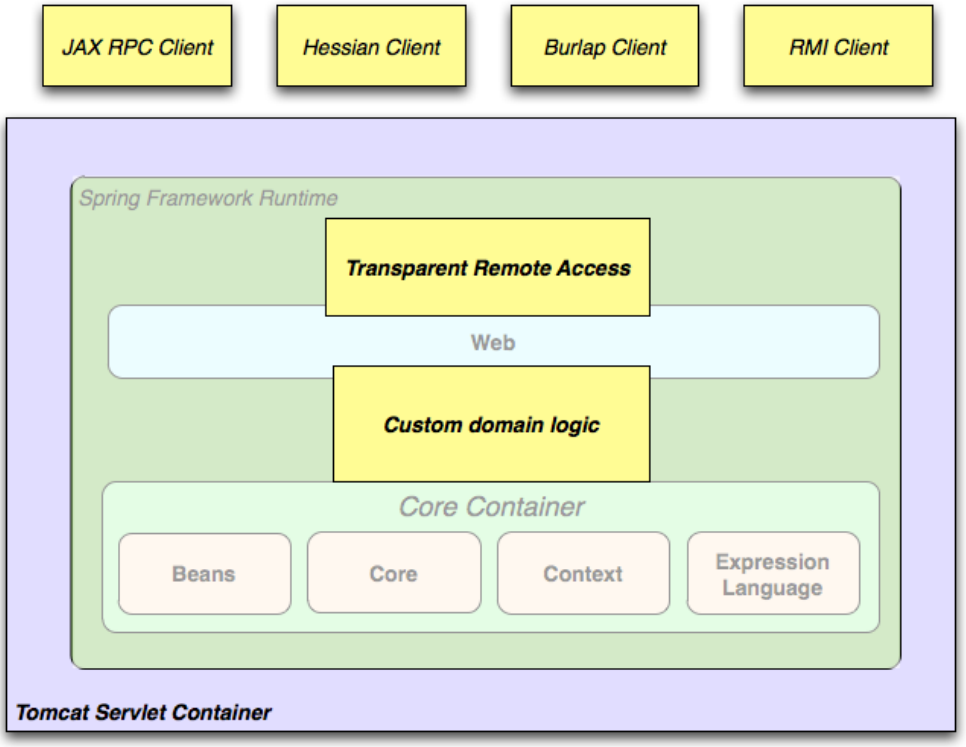
典型成熟的Spring web 应用

春天的 声明 事务管理功能 使web应用程序完全 事务,就像它是如果你使用EJB容器管理的 事务。 你所有的自定义业务逻辑可以实现 简单的pojo和管理的Spring的IoC容器。 额外的服务 包括支持发送电子邮件和验证,是独立的 web层,它可以让你选择在哪里执行验证规则。 春天的ORM支持是集成了JPA,Hibernate,JDO和iBatis; 例如,当使用Hibernate,您可以继续使用您现有的 Hibernate映射文件和标准 SessionFactory 配置。 形式 控制器无缝集成的web层与域模型, 删除需要 actionform 或其他类 HTTP参数值转换为你的域模型。



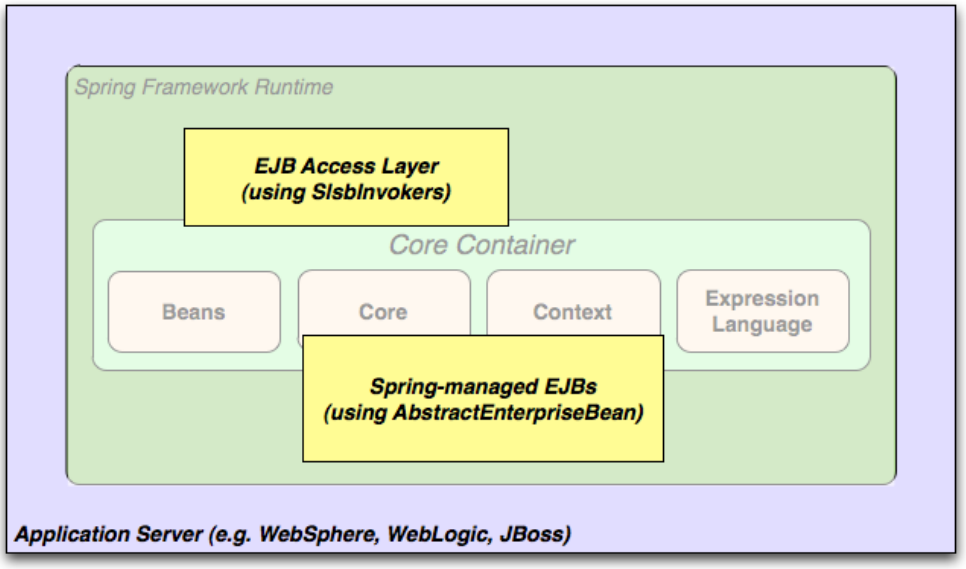
春天中间层使用第三方web 框架

有时情况不允许你完全切换到一个 不同的框架。 Spring框架并 不 强迫你使用一切在它,它不是一个 孤注一掷的 解决方案。 现有前端建立 与网络系统、Struts、挂毯、或其他UI框架可以集成 与一个基于Spring的中间层,它允许您使用弹簧 事务特性。 你只需要连接您的业务逻辑使用 一个 ApplicationContext 和使用 WebApplicationContext 整合您的web 层。



Remoting使用场景

当你需要通过web服务访问现有的代码,你可以 使用Spring的 黑森- , 麻袋, , Rmi - 或 JaxRpcProxyFactory 类。 启用远程访问现有的应用程序不是 困难的。



ejb -包装现有的pojo

Spring框架还提供了一个 访问和 抽象层 为Enterprise JavaBeans,使您可以重用 你现有的pojo并包装在使用无状态会话bean可伸缩的,故障安全的web应用程序可能需要声明 安全。

1.3.1A依赖管理和命名约定

依赖关系管理和依赖注入是不同的 的事情。 让那些漂亮的春天的特征到您的应用程序(如 依赖注入)你需要组装所有的图书馆需要(jar 文件),让他们到您在运行时类路径中,并可能在 编译时间。 这些依赖关系没有虚拟组件 注入,但物理资源在一个文件系统(通常)。 这个 过程的依赖关系管理涉及到定位这些资源, 存储它们并将它们添加到类路径。 依赖关系可以直接 (如我的应用程序依赖于Spring运行时)或间接(如我 应用取决于 commons-dbcp 取决于 共享池)。 间接依赖关系也被称为 “传递” ,这是那些依赖关系难以确定 和管理。

如果你打算使用弹簧你需要得到一个jar的副本 库组成件弹簧,你需要。 为了让这个 春天是容易打包为一组模块分开 尽可能多的依赖关系,例如如果你不想 编写一个web应用程序中,您不需要spring web模块。 参考 春天在本指南库模块我们使用速记命

名 公约 弹簧- * 或 春天- * . jar, 其中“*” 代表了模块的短名称(如。 spring核心 , spring-webmvc , spring jms 等等)。 实际的 jar文件的名字,你可能会以这种形式使用(见下文)或者它可能 不,通常它也有一个版本号在文件名称(如。 spring核心3 0 0 释放罐)。

一般来说,春天发布了4个不同的工件 的地方:

- 在社区下载站点 http://www.springsource.org/download/community 。 在这里你找到所有的弹簧罐捆扎在一起成为一个zip文件 为容易的下载。 这里的名字从版本3.0罐 形式 org.springframework。* - <版本> . jar 。
- Maven中央,这是默认的存储库,Maven 查询,并且不需要任何特殊配置使用。 许多公共库,弹簧取决于也 可以从Maven中央和一大段弹簧 社区使用Maven的依赖管理,所以这是 方便他们。 这里的名字是形式的罐子 春天- * - <版本> . jar 和 Maven groupId是 org.springframework 。
- 企业Bundle库(电子束记录),它是由 SpringSource和主机所有的库集成 春天。 两个Maven和Ivy存储器可用来所有 Spring jar和他们的依赖关系,以及大量的其他 公共库,人们用在应用程序与弹簧。 两 完整版本,还里程碑和开发快照 在这里部署。 jar文件的名称是在同一个表格 社区下载 (org.springframework。* - <版本> . jar), 依赖性也在这个“长”形式,与外部库 (不是从SpringSource)有前缀 com springsource 。 看到 faq 为更多的信息。
- 在一个公共Maven存储库托管在Amazon S3作为 开发快照和里程碑版本(一份期末 版本也在这里举行)。 这个jar文件名称相同的 形式是Maven中央,所以这是一个有用的地方去 开发版本的Spring使用与其他库部署 在Maven中央。

所以你首先需要决定如何管理你的 依赖性:大多数人使用一个自动化的系统像Maven或者常春藤,但是 你也可以手动下载所有的罐子自己。 当 获得与Maven和Ivy春天你然后决定哪个地方 你会得到它。 一般来说,如果你关心OSGi,使用电子束重熔, 因为它的房屋构件的兼容OSGi所有春天的 依赖关系,如Hibernate和Freemarker。 如果OSGi并不重要 你,任何一个地方工作,虽然有一些利弊之间 他们。 一般来说,选择一个地方或另一个用于你的项目,不要 混合。 这是特别重要,因为工件必然电子束重熔 使用不同的命名约定比Maven中央工件。

1.1为多。 一个比较的Maven中央和SpringSource电子束重熔 存储库

特性	Maven中央	电子束重熔
OSGi兼容	没有明确的	是的
数量的工件	成千上万的;所有种类	数以百计的;那些春天集成了
一致的命名惯例	没有	是的
命名约定:GroupId	各不相同。 新工件经常使用的域名,例如。 org.slf4j。 老年人通常只使用工件的名字,例如。 log4j。	域名的起源或主要包根,如。 org.springframework
命名约定:与ArtifactId设为	各不相同。 一般项目或模块名称、使用 连字符 “-” 分隔符,例如,logj4 spring核心。	束符号名称,来自主要的包 根,如 org.springframework.beans。 如果jar必须 打补丁,确保OSGi合规然后com。 springsource是 附加的,例如com springsource org apache log4j
命名约定:版本	各不相同。 许多新工件使用规则。 米或m m m。 X(m =数字,X =文本)。 老年人使用。一些既不答。 订购 被定义,但不能经常依赖,所以不严格吗 可靠的。	OSGi版本号m m m。 X,例如。3 0 0 rc3。 文本 预选赛上强加了字母排序的版本 相同的数值。
出版	通常通过rsync或源代码控制自动更新。 项目作者可以上传个人JIRA jar。	手册(JIRA SpringSource处理)
质量保证	根据政策。 精度是责任的 作者。	广泛的对OSGi manifest,Maven POM和常春藤 元数据。 QA团队执行的春天。
托管	Contegix。 由Sonatype与几个 镜子。	S3 SpringSource资助。
搜索实用程序	各种	http://www.springsource.com/repository
集成与SpringSource工具	集成通过STS Maven的依赖 管理	广泛的集成通过STS Maven,袋鼠, CloudFoundry

Spring依赖和依靠弹簧

虽然Spring提供了集成和支持一个巨大的 范围内的企业和其他外部工具,它特意保持 其强制性依赖一个绝对最小:你不应该 查找和下载(甚至自动)大量的jar 图书馆为了使用弹簧为简单的用例。 基本 依赖注入只有一个强制性的外部依赖, 并对日志记录(参见后面的一个更详细的描述 日志记录选项)。

接下来我们大纲的基本步骤需要配置一个 应用程序依赖于春天,首先与Maven然后用 艾薇。 在所有情况下,如果有不清楚的地方,请参阅文档 你的依赖关系管理系统,或看一些示例代码- 弹簧本身使用Ivy来管理依赖当它正在建设, 我们的样品主要是使用Maven。

Maven依赖管理

如果您正在使用Maven的依赖管理你甚至不 需要提供日志依赖明确。 例如,要 创建一个应用程序上下文和使用依赖注入来 配置一个应用程序,你会看起来像Maven的依赖 这个:

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>3.0.0.RELEASE</version>
    <scope>runtime</scope>
  </dependency>
</dependencies>
```

这是它。 注意范围可以声明为运行时如果你 不需要编译与弹簧的api,这是典型的案例 对于基本依赖注入的用例。

我们使用Maven中央命名约定在示例 以上,因此,适用于Maven中央或SpringSource S3 Maven 存储库。 使用S3 Maven存储库(例如里程碑或 开发者快照),您需要指定存储库位置 您的Maven配置。 对于完整版本:

```
<repositories>
  <repository>
    <id>com.springsource.repository.maven.release</id>
    <url>http://repo.springsource.org/release/</url>
    <snapshots><enabled>false</enabled></snapshots>
  </repository>
</repositories>
```

为里程碑:

```
<repositories>
  <repository>
    <id>com.springsource.repository.maven.milestone</id>
    <url>http://repo.springsource.org/milestone/</url>
    <snapshots><enabled>false</enabled></snapshots>
  </repository>
</repositories>
```

和快照:

```
<repositories>
  <repository>
    <id>com.springsource.repository.maven.snapshot</id>
    <url>http://repo.springsource.org/snapshot/</url>
    <snapshots><enabled>true</enabled></snapshots>
  </repository>
</repositories>
```

利用电子束记录您需要SpringSource使用一个不同的 命名约定的依赖性。 名字通常是容易 猜,比如在本例中,它是:

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>org.springframework.context</artifactId>
    <version>3.0.0.RELEASE</version>
    <scope>runtime</scope>
  </dependency>
</dependencies>
```

你还需要声明库的位置 明确(只有URL是重要的):

```
<repositories>
  <repository>
    <id>com.springsource.repository.bundles.release</id>
    <url>http://repository.springsource.com/maven/bundles/release/</url>
  </repository>
```

```
</repositories>
```

如果你管理你的依赖手工中的URL 库声明以上不是浏览,但有一个用户 接口在 http://www.springsource.com/repository 可以用来搜索并下载依赖关系。 它也有 方便的片段的Maven和Ivy的配置,你可以复制和 如果您使用的是那些粘贴工具。

艾薇依赖管理

如果你喜欢使用 艾薇 管理依赖 然后有类似的名称和配置选项。

配置艾薇指SpringSource的电子束重熔添加 以下你解析器 ivysettings.xml :

```
<resolvers>

  <url name="com.springsource.repository.bundles.release">

    <ivy pattern="http://repository.springsource.com/ivy/bundles/release/
     [organisation]/[module]/[revision]/[artifact]-[revision].[ext]" />
    <artifact pattern="http://repository.springsource.com/ivy/bundles/release/
     [organisation]/[module]/[revision]/[artifact]-[revision].[ext]" />

  </url>

  <url name="com.springsource.repository.bundles.external">

    <ivy pattern="http://repository.springsource.com/ivy/bundles/external/
      [organisation]/[module]/[revision]/[artifact]-[revision].[ext]" />
    <artifact pattern="http://repository.springsource.com/ivy/bundles/external/
      [organisation]/[module]/[revision]/[artifact]-[revision].[ext]" />

  </url>

</resolvers>
```

上面的XML也是无效的,因为线太长——如果 你复制粘贴然后删除额外的行结束的中间 url模式。

一旦艾薇配置为看看电子束记录添加一个依赖项是 容易的。 只是拉起包的详细信息页面上的问题 存储库浏览器,你会发现一个常春藤片段,为你准备好 包括在你的依赖关系部分。 例如(在 流):

```
<dependency org="org.springframework"
    name="org.springframework.core" rev="3.0.0.RELEASE" conf="compile->runtime"/>
```

1.3.2A日志

日志是一个非常重要的依赖,因为它是春天) 唯一强制性外部依赖,b)每个人都喜欢看一些 输出从他们使用的工具,和c)弹簧集成了很多 其它的工具都还做了一个选择的日志 依赖项。 的一个目标应用程序开发人员通常是 有统一的日志配置在一个中心位置对整个 应用程序,包括所有的外部组件。 这是更加困难 比它可能是由于有如此多的选择的日志 框架。

在春天的强制性日志依赖性是Jakarta Commons 日志记录API(JCL)。 我们编译也使对JCL JCL 日志 对类对象可见扩展Spring框架。 它是重要的用户,所有版本的春天 使用相同的日志库:迁移是容易因为向后 兼容性是与应用程序保存甚至扩展Spring。 我们这样做是在春天的一个模块依赖 明确在 通用日志 (规范化实现 的JCL),然后让所有其他模块依赖,在编译 时间。 如果您正在使用Maven例如,和想知道你挑选 依赖的 通用日志 ,那么它就是来自于 春天和专门从中央模块称为 spring核心 。

的好处 通用日志 是你 不需要什么要让你的应用程序的工作。 它有一个运行时 发现算法,寻找其他日志框架在好 已知在类路径的地方,使用一个它认为是适当的 (或者你可以告诉它哪一个如果你需要)。 如果没什么 可你很漂亮的日志只是从JDK (java跑龙套。 日志记录或7月短)。 你会发现你的春天 应用程序和记录到控制台的幸福好了大多数 的情况,这很重要。

不使用通用日志

不幸的是,运行时发现算法 通用日志 ,虽然方便了终端用户,是 有问题的。 如果时间可以倒转,开始春天了 作为一个新项目将使用不同的日志依赖性。 这个 第一选择应该是简单的日志立面为Java(SLF4J),它也使用了很多 其它的工具,人们使用弹簧在他们应用程序。

关掉 通用日志 很简单:只是要吗 当然这不是在运行时类路径中。 在Maven条款排除 的依赖,因此,Spring依赖项 是说,你只需要做一次。

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>3.0.0.RELEASE</version>
    <scope>runtime</scope>
```

```
    <exclusions>
      <exclusion>
        <groupId>commons-logging</groupId>
        <artifactId>commons-logging</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
</dependencies>
```

现在这个应用程序可能是破碎的,因为没有 实现的类路径上的JCL API,所以解决这一新的一个 必须提供。 在接下来的部分中,我们向您展示如何提供一个 选择实施JCL SLF4J作为一个例子,使用

使用SLF4J

SLF4J是一个更清洁和更有效的在运行时依赖比 通用日志 因为它使用编译时绑定 而不是运行时发现的其他日志框架它 集成了。 这也意味着,你必须更明确的关于什么 你想发生在运行时,并宣布它或配置它 因此。 SLF4J提供绑定很多常见的日志框架,所以你通常可以选择一个你已经使用,并结合, 配置和管理。

SLF4J提供绑定很多常见的日志框架, 包括JCL,它也确实相反:桥梁在其他 日志框架和本身。 所以使用SLF4J与弹簧你需要 取代通用日志 依赖与SLF4J-JCL 桥。 一旦你做了,然后从内部日志调用春天 将被翻译成日志调用SLF4J API,所以如果其他吗 图书馆在你的应用程序使用该API,然后你有一个单独的位置 配置和管理日志。

一种常见的选择可能是春天SLF4J桥,然后 提供显式绑定从SLF4J Log4J。 你需要提供4 依赖性(和排除现有的 通用日志): 这座桥,SLF4J API,绑定到Log4J,Log4J 实现本身。 在Maven这样的你也会这么做的

```
<dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>3.0.0.RELEASE</version>
      <scope>runtime</scope>
      <exclusions>
        <exclusion>
          <groupId>commons-logging</groupId>
          <artifactId>commons-logging</artifactId>
        </exclusion>
      </exclusions>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>jcl-over-slf4j</artifactId>
      <version>1.5.8</version>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
      <version>1.5.8</version>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
      <version>1.5.8</version>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.14</version>
      <scope>runtime</scope>
    </dependency>
  </dependencies>
```

这可能看起来像一个许多依赖关系只是为了得到一些 日志记录。 嗯,就是这样,但它 是 可选的,它 应表现出比香草 通用日志 与类加载器问题的方面,特别是如果你是在一个严格的容器 像一个OSGi平台。 据说也有性能优势 因为绑定是在编译时不运行时。

一个更普遍的选择在SLF4J用户,它使用更少的步骤 和产生更少的依赖,是直接绑定 logback 。 这消除了额外的 绑定的步骤,因为Logback SLF4J直接实现,所以你只需要 依靠两个库不是四(jcl-over-slf4j 和 logback)。 如果你这样做,你可能还需要排除slf4j-api依赖从其他外部依赖(不是弹簧), 因为你只需要一个版本的API类路径上。

使用Log4j

许多人使用 log4j 作为一个记录 框架配置和管理的目的。 它是有效的 和完善的,事实上它就是我们使用在运行时当我们 构建和测试弹簧。 春天还提供了一些实用程序 Log4j配置和初始化,所以它有一个可选的编译时 在一些模块依赖Log4j。

使Log4j工作默认JCL依赖性（通用日志）你所需要做的就是把Log4j的 类路径,和为它提供一个配置文件（log4j . properties 或 log4j xml 在根 在类路径中)。 所以对于Maven用户这是你的依赖 声明:

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>3.0.0.RELEASE</version>
    <scope>runtime</scope>
  </dependency>
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.14</version>
    <scope>runtime</scope>
  </dependency>
</dependencies>
```

这里有一个样品log4j。 属性用于日志记录的 控制台:

```
log4j.rootCategory=INFO, stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %t %c{2}:%L - %m%n

log4j.category.org.springframework.beans.factory=DEBUG
```

运行时容器与本机JCL

许多人运行他们的Spring应用程序在一个容器 本身提供了一个实现JCL。 IBM Websphere Application 服务器(是)是原型。 这常常导致问题, 不幸的是没有银弹的解决方案;简单地排除 通用日志 从你的应用程序是不够的 大多数情况下。

要清楚这一点:报告的问题通常是不 与JCL本身,或即使有 通用日志 :而 他们都与绑定 通用日志 到另一个 框架(经常Log4J)。 这可能会因为 通用日志 改变他们的方式运行时 发现旧版本之间(1.0)发现在一些 容器和现代版本,大多数人现在使用(1.1)。 春天不使用任何不寻常的部分的JCL API,所以没有什么 在那里中断,但只要弹簧或应用程序试图做的 任何日志你可以发现绑定到Log4J不是 工作。

在这种情况下是最简单的事情就是转化 类加载器层次结构(IBM称之为“父母最后”)这样 应用程序控制着JCL依赖,而不是容器。 这 选择并不总是开放的,但也有很多其他的建议 在公共领域对替代方法,和你的里程 可能取决于确切的版本和特性集的吗集装箱。

PartA II。 一个什么新弹簧3

2。 一个新特性和增强功能在Spring框架3.0

如果你一直使用Spring框架对于一些时间,你会 知道春天已经经历了两个重大修改:Spring 2.0中发布 2006年10月,Spring 2.5,2007年11月发布。 这是现在的时间为 第三个检修导致Spring Framework 3.0。

Java SE和Java EE的支持

Spring框架是现在基于Java 5和Java 6是完全支持。

此外,春天是兼容J2EE 1.4和Java EE 5,而 同时引入一些早期支持Java EE 6。

2.1 Java 5

整个框架代码已被修改,以利用Java 5的特性(如泛型、可变参数和其他语言的改进。 我们有 尽了最大努力仍然保持向后兼容的代码。 我们现在有 坚持使用泛型集合和地图,坚持使用泛型 FactoryBeans,也一致的解决方法在桥梁 Spring AOP API。 通用ApplicationListeners自动接收 特定事件类型只有。 所有的回调接口如 TransactionCallback和HibernateCallback声明一个通用的结果值 现在。 总的来说,核心代码库的春天是现在刚修订和 优化了Java 5。

Spring的抽象TaskExecutor已经更新为关闭 整合与Java 5的Java跑龙套。 并发的设施。 我们提供 一流的支持Callables和期货现在,以及 ExecutorService适配器,ThreadFactory集成等。这是 符合jsr - 236(并发实用程序对Java EE 6)至于 可能的。 此外,我们提供支持异步方法 调用通过使用新@Async注释(或EJB 3.1的 asynchronous注释)。

2.2一个改进的文档

Spring参考文档还明显被 更新,以反映所有的变更和新特性对于Spring框架 3.0。 而所做的一切努力都以确保没有错误 这个文档,一些错误可能不过已经爬了进去。 如果你 点任何拼写错误或更严重的错误,你可以抽出几个周期 在午餐,请把错误的注意弹簧 团队的 抚养一个 问题 。

2.3一个新的文章和教程

有许多优秀的文章和教程,展示如何得到 开始使用Spring框架3特性。 读他们的 弹簧文档 页面。

样品已经被改进和更新,以利用新特性在春天 框架3。 另外,样品已经搬出了源代码树到一个专门的SVN 库 可以在:

https://anonsvn.springframework.org/svn/spring-samples/

因此,样品都不再分布式和弹簧 框架3,需要单独下载从存储库 上面提到的。 然而,这个文档将继续参考 一些样品(特别是宠物诊所)来说明各种特性。

注意

为更多的信息在Subversion(SVN或短),请参见项目主页: http://subversion.apache.org/

2.4新模块的组织和构建系统

该框架模块已经修改,现在管理 一个源代码树分别与每个模块jar:

- org.springframework.aop
- org.springframework.beans
- org.springframework.context
- org.springframework.context.support
- org.springframework.expression
- org.springframework.instrument
- org.springframework.jdbc
- org.springframework.jms
- org.springframework.orm
- org.springframework.oxm
- org.springframework.test
- org.springframework.transaction
- org.springframework.web
- org.springframework.web.portlet
- org.springframework.web.servlet
- org.springframework.web.struts

We are now using a new Spring build system as known from Spring Web Flow 2.0. This gives us:

- Ivy-based "Spring Build" system
- consistent deployment procedure
- consistent dependency management
- consistent generation of OSGi manifests

Note:

The spring.jar artifact that contained almost the entire framework is no longer provided.

2.5 Overview of new features

This is a list of new features for Spring Framework 3.0. We will cover these features in more detail later in this section.

- Spring Expression Language
- IoC enhancements/Java based bean metadata
- General-purpose type conversion system and field formatting system
- Object to XML mapping functionality (OXM) moved from Spring Web Services project
- Comprehensive REST support
- @MVC additions
- Declarative model validation
- Early support for Java EE 6
- Embedded database support

2.5.1 Core APIs updated for Java 5

BeanFactory interface returns typed bean instances as far as possible:

- T getBean(Class<T> requiredType)
- T getBean(String name, Class<T> requiredType)
- Map<String, T> getBeansOfType(Class<T> type)

Spring's TaskExecutor interface now extends java.util.concurrent.Executor:

- extended AsyncTaskExecutor supports standard Callables with Futures

New Java 5 based converter API and SPI:

- stateless ConversionService and Converters
- superseding standard JDK PropertyEditors

Typed ApplicationListener<E>

2.5.2 Spring Expression Language

Spring introduces an expression language which is similar to Unified EL in its syntax but offers significantly more features. The expression language can be used when defining XML and Annotation based bean definitions and also serves as the foundation for expression language support across the Spring portfolio. Details of this new functionality can be found in the chapter Spring Expression Language (SpEL).

The Spring Expression Language was created to provide the Spring community a single, well supported expression language that can be used across all the products in the Spring portfolio. Its language features are driven by the requirements of the projects in the Spring portfolio, including tooling requirements for code completion support within the Eclipse based SpringSource Tool Suite。

下面是一个示例,说明了表达式语言可以 用于配置数据库设置的一些性质

```
<bean class="mycompany.RewardsTestDatabase">
   <property name="databaseName"
      value="#{systemProperties.databaseName}"/>
   <property name="keyGenerator"
      value="#{strategyBean.databaseKeyGenerator}"/>
</bean>
```

这个功能也可以如果你喜欢配置 你的组件使用注释:

```
@Repository
public class RewardsTestDatabase {

   @Value("#{systemProperties.databaseName}")
   public void setDatabaseName(String dbName) { ... }

   @Value("#{strategyBean.databaseKeyGenerator}")
   public void setKeyGenerator(KeyGenerator kg) { ... }
}
```

2.5.3A的控制反转(IoC)容器

基于Java bean的元数据

一些核心的功能 JavaConfig 项目已经被添加到Spring框架现在。 这意味着 下面的注释是现在直接支持:

- @ configuration
- @ bean
- @DependsOn
- @Primary
- @Lazy
- @ import
- @ImportResource
- 这个元素包含一个@ value

下面的示例是一个Java类提供基本配置 使用新的JavaConfig特点:

```
package org.example.config;

@Configuration
public class AppConfig {
    private @Value("#{jdbcProperties.url}") String jdbcUrl;
    private @Value("#{jdbcProperties.username}") String username;
    private @Value("#{jdbcProperties.password}") String password;

    @Bean
    public FooService fooService() {
        return new FooServiceImpl(fooRepository());
    }

    @Bean
    public FooRepository fooRepository() {
        return new HibernateFooRepository(sessionFactory());
    }

    @Bean
    public SessionFactory sessionFactory() {
        // wire up a session factory
        AnnotationSessionFactoryBean asFactoryBean =
            new AnnotationSessionFactoryBean();
        asFactoryBean.setDataSource(dataSource());
        // additional config
        return asFactoryBean.getObject();
    }

    @Bean
    public DataSource dataSource() {
        return new DriverManagerDataSource(jdbcUrl, username, password);
    }
}
```

得到这个工作你需要添加以下组件 扫描进入你的最小应用程序上下文XML文件。

```
<context:component-scan base-package="org.example.config"/>
<util:properties id="jdbcProperties" location="classpath:org/example/config/jdbc.properties"/>
```

或者你可以引导一个 @ configuration 类直接使用 所 :

```
public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);
    FooService fooService = ctx.getBean(FooService.class);
    fooService.doStuff();
}
```

看到 SectionA 5 12 2,一个Spring容器实例化使用 所 一个 完全信息 所 。

元数据定义bean组件内

@ bean 也支持带注释的方法 在弹簧组件。 他们使工厂bean定义 容器。 看到 bean定义元数据在 组件 为更多的信息

2.5.4A通用类型转换系统和字段格式 系统

一个通用 类型转换 系统 介绍了。 该系统目前使用? 类型转换,也可以被用于一个Spring容器和DataBinder当 绑定bean属性值。

此外,一个 格式化程序 SPI 介绍了格式字段值。 这SPI提供 一个更简单和更健壮的替代JavaBean PropertyEditors用于客户端环境如Spring MVC。

2.5.5A数据层

对象到XML的映射功能(OXM)从Spring Web 服务项目已经搬到核心的Spring框架现在。 这个 功能是发现的org.springframework.oxm 包。 更多的信息的使用 OXM 模块中可以找到 编组XML使用O / X 映射器 一章。

2.5.6A Web层

最令人兴奋的新功能是支持Web层 构建RESTful web服务和web应用程序。 也有一些 新的注释,可以用在任何web应用程序。

综合REST支持

服务器端支持构建RESTful应用程序一直 作为一个扩展现有的注解驱动的MVC web 框架。 客户端支持提供的 RestTemplate 类的精神,其他 模板类,比如 JdbcTemplate 和 JmsTemplate 。 服务器和客户端都休息 功能利用 HttpConverter 年代,以促进之间的转换对象和他们的代表在HTTP请求 和响应。

这个 MarshallingHttpMessageConverter 使用 这个 对象到XML的映射 功能提到 早期。

参考的章节 MVC 和 RestTemplate的 更多 信息。

@MVC添加

一个 MVC 名称空间被引入,这极大地简化了Spring MVC的配置。

额外的注释(如 @CookieValue 和 @RequestHeaders 已添加。 看到 映射的cookie的值 @CookieValue注释 和 映射属性与请求头 @RequestHeader注释的 为更多的信息。

2.5.7A声明式模型验证

几个 验证改进 , 包括JSR 303支持,使用Hibernate Validator作为默认提供者。

2.5.8A早期支持Java EE 6

我们提供支持异步方法调用通过 使用新@Async注释(或EJB 3.1的asynchronous 注释)。

JSR 303,JSF 2.0,JPA 2.0等

2.5.9A支持嵌入式数据库

方便支持 嵌入式Java数据库 引擎 ,包括HSQL,H2,德比,现在提供。

3。 一个新特性和增强功能在Spring框架3.1

这是一个列表的新功能,为Spring Framework 3.1。 一个数量的特性 没有专门的参考文档,但确实有完整的Javadoc。 在这样的 情况下,给出了完全限定的类名。 看到也 AppendixA C, 迁移到Spring Framework 3.1

3.1一个缓存抽象

- ChapterA 29日 缓存抽象
- 缓存抽象 (SpringSource团队博客)

3.2一个Bean定义概要文件

- XML配置文件 (SpringSource团队博客)
- 引入@Profile (SpringSource团队博客)
- 看到org.springframework.context.annotation.Configuration Javadoc
- 看到org.springframework.context.annotation.Profile Javadoc

3.3一个环境抽象

- 环境抽象 (SpringSource团队博客)

- 看到org.springframework.core.env。 环境Javadoc

3.4一个PropertySource抽象

- 统一物业管理 (SpringSource团队博客)
- 看到org.springframework.core.env。 环境Javadoc
- 看到org.springframework.core.env。 PropertySource Javadoc
- 看到org.springframework.context.annotation.PropertySource Javadoc

3.5代码等价的Spring的XML名称空间

基于代码的等价物,流行的Spring XML名称空间的元素 <上下文:组件扫描/ >,< tx:注解驱动/ > 和< mvc:注解驱动>已经开发出来,大多数在 形式的 @Enable 注释。 这些都是 设计用于结合春天的 @ configuration 类,这在 介绍了Spring框架3.0。

- 看到org.springframework.context.annotation.Configuration Javadoc
- 看到org.springframework.context.annotation.ComponentScan Javadoc
- 看到 org.springframework.transaction.annotation.EnableTransactionManagement Javadoc
- 看到 org.springframework.cache.annotation。 EnableCaching Javadoc
- 看到org.springframework.web.servlet.config.annotation.EnableWebMvc Javadoc
- 看到org.springframework.scheduling.annotation.EnableScheduling Javadoc
- 看到org.springframework.scheduling.annotation.EnableAsync Javadoc
- 看到 org.springframework.context.annotation.EnableAspectJAutoProxy Javadoc
- 看到 org.springframework.context.annotation.EnableLoadTimeWeaving Javadoc
- 看到 org.springframework.beans.factory.aspectj.EnableSpringConfigured Javadoc

3.6一个支持Hibernate 4. x

- 看到在新的类的Javadoc org.springframework.orm。 hibernate4包

3.7一个还是和TestContext框架支持@ configuration类和bean 定义概要文件

这个 @ContextConfiguration 注释现在支持提供 @ configuration 类配置 春天 还是和TestContext 。 此外,一个新的 @ActiveProfiles 注释已经 介绍了支持声明式配置的活跃bean 定义概要文件在 ApplicationContext 集成测试。

- 春天 3.1平方米:测试与@ configuration类和配置文件 (SpringSource团队博客)
- 看到 SectionA 11 3 5,一个春天还是和TestContext Frameworka
- 看到 一个章节上下文配置使用带注释的classesa 和 org.springframework.test.context.ContextConfiguration Javadoc
- 看到 org.springframework.test.context.ActiveProfiles Javadoc
- 看到 org.springframework.test.context.SmartContextLoader Javadoc
- 看到 org.springframework.test.context.support.DelegatingSmartContextLoader Javadoc
- 看到 org.springframework.test.context.support.AnnotationConfigContextLoader Javadoc

3.8一个c:名称空间更简洁的构造函数注入

- 一个章节XML与c-namespacea快捷方式

3.9一个支持对非标准JavaBeans注射 setter

Spring Framework 3.1之前,为了对产权注入 方法必须严格符合javabean属性签名规则, 即任何“setter”方法必须返回void。 现在可以 在Spring XML指定setter方法,返回任何对象类型。 这 是有用的在考虑设计api方法链接,在哪里 setter方法返回一个引用“本”。

3.10一个支持Servlet 3基于代码的配置Servlet 容器

新 WebApplicationInitializer 构建在Servlet 3.0的 ServletContainerInitializer 支持 提供一个程序化的替代传统的web . xml。

- 看到org.springframework.web.WebApplicationInitializer Javadoc
- Diff从春天的 温室参考应用 演示迁移 从web。 xml WebApplicationInitializer

3.11一个支持Servlet 3 MultipartResolver

- 看到 org.springframework.web.multipart.support.StandardServletMultipartResolver Javadoc

3.12一个JPA会引导没有 persistence . xml

在标准JPA,持久性单元得到定义通过 meta - inf / persistence . xml 文件在特定的jar文件 这将反过来得到搜索吗 entity 类。 在许多情况下,持久性。 xml不包含超过一个单位的名字 和依赖于违约和/或外部设置为所有其他问题 (如数据源使用,等等)。 出于这个原因,Spring框架 3.1提供了一个替代方案: LocalContainerEntityManagerFactoryBean 接受一个 “packagesToScan” 属性,指定基础包扫描 entity 类。 这是类似于 AnnotationSessionFactoryBean 的财产 相同的名称为土著Hibernate的设置,同时弹簧的 组件扫描功能,定期Spring bean。 实际上,这 允许xml自由JPA设置仅仅是在牺牲指定一个基地 包对于实体扫描:一个特别好的适合春天 应用程序依赖于组件扫描为Spring bean, 甚至可能引导使用基于Servlet 3.0 初始化器。

3.13新HandlerMethod-based支持类注释的控制器 处理

Spring Framework 3.1引入了一组新的支持类 处理请求使用带注释的控制器:

- RequestMappingHandlerMapping
- RequestMappingHandlerAdapter
- ExceptionHandlerExceptionResolver

这些类是替代现有的:

- DefaultAnnotationHandlerMapping
- AnnotationMethodHandlerAdapter
- AnnotationMethodHandlerExceptionResolver

新类是回应了许多要求 让注释控制器支持类更多的定制和开放 对扩展。 而以前可以配置一个自定义注释 控制器方法参数解析器,新的支持类 可以定制加工为任何受支持的方法参数或返回吗 值类型。

- 看到 org.springframework.web.method.support.HandlerMethodArgumentResolver Javadoc
- 看到 org.springframework.web.method.support.HandlerMethodReturnValueHandler Javadoc

第二个显著的区别是引入一个 HandlerMethod 抽象代表一个 @RequestMapping 法。 这种抽象是使用 在新支持类的 处理程序 实例。 例如一个 HandlerInterceptor 可以 把 处理程序 从 对象 到 HandlerMethod 和获得目标 控制器方法,其注释等。

新类是默认启用的MVC名称空间和通过 基于java的配置通过 @EnableWebMvc 。 这个 现有类持续可用但使用新的 类是推荐的前进。

看到 一个章节新的支持类 @RequestMapping 方法在一个Spring MVC 3.1 对于额外的 细节和一个列表的功能可能无法与新的支持类。

3.14 “消费” 和 “生产” 条件 @RequestMapping

改进支持指定媒体类型所消耗的方法 通过 ” 内容类型的 头以及 可生产的类型指定的通过 “接受” 头。 看到 一个章节消耗品媒体Typesa 和 一个章节Typesa可生产的媒体

3.15一个Flash属性和 RedirectAttributes

Flash属性现在可以被存储在一个 FlashMap 并保存在HTTP会话生存 一个重定向。 概述的一般支持flash的属性 在Spring MVC看到 SectionA 17.6,一个attributesa使用闪光灯 。

在注释的控制器,一个 @RequestMapping 方法可以添加闪光 属性声明一个方法的参数类型 RedirectAttributes 。 这个方法参数 现在还可以被用来得到精确控制属性用于 一个重定向的场景。 看到 一个章节指定attributesa重定向和闪光 为更多的细

节。

3.16一个URI模板变量增强

URI模板变量从当前请求中使用更多 的地方:

- URI模板变量是使用除了请求 当一个请求参数绑定 @ModelAttribute 方法 参数。
- @PathVariable方法参数值合并成 模型渲染之前,除了视图生成的内容 一个自动化的时尚如JSON序列化或XML 编组。
- 一个重定向为URI字符串可以包含占位符变量 (如。 “重定向:/博客/ {年} / {月}”)。 当 扩大占位符,URI模板变量 当前请求被自动考虑。
- 一个 @ModelAttribute 方法 参数可以实例化一个URI模板变量提供 有一个注册转换器或属性编辑器将从 一个字符串到目标对象类型。

3.17一个 @Valid 在 @RequestBody 控制器方法参数

一个 @RequestBody 方法参数可以 标注 @Valid 调用自动 验证类似于支持 @ModelAttribute 方法参数。 一个结果 MethodArgumentNotValidException 处理 DefaultHandlerExceptionResolver 和结果在一个 400年 响应代码。

3.18一个 @RequestPart 注释 控制器方法参数

这个新的注释提供了访问的内容 “多部分/格式数据“请求部分。 看到 SectionA 17 10 5,一个处理文件上传请求从编程clientsa 和 SectionA 17.10,一个弹簧的多部分(文件上传)supporta 。

3.19一个 UriComponentsBuilder 和 UriComponents

一个新的 UriComponents 类已添加, 这是一个不可变的容器组件的URI提供吗 访问所有包含URI组件。 一个新的 UriComponentsBuilder 类也 提供帮助创建 UriComponents 实例。 在一起的两个类给细粒度控制所有 方面的准备一个URI包括建设、扩张 从URI模板变量,和编码。

在大多数情况下,新类可以作为一个更灵活 替代现有的 UriTemplate 特别是 UriTemplate 依赖那些 同一类内部。

一个 ServletUriComponentsBuilder 子类 提供静态工厂方法复制信息 一个Servlet请求。 看到 SectionA 17.7,一个建筑 URI sa 。

4。 一个新特性和增强功能在Spring框架3.2

这一节将介绍什么是新的在Spring框架3.2。 看到也 AppendixA D, 迁移到Spring Framework 3.2

4.1一个支持Servlet 3基于异步请求处理

Spring MVC的编程模型现在提供明确的Servlet 3 异步支持。 @RequestMapping 方法可以 返回一个:

- java util并发调用 到 完整的处理在一个单独的线程管理任务执行人 在Spring MVC。
- org.springframework.web.context.request.async.DeferredResult 完成处理在稍后的时间从一个线程不了解 一个Spring MVC例如,在应对一些外部事件(JMS, AMQP等)。
- org.springframework.web.context.request.async.AsyncTask 包装 可调用的 和定制 超时值或任务执行器使用。

看到 SectionA 17 3 4,一个Processinga异步请求 。

4.2一个Spring MVC的测试框架

一流的支持Spring MVC应用程序测试用 流利的API和没有一个Servlet容器。 服务器端测试涉及使用 的 DispatcherServlet 而客户端休息 测试依赖 RestTemplate 。 看到 SectionA 11 3 6,一个测试FrameworkaSpring MVC 。

4.3一个内容协商改进

一个 ContentNegotiationStrategy 现在 用于解决从传入请求媒体类型 请求。 可用的实现是基于文件扩展名, 查询参数, “接受” 标题,或一个固定的内容类型。 等效选项以前只能在 ContentNegotiatingViewResolver但现在遍及。

ContentNegotiationManager 是中央 类使用在配置选项内容协商。 更多细节见 SectionA 17 15 4,一个配置内容Negotiationa 。

引入 ContentNegotiationManger 还使选择性后缀模式匹配为传入的请求。 有关详细信息,请参见的Javadoc RequestMappingHandlerMapping.setUseRegisteredSuffixPatternMatch 。

4.4一个 @ControllerAdvice 注释

类标注 @ControllerAdvice 可以包含 @ExceptionHandler , @InitBinder ,和 @ModelAttribute 方法和那些将 适用于 @RequestMapping 方法在 控制器层次相对于控制器层次在其中 他们宣布。 @ControllerAdvice 是 组件注释允许实现类自动 通过类路径扫描。

4.5一个矩阵变量

一个新的 @MatrixVariable 注释添加 支持提取矩阵变量从请求URI。 更多 细节见 一个章节Variablesa矩阵 。

4.6一个抽象基类,用于基于Servlet 3 +容器 初始化

一个抽象的基类的实现 WebApplicationInitializer 接口是 提供简化基于代码注册DispatcherServlet和 过滤器映射到它。 这个新类命名 AbstractDispatcherServletInitializer 和它的 子类 AbstractAnnotationConfigDispatcherServletInitializer 可以使用基于java的Spring配置。 更多细节见 SectionA 17.14,一个initializationa基于代码的Servlet容器 。

4.7一个 ResponseEntityExceptionHandler 类

一个方便的基类的 @ExceptionHandler 方法处理 标准的Spring MVC异常并返回 ResponseEntity 这允许定制和 编写响应HTTP消息转换器。 这可以作为一种 替代 DefaultHandlerExceptionResolver , 这确实相同的但返回吗 ModelAndView 相反。

看到修改后的 SectionA 17.11,一个处理exceptionsa 包括 信息定制默认Servlet容器的错误 页面。

4.8对泛型的支持的 RestTemplate 在 @RequestBody 参数

这个 RestTemplate 现在可以读一个HTTP吗 回应一个泛型类型(如。 列表<帐户>)。 有 三个新 交换() 方法,它接受 ParameterizedTypeReference ,一个新类 使捕获和传递泛型类型信息。

支持这个特性, HttpMessageConverter 是延长 GenericHttpMessageConverter 添加一个方法 阅读内容给指定的参数化的类型。 新 接口实现的 MappingJacksonHttpMessageConverter 也会被一个 新 Jaxb2CollectionHttpMessageConverter 这可以 阅读读一个通用的 收集 在 泛型类型是JAXB注释类型 @XmlRootElement 或 @XmlType 。

4.9一个杰克逊JSON 2和相关的改进

杰克逊JSON 2图书馆现在支持。 由于包装 变化在杰克逊的图书馆,有单独的类在Spring MVC 为好。 那些是 MappingJackson2HttpMessageConverter 和 MappingJackson2JsonView 。 其他相关 配置的改进包括支持漂亮的印刷等一个 JacksonObjectMapperFactoryBean 为了方便 定制的 ObjectMapper 在XML 配置。

4.10一个瓷砖3

现在支持瓷砖3除了瓷砖2. x。 配置 它应该非常类似于瓷砖2配置,即 结合 TilesConfigurer , TilesViewResolver 和 TilesView 除了使用 tiles3 而不是 tiles2 包。

还请注意,除了版本号变化,瓷砖 依赖关系也发生了变化。 你需要有一个子集或全部 瓦片请求api , 瓷砖api , 瓦片核心 , 瓷砖servlet , 瓷砖jsp , 瓷砖el 。

4.11一个 @RequestBody 改进

一个 @RequestBody 或一个 @RequestPart 参数可以遵循 通过一个 错误 参数成为可能 处理验证错误(由于一个 @Valid 注释)内的本地 @RequestMapping 法。 @RequestBody 现在还支持一个必需的 国旗。

4.12一个HTTP补丁方法

HTTP请求方法 补丁 现在可能用在吗 @RequestMapping 方法以及 RestTemplate 结合Apache HttpComponents HttpClient 4.2或更高版本。 JDK HttpURLConnection 不支持 补丁 法。

4.13一个排除模式映射的拦截器

拦截器现在支持URL模式映射到被排除在外。 MVC 名称空间和MVC JavaConfig两暴露这些选项。

4.14一元注释使用注射点和bean定义方法

为3.2,春天允许 @ autowired 和 这个元素包含一个@ value 作为元注释, 如建立自定义注入注解结合特定的限定符。 类似地,您可以构建定制的 @ bean 定义 注释 @ configuration 类, 如结合特定的限定符,@Lazy,@Primary等等。

4.15一个初始支持JCache 0.5

Spring提供了一种缓存管理器适配器为JCache、建筑与JCache 0.5 预览版。 全JCache支持是在明年,以及Java EE 7最后。

4.16一个支持 @DateTimeFormat 没有 Joda时间

这个 @DateTimeFormat 注释可以 现在被使用而无需依赖Joda时间图书馆。 如果Joda 时间不存在JDK SimpleDateFormat 将 被用来解析和打印日期模式。 当Joda时间是现在 将继续被用于优先 SimpleDateFormat 。

4.17全球日期和时间格式

现在可以定义全局格式的时候将会使用 解析和打印日期和时间类型。 看到 SectionA 7.7,一个配置全球formata日期与时间 对于 细节。

4.18一个新的测试功能

除了上述的包容的 Spring MVC的测试框架 在 这个 弹簧测试 模块, 春天 还是和TestContext框架 已被修订,支持吗 集成测试web应用程序和配置应用程序 与上下文初始化上下文。 为进一步的细节,请参考 以下。

- 配置和 加载一个 WebApplicationContext 在集成测试
- 配置 上下文层次 在集成测试
- 测试 请求和 会话作用域bean
- 改进 Servlet API 模拟
- 配置测试应用程序上下文与 ApplicationContextInitializers

4.19一个并发细化整个框架

Spring Framework 3.2包括微调的并发数据结构 在许多部分的框架,最小化锁和普遍提高 安排高并发创建范围/原型豆子。

4.20新gradle基础构建和搬到GitHub

建筑和导致从未简单的框架与 我们搬到一个基于gradle构建系统和源控制在GitHub上。 看到 从源代码构建 部分的自述和 贡献者指南 完整的细节。

4.21一个精制Java SE 7 / OpenJDK 7支持

最后但并非最不重要的是,Spring框架3.2附带精制Java 7的支持 在框架以及通过升级第三方依赖关系: 具体来说,CGLIB 3.0,ASM 4.0(两个来作为内联依赖关系与 春天现在)和AspectJ 1.7支持(下一个现有的AspectJ 1.6支持)。

PartA三世。 一个核心技术

这部分的参考文档涵盖了所有这些 技术是绝对不可或缺的春天 框架。

首先在这些是Spring框架的反转 控制(IoC)容器。 彻底治疗的Spring框架的 IoC容器是紧随其后的是春天的全面覆盖的 面向方面的编程(AOP)技术。 Spring框架已经 自己的AOP框架,它在概念上很容易理解, 成功地解决了80%的甜蜜点的AOP的要求吗 Java企业编程。

覆盖Spring的集成与AspectJ(目前 富有-从功能——当然,最成熟的AOP 实现在Java企业空间)也提供了。

最后,采用测试驱动开发(TDD) 软件开发方法当然是倡导的春天 团队,所以覆盖率Spring的支持集成测试 覆盖(与单元测试的最佳实践)。 春季团队 发现,正确使用IoC当然会让两个单位和 集成测试容易(在的存在和setter方法 适当的构造函数的类会使得他们更容易连接在一起 在一个测试而无需注册和设置服务定位器 诸如此类的)..... 这一章专门测试将有希望 说服你的这种。

- ChapterA 5, IoC容器
- ChapterA 6, 资源
- ChapterA 7, 验证、数据绑定、类型转换
- ChapterA 8, 春天表达式语言(?)
- ChapterA 9, 面向方面的编程与弹簧
- ChapterA 10, Spring AOP api
- ChapterA 11, 测试

5。 一个IoC容器

5.1一个介绍Spring IoC容器和豆类

这一章介绍了Spring框架的实现 控制反转(IoC) [1] 原理。 国际奥委会也被称为 依赖 注入 (DI)。 这是一个过程,即对象定义他们 依赖关系,即其他对象合作,只有通过 构造函数参数,参数到一个工厂方法或属性 上设置的对象实例构造或后回来吗 工厂方法。 容器然后 注入 那些 在创建bean的依赖项。 这一过程基本上是这个 逆,因此得名 控制反转 (IoC), bean本身的控制实例化或位置的 通过使用直接建设依赖的类,或一个机制这样的 随着 服务定位器 模式。

这个 org.springframework.beans 和 org.springframework.context 包是基础 Spring框架的IoC容器。 这个 BeanFactory 接口提供了一个先进的 配置机制能够处理任何类型的对象。 ApplicationContext 是一个接头界面的 BeanFactory。 它会更容易集成 与Spring的AOP功能;信息资源处理(用于 国际化),事件发布;和应用程序层具体 上下文如 WebApplicationContext 在web应用程序中使用。

简而言之, BeanFactory 提供 配置框架和基本功能,以及 ApplicationContext 添加更多 特定的功能。 这个 ApplicationContext 是一个完整的超集 的 BeanFactory ,是专门用于 在这一章在描述Spring的IoC容器。 对于 更多的信息在使用 BeanFactory 相反 的 ApplicationContext, 指 SectionA 5.15,一个BeanFactorya的 。

在春天,对象,形成的主干应用程序和 这由Spring IoC 容器 是 称为 bean 。 一个bean是一个对象,是 实例化,装配,否则一个 Spring IoC容器管理的。 否则,一个bean是一个简单的许多对象在您的应用程序。 豆, 依赖性 其中,是 反映在 配置元数据 使用集装箱。

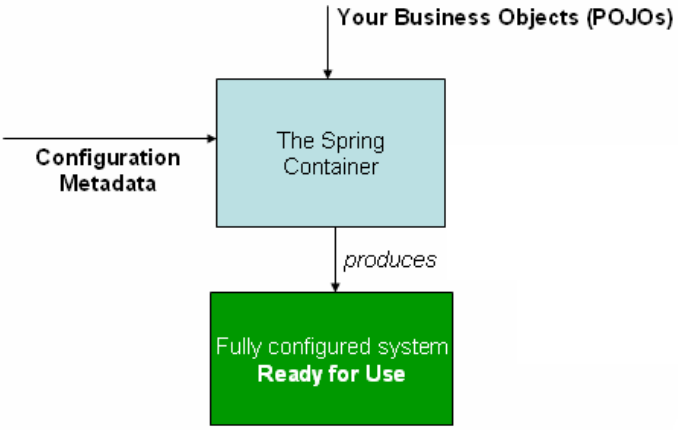
5.2一个集装箱概述

接口 org.springframework.context.ApplicationContext 代表了Spring IoC容器和负责实例化, 配置和组装上述豆子。 容器被 它的指令对象实例化、配置和组装 通过读取配置元数据。 配置元数据是 用XML表示,Java注释,或Java代码。 它允许您 表达的对象构成应用程序和丰富 这样的对象之间的相互依赖关系。

的几种实现 ApplicationContext 接口提供 与春天的开箱即用。 在独立应用程序中是很常见的 创建一个实例, ClassPathXmlApplicationContext 或 FileSystemXmlApplicationContext 。 虽然XML被传统的格式 定义配置元数据你可以通知容器来使用 Java注释或代码的元数据格式,提供少量 XML配置声明支持这些额外的 元数据格式。

在大多数应用程序场景,明确用户代码不需要 实例化一个或多个实例的Spring IoC容器。 例如, 在一个web应用程序场景中,一个简单的八(左右)行样板 J2EE web描述符中的XML web . xml 文件的 应用程序通常会足够了(参见 SectionA 5 14 4,一个方便 ApplicationContext 为web applicationsa实例化)。 如果你正在使用 SpringSource工具套件 eclipse动力开发环境 或 Spring Roo 这 样板配置可以很容易地创建几个鼠标点击或 击键。

下面的图是如何工作的高级视图春天。 你 应用程序类是结合配置元数据,以便之后 这个 ApplicationContext 创建和初始化, 你有一个完全配置和可执行的系统或应用程序。



Spring IoC容器

5.2.1A配置元数据

作为前面的图表显示,Spring IoC容器消耗 形式的 配置元数据 ,这个配置 元数据代表如何作为应用程序开发人员告诉Spring 容器实例化、配置和组装中的对象 应用程序。

配置元数据通常是在一个简单而提供 直观的XML格式,这是本章的大部分用来传达 关键的概念和特性Spring IoC容器。

> **注意**
>
> 基于xml的元数据是 不 唯一允许 形式的配置元数据。 Spring IoC容器本身 完全 脱离了格式,这 配置元数据实际上是写。

信息使用其他形式的元数据与春天 集装箱,请参阅:

- 基于注解的 配置 :Spring 2.5引入了支持 基于注解的配置元数据。
- 基于java的配置 : 从Spring 3.0,提供的许多特性 Spring JavaConfig 项目 成为Spring框架核心的一部分。 因此你 可以定义bean类的外部应用程序使用Java 而不是XML文件。 使用这些新特性,请参阅 @ configuration , @ bean, @ import 和 @DependsOn 注释。

Spring配置由至少一个,通常更多 比一个bean定义的容器必须管理。 基于xml的 配置元数据显示这些bean配置为 < bean / > 元素在一个顶级 < bean / > 元素。

这些bean定义对应于实际的对象构成 您的应用程序。 通常你定义服务层对象,数据 访问对象(DAOs)、表示对象(如Struts 行动情况下,基础设施对象 比如Hibernate SessionFactories 、JMS 队列 ,等等。 通常是 没有配置细粒度的域对象的容器,因为它通常是负责DAOs和业务逻辑创建和 负载的域对象。 然而,你可以使用Spring的集成 AspectJ来配置对象创建的无法控制 一个IoC容器。 看到 使用 AspectJ依赖注入与春天的域对象 。

下面的例子显示了xml的基本结构 配置元数据:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="..." class="...">
    <!-- collaborators and configuration for this bean go here -->
  </bean>

  <bean id="..." class="...">
    <!-- collaborators and configuration for this bean go here -->
  </bean>

  <!-- more bean definitions go here -->

</beans>
```

这个 id 属性是一个字符串,你使用 识别个人的bean定义。 这个 类 属性定义了类型的bean和使用完全限定 类名。 id属性的值是指合作 对象。 XML为指协作对象没有显示在 这个例子,看到 依赖性 为更多的信息。

5.2.2A实例化一个集装箱

实例化一个Spring IoC容器是直截了当的。 这个 位置路径或路径提供给一个 ApplicationContext 构造函数是 实际资源字符串,使容器加载配置 元数据从各种各样的外部资源,如本地文件 系统,从Java 类路径 ,等等。

```
ApplicationContext context =
    new ClassPathXmlApplicationContext(new String[] {"services.xml", "daos.xml"});
```

注意

在你了解Spring的IoC容器,您可能想知道 更多关于春天的 资源 抽象,如前文所述 ChapterA 6, 资源 ,这 提供了一个方便的机制来阅读InputStream从 位置中定义的URI语法。 特别是, 资源 路径用于构造 应用程序上下文所描述的那样 SectionA 6.7,一个应用程序上下文和 资源 pathsa 。

下面的例子显示了服务层对象 (services . xml) 配置文件:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans.xsd">

  <!-- services -->

  <bean id="petStore"
       class="org.springframework.samples.jpetstore.services.PetStoreServiceImpl">
    <property name="accountDao" ref="accountDao"/>
    <property name="itemDao" ref="itemDao"/>
    <!-- additional collaborators and configuration for this bean go here -->
  </bean>

  <!-- more bean definitions for services go here -->

</beans>
```

下面的例子显示了数据访问对象 daos xml 文件:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="accountDao"
     class="org.springframework.samples.jpetstore.dao.ibatis.SqlMapAccountDao">
    <!-- additional collaborators and configuration for this bean go here -->
  </bean>

  <bean id="itemDao" class="org.springframework.samples.jpetstore.dao.ibatis.SqlMapItemDao">
    <!-- additional collaborators and configuration for this bean go here -->
  </bean>

  <!-- more bean definitions for data access objects go here -->

</beans>
```

在前面的示例中,服务层包含的类 PetStoreServiceImpl 和两个数据访问对象 类型的 SqlMapAccountDao 和 SqlMapItemDao 基于 iBatis 对象/关系映射框架。 这个 财产 名称 元素指的JavaBean属性的名称,和 这个 Ref 元素指的是另一个bean的名称 定义。 这种关联id和ref元素表达了 协作对象之间的依赖关系。 对于细节的配置 对象的依赖项,见 依赖性 。

组成的基于xml的配置元数据

它可以有用bean定义跨越多个XML文件。 通常每个单独的XML配置文件中代表一个逻辑 层或模块在您的体系结构。

您可以使用应用程序上下文构造函数加载bean 定义所有这些XML片段。 这个构造函数 多个 资源 位置, 如上一节所示。 或者,使用一个或多个 退运 <进口/ > 元素加载 从另一个档案或bean定义文件。 例如:

```
<beans>

   <import resource="services.xml"/>
   <import resource="resources/messageSource.xml"/>
   <import resource="/resources/themeSource.xml"/>

   <bean id="bean1" class="..."/>
   <bean id="bean2" class="..."/>
```

```
</beans>
```

在前面的示例中,外部加载bean定义 从三个文件, services . xml , messageSource.xml ,和 themeSource.xml 。 所有的位置路径指的是相对于 定义文件做进口,所以 services . xml 必须在同一目录或 类路径位置文件做进口,而 messageSource.xml 和 themeSource.xml 必须在一个 资源 下面的位置的位置 导入文件。 正如您可以看到的,一个领先的削减是忽视,但鉴于 这些路径是相对而言的,它是更好的形式不使用斜杠 在所有。 文件的内容被导入,包括顶部 水平 < bean / > 元素,必须是有效的XML bean定义根据弹簧模式或DTD。

注意

它是可能的,但不推荐,参考文件 父目录使用一个相对 “. . / “路径。 这样创建一个 依赖一个文件,是当前应用程序之外。 在 特别的,这种引用是不推荐用于 “类路径: “url (例如, “类路径:. . / services . xml”),运行时 解决程序选择 “最近的 “类路径根,然后 看着它的父目录。 类路径配置的变化可能 导致一个不同的选择,不正确的目录。

你总是可以使用完全限定的资源位置代替 相对路径:例如, “文件:C:/ config /服务。 xml” 或 “类路径:/ config / services . xml” 。 然而,要知道你是 耦合应用程序的配置特定的绝对 位置。 它通常是更好的保持一个间接的 这样的绝对位置,例如,通过" $ {... } “占位符 这是解决在运行时对JVM系统属性。

5.2.3A使用集装箱

这个 ApplicationContext 是 界面一个先进的工厂能保持一个注册的 不同的bean及其依赖项。 使用方法 T getBean(字符串名称、类< T > requiredType) 你可以 您的bean检索实例。

这个 ApplicationContext 使您能够 读取bean定义和访问他们如下:

```
// create and configure beans
ApplicationContext context =
    new ClassPathXmlApplicationContext(new String[] {"services.xml", "daos.xml"});

// retrieve configured instance
PetStoreServiceImpl service = context.getBean("petStore", PetStoreServiceImpl.class);

// use configured instance
List userList = service.getUsernameList();
```

你使用 getBean() 检索实例的 你的豆子。 这个 ApplicationContext 接口有一些其他方法来检索bean,但最好你的 应用程序代码不应该使用它们。 事实上,您的应用程序代码 应该没有调用吗 getBean() 方法在 所有,因此不依赖于Spring api在所有。 例如,Spring的 与web框架集成提供了依赖注入 各种web框架类,如控制器和jsf托管 豆子。

5.3一个Bean概述

Spring IoC容器管理一个或多个 bean 。 这些bean创建配置元数据,你供应 的容器,例如,在形式的XML < bean / > 定义。

在集装箱本身,这些bean定义表示为 BeanDefinition 对象,它包含 (在其他信息)下面的元数据:

- 一个包限定类名: 通常 实际实现类定义的bean。
- 豆行为配置元素,这些国家如何Bean 应该表现在容器(范围、生命周期回调,所以呢 出)。
- 引用其他bean所需的bean来做它 工作;这些引用也被称为 合作者 或 依赖性 。
- 其他配置设置来设置在新创建的对象, 例如,连接数的使用在一个bean管理 连接池,或大小限制的池。

这个元数据转换成一组属性,弥补每个bean 定义。

5.1为多。 一个bean定义

财产	解释……
类	SectionA 5 3 2,一个beansa实例化
名称	SectionA 5 3 1,一个命名beansa
范围	SectionA 5.5,一个scopesaBean
构造函数参数	SectionA 5.4.1之前,一个injectiona依赖性
属性	SectionA 5.4.1之前,一个injectiona依赖性

自动装配模式	SectionA 5 4 5,一个collaboratorsa自动装配
延迟初始化模式	SectionA 5 4 4,一个延迟初始化 beansa
初始化方法	一个章节callbacksa初始化
破坏方法	一个章节callbacksa破坏

除了bean定义,包含信息 创建一个特定的bean, ApplicationContext 实现也 允许注册的现有对象以外的创建 容器,由用户。 这是通过访问ApplicationContext的 BeanFactory通过方法 getBeanFactory() 哪一个 返回BeanFactory实现 DefaultListableBeanFactory 。 DefaultListableBeanFactory 支持这 登记通过方法 registerSingleton(. .) 和 registerBeanDefinition(. .) 。 然而,典型的 应用程序的工作仅仅与豆类通过元数据定义的bean 定义。

5.3.1A命名豆

人各有一个或多个标识符。 这些标识符必须 独特的容器托管bean。 通常只有一个bean 一个标识符,但如果它需要不止一个, 额外的可以 认为是别名。

在基于xml的配置元数据,您使用 id 和/或 名称 属性来 指定bean标识符(年代)。 这个 id 属性 允许您指定一个id。传统究竟这些名字是 字母数字(' myBean” 、 “fooService” 等等),但可能特殊字符 为好。 如果你想介绍其他别名bean,您可以 还指定它们 名称 属性,隔开 一个逗号(,),分号(;),或 白色的空间。 作为一个历史的注意,在Spring 3.1版本之前, id 属性是一个类型的 xsd:ID ,可能的字符限制。 截至 3.1,现在 xsd:string 。 注意,bean id 独特性是由容器仍然执行,虽然不再通过XML 解析器。

你不需要提供一个名称或id为bean。 如果没有名字 或id提供明确,容器生成一个惟一的名称 这豆。 然而,如果你想要将这个 bean的名字,通过 使用 Ref 元素或 服务定位器 风格的查找, 你必须提供一个名称。 动机不提供一个名称 有关使用 内豆 和 自动装配 合作者 。

混淆一个bean外的bean定义

在bean定义本身,您可以提供超过一个名称 bean,通过结合使用一个指定的 名称 id 属性,和任意数量的其他名称 名称 属性。 这些名字可以等效 别名 来同样的bean,并且是有用的对于某些情况,如 允许每个组件在一个应用程 序来引用一个常见 通过使用bean名称依赖特定于该组件 本身。

指定所有别名定义的bean实际上是不 总是足够的,然而。 它有时需要介绍 一个 别名为bean定义的其他地方。 这是普遍的情形 在大型系统中,各子系 统之间的配置是分裂, 每个子系统有它自己的一组对象定义。 在基于xml的 配置元数据,您可以使用 <别名/ > 元素来完成这个。

> **Bean命名约定**
>
> 该公约是使用标准Java公约实例 当bean字段名 称命名。 即,bean名称开始 小写字母,骆驼包装 从那时起。 这样的例子 的名字是(没有引号) “accountManager” , “accountService” , “userDao” , ” loginController” ,等等。
>
> 一直让你配置命名豆子更容易阅读 和理解,如果 您正在使用Spring AOP它帮助很多 应用建议一 套bean相关的名字。

```
<alias name="fromName" alias="toName"/>
```

在这种情况下,bean在同一个集装箱得名 fromName ,也可以在使用这个别名 定义,被称为 toName 。

例如,配置元数据为一个可参考的子系统 到一个数据源通过名字 “subsystemA-dataSource。 配置 元数据子系统B可以引用数 据源通过名称 “subsystemB-dataSource’ 。 在编写应用程序,该应用程序使用的主要 这两个子系统的主要应用是指数据源 通过名字 “myapp数据源” 。 有这三个名字参考 相同的对象添加到MyApp配置元数据以下 别名定义:

```
<alias name="subsystemA-dataSource" alias="subsystemB-dataSource"/>
<alias name="subsystemA-dataSource" alias="myApp-dataSource" />
```

现在每个组件和主应用程序可以参考 数据源名称通过一个独特的和保证不发生冲突 与其他任何定义(有效地创建一个名称空 间),然而他们 指相同的bean。

5.3.2A实例化bean

一个bean定义实质上是一个创建一个或多个配方 对象。 容器看着菜谱命名豆当问, 和使用配置元数据封装的bean定义 创建 (或获得)一个实际对象。

如果你使用基于xml的配置元数据,您指定类型 (或类)的对象也被实例化的 类 属性的 < bean / > 元素。 这 类 属性,它的内部是一个 类 财产 在一个 BeanDefinition 实例,通常是 强制性的。 (例外,看到 一个章节实例使用一个实例工厂methoda 和 SectionA 5.7,一个inheritanceaBean定义)。 你使用 类 财产的方式有两种:

- 一般来说,指定bean类来构建的 情况下,容器本身直接创建bean调用 它的构造函数反映地,有点相当于Java代码使用 这个 新 操作符。
- 指定实际的类包含 静态 工厂方法时,就会到 创建对象,较常见的情况下,容器 调用一个 静态 , 工厂 方法在类来创建bean。 对象类型返回 的调用 静态 工厂方法可能 是同一类或另一个类完全。

与构造函数实例化

当你创建一个bean的构造函数方法,所有正常 类是可用的和兼容弹簧。 也就是说,类 正在开发不需要实现任何特定的接口或 是在一个特定的方式进行编码。 简单地指定bean类 应该足够了。 然而,这取决于你用什么类型的奥委会对 特定bean,您可能需要一个默认构造函数(空)。

Spring IoC容器可以管理几乎 任何 你想要它来管理类;它不是 限于管理真正的javabean。 大多数Spring用户更喜欢实际 只有一个默认的javabean(无参数)构造函数和 适当的setter和getter方法模仿中的属性 集装箱。 你也可以有更多的异国情调的非bean样式类 你的集装箱。 例如,如果您需要使用一个遗留的连接 池,绝对不符合JavaBean规范, 春天可以管理它。

内部类名称

如果你想配置一个bean定义 静态 嵌套类,您必须使用 二进制 内部类的名称。

例如,如果你有一个类称为 foo 在 com . example 包,这 foo 类有一个 静态 内在 类称为 酒吧 的值 " 类的 属性在一个bean定义会被.....

com例子foo $酒吧

注意使用 $ 字符的名称 将内部类的名字从外部类名。

与基于xml的配置元数据可以指定您的bean 类如下:

```
<bean id="exampleBean" class="examples.ExampleBean"/>

<bean name="anotherExample" class="examples.ExampleBeanTwo"/>
```

有关机构提供的参数 构造函数(如果需要)和设置对象实例属性之后 构造的对象,看到 注入 依赖性 。

与静态工厂方法实例化

当定义一个bean,您创建与静态工厂方法, 你使用 类 属性来指定类 包含 静态 工厂方法和一个 属性命名 工厂方法 指定名称 工厂方法本身。 你应该能够调用这个方法 (和可选的参数作为稍后描述)并返回一个活对象, 这后来被视为如果它已经创造了通过吗 构造函数。 一个用于这样一个bean定义是调用 静态 工厂在遗留代码。

下面的bean定义指定bean将 由调用工厂方法。 该定义没有指定 的类型(类)返回的对象,只有类包含 工厂方法。 在这个例子中, createInstance除外() 方法必须是一个 静态 法。

```
<bean id="clientService"
    class="examples.ClientService"
    factory-method="createInstance"/>
```

```
public class ClientService {
  private static ClientService clientService = new ClientService();
  private ClientService() {}

  public static ClientService createInstance() {
    return clientService;
  }
}
```

了解详细机制提供(可选参数) 工厂方法和设置对象实例属性的后 对象返回工厂,看到 依赖性和 详细配置 。

使用工厂方法实例化一个实例

通过一个类似实例化 静态工厂 方法 ,实例化一个实例工厂方法调用 非静态的方法从容器中现有的bean创建一个 新的bean。 使用这种机制,离开 类 属性空, 工厂bean 属性,指定bean的名称在当前(或 父/祖先)容器,包含实例方法 被调用来创建对象。 的名称设置工厂方法 本身与 工厂方法 属性。

```
<!-- the factory bean, which contains a method called createInstance() -->
<bean id="serviceLocator" class="examples.DefaultServiceLocator">
  <!-- inject any dependencies required by this locator bean -->
</bean>
```

```
<!-- the bean to be created via the factory bean -->
<bean id="clientService"
    factory-bean="serviceLocator"
    factory-method="createClientServiceInstance"/>
```

```
public class DefaultServiceLocator {
  private static ClientService clientService = new ClientServiceImpl();
  private DefaultServiceLocator() {}

  public ClientService createClientServiceInstance() {
    return clientService;
  }
}
```

一个工厂类也可以持有一个以上的工厂方法 所示:

```
<bean id="serviceLocator" class="examples.DefaultServiceLocator">
  <!-- inject any dependencies required by this locator bean -->
</bean>
<bean id="clientService"
    factory-bean="serviceLocator"
    factory-method="createClientServiceInstance"/>

<bean id="accountService"
    factory-bean="serviceLocator"
    factory-method="createAccountServiceInstance"/>
```

```
public class DefaultServiceLocator {
  private static ClientService clientService = new ClientServiceImpl();
  private static AccountService accountService = new AccountServiceImpl();

  private DefaultServiceLocator() {}

  public ClientService createClientServiceInstance() {
    return clientService;
  }

  public AccountService createAccountServiceInstance() {
    return accountService;
  }
}
```

这种方法表明,工厂bean本身是可以控制的 和配置通过依赖注入(DI)。 看到 依赖性和 详细配置 。

注意

在春天的文档, 工厂bean 指一个bean中配置Spring容器, 将创建对象通过一个吗 实例 或 静态 工厂方法。 相比之下, FactoryBean (注意 资本化)指的是一个spring特定 FactoryBean 。

5.4一个依赖

一个典型的企业应用程序不包含一个对象(或 豆在春天的说法)。 即使是最简单的应用程序有几个 对象,共同展示最终用户看到作为一个连贯的 应用程序。 下一节将解释如何去定义一个数量的 bean定义,独自站到一个完全实现应用程序 对象合作,以达到一个目标。

5.4.1A依赖注入

依赖注入 (DI)是一个过程,即 对象定义他们的依赖性,这是他们的工作,其他的对象 只能通过构造函数参数,参数到一个工厂方法,或属性设置对象实例,构建了之后 或返回工厂方法。 容器然后 注入 这些依赖项当它创建bean。 这一过程基本上是反,因此得名 控制反转 (IoC)的bean本身 控制实例化或位置的依赖自己 通过使用直接建设类,或 服务 定位器 模式。

代码是清洁与DI原理和去耦是更有效的 当对象提供它们的依赖项。 对象不 查找它的依赖性,不知道位置或类的 依赖关系。 这样,你的类更容易测试,特别是 当依赖在接口或抽象基类,它 允许存根或模拟实现用于单元测试。

存在于两个主要的变体迪, 无参依赖 注入 和 setter基于 依赖注入 。

无参依赖注入

无参 DI是完成的 容器调用构造函数与一些参数,每个 代表一个依赖项。 调用一个 静态 工厂 方法与特定的参数来构造bean是近 等价的,这个讨论治疗参数构造函数和 一个 静态 工厂方法类似。 以下 示例显示了一个类,只能依赖注入与 构造函数注入。 注意,没有什么 特殊 关于这个类,它是一个POJO,没有 依赖于容器特定的接口,基类或 注释。

```
public class SimpleMovieLister {

  // the SimpleMovieLister has a dependency on a MovieFinder
  private MovieFinder movieFinder;

  // a constructor so that the Spring container can 'inject' a MovieFinder
  public SimpleMovieLister(MovieFinder movieFinder) {
      this.movieFinder = movieFinder;
  }

  // business logic that actually 'uses' the injected MovieFinder is omitted...
}
```

构造函数参数的分辨率

构造函数参数分辨率匹配发生使用 参数的类型。 如果没有潜在的歧义存在的构造函数 参数定义的bean,那么顺序 构造函数的参数中定义的bean定义的顺序 这些参数是提供给适当的构造函数当吗 这个bean被实例化。 考虑下面的类:

```
package x.y;

public class Foo {

  public Foo(Bar bar, Baz baz) {
      // ...
  }
}
```

没有潜在的歧义存在,假设 酒吧 和 巴兹 类 不相关的继承。 因此下面的配置工作 很好,你不需要指定构造函数的参数指标 和/或类型显式的 < constructor - arg / > 元素。

```
<beans>
  <bean id="foo" class="x.y.Foo">
      <constructor-arg ref="bar"/>
      <constructor-arg ref="baz"/>
  </bean>

  <bean id="bar" class="x.y.Bar"/>
  <bean id="baz" class="x.y.Baz"/>

</beans>
```

当另一个bean引用,类型是已知的,匹配 可以发生(的情况也是前面的例子)。 当一个简单的 类型被使用,如 <值>真<值> ,春天不能 确定值的类型,因此不能匹配的类型没有 帮助。 考虑下面的类:

```
package examples;

public class ExampleBean {

  // No. of years to the calculate the Ultimate Answer
  private int years;

  // The Answer to Life, the Universe, and Everything
  private String ultimateAnswer;

  public ExampleBean(int years, String ultimateAnswer) {
      this.years = years;
      this.ultimateAnswer = ultimateAnswer;
  }
}
```

构造函数参数类型匹配

在前面的方案中,容器 可以 利用类型匹配与简单类型如果你 显式地指定类型的构造函数的参数使用 类型 属性。 例如:

```
<bean id="exampleBean" class="examples.ExampleBean">
<constructor-arg type="int" value="7500000"/>
<constructor-arg type="java.lang.String" value="42"/>
</bean>
```

构造函数参数的指数

使用 指数 属性来指定显式 构造函数参数的指数。 例如:

```
<bean id="exampleBean" class="examples.ExampleBean">
<constructor-arg index="0" value="7500000"/>
<constructor-arg index="1" value="42"/>
</bean>
```

除了解决歧义多简单 值,指定一个索引解决歧义在一个构造函数 有相同类型的两个参数。 注意, 指数是 0基础 。

构造函数参数的名字

在Spring 3.0中,您还可以使用构造函数的参数 名称值消歧:

```
<bean id="exampleBean" class="examples.ExampleBean">
<constructor-arg name="years" value="7500000"/>
<constructor-arg name="ultimateanswer" value="42"/>
</bean>
```

记住,使这项工作从盒子里你的代码 必须被编译和调试标记,以便Spring可以启用 查找参数的名字从构造函数。 如果你不能编译 你的代码与调试标记(或不愿意)可以使用 @ConstructorProperties JDK注释明确命名您的构造函数参数。 这个 样例类必须看起来如下:

```
package examples;

public class ExampleBean {

  // Fields omitted

  @ConstructorProperties({"years", "ultimateAnswer"})
  public ExampleBean(int years, String ultimateAnswer) {
      this.years = years;
      this.ultimateAnswer = ultimateAnswer;
  }
}
```

setter建立依赖注入

setter基于 DI是完成的 容器调用setter方法在你的bean调用之后 无参数构造函数或无参数 静态 工厂 方法来实例化bean。

下面的示例显示了一个类,它只能 依赖注入的使用纯setter注入。 这个类是 传统的Java。 这是一个POJO,没有依赖于容器 具体的接口,基类或注释。

```
public class SimpleMovieLister {

  // the SimpleMovieLister has a dependency on the MovieFinder
  private MovieFinder movieFinder;

  // a setter method so that the Spring container can 'inject' a MovieFinder
  public void setMovieFinder(MovieFinder movieFinder) {
      this.movieFinder = movieFinder;
  }

  // business logic that actually 'uses' the injected MovieFinder is omitted...
}
```

这个 ApplicationContext 支持 构造函数,对bean的setter基于DI它管理。 它还 支持基于setter经过一些依赖性都已经DI注射通过构造函数的方法。 你配置中的依赖关系 形式的一个 BeanDefinition ,你可以用它 与 属性编辑器 实例转换 属性从一种格式到另一个。 然而,大多数Spring用户不 使用这些类直接(编程),而是用一个 XML定义文件,然后在内部转换成的实例 这些类,用于加载整个Spring IoC容器 实例。

依赖解析过程

容器执行bean依赖决议如下:

1. 这个 ApplicationContext 创建 在初始化配置元数据,描述所有的 豆子。 配置元数据可以被指定通过XML、Java代码或 注释。
2. 对于每个bean,它所依赖的形式表达 属性,构造函数参数或参数 静态工厂方法如果您使用的不是正常的 构造函数。 这些依赖项提供了豆, 当bean实际上是创建了 。
3. 每个属性或构造函数的参数是一个实际的定义 值来设置,或引用的另一个bean 集装箱。
4. 每个属性或构造函数的参数,它是一个值 从其指定的格式转换到实际的类型的 属性或构造函数参数。 默认情况下弹簧可以转换一个 值在字符串格式提供给所有内置类型,如 int , 长 , 字符串 , 布尔 等。

Spring容器的配置每个bean验证的 集装箱被创建,包括验证bean是否参考

无参或setter建立迪吗?

因为你可以混合使用两种,构造函数——和setter基于DI,它是一个 好的经验法则使用构造函数参数为强制性 可选依赖依赖和setter。 注意,使用 的 @ required 注释一个setter可以用来制造setter要求 依赖关系。

春季团队一般主张setter注入,因为 大量的构造函数参数可以得到笨拙,特别是 当属性是可选的。 Setter方法也使对象的 类易于重构或重新注入后。 管理 通过 JMX mbean 是一个引人注目的使用 案例。

一些纯粹主义者支持注射无参。 供应所有 对象依赖意味着对象总是返回给客户机 (打电话)代码在一个完全初始化状态。 缺点是 的对象变得不大受重组和 重新注入。

属性指的是有效的豆子。 然而,该bean的属性本身 没有设置到豆吗 实际上是创建 。 豆子是单例范围并设置为可预先实例化(默认)是由容器创建。 范围被定义 在 SectionA 5.5,一个scopesaBean 否则,创建了bean 只有当它是要求。 创建bean可能导致一个图 要创建的bean,作为bean的依赖项和它所依赖的 依赖性(等等)创建和分配。

使用DI,让最适合一个特定的类。 有时,当处理的第三方类,你不 有来源,选择了你。 一个遗产类不得 揭露任何setter方法,构造函数注入是唯一的 可用DI。

通常你能够信任弹簧做正确的事。 它检测 配置问题,如引用不存在的豆类和 循环依赖,在容器加载时。 弹簧设置属性 和解析依赖项尽可能晚,当bean实际上是 创建的。 这意味着一个Spring容器加载正确的 稍后可以生成一个异常当你请求一个对象是否有 问题,对象创建或它的一个依赖项。 例如, bean抛出一个异常结果的缺失或无效 财产。 这可能会推迟一些配置的可见性 问题是为什么 ApplicationContext 默认实现单例bean实例化之前。 在成本 前期的一些时间和内存来创建这些bean之前 实际需要,你发现时配置问题 ApplicationContext 被创建,不迟。 你仍然可以覆盖这个缺省行为,这样单例bean 将懒惰的初始化,而不是预先实例化。

循环依赖

如果你使用主要构造函数注入,它是可能的 创建一个加入不能解析循环依赖的场景。

例如:A类需要B类的一个实例通过 构造函数注入,B类需要一个类的一个实例 通过构造函数注入。 如果你配置bean类的一种 和B注入到对方,Spring IoC容器检测 这个循环引用在运行时,并抛出了一个 BeanCurrentlyInCreationException 。

一个可能的解决方案是编辑源代码的一些类 通过设置配置而不是构造函数。 另外, 避免构造函数注入和使用setter注入只有。 在其他 话说,虽然不推荐,您可以配置循环 依赖关系与setter注入。

与 典型 案例(没有圆 依赖关系),一个圆形的依赖关系和bean B部队bean 一个bean注入其他之前被完全 初始化本身(经典鸡/蛋场景)。

如果没有循环依赖存在,当一个或更多的合作 豆子被注入依赖的bean,每个bean合作 是 完全 配置之前被注入 依赖bean。 这意味着如果bean一依赖的bean B,Spring IoC容器完全配置bean B之前 调用bean的setter方法a。换句话说,这个bean 实例化(如果不是预先实例化单例),其依赖关系 集,和相关的生命周期方法(如 配置初始化 方法 或 InitializingBean 回调方法)被调用。

依赖注入的例子

下面的示例使用基于xml的配置元数据 setter基于DI。 一个春天的一小部分的XML配置文件 指定一些bean定义:

```
<bean id="exampleBean" class="examples.ExampleBean">

<!-- setter injection using the nested <ref/> element -->
<property name="beanOne"><ref bean="anotherExampleBean"/></property>

<!-- setter injection using the neater 'ref' attribute -->
<property name="beanTwo" ref="yetAnotherBean"/>
<property name="integerProperty" value="1"/>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

```
public class ExampleBean {

  private AnotherBean beanOne;
  private YetAnotherBean beanTwo;
  private int i;

  public void setBeanOne(AnotherBean beanOne) {
      this.beanOne = beanOne;
  }

  public void setBeanTwo(YetAnotherBean beanTwo) {
      this.beanTwo = beanTwo;
  }

  public void setIntegerProperty(int i) {
      this.i = i;
  }
}
```

在前面的示例中,要匹配的制定者是宣布的 属性中指定的XML文件。 下面的例子使用 无参迪:

```
<bean id="exampleBean" class="examples.ExampleBean">

<!-- constructor injection using the nested <ref/> element -->
<constructor-arg>
  <ref bean="anotherExampleBean"/>
</constructor-arg>

<!-- constructor injection using the neater 'ref' attribute -->
<constructor-arg ref="yetAnotherBean"/>

<constructor-arg type="int" value="1"/>
```

```
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

```
public class ExampleBean {

  private AnotherBean beanOne;
  private YetAnotherBean beanTwo;
  private int i;

  public ExampleBean(
      AnotherBean anotherBean, YetAnotherBean yetAnotherBean, int i) {
      this.beanOne = anotherBean;
      this.beanTwo = yetAnotherBean;
      this.i = i;
  }
}
```

构造函数的参数中指定的bean定义将 用作参数的构造函数 ExampleBean 。

现在考虑这个例子的变种,而不是使用一个 构造函数,春天是告诉调用 静态 工厂 方法返回对象的一个实例:

```
<bean id="exampleBean" class="examples.ExampleBean"
   factory-method="createInstance">
<constructor-arg ref="anotherExampleBean"/>
<constructor-arg ref="yetAnotherBean"/>
<constructor-arg value="1"/>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

```
public class ExampleBean {

  // a private constructor
  private ExampleBean(...) {
    ...
  }

  // a static factory method; the arguments to this method can be
  // considered the dependencies of the bean that is returned,
  // regardless of how those arguments are actually used.
  public static ExampleBean createInstance (
         AnotherBean anotherBean, YetAnotherBean yetAnotherBean, int i) {

      ExampleBean eb = new ExampleBean (...);
      // some other operations...
      return eb;
  }
}
```

参数 静态 工厂方法是 提供通过 < constructor - arg / > 元素, 完全一样,如果一个构造函数实际上已经被使用。 这个类型的类返回了工厂方法不需要的 同一类型的类,该类包含 静态 工厂方法,虽然在这个例子中它是。 一个实例(非静态) 工厂方法将被用于一个本质上相同的时装(一边 从使用 工厂bean 属性而不是 这个 类 属性),所以细节将不会 在这里讨论。

5.4.2A依赖性和配置细节

正如上一节中提到的,您可以定义bean属性 和构造函数参数作为引用其他托管bean (合作者),或作为值定义内联。 Spring的基于xml的 配置元数据支持元素类型在它 <属性/ > 和 < constructor - arg / > 元素对于这个 目的。

直值(原语, 字符串 ,所以)

这个 价值 属性的 <属性/ > 元素指定一个属性或 构造函数参数作为一个人类可读的字符串表示。 正如前面提到的 , JavaBeans PropertyEditors 是用来转换这些吗 字符串值从一个 字符串 实际的类型的 房地产或参数。

```
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<!-- results in a setDriverClassName(String) call -->
<property name="driverClassName" value="com.mysql.jdbc.Driver"/>
<property name="url" value="jdbc:mysql://localhost:3306/mydb"/>
<property name="username" value="root"/>
<property name="password" value="masterkaoli"/>
</bean>
```

下面的例子使用了 p名称空间 更简洁的XML配置。

```
<beans xmlns="http://www.springframework.org/schema/beans"
```

```
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"
     destroy-method="close"
     p:driverClassName="com.mysql.jdbc.Driver"
     p:url="jdbc:mysql://localhost:3306/mydb"
     p:username="root"
     p:password="masterkaoli"/>

</beans>
```

前面的XML更简洁的;然而,输入错误发现 运行时,而不是设计时间,除非你使用一个IDE如 intellij idea 或 SpringSource工具 套件 (STS),支持自动属性完成当你 创建bean定义。 这样的IDE援助是高度 推荐。

您还可以配置一个 java util属性 实例:

```
<bean id="mappings"
   class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

 <!-- typed as a java.util.Properties -->
 <property name="properties">
   <value>
     jdbc.driver.className=com.mysql.jdbc.Driver
     jdbc.url=jdbc:mysql://localhost:3306/mydb
   </value>
 </property>
</bean>
```

Spring容器内的文本转换 <价值/ > 元素到一个 java util属性 实例通过使用 JavaBeans 属性编辑器 机制。 这 是一个不错的捷径,是为数不多的地方之一的弹簧组是怎么做的呢 支持使用嵌套的 <价值/ > 元素 在 价值 属性风格。

这个 idref 元素

这个 idref 元素是一个简单的错误证明方式 通过 id (字符串值——不是一个参考) 另一个bean在集装箱到 < constructor - arg / > 或 <属性/ > 元素。

```
<bean id="theTargetBean" class="..."/>

<bean id="theClientBean" class="...">
  <property name="targetName">
     <idref bean="theTargetBean" />
  </property>
</bean>
```

上面的bean定义片段 到底 等效(在运行时),下面的代码片段:

```
<bean id="theTargetBean" class="..." />

<bean id="client" class="...">
  <property name="targetName" value="theTargetBean" />
</bean>
```

第一种形式比第二个,因为使用 idref 标记允许容器来验证 在部署时 这个引用,命名 豆确实存在。 在第二个变化,没有验证 执行的值传递到 targetName 财产的 客户 bean。 输入错误只是发现(与大多数 可能致命的结果) 客户 bean是 其实实例化。 如果客户 bean是一个 原型 豆,这个错误 和由此产生的异常可能只被发现后很长时间 容器部署。

此外,如果引用bean是在相同的XML单位, bean的名字是bean id ,你可以使用 当地 属性,它允许XML解析器本身 验证bean id 之前,在XML文档解析时间。

```
<property name="targetName">
 <!-- a bean with id 'theTargetBean' must exist; otherwise an exception will be thrown -->
 <idref local="theTargetBean"/>
</property>
```

一个共同的地方(至少在版本早于Spring 2.0) 在< idref / >元素带来的价值是在配置吗 的 AOP拦截器 在一个 ProxyFactoryBean bean定义。 使用 < idref / >元素当你指定拦截器名称 防止你拼错一个拦截器id。

引用其他豆类(合作者)

这个 Ref 元素是最后的元素在 < constructor - arg / > 或 <属性/ > 定义元素。 在这里你设置 指定属性的值的一个bean的

引用 另一个bean(一个合作者)容器来管理。 引用的 豆是一种依赖的bean的财产将被设置,它是 初始化前的随需应变属性设置。(如果 合作者是一个单例bean,它可能被初始化了 容器)。 最终所有引用另一个对象的一个引用。 范围和验证取决于您指定id /名称的 其他对象通过 bean , 当地的, 或 父 属性。

指定目标bean通过 bean 属性的 < ref / > 标签是最一般的 形式,允许创建一个引用到任何bean在相同的 容器或父容器,不管它是在相同的 xml文件。 的价值 bean 属性可能是 一样 id 目标bean的属性,或作为 的一个值 名称 属性的目标 bean。

```
<ref bean="someBean"/>
```

指定目标bean通过 当地 属性利用能力的XML解析器来验证XML id 在同一个文件的引用。 的价值 当地 属性必须是一样的 id 目标bean的属性。 XML解析器 问题一个错误如果没有找到匹配的元素在同一个文件中。 作为 这样,使用当地的变体是最好的选择(为了了解 错误尽可能早的)如果目标bean是在相同的XML 文件。

```
<ref local="someBean"/>
```

指定目标bean通过 父 属性创建一个引用bean,是在一个父容器的 当前的容器。 的价值 父 属性可能是相同的或 id 属性 目标bean,或是一个值 名称 目标bean的属性,和目标bean必须在父母 容器的当前。 你使用这个bean引用主要变体 当你有一个层次结构的容器,你想把现有的 豆在父容器与代理,将会有相同的名称 父bean。

```
<!-- in the parent context -->
<bean id="accountService" class="com.foo.SimpleAccountService">
  <!-- insert dependencies as required as here -->
</bean>
```

```
<!-- in the child (descendant) context -->
<bean id="accountService"  <-- bean name is the same as the parent bean -->
   class="org.springframework.aop.framework.ProxyFactoryBean">
   <property name="target">
      <ref parent="accountService"/>  <!-- notice how we refer to the parent bean -->
   </property>
  <!-- insert other configuration and dependencies as required here -->
</bean>
```

内豆

一个 < bean / > 元素在 <属性/ > 或 < constructor - arg / > 元素定义了一个所谓的 内在bean 。

```
<bean id="outer" class="...">
<!-- instead of using a reference to a target bean, simply define the target bean inline -->
<property name="target">
  <bean class="com.example.Person"> <!-- this is the inner bean -->
    <property name="name" value="Fiona Apple"/>
    <property name="age" value="25"/>
  </bean>
</property>
</bean>
```

一种内在的bean定义不需要定义id或名称; 忽略这些值容器。 也忽略了 范围 国旗。 内在bean是 总是 匿名和他们 总是 创建外部bean。 这是 不 可能将内心的豆子进 其他的合作豆子比封闭的bean。

集合

在 <列表/ > , <设置/ > , <地图/ > ,和 <道具/ > 元素,设置属性和 参数的Java 收集 类型 列表 , 集 , 地图 ,和 属性 ,分别。

```
<bean id="moreComplexObject" class="example.ComplexObject">
<!-- results in a setAdminEmails(java.util.Properties) call -->
<property name="adminEmails">
  <props>
      <prop key="administrator">administrator@example.org</prop>
      <prop key="support">support@example.org</prop>
      <prop key="development">development@example.org</prop>
  </props>
</property>
<!-- results in a setSomeList(java.util.List) call -->
<property name="someList">
  <list>
      <value>a list element followed by a reference</value>
      <ref bean="myDataSource" />
  </list>
</property>
<!-- results in a setSomeMap(java.util.Map) call -->
<property name="someMap">
  <map>
      <entry key="an entry" value="just some string"/>
```

```
    <entry key ="a ref" value-ref="myDataSource"/>
  </map>
</property>
<!-- results in a setSomeSet(java.util.Set) call -->
<property name="someSet">
  <set>
    <value>just some string</value>
    <ref bean="myDataSource" />
  </set>
</property>
</bean>
```

地图的价值关键或价值,或一组值,也可以 再任何下列元素:

```
bean | ref | idref | list | set | map | props | value | null
```

收集合并

在Spring 2.0中,容器支持 合并 的集合。 应用程序开发人员 可以定义一个父样式 <列表/ > , <地图/ > , <设置/ > 或 <道具/ > 元素,和有孩子的风格 <列表/ > , <地图/ > , <设置/ > 或 <道具/ > 元素继承和重载值从父集合。 这 是,孩子集合的值合并的结果吗 父母和孩子的元素集合,有孩子的 集合的元素中指定的值覆盖父 收集。

这部分讨论了亲子bean合并 机制。 父母和孩子的读者不熟悉的bean定义 可能希望阅读吗 相关部门 在继续之前。

下面的例子演示了收集合并:

```
<beans>
<bean id="parent" abstract="true" class="example.ComplexObject">
  <property name="adminEmails">
    <props>
      <prop key="administrator">administrator@example.com</prop>
      <prop key="support">support@example.com</prop>
    </props>
  </property>
</bean>
<bean id="child" parent="parent">
  <property name="adminEmails">
    <!-- the merge is specified on the *child* collection definition -->
    <props merge="true">
      <prop key="sales">sales@example.com</prop>
      <prop key="support">support@example.co.uk</prop>
    </props>
  </property>
</bean>
<beans>
```

注意使用 合并= true 属性 这个 <道具/ > 元素的 adminEmails 财产的 孩子 bean定义。 当 孩子 豆是解决和实例化 容器,生成的实例有一个 adminEmails 属性 集合,包含结果的合并的孩子的 adminEmails 收集与父母的 adminEmails 收集。

```
administrator=administrator@example.com
sales=sales@example.com
support=support@example.co.uk
```

孩子 属性 集合的值设置 继承了所有的财产元素父 <道具/ > 和孩子的价值 支持 值将覆盖在父的价值 收集。

这个合并行为适用类似 <列表/ > , <地图/ > ,和 <设置/ > 集合类型。 在特定的情况下 的 <列表/ > 元素,语义 相关 列表 集合类型, 是,的概念 下令 收藏的价值, 是维护;父母的价值观之前所有的子列表的 值。 在的情况下 地图 , 集 ,和 属性 集合类型, 没有 订购的存在。 因此没有订购语义实际上影响了 集合类型,构成相关的 地图 , 集 ,和 属性 实现类型, 容器内部使用。

限制收集合并

你不能合并不同的集合类型(如 地图 和一个 列表),如果你试图这样做 一个适当的 异常 抛出。 这个 合并 属性必须被指定在降低, 继承、儿童定义;指定 合并 属性在一个父集合定义是冗余的,不会 导致所需的合并。 合并功能是只提供 在Spring 2.0和以后。

强类型集合(Java 5 +只有)

在Java 5和以后,您可以使用强类型集合(使用 泛型类型)。 也就是说,它可以声明一个 收集 的类型,这样它只能 包含 字符串 元素(例如)。 如果你 使用Spring依赖项注入一个强类型的吗 收集 成一个bean,您可以采取 利用弹簧的类型转换支持这样的元素 你的强类型 收集 实例转换为适当的类型添加之前 到 收集 。

```
public class Foo {
```

```
  private Map<String, Float> accounts;

  public void setAccounts(Map<String, Float> accounts) {
    this.accounts = accounts;
  }
}
```

```
<beans>
 <bean id="foo" class="x.y.Foo">
   <property name="accounts">
     <map>
       <entry key="one" value="9.99"/>
       <entry key="two" value="2.75"/>
       <entry key="six" value="3.99"/>
     </map>
   </property>
 </bean>
</beans>
```

当 账户 财产的 foo 豆是准备注入,泛型 元素类型信息的强类型 Map < String,浮动> 可以通过 反射。 因此弹簧的类型转换基础设施识别 各种价值元素的类型 浮 和字符串值 9.99, 2.75 ,和 3.99 转化为 实际 浮 类型。

空,空字符串值

春天 把空的参数属性等为空 字符串 。 下面的基于xml的配置 元数据代码片段设置电子邮件属性的空洞 字符串 值(“”)

```
<bean class="ExampleBean">
<property name="email" value=""/>
</bean>
```

前面的例子是相当于以下Java代码: exampleBean.setEmail(" ") 。 这个 < null / > 元素处理 空 值。 例如:

```
<bean class="ExampleBean">
<property name="email"><null/></property>
</bean>
```

上面的配置相当于以下Java代码: exampleBean.setEmail(空) 。

XML快捷与p名称空间

在p名称空间允许您使用 bean 元素的属性,而不是嵌套 <属性/ > 元素,来描述你的财产 价值观和/或合作豆子。

Spring 2.0,后来支持可扩展的配置格式 使用名称空间的 ,这是基于XML 模式定义。 这个 bean 配置格式 在本章中讨论的定义在一个XML模式文档。 然而, 没有定义的p名称空间在XSD文件和仅存在于 春天的核心。

下面的例子显示了两个XML片段,解决 同样的结果:第一次使用标准XML格式和第二次使用 p名称空间。

```
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:p="http://www.springframework.org/schema/p"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

 <bean name="classic" class="com.example.ExampleBean">
    <property name="email" value="foo@bar.com"/>
 </bean>

 <bean name="p-namespace" class="com.example.ExampleBean"
      p:email="foo@bar.com"/>
</beans>
```

该示例显示了一个属性在p名称空间称为电子邮件 该bean定义。 这告诉春天包含一个属性 宣言。 正如前面提到的,p名称空间没有 模式定义,所以你可以设置属性的名称的 属性名。

下一个示例包括两个bean定义,都有 另一个bean的引用:

```
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:p="http://www.springframework.org/schema/p"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

 <bean name="john-classic" class="com.example.Person">
    <property name="name" value="John Doe"/>
    <property name="spouse" ref="jane"/>
 </bean>
```

```
  <bean name="john-modern"
    class="com.example.Person"
    p:name="John Doe"
    p:spouse-ref="jane"/>

  <bean name="jane" class="com.example.Person">
    <property name="name" value="Jane Doe"/>
  </bean>
</beans>
```

正如您可以看到的,这个例子不仅包括一个属性值 使用p名称空间,但也使用一个特殊的格式申报 属性引用。 而第一个bean定义使用 <属性名= “配偶” ref = "简" / > 创建 从bean的引用 约翰 到bean 简 ,第二个bean定义使用 p:配偶ref = “简” 作为一个属性来做准确的 同样的事情。 在这种情况下 配偶 是属性名, 而 ref 部分表明这不是一个 直值而是一个引用到另一个 bean。

注意

不像的p名称空间灵活为标准的XML格式。 对于 示例中,格式为声明属性引用的冲突 属性,最终在 Ref ,而标准 XML格式不。 我们建议你选择你的方法 仔细和沟通这你的团队成员,以避免 产生的XML文档,使用这三种方法在相同的 时间。

XML快捷与c名称空间

类似 一个章节XML与p-namespacea快捷方式 , c名称空间 ,新引进的Spring 3.1中, 允许使用内联属性进行配置,构造函数参数而非嵌套 constructor - arg 元素。

让我们回顾的示例 一个章节injectiona无参的依赖 与 C 命名空间:

```
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:c="http://www.springframework.org/schema/c"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

 <bean id="bar" class="x.y.Bar"/>
 <bean id="baz" class="x.y.Baz"/>

 <-- 'traditional' declaration -->
 <bean id="foo" class="x.y.Foo">
    <constructor-arg ref="bar"/>
    <constructor-arg ref="baz"/>
    <constructor-arg value="foo@bar.com"/>
 </bean>

 <-- 'c-namespace' declaration -->
 <bean id="foo" class="x.y.Foo" c:bar-ref="bar" c:baz-ref="baz" c:email="foo@bar.com">

</beans>
```

这个 C: 名称空间使用相同的约定的 P: 一个(落后 ref 对于bean引用) 构造函数参数的设置由他们的名字。 和一样好,这需要宣称即使它没有定义在XSD模式 (但它存在在Spring核心)。

为罕见的情况下,构造函数参数名称不可以(通常如果字节码编译没有调试信息),一个人可以 使用回退到参数指标:

```
<-- 'c-namespace' index declaration -->
<bean id="foo" class="x.y.Foo" c:_0-ref="bar" c:_1-ref="baz">
```

注意

由于XML语法,该指数表示法需要领先的存在 _ 作为XML属性名称不能开始 与一些(尽管一些IDE允许它)。

在实践中,构造函数分辨率 机制 是相当有效的匹配参数 除非一个人真的需要,我们建议使用名称符号都会你的配置。

复合属性名

您可以使用复合或嵌套的属性名当你设置bean 属性,只要所有组件的路径除了最后的 属性名不 空 。 考虑以下 bean定义。

```
<bean id="foo" class="foo.Bar">
<property name="fred.bob.sammy" value="123" />
</bean>
```

这个 foo bean有一个 弗雷德 属性,它有一个 鲍勃 属性,它有一个 萨米 财产,最后 萨米 属性被设置为值 123年 。 为了工作, 弗

雷德 财产的 foo ,和 鲍勃 财产的 弗雷德 不得 空 在bean被构建,或一个 NullPointerException 抛出。

5.4.3A使用 取决于

如果一个bean是一个依赖另一个通常意味着一个bean 设置属性的另一个。 通常你完成这个的 < ref / > 元素 在基于xml的配置元数据。 然而,有时 bean之间的依赖关系不直接的;例如,一个静态的 初始化器在一个类需要被触发,比如数据库驱动程序 登记。 这个 取决于 属性可以显式 力一个或多个bean被初始化之前使用这个bean 元素初始化。 下面的例子使用了 取决于 属性来表达依赖于一个 单一的豆:

```
<bean id="beanOne" class="ExampleBean" depends-on="manager"/>

<bean id="manager" class="ManagerBean" />
```

表达依赖于多个bean,提供一个列表的bean的名字 的值的 取决于 属性,用逗号隔开, 空格、分号、用作有效的分隔符:

```
<bean id="beanOne" class="ExampleBean" depends-on="manager,accountDao">
<property name="manager" ref="manager" />
</bean>

<bean id="manager" class="ManagerBean" />
<bean id="accountDao" class="x.y.jdbc.JdbcAccountDao" />
```

注意

这个 取决于 属性在bean定义 可以同时指定一个初始化时间依赖和,对于吗 singleton bean 只是,一个相应的破坏时间依赖性。 依赖豆子, 定义一个 取决于 关系与一个给定的bean 首先被破坏之前,给定的bean本身遭到破坏。 因此 取决于 还可以控制关闭 订单。

5.4.4A延迟初始化的 bean

默认情况下, ApplicationContext 实现急切地创建和配置所有 singleton bean作为 初始化过程。 通常,这是以前的实例化 可取的,因为错误的配置或周围的环境 立即被发现,而不是数小时甚至数天之后。 当 这种行为是 不 可取的,可以防止 前实例化一个单例bean标记bean定义 延迟初始化的。 一个懒惰的初始化bean告诉IoC容器来 创建一个bean实例当它是第一个请求,而不是在 启动。

在XML中,这种行为的控制 懒init 属性 < bean / > 元素,例如:

```
<bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-init="true"/>

<bean name="not.lazy" class="com.foo.AnotherBean"/>
```

如果之前使用的配置 ApplicationContext ,bean命名 懒惰 不是急切地预先实例化的时候吗 ApplicationContext 是启动,而 这个 不是懒惰 豆是急切地预先实例化。

然而,当一个延迟初始化的bean是一个单例对象的依赖 bean, 不 延迟初始化的, ApplicationContext 创建 延迟初始化的bean在启动,因为它必须满足单例的 依赖关系。 延迟初始化的bean注入到一个单例bean 在其他地方,不是懒惰的初始化。

您还可以控制延迟初始化在集装箱的水平 使用 默认懒init 属性 < bean / > 元素,例如:

```
<beans default-lazy-init="true">
  <!-- no beans will be pre-instantiated... -->
</beans>
```

5.4.5A自动装配的合作者

Spring容器可以 自动装配 关系 在合作豆子。 你可以让春天解决合作者 (其他bean)自动为您的bean通过检查的内容 这个 ApplicationContext 。 自动装配有 以下优点:

- 自动装配可以显著减少需要指定属性 或构造函数参数。 (其他机制如豆的模板 讨论其他 本章 在这方面也有价值。)
- 自动装配可以更新一个配置对象演变。 对于 例子,如果你需要添加一个依赖关系,一个类,这种依赖性 可以满足自动而不需要重新修改吗 配置。 因此可以在自动装配特别有用 发展,没有否定选择切换到明确的 布线当代码库会变得更加稳定。

当使用基于xml的配置元数据 [2] ,你指定的bean定义自动装配模式的 自动装配 属性的 < bean / > 元素。 自动装配的功能五个模式。 你指定自动装配 每 bean,因此 可以选择哪些自动装配。

5.2为多。 一个自动装配模式

模式	解释
没有	(默认)没有自动装配。 Bean引用必须 定义通过 Ref 元素。 更改默认 不建议设置为较大的部署,因为 指定显式地给了更大的控制权和合作者 清晰。 在某种程度上,它文档的结构 系统。
别名	自动装配的属性名。 春天寻找一个bean 具有相同名称的属性,需要autowired的。 对于 例子,如果一个bean定义设置为自动装配的名字,和它 包含一个 主 属性(即,它有一个 setMaster(. .) 方法),弹簧查找 bean定义命名 主 ,使用它 设置此属性。
byType	允许一个属性是autowired的如果整整一个bean 属性的类型存在于容器。 如果超过一个 存在,一个致命的异常被抛出,这表明你可能 不使用 byType 自动装配的bean。 如果 没有匹配的豆子,什么也不会发生;财产不是 设置。
构造函数	类似于 byType ,但应用 到构造函数参数。 如果有不准确的一个bean 构造函数参数类型的容器,一个致命错误 提高。

与 byType 或 构造函数 自动装配模式,你可以线阵列和类型的集合。 在这种情况下 所有 自动装配的候选人在容器内, 匹配预期的类型提供了满足依赖关系。 你可以 如果预期强类型的地图自动装配关键类型 字符串 。 一个地图将包括autowired的价值观所有bean实例匹配预期的类型,和地图键将 包含相应的bean的名字。

你可以结合自动装配行为检查依赖性,这是 自动装配完成后进行。

局限性和缺点的自动装配

自动装配工作最好当它用于跨一个项目。 如果不使用自动装配在一般情况下,它可能是困惑 开发人员使用它来线只有一个或两个bean定义。

考虑的限制和缺点的自动装配:

- 显式的依赖在 财产 和 constructor - arg 设置总是覆盖 自动装配。 你不能自动装配所谓 简单 属性如原语, 字符串 ,和 类(和数组这样的简单属性)。 这个限制是 通过设计。
- 自动装配并不完全像显式布线。 不过,由于 指出在上面的表中,春天是小心避免猜测在 模棱两可的情况,可能意想不到的结果, 你的spring管理对象之间的关系不再是 明确地描述。
- 布线信息可能不能使用的工具,可能 从Spring容器生成文档。
- 多个bean定义在容器内可能匹配 指定的类型setter方法或构造函数的参数 autowired的。 为数组、集合或地图,这是不一定的 一个问题。 然而对于依赖关系,期望一个单一值,这 歧义是不随意解决。 如果没有独特的bean定义 可用,就抛出一个异常。

在后一种情况下,你有几种选择:

- 放弃自动装配支持显式布线。
- 避免自动装配为一个bean定义通过设置它 自动装配的候选人 属性来 假 在下一节中描述。
- 指定一个bean定义的 主要 候选人通过设置 主要 属性的 < bean / > 元素 真正的 。
- 如果您使用的是Java 5或更高版本,实现更多 细粒度的控制可提供基于注解的配置, 中描述的 SectionA 5.9,一个configurationa基于注解的容器 。

不包括一个bean从自动装配

每个bean上基础上,你可以排除一个bean从自动装配。 在 Spring的XML格式,设置 自动装配的候选人 属性的 < bean / > 元素 假 ;容器使特定bean 定义不能自动装配的基础设施(包括 注释风格配置如 @ autowired)。

你也可以限制自动装配候选人基于模式匹配 对bean名称。 顶级 < bean / > 元素接受一个或多个模式在它 默认自动装配候选人 属性。 例如, 自动装配的候选人地位来限制任何bean的名字结尾 库, 提供的值*存储库。 到 提供多种模式,定义在一个以逗号分隔的列表。 一个 显式值 真正的 或 假 对于一个bean定义 自动装配的候选人 属性 总是优先,并且这样的豆子,模式匹配规则 不适用。

这些技巧是很有用的,你再也不想咖啡豆是 其他bean注入由自动装配。 这并不意味着一个 排除bean本身不能被配置使用自动

装配。 相反, bean本身不是一个候选人对自动装配其他豆类。

5.4.6A方法注入

在大多数应用程序场景,大多数豆类的容器 单件 。 当一个 单例bean需要与另一个单例bean,或者一个 非单体bean需要与另一个单体豆, 你通常处理依赖通过定义一个bean的属性 其他的。 一个问题会在bean的生命周期是不同的。 假设singleton豆一个需要使用非单体(原型)bean B, 也许在每个方法调用在a .容器仅创建 单例bean一次,因此只有得到一个机会来设置 属性。 容器不能提供bean与一个新实例的一个 每次一豆B需要。

一个解决办法是放弃一些控制反转。 你可以 让豆一个意识到容器 通过 实施 ApplicationContextAware 接口,通过 做 getBean(" B ")调用容器 要求(一个典型的新)bean B 实例一个需要每次bean。 下面是一个例子 方法:

```
// a class that uses a stateful Command-style class to perform some processing
package fiona.apple;

// Spring-API imports
import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;

public class CommandManager implements ApplicationContextAware {

 private ApplicationContext applicationContext;

 public Object process(Map commandState) {
   // grab a new instance of the appropriate Command
   Command command = createCommand();
   // set the state on the (hopefully brand new) Command instance
   command.setState(commandState);
   return command.execute();
 }

 protected Command createCommand() {
   // notice the Spring API dependency!
   return this.applicationContext.getBean("command", Command.class);
 }

 public void setApplicationContext(ApplicationContext applicationContext)
                                        throws BeansException {
   this.applicationContext = applicationContext;
 }
}
```

前是不可取的,因为业务代码是意识到 和耦合到Spring框架。 方法注射,有些先进的 Spring IoC容器的特点,允许这个用例的处理 一个干净的 时尚。

查找方法注入

查找方法注入容器的能力覆盖 方法 容器托管bean ,返回 查找结果另一个名叫豆在容器。 查找 通常涉及一个原型bean在描述的场景 前一节。 Spring框架实现了这个方法注入 通过使用字节码生成从CGLIB来生成 动态子类,覆盖了 法。

> 你可以阅读更多关于动机的方法注入 这个博客条目 。

> **注意**
>
> 对于这个动态子类化工作,春天的类 容器将子类不能 最后 ,和 方法不能被覆盖 最后 要么。 同时, 测试一个类,它有一个 文摘 方法需要 你自己和子类的类提供一个存根实现 的 文摘 法。 最后,对象 被目标的方法注射不能序列化。 像春天的 3.2不需要添加CGLIB到您的类路径中,因为 CGLIB类打包在org。springframework和分布式 在spring核心JAR。 这样做既方便 以避免潜在的冲突与其他项目使用不同的版本的CGLIB。

看 commandManager 类 之前的代码片段,你看到Spring容器将 动态覆盖的实施 createCommand() 法。 你 commandManager 类将不会有任何的春天 依赖关系,可以看到在返工的例子:

```
package fiona.apple;

// no more Spring imports!

public abstract class CommandManager {

 public Object process(Object commandState) {
   // grab a new instance of the appropriate Command interface
   Command command = createCommand();
   // set the state on the (hopefully brand new) Command instance
```

```
    command.setState(commandState);
    return command.execute();
  }

  // okay... but where is the implementation of this method?
  protected abstract Command createCommand();
}
```

在客户端类中包含的方法注射(commandManager 在这种情况下),方法 需要签名的注入以下表格:

```
<public|protected> [abstract] <return-type> theMethodName(no-arguments);
```

如果这个方法是 文摘 , 动态生成子类实现了该方法。 否则, 动态生成子类覆盖混凝土方法定义在 原来的类。 例如:

```
<!-- a stateful bean deployed as a prototype (non-singleton) -->
<bean id="command" class="fiona.apple.AsyncCommand" scope="prototype">
<!-- inject dependencies here as required -->
</bean>

<!-- commandProcessor uses statefulCommandHelper -->
<bean id="commandManager" class="fiona.apple.CommandManager">
<lookup-method name="createCommand" bean="command"/>
</bean>
```

bean标识为 commandManager 调用它的 自己的方法 createCommand() 每当它需要一个 的新实例 命令 bean。 你必须小心部署 命令 bean作为原型,如果 这实际上是所需要的。 如果它是部署为一个 singleton ,同样的 实例的 命令 bean返回每个 时间。

提示

感兴趣的读者也可以找到 ServiceLocatorFactoryBean (在 org.springframework.beans.factory.config 包) 使用的。 ServiceLocatorFactoryBean中使用的方法是 类似于另一个实用程序类, ObjectFactoryCreatingFactoryBean ,但它允许 您指定您自己的查询接口而不是一个 spring特定查询界面。 参考这些javadoc 类以及这 博客条目 了解更多信息 ServiceLocatorFactoryBean。

任意方法替代

不太有用的形式的方法注入比查找方法注入 是能够代替任意方法在一个托管bean 另一个方法的实现。 用户可以安全地忽略其余的 节,直到功能实际上是 需要。

与基于xml的配置元数据,您可以使用 替换方法 元素替换现有的方法 实现与另一个部署bean。 考虑以下 类,有一个方法,我们想要computeValue覆盖:

```
public class MyValueCalculator {

public String computeValue(String input) {
  // some real code...
}

// some other methods...

}
```

一个类实现 org.springframework.beans.factory.support.MethodReplacer 接口提供了新方法定义。

```
/** meant to be used to override the existing computeValue(String)
  implementation in MyValueCalculator
*/
public class ReplacementComputeValue implements MethodReplacer {

  public Object reimplement(Object o, Method m, Object[] args) throws Throwable {
      // get the input value, work with it, and return a computed result
      String input = (String) args[0];
      ...
      return ...;
  }
}
```

该bean定义部署原始类和指定 方法覆盖会看起来像这样:

```
<bean id="myValueCalculator" class="x.y.z.MyValueCalculator">

<!-- arbitrary method replacement -->
<replaced-method name="computeValue" replacer="replacementComputeValue">
  <arg-type>String</arg-type>
```

```
</replaced-method>
</bean>

<bean id="replacementComputeValue" class="a.b.c.ReplacementComputeValue"/>
```

您可以使用一个或多个包含 <自变量类型/ > 元素在 <替换方法/ > 元素来表示 方法签名的方法被覆盖。 签名的 参数是必须的,如果方法重载和多 变异存在于类。 为方便起见,这些类型的字符串 子字符串的参数可以是一个完全限定类型名称。 对于 的例子,下面的所有比赛 以 :

```
  java.lang.String
 String
 Str
```

因为数量的参数通常是足够的区分 在每个可能的选择,这个快捷方式可以节省很多的打字,通过 允许您只输入字符串,将匹配的最短的一个 参数类型。

5.5一个Bean范围

当你创建一个bean定义,您创建一个 配方 创建实际的类的实例 定义的bean定义。 认为一个bean定义是一个配方 很重要,因为它意味着,与一个类,你可以创建很多吗 对象实例从单一配方。

你不仅可以控制各种依赖关系和配置 值将被插入一个对象,创建于一个 特定的bean定义,但也 范围 的 从一个特定的对象创建的bean定义。 这种方法是强大的 和灵活,你可以 选择 的范围 您创建的对象通过配置而不必再烤 一个对象的范围在Java类级别。 bean可以被定义 部署在一个数量的范围:盒子之外,Spring框架 支持5个范围,其中三只提供如果你使用 理解网络ApplicationContext 。

以下范围的支持从盒子里。 您还可以创建 一个自定义的范围。

5.3为多。 一个Bean范围

范围	描述
singleton	(默认)范围一个bean定义到一个 对象实例/ Spring IoC容器。
原型	范围一个bean定义任意数量的对象 实例。
请求	一个bean定义范围的生命周期 单一的HTTP请求;也就是说,每个HTTP请求有自己的实例 一个bean创建了后面的一个bean定义。 只有 有效的上下文中的理解网络弹簧 ApplicationContext 。
会话	范围一个bean定义的生命周期 HTTP 会话 。 只有有效的 上下文理解网络的春天 ApplicationContext 。
全球会话	一个bean定义范围的生命周期 全球HTTP 会话 。 通常只有 有效使用时,在一个portlet上下文。 只有有效的上下文中的一个 理解网络春天 ApplicationContext 。

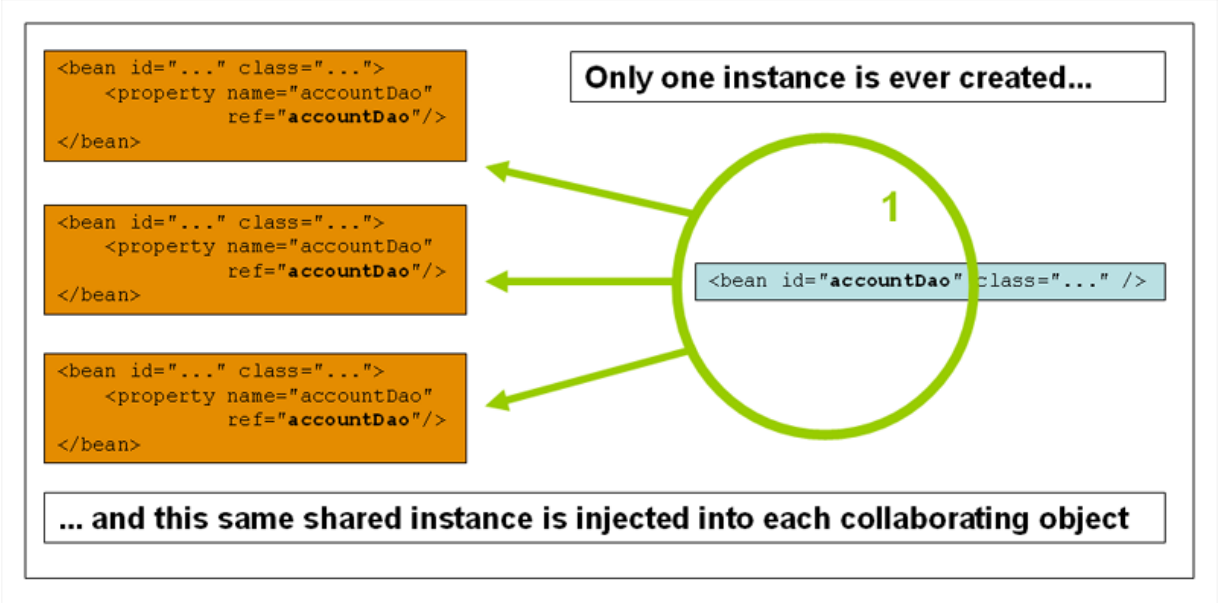
线程级的豆子

在Spring 3.0中,一个 线程范围 是可用的, 但不是默认注册。 有关更多信息,请参见 文档SimpleThreadScope 。 请示如何注册这个或 任何其他自定义范围,见 一个章节使用一个自定义scopea 。

5.5.1A singleton范围

只有一个 共享 一个单例实例bean 管理,所有请求bean id或id匹配的bean 在一个特定的定义结果被返回的bean实例 Spring容器。

换句话说,当你定义一个bean定义和它是 作用域作为一个单例,Spring IoC容器创建 到底 一个 对象的实例定义的bean定义。 这单实例存储在缓存这样的单例豆类, 所有后续请求和引用 这名叫 bean返回缓存的对象。



春天是一个单例bean的概念不同于单例 模式中定义的四人帮(GoF)模式的书。 “四人帮” 单例对象的范围将这样一个 和 只有一个 一个特定的类的实例被创建 每 类加载器 。 春天的范围 单例是最好的形容 每集装箱和每 bean 。 这意味着如果你定义一个bean为一个特定的 类在一个Spring容器,然后Spring容器创建一个 且只有一个 类的实例定义的 bean定义。 单例的范围是默认的范围 春天 。 定义一个bean作为一个单独的XML,你会 写,譬如:

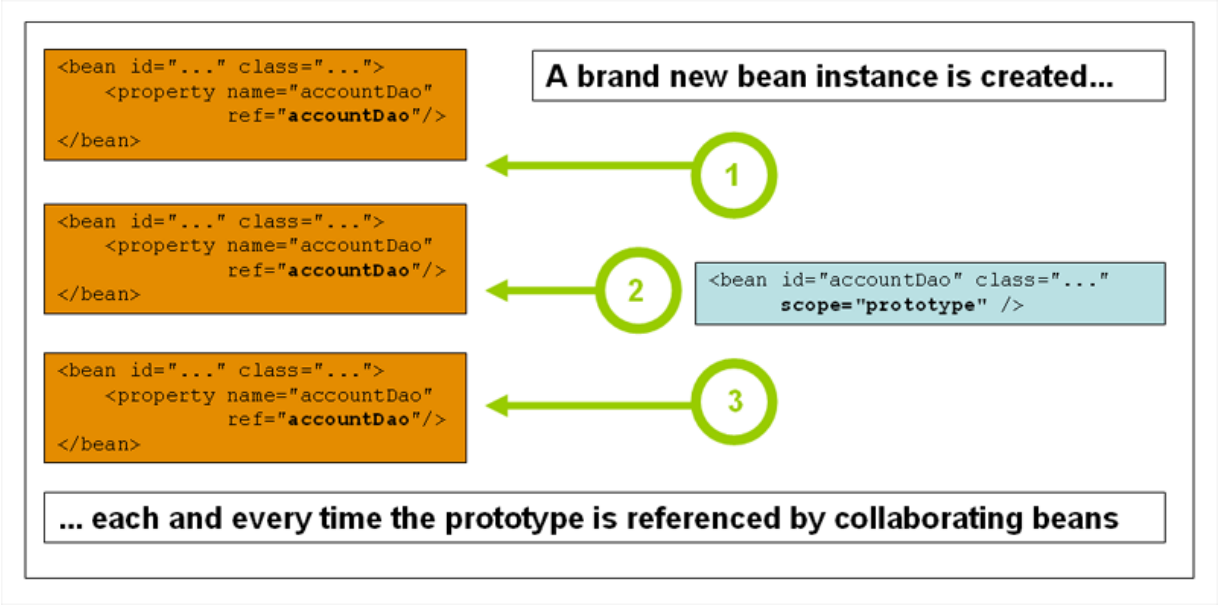
```
<bean id="accountService" class="com.foo.DefaultAccountService"/>

<!-- the following is equivalent, though redundant (singleton scope is the default) -->
<bean id="accountService" class="com.foo.DefaultAccountService" scope="singleton"/>
```

5.5.2A原型范围

单体的,原型范围部署结果。豆 创建一个新的bean实例 每次请求 对该特定bean是由。 即,bean注入另一个 bean或你要求它通过 getBean() 方法调用 在容器上。 作为一个规则,使用原型范围为所有状态 豆类和singleton范围为无状态bean。

下面的图展示了弹簧原型范围。 一个数据访问对象(DAO)不是通常配置为一个 原型,因为典型的刀不持有任何会话状态; 这只是作者更容易重用核心的单例 图。



下面的例子定义了一个bean作为原型在XML:

```
<!-- using spring-beans-2.0.dtd -->
<bean id="accountService" class="com.foo.DefaultAccountService" scope="prototype"/>
```

与其他范围,弹簧不管理完成 生命周期的一个原型bean:容器实例化、配置和 否则组装一个原型对象,并把它递给客户, 没有进一步的记录原型实例。 因此,尽管 初始化 生命周期回调方法被称为 在所有对象无论范围,对于原型、配置 破坏 生命周期回调 不

称为。 客户机代码必须清理 原型作用域对象和释放昂贵的资源 原型bean(s)控股。 让Spring容器释放 持有的资源原型作用域的豆子,尝试使用一个自定义的 豆后处理器 ,这 持有bean,需要清理。

在某些方面,Spring容器的角色对于一个 原型作用域bean是一个替代Java 新 操作符。 所有的生命周期管理过去这一点必须处理 客户端。 (有关的生命周期bean在Spring容器, 看到 SectionA 5.6.1,生命周期callbacksa)。

5.5.3A单例bean与原型bean依赖关系

当你使用单例范围bean与依赖的原型 豆子,要知道 在实例化时解析依赖项 时间 。 因此如果你依赖注入一个原型作用域bean成一个单例作用域的bean,一个新的原型bean实例化和 然后依赖注入到单例bean。 原型实例 是有史以来唯一实例提供给singleton范围 bean。

然而,假设您希望singleton范围bean来获得一个新的 bean的实例在运行时反复原型作用域。 你不能 依赖注入一个原型作用域bean到你的单例bean, 因为注射只发生 一旦 ,当 Spring容器实例化bean是singleton和解决和 其依赖项注入。 如果你需要一个新实例的原型bean 在运行时不止一次看到 SectionA 5 4 6,一个injectiona方法

5.5.4A请求、会话和全球会话作用域

这个 请求 , 会话 ,和 全球会话 范围 只有 如果你使用一个理解网络可用春天 ApplicationContext 实现(如 XmlWebApplicationContext)。 如果你使用这些范围 与常规Spring IoC容器如 ClassPathXmlApplicationContext ,你得到一个 IllegalStateException 抱怨一个未知的 豆范围。

初始web配置

支持bean的作用域的 请求 , 会话 ,和 全球会话 水平 (web范围bean),一些次要的初始配置之前,需要进行 你定义你的豆子。 (这个初始设置 不 所需的标准范围,singleton和原型)。

你如何完成这取决于您的特定初始设置 Servlet环境. .

如果你访问范围bean中Spring Web MVC,实际上,在 一个请求所处理的春天 DispatcherServlet ,或 DispatcherPortlet ,然后没有特殊设置 必要的: DispatcherServlet 和 DispatcherPortlet 已经暴露所有相关 状态。

如果你使用一个Servlet 2.4 + web容器,处理的请求 Spring的DispatcherServlet之外(例如,当使用JSF或 Struts),您需要添加以下 javax.servlet.ServletRequestListener 到 在您的web应用程序的声明 web . xml 文件:

```
<web-app>
...
<listener>
  <listener-class>
     org.springframework.web.context.request.RequestContextListener
  </listener-class>
</listener>
...
</web-app>
```

如果你使用一个年长的web容器(Servlet 2.3),使用提供的 javax.servlet.Filter 实现。 这个 以下代码片段的XML配置必须包含在 web . xml 文件的web应用程序,如果你想 访问web范围豆子在请求之外的春天的 DispatcherServlet在Servlet 2.3容器。 (过滤器映射 取决于周围的web应用程序的配置,所以你必须 适当的改变)。

```
<web-app>
..
<filter>
  <filter-name>requestContextFilter</filter-name>
  <filter-class>org.springframework.web.filter.RequestContextFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>requestContextFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
...
</web-app>
```

DispatcherServlet , RequestContextListener 和 RequestContextFilter 所有做完全一样的 东西,即HTTP请求对象绑定到 线程 这是服务请求。 这使得 豆子是请求和会话范围内可用的更低 调用链。

请求范围

考虑下面的bean定义:

```
<bean id="loginAction" class="com.foo.LoginAction" scope="request"/>
```

Spring容器创建一个新的实例 loginAction bean通过使用 loginAction bean定义每个HTTP 请求。 这是, loginAction bean 是作用域在 HTTP请求级别。 你可以改变内部状态 实例创建尽可能多的你想要的,因为其他实例 创建相同的 loginAction bean定义 不会看到这些变化在状态;他们是特别的一个 单个请求。 当请求完成加工,bean 作用域是这个请求就会被丢弃。

会话范围

考虑下面的bean定义:

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>
```

Spring容器创建一个新的实例 userPreferences bean通过使用 userPreferences bean定义的生命周期 单一的HTTP 会话 。 换句话说, userPreferences 豆是有效的范围 HTTP 会话 水平。 与 会 豆子,你可以改变内部 状态的实例创建尽可能多的你想要的,知道 其他HTTP 会话 实例 也使用相同的实例被创建 userPreferences bean定义看不到这些 状态的变化,因为他们是特定于 单个HTTP 会话 。 当HTTP 会话 最终丢弃,豆吗 这是局限于特定的HTTP 会话 也被丢弃。

全球会话范围

考虑下面的bean定义:

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="globalSession"/>
```

这个 全球会话 范围是相似的 标准HTTP 会话 范围(上面所描述的),和 仅适用于基于portlet的web应用程序的上下文。 这个 portlet规范定义了一个全局的概念 会话 这是所有portlet之间共享 建立了一个单个portlet的web应用程序。 bean定义 全球 会话 范围是作用域(或约束) 一辈子的全球portlet 会话 。

如果你写一个标准的基于servlet的web应用程序,您定义 一个或多个bean有 全球会话 范围, 标准HTTP 会话 范围使用,和 没有 错误了。

范围bean作为依赖项

Spring IoC容器实例化管理不仅你的 对象(豆子),而且连接的合作者(或 依赖关系)。 如果你想注入(例如)一个HTTP请求 作用域 bean到另一个bean,您必须注入AOP代理代替 作用域的bean。 这是,你需要注入一个代理对象,暴露了 相同的公共接口的作用 域对象,但也可以 获取真正的、目标对象相关的范围(例如, 一个HTTP请求)和委托方法调用到真正的对象。

> **注意**
>
> 你 不 需要使用 < aop:作用域的代理/ > 结合豆类 这是作用域作为 单件 或 原型 。

配置在下面的例子只有一线,但它 是重要的理解 一个 为什么 一个 以及 一个 如何 一个 它背后。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd">

  <!-- an HTTP Session-scoped bean exposed as a proxy -->
  <bean id="userPreferences" class="com.foo.UserPreferences" scope="session">

      <!-- instructs the container to proxy the surrounding bean -->
      <aop:scoped-proxy/>
  </bean>

  <!-- a singleton-scoped bean injected with a proxy to the above bean -->
  <bean id="userService" class="com.foo.SimpleUserService">

     <!-- a reference to the proxied userPreferences bean -->
     <property name="userPreferences" ref="userPreferences"/>

  </bean>
</beans>
```

创建这样一个代理,你插入一个孩子 < aop:作用域的代理/ > 元素到一个限定了作用域的bean 定义。 看到 一个章节选择类型的代理创建 和 AppendixA E, XML的基于配置)。 为什么定义bean的作用域在吗 请求 , 会话 , globalSession 和自定义范围

的水平要求 < aop:作用域的代理/ > 元素? 让我们检查 以下单例bean定义和对比它与你需要的 定义为上述范围。 (以下 userPreferences bean定义目前是 不完整。)

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>

<bean id="userManager" class="com.foo.UserManager">
  <property name="userPreferences" ref="userPreferences"/>
</bean>
```

在前面的示例中,单例bean userManager 被注射到HTTP引用 会话 范围bean userPreferences 。 这里的要点是, userManager 豆是一个单例:这将是 实例化 正好一次 每集装箱,和它的 依赖项(在本例中只有一个, userPreferences bean)也只有一次注射。 这意味着 userManager bean只会 操作完全相同 userPreferences 对象, 这就是,一个,原来的注射。

这是 不 你想要的行为当 注射到一个长期寿命较短的作用域bean范围豆, 例如注射一个HTTP 会话 作用域合作作为。豆 成单例bean的依赖。 相反,你需要一个单身 userManager 对象,和生命周期中的一个HTTP 会话 ,你需要一个 userPreferences 对象, 该对象是特定于说HTTP 会话 。 因此容器创建一个 对象,暴露了完全相同的公共接口 userPreferences 类(理想情况下一个对象 是 userPreferences 实例)可以获取真正的 userPreferences 对象的范围界定机制 (HTTP请求, 会话 等等)。 这个 这个代理 对象容器注入到 userManager 豆,这是不知道这个 userPreferences 引用是一个代理。 在这个 例如,当一个 userManager 实例 调用一个方法依赖注入 userPreferences 对象,它实际上是调用 方法在代理。 然后,代理获取真正需要的 userPreferences 对象(在本例中) HTTP 会话 ,代表该方法 调用到检索的实际 userPreferences 对象。

因此你需要以下,正确和完整、配置 当注入 请求- , 会话—— , 和 globalSession-scoped bean到合作 对象:

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session">
  <aop:scoped-proxy/>
</bean>

<bean id="userManager" class="com.foo.UserManager">
  <property name="userPreferences" ref="userPreferences"/>
</bean>
```

选择类型的代理创建

默认情况下,Spring容器创建一个代理,一个bean 这是明显的 < aop:作用域的代理/ > 元素, 一个 基于cglib代理创建类 。

注意:CGLIB代理只有截距公共方法 电话! 不叫非公有制方法在这样一个代理;他们吗 将不会被委托给目标对象的作用域。

或者,您可以配置Spring容器创建 标准JDK基于接口的代理这样的作用域豆类, 指定 假 的值 代理目标类 属性的 < aop:作用域的代理/ > 元素。 使用JDK 基于接口的代理意味着您不需要额外的 图书馆在你的应用程序的类路径效果这样的代理。 然而,这也意味着类的作用域bean必须 实现至少一个界面,和 所有 合作者的作用域bean注入必须参考 bean通过它的一个接口。

```
<!-- DefaultUserPreferences implements the UserPreferences interface -->
<bean id="userPreferences" class="com.foo.DefaultUserPreferences" scope="session">
  <aop:scoped-proxy proxy-target-class="false"/>
</bean>

<bean id="userManager" class="com.foo.UserManager">
  <property name="userPreferences" ref="userPreferences"/>
</bean>
```

对于更详细的信息关于选择基于类的或 基于接口的代理,看 SectionA 9.6,一个mechanismsa代理 。

5.5.5A自定义范围

在Spring 2.0中,bean范围界定机制是可扩展的。 你可以 定义自己的范围,甚至重新定义现有的范围,虽然 后者被认为是糟糕的实践和你 不能 覆盖内置 singleton 和 原型 作用域。

创建一个自定义的范围

将您的自定义范围(s)到Spring容器,你 需要实现 org.springframework.beans.factory.config.Scope 接口,它是这一节中描述。 对于一个想法如何 实现自己的范围,请参阅 范围 实现,提供Spring框架本身和 这个 范围Javadoc ,这解释了需要实现的方法 在更多的细节。

这个 范围 界面有四个方法来获取 对象的范围,删除它们从范围,并允许他们 摧毁了。

以下方法返回对象从底层的范围。 会话作用域实现,例如,返回 会话范围内的bean(如果它不存在,该方法返回一个新的 bean的实例之后,将它绑定到会话为未来 引用)。

```
Object get(String name, ObjectFactory objectFactory)
```

以下方法删除对象从底层的范围。 会话作用域实现例如,删除会话范围内 从底层的会话bean。 应该返回的对象,但你 可以返回null如果对象与指定的名字不是吗 发现。

```
Object remove(String name)
```

下面的方法注册回调的范围应该 当它被销毁或执行规定的对象范围 被摧毁。 请参考Javadoc或弹簧范围实现 更多的信息在破坏回调。

```
void registerDestructionCallback(String name, Runnable destructionCallback)
```

下面的方法获得对话标识 潜在的范围。 这个标识符是为每一个范围不同。 对于一个 会话作用域实现,这个标识符可以会话 标识符。

```
String getConversationId()
```

使用一个自定义的范围

在编写和测试一个或多个自定义 范围 实现,你需要做的 Spring容器意识到你的新范围(年代)。 下面的方法是 中央方法来注册一个新的 范围 与Spring容器:

```
void registerScope(String scopeName, Scope scope);
```

该方法中声明的 ConfigurableBeanFactory 接口,它 可以在大多数的混凝土 ApplicationContext 实现 船与Spring BeanFactory属性。通过

的第一个参数 套色定位观察仪(. .) 方法是唯一的名称与范围关联;这样的例子 名字在Spring容器本身 singleton 和 原型 。 第二个参数 套色定位观察仪(. .) 方法是一个实际的实例 定制 范围 实现 你想注册和使用。

假设您编写自定义 范围 实现,然后注册 它如下。

注意

下面的例子使用 SimpleThreadScope 哪一个 包含在春天,但不是默认注册。 这个 指令将同样的您自己的自定义 范围 实现。

```
Scope threadScope = new SimpleThreadScope();
beanFactory.registerScope("thread", threadScope);
```

然后,您创建的bean定义范围规则的遵守 您的自定义 范围 :

```
<bean id="..." class="..." scope="thread">
```

用一个自定义的 范围 的实现, 你是不限于程序性登记的范围。 你可以 也做 范围 登记 通过声明,使用 CustomScopeConfigurer 类:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd">

  <bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
     <property name="scopes">
        <map>
           <entry key="thread">
              <bean class="org.springframework.context.support.SimpleThreadScope"/>
           </entry>
        </map>
     </property>
  </bean>

  <bean id="bar" class="x.y.Bar" scope="thread">
     <property name="name" value="Rick"/>
     <aop:scoped-proxy/>
  </bean>
```

```
  <bean id="foo" class="x.y.Foo">
    <property name="bar" ref="bar"/>
  </bean>

</beans>
```

注意

当你把< aop:作用域的代理/ >在 FactoryBean 实现,它是 工厂bean本身是作用域,而不是返回的对象 getObject() 。

5.6一个定制bean的性质

5.6.1A生命周期回调

可以与容器的管理bean的生命周期,你 可以实现春天 InitializingBean 和 DisposableBean 接口。 这个 容器调用 afterPropertiesSet() 为 前和 destroy() 后者允许 bean来执行特定操作在初始化和销毁 你的豆子。

提示

jsr - 250 @PostConstruct 和 @PreDestroy 注释通常 被认为是最佳实践在现代接受生命周期回调 Spring 应用程序。 使用这些注释意味着您的bean是不 耦合弹簧特定的接口。 详情见 SectionA 5 9 6,一个 @PostConstruct 和 @PreDestroy 一个 。

如果你不想使用jsr - 250注释但你仍然 想消除耦合考虑使用init方法和销毁方法 对象定义元数据。

在内部,Spring框架使用 BeanPostProcessor 实现 处理任何回调接口可以找到并调用合适的 方法。 如果你需要定制功能或其他生命周期行为弹簧 不提供开箱即用的,您可以实现吗 BeanPostProcessor 你自己。 更多 信息,请参阅 SectionA 5.8,一个容器扩展Pointsa 。

除了初始化和销毁回调, spring管理对象也可以实现 生命周期 接口,以便这些对象 可以参与启动和关闭过程驱动的吗 容器的自己的生命周期。

生命周期回调接口描述 部分。

初始化回调

这个 org.springframework.beans.factory.InitializingBean 接口允许一个bean来执行初始化工作毕竟 必要的属性在bean已经设定的容器。 这个 InitializingBean 接口指定了一个 单一的方法:

```
void afterPropertiesSet() throws Exception;
```

建议您不要使用 InitializingBean 接口,因为它 不必要的代码来春夫妻。 另外,使用 @PostConstruct 注释 或指定一个POJO 初始化方法。 对于基于xml的配置元数据, 你使用 init方法 属性指定的名字 的方法有一个空白无参数签名。 例如, 以下定义:

```
<bean id="exampleInitBean" class="examples.ExampleBean" init-method="init"/>
```

```
public class ExampleBean {

  public void init() {
    // do some initialization work
  }
}
```

... 完全一样...

```
<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>
```

```
public class AnotherExampleBean implements InitializingBean {

  public void afterPropertiesSet() {
    // do some initialization work
  }
}
```

... 但没有几个春天的代码。

破坏回调

实施 org.springframework.beans.factory.DisposableBean 接口允许一个bean来得到一个回调当容器包含 它被摧毁。 这个 DisposableBean 接口指定了一个简单的方法:

```
void destroy() throws Exception;
```

建议您不要使用 DisposableBean 回调接口,因为 它不必要的代码来春夫妻。 另外,使用 @PreDestroy 注释 或指定一个 泛型方法,支持bean定义。 与基于xml的 配置元数据,您使用 销毁方法 属性 < bean / > 。 例如, 以下定义:

```
<bean id="exampleInitBean" class="examples.ExampleBean" destroy-method="cleanup"/>
```

```
public class ExampleBean {

  public void cleanup() {
    // do some destruction work (like releasing pooled connections)
  }
}
```

... 完全一样...

```
<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>
```

```
public class AnotherExampleBean implements DisposableBean {

  public void destroy() {
    // do some destruction work (like releasing pooled connections)
  }
}
```

... 但没有几个春天的代码。

默认的初始化和销毁方法

当你写回调函数,初始化和销毁方法做 不使用spring特定 InitializingBean 和 DisposableBean 回调接口,你 通常写方法等名称 init() , 初始化() , dispose() ,所以 在。 理想情况下,名称这样的生命周期回调方法 标准化的跨项目以便所有开发人员使用相同的方法 名称和确保一致性。

您可以配置Spring容器 看 指定的初始化和销毁的回调方法上的名字 每一 bean。 这意味着,你,作为一个应用程序 开发人员,可以编写应用程序类和使用一个初始化 调称为 init() ,无需配置 一个 init方法= “init” 属性与每个bean 定义。 Spring IoC容器 调用该方法当bean 创建(并符合标准的生命周期回调契约 前面描述的)。 这个特性还执行一致的命名 公约初始化和销毁方法回调。

假设你的初始化回调方法的命名 init() 并摧毁回调方法的命名 destroy() 。 你的类将像中的类 下面的例子。

```
public class DefaultBlogService implements BlogService {

  private BlogDao blogDao;

  public void setBlogDao(BlogDao blogDao) {
    this.blogDao = blogDao;
  }

  // this is (unsurprisingly) the initialization callback method
  public void init() {
    if (this.blogDao == null) {
      throw new IllegalStateException("The [blogDao] property must be set.");
    }
  }
}
```

```
<beans default-init-method="init">

  <bean id="blogService" class="com.foo.DefaultBlogService">
    <property name="blogDao" ref="blogDao" />
  </bean>

</beans>
```

的存在 默认init方法 属性 在顶级 < bean / > 元素属性 使Spring IoC容器识别方法被称为 INIT 在bean作为初始化方法的回调。 当一个bean创建并组装,如果有这样一个bean类 法,它是在适当的时候调用。

你配置回调的销毁方法同样(XML), 通过使用 默认销毁方法 属性 顶级 < bean / > 元素。

在现有的bean类已经回调方法,是吗 命名不大会,您可以覆盖默认的 指定(XML),方法名使用 init方法 和 销毁方法 属性的< bean / >本身。

Spring容器保证配置的初始化 回调后立即调用bean提供所有 依赖关系。 因此,初始化的回调是呼吁生豆 参考,这意味着AOP拦截器等等都没有 应用到bean。 一个目标bean是完全创建 第一 , 然后 AOP代理(例)以其拦截器链是应用。 如果目标bean和代理是单独定义,你的代码甚至可以与其进行交互 生目标bean,绕过代理。 因此,这将是不一致的, 应用拦截器到init方法,因为这样做会夫妇 目标bean的生命周期与其代理/拦截器和离开 奇怪的语义代码交互时直接向原始目标 bean。

结合生命周期机制

在Spring 2.5中,您有三个选项控制bean 生命周期行为: InitializingBean 和 DisposableBean 回调 接口;自定义 init() 和 destroy() 方法; @PostConstruct 和 @PreDestroy 注释 。 你可以 结合这些机制来控制一个bean。

> **注意**
>
> 如果多个生命周期机制配置为一个bean,并 每个机制都配置了一个不同的方法名称,然后每个 配置的方法是执行在下面列出的顺序。 然而,如果 配置相同的方法名,例如, init() 对于一个初始化方法——超过 其中的一个生命周期机制,该方法被执行一次, 在前一节中解释。

多个生命周期机制配置为相同的bean, 不同的初始化方法,被称为如下:

- 的方法 @PostConstruct
- afterPropertiesSet() 所定义的 InitializingBean 回调 接口
- 一个自定义配置 init() 方法

破坏的方法被称为以相同的顺序:

- 的方法 @PreDestroy
- destroy() 所定义的 DisposableBean 回调 接口
- 一个自定义配置 destroy() 方法

启动和关闭回调

这个 生命周期 接口定义了 基本方法的任何对象都有自己的生命周期需求 (如启动和停止一些背景过程):

```
public interface Lifecycle {
  void start();
  void stop();
  boolean isRunning();
}
```

任何spring管理对象可以实现该接口。 然后,当 本身的ApplicationContext启动和停止,它会级联这些 调用所有生命周期实现定义在这个上下文。 它 这是否通过指派给一个吗 LifecycleProcessor :

```
public interface LifecycleProcessor extends Lifecycle {
  void onRefresh();
  void onClose();
}
```

注意, LifecycleProcessor 是 本身的延伸 生命周期 接口。 它还添加了其他两个方法来应对上下文 被刷新和关闭。

启动和关闭的顺序调用可能是重要的。 如果一个 “依赖” 关系存在任何两个物体之间的依赖 侧将开始 在 它的依赖,它将 停止之前 它的依赖。 然而,有时 直接依赖关系是未知的。 你可能只知道对象的一个 某些类型的对象之前,应该开始另一个类型。 在那些 情况下, SmartLifecycle 接口 定义了另一个选择,即 getPhase() 方法定义在它的超接口相反, 阶段性 。

```
public interface Phased {
  int getPhase();
```

```
}

public interface SmartLifecycle extends Lifecycle, Phased {
  boolean isAutoStartup();
  void stop(Runnable callback);
}
```

在启动时,这些对象与最低的阶段开始前, 当停止,后面是相反的顺序。 因此,一个对象,该对象 实现 SmartLifecycle 和他 getPhase()方法返回 整数最小值 将 最早开始和最后停止。 在另一端的 谱,相位值 integer . max_value之间 将 表明该对象应该开始于去年和停止第一 (可能是因为它依赖于其他进程运行)。 当 考虑到相位值,同样重要的是知道的 默认的阶段对于任何“正常” 生命周期 对象,没有实现 SmartLifecycle 将是0。 因此,任何 负相位值将显示一个对象应该开始之前 这些标准组件(和停止在他们之后),反之亦然,任何 积极的相位值。

正如你可以看到停止法所定义的 SmartLifecycle 接受一个回调。 任何 实现 必须 调用这个回调的run() 方法之后,关闭过程完成实现的。 这 使异步关闭在必要时由于默认 实施 LifecycleProcessor 界面, DefaultLifecycleProcessor ,会等待 其超时的值对象组在每个阶段 调用这个回调。 每阶段的默认超时时间是30秒。 你 可以覆盖默认的生命周期处理器实例,通过定义一个豆吗 命名为“lifecycleProcessor”范围内。 如果你只是想 修改超时,然后定义以下会 足够的:

```
<bean id="lifecycleProcessor" class="org.springframework.context.support.DefaultLifecycleProcessor">
  <!-- timeout value in milliseconds -->
  <property name="timeoutPerShutdownPhase" value="10000"/>
</bean>
```

正如前面提到的, LifecycleProcessor 接口定义了回调方法的更新和关闭 背景和。 后者将只是把关机过程好像 停止()被称为明确,但它会发生当上下文 是关闭的。 “刷新”回调另一方面使另一个 特性的 SmartLifecycle 豆子。 当 上下文是刷新(毕竟已经实例化的对象, 初始化),将调用的回调,而那时 缺省生命周期处理器会检查返回的布尔值 每个 SmartLifecycle 对象的 isAutoStartup() 法。 如果“真正的”,然后, 对象将开始在这一点上,而不是等待一个显式的 调用上下文的或自己的start()方法(不像 上下文刷新,上下文的开始并不自动发生对于一个 标准上下文实现)。 “相”的价值以及任何 “依赖”关系将决定启动顺序相同 方法如上所述。

关闭Spring IoC容器优雅地在非web 应用

注意

本节仅适用于非web应用程序。 春天的 网络 ApplicationContext 实现已经有代码后关闭Spring IoC 集装箱优雅地在有关web应用程序关闭 下来。

如果您正在使用Spring的IoC容器在非web应用程序 环境;例如,在一个富客户端桌面环境;你 注册一个关机钩与JVM。 这样做可以确保一个优雅 关闭并调用相关的破坏方式单例bean 所以,所有资源被释放。 当然,您还必须配置 和实现这些破坏回调正确。

注册一个关机钩,你所说的 registerShutdownHook() 方法中声明 在 AbstractApplicationContext 类:

```
import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public final class Boot {

  public static void main(final String[] args) throws Exception {
    AbstractApplicationContext ctx
      = new ClassPathXmlApplicationContext(new String []{"beans.xml"});

    // add a shutdown hook for the above context...
    ctx.registerShutdownHook();

    // app runs here...

    // main method exits, hook is called prior to the app shutting down...
  }
}
```

5.6.2A ApplicationContextAware 和 BeanNameAware

当一个 ApplicationContext 创建一个 类,实现了 org.springframework.context.ApplicationContextAware 接口,类提供一个参考的 ApplicationContext 。

```
public interface ApplicationContextAware {

  void setApplicationContext(ApplicationContext applicationContext) throws BeansException;
}
```

因此大豆可以编程操作了 ApplicationContext 创造了他们, 通过 ApplicationContext 界面, 或者通过铸造引用到一个已知的子类这个接口,例如 ConfigurableApplicationContext ,它公开了 额外的功能。 一个使用将是程序化的检索 其他豆类。 有时这种能力是有用的;然而,在一般的你 应该避免使用它,因为它夫妇代码以春季和不遵循 控制反转的风格,合作者是提供给豆子 作为属性。 其他方法的ApplicationContext提供 文件资源,发布应用程序事件,和访问 MessageSource。 这些额外的功能描述 SectionA 5.14,一个附加的功能 ApplicationContext 一个

Spring 2.5的,自动装配是另一个替代获得 参考 ApplicationContext 。 这个 “传统” 构造函数 和 byType 自动装配模式(详见 SectionA 5 4 5,一个collaboratorsa自动装配)可以提供一个依赖的类型 ApplicationContext 对于一个构造函数 参数或setter方法参数,分别。 更多的灵活性, 包括自动装配领域的能力和多参数的方法, 使用新的基于注解的自动装配功能。 如果你这样做了, ApplicationContext autowired成是 域、构造参数或方法参数,是期待 ApplicationContext 如果字段类型, 构造函数或方法有问题 @ autowired 注释。 更多 信息,请参阅 SectionA 5 9 2,一个 @ autowired 一个 。

当创建一个类,它实现了ApplicationContext的 org.springframework.beans.factory.BeanNameAware 接口,类提供一个名称的引用定义在 它相关的对象定义。

```
public interface BeanNameAware {

  void setBeanName(string name) throws BeansException;
}
```

调用回调后的人口正常bean属性但 在一个初始化的回调如 InitializingBean 年代 afterPropertiesSet 或一个自定义的init方法。

5.6.3A其他 意识到 接口

除了 ApplicationContextAware 和 BeanNameAware 上面所讨论的,春天 提供一系列的 意识到 接口, 让豆子来指示容器,他们需要一个特定的 基础设施 依赖项。 最重要的 意识到 接口是总结如下,作为 一般来说,这个名字是一个好迹象的依赖 类型:

为多5 4 a 意识到 接口

名称	注入依赖	解释……
ApplicationContextAware	宣布 ApplicationContext	SectionA 5 6 2,一个 ApplicationContextAware 和 BeanNameAware 一个
ApplicationEventPublisherAware	事件发布者的封闭 ApplicationContext	SectionA 5.14,一个附加的功能 ApplicationContext 一个
BeanClassLoaderAware	类加载器来加载bean使用 类。	SectionA 5 3 2,一个beansa实例化
BeanFactoryAware	宣布 BeanFactory	SectionA 5 6 2,一个 ApplicationContextAware 和 BeanNameAware 一个
BeanNameAware	声明bean的名称	SectionA 5 6 2,一个 ApplicationContextAware 和 BeanNameAware 一个
BootstrapContextAware	资源适配器 BootstrapContext 容器运行在。 通常只在JCA意识 ApplicationContext 年代	ChapterA 25, JCA CCI
LoadTimeWeaverAware	定义 韦弗 加工 类定义在加载时间	SectionA 9 8 4,一个装入时编织与 AspectJ在春季Frameworka
MessageSourceAware	配置策略来解决信息(支持参数化和 国际化)	SectionA 5.14,一个附加的功能 ApplicationContext 一个
NotificationPublisherAware	Spring JMX通知出版商	SectionA 24.7,一个Notificationsa
PortletConfigAware	电流 PortletConfig 容器运行在。 有效只	ChapterA 20, Portlet MVC框架

	有在理解网络弹簧 ApplicationContext	
PortletContextAware	电流 PortletContext 容器运行在。 有效只有在理解网络弹簧 ApplicationContext	ChapterA 20, Portlet MVC框架
ResourceLoaderAware	配置的装载机为低级的 资源	ChapterA 6, 资源
ServletConfigAware	电流 ServletConfig 容器运行在。 有效只有在理解网络弹簧 ApplicationContext	ChapterA 17, Web MVC框架
ServletContextAware	电流 ServletContext 容器运行在。 有效只有在理解网络弹簧 ApplicationContext	ChapterA 17, Web MVC框架

请再次注意,使用这些接口的关系你的代码到春天 API和不遵循控制反转的风格。 因此,他们是 推荐用于基础设施,需要编程访问 bean 容器。

5.7一个Bean定义继承

一个bean定义可以包含大量的配置信息, 包括构造函数参数,属性值和特定容器 信息,如初始化方法,静态工厂方法的名字, 等等。 一个孩子bean定义继承配置数据从一个家长 定义。 孩子定义可以覆盖一些值,或添加其他人, 作为需要。 父母和孩子使用bean定义可以节省很多 打字。 实际上,这是一个形式的模板。

如果您使用一个 ApplicationContext 接口编程,孩子bean定义为代表的 ChildBeanDefinition 类。 大多数用户不工作 他们在这一水平,相反配置bean定义 在类似的声明 ClassPathXmlApplicationContext 。 当你使用 基于xml的配置元数据,您显示孩子的bean定义 使用 父 属性,指定父bean 作为这个属性的值。

```
<bean id="inheritedTestBean" abstract="true"
   class="org.springframework.beans.TestBean">
  <property name="name" value="parent"/>
  <property name="age" value="1"/>
</bean>

<bean id="inheritsWithDifferentClass"
     class="org.springframework.beans.DerivedTestBean"
     parent="inheritedTestBean" init-method="initialize">

  <property name="name" value="override"/>
  <!-- the age property value of 1 will be inherited from  parent -->

</bean>
```

一个孩子bean定义使用bean类从父定义 如果没有指定,但是也可以重写它。 在后一种情况下, 孩子bean类必须兼容父,也就是说,它必须 接受父母的财产价值。

一个孩子bean定义继承的构造函数的参数值,属性 值,方法覆盖父,选项来添加新的 值。 任何初始化方法,破坏的方法,和/或 静态工厂方法设置,您指定将 覆盖相应的父设置。

其余设置 总是 来自 孩子的定义: 取决于 , 自动装配 模式 , 依赖检查 , singleton , 范围 , 懒惰 INIT 。

前面的例子明确标志着父的bean定义为 摘要通过使用 文摘 属性。 如果父 定义并没有指定一个类,明确标记父bean 定义为 文摘 是必需的,如下:

```
<bean id="inheritedTestBeanWithoutClass" abstract="true">
   <property name="name" value="parent"/>
   <property name="age" value="1"/>
</bean>

<bean id="inheritsWithClass" class="org.springframework.beans.DerivedTestBean"
   parent="inheritedTestBeanWithoutClass" init-method="initialize">
  <property name="name" value="override"/>
  <!-- age will inherit the value of 1 from the parent bean definition-->
</bean>
```

父豆不能被实例化在它自己的,因为它是 不完整的,而且它也是明确标记为 文摘 。 当一个定义是 文摘 喜欢这个,它是可用的只有作为一个纯粹 模板的bean定义,作为父母对孩子的定义 定义。 尝试使用这样一个 文摘 父bean 在它自己的,称之为一个ref属性的另一个bean或做 一个明确的 getBean() 调用父bean id,返回一个错误。 同样,容器的内部 preInstantiateSingletons() 方法忽略了bean 定义为抽象的定义。

注意

ApplicationContext pre实例化所有 单例对象默认情况下。 因此,它是重要的(至少对 单例bean),如果你有一个(父)bean定义你 打算只使用作为模板,这个定义指定了一个类, 你必须确保设置 文摘 属性 真正的 ,否则应用程序上下文将 实际上(试图)预实例化的 文摘 bean。

5.8容器扩展点

通常,应用程序开发人员不需要子类 ApplicationContext 实现类。 相反,Spring IoC容器可以延长插入 实现特殊的集成接口。 接下来的几个部分 描述这些集成接口。

5.8.1A定制bean使用 BeanPostProcessor

这个 BeanPostProcessor 接口定义了 回调方法 那你可以实现来提供 你自己的(或覆盖容器的默认)实例化的逻辑, 依赖性解析逻辑,等等。 如果你想实现一些 自定义逻辑Spring容器实例化后完成, 配置和初始化一个bean,您可以插入一个或 更多的 BeanPostProcessor 实现。

您可以配置多个 BeanPostProcessor 情况下,您可以控制顺序这些 BeanPostProcessor 年代执行通过设置 秩序 财产。 你可以设置这个属性只是如果 BeanPostProcessor 实现了 下令 接口;如果你写你自己的 BeanPostProcessor 你应该考虑 实施 下令 界面太。 对于 进一步的细节,请咨询的Javadoc BeanPostProcessor 和 下令 接口。 参见下面的说明上 编程式注册 BeanPostProcessors

注意

BeanPostProcessor 年代操作bean(或对象) 实例 ;也就是说,Spring IoC容器 实例化一个bean实例和 然后 BeanPostProcessor 年代做他们的工作。

BeanPostProcessor 年代的范围被 每集装箱 。 这是唯一相关的如果你是 使用容器层次结构。 如果你定义一个 BeanPostProcessor 在一个容器,它 将 只有 豆子,后处理 集装箱。 换句话说,bean中定义的一个集装箱不是 后期处理的 BeanPostProcessor 定义在另一个 容器,即使两个容器都属于同一个层次。

改变实际的bean定义(即。 , 蓝图 定义bean),您需要使用一个不是 BeanFactoryPostProcessor 所述 在 SectionA 5 8 2,一个定制配置元数据与一个 BeanFactoryPostProcessor 一个 。

这个 org.springframework.beans.factory.config.BeanPostProcessor 接口由完全两个回调方法。 当这样一个类 注册为后处理与容器,每个bean实例 这是由容器、后处理器得到一个回调的 容器既 之前 容器初始化 方法(如InitializingBean的afterPropertiesSet() 和任何宣布init方法)被称为以及 在 任何bean初始化回调。 后处理器可以 任何动作与bean实例,包括忽略了回调 完全。 一个豆后处理器通常检查回调 接口或可能包装一个bean与一个代理。 一些Spring AOP 基础设施类都实现为bean后处理器在秩序 提供代理包装逻辑。

一个 ApplicationContext 自动检测 任何bean中定义的 配置元数据而实现 BeanPostProcessor 接口。 这个 ApplicationContext 寄存器这些bean作为 后处理器,这样他们可以被称为后在bean创建。 豆后处理器可以部署在容器就像任何其他 豆子。

编程方式注册 BeanPostProcessors

而推荐方法 BeanPostProcessor 登记是通过 ApplicationContext 自动检测(如上所述),它也是 可以注册他们 编程 反对 ConfigurableBeanFactory 使用 addBeanPostProcessor 法。 这可能是有用的 当需要评估条件逻辑注册之前,甚至 抄袭bean后处理器在上下文层次结构中。 注意 然而, BeanPostProcessors 添加编程 不尊重 下令 接口 。 这是 秩序的 登记 ,决定了执行顺序。 还请注意 这 BeanPostProcessors 注册 以编程方式总是加工前注册通过 自动检测、不管任何明确的顺序。

BeanPostProcessors 和AOP 汽车代理

类,实现了 BeanPostProcessor 接口是 特殊 并以不同的方式对待 集装箱。 所有 BeanPostProcessors 和 豆类,他们直接引用 是 在启动时实例化的一部分,特殊的启动阶段 ApplicationContext 。 接下来,所有 BeanPostProcessors 被注册在一个 排序的时尚和适用于所有进一步的豆子在容器。 因为AOP汽车代理被实现为一个 BeanPostProcessor 本身,既不 BeanPostProcessors 还是豆子他们参考 直接有资格获得汽车代理,因此没有方面编织 到他们。

任何这样的bean,您应该看到一个信息日志信息: 一个 豆foo没有资格获得处理的所有 BeanPostProcessor 接口(例如:为不合格 汽车代理) 一个 。

请注意,如果你有豆子连接到你的 BeanPostProcessor 使用自动装配或 @ resource (可能跌回自动装配),春天可能获得意想不到的豆子当搜索类型匹配依赖候选人, 因而使他们得到汽车代理或其他类型的bean后处理。 例如,如果你有一个依赖项标注 @ resource 如果该字段/ setter的名字不直接对应于声明的名称bean和 没有名字的属性是使用,然后春天将访问其他豆类为匹配他们的类型。

下面的例子展示了如何编写,注册和使用 BeanPostProcessors 在一个 ApplicationContext 。

例如:你好世界, BeanPostProcessor 风格

这第一个示例演示了基本的用法。 这个例子显示了一个 定制 BeanPostProcessor 实现 调用 toString() 每个bean的方法 因为它是由容器和打印生成的字符串 系统控制台。

下面找到自定义 BeanPostProcessor 实现类 定义:

```
package scripting;

import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.beans.BeansException;

public class InstantiationTracingBeanPostProcessor implements BeanPostProcessor {

  // simply return the instantiated bean as-is
  public Object postProcessBeforeInitialization(Object bean, String beanName)
                                                    throws BeansException {
      return bean; // we could potentially return any object reference here...
  }

  public Object postProcessAfterInitialization(Object bean, String beanName)
                                                    throws BeansException {
      System.out.println("Bean '" + beanName + "' created : " + bean.toString());
      return bean;
  }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:lang="http://www.springframework.org/schema/lang"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/lang
        http://www.springframework.org/schema/lang/spring-lang.xsd">

  <lang:groovy id="messenger"
        script-source="classpath:org/springframework/scripting/groovy/Messenger.groovy">
      <lang:property name="message" value="Fiona Apple Is Just So Dreamy."/>
  </lang:groovy>

  <!--
      when the above bean (messenger) is instantiated, this custom
      BeanPostProcessor implementation will output the fact to the system console
  -->
  <bean class="scripting.InstantiationTracingBeanPostProcessor"/>

</beans>
```

注意, InstantiationTracingBeanPostProcessor 只是 定义。 它甚至没有一个名字,因为它是一个bean可以 是依赖注入就像任何其他bean。 (前面的 配置还定义了一个bean是由一个Groovy脚本。 这个 动态语言支持Spring 2.0中详细章节的题目为 ChapterA 28日 动态语言支持)。

以下简单的Java应用程序执行前面的代码和 配置:

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.scripting.Messenger;

public final class Boot {

  public static void main(final String[] args) throws Exception {
      ApplicationContext ctx = new ClassPathXmlApplicationContext("scripting/beans.xml");
      Messenger messenger = (Messenger) ctx.getBean("messenger");
      System.out.println(messenger);
  }
}
```

前应用程序的输出类似 以下:

```
Bean 'messenger' created : org.springframework.scripting.groovy.GroovyMessenger@272961
```

```
org.springframework.scripting.groovy.GroovyMessenger@272961
```

示例: RequiredAnnotationBeanPostProcessor

使用回调接口或注释联同一个 定制 BeanPostProcessor 实现 是一个普遍的方法扩展Spring IoC容器。 一个例子是 春天的 RequiredAnnotationBeanPostProcessor 一个一 BeanPostProcessor 实现 船只与春天分布,确保JavaBean 在bean的属性标记为(任意)注解 实际上(配置为)依赖注入与价值。

5.8.2A定制配置元数据与一个 BeanFactoryPostProcessor

接下来,我们将扩展点看 org.springframework.beans.factory.config.BeanFactoryPostProcessor 。 这个接口的语义是相似的 BeanPostProcessor ,其中一个主要的 的区别: BeanFactoryPostProcessor 年代操作 bean配置元数据 ;也就是说,Spring IoC 容器允许 BeanFactoryPostProcessors 阅读 配置元数据和潜在地改变它 之前 容器实例化bean的任何其他 比 BeanFactoryPostProcessors 。

您可以配置多个 BeanFactoryPostProcessors ,你可以控制的顺序 这些 BeanFactoryPostProcessors 执行的 设置 秩序 财产。 但是,你只能设置 这个属性如果 BeanFactoryPostProcessor 实现了 下令 接口。 如果你写你自己的 BeanFactoryPostProcessor ,你应该 考虑实施 下令 接口 太。 咨询的Javadoc BeanFactoryPostProcessor 和 下令 接口为更多的细节。

> **注意**
>
> 如果你想改变实际的bean 实例 (即。 ,创建的对象从配置元数据),然后你 而是需要使用 BeanPostProcessor (上面所描述的在 SectionA 5 8 1,一个定制bean使用 BeanPostProcessor 一个)。 而 它在技术上是可能的工作与bean实例在一个 BeanFactoryPostProcessor (如。 ,使用 BeanFactory.getBean()),这样做的原因 过早的bean实例化,违反了标准集装箱的生命周期。 这可能导致负面影响比如绕过bean邮报 处理。
>
> 同时, BeanFactoryPostProcessors 是作用域 每集装箱 。 这是唯一相关的如果你是 使用容器层次结构。 如果你定义一个 BeanFactoryPostProcessor 在一个 容器,它将 只有 被应用到bean 定义在容器。 在一个容器的Bean定义 不会的后续处理 BeanFactoryPostProcessors 在另一个容器,甚至 如果两个容器都属于同一个层次。

一个bean工厂后处理器是当它是自动执行 宣布在一个 ApplicationContext , 为了申请更改配置元数据定义 集装箱。 弹簧包括一系列预定义的bean工厂 后处理器,如 PropertyOverrideConfigurer 和 来完成 。 一个自定义 BeanFactoryPostProcessor 也可以使用, 例如,注册自定义属性编辑器。

一个 ApplicationContext 自动 检测到任何bean的部署到它实现 BeanFactoryPostProcessor 接口。 它 使用这些bean作为 bean工厂后处理器,在 适当的时间。 你可以将这些部署后处理器bean作为你 将任何其他bean。

> **注意**
>
> 与 BeanPostProcessor 年代,你通常 不需要配置吗 BeanFactoryPostProcessor 年代 对于延迟初始化。 如果没有其他bean引用 豆(工厂)后处理程序 , 后处理器将不会得到实例化,在所有。 因此,标记它 延迟初始化将被忽略, 豆(工厂)后处理程序 将 实例化急切地即使你设置 默认懒init 属性 真正的 在声明的 < bean / > 元素。

示例: 来完成

你使用 来完成 到 外部化属性值从一个bean定义在一个单独的 文件使用标准的Java 属性 格式。 这样做使人部署一个应用程序来定制 与环境相关的属性如数据库url和密码, 没有风险的复杂性或修改主XML定义文件 或文件的容器。

考虑下面的基于xml的配置元数据片段, 一个 数据源 用占位符 值被定义。 这个例子显示了属性的配置 外部 属性 文件。 在运行时,一个 来完成 应用于 元数据,将取代一些属性的数据源。 值 来取代被指定为 占位符 表单的 $ { property - name }这是蚂蚁/ log4j / JSP EL风格。

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="locations" value="classpath:com/foo/jdbc.properties"/>
</bean>

<bean id="dataSource" destroy-method="close"
    class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName" value="${jdbc.driverClassName}"/>
  <property name="url" value="${jdbc.url}"/>
```

```
  <property name="username" value="${jdbc.username}"/>
  <property name="password" value="${jdbc.password}"/>
</bean>
```

实际的值来自另一个文件在标准的Java 属性 格式:

```
jdbc.driverClassName=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:hsql://production:9002
jdbc.username=sa
jdbc.password=root
```

因此,字符串 $ { jdbc用户名} 代替 在运行时的价值' sa ',这同样适用于其他占位符 值,匹配键在属性文件。 这个 来完成 检查 在大多数属性和属性占位符的bean定义。 此外,占位符前缀和后缀可以定制。

与 上下文 名称空间中引入弹簧 2.5,可以配置属性占位符和一个专用的 配置元素。 一个或多个位置可以提供作为一个 逗号分隔列表 位置 属性。

```
<context:property-placeholder location="classpath:com/foo/jdbc.properties"/>
```

这个 来完成 不仅 查找的属性 属性 文件 你指定。 默认情况下它还检查对Java 系统 属性如果不能找到属性 在指定的属性文件。 您可以定制这个行为通过设置 systemPropertiesMode configurer财产与 以下三个支持整数价值:

- 从来没有 (0):从不检查系统属性
- 撤退 (1):检查系统属性如果无法在指定的属性文件。 这是默认的。
- 覆盖 (2):检查系统属性之前,首先在指定的属性文件。 这允许系统属性重写任何其他财产的来源。

咨询的Javadoc 来完成 为更多的信息。

类名称替换

您可以使用 来完成 代替 类的名字,有时候是有用的,当你必须选择一个 特定的实现类在运行时。 例如:

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="locations">
     <value>classpath:com/foo/strategy.properties</value>
  </property>
  <property name="properties">
     <value>custom.strategy.class=com.foo.DefaultStrategy</value>
  </property>
</bean>

<bean id="serviceStrategy" class="${custom.strategy.class}"/>
```

如果类不能在运行时解析为一个有效的类, 分辨率的bean失败当它即将被创建,这是 在preInstantiateSingletons() 阶段 的 ApplicationContext 对于一个 非懒惰init bean。

示例: PropertyOverrideConfigurer

这个 PropertyOverrideConfigurer ,另一个bean 工厂后处理器,类似 来完成 ,但与 后者,原始的定义可以有默认值或没有 在所有bean属性的值。 如果一个覆盖 属性 文件没有的条目 某些bean属性,默认的上下文定义使用。

注意,该bean定义 不 意识到 被覆盖,所以它不会立即明显的从XML 定义文件,覆盖configurer正在被使用。 在案件的 多个PropertyOverrideConfigurer 实例 定义不同的值相同的bean属性,最后一个 赢了,由于覆盖机制。

属性文件配置线采取这种格式:

```
beanName.property=value
```

例如:

```
dataSource.driverClassName=com.mysql.jdbc.Driver
dataSource.url=jdbc:mysql:mydb
```

这个示例文件可以使用集装箱的定义 包含一个bean名为 数据源 ,这已经 司机 和 url 属性。

复合属性名也支持,只要每一个 组件的路径除了最后的财产被重载 已非空(可能是初始化的构造函数)。 在这个 例子.....

```
foo.fred.bob.sammy=123
```

... 这个 萨米 财产的 鲍勃 财产的 弗雷德 财产 的 foo bean设置为标量值 123年 。

> **注意**
>
> 指定覆盖值总是 文字 值;他们不是翻译成bean的引用。 本公约 也适用于原始的值在XML bean定义 指定一个bean引用。

与 上下文 名称空间中引入弹簧 2.5,可以配置属性覆盖与一个专用的 配置元素:

```
<context:property-override location="classpath:override.properties"/>
```

5.8.3A定制实例化逻辑与一个 FactoryBean

实现 org.springframework.beans.factory.FactoryBean 接口的对象的 本身 工厂 。

这个 FactoryBean 接口是一个点 可插入性到Spring IoC容器实例化的逻辑。 如果你 有复杂的初始化代码,更好的表达在Java作为吗 反对(可能)详细数量的XML,您可以创建自己的 FactoryBean 、编写复杂的 在这个类的初始化,然后塞你的自定义 FactoryBean 进入容器。

这个 FactoryBean 接口提供了 三个方法:

- 对象的getObject() :返回一个实例 这个工厂的对象创建。 可能的实例 共享,取决于这个工厂返回单例对象或 原型。
- 布尔isSingleton() :返回 真正的 如果这 FactoryBean 返回单例对象, 假 否则。
- 类getObjectType() :返回对象 类型返回的 getObject() 方法或 空 如果类型是无法提前知道。

这个 FactoryBean 概念和接口 是用在许多地方在Spring框架;超过50个吗 的实现 FactoryBean 接口与弹簧本身的船。

当你需要问一个容器对于一个实际的 FactoryBean 实例本身而不是bean 它产生,前言bean的id与&符号 (&)当调用 getBean() 方法 ApplicationContext 。 所以对于一个给定的 FactoryBean id的 myBean ,调用 getBean(“myBean”) 在集装箱返回的产品 FactoryBean ;然而,调用 getBean(“&myBean”) 返回 FactoryBean 实例 本身。

5.9基于注解的容器的配置

一个替代XML设置是由基于注解的 配置依赖于字节码元数据组件连接 而不是尖括号声明。 而不是使用XML来描述 豆布线,开发人员移动配置成组件类 通过使用注释的本身相关的类、方法或字段 宣言。 如前所述在 一个章节例子: RequiredAnnotationBeanPostProcessor 一个 ,使用一个 BeanPostProcessor 结合 注释是一个普遍的方法扩展Spring IoC容器。 对于 例,Spring 2.0引入强制要求的可能性 属性与 @ required 注释。 Spring 2.5能跟随 相同的一般方法来驱动Spring的依赖项注入。 本质上, @ autowired 注释 提供了同样的功能描述 SectionA 5 4 5,一个collaboratorsa自动装配 但是随着越来越多的细粒度控制和 更广泛的适用性。 Spring 2.5还增加了支持jsr - 250注释 如 @PostConstruct ,和 @PreDestroy 。 Spring 3.0添加支持 jsr - 330(依赖注入Java)中包含注释 javax。 注射包如 @ inject 和 @ named 。 这些注释的细节中可以找到 相关部门 。

> **是注释比XML配置弹簧吗?**
>
> 介绍了基于注解的配置 的问题:这种方法是更好的比XML。 简短的回答 是 这取决于 。 长一点的回答,每个方法 有其自身的优缺点,通常由开发人员决定吗 这策略适合她更好。 由于它们定义的方法, 注释提供了大量的上下文在他们的声明中,导致 更短更简洁的配置。 然而,XML擅长连接 组件不碰他们的源代码或重新编译它们。 一些 开发人员更喜欢具有布线接近源而其他人认为 带注释的类,不再是pojo,此外, 配置变得分散,难以控制。
>
> 无论选择,春天可以容纳两种风格,甚至混合 他们在一起。 值得指出的是,通过它 JavaConfig 选项,春天允许注释 用于非侵入性的方式,不接触目标组件 源代码和这方面的工具,所有配置风格 支持的 SpringSource工具套件 。

> **注意**
>
> 注释注入执行 之前 XML注入,因此后者配置 将覆盖前通过这两种方法的属性连接。

像往常一样,您可以将这些看作个人bean定义,但是 他们也可以隐式注册通过包括下面的标签在一个 基于xml的Spring配置(注意包容的 上下文 名称空间):

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
```

```
        http://www.springframework.org/schema/context/spring-context.xsd">

    <context:annotation-config/>

</beans>
```

(隐式注册后处理器包括 AutowiredAnnotationBeanPostProcessor , CommonAnnotationBeanPostProcessor , PersistenceAnnotationBeanPostProcessor ,如 以及上述 RequiredAnnotationBeanPostProcessor)。

注意

<上下文:注释配置/ > 只希望 注释在豆子在同一个应用程序上下文 定义。 这意味着,如果你把 <上下文:注释配置/ > 在一个 WebApplicationContext 对于一个 DispatcherServlet ,它只检查 @ autowired 豆子在你的控制器,和 不是你的服务。 看到 SectionA 17.2,一个了 DispatcherServlet 一个 更多 信息。

5.9.1A @ required

这个 @ required 注释适用于 bean属性的setter方法,像下面的例子:

```
public class SimpleMovieLister {

  private MovieFinder movieFinder;

  @Required
  public void setMovieFinder(MovieFinder movieFinder) {
     this.movieFinder = movieFinder;
  }

  // ...
}
```

这个注释仅仅表明,受影响的bean属性必须 被填充在配置时,通过一个明确的财产价值 一个bean定义或通过自动装配。 容器抛出一个异常 如果受影响的bean属性没有被填充的,这允许 热切的和明确的故障,避免 NullPointerException 年代以后或类似。 这是 还是建议你把断言到bean类本身,因为 例,变成一个init方法。 这样做需要强制执行这些引用 和价值观甚至当你使用类外的容器。

5.9.2A @ autowired

正如预期的那样,您可以应用 @ autowired 注释“传统” setter方法:

```
public class SimpleMovieLister {

  private MovieFinder movieFinder;

  @Autowired
  public void setMovieFinder(MovieFinder movieFinder) {
     this.movieFinder = movieFinder;
  }

  // ...
}
```

注意

JSR 330的@ inject注释可以用来代替弹簧的 @ autowired 注释在下面的例子。 看到 这里 为更多的细节

你也可以应用注释方法具有任意名称 和/或多个参数:

```
public class MovieRecommender {

  private MovieCatalog movieCatalog;

  private CustomerPreferenceDao customerPreferenceDao;

  @Autowired
  public void prepare(MovieCatalog movieCatalog,
                      CustomerPreferenceDao customerPreferenceDao) {
     this.movieCatalog = movieCatalog;
     this.customerPreferenceDao = customerPreferenceDao;
  }

  // ...
}
```

你可以申请 @ autowired 到 构造函数和字段:

```
public class MovieRecommender {

  @Autowired
  private MovieCatalog movieCatalog;

  private CustomerPreferenceDao customerPreferenceDao;

  @Autowired
  public MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {
      this.customerPreferenceDao = customerPreferenceDao;
  }

  // ...
}
```

也可以提供 所有 bean的一个 特定类型的 ApplicationContext 通过添加注释字段或方法,预计的数组 该类型:

```
public class MovieRecommender {

  @Autowired
  private MovieCatalog[] movieCatalogs;

  // ...
}
```

这同样适用于类型集合:

```
public class MovieRecommender {

  private Set<MovieCatalog> movieCatalogs;

  @Autowired
  public void setMovieCatalogs(Set<MovieCatalog> movieCatalogs) {
      this.movieCatalogs = movieCatalogs;
  }

  // ...
}
```

即使输入地图可以autowired只要预期的主要类型 字符串 。 映射的值将包含所有的咖啡豆 预期的类型和键将包含相应的 bean 名称:

```
public class MovieRecommender {

  private Map<String, MovieCatalog> movieCatalogs;

  @Autowired
  public void setMovieCatalogs(Map<String, MovieCatalog> movieCatalogs) {
      this.movieCatalogs = movieCatalogs;
  }

  // ...
}
```

默认情况下,自动装配失败每当 零 候选人bean是可用的,默认行为是把注释 方法、构造函数和字段指示 需要 依赖关系。 这种行为可以被改变 如下图所示。

```
public class SimpleMovieLister {

  private MovieFinder movieFinder;

  @Autowired(required=false)
  public void setMovieFinder(MovieFinder movieFinder) {
      this.movieFinder = movieFinder;
  }

  // ...
}
```

注意

只有 一个带注释的构造函数/类 可以 标记为 需要 ,但多个非必需 构造函数可以得到注释。 在这种情况下,每一个被认为是在 候选人和弹簧使用 贪婪的 构造函数的依赖可以满足,那是构造函数 有最大数量的参数。

@ autowired ' s 需要 属性是推荐的 @ required 注释。 这个 需要 属性表示该属性 不需要自动装配的目的,房地产是忽略了如果它 不能autowired的。 @ required ,在 另一方面,更强,它执行财产设定的 任何方式支持的容器。 如果没有值是注射,一个 相应的发生异常。

您还可以使用 @ autowired 对于 接口是众所周知的可分解的依赖关系: BeanFactory , ApplicationContext , 环境 , ResourceLoader , ApplicationEventPublisher ,和 MessageSource 。 这些接口和他们 扩展接口,如 ConfigurableApplicationContext 或 ResourcePatternResolver ,自动 解决,没有特殊设置必要的。

```
public class MovieRecommender {

  @Autowired
  private ApplicationContext context;

  public MovieRecommender() {
  }

  // ...
}
```

注意

@ autowired , @ inject , @ resource ,和 这个元素包含一个@ value 注释是由一个 春天 BeanPostProcessor 实现 这反过来意味着你 不能 应用这些注释在你自己的 BeanPostProcessor 或 BeanFactoryPostProcessor 类型(如果有的话)。 这些 类型必须是“有线”明确通过XML或使用一个春天 @ bean 法。

5.9.3A微调自动装配与限定词基于注解的

因为自动装配的类型可能导致多个候选人,这是 常常需要有更多的控制的选择过程。 一个方法 完成这是Spring qualifier 注释。 你可以将 限定符与特定的参数值,缩小组类型 匹配,这样一个特定的bean是为每个参数选择。 在 简单的情况下,这可能是一个纯描述性的值:

```
public class MovieRecommender {

  @Autowired
  @Qualifier("main")
  private MovieCatalog movieCatalog;

  // ...
}
```

这个 qualifier 注释也可以 构造函数参数指定个人或方法参数:

```
public class MovieRecommender {

  private MovieCatalog movieCatalog;

  private CustomerPreferenceDao customerPreferenceDao;

  @Autowired
  public void prepare(@Qualifier("main") MovieCatalog movieCatalog,
                      CustomerPreferenceDao customerPreferenceDao) {
      this.movieCatalog = movieCatalog;
      this.customerPreferenceDao = customerPreferenceDao;
  }

  // ...
}
```

相应的bean定义如下所示。 bean 限定符值“主要”是有线与构造函数参数,是 合格的用相同的值。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd">

  <context:annotation-config/>

  <bean class="example.SimpleMovieCatalog">
      <qualifier value="main"/>
      <!-- inject any dependencies required by this bean -->
  </bean>

  <bean class="example.SimpleMovieCatalog">
      <qualifier value="action"/>
      <!-- inject any dependencies required by this bean -->
```

```
  </bean>

  <bean id="movieRecommender" class="example.MovieRecommender"/>

</beans>
```

对于一个后备匹配,该bean的名字被认为是一个默认的限定符 值。 因此你可以定义bean id“主要”代替 嵌套的限定符元素,导致相同的匹配结果。 然而, 虽然您可以使用本公约来引用特定bean的名字, @ autowired 根本上是关于 与可选的语义限定符受类型驱动注入。 这意味着 限定符的值,即使bean名称撤退,总有缩小 语义在组类型匹配;他们不语义表达 引用一个独特的bean id。良好的限定符值是“主要”或 “中东”或“持久”,表达一个特定的组件的特点 这是独立于bean id,这可能是自动生成的情况 一个匿名的bean定义类似于前面的 的例子。

限定符也适用于类型集合,正如上面所讨论的,对 个例子, 集< MovieCatalog > 。 在这种情况下,所有 匹配的豆子根据宣布限定符作为注射 收集。 这意味着限定符不必是独特的;他们 而仅仅构成过滤标准。 例如,您可以定义 多个 MovieCatalog bean与相同的限定符 值“行动”,所有这些将被注入到一个 集< MovieCatalog > 标注 @ qualifier(“行动”)。

> **提示**
>
> 如果你打算表达注解驱动喷射的名字,不要 主要使用 @ autowired ,即使是 技术能力指一个bean名称通过qualifier 值。 相反,使用 jsr - 250 @ resource 注释,这是 语义定义,确定具体的目标组件的 独特的名字,声明的类型被无关的匹配 过程。
>
> 作为一个特定结果的语义差异,豆类, 是自己定义为一个集合或地图类型不能被注入 通过 @ autowired ,因为类型匹配 没有被正确适用于他们。 使用 @ resource 对于这样的豆子,指 特定的集合或地图由独特的名字,豆
>
> @ autowired 适用于油田, 构造函数,进行方法,允许通过缩小 限定符注释在参数水平。 相比之下, @ resource 只支持字段 和bean属性setter方法的一个参数。 作为一个 因此,坚持限定符如果你注入目标是构造函数或方法进行。

您可以创建自己的自定义限定符注释。 简单地定义一个 注释和提供 qualifier 注释在您的定义:

```
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Genre {

  String value();
}
```

然后你可以提供自定义修饰符在autowired的字段和 参数:

```
public class MovieRecommender {

  @Autowired
  @Genre("Action")
  private MovieCatalog actionCatalog;

  private MovieCatalog comedyCatalog;

  @Autowired
  public void setComedyCatalog(@Genre("Comedy") MovieCatalog comedyCatalog) {
     this.comedyCatalog = comedyCatalog;
  }

  // ...
}
```

接下来,为候选人提供信息的bean定义。 你 可以添加 <限定符/ > 标签的子元素 < bean / > 标签,然后指定 类型 和 价值 以匹配您的自定义 限定符注释。 是与类型的完全限定 类名的注释。 或者,作为一个方便的如果没有风险的 存在冲突的名称,您可以使用短的类名。 两 方法是通过以下例子。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans.xsd
     http://www.springframework.org/schema/context
     http://www.springframework.org/schema/context/spring-context.xsd">

  <context:annotation-config/>

  <bean class="example.SimpleMovieCatalog">
```

```
    <qualifier type="Genre" value="Action"/>
    <!-- inject any dependencies required by this bean -->
  </bean>

  <bean class="example.SimpleMovieCatalog">
    <qualifier type="example.Genre" value="Comedy"/>
    <!-- inject any dependencies required by this bean -->
  </bean>

  <bean id="movieRecommender" class="example.MovieRecommender"/>

</beans>
```

在 SectionA 5.10,一个类路径扫描和管理componentsa ,你将看到一个 基于注解的替代提供限定符元数据在XML。 具体地说,看到 SectionA 5 10 7,一个提供元数据与annotationsa限定词 。

在某些情况下,它可能不足以使用注释没有 值。 这可能是有用的注释是一个更通用的 目的和可以应用在几个不同类型的依赖关系。 例如,您可以提供一个 离线 目录, 将搜索网络连接可用时没有。 首先定义 简单的注释:

```
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Offline {

}
```

然后添加注释的字段或属性是 autowired的:

```
public class MovieRecommender {

  @Autowired
  @Offline
  private MovieCatalog offlineCatalog;

  // ...
}
```

现在只需要一个预选赛的bean定义 类型 :

```
<bean class="example.SimpleMovieCatalog">
  <qualifier type="Offline"/>
  <!-- inject any dependencies required by this bean -->
</bean>
```

您还可以定义定制的限定符注释命名接受 属性除了或而不是简单的 价值 属性。 如果多个属性值然后 指定一个字段或参数 autowired的,一个bean定义必须 匹配 所有 这样的属性值被认为是一个 自动装配的候选人。 作为一个例子,考虑下面的注释定义:

```
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface MovieQualifier {

  String genre();

  Format format();
}
```

在这种情况下 格式 是一个枚举:

```
public enum Format {

  VHS, DVD, BLURAY
}
```

这个字段是加注autowired的定义修饰符和 包括两个属性的值: 流派 和 格式 。

```
public class MovieRecommender {

  @Autowired
  @MovieQualifier(format=Format.VHS, genre="Action")
  private MovieCatalog actionVhsCatalog;

  @Autowired
  @MovieQualifier(format=Format.VHS, genre="Comedy")
  private MovieCatalog comedyVhsCatalog;

  @Autowired
```

```
  @MovieQualifier(format=Format.DVD, genre="Action")
  private MovieCatalog actionDvdCatalog;

  @Autowired
  @MovieQualifier(format=Format.BLURAY, genre="Comedy")
  private MovieCatalog comedyBluRayCatalog;

  // ...
}
```

最后,该bean定义应该包含匹配的限定符 值。 这个例子还说明bean 元 属性可能用于代替 <限定符/ > 子元素。 如果可用, < 限定符/ > 和它的属性优先, 但是自动装配机制回落在值中提供 < meta / > 标记如果没有这样的限定符是礼物, 在过去的两个bean定义在下面的例子。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:context="http://www.springframework.org/schema/context"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">

  <context:annotation-config/>

  <bean class="example.SimpleMovieCatalog">
    <qualifier type="MovieQualifier">
      <attribute key="format" value="VHS"/>
      <attribute key="genre" value="Action"/>
    </qualifier>
    <!-- inject any dependencies required by this bean -->
  </bean>

  <bean class="example.SimpleMovieCatalog">
    <qualifier type="MovieQualifier">
      <attribute key="format" value="VHS"/>
      <attribute key="genre" value="Comedy"/>
    </qualifier>
    <!-- inject any dependencies required by this bean -->
  </bean>

  <bean class="example.SimpleMovieCatalog">
    <meta key="format" value="DVD"/>
    <meta key="genre" value="Action"/>
    <!-- inject any dependencies required by this bean -->
  </bean>

  <bean class="example.SimpleMovieCatalog">
    <meta key="format" value="BLURAY"/>
    <meta key="genre" value="Comedy"/>
    <!-- inject any dependencies required by this bean -->
  </bean>

</beans>
```

5.9.4A CustomAutowireConfigurer

这个 CustomAutowireConfigurer 是 BeanFactoryPostProcessor 这使您 注册自己的自定义限定符注释类型,即使它们 没有标注春天的 qualifier 注释。

```
<bean id="customAutowireConfigurer"
   class="org.springframework.beans.factory.annotation.CustomAutowireConfigurer">
 <property name="customQualifierTypes">
    <set>
      <value>example.CustomQualifier</value>
    </set>
 </property>
</bean>
```

特定的实现 AutowireCandidateResolver 这是激活 对于应用程序上下文依赖于Java版本。 在版本 早于Java 5注释的限定符, 不支持 因此自动装配候选人仅根据 自动装配的候选人 每个bean定义的价值一样 通过任何 默认自动装配候选人 模式(s) 上可用 < bean / > 元素。 在Java 5或 后来的存在 qualifier 任何自定义注释的注释和注册 CustomAutowireConfigurer 也将扮演一个 的角色。

不管Java版本,当多个bean符合 自动装配的候选人,确定一个“主”候选人的 相同的:如果整整一个bean定义的候选人中有一个主要 属性设置为 真正的 ,它 将被选择。

5.9.5A @ resource

春天也支持注射使用jsr - 250 @ resource 注释字段或bean 属性的setter方法。 这是一个常见的模式在Java EE 5和6, 在JSF托管bean示例1.2或jax - ws 2.0的端点。 Spring支持 这种模式也为spring管理对象。

@ resource 取一个名称属性,和 默认情况下春天,的值作为解释bean名称注射。 换句话说,它遵循 则 语义, 证明在这个例子:

```
public class SimpleMovieLister {

  private MovieFinder movieFinder;

  @Resource(name="myMovieFinder")
  public void setMovieFinder(MovieFinder movieFinder) {
     this.movieFinder = movieFinder;
  }
}
```

如果没有名字是显式地指定,默认的名称源自 字段名或setter方法。 在案件的一个领域,它需要的字段 名字,以防setter方法,它需要bean属性名。 所以 下面的例子将bean名称” movieFinder” 注入它的setter方法:

```
public class SimpleMovieLister {

  private MovieFinder movieFinder;

  @Resource
  public void setMovieFinder(MovieFinder movieFinder) {
     this.movieFinder = movieFinder;
  }
}
```

注意

这个名字提供注释被解析为bean名称 这个 ApplicationContext 的 CommonAnnotationBeanPostProcessor 是意识到。 这个 通过JNDI名称可以解决如果你配置Spring的 SimpleJndiBeanFactory 明确。 然而,建议你依赖默认行为和 简单地使用Spring的JNDI查找功能来保护级别的 间接寻址。

专属的案例 @ resource 没有显式指定使用名字,类似 @ autowired , @ resource 发现一个主要类型匹配 而不是一个特定的命名bean,并解决了著名的可分解的 依赖性: BeanFactory , ApplicationContext, ResourceLoader, ApplicationEventPublisher ,和 MessageSource 接口。

因此在接下来的例子中, customerPreferenceDao 场第一个寻找一个bean customerPreferenceDao命名,然后回落到一个主类型匹配 类型 customerPreferenceDao 。 “上下文”字段 是基于已知的可分解的注入依赖类型 ApplicationContext 。

```
public class MovieRecommender {

  @Resource
  private CustomerPreferenceDao customerPreferenceDao;

  @Resource
  private ApplicationContext context;

  public MovieRecommender() {
  }

  // ...
}
```

5.9.6A @PostConstruct 和 @PreDestroy

这个 CommonAnnotationBeanPostProcessor 不仅 认识到 @ resource 注解,但 还jsr - 250 生命周期 注释。 引入 Spring 2.5,支持这些注释提供了另一个 替代中描述 初始化 回调 和 破坏 回调 。 只要 CommonAnnotationBeanPostProcessor 注册 在春天 ApplicationContext ,一个 法携带这些注释是在同一点的调用 生命周期作为相应的春天生命周期接口方法或 显式声明回调方法。 在下面的示例中,缓存会 在初始化和清除来预装在毁灭。

```
public class CachingMovieLister {

  @PostConstruct
  public void populateMovieCache() {
     // populates the movie cache upon initialization...
  }

  @PreDestroy
  public void clearMovieCache() {
     // clears the movie cache upon destruction...
  }
```

```
}
```

注意

对于细节的影响结合各种生命周期 机制,看到 一个章节结合生命周期mechanismsa 。

5.10类路径扫描和管理组件

大多数的例子在这一章使用XML指定的配置 元数据产生每个 BeanDefinition 在Spring容器。 前一节 (SectionA 5.9,一个configurationa基于注解的容器)演示了如何提供 很多配置元数据通过源代码级别的注释。 甚至 在那些例子,然而,“基地” bean定义明确 在XML文件中,而注释只驱动依赖 注射。 本节描述一个选项为隐式检测 候选组件 通过扫描类路径。 候选人组件类,与一个过滤条件和 有一个相应的bean定义注册容器。 这 消除了需要使用XML来执行bean注册,相反你可以 使用注释(例如component),AspectJ表达式,或者你的类型 自定义过滤条件来选择哪个类将bean 定义注册容器。

注意

从Spring 3.0,提供的许多特性 Spring JavaConfig 项目 是Spring框架核心的一部分。 这允许您 定义bean 使用Java,而不是使用传统的XML文件。 把 一看 @ configuration , @ bean , @ import ,和 @DependsOn 注释的例子 使用这些新特性。

5.10.1A component 和进一步的刻板印象 注释

在Spring 2.0和以后, @Repository 注释标记任何 类,它实现了角色或 原型 (也 被称为数据访问对象或刀)的存储库。 在使用的这个标志是自动翻译中描述的异常 SectionA 15 2 2,一个translationa除外 。

Spring 2.5引入了进一步构造型注解: component , @ service ,和 controller 。 component 是一个通用的原型吗 spring管理组件。 @Repository , @ service ,和 controller 是专门化的 component 为更具体的用例,因为 例,在持久性、服务和表示层,分别。 因此,你可以标注你的组件类 component ,但他们通过标注 @Repository , @ service ,或 controller 相反,你的类是更多 正确地适合加工的工具或联想方面。 对于 的例子,这些构造型注解使理想目标为切入点。 它 也有可能的是, @Repository , @ service ,和 controller 可能携带额外的语义 在未来版本的Spring框架。 因此,如果你选择 之间使用 component 或 @ service 为你的服务层, @ service 显然是更好的选择。 同样,如上所述, @Repository 是 已经支持作为一个标记为自动异常在你。翻译 持久层。

5.10.2A自动检测类和注册的bean 定义

春天可以自动检测的原型类和寄存器 相应 BeanDefinition 年代的 ApplicationContext 。 例如, 以下两个类都有资格获得这种自动:

```
@Service
public class SimpleMovieLister {

  private MovieFinder movieFinder;

  @Autowired
  public SimpleMovieLister(MovieFinder movieFinder) {
      this.movieFinder = movieFinder;
  }
}
```

```
@Repository
public class JpaMovieFinder implements MovieFinder {
  // implementation elided for clarity
}
```

自动侦测到这些类和寄存器对应的豆子,你 需要包括以下元素在XML的基本方案 元素是一种常见的父母包了两个类。 (另外, 你可以指定一个以逗号分隔的列表,包括父包的 每个类)。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

  <context:component-scan base-package="org.example"/>
```

```
</beans>
```

提示

使用 <上下文:组件扫描> 隐式 使功能的 <上下文:注释配置> 。 通常没有必要包括 <上下文:注释配置> 元素当使用 <上下文:组件扫描> 。

注意

扫描的classpath包需要存在 相应的目录条目在类路径中。 当您构建jar 与蚂蚁,确保你做的 不 激活 文件只有开关JAR任务。

此外, AutowiredAnnotationBeanPostProcessor 和 CommonAnnotationBeanPostProcessor 都 包括隐式在使用组件扫描元件。 这意味着 这两个组件 和 有线 在一起,没有任何bean配置元数据中提供的 XML。

注意

你可以禁用注册 AutowiredAnnotationBeanPostProcessor 和 CommonAnnotationBeanPostProcessor 通过 包括 注释配置 属性与一个 值为false。

5.10.3A使用过滤器来定制扫描

默认情况下,类标注 component , @Repository , @ service , controller ,或一个自定义注释, 本身是标注 component 是 只有发现候选人组件。 但是,您可以修改和扩展 这种行为只是通过应用自定义过滤器。 在 包括过滤器 或 排他过滤器 子元素的组件扫描 元素。 每个过滤器 元素要求 类型 和 表达式 属性。 下表描述了 过滤选项。

5.5为多。 一个过滤器类型

滤波器类型	示例表达式	描述
注释	org.example.SomeAnnotation	一个注释出席类型在目标水平 组件。
可转让的	org.example.SomeClass	一个类(或接口),目标组件 可转让的(扩展/执行)。
AspectJ	org的例子. . *服务+	AspectJ表达式类型相匹配的目标 组件。
Regex	org \例子\违约。*	一个正则表达式表达与目标组件 类名。
定制	org.example.MyTypeFilter	一个自定义实现的 org.springframework.core.type .TypeFilter 接口。

下面的例子显示了XML配置忽略所有 @Repository 注释和使用“存根” 存储库相反。

```
<beans>

  <context:component-scan base-package="org.example">
    <context:include-filter type="regex" expression=".*Stub.*Repository"/>
    <context:exclude-filter type="annotation"
                            expression="org.springframework.stereotype.Repository"/>
  </context:component-scan>

</beans>
```

注意

你也可以禁用默认的过滤器通过提供 使用默认的过滤器= " false " 的属性 <组件扫描/ >元素。 这将禁用自动生效 检测类标注 component , @Repository , @ service ,或 controller 。

5.10.4A定义bean组件内的元数据

弹簧组件也可以贡献的bean定义元数据 集装箱。 你这么做同样的 @ bean 注释 用于定义bean内的元数据 @ configuration

带注释的类。 这是一个简单的例子:

```
@Component
public class FactoryMethodComponent {

  @Bean @Qualifier("public")
  public TestBean publicInstance() {
    return new TestBean("publicInstance");
  }

  public void doWork() {
    // Component method implementation omitted
  }
}
```

这个类是一个弹簧组件包含特定于应用程序的代码 包含在其 doWork() 法。 然而,它 也贡献一个bean定义,有一个工厂方法指该方法 publicInstance() 。 这个 @ bean 注释标识工厂方法和 其他bean定义属性,比如限定符值通过 qualifier 注释。 其他方法级 可以指定注释是 scope , @Lazy ,定义修饰符注释。 Autowired的 支持字段和方法如之前讨论的,额外的 支持自动装配的 @ bean 方法:

```
@Component
public class FactoryMethodComponent {

  private static int i;

  @Bean @Qualifier("public")
  public TestBean publicInstance() {
    return new TestBean("publicInstance");
  }

  // use of a custom qualifier and autowiring of method parameters

  @Bean
  protected TestBean protectedInstance(@Qualifier("public") TestBean spouse,
                                       @Value("#{privateInstance.age}") String country) {
    TestBean tb = new TestBean("protectedInstance", 1);
    tb.setSpouse(tb);
    tb.setCountry(country);
    return tb;
  }

  @Bean @Scope(BeanDefinition.SCOPE_SINGLETON)
  private TestBean privateInstance() {
    return new TestBean("privateInstance", i++);
  }

  @Bean @Scope(value = WebApplicationContext.SCOPE_SESSION,
           proxyMode = ScopedProxyMode.TARGET_CLASS)
  public TestBean requestScopedInstance() {
    return new TestBean("requestScopedInstance", 3);
  }
}
```

这个例子的autowires 字符串 方法 参数 国家 的价值 年龄 另一个bean属性命名 privateInstance 。 一个春天的表达式语言元素 定义了该属性的值通过符号 # { <表达式> } 。 对于 这个元素包含一个@ value 注释, 一个表达式解析器是预先配置的寻找bean名称当 解析表达式的文本。

这个 @ bean 方法在一个弹簧组件 经过不同的处理比同行在春天 @ configuration 类。 不同的是, component 类是不增强,CGLIB 拦截的调用方法和字段。 CGLIB代理是 方法调用方法或字段中 @ configuration 类 @ bean 方法 创建bean引用协作对象的元数据。 方法 不 与普通的Java调用语义。 相比之下, 调用一个方法或字段在一个 component 类 @ bean 方法已经 标准Java 语义。

5.10.5A命名个组件

当一个组件是作为扫描过程,它的 bean的名字是生成的 BeanNameGenerator 策略已知, 扫描仪。 默认情况下,任何弹簧构造型注解 (component , @Repository , @ service ,和 controller),其中包含一个 名称 值从而提供名字 相应的bean定义。

如果这样的注释不包含 名称 值或 检测到任何其他组件(如自定义过滤器所发现的), 默认的bean名称生成器返回小写形式的确立 类名。 例如,如果以下两个组件检测到, 名字会myMovieLister和movieFinderImpl:

```
@Service("myMovieLister")
public class SimpleMovieLister {
  // ...
}
```

```
@Repository
```

```
public class MovieFinderImpl implements MovieFinder {
  // ...
}
```

注意

如果你不想依赖默认bean命名策略,你 可以提供一个自定义的bean命名策略。 首先,实现 BeanNameGenerator 接口, 一定要包括一个默认的无参数构造函数。 然后,提供 完全限定类名当配置扫描:

```
<beans>

  <context:component-scan base-package="org.example"
                  name-generator="org.example.MyNameGenerator" />

</beans>
```

作为一般规则,考虑指定名字和注释 每当其他组件可以使显式地引用它。 在 另一方面,自动生成的名字是足够每当容器 负责布线。

5.10.6A提供一个组件范围

与spring管理组件在一般情况下,默认的和最 常见的范围是singleton。个组件 然而,有时 你需要其他范围,用一个新的Spring 2.5提供 scope 注释。 仅仅提供名称 范围内的注释:

```
@Scope("prototype")
@Repository
public class MovieFinderImpl implements MovieFinder {
  // ...
}
```

注意

提供一个自定义的策略范围解析而不是 依靠基于注解的方法,实现了 ScopeMetadataResolver 界面, 而且一定要包括一个默认的无参数构造函数。 然后,提供 完全限定类名当配置扫描:

```
<beans>

  <context:component-scan base-package="org.example"
                  scope-resolver="org.example.MyScopeResolver" />

</beans>
```

当使用某些单体范围,可能需要 生成代理的范围对象。 描述的推理 一个章节bean作为dependenciesa范围 。 为了这个目的,一个 范围代理 属性是可用的 组件扫描的元素。 三个可能的值是:不,接口和 targetClass。 例如,下面的配置会导致 标准JDK动态代理:

```
<beans>

  <context:component-scan base-package="org.example"
                  scoped-proxy="interfaces" />

</beans>
```

提供元数据与5.10.7A预选赛注释

这个 qualifier 注释是讨论 在 SectionA 5 9 3,一个微调与qualifiersa基于注解的自动装配 。 这个例子 在这部分演示使用 qualifier 注释和定义修饰符 注释提供细粒度控制当你解决自动装配 候选人。 因为这些例子是基于XML bean定义, 限定符元数据提供候选人bean定义使用 这个 预选赛 或 元 子元素 的 bean 元素在XML。 当依赖 类路径扫描自动组件,你提供 限定符元数据与类型水平上加注解的候选类。 这个 以下三个例子证明这种技术:

```
@Component
@Qualifier("Action")
public class ActionMovieCatalog implements MovieCatalog {
  // ...
}
```

```
@Component
@Genre("Action")
public class ActionMovieCatalog implements MovieCatalog {
```

```
  // ...
}
```

```
@Component
@Offline
public class CachingMovieCatalog implements MovieCatalog {
  // ...
}
```

注意

与大多数基于注解的替代品,请记住 注释的元数据绑定类定义本身,而 使用XML允许多个bean 相同的 类型提供元数据的变化他们的限定符, 因为这是每个元数据提供而不是 每个类。

5.11一个使用JSR 330标准注释

从Spring 3.0,弹簧提供了支持jsr - 330标准注释(依赖注入)。 这些注释是扫描一样弹簧注释。 你只需要有相关的罐子在您的类路径下。

注意

如果您正在使用Maven javax.inject 工件可用 在标准Maven存储库 (http://repo1.maven.org/maven2/javax/inject/javax.inject/1/)。 你可以添加以下依赖你的文件pom . xml中:

```
<dependency>
  <groupId>javax.inject</groupId>
  <artifactId>javax.inject</artifactId>
  <version>1</version>
</dependency>
```

5.11.1A依赖注入与 @ inject 和 @ named

而不是 @ autowired , @javax.inject.Inject 可以用如下:

```
import javax.inject.Inject;

public class SimpleMovieLister {

  private MovieFinder movieFinder;

  @Inject
  public void setMovieFinder(MovieFinder movieFinder) {
    this.movieFinder = movieFinder;
  }
  // ...
}
```

与 @ autowired ,可以使用 @ inject 在类级别,字段级,方法级和构造函数参数的水平。 如果你想使用限定名称,应该注入的依赖, 你应该使用 @ named 注释如下:

```
import javax.inject.Inject;
import javax.inject.Named;

public class SimpleMovieLister {

  private MovieFinder movieFinder;

  @Inject
  public void setMovieFinder(@Named("main") MovieFinder movieFinder) {
    this.movieFinder = movieFinder;
  }
  // ...
}
```

5.11.2A @ named :一个标准相当于 component 注释

而不是 component , @javax.inject.Named 可以用如下:

```
import javax.inject.Inject;
import javax.inject.Named;

@Named("movieListener")
public class SimpleMovieLister {
```

```
  private MovieFinder movieFinder;

  @Inject
  public void setMovieFinder(MovieFinder movieFinder) {
      this.movieFinder = movieFinder;
  }
  // ...
}
```

这是非常常见的使用 component 没有 指定组件的名称。 @ named 可以用在类似的方式:

```
import javax.inject.Inject;
import javax.inject.Named;

@Named
public class SimpleMovieLister {

  private MovieFinder movieFinder;

  @Inject
  public void setMovieFinder(MovieFinder movieFinder) {
      this.movieFinder = movieFinder;
  }
  // ...
}
```

当使用 @ named ,可以使用 组件扫描在完全相同的方式在使用Spring注释:

```
<beans>
   <context:component-scan base-package="org.example"/>
</beans>
```

5.11.3A限制的标准方法

当使用标准的注释,重要的是要知道 一些重要的特性是不提供如下表所示:

5.6为多。 一个春天的注释与标准注释

春天	javax.inject。*	javax。 注入限制/评论
@ autowired	@ inject	@ inject没有“要求”属性
component	@ named	一个
scope(“单”)	@ singleton	jsr - 330默认范围是喜欢春天的 原型 。 然而,为了保持符合春天的一般违约, 一个 jsr - 330 bean声明在Spring容器是一个 singleton 默认情况下。 为了使用一个 范围以外的其他 singleton ,你应该使用Spring的 scope 注释。 javax.inject 还提供了一个 scope 注释。 然而,这是只用于创建您自己的注释。
qualifier	@ named	一个
这个元素包含一个@ value	一个	没有等效
@ required	一个	没有等效
@Lazy	一个	没有等效

5.12一个基于java的容器的配置

5.12.1A基本概念: @ bean 和 @ configuration

在春天的中央工件的新java配置支持 @ configuration 注释的类和 @ bean 带注释的方法。

这个 @ bean 注释被用来显示 方法实例化、配置和初始化一个新对象的管理 Spring IoC容器。 对于那些熟悉Spring的 < bean / > XML配置的

全@ configuration vs“简装版”@Beans模式?

当 @ bean 方法内部声明 类 不 标注 @

@ bean 注释扮演同样的角色 < bean / > 元素。 您可以使用 @ bean 带注释的方法 任何春天 component ,然而,他们是大多数 通常用于 @ configuration 豆子。

注释一个类与 @ configuration 表明,其主要用途是作为一个来源的bean定义。 此外, @ configuration 国米bean类允许 依赖关系被定义为简单的调用其他 @ bean 方法在同一个班。 尽可能简单的 @ configuration 类将如下:

configuration 他们被称为被 处理在一个 “lite” 模式。 例如,bean方法中声明 component 或者甚至在一个 平原旧 类 将被视为 “简装版” 。

与全面 @ configuration ,清淡的 @ bean 方法不能轻易宣布国米bean 依赖关系。 通常一 @ bean 方法不应该 调用另一个 @ bean 方法当操作在 “建兴” 模式。

只有使用 @ bean 方法在 @ configuration 类是推荐的方法 确保 ‘满载’ 模式总是使用。 这将防止相同的 @ bean 不小心被调用的方法 多次和有助于减少微妙的错误,很难追踪 当操作在 “精简” 模式。

```
@Configuration
public class AppConfig {
  @Bean
  public MyService myService() {
    return new MyServiceImpl();
  }
}
```

这个 AppConfig 级以上就相当于这个 第二年春天 < bean / > XML:

```
<beans>
  <bean id="myService" class="com.acme.services.MyServiceImpl"/>
</beans>
```

这个 @ bean 和 @ configuration 注释将深入讨论下面的小节。 不过,首先,我们会 覆盖的各种方法创建一个spring容器使用基于java的 配置。

5.12.2A Spring容器实例化使用 所

下面的部分文档春天的 所 ,新的Spring 3.0中。 这种多用途的 ApplicationContext 实现 不仅能够接受 @ configuration 类输入,但也平原 component 类和类 jsr - 330元数据注释。

当 @ configuration 类是提供作为输入, 这个 @ configuration 类本身注册为bean 定义,和所有的声明 @ bean 方法在 类也登记为bean定义。

当 component 和jsr - 330提供的类, 他们被注册为bean定义和假设DI 元数据如 @ autowired 或 @ inject 用在这些类在哪里 必要的。

结构简单

在同样的方式,使用Spring XML文件作为输入当 实例化一个 ClassPathXmlApplicationContext , @ configuration 课程可能被用作输入当 实例化一个 所 。 这允许完全使用xml免费Spring容器:

```
public static void main(String[] args) {
  ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);
  MyService myService = ctx.getBean(MyService.class);
  myService.doStuff();
}
```

正如上面提到的, 所 并不局限于 工作只有 @ configuration 类。 任何 component 或jsr - 330带注释的类可以提供 作为输入到构造函数。 例如:

```
public static void main(String[] args) {
  ApplicationContext ctx = new AnnotationConfigApplicationContext(MyServiceImpl.class, Dependency1.class, Dependency2.class);
  MyService myService = ctx.getBean(MyService.class);
  myService.doStuff();
}
```

上述假设 MyServiceImpl , Dependency1 和 Dependency2 使用 Spring依赖项注入等注释 @ autowired 。

建筑容器以编程方式使用 寄存器(类< ? >...)

一个 所 可能是 使用无参数构造函数实例化,然后配置使用 寄存器() 法。 这种方法尤其 当通过编程建立一个有用 所 。

```
public static void main(String[] args) {
  AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
  ctx.register(AppConfig.class, OtherConfig.class);
  ctx.register(AdditionalConfig.class);
  ctx.refresh();
  MyService myService = ctx.getBean(MyService.class);
  myService.doStuff();
}
```

使组件扫描与 扫描(String……)

经验丰富的弹簧用户将熟悉以下 常用的XML声明从春天的 背景: 名称空间

```
<beans>
  <context:component-scan base-package="com.acme"/>
</beans>
```

在上面的例子中, com acme 包将 扫描,寻找任何 component 注释 类,这些类将被登记为Spring bean定义 在容器内。 所 暴露了 扫描(String……) 方法以允许相同的 组件扫描 功能:

```
public static void main(String[] args) {
  AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
  ctx.scan("com.acme");
  ctx.refresh();
  MyService myService = ctx.getBean(MyService.class);
}
```

注意

记住, @ configuration 类 元注释与 component ,所以他们 组件扫描的候选人! 在上面的示例中,假设 AppConfig 是宣布在 com acme 包(或任何包下面,它 会捡起在叫吗 扫描() ,和 在 refresh() 所有的 @ bean 方法将处理和登记为bean定义中 容器。

支持web应用程序时 AnnotationConfigWebApplicationContext

一个 WebApplicationContext 的变体 所 可与 AnnotationConfigWebApplicationContext 。 这 实现可用于当配置弹簧 ContextLoaderListener servlet侦听器,Spring MVC DispatcherServlet 等等。下面是一个 web . xml 片段,配置一个典型的 Spring MVC web应用程序。 注意使用 contextClass 上下文参数和初始化:

```
<web-app>
  <!-- Configure ContextLoaderListener to use AnnotationConfigWebApplicationContext
       instead of the default XmlWebApplicationContext -->
  <context-param>
     <param-name>contextClass</param-name>
     <param-value>
        org.springframework.web.context.support.AnnotationConfigWebApplicationContext
     </param-value>
  </context-param>

  <!-- Configuration locations must consist of one or more comma- or space-delimited
       fully-qualified @Configuration classes. Fully-qualified packages may also be
       specified for component-scanning -->
  <context-param>
     <param-name>contextConfigLocation</param-name>
     <param-value>com.acme.AppConfig</param-value>
  </context-param>

  <!-- Bootstrap the root application context as usual using ContextLoaderListener -->
  <listener>
     <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>

  <!-- Declare a Spring MVC DispatcherServlet as usual -->
  <servlet>
     <servlet-name>dispatcher</servlet-name>
     <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
     <!-- Configure DispatcherServlet to use AnnotationConfigWebApplicationContext
          instead of the default XmlWebApplicationContext -->
     <init-param>
        <param-name>contextClass</param-name>
        <param-value>
           org.springframework.web.context.support.AnnotationConfigWebApplicationContext
        </param-value>
     </init-param>
     <!-- Again, config locations must consist of one or more comma- or space-delimited
          and fully-qualified @Configuration classes -->
```

```
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>com.acme.web.MvcConfig</param-value>
    </init-param>
  </servlet>

  <!-- map all requests for /app/* to the dispatcher servlet -->
  <servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/app/*</url-pattern>
  </servlet-mapping>
</web-app>
```

5.12.3A使用 @ bean 注释

@ bean 是方法级注释和 直接模拟的XML < bean / > 元素。 这个 注释支持一些属性提供 < bean / > ,如: init方法 , 销毁方法 , 自动装配 和 名称 。

您可以使用 @ bean 注释在 @ configuration 注释或在一个 component 带注释的类。

声明一个bean

声明一个bean,只是标注的方法 @ bean 注释。 你使用这种方法 注册一个bean定义在一个 ApplicationContext 的 类型指定为方法的返回值。 默认情况下,该bean 的名字是相同的方法名称。 以下是一个简单的 的例子 @ bean 方法声明:

```
@Configuration
public class AppConfig {

  @Bean
  public TransferService transferService() {
    return new TransferServiceImpl();
  }

}
```

前面的配置就是相当于以下 Spring XML:

```
<beans>
  <bean id="transferService" class="com.acme.TransferServiceImpl"/>
</beans>
```

既使一个bean声明命名 transferService 可用的 ApplicationContext ,绑定到一个对象 类型的实例 TransferServiceImpl :

```
transferService -> com.acme.TransferServiceImpl
```

接收生命周期回调

任何类定义 @ bean 注释支持 常规的生命周期回调函数,可以使用 @PostConstruct 和 @PreDestroy jsr - 250的注解,请参阅 jsr - 250 注释 为进一步的细节。

常规的弹簧 生命周期 回调也完全支持。 如果一个bean 实现 InitializingBean , DisposableBean , 或 生命周期 ,各自的方法叫的 集装箱。

标准组 *意识到 接口如 BeanFactoryAware , BeanNameAware , MessageSourceAware , ApplicationContextAware ,和所以在也完全支持。

这个 @ bean 注释支持 指定任意的初始化和销毁的回调方法, 就像春天的XML的 init方法 和 销毁方法 属性 bean 元素:

```
public class Foo {
  public void init() {
    // initialization logic
  }
}

public class Bar {
  public void cleanup() {
    // destruction logic
  }
}

@Configuration
public class AppConfig {
  @Bean(initMethod = "init")
  public Foo foo() {
```

```
    return new Foo();
  }
  @Bean(destroyMethod = "cleanup")
  public Bar bar() {
    return new Bar();
  }
}
```

当然,对于 foo 以上,这将是 同样有效的调用 init() 方法直接在 施工:

```
@Configuration
public class AppConfig {
  @Bean
  public Foo foo() {
    Foo foo = new Foo();
    foo.init();
    return foo;
  }

  // ...
}
```

提示

当你工作直接在Java中,你可以做任何你喜欢的和 你的对象,并不总是需要依靠容器 生命周期!

bean指定范围

使用 scope 注释

你可以指定你的bean定义 @ bean 注释应该有一个特定的 范围。 你可以使用任何标准的范围中指定的 Bean范围 部分。

默认范围是 singleton ,但你可以 覆盖这个与 scope 注释:

```
@Configuration
public class MyConfiguration {
  @Bean
  @Scope("prototype")
  public Encryptor encryptor() {
    // ...
  }
}
```

scope和作用域的代理

Spring提供了方便的工作方式与作用域依赖关系 通过 范围 代理 。 最简单的方法来创建这样一个代理在使用 XML配置是 < aop:作用域的代理/ > 元素。 配置您的Java bean与一个scope注释 提供了同等的支持与proxyMode属性。 默认的是 没有代理(ScopedProxyMode.NO),但您可以指定 ScopedProxyMode.TARGET_CLASS 或 ScopedProxyMode.INTERFACES 。

如果你港口的作用域内的代理从XML实例参考 文档(参见前面的链接)到我们 @ bean 使用Java,它看起来像 以下:

```
// an HTTP Session-scoped bean exposed as a proxy
@Bean
@Scope(value = "session", proxyMode = ScopedProxyMode.TARGET_CLASS)
public UserPreferences userPreferences() {
 return new UserPreferences();
}

@Bean
public Service userService() {
 UserService service = new SimpleUserService();
 // a reference to the proxied userPreferences bean
 service.setUserPreferences(userPreferences());
 return service;
}
```

定制bean命名

默认情况下,配置类使用 @ bean 方法的名称的名称 产生的bean。 这个功能可以被覆盖,然而,由于 名称 属性。

```
@Configuration
public class AppConfig {

  @Bean(name = "myFoo")
  public Foo foo() {
    return new Foo();
```

```
  }

}
```

Bean混淆

作为讨论 SectionA 5 3 1,一个命名beansa ,有时候 希望给一个bean多个名称,否则称为 Bean混淆 。 这个 名称 属性的 @ bean 注释接受一个字符串 数组为这个目的。

```
@Configuration
public class AppConfig {

  @Bean(name = { "dataSource", "subsystemA-dataSource", "subsystemB-dataSource" })
  public DataSource dataSource() {
      // instantiate, configure and return DataSource bean...
  }

}
```

5.12.4A使用 @ configuration 注释

@ configuration 是一个类级别注释 表明一个对象是一个源的bean定义。 @ configuration 类声明bean通过 公共 @ bean 带注释的方法。 调用 @ bean 方法 @ configuration 类还可以用来 定义国米bean依赖关系。 看到 SectionA 5 12 1,一个基本概念: @ bean 和 @ configuration 一个 对于 一个总体介绍。

注入国米bean依赖

当 @ bean 年代都依赖一个 另一个,表示依赖关系很简单,有一个bean 方法调用另一个:

```
@Configuration
public class AppConfig {

  @Bean
  public Foo foo() {
      return new Foo(bar());
  }

  @Bean
  public Bar bar() {
      return new Bar();
  }

}
```

在上面的例子中, foo bean接收参考 到 酒吧 通过构造函数注入。

> **注意**
>
> 这个方法的声明国米bean依赖只能当 这个 @ bean 方法内部声明的 @ configuration 类。 你不能申报国米bean依赖使用普通 component 类。

查找方法注入

正如前面提到的, 查找方法注入 是一种先进的功能,你应该吗 使用很少。 它是有用的情况下,有一个单例范围bean有一个 依赖一个原型作用域bean。 使用Java为这种类型的 配置提供了一种自然的方式实现此模式。

```
public abstract class CommandManager {
  public Object process(Object commandState) {
      // grab a new instance of the appropriate Command interface
      Command command = createCommand();

      // set the state on the (hopefully brand new) Command instance
      command.setState(commandState);
      return command.execute();
  }

  // okay... but where is the implementation of this method?
  protected abstract Command createCommand();
}
```

使用java配置支持,您可以创建的一个子类 commandManager 在抽象的 createCommand() 方法被覆盖的方式 它看起来新(原型)命令对象:

```
@Bean
@Scope("prototype")
public AsyncCommand asyncCommand() {
  AsyncCommand command = new AsyncCommand();
  // inject dependencies here as required
  return command;
}

@Bean
public CommandManager commandManager() {
  // return new anonymous implementation of CommandManager with command() overridden
  // to return a new prototype Command object
  return new CommandManager() {
      protected Command createCommand() {
          return asyncCommand();
      }
  }
}
```

进一步的信息关于基于java的配置工作 内部

下面的示例显示了一个 @ bean 注释 方法被称为两次:

```
@Configuration
public class AppConfig {

  @Bean
  public ClientService clientService1() {
    ClientServiceImpl clientService = new ClientServiceImpl();
    clientService.setClientDao(clientDao());
    return clientService;
  }
  @Bean
  public ClientService clientService2() {
    ClientServiceImpl clientService = new ClientServiceImpl();
    clientService.setClientDao(clientDao());
    return clientService;
  }

  @Bean
  public ClientDao clientDao() {
    return new ClientDaoImpl();
  }
}
```

clientDao() 被称为一旦在吗 clientService1() 一旦在 clientService2() 。 因为这个方法创建一个新的 实例的 ClientDaoImpl 并返回它,你会 通常期望拥有两个实例(一个用于每个服务)。 这绝对 将有问题的:在春天,实例化bean有一个吗 singleton 默认范围。 这是神奇的 来自:所有 @ configuration 类子类在 启动时间与 CGLIB 。 在子类,孩子 方法检查容器首先对于任何缓存(作用域)bean之前 调用父方法和创建一个新的实例。 注意,在春天 3.2,不需要添加到您的类路径中,因为,CGLIB CGLIB类已经重新打包在org。 springframework和包括 直接在spring核心JAR。

注意

这种行为可能是不同的根据你的范围 bean。 我们正在谈论单身在这里。

注意

存在一些限制因为CGLIB动态 添加特性在启动时间:

- 配置类不应该是最后的
- 他们应该有一个不带参数的构造函数

5.12.5A构成基于java的配置

使用 @ import 注释

一样 <进口/ > 元素用于 在Spring XML文件来帮助在模块化配置, @ import 注释允许加载 @ bean 定义从另一个配置 类:

```
@Configuration
public class ConfigA {
  public @Bean A a() { return new A(); }
}
```

```
@Configuration
@Import(ConfigA.class)
public class ConfigB {
  public @Bean B b() { return new B(); }
}
```

现在,而不是需要同时指定 ConfigA.class 和 ConfigB.class 当实例化背景下,只有 ConfigB 需要 提供 明确:

```
public static void main(String[] args) {
  ApplicationContext ctx = new AnnotationConfigApplicationContext(ConfigB.class);

  // now both beans A and B will be available...
  A a = ctx.getBean(A.class);
  B b = ctx.getBean(B.class);
}
```

这种方法简化了容器实例化,因为只有一个类 需要处理,而不需要开发人员记住 一个潜在的大量的 @ configuration 类 在施工期间。

注入依赖进口 @ bean 定义

上面的例子是有效的,但过于简单。 在大多数实际 场景,豆子会彼此依赖跨 配置类。 当使用XML,这不是一个问题,本身, 因为没有编译器的参与,和一个可以简单地声明 ref = " someBean" 并且相信春天会工作 在容器初始化。 当然,当使用 @ configuration 类,Java编译器的地方 约束的配置模型,引用其他 bean必须有效的Java语法。

幸运的是,解决这个问题很简单。 记住, @ configuration 类是最终只是另一个 豆在容器——这意味着他们可以利用 @ autowired 注入元数据就像任何其他 豆!

让我们考虑一个更真实的场景中与几个 @ configuration 类,每个取决于豆子 宣布在 其他:

```
@Configuration
public class ServiceConfig {
  private @Autowired AccountRepository accountRepository;

  public @Bean TransferService transferService() {
      return new TransferServiceImpl(accountRepository);
  }
}

@Configuration
public class RepositoryConfig {
  private @Autowired DataSource dataSource;

  public @Bean AccountRepository accountRepository() {
      return new JdbcAccountRepository(dataSource);
  }
}

@Configuration
@Import({ServiceConfig.class, RepositoryConfig.class})
public class SystemTestConfig {
  public @Bean DataSource dataSource() { /* return new DataSource */ }
}

public static void main(String[] args) {
  ApplicationContext ctx = new AnnotationConfigApplicationContext(SystemTestConfig.class);
  // everything wires up across configuration classes...
  TransferService transferService = ctx.getBean(TransferService.class);
  transferService.transfer(100.00, "A123", "C456");
}
```

完全资格进口咖啡豆,易于导航

在上面的场景中,使用 @ autowired 作品 好,并提供所需的模块化,但确定到底 autowired的bean定义声明仍有点吗 模棱两可的。 例如,作为一个开发人员查看 ServiceConfig ,你怎么知道的确切位置 AccountRepository @ autowired bean声明吗? 这不是显式的代码,这可能是刚刚好。 记得 , SpringSource工具套件 提供的工具,可以渲染 图表表明一切都是有线——这可能是所有的你 需要。 同时,您的Java IDE可以很容易地找到所有声明和使用 的 AccountRepository 式,将很快 显示你的位置 @ bean 方法 返回类型。

这种模糊性的情况下是不能接受的,你想 直接导航从在IDE从一个吗 @ configuration 类到另一个,可以考虑 自动装配配置类本身:

```
@Configuration
public class ServiceConfig {
  private @Autowired RepositoryConfig repositoryConfig;
```

```
  public @Bean TransferService transferService() {
      // navigate 'through' the config class to the @Bean method!
      return new TransferServiceImpl(repositoryConfig.accountRepository());
  }
}
```

在上面的情况,它是完全明确的地方 AccountRepository 被定义。 然而, ServiceConfig 现在是紧耦合的吗 RepositoryConfig ,这是一种折衷。 这种紧密 耦合可以稍微缓解通过使用基于接口的或 摘要基于类的 @ configuration 类。 考虑以下:

```
@Configuration
public class ServiceConfig {
  private @Autowired RepositoryConfig repositoryConfig;

  public @Bean TransferService transferService() {
      return new TransferServiceImpl(repositoryConfig.accountRepository());
  }
}

@Configuration
public interface RepositoryConfig {
  @Bean AccountRepository accountRepository();
}

@Configuration
public class DefaultRepositoryConfig implements RepositoryConfig {
  public @Bean AccountRepository accountRepository() {
      return new JdbcAccountRepository(...);
  }
}

@Configuration
@Import({ServiceConfig.class, DefaultRepositoryConfig.class}) // import the concrete config!
public class SystemTestConfig {
  public @Bean DataSource dataSource() { /* return DataSource */ }
}

public static void main(String[] args) {
  ApplicationContext ctx = new AnnotationConfigApplicationContext(SystemTestConfig.class);
  TransferService transferService = ctx.getBean(TransferService.class);
  transferService.transfer(100.00, "A123", "C456");
}
```

现在 ServiceConfig 是松散耦合与尊重 具体 DefaultRepositoryConfig ,和 内置的IDE工具仍然是有用的:它将是容易的 开发人员得到一个类型层次结构的 RepositoryConfig 实现。 通过这种方式, 导航 @ configuration 类及其 依赖关系变得不平常的不同过程 导航基于接口的代码。

结合Java和XML配置

春天的 @ configuration 类支持不 目标是100%完成替代弹簧XML。 一些设施 如弹簧XML名称空间是一个理想的方式来配置集装箱。 在情况下,XML是方便的和必要的,你有一个 选择:要么实例化容器在一个 “以xml为中心的” 方式使用, 例如, ClassPathXmlApplicationContext ,或在一个 “以java为中心的 “时尚使用 所 和 @ImportResource 注释导入XML作为 需要。

以xml为中心的使用 @ configuration 类

它最好是引导Spring容器从XML 和包括 @ configuration 类在一个特别 时尚。 例如,在一个大型的现有代码库,使用弹簧XML,这将会更容易创建 @ configuration 根据基础类,包括他们从现有的XML 文件。 下面你会发现使用的选项 @ configuration 类在这种 “以xml为中心” 的情况。

宣布 @ configuration 类作为普通 春天 < bean / > 元素

记住, @ configuration 类 最终只是bean定义在容器。 在这个例子中, 我们创建一个 @ configuration 类命名 AppConfig 并将其包括在 系统测试配置xml 作为一个 < bean / > 定义。 因为 <上下文:注释配置/ > 切换 在,容器将识别 @ configuration 注释和处理 @ bean 方法中声明的 AppConfig 正确。

```
@Configuration
public class AppConfig {
  private @Autowired DataSource dataSource;

  public @Bean AccountRepository accountRepository() {
      return new JdbcAccountRepository(dataSource);
  }

  public @Bean TransferService transferService() {
      return new TransferService(accountRepository());
  }
}
```

```
system-test-config.xml
<beans>
  <!-- enable processing of annotations such as @Autowired and @Configuration -->
  <context:annotation-config/>
  <context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>

  <bean class="com.acme.AppConfig"/>

  <bean class="org.springframework.jdbc.datasource.DriverManagerDataSource">
     <property name="url" value="${jdbc.url}"/>
     <property name="username" value="${jdbc.username}"/>
     <property name="password" value="${jdbc.password}"/>
  </bean>
</beans>
```

```
jdbc.properties
jdbc.url=jdbc:hsqldb:hsql://localhost/xdb
jdbc.username=sa
jdbc.password=
```

```
public static void main(String[] args) {
  ApplicationContext ctx = new ClassPathXmlApplicationContext("classpath:/com/acme/system-test-config.xml");
  TransferService transferService = ctx.getBean(TransferService.class);
  // ...
}
```

注意

在 系统测试配置xml 以上, AppConfig < bean / > 不声明一个 id 元素。 而这将是可行的 所以,它是不必要的假设没有其他bean会参考 它,就不太可能会显式地去 容器的名字。 同样的 数据源 豆——它是只autowired的类型,所以一个显式的bean id 没有严格要求。

使用 <上下文:组件扫描/ > 到 接 @ configuration 类

因为 @ configuration 是元注释与 component , @ configuration 注释类 组件扫描自动候选人。 使用相同的 场景如上所述,我们可以重新定义 系统测试配置xml 利用 组件扫描。 注意,在这种情况下,我们不需要 显式地声明 <上下文:注释配置/ > ,因为 <上下文:组件扫描/ > 使所有 相同 功能。

```
system-test-config.xml
<beans>
  <!-- picks up and registers AppConfig as a bean definition -->
  <context:component-scan base-package="com.acme"/>
  <context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>

  <bean class="org.springframework.jdbc.datasource.DriverManagerDataSource">
     <property name="url" value="${jdbc.url}"/>
     <property name="username" value="${jdbc.username}"/>
     <property name="password" value="${jdbc.password}"/>
  </bean>
</beans>
```

@ configuration 类中心使用XML @ImportResource

在应用 @ configuration 类 是主要的机制来配置容器,它仍然会吗 可能有必要使用至少一些XML。 在这些场景中, 简单地使用 @ImportResource 和定义只有尽可能多的 XML是必要的。 这样做达到一个“篇”的方法来 配置容器和保持XML到最低限度。

```
@Configuration
@ImportResource("classpath:/com/acme/properties-config.xml")
public class AppConfig {
  private @Value("${jdbc.url}") String url;
  private @Value("${jdbc.username}") String username;
  private @Value("${jdbc.password}") String password;

  public @Bean DataSource dataSource() {
     return new DriverManagerDataSource(url, username, password);
  }
}
```

```
properties-config.xml
<beans>
  <context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>
</beans>
```

```
jdbc.properties
```

```
jdbc.url=jdbc:hsqldb:hsql://localhost/xdb
jdbc.username=sa
jdbc.password=
```

```
public static void main(String[] args) {
  ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);
  TransferService transferService = ctx.getBean(TransferService.class);
  // ...
}
```

5.13一个注册一个 LoadTimeWeaver

这个 LoadTimeWeaver 使用弹簧动态吗 变换类被加载到Java虚拟机(JVM)。

使装入时编织添加 @EnableLoadTimeWeaving 你的 @ configuration 类:

```
@Configuration
@EnableLoadTimeWeaving
public class AppConfig {

}
```

另外对于XML配置使用 背景:加载时间织布 元素:

```
<beans>
    <context:load-time-weaver/>
</beans>
```

一旦配置 ApplicationContext 。 任何bean内 ApplicationContext 可以实现 LoadTimeWeaverAware ,从而获得一个 装载时编织器实例的引用。 这是特别有用 结合 Spring的JPA支持 在 可能需要装入时编织为JPA类转换。 咨询 这个 LocalContainerEntityManagerFactoryBean Javadoc 更多的细节。 更多关于AspectJ装入时编织,看到 SectionA 9 8 4,一个装入时编织与AspectJ在春季Frameworka 。

5.14一个额外的功能 ApplicationContext

作为讨论的是章介绍, org.springframework.beans.factory 包提供了基本 功能管理和操纵豆类,包括在一个 程序化的方法。 这个 org.springframework.context 包 添加 ApplicationContext 接口,它 扩展了 BeanFactory 接口,在 除了扩展其他接口来提供额外的功能 在一个更 应用面向框架风格 。 许多 人们使用 ApplicationContext 在一个 完全声明式的时尚,甚至以编程方式创建它,但是 而是依赖支持类等 ContextLoader 自动实例化一个 ApplicationContext 正常的一部分 启动过程的J2EE web 应用程序。

提高 BeanFactory 功能 更多的面向框架风格上下文包还提供了 以下功能:

- 访问消息i18n风格 ,通过 MessageSource 接口。
- 对资源的访问 ,比如url和文件, 通过 ResourceLoader 接口。
- 事件发表 到bean实现 ApplicationListener 接口,通过 使用 ApplicationEventPublisher 接口。
- 加载多个(分层)上下文 , 允许每个集中在一个特定的层,如web 层的一个应用程序,通过 HierarchicalBeanFactory 接口。

5.14.1A国际化使用 MessageSource

这个 ApplicationContext 接口 扩展接口称为 MessageSource , 因此提供国际化(i18n)功能。 春天 还提供了接口 HierarchicalMessageSource ,它可以解决 信息分层次。 这些接口提供了基础 在这春天消息解决效果。 在这些定义的方法 界面包括:

- 字符串(字符串的代码,对象getMessage[]参数,字符串 默认情况下,地区loc) :基本方法用于检索 消息从 MessageSource 。 当没有 消息是发现指定场所的,默认的消息 使用。 任何参数传递成为替代值,使用 MessageFormat 提供的功能 标准库。
- 字符串(字符串的代码,对象getMessage[]参数,语言环境 loc) :基本上与前面的方法,但是 有一点区别:不可以指定默认消息; 如果 消息不能被发现,一个 NoSuchMessageException 抛出。
- 字符串getMessage(MessageSourceResolvable可分解的, 地区地区) :所有属性用于前面的 方法也包裹在一个类命名 MessageSourceResolvable ,您可以 使用这种方法。

当一个 ApplicationContext 正在加载, 它会自动搜索 MessageSource bean中定义的上下文。 bean必须有名称 MessageSource 。 如果这样一个 豆是发现,所有调用前面的方法都是委托给 消息源。 如果没有消息来源是发现, ApplicationContext 试图找到一个 父包含bean使用相同的名称。 如果是,它使用的bean 随着 MessageSource 。 如果

ApplicationContext 无法找到任何源 的消息,一个空的 DelegatingMessageSource 是 实例化为了能够接受调用定义的方法以上。

弹簧提供了两个 MessageSource 实现, ResourceBundleMessageSource 和 StaticMessageSource 。 同时实现 HierarchicalMessageSource 为了做 嵌套的消息。 这个 StaticMessageSource 很少 但提供了编程方式使用添加消息源。 这个 ResourceBundleMessageSource 显示在 下面的例子:

```
<beans>
<bean id="messageSource"
    class="org.springframework.context.support.ResourceBundleMessageSource">
  <property name="basenames">
    <list>
      <value>format</value>
      <value>exceptions</value>
      <value>windows</value>
    </list>
  </property>
</bean>
</beans>
```

在示例假设您有三个资源包定义 在您的类路径称为 格式 , 异常 和 Windows 。 任何请求 解决一个消息将被处理的标准方式解决。JDK 通过resourcebundle的消息。 为目的的例子,假设 内容的两个以上的资源包文件...

```
# in format.properties
message=Alligators rock!
```

```
# in exceptions.properties
argument.required=The '{0}' argument is required.
```

一个程序执行 MessageSource 功能是下一个示例所示。 记住,所有 ApplicationContext 实现也 MessageSource 实现,因此可以转换为 这个 MessageSource 接口。

```
public static void main(String[] args) {
  MessageSource resources = new ClassPathXmlApplicationContext("beans.xml");
  String message = resources.getMessage("message", null, "Default", null);
  System.out.println(message);
}
```

从上面的输出结果程序将.....

```
Alligators rock!
```

所以总结一下, MessageSource 定义 在一个文件叫做 它指明 ,它存在的根本原因 你的类路径。 这个 MessageSource bean 定义 这个概念的 “可伸缩反应堆” basenames 财产。 这三个文件传递进来 列表的 basenames 属性文件存在 你的类路径的根和被称为 格式属性 , 异常属性 ,和 窗口属性 分别。

下一个例子显示了参数传递到信息查找;这些 参数将被转换成字符串和插入的占位符 查找信息。

```
<beans>

  <!-- this MessageSource is being used in a web application -->
  <bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basename" value="exceptions"/>
  </bean>

  <!-- lets inject the above MessageSource into this POJO -->
  <bean id="example" class="com.foo.Example">
    <property name="messages" ref="messageSource"/>
  </bean>

</beans>
```

```
public class Example {

  private MessageSource messages;

  public void setMessages(MessageSource messages) {
    this.messages = messages;
  }

  public void execute() {
    String message = this.messages.getMessage("argument.required",
        new Object [] {"userDao"}, "Required", null);
    System.out.println(message);
  }
```

```
}
```

得到的输出结果的调用 execute() 方法将……

```
The userDao argument is required.
```

关于国际化(i18n),春天的不同 MessageResource 实现遵循相同的 语言环境的分辨率和回退的规则为标准JDK ResourceBundle 。 总之,和继续 例子 MessageSource 前面定义的,如果你想要的 解决信息对英国(扎)地区,你将创建 文件称为 格式en gb属性 , 异常en gb属性 ,和 windows en gb属性 分别。

一般来说,语言环境决议是由周围的环境 的应用程序。 在这个例子中,语言环境对(英国) 消息将被解决是手动指定。

```
# in exceptions_en_GB.properties
argument.required=Ebagum lad, the '{0}' argument is required, I say, required.
```

```
public static void main(final String[] args) {
  MessageSource resources = new ClassPathXmlApplicationContext("beans.xml");
  String message = resources.getMessage("argument.required",
      new Object [] {"userDao"}, "Required", Locale.UK);
  System.out.println(message);
}
```

结果输出上述程序将运行 被……

```
Ebagum lad, the 'userDao' argument is required, I say, required.
```

你也可以使用 MessageSourceAware 接口来获取一个引用 MessageSource 这被定义。 任何bean, 定义在一个 ApplicationContext 实现 这个 MessageSourceAware 接口是注射 应用程序上下文的 MessageSource 当 bean创建和配置。

注意

作为一种替代 ResourceBundleMessageSource ,弹簧提供了一个 ReloadableResourceBundleMessageSource 类。 这 变体支持相同的包文件格式,但更灵活的比 基于标准JDK ResourceBundleMessageSource 实现。 特别是,它允许阅读文件 从任何Spring资源位置(不只是从类路径)和 支持热重载的包属性文件(而有效 缓存它们之间)。 检查 ReloadableResourceBundleMessageSource javadoc的 细节。

5.14.2A标准和定制的事件

事件处理的 ApplicationContext 提供通过 ApplicationEvent 类和 ApplicationListener 接口。 如果一个bean 实现 ApplicationListener 接口是部署到上下文,每次一个 ApplicationEvent 被发布到 ApplicationContext ,那豆是通知。 从本质上讲,这是标准的 观察者 设计 模式。 Spring提供了以下标准事件:

5.7为多。 一个内置的事件

事件	解释
ContextRefreshedEvent	发表当 ApplicationContext 初始化 或刷新,例如,使用 refresh() 方法 ConfigurableApplicationContext 接口。 这里的“初始化”意味着所有bean加载, 后处理器bean是检测并激活,单例对象是 预先实例化, ApplicationContext 对象是准备好了 对于使用。 只要上下文没有被关闭,一个刷新可以 多次被触发,只要这样的选择 ApplicationContext 其实 支持这种“热”刷新。 例如, XmlWebApplicationContext 支持热 刷新, GenericApplicationContext 不。
ContextStartedEvent	发表当 ApplicationContext 是开始, 使用 start() 方法 ConfigurableApplicationContext 接口。 “开始”这意味着所有 生命周期 bean接收一个显式的 开始信号。 通常这个信号是用来启动bean之后 一个显式的停止,但它也可以用来启动组件 没有被配置为自动运行,例如,组件 这还没有开始初始化。
ContextStoppedEvent	发表当 ApplicationContext 是停止了, 使用 停止() 方法 ConfigurableApplicationContext 接口。 “停止”这意味着所有 生命周期 bean接收一个显式的 停止信号。 一个停止上下文可能重启通过 start() 调用。
	发表当 ApplicationContext 是封闭的,使用 这个 close() 方法

ContextClosedEvent	ConfigurableApplicationContext 接口。 “封闭”这意味着所有单例bean 摧毁了。 一个封闭的环境达到生命终止,不得 刷新或重启。
RequestHandledEvent	一个网络自身事件告诉所有的豆子,一个HTTP请求 已经维修。 这个事件发表 在 在请求完成。 这个事件是 只适用于使用Spring的web应用程序 DispatcherServlet 。

您还可以创建和发布您自己的自定义事件。 这个例子 演示了一个简单的类,它扩展了春天的 ApplicationEvent 基类:

```
public class BlackListEvent extends ApplicationEvent {
  private final String address;
  private final String test;

  public BlackListEvent(Object source, String address, String test) {
      super(source);
      this.address = address;
      this.test = test;
  }

  // accessor and other methods...
}
```

发布一个自定义 ApplicationEvent ,叫 publishEvent() 方法在一个 ApplicationEventPublisher 。 通常这 通过创建一个类,它实现了吗 ApplicationEventPublisherAware 和 它注册作为一个Spring bean。 下面的例子演示了这样一个 类:

```
public class EmailService implements ApplicationEventPublisherAware {

  private List<String> blackList;
  private ApplicationEventPublisher publisher;

  public void setBlackList(List<String> blackList) {
      this.blackList = blackList;
  }

  public void setApplicationEventPublisher(ApplicationEventPublisher publisher) {
      this.publisher = publisher;
  }

  public void sendEmail(String address, String text) {
      if (blackList.contains(address)) {
          BlackListEvent event = new BlackListEvent(this, address, text);
          publisher.publishEvent(event);
          return;
      }
      // send email...
  }
}
```

在配置时,Spring容器将检测到 EmailService 实现 ApplicationEventPublisherAware 并将 自动调用 setApplicationEventPublisher() 。 在现实中, 参数传递将Spring容器本身;你只是 与应用程序交互上下文通过它 ApplicationEventPublisher 接口。

接受定制 ApplicationEvent ,创建 一个类,它实现了 ApplicationListener 并注册它作为一个Spring bean。 下面的例子演示了这样 一个类:

```
public class BlackListNotifier implements ApplicationListener<BlackListEvent> {

  private String notificationAddress;

  public void setNotificationAddress(String notificationAddress) {
      this.notificationAddress = notificationAddress;
  }

  public void onApplicationEvent(BlackListEvent event) {
        // notify appropriate parties via notificationAddress...
  }
}
```

注意, ApplicationListener 是 通用参数化与该类型的自定义事件, BlackListEvent 。 这意味着 onApplicationEvent() 方法可以保持类型安全的, 避免需要向下类型转换。 你可以注册成为许多事件 听众如你所愿,但请注意,默认情况下事件监听器接收 事件同步。 这意味着 publishEvent() 方法将阻塞直到所有听众 已完成处理事件。 这样做的好处同步和 单线程的方法是,当一个监听器接收到一个事件,它 是在事务上下文的出版商如果事务 上下文是可用的。 如果一个策略来事件发布变得 有必要,请参考 JavaDoc春天的 ApplicationEventMulticaster 接口。

下面的示例展示了使用bean定义 注册和配置上面的每个类:

```
<bean id="emailService" class="example.EmailService">
 <property name="blackList">
    <list>
        <value>known.spammer@example.org</value>
        <value>known.hacker@example.org</value>
        <value>john.doe@example.org</value>
    </list>
 </property>
</bean>

<bean id="blackListNotifier" class="example.BlackListNotifier">
 <property name="notificationAddress" value="blacklist@example.org"/>
</bean>
```

把它们结合在一起,当 sendEmail() 方法 EmailService 豆被调用时,如果有 任何电子邮件,应该列入黑名单,一个自定义事件类型 BlackListEvent 发表。 这个 blackListNotifier bean注册作为一个 ApplicationListener 从而接收 BlackListEvent ,这时它就可以通知 适当的政党。

注意

春天的事件机制是专为简单的沟通 Spring bean之间在同一应用程序上下文。 然而,对于 更复杂的企业集成的需要, 分别维护 春天 集成 项目建设提供了全面的支持 轻量级的, 面向模式 、事件驱动的体系结构,建立 著名的春天的编程模型。

5.14.3A方便访问底层的资源

为获得最佳的使用和理解应用程序上下文,用户 一般应该熟悉Spring的吗 资源 抽象,如前文所述 章 ChapterA 6, 资源 。

一个应用程序上下文是一个 ResourceLoader ,它可用于负载 资源 年代。 资源 本质上是一个更加丰富的功能吗 版本的JDK类 java净url ,事实上, 的实现 资源 包装一个 实例的 java净url 在适当的地方。 一个 资源 可以获得低级资源 从几乎任何位置在一个透明的方式,包括从 类路径,文件系统位置,任何地方可描写的标准 URL,以及其他一些变化。 如果资源位置的字符串是一个 简单的路径没有任何特殊的前缀,这些资源来自 是具体的和适当的到实际的应用程序上下文类型。

您可以配置一个bean部署到应用程序上下文 实现特殊的回调接口, ResourceLoaderAware ,自动 在初始化的时候召回与应用程序上下文本身 传入的 ResourceLoader 。 你可以 还公开属性的类型 资源 , 被用来访问静态资源;他们将被注入到它像任何其他属性。 你可以指定那些 资源 属性为简单的字符串的路径, 和依赖特殊的JavaBean 属性编辑器 这是自动 登记的上下文,这些文本字符串转换到实际 资源 当bean对象 部署。

位置路径或路径提供给一个 ApplicationContext 构造函数实际上是 资源字符串,在简单的形式是恰当的治疗 特定的上下文实现。 ClassPathXmlApplicationContext 对待一个简单的 位置路径作为一个类路径的位置。 您还可以使用位置路径 (资源字符串)和特殊前缀力载荷的定义 从类路径或一个URL,而不管实际的上下文类型。

5.14.4A方便 ApplicationContext 为web应用程序的实例化

您可以创建 ApplicationContext 实例声明通过使用,例如,一个 ContextLoader 。 当然,你也可以创建 ApplicationContext 实例 以编程方式通过使用其中一个 ApplicationContext 实现。

这个 ContextLoader 机制有两种 口味: ContextLoaderListener 和 ContextLoaderServlet 。 他们有相同的 功能,但区别在于听者版本是不可靠的 Servlet 2.3容器。 在Servlet 2.4规范,Servlet上下文 听众必须执行后立即为web Servlet上下文 创建应用程序,并可用于服务第一个请求(和 当Servlet上下文即将被关闭)。 作为这样一个Servlet 上下文侦听器是一个理想的地方初始化弹簧 ApplicationContext 。 所有的事情都是公平的, 你应该喜欢 ContextLoaderListener ; 更多信息兼容性,看看的Javadoc ContextLoaderServlet 。

你可以注册一个 ApplicationContext 使用 ContextLoaderListener 如下:

```
<context-param>
<param-name>contextConfigLocation</param-name>
<param-value>/WEB-INF/daoContext.xml /WEB-INF/applicationContext.xml</param-value>
</context-param>

<listener>
<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

<!-- or use the ContextLoaderServlet instead of the above listener
<servlet>
<servlet-name>context</servlet-name>
<servlet-class>org.springframework.web.context.ContextLoaderServlet</servlet-class>
<load-on-startup>1</load-on-startup>
```

```
</servlet>
-->
```

侦听器检查 contextConfigLocation 参数。 如果参数不存在,侦听器使用 / - inf /中 作为默认的设置。 当 参数 并 存在,侦听器分离了 通过使用预定义的字符串分隔符(逗号、分号和空格) 和使用值作为位置将应用程序上下文 搜查了。 ant是基于路径模式也支持。 的例子是 / - inf / *上下文xml 所有文件名称以结束 与“上下文。 xml”,居住在“不想”目录,和 / - inf / * * / *上下文xml ,因为所有这些文件在任何 子目录的“不想”。

您可以使用 ContextLoaderServlet 而不是 ContextLoaderListener 。 Servlet使用 contextConfigLocation 参数只是作为侦听器 确实。

5.14.5A部署一个弹簧ApplicationContext作为J2EE RAR文件

在Spring 2.5和以后,可以部署一个春天 ApplicationContext作为RAR文件,封装上下文和它的所有 要求bean类和库jar在J2EE RAR部署单元。 这 相当于一个独立的ApplicationContext引导,就 托管在J2EE环境中,能够访问J2EE服务器 设施。 RAR部署是一种更自然的替代的场景 部署一个无头WAR文件,实际上,一个WAR文件没有任何HTTP 入口点,仅用于引导一个春天 ApplicationContext在J2EE环境中。

RAR部署是理想的应用程序上下文,不需要HTTP 入口点而是只有消息端点和预定 乔布斯。 豆子在这样的环境下可以使用应用服务器等资源 JTA事务管理器和JDBC数据源的jndi绑定和JMS ConnectionFactory的实例,也可以注册平台的JMX 服务器-所有通过弹簧的标准事务管理和JNDI和 JMX支持设施。 应用程序组件还可以与其进行交互 应用程序服务器的JCA WorkManager 通过弹簧的 TaskExecutor 抽象。

检查出的JavaDoc SpringContextResourceAdapter 类的配置细节 参与RAR部署。

对于一个简单的部署一个弹簧ApplicationContext作为 J2EE RAR文件: 包所有应用程序类成一个RAR文件, 这是一个标准的JAR文件和一个不同的文件扩展名。 添加所有 所需的库jar到根的RAR存档。 添加一个 “meta - inf / ra。 xml部署描述符”(见 SpringContextResourceAdapter 年代JavaDoc)和 相应的Spring XML bean定义文件(s)(一般 “meta - inf /中”),并将由此产生的RAR文件到 您的应用程序服务器的部署目录中。

注意

这样的RAR部署单元通常是独立的;他们不 组件公开到外面的世界,即使是对其他模块的 相同的应用程序。 交互与一个基于rar ApplicationContext 通过JMS目的地通常发生,它与其他股票 模块。 一个基于rar ApplicationContext也可以,例如,时间表 有些工作,应对新的文件在文件系统(或相似的)。 如果它 需要允许同步访问从外面看,它可以为例 出口RMI端点,它当然可以被其他应用程序使用 模块在同一台机器上。

5.15一个BeanFactory

这个 BeanFactory 提供底层基础 对于Spring的IoC功能,但它只是直接使用在集成 与其他第三方框架和现在主要在自然历史 对于大多数用户的春天。 这个 BeanFactory 和 相关的接口,如 BeanFactoryAware , InitializingBean , DisposableBean ,仍然出现在春天的 向后兼容性的目的与众多的第三方 框架,结合弹簧。 经常第三方组件, 可以不使用更现代的等价物如 @PostConstruct 或 @PreDestroy 为了保持兼容的JDK 1.4或 避免依赖于jsr - 250。

本节提供了额外的背景的差异 之间的 BeanFactory 和 ApplicationContext 和如何访问 IoC容器直接通过一个经典的单例查找。

5.15.1A BeanFactory 或 ApplicationContext 吗?

使用一个 ApplicationContext 除非你 有一个充分的理由不这样做。

因为 ApplicationContext 包括所有的功能 BeanFactory ,它通常是被推荐的 在 BeanFactory ,除了一些 情况,比如在一个 applet 在记忆 消费可能是关键和一些额外的千字节可能使一个 差异。 然而,对于大多数典型的企业应用程序和 系统中, ApplicationContext 就是 你会想要使用。 Spring 2.0,后来使 重 使用 BeanPostProcessor 扩展点 (影响代理等等)。 如果你只使用一个普通的 BeanFactory ,大量的支持 如交易和AOP不会生效,至少不是没有 一些额外的步骤在你的部分。 这种情况可能是令人困惑,因为 没有什么实际上是错误的配置。

下表列出了所提供的功能 BeanFactory 和 ApplicationContext 接口和 实现。

5.8为多。 一个特征矩阵

特性	BeanFactory	ApplicationContext
Bean实例化/布线	是的	是的
自动 BeanPostProcessor 登记	没有	是的
自动 BeanFactoryPostProcessor 登记	没有	是的
方便 MessageSource 访问(i18n)	没有	是的
ApplicationEvent 出版	没有	是的

明确注册一个bean后处理器与一个 BeanFactory 实现,你必须 写这样的代码:

```
ConfigurableBeanFactory factory = new XmlBeanFactory(...);

// now register any needed BeanPostProcessor instances
MyBeanPostProcessor postProcessor = new MyBeanPostProcessor();
factory.addBeanPostProcessor(postProcessor);

// now start using the factory
```

明确注册一个 BeanFactoryPostProcessor 当使用一个 BeanFactory 实现,你必须 写这样的代码:

```
XmlBeanFactory factory = new XmlBeanFactory(new FileSystemResource("beans.xml"));

// bring in some property values from a Properties file
PropertyPlaceholderConfigurer cfg = new PropertyPlaceholderConfigurer();
cfg.setLocation(new FileSystemResource("jdbc.properties"));

// now actually do the replacement
cfg.postProcessBeanFactory(factory);
```

在这两种情况下,明确登记步骤是不方便, 是一个原因为什么不同 ApplicationContext 实现 优先高于平原 BeanFactory 实现在绝大多数弹簧支持应用程序, 尤其是当使用 BeanFactoryPostProcessors 和 BeanPostProcessors 。 这些机制实现 重要的功能,如财产占位符替换和 AOP。

5.15.2A胶水代码和邪恶的单例

最好是写在一个依赖项注入最应用程序代码 (DI)风格,代码是服务的一个Spring IoC容器,有 自己的依赖关系由容器提供创建时,是完全没有意识到容器。 然而,对于小胶水 层的代码,有时需要领带其他代码在一起,你 有时候需要一个单例(或准单例)风格访问一个春天 IoC容器。 例如,第三方代码可能试图建构新的 对象直接(class . forname() 风格),没有了 能力的获得这些对象一个 Spring IoC容器。 如果 构造的对象由第三方代码是一个小的存根或代理, 然后使用一个单例风格访问Spring IoC容器吗 得到一个真正的对象来代表,然后控制反转仍 得到的大多数代码(对象走出 容器)。 因此大多数代码仍然没有意识到容器或它如何 被访问,仍脱离其他代码,所有随后的吗 福利。 ejb stub也可以使用这种方法来代表/代理一个 普通的Java实现对象,是从一个 Spring IoC容器。 虽然Spring IoC容器本身理想情况下不需要一个 单例,它可能是不切实际的在内存使用方面的或 初始化时间(当使用bean在Spring IoC容器这样的 作为一个Hibernate SessionFactory)对于每个 bean来使用它自己的,非单体Spring IoC容器。

查找应用程序上下文在服务定位器风格 有时唯一的选择来访问共享的spring管理 组件,如在EJB 2.1环境,或当你想要分享 一个作为父母,WebApplicationContexts ApplicationContext跨 WAR文件。 在这种情况下你应该考虑使用的实用程序类 ContextSingletonBeanFactoryLocator 定位器,描述在这 SpringSource团队博客条目 。

[1] 看到 背景
[2] 看到 SectionA 5.4.1之前,一个injectiona依赖性

6.一个资源

6.1一个介绍

Java的标准 java净url 类和 标准处理程序对于各种URL前缀不幸的是不完全的 充足的足够的所有访问底层的资源。 例如, 没有标准化的 url 实现 这可能是用于访问资源,需要获得 类路径或相对于一个 ServletContext 。 虽然它是可能的 新处理程序注册为专业 url 前缀(类似于现有的处理程序等的前缀 http:),这通常是很复杂的, url 接口仍然缺乏一些可取的 功能,如一个方法来检

查存在的 资源被指出。

6.2一个的 资源 接口

春天的 资源 接口指的是 成为一个更有能力的接口访问底层的抽象 资源。

```
public interface Resource extends InputStreamSource {
    boolean exists();
    boolean isOpen();
    URL getURL() throws IOException;
    File getFile() throws IOException;
    Resource createRelative(String relativePath) throws IOException;
    String getFilename();
    String getDescription();
}
```

```
public interface InputStreamSource {
    InputStream getInputStream() throws IOException;
}
```

一些最重要的方法 资源 界面:

- getInputStream() :定位并打开了 资源,返回一个 InputStream 阅读 从资源。 预计每个调用返回 新鲜 InputStream 。 都有责任 调用者关闭流。
- 存在() :返回一个 布尔 指示是否这个资源实际 存在于物理形式。
- isOpen() :返回一个 布尔 指示是否这个资源代表 一个手柄,一个开放的流。 如果 真正的 , InputStream 不能读多次,和 必须读一次只有然后关闭以避免资源泄漏。 将 是 假 对于所有通常的资源实现, 除了 InputStreamResource 。
- getDescription() :返回一个描述 对于这个资源,用于错误输出当处理 资源。 这通常是完全合格的文件名称或实际的 资源 的URL。

其他的方法让你获得一个实际 url 或 文件 对象 代表资源(如果底层的实现是兼容的, 和支持该功能)。

这个 资源 抽象是使用 广泛的在春天本身,作为一个参数类型在许多方法 当一个资源需要签名。 其他方法在某些Spring api (如构造函数不同 ApplicationContext 实现),拿一个 字符串 在朴实或简单的形式用于 创建一个 资源 恰当 上下文实现,或通过特殊的前缀 字符串 路径,允许调用者指定一个 特定 资源 实现必须 创建和使用。

而 资源 接口用于 很多春天和春季之前,它实际上是非常有用的使用作为一个 一般的实用程序类本身在您自己的代码,因为对资源的访问, 即使你的代码不知道或关心的任何其他部分的弹簧。 虽然这夫妇代码到春天,真的只有夫妻到这 小套实用工具类,它是作为一个更有能力 替代 url ,可以考虑 相当于其他库用于这一目的。

重要的是要注意, 资源 抽象不能替代 功能:它将它在可能的情况下。 例如,一个 UrlResource 将URL,使用包装 url 做它的工作。

6.3内置 资源 实现

有许多 资源 实现,来直接提供开箱即用的 春天:

6.3.1A UrlResource

这个 UrlResource 包装 java净url 的,并且可以用于访问任何 对象,通常可以通过一个URL,比如文件、一个HTTP 目标,FTP target,等。所有url都标准化 字符串 表示,这样适当的 标准化的前缀是用来表示一个URL类型从另一个。 这包括 文件: 访问文件系统路径, http: 为通过HTTP协议访问的资源, ftp: 通过FTP访问资源,等等。

一个 UrlResource 是由Java代码吗 显式地使用 UrlResource 构造函数,但 经常会被隐式地创建当你叫一个API中采用的方法吗 一个 字符串 的参数,它是用来代表一个 路径。 对于后一种情况,一个javabean 属性编辑器 将最终决定 哪种类型的 资源 创建。 如果 路径字符串包含几个知名(,)前缀如 类路径: ,它将创建一个合适的专业 资源 为该前缀。 然而,如果它 不能识别前缀,它会认为这只是一个标准 URL字符串,并将创建一个 UrlResource 。

6.3.2A ClassPathResource

这个类代表一个资源应得到 类路径。 它使用线程上下文类加载器要么,给定 类装入器,或一个给定的类加载资源。

这 资源 实现 支持分辨率 java输入输出文件 如果类 道路资源驻留在文件系统,但不是为类路径 资源驻留在一个罐子里,没有被扩展(由 servlet引擎,或任何环境)文件系统。 到 地址这各种 资源 实现永远支持分辨率作为 java净url 。

一个 ClassPathResource 是由Java代码吗 显式地使用 ClassPathResource 构造函数,但往往会被隐式地创建当你调用一个API方法将一个 字符串 论点 代表一个路径。 对于后一种情况,一个javabean 属性编辑器 将承认特别 前缀 类路径: 在字符串的路径,并创建一个 ClassPathResource 在这种情况下。

6.3.3A FileSystemResource

这是一个 资源 实现 对于 java输入输出文件 处理。 显然,它支持 分辨率作为 文件 ,作为一个 url 。

6.3.4A ServletContextResource

这是一个 资源 实现 对于 ServletContext 资源, 解释相对路径在相关的web应用程序的根 目录。

这总是支持流访问和URL访问,但只允许 java输入输出文件 当web应用程序访问 档案是展开的,资源是物理文件系统上。 是否扩大,像这样的文件系统,或者 直接访问从罐子里或其他地方像一个DB(它的 可以想象的)实际上是依赖于Servlet容器。

6.3.5A InputStreamResource

一个 资源 实现一个 鉴于 InputStream 。 这只能 使用如果没有具体的 资源 实现是适用的。 特别是,喜欢 ByteArrayResource 或任何基于文件的 资源 实现, 可能的。

与其他 资源 实现,这是一个描述符的 已经 打开资源——因此返回 真正的 从 isOpen() 。 不 使用它如果你需要保持资源描述符某处,或如果你 需要多次读取流。

6.3.6A ByteArrayResource

这是一个 资源 实现 对于一个给定的字节数组。 它创建了一个 ByteArrayInputStream 对于给定的字节 数组。

它是有用的对于加载内容从任何给定的字节数组,没有 不得不求助于一个一次性 InputStreamResource 。

6.4一个的 ResourceLoader

这个 ResourceLoader 接口指的是 去实现对象可以返回(即负载) 资源 实例。

```
public interface ResourceLoader {
    Resource getResource(String location);
}
```

所有应用程序上下文实现 ResourceLoader 接口,因此所有 应用程序上下文可以用来获得 资源 实例。

当你打电话 getResource() 在一个特定的 应用程序上下文,和位置路径指定的没有 特定的前缀,您将会得到一个 资源 类型,是恰当的 特定的应用程序上下文。 例如,假设以下代码片段 代码被执行的反对 ClassPathXmlApplicationContext 实例:

```
Resource template = ctx.getResource("some/resource/path/myTemplate.txt");
```

返回的是什么会 ClassPathResource ,如果同样的方法被执行死刑 反对 FileSystemXmlApplicationContext 实例, 你会得到一个 FileSystemResource 。 对于一个 WebApplicationContext ,你会得到一个 ServletContextResource ,等等。

这样,你可以加载资源在一个时尚的适合 特定的应用程序上下文。

另一方面,你也可能迫使 ClassPathResource 被使用,不管 应用程序上下文类型,通过指定特殊 类路径: 前缀:

```
Resource template = ctx.getResource("classpath:some/resource/path/myTemplate.txt");
```

同样,一个人可以强制 UrlResource 是 使用指定的任何标准 java净url 前缀:

```
Resource template = ctx.getResource("file:/some/resource/path/myTemplate.txt");
```

```
Resource template = ctx.getResource("http://myhost.com/resource/path/myTemplate.txt");
```

下面的表总结了战略转换 字符串 年代 资源 年代:

6.1为多。 一个资源字符串

前缀	例子	解释
类路径:	类路径:com/myapp/config.xml	从类路径加载。
文件:	文件:/数据/ config . xml	加载 url ,从 文件系统。 [1]
http:	http://myserver/logo.png	加载 url 。
(无)	/数据/ config . xml	取决于底层 ApplicationContext 。

[1] 但看到也 SectionA 6 7 3,一个 FileSystemResource caveatsa 。

6.5一个的 ResourceLoaderAware 接口

这个 ResourceLoaderAware 接口是 一个特殊的标记接口,确定对象,期望提供 与 ResourceLoader 参考。

```
public interface ResourceLoaderAware {

  void setResourceLoader(ResourceLoader resourceLoader);
}
```

当一个类实现 ResourceLoaderAware 并部署到一个 应用程序上下文(它作为spring托管bean),这是公认的 ResourceLoaderAware 由应用程序 上下文。 应用程序上下文将调用 setResourceLoader(ResourceLoader) ,提供 本身作为参数(记住,所有应用程序上下文在春天 实现 ResourceLoader 接口)。

当然,因为一个 ApplicationContext 是 ResourceLoader ,该bean也可以 实现 ApplicationContextAware 界面和使用提供的应用程序上下文直接加载 资源,但一般来说,最好是使用专业的 ResourceLoader 接口如果是所有 这是需要的。 代码就被耦合到资源加载 接口,它可以被认为是一个实用的界面,而不是整体 春天 ApplicationContext 接口。

在Spring 2.5中,您可以依赖的自动装配 ResourceLoader 作为一种替代 实施 ResourceLoaderAware 接口。 “传统” 构造函数 和 byType 自动装配模式(详见 SectionA 5 4 5,一个collaboratorsa自动装配) 现在能够提供一个依赖的类型 ResourceLoader 无论是一个 构造函数参数或setter方法参数分别。 更多的灵活性 (包括自动装配领域的能力和多个参数方法),可以考虑 使用新的基于注解的自动装配功能。 在这种情况下, ResourceLoader 将autowired的到一个领域, 构造函数参数或方法参数,是期待 ResourceLoader 类型只要字段, 构造函数或方法有问题 @ autowired 注释。 有关更多信息, 看到 SectionA 5 9 2,一个 @ autowired 一个 。

6.6一个 资源 作为依赖项

如果bean本身是要确定和供应资源 路径通过某种形式的动态过程,它可能是有意义的 bean使用 ResourceLoader 接口 加载资源。 考虑一个例子装载一个模板的一些 排序,具体资源需要取决于角色的 用户。 如果资源是静态的,它是有意义的,消除了使用的 ResourceLoader 界面完全, 和只有bean公开 资源 属性需要,预计他们将会注入到它。

是什么让它琐碎,然后注入这些属性,是所有 应用程序上下文注册和使用一个特殊的javabean 属性编辑器 可转换 字符串 路径 资源 对象。 所以如果 myBean 有一个模板属性的类型 资源 ,它可以配置一个 简单的字符串资源的,如下所示:

```
<bean id="myBean" class="...">
  <property name="template" value="some/resource/path/myTemplate.txt"/>
</bean>
```

注意,资源路径没有前缀,所以因为 应用程序上下文本身是要用作 ResourceLoader ,资源本身将 加载通过 ClassPathResource , FileSystemResource ,或 ServletContextResource (适当地) 根据具体类型的上下文。

如果有一个需要强迫一个特定的 资源 类型被使用,那么一个前缀可能 被使用。 以下两个例子展示如何强制 ClassPathResource 和一个 UrlResource (后者被用来访问 文件系统文件)。

```
<property name="template" value="classpath:some/resource/path/myTemplate.txt">
```

```
<property name="template" value="file:/some/resource/path/myTemplate.txt"/>
```

6.7一个应用程序上下文和 资源 路径

6.7.1A构建应用程序上下文

一个应用程序上下文构造函数(用于一个特定的应用程序 上下文类型)通常以一个字符串或字符串数组作为 位置路径(s)的资源(s)如XML文件构成 定义的上下文。

当这样一个位置路径没有前缀,具体 资源 类型由,路径和 用来加载bean定义,取决于和是适当的 特定的应用程序上下文。 例如,如果您创建一个 ClassPathXmlApplicationContext 如下:

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("conf/appContext.xml");
```

该bean定义将从类路径装入,作为一个 ClassPathResource 将 使用。 但如果你创建一个 FileSystemXmlApplicationContext 作为 如下:

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("conf/appContext.xml");
```

该bean定义将从文件系统加载的位置,在 这种情况下相对于当前工作目录。

注意,使用特殊的类路径前缀或标准 URL前缀的位置路径将会覆盖默认的类型的 资源 创建加载定义。 所以这 FileSystemXmlApplicationContext ...

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("classpath:conf/appContext.xml");
```

... 将会加载它的bean定义从类路径中。 然而,这仍然是一个 FileSystemXmlApplicationContext 。 如果它是 随后用作 ResourceLoader , 任何无前缀的路径仍将被作为文件系统路径。

构建 ClassPathXmlApplicationContext 实例——快捷键

这个 ClassPathXmlApplicationContext 公开了一个数量的构造函数实例化启用方便。 基本的想法是,一个供应仅仅包含字符串数组 刚刚的文件名的XML文件本身(没有领导 路径信息),和一个 也 供应一个 类 ; ClassPathXmlApplicationContext 将获得从提供的类路径信息。

一个例子将希望明确这一点。 考虑一个目录 布局看起来像这样:

```
com/
  foo/
    services.xml
    daos.xml
    MessengerService.class
```

一个 ClassPathXmlApplicationContext 实例 由bean中定义的 “services . xml” 和 “daos xml的 可以被实例化像 所以...

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
    new String[] {"services.xml", "daos.xml"}, MessengerService.class);
```

请做参考的javadoc ClassPathXmlApplicationContext 类 各种构造函数的细节。

6.7.2A通配符在应用程序上下文构造函数资源路径

在应用程序上下文的资源路径构造函数值可能 是一个简单的路径(如上所示),它有一个一对一的映射 目标资源,或交替可能包含特殊的 “classpath *: “ 前缀和/或内部ant是基于正则表达式匹配使用 春天的 PathMatcher 实用程序)。 后者的两 实际上是通配符

一个使用这种机制是通过组件样式时做 应用程序装配。 所有组件可以 “发布” 上下文定义 一个众所周知的位置路径片段,当最终的应用程序 上下文创建使用相同的路径前缀的通过 classpath *: ,所有的组件将被碎片 自动弹出。

注意,这个它是特定于使用资源的路径 应用程序上下文构造函数(或当使用 PathMatcher 实用程序类层次结构直接), 和解决在施工时间。 它没有关系 资源 类型本身。 这是不可能的 使用 classpath *: 建立实际的前缀 资源 作为一个资源点才 一次一个资源。

ant是基于模式

当路径位置包含一个ant是基于模式,例如:

```
/WEB-INF/*-context.xml
com/mycompany/**/applicationContext.xml
file:C:/some/path/*-context.xml
classpath:com/mycompany/**/applicationContext.xml
```

... 解析器是一个更为复杂的但定义过程 试图解决通配符。 它产生了一个资源的路径了 最后,得到非通配符段URL从它。 如果这个URL 不是一个“jar:“URL或特定容器变体(如。 “ 邮政编码: “在WebLogic。” wsjar “在 WebSphere等等),那么一个java输入输出文件 是 得到它,用来解决通配符的遍历 文件系统。 对于一个jar URL,解析器可以获取一个java.net.JarURLConnection 从它或手动 解析jar URL,然后遍历jar文件的内容 解决通配符。

影响可移植性

如果指定的路径已经是一个文件的URL(要么 显式或隐式地因为基地 ResourceLoader 是 文件系统,那么它是保证工作 完全便携时尚。

如果类路径中指定的路径是一个位置,那么 解析器必须获得最后非通配符路径段URL通过 classloader getresource() 调用。 因为这 只是一个节点的路径(而不是文件结束时),它实际上是吗 未定义(在 类加载器 javadoc) 到底什么样的一个URL返回在这种情况下。 在实践中,它 始终是一个 java输入输出文件 代表 目录路径资源可以解析为一个文件系统 位置,或者一个jar URL的某种资源的类路径 解析为一罐的位置。 不过,有一个可移植性关注 这个操作。

如果一个jar URL获得过去非通配符段, 这个解析器必须能够得到一个 java.net.JarURLConnection 从它,或 手动解析jar URL,能够行走的内容 jar和解决通配符。 这将工作在大多数环境中, 但会失败在别人,强烈建议 通配符解决资源来自jar被彻底 测试您的特定环境在你依赖它。

这个 classpath *: 前缀

当构建一个基于xml的应用程序上下文,一个位置 字符串可以使用特殊的 classpath *: 前缀:

```
ApplicationContext ctx =
    new ClassPathXmlApplicationContext("classpath*:conf/appContext.xml");
```

这种特殊的前缀指定类路径中所有资源 匹配给定的名称必须获得(在内部,这本质上 发生通过 ClassLoader.getResources(...) 调用),然后合并以形成最终的应用程序上下文 定义。

> **Classpath *:可移植性**
>
> 通配符类路径依赖 getResources() 方法 潜在的类加载器。 因为大多数应用服务器现在供应 他们自己的类加载器实现的,这种行为可能会有所不同 尤其是在处理jar文件。 一个简单的测试来检查 classpath * 作品是使用类加载器来加载一个文件 在一个jar的类路径: .getClassLoader getClass()().getResources(“< someFileInsideTheJar >”) 。 试试这个测试的文件具有相同的名字,但被放置 在两个不同的位置。 如果一个不合适的结果 回来,检查应用程序服务器文档设置 这可能会影响行为的类加载器。

“ classpath *: “前缀也可以结合 与 PathMatcher 模式在其余的位置路径,对于 例 “ classpath *:meta - inf / *豆xml ” 。 在这个 情况下,分辨率的策略很简单:一个 ClassLoader.getResources()调用是使用在过去的非通配符路径 段获得所有匹配的资源的类装入器 层次结构,然后从每个资源相同的PathMatcher分辨率 上面描述的策略用于通配符子路径。

其他笔记有关通配符

请注意, “ classpath *: “当 结合ant是基于模式只会工作可靠,至少 一个根目录的模式开始之前,除非实际的目标 文件驻留在文件系统。 这意味着模式像 “ classpath *:* . xml “不会检索文件 根的jar文件,而仅仅从根的扩大 目录。 这源于一个限制在JDK的 ClassLoader.getResources() 方法只有 返回文件系统的位置对于一个传入空字符串(指示 潜在的根搜索)。

ant是基于模式” 类路径: “ 资源是不能保证找到匹配的资源如果根 包来搜索可在多个类路径位置。 这 是因为资源如

```
com/mycompany/package1/service-context.xml
```

可能只在一个位置,但是当一个路径如

```
classpath:com/mycompany/**/service-context.xml
```

用于尝试解决它,解析器将工作了(第一次)的URL 返回 "com/mycompany getResource(") ;。 如果 这个基础包节点存在于多个类加载器的位置, 实际资源可能不是下面结束。 因此,最好是用 " classpath *: "与同一ant是基于模式 这种情况下,它将搜索所有的类路径位置,包含 根包。

6.7.3A FileSystemResource 警告

一个 FileSystemResource 这不是附加 一个 FileSystemApplicationContext (即一个 FileSystemApplicationContext 不是实际的 ResourceLoader)将治疗绝对vs。 正如你期望的相对路径。 相对路径是相对的 当前工作目录,而绝对路径指的是相对于根 的文件系统。

为向后兼容(历史)原因,然而,这 变化当 FileSystemApplicationContext 是 这个 ResourceLoader 。 这个 FileSystemApplicationContext 只是强迫所有 附加 FileSystemResource 实例来治疗 所有的位置路径为相对,他们是否从一个领先的削减 或不是。 在实践中,这意味着以下是等价的:

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("conf/context.xml");
```

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("/conf/context.xml");
```

像以下:(尽管它会让他们感觉 是不同的,作为一个案例是相对的和其他绝对。)

```
FileSystemXmlApplicationContext ctx = ...;
ctx.getResource("some/resource/path/myTemplate.txt");
```

```
FileSystemXmlApplicationContext ctx = ...;
ctx.getResource("/some/resource/path/myTemplate.txt");
```

在实践中,如果真正的绝对路径,它需要的文件系统是 更好的放弃使用绝对路径与 FileSystemResource / FileSystemXmlApplicationContext ,只是力量 使用一个 UrlResource ,通过使用 文件: URL前缀。

```
// actual context type doesn't matter, the Resource will always be UrlResource
ctx.getResource("file:/some/resource/path/myTemplate.txt");
```

```
// force this FileSystemXmlApplicationContext to load its definition via a UrlResource
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("file:/conf/context.xml");
```

7。 一个验证、数据绑定、类型转换

7.1一个介绍

有优点和缺点考虑验证业务逻辑, 和弹簧提供了一个设计验证(和数据绑定),不 排除或其中之一。 具体验证不应该绑定到 web层,应该很容易定位,它应该有可能塞 在任何验证器可用。 考虑到上述,春天已经来了 一个 验证器 界面,都是基本的 非常有用的在每层细节的一个应用程序。

数据绑定是有用的因为允许用户输入是动态绑定 领域模型的一个应用程序(或者任何你使用的对象 处理用户输入)。 Spring提供了所谓的 DataBinder 就这么干吧。 这个 验证器 和 DataBinder 弥补 验证 包,它主要是用于但不 限于MVC框架。

这个 BeanWrapper 是一项基本 在Spring框架和概念用于很多地方。 然而, 你可能不会有需要使用 BeanWrapper 直接。 因为这是 参考文档不过,我们认为一些解释可能在 订单。 我们将解释 BeanWrapper 在 这一章,因为如果你要使用它,你会最 可能这样做当试图将数据绑定到对象。

jsr - 303 Bean验证

Spring框架支持jsr - 303 Bean验证调整 它春天的 验证器 接口。

一个应用程序可以选择支持jsr - 303 Bean验证一旦在全球范围内, 中描述的 SectionA 7.8,一个弹簧3 Validationa ,使用它 专门为所有验证需求。

应用程序也可以注册 额外的春天 验证器 实例每 DataBinder 实例中描述的那样 SectionA 7 8 3,一个配置一个DataBindera 。 这可能是有用的 插入验证逻辑没有注释的使用。

春天的DataBinder和低层BeanWrapper都使用 PropertyEditors解析和格式化属性值。 这个 属性编辑器 概念的一部分 JavaBeans规范,也是本章中解释。 弹簧3 介绍了一种 "核心。 转换" 方案,提供了一个通用类型 转换工具,以及一个更高层次的 "格式" 包 格式化UI字段值。 这些新的包可能被用作简单的 PropertyEditors替代品,也将讨论在这 一章。

7.2验证使用Spring的 验证器 接口

弹簧特性一个 验证器 接口 你可以使用验证对象。 这个 验证器 接口工作使用一个 错误 因此,在验证对象, 验证器可以报告验证失败 错误 对象。

让我们考虑一个小数据对象:

```
public class Person {

  private String name;
  private int age;

  // the usual getters and setters...
}
```

我们将提供验证行为 人 类通过实现以下两个 方法 org.springframework.validation.Validator 接口:

- 支持(类) ——这 验证器 验证的实例 提供 类 吗?
- 验证(对象, org.springframework.validation.Errors) -验证 给定对象和案例验证错误,注册那些 给定的 错误 对象

实现 验证器 相当 简单,尤其是当你知道的 ValidationUtils 辅助类,春天 框架还提供了。

```
public class PersonValidator implements Validator {

    /**
    * This Validator validates *just* Person instances
    */
    public boolean supports(Class clazz) {
        return Person.class.equals(clazz);
    }

    public void validate(Object obj, Errors e) {
        ValidationUtils.rejectIfEmpty(e, "name", "name.empty");
        Person p = (Person) obj;
        if (p.getAge() < 0) {
            e.rejectValue("age", "negativevalue");
        } else if (p.getAge() > 110) {
            e.rejectValue("age", "too.darn.old");
        }
    }
}
```

正如您可以看到的, 静态 rejectIfEmpty(. .) 方法 ValidationUtils 类是用来拒绝 ' name ' 财产如果是 空 或 空字符串。 看看Javadoc的吗 ValidationUtils 类来看看功能它 除了前面所示的例子了。

虽然你可以实现一个单 验证器 类来验证每个 嵌套对象在一个富裕的对象,它可能是更好的封装 验证逻辑对于每个嵌套类的对象在它自己的 验证器 实现。 一个简单的例子 的 “富有” 对象将是一个 客户 这是由两种 字符串 属性(第一和第二名)和一个复杂 地址 对象。 地址 对象可能是独立使用的 客户 对象,所以一个截然不同的 AddressValidator 已经实现了。 如果你想要你 CustomerValidator 重用逻辑包含 在 AddressValidator 类不通过 去复制粘贴,你可以依赖注入或实例化 AddressValidator 在你 CustomerValidator ,使用它像这样:

```
public class CustomerValidator implements Validator {

    private final Validator addressValidator;

    public CustomerValidator(Validator addressValidator) {
        if (addressValidator == null) {
            throw new IllegalArgumentException(
              "The supplied [Validator] is required and must not be null.");
        }
        if (!addressValidator.supports(Address.class)) {
            throw new IllegalArgumentException(
              "The supplied [Validator] must support the validation of [Address] instances.");
        }
        this.addressValidator = addressValidator;
    }

    /**
    * This Validator validates Customer instances, and any subclasses of Customer too
    */
    public boolean supports(Class clazz) {
        return Customer.class.isAssignableFrom(clazz);
    }

    public void validate(Object target, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName", "field.required");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "surname", "field.required");
        Customer customer = (Customer) target;
```

```
        try {
            errors.pushNestedPath("address");
            ValidationUtils.invokeValidator(this.addressValidator, customer.getAddress(), errors);
        } finally {
            errors.popNestedPath();
        }
    }
}
```

验证错误报告 错误 对象传递到验证器。 在 Spring Web MVC的情况下你可以使用 <春天:bind / > 标签检查错误消息,当然你也可以检查 错误对象自己。 更多的信息关于这些方法它提供了可以 被发现从Javadoc。

7.3一个解决代码错误消息

我们讨论关于数据绑定和验证。 输出消息 相应的验证错误的最后一件事就是我们需要讨论。 在我们的示例所示,我们拒绝了名称 和 年龄 场。 如果我们要输出误差 消息通过使用 MessageSource ,我们将 使用错误代码我们给当拒绝字段(' name ' 和 “年龄” 在这种情况下)。 当你调用(无论是直接或间接, 使用例如 ValidationUtils 类) rejectValue 或另一个 拒绝 方法 错误接口,底层 实现将不仅注册代码,你已经通过了,但也 许多额外的错误代码。 这是什么错误代码注册 取决于 MessageCodesResolver 这 是使用。 默认情况下, DefaultMessageCodesResolver 使用,对吗 例子不仅注册消息的代码你给,但也 信息,包括字段名你传递给拒绝的方法。 所以 如果你拒绝一个字段使用 rejectValue(“时代” , “太老”) ,除了 太老 代码,春天也将寄存器 太老 和 太老的年龄智力 (第一个将包括 字段名,第二个将包括字段类型);这是 作为一个方便的辅助开发人员针对错误消息和 诸如此类的。

更多信息 MessageCodesResolver 和默认 策略可以发现网上的Javadocs为 MessageCodesResolver 和 DefaultMessageCodesResolver 分别。

7.4一个Bean操纵和 BeanWrapper

这个 org.springframework.beans 包坚持 JavaBeans标准提供的太阳。 是一个简单的JavaBean类 一个默认的无参数的构造函数,它遵循命名约定, (通过一个例子)属性命名 bingoMadness 会有一个setter方法 setBingoMadness(. .) 和一个getter方法 getBingoMadness() 。 更多 JavaBeans和规范的信息,请参考太阳的 网站(java.sun.com/products/javabeans)。

一个非常重要的类在bean包是 BeanWrapper 接口和相应的 实现(BeanWrapperImpl)。 作为引用 Javadoc, BeanWrapper 提供 功能设置和获取属性值(单独或批量), 得到属性描述符,并查询属性,以确定它们是否 可读或写。 同时, BeanWrapper 提供了支持嵌套的属性,使设置的属性 在子属性到一个无限的深度。 然后, BeanWrapper 支持能够添加 标准JavaBeans PropertyChangeListeners 和 VetoableChangeListeners ,而不 需要支持的代码在目标类。 最后但并非最不重要, BeanWrapper 提供支持 设置索引属性。 这个 BeanWrapper 通常不使用 应用程序代码直接,而是由 DataBinder 和 BeanFactory 。

的方式 BeanWrapper 作品部分 显示它的名字: 它封装了一个bean 执行 上的动作,bean,如设置和获取属性。

7.4.1A设置和获取基本和嵌套的属性

设置和获取属性使用 setPropertyValue(s) 和 getPropertyValue(s) 方法,都有一个 两个重载的变体。 他们都是更详细地描述 Javadoc春天伴随着。 重要的是要知道,那里 是几个约定指示对象的属性。 一个 几个例子:

7.1为多。 一个属性的范例

表达式	解释
名称	表明房地产 名称 对应方法 getName() 或 isName() 和 setName(. .)
帐户名称	表明嵌套的属性 名称 的 房地产 帐户 相应的如 方法 .setName getAccount()() 或 getAccount(). getname()
帐户[2]	表示 第三 元素的 索引属性 帐户 。 索引属性 类型可以是 数组 , 列表 或其他 自然有序 收集
帐户(公司名称)	显示值的映射条目索引键 公司名称 地图的属性 帐户

下面你会发现一些例子的使用 BeanWrapper 获取和设置 属性。

(下一节不是至关重要的,你如果 你不打算工作 BeanWrapper 直接。 如果你只是 使用 DataBinder 和 BeanFactory 和他们的开箱即用的 实现,你应该跳至有关部分 PropertyEditors)。

考虑以下两类:

```
public class Company {
    private String name;
    private Employee managingDirector;

    public String getName() {
        return this.name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Employee getManagingDirector() {
        return this.managingDirector;
    }
    public void setManagingDirector(Employee managingDirector) {
        this.managingDirector = managingDirector;
    }
}
```

```
public class Employee {
    private String name;
    private float salary;

    public String getName() {
        return this.name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public float getSalary() {
        return salary;
    }
    public void setSalary(float salary) {
        this.salary = salary;
    }
}
```

下面的代码片段显示了一些示例,演示如何检索 和操作的一些属性实例化 公司 和 员工 :

```
BeanWrapper company = BeanWrapperImpl(new Company());
// setting the company name..
company.setPropertyValue("name", "Some Company Inc.");
// ... can also be done like this:
PropertyValue value = new PropertyValue("name", "Some Company Inc.");
company.setPropertyValue(value);

// ok, let's create the director and tie it to the company:
BeanWrapper jim = BeanWrapperImpl(new Employee());
jim.setPropertyValue("name", "Jim Stravinsky");
company.setPropertyValue("managingDirector", jim.getWrappedInstance());

// retrieving the salary of the managingDirector through the company
Float salary = (Float) company.getPropertyValue("managingDirector.salary");
```

7.4.2A内置 属性编辑器 实现

弹簧使用的概念 PropertyEditors 到 效应之间的转换一个 对象 和一个 字符串 。 如果你仔细想想,它有时可能 是方便能够以不同的方式表示属性比 对象本身。 例如,一个 日期 可以 代表在一个人类可读的方式(如 字符串 ” 2007-14-09 ”),而 我们仍然能够将人类可读的形式回到最初 日期(或甚至更好:转换任何日期中输入一个人类可读的形式, 回到 日期 对象)。 这种行为可以通过 注册自定义的编辑器 的类型 java bean属性编辑器 。 注册 定制编辑器在一个 BeanWrapper 或 在一个特定的IoC容器交替就像前面提到过的 章,给它的知识如何转换属性 所需的类型。 阅读更多关于 PropertyEditors 在的Javadoc java bean 包提供的太阳。

几个例子,在春天使用财产编辑:

- 设置属性在bean 是通过使用 PropertyEditors 。 当提及 以 作为一个房地产的价值 一些豆你宣布在XML文件中,弹簧将(如果setter 相应的属性 类 参数)使用 ClassEditor 试图解决这一参数 一个 类 对象。
- 解析HTTP请求参数 在春天的 MVC框架是通过使用各种各样的 PropertyEditors 你可以手动绑定在所有 子类的 CommandController 。

春天有很多内置的 PropertyEditors 让生活简单。 每一个是下面列出的,他们都是 位于 org.springframework.beans.propertyeditors 包。 大部分,但不是全部(如上下文),默认注册通过 BeanWrapperImpl 。 在属

性编辑器是什么 可配置在一些时尚,你可以注册自己的课程仍然 变体来覆盖默认的一个:

为多7 2一个内置 PropertyEditors

类	解释
ByteArrayPropertyEditor	编辑字节数组。 只会被转换成字符串 对应的字节表示。 默认注册 通过 BeanWrapperImpl 。
ClassEditor	解析字符串代表实际的类和类 倒过来。 当一个类是没有找到,一个 IllegalArgumentException 抛出。 默认注册由 BeanWrapperImpl 。
CustomBooleanEditor	定制属性编辑器 布尔 属性。 默认注册 通过 BeanWrapperImpl ,但是,可以 被注册自定义的实例,它作为自定义 编辑器。
CustomCollectionEditor	属性编辑器集合,将任何源 收集 对于一个给定的目标 收集 类型。
CustomDateEditor	可定制的属性编辑java跑龙套。 目前为止,支持 一个自定义DateFormat。 没有注册的默认情况下。 必须用户 注册需要以适当的格式。
CustomNumberEditor	可定制的属性编辑任意数量子类像 整数 , 长 , 浮 , 双 。 默认注册由 BeanWrapperImpl , 但可以被注册自定义的实例,它作为一个 自定义编辑器。
FileEditor	能够解决字符串 java输入输出文件 对象。 注册的 违约 BeanWrapperImpl 。
InputStreamEditor	单向属性编辑器,可以将一个文本字符串 和生产(通过一个中间 ResourceEditor 和 资源) 一个 InputStream ,所以 InputStream 属性可能是 直接设置为字符串。 注意,默认使用 不会 关闭 InputStream 为你! 默认注册由 BeanWrapperImpl 。
LocaleEditor	能够解决字符串 地区 对象,反之亦然(字符串 格式是(语言)_(国家)_(变种),这是相同的 事 toString()方法的现场提供)。 注册的 违约 BeanWrapperImpl 。
PatternEditor	能够解决字符串到JDK 1.5 模式 对象,反之亦然。
PropertiesEditor	能够转换字符串(格式化使用格式 为定义在java . lang的Javadoc。 属性类) 属性 对象。 默认注册 通过 BeanWrapperImpl 。
StringTrimmerEditor	属性编辑器,阀内件的字符串。 还允许 一个空字符串转换成一个 空 值。 不是注册在默认情况下,用户必须注册为 需要。
URLEditor	能够解决一个URL的字符串表示的 实际 url 对象。 默认注册 通过 BeanWrapperImpl 。

弹簧使用 java.beans.PropertyEditorManager 设置 搜索路径属性编辑,他们可能需要。 搜索 路径还包括 太阳bean编辑 ,其中包括 属性编辑器 实现类型 如 字体 , 颜色 ,和 大部分的原始类型。 还要注意标准JavaBeans 基础设施将自动发现 属性编辑器 类(没有你 需要注册他们明确)如果他们在同一个包 他们处理的类,这个类的名称相同, “编辑器” 附加的;例如,一个可能的下面的类和包的结构,这就足够了 FooEditor 类被认可和使用的 属性编辑器 对于 foo 类型属性。

```
com
  chank
    pop
      Foo
      FooEditor   // the PropertyEditor for the Foo class
```

请注意,您还可以使用标准的 BeanInfo JavaBeans机制在这里 (描述 在这里没有惊人的细节)。 下面的示例使用找到的 BeanInfo 机制来明确 注册一个或多个 属性编辑器 实例与属性相关的类。

```
com
  chank
    pop
      Foo
      FooBeanInfo   // the BeanInfo for the Foo class
```

这是Java源代码引用 FooBeanInfo 类。 这将关联 CustomNumberEditor 与 年龄 财产的 foo 类。

```
public class FooBeanInfo extends SimpleBeanInfo {

    public PropertyDescriptor[] getPropertyDescriptors() {
        try {
            final PropertyEditor numberPE = new CustomNumberEditor(Integer.class, true);
```

```
            PropertyDescriptor ageDescriptor = new PropertyDescriptor("age", Foo.class) {
                public PropertyEditor createPropertyEditor(Object bean) {
                    return numberPE;
                };
            };
            return new PropertyDescriptor[] { ageDescriptor };
        }
        catch (IntrospectionException ex) {
            throw new Error(ex.toString());
        }
    }
}
```

注册附加自定义 PropertyEditors

当bean属性设置为一个字符串值,一个Spring IoC 集装箱最终使用标准JavaBeans PropertyEditors 将这些字符串的 复杂类型的属性。 弹簧预注册一个数量的定制 PropertyEditors (例如,将一个 类名表示为一个字符串变成一个真正的 类 对象)。 此外,Java的标准 JavaBeans 属性编辑器 查找 机制允许一个 属性编辑器 对于一个类 简单地命名为适当地放置在相同的包 类提供了支持,自动发现。

如果有一个需要注册其他自定义 PropertyEditors 有几种机制 可用。 最手工方式,这是通常不方便 或推荐的,是直接使用 registerCustomEditor() 方法 ConfigurableBeanFactory 界面, 假设你有一个 BeanFactory 参考。 另一个,稍微方便,机制是使用 特殊bean工厂后处理器称为 CustomEditorConfigurer 。 尽管bean工厂 后处理器可以使用 BeanFactory 的实现, CustomEditorConfigurer 有一个嵌套的属性 设置,所以强烈建议使用它的 ApplicationContext ,它可能是 在类似的方式部署到任何其他bean,并自动 发现和应用。

请注意,所有bean工厂和应用程序上下文 自动使用大量的内置属性编辑器,通过他们的 使用一种叫 BeanWrapper 办理产权转换。 标准属性编辑, BeanWrapper 寄存器中列出 前一节 。 此外, ApplicationContexts 还覆盖或 添加一个额外的数量的编辑处理资源查找的 的方式适当的特定应用程序上下文类型。

标准JavaBeans 属性编辑器 实例被用来转换属性值表示为字符串 实际的复杂类型的属性。 CustomEditorConfigurer ,一个bean工厂 后处理器,可以用来方便地添加额外的支持 属性编辑器 实例一个 ApplicationContext 。

考虑一个用户类 ExoticType ,和 另一个类 DependsOnExoticType 需要 ExoticType 设置为一个属性:

```
package example;

public class ExoticType {

    private String name;

    public ExoticType(String name) {
        this.name = name;
    }
}

public class DependsOnExoticType {

    private ExoticType type;

    public void setType(ExoticType type) {
        this.type = type;
    }
}
```

当事情是适当的设置,我们希望能够分配 类型属性作为一个字符串,它一个 属性编辑器 将在幕后 转换成一个实际的 ExoticType 实例:

```
<bean id="sample" class="example.DependsOnExoticType">
    <property name="type" value="aNameForExoticType"/>
</bean>
```

这个 属性编辑器 实现 可能类似于:

```
// converts string representation to ExoticType object
package example;

public class ExoticTypeEditor extends PropertyEditorSupport {

    public void setAsText(String text) {
        setValue(new ExoticType(text.toUpperCase()));
    }
}
```

最后,我们使用 CustomEditorConfigurer 到 注册这个新 属性编辑器 与 这个 ApplicationContext ,然后 可以根据需要使用它:

```
<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
 <property name="customEditors">
  <map>
   <entry key="example.ExoticType" value="example.ExoticTypeEditor"/>
  </map>
 </property>
</bean>
```

使用 PropertyEditorRegistrars

另一个机制的财产登记的编辑 Spring容器是创建和使用 PropertyEditorRegistrar 。 这 接口是特别有用,当你需要使用相同的设置 属性编辑器的几种不同情况:写一个 相应的注册和重用,在每种情况下。 PropertyEditorRegistrars 工作结合 一个接口称为 PropertyEditorRegistry ,一个接口 这是实现的春天 BeanWrapper (和 DataBinder)。 PropertyEditorRegistrars 尤其 当结合使用方便的 CustomEditorConfigurer (介绍 这里),它公开了一个属性,名为 setPropertyEditorRegistrars(. .) : PropertyEditorRegistrars 添加到 CustomEditorConfigurer 在这个时尚可以 容易共享与 DataBinder 和 Spring MVC 控制器 。 此外, 它避免了需要同步在定制编辑器:一个 PropertyEditorRegistrar 预计 创建新的 属性编辑器 每个bean创建实例尝试。

使用 PropertyEditorRegistrar 也许是最好的为例进行了阐述。 首先,你需要 创建自己的 PropertyEditorRegistrar 实现:

```
package com.foo.editors.spring;

public final class CustomPropertyEditorRegistrar implements PropertyEditorRegistrar {

    public void registerCustomEditors(PropertyEditorRegistry registry) {

        // it is expected that new PropertyEditor instances are created
        registry.registerCustomEditor(ExoticType.class, new ExoticTypeEditor());

        // you could register as many custom property editors as are required here...
    }
}
```

也看见了 org.springframework.beans.support.ResourceEditorRegistrar 例如 PropertyEditorRegistrar 实现。 请注意,在其实现的 registerCustomEditors(. .) 方法它创建 每个属性编辑器的新实例。

接下来我们配置一个 CustomEditorConfigurer 和注入一个实例 我们的 CustomPropertyEditorRegistrar 进 它:

```
<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="propertyEditorRegistrars">
        <list>
            <ref bean="customPropertyEditorRegistrar"/>
        </list>
    </property>
</bean>

<bean id="customPropertyEditorRegistrar"
     class="com.foo.editors.spring.CustomPropertyEditorRegistrar"/>
```

最后,在一点离开的焦点 一章,对于那些使用 Spring的MVC web 框架 ,使用 PropertyEditorRegistrars 在 结合数据绑定 控制器 (如 SimpleFormController)可以非常方便的。 找到下面的示例使用一个 PropertyEditorRegistrar 在 实现的一 initBinder(. .) 方法:

```
public final class RegisterUserController extends SimpleFormController {

    private final PropertyEditorRegistrar customPropertyEditorRegistrar;

    public RegisterUserController(PropertyEditorRegistrar propertyEditorRegistrar) {
        this.customPropertyEditorRegistrar = propertyEditorRegistrar;
    }

    protected void initBinder(HttpServletRequest request, ServletRequestDataBinder binder)
                    throws Exception {
        this.customPropertyEditorRegistrar.registerCustomEditors(binder);
    }

    // other methods to do with registering a User
}
```

这种风格的 属性编辑器 注册可以导致简洁代码(实施 initBinder(. .) 只是一个线长!),然后呢 允许普通 属性编辑器 登记代码封装在一个类,然后共享 在尽可能多的 控制器 作为 需要。

7.5一个弹簧3类型转换

弹簧3引入了一种 核心转换 包, 提供了一个通用类型转换系统。 该系统定义了一个SPI来 执行类型转换逻辑,以及一个API来执行类型 在运行时转换。 在Spring容器,这个系统可以使用 作为一个替代PropertyEditors将外部化bean属性 值字符串所需的属性类型。 公共API也可以使用 在您的应用程序的任何地方,类型转换是必要的。

7.5.1A转换器SPI

SPI的实现类型转换的逻辑是简单的和强烈 类型:

```
package org.springframework.core.convert.converter;

public interface Converter<S, T> {

    T convert(S source);

}
```

创建你自己的转换器,只是上面实现接口。 参数化的类型你年代来回转换,和T作为类型 你是皈依。 为每次调用转换(S),源参数保证不空。 你的转换器可以抛出任何异常 转换失败。 一个IllegalArgumentException应该被扔到报告 一个无效的源值。 照顾,以确保你的转换器 实现线程安全的。

数转换器的实现提供了 核心转换支持 包作为一个方便。 这些包括转换器从字符串到数字和其他常见的类型。 考虑 StringToInteger 作为一个示例转换器 实现:

```
package org.springframework.core.convert.support;

final class StringToInteger implements Converter<String, Integer> {

    public Integer convert(String source) {
        return Integer.valueOf(source);
    }

}
```

7.5.2A ConverterFactory

当你需要集中转换逻辑为整个 类层次结构,例如,当从字符串转换 java . lang。 枚举对象,实现 ConverterFactory :

```
package org.springframework.core.convert.converter;

public interface ConverterFactory<S, R> {

    <T extends R> Converter<S, T> getConverter(Class<T> targetType);

}
```

参数化年代是要转换的类型从和R 基础类型定义 范围 类你可以 皈依。 然后实现getConverter(类< T >),T是一个 子类的R。

考虑 StringToEnum ConverterFactory 作为一个例子:

```
package org.springframework.core.convert.support;

final class StringToEnumConverterFactory implements ConverterFactory<String, Enum> {

    public <T extends Enum> Converter<String, T> getConverter(Class<T> targetType) {
        return new StringToEnumConverter(targetType);
    }

    private final class StringToEnumConverter<T extends Enum> implements Converter<String, T> {

        private Class<T> enumType;

        public StringToEnumConverter(Class<T> enumType) {
            this.enumType = enumType;
        }

        public T convert(String source) {
            return (T) Enum.valueOf(this.enumType, source.trim());
        }
    }
}
```

7.5.3A GenericConverter

当您需要一个复杂的转换实现,考虑 这个GenericConverter接口。 与一个更灵活但不强烈 类型的签名,一个GenericConverter 支持转换之间的多 源和目标类型。 此外,一个GenericConverter使可用 源和目标字段上下文可以使用在实现你的 转换逻辑。 这种上下文允许一个类型转换来推动 一个字段注释,或泛型信息上声明一个字段 签名。

```
package org.springframework.core.convert.converter;

public interface GenericConverter {

    public Set<ConvertiblePair> getConvertibleTypes();

    Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType);

}
```

实现一个GenericConverter,有getConvertibleTypes()返回 支持的源- >目标类型配对。 然后实现 转换(对象,TypeDescriptor,TypeDescriptor)来实现你的 转换逻辑。 源TypeDescriptor提供访问 源字段值被转换着。 目标TypeDescriptor提供访问目标字段转换后 值将被设置。

A good example of a GenericConverter is a converter that converts between a Java Array and a Collection. Such an ArrayToCollectionConverter introspects the field that declares the target Collection type to resolve the Collection's element type. This allows each element in the source array to be converted to the Collection element type before the Collection is set on the target field.

Note

Because GenericConverter is a more complex SPI interface, only use it when you need it. Favor Converter or ConverterFactory for basic type conversion needs.

ConditionalGenericConverter

Sometimes you only want a Converter to execute if a specific condition holds true. For example, you might only want to execute a Converter if a specific annotation is present on the target field. Or you might only want to execute a Converter if a specific method, such as static valueOf method, is defined on the target class. ConditionalGenericConverter is an subinterface of GenericConverter that allows you to define such custom matching criteria:

```
public interface ConditionalGenericConverter extends GenericConverter {

    boolean matches(TypeDescriptor sourceType, TypeDescriptor targetType);

}
```

A good example of a ConditionalGenericConverter is an EntityConverter that converts between an persistent entity identifier and an entity reference. Such a EntityConverter might only match if the target entity type declares a static finder method e.g. findAccount(Long). You would perform such a finder method check in the implementation of matches(TypeDescriptor, TypeDescriptor).

7.5.4 ConversionService API

The ConversionService defines a unified API for executing type conversion logic at runtime. Converters are often executed behind this facade interface:

```
package org.springframework.core.convert;

public interface ConversionService {

    boolean canConvert(Class<?> sourceType, Class<?> targetType);

    <T> T convert(Object source, Class<T> targetType);

    boolean canConvert(TypeDescriptor sourceType, TypeDescriptor targetType);

    Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType);

}
```

Most ConversionService implementations also implement ConverterRegistry, which provides an SPI for registering converters. Internally, a ConversionService implementation delegates to its registered converters to carry out type

conversion logic.

A robust ConversionService implementation is provided in the core.convert.support package. GenericConversionService is the general-purpose implementation suitable for use in most environments. ConversionServiceFactory provides a convenient factory for creating common ConversionService configurations.

7.5.5 Configuring a ConversionService

A ConversionService is a stateless object designed to be instantiated at application startup, then shared between multiple threads. In a Spring application, you typically configure a ConversionService instance per Spring container (or ApplicationContext). That ConversionService will be picked up by Spring and then used whenever a type conversion needs to be performed by the framework. You may also inject this ConversionService into any of your beans and invoke it directly.

> **注意**
>
> 如果没有ConversionService登记与春天,原来的 基于属性编辑器使用系统。

注册一个默认的ConversionService与春天,添加 以下的bean定义id conversionService :

```
<bean id="conversionService"
    class="org.springframework.context.support.ConversionServiceFactoryBean"/>
```

一个默认的ConversionService之间可以转换字符串、数字、 枚举、集合、地图和其他常见类型。 补充或 覆盖默认的转换器使用您自己的自定义转换器(s),设置 这个 转换器 财产。 属性值可以实现 这两个转换器,ConverterFactory,或GenericConverter 接口。

```
<bean id="conversionService"
    class="org.springframework.context.support.ConversionServiceFactoryBean">
  <property name="converters">
    <list>
      <bean class="example.MyCustomConverter"/>
    </list>
  </property>
</bean>
```

它也是常见的使用ConversionService在Spring MVC 应用程序。 看到 SectionA 7 6 5,一个配置格式在春天MVCa 有关使用 < mvc:注解驱动/ > 。

在某些情况下,您可能希望在应用格式 转换。 看到 SectionA 7 6 3,一个FormatterRegistry SPIa 对于 细节使用FormattingConversionServiceFactoryBean 。

7.5.6A使用ConversionService编程方式

工作ConversionService实例编程,简单 注入一个参考就像任何其他bean:

```
@Service
public class MyService {

    @Autowired
    public MyService(ConversionService conversionService) {
        this.conversionService = conversionService;
    }

    public void doIt() {
        this.conversionService.convert(...)
    }
}
```

7.6一个弹簧3字段格式

正如前面讨论的, 核心转换 是一个通用的类型 转换系统。 它提供了一个统一的ConversionService API以及 一个强类型的转换器SPI实现转换逻辑从一个 类型到另一个。 Spring容器使用这个系统来绑定bean属性 值。 此外,两个弹簧表达式语言(?)和 使用这个系统来绑定DataBinder字段值。 例如,当? 需要强迫一个 短 一个 长 完成一个 表达式。 setValue(对象豆、对象值) 尝试,核心。 执行强制转换系统。

现在考虑一个典型的类型转换需求客户 环境,如web和桌面应用程序。 在这样的环境中, 你通常把 从字符串 支持 客户端回传的

过程,以及背 到字符串 到 支持视图渲染过程。 此外,你经常需要 本地化的字符串值。 更普遍 核心转换 转换器SPI不解决这些格式化 直接要求。 直接地址,弹簧3引入了一个 方便,提供了一种简单的格式化器SPI和健壮的替代 PropertyEditors为客户端环境。

一般来说,使用转换器SPI当你需要实现 通用类型转换逻辑;例如,对于之间的转换 一个java跑龙套。 日期和和java朗长。 使用Formatter SPI当你 工作在客户端环境,如一个web应用程序,和需要 解析和打印本地化的字段值。 这个ConversionService提供了一个 统一的类型转换API对于spi。

7.6.1A格式化器SPI

格式化程序SPI来实现字段格式的逻辑是简单的和 强类型:

```
package org.springframework.format;

public interface Formatter<T> extends Printer<T>, Parser<T> {
}
```

在格式化程序扩展从打印机和解析器积木吗 接口:

```
public interface Printer<T> {
    String print(T fieldValue, Locale locale);
}
```

```
import java.text.ParseException;

public interface Parser<T> {
    T parse(String clientValue, Locale locale) throws ParseException;
}
```

创建您自己的格式器,只需实现格式化程序 上面的接口。 不要被参数化对象的类型,你想 格式,例如, java util日期 。 实现 这个print() 操作打印一个实例的T 在客户端显示的语言环境。 实现 parse() 操作的实例来解析从T 格式化的表示返回客户端语言环境。 你 格式化程序应该抛出ParseException或IllegalArgumentException如果一个 解析失败。 照顾,以确保你的格式化程序实现 是线程安全的。

几个格式化程序实现中提供 格式 子包作为一种便利。 这个 号码 包提供一个NumberFormatter, CurrencyFormatter,PercentFormatter格式化java朗号码 对象使用一个java文本numberformat。 这个 datetime 包提供了一个DateFormatter格式化 java跑龙套。 日期对象与一个java文本dateformat。 这个 datetime joda 包提供了全面的datetime格式支持基于 Joda时间图书馆 。

考虑 DateFormatter 作为一个例子 格式化程序 实现:

```
package org.springframework.format.datetime;

public final class DateFormatter implements Formatter<Date> {

    private String pattern;

    public DateFormatter(String pattern) {
        this.pattern = pattern;
    }

    public String print(Date date, Locale locale) {
        if (date == null) {
            return "";
        }
        return getDateFormat(locale).format(date);
    }

    public Date parse(String formatted, Locale locale) throws ParseException {
        if (formatted.length() == 0) {
            return null;
        }
        return getDateFormat(locale).parse(formatted);
    }

    protected DateFormat getDateFormat(Locale locale) {
        DateFormat dateFormat = new SimpleDateFormat(this.pattern, locale);
        dateFormat.setLenient(false);
        return dateFormat;
    }

}
```

春季团队欢迎社区驱动的格式化程序的贡献; 看到 http://jira.springframework.org 做出贡献。

7.6.2A注解驱动的格式

正如您将看到的,字段格式可以配置的字段类型 或注释。 绑定一个标注格式器,实现 AnnotationFormatterFactory:

```
package org.springframework.format;

public interface AnnotationFormatterFactory<A extends Annotation> {

    Set<Class<?>> getFieldTypes();

    Printer<?> getPrinter(A annotation, Class<?> fieldType);

    Parser<?> getParser(A annotation, Class<?> fieldType);

}
```

参数化一个是场annotationType你想关联 格式化逻辑,例如 org.springframework.format.annotation.DateTimeFormat 。 有 getFieldTypes() 返回类型的字段 注释可用于。 有 getPrinter() 返回一个打印机来打印一个带注释的字段的值。 有 getParser() 返回一个解析器来解析 clientValue为一个带注释的字段。

下面的示例AnnotationFormatterFactory实现绑定 @NumberFormat formatter的注释。 这个注释允许 要么一个号码风格或模式指定:

```
public final class NumberFormatAnnotationFormatterFactory
        implements AnnotationFormatterFactory<NumberFormat> {

    public Set<Class<?>> getFieldTypes() {
        return new HashSet<Class<?>>(asList(new Class<?>[] {
            Short.class, Integer.class, Long.class, Float.class,
            Double.class, BigDecimal.class, BigInteger.class }));
    }

    public Printer<Number> getPrinter(NumberFormat annotation, Class<?> fieldType) {
        return configureFormatterFrom(annotation, fieldType);
    }

    public Parser<Number> getParser(NumberFormat annotation, Class<?> fieldType) {
        return configureFormatterFrom(annotation, fieldType);
    }

    private Formatter<Number> configureFormatterFrom(NumberFormat annotation,
                                                     Class<?> fieldType) {
        if (!annotation.pattern().isEmpty()) {
            return new NumberFormatter(annotation.pattern());
        } else {
            Style style = annotation.style();
            if (style == Style.PERCENT) {
                return new PercentFormatter();
            } else if (style == Style.CURRENCY) {
                return new CurrencyFormatter();
            } else {
                return new NumberFormatter();
            }
        }
    }
}
```

触发格式化,只是注释字段@NumberFormat:

```
public class MyModel {

    @NumberFormat(style=Style.CURRENCY)
    private BigDecimal decimal;

}
```

格式注释API

一个便携式格式注释API中存在 org.springframework.format.annotation 包。 使用java . lang @NumberFormat格式化。 数字领域。 使用 @DateTimeFormat格式化java跑龙套。 日期、java跑龙套的日历, java跑龙套。 长,或Joda时间字段。

下面的例子使用@DateTimeFormat格式化一个java util日期 作为一个ISO日期(yyyy mm dd):

```
public class MyModel {

    @DateTimeFormat(iso=ISO.DATE)
    private Date date;

}
```

7.6.3A FormatterRegistry SPI

这个是一个注册FormatterRegistry SPI格式化程序和 转换器。 FormattingConversionService 是 一个实现FormatterRegistry适合大多数环境。 这个实现可以被配置以编程方式或声明 作为一个Spring bean使用FormattingConversionServiceFactoryBean 。 因为这个实现还实现了 conversionService ,它可以直接 配置为使用Spring的DataBinder和弹簧表达式 语言(?)。

回顾FormatterRegistry SPI如下:

```
package org.springframework.format;

public interface FormatterRegistry extends ConverterRegistry {

    void addFormatterForFieldType(Class<?> fieldType, Printer<?> printer, Parser<?> parser);

    void addFormatterForFieldType(Class<?> fieldType, Formatter<?> formatter);

    void addFormatterForFieldType(Formatter<?> formatter);

    void addFormatterForAnnotation(AnnotationFormatterFactory<?, ?> factory);

}
```

如上所示,格式化程序可以由fieldType注册或 注释。

这个FormatterRegistry SPI允许您配置格式规则 集中,而不是复制这样的配置在你 控制器。 例如,您可能希望执行,所有日期字段 格式化一个特定的方式,或字段与特定的注释是吗 在一个特定的方式格式化。 FormatterRegistry共享,您定义 这些规则一旦和他们应用每当需要格式化。

7.6.4A FormatterRegistrar SPI

这个是一个注册FormatterRegistrar SPI格式化程序和 通过FormatterRegistry转换器:

```
package org.springframework.format;

public interface FormatterRegistrar {

    void registerFormatters(FormatterRegistry registry);

}
```

一个FormatterRegistrar是有用,当注册多个相关 转换器和格式化程序用于给定格式类别,例如日期 格式化。 它也可以是有用的,声明式登记 不够的。 例如当一个格式化程序需要索引在 特定的字段类型不同于自己的< T >或当注册 打印机/解析器对。 下一节提供了更多的信息 转换器和格式化程序登记。

7.6.5A配置格式在Spring MVC

在一个Spring MVC应用程序中,您可以配置一个自定义的 ConversionService实例明确的属性 注解驱动的 元素的名称空间的MVC。 这 ConversionService将随时使用一个类型转换是 需要在控制器模型绑定。 如果没有显式配置, Spring MVC将自动默认格式器和转换器登记 对于常见的类型,例如数字和日期。

依赖默认格式规则,没有自定义配置 需要在你的Spring MVC配置XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:mvc="http://www.springframework.org/schema/mvc"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/mvc
      http://www.springframework.org/schema/mvc/spring-mvc.xsd">

   <mvc:annotation-driven/>

</beans>
```

只有一行字的配置,默认格式化程序用于数字 和日期类型将被安装,包括支持 @NumberFormat和@DateTimeFormat注释。 Joda完全支持 时间格式库也安装如果Joda时间上是否存在 类路径。

注入一个ConversionService实例使用自定义格式器和 转换器注册,设置转换服务属性,然后 指定自定义转换器,格式器,或

FormatterRegistrars作为属性 的FormattingConversionServiceFactoryBean:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:mvc="http://www.springframework.org/schema/mvc"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/mvc
      http://www.springframework.org/schema/mvc/spring-mvc.xsd">

   <mvc:annotation-driven conversion-service="conversionService"/>

   <bean id="conversionService"
       class="org.springframework.format.support.FormattingConversionServiceFactoryBean">
      <property name="converters">
         <set>
            <bean class="org.example.MyConverter"/>
         </set>
      </property>
      <property name="formatters">
         <set>
            <bean class="org.example.MyFormatter"/>
            <bean class="org.example.MyAnnotationFormatterFactory"/>
         </set>
      </property>
      <property name="formatterRegistrars">
         <set>
            <bean class="org.example.MyFormatterRegistrar"/>
         </set>
      </property>
   </bean>

</beans>
```

注意

看到 SectionA 7 6 4,一个FormatterRegistrar SPIa 和 这个 FormattingConversionServiceFactoryBean 更多信息FormatterRegistrars时使用。

7.7一个配置一个全球日期&时间格式

默认情况下,日期和时间字段,不标注 @DateTimeFormat 从字符串转换 使用的 dateformat短 风格。 如果你喜欢, 你可以改变这个定义你自己的全球格式。

您将需要确保弹簧不注册违约 格式化程序,相反你应该登记所有手动格式化器。 使用 org.springframework.format.datetime.joda.JodaTimeFormatterRegistrar 或 org.springframework.format.datetime.DateFormatterRegistrar 取决于您使用类Joda时间图书馆。

例如,下面的Java配置将注册一个全球 " yyyyMMdd 的格式。 这个示例并不依赖 Joda时间图书馆:

```
@Configuration
public class AppConfig {

  @Bean
  public FormattingConversionService conversionService() {

    // Use the DefaultFormattingConversionService but do not register defaults
    DefaultFormattingConversionService conversionService = new DefaultFormattingConversionService(false);

    // Ensure @NumberFormat is still supported
    conversionService.addFormatterForFieldAnnotation(new NumberFormatAnnotationFormatterFactory());

    // Register date conversion with a specific global format
    DateFormatterRegistrar registrar = new DateFormatterRegistrar();
    registrar.setFormatter(new DateFormatter("yyyyMMdd"));
    registrar.registerFormatters(conversionService);

    return conversionService;
  }
}
```

如果你更喜欢基于XML的配置可以使用 FormattingConversionServiceFactoryBean 。 这是相同的 例子,这次使用Joda时间:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
```

```
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd>

  <bean id="conversionService" class="org.springframework.format.support.FormattingConversionServiceFactoryBean">
    <property name="registerDefaultFormatters" value="false" />
    <property name="formatters">
    <set>
      <bean class="org.springframework.format.number.NumberFormatAnnotationFormatterFactory" />
    </set>
    </property>
    <property name="formatterRegistrars">
    <set>
     <bean class="org.springframework.format.datetime.joda.JodaTimeFormatterRegistrar">
       <property name="dateFormatter">
         <bean class="org.springframework.format.datetime.joda.DateTimeFormatterFactoryBean">
           <property name="pattern" value="yyyyMMdd"/>
         </bean>
       </property>
     </bean>
   </set>
   </property>
  </bean>
 </beans>
```

注意

Joda时间提供了单独的不同类型来表示 日期 , 时间 和 日期-时间 值。 这个 DateFormatter , timeFormatter 和 dateTimeFormatter 属性的 JodaTimeFormatterRegistrar 应该 被用于配置不同的格式为每个类型。 这个 DateTimeFormatterFactoryBean 提供了一个 方便地创建格式化器。

如果您正在使用Spring MVC记得显式配置 转换服务使用。 为基于Java @ configuration 这意味着延长 WebMvcConfigurationSupport 类并覆盖 这个 mvcConversionService() 法。 对于XML你应该 使用 " 转换服务的 属性的 mvc:注解驱动的 元素。 看到 SectionA 7 6 5,一个配置格式在春天MVCa 详情。

7.8一个弹簧3验证

弹簧3介绍了几种增强其验证支持。 首先,jsr - 303 API Bean验证是现在完全支持。 第二, 以编程方式使用时,弹簧的 DataBinder现在可以验证对象 以及绑定到他们。 第三,Spring MVC现在支持声明 验证输入controller。

7.8.1A概述jsr - 303的Bean验证API

jsr - 303规范的验证约束声明和元数据 对于Java平台。 使用该API,您标注了域模型 属性与声明性验证约束和运行时 强制执行它们。 有很多内置的约束可以采取 利用。 你也可以定义自己的自定义的约束。

为了说明这一点,考虑一个简单的PersonForm模型两个 属性:

```
public class PersonForm {
    private String name;
    private int age;
}
```

jsr - 303允许您定义声明性验证约束 反对这样的属性:

```
public class PersonForm {

    @NotNull
    @Size(max=64)
    private String name;

    @Min(0)
    private int age;

}
```

当这个类的一个实例验证了一个jsr - 303验证器, 这些限制将被强制执行。

对于一般信息关于jsr - 303,请参阅 Bean验证 规范 。 关于具体的功能 默认的参考实现,看到 Hibernate Validator 文档。 学习如何设置一个jsr - 303实现作为一个 Spring bean,继续阅读。

7.8.2A配置Bean验证实现

Spring提供了完全支持jsr - 303 API Bean验证。 这包括方便的支持引导一个jsr - 303 作为一个Spring bean实现。 这允许一个 javax.validation.ValidatorFactory 或 javax.validation.Validator 注射无论 验证是必要的在您的应用程序。

使用 LocalValidatorFactoryBean 到 配置一个默认的jsr - 303验证器作为一个Spring bean:

```
<bean id="validator"
    class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"/>
```

上面的基本配置将触发jsr - 303来初始化 使用其默认引导机制。 jsr - 303的供应商,如 Hibernate Validator,预计将出现在类路径,并将 被自动检测到。

注入一个验证器

LocalValidatorFactoryBean 实现两 javax.validation.ValidatorFactory 和 javax.validation.Validator 以及弹簧的 org.springframework.validation.Validator 。 你可能注入 引用这两种接口到bean,需要 调用验证逻辑。

注入一个参考 javax.validation.Validator 如果 你更喜欢使用jsr - 303 API直接:

```
import javax.validation.Validator;

@Service
public class MyService {

    @Autowired
    private Validator validator;
```

注入一个参考 org.springframework.validation.Validator 如果您的bean 需要弹簧验证API:

```
import org.springframework.validation.Validator;

@Service
public class MyService {

    @Autowired
    private Validator validator;

}
```

配置自定义约束

每一个jsr - 303验证约束包括两个部分。 首先, 一个@Constraint注释来声明约束及其 可配置特性。 第二,一个实现的 javax.validation.ConstraintValidator 接口, 实现约束的行为。 把一个声明 一个实现,每个@Constraint注释引用 相应的 ValidationConstraint实现类。 在运行时,一个 ConstraintValidatorFactory 实例化所引用的 实现当约束注释中遇到你 域模型。

默认情况下, LocalValidatorFactoryBean 配置 SpringConstraintValidatorFactory 使用 Spring创建ConstraintValidator实例。 这允许你 自定义ConstraintValidators受益于依赖注入像 任何其他Spring bean。

下面是一个例子,一个自定义@Constraint宣言, 后跟一个相关的 ConstraintValidator 实现,使用弹簧用于依赖注入:

```
@Target({ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy=MyConstraintValidator.class)
public @interface MyConstraint {
}
```

```
import javax.validation.ConstraintValidator;

public class MyConstraintValidator implements ConstraintValidator {

    @Autowired;
    private Foo aDependency;

    ...
}
```

正如您可以看到的,一个ConstraintValidator实现可能有其 依赖性的@ autowired像任何其他Spring bean。

额外的配置选项

默认 LocalValidatorFactoryBean 配置应该证明足够的大多数情况下。 有一个 数量的其他配置选项不同的jsr - 303结构, 从信息插值到遍历解析。 看到JavaDocs 的 LocalValidatorFactoryBean 更多 这些选项的信息。

7.8.3A DataBinder配置一个

因为春天3,DataBinder实例都可以配置 验证器。 一旦配置完成,验证器可以被调用 粘合剂validate() 。 任何验证错误自动 添加到活页夹BindingResult。

当使用DataBinder编程方式,这可以利用 调用验证逻辑绑定后向目标对象:

```
Foo target = new Foo();
DataBinder binder = new DataBinder(target);
binder.setValidator(new FooValidator());

// bind to the target object
binder.bind(propertyValues);

// validate the target object
binder.validate();

// get BindingResult that includes any validation errors
BindingResult results = binder.getBindingResult();
```

一个DataBinder也可以配置多个 验证器 实例 通过 dataBinder.addValidators 和 dataBinder.replaceValidators 。 这是有用的在jsr - 303结合全球配置Bean验证 用弹簧 验证器 配置 在本地一个DataBinder实例。 看到 一个章节配置一个验证器使用的弹簧MVCa 。

7.8.4A Spring MVC 3验证

从弹簧3,Spring MVC有能力 自动验证输入controller。 在之前的版本中,这是 开发人员手动调用验证逻辑。

触发controller输入验证

触发验证一个controller输入,只需标注 @Valid输入参数为:

```
@Controller
public class MyController {

    @RequestMapping("/foo", method=RequestMethod.POST)
    public void processFoo(@Valid Foo foo) { /* ... */ }
```

Spring MVC将验证@Valid对象绑定后这么长时间为 一个适当的验证器被配置。

注意

@Valid的注释部分的标准jsr - 303豆 验证API,并不是一个spring特定构造。

配置一个验证器使用Spring MVC

Validator实例时调用一个方法@Valid论点 遇到可能以两种方式配置。 首先,你可以叫 binder.setValidator(Validator)内的 @InitBinder controller 回调。 这允许您配置一个验证器实例/ controller类:

```
@Controller
public class MyController {

    @InitBinder
    protected void initBinder(WebDataBinder binder) {
        binder.setValidator(new FooValidator());
    }

    @RequestMapping("/foo", method=RequestMethod.POST)
    public void processFoo(@Valid Foo foo) { ... }

}
```

第二,你可以叫setValidator(Validator)在全球 WebBindingInitializer。 这允许您配置一个验证器 在所有@Controllers实例。 这可以实现很容易使用 Spring MVC的命名空间:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <mvc:annotation-driven validator="globalValidator"/>
```

```
</beans>
```

结合全球和当地的一个验证器,配置 全球验证程序如上所示,然后添加一个本地验证器:

```
@Controller
public class MyController {

    @InitBinder
    protected void initBinder(WebDataBinder binder) {
        binder.addValidators(new FooValidator());
    }

}
```

配置一个jsr - 303验证器使用Spring MVC

与jsr - 303,一个单一的 javax.validation.Validator 实例通常验证 所有 模型对象 这声明验证约束。 配置一个jsr - 303支持 验证器与Spring MVC,只需添加一个jsr - 303提供者,如 Hibernate Validator,到您的类路径中。 Spring MVC将检测它, 自动启用jsr - 303支持在所有控制器。

Spring MVC的配置才能使jsr - 303的支持 如下所示:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <!-- JSR-303 support will be detected on classpath and enabled automatically -->
    <mvc:annotation-driven/>

</beans>
```

在这种最小配置,任何时候@Valid controller 输入时,它将被验证了jsr - 303提供者。 jsr - 303,反过来,将执行任何约束声明反对了 输入。 任何ConstraintViolations将自动被公开为 BindingResult可渲染的错误标准Spring MVC形式 标签。

8。 一个春天的表达式语言(?)

8.1一个介绍

春天表达式语言(简称?)是一个强大的 表达式语言,支持查询和操作一个对象 图在运行时。 语言的语法类似于统一EL但提供 额外的功能,最明显的是方法调用和基本的字符串 模板的功能。

虽然有几个其他的Java表达式语言可用, OGNL,MVEL,JBoss EL等等,春天表达式语言 被建立提供春季社区提供单井支持吗 表达式语言,可以使用所有产品在春天 投资组合。 它是由语言特性的要求 项目在春天的投资组合,包括工具要求代码 完成基于eclipse的党内支持SpringSource工具套件。 这 说,?是基于技术无关的API允许其他 表达式语言实现集成应该需要 出现。

而作为基础?表达式求值的在 弹簧组合,它不直接绑定到春天和可用 独立。 为了是自包含的,许多例子在这 章使用?就好像它是一个独立的表达式语言。 这 需要创建一些引导基础设施类等 解析器。 大多数Spring用户不需要处理这个基础设施 只有作者表达,而是将字符串评价。 一个例子 这种典型的使用是集成到创建的XML或? 基于注释的bean定义的部分所示 表达式支持定义bean 定义。

本章涵盖的特征表达式语言,它的 API,它的语言的语法。 在几个地方一个发明家,发明家的 社会阶级是作为目标对象表达式求值的。 这些类声明和使用的数据填充它们列在 一章的结束。

8.2一个特性概述

表达式语言支持以下功能

- 字面表达式
- 布尔和关系运算符
- 正则表达式
- 类表达式

- 访问属性、数组、列表、地图
- 方法调用
- 关系运算符
- 分配
- 调用构造函数
- Bean引用
- 阵列结构
- 内联列表
- 三元操作符
- 变量
- 用户定义函数
- 收集投影
- 集合选择
- 模板化的表达式

8.3一个表达式求值的表达式使用Spring接口

本节介绍了简单的使用接口和它? 表达式语言。 完整的语言参考中可以找到 部分 语言 参考 。

下面的代码介绍了API来评估文字? “你好,世界” 的字符串表达式。

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("'Hello World'");
String message = (String) exp.getValue();
```

的价值 消息变量只是“你好,世界” 。

类和接口的?你是最有可能使用 位于包 org.springframework.expression 和它的子包和 ?支持 。

接口 ExpressionParser 是 负责解析表达式的字符串。 在这个例子中, 表达式的字符串是一个字符串,表示为周围的单 引用。 接口 表达式 是 负责评估前面定义的表达式字符串。 有 是两个例外,可以扔, ParseException 和 EvaluationException 当调用 ” parser.parseExpression ” 和 ” exp getvalue 分别的。

?支持广泛的特性,比如调用方法, 访问属性、调用构造函数。

作为一个例子的方法调用,我们称之为“concat的方法 字符串文字。

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("'Hello World'.concat('!')");
String message = (String) exp.getValue();
```

信息的价值是现在“Hello World !” 。

作为一个例子,调用JavaBean属性的字符串属性 “字节” 可以称为如下所示。

```
ExpressionParser parser = new SpelExpressionParser();

// invokes 'getBytes()'
Expression exp = parser.parseExpression("'Hello World'.bytes");

byte[] bytes = (byte[]) exp.getValue();
```

嵌套属性?还支持使用标准的“点点” 符号, 即prop1.prop2。 prop3和设置属性值

公共字段也可能被访问。

```
ExpressionParser parser = new SpelExpressionParser();

// invokes 'getBytes().length'
Expression exp = parser.parseExpression("'Hello World'.bytes.length");

int length = (Integer) exp.getValue();
```

字符串的构造函数可以调用而不是使用一个字符串 文字。

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("new String('hello world').toUpperCase()");
```

```
String message = exp.getValue(String.class);
```

注意这里使用的通用方法 公共< T > T getValue(类< T > desiredResultType) 。 使用这种方法 删除需要铸造表达式的值到预期的结果 类型。 一个 EvaluationException 将会抛出如果 价值不能抛到类型 T 或转换使用 注册的类型转换器。

更常见的用法是提供一个表达式?字符串 是针对特定对象实例评估(称为根对象)。 这里有两个选项,选择哪一个取决于对象 对表达式进行评估将改变每个 调用表达式求值。 在接下来的例子中 我们检索 名称 产权的一个实例 发明家类。

```
// Create and set a calendar
GregorianCalendar c = new GregorianCalendar();
c.set(1856, 7, 9);

// The constructor arguments are name, birthday, and nationality.
Inventor tesla = new Inventor("Nikola Tesla", c.getTime(), "Serbian");

ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("name");
EvaluationContext context = new StandardEvaluationContext(tesla);

String name = (String) exp.getValue(context);
```

在过去的 线,这个值的字符串变量的名字将被设置为“尼古拉 特斯拉” 。 类StandardEvaluationContext就是你可以指定 对象的“name”属性将依据。 这是机制 使用如果根对象是不可能改变的,它可以简单地设置一次 在评估上下文。 如果根对象可能会改变 多次,它可以提供在每次调用 getValue , 作为下一个示例显示:

```
/ Create and set a calendar
GregorianCalendar c = new GregorianCalendar();
c.set(1856, 7, 9);

// The constructor arguments are name, birthday, and nationality.
Inventor tesla = new Inventor("Nikola Tesla", c.getTime(), "Serbian");

ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("name");

String name = (String) exp.getValue(tesla);
```

在这种情况下,发明家 特斯拉 一直 直接供给 getValue 和表达 评估基础设施创建和管理一个默认评价上下文 在内部,它不需要一个提供。

StandardEvaluationContext相对昂贵的构造和 在重复使用它建立缓存状态,使后续 表达评估执行更迅速。 因为这个原因,它是 更好的缓存和重用他们在可能的情况下,而不是构建一个新的 一个用于每个表达式求值。

在某些情况下,它可以理想的使用配置评估上下文和 但仍然提供一个不同的根对象在每次调用 getValue 。 getValue 允许两个上指定相同的调用。 在这些情况下根对象通过调用被覆盖 任何(可能为空)指定的评估上下文。

注意

在独立使用?有必要创建解析器, 解析表达式和也许提供评估上下文和根 上下文对象。 然而,更常见的用法是提供只有?表达式字符串的一部分吗 配置文件,例如Spring bean或Spring Web Flow 定义。 在这种情况下,解析器、评估上下文,根对象 和任何预定义的变量都是建立隐式,要求 用户指定任何其他比表达式。

最后介绍的例子,使用布尔运算符 显示使用发明家对象在前面的例子。

```
Expression exp = parser.parseExpression("name == 'Nikola Tesla'");
boolean result = exp.getValue(context, Boolean.class);  // evaluates to true
```

8.3.1A EvaluationContext接口的

接口 EvaluationContext 是 使用在评估一个表达式来解析属性、方法 字段,并帮助执行类型转换。 开箱即用 的实现, StandardEvaluationContext ,使用 反射来操纵对象,缓存 java朗反映 ' s 方法 , 领域 ,和 构造函数 为提高性能的实例。

这个 StandardEvaluationContext 就是你 可以指定根对象评估反对通过方法 setRootObject() 或通过根对象 构造函数。 您还可以指定变量和函数 将用于表达式使用方法吗 setVariable() 和 registerFunction() 。 使用变量和 函数是描述在语言参考部分 变量 和 功能 。 这个 StandardEvaluationContext 也是,你可以吗 注册自定义 ConstructorResolver 年代, MethodResolver 年代, PropertyAccessor 年代延长?如何评估 表达式。 请参考这些类的JavaDoc更多 细节。

类型转换

默认情况下?使用转换服务可以在春天 核心（org.springframework.core.convert.ConversionService）。 这个转换服务有许多转换器建在为常见 转换但也完全可扩展的如此定义之间的转换 类型可以被添加。 另外它有关键的能力,它是 仿制药知道。 这意味着在使用中泛型类型 表情,?会尝试转换维护类型 遇到的任何对象的正确性。

在实践中这意味着什么? 假设任务,使用 setValue() ,是被用来设置一个 列表 财产。 属性的类型实际上是 列表<布尔> 。 会认识到,? 元素列表需要转换 布尔 在被放置在它。 一个简单的 示例:

```
class Simple {
   public List<Boolean> booleanList = new ArrayList<Boolean>();
}

Simple simple = new Simple();

simple.booleanList.add(true);

StandardEvaluationContext simpleContext = new StandardEvaluationContext(simple);

// false is passed in here as a string.  SpEL and the conversion service will
// correctly recognize that it needs to be a Boolean and convert it
parser.parseExpression("booleanList[0]").setValue(simpleContext, "false");

// b will be false
Boolean b = simple.booleanList.get(0);
```

8.4一个表达式支持定义bean定义

?表情可以使用XML或基于注解的 配置元数据定义BeanDefinitions。 在这两种情况下, 语法定义表达式的形式 # { <表达式 字符串> } 。

8.4.1A基于XML的配置

一个属性或constructor - arg值可以设置使用表达式 如下所示

```
<bean id="numberGuess" class="org.spring.samples.NumberGuess">
   <property name="randomNumber" value="#{ T(java.lang.Math).random() * 100.0 }"/>

   <!-- other properties -->
</bean>
```

变量“systemProperties’ 是预定义的,所以你可以使用它 在你的表达式如下所示。 注意,您不需要前缀 预定义的变量用“#” 符号在这个上下文。

```
<bean id="taxCalculator" class="org.spring.samples.TaxCalculator">
   <property name="defaultLocale" value="#{ systemProperties['user.region'] }"/>

   <!-- other properties -->
</bean>
```

你也可以参考其他bean属性的名字, 的例子。

```
<bean id="numberGuess" class="org.spring.samples.NumberGuess">
   <property name="randomNumber" value="#{ T(java.lang.Math).random() * 100.0 }"/>

   <!-- other properties -->
</bean>


<bean id="shapeGuess" class="org.spring.samples.ShapeGuess">
   <property name="initialShapeSeed" value="#{ numberGuess.randomNumber }"/>

   <!-- other properties -->
</bean>
```

8.4.2A基于注解的配置

这个 这个元素包含一个@ value 注释可以放在字段, 方法和构造函数参数的方法/指定一个默认的 值。

这里有一个例子来设置默认字段值 变量。

```
public static class FieldValueTestBean

  @Value("#{ systemProperties['user.region'] }")
  private String defaultLocale;
```

```
  public void setDefaultLocale(String defaultLocale)
  {
    this.defaultLocale = defaultLocale;
  }

  public String getDefaultLocale()
  {
    return this.defaultLocale;
  }

}
```

但在一个相当于属性setter方法显示 下面。

```
public static class PropertyValueTestBean

  private String defaultLocale;

  @Value("#{ systemProperties['user.region'] }")
  public void setDefaultLocale(String defaultLocale)
  {
    this.defaultLocale = defaultLocale;
  }

  public String getDefaultLocale()
  {
    return this.defaultLocale;
  }

}
```

Autowired的方法和构造函数也可以使用 这个元素包含一个@ value 注释。

```
public class SimpleMovieLister {

  private MovieFinder movieFinder;
  private String defaultLocale;

  @Autowired
  public void configure(MovieFinder movieFinder,
                        @Value("#{ systemProperties['user.region'] }"} String defaultLocale) {
      this.movieFinder = movieFinder;
      this.defaultLocale = defaultLocale;
  }

  // ...
}
```

```
public class MovieRecommender {

  private String defaultLocale;

  private CustomerPreferenceDao customerPreferenceDao;

  @Autowired
  public MovieRecommender(CustomerPreferenceDao customerPreferenceDao,
                          @Value("#{systemProperties['user.country']}"} String defaultLocale) {
      this.customerPreferenceDao = customerPreferenceDao;
      this.defaultLocale = defaultLocale;
  }

  // ...
}
```

8.5语言参考

8.5.1A字面表达式

这个类型的文字表达式支持都是字符串,日期, 数值(int,真实,和十六进制),布尔和null。 字符串是 由单引号分隔。 将单引号本身在一个字符串使用 两个单引号字符。 下面的清单显示了简单的使用 文字。 通常他们不会被用于隔离,但 作为一个更复杂的表达式,例如使用文字在一个 边的一个逻辑比较运算符。

```
ExpressionParser parser = new SpelExpressionParser();

// evals to "Hello World"
String helloWorld = (String) parser.parseExpression("'Hello World'").getValue();

double avogadrosNumber  = (Double) parser.parseExpression("6.0221415E+23").getValue();

// evals to 2147483647
```

```
int maxValue = (Integer) parser.parseExpression("0x7FFFFFFF").getValue();

boolean trueValue = (Boolean) parser.parseExpression("true").getValue();

Object nullValue = parser.parseExpression("null").getValue();
```

数据支持使用负号,指数 符号,小数点。 默认情况下真正的数字解析使用 Double.parseDouble()。

8.5.2A属性、数组、列表、地图、索引器

导航与产权引用很容易,只需使用一个时期 显示一个嵌套的属性值。 类的实例的发明家,加感 和特斯拉,来填充数据列在部分 类用于 例子 。 导航“下”,得到特斯拉的出生年份和 加感的出生的城市下面的表达式是使用。

```
// evals to 1856
int year = (Integer) parser.parseExpression("Birthdate.Year + 1900").getValue(context);


String city = (String) parser.parseExpression("placeOfBirth.City").getValue(context);
```

不区分大小写是允许第一个字母的财产 的名字。 数组和列表的内容得到了使用方 括号表示法。

```
ExpressionParser parser = new SpelExpressionParser();

// Inventions Array
StandardEvaluationContext teslaContext = new StandardEvaluationContext(tesla);

// evaluates to "Induction motor"
String invention = parser.parseExpression("inventions[3]").getValue(teslaContext,
                                                        String.class);


// Members List
StandardEvaluationContext societyContext = new StandardEvaluationContext(ieee);

// evaluates to "Nikola Tesla"
String name = parser.parseExpression("Members[0].Name").getValue(societyContext, String.class);

// List and Array navigation
// evaluates to "Wireless communication"
String invention = parser.parseExpression("Members[0].Inventions[6]").getValue(societyContext,
                                                                  String.class);
```

地图的内容得到了指定文字的关键 括号内的值。 在这种情况下,因为钥匙的人员 地图都是字符串,我们可以指定字符串字面值。

```
// Officer's Dictionary

Inventor pupin = parser.parseExpression("Officers['president']").getValue(societyContext,
                                                        Inventor.class);

// evaluates to "Idvor"
String city =
   parser.parseExpression("Officers['president'].PlaceOfBirth.City").getValue(societyContext,
                                                                  String.class);

// setting values
parser.parseExpression("Officers['advisors'][0].PlaceOfBirth.Country").setValue(societyContext,
                                                                  "Croatia");
```

8.5.3A内联列表

列表可以直接表达的表达式中使用{ }符号。

```
// evaluates to a Java list containing the four numbers
List numbers = (List) parser.parseExpression("{1,2,3,4}").getValue(context);

List listOfLists = (List) parser.parseExpression("{{'a','b'},{'x','y'}}").getValue(context);
```

{ }本身意味着一个空列表。 由于性能原因,如果 列表本身是完全由固定文字然后创建一个常数列表 代表表达式,而不是建立一个新的列表中每一个评价。

8.5.4A阵列结构

数组可以使用熟悉的Java语法,可选地 提供一个初始化的数组填充在施工时间。

```
int[] numbers1 = (int[]) parser.parseExpression("new int[4]").getValue(context);

// Array with initializer
int[] numbers2 = (int[]) parser.parseExpression("new int[]{1,2,3}").getValue(context);

// Multi dimensional array
int[][] numbers3 = (int[][]) parser.parseExpression("new int[4][5]").getValue(context);
```

目前还不允许提供一个初始化器在构造 一个多维数组。

8.5.5A方法

方法调用使用典型的Java编程语法。 你可能 还在文字调用方法。 也支持可变参数。

```
// string literal, evaluates to "bc"
String c = parser.parseExpression("'abc'.substring(2, 3)").getValue(String.class);

// evaluates to true
boolean isMember = parser.parseExpression("isMember('Mihajlo Pupin')").getValue(societyContext,
                                                                                 Boolean.class);
```

8.5.6A运营商

关系运算符

关系运算符;平等,不平等,小于,小于 或等于、大于、大于或等于支持使用 标准算子符号。

```
// evaluates to true
boolean trueValue = parser.parseExpression("2 == 2").getValue(Boolean.class);

// evaluates to false
boolean falseValue = parser.parseExpression("2 < -5.0").getValue(Boolean.class);

// evaluates to true
boolean trueValue = parser.parseExpression("'black' < 'block'").getValue(Boolean.class);
```

除了标准的关系运算符?支持 “运算符” 和基于正则表达式的匹配操作符。

```
// evaluates to false
boolean falseValue = parser.parseExpression("'xyz' instanceof T(int)").getValue(Boolean.class);

// evaluates to true
boolean trueValue =
    parser.parseExpression("'5.00' matches '^-?\\d+(\\.\\d{2})?$'").getValue(Boolean.class);

//evaluates to false
boolean falseValue =
    parser.parseExpression("'5.0067' matches '^-?\\d+(\\.\\d{2})?$'").getValue(Boolean.class);
```

每个符号操作符也可以被指定为一个纯粹的字母等效。 这就避免了 问题,有特殊含义的符号使用的文档类型中 表达式是嵌入式(如。 一个XML文档)。 显示文本等价物 在这里:lt(“<”),gt(' > '),勒(“< =”)、通用电气(“> =”), eq(' = = ')、ne(“! =”),div(“/”)、国防部(“%”),而不是(' !”)。 这些都是不区分大小写的。

逻辑运算符

逻辑运算符,支持和,或者,不。 他们使用了下面。

```
// -- AND --

// evaluates to false
boolean falseValue = parser.parseExpression("true and false").getValue(Boolean.class);

// evaluates to true
String expression = "isMember('Nikola Tesla') and isMember('Mihajlo Pupin')";
boolean trueValue = parser.parseExpression(expression).getValue(societyContext, Boolean.class);

// -- OR --

// evaluates to true
boolean trueValue = parser.parseExpression("true or false").getValue(Boolean.class);

// evaluates to true
String expression = "isMember('Nikola Tesla') or isMember('Albert Einstein')";
boolean trueValue = parser.parseExpression(expression).getValue(societyContext, Boolean.class);

// -- NOT --
```

```
// evaluates to false
boolean falseValue = parser.parseExpression("!true").getValue(Boolean.class);


// -- AND and NOT --
String expression =  "isMember('Nikola Tesla') and !isMember('Mihajlo Pupin')";
boolean falseValue = parser.parseExpression(expression).getValue(societyContext, Boolean.class);
```

数学运算符

加法操作符可以用在数字、字符串和日期。 减法可以用在数字和日期。 乘法和 部门只能使用在数字。 其他数学运算符 支持模量(%)和指数功率(^)。 标准算子 优先执行。 下面演示了这些操作符。

```
// Addition
int two = parser.parseExpression("1 + 1").getValue(Integer.class); // 2

String testString =
   parser.parseExpression("'test' + ' ' + 'string'").getValue(String.class);  // 'test string'

// Subtraction
int four =  parser.parseExpression("1 - -3").getValue(Integer.class); // 4

double d = parser.parseExpression("1000.00 - 1e4").getValue(Double.class); // -9000

// Multiplication
int six =  parser.parseExpression("-2 * -3").getValue(Integer.class); // 6

double twentyFour = parser.parseExpression("2.0 * 3e0 * 4").getValue(Double.class); // 24.0

// Division
int minusTwo =  parser.parseExpression("6 / -3").getValue(Integer.class); // -2

double one = parser.parseExpression("8.0 / 4e0 / 2").getValue(Double.class); // 1.0

// Modulus
int three =  parser.parseExpression("7 % 4").getValue(Integer.class); // 3

int one = parser.parseExpression("8 / 5 % 2").getValue(Integer.class); // 1

// Operator precedence
int minusTwentyOne = parser.parseExpression("1+2-3*8").getValue(Integer.class); // -21
```

8.5.7A赋值

设置的属性是通过使用赋值运算符。 这通常是一个调用中完成的 setValue 但也可以做在一个调用 getValue 。

```
Inventor inventor = new Inventor();
StandardEvaluationContext inventorContext = new StandardEvaluationContext(inventor);

parser.parseExpression("Name").setValue(inventorContext, "Alexander Seovic2");

// alternatively

String aleks = parser.parseExpression("Name = 'Alexandar Seovic'").getValue(inventorContext,
                                                        String.class);
```

8.5.8A类型

特殊的“T”操作符可以用来指定的一个实例 java . lang。 类(“类型”)。 静态方法调用使用这个 运营商也。 这个 StandardEvaluationContext 使用 TypeLocator 找到类型和 StandardTypeLocator (可替换) 建造一个理解java。 朗包。 这意味着T() 引用类型在java。 朗不需要完全限定的, 但所有其他类型引用必须。

```
Class dateClass = parser.parseExpression("T(java.util.Date)").getValue(Class.class);

Class stringClass = parser.parseExpression("T(String)").getValue(Class.class);

boolean trueValue =
   parser.parseExpression("T(java.math.RoundingMode).CEILING < T(java.math.RoundingMode).FLOOR")
  .getValue(Boolean.class);
```

8.5.9A构造函数

构造函数可以调用使用新的操作符。 完全 限定类名应该用于所有但原始类型和 字符串(整数、浮点等,可以用)。

```
Inventor einstein =
  p.parseExpression("new org.spring.samples.spel.inventor.Inventor('Albert Einstein',
                                                        'German')")
```

```
                                                .getValue(Inventor.class);

//create new inventor instance within add method of List
p.parseExpression("Members.add(new org.spring.samples.spel.inventor.Inventor('Albert Einstein',
                                                'German'))")
                                                .getValue(societyContext);
```

8.5.10A变量

变量可以引用表达式使用语法 # variableName。 变量是设置使用方法setVariable在 StandardEvaluationContext。

```
Inventor tesla = new Inventor("Nikola Tesla", "Serbian");
StandardEvaluationContext context = new StandardEvaluationContext(tesla);
context.setVariable("newName", "Mike Tesla");

parser.parseExpression("Name = #newName").getValue(context);

System.out.println(tesla.getName()) // "Mike Tesla"
```

#这和# root变量

变量#这是总是定义和引用当前 评价对象(对不合格的参考解决)。 变量#根总是定义,指根 上下文对象。 虽然#这可能不同的表达式作为组件 评价,#根总是指根。

```
// create an array of integers
List<Integer> primes = new ArrayList<Integer>();
primes.addAll(Arrays.asList(2,3,5,7,11,13,17));

// create parser and set variable 'primes' as the array of integers
ExpressionParser parser = new SpelExpressionParser();
StandardEvaluationContext context = new StandardEvaluationContext();
context.setVariable("primes",primes);

// all prime numbers > 10 from the list (using selection ?{...})
// evaluates to [11, 13, 17]
List<Integer> primesGreaterThanTen =
          (List<Integer>) parser.parseExpression("#primes.?[#this>10]").getValue(context);
```

8.5.11A功能

您可以扩展通过注册用户定义函数?可以 被称为在表达式的字符串。 这个功能是注册 这个 StandardEvaluationContext 使用法。

```
public void registerFunction(String name, Method m)
```

引用的Java方法提供实现的 函数。 例如,一个实用程序方法来扭转一个字符串显示 下面。

```
public abstract class StringUtils {

  public static String reverseString(String input) {
    StringBuilder backwards = new StringBuilder();
    for (int i = 0; i < input.length(); i++)
      backwards.append(input.charAt(input.length() - 1 - i));
    }
    return backwards.toString();
  }
}
```

该方法然后注册评估上下文和可以 被使用在一个表达式的字符串。

```
ExpressionParser parser = new SpelExpressionParser();
StandardEvaluationContext context = new StandardEvaluationContext();

context.registerFunction("reverseString",
                    StringUtils.class.getDeclaredMethod("reverseString",
                                                new Class[] { String.class }));

String helloWorldReversed =
        parser.parseExpression("#reverseString('hello')").getValue(context, String.class);
```

8.5.12A Bean引用

如果评估上下文已经配置了一个bean解析器是有可能的 查找bean从一个表达式使用符号(@)。

```
ExpressionParser parser = new SpelExpressionParser();
```

```
StandardEvaluationContext context = new StandardEvaluationContext();
context.setBeanResolver(new MyBeanResolver());

// This will end up calling resolve(context,"foo") on MyBeanResolver during evaluation
Object bean = parser.parseExpression("@foo").getValue(context);
```

8.5.13A三元操作符(是以if - then - else)

您可以使用三元操作符用于执行是以if - then - else 条件逻辑表达式内。 一个最小的例子是:

```
String falseString =
        parser.parseExpression("false ? 'trueExp' : 'falseExp'").getValue(String.class);
```

在这种情况下,逻辑错误导致返回字符串 价值”falseExp’ 。 一个更实际的例子如下所示。

```
parser.parseExpression("Name").setValue(societyContext, "IEEE");
societyContext.setVariable("queryName", "Nikola Tesla");

expression = "isMember(#queryName)? #queryName + ' is a member of the ' " +
        "+ Name + ' Society' : #queryName + ' is not a member of the ' + Name + ' Society'";

String queryResultString =
            parser.parseExpression(expression).getValue(societyContext, String.class);
// queryResultString = "Nikola Tesla is a member of the IEEE Society"
```

也见下一节在猫王运营商一个更 短三目操作符的语法。

8.5.14A猫王算子

猫王运营商是一个缩短的三元操作符的语法 和用于 groovy 语言。 与三元操作符语法通常你需要重复 变量两次,例如:

```
String name = "Elvis Presley";
String displayName = name != null ? name : "Unknown";
```

相反,你可以使用猫王算子,命名为相似 到埃尔维斯的发型。

```
ExpressionParser parser = new SpelExpressionParser();

String name = parser.parseExpression("null?:'Unknown'").getValue(String.class);

System.out.println(name);  // 'Unknown'
```

这是一个更复杂的例子。

```
ExpressionParser parser = new SpelExpressionParser();

Inventor tesla = new Inventor("Nikola Tesla", "Serbian");
StandardEvaluationContext context = new StandardEvaluationContext(tesla);

String name = parser.parseExpression("Name?:'Elvis Presley'").getValue(context, String.class);

System.out.println(name); // Mike Tesla

tesla.setName(null);

name = parser.parseExpression("Name?:'Elvis Presley'").getValue(context, String.class);

System.out.println(name); // Elvis Presley
```

8.5.15A安全导航操作符

安全的导航操作符是用来避免 NullPointerException 和来自 groovy 语言。 通常当你有对象的引用你可能 需要确认它不是空之前访问方法或 对象的属性。 为了避免这种情况,安全导航操作符 只会返回null而不是抛出异常。

```
ExpressionParser parser = new SpelExpressionParser();

Inventor tesla = new Inventor("Nikola Tesla", "Serbian");
tesla.setPlaceOfBirth(new PlaceOfBirth("Smiljan"));

StandardEvaluationContext context = new StandardEvaluationContext(tesla);

String city = parser.parseExpression("PlaceOfBirth?.City").getValue(context, String.class);
System.out.println(city); // Smiljan

tesla.setPlaceOfBirth(null);
```

```
city = parser.parseExpression("PlaceOfBirth?.City").getValue(context, String.class);

System.out.println(city); // null - does not throw NullPointerException!!!
```

注意

猫王算子可以用于应用默认值 表达式,例如在一个 这个元素包含一个@ value 表达式:

```
@Value("#{systemProperties['pop3.port'] ?: 25}")
```

这将注入系统属性 pop3港口 如果它 如果不定义或25。

8.5.16A集选择

选择是一个强大的表达式语言功能,允许你 改变一些源集合到另一个通过选择从 条目。

选择使用语法 ?[selectionExpression] 。 这将过滤 收集和返回一个新的集合包含的一个子集 原始元素。 例如,选择将使我们能够很容易地得到一个 塞尔维亚发明家的列表:

```
List<Inventor> list = (List<Inventor>)
    parser.parseExpression("Members.?[Nationality == 'Serbian']").getValue(societyContext);
```

选择是可能的在两个列表和地图。 在前一种情况下 选择标准是依据每个列表元素 同时对地图的选择标准是依据每个 映射项 (对象的Java类型 . entry)。 地图 条目有他们的键和值可以作为属性的使用 选择。

这个表达式将返回一个新的地图构成的元素 原来的地图,输入值小于27。

```
Map newMap = parser.parseExpression("map.?[value<27]").getValue();
```

除了返回所有选中的元素,它是可能的 检索只是第一个或最后一个值。 获得第一个条目 匹配选择语法 ^[...] 同时, 获得最后的匹配选择语法 $[...] 。

8.5.17A收集投影

投影允许一个集合来驱动的评估 子表达式,其结果是一个新的集合。 的语法 投影是 ![projectionExpression] 。 最容易 理解的例子,假设我们有一个列表的发明者,但希望 城市中出生。 实际上我们需要评估 “placeOfBirth。 城市的每一个进入的发明家列表。 使用 投影:

```
// returns [ 'Smiljan', 'Idvor' ]
List placesOfBirth = (List)parser.parseExpression("Members.![placeOfBirth.city]");
```

地图也可以用来驱动投影和在这种情况下 投影计算表达式对每个条目在地图 (表示为一个Java . entry)。 的结果 投影在地图上是一个列表组成的评价 对每个地图投影表达条目。

8.5.18A表达式模板

表达式模板允许一个混合的字面文本与一个或 更多的评估模块。 每个评估块分隔与前缀 和后缀字符,您可以定义,一个共同的选择是使用 # { } 为分隔符。 例如,

```
String randomPhrase =
  parser.parseExpression("random number is #{T(java.lang.Math).random()}",
                    new TemplateParserContext()).getValue(String.class);

// evaluates to "random number is 0.7038186818312008"
```

字符串是评估通过连接文字文本的随机 数字与计算表达式的结果在# { } 分隔符,在这种情况下的结果,随机()方法调用。 这个 第二个参数的方法 parseExpression() 是 类型 ParserContext 。 这个 ParserContext 接口用于 影响表达式解析为支持 表达式模板功能。 的定义 TemplateParserContext 如下所示。

```
public class TemplateParserContext implements ParserContext {

  public String getExpressionPrefix() {
    return "#{";
  }

  public String getExpressionSuffix() {
    return "}";
  }
```

```
  public boolean isTemplate() {
    return true;
  }
}
```

8.6的例子中使用的类

发明家java

```
package org.spring.samples.spel.inventor;

import java.util.Date;
import java.util.GregorianCalendar;

public class Inventor {

  private String name;
  private String nationality;
  private String[] inventions;
  private Date birthdate;
  private PlaceOfBirth placeOfBirth;


  public Inventor(String name, String nationality)
  {
    GregorianCalendar c= new GregorianCalendar();
    this.name = name;
    this.nationality = nationality;
    this.birthdate = c.getTime();
  }
  public Inventor(String name, Date birthdate, String nationality) {
    this.name = name;
    this.nationality = nationality;
    this.birthdate = birthdate;
  }

  public Inventor() {
  }

  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public String getNationality() {
    return nationality;
  }
  public void setNationality(String nationality) {
    this.nationality = nationality;
  }
  public Date getBirthdate() {
    return birthdate;
  }
  public void setBirthdate(Date birthdate) {
    this.birthdate = birthdate;
  }
  public PlaceOfBirth getPlaceOfBirth() {
    return placeOfBirth;
  }
  public void setPlaceOfBirth(PlaceOfBirth placeOfBirth) {
    this.placeOfBirth = placeOfBirth;
  }
  public void setInventions(String[] inventions) {
    this.inventions = inventions;
  }
  public String[] getInventions() {
    return inventions;
  }
}
```

PlaceOfBirth.java

```
package org.spring.samples.spel.inventor;

public class PlaceOfBirth {

    private String city;
    private String country;

    public PlaceOfBirth(String city) {
        this.city=city;
    }
    public PlaceOfBirth(String city, String country)
    {
```

```
        this(city);
        this.country = country;
    }


    public String getCity() {
        return city;
    }
    public void setCity(String s) {
        this.city = s;
    }
    public String getCountry() {
        return country;
    }
    public void setCountry(String country) {
        this.country = country;
    }



}
```

社会java

```
package org.spring.samples.spel.inventor;

import java.util.*;

public class Society {

    private String name;

    public static String Advisors = "advisors";
    public static String President = "president";

    private List<Inventor> members = new ArrayList<Inventor>();
    private Map officers = new HashMap();

    public List getMembers() {
        return members;
    }

    public Map getOfficers() {
        return officers;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public boolean isMember(String name)
    {
        boolean found = false;
        for (Inventor inventor : members) {
            if (inventor.getName().equals(name))
            {
                found = true;
                break;
            }
        }
        return found;
    }


}
```

9。 一个面向方面的编程与弹簧

9.1一个介绍

面向方面编程 (AOP)补充 面向对象编程(OOP)通过提供另一种思维方式 关于程序结构。 关键单元的模块化在OOP是类, 而在AOP单元模块化是 方面 。 方面使模块化的问题如事务 管理,跨越多个类型和对象。 (这种忧虑是 通常称为 横切 担忧在AOP文学)。

的关键部件之一是春天了 aop 框架 。 虽然Spring IoC容器不依赖 AOP,这意味着您不需要使用AOP如果你不想,AOP 补充了Spring IoC提供一个非常能干的中间件 解决方案。

AOP是用于Spring框架来.....

- ... 提供声明性企业服务,特别是作为一个 替代EJB声明式服务。 最重要的 服务是 声明式事务 管理 。
- ... 允许用户实现自定义方面,以补充他们的 使用OOP和AOP。

如果你有兴趣只有在通用的声明式服务 或其他预包装的声明式中间件服务如池,你 不需要直接处理Spring AOP,可以跳过大部分的吗 一章。

Spring 2.0 AOP

Spring 2.0引入了一个更简单和更强大的写作方式 或者使用一个定制方面 基于 方法 或 @ aspectj注释 风格 。 这两种风格提供完全类型建议和使用 AspectJ切入点语言,同时仍然使用Spring AOP进行 编织。

Spring 2.0模式,讨论了基于@ aspectj AOP支持 在这一章。 Spring 2.0 AOP仍然完全向后兼容 Spring 1.2 AOP,低层AOP支持所提供的弹簧1.2 api了 以下 章 。

9.1.1A AOP概念

让我们首先定义一些中央AOP概念和 术语。 这些条款不是spring特定... 不幸的是,AOP 术语并不是非常直观的;然而,它甚至会 更令人困惑的如果弹簧使用自己的术语。

- 方面 :一个模块化的担忧 跨越多个类。 事务管理是一个很好的 的例子在企业Java应用程序的横切关注点。 在Spring AOP,方面使用常规类(基于方法)或定期 类注释与 @Aspect 注释(@ aspectj 风格)。
- 连接点 :一个点在执行 的一个程序,如执行一个方法或处理 一个例外。 在Spring AOP,连接点 总是 代表一个方法执行。
- 建议 :由一个方面采取行动在一个 特定的连接点。 不同类型的建议包括 “约” , “之前” 和 “之后” 的建议。 (建议类型下面讨论)。 许多的AOP框架,包括弹簧、模型一个建议 拦截器 ,维护一个链的 拦截器 周围 连接点。
- 切入点 :一个谓词匹配连接 点。 的建议是关联到一个切入点表达式和运行在 任何连接点的切入点匹配(例如,执行 的一个方法和一个特定的名称)。 连接点的概念作为 匹配切入点表达式是AOP的核心,弹簧使用 AspectJ切入点表达式语言默认情况下。
- 介绍 :声明额外 方法或字段代表一个类型。 Spring AOP允许你 引入新的接口(以及相应的实现)任何 建议对象。 例如,您可以使用一个介绍做一个 bean实现一个 IsModified 接口,简化缓存。 (介绍被称为 类型间声明在AspectJ社区。)
- 目标对象 :对象的咨询顾问 一个或多个方面。 也被称为 建议 对象。 因为Spring AOP实现 使用运行时代理,这个对象将永远是一个 代理 对象。
- AOP代理 :一个对象创建的AOP 框架为实现方面合同(建议方法 死刑等等)。 在Spring框架,AOP代理将 一个JDK动态代理或CGLIB代理。
- 编织 :链接方面与其他 应用程序类型或对象来创建一个建议的对象。 这可以 在编译时完成(使用AspectJ编译器为例), 加载时间,或在运行时。 Spring AOP,像其他纯Java AOP 框架,在运行时执行编织。

类型的建议:

- 建议之前 :建议执行 在一个连接点,但没有能力阻止 执行流继续连接点(除非它抛出一个 例外)。
- 回国后的建议 :建议是 完成后执行连接点通常:例如,如果一个 方法返回没有抛出异常。
- 在投掷的建议 :建议是 如果一个方法执行退出通过抛出异常。
- (最后)后建议 :建议是 执行无论手段连接点退出(正常 或异常返回)。
- Around通知 :建议包围 连接点比如方法调用。 这是最强大的 种建议。 在建议可以执行自定义行为之前和 在方法调用。 它还负责选择 是否继续进行连接点或快捷方式的建议 方法执行通过返回自己的返回值或抛出 例外。

约的建议是最一般类型的建议。 因为春天 AOP,像AspectJ,提供全方位的建议类型,我们建议 那你用最强大的建议类型,可以实现 所需的行为。 例如,如果您只需要更新一个缓存与 一个方法的返回值,因此你最好实现之后 返回的建议比周围的建议,尽管around通知可以 完成同样的事情。 使用最具体的类型提供建议 一个简单的编程模型和更少的潜在错误。 例如, 你不需要调用proceed() 方法 在 连接点 用于在建议, 因而不能失败来调用它。

在Spring 2.0中,所有的建议都是静态类型的参数,这样 和你一起工作的建议参数适当的类型(类型 返回值的方法执行例如)而不是 对象 数组。

连接点的概念,与切入点,是关键 AOP有别于旧技术只提供 拦截。 切入点使建议是独立的目标 面向对象的层次结构。 例如,一个在建议提供 声明式事务管理可以应用于一组方法 跨越多个对象(如所有业务操作 服务层)。

9.1.2A Spring AOP功能和目标

Spring AOP是用纯Java实现。 这里不需要 特殊的编译过程。 Spring AOP不需要控制 类加载器层次结构,因此是适合用于Servlet 容器或应用程序服务器。

目前只支持Spring AOP方法执行连接点 (建议执行方法的Spring bean)。 场拦截 还没有实现,尽管支持场拦截可以吗 添加不打破Spring AOP的核心api。 如果你需要建议 字段访问和更新的连接点,考虑之类的语言 AspectJ。

Spring AOP的AOP方法不同于大多数其他AOP 框架。 我们的目标不是要提供最完整的AOP 实现(虽然Spring AOP相当有能

力);而是去 提供一个紧密集成Spring AOP实现之间和奥委会 帮助解决常见的问题在企业应用程序。

因此,例如,Spring框架的AOP功能 通常结合使用Spring IoC容器。 方面 配置使用正常的bean定义语法(虽然这允许 强大的“自动代理”功能):这是一个关键的区别 其他的AOP实现。 有些事情你不能轻易或 有效地与Spring AOP,如建议非常细粒度的对象 (如域对象通常):AspectJ是最好的选择在这样的 情况下。 然而,我们的经验是,Spring AOP提供了一个优秀的 解决大多数问题,企业Java应用程序 服从AOP。

Spring AOP永远努力竞争,AspectJ提供 AOP解决方案。综合 我们相信这两个基于代理的框架 像Spring AOP和成熟的框架如AspectJ是有价值的, ,他们是互补的,而不是在竞争。 Spring 2.0 无缝地集成了Spring AOP和AspectJ和IoC,使所有 使用AOP提供的在一个一致的基于spring 应用程序体系结构。 这种集成不影响弹簧 AOP API或AOP联盟API:Spring AOP仍然是向后兼容的。 看到 下面的章节 对于一个 Spring AOP的讨论api。

注意

的一个中心原则的Spring框架的 具有非侵袭性 ,这是你的想法 不应该被迫介绍特定于框架的类和 接口业务/域模型。 然而,在一些地方 Spring框架确实给你选择介绍弹簧 特定于框架的依赖关系到你的代码库:基本原理在 给你这样的选项,因为在某些情况下它可能是 单纯的容易阅读或一些特定功能的代码块 在这样一种方式。 Spring框架(几乎)总是为你提供 选择虽然:你可以自由做出明智的决策, 哪个选项最适合您的特定用例或场景。

这样的一个选择,这一章是相关的 而AOP框架(AOP风格)选择。 你有 选择AspectJ和/或Spring AOP,你也有选择的 要么@ aspectj注释风格方法或Spring XML 方式方法。 事实上,这一章选择 介绍了@ aspectj风格方法首先不应该被当作一个 迹象表明,春季团队支持@ aspectj的注释风格 方法在Spring XML方式。

看到 SectionA 9.4,一个选择AOP声明使用风格 对于一个 更完整的讨论识大体、每个风格。

9.1.3A AOP代理

Spring AOP默认使用标准J2SE 动态 代理 对AOP代理。 这使得任何接口(或设置 接口)进行代理。

Spring AOP还可以使用CGLIB代理。 这是需要代理 类,而不是接口。 默认情况下使用CGLIB是如果一个企业 对象没有实现一个接口。 因为它是良好的实践 程序接口而不是类、业务类通常 将实现一个或多个业务接口。 它是可能的 力使用CGLIB ,在 那些(希望罕见的)情况下,你需要建议的方法 不上声明一个接口,或者你需要通过一个代理对象 到一个方法作为一个具体的类型。

它是重要的去把握事实是Spring AOP 基于代理的 。 看到 SectionA 9 6 1,一个理解AOP proxiesa 对于一个彻底的检查 正是这实际上意味着实现细节。

9.2一个@ aspectj支持

@ aspectj指的是一个风格的声明方面作为普通Java 类注释Java 5注释。 @ aspectj风格的 引入的 AspectJ 项目 AspectJ 5的一部分发布。 Spring 2.0解释 相同的注释为AspectJ 5、图书馆使用AspectJ提供的 切入点解析和匹配。 AOP运行时仍然是纯粹的Spring AOP 虽然,没有依赖AspectJ编译器或 韦弗。

使用AspectJ编译器和韦弗允许使用 全AspectJ语言,并讨论了 SectionA 9.8,一个使用AspectJ和弹簧applicationsa 。

9.2.1A启用@ aspectj支持

使用@ aspectj切面在Spring配置你需要的 使弹簧支持配置Spring AOP基于@ aspectj 方面,和 自动代理 豆子基于是否或 不建议他们通过那些方面。 通过自动代理我们意味着如果 弹簧确定bean是建议由一个或多个方面,它将 自动生成一个代理的bean来截距法 调用和确保的建议是根据需要执行。

@ aspectj的支持可以启用XML或Java风格 配置。 在这两种情况下你还需要确保 AspectJ的 aspectjweaver.jar 图书馆在你的应用程序的类路径(版本1 6 8或更高)。 这个图书馆是可用的 “自由” 目录的一个AspectJ分布或通过Maven中央存储库。

使@ aspectj支持Java配置

使@ aspectj支持使用Java @ configuration 添加 @EnableAspectJAutoProxy 注释:

```
@Configuration
@EnableAspectJAutoProxy
public class AppConfig {

}
```

使@ aspectj支持使用XML配置

使@ aspectj支持基于XML的配置与使用 aop:aspectj火狐的一个插件 元素:

```
<aop:aspectj-autoproxy/>
```

这个假设您正在使用模式中描述的支持 AppendixA E, XML的基于配置 。 看到 SectionA e 2 7,一个了 aop schemaa 如何导入标签 aop命名空间。

如果您正在使用DTD,仍有可能使@ aspectj 支持通过添加以下定义您的应用程序 背景:

```
<bean class="org.springframework.aop.aspectj.annotation.AnnotationAwareAspectJAutoProxyCreator" />
```

9.2.2A声明一个方面

与@ aspectj支持启用,任何bean中定义 应用程序上下文的类,是一个@ aspectj方面(有 @Aspect 注释)将自动 检测到春天和用于配置Spring AOP。 以下 示例显示所需的最小的定义不是很有用 方面:

一个正规的bean定义的应用程序上下文,指向 一个bean类的 @Aspect 注释:

```
<bean id="myAspect" class="org.xyz.NotVeryUsefulAspect">
  <!-- configure properties of aspect here as normal -->
</bean>
```

和 NotVeryUsefulAspect 类 定义,标注 org aspectj朗注释方面 注释;

```
package org.xyz;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class NotVeryUsefulAspect {

}
```

方面(类标注 @Aspect)可能有方法和字段只是 像任何其他类。 他们还可能包含切入点,建议,和 简介(类型间)声明。

Autodetecting方面通过组件扫描

你可以注册方面类在你的春天一般的咖啡豆 XML配置,或者自动侦测他们通过类路径扫描- 就像任何其他spring管理bean。 然而,请注意, @Aspect 注释是 不 足够的自动识别在类路径中:为此, 您需要添加一个单独的 component 注释 (或另外一个定义构造型注解,合格, 按规则Spring的组件扫描)。

建议方面与其他方面?

在Spring AOP,它是 不 可能有 方面自己是目标的其他方面的建议。 这个 @Aspect 注释在类标记为一个方面,因此排除了它从汽车代理。

9.2.3A声明一个切入点

回想一下,确定感兴趣的切入点连接点,因此 使我们能够控制当建议执行。 Spring AOP只有 支持方法执行连接点为Spring bean ,所以 你能想到的一个切入点匹配方法的执行 Spring bean。 一个切入点声明有两个部分:一个签名 由一个名称和任何参数,和一个切入点表达式 决定 到底 这方法执行我们吗 感兴趣的。 在AOP @ aspectj注释风格,一个切入点 签名是由常规方法定义和切入点 表达式是表示使用 @Pointcut 注释(该方法服务 作为切入点签名 必须 有一个 无效 返回类型)。

一个例子会使这个区别一个切入点 签名和一个切入点表达清楚。 下面的例子定义了 一个切入点命名 ” anyOldTransfer” 这将匹配 执行任何方法命名 “转让” :

```
@Pointcut("execution(* transfer(..))")// the pointcut expression
private void anyOldTransfer() {}// the pointcut signature
```

切入点表达式的值的形式 @Pointcut 注释是一个常规AspectJ 5切入点表达式。 对于一个完整的讨论AspectJ的切入点 语言,看到 AspectJ 编程指南 (和基于Java 5的扩展, AspectJ 5开发人员笔记本)或一个AspectJ的书籍等 一个 Eclipse AspectJ 一个等人通过Colyer或 一个 AspectJ在 行动 一个 Ramnivas Laddad通过。

支持切入点指示器

Spring AOP支持以下AspectJ切入点指示器 (PCD)用于切入点表达式:

- 执行 ——匹配方法 执行连接点,这是你的主要切入点指示器 将使用在使用Spring AOP吗
- 在 ——限制匹配连接点 在某些类型(简单地执行一个方法声明 在一个匹配类型在使用Spring AOP)
- 这 ——限制匹配连接点 (执行方法在使用Spring AOP)的bean 参考(Spring AOP代理)是给定的一个实例 类型
- 目标 ——限制匹配连接点 (执行方法在使用Spring AOP)目标 对象(应用程序对象代理)的一个实例 给定类型
- args ——限制匹配连接点 (执行方法在使用Spring AOP) 给出的实例的参数类型
- @ target时 ——限制匹配连接点(执行方法当 使用Spring AOP)类的对象有一个执行 注释指定类型的
- @args —— 限制匹配连接点(执行方法当 使用Spring AOP)在运行时类型的实际参数 通过给定的类型有标注(年代)
- @within ——限制匹配连接点类型内给定 注释(执行方法中声明的类型的 鉴于注释在使用Spring AOP)
- @annotation ——限制匹配的加入 点的主题(方法执行连接点 在Spring AOP)已给定的注释

其他切入点类型

完整的AspectJ切入点语言支持额外的 切入点指示器,在春天不支持。 这些都是: 电话,得到集,preinitialization,staticinitialization,初始化,处理程序,,,,adviceexecution withincode cflow cflowbelow,如果,@this ,和 @withincode 。 使用这些切入点指示器在切入点表达式 Spring AOP解释将导致一个 IllegalArgumentException 被扔。

这个组由Spring AOP切入点指示器可能 扩展在将来的版本中支持更多的AspectJ切入点 指示器。

因为Spring AOP限制只匹配方法执行 加入点,讨论的切入点指示器上面了 比你会发现窄的定义在AspectJ编程 指南。 此外,AspectJ本身基于类型的语义和在一个 执行连接点均 这 ” 和 ” 目标 “引用同一个对象——对象 执行该方法。 Spring AOP是一个基于代理的系统和 区分代理对象本身(绑定到 ” 这 ”)和目标对象背后的代理 (绑定到 “ 目标 ”)。

注意

由于Spring的基于代理的AOP框架性质, 受保护的方法被定义 不 拦截,既没有JDK代理(这并不适用) 也不是为CGLIB代理(这在技术上是可能的,但不 值得推荐的对AOP的目的)。 因此,任何给定的切入点 将匹配与 公共方法只有 !

如果你的拦截需求包括保护/私有方法 甚至是构造函数,考虑使用台弹力 原生AspectJ编织 相反 Spring的基于代理的AOP框架。 这就构成了一个不同的 AOP使用模式具有不同特点,所以一定要做 你自己熟悉织造前作出决定。

Spring AOP还支持额外的PCD命名 ” bean ” 。 这金刚石可以限制匹配 到一个特定的连接点名为Spring bean,或者一组命名的 Spring bean(当使用通配符)。 “ bean “金刚石 具有以下形式:

```
bean(idOrNameOfBean)
```

“ idOrNameOfBean “令牌可以被命名 任何Spring bean:有限的通配符支持使用 ” * 字符被提供,所以如果你建立 一些命名约定你的Spring bean,您可以很容易的 写一个 “ bean “金刚石表达式找出它们。 作为 情况与其他切入点指示器, ” bean “金刚石可以& &” || ed,” 艾德,! (否定)太。

注意

请注意, “ bean 的金刚石是 只有 支持Spring AOP和 不 在本机AspectJ编织。 这是一个 spring特定扩展 标准PCDs AspectJ 定义了。

“ bean “金刚石运行在 实例 水平(建立在春天 bean名称概念),而不是只在类型水平 (这是编织基于AOP是有限的)。 基于实例的切入点指示器是一个特殊的能力 Spring的基于代理的AOP框架及其紧密集成 与 Spring bean工厂,它是自然的和 简单的识别特定bean的名字。

结合切入点表达式

切入点表达式可以组合使用“& &”、“||” 和“!” 。 也可以参考切入点表达式的名字。 下面的例子显示了三个切入点表达式: anyPublicOperation (如果一个方法相匹配 执行连接点代表执行任何公共方法); inTrading (如果一个方法执行匹配 在交易模块), tradingOperation (如果一个方法执行匹配代表任何公共方法 交易模块)。

```
@Pointcut("execution(public * *(..))")
private void anyPublicOperation() {}
```

```
    @Pointcut("within(com.xyz.someapp.trading..*)")
    private void inTrading() {}

    @Pointcut("anyPublicOperation() && inTrading()")
    private void tradingOperation() {}
```

这是一个最佳实践,构建更复杂的切入点表达式 出的小名叫组件如上所示。 指 切入点的名字,正常的Java可见性规则应用(你可以看到 私人的切入点在同一类型、保护中的切入点 层次结构、公共切入点任何地方等等)。 能见度不 影响切入点 匹配 。

分享共同切入点定义

在使用企业应用程序,你经常想 指模块的应用程序和特定的操作集合 从几方面。 我们建议定义一个“SystemArchitecture”方面,抓住共同切入点表达式 为了这个目的。 一个典型的这样的方面看起来如下:

```
package com.xyz.someapp;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class SystemArchitecture {

  /**
   * A join point is in the web layer if the method is defined
   * in a type in the com.xyz.someapp.web package or any sub-package
   * under that.
   */
  @Pointcut("within(com.xyz.someapp.web..*)")
  public void inWebLayer() {}

  /**
   * A join point is in the service layer if the method is defined
   * in a type in the com.xyz.someapp.service package or any sub-package
   * under that.
   */
  @Pointcut("within(com.xyz.someapp.service..*)")
  public void inServiceLayer() {}

  /**
   * A join point is in the data access layer if the method is defined
   * in a type in the com.xyz.someapp.dao package or any sub-package
   * under that.
   */
  @Pointcut("within(com.xyz.someapp.dao..*)")
  public void inDataAccessLayer() {}

  /**
   * A business service is the execution of any method defined on a service
   * interface. This definition assumes that interfaces are placed in the
   * "service" package, and that implementation types are in sub-packages.
   *
   * If you group service interfaces by functional area (for example,
   * in packages com.xyz.someapp.abc.service and com.xyz.def.service) then
   * the pointcut expression "execution(* com.xyz.someapp..service.*.*(..))"
   * could be used instead.
   *
   * Alternatively, you can write the expression using the 'bean'
   * PCD, like so "bean(*Service)". (This assumes that you have
   * named your Spring service beans in a consistent fashion.)
   */
  @Pointcut("execution(* com.xyz.someapp.service.*.*(..))")
  public void businessService() {}

  /**
   * A data access operation is the execution of any method defined on a
   * dao interface. This definition assumes that interfaces are placed in the
   * "dao" package, and that implementation types are in sub-packages.
   */
  @Pointcut("execution(* com.xyz.someapp.dao.*.*(..))")
  public void dataAccessOperation() {}

}
```

切入点这样定义的一个方面,可以引用 任何地方,你需要一个切入点表达式。 例如,为了使 服务层的事务,你可以写:

```
<aop:config>
  <aop:advisor
      pointcut="com.xyz.someapp.SystemArchitecture.businessService()"
      advice-ref="tx-advice"/>
</aop:config>

<tx:advice id="tx-advice">
  <tx:attributes>
    <tx:method name="*" propagation="REQUIRED"/>
  </tx:attributes>
```

```
</tx:advice>
```

这个 < aop:配置> 和 < aop:顾问> 元素进行 SectionA 9.3,一个supporta基于AOP 。 讨论了事务元素 ChapterA 12, 事务管理 。

例子

Spring AOP用户可能使用 执行 切入点指示器最常。 这个 一个执行表达式的格式是:

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern)
        throws-pattern?)
```

所有部件除了返回类型模式(ret型模式在 上面的代码片段),名称模式和参数模式是可选的。 返回的类型模式决定了的返回类型方法必须为了一个连接点匹配。 大多数 你经常使用 * 作为返回类型 模式匹配的任何返回类型。 一个完全限定类型名称 将匹配只有在方法返回给定的类型。 这个名字 模式匹配方法名。 您可以使用 * 通配符是全部或部分名称模式。 参数模式是 稍微复杂的: () 匹配方法, 不带参数,而 (. .) 匹配任何 数目的参数(零或更多)。 该模式 (*) 以一个参数匹配方法的任何 类型, (*,字符串) 匹配方法采取两种 参数,第一可以是任何类型,第二必须是一个字符串。 咨询 语言语义 部分的AspectJ编程指南 为更多的信息。

一些常见的例子给出了切入点表达式 下面。

- 执行任何公共方法:

  ```
  execution(public * *(..))
  ```

- 任何方法的执行与一个名称开头 “设置”:

  ```
  execution(* set*(..))
  ```

- 执行定义的任何方法 AccountService 接口:

  ```
  execution(* com.xyz.service.AccountService.*(..))
  ```

- 执行任何方法中定义的服务 包:

  ```
  execution(* com.xyz.service.*.*(..))
  ```

- 执行任何方法中定义的服务包 或一个包里:

  ```
  execution(* com.xyz.service..*.*(..))
  ```

- 任何连接点(方法执行只有在Spring AOP)内 服务包:

  ```
  within(com.xyz.service.*)
  ```

- 任何连接点(方法执行只有在Spring AOP)内 服务包或包里:

  ```
  within(com.xyz.service..*)
  ```

- 任何连接点(方法执行只有在Spring AOP) 代理实现了 AccountService 接口:

  ```
  this(com.xyz.service.AccountService)
  ```

  “这更多的是常用的在一个绑定形式:- 看到下面的部分建议对于如何制作代理 对象可以在身体的建议。

- 任何连接点(方法执行只有在Spring AOP) 目标对象实现 AccountService 接口:

  ```
  target(com.xyz.service.AccountService)
  ```

  “目标” 更多地使用一个绑定形式:- 见以下部分的建议如何使目标 对象可以在身体的建议。

- 任何连接点(方法执行只有在Spring AOP) 接受一个参数,并在运行时传递的参数 是 可序列化的 :

  ```
  args(java.io.Serializable)
  ```

  “args” 更多地使用一个绑定形式:-看到 以下部分的建议对于如何制作方法参数 身体中可用的建议。
  注意,本例中给出的切入点是不同的 执行(* *(java io序列化)) :args 版本匹配如果在运行时传递的参数是可序列化的, 执行版本匹配如果方法签名声明 单一类型的参数 可序列化的 。

- 任何连接点(方法执行只有在Spring AOP) 目标对象有一个 transactional 注释:

```
@target(org.springframework.transaction.annotation.Transactional)
```

  “@ target时”也可以用于绑定形式:——看 以下部分的建议如何使注释 对象可以在身体的建议。
- 任何连接点(方法执行只有在Spring AOP) 声明的类型的目标对象有一个 transactional 注释:

```
@within(org.springframework.transaction.annotation.Transactional)
```

  “@within”也可以用于绑定形式:——看 以下部分的建议如何使注释 对象可以在身体的建议。
- 任何连接点(方法执行只有在Spring AOP) 执行方法 transactional 注释:

```
@annotation(org.springframework.transaction.annotation.Transactional)
```

  “@annotation”也可以用于绑定形式:- 见以下部分的建议如何使注释 对象可以在身体的建议。
- 任何连接点(方法执行只有在Spring AOP) 接受一个参数,并在运行时的类型 参数传递有 @Classified 注释:

```
@args(com.xyz.security.Classified)
```

  “@args”也可以用于绑定形式:——看 以下部分的建议如何使注释 对象(s)中可用的建议的身体。
- 任何连接点(方法执行只有在Spring AOP) Spring bean名为“ tradeService ”:

```
bean(tradeService)
```

- 任何连接点(方法执行只有在Spring AOP) Spring bean有名称,匹配通配符表达式” *服务 ”:

```
bean(*Service)
```

编写好的切入点

在编译期间,AspectJ切入点过程,为了优化匹配性能。 检查代码 和确定每个连接点匹配(静态或动态)给定的切入点是一个昂贵的过程。 (动态 比赛意味着匹配不能完全决定从静态分析和测试将放置在代码 决定是否有一个实际的比赛当代码运行)。 第一次遇到一个切入点声明, AspectJ将重写它到一个最佳状态匹配过程。 这是什么意思? 主要的切入点 是重写的析取范式(析取范式)和组件的切入点是排序,这样那些吗 组件是首先检查评估更便宜。 这意味着您不必担心理解 不同的切入点的性能指示器和可能供应以任何顺序排列在一个切入点声明。

然而,AspectJ只能使用它是什么告诉,为获得最佳性能的匹配你应该 想想他们正试图实现和缩小搜索空间尽可能匹配的 定义。 现有的指示器自然分为三组:燃起理智之火,范围和背景:

- 指示器燃起理智之火是那些选择特定类型的连接点。 例如:执行、获取、设置、电话,处理程序
- 范围指示器是那些选择一组加入的兴趣点(可能很多种类)。 例如:在,withincode
- 上下文指示器是那些匹配(和可选的绑定)基于上下文。 例如:这,目标,@annotation

一个写好的切入点应该尝试至少包括第一两种类型(燃起理智之火和范围),同时 上下文指示器可能包括如果希望匹配基于连接点上下文,或绑定,上下文 使用的建议。 仅仅提供要么燃起理智之火还是上下文指示器指示者将工作但 可能影响织造性能(时间和内存使用)由于所有的额外的处理和分析。 范围 指示器是非常快和他们的使用来匹配意味着AspectJ很快可以解散群 连接点,不应该被进一步加工——这就是为什么一个好的切入点应该包括 如果有可能的话。

9.2.4A声明的建议

的建议是关联到一个切入点表达式,并运行之前, 之后,或在方法执行匹配的切入点。 切入点 表达式可以是一个简单的参考为命名的切入点,或一个 切入点表达式中声明的地方。

建议之前

之前在一个方面建议申报使用 @ before一样 注释:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class BeforeExample {

  @Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
  public void doAccessCheck() {
    // ...
```

```
  }

}
```

如果使用一个合适的切入点表达式我们可以重写 以上的例子:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class BeforeExample {

  @Before("execution(* com.xyz.myapp.dao.*.*(..))")
  public void doAccessCheck() {
    // ...
  }

}
```

回国后的建议

回国后建议运行当匹配方法执行 返回正常。 这是宣布使用 @AfterReturning 注释:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterReturning;

@Aspect
public class AfterReturningExample {

  @AfterReturning("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
  public void doAccessCheck() {
    // ...
  }

}
```

注意:当然可能有多个建议 声明和其他成员,都在相同的方面。 我们只是显示一个建议在这些例子来。宣言 对这个问题的关注讨论的时间。

有时你需要访问的建议身体实际值 这是返回。 您可以使用的形式 @AfterReturning 这将返回 值:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterReturning;

@Aspect
public class AfterReturningExample {

  @AfterReturning(
    pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()",
    returning="retVal")
  public void doAccessCheck(Object retVal) {
    // ...
  }

}
```

名称中使用的 返回 属性必须 对应于一个参数的名称的建议方法。 当一个 方法执行返回,返回值将被传递到 建议方法相应的参数值。 一个 返回 条款也限制只匹配 这些方法执行,返回一个指定类型的价值 (对象 在这种情况下,它将匹配任何 返回值)。

请注意,这是 不 可能 返回一个完全不同的参考在使用后返回 建议。

在投掷的建议

在运行时抛出建议匹配方法执行退出 通过抛出异常。 这是宣布使用 @AfterThrowing 注释:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterThrowing;

@Aspect
public class AfterThrowingExample {

  @AfterThrowing("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
  public void doRecoveryActions() {
    // ...
  }

}
```

你经常想要建议只运行一个给定的时候例外 类型是扔,你也经常需要访问扔 异常的建议的身体。 使用 扔 属性既限制匹配(如果

需要,使用 Throwable 随着异常类型 否则),并将其绑定到一个忠告,抛出异常 参数。

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterThrowing;

@Aspect
public class AfterThrowingExample {

  @AfterThrowing(
    pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()",
    throwing="ex")
  public void doRecoveryActions(DataAccessException ex) {
    // ...
  }

}
```

名称中使用的 扔 属性必须 对应于一个参数的名称的建议方法。 当一个 方法执行退出通过抛出异常,异常 传递给建议方法相应的参数值。 一个 扔 条款也限制只匹配 这些方法执行,抛出一个异常指定类型的 (DataAccessException 在这种情况下)。

(最后)后建议

(最后)建议后运行一个匹配方法执行。然而 退出。 这是宣布使用 @After 注释。 建议后必须准备处理包括正常的和 异常返回条件。 它通常用于释放 资源等。

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.After;

@Aspect
public class AfterFinallyExample {

  @After("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
  public void doReleaseLock() {
    // ...
  }

}
```

Around通知

最后一种建议是around通知。 Around通知运行 “周围”的一个匹配的方法执行。 有机会去做的工作 前后方法执行,以确定何时、如何 即使,这个方法实际上可以执行在所有。 Around通知 是常用的如果你需要共享状态之前和之后的一个方法 在一个线程安全的方式执行(启动和停止一个定时器, 例)。 总是使用最强大的形式的建议,满足你 需求(即不使用around通知如果以前简单的建议 会做)。

在宣布使用建议 @Around 注释。 第一个参数 建议的方法必须的类型 ProceedingJoinPoint 。 主体内 的建议,称 proceed() 在 ProceedingJoinPoint 导致 底层方法来执行。 这个 进行 方法 也可以称为传入一个吗 Object[] —— 值在数组将作为参数的方法 执行当它收益。

进行的行为的时候叫 Object[]有点不同 的行为对周围进行建议编制的AspectJ 编译器。 建议写左右使用传统的AspectJ 语言,传递的参数的数量进行必须匹配 数目的参数传递到周围的建议(不是数量的 参数被底层连接点),值传递给 按照给定的参数位置取代原来的值 连接点的实体价值势必(别担心如果 这没有意义现在!)。 采用的方法是春天 简单的和一个更好的匹配它的基于代理的,只执行 语义。 你只需要知道这种差异如果你 编译@ aspectj切面写给春天和使用进行 参数与AspectJ编译器和韦弗。 有一个办法 写等方面,是100%兼容跨Spring AOP和 AspectJ,这是下一节中讨论的建议 参数。

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.ProceedingJoinPoint;

@Aspect
public class AroundExample {

  @Around("com.xyz.myapp.SystemArchitecture.businessService()")
  public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {
    // start stopwatch
    Object retVal = pjp.proceed();
    // stop stopwatch
    return retVal;
  }

}
```

返回的值在建议将返回值 被调用者的方法。 一个简单的缓存方面例如 可以返回一个值从一个缓存,如果它有一个、调用 proceed() 如果它不。 注意,进行可能调用一次,很多次,或 不是所有在身体周围的建议,所有这些都是 完全合法。

建议参数

Spring 2.0提供了充分的类型化的建议——这意味着你声明 你需要的参数在建议签名(正如我们看到的 上面的例子返回和投掷)而不是工作 Object[] 数组的所有时间。 我们将看到如何 使参数和其他上下文值提供给建议的身体 在一个时刻。 首先让我们看一看如何编写通用的建议 可以找出方法目前的建议 建议。

访问当前 连接点

任何建议方法可以宣布作为它的第一个参数,一个 类型的参数 org aspectj朗的连接点 (请 注意周围的建议是 需要 申报 第一个参数的类型 ProceedingJoinPoint ,这是一个 子类的 连接点 。 这个 连接点 接口提供了许多 有用的方法如 getArgs() (返回 方法参数), getThis() (返回 代理对象), getTarget() (返回 目标对象), getSignature() (返回一个 描述的方法是建议)和 toString() (打印一个有用的描述 该方法被建议)。 请查阅Javadocs完全 细节。

传递参数给建议

我们已经看到了如何绑定返回值或异常 值(使用后回国后和投掷的建议)。 让 参数值提供给建议的身体,你可以使用 绑定形式的 args 。 如果一个参数的名字是使用 代替一个类型名称的参数表达式,那么这个值的 相应的参数将作为参数传递价值 调用的建议。 一个例子应该使这个更清晰。 假设 你想建议的执行,拿一个dao操作 帐户对象作为第一个参数,你需要访问 账户在建议的身体。 您可以编写以下:

```
@Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation() &&" +
        "args(account,..)")
public void validateAccount(Account account) {
  // ...
}
```

这个 args(账户,. .) 部分的切入点 表达式有两个目的:首先,这限制了匹配 只有那些方法执行的方法需要至少一个 参数和参数传递给该参数是一个实例 的 帐户 ;其次,它使实际的 帐户 对象提供给建议通过 这个 帐户 参数。

另一种方式写这是声明一个切入点, “提供” 帐户 对象价值当它 匹配一个连接点,然后只是参考命名的切入点从 这个建议。 这将如下所示:

```
@Pointcut("com.xyz.myapp.SystemArchitecture.dataAccessOperation() &&" +
         "args(account,..)")
private void accountDataAccessOperation(Account account) {}

@Before("accountDataAccessOperation(account)")
public void validateAccount(Account account) {
  // ...
}
```

感兴趣的读者是再一次提到了AspectJ 编程指南为更多的细节。

代理对象(这),目标对象 (目标)、注释(@within, @ target时,@annotation,@args)都可以绑定在一个类似的 时尚。 下面的例子显示了如何匹配 执行的方法的一种 @Auditable 注释,并提取 审计代码。

第一个定义 @Auditable 注释:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Auditable {
    AuditCode value();
}
```

然后建议匹配的执行 @Auditable 方法:

```
@Before("com.xyz.lib.Pointcuts.anyPublicMethod() && " +
        "@annotation(auditable)")
public void audit(Auditable auditable) {
  AuditCode code = auditable.value();
  // ...
}
```

建议参数和泛型

Spring AOP可以处理泛型类声明和使用 方法参数。 假设您有一个泛型类型如下:

```
public interface Sample<T> {
  void sampleGenericMethod(T param);
  void sampleGenericCollectionMethod(Collection>T> param);
```

```
}
```

你可以限制拦截的方法对特定类型 只需输入参数类型参数的建议 参数类型你想要拦截的方法:

```
@Before("execution(* ..Sample+.sampleGenericMethod(*)) && args(param)")
public void beforeSampleMethod(MyType param) {
  // Advice implementation
}
```

这个作品是很明显的我们已经讨论过的 以上。 然而,需要指出的是,这个工作不了 泛型集合。 所以你不能定义切入点像 这个:

```
@Before("execution(* ..Sample+.sampleGenericCollectionMethod(*)) && args(param)")
public void beforeSampleMethod(Collection<MyType> param) {
  // Advice implementation
}
```

做这项工作我们会检查每一个元素的 集合,它是不合理的,因为我们还不能决定 治疗 空 值在一般。 实现 类似于这个你必须输入参数 收集< ? > 和手动 检查类型的元素。

确定参数名称

参数绑定在通知调用依赖于匹配 切入点表达式中使用的名称,名称声明的参数 (建议和切入点)方法签名。 参数名称 不 可以通过Java反射,所以 Spring AOP使用以下策略来确定参数 名称:

1. 如果参数名称由用户指定 明确,然后指定参数名称用:两个 建议和切点注释有一个可选的 “argNames”属性可以用来指定参数 带注释的方法的名字——这些参数名称 是 在运行时可用。 对于 示例:

   ```
   @Before(
     value="com.xyz.lib.Pointcuts.anyPublicMethod() && target(bean) && @annotation(auditable)",
     argNames="bean,auditable")
   public void audit(Object bean, Auditable auditable) {
    AuditCode code = auditable.value();
    // ... use code and bean
   }
   ```

   如果第一个参数的 连接点 , ProceedingJoinPoint ,或 JoinPoint.StaticPart 式,你 会省去参数的名字从价值的 “argNames”属性。 例如,如果您修改前面的 建议接收连接点的对象,“argNames” 属性不需要包括:

   ```
   @Before(
     value="com.xyz.lib.Pointcuts.anyPublicMethod() && target(bean) && @annotation(auditable)",
     argNames="bean,auditable")
   public void audit(JoinPoint jp, Object bean, Auditable auditable) {
    AuditCode code = auditable.value();
    // ... use code, bean, and jp
   }
   ```

   特殊优待的第一个参数 连接点 , ProceedingJoinPoint ,和 JoinPoint.StaticPart 类型是 特别是方便建议,不收取任何其他连接点上下文。 在这种情况下,你可以简单地省略了 “argNames”属性。 例如,下面的建议不需要 宣布“argNames”属性:

   ```
   @Before(
     "com.xyz.lib.Pointcuts.anyPublicMethod()")
   public void audit(JoinPoint jp) {
    // ... use jp
   }
   ```

2. 使用 ” argNames” 属性是一个 小笨手笨脚,所以如果 ” argNames” 属性 没有被指定,那么Spring AOP将着眼于调试吗 信息类,并试图确定参数 局部变量的名称表。 这些信息将 现在只要类被编译和调试 信息(' - g:增值” 在最低限度)。 这个 后果与这个国旗的编译是:(1)你的代码 会稍微容易理解(逆向),(2) 类文件大小将略微大(一般 无关紧要的),(3)优化以删除未使用的地方 变量不会被应用到你的编译器。 换句话说, 你应该遇到任何艰难建筑这个标志 在。
   如果一个@ AspectJ方面已编制的AspectJ 编译器(ajc)即使没有调试信息还有 不需要添加argNames属性的 编译器将保留所需的信息。
3. 如果代码被编译时没有必要的调试 信息,然后Spring AOP将试图推断出配对 绑定变量的参数(例如,如果只有一个 变量是绑定在切入点表达式,和建议 方法只接受一个参数,结对是显而易见的!)。 如果 绑定的变量是模糊给出可用的 信息,然后一个 AmbiguousBindingException 将 扔。
4. 如果上述所有策略都失败那么一个 IllegalArgumentException 将 扔。

进行参数

我们说,我们会早些时候描述如何编写一个 进行调用 与参数 工作 在Spring AOP和AspectJ一致。 解决方案是简单 确保建议签名绑定每个方法 参数的顺序。 例如:

```
@Around("execution(List<Account> find*(..)) &&" +
        "com.xyz.myapp.SystemArchitecture.inDataAccessLayer() && " +
        "args(accountHolderNamePattern)")
public Object preProcessQueryPattern(ProceedingJoinPoint pjp, String accountHolderNamePattern)
throws Throwable {
  String newPattern = preProcess(accountHolderNamePattern);
  return pjp.proceed(new Object[] {newPattern});
}
```

在很多情况下你会做这个绑定无论如何(如 上面的例子)。

建议订购

会发生什么当多个建议所有想要运行在 同样的连接点? Spring AOP遵循相同的优先规则 AspectJ来确定订单的建议执行。 最高的 优先建议首先运行的“在”(所以给两块 之前建议的,最高优先级运行第一)。 “在 出路”从一个连接点,最高优先级的跑最后的建议 (所以给两块的建议后,最高 优先将运行第二)。

当两块中定义的建议 不同 方面都需要运行在相同的 连接点,除非你指定否则执行顺序是 未定义的。 你可以控制执行的顺序指定 优先。 这样做是在正常的弹簧,要么 实施 org.springframework.core.Ordered 界面方面类或注释的 秩序 注释。 给定两个方面, 低价值方面返回从 下令getvalue() (或注释值) 优先级越高。

当两块中定义的建议 这个 相同 方面都需要运行在相同的连接点, 排序是未定义的(因为没有方法来检索 声明顺序通过反射对javac编译后的类)。 考虑 崩溃这样的建议方法到一个建议方法/连接点 在各个方面类或重构建议到单独的 方面可以订购类—这在方面水平。

9.2.5A介绍

介绍(称为类型间声明在AspectJ)启用 一个方面声明,建议对象执行一个给定的接口, 并提供一个实现该接口的代表 对象。

摘要提出一种使用 @DeclareParents 注释。 这 注释用于声明,匹配类型有一个新的父 (因此得名)。 例如,给定一个接口 UsageTracked ,和一个实现 该接口 DefaultUsageTracked ,以下 方面声明,所有服务接口的实现者也 实现 UsageTracked 接口。 (在 为了通过JMX公开统计数据为例。)

```
@Aspect
public class UsageTracking {

  @DeclareParents(value="com.xzy.myapp.service.*+",
                  defaultImpl=DefaultUsageTracked.class)
  public static UsageTracked mixin;

  @Before("com.xyz.myapp.SystemArchitecture.businessService() &&" +
          "this(usageTracked)")
  public void recordUsage(UsageTracked usageTracked) {
    usageTracked.incrementUseCount();
  }

}
```

接口的实现是由这个类型的 带注释的字段。 这个 价值 属性的 @DeclareParents 注释是一个AspectJ 类型模式:-任何bean的一个匹配的类型将会实现 UsageTracked接口。 注意,在上面的之前的建议 例,服务bean可以直接用作实现的 UsageTracked 接口。 如果访问 你会写编程bean以下:

```
UsageTracked usageTracked = (UsageTracked) context.getBean("myService");
```

9.2.6A方面的实例化模型

(这是一个高级主题,所以如果你只是开始 AOP你可以跳过它,直到后来。)

默认情况下会有一个单一的实例,在每一个方面 应用程序上下文。 AspectJ调用这个singleton实例化 模型。 可以定义方面,以替代生命周期:- Spring支持AspectJ的 关于perthis 和 pertarget 实例化模型(percflow, percflowbelow, 和 pertypewithin 不是 目前支持)。

一个“关于perthis”方面是宣布通过指定一个 关于perthis 条款 @Aspect 注释。 让我们看一个 例子,然后我们将解释它是如何工作的。

```
@Aspect("perthis(com.xyz.myapp.SystemArchitecture.businessService())")
public class MyAspect {

  private int someState;

  @Before(com.xyz.myapp.SystemArchitecture.businessService())
  public void recordServiceUsage() {
    // ...
  }

}
```

的效果 '关于perthis" 条款是一个 方面的实例将被创建为每个独特的服务对象执行 一个业务服务(每个独特的对象绑定到 '这' 在连接点 切入点表达式相匹配)。 方面的实例被创建 第一次一个方法被调用的服务对象。 方面 超出范围当服务对象超出范围。 之前 方面的实例被创建,没有建议在它执行。 作为 一旦创建了实例方面,建议内部声明 它将执行在匹配的连接点,但只有当服务对象 是一个这方面有关。 看到AspectJ编程 更多的信息来指导每条款。

这个 "pertarget" 实例化模型工作在 关于perthis完全一样,但创建一个方面的实例 每一个独特的目标对象在匹配连接点。

9.2.7A例子

现在,您已经看到了如何工作的所有组成部分,让我们 将它们组合在一起来做一些有用的!

执行业务服务可以有时失败由于 并发性问题(例如,僵局失败者)。 如果操作是 重试,它很可能下一次成功。 对于业务 服务那里是合适的,在这样的条件下重试(幂等 操作,不需要返回到用户冲突 分辨率),我们想透明地重试操作来避免 客户看到 PessimisticLockingFailureException 。 这是一个 要求清楚地穿过多个服务在服务 层,因此是理想的实现通过一个方面。

因为我们想重试操作,我们将需要使用约 建议,这样我们可以调用多次进行。 下面是基本的 方面实现看起来:

```
@Aspect
public class ConcurrentOperationExecutor implements Ordered {

  private static final int DEFAULT_MAX_RETRIES = 2;

  private int maxRetries = DEFAULT_MAX_RETRIES;
  private int order = 1;

  public void setMaxRetries(int maxRetries) {
    this.maxRetries = maxRetries;
  }

  public int getOrder() {
    return this.order;
  }

  public void setOrder(int order) {
    this.order = order;
  }

  @Around("com.xyz.myapp.SystemArchitecture.businessService()")
  public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {
    int numAttempts = 0;
    PessimisticLockingFailureException lockFailureException;
    do {
      numAttempts++;
      try {
        return pjp.proceed();
      }
      catch(PessimisticLockingFailureException ex) {
        lockFailureException = ex;
      }
    }
    while(numAttempts <= this.maxRetries);
    throw lockFailureException;
  }

}
```

注意方面实现了 下令 所以我们可以设置界面 的优先级高于事务方面的建议(我们想要一个 新鲜的事务我们每一次重试)。 这个 maxRetries 和 秩序 属性都需要配置的 春天。 主要的行动发生在 doConcurrentOperation 约的建议。 注意,对于 目前我们应用重试逻辑所有 businessService()年代 。 我们试着继续,如果我们失败了 与一个 PessimisticLockingFailureException 我们 只是试一次,除非我们已经用尽了所有的重试 尝试。

相应的Spring配置是:

```
<aop:aspectj-autoproxy/>

<bean id="concurrentOperationExecutor"
```

```
 class="com.xyz.myapp.service.impl.ConcurrentOperationExecutor">
   <property name="maxRetries" value="3"/>
   <property name="order" value="100"/>
</bean>
```

完善的方面,这样它只重试幂等 操作,我们可以定义一个 幂等 注释:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Idempotent {
  // marker annotation
}
```

和使用注释来注释实现的服务 操作。 更改方面只有重试幂等操作 只是涉及炼油切入点表达式,因此只有 @Idempotent 操作匹配:

```
@Around("com.xyz.myapp.SystemArchitecture.businessService() && " +
        "@annotation(com.xyz.myapp.service.Idempotent)")
public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {
  ...
}
```

9.3基于AOP支持

如果你不能使用Java 5,或者只是喜欢一个基于xml的 格式,然后Spring 2.0还提供了支持定义方面使用 新的“aop”名称空间标签。 完全相同的切入点表达式和建议 支持各种当使用@ aspectj风格,因此在这 一节中,我们将关注新 语法 并参考了 读者在前一节的讨论(SectionA 9.2,一个supporta@ aspectj)了解写作的切入点 表情和绑定参数的建议。

使用aop命名空间标记描述了在这一节中,您需要 导入spring aop模式中描述 AppendixA E, XML的基于配置 。 看到 SectionA e 2 7,一个了 aop schemaa 如何导入标记 aop命名空间。

在Spring配置,所有方面和顾问的元素 必须被放置在一个吗 < aop:配置> 元素 (你可以有一个以上的 < aop:配置> 元素 在一个应用程序上下文配置)。 一个 < aop:配置> 元素可以包含切入点, 顾问,元素(注意这些方面必须声明在那 顺序)。

> **警告**
>
> 这个 < aop:配置> 风格的配置 大量使用Spring的 汽车代理 机制。 这可能会导致 问题(如建议没有被编织) 如果你已经在使用 显式汽车代理通过使用 BeanNameAutoProxyCreator 或者诸如此类的。 这个 推荐使用模式是使用要么只是 < aop:配置> 风格,或只是 AutoProxyCreator 风格。

9.3.1A声明一个方面

使用模式的支持,一个方面是一个简单的普通Java 对象定义为一个bean在Spring应用程序上下文。 国家 和行为是捕获对象的字段和方法, 切入点和建议信息捕获在XML。

一个方面是宣布使用< aop:方面>元素,和 支持bean的引用使用 Ref 属性:

```
<aop:config>
 <aop:aspect id="myAspect" ref="aBean">
  ...
 </aop:aspect>
</aop:config>

<bean id="aBean" class="...">
 ...
</bean>
```

bean支持方面(“ aBean “在这个 例)当然可以被配置和依赖注入的一样 其他Spring bean。

9.3.2A声明一个切入点

一个名叫切入点可以被声明在一个< aop:配置> 元素,使切入点定义共享跨几个 方面和顾问。

一个切入点代表执行任何业务服务 服务层可以定义如下:

```
<aop:config>

 <aop:pointcut id="businessService"
     expression="execution(* com.xyz.myapp.service.*.*(..))"/>

</aop:config>
```

注意,切入点表达式本身是使用相同的AspectJ 切入点表达式语言中描述 SectionA 9.2,一个supporta@ aspectj 。 如果您使用的是基于模式 声明风格与Java 5,您可以参考切入点定义命名 在类型(@Aspects)在切入点表达式,但是这种特性 不可以在JDK 1.4及以下(它依赖Java 5具体 AspectJ反射api)。 在JDK 1.5因此,另一种定义 上面的切入点是:

```
<aop:config>

  <aop:pointcut id="businessService"
      expression="com.xyz.myapp.SystemArchitecture.businessService()"/>

</aop:config>
```

假设你有一个 SystemArchitecture 方面 中描述的 一个章节分享共同的切入点definitionsa 。

声明一个切入点在一个方面是非常相似的声明 一个顶级的切入点:

```
<aop:config>

  <aop:aspect id="myAspect" ref="aBean">

    <aop:pointcut id="businessService"
        expression="execution(* com.xyz.myapp.service.*.*(..))"/>

    ...

  </aop:aspect>

</aop:config>
```

同样的方式在一个@ aspectj方面,切入点声明的使用 基于模式的定义风格可能收集连接点上下文。 对于 例如,下面的“本”的切入点收集对象作为加入 点上下文并将其传递到建议:

```
<aop:config>

  <aop:aspect id="myAspect" ref="aBean">

    <aop:pointcut id="businessService"
        expression="execution(* com.xyz.myapp.service.*.*(..)) &amp;amp;&amp;amp; this(service)"/>
    <aop:before pointcut-ref="businessService" method="monitor"/>
    ...

  </aop:aspect>

</aop:config>
```

必须声明的建议得到收集的连接点 上下文通过包括参数匹配的名字:

```
public void monitor(Object service) {
    ...
}
```

当结合切入点子表达式,“& &”是尴尬的 在一个XML文档,所以关键词”和“,”或“和”不是“可以 用来代替“& &”、“| |”和“! 分别的。 例如, 前面的切入点可能更好的写为:

```
<aop:config>

  <aop:aspect id="myAspect" ref="aBean">

    <aop:pointcut id="businessService"
        expression="execution(* com.xyz.myapp.service.*.*(..)) and this(service)"/>
    <aop:before pointcut-ref="businessService" method="monitor"/>
    ...

  </aop:aspect>

</aop:config>
```

注意,在这种方式的切入点定义引用他们 XML id和不能被用作命名为切入点,形成复合 切入点。 指定切入点支持基于模式的定义 风格就更有限了,@ aspectj 风格。

9.3.3A声明的建议

相同的五个建议类型的支持对于@ aspectj 风格,和他们有相同的语义。

建议之前

建议在匹配之前运行方法执行。 这是 宣布在一个 < aop:方面> 使用 < aop:在>元素。

```
<aop:aspect id="beforeExample" ref="aBean">

  <aop:before
    pointcut-ref="dataAccessOperation"
    method="doAccessCheck"/>

  ...

</aop:aspect>
```

这里 dataAccessOperation 是的id吗 切入点定义在顶部(< aop:配置>) 水平。 定义切入点内联相反,取代 切入点ref 属性与一个 切入点 属性:

```
<aop:aspect id="beforeExample" ref="aBean">

  <aop:before
    pointcut="execution(* com.xyz.myapp.dao.*.*(..))"
    method="doAccessCheck"/>

  ...

</aop:aspect>
```

当我们在讨论中提到的@ aspectj风格,使用命名 切入点可以大大提高你的可读性 代码。

方法属性识别方法 (doAccessCheck),提供身体的 建议。 这个方法必须定义bean的引用 方面的元素,其中包含的建议。 在一个数据访问操作 是执行(一个方法执行连接点匹配的切入点 表达式),“doAccessCheck”方法的方面上的bean 调用。

回国后的建议

回国后建议运行当匹配方法执行 正常完成。 它是在一个声明 < aop:方面> 在像以前那样 建议。 例如:

```
<aop:aspect id="afterReturningExample" ref="aBean">

  <aop:after-returning
    pointcut-ref="dataAccessOperation"
    method="doAccessCheck"/>

  ...

</aop:aspect>
```

正如在@ aspectj风格,可以拿到的 返回值在建议的身体。 使用返回的属性 指定名称的参数,返回值应该是 通过:

```
<aop:aspect id="afterReturningExample" ref="aBean">

  <aop:after-returning
    pointcut-ref="dataAccessOperation"
    returning="retVal"
    method="doAccessCheck"/>

  ...

</aop:aspect>
```

doAccessCheck的方法必须声明一个参数命名的 retval 。 这个参数约束的类型 匹配相同的方式为@AfterReturning描述。 对于 例,方法签名可以声明为:

```
public void doAccessCheck(Object retVal) {...
```

在投掷的建议

时,就会执行后扔建议匹配的方法执行 出口通过抛出异常。 它是在一个声明 < aop:方面> 使用后扔 元素:

```
<aop:aspect id="afterThrowingExample" ref="aBean">

  <aop:after-throwing
    pointcut-ref="dataAccessOperation"
    method="doRecoveryActions"/>

  ...

</aop:aspect>
```

正如在@ aspectj风格,可以拿到的 抛出异常在建议的身体。 使用投掷属性 指定名称的参数的异常应 通过:

```
<aop:aspect id="afterThrowingExample" ref="aBean">

  <aop:after-throwing
    pointcut-ref="dataAccessOperation"
    throwing="dataAccessEx"
    method="doRecoveryActions"/>

  ...

</aop:aspect>
```

doRecoveryActions的方法必须声明一个参数命名的 dataAccessEx 。 这个参数约束的类型 匹配相同的方式为@AfterThrowing描述。 例如, 方法签名可以声明为:

```
public void doRecoveryActions(DataAccessException dataAccessEx) {...
```

(最后)后建议

(最后)建议后运行一个匹配方法执行。然而 退出。 这是宣布使用 在 元素:

```
<aop:aspect id="afterFinallyExample" ref="aBean">

  <aop:after
    pointcut-ref="dataAccessOperation"
    method="doReleaseLock"/>

  ...

</aop:aspect>
```

Around通知

最后一种建议是around通知。 Around通知运行 “周围” 的一个匹配的方法执行。 有机会去做的工作 前后方法执行,以确定何时、如何 即使,这个方法实际上可以执行在所有。 Around通知 是常用的如果你需要共享状态之前和之后的一个方法 在一个线程安全的方式执行(启动和停止一个定时器, 例)。 总是使用最强大的形式的建议,满足你 要求;不要使用around通知如果以前简单的建议 做的。

在宣布使用建议 aop:约 元素。 的第一个参数 建议的方法必须是类型 ProceedingJoinPoint 。 主体内 的建议,称 proceed() 在 ProceedingJoinPoint 导致 底层方法来执行。 这个 进行 方法 也可以调用传入一个吗 Object[] —— 数组中的值将作为参数的方法 执行当它收益。 看到 一个章节advicea周围 对笔记进行调用 与一个 Object[] 。

```
<aop:aspect id="aroundExample" ref="aBean">

  <aop:around
    pointcut-ref="businessService"
    method="doBasicProfiling"/>

  ...

</aop:aspect>
```

实施 doBasicProfiling 的建议是完全一样的(减去@ aspectj的例子 注释当然):

```
public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {
  // start stopwatch
  Object retVal = pjp.proceed();
  // stop stopwatch
  return retVal;
}
```

建议参数

基于模式的支持完全类型声明风格建议 在同样的方式描述的支持,通过匹配@ aspectj 切入点参数的名字对方法参数的建议。 看到 一个章节parametersa建议 详情。 如果你 想要显式地指定参数名称(不建议方法 依靠检测策略之前所描述的),那么这是通过使用 自变量名称 属性的建议 元素,它是在相同的方式对待 “argNames” 属性在一个注释中描述的建议 一个章节确定参数namesa 。 例如:

```
<aop:before
 pointcut="com.xyz.lib.Pointcuts.anyPublicMethod() and @annotation(auditable)"
 method="audit"
 arg-names="auditable"/>
```

这个 自变量名称 属性接受 用逗号分隔的参数名称的列表。

在下面发现一个稍复杂一点的例子基于xsd 的方法,说明了一些周围的建议结合使用 一个数量的强类型的参数。

```
package x.y.service;

public interface FooService {

  Foo getFoo(String fooName, int age);
}

public class DefaultFooService implements FooService {

  public Foo getFoo(String name, int age) {
    return new Foo(name, age);
  }
}
```

接下来是方面。 注意到的事实 配置文件(. .) 方法接受一个数量的 强类型的参数,第一个恰好是连接 点用来进行方法调用:的出现 参数是一个迹象表明 配置文件(. .) 是被用作吗 周围 建议:

```
package x.y;

import org.aspectj.lang.ProceedingJoinPoint;
import org.springframework.util.StopWatch;

public class SimpleProfiler {

  public Object profile(ProceedingJoinPoint call, String name, int age) throws Throwable {
    StopWatch clock = new StopWatch(
        "Profiling for '" + name + "' and '" + age + "'");
    try {
      clock.start(call.toShortString());
      return call.proceed();
    } finally {
      clock.stop();
      System.out.println(clock.prettyPrint());
    }
  }
}
```

最后,这是XML配置需要 影响执行上面的建议为特定的加入 点:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd">

  <!-- this is the object that will be proxied by Spring's AOP infrastructure -->
  <bean id="fooService" class="x.y.service.DefaultFooService"/>

  <!-- this is the actual advice itself -->
  <bean id="profiler" class="x.y.SimpleProfiler"/>

  <aop:config>
    <aop:aspect ref="profiler">

      <aop:pointcut id="theExecutionOfSomeFooServiceMethod"
              expression="execution(* x.y.service.FooService.getFoo(String,int))
              and args(name, age)"/>

      <aop:around pointcut-ref="theExecutionOfSomeFooServiceMethod"
            method="profile"/>

    </aop:aspect>
  </aop:config>

</beans>
```

如果我们有以下驱动脚本,我们会得到输出 这样的事情在标准输出:

```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import x.y.service.FooService;

public final class Boot {

  public static void main(final String[] args) throws Exception {
    BeanFactory ctx = new ClassPathXmlApplicationContext("x/y/plain.xml");
    FooService foo = (FooService) ctx.getBean("fooService");
    foo.getFoo("Pengo", 12);
```

```
  }
}
```

```
StopWatch 'Profiling for 'Pengo' and '12'': running time (millis) = 0
---------------------------------------
ms     %     Task name
---------------------------------------
00000  ?  execution(getFoo)
```

建议订购

当多个建议需要执行相同的连接点 (执行方法)排序规则中描述 一个章节orderinga建议 。 之间的优先 方面是由要么添加 秩序注释到bean的支持 方面或通过bean实现 下令 接口。

9.3.4A介绍

介绍(称为类型间声明在AspectJ)启用 一个方面声明,建议对象执行一个给定的接口, 并提供一个实现该接口的代表 对象。

摘要提出一种使用 aop:声明的父母 元素在一个 aop:方面 这个元素用来声明 匹配类型有一个新的父(因此得名)。 例如,假定有一个 接口 UsageTracked ,和一个 实现该接口的 DefaultUsageTracked ,以下方面 声明,所有服务接口的实现者也实现了 UsageTracked 接口。 (为了 通过JMX公开统计数据为例。)

```
<aop:aspect id="usageTrackerAspect" ref="usageTracking">

  <aop:declare-parents
     types-matching="com.xzy.myapp.service.*+"
     implement-interface="com.xyz.myapp.service.tracking.UsageTracked"
     default-impl="com.xyz.myapp.service.tracking.DefaultUsageTracked"/>

  <aop:before
   pointcut="com.xyz.myapp.SystemArchitecture.businessService()
          and this(usageTracked)"
   method="recordUsage"/>

</aop:aspect>
```

类支持 usageTracking bean将 控制方法:

```
public void recordUsage(UsageTracked usageTracked) {
   usageTracked.incrementUseCount();
}
```

接口的实现是由 实现接口 属性。 的价值 类型匹配 属性是一个AspectJ类型模式 :-任何bean的一个匹配的类型将会实现 UsageTracked 接口。 注意,在 在以上示例的建议,服务bean可以直接使用 作为实现的 UsageTracked 接口。 如果访问bean 以编程方式你会写 以下:

```
UsageTracked usageTracked = (UsageTracked) context.getBean("myService");
```

9.3.5A方面的实例化模型

唯一支持实例化模型方案定义方面 是单例模式。 其他实例化模型可以支持 将来的版本中。

9.3.6A顾问

“顾问” 的概念提出了从AOP支持 在Spring 1.2中定义并没有直接等效在AspectJ。 一个顾问就像一个小的独立的方面,都有一个单独的块 的建议。 这个建议本身就是一个由豆,和必须 实现一个接口描述的建议 SectionA 10 3 2,一个建议类型Springa 。 顾问可以利用 AspectJ切入点表达式虽然。

Spring 2.0支持顾问的概念 < aop:顾问> 元素。 你会最常见的 看到它结合使用事务的建议,也有它 自己的名称空间支持 Spring 2.0中。 这是它的样子:

```
<aop:config>

  <aop:pointcut id="businessService"
      expression="execution(* com.xyz.myapp.service.*.*(..))"/>

  <aop:advisor
     pointcut-ref="businessService"
     advice-ref="tx-advice"/>

</aop:config>
```

```
<tx:advice id="tx-advice">
  <tx:attributes>
    <tx:method name="*" propagation="REQUIRED"/>
  </tx:attributes>
</tx:advice>
```

以及 切入点ref 属性用于 上面的例子,你也可以使用 切入点 属性 定义一个切入点表达式内联。

定义优先级的一个顾问,所以建议可以 参与排序,使用 秩序 属性 定义 下令 价值的顾问。

9.3.7A例子

让我们来看看并发锁失败重试的例子 SectionA 9 2 7,一个Examplea 看起来当重写使用 模式支持。

执行业务服务可以有时失败由于 并发性问题(例如,僵局失败者)。 如果操作是 重试,它很有可能下次会成功的。 对于 业务服务 其适合于重试在这样的条件下 (幂等操作,不需要返回到用户 冲突解决),我们想透明地重试操作 避免客户看到 PessimisticLockingFailureException 。 这是一个 要求清楚地穿过多个服务在服务 层,因此是理想的实现通过一个方面。

因为我们想重试操作,我们将需要使用约 建议,这样我们可以调用多次进行。 下面是基本的 方面实现看起来(它只是一个普通的 Java类使用 模式支持):

```
public class ConcurrentOperationExecutor implements Ordered {

  private static final int DEFAULT_MAX_RETRIES = 2;

  private int maxRetries = DEFAULT_MAX_RETRIES;
  private int order = 1;

  public void setMaxRetries(int maxRetries) {
    this.maxRetries = maxRetries;
  }

  public int getOrder() {
    return this.order;
  }

  public void setOrder(int order) {
    this.order = order;
  }

  public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {
    int numAttempts = 0;
    PessimisticLockingFailureException lockFailureException;
    do {
      numAttempts++;
      try {
        return pjp.proceed();
      }
      catch(PessimisticLockingFailureException ex) {
        lockFailureException = ex;
      }
    }
    while(numAttempts <= this.maxRetries);
    throw lockFailureException;
  }

}
```

注意方面实现了 下令 所以我们可以设置界面 的优先级高于事务方面的建议(我们想要一个 新鲜的事务我们每一次重试)。 这个 maxRetries 和 秩序 属性都需要配置的 春天。 主要的行动发生在 doConcurrentOperation 在建议的方法。 我们试图 接下来,如果我们不能用 PessimisticLockingFailureException 我们只是尝试 我们已经筋疲力尽了,除非我们所有的重试尝试。

这个类是相同的,使用@ aspectj的例子, 但随着注释删除。

相应的Spring配置是:

```
<aop:config>

  <aop:aspect id="concurrentOperationRetry" ref="concurrentOperationExecutor">

    <aop:pointcut id="idempotentOperation"
      expression="execution(* com.xyz.myapp.service.*.*(..))"/>

    <aop:around
      pointcut-ref="idempotentOperation"
      method="doConcurrentOperation"/>

  </aop:aspect>
```

```
</aop:config>

<bean id="concurrentOperationExecutor"
 class="com.xyz.myapp.service.impl.ConcurrentOperationExecutor">
   <property name="maxRetries" value="3"/>
   <property name="order" value="100"/>
</bean>
```

注意,这里我们假设所有的业务 服务是等幂的。 如果不是这种情况我们可以改进 方面,只能重试真正幂等操作,通过 引入一个 幂等 注释:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Idempotent {
  // marker annotation
}
```

和使用注释来注释实现的服务 操作。 更改方面只有幂等操作重试 只是涉及炼油切入点表达式,因此只有 @Idempotent 操作匹配:

```
<aop:pointcut id="idempotentOperation"
    expression="execution(* com.xyz.myapp.service.*.*(..)) and
            @annotation(com.xyz.myapp.service.Idempotent)"/>
```

9.4一个选择AOP声明样式使用

一旦你已经决定,一个方面是最好的方法 实现一个给定的要求,如何决定使用Spring之间 AOP和AspectJ之间,方面语言(代码)风格,@ AspectJ 注释风格,或Spring XML风格吗? 这些决定的影响 由多种因素,包括应用程序的需求,发展 工具和团队熟悉AOP。

9.4.1A Spring AOP和AspectJ完整吗?

使用最简单的事情可以工作。 Spring AOP是简单的比 使用全AspectJ是没有要求介绍AspectJ 编译器/韦弗进入您的开发和构建过程。 如果你只 需要建议执行操作在Spring bean,那么春天 AOP是正确的选择。 如果你需要建议对象进行管理 Spring容器(如域对象一般),然后你会 需要使用AspectJ。 你还需要使用AspectJ如果你希望 建议加入点除了简单的方法执行(例如, 字段获取或设置的连接点,等等)。

当使用AspectJ,您可以选择AspectJ语言 语法(也称为“代码风格”)或@ aspectj注释 风格。 显然,如果你不使用Java 5 +然后别无选择 为你做... 使用代码风格。 如果方面起到了很大的作用在你的 设计,你可以使用 AspectJ开发工具 (AJDT) Eclipse插件,然后AspectJ语言语法 优先选择:它是清洁和简单的,因为语言是 故意设计为写作方面。 如果你不使用Eclipse, 或只有几个方面不扮演重要角色在你的 应用程序,那么您可能需要考虑使用@ aspectj风格和 坚持常规Java编译在IDE,并添加一个 方面编织阶段构建脚本。

9.4.2A @ aspectj或XML为Spring AOP吗?

如果你选择使用Spring AOP,然后你有一个选择 @ aspectj或XML风格。 显然如果你不运行在Java 5 +,然后 XML风格是合适的选择,因为Java 5项目有 各种权衡考虑。

XML风格将对现有Spring用户最熟悉。 它 可以使用任何的JDK级别(指从内部命名为切入点 切入点表达式并仍然需要Java 5 +虽然)和后盾 真正的pojo。 当使用AOP作为一种工具来配置企业服务 然后XML可以是一个不错的选择(一个好的测试是您是否考虑 切入点表达式的一部分,您的配置你可能想 改变独立)。 与XML风格可以说是清晰 您的配置中存在哪些方面的系统。

XML风格有两个缺点。 首先它不完全 封装的实现要求它地址的 单一的地方。 干燥原理说,应该有一个单独的, 明确的,权威的表示知识的任何一块 在一个系统。 当使用XML样式,知识的 如何 一个要求是实现跨越 声明支持bean的类和XML的 配置文件。 当使用@ aspectj风格有一个单一的 模块-方面——信息封装。 其次,XML样式稍微限制它可以表达 比@ aspectj风格:只有“单体”方面的实例化模型 是支持,是不可能结合的切入点声明命名 在XML。 例如,在@ aspectj风格你可以写点东西 如:

```
@Pointcut(execution(* get*()))
public void propertyAccess() {}

@Pointcut(execution(org.xyz.Account+ *(..))
public void operationReturningAnAccount() {}

@Pointcut(propertyAccess() && operationReturningAnAccount())
public void accountPropertyAccess() {}
```

在XML样式我可以声明前两个切入点:

```
<aop:pointcut id="propertyAccess"
```

```
    expression="execution(* get*())"/>

<aop:pointcut id="operationReturningAnAccount"
    expression="execution(org.xyz.Account+ *(..))"/>
```

XML方法的缺点是,你不能定义 " accountPropertyAccess "切入点通过结合这些 定义。

@ aspectj风格的支持额外的实例化模型, 富裕切入点组合。 它的优势是保持方面 作为一个模块化的单位。 它也有优势的@ aspectj切面可以 理解(因此消耗)都通过Spring AOP和AspectJ -所以如果 你以后决定你需要实现的功能,AspectJ 额外的需求 然后很容易迁移到一个 基于aspectj的方法。 在平衡弹簧团队喜欢@ aspectj 只要你有风格方面,远不止简单的"配置" 企业服务。

9.5混合类型方面

它是完全有可能混合风格方面的@ aspectj使用 自动代理支持,方案定义 < aop:方面> 方面, < aop:顾问> 宣布顾问甚至 代理和拦截器定义使用Spring 1.2风格在同一 配置。 所有这些都是使用相同的底层实现 支持机制,将共存没有任何困难。

9.6代理机制

Spring AOP使用JDK的动态代理或CGLIB要么创建 代表一个给定的目标对象。 (JDK动态代理是首选 每当你有一个选择)。

如果目标对象进行代理实现至少一个接口 然后将使用JDK的动态代理。 所有的接口实现 由目标类型将代理。 如果目标对象不实现任何接口然后CGLIB代理将被创建。

如果你想强迫使用CGLIB代理(例如, 代理的每个方法定义的目标对象,而不只是这些 通过其接口实现),你可以这样做。 然而,有一些 问题需要考虑:

- 最后 方法不能被建议,因为他们 不能被覆盖。
- Spring 3.2的,不需要添加CGLIB到你 项目类路径,CGLIB类下重新打包 org。 springframework和包括直接在spring核心JAR。 这 意味着基于cglib代理支持'只是作品的相同的方式 JDK动态代理总是有。
- 你的代理对象的构造函数会调用两次。 这是一个自然的结果,即代理模型CGLIB 为每个代理生成子类对象。 对于每个代理实例,两个对象的创建:实际的代理对象和一个 子类的实例实现的建议。 这种行为是 没有表现出在使用JDK代理。 通常,调用构造函数 代理的类型两次,不是问题,因为通常只 作业发生,没有真正的逻辑实现 构造函数。

强制使用CGLIB代理设置 的价值 代理目标类 属性的 < aop:配置> 元素为真:

```
<aop:config proxy-target-class="true">
  <!-- other beans defined here... -->
</aop:config>
```

迫使CGLIB代理当使用@ aspectj火狐的一个插件的支持, 设置 "代理目标类的 属性的 < aop:aspectj火狐的一个插件> 元素真正的 :

```
<aop:aspectj-autoproxy proxy-target-class="true"/>
```

注意

多个 < aop:config / > 部分 倒塌成一个单一的统一汽车代理创造者在运行时,这 应用 最强 代理设置,任何的 这个 < aop:config / > 部分(通常来自 不同的XML bean定义文件)指定。 这也适用于 < tx:注解驱动/ > 和 < aop:aspectj火狐的一个插件/ > 元素。

需要明确的是:使用 " 代理目标类= " true " " 在 < tx:注解驱动/ > , < aop:aspectj火狐的一个插件/ > 或 < aop:config / > 元素将强制使用 CGLIB代理 对于所有三个人 。

9.6.1A理解AOP代理

Spring AOP是 基于代理的 。 这是至关 重要的是你掌握了语义的最后一个语句 其实就是在你写你自己的方面或使用任何 春天提供的基于aop方面Spring框架。

考虑第一场景,你有一个普通的, 联合国代理,没有什么特殊之处,直接对象引用,因为 下面的代码片段所示。

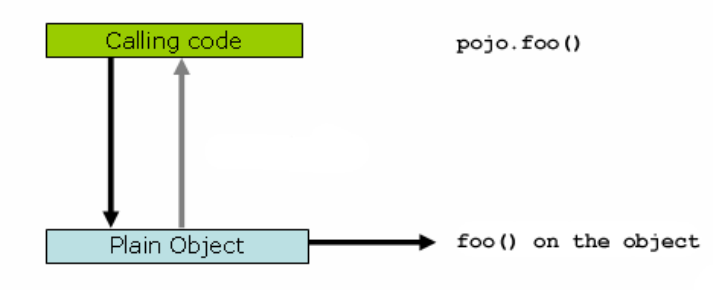
```
public class SimplePojo implements Pojo {

  public void foo() {
```

```
    // this next method invocation is a direct
    call on the 'this' reference
    this.bar();
  }

  public void bar() {
    // some logic...
  }
}
```

如果你对某个对象调用方法的引用,该方法 调用 直接 在对象引用,可以 下面看到。



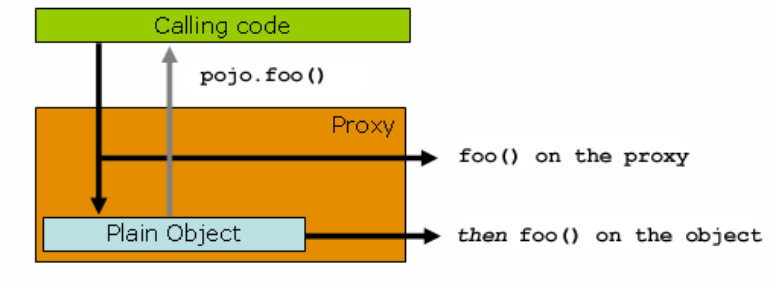

```
public class Main {

  public static void main(String[] args) {

    Pojo pojo = new SimplePojo();

    // this is a direct method call on the 'pojo' reference
    pojo.foo();
  }
}
```

事情变化时稍加参考,客户端代码已经是 一个代理。 考虑下面的图和代码片段。




```
public class Main {

  public static void main(String[] args) {

    ProxyFactory factory = new ProxyFactory(new SimplePojo());
    factory.addInterface(Pojo.class);
    factory.addAdvice(new RetryAdvice());

    Pojo pojo = (Pojo) factory.getProxy();

    // this is a method call on the proxy!
    pojo.foo();
  }
}
```

关键的是要理解是,客户端代码内部 这个 主要(. .) 的 主要 类 有一个引用代理吗 。 这意味着 上的方法调用该对象引用将呼吁代理,和作为 这样的代理将能够代表所有的拦截器 (建议),相关的特定的方法调用。 然而,一旦 电话终于达到目标对象, SimplePojo 参考在这种情况下,任何方法 调用,它可能使在本身,如 这条() 或 这foo() ,将被调用的反对 这 参考, 不 代理。 这有着重要的意义。 它 意味着自我调用是 不 要结果 相关的建议与方法调用获得机会 执行。

好吧,那么什么是必须要做的吗? 最好的方法(最好是使用术语)是松散这里重构代码,这样 自动调用不会发生。 当然,这确实需要一些工作 你的一部分,但它是最好的,至少侵入性的方法。 接下来的 方法绝对是可怕的,我几乎不愿点它 准确,因为它太可怕。 你可以(窒息!)完全的领带 在你的类的逻辑Spring AOP通过这样做:

```
public class SimplePojo implements Pojo {

  public void foo() {
    // this works, but... gah!
    ((Pojo) AopContext.currentProxy()).bar();
  }
```

```
  public void bar() {
    // some logic...
  }
}
```

这完全夫妇Spring AOP代码, 和 它使类本身知道的事实 它被用于一个AOP上下文,苍蝇在面对AOP。 它还需要一些额外的配置当代理正在 创建:

```
public class Main {

  public static void main(String[] args) {

    ProxyFactory factory = new ProxyFactory(new SimplePojo());
    factory.adddInterface(Pojo.class);
    factory.addAdvice(new RetryAdvice());
    factory.setExposeProxy(true);

    Pojo pojo = (Pojo) factory.getProxy();

    // this is a method call on the proxy!
    pojo.foo();
  }
}
```

最后,必须指出AspectJ没有这个 自动调用问题,因为它不是一个基于代理的AOP 框架。

9.7一个编程创建@ aspectj代理

除了声明方面在你的配置或者使用 < aop:配置> 或 < aop:aspectj火狐的一个插件> ,这也是可能的 以编程方式创建代理,建议目标对象。 为 全面详细的Spring的AOP API,见下一章。 这里我们想 专注于能够自动创建代理使用@ aspectj 方面。

类 org.springframework.aop.aspectj.annotation.AspectJProxyFactory 可以用来创建一个代理为目标对象,建议由一个吗 或多个@ aspectj切面。 对于这类基本用法很简单,如 下面的插图。 看到满信息。Javadocs

```
// create a factory that can generate a proxy for the given target object
AspectJProxyFactory factory = new AspectJProxyFactory(targetObject);

// add an aspect, the class must be an @AspectJ aspect
// you can call this as many times as you need with different aspects
factory.addAspect(SecurityManager.class);

// you can also add existing aspect instances, the type of the object supplied must be an @AspectJ aspect
factory.addAspect(usageTracker);

// now get the proxy object...
MyInterfaceType proxy = factory.getProxy();
```

9.8一个使用AspectJ和Spring应用程序

一切我们已经覆盖到目前为止在这一章是纯Spring AOP。 在本节中,我们将看看如何使用AspectJ 编译器/韦弗取代,或者除了,Spring AOP如果你需要去 超出了Spring AOP提供的设施独自。

春天的船舶小AspectJ方面库,它是可用的 独立的在你的分配 弹簧方面jar ,你将需要添加这个 到您的类路径中为了使用方面在它。 SectionA 9 8 1,一个使用AspectJ依赖注入与域对象 Springa 和 SectionA 9 8 2,一个其他弹簧方面AspectJa 讨论的内容库,以及如何使用它。 SectionA 9 8 3,一个AspectJ方面使用Spring配置IoCa 讨论了如何依赖注入AspectJ 方面编织使用AspectJ编译器。 最后, SectionA 9 8 4,一个装入时编织与AspectJ在春季Frameworka 介绍了装入时编织的 Spring应用程序使用AspectJ。

9.8.1A使用AspectJ依赖注入与域对象 春天

Spring容器实例化bean中定义和配置 你的应用程序上下文。 也可以问一个bean工厂 配置一个 预先存在的 对象给定的名称bean定义包含配置应用。 这个 弹簧方面jar 包含一个 注解驱动方面,使用这种能力允许 依赖注入的 任何对象 。 支持是 用于创建的对象 以外的控制 任何容器 。 域对象往往属于这 类别,因为他们往往是通过编程方式创建使用 新 操作符,或由一个ORM工具的结果 数据库查询。

这个 @Configurable 注释标志 一个类作为资格台弹力配置。 在最简单的 情况下可以使用它就像一个标记注释:

```
package com.xyz.myapp.domain;

import org.springframework.beans.factory.annotation.Configurable;
```

```
@Configurable
public class Account {
  // ...
}
```

当用作标记接口,通过这种方式,弹簧将配置 注释类型的新实例(帐户 在 这种情况下)使用bean定义(通常原型作用域) 完全限定名称相同类型名称 (com xyz myapp域帐户)。 因为默认的 名字对于一个bean是完全限定名称的类型,方便 方法声明的原型定义就是省略了 id 属性:

```
<bean class="com.xyz.myapp.domain.Account" scope="prototype">
 <property name="fundsTransferService" ref="fundsTransferService"/>
</bean>
```

如果你想要显式地指定bean的名称。原型 定义来使用,你可以直接在注释:

```
package com.xyz.myapp.domain;

import org.springframework.beans.factory.annotation.Configurable;

@Configurable("account")
public class Account {
  // ...
}
```

春天将寻找一个bean定义命名 “ 帐户 ” 并使用它作为定义配置 新 帐户 实例。

您还可以使用自动装配避免必须指定一个 专用的bean定义在所有。 有弹簧应用自动装配 使用 “ 自动装配 ” 属性的 @Configurable 注释:指定要么 @Configurable(自动装配=自动装配的类型) 或 @Configurable(自动装配=自动装配的名字 对于 自动装配的类型或名称分别。 作为替代,截至 Spring 2.5最好是指定显式,注解驱动的 依赖注入你的 @Configurable bean使用 @ autowired 或 @ inject 在字段或方法水平(见 SectionA 5.9,一个configurationa基于注解的容器 为进一步的细节)。

最后你可以使Spring依赖性检查对象 引用新创建和配置对象通过使用 dependencyCheck 属性(例如: @Configurable(自动装配=自动装配的名字,dependencyCheck = true))。 如果这个属性被设置为true,那么春天将验证后 配置,所有属性(这不是原语或 集合)已经被建立。

使用注释本身并没有什么当然。 这是 AnnotationBeanConfigurerAspect 在 弹簧方面jar ,作用于 存在的注释。 在本质方面说 “回来后 从初始化一个新的对象类型的注释 @Configurable ,配置新 使用Spring创建的对象按照属性的 注释” 。 在这种背景下, 初始化 指 新实例化的对象(如。 ,对象实例 “ 新 '操作符),以及 可序列化的 对象,接受着 反序列化(如。 ,通过 readResolve())。

注意

一个关键的短语在上面的段落是 ‘ 在 精华 ” 。 大多数情况下,精确的语义 ” 回来后,初始化一个新的 对象 “会没事的…… 在这种背景下, “ 在 初始化 “意味着依赖项将 注入 在 对象已经构成- 这意味着依赖项将不可以使用的 类的构造函数体。 如果你想要的依赖项 注入 之前 身体的构造函数执行, 从而可以在主体中使用的构造函数,然后 您需要定义这个的 @Configurable 宣言像 所以:

```
@Configurable(preConstruction=true)
```

你可以找到更多的信息关于该语言的语义 各种类型的切入点在AspectJ 在 这个附录 的 AspectJ 编程指南 。

对于这个工作带注释的类型必须与外人 AspectJ韦弗-您可以使用Ant或Maven构建时任务去做 这(见例如 AspectJ 开发环境导)或装入时编织(见 SectionA 9 8 4,一个装入时编织与AspectJ在春季Frameworka)。 这个 AnnotationBeanConfigurerAspect 本身需要 配置到春天(为了获得一个bean的引用 工厂也被用来配置新对象)。 如果你是使用基于Java的配置简单的添加 @EnableSpringConfigured 任何 @ configuration 类。

```
@Configuration
@EnableSpringConfigured
public class AppConfig {

}
```

如果你更喜欢基于XML的配置,春天 上下文 名称空间 定义了一个方便 背景:spring配置 元素:

```
<context:spring-configured/>
```

如果您正在使用DTD,而不是模式,相当于 定义是:

```
<bean
    class="org.springframework.beans.factory.aspectj.AnnotationBeanConfigurerAspect"
    factory-method="aspectOf"/>
```

实例的 @Configurable 对象 创建 之前 方面已经配置将 结果在消息被发布到调试日志,没有配置 发生的对象。 一个例子可能是一个bean在春季 配置域对象创建时初始化 春天。 在这种情况下,你可以使用 “取决于” bean属性 手动指定bean取决于配置 方面。

```
<bean id="myService"
 class="com.xzy.myapp.service.MyService"
 depends-on="org.springframework.beans.factory.aspectj.AnnotationBeanConfigurerAspect">

 <!-- ... -->

</bean>
```

注意

不激活 @Configurable 处理通过bean configurer方面,除非你真的 意思是在运行时依赖其语义。 特别是,确保 你不要使用 @Configurable 在bean类,注册为普通弹簧豆 容器:你会得到双倍的初始化,否则一旦 通过容器一旦通过方面。

单元测试 @Configurable 对象

的一个目标 @Configurable 支持是使 独立的单元测试的域对象没有困难 硬编码查找有关。 如果 @Configurable 类型没有被编织 通过AspectJ然后注释没有影响在单元测试, 你可以简单地设置模拟或存根属性引用的对象 根据试验和正常运作。 如果 @Configurable 类型 有 被编织的AspectJ然后你仍然可以 容器外部的单元测试是正常的,但你会看到 警告消息每次你构建一个 @Configurable 对象表示它 尚未由Spring配置。

使用多种应用程序上下文

这个 AnnotationBeanConfigurerAspect 使用 来实现 @Configurable 支持 是一个AspectJ singleton方面。 一个单例的范围方面是 相同的范围 静态 成员,也就是说 有一个方面实例类加载器,定义了每类型。 这意味着如果你定义内的多种应用程序上下文 相同的类加载器层次结构,你需要考虑在哪里定义 @EnableSpringConfigured bean和在哪里 地方 弹簧方面jar 在 类路径。

考虑一个典型的春天web配置共享 父应用程序上下文定义公共业务服务和 一切都需要支持他们,和一个孩子应用程序上下文 包含特定于每个servlet定义的servlet。 所有的 这些上下文将并存在同一个类加载器层次结构, 所以 AnnotationBeanConfigurerAspect 只能 举行一个引用其中一个。 在这种情况下我们建议定义 这个 @EnableSpringConfigured bean的 共享(父)应用程序上下文:这个定义的服务 你可能会想注入到域对象。 一个后果是 那你不能配置域对象的引用的豆子 定义在这个孩子(servlet特定)上下文使用 @Configurable机制(可能不是你想要做的 无论如何!)。

当部署多个web应用程序在相同的容器, 确保每个web应用程序加载类型 弹簧方面jar 使用自己的 类加载器(例如,通过放置 弹簧方面jar 在 “- inf / lib”)。 如果 弹簧方面jar 只有添加到吗 容器类路径(因此广泛加载共享父 类加载器),所有web应用程序将共享相同的方面实例 这可能不是你想要的。

9.8.2A其他弹簧方面AspectJ

除了 @Configurable 方面, 弹簧方面jar 包含一个AspectJ方面,可以用来驱动弹簧的 事务管理的类型和的方法 transactional 注释。 这是 主要用于用户谁想要使用Spring框架的 事务支持Spring容器之外。

方面解释 transactional 注释是 AnnotationTransactionAspect 。 当使用这个 方面,你必须标注 实现 类 (和/或方法在这类), 不 这个 接口(如果有的话),类实现。 AspectJ遵循Java的 规则,接口是注解 不 继承 。

一个 transactional 注释 类指定缺省的事务语义的执行 任何 公共 操作类。

一个 transactional 注释 方法在类中重写默认的事务语义 给定类的注释(如果存在)。 方法 公共 , 保护 ,默认的 能见度可能都是注释。 注释 保护 和默认的能见度方法直接是唯一的办法 事务界定为执行这些方法。

对AspectJ的程序员需要使用Spring配置 和事务管理的支持,但不愿(或不能)用 注释, 弹簧方面jar 还包含 文摘 方面,您可以扩展

提供自己的切入点定义。 看到的来源 AbstractBeanConfigurerAspect 和 AbstractTransactionAspect 方面更多 信息。 作为一个例子,下面的摘录展示了如何 写一个方面配置的所有实例对象中定义的 域模型使用原型匹配的bean定义 完全限定类名:

```
public aspect DomainObjectConfiguration extends AbstractBeanConfigurerAspect {

  public DomainObjectConfiguration() {
    setBeanWiringInfoResolver(new ClassNameBeanWiringInfoResolver());
  }

  // the creation of a new bean (any object in the domain model)
  protected pointcut beanCreation(Object beanInstance) :
    initialization(new(..)) &&
    SystemArchitecture.inDomainModel() &&
    this(beanInstance);

}
```

9.8.3A AspectJ方面使用Spring IoC配置

当使用AspectJ方面与Spring应用程序,它是自然的 希望和期待都可以配置等方面使用 春天。 AspectJ运行时本身负责方面创造, 和手段的配置创建AspectJ方面通过弹簧 取决于AspectJ实例化模型(” / xxx “条款)所使用的方面。

大多数的AspectJ方面 singleton 方面。 配置这些方面非常简单:简单地创建一个 bean定义引用类型作为正常的方面,包括 bean属性 “工厂方法= " aspectOf” 。 这 确保弹簧获得方面实例为它通过询问AspectJ 而不是试图创建一个实例本身。 例如:

```
<bean id="profiler" class="com.xyz.profiler.Profiler"
    factory-method="aspectOf">
  <property name="profilingStrategy" ref="jamonProfilingStrategy"/>
</bean>
```

非单体方面很难配置:但是这是 可以通过创建原型bean定义和使用 @Configurable 支持 弹簧方面jar 配置 方面一旦bean实例创建AspectJ 运行时。

如果你有一些@ aspectj切面,你想织与 AspectJ(例如,使用加载时编织为领域模型类型) 和其他@ aspectj切面,你想使用Spring AOP和 这些方面都是使用Spring配置的,那么您将需要 告诉Spring AOP @ aspectj自动代理支持这确切的子集 @ aspectj切面的定义在配置应该用于 自动代理。 为此,您可以使用一个或多个 <包括/ > 元素在 < aop:aspectj火狐的一个插件/ > 宣言。 每个 <包括/ > 元素指定一个名称的模式, 和只有bean与名称相匹配的至少一个模式 用于Spring AOP火狐的一个插件配置:

```
<aop:aspectj-autoproxy>
  <aop:include name="thisBean"/>
  <aop:include name="thatBean"/>
</aop:aspectj-autoproxy>
```

注意

不要被误导的名称吗 < aop:aspectj火狐的一个插件/ > 元素:使用它 将会创建 Spring AOP 代理 。 @ aspectj风格的方面声明的只是 这里所用,但AspectJ运行时 不 涉及。

9.8.4A装入时编织与AspectJ在Spring框架

装入时编织(LTW)指的是过程的编织AspectJ 方面的到一个应用程序类文件,因为他们已被载入 Java虚拟机(JVM)。 本节的重点是 配置和使用LTW在特定条件下的弹簧 框架:这部分不介绍LTW虽然。 全 的具体细节和配置LTW LTW只有AspectJ (与弹簧不参与),请参阅 LTW 部分的AspectJ开发环境指南 。

Spring框架的增值,带给AspectJ LTW是 在启用多细粒度控制在编织的过程。 “香草” AspectJ LTW是影响使用Java(5 +)代理, 这是 开启通过指定一个VM参数当启动JVM。 这是 因此一项jvm设置,会很好,但是经常在某些情况下 是有点太粗。 春天使 LTW允许您打开LTW 在一个 每- 类加载器 基础上, 这显然是更细粒度和可以在一个更有意义吗 “单个jvm多个应用程序的环境 (如被发现在一个 典型的应用程序服务器环境)。

进一步的, 在某些 环境 ,这种支持使装入时编织 不作任何修改的应用程序服务器的 启动脚本 这将需要添加 -javaagent:/ / aspectjweaver.jar路径 或(我们稍后将讲述在这 部分) -javaagent:路径/去/ org.springframework.instrument - { version } . jar (原名 弹簧剂罐)。 开发者仅仅修改 一个或多个文件,形成应用程序上下文加载时启用 编织而不是依靠管理员通常负责 的部署配置如启动脚本。

现在推销结束了,让我们先步行通过 简单的例子AspectJ LTW使用Spring,紧随其后的是详细的 具体介绍了关于元素下面的例

子。 对于一个 完整的例子,请参阅宠物诊所 样品 应用程序。

第一个例子

让我们假设你是一个应用程序的开发人员已经 负责诊断一些性能问题的原因在 系统。 而不是打破一个分析工具,我们要 做的是开关在一个简单的分析方面,它将使我们能够非常 很快得到一些性能指标,这样我们就可以申请一个 细粒度的分析工具,特定区域立即 后来。

注意

本文的示例使用XML样式配置,它是 也可以配置和使用@ aspectj与 Java配置 。 特别是 @EnableLoadTimeWeaving 可以使用注释来代替 <上下文:加载时间韦弗/ > (见 低于 详情)。

这里是分析方面。 没什么新奇,只是一个 应急的基于时间的分析器,使用@ aspectj风格的 方面声明。

```
package foo;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.util.StopWatch;
import org.springframework.core.annotation.Order;

@Aspect
public class ProfilingAspect {

    @Around("methodsToBeProfiled()")
    public Object profile(ProceedingJoinPoint pjp) throws Throwable {
        StopWatch sw = new StopWatch(getClass().getSimpleName());
        try {
            sw.start(pjp.getSignature().getName());
            return pjp.proceed();
        } finally {
            sw.stop();
            System.out.println(sw.prettyPrint());
        }
    }

    @Pointcut("execution(public * foo..*.*(..))")
    public void methodsToBeProfiled(){}
}
```

我们还将需要创建一个 " meta - inf / aop xml 的文件,通知AspectJ 韦弗,我们想编织我们的 ProfilingAspect 为我们的类。 这个文件 公约,即存在一个文件(或文件)在Java 类路径称为 " meta - inf / aop xml " 是标准 AspectJ。

```
<!DOCTYPE aspectj PUBLIC
        "-//AspectJ//DTD//EN" "http://www.eclipse.org/aspectj/dtd/aspectj.dtd">
<aspectj>

    <weaver>

        <!-- only weave classes in our application-specific packages -->
        <include within="foo.*"/>

    </weaver>

    <aspects>

        <!-- weave in just this aspect -->
        <aspect name="foo.ProfilingAspect"/>

    </aspects>

  </aspectj>
```

现在到spring特定部分的配置。 我们需要 配置一个 LoadTimeWeaver (所有 后来解释说,只是把它在信任现在)。 这个装载时编织器 是必要的组件负责织造方面吗 配置在一个或者多个 meta - inf / aop xml " 文件到您的应用程序中的类。 好的方面是它 不需要大量的配置,如下所示(那里 有一些更多的选项,您可以指定,但这些都是详细的吗 后)。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:context="http://www.springframework.org/schema/context"
      xsi:schemaLocation="
http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
```

```
    http://www.springframework.org/schema/context/spring-context.xsd">

    <!-- a service object; we will be profiling its methods -->
    <bean id="entitlementCalculationService"
        class="foo.StubEntitlementCalculationService"/>

    <!-- this switches on the load-time weaving -->
    <context:load-time-weaver/>

</beans>
```

现在,所有需要的工件到位——方面, “ meta - inf / aop xml 的文件,和春天 配置-,让我们创建一个简单的驱动程序类和一个主要(. .) 方法来演示LTW在 行动。

```
package foo;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public final class Main {

    public static void main(String[] args) {

        ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml", Main.class);

        EntitlementCalculationService entitlementCalculationService
            = (EntitlementCalculationService) ctx.getBean("entitlementCalculationService");

        // the profiling aspect is 'woven' around this method execution
        entitlementCalculationService.calculateEntitlement();
    }
}
```

还有最后一件事要做。 介绍本节 是说,一个可以打开LTW选择性地在吗 每- 类加载器 与Spring的基础,这是 真正的。 然而,对于这个示例,我们将使用一个Java代理 (提供弹簧)开关在LTW。 这是命令行 我们将使用运行上面的 主要 类:

```
java -javaagent:C:/projects/foo/lib/global/spring-instrument.jar foo.Main
```

“ -javaagent “是一个Java 5 +的标志 指定和启用 代理 对仪器程序运行在JVM 。 春天 框架附带这样一个代理, InstrumentationSavingAgent ,这是包装 在 弹簧仪器jar 这 提供的值 -javaagent 参数在上面的例子中。

的输出执行 主要 程序将类似于下面。 (我已经介绍了 thread . sleep(. .) 声明到 calculateEntitlement() 实现, 分析器真正捕捉其他东西比0毫秒- 这个 01234年 毫秒是 不 介绍了一种开销由AOP:))

```
Calculating entitlement

StopWatch 'ProfilingAspect': running time (millis) = 1234
------ ----- ----------------------------
ms     %     Task name
------ ----- ----------------------------
01234  100%  calculateEntitlement
```

因为这是使用AspectJ LTW影响全面的,我们不是 只是限于建议Spring bean;下面的轻微变化 在 主要 程序将产生相同的 结果。

```
package foo;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public final class Main {

    public static void main(String[] args) {

        new ClassPathXmlApplicationContext("beans.xml", Main.class);

        EntitlementCalculationService entitlementCalculationService =
            new StubEntitlementCalculationService();

        // the profiling aspect will be 'woven' around this method execution
        entitlementCalculationService.calculateEntitlement();
    }
}
```

注意在上面的项目我们只是引导 Spring容器,然后创建一个新的实例 StubEntitlementCalculationService 完全 春天的上下文之外…… 配置建议仍然会编织 在。

无可否认的是简单的...的例子 然而基本的 在春天的LTW支持都被介绍在上面的 例,这一节的其余部分将解释背后的 “为什么” 每一位的详细配置和使用情况。

注意

这个 ProfilingAspect 用于这 的例子可能是最基本的,但是它是非常有用的。 这是一个很好的例子 一个开发方面,开发人员可以使用在 开发(当然),然后很容易排除从构建 的应用程序被部署到UAT或生产。

方面

你使用的方面在LTW必须AspectJ方面。 他们 可以写在要么AspectJ语言本身或者你可以写吗 你在@ aspectj风格方面。 后者的选择当然是 只有一个选项,如果您使用的是Java 5 +,但这确实意味着你的 然后两方面有效的AspectJ 和 春天 AOP方面。 此外,编译后的类需要方面 可以在类路径中。

” meta - inf / aop xml ”

AspectJ LTW基础设施配置使用一个或多个 ” meta - inf / aop xml 的文件,这是Java 类路径(或直接,或更通常在jar文件)。

结构和这个文件的内容是详细的在主 AspectJ参考文档,感兴趣的读者 称为 到该资源的 。 (我理解这部分是短暂的, 但 “ aop xml “文件- - -有100% AspectJ 没有spring特定信息或语义,适用于它,所以 没有额外的值,我可以贡献或结果),所以 而不是重复比较满意的部分,AspectJ 开发人员写的,我只是指导你。)

需要的库(jar)

至少你将需要以下库使用 Spring框架的支持AspectJ LTW:

1. spring aop jar (版本 2.5或更高版本,加上所有强制性依赖)
2. aspectjweaver.jar (版本1 6 8或更高)

如果你正在使用 弹簧提供代理 让仪器 ,你也将需要:

1. 弹簧仪器jar

Spring配置

在春天的关键组件的LTW支持 LoadTimeWeaver 接口(在 org.springframework.instrument.classloading 包),和众多的实现它的那艘船 春天分布。 一个 LoadTimeWeaver 负责添加一个或更多 java.lang.instrument.ClassFileTransformers 一个 类加载器 在运行时,它可以打开门 各种各样的有趣的应用程序,其中一个发生的 LTW方面。

提示

如果你不熟悉这个想法的运行时类文件 转换,你是鼓励阅读Javadoc API 文档 java朗仪器 包在继续之前。 这不是一个巨大的繁琐,因为有 ——而恼人地——珍贵的小文档那里…… 关键 接口和类至少会被放置在你的面前, 参考当你阅读本部分。

配置一个 LoadTimeWeaver 对于一个特定的 ApplicationContext 也很容易吗 添加一行。 (请注意,你几乎肯定会需要 是使用一个 ApplicationContext 作为你的 Spring容器——通常是一个 BeanFactory 是不够的,因为 LTW的支持使得使用 BeanFactoryPostProcessors)。

使Spring框架的LTW支持,你需要 配置一个 LoadTimeWeaver ,这 通常是通过使用 @EnableLoadTimeWeaving 注释。

```
@Configuration
@EnableLoadTimeWeaving
public class AppConfig {

}
```

或者,如果你喜欢,可以使用基于XML的配置 <上下文:加载时间韦弗/ > 元素。 注意 元素定义在 “ 上下文 “名称空间。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:context="http://www.springframework.org/schema/context"
      xsi:schemaLocation="
http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">

    <context:load-time-weaver/>
```

```
</beans>
```

上面的配置将定义和注册一个号码的 自动为您LTW-specific基础设施bean,这样 作为一个 LoadTimeWeaver 和一个 AspectJWeavingEnabler 。 默认 LoadTimeWeaver 是 DefaultContextLoadTimeWeaver 类,它 装修一个自动检测的尝试 LoadTimeWeaver :确切的类型的 LoadTimeWeaver 那将是 “自动检测”是依赖于你的运行时环境 (概括如下表)。

为多9 1一个 DefaultContextLoadTimeWeaver LoadTimeWeavers

运行时环境	LoadTimeWeaver 实现
运行在 Bea的 Weblogic 10	WebLogicLoadTimeWeaver
运行在 IBM WebSphere Application Server 7	WebSphereLoadTimeWeaver
运行在 甲骨文的 OC4J	OC4JLoadTimeWeaver
运行在 GlassFish	GlassFishLoadTimeWeaver
运行在 JBoss AS	JBossLoadTimeWeaver
JVM始于春天 InstrumentationSavingAgent (java -javaagent:路径/去/弹簧仪器jar)	InstrumentationLoadTimeWeaver
回退,期望底层的类加载器通常根据约定 (如适用 TomcatInstrumentableClassLoader 和 树脂)	ReflectiveLoadTimeWeaver

注意,这些只是 LoadTimeWeavers 这是个 当使用 DefaultContextLoadTimeWeaver :它 当然是可以指定哪些 LoadTimeWeaver 实现,你 希望使用。

指定一个特定的 LoadTimeWeaver 与Java配置实现 LoadTimeWeavingConfigurer 接口和覆盖 这个 getLoadTimeWeaver() 方法:

```
@Configuration
@EnableLoadTimeWeaving
public class AppConfig implements LoadTimeWeavingConfigurer {
    @Override
    public LoadTimeWeaver getLoadTimeWeaver() {
        return new ReflectiveLoadTimeWeaver();
    }
}
```

如果您使用的是基于XML的配置,你可以指定完全限定 类名的值的 韦弗类 ” 属性 <上下文:加载时间韦弗/ > 元素:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:context="http://www.springframework.org/schema/context"
      xsi:schemaLocation="
http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">

    <context:load-time-weaver
          weaver-class="org.springframework.instrument.classloading.ReflectiveLoadTimeWeaver"/>

</beans>
```

这个 LoadTimeWeaver 这是 定义并登记的配置可以 后来恢复从Spring容器使用众所周知的名字 ” LoadTimeWeaver ” 。 请记住, LoadTimeWeaver 存在只是作为一个 春天的机制基础设施LTW添加一个或多个 ClassFileTransformers 。 实际的 ClassFileTransformer 这是否就是LTW ClassPreProcessorAgentAdapter (从 org.aspectj.weaver.loadtime 包)类。 看到 Javadoc的类级 ClassPreProcessorAgentAdapter 类作进一步 细节,因为细节实际上是影响如何编织 超出了本节的范围。

最后还有一个属性配置左来讨论: “ aspectjWeaving ” 属性 (或 “ aspectj编织 如果您使用的是XML)。 这是一个简单的 属性 控制是否启用LTW与否,事情就是这么简单 为,。 它接受三种可能的值,总结如下, 用默认值如果属性不存在被 ” 自动检测 “

9.2为多。 一个AspectJ编织属性值

注释值	XML值	解释
启用	在	AspectJ编织,和方面将编织 在加载时适当。
禁用	掉	LTW是关闭的... 没有方面将编织在 加载时。
自动检测	自动检测	如果春天LTW基础设施可以找到 至少有一个 “ meta - inf / aop xml 的文件,然后 AspectJ编织是打开的,否则它是关闭的。这是默认的 值。

特定于环境的配置

最后这部分包含任何额外的设置和 配置,您将需要在使用Spring的LTW支持 环境如应用服务器和web容器。

tomcat

apache tomcat 的默认的类装入器 不支持类转换这就是为什么Spring提供了一个增强的实现 地址这一需要。 命名 TomcatInstrumentableClassLoader 、装载机工作 在Tomcat 5.0及以上,可以分别登记的 每个 web应用程序 如下:

- Tomcat 6.0。 x或更高
  1. 复制 org.springframework.instrument.tomcat.jar 进 $卡特琳娜家 / lib, $卡特琳娜家 代表的根 Tomcat安装)
  2. 指导Tomcat使用自定义类加载器(相反 默认的)通过编辑web应用程序上下文 文件:

     ```
     <Context path="/myWebApp" docBase="/my/webApp/location">
       <Loader
         loaderClass="org.springframework.instrument.classloading.tomcat.TomcatInstrumentableClassLoader"/>
     </Context>
     ```

     Apache Tomcat 6.0。 x(类似于5 0 x / 5 5 x)系列支持多种上下文位置:
     - 服务器配置文件—— 卡特琳娜家美元/ conf / server . xml
     - 默认上下文配置- 卡特琳娜家美元/ conf /上下文xml —— 影响到所有部署web应用程序
     - 应用程序配置,每网络可以部署在服务器端,要么 卡特琳娜家美元/ conf /[enginename]/[hostname]/[webapp]上下文xml 或嵌入 这个web存档在 meta - inf /上下文xml

     为了提高效率,嵌入式/ web应用程序配置风格 推荐的,因为它将只影响应用程序使用 自定义类加载器和不需要任何更改服务器配置。 看到Tomcat . x 文档 为更多的细节关于可用的上下文位置。
- Tomcat 5 0 x / 5 5倍
  1. 复制 org.springframework.instrument.tomcat.jar 进 $卡特琳娜家 /服务器/ lib, $卡特琳娜家 代表的根 Tomcat的安装。
  2. 指导Tomcat使用自定义类加载器相反 默认的一个通过编辑web应用程序上下文 文件:

     ```
     <Context path="/myWebApp" docBase="/my/webApp/location">
       <Loader
         loaderClass="org.springframework.instrument.classloading.tomcat.TomcatInstrumentableClassLoader"/>
     </Context>
     ```

     Tomcat 5.0。 x和5.5。 x系列支持多种上下文位置:
     - 服务器配置文件—— 卡特琳娜家美元/ conf / server . xml
     - 默认上下文配置- 卡特琳娜家美元/ conf /上下文xml —— 影响到所有部署web应用程序
     - 应用程序配置,每网络可以部署在服务器端,要么 卡特琳娜家美元/ conf /[enginename]/[hostname]/[webapp]上下文xml 或嵌入 这个web存档在 meta - inf /上下文xml

     为了提高效率,嵌入式web配置风格推荐 推荐的,因为它将只影响应用程序类装入器使用。 看到Tomcat 5. x 文档 为更多的细节关于可用的上下文位置。
     Tomcat版本5 5 20之前包含一些小问题 XML配置解析,避免使用 装载机 标签内 server . xml 配置,无论一个类 装载机是指定或它是否为官方或一个自定义的 一。 看到Tomcat的bugzilla的 更多的 细节 。
     在Tomcat 5.5。 x,版本5 5 20或以后,你应该设置 useSystemClassLoaderAsParent 到 假 来解决这个问题:

     ```
     <Context path="/myWebApp" docBase="/my/webApp/location">
       <Loader
         loaderClass="org.springframework.instrument.classloading.tomcat.TomcatInstrumentableClassLoader"
           useSystemClassLoaderAsParent="false"/>
     </Context>
     ```

这个设置是不需要在Tomcat 6或更高。

或者,考虑使用弹簧提供了通用的 虚拟机代理,必须在Tomcat启动脚本(见上图)。 这将使仪器可供所有已部署的web 的应用程序,无论他们发生什么类加载器上运行。

应用服务器,WebSphere,OC4J,树脂,GlassFish、JBoss

BEA WebLogic的最新版本(版本10以上),IBM WebSphere Application Server(version 7及以上), 甲骨文Java EE容器(OC4J 10.1.3.1版及以上),树脂(3.1以上)和JBoss(5。 x或以上) 提供一个类加载器,能够本地仪表。 春天的家乡LTW利用这样的类加载器来启用AspectJ编织。 您可以启用LTW只需激活装入时编织 如前面所述。 具体地说,你做 不 需要修改启动脚本添加 - javaagent:路径/去/弹簧仪器jar 。

注意,GlassFish仪表能够只在类加载器它的耳朵环境。 对GlassFish的web应用程序,遵循Tomcat安装指令如前所述。

注意,在JBoss 6。 x,这个应用程序服务器扫描需要被禁用,以防止它加载类 在应用程序的真正开始。 一个快速的解决方法是增加一个文件夹命名为工件 - inf / jboss扫描xml 包含以下内容:

```
<scanning xmlns="urn:jboss:scanning:1.0"/>
```

泛型Java应用程序

当类仪表需要环境,不支持或 不支持现有的吗 LoadTimeWeaver 实现, 一个JDK剂可以是唯一的解决方案。 对于这种情况,春天提供 InstrumentationLoadTimeWeaver , 这需要一个spring特定(但很一般)VM剂, org.springframework.instrument - { version } . jar (原名 弹簧剂罐)。

使用它,您必须启动虚拟机与春天剂,通过 提供以下JVM选项:

```
-javaagent:/path/to/org.springframework.instrument-{version}.jar
```

注意,这需要修改虚拟机启动脚本可能阻止你使用这个应用程序服务器 环境(取决于您的操作策略)。 此外,JDK代理将仪器的 整个 虚拟机可以证明昂贵的。

由于性能原因,推荐使用这个配置只有如果你的目标环境 (如 Jetty)没有(或不支持)专用LTW。

9.9进一步资源

更多的信息在AspectJ可以找到 AspectJ网站 。

这本书 Eclipse AspectJ 艾德里安·Colyer等。 艾尔。(addison - wesley,2005)提供了一个全面的介绍和 AspectJ语言参考。

这本书 AspectJ在行动 Ramnivas Laddad由 (曼宁,2003)是强烈推荐的,这本书的焦点是在 AspectJ,但很多一般AOP主题探索 (在一些深度)。

10。 一个Spring AOP api

10.1一个介绍

前面的章节描述了Spring 2.0和以后的版本的 支持AOP使用@ aspectj和基于方面的定义。 在 这一章中,我们讨论了低层 Spring AOP api和AOP 支持Spring 1.2的应用程序中使用。 为新的应用程序,我们 建议使用Spring AOP支持2.0及以后版本中所描述的 前一章,但当工作与现有应用程序,或者当 阅读书籍和文章,你可能遇到Spring 1.2风格的例子。 Spring 3.0是Spring 1.2向后兼容和一切 在本章中介绍是完全支持Spring 3.0中。

10.2一个切入点API在春天

让我们看看如何处理关键切入点春天 概念。

10.2.1A概念

春天的切入点模型支持独立的切入点的重用 建议类型。 可以使用同样的目标不同的建议 切入点。

这个 org.springframework.aop.Pointcut 接口 是中央接口,用于目标建议特定的类 和方法。 完整的界面如下所示:

```
public interface Pointcut {
    ClassFilter getClassFilter();
    MethodMatcher getMethodMatcher();
}
```

分裂 切入点 接口 分为两部分允许重用类和方法匹配部分,和 细粒度组成操作(如执行一个“联盟” 另一种方法匹配器)。

这个 ClassFilter 接口用于 限制切入点来一组给定的目标类。 如果 matches() 方法总是返回true,所有目标 类将匹配:

```
public interface ClassFilter {
    boolean matches(Class clazz);
}
```

这个 MethodMatcher 接口是 通常更重要。 完整的界面如下所示:

```
public interface MethodMatcher {
    boolean matches(Method m, Class targetClass);
    boolean isRuntime();
    boolean matches(Method m, Class targetClass, Object[] args);
}
```

这个 匹配(方法、类) 方法用于 测试这个切入点是否会匹配一个给定的方法对一个目标 类。 这种评价可以执行一个AOP代理创建时, 避免需要一个测试在每个方法调用。 如果2参数 匹配方法返回true对于一个给定的方法 isRuntime() MethodMatcher 方法返回 诚然,3参数匹配方法将在每个方法调用 调用。 这使得一个切入点来看看参数传递给 方法调用之前立即目标建议 执行。

最MethodMatchers是静态的,这意味着他们的 isRuntime() 方法返回false。 在这种情况下, 3参数匹配方法永远不会被调用。

提示

如果可能的话,尽量使切入点的静态,允许AOP 框架来缓存结果评价的切入点当AOP 创建代理。

10.2.2A操作切入点

Spring支持操作切入点:值得注意的是, 联盟 和 路口 。

- 联盟意味着方法,要么切入点匹配。
- 十字路口意味着方法这两个切入点 匹配。
- 联盟通常是更有用的。
- 切入点可以组合使用静态方法 org.springframework.aop.support.Pointcuts 类,或使用 ComposablePointcut 类 同一个包。 然而,使用AspectJ切入点表达式 通常是一个更简单的方法。

10.2.3A AspectJ切入点表达式

因为2.0,最重要类型的切入点用春天 org.springframework.aop.aspectj.AspectJExpressionPointcut 。 这是一个切入点,使用AspectJ提供的库来解析一个 AspectJ切入点表达式字符串。

看到前面一章讨论支持AspectJ 切入点原语。

10.2.4A便利切入点实现

Spring提供了一些方便的切入点实现。 一些 可以用现成的,其他人是打算再在吗 特定于应用程序的切入点。

静态切入点

静态的切入点是基于方法和目标类, 不能考虑到方法的参数。 静态的切入点是 足够—— 和最好的 ——对于大多数用途。 这是可能的春天要评估一个静态的切入点只有一次,当一个 第一次被调用的方法:在那之后,没有需要评估 切入点又与每个方法调用。

让我们考虑一些静态的切入点实现包括 与春天。

正则表达式的切入点

一个明显的方式来指定静态的切入点是常规 表达式。 几个AOP框架除了弹簧使这个 可能的。 org.springframework.aop.support.JdkRegexpMethodPointcut 正则表达式是一个通用的切入点,使用正常吗 表达式支持 JDK 1.4 +。

使用 JdkRegexpMethodPointcut 类, 你可以提供一个列表的模式字符串。 如果任何这些是一个 匹配,切入点将评估为true。 (所以结果是 有效的结合这些切入点。)

使用如下所示:

```
<bean id="settersAndAbsquatulatePointcut"
  class="org.springframework.aop.support.JdkRegexpMethodPointcut">
  <property name="patterns">
    <list>
      <value>.*set.*</value>
      <value>.*absquatulate</value>
    </list>
  </property>
</bean>
```

Spring提供了一个方便的类, RegexpMethodPointcutAdvisor ,这允许我们 还引用一个建议(请记住,一个建议可以是一个 拦截器,在建议,抛出建议等)。 在幕后, 弹簧将使用一个 JdkRegexpMethodPointcut 。 使用 RegexpMethodPointcutAdvisor 简化了布线, 一个bean封装两个切入点和通知,如图所示 下图:

```
<bean id="settersAndAbsquatulateAdvisor"
  class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
  <property name="advice">
    <ref local="beanNameOfAopAllianceInterceptor"/>
  </property>
  <property name="patterns">
    <list>
      <value>.*set.*</value>
      <value>.*absquatulate</value>
    </list>
  </property>
</bean>
```

RegexpMethodPointcutAdvisor 可以用 任何意见类型。

属性驱动的切入点

一个重要的类型的静态的切入点是一个 元数据驱动的 切入点。 这种使用价值 元数据属性:通常,源代码级别的元数据。

动态切入点

动态的切入点是昂贵的比静态的评价 切入点。 他们考虑的方法 参数 以及静态信息。 这 意味着他们必须评估每个方法调用; 结果不能被缓存的,作为参数会有所不同。

主要的例子是 控制流 切入点。

控制流的切入点

弹簧控制流的切入点在概念上类似于 AspectJ cflow 切入点,虽然不 强大的。 (目前还没有方法能够指定一个切入点 执行以下一个连接点匹配另一个切入点。) 一个控制 流切入点匹配当前调用堆栈。 例如,它可能 火如果连接点是调用一个方法 com mycompany web 包,或由 SomeCaller 类。 控制流的切入点是 指定使用 org.springframework.aop.support.ControlFlowPointcut 类。

> **注意**
>
> 控制流的切入点是更昂贵的 在运行时评估甚至比其他动态的切入点。 在Java 1.4,成本大约是5次,其他的动态 切入点。

10.2.5A切入点超类

Spring提供了有用的切入点超类来帮助你 实现自己的切入点。

因为静态的切入点是最有用的,你可能会子类 StaticMethodMatcherPointcut,如下所示。 这需要实现 只是一个抽象方法(尽管

它可以覆盖其他 方法自定义行为):

```
class TestStaticPointcut extends StaticMethodMatcherPointcut {

    public boolean matches(Method m, Class targetClass) {
        // return true if custom criteria match
    }
}
```

还有超类动态的切入点。

您可以使用自定义切入点的任何意见类型在Spring 1.0 RC2及以上。

10.2.6A定制切入点

因为切入点在Spring AOP是Java类,而不是 语言特性(如AspectJ)可以声明自定义 切入点,无论是静态或动态。 自定义切入点在春天可以 任意复杂的。 然而,使用AspectJ切入点表达式 语言是建议如果可能的话。

注意

后来版本的春天可能提供支持”语义 切入点”作为提供江淮:例如,“所有的方法,改变 实例变量在目标对象。”

建议在春天10.3 API

现在让我们看看如何Spring AOP处理建议。

10.3.1A建议生命周期

每个建议是Spring bean。 一个实例可以共享的建议 在所有建议的对象,或独特的每个建议的对象。 这 对应 每个类 或 每个建议。

每个类的建议是最常用的。 它是适合通用 建议如事务顾问。 这些不依赖于状态的 这个代理对象或添加新的状态;他们只是行为的方法和 参数。

每个建议适合的介绍,来支持 mixin。 在这种情况下,建议增加国家的代理 对象。

它可以使用一个混合的共享和每个建议 同样的AOP代理。

在春天10.3.2A建议类型

弹簧提供几个建议类型的盒子,是 可扩展支持任意类型的建议。 让我们看看基本 概念和标准建议类型。

拦截around通知

最基本的建议类型在春天 拦截around通知 。

春天是符合AOP联盟界面左右 建议使用方法拦截。 MethodInterceptors实施 在建议应该实现以下接口:

```
public interface MethodInterceptor extends Interceptor {

    Object invoke(MethodInvocation invocation) throws Throwable;
}
```

这个 MethodInvocation 参数 invoke() 方法暴露了方法 调用;目标连接点;AOP代理;和参数 该方法。 这个 invoke() 方法应该返回 调用的结果:返回值的连接点。

一个简单的 MethodInterceptor 实现 看起来如下:

```
public class DebugInterceptor implements MethodInterceptor {

    public Object invoke(MethodInvocation invocation) throws Throwable {
        System.out.println("Before: invocation=[" + invocation + "]");
        Object rval = invocation.proceed();
        System.out.println("Invocation returned");
        return rval;
    }
}
```

注意调用MethodInvocation的 proceed() 法。 这个顺序从上到下 拦截器链向连接点。 大多数拦截器将 调用此方法,并返回它的返回值。 然而,一个 MethodInterceptor,像任何around通知,可以返回一个不同的 值或者抛出一个异常,而不是调用proceed方法。 然而,你不想这样做没有好理由!

注意

与其他AOP MethodInterceptors提供互操作性 联盟兼容的AOP实现。 其他的建议类型 讨论在本节的其余部分实现常见的AOP 的概念,但在一个spring特定方式。 虽然有一个优势 在使用最具体的建议类型,坚持MethodInterceptor 在你可能会建议,如果想要运行在另一个方面 AOP框架。 注意,切入点目前不互操作 框架之间,AOP联盟目前尚未定义 切入点的接口。

建议之前

一个简单的建议类型是 **之前 建议** 。 这并不需要一个 MethodInvocation 对象,因为它只会 进入方法之前调用。

主要的优势的建议是,之前是没有必要的 调用 proceed() 方法,因此也没有 可能无意中未能进行了拦截 链。

这个 MethodBeforeAdvice 界面显示 下面。 (春天的API设计将允许对字段之前的建议, 虽然通常的对象适用于现场拦截和它的 不太可能会实现它,春天)。

```
public interface MethodBeforeAdvice extends BeforeAdvice {

    void before(Method m, Object[] args, Object target) throws Throwable;
}
```

注意,返回类型 无效 。 建议之前 可以插入自定义行为在连接点之前执行,但不能 改变返回值。 如果之前建议抛出一个异常,这个 将中止进一步执行拦截器链。 异常 将传播支持拦截器链。 如果不加以控制,或 在签名的调用方法时,它将直接传递给 客户端,否则将被包装在一个未经检查的异常的 AOP代理。

之前的一个例子建议在春天,它统计所有的方法 调用:

```
public class CountingBeforeAdvice implements MethodBeforeAdvice {

    private int count;

    public void before(Method m, Object[] args, Object target) throws Throwable {
        ++count;
    }

    public int getCount() {
        return count;
    }
}
```

提示

之前的建议可以被用于任何切入点。

抛出建议

抛出建议 之后才调用 返回的连接点如果连接点抛出一个异常。 弹簧提供类型抛出的建议。 注意,这意味着 org.springframework.aop.ThrowsAdvice 接口并 不包含任何方法:它是一个标记接口,确认了 给定对象实现一个或多个类型的抛出的建议方法。 这些 应该是在形式的:

```
afterThrowing([Method, args, target], subclassOfThrowable)
```

只有最后一个参数是必需的。 方法签名可能 要么一个或四个参数,取决于建议吗 方法是感兴趣的方法和参数。 以下 类的实例抛出的建议。

下面的建议是如果一个调用 RemoteException 抛出(包括 子类):

```
public class RemoteThrowsAdvice implements ThrowsAdvice {

    public void afterThrowing(RemoteException ex) throws Throwable {
        // Do something with remote exception
    }
}
```

以下的建议是如果一个调用 ServletException 抛出。 与 以上建议,它声明4参数,以便它可以使用 调用方法,方法参数和目标对象:

```
public class ServletThrowsAdviceWithArguments implements ThrowsAdvice {

    public void afterThrowing(Method m, Object[] args, Object target, ServletException ex) {
        // Do something with all arguments
    }
}
```

最后的示例说明了这两种方法可以 用在一个类,它可以处理 RemoteException 和 ServletException 。 任何数量的抛出的建议方法可以合并成一个类。

```
public static class CombinedThrowsAdvice implements ThrowsAdvice {

    public void afterThrowing(RemoteException ex) throws Throwable {
        // Do something with remote exception
    }

    public void afterThrowing(Method m, Object[] args, Object target, ServletException ex) {
        // Do something with all arguments
    }
}
```

注意: 如果一个抛出建议方法抛出一个 异常本身,它将覆盖原来的异常(即改变 抛出的异常给用户)。 压倒一切的异常将 通常是一个RuntimeException;这是兼容任何方法 签名。 然而,如果一个抛出建议方法会抛出一个检查 例外,它必须匹配目标的声明的例外 方法,因此在某种程度上耦合到特定目标的方法 签名。 不抛出未申报检查异常 这是不符合目标方法的 签名!

提示

抛出的建议可以被用于任何切入点。

回国后的建议

回国后在春天的建议必须实现 org.springframework.aop.AfterReturningAdvice 界面,如下所示:

```
public interface AfterReturningAdvice extends Advice {

    void afterReturning(Object returnValue, Method m, Object[] args, Object target)
            throws Throwable;
}
```

回国后的建议已经获得返回值(它不能修改),调用方法,方法参数和 目标。

以下所有成功后返回的建议项 方法调用,没有抛出的异常:

```
public class CountingAfterReturningAdvice implements AfterReturningAdvice {

    private int count;

    public void afterReturning(Object returnValue, Method m, Object[] args, Object target)
            throws Throwable {
        ++count;
    }

    public int getCount() {
        return count;
    }
}
```

这个建议不改变执行路径。 如果它抛出一个 例外,这将被扔了拦截器链代替 返回值。

提示

回国后的建议可以被用于任何切入点。

介绍的建议

春天把介绍的建议当成一种特殊的 拦截的建议。

介绍需要一个 IntroductionAdvisor , 和一个 IntroductionInterceptor ,实现 以下界面:

```
public interface IntroductionInterceptor extends MethodInterceptor {

    boolean implementsInterface(Class intf);
}
```

这个 invoke() 方法继承了AOP 联盟 MethodInterceptor 接口必须实现 简介:也就是说,如果被调用的方法是在一个介绍 接口,介绍拦截器负责处理 方法调用——它不能调用 proceed() 。

介绍的建议不能用于任何切入点,因为它 仅适用于在类中,而不是方法,水平。 你只能使用 介绍与建议 IntroductionAdvisor , 这有以下方法:

```
public interface IntroductionAdvisor extends Advisor, IntroductionInfo {

    ClassFilter getClassFilter();

    void validateInterfaces() throws IllegalArgumentException;
}

public interface IntroductionInfo {

    Class[] getInterfaces();
}
```

没有 MethodMatcher ,和 因此没有 切入点 ,相关 介绍的建议。 只有类过滤是合乎逻辑的。

这个 getInterfaces() 方法返回 接口引入的这个顾问。

这个 validateInterfaces() 方法是在公司内部使用, 看是否可以实现介绍了接口的配置 IntroductionInterceptor 。

让我们看一个简单的例子从弹簧测试套件。 让我们 假设我们想要介绍以下接口与一个或多个 对象:

```
public interface Lockable {
    void lock();
    void unlock();
    boolean locked();
}
```

这说明了一个 **Mixin** 。 我们 希望能够投建议可锁定的对象,无论他们的 类型和调用方法锁定和解锁。 如果我们调用lock()方法, 我们希望所有的setter方法抛出 LockedException 。 因此我们可以添加一个方面 提供了能够使对象不变的,没有他们有 任何对它的认识:一个很好的例子,AOP。

首先,我们需要一个 IntroductionInterceptor 这确实沉重的 提升。 在本例中,我们扩展了 org.springframework.aop.support.DelegatingIntroductionInterceptor 方便的类。 我们可以实现 IntroductionInterceptor 直接,但使用 DelegatingIntroductionInterceptor 最适合最 情况下。

这个 DelegatingIntroductionInterceptor 是 用来代表一个介绍到实际实现的 引入界面(s)、隐匿使用拦截来做 所以。 委托可以设置为任何对象使用一个构造函数 参数,默认的代表(当使用不带参数的构造器) 这是。 因此在下面的例子中,代表了 LockMixin 子类的 DelegatingIntroductionInterceptor 。 给定一个代表 (默认情况下本身),一个 DelegatingIntroductionInterceptor 实例看起来 对于所有接口实现的代表(除了 IntroductionInterceptor),并将支持对任何 介绍 他们的。 它的子类可能如 LockMixin 调用 suppressInterface(类intf) 方法抑制 接口不应该暴露。 然而,无论多少 接口一个 IntroductionInterceptor 准备 支持, IntroductionAdvisor 使用将 控制接口实际上是暴露。 一个介绍界面 将隐瞒任何实现相同的接口的 目标。

因此LockMixin子类 DelegatingIntroductionInterceptor 并实现了 可锁定的本身。 超类自动回升,可锁定的 可以支持的介绍,所以我们不需要指定。 我们可以引入任意数量的接口在这种方式。

注意使用 锁 实例变量。 这有效地增加了额外的状态来了在目标 对象。

```
public class LockMixin extends DelegatingIntroductionInterceptor
    implements Lockable {

    private boolean locked;

    public void lock() {
        this.locked = true;
    }

    public void unlock() {
        this.locked = false;
    }

    public boolean locked() {
```

```
        return this.locked;
    }

    public Object invoke(MethodInvocation invocation) throws Throwable {
        if (locked() && invocation.getMethod().getName().indexOf("set") == 0)
            throw new LockedException();
        return super.invoke(invocation);
    }

}
```

通常并不需要覆盖 invoke() 方法: DelegatingIntroductionInterceptor 实现—— 这调用委托方法如果方法介绍,否则吗 向着连接点——通常是足够的。 在 目前情况下,我们需要添加一个检查:不可以调用setter方法 如果在锁定模式。

需要的是简单的介绍顾问。 所有需要做的 是举行一个截然不同的 LockMixin 实例,并指定 介绍了接口的——在这种情况下,只是 可封闭的 。 一个更复杂的例子可能需要 引用介绍拦截器(这将是定义为一个 原型):在这种情况下,没有配置相关的 LockMixin ,所以我们简单地创建它使用 新 。

```
public class LockMixinAdvisor extends DefaultIntroductionAdvisor {

    public LockMixinAdvisor() {
        super(new LockMixin(), Lockable.class);
    }
}
```

我们可以把这个顾问非常简单:它不需要 配置。 (然而,这 是 必要的: 不可能使用一个 IntroductionInterceptor 没有 IntroductionAdvisor)。 像往常一样同 介绍,该顾问必须每个,因为它是有状态的。 我们 需要一个不同的实例 LockMixinAdvisor ,和 因此 LockMixin ,对于每个建议的对象。 这个 顾问由部分建议对象的状态。

我们可以把这个顾问通过编程,使用 Advised.addAdvisor() 法,或(推荐的 在XML配置,就像任何其他顾问。 所有代理创建 选择下面讨论,包括“自动代理的创造者”,正确 处理介绍和有状态的混合。

10.4一个顾问API在春天

在春天,一个顾问是一个方面,它只包含一个 建议与对象关联一个切入点表达式。

除了特殊情况的介绍,任何顾问可以 用于任何建议。 org.springframework.aop.support.DefaultPointcutAdvisor 是最常用的顾问类。 例如,它可以使用 一个 MethodInterceptor , BeforeAdvice 或 ThrowsAdvice 。

它是可能的混合顾问和咨询类型在春天在相同的 AOP代理。 例如,您可以使用一个拦截around通知,抛出 建议和在一个代理配置之前建议:春天将 自动创建必要的拦截器链。

10.5一个使用ProxyFactoryBean创建AOP代理

如果你使用Spring IoC容器(一个ApplicationContext或 为你的业务对象BeanFactory)——和你应该! ——你会 想要使用Spring的AOP FactoryBeans之一。 (请记住,一个工厂 介绍了一个间接层bean,使它创建的对象 不同的类型)。

注意

Spring 2.0 AOP支持下还使用工厂bean 覆盖。

基本的方法来创建一个AOP代理在春天是使用 org.springframework.aop.framework.ProxyFactoryBean 。 这给完全控制切入点和通知,将申请, 和它们的顺序。 然而,有简单的选项是可取的 如果你不需要这样的控制。

10.5.1A基础知识

这个 ProxyFactoryBean 像其他弹簧 FactoryBean 实现,介绍一个级别的 间接寻址。 如果你定义一个 ProxyFactoryBean 与名称 foo ,什么对象引用 foo 看到的并不是 ProxyFactoryBean 实例本身,而是一个对象 创造的 ProxyFactoryBean 的实施 这个 getObject() 法。 该方法将创建一个 AOP代理包装一个目标对象。

最重要的优点之一,使用 ProxyFactoryBean 或另一个奥委会知道类来创建 AOP代理,是这意味着建议和切入点也可以 国际奥委会所管理的。 这是一个强大的功能,使某些方法 这是很难实现与其他AOP框架。 例如,一个 建议参考应用程序对象本身(除了目标, 这应该可以在任何AOP框架),受益于所有的吗 可插入性提供了依赖注入。

10.5.2A JavaBean属性

同大多数 FactoryBean 实现提供弹簧, ProxyFactoryBean 类本身就是一个JavaBean。 它的 属性用来:

- 指定目标你想代理。
- 指定是否使用CGLIB(见下文和也 SectionA 10 5 3,一个JDK -和proxiesacglib的基础)。

一些关键属性是继承 org.springframework.aop.framework.ProxyConfig (父类的所有AOP代理工厂在春天)。 这些关键 属性包括:

- proxyTargetClass : 真正的 如果目标类进行代理,而非目标类的 接口。 如果这个属性值设置为 真正的 ,然后CGLIB代理将被创建(参见 也 SectionA 10 5 3,一个JDK -和proxiesacglib的基础)。
- 优化 :控制是否或不是 积极优化应用于代理 创建 通过CGLIB 。 人们不应轻率地使用此设置 除非一个人完全了解有关AOP代理处理 优化。 这是目前仅用于CGLIB代理;它 没有效应与JDK动态代理。
- 冻 :如果一个代理配置 冻 ,然后更改配置都没有 不再允许。 这是有用的作为一个轻微的优化和 这些情况下当你不想让来电者能够操纵 代理(通过 建议 接口) 在代理已经创建。 此属性的默认值 是 假 ,所以变化如添加额外的 的建议是允许的。
- exposeProxy :决定是否或不是 目前的代理应该是公开的 ThreadLocal 所以,它可以访问的 目标。 如果一个目标需要获得代理和 exposeProxy 属性设置为 真正的 ,目标可以使用 AopContext.currentProxy() 法。

其他属性具体到 ProxyFactoryBean 包括:

- proxyInterfaces :字符串数组,数组的接口 的名字。 如果这不是提供,CGLIB代理的目标类 将被使用(但亦见吗 SectionA 10 5 3,一个JDK -和proxiesacglib的基础)。
- interceptorNames :字符串数组的 顾问 ,拦截器或其他建议 名称适用。 的顺序是重要的,在第一次名列第一 服务为基础。 也就是说,在列表中第一个拦截器 将会是第一个可以截取调用。
  这个名字是bean名称在当前的工厂,包括 从祖先工厂bean名称。 你不能提到bean 这里引用自这样做会导致 ProxyFactoryBean 忽略singleton 设置的建议。
  你可以添加一个拦截器的名字与一个星号 (*)。 这将导致所有的应用 顾问bean与名称开始的前一部分的星号 应用。 使用这个特性的一个示例中可以找到 SectionA 10 5 6,一个使用“全球化”advisorsa 。
- 单例:工厂是否应该返回一个单一的 对象,不管多久 getObject() 方法被调用。 几个 FactoryBean 的实现提供了这样一个方法。 默认值是 真正的 。 如果你想要使用有状态的建议——对 例,对有状态的混合——使用原型建议加上 单件价值 假 。

10.5.3A JDK -和cglib建立代理

这部分作为明确的文档上 ProxyFactoryBean 选择创建一个要么 一个JDK -和基于cglib代理特定目标对象(这是 是代理)。

> **注意**
>
> 的行为 ProxyFactoryBean 与 关于创建JDK -或cglib基于代理之间变化 版本1.2。 x和2.0的春天。 这个 ProxyFactoryBean 现在展览相似的语义 关于接口的自动的 TransactionProxyFactoryBean 类。

如果类的目标对象,进行代理(以下简称 简单地称为目标类)不实现任何 接口,然后基于cglib代理将被创建。 这是 简单的情况,因为代理是接口基于JDK和没有 接口意味着JDK代理不可能。 一个简单的插头在 目标bean,并指定拦截器的列表通过 interceptorNames 财产。 注意,一个基于cglib 代理将创建即使 proxyTargetClass 财产的 ProxyFactoryBean 已经设置为 假 。 (很明显这毫无意义,是最好的 将bean定义,因为它是在最好的冗余,在 糟糕的混乱。)

如果目标类实现一个(或多个)接口,那么 类型的代理创建依赖于配置的 ProxyFactoryBean 。

如果 proxyTargetClass 财产的 ProxyFactoryBean 已经设置为 真正的 ,那么将创建基于cglib代理。 这 是有意义的,是符合最少意外原则。 即使 proxyInterfaces 财产的 ProxyFactoryBean 已经被设置为一个或多个 完全限定的接口名称,这一事实 proxyTargetClass 属性设置为 真正的 将 导致基于cglib 代理在效应。

如果 proxyInterfaces 财产的 ProxyFactoryBean 已经被设置为一个或多个 完全限定的接口名称,然后代理将被创建为基础的jdk。 创建的代理将执行所有的接口都是 指定的 proxyInterfaces 房地产; 目标类实现一个整体发生更多的接口比 指定的 proxyInterfaces 房地产,这是 很好但这些额外的接口将不会 实现返回的代理。

如果 proxyInterfaces 财产的 ProxyFactoryBean 已经 不 被设置,但目标类 并实现一个(或 更多) 接口,那么 ProxyFactoryBean 将自动检测这一事实吗 目标类并实现至少一个接口和一个 代理将被创建为基础的jdk。 接口,实际上是 代理将 所有 的接口 目标类实现;实际上,这是仅仅供应一样 一个列表的每个接口,目标类实现 这个 proxyInterfaces 财产。 然而,

它是 更少的工作,并且更不容易输入错误。

10.5.4A代理接口

让我们看一个简单的例子 ProxyFactoryBean 在行动。 这个例子 包括:

- 一个 目标bean 那将是代理。 这 是 “personTarget “bean定义在下面的例子。
- 一个顾问和拦截器用来提供建议。
- AOP代理bean定义指定目标对象(personTarget bean)和接口代理,以及 建议申请。

```
<bean id="personTarget" class="com.mycompany.PersonImpl">
  <property name="name" value="Tony"/>
  <property name="age" value="51"/>
</bean>

<bean id="myAdvisor" class="com.mycompany.MyAdvisor">
  <property name="someProperty" value="Custom string property value"/>
</bean>

<bean id="debugInterceptor" class="org.springframework.aop.interceptor.DebugInterceptor">
</bean>

<bean id="person"
  class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="proxyInterfaces" value="com.mycompany.Person"/>

  <property name="target" ref="personTarget"/>
  <property name="interceptorNames">
    <list>
      <value>myAdvisor</value>
      <value>debugInterceptor</value>
    </list>
  </property>
</bean>
```

注意, interceptorNames 财产需要 字符串列表:bean的名称或顾问的拦截器 目前的工厂。 顾问,拦截器,前,后返回和 建议可以使用抛出对象。 顾问的排序是 显著的。

> **注意**
>
> 你也许想知道为什么这个列表并不持有bean 引用。 原因是,如果ProxyFactoryBean的 单例属性设置为false,它必须能够返回 独立代理实例。 如果任何顾问本身就是一个 原型,一个独立的实例需要返回,所以它的 必要的能够获得的一个实例的原型 工厂;持有一个参考不是足够的。

“人” bean定义可以使用上面的地方 人的实现,如下所示:

```
Person person = (Person) factory.getBean("person");
```

在相同的奥委会其他bean上下文可以表达一个强类型 依赖它,像一个普通的Java对象:

```
<bean id="personUser" class="com.mycompany.PersonUser">
 <property name="person"><ref local="person"/></property>
</bean>
```

这个 PersonUser 类在这个例子会 揭露一个属性的类型的人。 至于它的关注,AOP 代理可以使用透明地在地方 “真实” 的人实现。 然而,它的类将是一个动态代理类。 它 可能丢给吗 建议 接口 (下面讨论)。

它可以隐藏目标和代理之间的区别 使用一个匿名 内在bean ,如下所示。 只有 ProxyFactoryBean 定义是不同的,建议 是只包括的完整性:

```
<bean id="myAdvisor" class="com.mycompany.MyAdvisor">
 <property name="someProperty" value="Custom string property value"/>
</bean>

<bean id="debugInterceptor" class="org.springframework.aop.interceptor.DebugInterceptor"/>

<bean id="person" class="org.springframework.aop.framework.ProxyFactoryBean">
 <property name="proxyInterfaces" value="com.mycompany.Person"/>
 <!-- Use inner bean, not local reference to target -->
 <property name="target">
  <bean class="com.mycompany.PersonImpl">
   <property name="name" value="Tony"/>
   <property name="age" value="51"/>
  </bean>
 </property>
```

```
  <property name="interceptorNames">
    <list>
      <value>myAdvisor</value>
      <value>debugInterceptor</value>
    </list>
  </property>
</bean>
```

这个的优点是只有一个类型的对象 人 :如果我们希望防止有用的用户 应用程序上下文获取一个引用联合国建议的对象, 或需要避免歧义与Spring IoC 自动装配 。 也可以说是一个优势 的定义是独立的。ProxyFactoryBean 然而,有 有时能够获得联合国建议的目标吗 工厂可能实际上是一个 优势 :对于 例,在特定的测试场景。

10.5.5A代理类

如果你需要代理类,而不是一个或更多 接口?

想象一下,在我们的例子中,没有 人 接口:我们需要建议一个类称为 人 没有实现任何业务接口。 在这种情况下,您可以配置弹簧使用CGLIB代理,而 比动态代理。 只要设置 proxyTargetClass 属性为true的上面ProxyFactoryBean。 虽然最好 程序接口,而不是类的能力,建议 实现接口的类不可能是有用的在处理 遗留代码。 (一般来说,春天不是规定性的。 虽然它使它 容易申请好实践,它避免了迫使一个特定的 方法。)

如果你想,你可以强制使用CGLIB在任何情况下,即使 如果你有接口。

CGLIB代理作品通过生成目标类的一个子类 在运行时。 弹簧配置这个生成子类来委派方法 调用原始目标:子类是用来实现 装饰模式,编织的建议。

CGLIB代理一般应该是透明的,用户。 然而, 有一些问题需要考虑:

- 最后 方法不能被建议,因为他们 不能被覆盖。
- 没有需要添加到classpath CGLIB。 像春天的 3.2,CGLIB被重新包装,包括在spring核心JAR。 在 句话说,cglib基于AOP将工作“开箱即用”的会做的一样 JDK动态代理。

有小的性能区别,CGLIB代理 动态代理。 在Spring 1.0中,动态代理是稍快。 然而,这可能会改变在未来。 性能不应该 决定性的考虑在这种情况下。

10.5.6A使用“全球”顾问

通过附加一个星号,拦截器的名字,所有的顾问与 bean名称匹配前面的部分星号,将被添加到 顾问链。 这可以派上用场,如果你需要添加一组标准 “全球”顾问:

```
<bean id="proxy" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target" ref="service"/>
  <property name="interceptorNames">
    <list>
      <value>global*</value>
    </list>
  </property>
</bean>

<bean id="global_debug" class="org.springframework.aop.interceptor.DebugInterceptor"/>
<bean id="global_performance" class="org.springframework.aop.interceptor.PerformanceMonitorInterceptor"/>
```

10.6一个简洁的代理定义

特别是当定义事务代理,你可能最终得到 许多类似的代理定义。 父母和孩子的使用bean 定义,以及内心的bean定义,可以导致更加简洁 和更简洁的代理定义。

首先一个家长, 模板 ,bean的定义是 创建的代理:

```
<bean id="txProxyTemplate" abstract="true"
      class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
  <property name="transactionManager" ref="transactionManager"/>
  <property name="transactionAttributes">
    <props>
      <prop key="*">PROPAGATION_REQUIRED</prop>
    </props>
  </property>
</bean>
```

这将永远不会被实例化本身,所以实际上可能 不完整的。 然后每个代理需要创建bean只是个孩子 定义,它将目标的代理作为一

个内部bean 定义,因为目标永远不会被用在它自己的 不管怎样。

```
<bean id="myService" parent="txProxyTemplate">
  <property name="target">
    <bean class="org.springframework.samples.MyServiceImpl">
    </bean>
  </property>
</bean>
```

当然可能覆盖属性从父 模板,如在这种情况下,事务传播 设置:

```
<bean id="mySpecialService" parent="txProxyTemplate">
  <property name="target">
    <bean class="org.springframework.samples.MySpecialServiceImpl">
    </bean>
  </property>
  <property name="transactionAttributes">
    <props>
      <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>
      <prop key="find*">PROPAGATION_REQUIRED,readOnly</prop>
      <prop key="load*">PROPAGATION_REQUIRED,readOnly</prop>
      <prop key="store*">PROPAGATION_REQUIRED</prop>
    </props>
  </property>
</bean>
```

注意,在上面的例子中,我们有明确标志着父母 bean定义为 文摘 通过使用 文摘 属性,描述 以前 ,所以它可能 实际上不是曾经被实例化。 应用程序上下文(但不是简单的 bean工厂)将默认情况下预实例化所有单件。 这是 因此重要的(至少对单例bean),如果你有一个 (父)bean定义你想只使用作为模板, 这个定义指定了一个类,您必须确保设置 文摘 属性 真正的 , 否则,应用程序上下文实际上会尝试预实例化 它。

10.7一个AOP代理以编程方式创建与ProxyFactory

它很容易使用Spring AOP代理以编程方式创建。 这 使您能够使用Spring AOP没有依赖Spring IoC。

以下清单显示了创建一个代理为目标对象, 与一个拦截器和一个顾问。 接口实现的 目标对象将自动代理:

```
ProxyFactory factory = new ProxyFactory(myBusinessInterfaceImpl);
factory.addAdvice(myMethodInterceptor);
factory.addAdvisor(myAdvisor);
MyBusinessInterface tb = (MyBusinessInterface) factory.getProxy();
```

第一步是构建一个对象的类型 org.springframework.aop.framework.ProxyFactory 。 你可以 创建这个与目标对象,在上面的示例中,或指定 接口是在另一个构造函数,代理

您可以添加建议(拦截器作为一种专业类型的建议) 和/或顾问,和操纵他们的终身ProxyFactory。 如果你添加一个IntroductionInterceptionAroundAdvisor,可以导致代理 实现其他接口。

也有方便的方法在ProxyFactory(被继承 AdvisedSupport),这允许您添加其他建议 类型如之前和抛出的建议。AdvisedSupport是父类的 两ProxyFactory和ProxyFactoryBean。

提示

AOP代理创建集成与IoC框架是最好的 实践在大多数应用程序。 我们建议你具体化 配置从Java代码使用AOP,如一般。

10.8一个操纵建议对象

然而你创建AOP代理,你可以操纵他们使用 org.springframework.aop.framework.Advised 接口。 任何AOP代理可以转换为这个接口,接口的任何其他 实现了。 这个接口包括以下方法:

```
Advisor[] getAdvisors();

void addAdvice(Advice advice) throws AopConfigException;

void addAdvice(int pos, Advice advice)
      throws AopConfigException;

void addAdvisor(Advisor advisor) throws AopConfigException;

void addAdvisor(int pos, Advisor advisor) throws AopConfigException;
```

```
int indexOf(Advisor advisor);

boolean removeAdvisor(Advisor advisor) throws AopConfigException;

void removeAdvisor(int index) throws AopConfigException;

boolean replaceAdvisor(Advisor a, Advisor b) throws AopConfigException;

boolean isFrozen();
```

这个 getAdvisors() 方法将返回一个顾问 对于每一个顾问,拦截器或其他建议,已添加到类型 工厂。 如果你添加了一个顾问,顾问在这个索引返回 将您添加的对象。 如果你添加了一个拦截器或其他 建议类型,弹簧将包裹这在一个顾问与切入点 总是返回true。 因此如果你添加了一个 MethodInterceptor ,返回这个索引advisor 将是一个 DefaultPointcutAdvisor 返回你的 MethodInterceptor 和一个切入点匹配所有 类和方法。

这个 addAdvisor() 方法可以用来添加任何 顾问。 通常顾问控股切入点和通知将是 通用 DefaultPointcutAdvisor ,可以使用任何建议或切入点(但不是介绍)。

默认情况下,它可以添加或删除顾问或拦截器 即使已经创建一个代理。 唯一的限制是,它是 无法添加或删除一个介绍顾问,因为现有的代理 从工厂将不会显示界面变化。 (你可以获得一个新的 代理从工厂来避免这个问题。)

一个简单的例子,铸造一个AOP代理 建议 接口和检查和处理 建议:

```
Advised advised = (Advised) myObject;
Advisor[] advisors = advised.getAdvisors();
int oldAdvisorCount = advisors.length;
System.out.println(oldAdvisorCount + " advisors");

// Add an advice like an interceptor without a pointcut
// Will match all proxied methods
// Can use for interceptors, before, after returning or throws advice
advised.addAdvice(new DebugInterceptor());

// Add selective advice using a pointcut
advised.addAdvisor(new DefaultPointcutAdvisor(mySpecialPointcut, myAdvice));

assertEquals("Added two advisors",
    oldAdvisorCount + 2, advised.getAdvisors().length);
```

注意

这是怀疑它是可取的(没有双关) 修改一个业务对象的建议在生产,虽然没有 怀疑合法使用案例。 然而,它可以是非常有用的 发展:例如,在测试。 我有时会发现它非常 有用的能够添加测试代码的形式,一个拦截器或其他建议,进入一个方法调用我想测试。 (例,建议可以在一个事务创建的 方法:例如,运行SQL检查数据库是正确的 更新之前,标志着对回滚事务。)

这取决于你如何创建代理,您通常可以设置一个 冻 旗,在这种情况下 建议 isFrozen() 方法将 返回true,任何试图修改建议通过添加或删除 将导致一个 AopConfigException 。 能够 冻结的状态建议对象有用的在某些情况下,对于 例,以防止调用代码删除安全拦截器。 它可能 也可用于Spring 1.1允许积极优化如果运行时 建议修改是不需要知道。

10.9一个使用“汽车代理”设施

到目前为止,我们已经被认为是显式创建的AOP代理使用 ProxyFactoryBean 或类似的工厂bean。

春天还允许我们使用“汽车代理“bean定义,它可以 自动代理选定的bean定义。 这是建立在春天 “豆后置处理程序”基础设施,允许修改任何 作为容器加载bean定义。

在这个模型中,您将设置一些特殊bean定义XML bean定义文件来配置自动代理的基础设施。 这 允许您声明的目标符合汽车代理:你 不需要使用 ProxyFactoryBean 。

有两种方法可以做到这一点:

- 使用一个自动代理的创造者,是指特定的豆子在 当前上下文。
- 一个特殊情况的汽车代理创建,值得 单独考虑;汽车代理创建源代码级别的驱动 元数据属性。

10.9.1A火狐的一个插件bean定义

这个 org.springframework.aop.framework.autoproxy 包提供了以下标准汽车代理的创造者。

BeanNameAutoProxyCreator

这个 BeanNameAutoProxyCreator 类是一个 BeanPostProcessor 自动创建AOP 代理bean与名称匹配的文字值或 通配符。

```
<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
 <property name="beanNames" value="jdk*,onlyJdk"/>
 <property name="interceptorNames">
  <list>
   <value>myInterceptor</value>
  </list>
 </property>
</bean>
```

与 ProxyFactoryBean ,有一个 interceptorNames 财产而不是一个列表的 拦截器,允许正确行为的原型顾问。 命名 “拦截器” 可以顾问或任何类型的建议。

与汽车代理一般来说,主要的点使用 BeanNameAutoProxyCreator 是应用相同的吗 一直到多个对象的配置,以最小的体积的配置。 这是一个受欢迎的选择申请声明 交易到多个对象。

Bean定义的名字匹配,如 “jdkMyBean” 和 “onlyJdk” 在上面的例子中,是与普通的旧bean定义 目标类。 AOP代理将被自动创建 BeanNameAutoProxyCreator 。 同样的建议 应用到所有匹配的豆子。 注意,如果使用顾问(相当 比拦截器在上面的例子中),可以申请的切入点 不同不同的豆子。

DefaultAdvisorAutoProxyCreator

一个更一般的和非常强大的汽车代理创造者 DefaultAdvisorAutoProxyCreator 。 这将 自动应用合格的顾问在当前背景下,没有 需要包括特定bean的名字在汽车代理顾问的 bean定义。 它提供了相同的价值一致的配置 和避免重复作为 BeanNameAutoProxyCreator 。

使用此机制包括:

- 指定 DefaultAdvisorAutoProxyCreator bean 定义。
- 指定任意数量的顾问在同一或相关 上下文。 注意,这些 必须 是顾问, 不只是拦截器或其他建议。 这是必要的,因为 必须有一个切入点来评估、检查合格证的 每个建议候选人bean定义。

这个 DefaultAdvisorAutoProxyCreator 将 自动评价包含在每个顾问的切入点,看看 什么(如果有的话)的建议应该适用于每个业务对象(如 “businessObject1” 和 “businessObject2” 的例子)。

这意味着任何数量的顾问可以应用 自动为每个业务对象。 如果没有切入点的 顾问的任何方法相匹配的业务对象,该对象将不会是代理。 作为bean定义添加新的业务对象, 他们将会被自动代理如果必要的。

一般自动代理的优势使它 不可能获得一个调用者或依赖项联合国建议的对象。 调用getBean(“businessObject1”)在这个 ApplicationContext将 返回一个AOP代理,而不是目标业务对象。 (“内在豆” 成语前面显示还提供这种好处。)

```
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>

<bean class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
 <property name="transactionInterceptor" ref="transactionInterceptor"/>
</bean>

<bean id="customAdvisor" class="com.mycompany.MyAdvisor"/>

<bean id="businessObject1" class="com.mycompany.BusinessObject1">
 <!-- Properties omitted -->
</bean>

<bean id="businessObject2" class="com.mycompany.BusinessObject2"/>
```

这个 DefaultAdvisorAutoProxyCreator 非常 有用的,如果你想应用相同的建议始终对许多 业务对象。 一旦基础设施定义就位, 你可以简单的添加新的业务对象不包括特定的 代理的配置。 你也可以减少在额外的方面非常 很容易——例如,跟踪或性能监控方面, 最小的变化来配置。

DefaultAdvisorAutoProxyCreator提供支持的过滤 (使用一个命名约定,以便只有特定的顾问 评估,允许使用多个不同的配置, AdvisorAutoProxyCreators在同一个工厂)和排序。 顾问 可以实现 org.springframework.core.Ordered 接口,以确保正确的订购,如果这是一个问题。 这个 TransactionAttributeSourceAdvisor用在上面的例子中有一个 可配置的秩序价值;默认设置是无序的。

AbstractAdvisorAutoProxyCreator

这是DefaultAdvisorAutoProxyCreator的超类。 你 可以创建自己的汽车代理创作者通过子类化这类,在吗 这是不大可能发生的顾问定义提供不足 定制的行为的框架 DefaultAdvisorAutoProxyCreator 。

10.9.2A使用元数据驱动的汽车代理

一个特别重要的类型的汽车代理是由 元数据。 这就形成了一个类似的编程模型来。net ServicedComponents 。 而不是定义元数据 在XML描述符、事务管理和配置 其他企业服务是关押在源代码级别的属性。

在本例中,您使用 DefaultAdvisorAutoProxyCreator ,结合 顾问,理解元数据属性。 元数据的细节 在切入点中举行的部分候选人顾问,而不是 汽车代理创建类本身。

这是真正的一个特例 DefaultAdvisorAutoProxyCreator ,但是值得 考虑自己的。 (元数据知道代码是在切入点 中包含的顾问,而不是AOP框架本身。)

这个 /属性 JPetStore目录的 示例应用程序展示了使用属性驱动的汽车代理。 在这种情况下,不需要使用 TransactionProxyFactoryBean 。 简单地定义 事务属性在业务对象是足够的,因为 使用元数据清楚的切入点。 该bean定义包括 下面的代码,在 / - inf / declarativeServices.xml 。 注意,这是通用的,可以用JPetStore外:

```
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>

<bean class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
 <property name="transactionInterceptor" ref="transactionInterceptor"/>
</bean>

<bean id="transactionInterceptor"
   class="org.springframework.transaction.interceptor.TransactionInterceptor">
 <property name="transactionManager" ref="transactionManager"/>
 <property name="transactionAttributeSource">
   <bean class="org.springframework.transaction.interceptor.AttributesTransactionAttributeSource">
     <property name="attributes" ref="attributes"/>
   </bean>
 </property>
</bean>

<bean id="attributes" class="org.springframework.metadata.commons.CommonsAttributes"/>
```

这个 DefaultAdvisorAutoProxyCreator bean 定义(这个名字并不重要,因此它甚至可以省略) 将接所有合格的切入点在当前应用程序上下文。 在这种情况下,“transactionAdvisor“bean定义,类型 TransactionAttributeSourceAdvisor ,将适用于 类或方法携带事务属性。 这个 TransactionAttributeSourceAdvisor取决于TransactionInterceptor, 通过构造函数依赖。 示例解决这个通过自动装配。 这个 AttributesTransactionAttributeSource 取决于 的实现 org.springframework.metadata.Attributes 接口。 在 这个片段,“属性” bean,用满足这个雅加达 共享属性的API来获取属性信息。 (应用程序 代码必须被编译使用共享属性编译 任务。)

这个 /注释 JPetStore目录的 示例应用程序包含了一个类似的例子,汽车的代理 由JDK 1.5 +注释。 下面的配置使 Spring的自动检测 事务性 注释,导致隐性代理bean包含这个 注释:

```
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>

<bean class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
 <property name="transactionInterceptor" ref="transactionInterceptor"/>
</bean>

<bean id="transactionInterceptor"
   class="org.springframework.transaction.interceptor.TransactionInterceptor">
 <property name="transactionManager" ref="transactionManager"/>
 <property name="transactionAttributeSource">
   <bean class="org.springframework.transaction.annotation.AnnotationTransactionAttributeSource"/>
 </property>
</bean>
```

这个 TransactionInterceptor 这里定义取决于 在一个 PlatformTransactionManager 定义,它是 不包括在这个通用的文件(尽管它可以),因为它将 特定于应用程序的事务需求(通常是 JTA,因为在这个例子中,或Hibernate,JDO或者JDBC):

```
<bean id="transactionManager"
   class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

提示

如果你只需要声明式事务管理、使用 这些通用的XML定义会导致弹簧自动 代理所有的类或方法与事务属性。 你不会 需要直接处理AOP,和编程模型是相似的 到那的。 净ServicedComponents。

这种机制是可扩展的。 它可以做到自动代理 基于自定义属性。 你需要:

- 定义您的自定义属性。
- 指定一个顾问与必要的建议,包括 触发的切入点,存在的自定义属性 在一个类或方法。 你可以使用现有的建议, 只是实现一个静态的切入点,拿起自定义 属性。

有可能这样的顾问是不同的每个建议类 (例如,混合):他们只是需要被定义为原型, 而非单例,bean定义。 例如, LockMixin 介绍从春天拦截 测试套件,如上图所示,可以结合使用一个 属性驱动的切入点来目标mixin,如下所示。 我们使用 通用 DefaultPointcutAdvisor 、配置使用 JavaBean属性:

```
<bean id="lockMixin" class="org.springframework.aop.LockMixin"
   scope="prototype"/>

<bean id="lockableAdvisor" class="org.springframework.aop.support.DefaultPointcutAdvisor"
   scope="prototype">
  <property name="pointcut" ref="myAttributeAwarePointcut"/>
  <property name="advice" ref="lockMixin"/>
</bean>

<bean id="anyBean" class="anyclass" ...
```

如果属性清楚切入点匹配任何方法 anyBean 或其他bean定义,mixin将 应用。 请注意,这两个 LockMixin 和 lockableAdvisor 定义原型。 这个 myAttributeAwarePointcut 切入点可以是一个单例 定义,因为它不保存状态为个人建议 对象。

使用TargetSources 10.10

弹簧提供的概念 TargetSource , 表示在 org.springframework.aop.TargetSource 接口。 这个接口负责返回 “目标对象” 实现连接点。 这个 TargetSource 实现要求目标实例每次AOP代理 处理一个方法调用。

开发人员使用Spring AOP通常不需要直接 与TargetSources,但是这提供了一个强有力的工具支持 池、热交换和其他复杂的目标。 例如,一个 池TargetSource可以返回一个不同的目标实例为每个 调用,使用池来管理实例。

如果你不指定一个TargetSource,一个默认实现 使用一个本地对象的。 相同的目标是为每一个返回 调用(如你所愿)。

让我们看一下标准目标来源提供弹簧,和 如何使用它们。

> **提示**
>
> 当使用一个自定义的目标源的,你的目标通常会需要 是一个原型,而不是一个单例bean定义。 这允许 春天来创建一个新的目标实例需要时。

10.10.1A热可切换目标来源

这个 org.springframework.aop.target.HotSwappableTargetSource 的存在是为了让目标的AOP代理要切换,同时允许 调用者保持他们对它的引用。

改变目标源的目标立即生效。 这个 HotSwappableTargetSource 是线程安全的。

你可以改变目标通过 交换() 方法 在HotSwappableTargetSource如下:

```
HotSwappableTargetSource swapper =
   (HotSwappableTargetSource) beanFactory.getBean("swapper");
Object oldTarget = swapper.swap(newTarget);
```

所需的XML定义如下所示:

```
<bean id="initialTarget" class="mycompany.OldTarget"/>

<bean id="swapper" class="org.springframework.aop.target.HotSwappableTargetSource">
  <constructor-arg ref="initialTarget"/>
</bean>

<bean id="swappable" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="targetSource" ref="swapper"/>
</bean>
```

上面的 交换() 调用目标的变化 可切换的bean。 客户持有一个引用bean将 不知道这种改变,但是会立即开始触及新 目标。

尽管这个例子并没有添加任何建议,它不是 需要添加的建议使用 TargetSource —— 课程任何 TargetSource 可以一起使用吗具有任意的建议。

10.10.2A池目标源

使用池目标源提供了一个类似的编程模型 与无状态会话ejb,池相同的实例 的维护,将免费对象方法调用的 池。

一个关键的区别SLSB春池,池 那个春天池可以应用到任何一个POJO。 与春天在 一般,这个服务可以被应用于一种非侵入性的方法。

Spring提供了开箱即用的支持Jakarta Commons池 1.3,它提供了一个相当有效的池实现。 你会 需要在您的应用程序共享池Jar的类路径中使用这个 特性。 它也可以子类 org.springframework.aop.target.AbstractPoolingTargetSource 支持任何其他池API。

示例配置如下所示:

```
<bean id="businessObjectTarget" class="com.mycompany.MyBusinessObject"
   scope="prototype">
  ... properties omitted
</bean>

<bean id="poolTargetSource" class="org.springframework.aop.target.CommonsPoolTargetSource">
  <property name="targetBeanName" value="businessObjectTarget"/>
  <property name="maxSize" value="25"/>
</bean>

<bean id="businessObject" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="targetSource" ref="poolTargetSource"/>
  <property name="interceptorNames" value="myInterceptor"/>
</bean>
```

请注意,目标对象—— “businessObjectTarget” 例子—— 必须 是一个原型。 这允许 PoolingTargetSource 实现来创建新的 目标的实例来种植池是必要的。 看到javadoc 对于 AbstractPoolingTargetSource 和具体的 您希望使用子类的属性信息: “最大容量” 是最基本的,总是会出现。

在这种情况下, “myInterceptor” 是一个拦截器,该拦截器的名称 需要定义在相同的奥委会上下文。 然而,它不是 有必要指定拦截器使用池。 如果你只想要 池,没有其他建议,不要设置interceptorNames产权 所有。

可以配置弹簧以便能够施放 池对象的 org.springframework.aop.target.PoolingConfig 接口,它公开了信息和当前的配置 池的大小通过介绍。 您需要定义一个 顾问:

```
<bean id="poolConfigAdvisor" class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
  <property name="targetObject" ref="poolTargetSource"/>
  <property name="targetMethod" value="getPoolingConfigMixin"/>
</bean>
```

这个顾问是通过调用一个方便的方法获得的 AbstractPoolingTargetSource 类,因此使用 MethodInvokingFactoryBean。 这个顾问的名字(“poolConfigAdvisor” 这里)必须列表中的拦截器的名字在ProxyFactoryBean 暴露的合并对象。

演员将如下所示:

```
PoolingConfig conf = (PoolingConfig) beanFactory.getBean("businessObject");
System.out.println("Max pool size is " + conf.getMaxSize());
```

注意

池无状态的服务对象通常没有必要。 我们 不相信它应该是默认的选择,因为大多数无状态 对象自然是线程安全的,和实例池是有问题的 如果资源被缓存。

简单的连接池是可以使用汽车代理。 这是可能的 设置TargetSources使用任何自动代理的创造者。

10.10.3A原型目标源

建立一个 “原型” 目标源类似于一个池 TargetSource。 在这种情况下,一个新实例将创建的目标 在每个方法调用。 虽然在创建一个新对象的成本 不高在现代JVM,连接的成本的新对象 (满足其奥委会依赖性)可能更昂贵。 因此你 不应该用这种方法没有很好的理由。

要做到这一点,您可以修改 poolTargetSource 上面所示的定义如下。 (我也改变了名字,为了清晰。)

```
<bean id="prototypeTargetSource" class="org.springframework.aop.target.PrototypeTargetSource">
  <property name="targetBeanName" ref="businessObjectTarget"/>
</bean>
```

只有一个属性:目标bean的名称。 继承是用于TargetSource实现确保 一致的命名。 与池目标源、目标bean 必须是一个原型的bean定义。

10.10.4A ThreadLocal 目标来源

ThreadLocal 目标来源是有用的如果 你需要一个对象被创建为每个传入请求(每个线程 这是)。 的概念 ThreadLocal 提供一个jdk宽设施来透明地存储资源与一个线程。 设置一个 ThreadLocalTargetSource 相当 一样多的被解释为其他类型的目标 来源:

```
<bean id="threadlocalTargetSource" class="org.springframework.aop.target.ThreadLocalTargetSource">
  <property name="targetBeanName" value="businessObjectTarget"/>
</bean>
```

注意

有严重的问题(比如ThreadLocals)引用(可能导致 内存泄漏)当不正确的使用多线程和 多类加载器环境。 一个人应该总是考虑包装一个 在一些其他类和threadlocal从来没有直接使用 ThreadLocal 本身(除了当然的包装类)。 同样,一个人应该永远记住要正确设置和 扰乱(后者只需要调用 threadlocal集(空))资源的局部 线程。 取消在任何情况下应做自不取消它 可能会导致问题行为。 Spring的ThreadLocal支持 这是否对你和应该总是被认为是支持使用吗 没有其他合适的处理代码(比如ThreadLocals)引用。

10.11一个定义新的 建议 类型

Spring AOP设计是可扩展的。 而拦截 实现策略是目前在内部使用,它是可能的 支持任意的建议类型除了开箱即用 拦截around通知,之前,抛出建议和回国后 建议。

这个 org.springframework.aop.framework.adapter 包是一个SPI包允许支持新的自定义类型的建议 被添加在不改变核心框架。 唯一约束在一个 定制 建议 类型是必须的 实现 org.aopalliance.aop.Advice 标记接口。

请参考 org.springframework.aop.framework.adapter 包的 Javadocs为进一步的信息。

10.12进一步资源

请参阅示例应用程序的Spring为进一步的例子 Spring AOP的:

- 的默认配置的JPetStore演示了使用 TransactionProxyFactoryBean 声明性 事务管理。
- 这个 /属性 JPetStore目录的 演示了使用声明式事务的属性驱动 管理。

11.测试

11.1一个介绍弹簧测试

测试是不可分割的一部分,企业软件开发。 这 章主要是在国际奥委会的增值原则 单元测试 和春天的好处 框架的支持 集成 测试 。 (彻底治疗的测试在企业 超出了本文的范围参考手册。)

11.2一个单元测试

依赖注入应该让你的代码更少的依赖 集装箱比传统的Java EE开发。 pojo的 ,您的应用程序应该在JUnit测试或者TestNG测试,简单的实例化对象使用 新 运营商, 没有弹簧或其他容器 。 您可以使用 模拟对象 (结合其他 有价值的测试技术来测试您的代码)隔离。 如果你遵循 春天的架构建议,由此产生的清洁分层 你的代码库和组件将促进单元测试更加容易。 例如,您可以测试服务层对象存根或嘲笑道 或存储库的接口,而不需要访问持久数据而 运行单元测试。

真正的单元测试通常运行得非常快,因为没有 运行时基础设施设置。 强调真正的单元测试的一部分 你的开发方法将提高你的生产率。 你可能不需要 这部分的测试章节帮助你编写有效的单元测试 对于你的国际奥委会建立应用程序。 对于某些单元测试场景, 然而,Spring框架提供了以下模拟对象和 测试支持类。

11.2.1A模拟对象

环境

这个 org.springframework.mock.env 包 包含模拟的实现 环境 和 PropertySource 抽象介绍 Spring 3.1(见 SectionA 3.3,一个Abstractiona环境 和 SectionA 3.4,一个PropertySource Abstractiona)。 MockEnvironment 和 MockPropertySource 是有用的对于发展中 容器外 测试代码,取决于 与环境相关的属性。

JNDI

这个 org.springframework.mock.jndi 包 包含一个实现JNDI SPI,您可以使用它来设置 一个简单的JNDI环境测试套件或独立的应用程序。 如果,例如,JDBC 数据源 年代得到绑定到 相同的JNDI名称在测试代码在一个Java EE容器,你可以 两个应用程序 重用代码和配置在测试场景 没有修改。

Servlet API

这个 org.springframework.mock.web 包 包含一组全面的Servlet API模拟对象,针对 使用Spring的Web MVC框架,它是有用的为测试Web 上下文和控制器。 这些模拟对象通常更 方便的使用比动态模拟对象如 EasyMock 或现有的Servlet API mock 对象如 MockObjects 。

Portlet API

这个 org.springframework.mock.web.portlet 包包含一组Portlet API模拟对象,针对使用 与Spring MVC框架的Portlet。

11.2.2A单元测试支持类

通用实用工具

这个 org.springframework.test.util 包 包含 ReflectionTestUtils ,这是一个 基于映像的实用方法的集合。 开发者使用这些 方法在单元和集成测试场景中,他们需要 设置一个非 公共 字段或调用一个 非 公共 setter方法在测试应用程序 代码包括,例如:

- ORM框架如JPA和Hibernate,宽恕 私人 或 保护 领域 访问,而不是 公共 setter方法 属性在一个域的实体。
- Spring的支持等注释 @ autowired , @ inject ,和 @ resource, 提供依赖 注入 私人 或 保护 字段,setter方法, 配置方法。

Spring MVC

这个 org.springframework.test.web 包 包含 ModelAndViewAssert ,您可以使用它在 结合JUnit,TestNG,或其他任何测试框架为单位 测试处理Spring MVC ModelAndView 对象。

> **单元测试Spring MVC控制器**
>
> 测试你的Spring MVC 控制器 年代,使用 ModelAndViewAssert 结合 MockHttpServletRequest , MockHttpSession 等等的 org.springframework.mock.web 包。

11.3一个集成测试

11.3.1A概述

重要的是能够执行一些集成测试 不需要部署到应用程序服务器或连接 其他企业的基础设施。 这将使您能够测试这样的东西为:

- 正确的布线的Spring IoC容器上下文。
- 数据访问使用JDBC或一个ORM工具。 这将包括 事情的正确性,SQL语句,Hibernate查询,JPA 实体映射等。

Spring框架提供了一流的支持集成 测试在 弹簧测试 模块。 实际的名字JAR文件可能包括发行版本 ,也可能是长期的 org.springframework.test 形式,这取决于你在哪里 把它从(请参阅 部分 依赖管理 对于一个解释)。 这个库包含 这个 org.springframework.test 包,其中包含 有价值的类集成测试与Spring容器。 这 测试并不依赖于一个应用程序服务器或其他部署 环境。 这样的测试是运行慢比单元测试,但快得多 比同等的仙人掌测试或远程测试,依靠部署 到一个应用程序服务器。

在Spring 2.5和以后,单元测试和集成测试支持 的形式提供了注解驱动的 春天还是和TestContext框架 。 这个 还是和 TestContext框架是不可知论者的实际测试框架在使用中, 从而使仪器的测试在各种环境中包括 JUnit,TestNG,等等。

> **JUnit 3.8支持已被弃用**

Spring 3.0的,遗留JUnit 3.8基类的层次结构(例如, AbstractDependencyInjectionSpringContextTests , AbstractTransactionalDataSourceSpringContextTests , 等)是正式弃用,将被删除在后面的版本。 任何测试类基于这段代码应该迁移到 春天还是和TestContext 框架 。

在Spring 3.1,JUnit 3.8基类在春季 还是和TestContext框架(即, AbstractJUnit38SpringContextTests 和 AbstractTransactionalJUnit38SpringContextTests) 和 @ExpectedException 一直 正式弃用,将被删除在后面的版本。 任何测试 类基于此代码应该迁移到JUnit 4或者TestNG 提供支持 春天 还是和TestContext框架 。 同样,任何测试的方法 @ExpectedException 应该修改为 使用内置的支持在JUnit和预期的异常 TestNG。

11.3.2A目标的集成测试

Spring的集成测试支持具有以下主要 目标:

- 管理 Spring IoC 集装箱缓存 在测试执行。
- 提供 依赖 注入测试夹具实例 。
- 提供 事务 管理 适合集成测试。
- 供应 spring特定基类 ,帮助开发人员在编写集成测试。

接下来的几个小节描述每个目标并提供链接 实现和配置细节。

上下文管理和缓存

春天还是和TestContext框架提供了一致的加载 春天 ApplicationContext 年代和 WebApplicationContext 年代以及缓存 这些上下文。 支持缓存加载上下文很重要, 因为启动时间可以成为一个问题一个不是因为开销 春天的本身,而是因为对象实例化的春天 集装箱需要时间来实例化。 例如,一个项目与50 100的Hibernate映射文件可能需要10到20秒加载 映射文件,导致成本运行每个测试之前在 每一个测试夹具导致较慢的整体测试运行,减少 开发人员的生产力。

测试类通常声明数组要么 资源位置 对于XML配置元数据的一个 经常在类路径中一个或一组 注释 类 这是用来配置应用程序。 这些 位置或类是一样的或类似的规定 web . xml 或其他部署配置 文件。

默认情况下,加载后,配置的 ApplicationContext 重用对于每个 测试。 因此安装成本发生只有一次每个测试套件, 随后的测试执行速度更快。 在这种背景下,这个词 测试套件 意味着所有的测试运行在相同的JVM的一个 例如,所有的测试运行从一个Ant、Maven或构建一Gradle 给定的项目或模块。 在不太可能的情况下,一个测试腐蚀 应用程序上下文,需要重新加载一个例如,通过修改 bean定义或一个应用程序对象的状态一个还是和TestContext 框架可以配置为重新加载配置和重建 应用程序上下文在执行下一个测试。

看到 一个章节上下文managementa 和 一个章节上下文cachinga 还是和TestContext与 框架。

依赖注入的测试夹具

当加载应用程序还是和TestContext框架上下文,它 可以选择配置您的测试类实例通过依赖 注射。 这提供了一种方便的机制设置测试 夹具使用预配置的bean从您的应用程序上下文。 一个 强大的好处在于,您可以重用应用程序上下文跨越 不同的测试场景(如。 ,用于配置spring管理对象 图形、事务代理, 数据源 年代, 等),从而避免了需要复制复杂的测试夹具设置 对个人的测试用例。

作为一个例子,考虑的情况我们上课, HibernateTitleRepository ,实现了数据 访问逻辑为 标题 域的实体。 我们希望 编写集成测试,测试以下领域:

- Spring配置:基本上,就是一切有关 的配置 HibernateTitleRepository bean正确和 礼物吗?
- Hibernate映射文件配置:就是一切映射 正确,正确的延迟加载设置 地方吗?
- 的逻辑 HibernateTitleRepository :不同的配置 这个类的实例执行如预期的吗?

看到依赖注入的测试夹具的 还是和TestContext框架 。

事务管理

一个常见的问题在测试中,访问一个真正的数据库是他们的 影响持久性存储的状态。 甚至当你使用一个 开发数据库,更改状态可能会影响未来的测试。 同时,许多操作一个如插入或修改一个持久数据 不能被执行(或验证)外的事务。

还是和TestContext框架解决了这个问题。 默认情况下, 框架将创建和回滚事务为每个测试。 你 简单地编写代码,可以假定已

经存在一个事务。 如果你 调用代理对象在你的测试以事务的方式,他们将采取的行动 正确,根据其配置的事务语义。 在 另外,如果一个测试方法删除选中的内容表, 在事务管理的运行测试,事务 将回滚在默认情况下,数据库将会回到它的状态吗 执行测试之前。 事务支持提供一个 测试通过 PlatformTransactionManager bean 定义在测试的应用程序上下文。

如果你想要一个事务提交一个不寻常,但偶尔 当你想要一个特别有用的测试来填充或修改 数据库一个还是和TestContext框架可以指示导致 事务提交,而不是通过回滚 @TransactionConfiguration 和 @Rollback 注释。

看到事务管理的 还是和TestContext框架 。

支持类的集成测试

春天还是和TestContext框架提供了几个 文摘 支持类,简化编写的 集成测试。 这些基础的测试类提供定义良好的钩子 成测试框架以及方便的实例变量和 方法,使您能够访问:

- 这个 ApplicationContext ,执行 明确的bean查找或测试上下文的状态作为一个 整个。
- 一个 JdbcTemplate 为执行SQL 语句来查询数据库。 这样的查询可以被用来 确认数据库状态都 之前 和 在 执行数据库相关的应用程序 代码,和弹簧确保这样的查询运行的范围 相同的事务作为应用程序代码。 当结合使用 与一个ORM工具,一定要避免 假 阳性 。

此外,您可能想要创建自己的定制, 应用程序范围的超类与实例变量和方法具体 你的项目。

看到支持类 还是和TestContext 框架 。

11.3.3A JDBC测试支持

这个 org.springframework.test.jdbc 包 包含 JdbcTestUtils ,这是一个收集的 JDBC相关实用功能旨在简化标准数据库 测试场景。 注意, AbstractTransactionalJUnit4SpringContextTests 和 AbstractTransactionalTestNGSpringContextTests 提供便利的方法,委托 JdbcTestUtils 在内部。

这个 spring jdbc 模块提供支持 配置和启动嵌入式数据库可以使用 集成测试与数据库进行交互。 有关详细信息,请参见 SectionA 14.8,一个supporta嵌入式数据库 和 SectionA 14 8 8,测试数据访问逻辑与嵌入式数据库 。

11.3.4A注释

弹簧测试注释

Spring框架提供了以下组 spring特定 注释,可以使用 你的单元测试和集成测试结合还是和TestContext 框架。 参考各自的Javadoc为进一步的信息, 包括默认属性值,属性别名,等等。

- **@ContextConfiguration**
  定义了类级元数据,用于确定如何 加载和配置一个 ApplicationContext 集成 测试。 具体地说, @ContextConfiguration 声明 要么 应用程序上下文资源 位置 或 带注释的 类 这将用于加载 上下文。
  资源位置通常是XML配置文件 位于类路径;然而,带注释的类通常 @ configuration 类。 然而, 资源位置也可指文件在文件系统, 带注释的类可以组件类等。

```
@ContextConfiguration("/test-config.xml")
public class XmlApplicationContextTests {
    // class body...
}
```

```
@ContextConfiguration(classes = TestConfig.class)
public class ConfigClassApplicationContextTests {
    // class body...
}
```

  作为一种替代或除了声明资源 位置或注释的类, @ContextConfiguration 可用于 申报 ApplicationContextInitializer 类。

```
@ContextConfiguration(initializers = CustomContextIntializer.class)
public class ContextInitializerTests {
    // class body...
}
```

  @ContextConfiguration 可能 有选择地用于声明 ContextLoader 策略。 注意, 然而,你通常不需要显式地配置 装载机因为默认加载程序支持要么资源 位置 或注释 类 以及 初始化 。

```
@ContextConfiguration(locations = "/test-context.xml", loader = CustomContextLoader.class)
public class CustomLoaderXmlApplicationContextTests {
    // class body...
}
```

注意

@ContextConfiguration 提供支持 继承 资源 位置或配置类以及上下文初始化 默认的超类声明。

看到 一个章节上下文managementa 和 javadoc的 @ContextConfiguration 对于 进一步的细节。

- **@WebAppConfiguration**

  一个类级别注释,用于声明 ApplicationContext 加载一 集成测试应该是一个 WebApplicationContext 。 仅仅 存在 @WebAppConfiguration 在一个 确保测试类 WebApplicationContext 将加载 对于测试,使用默认值 “文件:src / main / webapp” 对于根的路径 web应用程序的(即。 , 资源基础 路径)。 资源的基本路径是在幕后使用 创建一个 MockServletContext 提供 随着 ServletContext 为测试的 WebApplicationContext 。

```
@ContextConfiguration
@WebAppConfiguration
public class WebAppTests {
    // class body...
}
```

以覆盖默认,指定一个不同的基础资源 路径通过 隐式 价值 属性。 两 类路径: 和 文件: 资源 前缀是支持的。 如果没有资源 前缀是提供了路径 被认为是一个文件系统资源。

```
@ContextConfiguration
@WebAppConfiguration("classpath:test-web-resources")
public class WebAppTests {
    // class body...
}
```

注意, @WebAppConfiguration 必须结合使用 @ContextConfiguration ,无论是在 一个测试类或在一个测试类层次结构。 看到 javadoc的 @WebAppConfiguration 对于 进一步的细节。

- **@ContextHierarchy**

  一个类级别注释,用于定义的层次结构 ApplicationContext 年代的集成 测试。 @ContextHierarchy 应该 声明一个列表的一个或多个 @ContextConfiguration 情况下,每 其中定义了一个水平层次的上下文。 以下 示例演示使用 @ContextHierarchy 在单个 测试类;然而, @ContextHierarchy 也可以使用 在一个测试类层次结构。

```
@ContextHierarchy({
    @ContextConfiguration("/parent-config.xml"),
    @ContextConfiguration("/child-config.xml")
})
public class ContextHierarchyTests {
    // class body...
}
```

```
@WebAppConfiguration
@ContextHierarchy({
    @ContextConfiguration(classes = AppConfig.class),
    @ContextConfiguration(classes = WebConfig.class)
})
public class WebIntegrationTests {
    // class body...
}
```

如果你需要合并或覆盖配置对于一个给定的 层次的上下文层次结构在一个测试类层次结构,你 必须显式地名字,水平通过提供相同的价值吗 名称 属性 @ContextConfiguration 在每个 相应级别的类层次结构。 看到 一个章节上下文hierarchiesa 和 javadoc的 @ContextHierarchy 对于 进一步的例子。

- **@ActiveProfiles**

  一个类级别注释,用于指明哪 Bean定义概要文件 应该主动当 加载一个 ApplicationContext 对于 测试类。

```
@ContextConfiguration
@ActiveProfiles("dev")
public class DeveloperTests {
    // class body...
}
```

```
@ContextConfiguration
@ActiveProfiles({"dev", "integration"})
public class DeveloperIntegrationTests {
    // class body...
```

```
}
```

注意

@ActiveProfiles 提供 支持 继承 活跃的bean定义 配置文件默认声明的超类。

看到 一个章节上下文配置与环境profilesa 和的Javadoc @ActiveProfiles 为例,进一步的细节。

- **@DirtiesContext**

  表明,底层的春天 ApplicationContext 一直 脏 在执行一个测试(即, 修改或破坏一个以某种方式为例,通过改变 一个单例bean的状态),应该关闭,不管 是否通过测试。 当一个应用程序上下文被标记 脏 ,这是远离测试 框架的缓存和关闭。 因此,底层的 Spring容器将重建对于任何后续测试 需要一个上下文相同的配置元数据。

  @DirtiesContext 可以作为 两类级别和方法级注释在相同的测试 类。 在这样的场景中, ApplicationContext 被标记为 脏 在任何这样的带注释的方法 作为在整个类。 如果 ClassMode 设置为 在每个测试方法 ,上下文是 标志着脏后班上每个测试方法。

  下面的例子解释当上下文会 脏为各种配置场景:

  - 在当前的测试类,当上声明一个类 与类模式设置为 下课后 (即。 , 默认类模式)。

    ```
    @DirtiesContext
    public class ContextDirtyingTests {
        // some tests that result in the Spring container being dirtied
    }
    ```

  - 在每个测试方法在当前的测试类,当 声明在类与类模式设置为 在每个测试方法。

    ```
    @DirtiesContext(classMode = ClassMode.AFTER_EACH_TEST_METHOD)
    public class ContextDirtyingTests {
        // some tests that result in the Spring container being dirtied
    }
    ```

  - 在当前的测试,当上声明一个方法。

    ```
    @DirtiesContext
    @Test
    public void testProcessWhichDirtiesAppCtx() {
        // some logic that results in the Spring container being dirtied
    }
    ```

  如果 @DirtiesContext 用于 测试其上下文配置作为一个上下文层次通过 @ContextHierarchy , hierarchyMode 标记可以用来控制如何 上下文缓存清除。 默认情况下一个 详尽的 算法将被使用,清除 上下文缓存不仅包括当前水平但也都 其他上下文层次结构,共享一个共同的祖先上下文 当前测试;所有 ApplicationContext 年代,驻留在一个 子层次的共同祖先上下文将被移除 上下文缓存和关闭。 如果 详尽的 算法忽略了特定的用例,简单 当前水平 算法可以指定 相反,如下所示。

  ```
  @ContextHierarchy({
      @ContextConfiguration("/parent-config.xml"),
      @ContextConfiguration("/child-config.xml")
  })
  public class BaseTests {
      // class body...
  }

  public class ExtendedTests extends BaseTests {

      @Test
      @DirtiesContext(hierarchyMode = HierarchyMode.CURRENT_LEVEL)
      public void test() {
          // some logic that results in the child context being dirtied
      }
  }
  ```

  为进一步的细节有关 详尽的 和 当前水平 算法的Javadoc中看到 DirtiesContext.HierarchyMode 。

- **@TestExecutionListeners**

  定义元数据配置这类级别 TestExecutionListener 年代应该 注册 TestContextManager 。 通常情况下, @TestExecutionListeners 是 结合使用 @ContextConfiguration 。

  ```
  @ContextConfiguration
  @TestExecutionListeners({CustomTestExecutionListener.class, AnotherTestExecutionListener.class})
  public class CustomTestExecutionListenerTests {
      // class body...
  }
  ```

  @TestExecutionListeners 支持 继承 听众默认情况下。 看到 Javadoc为例,进一步的细节。

- **@TransactionConfiguration**
  定义了类级元数据配置事务 测试。 具体地说,该bean的名称 PlatformTransactionManager 这 应该被用来驱动事务可以显式地指定如果 有多个bean类型 PlatformTransactionManager 在 测试的 ApplicationContext 如果 bean名称所需的 PlatformTransactionManager 不是 “transactionManager” 。 此外,您可以更改 defaultRollback 旗帜 假 。 通常情况下, @TransactionConfiguration 是结合使用 @ContextConfiguration 。

```
@ContextConfiguration
@TransactionConfiguration(transactionManager = "txMgr", defaultRollback = false)
public class CustomConfiguredTransactionalTests {
    // class body...
}
```

> **注意**
>
> 如果默认的惯例是足够的为您的测试 配置,您可以避免使用 @TransactionConfiguration 完全。 换句话说,如果你只有一个事务 经理一个或如果你有多个事务经理但 事务管理器测试被命名为“transactionManager”或 指定通过 TransactionManagementConfigurer 一个 如果你想交易自动回滚,然后 不需要注释您的测试类 @TransactionConfiguration 。

- **@Rollback**
  指示是否交易的注释测试 方法应该 回滚 在测试之后 法完成了。 如果 真正的 ,事务是 回滚;否则,提交事务。 使用 @Rollback 以覆盖默认 回滚国旗在类级配置。

```
@Rollback(false)
@Test
public void testProcessWithoutRollback() {
    // ...
}
```

- **@BeforeTransaction**
  指明注释的 公共无效 方法应该被执行 之前 一个事务 是开始测试方法的配置为运行在一个事务 通过 transactional 注释。

```
@BeforeTransaction
public void beforeTransaction() {
    // logic to be executed before a transaction is started
}
```

- **@AfterTransaction**
  指明注释的 公共无效 方法应该被执行 在 一个事务 已经结束的测试方法的配置为运行在一个事务 通过 transactional 注释。

```
@AfterTransaction
public void afterTransaction() {
    // logic to be executed after a transaction has ended
}
```

- **@NotTransactional**
  存在该注释指明注释的 测试方法必须 不 执行在一个事务 上下文。

```
@NotTransactional
@Test
public void testProcessWithoutTransaction() {
    // ...
}
```

> **@NotTransactional不**
>
> Spring 3.0的, @NotTransactional 是不宜用在 支持移动 事务性 测试 方法到一个单独的(非事务性)测试类或一个 @BeforeTransaction 或 @AfterTransaction 法。 作为一个 替代注释整个类 transactional ,考虑注释 个人方法 transactional ,这样做允许 混合的事务性和非事务性方法在相同的测试类,而不需要使用 @NotTransactional 。

标准注释支持

下面的注释是支持标准语义 所有配置的春天还是和TestContext框架。 注意, 这些注释并不特定于测试,可以在任何地方使用Spring框架。

- **@ autowired**
- **qualifier**
- **@ resource** (javax.annotation) 如果jsr - 250存在
- **@ inject** (javax.inject) 如果jsr - 330 礼物
- **@ named** (javax.inject) 如果jsr - 330 礼物
- **persistencecontext** (javax.persistence) 如果JPA存在
- **@PersistenceUnit** (javax.persistence) 如果JPA存在
- **@ required**
- **transactional**

> **jsr - 250生命周期注释**
>
> 在春天还是和TestContext框架 @PostConstruct 和 @PreDestroy 可使用标准吗 在任何应用程序组件语义中配置的 ApplicationContext ;然而,这些 生命周期内注释有有限的使用实际的测试 类。
>
> 如果一个方法在一个测试类标注 @PostConstruct ,该方法将 之前执行任何 之前 方法 底层测试框架(如。 、方法与JUnit的注释 @ before一样),这将适用于每一个 测试方法在测试类中。 另一方面,如果一个方法在一个 测试类标注 @PreDestroy ,该方法将 **从来没有** 被执行。 在一个测试类 因此推荐使用测试生命周期回调的 底层测试框架代替 @PostConstruct 和 @PreDestroy 。

弹簧JUnit测试注释

下面的注释是 只有 支持 当结合使用的 SpringJUnit4ClassRunner 或 这个 JUnit 支持类。

- **@IfProfileValue**
  指明注释的测试都启用了一个特定的 测试环境。 如果配置的 ProfileValueSource 返回一个匹配 价值 所提供的 名称 , 测试是启用的。 这个注释就可以应用到整个 类或个人方法。 类级别使用覆盖 方法级的用法。

```
@IfProfileValue(name="java.vendor", value="Sun Microsystems Inc.")
@Test
public void testProcessWhichRunsOnlyOnSunJvm() {
    // some logic that should run only on Java VMs from Sun Microsystems
}
```

  或者,您可以配置 @IfProfileValue 列表的 值 (或 语义) 实现testng像支持 测试组 在JUnit环境。 考虑下面的例子:

```
@IfProfileValue(name="test-groups", values={"unit-tests", "integration-tests"})
@Test
public void testProcessWhichRunsForUnitOrIntegrationTestGroups() {
    // some logic that should run only for unit and integration test groups
}
```

- **@ProfileValueSourceConfiguration**
  类级别注释指定什么类型的 ProfileValueSource 使用当检索 剖面值 配置通过 @IfProfileValue 注释。 如果 @ProfileValueSourceConfiguration 是 不宣布为测试, SystemProfileValueSource 被 默认的。

```
@ProfileValueSourceConfiguration(CustomProfileValueSource.class)
public class CustomProfileValueSourceTests {
    // class body...
}
```

- **@Timed**
  指明注释的测试方法必须完成执行 在指定的时间(以毫秒为单位)。 如果文本执行 时间超过指定时间,测试失败。 这个时期包括执行测试方法本身, 任何重复的测试(见 @Repeat),以及任何 设置 或 拆掉 的 测试夹具。

```
@Timed(millis=1000)
public void testProcessWithOneSecondTimeout() {
    // some logic that should not take longer than 1 second to execute
}
```

  春天的 @Timed 注释已经 不同的语义比JUnit的 @ test来(timeout =...) 支持。 具体来说,由于方式处理测试JUnit 执行超时(即,通过执行测试方法 单独 线程), @ test来(timeout =...) 适用于 每个迭代 对于重复和 先发制人地失败测试如果测试花费的时间太长。 春天的 @Timed 另一方面,乘以 总 测试执行时间(包括所有 重复)和不预先失败测试而是 等待测试完成之前失败的。
- **@Repeat**

指明注释的测试方法必须被执行 反复。 多少次,测试方法是 指定执行的注释。
执行的范围包括执行重复的 测试方法本身以及任何 设置 或 拆掉 的测试夹具。

```
@Repeat(10)
@Test
public void testProcessRepeatedly() {
    // ...
}
```

11.3.5A春天还是和TestContext框架

这个 春天 还是和TestContext 框架 (位于 org.springframework.test.context 包)提供 通用,注解驱动的单元测试和集成测试 支持 不可知论者的测试框架使用。 还是和TestContext框架还 地方很大的重视 约定优于 配置 与合理的默认值,可以被覆盖 通过基于注解的配置。

除了通用的测试基础设施,还是和TestContext 框架提供了明确的支持JUnit和TestNG的形式 文摘 支持类。 对于JUnit,春天也提供一个自定义的JUnit 跑 这允许 一个写所谓的 POJO测试类 。 POJO测试 类不需要扩展一个特定的类层次结构。

以下部分概述的内部构造 还是和TestContext框架。 如果你唯一感兴趣的就是使用这个框架 和不一定感兴趣与您自己的自定义扩展它 听众或自定义加载器,请直接去配置 (上下文管理 , 依赖注入 , 事务管理), 支持类 ,和 注释支持 部分。

关键抽象

框架的核心包括 还是和TestContext 和 TestContextManager 类和 TestExecutionListener , ContextLoader ,和 SmartContextLoader 接口。 一个 TestContextManager 上创建一个每个测试的基础 (如。 ,执行一个测试方法在JUnit)。 这个 TestContextManager 反过来管理一个 还是和TestContext 保存当前的上下文 测试。 这个 TestContextManager 还更新状态的 还是和TestContext 作为测试的过程 和代表 TestExecutionListener 年代, 这仪器实际测试执行通过提供依赖 注入,管理事务,等等。 一个 ContextLoader (或 SmartContextLoader)负责 加载一个 ApplicationContext 对于一个给定的 测试类。 请教Javadoc和弹簧测试套件进行进一步的 信息和各种实现的例子。

- 还是和TestContext :封装上下文 在这一个测试被执行,不可知论者的实际测试 框架在使用,并且提供了上下文管理和缓存支持测试实例,它是负责任的。 这个 还是和TestContext 也代表一个 ContextLoader (或 SmartContextLoader)加载一个 ApplicationContext 如果 要求。
- TestContextManager :主入口 点到 春天还是和TestContext框架 , 管理着一个 还是和TestContext 和 信号事件所有注册的 TestExecutionListener 年代在 定义良好的测试执行点:
  - 之前 类方法之前 的 一个特定的测试框架
  - 测试实例准备
  - 之前 方法之前 的 特定的测试框架
  - 在任何 在方法 的 特定的测试框架
  - 在任何 在类方法 的 特定的测试框架
- TestExecutionListener :定义 一个 侦听器 API,用于对测试执行 事件发表的 TestContextManager 与该侦听器注册。 Spring提供了四 TestExecutionListener 实现 这是默认配置: ServletTestExecutionListener , DependencyInjectionTestExecutionListener , DirtiesContextTestExecutionListener ,和 TransactionalTestExecutionListener 。 分别,他们支持Servlet API模拟为一个 WebApplicationContext 、依赖 注入的测试实例,处理 @DirtiesContext 注释, 与默认的回滚事务测试执行语义。
- ContextLoader :战略 界面介绍了Spring 2.5中加载一个 ApplicationContext 为一个集成 测试管理的春天还是和TestContext框架。
  在Spring 3.1,实现 SmartContextLoader 而不是这 接口以提供支持注释类和 活跃的bean定义概要文件。
- SmartContextLoader :扩展 的 ContextLoader 接口 Spring 3.1中引入的。
  这个 SmartContextLoader SPI 取代了 ContextLoader SPI, 介绍了Spring 2.5。 具体地说,一个 SmartContextLoader 可以选择 过程资源 位置 、注释 类 或上下文 初始化 。 此外,一个 SmartContextLoader 可以设置活动 bean定义概要文件的上下文,它加载。
  Spring提供了以下的实现:
  - DelegatingSmartContextLoader :一个 两个默认加载器内部的代表 AnnotationConfigContextLoader 或 GenericXmlContextLoader 既不同 在配置测试类的声明或 有默认位置或默认配置 类。
  - WebDelegatingSmartContextLoader : 两个默认加载器之一,代表们在内部一个 AnnotationConfigWebContextLoader 或 GenericXmlWebContextLoader 根据 无论是在配置宣布测试类或 有默认位置或默认配置 类。 一个web ContextLoader 将 只用于如果 @WebAppConfiguration 存在 在测试类。
  - AnnotationConfigContextLoader : 加载一个标准 ApplicationContext 从 注释类 。

- AnnotationConfigWebContextLoader : 加载一个 WebApplicationContext 从 注释类 。
- GenericXmlContextLoader :加载一个 标准 ApplicationContext 从 XML 资源位置 。
- GenericXmlWebContextLoader :加载一个 WebApplicationContext 从XML 资源位置 。
- GenericPropertiesContextLoader : 加载一个标准 ApplicationContext 从Java 属性文件。

接下来的小节解释如何配置 还是和TestContext 框架通过注释和 提供工作的例子,如何编写单元测试和集成测试 这个框架。

上下文管理

每个 还是和TestContext 提供上下文 管理和缓存支持测试实例是负责任的 对。 测试实例不自动接收访问 配置 ApplicationContext 。 然而, 如果一个测试类实现了 ApplicationContextAware 接口, 参考 ApplicationContext 提供 到测试实例。 注意, AbstractJUnit4SpringContextTests 和 AbstractTestNGSpringContextTests 实现 ApplicationContextAware 因此 提供访问 ApplicationContext 自动。

@ autowired ApplicationContext

作为一种替代方法来实现 ApplicationContextAware 界面,你 可以注入应用程序上下文为您的测试类通过 吗 @ autowired 注释可以在一个字段 或setter方法。 例如:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
public class MyTest {

    @Autowired
    private ApplicationContext applicationContext;

    // class body...
}
```

同样,如果您的测试配置加载 WebApplicationContext ,你可以注射 web应用程序上下文到您的测试如下:

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration
public class MyWebAppTest {

    @Autowired
    private WebApplicationContext wac;

    // class body...
}
```

依赖注入通过 @ autowired 提供的 DependencyInjectionTestExecutionListener 哪一个 默认配置(见吗 一个章节依赖注入的测试fixturesa)。

测试类,使用还是和TestContext框架不需要 扩展任何特定的类或实现特定的接口 配置他们的应用程序上下文。 相反,配置是实现 只需声明 @ContextConfiguration 注释在 类级别。 如果您的测试类没有显式地声明应用程序 上下文资源 位置 或注释 类 ,配置的 ContextLoader 决定如何加载 从默认的上下文位置或默认的配置类。 在 除了上下文资源 位置 和注释 类 ,一个应用程序上下文也可以 通过应用程序上下文配置 初始化 。

接下来的小节解释如何配置一个 ApplicationContext 通过XML配置 文件,带注释的类(一般 @ configuration 类),或上下文 初始化使用Spring的 @ContextConfiguration 注释。 或者,您可以实现您自己的自定义和配置 SmartContextLoader 对于高级使用 情况下。

上下文配置XML资源

加载一个 ApplicationContext 对于 你的测试使用XML配置文件,标注您的测试类 与 @ContextConfiguration 和 配置 位置 属性与一个数组 包含的资源位置的XML配置元数据。 一个 平原或相对路径一个例如 “上下文xml” 一个将被视为一个类路径 是相对于资源 包中定义的测试类。 一个路径开始 斜杠是当作一个绝对路径位置,例如 “/ org/example/config.xml” 。 一个路径代表 资源URL(即。 ,一个路径前缀 类路径: , 文件: , http: 等)将被使用 作为 是 。

```
@RunWith(SpringJUnit4ClassRunner.class)
// ApplicationContext will be loaded from "/app-config.xml" and
// "/test-config.xml" in the root of the classpath
@ContextConfiguration(locations={"/app-config.xml", "/test-config.xml"})
public class MyTest {
    // class body...
}
```

@ContextConfiguration 支持一个 别名 位置 属性通过 标准Java 价值 属性。 因此,如果你不 需要声明额外的属性 @ContextConfiguration ,你可以省略的 宣言 位置 属性名称和 声明的资源位置通过使用简写形式 通过以下例子。

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration({"/app-config.xml", "/test-config.xml"})
public class MyTest {
    // class body...
}
```

同时如果您省略 位置 和 价值 属性从 @ContextConfiguration 注释, 还是和TestContext框架将试图发现一个默认的XML资源 位置。 具体地说, GenericXmlContextLoader 检测到一个默认位置基于测试类的名称。 如果 你的类命名 com例子mytest , GenericXmlContextLoader 加载应用程序 上下文从 “类路径:/ com/example/mytest-context.xml” 。

```
package com.example;

@RunWith(SpringJUnit4ClassRunner.class)
// ApplicationContext will be loaded from
// "classpath:/com/example/MyTest-context.xml"
@ContextConfiguration
public class MyTest {
    // class body...
}
```

上下文配置使用带注释的类

加载一个 ApplicationContext 对于 你的测试用 注释类 (见 SectionA 5.12,一个configurationa基于java的容器),注释您的测试类 @ContextConfiguration 和配置 类 一个数组,其中包含属性 带注释的类的引用。

```
@RunWith(SpringJUnit4ClassRunner.class)
// ApplicationContext will be loaded from AppConfig and TestConfig
@ContextConfiguration(classes = {AppConfig.class, TestConfig.class})
public class MyTest {
    // class body...
}
```

如果您省略了 类 属性从 @ContextConfiguration 注释, 还是和TestContext框架将尝试检测存在违约 配置类。 具体地说, AnnotationConfigContextLoader 将检测所有 静态内部类的测试类,满足要求 配置类实现规定的Javadoc @ configuration 。 在接下来的 的例子, OrderServiceTest 类声明一个 静态内部配置类命名 配置 这将被自动用于加载 ApplicationContext 为测试类。 注意,配置类的名称是任意的。 在 另外,一个测试类可以包含多个静态内部 如果需要配置类。

```
@RunWith(SpringJUnit4ClassRunner.class)
// ApplicationContext will be loaded from the
// static inner Config class
@ContextConfiguration
public class OrderServiceTest {

    @Configuration
    static class Config {

        // this bean will be injected into the OrderServiceTest class
        @Bean
        public OrderService orderService() {
            OrderService orderService = new OrderServiceImpl();
            // set properties, etc.
            return orderService;
        }
    }

    @Autowired
    private OrderService orderService;

    @Test
    public void testOrderService() {
        // test the orderService
    }

}
```

混合的XML资源和注释的类

它有时是理想的混合XML资源和注释 类(即。 ,通常是 @ configuration 类)来配置一个 ApplicationContext 为你的测试。 对于 例子,如果你使用XML配置在生产中,您可能会决定 你想使用 @ configuration 类配置特定的spring管理组件为您的 测试,或反之亦然。 如前所述在 一个章节弹簧测试Annotationsa 还是和TestContext 框架不允许您声明 两 通过 @ContextConfiguration ,但这并 不意味着你不能同时使用。

如果你想要使用XML **和** @ configuration 类配置 你的测试,你将不得不选择一个作为 条目 点 ,你将需要包括或导入 其他的。 例如,在XML可以包含 @ configuration 类通过组件 扫描或定义为Spring bean,正常在XML;相反,在一个 @ configuration 类可以使用 @ImportResource 导入XML 配置文件。 注意,这个行为是语义上的 相当于你如何配置您的应用程序在生产:在 生产配置您将定义或者一组XML资源 位置或一组 @ configuration 类,生产 ApplicationContext 将被加载, 但你仍然可以自由添加或导入其他类型的 配置。

上下文配置与上下文初始化

配置一个 ApplicationContext 为测试使用 上下文初始化,注释您的测试类 @ContextConfiguration 和配置 初始化 一个数组,其中包含属性 引用的类,实现的 ApplicationContextInitializer 。 这个 宣布上下文初始化器将被用来初始化 ConfigurableApplicationContext 这是 为测试加载。 注意,具体 ConfigurableApplicationContext 类型 由每个宣布初始化必须兼容 类型的 ApplicationContext 由 这个 SmartContextLoader 在使用(即, 通常一个 GenericApplicationContext)。 此外,初始化的顺序调用依赖 他们是否实现弹簧的 下令 接口或标注 春天的 @Order 注释。

```
@RunWith(SpringJUnit4ClassRunner.class)
// ApplicationContext will be loaded from TestConfig
// and initialized by TestAppCtxInitializer
@ContextConfiguration(
    classes = TestConfig.class,
    initializers = TestAppCtxInitializer.class)
public class MyTest {
    // class body...
}
```

也可以省略的声明XML配置 文件或注释的类 @ContextConfiguration 完全和 相反只声明 ApplicationContextInitializer 类然后负责注册豆在语境中 例如,通过编程加载bean定义XML文件 或配置类。

```
@RunWith(SpringJUnit4ClassRunner.class)
// ApplicationContext will be initialized by EntireAppInitializer
// which presumably registers beans in the context
@ContextConfiguration(initializers = EntireAppInitializer.class)
public class MyTest {
    // class body...
}
```

上下文配置继承

@ContextConfiguration 支持 布尔 inheritLocations 和 inheritInitializers 属性,指示是否 资源位置或注释的类和上下文初始化 声明父类的应该 继承 。 这个 默认值为双方的旗帜是 真正的 。 这意味着 这一个测试类继承了资源位置或注释的类 以及上下文初始化父类声明的任何。 具体来说,资源位置或注释的类测试 类都添加到列表的资源位置或注释 类声明的超类。 同样,初始化对于一个 鉴于测试类将被添加到组定义的初始化 测试超类。 因此,子类可以选择 扩展 资源位置、注释 类,或上下文初始化。

如果 @ContextConfiguration ' s inheritLocations 或 inheritInitializers 属性设置为 假 、资源位置或注释的类 和上下文初始化器,分别为测试类 影子 和有效地替代配置 超类所定义的。

在接下来的例子中,使用XML资源的位置, ApplicationContext 对于 ExtendedTest 将加载从 “基本配置xml” **和** “扩展配置xml” , 这个顺序。 bean定义在 “扩展配置xml” 因此可能 覆盖 (即。 ,替换)中定义的 “基本配置xml” 。

```
@RunWith(SpringJUnit4ClassRunner.class)
// ApplicationContext will be loaded from "/base-config.xml"
// in the root of the classpath
@ContextConfiguration("/base-config.xml")
public class BaseTest {
    // class body...
}

// ApplicationContext will be loaded from "/base-config.xml" and
// "/extended-config.xml" in the root of the classpath
@ContextConfiguration("/extended-config.xml")
public class ExtendedTest extends BaseTest {
    // class body...
}
```

同样,在接下来的例子中,使用带注释的类, 这个 ApplicationContext 对于 ExtendedTest 将被加载的 BaseConfig **和** ExtendedConfig 类,这个顺序。 bean 定义在 ExtendedConfig 因此可能 覆盖(即。 ,替换)中定义的 BaseConfig 。

```
@RunWith(SpringJUnit4ClassRunner.class)
// ApplicationContext will be loaded from BaseConfig
@ContextConfiguration(classes = BaseConfig.class)
public class BaseTest {
```

```
    // class body...
}

// ApplicationContext will be loaded from BaseConfig and ExtendedConfig
@ContextConfiguration(classes = ExtendedConfig.class)
public class ExtendedTest extends BaseTest {
    // class body...
}
```

在接下来的例子中,使用上下文初始化器, ApplicationContext 对于 ExtendedTest 将初始化使用 BaseInitializer **和** ExtendedInitializer 。 但是请注意,初始化的顺序被调用 取决于他们是否实现弹簧的 下令 接口或标注 春天的 @Order 注释。

```
@RunWith(SpringJUnit4ClassRunner.class)
// ApplicationContext will be initialized by BaseInitializer
@ContextConfiguration(initializers=BaseInitializer.class)
public class BaseTest {
    // class body...
}

// ApplicationContext will be initialized by BaseInitializer
// and ExtendedInitializer
@ContextConfiguration(initializers=ExtendedInitializer.class)
public class ExtendedTest extends BaseTest {
    // class body...
}
```

上下文配置与环境概要文件

Spring 3.1引入了一流的支持框架 环境的概念和配置文件(即 bean 定义概要文件),和集成测试可以 配置为激活特定bean定义配置文件不同 测试场景。 这是通过一个测试类注解 这个 @ActiveProfiles 注释和 提供一个概要文件的列表应该激活当加载 ApplicationContext 对于测试。

> **注意**
>
> @ActiveProfiles 可以用 与任何实施新的 SmartContextLoader SPI,但 @ActiveProfiles 不支持 实现老 ContextLoader SPI。

让我们看看一些例子使用XML配置和 @ configuration 类。

```
<!-- app-config.xml -->
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:jdbc="http://www.springframework.org/schema/jdbc"
   xmlns:jee="http://www.springframework.org/schema/jee"
   xsi:schemaLocation="...">

   <bean id="transferService"
      class="com.bank.service.internal.DefaultTransferService">
      <constructor-arg ref="accountRepository"/>
      <constructor-arg ref="feePolicy"/>
   </bean>

   <bean id="accountRepository"
      class="com.bank.repository.internal.JdbcAccountRepository">
      <constructor-arg ref="dataSource"/>
   </bean>

   <bean id="feePolicy"
      class="com.bank.service.internal.ZeroFeePolicy"/>

   <beans profile="dev">
      <jdbc:embedded-database id="dataSource">
         <jdbc:script
            location="classpath:com/bank/config/sql/schema.sql"/>
         <jdbc:script
            location="classpath:com/bank/config/sql/test-data.sql"/>
      </jdbc:embedded-database>
   </beans>

   <beans profile="production">
      <jee:jndi-lookup id="dataSource"
         jndi-name="java:comp/env/jdbc/datasource"/>
   </beans>

</beans>
```

```
package com.bank.service;

@RunWith(SpringJUnit4ClassRunner.class)
// ApplicationContext will be loaded from "classpath:/app-config.xml"
@ContextConfiguration("/app-config.xml")
```

```
@ActiveProfiles("dev")
public class TransferServiceTest {

  @Autowired
  private TransferService transferService;

  @Test
  public void testTransferService() {
    // test the transferService
  }
}
```

当 TransferServiceTest 运行时,其 ApplicationContext 将加载从 这个 应用配置xml 配置文件在根 在类路径中。 如果你检查应用配置xml 你会注意到 AccountRepository bean有一个 依赖一个 数据源 豆;然而, 数据源 不是定义为一个顶级bean。 相反, 数据源 是定义了两次:一次在吗 生产 剖面和一次 Dev 概要文件。

通过标注 TransferServiceTest 与 @ActiveProfiles(“dev”) 我们指导 春天还是和TestContext框架来加载 ApplicationContext 积极 配置文件设置为 { “dev " } 。 因此,嵌入式 数据库将被创建,和 AccountRepository bean的引用将连接到发展 数据源 。 和这是可能我们 希望在集成测试。

下面的代码清单演示如何实现 相同的配置和集成测试但使用 @ configuration 类而不是 XML。

```
@Configuration
@Profile("dev")
public class StandaloneDataConfig {

   @Bean
   public DataSource dataSource() {
      return new EmbeddedDatabaseBuilder()
         .setType(EmbeddedDatabaseType.HSQL)
         .addScript("classpath:com/bank/config/sql/schema.sql")
         .addScript("classpath:com/bank/config/sql/test-data.sql")
         .build();
   }
}
```

```
@Configuration
@Profile("production")
public class JndiDataConfig {

   @Bean
   public DataSource dataSource() throws Exception {
      Context ctx = new InitialContext();
      return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");
   }
}
```

```
@Configuration
public class TransferServiceConfig {

   @Autowired DataSource dataSource;

   @Bean
   public TransferService transferService() {
      return new DefaultTransferService(accountRepository(),
         feePolicy());
   }

   @Bean
   public AccountRepository accountRepository() {
      return new JdbcAccountRepository(dataSource);
   }

   @Bean
   public FeePolicy feePolicy() {
      return new ZeroFeePolicy();
   }

}
```

```
package com.bank.service;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(
  classes = {
     TransferServiceConfig.class,
     StandaloneDataConfig.class,
     JndiDataConfig.class})
@ActiveProfiles("dev")
public class TransferServiceTest {

  @Autowired
  private TransferService transferService;
```

```
    @Test
    public void testTransferService() {
        // test the transferService
    }
}
```

在这个变化,我们已经把XML配置成 三个独立的 @ configuration 类:

- TransferServiceConfig :获得一个 数据源 通过依赖注入使用 @ autowired
- StandaloneDataConfig :定义了一个 数据源 嵌入式数据库合适 对于开发人员测试
- JndiDataConfig :定义了一个 数据源 从JNDI检索,在 生产环境

与基于xml的配置示例,我们仍然注释 TransferServiceTest 与 @ActiveProfiles(“dev”),但是这一次 我们指定的所有三个配置类通过 @ContextConfiguration 注释。 这个 体的测试类本身仍然是完全不变。

加载WebApplicationContext

Spring 3.2引入了支持加载 WebApplicationContext 在集成 测试。 指导还是和TestContext框架加载 WebApplicationContext 而不是一个 标准 ApplicationContext ,简单地 注释相应的测试类 @WebAppConfiguration 。

存在 @WebAppConfiguration 在您的测试类 指示还是和TestContext框架(TCF) WebApplicationContext (WAC)应该 为你的集成测试加载。 在后台TCF使 确保一个 MockServletContext 是 创建和提供给你的测试的WAC。 默认情况下,基础资源 路径为你 MockServletContext 将 被设置为 “src / main / webapp” 。 这是解释 作为一个路径相对于根你的JVM(即。 ,正常情况下的路径 你的项目)。 如果您熟悉的目录结构 web应用程序在一个Maven项目,您就会知道 “src / main / webapp” 是默认的位置吗 根你的战争。 如果你需要覆盖这个默认,只需提供 另一个路径 @WebAppConfiguration 注释(如 @WebAppConfiguration(" src / webapp /测试”))。 如果你想引用一个基地资源路径从类路径 而不是文件系统,使用Spring的 类路径: 前缀。

请注意弹簧的测试支持 WebApplicationContexts 就等同于 支持标准 ApplicationContexts 。 当测试与 一个 WebApplicationContext 你可以自由 声明不是XML配置文件或 @ configuration 类通过 @ContextConfiguration 。 你的当然也可以使用别的测试等注释 @TestExecutionListeners , @TransactionConfiguration , @ActiveProfiles 等。

下面的例子演示一些不同 加载配置选项 WebApplicationContext 。

ExampleA 11 1一个约定

```
@RunWith(SpringJUnit4ClassRunner.class)

// defaults to "file:src/main/webapp"
@WebAppConfiguration

// detects "WacTests-context.xml" in same package
// or static nested @Configuration class
@ContextConfiguration

public class WacTests {
    //...
}
```

上面的示例演示了还是和TestContext框架的 支持 约定优于配置 。 如果你 注释一个测试类 @WebAppConfiguration 不指定 一个资源基本路径,资源路径将有效的默认 “文件:src / main / webapp” 。 同样,如果你声明 @ContextConfiguration 没有 指定资源 位置 , 注释 类 或上下文 初始化 ,春天将尝试 检测存在你的配置(即,使用约定。 “WacTests-context.xml” 在同一个包 WacTests 类或静态嵌套 @ configuration 类)。

ExampleA 11.2。 一个默认的资源语义

```
@RunWith(SpringJUnit4ClassRunner.class)

// file system resource
@WebAppConfiguration("webapp")

// classpath resource
@ContextConfiguration("/spring/test-servlet-config.xml")

public class WacTests {
    //...
}
```

这个例子演示了如何显式地声明一个资源 基本路径与 @WebAppConfiguration 和 一个XML资源位置与 @ContextConfiguration 。 重要的 注意这里是不同的语义路径与这两个 注释。 默认情况下, @WebAppConfiguration 资源路径 文件系统的基础,然而, @ContextConfiguration 资源 基于位置的类路径。

ExampleA 11.3。 一个显式的资源语义

```
@RunWith(SpringJUnit4ClassRunner.class)

// classpath resource
@WebAppConfiguration("classpath:test-web-resources")

// file system resource
@ContextConfiguration("file:src/main/webapp/WEB-INF/servlet-config.xml")

public class WacTests {
    //...
}
```

在这第三个例子中,我们看到,我们可以覆盖默认的 资源语义注释都通过指定一个春天 资源前缀。 在这个例子的对比评论的 前面的例子。

处理网络模拟

提供全面的web测试支持,Spring 3.2 引入了一个新的 ServletTestExecutionListener 这是 默认启用。 当测试对 WebApplicationContext 这 TestExecutionListener 设置默认线程局部状态通过Spring Web的 RequestContextHolder 在每个测试之前 方法并创建一个 MockHttpServletRequest , MockHttpServletResponse ,和 ServletWebRequest 基于基础配置资源路径通过 @WebAppConfiguration 。 ServletTestExecutionListener 也 确保 MockHttpServletResponse 和 ServletWebRequest 可以被注入 到测试实例,一旦测试完成它清除 线程局部状态。

一旦你有一个 WebApplicationContext 加载为你 测试你可能发现你需要和网络交互戏 例如,设置您的测试夹具或执行断言 在调用您的web组件。 下面的例子 演示,模拟可以在你的测试实例autowired的。 注意, WebApplicationContext 和 MockServletContext 都 缓存整个测试套件;但是,其他模拟管理 每个测试方法的 ServletTestExecutionListener 。

ExampleA 11.4。 一个注入模拟

```
@WebAppConfiguration
@ContextConfiguration
public class WacTests {

    @Autowired WebApplicationContext wac; // cached

    @Autowired MockServletContext servletContext; // cached

    @Autowired MockHttpSession session;

    @Autowired MockHttpServletRequest request;

    @Autowired MockHttpServletResponse response;

    @Autowired ServletWebRequest webRequest;

    //...
}
```

上下文缓存

一旦还是和TestContext框架装载一个 ApplicationContext (或 WebApplicationContext)为测试, 上下文将被缓存和重用 **所有** 随后声明相同的测试 独特的上下文配置在相同的测试套件。 了解 如何缓存作品,重要的是要理解是什么意思吗 独特的 和 测试 套件 。

一个 ApplicationContext 可以 独特 确定的组合 配置参数,用于加载它。 因此, 结合独特的配置参数是用来生成一个 关键 在这上下文缓存。 这个 还是和TestContext框架使用下面的配置参数 构建上下文缓存键:

- 位置 (从 @ContextConfiguration)
- 类 (从 @ContextConfiguration)

- contextInitializerClasses (从 @ContextConfiguration)
- ContextLoader (从 @ContextConfiguration)
- activeProfiles (从 @ActiveProfiles)
- resourceBasePath (从 @WebAppConfiguration)

例如,如果 TestClassA 指定 { "应用程序配置。 xml” 、 “测试配置xml " } 为 位置 (或 价值)属性 的 @ContextConfiguration , 还是和TestContext框架将载荷对应的 ApplicationContext 并将其存储在一个 静态 上下文缓存的一个关键,是基于下 只在那些地点。 所以如果 TestClassB 还定义了 { "应用程序配置。 xml” 、 “测试配置xml " } 对于它的位置(或隐或显通过 但没有定义继承) @WebAppConfiguration 、不同的 ContextLoader ,不同的活动 配置文件,或不同的上下文的初始化,然后同样的 ApplicationContext 将共享 这两个测试类。 这意味着设置成本加载一个 应用程序上下文发生只有一次(每个测试套件), 随后的测试执行速度更快。

测试套件和分叉的过程

春天还是和TestContext商店应用程序上下文框架 在一个 静态 缓存。 这意味着上下文 简直是存储在一个吗 静态 变量。 在 句话说,如果测试执行在不同的流程静态缓存 将被清除每个测试执行之间,这将吗 有效的禁用缓存机制。

受益于缓存机制,所有的测试必须运行 在相同的过程或测试套件。 这可以通过 执行所有测试作为一个群体在一个IDE。 同样的,当 执行测试和构建框架(如Ant、Maven或Gradle 它是重要的,以确保构建框架不 叉在测试的。 例如,如果 forkMode 对于Maven插件的设置 总是 或 pertest ,还是和TestContext框架不会能够缓存应用程序上下文测试类和之间 构建过程将运行很慢的结果。

在不太可能的情况下,一个测试应用程序导致腐败 上下文和需要重新加载一个例如,通过修改一个bean 定义或一个应用程序对象的状态一个你可以标注 您的测试类或测试方法 @DirtiesContext (见讨论 @DirtiesContext 在 一个章节弹簧测试Annotationsa)。 这指示 春天从缓存中移除上下文和重建 应用程序上下文在执行下一个测试。 注意,支持 为 @DirtiesContext 注释是 提供的 DirtiesContextTestExecutionListener 这是 默认启用。

上下文层次

当编写集成测试,依靠一个加载弹簧 ApplicationContext ,它往往是 足以测试单个上下文;然而,有次 当它是有利的,甚至是必要的测试对比的一个层次 ApplicationContext 年代。例如,如果 您正在开发一个Spring MVC web应用程序通常会 有一根 WebApplicationContext 通过弹簧加载的 ContextLoaderListener 和一个 孩子 WebApplicationContext 加载通过 春天的 DispatcherServlet 。 这个结果在一个 父子层次结构,共享组件和上下文 基础设施配置中声明根上下文和 在孩子的消费环境网络自身组件。 另一个使用 案例可以发现在春天批处理应用程序,你经常有 父上下文,提供配置为共享的批处理 基础设施和一个孩子上下文特定的配置 批处理作业。

Spring框架的第3.2.2章,可以写 集成测试,使用上下文层次通过声明上下文 配置通过 @ContextHierarchy 注释,要么在一个单独的测试类或在一个测试类 层次结构。 如果一个上下文层次结构是宣布在多个类 在一个测试类层次结构也可以合并或覆盖 上下文配置为一个特定的,名叫水平在上下文 层次结构。 当合并配置对于一个给定的水平 层次结构配置资源类型(即。 、XML配置 文件或注释类)必须是一致的;否则,它是 完全可以接受有不同的水平在一个上下文层次结构 配置使用不同的资源类型。

以下示例展示常见基础junit 配置场景集成测试,需要使用 上下文层次结构。

ExampleA 11.5。 一个测试类与上下文层次结构

ControllerIntegrationTests 代表一个 典型的集成测试场景一个Spring MVC web 应用程序通过声明一个上下文层次结构包括两个 的水平,一个用于 根 WebApplicationContext (加载使用 TestAppConfig @ configuration 类)和一个用于 dispatcher servlet WebApplicationContext (加载使用 这个 WebConfig @ configuration 类)。 这个 WebApplicationContext 这是 Autowired的 到测试实例是一个用于 孩子上下文(即。 ,最低的上下文 层次结构)。

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextHierarchy({
    @ContextConfiguration(classes = TestAppConfig.class),
    @ContextConfiguration(classes = WebConfig.class)
})
public class ControllerIntegrationTests {

    @Autowired
    private WebApplicationContext wac;

    // ...
}
```

ExampleA 11.6。 一个类层次结构和隐式父上下文

以下测试类定义一个上下文层次结构在一个 测试类的层次结构。 AbstractWebTests 声明了配置一个根 WebApplicationContext 在一个 弹簧驱动的web应用程序。 但是请注意, AbstractWebTests 没有声明 @ContextHierarchy ,因此, 子类的 AbstractWebTests 可以选择 参与一个上下文层次或简单地遵循标准的 语义 @ContextConfiguration 。 SoapWebServiceTests 和 RestWebServiceTests 两个扩展 AbstractWebTests 和定义一个上下文 层次通过 @ContextHierarchy 。 这个 结果是,三个应用程序上下文(一个用于将被装载 每个声明的 @ContextConfiguration), 应用程序上下文加载基于配置 AbstractWebTests 将被设置为父吗 上下文的每个上下文加载的混凝土 子类。

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration("file:src/main/webapp/WEB-INF/applicationContext.xml")
public abstract class AbstractWebTests {}

@ContextHierarchy(@ContextConfiguration("/spring/soap-ws-config.xml")
public class SoapWebServiceTests extends AbstractWebTests {}

@ContextHierarchy(@ContextConfiguration("/spring/rest-ws-config.xml")
public class RestWebServiceTests extends AbstractWebTests {}
```

ExampleA 11.7。 一个类层次结构和上下文层次结构合并 配置

下面的类演示使用 命名 层次的水平,以 合并 配置为特定的水平 一个上下文层次结构。 BaseTests 定义了两个 水平的层次结构, 父 和 孩子 。 ExtendedTests 延伸 BaseTests 和指示春天 还是和TestContext框架来合并上下文配置 孩子 层次水平,只需确保 声明的名称通过 ContextConfiguration ' s 名称 属性都是 “孩子” 。 结果是三个应用程序 上下文将被加载:一个用于 “/应用程序配置xml” ,一个用于 “/用户配置xml” ,和一个用于 { " /用户配置。 xml” 、 “/订单配置xml " } 。 作为 与前面的示例中,应用程序上下文加载 “/应用程序配置xml” 将被设置为父吗 上下文语境的加载 “/用户配置xml” 和 { " /用户配置。 xml” 、 “/订单配置xml " } 。

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextHierarchy({
    @ContextConfiguration(name = "parent", locations = "/app-config.xml"),
    @ContextConfiguration(name = "child",  locations = "/user-config.xml")
})
public class BaseTests {}

@ContextHierarchy(
    @ContextConfiguration(name = "child",  locations = "/order-config.xml")
)
public class ExtendedTests extends BaseTests {}
```

ExampleA 11.8。 一个类层次结构和覆盖上下文层次结构 配置

与前面的示例,这个示例演示了 如何 覆盖 配置对于一个给定的 命名上下文层次水平通过设置 ContextConfiguration ' s inheritLocations 旗帜 假 。 因此,应用程序上下文 ExtendedTests 将只加载来自哪里 “/测试用户配置xml” 并将其父母 设置到上下文加载 “/应用程序配置xml” 。

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextHierarchy({
    @ContextConfiguration(name = "parent", locations = "/app-config.xml"),
    @ContextConfiguration(name = "child",  locations = "/user-config.xml")
})
public class BaseTests {}

@ContextHierarchy(
    @ContextConfiguration(
        name = "child",
        locations = "/test-user-config.xml",
        inheritLocations = false
))
public class ExtendedTests extends BaseTests {}
```

在一个上下文弄脏一个上下文层次结构

如果 @DirtiesContext 用于 测试其上下文配置作为一个上下文层次结构, hierarchyMode 标记可以用来控

制如何 上下文缓存清除。 详情咨询洽谈 的 @DirtiesContext 在 一个章节弹簧测试Annotationsa 和 Javadoc 对于 @DirtiesContext 。

依赖注入的测试夹具

当你使用 DependencyInjectionTestExecutionListener 一个这 配置默认情况下一个的依赖性测试实例 注入 从豆在应用程序上下文 你配置了 @ContextConfiguration 。 你可以使用setter 注入,字段注入,或者两者兼有,根据注释你 选择和你是否放在setter方法或字段。 对于 一致性与注释支持Spring 2.5中引入的, 3.0,您可以使用Spring的 @ autowired 注释或 @ inject 注释从 JSR 300。

> **提示**
>
> 还是和TestContext框架没有仪器的方式 哪一个测试实例。 因此使用 @ autowired 或 @ inject 对于构造函数没有任何影响 为测试类。

因为 @ autowired 用于 执行 自动装配的 类型 ,如果你有多个bean的定义 相同的类型,你不能依靠这种方法对于那些特别的豆子。 在这种情况下,您可以使用 @ autowired 在 结合 qualifier 。 像春天的 3.0你也可以选择使用 @ inject 在 结合 @ named 。 另外, 如果您的测试类可以访问它的 ApplicationContext ,您可以执行一个显式的 查找使用(例如)调用 applicationcontext getbean(“titleRepository”) 。

如果你不希望依赖注入应用到您的测试 实例,只是没有注释字段或setter方法 @ autowired 或 @ inject 。 或者,您可以禁用依赖注入通过显式配置类完全 与 @TestExecutionListeners 和省略 DependencyInjectionTestExecutionListener.class 从 听众的名单。

考虑场景的测试 HibernateTitleRepository 类,如中概述 这个 目标 部分。 这个 接下来的两个代码清单演示使用 @ autowired 在田野和setter方法。 提出了应用程序上下文配置毕竟示例代码 清单。

> **注意**
>
> 依赖注入的行为在以下代码清单 不是特定于JUnit。 同样的DI技术可用于 结合任何测试框架。
>
> 下面的例子使调用静态断言方法 如 assertNotNull() 但是没有追加到 电话 断言 。 在这种情况下,假设 方法是通过一个正确导入 进口 静态 声明这不是示例中所示。

第一个代码清单显示了一个基于junit的实现 测试类,使用 @ autowired 实地 注射。

```
@RunWith(SpringJUnit4ClassRunner.class)
// specifies the Spring configuration to load for this test fixture
@ContextConfiguration("repository-config.xml")
public class HibernateTitleRepositoryTests {

    // this instance will be dependency injected by type
    @Autowired
    private HibernateTitleRepository titleRepository;

    @Test
    public void findById() {
        Title title = titleRepository.findById(new Long(10));
        assertNotNull(title);
    }
}
```

或者,您可以配置类来使用 @ autowired 看到的setter注入 下面。

```
@RunWith(SpringJUnit4ClassRunner.class)
// specifies the Spring configuration to load for this test fixture
@ContextConfiguration("repository-config.xml")
public class HibernateTitleRepositoryTests {

    // this instance will be dependency injected by type
    private HibernateTitleRepository titleRepository;

    @Autowired
    public void setTitleRepository(HibernateTitleRepository titleRepository) {
        this.titleRepository = titleRepository;
    }

    @Test
    public void findById() {
        Title title = titleRepository.findById(new Long(10));
        assertNotNull(title);
```

```
    }
}
```

前面的代码清单中使用相同的XML上下文文件 引用的 @ContextConfiguration 注释(即 存储库配置xml),这 看起来像这样:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

   <!-- this bean will be injected into the HibernateTitleRepositoryTests class -->
   <bean id="titleRepository" class="com.foo.repository.hibernate.HibernateTitleRepository">
      <property name="sessionFactory" ref="sessionFactory"/>
   </bean>

   <bean id="sessionFactory"
         class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
      <!-- configuration elided for brevity -->
   </bean>

</beans>
```

注意

如果你从一个弹簧提供扩展测试基类, 碰巧使用 @ autowired 在它的一个 setter方法,你可能会有多个 bean类型的影响 在你的应用程序上下文定义:例如,多个 数据源 豆子。 在这种情况下,你 可以覆盖setter方法和使用 qualifier 注释来表示 具体目标bean如下,但确保委托 覆盖超类中的方法一样。

```
// ...

    @Autowired
    @Override
    public void setDataSource(@Qualifier("myDataSource") DataSource dataSource) {
        super.setDataSource(dataSource);
    }

// ...
```

指定限定符值表示特定的 数据源 bean注入,缩小 这个组类型匹配到特定的bean。 它的值匹配 反对 <限定符> 中的声明 相应 < bean > 定义。 bean 的名字是用作后备限定符的值,所以你可能会有效 也指向一个特定的bean的名称(如上图所示,假设 “一” 是bean id)。

测试请求和会话作用域的豆子

请求和会话 范围bean 已经由弹簧几年后的现在, 但它总是有点不平凡的去测试它们。 在Spring 3.2 现在是一个微风来测试你会和会话范围内的豆子 通过以下步骤。

- 确保 WebApplicationContext 就会加载 你的测试通过标注您的测试类 @WebAppConfiguration 。
- 注入模拟请求或会话到您的测试实例和 准备你的测试夹具是适当的。
- 调用您的web组件,你获取 配置 WebApplicationContext (即。 通过依赖注入)。
- 针对模拟执行断言。

下面的代码片段显示了XML配置 登录用例。 注意, userService bean有一个 依赖一个会 loginAction bean。 同时, loginAction 被实例化使用 ?表情 那检索用户名 和密码从当前HTTP请求。 在我们的测试,我们会想 配置这些请求参数通过模拟管理 还是和TestContext框架。

ExampleA 11.9。 一个会bean配置

```
<beans>

  <bean id="userService"
     class="com.example.SimpleUserService"
     c:loginAction-ref="loginAction" />

  <bean id="loginAction" class="com.example.LoginAction"
     c:username="#{request.getParameter('user')}"
     c:password="#{request.getParameter('pswd')}"
     scope="request">
    <aop:scoped-proxy />
  </bean>

</beans>
```

在 RequestScopedBeanTests 我们将两 这个 userService (即。 ,这个主题下的测试) 和 MockHttpServletRequest 到我们的测试 实例。 在我们 requestScope() 测试方法我们 建立我们的测试夹具通过设置请求参数提供 MockHttpServletRequest 。 当 loginUser() 在我们的方法被调用 userService 我们保证用户服务 访问会 loginAction 为 电流 MockHttpServletRequest (即。 ,我们 只是设置参数)。 然后我们可以执行断言反对 结果基于已知输入的用户名和密码。

ExampleA 11.10。 一个会豆测试

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
@WebAppConfiguration
public class RequestScopedBeanTests {

    @Autowired UserService userService;
    @Autowired MockHttpServletRequest request;

    @Test
    public void requestScope() {

        request.setParameter("user", "enigma");
        request.setParameter("pswd", "$pr!ng");

        LoginResults results = userService.loginUser();

        // assert results
    }
}
```

下面的代码片段是一个类似我们在上面看到的 一个会豆;然而,这一次的 userService bean依赖于一个会话范围内 userPreferences bean。 注意, userPreferences 实例化bean使用? 表达式,检索 主题 从 当前的HTTP会话。 在我们的测试, 我们将需要配置一个主题 模拟会话管理的还是和TestContext框架。

ExampleA 11.11。 一个会话范围内的bean配置

```
<beans>

  <bean id="userService"
     class="com.example.SimpleUserService"
     c:userPreferences-ref="userPreferences" />

  <bean id="userPreferences"
     class="com.example.UserPreferences"
     c:theme="#{session.getAttribute('theme')}"
     scope="session">
    <aop:scoped-proxy />
  </bean>

</beans>
```

在 SessionScopedBeanTests 我们把 userService 和 MockHttpSession 到我们的测试实例。 在 我们 sessionScope() 测试方法我们建立我们的测试 夹具通过设定预期的 “主题” 属性提供的 MockHttpSession 。 当 processUserPreferences() 在我们的方法被调用 userService 我们保证用户服务 访问会话范围内 userPreferences 为 电流 MockHttpSession ,我们可以执行 断言对结果基于配置的主题。

ExampleA 11.12。 一个会话范围内的bean测试

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
@WebAppConfiguration
public class SessionScopedBeanTests {

  @Autowired UserService userService;
  @Autowired MockHttpSession session;

  @Test
  public void sessionScope() throws Exception {

    session.setAttribute("theme", "blue");

    Results results = userService.processUserPreferences();

    // assert results
  }
}
```

事务管理

在还是和TestContext框架,事务管理 TransactionalTestExecutionListener 。 注意, TransactionalTestExecutionListener 配置 默认情况下,即使你不显式地声明 @TestExecutionListeners 在你的测试 类。 使支持事务,然而,您必须提供一个 PlatformTransactionManager bean的 应用程序上下文加载 @ContextConfiguration 语义。 在 另外,你必须声明 transactional 要么在类或方法为你的测试水平。

类级别的事务配置(即。 ,设定一个 显式事务管理器bean的名称和默认的回滚 国旗),请参阅 @TransactionConfiguration 条目 注释 支持 部分。

如果事务是不支持整个测试类,你可以 注释方法明确与 transactional 。 控制是否 事务应该坚持一个特定的测试方法,您可以使用 @Rollback 注释覆盖 类级别默认回滚设置。

AbstractTransactionalJUnit4SpringContextTests 和 AbstractTransactionalTestNGSpringContextTests 是预先配置的事务支持类 水平。

有时候你需要执行某些代码之前或之后 事务性测试方法但以外的事务上下文,因为 的例子,来验证初始数据库执行之前的状态 测试或验证预期的事务提交行为试验后 执行(如果测试配置不回滚事务)。 TransactionalTestExecutionListener 支持 @BeforeTransaction 和 @AfterTransaction 注释,正是 这样的场景。 只是注释任何 公共无效 方法在您的测试类,其中的一个注释, TransactionalTestExecutionListener 确保 你 事务方法之前 或 在 事务方法 在适当的执行 时间。

提示

任何 方法之前 (如方法 标注JUnit的 @ before一样)和任何 在方法 (如的方法 JUnit的 @After)执行 **在** 一个事务。 此外,方法 标注 @BeforeTransaction 或 @AfterTransaction 天生不 执行测试标注 @NotTransactional 。 然而, @NotTransactional 是废弃的 Spring 3.0。

以下示例展示了一个基于junit虚构 集成测试场景突出几个事务相关 注释。 咨询 注释支持 为进一步的信息和配置节的例子。

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
@TransactionConfiguration(transactionManager="txMgr", defaultRollback=false)
@Transactional
public class FictitiousTransactionalTest {

    @BeforeTransaction
    public void verifyInitialDatabaseState() {
        // logic to verify the initial state before a transaction is started
    }

    @Before
    public void setUpTestDataWithinTransaction() {
        // set up test data within the transaction
    }

    @Test
    // overrides the class-level defaultRollback setting
    @Rollback(true)
    public void modifyDatabaseWithinTransaction() {
        // logic which uses the test data and modifies database state
    }

    @After
    public void tearDownWithinTransaction() {
        // execute "tear down" logic within the transaction
    }

    @AfterTransaction
    public void verifyFinalDatabaseState() {
        // logic to verify the final state after transaction has rolled back
    }

}
```

避免假阳性当测试ORM代码

当你测试应用程序代码,操作的状态 Hibernate会话,确保 冲洗 这个 在测试方法的基础会话执行该代码。 失败 冲洗底层会话可以生产 假 阳性 :您的测试可以通过,但相同的代码抛出 例外在现场、生产环境。 在接下来的 hibernate的基础例子测试用例,一个方法演示了一个假 积极的,和其他方法正确结果的公开 冲洗会话。 注意,这适用于JPA和任何其他ORM 框架,维护一个内存 单位 工作 。

```
// ...

@Autowired
private SessionFactory sessionFactory;

@Test // no expected exception!
public void falsePositive() {
    updateEntityInHibernateSession();
    // False positive: an exception will be thrown once the session is
    // finally flushed (i.e., in production code)
}

@Test(expected = GenericJDBCException.class)
public void updateWithSessionFlush() {
    updateEntityInHibernateSession();
    // Manual flush is required to avoid false positive in test
    sessionFactory.getCurrentSession().flush();
}

// ...
```

还是和TestContext框架支持类

JUnit支持类

这个 org.springframework.test.context.junit4 包提供了支持类基于JUnit 4.5 +测试 情况下。

- AbstractJUnit4SpringContextTests : 抽象基类,测试集 春天 还是和TestContext框架 使用显式的 ApplicationContext 测试支持在一个 JUnit 4.5 +环境。
  当你扩展 AbstractJUnit4SpringContextTests ,你可以 访问以下 保护 实例 变量:
  - ApplicationContext :使用这个变量 执行明确的bean查找或测试状态的 环境作为一个整体。
- AbstractTransactionalJUnit4SpringContextTests : 文摘 事务性 扩展的 AbstractJUnit4SpringContextTests 这也 增加了一些方便的功能用于JDBC访问。 预计 javax.sql.DataSource bean和 PlatformTransactionManager bean 被定义在 ApplicationContext 。 当 你延长 AbstractTransactionalJUnit4SpringContextTests 你可以访问以下 保护 实例 变量:
  - ApplicationContext :继承 这个 AbstractJUnit4SpringContextTests 超类。 使用这个变量执行显式的bean查找 或测试上下文的状态作为一个整体。
  - JdbcTemplate :使用该变量 执行SQL语句来查询数据库。 这样的查询可以 被用来确认数据库状态两个 之前 到 和 在 执行 数据库相关的应用程序代码,和弹簧确保 这样的查询运行在同一事务的范围的 应用程序代码。 当结合使用一个 ORM工具, 一定要避免 假 阳性 。

> **提示**
>
> 这些类是一个方便的扩展。 如果你不 想要您的测试类绑定到一个spring特定类 一个层次举个例子,如果你 想直接扩展类 你是测试一个您可以配置自己的自定义测试类的 使用 @RunWith(SpringJUnit4ClassRunner.class) , @ContextConfiguration , @TestExecutionListeners ,所以 在。

弹簧JUnit跑

这个 春天还是和TestContext框架 提供 全面整合与JUnit 4.5 +通过一个自定义的跑步者(测试 一个JUnit 4.5 4.10)。 通过标注测试类 @RunWith(SpringJUnit4ClassRunner.class) 、开发人员 可以实现标准基于junit单元测试和集成测试和 同时获得益处还是和TestContext框架如 支持加载应用程序上下文,依赖注入的测试 情况下,事务测试方法执行,等等。 这个 下面的代码清单显示的最小要求 配置一个测试类来运行自定义弹簧跑步者。 @TestExecutionListeners 配置 用一个空的列表以禁用默认的听众,这 否则将需要一个ApplicationContext配置通过 @ContextConfiguration 。

```
@RunWith(SpringJUnit4ClassRunner.class)
@TestExecutionListeners({})
public class SimpleTest {

    @Test
    public void testMethod() {
        // execute test logic...
    }
}
```

TestNG支持类

这个 org.springframework.test.context.testng 包提供了支持基于测试用例的类TestNG。

- AbstractTestNGSpringContextTests : 抽象基类,测试集 春天 还是和TestContext框架 使用显式的 ApplicationContext 测试支持在一个 TestNG的环境。
  当你扩展 AbstractTestNGSpringContextTests ,你可以 访问以下 保护 实例 变量:
  - ApplicationContext :使用这个变量 执行明确的bean查找或测试状态的 环境作为一个整体。
- AbstractTransactionalTestNGSpringContextTests : 文摘 事务性 扩展的 AbstractTestNGSpringContextTests 添加 一些方便的功能为JDBC访问。 预计 javax.sql.DataSource bean和 PlatformTransactionManager bean 被定义在 ApplicationContext 。 当 你延长 AbstractTransactionalTestNGSpringContextTests , 你可以访问以下 保护 实例 变量:
  - ApplicationContext :继承 这个 AbstractTestNGSpringContextTests 超类。 使用这个变量执行显式的bean查找 或测试上下文的状态作为一个整体。
  - JdbcTemplate :使用该变量 执行SQL语句来查询数据库。 这样的查询可以 被用来确认数据库状态两个 之前 到 和 在 执行 数据库相关的应用程序代码,和弹簧确保 这样的查询运行在同一事务的范围的 应用程序代码。 当结合使用一个 ORM工具, 一定要避免 假 阳性 。

提示

这些类是一个方便的扩展。 如果你不 想要您的测试类绑定到一个spring特定类 一个层次举个例子,如果你 想直接扩展类 你是测试一个您可以配置自己的自定义测试类的 使用 @ContextConfiguration , @TestExecutionListeners 等等, 和手动插装您的测试类 TestContextManager 。 看到的源代码 AbstractTestNGSpringContextTests 对于一个 的例子来测试您的测试类。

11.3.6A Spring MVC的测试框架

这个 Spring MVC的测试框架 提供第一 类JUnit测试支持客户端和服务器端Spring MVC代码 通过一个流利的API。 通常它加载实际的Spring配置 通过 还是和TestContext框架 和总是使用 DispatcherServlet 处理请求因此 近似完全集成测试不需要运行Servlet 集装箱。

客户端测试 RestTemplate 的和 允许测试的代码的依赖 RestTemplate 不需要一个运行的服务器 回应请求。

独立项目

Spring框架包含在3.2之前,Spring MVC测试 框架已经存在作为一个单独的项目在GitHub上它增长和演化的实际使用,反馈,和贡献的 许多。

独立 弹簧测试mvc 项目 仍在GitHub,可用于 结合Spring框架里。 应用升级到3.2 应该取代 弹簧测试mvc 依赖与 一个依赖 弹簧测试 。

这个 弹簧测试 模块使用一个不同的 包 org.springframework.test.web 但 否则是几乎相同的只有两个例外。 一个是支持 在3.2新特性(如异步web请求)。 其他相关 的选项 来创建一个 MockMvc 实例。 在Spring框架3.2,这只能通过还是和TestContext 框架,它提供缓存加载的好处 配置。

服务器端测试

Spring Framework 3.2之前,最可能的方法来测试一个春天 MVC控制器是编写一个单元测试,实例化 控制器,它与模拟或存根注入依赖项,然后调用 其方法,直接使用 MockHttpServletRequest 和 MockHttpServletResponse 在必要时。

虽然这是很容易做到的,控制器有很多 注释,大部分还有待验证。 请求映射、数据绑定, 类型转换和验证只是几个例子的什么不是 测试了。 此外,还有其他类型的注释等方法 @InitBinder , @ModelAttribute ,和 @ExceptionHandler 得到调用, 请求处理的一部分。

Spring MVC测试背后的想法是能够重写这些 控制器测试通过执行实际的请求和生成反应, 因为他们将在运行时,一路上调用控制器通过 Spring MVC DispatcherServlet 。 控制器可以 还是注射模拟依赖,所以测试可以保持关注 web层。

Spring MVC测试建立在熟悉的“模拟”的实现 Servlet API中可用 弹簧测试 模块。 这允许执行请求和生成反应没有 需要运行在一个Servlet容器。 大部分 一切都应该工作在运行时是否除了JSP 渲染,不提供一个Servlet容器之外。 此外,如果您熟悉如何 MockHttpServletResponse 的作品,你就会知道 转发和重定向不实际执行。 相反“转发” 和“重定向”url保存,可以断言在测试中。 这意味着 如果您使用的是JSP,您可以验证JSP页面请求 被转发。

其他一切呈现包括 @ResponseBody 方法和 视图 类型(除了jsp等) Freemarker,速度,和其他Thymeleaf渲染HTML、JSON、XML等应按预期工作,并且响应将包含 生成的内容。

下面是一个例子,一个测试请求帐户信息在 JSON格式:

```
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration("test-servlet-context.xml")
```

```
public class ExampleTests {

    @Autowired
    private WebApplicationContext wac;

    private MockMvc mockMvc;

    @Before
    public void setup() {
        this.mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).build();
    }

    @Test
    public void getAccount() throws Exception {
        this.mockMvc.perform(get("/accounts/1").accept("application/json;charset=UTF-8"))
          .andExpect(status().isOk())
          .andExpect(content().contentType("application/json"))
          .andExpect(jsonPath("$.name").value("Lee");
    }

}
```

测试依赖 WebApplicationContext 支持 还是和TestContext框架 。 它加载弹簧 配置从一个XML配置文件位于相同的包 为测试类(也支持JavaConfig)和注入创建的 WebApplicationContext 到测试所以 MockMvc 可以用它创建的实例。

这个 MockMvc 然后被用来执行 请求 “/账户/ 1” 和验证结果 响应状态是200,响应内容类型 “application / json” 和响应内容有一个JSON 财产称为“名称”与“李”的价值。 JSON内容是检查 借助Jayway的 JsonPath项目 。 还有很多其他的选择结果的验证 执行请求和那些将在稍后讨论。

静态导入

的流利的API在上面的例子中需要几个静态 进口,如 MockMvcRequestBuilders。* , MockMvcResultMatchers。* ,和 MockMvcBuilders。* 。 一个简单的方法来找到这些 类是搜索类型匹配 “MockMvc *” 。 如果使用Eclipse,一定要添加它们 为“最喜欢的静态成员“在Eclipse首选项下 Java - >编辑- > - >内容帮助 最爱 。 这将允许使用内容帮助之后 输入的第一个字符的静态方法的名字。 其他ide(如。 IntelliJ)可能不需要任何额外的配置。 只是检查 支持代码完成静态成员。

设置选项

服务器端测试设置的目标是创建一个实例 MockMvc 可用于执行请求。 有两个主要选项。

第一个选项是指通过Spring MVC的配置 这个 还是和TestContext框架 ,加载弹簧 配置和注入 WebApplicationContext 进入测试 用来创建一个 MockMvc :

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration("my-servlet-context.xml")
public class MyWebTests {

    @Autowired
    private WebApplicationContext wac;

    private MockMvc mockMvc;

    @Before
    public void setup() {
        this.mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).build();
    }

    // ...

}
```

第二个选择是简单的注册一个控制器实例 没有加载任何Spring配置。 而不是Spring MVC的基本 配置适合测试注释的控制器自动创建的。 创建的配置相比 MVC JavaConfig(和MVC的名称空间),可以定制 一个学位,通过构建式方法:

```
public class MyWebTests {

    private MockMvc mockMvc;

    @Before
    public void setup() {
        this.mockMvc = MockMvcBuilders.standaloneSetup(new AccountController()).build();
    }

    // ...

}
```

你应该使用哪个选项?

这个 “webAppContextSetup” 加载实际 Spring MVC配置导致一个更完整的集成 测试。 自 还是和TestContext框架 缓存加载Spring配置,它有助于保持测试运行快甚至 随着越来越多的测试得到补充。 此外,你可以注入模拟服务 到控制器通过Spring配置,为了保持 专注于测试web层。 下面是一个示例声明 mock服务5:

```
<bean id="accountService" class="org.mockito.Mockito" factory-method="mock">
  <constructor-arg value="org.example.AccountService"/>
</bean>
```

然后你可以注入mock服务到测试顺序设置 和验证的期望:

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration("test-servlet-context.xml")
public class AccountTests {

    @Autowired
    private WebApplicationContext wac;

    private MockMvc mockMvc;

    @Autowired
    private AccountService accountService;

    // ...

}
```

这个 “standaloneSetup” 另一方面是 有点接近一个单元测试。 它测试一个控制器在一个时间, 控制器可以注射模拟依赖手工,和它 不需要加载Spring配置。 这样的测试更 集中在风格和使它更容易看到哪个控制器 测试,是否有具体的Spring MVC配置需要 工作,等等。 “standaloneSetup” 也是一个非常方便的方式 写特别的测试,以验证一些行为或调试 问题。

就像集成与单元测试,没有正确的或 错误的答案。 使用 “standaloneSetup” 确实意味着需要一些 额外的 “webAppContextSetup” 测试,验证了Spring MVC 配置。 或者,你可以决定写所有测试用 “webAppContextSetup” 和总是测试与实际的Spring MVC 配置。

执行请求

执行请求,使用适当的HTTP方法和 额外的构建式方法对应的属性 MockHttpServletRequest 。 例如:

```
mockMvc.perform(post("/hotels/{id}", 42).accept(MediaType.APPLICATION_JSON));
```

除了所有的HTTP方法,您还可以执行文件 上传请求,它在内部创建一个实例 MockMultipartHttpServletRequest :

```
mockMvc.perform(fileUpload("/doc").file("a1", "ABC".getBytes("UTF-8")));
```

查询字符串参数可以指定的URI 模板:

```
mockMvc.perform(get("/hotels?foo={foo}", "bar"));
```

或通过添加Servlet请求参数:

```
mockMvc.perform(get("/hotels").param("foo", "bar"));
```

如果应用程序代码依赖于Servlet请求参数, 不检查查询字符串,通常是这样,那么它 不管你多么添加参数。 请记住, 参数中提供的URI模板将解码而 参数提供的 参数(...) 方法 预计将解码。

在大多数情况下它比保留上下文路径和 Servlet路径从请求URI。 如果你必须测试和完整的 请求URI,一定要设置 contextPath 和 servletPath 因此,请求映射 将工作:

```
mockMvc.perform(get("/app/main/hotels/{id}").contextPath("/app").servletPath("/main"))
```

看着上面的例子中,它将繁琐的设置 servletPath contextPath和与每个执行的请求。 这就是为什么 你可以定义默认请求属性当构建 MockMvc :

```
public class MyWebTests {

    private MockMvc mockMvc;
```

```
    @Before
    public void setup() {
        mockMvc = standaloneSetup(new AccountController())
            .defaultRequest(get("/")
                .contextPath("/app").servletPath("/main")
                .accept(MediaType.APPLICATION_JSON).build();
    }

}
```

上面的属性将应用于每个请求执行 通过 MockMvc 。 如果相同的属性 还指定在一个给定的请求,它将覆盖默认值。 这就是为什么,HTTP方法和URI并不重要,当设置 默认请求属性,因为他们必须被指定在每个 请求。

定义预期

期望可以定义通过附加一个或更多 .andExpect(. .) 在调用来执行 要求:

```
mockMvc.perform(get("/accounts/1")).andExpect(status().isOk());
```

MockMvcResultMatchers。* 定义了大量的 静态成员,其中一些与其他方法的返回类型, 因为坚持执行的结果要求。 秋天的断言 在两大类。

第一类断言验证的属性 反应,我。 e响应状态,标题和内容。 这些都是 最重要的事情来测试。

第二类断言超越响应,和 允许检查Spring MVC特定构造如这 处理请求的控制器方法,无论是一个例外是 提出和处理,模型的内容是,什么观点 选择,flash也加入进来,属性等等。 它也 可以验证特定的构造(如Servlet请求和 会话属性。 下面的测试断言绑定/验证 失败:

```
mockMvc.perform(post("/persons"))
  .andExpect(status().isOk())
  .andExpect(model().attributeHasErrors("person"));
```

很多时候,当写作测试,它是有用的结果转储 执行的请求。 这个可以做如下,在那里 print() 是一个静态导入的 MockMvcResultHandlers :

```
mockMvc.perform(post("/persons"))
  .andDo(print())
  .andExpect(status().isOk())
  .andExpect(model().attributeHasErrors("person"));
```

只要请求处理导致一个未处理的异常, print() 方法将打印所有可用的 结果数据 system . out 。

在某些情况下,您可能想要直接访问结果 和验证一些,否则不能验证。 这可以 通过附加 .andReturn() 最后在 所有的期望:

```
MvcResult mvcResult = mockMvc.perform(post("/persons")).andExpect(status().isOk()).andReturn();
// ...
```

当所有的测试重复相同的期望,您可以定义 常见的预期一旦当构建 MockMvc :

```
standaloneSetup(new SimpleController())
    .alwaysExpect(status().isOk())
    .alwaysExpect(content().contentType("application/json;charset=UTF-8"))
    .build()
```

注意,期望 总是 应用 和不能被覆盖,不创建一个单独的 MockMvc 实例。

当JSON响应内容包含超媒体链接创建 与 春天 HATEOAS ,生成的链接可以验证:

```
mockMvc.perform(get("/people").accept(MediaType.APPLICATION_JSON))
  .andExpect(jsonPath("$.links[?(@.rel == 'self')].href").value("http://localhost:8080/people"));
```

当XML响应内容包含超媒体链接创建 春天 HATEOAS ,生成的链接可以验证:

```
Map<String, String> ns = Collections.singletonMap("ns", "http://www.w3.org/2005/Atom");
mockMvc.perform(get("/handle").accept(MediaType.APPLICATION_XML))
  .andExpect(xpath("/person/ns:link[@rel='self']/@href", ns).string("http://localhost:8080/people"));
```

滤波器注册

当建立一个 MockMvc ,你可以 注册一个或多个 滤波器 实例:

```
mockMvc = standaloneSetup(new PersonController()).addFilters(new CharacterEncodingFilter()).build();
```

过滤器将调用注册通过 MockFilterChain 从 弹簧测试 和过去的过滤器将代表 这个 DispatcherServlet 。

进一步的服务器端测试例子

该框架的测试包括 许多 样品测试 为了演示如何使用Spring MVC 测试。 浏览这些例子进一步的想法。 也 spring mvc展示 已经全部测试覆盖基于Spring MVC测试。

端REST测试

客户端测试代码使用 RestTemplate 。 预期的目标是定义 请求和提供“存根”反应:

```
RestTemplate restTemplate = new RestTemplate();

MockRestServiceServer mockServer =  MockRestServiceServer.createServer(restTemplate);
mockServer.expect(requestTo("/greeting")).andRespond(withSuccess("Hello world", "text/plain"));

// use RestTemplate ...

mockServer.verify();
```

在上面的例子中, MockRestServiceServer ——中央类测试,配置客户端休息 RestTemplate 用一个自定义的 ClientHttpRequestFactory 声称 实际的请求与预期并返回“存根”反应。 在 这种情况下我们期望单个请求“/问候”,想返回200年响应以“text / plain”内容。 我们可以定义许多 额外的请求和响应是必要的。存根

一旦预期请求和存根反应已经定义, RestTemplate 可以用在客户端代码吗 通常的。 最后的测试 mockServer.verify() 可以用来验证所有期望的要求进行。

静态导入

就像服务器端测试,流利的API,用于客户端 测试需要几个静态导入。 这些都很容易找到的 搜索 “MockRest *” 。 Eclipse用户应该添加 “MockRestRequestMatchers。*” 和 “MockRestResponseCreators。*” 为“最喜欢的 静态成员“在 Eclipse首选项下 Java - > - >编辑- >内容帮助最爱 。 这允许 使用内容辅助输入第一个字符后的静态 方法名。 其他ide(例如。IntelliJ)可能不需要任何额外的 配置。 只是检查支持代码完成静态 成员。

进一步的例子,其他的客户端测试

Spring MVC测试的测试包括 例子 测试 其他客户端测试。

11.3.7A宠物诊所的例子

PetClinic应用程序很好地,可以从 样本库 ,说明了 的几个特点 春天还是和TestContext框架 在JUnit 4.5 +环境。 大多数测试功能包含在 AbstractClinicTests ,对于这部分清单 如下所示:

```
import static org.junit.Assert.assertEquals;
// import ...

@ContextConfiguration
public abstract class AbstractClinicTests extends AbstractTransactionalJUnit4SpringContextTests {

    @Autowired
    protected Clinic clinic;

    @Test
    public void getVets() {
        Collection<Vet> vets = this.clinic.getVets();
        assertEquals("JDBC query must show the same number of vets",
            super.countRowsInTable("VETS"), vets.size());
        Vet v1 = EntityUtils.getById(vets, Vet.class, 2);
        assertEquals("Leary", v1.getLastName());
        assertEquals(1, v1.getNrOfSpecialties());
        assertEquals("radiology", (v1.getSpecialties().get(0)).getName());
        // ...
    }

    // ...
}
```

注释:

- 这个测试用例扩展了 AbstractTransactionalJUnit4SpringContextTests 类,它继承了配置用于依赖注入 (通过 DependencyInjectionTestExecutionListener)和 事务行为(通过 TransactionalTestExecutionListener)。
- 这个 诊所 一个实例变量的 应用程序对象正在测试一个设置依赖项注入 通过 @ autowired 语义。
- 这个 testGetVets() 方法说明了 如何使用继承吗 countRowsInTable() 方法容易验证 的行数在一个给定的表,从而验证正确的行为 被测试的应用程序代码。 这允许更强的测试 和减少依赖准确的测试数据。 例如,您可以 在数据库中添加额外的行不打断测试。
- 像许多集成测试,使用一个数据库,大部分的 测试 AbstractClinicTests 取决于 最小数量的数据在数据库中已经在测试用例运行。 或者,您可以选择填充数据库内 测试夹具设置测试用例的一个再次,在相同的 事务作为测试。

PetClinic应用程序很好地支持三种数据访问技术: JDBC,Hibernate和JPA。 通过声明 @ContextConfiguration 没有任何特定的 资源的位置, AbstractClinicTests 类 将其应用程序上下文加载默认位置, AbstractClinicTests-context.xml ,它声明了一个常见 数据源 。 子类指定附加 上下文位置,必须声明一个 PlatformTransactionManager 和一个具体的 实施 诊所 。

例如,Hibernate实现的测试等 包含以下的实现。 对于这个示例, HibernateClinicTests 不包含一个单一的线吗 代码:我们只需要声明 @ContextConfiguration 和测试 继承 AbstractClinicTests 。 因为 @ContextConfiguration 声明没有 任何特定的资源的位置, 春天还是和TestContext 框架 加载一个应用程序上下文从所有的豆子 定义在 AbstractClinicTests-context.xml (即。 , 继承位置)和 HibernateClinicTests-context.xml , HibernateClinicTests-context.xml 可能覆盖 bean定义在 AbstractClinicTests-context.xml 。

```
@ContextConfiguration
public class HibernateClinicTests extends AbstractClinicTests { }
```

在一个大型的应用程序中,Spring配置通常是 跨越多个文件。 因此,配置位置 通常一个共同的基类中指定的所有特定于应用程序的 集成测试。 这样的一个基类也可能添加有用的实例 变量一个人口依赖注入,自然一个如 SessionFactory 对于一个应用程序使用 Hibernate。

只要有可能,你应该完全相同的弹簧 配置文件在已部署的集成测试 环境。 一个可能的角度的差别是数据库连接 池和事务基础设施。 如果你部署一个 成熟的应用程序服务器,您可能会使用它的连接池 (可以通过JNDI)和JTA的实现。 因此在生产你 将使用一个 JndiObjectFactoryBean 或 < jee:jndi查找> 为 数据源 和 JtaTransactionManager 。 JNDI和JTA不会 可在容器外集成测试,那么你应该使用一个 像下议院DBCP组合 BasicDataSource 和 DataSourceTransactionManager 或 HibernateTransactionManager 为他们。 你可以 这个差异的行为因素到一个单独的XML文件,有选择 在应用程序服务器和一个 “当地” 配置独立于所有 其他配置,这将不是测试和生产之间的不同 环境。 此外,建议使用属性文件 连接设置。 看到 PetClinic应用程序很好地为例。

11.4进一步资源

参考下面的参考资料了解更多信息 测试:

- JUnit : 一个 一个面向程序员测试框架为Java 一个 。 使用Spring框架的测试套件。
- TestNG :一个测试 灵感来自JUnit框架添加了支持Java 5注释, 测试组,数据驱动测试,分布式测试等。
- MockObjects.com :网站 致力于模拟对象,一个技术,提高设计的代码 在测试驱动开发。
- “模拟 对象” :在维基百科上的文章。
- EasyMock :Java 图书馆 一个 模仿对象提供的接口(和 对象通过类扩展)通过生成它们在飞行中使用 Java的代理机制。 一个 使用Spring框架 在它的测试套件。
- JMock :图书馆, 支持测试驱动开发的Java代码与模仿对象。
- 5 :Java模拟 图书馆基于 测试间谍 模式。
- DbUnit : JUnit扩展(也可用与Ant和Maven)目标 数据库驱动的项目,在其他方面,使您的数据库 到一个已知状态测试运行之间。
- 这个 磨床 :Java负载测试框架。

PartA IV。 一个数据访问

这部分的参考文档涉及数据 访问和之间的交互数据访问层和 业务或服务层。

春天的综合事务管理支持覆盖 在一些细节,其次是全面覆盖各种各样的数据访问 框架和技术,Spring框架集成 与。

- ChapterA 12, 事务管理
- ChapterA 13, DAO支持
- ChapterA 14, 数据访问与JDBC
- ChapterA 15, 对象关系映射(ORM)数据访问
- ChapterA 16, 编组XML使用O / X映射器

12。 一个事务管理

12.1一个介绍Spring框架事务管理

综合事务支持是其中最引人注目的 理由使用Spring框架。 Spring框架提供了一个 一致的抽象为事务管理,提供 以下好处:

- 一致的编程模型在不同事务api 如Java Transaction API(JTA)、JDBC、Hibernate、Java持久性 API(JPA)和Java数据对象(JDO)。
- 支持 声明 事务管理 。
- 简单的API,用于 编程 事务 管理比复杂事务api如JTA。
- 优秀的整合与Spring的数据访问 抽象。

以下部分描述了Spring框架的事务 的增值和技术。 (本章还包括讨论 最佳实践、应用服务器集成,解决常见的 问题。)

- 优势的弹簧 框架的事务支持模型 描述 为什么 你会使用Spring框架的 事务抽象而不是EJB容器管理的事务 (CMT)或选择驱动本地事务通过专有 如Hibernate API。
- 理解春天 框架事务抽象 概述了核心类和 描述了如何配置和获取 数据源 实例从各种各样的 来源。
- 同步 资源交易 描述如何在应用程序代码 确保资源创建、重用,并清理干净 正确。
- 声明式事务 管理 描述了支持声明式事务 管理。
- 编程 事务管理 覆盖支持编程(, 是,显式地进行编码)事务管理。

12.2一个优势的Spring框架的事务支持模型

传统上,Java EE开发人员有两种选择 事务管理: 全球 或 当地 交易,它们都有深刻的 的局限性。 全球和本地事务管理了 接下来的两个部分,然后讨论了如何Spring框架的 事务管理支持地址的限制全球和 本地事务模型。

12.2.1A全局事务

全局事务使您能够处理多个事务 资源,一般关系数据库和消息队列。 这个 应用服务器管理全局事务通过JTA,这是 一个笨重的API来使用(部分是由于其异常模型)。 此外,一个JTA UserTransaction 通常需要来自JNDI,意味着你 也 需要使用JNDI为了使用JTA。 显然使用全局事务将限制任何潜在的重用 应用程序代码,因为JTA通常仅能在一个应用程序 服务器环境。

此前,首选的方法是通过使用全局事务 EJB cmt (容器管理 事务):CMT是形式的 **声明式事务管理** (有别于 **程序性事务 管理**)。 EJB CMT删除事务相关的需要 JNDI查找,当然,使用EJB本身就需要 使用JNDI。 它删除的大部分,但不是全部需要编写Java代码控制事务。 重大的缺点是CMT挂钩 JTA和一个应用程序服务器环境。 同时,它才可用 一个选择实现业务逻辑ejb,或者至少后面事务性EJB facade。 底片的EJB一般来说是如此伟大 这不是一个有吸引力的,尤其是在面对 引人注目的方法声明式事务管理。

12.2.2A本地事务

本地事务是特定于资源,如一个事务 关联到一个JDBC连接。 本地事务可能会更容易 使用,但有明显的缺点:他们无法跨越 多个事务资源。 例如,代码管理 交易使用JDBC连接不能运行在一个全球JTA 事务。 因为应用程序服务器是没有参与 事务管理,它不能帮助确保正确性跨越 多个资源。 (值得注意的是,大多数应用程序使用 单独的事务资源。) 另一个缺点是当地的 事务是侵入性的编程模型。

12.2.3A Spring框架的一致的编程模型

春天解决全球和本地的缺点 事务。 它使应用程序开发人员使用 一致 编程模型 在任何 环境 。 你写你的代码一次,并且它可以受益 从不同的事务管理策略在不同的 环境。 Spring框架提供了声明式和 编程式事务管理。 大多数用户喜欢声明 事务管理,建议在大多数情况下。

与程序性事务管理,开发人员的工作 Spring框架事务的抽象,它可以运行在任何 底层事务基础设施。 与首选的声明式模型,开发人员通常写小或 没有代码相关的事务管理,因此不依赖 Spring框架事务API,或任何其他事务 API。

你需要一个应用服务器的事务 管理?

Spring框架的事务管理支持变化 传统的规则,当一个企业Java应用程序的需要 一个应用程序服务器。

特别是,您不需要一个应用程序服务器的简单 声明性事务通过ejb。 事实上,即使你的 应用程序服务器具有强大功能JTA,你可能决定 Spring框架的声明性事务提供更多的力量和一个 更有效率的编程模型与EJB CMT。

通常你需要一个应用程序服务器的JTA能力只有如果您的应用程序需要处理多个事务 资源,而不是一个要求对于很多应用程序。 许多 高端应用程序使用一个单一的、高度可扩展的数据库(如Oracle RAC)相反。 独立的事务管理器等 于Atomikos交易 和 JOTM 是其他 选项。 当然,你可能需要其他应用服务器功能 如Java消息服务(JMS)和J2EE连接器体系结构 (JCA)。

Spring框架 给你选择的什么时候 你的应用程序和规模完全加载应用程序 服务器 。 的日子一去不复返了唯一的选择 使用EJB CMT或JTA是写代码和本地事务如 那些在JDBC连接,面临巨额返工如果你需要 代码运行在全球,容器管理的事务。 与 Spring框架,只有一些bean定义在你的配置文件,而不是你的代码,需要改变。

12.3一个了解Spring框架事务抽象

Spring事务的关键抽象的概念 事务策略 。 一个事务策略是 定义的 org.springframework.transaction.PlatformTransactionManager 接口:

```
public interface PlatformTransactionManager {

  TransactionStatus getTransaction(TransactionDefinition definition)
    throws TransactionException;

  void commit(TransactionStatus status) throws TransactionException;

  void rollback(TransactionStatus status) throws TransactionException;
}
```

这主要是一个服务提供程序接口(SPI),尽管它 可以用 编程 从你 应用程序代码。 因为 PlatformTransactionManager 是一个接口 ,它可以很容易地嘲笑或粗短的作为 必要的。 它不是绑定到一个JNDI查找策略等。 PlatformTransactionManager 实现像任何其他对象定义(或bean)在Spring框架奥委会 集装箱。 这个好处足以使得Spring框架交易一个 有价值的抽象甚至当你使用JTA。 事务代码可以 被测试更容易比直接使用JTA。

又符合春天的哲学 TransactionException 可以抛出 任何 PlatformTransactionManager 接口的方法是 unchecked (即,它扩展了 这个 java朗runtimeexception 类)。 事务基础设施故障是几乎总是致命的。 在罕见的 情况下,应用程序代码可以恢复一个事务 失败,应用程序开发人员仍然可以选择捕捉和处理 TransactionException 。 最关键的一点是 开发人员不 迫使 这样做。

这个 getTransaction(. .) 方法返回一个 TransactionStatus 对象,根据一个 TransactionDefinition 参数。 这个 返回 TransactionStatus 可能代表 一个新的事务,或能代表一个现有的交易如果匹配 交易存在于当前调用堆栈。 在暗示这 后一种情况是,与Java EE事务上下文,一个 TransactionStatus 关联到一个 **线程** 的执行。

这个 TransactionDefinition 接口 指定:

- **隔离** :的程度 这个交易是孤立于其他事务的工作。 对于 的例子,可以看到未提交该事务从其他写 交易吗?
- **传播** :通常,所有 事务范围内执行的代码将运行在该事务。 然而,您可以选择指定行为的事件 一个事务性方法时执行一个事务上下文 已经存在。 对于 示例中,代码可以继续运行在现有的事务(常见的情况),或现有的事务可以暂停和一个新的 交易创造了。 Spring提供了所有的事务 传播选项从EJB CMT熟悉 。 阅读 事务传播的语义在春天,看到 SectionA 12 5 7,一个propagationa事务 。

- **超时** 多长时间这 事务运行时间自动回滚 由底层事务基础设施。
- **只读状态** :一个只读 事务可以用在你的代码读取但不修改数据。 只读事务可以是一个有用的优化在某些情况下, 例如当你使用Hibernate。

这些设置反映标准事务的概念。 如果 必要的,请参阅参考资料,讨论事务隔离级别 和其他核心事务的概念。 理解这些概念是 基本使用Spring框架或任何事务管理 解决方案。

这个 TransactionStatus 接口 提供了一个简单的方法来控制事务的事务代码 执行和查询交易状态。 应该熟悉的概念, 因为他们是常见的所有事务api:

```
public interface TransactionStatus extends SavepointManager {
    boolean isNewTransaction();
    boolean hasSavepoint();
    void setRollbackOnly();
    boolean isRollbackOnly();
    void flush();
    boolean isCompleted();
}
```

无论你选择声明或编程 事务管理在春天,定义正确的 PlatformTransactionManager 实现 是绝对必要的。 你通常定义这个实现通过 依赖注入。

PlatformTransactionManager 实现通常需要知识的环境中 他们的工作:JDBC,JTA,Hibernate,等等。 下面的例子展示 如何定义一个地方吗 PlatformTransactionManager 实现。 (这个例子使用纯JDBC)。

你定义一个JDBC 数据源

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
  <property name="driverClassName" value="${jdbc.driverClassName}" />
  <property name="url" value="${jdbc.url}" />
  <property name="username" value="${jdbc.username}" />
  <property name="password" value="${jdbc.password}" />
</bean>
```

相关 PlatformTransactionManager 将bean定义的引用 数据源 定义。 它会看起来像这样:

```
<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"/>
</bean>
```

如果你使用JTA在一个Java EE容器然后你使用一个容器 数据源 ,通过JNDI, 结合Spring的 JtaTransactionManager 。 这就是JTA和JNDI查找版本将会看起来像:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/jee
    http://www.springframework.org/schema/jee/spring-jee.xsd">

  <jee:jndi-lookup id="dataSource" jndi-name="jdbc/jpetstore"/>

  <bean id="txManager" class="org.springframework.transaction.jta.JtaTransactionManager" />

  <!-- other <bean/> definitions here -->

</beans>
```

这个 JtaTransactionManager 不需要 知道 数据源 ,或任何其他 特定的资源,因为它使用容器的全局事务 管理基础设施。

注意

上述定义的 数据源 bean 使用 < jndi查找/ > 标签从 JEE 名称空间。 基于架构的更多信息 配置,请参阅 AppendixA E, XML的基于配置 ,和更多 信息 < jee / > 标签看到部分 题为 SectionA e 2 3,的 JEE schemaa 。

您还可以使用Hibernate本地事务容易,见 下面的例子。 在这种情况下,您需要定义一个Hibernate LocalSessionFactoryBean , 您的应用程序 代码将使用获得Hibernate 会话 实例。

这个 数据源 bean定义将 类似于当地的JDBC示例先前所显示的,因此不是 如下面的示例所示。

注意

如果 数据源 使用的任何 非jta事务管理器,通过JNDI查找和管理Java EE容器,那么它应该是事务性的,因为春天 框架,而不是Java EE容器,将管理 事务。

这个 txManager 在本例中是豆的 HibernateTransactionManager 类型。 以同样的方式 随着 DataSourceTransactionManager 需要一个 参考 数据源 , HibernateTransactionManager 需要一个参考 这个 SessionFactory 。

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
 <property name="dataSource" ref="dataSource" />
 <property name="mappingResources">
 <list>
  <value>org/springframework/samples/petclinic/hibernate/petclinic.hbm.xml</value>
 </list>
 </property>
 <property name="hibernateProperties">
  <value>
   hibernate.dialect=${hibernate.dialect}
  </value>
 </property>
</bean>

<bean id="txManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
 <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

如果您在使用Hibernate和Java EE容器管理的JTA 交易,那么你就应该使用相同的 JtaTransactionManager 与前面的JTA JDBC示例。

```
<bean id="txManager" class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

注意

如果你使用JTA,然后你的事务管理器定义 看起来一样不管你用什么样的数据访问技术,是它 JDBC、JPA支持Hibernate或其他技术。 这是由于 事实上,JTA事务是全局事务,可以招募 任何事务性资源。

在所有这些情况下,应用程序代码不需要改变。 你 可以改变交易管理仅仅通过改变配置, 即使这变化意味着要从局部到全局事务或副 亦然。

12.4一个同步资源交易

它现在应该清楚你如何创建不同的事务 经理,以及他们如何与相关资源需要 同步到事务(例如 DataSourceTransactionManager 到一个JDBC 数据源 , HibernateTransactionManager 到一个Hibernate SessionFactory ,等等)。 本节 描述如何在应用程序代码中,直接或间接地使用 持久性API,如JDBC、Hibernate或JDO,确保这些 资源创建、重用,并清理干净。 这个部分还 讨论了事务同步触发(可选) 通过相关的 PlatformTransactionManager 。

12.4.1A高级同步方法

首选的方法是使用Spring的最高水平模板 建立持久性api或使用本机集成ORM api 事务——意识到工厂bean或代理来管理本机 资源工厂。 这些感知事务的解决方案在内部处理 资源创建和重用,清理,可选的事务 同步的资源,和异常的映射。 因此用户数据 访问代码没有解决这些任务,但可以聚焦 纯粹是在非样板持久性逻辑。 通常,您使用 本机ORM API或采取一个 模板 方法用于JDBC 访问通过使用 JdbcTemplate 。 这些解决方案 在接下来的章节中详细的参考文档。

12.4.2A低级同步方法

类比如 DataSourceUtils (JDBC), EntityManagerFactoryUtils (JPA), SessionFactoryUtils (用于Hibernate), PersistenceManagerFactoryUtils (JDO)等 在一个较低的水平上存在。 当你想要的应用程序代码来处理 直接与资源类型的本地持久化api,您可以使用 这些类来确保适当的Spring框架管理实例 ,给出了事务(可选地)同步,和异常 在这个过程中发生的正确

映射到一个一致的 API。

例如,JDBC的情况,而不是传统的JDBC 方法调用 getConnection() 方法 数据源 ,你相反使用Spring的 org.springframework.jdbc.datasource.DataSourceUtils 类如下:

```
Connection conn = DataSourceUtils.getConnection(dataSource);
```

如果现有的事务已经有一个连接同步 (链接),该实例返回。 否则,该方法调用 触发器创建一个新的连接,这是(可选) 同步到任何现有的事务,并把这些信息 随后的重用在同一事务。 如前所述,任何 SQLException 包装在一个春天 框架 CannotGetJdbcConnectionException 之一, Spring框架的层次结构的DataAccessExceptions无限制。 这 方法让你可以获得更多的信息比很容易从 SQLException ,并确保可移植性 跨数据库,甚至跨越不同的持久化技术。

这种方法还没有春天事务管理工作 (事务同步是可选的),所以你可以使用它不管 不是你是使用Spring事务管理。

当然,一旦你使用了Spring JDBC支持,JPA支持 或Hibernate支持,你通常不喜欢使用 DataSourceUtils 或其他辅助类, 因为你会变得更快乐工作穿过春天的抽象 比直接与相关的api。 例如,如果您使用弹簧 JdbcTemplate 或 jdbc对象 包来简化你的使用 JDBC,正确连接检索 发生在幕后,你不需要写任何特别 代码。

12.4.3A TransactionAwareDataSourceProxy

非常最低级别的存在 TransactionAwareDataSourceProxy 类。 这是一个 代理为目标 数据源 ,这 包装了目标 数据源 添加 spring管理事务的意识。 在这方面,它是类似的 一个事务性JNDI 数据源 作为 提供了一个Java EE服务器。

它应该几乎从不是必要或可取的使用这个 类,除了现有的代码必须调用和通过了一项标准 JDBC 数据源 接口的实现。 在这种情况下,它是可能的,这个代码是可用的,但参与 在春季管理事务。 最好是写你的新代码 通过使用上面提到的更高层次的抽象。

12.5一个声明式事务管理

注意

大多数Spring框架用户选择声明式事务 管理。 这个选项有至少影响应用程序代码, 因此是最符合理想的一个 非侵入性的 轻量级容器。

Spring框架的声明式事务管理了 可能与Spring面向方面编程(AOP),虽然,作为 事务方面代码带有Spring框架分布 和可以用于一个样板时尚,AOP概念不一般 必须理解有效地利用这段代码。

Spring框架的声明式事务管理是相似的 到EJB CMT,您可以指定事务行为(或缺乏) 方法级到个人。 就可以做 setRollbackOnly() 调用事务内 如果有必要,上下文 两者的不同类型的事务 管理是:

- 与EJB CMT,这与JTA,Spring框架的 声明式事务管理工作在任何环境。 它可以 使用JTA事务或本地事务使用JDBC、JPA, Hibernate或JDO通过简单的调整配置文件。
- 你可以使用Spring框架声明式事务 管理任何类,而不只是特殊的类,比如 ejb。
- Spring框架提供了声明 回滚 规则 , 一个功能没有EJB等效。 两 编程和声明支持回滚规则 提供了。
- Spring框架使您可以自定义事务 行为,通过使用AOP。 例如,您可以插入自定义行为 事务回滚的情况。 你也可以添加任意的建议, 随着事务的建议。 与EJB CMT,你不能影响 容器的事务管理除外 setRollbackOnly() 。
- Spring框架不支持事务的传播 在远程调用上下文,高端应用服务器。 如果 你需要这个功能,我们建议您使用EJB。 然而, 使用前仔细考虑这样一种特性,因为正常情况下,一个 不希望交易跨越远程调用。

回滚规则的概念很重要:他们让你 指定哪些异常(和throwables)应该 导致自动回滚。 你指定这个声明,在 配置,而不是Java代码。 所以,尽管你仍然可以调用 setRollbackOnly() 在 TransactionStatus 对象可以回滚 回当前事务,通常您可以指定一个规则, MyApplicationException 必须总是结果在回滚。 其显著的优势,这个选项是业务 对象不依赖于事务基础设施。 例如,他们 通常不需要进口弹簧事务api或其他弹簧 api。

虽然EJB容器的默认行为的自动回滚 事务在一个 系统异常 (通常是一个运行时 例外),EJB CMT不回滚事务自动在一个 应用程序异常 (即检查异常 除了 java rmi remoteexception)。 而 春天的默认行为的声明式事务管理 遵循 EJB公约(回滚是自动只在未经检查的异常),它 经常是有用的定制此行

在哪里 TransactionProxyFactoryBean 吗?

声明式事务配置Spring 2.0的版本 及以上很大区别的从以前版本的春天。 这个 主要的区别是,不再有任何需要配置 TransactionProxyFactoryBean 豆子。

2.0配置风格的pre春天依然是100%有效的 配置;认为新 < tx:标签/ > 作为 简单地定义 TransactionProxyFactoryBean bean 代表你。

为。

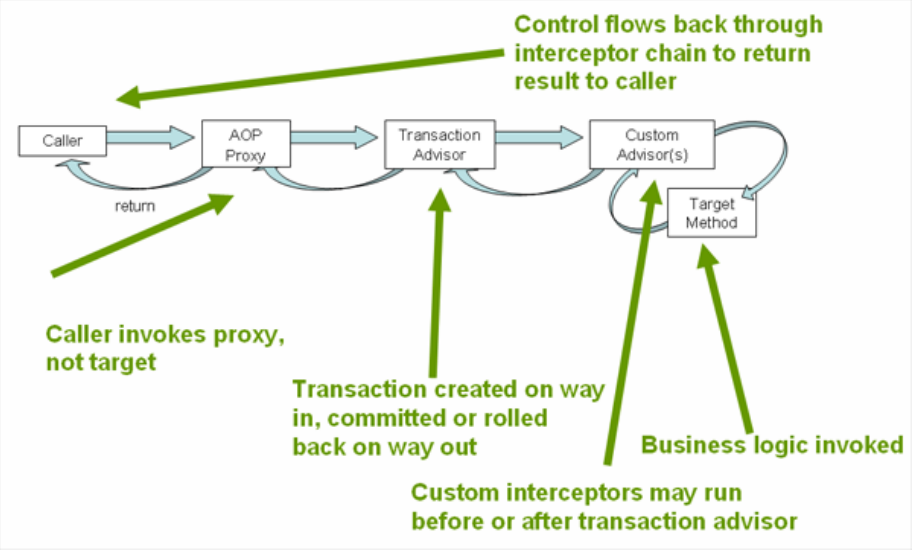
12.5.1A了解Spring框架的声明性事务 实现

它并不足以告诉你简单的注释你类 与 transactional 注释,添加 @EnableTransactionManagement 你 配置,然后期待你了解它是如何运作的。 这 部分解释了Spring框架的内部运作的 声明式事务基础设施在发生 事务相关的问题。

最重要的概念掌握关于春天 框架的声明式事务的支持,这种支持是 启用 通过AOP 代理 ,事务性的建议是驱动 通过 元数据 (目前XML -或基于注释)。 结合AOP与元数据收益率AOP代理事务 使用 TransactionInterceptor 结合 用一个合适 PlatformTransactionManager 实现驱动交易 约方法 调用 。

> **注意**
>
> Spring AOP覆盖着 ChapterA 9, 面向方面的编程与弹簧 。

从概念上讲,调用一个方法在一个事务代理看起来像 这.....



12.5.2A声明式事务实现的例子

考虑下面的界面,和它的服务员 实现。 这个示例使用 foo 和 酒吧 类作为占位符,这样您就可以 专注于事务的使用没有专注于一个特定的 域模型。 对于本示例的目的,这一事实 DefaultFooService 类抛出 UnsupportedOperationException 实例 在身体的每个实现的方法是好的,它可以让你看到 创建回滚事务,然后在回应的 UnsupportedOperationException 实例。

```
// the service interface that we want to make transactional

package x.y.service;

public interface FooService {

  Foo getFoo(String fooName);

  Foo getFoo(String fooName, String barName);

  void insertFoo(Foo foo);

  void updateFoo(Foo foo);

}
```

```
// an implementation of the above interface

package x.y.service;

public class DefaultFooService implements FooService {

  public Foo getFoo(String fooName) {
    throw new UnsupportedOperationException();
  }

  public Foo getFoo(String fooName, String barName) {
    throw new UnsupportedOperationException();
  }
```

```
  public void insertFoo(Foo foo) {
    throw new UnsupportedOperationException();
  }

  public void updateFoo(Foo foo) {
    throw new UnsupportedOperationException();
  }

}
```

假设前两个方法 fooService 接口 , getFoo(字符串) 和 getFoo(String,String), 必须执行事务的上下文中与只读 语义,和其他的方法 ,insertFoo(Foo) 和 updateFoo(Foo), 必须执行上下文中的一个吗 事务与读写语义。 下面的配置是 详细解释在接下来的几个段落。

```
<!-- from the file 'context.xml' -->
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop.xsd">

  <!-- this is the service object that we want to make transactional -->
  <bean id="fooService" class="x.y.service.DefaultFooService"/>

  <!-- the transactional advice (what 'happens'; see the <aop:advisor/> bean below) -->
  <tx:advice id="txAdvice" transaction-manager="txManager">
  <!-- the transactional semantics... -->
  <tx:attributes>
    <!-- all methods starting with 'get' are read-only -->
    <tx:method name="get*" read-only="true"/>
    <!-- other methods use the default transaction settings (see below) -->
    <tx:method name="*"/>
  </tx:attributes>
  </tx:advice>

  <!-- ensure that the above transactional advice runs for any execution
    of an operation defined by the FooService interface -->
  <aop:config>
  <aop:pointcut id="fooServiceOperation" expression="execution(* x.y.service.FooService.*(..))"/>
  <aop:advisor advice-ref="txAdvice" pointcut-ref="fooServiceOperation"/>
  </aop:config>

  <!-- don't forget the DataSource -->
  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
  <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
  <property name="url" value="jdbc:oracle:thin:@rj-t42:1521:elvis"/>
  <property name="username" value="scott"/>
  <property name="password" value="tiger"/>
  </bean>

  <!-- similarly, don't forget the PlatformTransactionManager -->
  <bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"/>
  </bean>

  <!-- other <bean/> definitions here -->

</beans>
```

检查前面的配置。 你想让一个服务 对象, fooService 豆、事务。 这个 事务语义被封装在应用 < tx:建议/ > 定义。 这个 < tx:建议/ > 定义读起来 一个 ... 所有的方法在开始 ' GET ' 是执行上下文中的只读 事务,和所有其他的方法都是用默认执行 事务语义 一个 。 这个 事务管理器 属性的 < tx:建议/ > 标签被设置的名称 PlatformTransactionManager bean, 去 驱动 交易,在这种情况下, txManager bean。

提示

你可以省略的 事务管理器 属性在事务的建议 (< tx:建议/ >)如果bean的名称 PlatformTransactionManager 那你 想线在有名字 transactionManager 。 如果 这个 PlatformTransactionManager bean 那你想线在有任何其他的名字,那么你必须使用 事务管理器 属性明确,如 前面的例子。

这个 < aop:config / > 定义确保 定义事务的建议 txAdvice bean 执行在适当的分项目。 首先定义一个 切入点匹配执行任何操作的定义 fooService 接口 (fooServiceOperation)。 然后你将 切入点与 txAdvice 使用一个顾问。 这个 结果表明,在执行

fooServiceOperation ,建议定义为 txAdvice 将运行。

表达式中定义 < aop:切入点/ > 元素是一个AspectJ切入点 表达式;看到 ChapterA 9, 面向方面的编程与弹簧 为更多的细节在切入点 表达式在Spring 2.0。

一个共同的要求就是让整个服务层 事务。 最好的方法是简单地改变切入点 表达式来匹配任何操作在你的服务层。 对于 示例:

```
<aop:config>
  <aop:pointcut id="fooServiceMethods" expression="execution(* x.y.service.*.*(..))"/>
  <aop:advisor advice-ref="txAdvice" pointcut-ref="fooServiceMethods"/>
</aop:config>
```

注意

在这个例子中它是假定你所有的服务 接口中定义的 x y服务 包,看到 ChapterA 9, 面向方面的编程与弹簧 更多 细节。

现在,我们已经分析了配置,您可能会问 你自己, 一个 好吧…… 但是什么配置 其实做什么? 一个 。

上面的配置将被用来创建一个事务 代理在对象创建的 fooService bean定义。 这个 代理将配置事务的建议,因此,当一个 调用适当的方法 在代理 ,一个 事务开始,暂停,标记为只读的,等等, 根据事务的配置与之关联的方法。 考虑下面的程序,测试驱动器上面 配置:

```
public final class Boot {

  public static void main(final String[] args) throws Exception {
    ApplicationContext ctx = new ClassPathXmlApplicationContext("context.xml", Boot.class);
    FooService fooService = (FooService) ctx.getBean("fooService");
    fooService.insertFoo (new Foo());
  }
}
```

运行的输出将类似于前面的程序 以下。 (Log4J输出和堆栈跟踪的 抛出UnsupportedOperationException由insertFoo(. .)方法 DefaultFooService类已经被截断为清晰起见)。

```
  <!-- the Spring container is starting up... -->
[AspectJInvocationContextExposingAdvisorAutoProxyCreator] - Creating implicit proxy
    for bean 'fooService' with 0 common interceptors and 1 specific interceptors
  <!-- the DefaultFooService is actually proxied -->
[JdkDynamicAopProxy] - Creating JDK dynamic proxy for [x.y.service.DefaultFooService]

  <!-- ... the insertFoo(..) method is now being invoked on the proxy -->

[TransactionInterceptor] - Getting transaction for x.y.service.FooService.insertFoo
  <!-- the transactional advice kicks in here... -->
[DataSourceTransactionManager] - Creating new transaction with name [x.y.service.FooService.insertFoo]
[DataSourceTransactionManager] - Acquired Connection
    [org.apache.commons.dbcp.PoolableConnection@a53de4] for JDBC transaction

  <!-- the insertFoo(..) method from DefaultFooService throws an exception... -->
[RuleBasedTransactionAttribute] - Applying rules to determine whether transaction should
    rollback on java.lang.UnsupportedOperationException
[TransactionInterceptor] - Invoking rollback for transaction on x.y.service.FooService.insertFoo
    due to throwable [java.lang.UnsupportedOperationException]

   <!-- and the transaction is rolled back (by default, RuntimeException instances cause rollback) -->
[DataSourceTransactionManager] - Rolling back JDBC transaction on Connection
    [org.apache.commons.dbcp.PoolableConnection@a53de4]
[DataSourceTransactionManager] - Releasing JDBC Connection after transaction
[DataSourceUtils] - Returning JDBC Connection to DataSource

Exception in thread "main" java.lang.UnsupportedOperationException
    at x.y.service.DefaultFooService.insertFoo(DefaultFooService.java:14)
   <!-- AOP infrastructure stack trace elements removed for clarity -->
    at $Proxy0.insertFoo(Unknown Source)
    at Boot.main(Boot.java:11)
```

12.5.3A声明式事务回滚

上一节概述了如何指定的基本知识 事务设置类,通常服务层的类, 在应用程序中声明。 本节描述如何 控制回滚的事务在一个简单的声明 时尚。

推荐的方式来表示对Spring框架的 交易的基础设施,一个事务的工作是卷 回来是抛出 异常 从代码 这是当前事务的上下文中执行。 春天 框架的事务基础设施代码将抓住任何未处理的 异常 因为它的泡沫在呼叫 堆栈,并确定是否标志着交易 回滚。

在默认配置中,Spring框架的事务 基础设施代码 只有 标志着交易 回滚的情况下运行时,未经检查的异常,即当 抛出异常的子类的一个实例或 RuntimeException 。 (误差 年代也会——默认情况下)的结果 在回滚)。 Checked异常,抛出一个事务性的 方法做 不 结果在回滚在默认 配置。

您可以配置哪些 异常 马克一个事务的类型 回滚,包括已检查的异常。 下面的XML片段 演示了如何配置回滚一个检查, 特定于应用程序的 异常 类型。

```
<tx:advice id="txAdvice" transaction-manager="txManager">
  <tx:attributes>
  <tx:method name="get*" read-only="true" rollback-for="NoProductInStockException"/>
  <tx:method name="*"/>
  </tx:attributes>
</tx:advice>
```

您还可以指定“不回滚规则”,如果你做 不 想要一个事务回滚,当一个 抛出异常。 下面的例子告诉Spring框架的 事务基础设施提交服务员事务甚至在 面对一个未处理的 InstrumentNotFoundException 。

```
<tx:advice id="txAdvice">
  <tx:attributes>
  <tx:method name="updateStock" no-rollback-for="InstrumentNotFoundException"/>
  <tx:method name="*"/>
  </tx:attributes>
</tx:advice>
```

当Spring框架的事务基础设施捕获 例外,咨询来确定是否配置回滚规则 标记为回滚的事务, 最强 匹配规则赢了。 所以对于以下配置,任何 例外之外的其他 InstrumentNotFoundException 结果在一个 回滚事务的服务员。

```
<tx:advice id="txAdvice">
  <tx:attributes>
  <tx:method name="*" rollback-for="Throwable" no-rollback-for="InstrumentNotFoundException"/>
  </tx:attributes>
</tx:advice>
```

您还可以显示一个需要回滚 编程 。 虽然很简单,这 过程是相当入侵,代码紧密耦合的春天 框架的事务基础设施:

```
public void resolvePosition() {
  try {
    // some business logic...
  } catch (NoProductInStockException ex) {
    // trigger rollback programmatically
    TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();
  }
}
```

强烈建议您使用声明性方法 回滚如果可能的话。 程序化的回滚是可用的应该 你绝对需要它,但它的用法苍蝇在面对实现 基于pojo的架构,清洁

12.5.4A配置不同的事务语义不同 bean

考虑的情况你有许多的服务层 对象,你想申请一个 完全不同 他们每个人的事务配置。 这可以通过定义 截然不同的 < aop:顾问/ > 元素有不同 切入点 和 建议裁判 属性 值。

作为一个点的比较,首先假设你所有的服务 层类定义在一个根 x y服务 包。 让所有bean实例的类的定义在这 包(或子包)和名称结尾 服务 有默认的事务配置, 你会写以下:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="
  http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/tx
  http://www.springframework.org/schema/tx/spring-tx.xsd
  http://www.springframework.org/schema/aop
  http://www.springframework.org/schema/aop/spring-aop.xsd">

  <aop:config>

    <aop:pointcut id="serviceOperation"
        expression="execution(* x.y.service..*Service.*(..))"/>

    <aop:advisor pointcut-ref="serviceOperation" advice-ref="txAdvice"/>
```

```
  </aop:config>

  <!-- these two beans will be transactional... -->
  <bean id="fooService" class="x.y.service.DefaultFooService"/>
  <bean id="barService" class="x.y.service.extras.SimpleBarService"/>

  <!-- ... and these two beans won't -->
  <bean id="anotherService" class="org.xyz.SomeService"/> <!-- (not in the right package) -->
  <bean id="barManager" class="x.y.service.SimpleBarManager"/> <!-- (doesn't end in 'Service') -->

  <tx:advice id="txAdvice">
    <tx:attributes>
      <tx:method name="get*" read-only="true"/>
      <tx:method name="*"/>
    </tx:attributes>
  </tx:advice>

  <!-- other transaction infrastructure beans such as a PlatformTransactionManager omitted... -->

</beans>
```

下面的例子展示了如何配置两个不同的豆子 与完全不同的事务设置。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="
  http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/tx
  http://www.springframework.org/schema/tx/spring-tx.xsd
  http://www.springframework.org/schema/aop
  http://www.springframework.org/schema/aop/spring-aop.xsd">

  <aop:config>

    <aop:pointcut id="defaultServiceOperation"
        expression="execution(* x.y.service.*Service.*(..))"/>

    <aop:pointcut id="noTxServiceOperation"
        expression="execution(* x.y.service.ddl.DefaultDdlManager.*(..))"/>

    <aop:advisor pointcut-ref="defaultServiceOperation" advice-ref="defaultTxAdvice"/>

    <aop:advisor pointcut-ref="noTxServiceOperation" advice-ref="noTxAdvice"/>

  </aop:config>

  <!-- this bean will be transactional (see the 'defaultServiceOperation' pointcut) -->
  <bean id="fooService" class="x.y.service.DefaultFooService"/>

  <!-- this bean will also be transactional, but with totally different transactional settings -->
  <bean id="anotherFooService" class="x.y.service.ddl.DefaultDdlManager"/>

  <tx:advice id="defaultTxAdvice">
    <tx:attributes>
      <tx:method name="get*" read-only="true"/>
      <tx:method name="*"/>
    </tx:attributes>
  </tx:advice>

  <tx:advice id="noTxAdvice">
    <tx:attributes>
      <tx:method name="*" propagation="NEVER"/>
    </tx:attributes>
  </tx:advice>

  <!-- other transaction infrastructure beans such as a PlatformTransactionManager omitted... -->

</beans>
```

12.5.5A < tx:建议/ > 设置

本节总结了各种事务设置 可以指定使用吗 < tx:建议/ > 标签。 默认 < tx:建议/ > 设置如下:

- 传播设置 是 要求。
- 隔离级别是 默认的。
- 事务是读/写。
- 事务超时默认的默认超时 潜在的交易系统,或没有如果不会超时 支持。
- 任何 RuntimeException 触发 回滚,和任何检查 异常 不。

您可以更改这些默认设置;各种属性的 这个 < tx:方法/ > 是嵌在标签 < tx:建议/ > 和 < tx:属性/ > 标签进行了总结 下图:

为多12 1一个 < tx:方法/ > 设置

属性	要求?	默认	描述
名称	是的		方法名(s)的事务 属性是相联系的。 通配符(*)字符 可以用来副相同的事务属性 设置与一些方法,例如, 获得* , 处理* , 在*事件 ,等等。
传播	没有	需要	事务传播行为。
隔离	没有	默认	事务隔离级别。
超时	没有	1	事务超时的值(以秒为单位)。
只读	没有	假	这是事务只读?
回滚	没有		例外(s) 这引发 回滚,以逗号分隔。 例如, com.foo.MyBusinessException,ServletException。
没有回滚	没有		例外(s) ,做 不 触发回滚,以逗号分隔。 例如, com.foo.MyBusinessException,ServletException。

12.5.6A使用 transactional

除了基于xml的声明式事务的方法 配置,您可以使用一个基于注解的方法。 宣布 事务语义直接在Java源代码把 声明更接近受灾的代码。 没有太多的危险 不应有的耦合,因为代码是用来事务性的 几乎总是这样无论如何部署。

易用性所提供的使用 transactional 注释是最好的 通过一个具体实例说明了,这是解释的文本 遵循。 考虑下面的类定义:

```
// the service class that we want to make transactional
@Transactional
public class DefaultFooService implements FooService {

  Foo getFoo(String fooName);

  Foo getFoo(String fooName, String barName);

  void insertFoo(Foo foo);

  void updateFoo(Foo foo);
}
```

当上面的POJO被定义为在一个Spring IoC bean 容器,bean实例可以通过添加仅仅事务 一个 行XML配置:

```
<!-- from the file 'context.xml' -->
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop.xsd">

  <!-- this is the service object that we want to make transactional -->
  <bean id="fooService" class="x.y.service.DefaultFooService"/>

  <!-- enable the configuration of transactional behavior based on annotations -->
  <tx:annotation-driven transaction-manager="txManager"/>

  <!-- a PlatformTransactionManager is still required -->
  <bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <!-- (this dependency is defined somewhere else) -->
  <property name="dataSource" ref="dataSource"/>
  </bean>

  <!-- other <bean/> definitions here -->
```

```
</beans>
```

提示

你可以省略的 事务管理器 属性 < tx:注解驱动/ > 标签 如果bean的名称 PlatformTransactionManager 那你 想线在有名字 transactionManager 。 如果 这个 PlatformTransactionManager bean 那你想依赖 注入有任何其他的名字,然后你有 使用 事务管理器 属性 明确,如前面的例子。

注意

这个 @EnableTransactionManagement 注释提供了相同的功能如果您使用的是基于Java的 配置。 只需 添加注释 @ configuration 类。 看到Javadoc 详情。

你可以把 transactional 注释一个接口定义之前,一个方法,一个接口 类定义,或一个 公共 方法在类。 然而,仅仅是存在的 transactional 注释是不够的 激活事务性行为。 这个 transactional 注释只是 元数据,可以被一些运行时基础设施 transactional 了解并能使用 元数据配置适当的bean与事务性行为。 在前面的示例中, < tx:注解驱动/ > 元素 开关在 事务性行为。

方法可视性和 transactional

当使用代理,你应该申请 transactional 注释只 方法 公共 可见性。 如果你 注释保护,私人或包可见的方法 transactional 注释,没有错误 提高了,但是带注释的方法不存在配置 事务设置。 考虑使用AspectJ(见下文)如果你 需要标注非公有制的方法。

提示

春天建议你只标注具体类(和 方法的具体类) transactional 注释,而不是 来注解的接口。 你当然可以把 transactional 注释一个 接口(或一个接口方法),但这只能像你 期望它如果您使用的是基于接口的代理。 事实上, Java注释是 没有继承接口 意味着,如果您使用的是基于类的代理 (代理目标类= " true ")或编织的基础 方面(模式 = " aspectj"),那么事务 设置不认可的代理和编织 基础设施,和对象将不会被包裹在一个 事务代理,这将是明显的坏 。

注意

在代理模式(默认),只有外部方法调用 通过代理来截获。 这意味着 自动调用,实际上,在目标对象的方法调用另一种方法的目标对象,不会导致一个实际 事务在运行时即使调用方法是标记 transactional 。

考虑使用AspectJ模式(请参阅模式属性表 下图)如果你期望自我调用是用事务 好。 在 这种情况下,是不会有一个代理在第一个地方;相反, 目标类将编织(即它的字节代码将被修改) 才能 transactional 到运行时 行为的一种方法。

12.2为多。 一个注解驱动的事务设置

XML属性	注释属性	默认	描述
事务管理器	N / A(见 TransactionManagementConfigurer Javadoc)	transactionManager	事务管理器要使用的名称。 只需要 如果事务管理器的名称不是 transactionManager ,例子 以上。
模式	模式	代理	默认的模式 “代理” 过程注释 豆子是代理使用Spring的AOP框架(以下 代理语义,正如上面所讨论的,申请方法调用 通过代理来只)。 另一种模式 “aspectj” 而不是编织类与春天的的影响 AspectJ事务方面,修改目标类字节 代码,适用于任何类型的方法调用。 AspectJ编织 需要弹簧方面。 jar的类路径以及 装入时编织(或编译时编织)启用。 (见 一个章节弹簧configurationa 有关如何设置 装入时编织了。)
			只适用于代理模式。 控制什么类型的 事务代理创建类标注 这个 transactional 注释。 如果 代理目标类 属性设置 到 真正的

代理目标类	proxyTargetClass	假	,然后基于类的代理 创建的。 如果 代理目标类 是 假 或者如果属性是省略了,然后 基于接口的代理创建标准JDK。 (见 SectionA 9.6,一个mechanismsa代理 对于一个详细的检查 不同的代理类型。)
秩序	秩序	下令最低优先级	定义事务的顺序的建议 应用于bean注释吗transactional 。 (更多 相关规定信息订购AOP的建议, 看到 一个章节orderinga建议)。 没有 指定顺序意味着AOP子系统决定订单的建议。

注意

这个 代理目标类 属性控制什么 类型的事务代理创建类标注 这个 transactional 注释。 如果 代理目标类 设置为 真正的 ,基于类创建代理。 如果 代理目标类 是 假 或 如果省略该属性,标准JDK基于接口的代理 创建的。 (见 SectionA 9.6,一个mechanismsa代理 对于一个讨论 不同的代理类型。)

注意

@EnableTransactionManagement 和 < tx:注解驱动/ > 只希望 transactional 在bean在相同的 它们被定义在应用程序上下文。 这意味着,如果你把 注解驱动的配置在一个 WebApplicationContext 对于一个DispatcherServlet ,它只检查 transactional 豆子在你 控制器,而不是你的服务。 看到 SectionA 17.2,一个了 DispatcherServlet 一个 为更多的信息。

最派生位置优先当评估 事务设置方法。 在 下面的例子中, DefaultFooService 类注释的类 水平与设置只读事务,但是transactional 注释 updateFoo(Foo) 方法在同一个班花 优先考虑的事务设置定义在类 水平。

```
@Transactional(readOnly = true)
public class DefaultFooService implements FooService {

  public Foo getFoo(String fooName) {
    // do something
  }

  // these settings have precedence for this method
  @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
  public void updateFoo(Foo foo) {
    // do something
  }
}
```

transactional 设置

这个 transactional 注释是 元数据,指定一个接口、类或方法必须有 事务性语义;例如, 一个 启动一个品牌 新只读事务当调用该方法时,暂停任何 现有事务 一个 。 默认 transactional 设置为 如下:

- 传播设置是 PROPAGATION_REQUIRED。
- 隔离级别是 隔离违约。
- 事务是读/写。
- 事务超时默认的默认超时 潜在的交易系统,或任何人如果不会超时 支持。
- 任何 RuntimeException 触发 回滚,和任何检查 异常 不。

这些默认设置可以改变,各种属性的 这个 transactional 注释是 概括如下表:

为多12 3 transactional 属性

财产	类型	描述
价值	字符串	可选的限定指定事务管理器使用。
传播	枚举: 传播	可选的传播环境。
隔离	枚举: 隔离	可选的隔离级别。

readOnly	布尔	读/写和只读事务
超时	int(秒粒度)	事务超时。
将	数组的 类 对象,这 必须来自 可抛出。	可选的数组的异常类 **必须** 导致回滚。
rollbackForClassname	数组的类名。 类必须是来自 可抛出。	可选的数组的名字的异常类 **必须** 引起 回滚。
noRollbackFor	数组的 类 对象,这 必须来自 可抛出。	可选的数组的异常类 **不得** 导致回滚。
noRollbackForClassname	数组的 字符串 类名, 它必须是来自 可抛出。	可选的数组的名字的异常类 **不得** 引起 回滚。

目前你不能有明确的控制的名称 事务,“名字”意味着事务名称,将 显示在一个事务监视器,如果适用(例如,服务器的 事务监视器),并在日志输出。 声明性 事务、事务的名字总是完全限定的类 名+”。 “+方法 类的名称以事务方式建议。 例如,如果 handlePayment(. .) 方法 BusinessService 类开始一个事务, 事务的名称是: com.foo.BusinessService.handlePayment 。

多个事务经理与 transactional

大多数Spring应用程序只需要一个事务管理器,但可能存在的情况 在你想要的多个独立的事务管理器在一个单一的应用程序。 的值属性 transactional 注释可以 被用来选择指定的标识 PlatformTransactionManager 被使用。 这可以是bean名称或限定符值的事务管理器bean。 例如,使用限定符符号,下面的Java代码

```
public class TransactionalService {

  @Transactional("order")
  public void setSomething(String name) { ... }

  @Transactional("account")
  public void doSomething() { ... }
}
```

可以结合以下事务管理器bean声明在应用程序上下文。

```
<tx:annotation-driven/>

<bean id="transactionManager1" class="org.springframework.jdbc.DataSourceTransactionManager">
  ...
  <qualifier value="order"/>
</bean>

<bean id="transactionManager2" class="org.springframework.jdbc.DataSourceTransactionManager">
  ...
  <qualifier value="account"/>
</bean>
```

在这种情况下,两个方法 TransactionalService 将运行在另 事务管理器,来区分“秩序”和“账户”限定符。 默认 < tx:注解驱动的> 目标bean的名字 transactionManager 将 仍然被使用如果没有特别限定PlatformTransactionManager bean是发现。

自定义快捷注释

如果你发现你总是使用相同的属性 transactional 在许多不同的方法,那么春天的元注释支持允许您定义自定义快捷键 注释为您的特定的用例。 例如,定义以下注释

```
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Transactional("order")
public @interface OrderTx {
}

@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Transactional("account")
public @interface AccountTx {
}
```

让我们写前一节的例子一样

```
public class TransactionalService {

  @OrderTx
  public void setSomething(String name) { ... }

  @AccountTx
  public void doSomething() { ... }
}
```

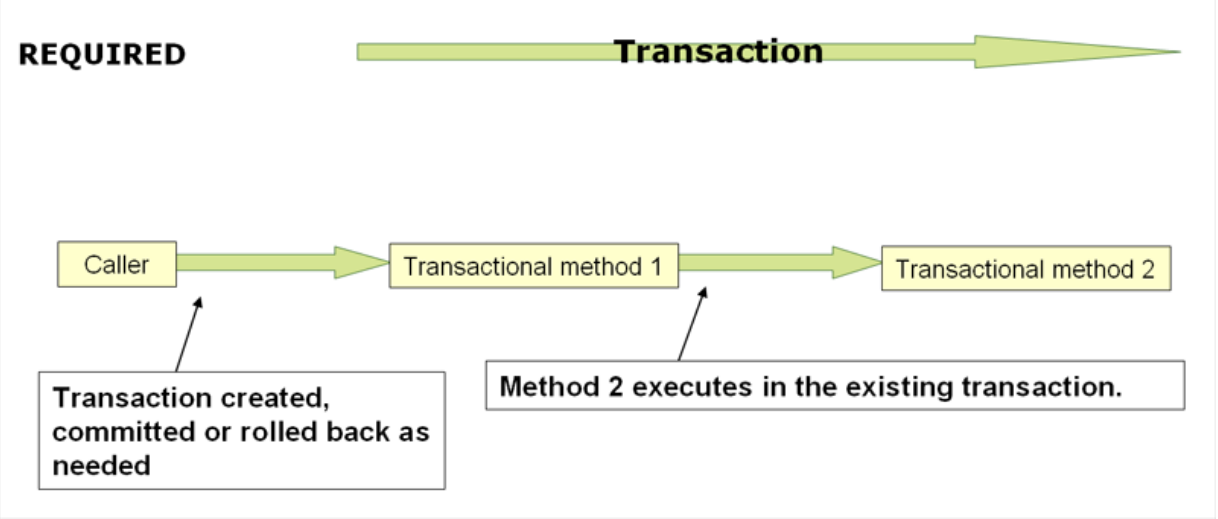
这里我们用语法来定义事务管理器限定符,但也可以 包括传播行为,回滚规则、超时等。

12.5.7A事务传播

本节描述一些语义的事务传播 在春天。 请注意,这部分不介绍 事务传播正确,而是细节一些语义 关于事务传播在春天。

在spring管理事务,注意之间的区别 物理 和 逻辑 事务,以及如何设置适用于此。传播 差异。

需要

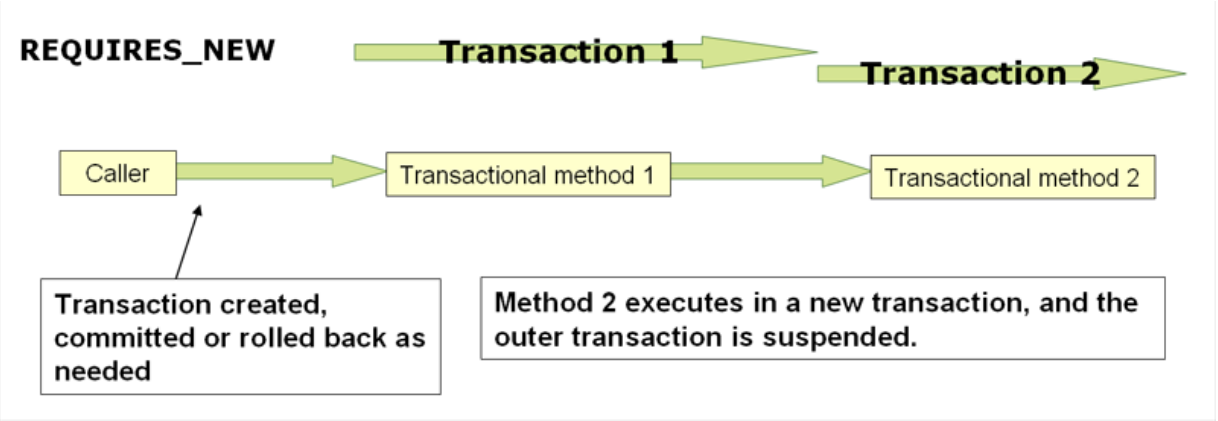


PROPAGATION_REQUIRED

当传播设置 PROPAGATION_REQUIRED ,一个 逻辑 创建事务范围为每个 方法应用在这些设置。 每一个这样的逻辑 事务范围可以确定只回滚状态分别, 与外部事务逻辑上独立的范围 内部事务范围。 当然,在案件的标准 PROPAGATION_REQUIRED 行为,所有这些范围 将被映射到同一个物理事务。 所以一个回滚只有 标志设置在内部事务范围并影响外 事务的机会实际上commit(正如你期望它 到)。

然而,如果一个内部事务范围设置 回滚只有标记,外层事务还没有决定 回滚本身,所以回滚(默默地引发的内部 事务范围)是意想不到的。 相应的 UnexpectedRollbackException 扔在那 点。 这是 预期行为 所以, 调用者的事务不能被误导的假设:一个承诺了,其实那是不。 所以如果一个内部事务(其中外部调用者不知道)标志着交易默默 只回滚,外部调用者还调用commit。 外部调用者 需要接收一个 UnexpectedRollbackException 来清楚地指出,执行回滚而。

RequiresNew



PROPAGATION_REQUIRES_NEW

PROPAGATION_REQUIRES_NEW 形成鲜明对比的是, PROPAGATION_REQUIRED ,使用一个 完全 独立事务对于每个 事务

范围的影响。 在这种情况下,底层物理 交易是不同的,因此可以提交或回滚 独立,与外部事务不受内部 事务的回滚状态。

嵌套

传播嵌套 使用 单 物理事务与多个 保存点,它可以回滚。 这样的部分回滚允许一个 事务范围内触发一个回滚 对其 范围 ,与外部交易能够继续 尽管一些操作的物理事务已经卷 回来。 这个设置通常映射到JDBC保存点,所以将 只使用JDBC资源交易。 看到春天的 DataSourceTransactionManager 。

12.5.8A建议事务操作

假设您想要执行 两 事务性 和 一些基本的分析建议。 如何 你这个上下文中的作用 < tx:注解驱动/ > 吗?

当你调用 updateFoo(Foo) 方法,你想看到下列行动:

1. 启动配置分析方面。
2. 事务的建议执行。
3. 方法建议对象执行。
4. 事务提交。
5. 分析方面的报道准确时间整个 事务性方法调用。

注意

这一章是不关心在任何伟大的解释AOP 细节(除非它适用于事务)。 看到 ChapterA 9, 面向方面的编程与弹簧 以下为详细报道AOP 配置和AOP一般。

这是一个简单的代码上面讨论分析方面。 这个 订购的建议是控制通过 下令 接口。 详细了解 建议订购,看到 一个章节 orderinga建议 。

```
package x.y;

import org.aspectj.lang.ProceedingJoinPoint;
import org.springframework.util.StopWatch;
import org.springframework.core.Ordered;

public class SimpleProfiler implements Ordered {

  private int order;

  // allows us to control the ordering of advice
  public int getOrder() {
    return this.order;
  }

  public void setOrder(int order) {
    this.order = order;
  }

  // this method *is* the around advice
  public Object profile(ProceedingJoinPoint call) throws Throwable {
    Object returnValue;
    StopWatch clock = new StopWatch(getClass().getName());
    try {
      clock.start(call.toShortString());
      returnValue = call.proceed();
    } finally {
      clock.stop();
      System.out.println(clock.prettyPrint());
    }
    return returnValue;
  }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
   http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans.xsd
   http://www.springframework.org/schema/tx
   http://www.springframework.org/schema/tx/spring-tx.xsd
   http://www.springframework.org/schema/aop
   http://www.springframework.org/schema/aop/spring-aop.xsd">

  <bean id="fooService" class="x.y.service.DefaultFooService"/>
```

```
<!-- this is the aspect -->
<bean id="profiler" class="x.y.SimpleProfiler">
  <!-- execute before the transactional advice (hence the lower order number) -->
  <property name="order" value="1"/>
</bean>

<tx:annotation-driven transaction-manager="txManager" order="200"/>

<aop:config>
  <!-- this advice will execute around the transactional advice -->
  <aop:aspect id="profilingAspect" ref="profiler">
    <aop:pointcut id="serviceMethodWithReturnValue"
          expression="execution(!void x.y..*Service.*(..))"/>
    <aop:around method="profile" pointcut-ref="serviceMethodWithReturnValue"/>
  </aop:aspect>
</aop:config>

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
  <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
  <property name="url" value="jdbc:oracle:thin:@rj-t42:1521:elvis"/>
  <property name="username" value="scott"/>
  <property name="password" value="tiger"/>
</bean>

<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"/>
</bean>

</beans>
```

以上配置的结果是一个 fooService 豆,剖析和事务 方面应用 按照顺序 。 你 配置任意数量的额外的方面以相似的方式。

下面的例子效果相同的设置如上,但使用 纯XML声明性方法。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
  http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/tx
  http://www.springframework.org/schema/tx/spring-tx.xsd
  http://www.springframework.org/schema/aop
  http://www.springframework.org/schema/aop/spring-aop.xsd">

  <bean id="fooService" class="x.y.service.DefaultFooService"/>

  <!-- the profiling advice -->
  <bean id="profiler" class="x.y.SimpleProfiler">
    <!-- execute before the transactional advice (hence the lower order number) -->
    <property name="order" value="1"/>
  </bean>

  <aop:config>

    <aop:pointcut id="entryPointMethod" expression="execution(* x.y..*Service.*(..))"/>

    <!-- will execute after the profiling advice (c.f. the order attribute) -->
    <aop:advisor
        advice-ref="txAdvice"
        pointcut-ref="entryPointMethod"
        order="2"/> <!-- order value is higher than the profiling aspect -->

    <aop:aspect id="profilingAspect" ref="profiler">
      <aop:pointcut id="serviceMethodWithReturnValue"
            expression="execution(!void x.y..*Service.*(..))"/>
      <aop:around method="profile" pointcut-ref="serviceMethodWithReturnValue"/>
    </aop:aspect>

  </aop:config>

  <tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
      <tx:method name="get*" read-only="true"/>
      <tx:method name="*"/>
    </tx:attributes>
  </tx:advice>

  <!-- other <bean/> definitions such as a DataSource and a PlatformTransactionManager here -->

</beans>
```

以上配置的结果将是一个 fooService 豆,剖析和事务 方面应用 这个顺序 。 如果你想要 配置建议执行 在 这个 事务的建议, 之前 这个 事务的建议,然后你只是交换的价值 bean的配置方面 秩序 属性,以便它 高于建议的秩序价值的事务。

你在类似的方式配置额外的方面。

12.5.9A使用 transactional 与 AspectJ

也可以使用Spring框架的 transactional 支持外的 Spring容器通过AspectJ方面。 这样做,你首先 注释您的类(和可选类的方法) transactional 注释,然后你 链接(织)应用程序的 org.springframework.transaction.aspectj.AnnotationTransactionAspect 定义在 弹簧方面jar 文件。 方面必须 还配置一个事务管理器。 你当然可以使用 Spring框架的IoC容器照顾了依赖注入 这个方面。 最简单的方法来配置事务管理 方面是使用 < tx:注解驱动/ > 元素和指定 模式 属性 AspectJ 中描述的 SectionA 12 5 6,一个使用 transactional 一个 。 因为我们的注意力是 在运行的应用程序的Spring容器外,我们将展示 你怎么做编程。

> **注意**
>
> 继续之前,您可能想读 SectionA 12 5 6,一个使用 transactional 一个 和 ChapterA 9, 面向方面的编程与弹簧 分别。

```
// construct an appropriate transaction manager
DataSourceTransactionManager txManager = new DataSourceTransactionManager(getDataSource());

// configure the AnnotationTransactionAspect to use it; this must be done before executing any transactional methods
AnnotationTransactionAspect.aspectOf().setTransactionManager(txManager);
```

> **注意**
>
> 当使用这方面,你必须标注 实现 类(和/或方法在这 类), 不 接口(如果有的话)的类 实现了。 AspectJ的规则, 遵循Java注解的接口 是 没有继承 。

这个 transactional 注释 类指定缺省的事务语义的执行 班上的任何方法。

这个 transactional 注释 方法在类中重写默认的事务语义 给定类的注释(如果存在)。 任何方法可能是注释, 不管能见度。

你的应用程序与编织 AnnotationTransactionAspect 你必须建立 你的应用程序使用AspectJ(请参阅 AspectJ 开发指南)或使用加载时编织。 看到 SectionA 9 8 4,一个装入时编织与AspectJ在春季Frameworka 讨论与加载时编织 AspectJ。

12.6一个程序性事务管理

Spring框架提供了两种方法的程序性事务 管理:

- 使用 TransactionTemplate 。
- 使用 PlatformTransactionManager 直接实现。

春季团队一般建议 TransactionTemplate 对于程序性事务 管理。 第二种方法是类似于使用JTA UserTransaction API,尽管例外 处理还是少了一些麻烦。

12.6.1A使用 TransactionTemplate

这个 TransactionTemplate 采用相同的 方法作为其他弹簧 模板 如 JdbcTemplate 。 它使用一个回调方法, 免费的应用程序代码不必进行采集和样板 释放事务资源,导致代码 目的驱动的,在这写的代码仅仅关注 开发者想要做什么。

> **注意**
>
> 您将看到以下的例子中,使用 TransactionTemplate 绝对夫妻你 春天的事务基础设施和api。 是否 编程式事务管理是适合你的发展 需要的是一个决定,你将不得不让自己。

应用程序必须执行的代码在一个事务上下文,和 这将使用 TransactionTemplate 明确地, 看起来像下面的。 你,作为应用程序开发人员,写一个 TransactionCallback 实现 (通常表示为一个匿名内部类),其中包含代码 那你需要执行事务的上下文中。 然后通过 您的自定义的一个实例 TransactionCallback 到 执行(. .) 方法暴露在 TransactionTemplate 。

```
public class SimpleService implements Service {

  // single TransactionTemplate shared amongst all methods in this instance
  private final TransactionTemplate transactionTemplate;

  // use constructor-injection to supply the PlatformTransactionManager
  public SimpleService(PlatformTransactionManager transactionManager) {
```

```
    Assert.notNull(transactionManager, "The 'transactionManager' argument must not be null.");
    this.transactionTemplate = new TransactionTemplate(transactionManager);
  }

  public Object someServiceMethod() {
    return transactionTemplate.execute(new TransactionCallback() {

      // the code in this method executes in a transactional context
      public Object doInTransaction(TransactionStatus status) {
        updateOperation1();
        return resultOfUpdateOperation2();
      }
    });
  }
}
```

如果没有返回值,使用方便 TransactionCallbackWithoutResult 类和 匿名类如下:

```
transactionTemplate.execute(new TransactionCallbackWithoutResult() {

  protected void doInTransactionWithoutResult(TransactionStatus status) {
    updateOperation1();
    updateOperation2();
  }
});
```

代码在回调可以回滚事务通过调用 这个 setRollbackOnly() 方法对提供的 TransactionStatus 对象:

```
transactionTemplate.execute(new TransactionCallbackWithoutResult() {

  protected void doInTransactionWithoutResult(TransactionStatus status) {
    try {
      updateOperation1();
      updateOperation2();
    } catch (SomeBusinessExeption ex) {
      status.setRollbackOnly();
    }
  }
});
```

指定事务设置

你可以指定事务设置,比如传播 模式下,隔离级别、超时等等的 TransactionTemplate 通过编程方式和 在配置。 TransactionTemplate 实例 默认情况下有 默认 事务设置 。 下面的例子显示了 编程定制的事务设置 特定 TransactionTemplate:

```
public class SimpleService implements Service {

  private final TransactionTemplate transactionTemplate;

  public SimpleService(PlatformTransactionManager transactionManager) {
    Assert.notNull(transactionManager, "The 'transactionManager' argument must not be null.");
    this.transactionTemplate = new TransactionTemplate(transactionManager);

    // the transaction settings can be set here explicitly if so desired
    this.transactionTemplate.setIsolationLevel(TransactionDefinition.ISOLATION_READ_UNCOMMITTED);
    this.transactionTemplate.setTimeout(30); // 30 seconds
    // and so forth...
  }
}
```

下面的例子定义了一个 TransactionTemplate 和一些自定义 事务设置,使用Spring XML配置。 这个 sharedTransactionTemplate 然后可以注入 许多服务是必需的。

```
<bean id="sharedTransactionTemplate"
    class="org.springframework.transaction.support.TransactionTemplate">
  <property name="isolationLevelName" value="ISOLATION_READ_UNCOMMITTED"/>
  <property name="timeout" value="30"/>
</bean>"
```

最后,实例的 TransactionTemplate 类是线程安全的,在那 实例不维护任何会话状态。 TransactionTemplate 实例 做 然而维护配置状态,所以当一个 类的数量可能共享一个单一的实例 TransactionTemplate ,如果一个类需要使用 TransactionTemplate 与不同的设置(为 例子,一个不同的隔离级别),那么你需要创建两个 截然不同的 TransactionTemplate 实例。

12.6.2A使用 PlatformTransactionManager

你也可以使用 org.springframework.transaction.PlatformTransactionManager 直接管理你的交易。 简单地通过实施 这个 PlatformTransactionManager 你是 使用你的bean通过bean的引用。 然后,使用 TransactionDefinition 和 TransactionStatus 对象可以 启动事务,回滚,并提交。

```
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
// explicitly setting the transaction name is something that can only be done programmatically
def.setName("SomeTxName");
def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);

TransactionStatus status = txManager.getTransaction(def);
try {
  // execute your business logic here
}
catch (MyException ex) {
  txManager.rollback(status);
  throw ex;
}
txManager.commit(status);
```

12.7一个选择编程和声明式事务 管理

编程式事务管理通常是一个好主意只有 你有一个小数量的事务操作。 例如,如果你 有一个web应用程序,需要事务只针对某个更新吗 操作,您可能不希望使用Spring事务代理设置 或任何其他技术。 在这种情况下,使用 TransactionTemplate 可能 是一个好的方法。 能够设置事务名称明确地也 一些只能用编程的方法 事务管理。

另一方面,如果您的应用程序有许多事务 操作、声明式事务管理通常是值得的。 它 保持事务管理的业务逻辑,并不是困难的 配置。 当使用Spring框架,而不是EJB CMT, 声明式事务管理配置的成本是很大的 减少了。

12.8一个特定于应用服务器的集成

春天的事务抽象通常是应用程序服务器 不可知论者。 此外,春天的 JtaTransactionManager 类,它可以选择 执行JNDI查找JTA UserTransaction 和 transactionManager 对象,自动指定 位置对后者的对象,不同的应用程序服务器。 有 访问JTA transactionManager 允许 为增强的事务语义,特别是支持事务 悬架。 看到 JtaTransactionManager JavaDocs 详情。

春天的 JtaTransactionManager 是 标准的选择上运行的Java EE应用服务器,是众所周知的 所有常见的服务器上工作。 先进的功能,如事务 悬挂在许多服务器工作——包括GlassFish、JBoss, Geronimo,甲骨文OC4J——没有任何特殊配置要求。 然而,对于完全支持事务悬挂和进一步的先进 集成、弹簧船舶特殊适配器为IBM WebSphere,BEA WebLogic 服务器和OC4J甲骨文。 这些适配器进行以下 部分。

对于标准的场景,包括WebLogic服务器, WebSphere和OC4J,考虑使用方便 < tx:jta事务管理器/ > 配置 元素。 在配置时,该元素自动检测 底层服务器并选择最好的事务管理器可用 为平台。 这意味着您不需要配置 特定的适配器类(如在以下部分中讨论)明确;相反,它们是自动选择,标准 JtaTransactionManager 作为默认的回退。

12.8.1A IBM WebSphere

在WebSphere 6 1 0 9以上,推荐的弹簧JTA 事务管理器来使用 WebSphereUowTransactionManager 。 这种特殊的 适配器利用IBM的 UOWManager API, 这是可在WebSphere应用程序服务器6 0 2 19,后来呢 和6 1 0 9和以后。 用这个适配器,台弹力事务 悬挂(暂停/恢复作为发起 PROPAGATION_REQUIRES_NEW)是由官方支持 IBM！

12.8.2A BEA WebLogic服务器

在WebLogic Server 9.0或以上,您通常会使用 WebLogicJtaTransactionManager 而不是 股票 JtaTransactionManager 类。 这种特殊的 特定于weblogic的子类的正常 JtaTransactionManager 支持全部的能量 春天的事务定义在weblogic管理事务 环境,超越标准JTA事务语义:功能包括 名字,事务隔离级别,和适当的恢复 交易在所有的情况下。

OC4J 12.8.3A甲骨文

春天一个特殊的适配器类船只OC4J 10 1 3或更高 称为 OC4JJtaTransactionManager 。 这个类是 类似于 WebLogicJtaTransactionManager 类在前一节中讨论的,提供类似的增值上 OC4J:事务名称和事务隔离级别。

完整的JTA功能,包括交易暂停, 与春天的正常工作 JtaTransactionManager 在 OC4J为好。 特殊的 OC4JJtaTransactionManager 适配器只是提供 超出标准的JTA的增值。

12.9一个解决常见的问题

12.9.1A使用错误的事务管理器为一个特定的 数据源

使用 正确 PlatformTransactionManager 实现 根据您选择的事务技术和需求。 使用得当,Spring框架仅仅提供了一个简单的和便携式抽象。 如果您使用的是全局事务,你 必须 使用 org.springframework.transaction.jta.JtaTransactionManager 类(或一个 应用 特定于服务器的子类 它为所有你的事务。 操作。 否则事务基础设施试图执行 本地事务容器等资源 数据源 实例。 这样的地方 交易没有意义,和一个好的应用程序服务器对待 他们是错误的。

12.10进一步资源

为更多的信息关于Spring框架的事务 支持:

- 分布式 交易在春天,有和没有XA 是JavaWorld 演讲中,SpringSource的大卫Syer指导您通过 七个模式在春天分布式事务的应用程序, 他们三个和四个没有使用XA。
- Java 事务设计策略 是一本可以从吗 InfoQ 这提供了一种有节奏的 介绍事务在Java。 它还包括并排 的例子,如何配置和使用事务和春天 框架和EJB3。

13。 一个DAO支持

13.1一个介绍

数据访问对象(DAO)支持在春天是旨在使它 易于使用的数据访问技术像JDBC、JPA或Hibernate, JDO以一致的方式。 这允许一个切换的 提到的持久性技术相当容易,它也允许 一个代码而不用担心捕获异常,是特定的 每个技术。

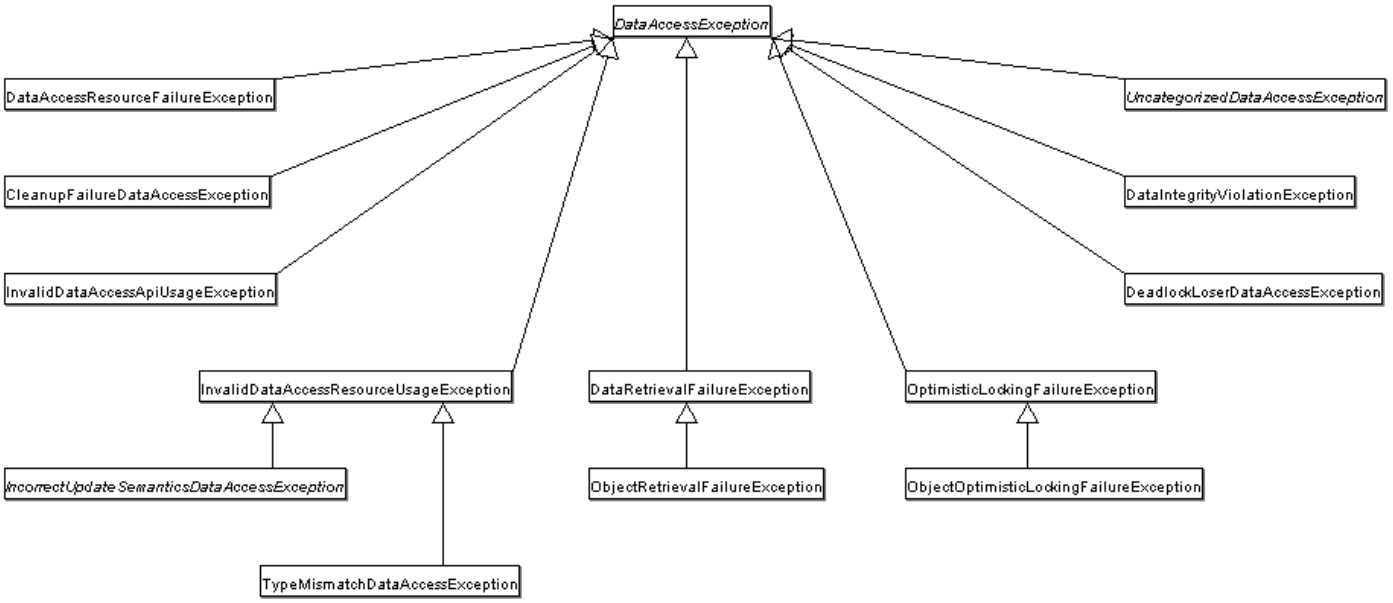
13.2一个一致的异常层次结构

Spring提供了一个方便的翻译从特定于技术的 异常喜欢 SQLException 到自己的异常 类层次结构的 DataAccessException 随着 根异常。 这些异常包装原始异常所以有 从来没有任何风险,你可能失去任何信息,可能 出了差错。

除了JDBC例外,春天也可以包装 hibernate具体的异常,将他们从专有、检查 异常(对于版本3.0之前的Hibernate Hibernate), 一套集中运行时异常(这同样适用于JDO和JPA 例外)。 这允许一个处理最持久的异常,这 是不可恢复的,只有在适当的层,而不用吗 恼人的样板catch和throw街区和异常声明 一个的dao。 (一个陷阱和处理异常仍可以在任何地方一个需求 来虽然。) 正如上面提到的,JDBC异常(包括 数据库方言)也转换为相同的层次结构, 这意味着一个可以执行一些操作与JDBC在一致的 编程模型。

上述适用于各种模板类在温泉 支持各种ORM框架。 如果一个人使用拦截器的基础 类然后应用程序必须关心处理 HibernateExceptions 和 JDOExceptions 本身,最好是通过委托给 SessionFactoryUtils ” convertHibernateAccessException(. .) 或 convertJdoAccessException() 分别的方法。 这些方法将例外是兼容的 异常的 org.springframework.dao 异常 层次结构。 作为 JDOExceptions 不加以制止,他们可以吗 也只会抛出,牺牲泛型DAO的抽象的方面 例外虽然。

异常层次结构,弹簧提供如下图所示。 (请注意,类层次结构的详细图像显示只有一个 子集的整个 DataAccessException 层次结构)。



13.3一个注释用于配置刀或存储库类

最好的方式来保证你的数据访问对象(dao)或 存储库提供例外翻译是使用 @Repository 注释。 这个注释 还允许组件扫描支持找到和配置您的dao 和仓库,不需要提供XML配置条目 他们。

```
@Repository
public class SomeMovieFinder implements MovieFinder {

    // ...

}
```

任何刀或存储库实现将需要访问 持久性资源,根据持久性技术使用;为 例子,一个基于JDBC的存储库将需要访问JDBC 数据源 ,一个jpa基础库需要 访问一个 EntityManager 。 最简单的方法 为了完成这个是有这个资源依赖注射使用 这个 @ autowired、 , @ inject , @ resource 或 persistencecontext 注释。 这是一个 例子为一个JPA存储库:

```
@Repository
public class JpaMovieFinder implements MovieFinder {

    @PersistenceContext
    private EntityManager entityManager;

    // ...

}
```

如果您使用的是经典的Hibernate api比你可以注入 SessionFactory:

```
@Repository
public class HibernateMovieFinder implements MovieFinder {

    private SessionFactory sessionFactory;

    @Autowired
    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    // ...

}
```

最后的示例中,我们将展示这是典型的JDBC支持。 你 会有 数据源 注入 初始化方法,您将创建一个 JdbcTemplate 和其他数据访问支持类 像 SimpleJdbcCall 等使用这种 数据源 。

```
@Repository
public class JdbcMovieFinder implements MovieFinder {

    private JdbcTemplate jdbcTemplate;

    @Autowired
    public void init(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }
```

```
    // ...
}
```

注意

请见具体覆盖每个持久性技术 有关如何配置应用程序上下文采取 利用这些注释。

14。 一个数据访问与JDBC

14.1一个介绍Spring框架JDBC

所提供的增值Spring框架JDBC抽象 也许最好操作的顺序显示在下表中列出。 该表显示了春天什么行动会照顾和哪些动作 是你的责任,应用程序开发人员。

14.1为多。 一个Spring JDBC -谁做什么?

行动	春天	你
定义连接参数。		x
打开连接。	x	
指定SQL语句。		x
声明参数并提供参数值		x
准备和执行该语句。	x	
设置循环来遍历结果(如果 任何)。	x	
做这个工作对于每个迭代。		x
处理任何例外。	x	
处理事务。	x	
关闭连接、语句、结果集。	x	

Spring框架负责的所有底层细节 可以使JDBC API开发这些乏味的和。

选择一个14.1.1A JDBC数据库访问方法

你可以选择几种方法之间形成的基础 JDBC数据库访问。 除了三种口味的JdbcTemplate, 一个新的SimpleJdbcInsert和SimplejdbcCall方法优化数据库 元数据和对象的RDBMS需要更多的面向对象的风格 方法相似,JDO查询设计。 一旦你开始使用一个 这些方法,你仍然可以混合和匹配,包括一个功能 从一个不同的方法。 所有的方法都需要一个JDBC 2.0兼容的 司机,和一些高级功能需要一个JDBC 3.0驱动程序。

注意

Spring 3.0更新以下所有方法使用Java 5 支持如泛型和可变参数。

- **JdbcTemplate** 是典型的 Spring JDBC方法和最受欢迎的。 这个“最低水平” 方法和所有其他人使用JdbcTemplate在后台,和所有 更新Java 5泛型和可变参数等的支持。
- **NamedParameterJdbcTemplate** 包装 JdbcTemplate 提供命名参数 而不是传统的JDBC”？“占位符。 这种方法 提供了更好的文档和易用性当你有多重 参数的SQL语句。
- **SimpleJdbcInsert和 SimpleJdbcCall** 优化数据库元数据限制 数量的必要的配置。 这种方法简化了编码 所以,你只需要提供的表的名称或过程 并提供参数的映射匹配列名称。 这仅适用于如果数据库提供足够的元数据。 如果 数据库不提供此元数据,您将必须提供 显式配置的参数。
- **RDBMS对象包括MappingSqlQuery, SqlUpdate和StoredProcedure** 要求您创建 可重用和线程安全对象初始化期间

您的数据 访问层。 这种方法是仿照JDO查询在你 定义你的查询字符串,声明参数和编译查询。 一旦你这样做,执行方法可以多次调用与 各种参数值传递。

14.1.2A包的层次结构

Spring框架的JDBC抽象框架包括四个 不同的包,即 核心 , 数据源 , 对象 ,和 支持 。

这个 org.springframework.jdbc.core 包 包含 JdbcTemplate 类和它的各种 回调接口,加上各种相关类。 一个分包 命名 org.springframework.jdbc.core.simple 包含 这个 SimpleJdbcInsert 和 SimpleJdbcCall 类。 另一个分包命名 org.springframework.jdbc.core.namedparam 包含 NamedParameterJdbcTemplate 类和相关 支持类。 看到 SectionA 14.2,一个使用JDBC核心类来控制基本JDBC加工 误差handlinga , SectionA 14.4,一个operationsaJDBC批处理 ,和 SectionA 14.5,一个简化JDBC操作与SimpleJdbc classesa

这个 org.springframework.jdbc.datasource 包 包含一个工具类,用于容易 数据源 访问,和各种简单的 数据源 实现,可以 用于测试和运行未修改的JDBC代码之外的Java EE 集装箱。 一个分包命名 org.springfamework.jdbc.datasource.embedded 提供 支持创建内存数据库实例使用Java数据库 引擎如HSQL和H2。 看到 SectionA 14.3,一个connectionsa控制数据库 和 SectionA 14.8,一个supporta嵌入式数据库

这个 org.springframework.jdbc.object 包 包含类,代表RDBMS的查询、更新和存储 程序是线程安全的,可重用的对象。 看到 SectionA 14.6,一个建模JDBC操作作为Java objectsa 。 这种方法是通过JDO建模,尽管 课程对象返回的查询 一个 断开 一个 从 数据库。 这种更高层次的JDBC抽象取决于 较低级别的抽象的 org.springframework.jdbc.core 包。

这个 org.springframework.jdbc.support 包提供 SQLException 翻译功能和一些 实用工具类。 期间抛出的异常处理JDBC。 翻译 对异常的定义 org.springframework.dao 包。 这意味着代码使用Spring JDBC抽象层 不需要实现JDBC或rdbms特有的错误处理。 所有 例外是不翻译,给你选择 捕获异常,您可以恢复,同时允许其他 例外的传递给调用者。 看到 SectionA 14 2 3,一个 SQLExceptionTranslator 一个 。

使用JDBC 14.2核心类来控制基本JDBC加工 错误处理

14.2.1A JdbcTemplate

这个 JdbcTemplate 类是中央类 在JDBC核心包。 它处理创建和释放 资源,帮助您避免一些常见错误,如忘记 关闭连接。 它执行的基本任务核心JDBC 工作流如语句创建和执行,使应用程序 代码提供的SQL和提取结果。 这个 JdbcTemplate 类执行SQL查询,更新 语句和存储过程调用,执行迭代 ResultSet 年代和提取返回 参数值。 它还捕获异常并将它们转换为JDBC通用,更多信息,定义的异常层次结构 org.springframework.dao 包。

当你使用 JdbcTemplate 对于你的 代码,你只需要实现回调接口,给他们一个 明确定义的合同。 这个 PreparedStatementCreator 回调 接口创建一份事先准备好的声明中给定一个 连接 提供的这个类, 提供SQL和任何必要的参数。 也同样如此 CallableStatementCreator 接口,它 创建可调用语句。 这个 RowCallbackHandler 接口提取 从每一行的值 ResultSet 。

这个 JdbcTemplate 可以用在一刀吗 实现通过直接实例化与 数据源 参考,或者被配置在 一个Spring IoC容器和给DAOs作为bean引用。

> **注意**
>
> 这个 数据源 总是应该 配置为一个bean在Spring IoC容器。 在第一种情况下 bean是直接交给服务;在第二种情况下它是 给有准备的模板。

所有SQL签发这类记录的 调试 水平属于对应 完全限定类名的模板实例(一般 JdbcTemplate ,但它可能是不同的,如果你是 使用一个自定义的子类 JdbcTemplate 类)。

JdbcTemplate类的例子使用

本节提供了一些示例 JdbcTemplate 类的使用。 这些例子是 不是一个详尽的清单的所有功能暴露出来的 JdbcTemplate ,看到服务员的Javadocs 那。

查询(选择)

这是一个简单的查询获取的行数 关系:

```
int rowCount = this.jdbcTemplate.queryForInt("select count(*) from t_actor");
```

一个简单的查询使用绑定变量:

```
int countOfActorsNamedJoe = this.jdbcTemplate.queryForInt(
        "select count(*) from t_actor where first_name = ?", "Joe");
```

查询 字符串：

```
String lastName = this.jdbcTemplate.queryForObject(
        "select last_name from t_actor where id = ?",
        new Object[]{1212L}, String.class);
```

查询和填充 单 域 对象:

```
Actor actor = this.jdbcTemplate.queryForObject(
        "select first_name, last_name from t_actor where id = ?",
        new Object[]{1212L},
        new RowMapper<Actor>() {
            public Actor mapRow(ResultSet rs, int rowNum) throws SQLException {
                Actor actor = new Actor();
                actor.setFirstName(rs.getString("first_name"));
                actor.setLastName(rs.getString("last_name"));
                return actor;
            }
        });
```

查询和填充一个数量的域对象:

```
List<Actor> actors = this.jdbcTemplate.query(
        "select first_name, last_name from t_actor",
        new RowMapper<Actor>() {
            public Actor mapRow(ResultSet rs, int rowNum) throws SQLException {
                Actor actor = new Actor();
                actor.setFirstName(rs.getString("first_name"));
                actor.setLastName(rs.getString("last_name"));
                return actor;
            }
        });
```

如果最后两段代码实际上存在于相同的 应用程序,它会有道理删除重复呈现 在这两个 RowMapper 匿名内部 类,并将其提取到一个类(通常是一个 静态 内部类),然后可以引用 根据需要由DAO方法。 例如,它可能是更好的编写 最后的代码片段如下:

```
public List<Actor> findAllActors() {
    return this.jdbcTemplate.query( "select first_name, last_name from t_actor", new ActorMapper());
}

private static final class ActorMapper implements RowMapper<Actor> {

    public Actor mapRow(ResultSet rs, int rowNum) throws SQLException {
        Actor actor = new Actor();
        actor.setFirstName(rs.getString("first_name"));
        actor.setLastName(rs.getString("last_name"));
        return actor;
    }
}
```

更新(插入/更新/删除)与jdbcTemplate

你使用 更新(. .) 方法 执行插入、更新和删除操作。 参数值 通常提供的var args或者作为一个对象 数组。

```
this.jdbcTemplate.update(
        "insert into t_actor (first_name, last_name) values (?, ?)",
        "Leonor", "Watling");
```

```
this.jdbcTemplate.update(
        "update t_actor set = ? where id = ?",
        "Banjo", 5276L);
```

```
this.jdbcTemplate.update(
        "delete from actor where id = ?",
        Long.valueOf(actorId));
```

其他jdbcTemplate操作

您可以使用 执行(. .) 方法 执行任意SQL,因此该方法通常用于 DDL语句。 这是严重超载与变体以 回调接口、绑定变量数组,等

等。

```
this.jdbcTemplate.execute("create table mytable (id integer, name varchar(100))");
```

下面的示例调用一个简单的存储过程。 更多的 复杂的存储过程支持 覆盖后 。

```
this.jdbcTemplate.update(
        "call SUPPORT.REFRESH_ACTORS_SUMMARY(?)",
        Long.valueOf(unionId));
```

JdbcTemplate 最佳实践

实例的 JdbcTemplate 类 一旦配置线程安全的 。 这是重要的 因为它意味着您可以配置的单独一个实例 JdbcTemplate 然后安全地注入这 共享 引用到多个dao(或 存储库)。 这个 JdbcTemplate 是有状态的,在 ,它维护一个引用 数据源 ,但这个状态 不会话状态。

一个常见的做法在使用 JdbcTemplate 类(和相关的 NamedParameterJdbcTemplate 类)是配置 数据源 在Spring的配置文件, 然后依赖注入, 共享 数据源 bean到你的刀 类; JdbcTemplate 会创建在 setter的 数据源 。 这导致 对DAOs看起来部分如下:

```
public class JdbcCorporateEventDao implements CorporateEventDao {

    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    // JDBC-backed implementations of the methods on the CorporateEventDao follow...
}
```

相应的配置可能看起来像这样。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <bean id="corporateEventDao" class="com.example.JdbcCorporateEventDao">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
        <property name="driverClassName" value="${jdbc.driverClassName}"/>
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>

    <context:property-placeholder location="jdbc.properties"/>

</beans>
```

显式配置的另一种选择是使用 组件扫描和注释支持依赖注入。 在 这种情况下你注释类的 @Repository (这使它一个候选人 组件扫描)和注释 数据源 setter方法 @ autowired 。

```
@Repository
public class JdbcCorporateEventDao implements CorporateEventDao {

    private JdbcTemplate jdbcTemplate;

    @Autowired
    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    // JDBC-backed implementations of the methods on the CorporateEventDao follow...
}
```

相应的XML配置文件 看起来像下面的:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <!-- Scans within the base package of the application for @Components to configure as beans -->
    <context:component-scan base-package="org.springframework.docs.test" />

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
        <property name="driverClassName" value="${jdbc.driverClassName}"/>
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>

    <context:property-placeholder location="jdbc.properties"/>

</beans>
```

如果您使用的是春天的 JdbcDaoSupport 类,和你的不同 jdbc支持DAO类扩展从它,然后你的子类继承了一个 setDataSource(..) 方法从 JdbcDaoSupport 类。 你 可以选择是否从这个类继承。 这个 JdbcDaoSupport 类提供一个 只是方便使用。

无论上述模板初始化风格 你选择使用(或不),很少需要创建一个新的 实例 JdbcTemplate 类每一次你 要执行的SQL。 一旦配置完成, JdbcTemplate 实例是线程安全的。 你可能 要多 JdbcTemplate 实例如果你 应用程序访问多个数据库,这就需要多个 数据源 ,随后多个 不同配置 JdbcTemplates 。

14.2.2A NamedParameterJdbcTemplate

这个 NamedParameterJdbcTemplate 类添加了 支持编程JDBC语句使用命名参数,如 反对编程JDBC语句只使用经典的占位符(' 吗?")参数。 这个 NamedParameterJdbcTemplate 类封装了一个 JdbcTemplate ,代表对包装的 JdbcTemplate 做了大量的工作。 本节 只有那些区域的描述 NamedParameterJdbcTemplate 类,不同于 这个 JdbcTemplate 本身,即编程JDBC 语句使用命名参数。

```
// some JDBC-backed DAO class...
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

public void setDataSource(DataSource dataSource) {
    this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);
}

public int countOfActorsByFirstName(String firstName) {

    String sql = "select count(*) from T_ACTOR where first_name = :first_name";

    SqlParameterSource namedParameters = new MapSqlParameterSource("first_name", firstName);

    return namedParameterJdbcTemplate.queryForInt(sql, namedParameters);
}
```

注意使用命名参数符号价值 分配给 SQL 变量和相应的 插入的值 namedParameters 类型的变量(MapSqlParameterSource)。

或者,你可以通过命名参数和他们的 相应值 NamedParameterJdbcTemplate 实例通过使用 地图 欧式风格的 剩余的方法公开 NamedParameterJdbcOperations 和 实现的 NamedParameterJdbcTemplate 类也遵循类似的模式,这里不介绍。

下面的示例展示了使用 地图 的风格。

```
// some JDBC-backed DAO class...
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

public void setDataSource(DataSource dataSource) {
    this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);
}

public int countOfActorsByFirstName(String firstName) {

    String sql = "select count(*) from T_ACTOR where first_name = :first_name";

    Map namedParameters = Collections.singletonMap("first_name", firstName);

    return this.namedParameterJdbcTemplate.queryForInt(sql, namedParameters);
}
```

一个不错的特点有关 NamedParameterJdbcTemplate (和现有的 相同的Java包) SqlParameterSource 接口。 你已经看到一个例子,一个实现这个 接口在前面的代码片段(MapSqlParameterSource 类)。 一个 SqlParameterSource 是一个来源的 命

名参数值 NamedParameterJdbcTemplate 。 这个 MapSqlParameterSource 类是一个非常简单的 实现是一个简单的适配器周围 java util地图 ,那里的钥匙的 参数名称和值的参数值。

另一个 SqlParameterSource 实现是 BeanPropertySqlParameterSource 类。 这类 包装一个任意的JavaBean(即,一个类的实例 坚持 JavaBean 约定),并使用JavaBean的属性作为包装 命名参数值的来源。

```
public class Actor {

    private Long id;
    private String firstName;
    private String lastName;

    public String getFirstName() {
        return this.firstName;
    }

    public String getLastName() {
        return this.lastName;
    }

    public Long getId() {
        return this.id;
    }

    // setters omitted...

}
```

```
// some JDBC-backed DAO class...
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

public void setDataSource(DataSource dataSource) {
    this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);
}

public int countOfActors(Actor exampleActor) {

    // notice how the named parameters match the properties of the above 'Actor' class
    String sql =
        "select count(*) from T_ACTOR where first_name = :firstName and last_name = :lastName";

    SqlParameterSource namedParameters = new BeanPropertySqlParameterSource(exampleActor);

    return this.namedParameterJdbcTemplate.queryForInt(sql, namedParameters);
}
```

请记住, NamedParameterJdbcTemplate 类 包装 一个经典的 JdbcTemplate 模板;如果你需要访问包裹 JdbcTemplate 实例来访问功能 仅出现在 JdbcTemplate 类,你可以 使用 getJdbcOperations() 方法来访问 包装 JdbcTemplate 通过 JdbcOperations 接口。

看到也 一个章节 JdbcTemplate 最佳practicesa 对于 使用指南的 NamedParameterJdbcTemplate 类在上下文 一个应用程序。

14.2.3A SQLExceptionTranslator

SQLExceptionTranslator 是一个 接口的实现类,它可以转换 SQLExceptions 和春天的 org.springframework.dao.DataAccessException , 这是不可知论者关于数据访问策略。 实现可以 是通用的(例如,使用SQLState编码JDBC)或专有的 (例如,使用Oracle错误代码),为更精确。

SQLErrorCodeSQLExceptionTranslator 是 实施 SQLExceptionTranslator 这是默认情况下使用的。 这个实现使用特定的供应商代码。 它是更为精准 SQLState 实现。 错误代码翻译是基于代码保存在JavaBean类型 类称为 SQLErrorCodes 。 这类被创建 和填充的 SQLErrorCodesFactory 作为 顾名思义是一个工厂来创建 SQLErrorCodes 基于内容的 配置文件命名 sql错误代码的xml 。 这个文件是 填充供应商编码和基于 DatabaseProductName 来自 该方法 。 对于实际的代码 你正在使用的数据库使用。

这个 SQLErrorCodeSQLExceptionTranslator 适用于以下序列匹配规则:

注意

这个 SQLErrorCodesFactory 被 默认定义错误代码和自定义异常的翻译。 他们是在一个文件夹命名为抬头 sql错误代码的xml 从类路径和 匹配的 SQLErrorCodes 实例 基于数据库名称位于从数据库的元数据 数据库在使用。

1. 任何自定义翻译实现子类。 通常 提供的混凝土 SQLErrorCodeSQLExceptionTranslator 使用 所以这个规则不适用。 它只适用于如果您有其实 提供了一个子类实现。
2. 任何自定义实现的 SQLExceptionTranslator 接口,是 提供作为 customSqlExceptionTranslator 财产的 这个 SQLErrorCodes 类。
3. 的实例的列表 CustomSQLErrorCodesTranslation 类, 提供 customTranslations 财产的 SQLErrorCodes 类,是 寻找一个匹配。
4. 错误代码匹配是应用。
5. 使用回退的翻译。 SQLExceptionSubclassTranslator 是 默认的回退的翻译。 如果这个翻译不是可用的 然后接下来的回退的翻译是 SQLStateSQLExceptionTranslator 。

您可以扩展 SQLErrorCodeSQLExceptionTranslator:

```
public class CustomSQLErrorCodesTranslator extends SQLErrorCodeSQLExceptionTranslator {

    protected DataAccessException customTranslate(String task, String sql, SQLException sqlex) {
        if (sqlex.getErrorCode() == -12345) {
            return new DeadlockLoserDataAccessException(task, sqlex);
        }
        return null;
    }
}
```

在这个例子中,特定的错误代码 -12345年 是翻译和其他错误被翻译成默认吗 翻译的实现。 使用这个自定义翻译,它是 要通过它来了 JdbcTemplate 通过 该方法 setExceptionTranslator 和使用这 JdbcTemplate 对于所有的数据访问 处理这个翻译是必要的。 这里是一个例子,说明 这个自定义翻译可以使用:

```
private JdbcTemplate jdbcTemplate;

public void setDataSource(DataSource dataSource) {
    // create a JdbcTemplate and set data source
    this.jdbcTemplate = new JdbcTemplate();
    this.jdbcTemplate.setDataSource(dataSource);
    // create a custom translator and set the DataSource for the default translation lookup
    CustomSQLErrorCodesTranslator tr = new CustomSQLErrorCodesTranslator();
    tr.setDataSource(dataSource);
    this.jdbcTemplate.setExceptionTranslator(tr);
}

public void updateShippingCharge(long orderId, long pct) {
    // use the prepared JdbcTemplate for this update
    this.jdbcTemplate.update(
        "update orders" +
            " set shipping_charge = shipping_charge * ? / 100" +
            " where id = ?"
        pct, orderId);
}
```

自定义翻译传递一个数据源,以便查找 错误代码在 sql错误代码的xml 。

14.2.4A执行语句

执行一条SQL语句只需要很少的代码。 你需要一个 数据源 和一个 JdbcTemplate ,包括方便 方法 所提供的 JdbcTemplate 。 这个 下面的例子显示了您需要包括对最小但完全 功能类,创建一个新表:

```
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

public class ExecuteAStatement {

    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    public void doExecute() {
        this.jdbcTemplate.execute("create table mytable (id integer, name varchar(100))");
    }
}
```

14.2.5A运行查询

一些查询方法返回一个值。 检索一个计数或 特定的值从一行,使用 queryForInt(. .) , queryForLong(. .) 或 queryForObject(. .) 。 后者将 返回JDBC 类型 Java类 作为参数传入。 如果类型转换是无效的,那么一个 InvalidDataAccessApiUsageException

是 扔。 下面是一个示例,它包含两个查询方法(一个 int 和一个查询 字符串 。

```
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

public class RunAQuery {

    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    public int getCount() {
        return this.jdbcTemplate.queryForInt("select count(*) from mytable");
    }

    public String getName() {
        return (String) this.jdbcTemplate.queryForObject("select name from mytable", String.class);
    }

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
}
```

除了单个结果的查询方法,几种方法 返回一个列表的条目,查询返回的每一行。 这个 最通用的方法是 queryForList(. .) 哪一个 返回一个 列表 每个条目是一个地方吗 地图 每个条目在地图 代表列值的行。 如果你添加一个方法 以上的例子来检索一个列表的所有行,它将看起来像 这个:

```
private JdbcTemplate jdbcTemplate;

public void setDataSource(DataSource dataSource) {
    this.jdbcTemplate = new JdbcTemplate(dataSource);
}

public List<Map<String, Object>> getList() {
    return this.jdbcTemplate.queryForList("select * from mytable");
}
```

返回的列表将会看起来像这样:

```
[{name=Bob, id=1}, {name=Mary, id=2}]
```

14.2.6A更新数据库

下面的示例显示了一个列更新为某小学 关键。 在这个例子中,一个SQL语句行占位符 参数。 参数值可以作为传入可变参数或另外作为一个对象数组。 因此原语应包装 在原始包装器类显式或使用汽车拳击。

```
import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

public class ExecuteAnUpdate {

    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    public void setName(int id, String name) {
        this.jdbcTemplate.update(
                "update mytable set name = ? where id = ?",
                name, id);
    }
}
```

14.2.7A检索自动生成的键值

一个 Update() 便利方法 支持主键生成检索 由数据库。 这支持JDBC 3.0标准的一部分;请参阅 规范的13.6章的细节。 这个方法使用一个 PreparedStatementCreator 作为第一个参数, 这是所需的insert语句中指定。 这个 其他参数是一个 钥匙扣 ,其中包含了 生成的密钥成功回来更新。 没有一个 标准单创建一个适当的方式 PreparedStatement (这也解释了为什么这个方法签名是这样)。 下面的例子适用于甲骨文但 不能用于其他平台:

```
final String INSERT_SQL = "insert into my_test (name) values(?)";
final String name = "Rob";
```

```
KeyHolder keyHolder = new GeneratedKeyHolder();
jdbcTemplate.update(
    new PreparedStatementCreator() {
        public PreparedStatement createPreparedStatement(Connection connection) throws SQLException {
            PreparedStatement ps =
                connection.prepareStatement(INSERT_SQL, new String[] {"id"});
            ps.setString(1, name);
            return ps;
        }
    },
    keyHolder);

// keyHolder.getKey() now contains the generated key
```

14.3控制数据库连接

14.3.1A 数据源

春天获得数据库的连接通过一个 数据源 。 一个 数据源 JDBC的一部分吗 规范和是一个广义连接工厂。 它允许一个 容器或框架来隐藏连接池、事务 管理问题从应用程序代码。 作为一名开发人员,您需要 不知道如何连接到数据库,这就是 责任的管理员设置数据源。 你 最可能满足这两种角色为您开发和测试代码,但是你没有 一定要知道如何生产数据源 配置。

当使用Spring的JDBC层,你可以获得一个从JNDI数据源 或者你自己配置和连接池的实现提供了 由第三方。 流行的实现是Apache Jakarta Commons DBCP和C3P0。 实现在春季分布意味着只有 出于测试目的,不提供池。

本节使用Spring的 DriverManagerDataSource 实现和 后文将介绍几个额外的实现。

注意

只使用 DriverManagerDataSource 类应该仅仅用于测试目的,因为它没有 提供池,将多个请求时表现不佳的一个 连接是由。

你获得一个连接 DriverManagerDataSource 你通常得到一个 JDBC连接。 指定完全限定类名的JDBC 司机这样 DriverManager 可以加载 驱动程序类。 接下来,提供一个URL,JDBC驱动程序之间的变化。 (参考文档为你的司机正确的价值。) 然后 提供一个用户名和密码来连接到数据库。 这是一个 如何配置一个的例子 DriverManagerDataSource 在Java代码中:

```
DriverManagerDataSource dataSource = new DriverManagerDataSource();
dataSource.setDriverClassName("org.hsqldb.jdbcDriver");
dataSource.setUrl("jdbc:hsqldb:hsql://localhost:");
dataSource.setUsername("sa");
dataSource.setPassword("");
```

下面是相应的XML配置:

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>

<context:property-placeholder location="jdbc.properties"/>
```

下面的例子显示了基本的连通性和 配置和C3P0 DBCP。 了解更多的选项,帮助 控制池功能,请参阅产品文档 各自的连接池实现。

DBCP配置:

```
<bean id="dataSource"
        class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>

<context:property-placeholder location="jdbc.properties"/>
```

C3P0配置:

```
<bean id="dataSource"
      class="com.mchange.v2.c3p0.ComboPooledDataSource" destroy-method="close">
   <property name="driverClass" value="${jdbc.driverClassName}"/>
   <property name="jdbcUrl" value="${jdbc.url}"/>
   <property name="user" value="${jdbc.username}"/>
   <property name="password" value="${jdbc.password}"/>
</bean>

<context:property-placeholder location="jdbc.properties"/>
```

14.3.2A DataSourceUtils

这个 DataSourceUtils 类是一个方便的 和强大的助手类,它提供了 静态 方法来获取连接从JNDI和紧密联系如果 必要的。 它支持创建连接,例如, DataSourceTransactionManager 。

14.3.3A SmartDataSource

这个 SmartDataSource 接口 应该实现的类,它可以提供一个连接到一个吗 关系数据库。 它扩展了 数据源 接口允许类 用它来查询的连接是否应该关闭在一个给定的 操作。 这种用法是有效的,当你知道你会重用 连接。

14.3.4A AbstractDataSource

AbstractDataSource 是一个 文摘 基类 春天的 数据源 实现 实现了代码,是常见的所有 数据源 实现。 你延长 AbstractDataSource 类如果你 正在编写您自己的 数据源 实现。

14.3.5A SingleConnectionDataSource

这个 SingleConnectionDataSource 类是一个 实施 SmartDataSource 接口,封装了一个 单 连接 这是 不 每次使用后关闭。 显然,这不是 多线程能力。

如果任何客户端代码调用 **密切** 在 假设一个合用的连接,当使用持久化工具,设置 这个 suppressClose 财产 真正的 。 这 设置返回一个关闭抑制代理包装物理 连接。 注意,你将不能投这个 一个土生土长的甲骨文 连接 或类似的 了。

这是一个测试类。 例如,它方便 测试代码外部应用程序服务器,联同一个 简单的JNDI环境。 相比之下, DriverManagerDataSource ,它重用相同的 连接所有的时间,避免过度创建物理 连接。

14.3.6A DriverManagerDataSource

这个 DriverManagerDataSource 类是一个 实施标准 数据源 界面,配置一个普通的JDBC驱动程序通过bean属性, 并返回一个新的 连接 每一 时间。

这个实现是有用的为测试和独立 环境外的Java EE容器,或者作为一个 数据源 在一个Spring IoC bean 容器,或结合一个简单的JNDI环境。 池假设 连接关闭() 电话只会 关闭连接,所以任何 数据源 知道持久性代码应该 工作。 然而,使用javabean样式的连接池如 commons-dbcp 是如此容易,即使在一个测试环境, 它几乎总是比使用这样的一个连接池了 DriverManagerDataSource 。

14.3.7A TransactionAwareDataSourceProxy

TransactionAwareDataSourceProxy 是一个代理 为一个目标 数据源 ,它包装了, 目标 数据源 添加的意识 spring管理事务。 在这方面,它类似于一个 事务性JNDI 数据源 提供 通过一个Java EE服务器。

> **注意**
>
> 这是很少需要使用这个类,除非已经 现有的代码,必须调用和传递标准的JDBC 数据源 接口的实现。 在 这种情况下,它可能仍然有这个代码可用,同时 同时有这段代码参与春季管理事务。 它 通常比自己编写新的代码使用更高的吗 为资源管理层次的抽象,例如 JdbcTemplate 或 DataSourceUtils 。

(见 TransactionAwareDataSourceProxy Javadocs更多 细节。)

14.3.8A DataSourceTransactionManager

这个 DataSourceTransactionManager 类是一个 PlatformTransactionManager 实现 对于单一的JDBC数据源。 它将一个JDBC连接的 指定数据源到当前执行的线程,有可能 允许一个线程每数据源连接。

应用程序代码是必须的 通过检索JDBC连接 DataSourceUtils.getConnection(数据源) 而不是 Java EE标准 数据源 getconnection 。 它 抛出未经检查的 org.springframework.dao 异常 而不是检查 SQLExceptions 。 所有 框架类像 JdbcTemplate 使用这个 战略隐式。 如果不使用该事务经理, 查找策略的行为就像常见的——它就可以 在任何情况下使用。

这个 DataSourceTransactionManager 类 支持自定义隔离级别,超时,得到应用 适当的JDBC语句查询超时。 支持后者, 应用程序代码必须使用 JdbcTemplate 或 调用 DataSourceUtils.applyTransactionTimeout(. .) 为创建的每个方法声明。

这个实现可以用来代替 JtaTransactionManager 在单资源 情况下,因为它不需要容器支持JTA。 切换 两者之间只是一个物质的配置,如果你坚持 需要连接查找模式。 JTA不支持自定义 隔离级别!

14.3.9A NativeJdbcExtractor

有时你需要访问特定于供应商的JDBC方法 不同于标准的JDBC API。 这可能会有问题,如果你是 运行在一个应用程序服务器或一个 数据源 把 连接 , 声明 和 ResultSet 对象有自己的包装器对象。 为获得本机对象您可以配置您的 JdbcTemplate 或 OracleLobHandler 与 NativeJdbcExtractor 。

这个 NativeJdbcExtractor 有各种口味 来匹配你的执行环境:

- SimpleNativeJdbcExtractor
- C3P0NativeJdbcExtractor
- CommonsDbcpNativeJdbcExtractor
- JBossNativeJdbcExtractor
- WebLogicNativeJdbcExtractor
- WebSphereNativeJdbcExtractor
- XAPoolNativeJdbcExtractor

通常 SimpleNativeJdbcExtractor 是 足以展开一个 连接 对象在 大多数环境。 看到更多细节的Javadocs。

14.4一个JDBC批处理操作

大多数JDBC驱动程序提供改进的性能如果你批多个 调用同一个预置语句。 通过分组成批量更新你 限制数量的往返数据库。

14.4.1A JdbcTemplate基本批处理操作的

你完成 JdbcTemplate 批 通过实现两种方法处理的一个特殊的接口, BatchPreparedStatementSetter ,并通过在 作为第二个参数在你 batchUpdate 方法调用。 使用 getBatchSize 方法 提供电流的大小批量。 使用 setValues 方法设置的值 参数的预置语句。 这种方法被称为 您指定的次数,在 getBatchSize 调用。 下面的示例更新 演员表基于条目列表。 整个列表用作 批处理在这个例子中:

```
public class JdbcActorDao implements ActorDao {
   private JdbcTemplate jdbcTemplate;

   public void setDataSource(DataSource dataSource) {
      this.jdbcTemplate = new JdbcTemplate(dataSource);
   }

   public int[] batchUpdate(final List<Actor> actors) {
      int[] updateCounts = jdbcTemplate.batchUpdate(
            "update t_actor set first_name = ?, last_name = ? where id = ?",
            new BatchPreparedStatementSetter() {
               public void setValues(PreparedStatement ps, int i) throws SQLException {
                  ps.setString(1, actors.get(i).getFirstName());
                  ps.setString(2, actors.get(i).getLastName());
                  ps.setLong(3, actors.get(i).getId().longValue());
               }

               public int getBatchSize() {
                  return actors.size();
               }
            } );
      return updateCounts;
   }

   // ... additional methods
}
```

如果你是处理一个流的更新或阅读 文件,那么您可能有一个首选的批量大小,但最后一批 可能没有,数量的条目。 在这种情况下你可以使用 InterruptibleBatchPreparedStatementSetter 接口,它允许您中断一批一旦输入源 是疲惫。 这个

isBatchExhausted 方法允许 你信号结束的批处理。

14.4.2A批量操作的对象列表

两 JdbcTemplate 和 NamedParameterJdbcTemplate 提供了另一种 提供批量更新的方法。 而不是实现一个特殊的 批处理接口,您提供的所有参数值调用列表。 该框架循环遍历这些值,并使用一个内部的准备 声明setter。 API的变化取决于你是否使用命名 参数。 你提供的命名参数的数组 SqlParameterSource ,一个条目的每个成员 批处理。 您可以使用 SqlParameterSource.createBatch 方法创建 这个数组,数组传入要么JavaBeans或数组的地图 包含参数值。

这个例子显示了一个批量更新使用命名参数:

```
public class JdbcActorDao implements ActorDao {
   private NamedParameterTemplate namedParameterJdbcTemplate;

   public void setDataSource(DataSource dataSource) {
      this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);
   }

   public int[] batchUpdate(final List<Actor> actors) {
      SqlParameterSource[] batch = SqlParameterSourceUtils.createBatch(actors.toArray());
      int[] updateCounts = namedParameterJdbcTemplate.batchUpdate(
         "update t_actor set first_name = :firstName, last_name = :lastName where id = :id",
         batch);
      return updateCounts;
   }

   // ... additional methods
}
```

对于一个SQL语句使用经典”吗? “占位符,你 通过在一个列表包含一个对象数组更新值。 这 对象数组必须有一个条目在SQL 各占位符 语句,它们必须在同一个订单中定义的 SQL语句。

同样的例子使用经典的JDBC”? “占位符:

```
public class JdbcActorDao implements ActorDao {
   private JdbcTemplate jdbcTemplate;

   public void setDataSource(DataSource dataSource) {
      this.jdbcTemplate = new JdbcTemplate(dataSource);
   }

   public int[] batchUpdate(final List<Actor> actors) {
      List<Object[]> batch = new ArrayList<Object[]>();
      for (Actor actor : actors) {
         Object[] values = new Object[] {
               actor.getFirstName(),
               actor.getLastName(),
               actor.getId());
         batch.add(values);
      }
      int[] updateCounts = jdbcTemplate.batchUpdate(
            "update t_actor set first_name = ?, last_name = ? where id = ?",
            batch);
      return updateCounts;
   }

   // ... additional methods
}
```

上述所有批量更新方法返回一个int数组 包含影响的数量为每个批处理输入的行。 这个数 是报道的JDBC驱动程序。 如果计数不可用,JDBC 驱动程序返回一个2值。

14.4.3A与多个批次的批处理操作

最后一个示例批处理批量更新,是如此 大,你想把他们分成几个较小的批次。 你 当然可以做到这一点的方法上面提到的通过 多个调用 batchUpdate 法,但 现在有一个更方便的方法。 该方法以,除了 SQL语句,一个对象集合包含参数, 更新的数量为每个批处理和一个 ParameterizedPreparedStatementSetter 设置 值参数的预置语句。 该框架循环 通过所提供的值和打破成批量更新调用的 指定大小。

这个例子显示了一个批量更新使用批量大小的 100:

```
public class JdbcActorDao implements ActorDao {
   private JdbcTemplate jdbcTemplate;

   public void setDataSource(DataSource dataSource) {
      this.jdbcTemplate = new JdbcTemplate(dataSource);
```

```
    }

    public int[][] batchUpdate(final Collection<Actor> actors) {
        int[][] updateCounts = jdbcTemplate.batchUpdate(
                "update t_actor set first_name = ?, last_name = ? where id = ?",
                actors,
                100,
                new ParameterizedPreparedStatementSetter<Actor>() {
                    public void setValues(PreparedStatement ps, Actor argument) throws SQLException {
                        ps.setString(1, argument.getFirstName());
                        ps.setString(2, argument.getLastName());
                        ps.setLong(3, argument.getId().longValue());

                    }
                } );
        return updateCounts;
    }

    // ... additional methods
}
```

批处理更新方法对于这个调用返回的数组 int数组包含一个数组条目与数组每一批的 受影响的行数为每个更新。 顶级数组的长度 表示数量的批次执行和二级数组的 长度表示数量的更新在这批处理。 的数量 更新在每批应提供所有的批量大小 除了最后一个批次,可能要少一些,这取决于 更新对象提供的总数。 更新计数为每个更新 语句是一个报道的JDBC驱动程序。 如果数量不是 可用,JDBC驱动程序返回一个2值。

14.5一个简化JDBC操作与SimpleJdbc类

这个 SimpleJdbcInsert 和 SimpleJdbcCall 类提供一个简化的 配置利用数据库元数据,可以 这可以通过JDBC驱动程序。 这意味着有更少的配置 前面,虽然你可以覆盖或关掉元数据处理如果 你更愿意提供所有的细节在你的代码。

使用SimpleJdbcInsert 14.5.1A插入数据

首先,让我们看看 SimpleJdbcInsert 类与最少的 配置选项。 你应该实例化 SimpleJdbcInsert 在数据访问层的 初始化方法。 对于这个示例,初始化方法是 setDataSource 法。 你不需要子类 这个 SimpleJdbcInsert 类,简单地创建一个新的 实例并设置表名称使用 withTableName 法。 配置方法 这类遵循“流体”风格,返回的实例 SimpleJdbcInsert ,它允许您链所有 配置方法。 这个示例只使用一个配置方法; 你将看到的例子,以后多个参数。

```
public class JdbcActorDao implements ActorDao {
    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcInsert insertActor;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        this.insertActor =
                new SimpleJdbcInsert(dataSource).withTableName("t_actor");
    }

    public void add(Actor actor) {
        Map<String, Object> parameters = new HashMap<String, Object>(3);
        parameters.put("id", actor.getId());
        parameters.put("first_name", actor.getFirstName());
        parameters.put("last_name", actor.getLastName());
        insertActor.execute(parameters);
    }

    // ... additional methods
}
```

这里使用execute方法接受一个平原 java跑龙套地图 作为其唯一的参数。 这个 在这里,需要注意是这些密钥用于地图必须 匹配的列名表中定义的数据库。 这是 因为我们读元数据以构建实际的插入 语句。

自动生成的键SimpleJdbcInsert 14.5.2A检索使用

下面的例子使用了相同的插入是前面的,但不是 通过id获取自动生成的键,并将它保存在 新演员对象。 当你创建 SimpleJdbcInsert ,除了指定 表的名字,您指定的名称的列生成的密钥 usingGeneratedKeyColumns 法。

```
public class JdbcActorDao implements ActorDao {
    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcInsert insertActor;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        this.insertActor =
                new SimpleJdbcInsert(dataSource)
                        .withTableName("t_actor")
```

```
                .usingGeneratedKeyColumns("id");
    }

    public void add(Actor actor) {
        Map<String, Object> parameters = new HashMap<String, Object>(2);
        parameters.put("first_name", actor.getFirstName());
        parameters.put("last_name", actor.getLastName());
        Number newId = insertActor.executeAndReturnKey(parameters);
        actor.setId(newId.longValue());
    }

    // ... additional methods
}
```

主要区别在执行插入由第二个 方法是,你不添加id映射和你打电话 executeReturningKey 法。 这返回一个 java朗数 对象,您可以创建一个 数值类型的实例是用于我们的域名,你 不能依赖所有数据库返回一个特定的Java类在这里; java朗数 是基类,您可以依靠吗 在。 如果你有多个自动生成的列,或者生成的值 正非数字,然后你可以使用吗 钥匙扣 这是 返回 executeReturningKeyHolder 法。

对于一个SimpleJdbcInsert 14.5.3A指定列

你可以限制列插入指定的列表 列名与 usingColumns 方法:

```
public class JdbcActorDao implements ActorDao {
    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcInsert insertActor;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        this.insertActor =
                new SimpleJdbcInsert(dataSource)
                        .withTableName("t_actor")
                        .usingColumns("first_name", "last_name")
                        .usingGeneratedKeyColumns("id");
    }

    public void add(Actor actor) {
        Map<String, Object> parameters = new HashMap<String, Object>(2);
        parameters.put("first_name", actor.getFirstName());
        parameters.put("last_name", actor.getLastName());
        Number newId = insertActor.executeAndReturnKey(parameters);
        actor.setId(newId.longValue());
    }

    // ... additional methods
}
```

执行插入是相同的,如果你有依靠 在元数据来确定哪些列使用。

14.5.4A使用SqlParameterSource提供参数值

使用 地图 提供参数值 工作很好,但这不是最方便的类来使用。 春天 提供了几个实现的 SqlParameterSource 接口,可以使用 相反, 一是 BeanPropertySqlParameterSource , 这是一个非常方便的类,如果你有一个javabean兼容的类 包含你的价值观。 它将使用相应的getter方法 提取参数值。 这里是一个例子:

```
public class JdbcActorDao implements ActorDao {
    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcInsert insertActor;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        this.insertActor =
                new SimpleJdbcInsert(dataSource)
                        .withTableName("t_actor")
                        .usingGeneratedKeyColumns("id");
    }

    public void add(Actor actor) {
        SqlParameterSource parameters = new BeanPropertySqlParameterSource(actor);
        Number newId = insertActor.executeAndReturnKey(parameters);
        actor.setId(newId.longValue());
    }

    // ... additional methods
}
```

另一个选择是 MapSqlParameterSource 就像一个地图,但 提供一个更方便 addValue 方法, 可以进行链接。

```
public class JdbcActorDao implements ActorDao {
```

```
    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcInsert insertActor;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        this.insertActor =
                new SimpleJdbcInsert(dataSource)
                        .withTableName("t_actor")
                        .usingGeneratedKeyColumns("id");
    }

    public void add(Actor actor) {
        SqlParameterSource parameters = new MapSqlParameterSource()
                .addValue("first_name", actor.getFirstName())
                .addValue("last_name", actor.getLastName());
        Number newId = insertActor.executeAndReturnKey(parameters);
        actor.setId(newId.longValue());
    }

    // ... additional methods
}
```

正如您可以看到的,该配置是相同的,只有 执行代码的改变使用这些替代输入 类。

14.5.5A与SimpleJdbcCall调用一个存储过程

这个 SimpleJdbcCall 类利用元数据 在数据库中查找的名字 在 和 出 参数,所以,你不必明确声明它们。 你可以 声明参数如果你喜欢去做,或者如果你有参数 如 数组 或 Struct 没有一个 自动映射到一个Java类。 第一个例子展示了一个简单的 过程只返回标量值 VARCHAR 和 日期 格式从MySQL数据库。 示例程序 读取指定的演员进入和返回 first_name , last_name ,和 出生日期 列在表格 的 出 参数。

```
CREATE PROCEDURE read_actor (
  IN in_id INTEGER,
  OUT out_first_name VARCHAR(100),
  OUT out_last_name VARCHAR(100),
  OUT out_birth_date DATE)
BEGIN
  SELECT first_name, last_name, birth_date
  INTO out_first_name, out_last_name, out_birth_date
  FROM t_actor where id = in_id;
END;
```

这个 在id 参数包含 id 的演员你查找。 这个 出 参数返回数据从表中读取。

这个 SimpleJdbcCall 在一个类似的声明吗 方式到 SimpleJdbcInsert 。 你应该 实例化和配置类的初始化方法的 数据访问层。 StoredProcedure类相比,你没有 创建一个子类,您不必声明的参数 在数据库中查找元数据。 以下 是一个例子,一个 SimpleJdbcCall配置使用上述存储 程序。 唯一的配置选项,除了 数据源 ,是存储的名称 程序。

```
public class JdbcActorDao implements ActorDao {
    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcCall procReadActor;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        this.procReadActor =
                new SimpleJdbcCall(dataSource)
                        .withProcedureName("read_actor");
    }

    public Actor readActor(Long id) {
        SqlParameterSource in = new MapSqlParameterSource()
                .addValue("in_id", id);
        Map out = procReadActor.execute(in);
        Actor actor = new Actor();
        actor.setId(id);
        actor.setFirstName((String) out.get("out_first_name"));
        actor.setLastName((String) out.get("out_last_name"));
        actor.setBirthDate((Date) out.get("out_birth_date"));
        return actor;
    }

    // ... additional methods
}
```

您编写的代码的执行调用涉及 创建一个 SqlParameterSource 包含在 参数。 这是 重要的名称相匹配的输入值和提供的 参数名称宣布 在存储过程。 此案没有匹配,因为你使用 元数据来确定数据库对象应该是指在一个 存储过程。 什么是指定源为存储程序不一定是它是存储在数据库中。 一些 数据库变换名字大写而其他人使用小写 或用例指定。

这个 执行 方法以在参数 并返回一个包含任意地图 出 参数由 这个名称所指定的存储过程。 在这种情况下他们 出第一名,去年

的名字 和 出出生日期 。

的最后一部分 执行 方法创建 一个演员实例使用返回检索的数据。 再次,它是 重要的使用的名字 出 参数作为他们 在存储过程中声明。 同时, 这个案子的名字 出 参数存储在 地图匹配的结果 出 参数名称 数据库,数据库之间可能有所变化。 到 让你的代码的可移植性更好你应该做一个不区分大小写的查询或 指导弹簧使用 CaseInsensitiveMap 从 Jakarta Commons项目。 做后者,你创建你自己的 JdbcTemplate 并设置 setResultsMapCaseInsensitive 财产 真正的 。 然后你通过这个定制的 JdbcTemplate 实例的构造函数 你 SimpleJdbcCall 。 你必须包括 commons集合jar 在您的类路径 这工作。 这里是一个例子,这个配置:

```
public class JdbcActorDao implements ActorDao {
   private SimpleJdbcCall procReadActor;

   public void setDataSource(DataSource dataSource) {
      JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
      jdbcTemplate.setResultsMapCaseInsensitive(true);
      this.procReadActor =
            new SimpleJdbcCall(jdbcTemplate)
                  .withProcedureName("read_actor");
   }


   // ... additional methods
}
```

通过采取这种行动,你避免利益冲突的情况下使用 的名字你的返回 出 参数。

14.5.6A显式地声明参数来使用 SimpleJdbcCall

你已经了解了参数推导了基于元数据, 但是你可以声明然后明确如果你希望。 这可以通过创建 和配置 SimpleJdbcCall 与 declareParameters 方法,该方法接受一个变量 数量的 SqlParameter 对象作为输入。 看到 下一节详细讨论如何定义一个 SqlParameter 。

注意

显式声明是必要的,如果你使用的数据库 不是一个弹簧支持数据库。 目前弹簧支持元数据 存储过程调用的查找以下数据库:Apache Derby,DB2、MySQL、微软SQL Server、Oracle和Sybase。 我们也 支持元数据查找存储功能:MySQL,微软 SQL服务器,和甲骨文。

你可以选择要申报一个,一些或所有的参数 明确。 参数元数据仍然是用在哪里的你不 显式声明的参数。 到 旁路的所有处理元数据查找潜在的参数和 只使用声明的参数,你调用该方法 withoutProcedureColumnMetaDataAccess 作为 《宣言》。 假设你有两个或两个以上不同的电话 为一个数据库函数签名声明。 在这种情况下你所说的 useInParameterNames 指定的列表 参数名称包括对于一个给定的签名。

下面的示例显示了一个全面申报过程调用,使用 信息从前面的例子。

```
public class JdbcActorDao implements ActorDao {
   private SimpleJdbcCall procReadActor;

   public void setDataSource(DataSource dataSource) {
      JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
      jdbcTemplate.setResultsMapCaseInsensitive(true);
      this.procReadActor =
            new SimpleJdbcCall(jdbcTemplate)
                  .withProcedureName("read_actor")
                  .withoutProcedureColumnMetaDataAccess()
                  .useInParameterNames("in_id")
                  .declareParameters(
                        new SqlParameter("in_id", Types.NUMERIC),
                        new SqlOutParameter("out_first_name", Types.VARCHAR),
                        new SqlOutParameter("out_last_name", Types.VARCHAR),
                        new SqlOutParameter("out_birth_date", Types.DATE)
                  );
   }


   // ... additional methods
}
```

执行和结束的结果是两个例子 相同的;这一指定所有细节明确而不是依赖 元数据。

SqlParameters 14.5.7A如何定义

定义一个参数为SimpleJdbc类和也的 RDBMS操作类,覆盖着 SectionA 14.6,一个建模JDBC操作作为Java objectsa , 你 使用一个 SqlParameter 或它的一个子类。 你 通常指定参数名称和SQL类型的构造函数。 SQL类型被指定使用 java sql类型 常量。 我们已经看到 声明:

```
new SqlParameter("in_id", Types.NUMERIC),
new SqlOutParameter("out_first_name", Types.VARCHAR),
```

第一线的 SqlParameter 声明了一个在参数。 在参数可以用于存储 过程调用和查询使用 SqlQuery 和它的子类中覆盖 下一节。

第二行用 SqlOutParameter 声明了一个 出 参数中使用存储过程 调用。 还有一个 SqlInOutParameter 对于 inout 参数,参数,提供一个 在 值的过程,也返回 值。

注意

只有参数声明为 SqlParameter 和 SqlInOutParameter 将被用来提供吗 输入值。 这是不同的 StoredProcedure 类,它为向后 兼容性原因允许输入值提供 参数声明为 SqlOutParameter 。

因为在参数,除了名字和SQL类型,你 可以指定一个量表数值数据或类型名称为自定义数据库 类型。 对于 出 参数,您可以提供一个 RowMapper 处理映射从返回的行 一个 Ref 光标。 另一个选项是指定一个 SqlReturnType 这提供了一个机会 定义定制的处理的返回值。

14.5.8A使用SimpleJdbcCall调用一个存储功能

你叫一个存储函数在几乎相同的方式调用 存储过程,除了你提供一个函数的名字,而不是一个 程序的名字。 你使用 withFunctionName 方法配置一部分表明我们想做一个 调用一个函数,对应的字符串函数调用 生成的。 一个专门的执行调用, executeFunction, 是用来执行函数 和它返回函数的返回值是一个对象的指定 类型,这意味着你不需要检索的返回值 结果图。 一个 类似的便利方法命名 executeObject 是 也可用于存储过程,只有一个 出 参数。 下面的例子是基于一个存储函数命名 让演员的名字 返回一个演员的姓名。 这里是MySQL源对于这个函数:

```
CREATE FUNCTION get_actor_name (in_id INTEGER)
RETURNS VARCHAR(200) READS SQL DATA
BEGIN
  DECLARE out_name VARCHAR(200);
  SELECT concat(first_name, ' ', last_name)
    INTO out_name
    FROM t_actor where id = in_id;
  RETURN out_name;
END;
```

我们再来调用这个函数创建一个 SimpleJdbcCall 在初始化 法。

```
public class JdbcActorDao implements ActorDao {
    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcCall funcGetActorName;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
        jdbcTemplate.setResultsMapCaseInsensitive(true);
        this.funcGetActorName =
                new SimpleJdbcCall(jdbcTemplate)
                        .withFunctionName("get_actor_name");
    }

    public String getActorName(Long id) {
        SqlParameterSource in = new MapSqlParameterSource()
                .addValue("in_id", id);
        String name = funcGetActorName.executeFunction(String.class, in);
        return name;
    }

    // ... additional methods
}
```

execute方法使用 返回一个 字符串 包含返回值 函数调用。

14.5.9A返回结果集/ REF光标从一个SimpleJdbcCall

调用一个存储过程或函数,返回一个结果集 是有点棘手。 一些数据库返回结果集在JDBC 结果处理而其他人需要明确注册 出 参数的特定类型。 这两种方法都需要 额外的处理来遍历结果集和处理 返回的行。 与 SimpleJdbcCall 你使用 这个

returningResultSet 方法和声明一个 RowMapper 实现用于一个 具体的参数。 如果结果集返回在 结果处理,没有名字的定义,所以返回的 结果必须匹配的顺序进行声明 RowMapper 实现。 指定的名称是 还用于存储处理结果列表在结果图 返回执行语句。

下面的例子使用了一个存储过程中不带 参数和返回所有行从t演员表。 这是 MySQL源对于这个过程:

```
CREATE PROCEDURE read_all_actors()
BEGIN
 SELECT a.id, a.first_name, a.last_name, a.birth_date FROM t_actor a;
END;
```

调用这个过程你声明 RowMapper 。 因为要映射到类 遵循JavaBean的规则,你可以使用 ParameterizedBeanPropertyRowMapper 这是 由传递所需的类映射到的 newInstance 法。

```
public class JdbcActorDao implements ActorDao {
    private SimpleJdbcCall procReadAllActors;

    public void setDataSource(DataSource dataSource) {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
        jdbcTemplate.setResultsMapCaseInsensitive(true);
        this.procReadAllActors =
                new SimpleJdbcCall(jdbcTemplate)
                        .withProcedureName("read_all_actors")
                        .returningResultSet("actors",
                                ParameterizedBeanPropertyRowMapper.newInstance(Actor.class));
    }

    public List getActorsList() {
        Map m = procReadAllActors.execute(new HashMap<String, Object>(0));
        return (List) m.get("actors");
    }

    // ... additional methods
}
```

执行调用传入一个空的地图,因为这个调用 不带任何参数。 演员的名单然后检索得到的 结果地图并返回给调用者。

14.6一个建模JDBC操作作为Java对象

这个 org.springframework.jdbc.object 包 包含类允许您访问数据库在一个更多 面向对象的方式。 作为一个例子,你可以执行查询和获取 返回的结果作为一个列表包含业务对象与关系 列数据映射到业务对象的属性。 你也可以 执行存储过程和运行更新、删除和插入 语句。

注意

许多Spring开发者相信各种RDBMS操作 下面描述类(除 StoredProcedure 类)通常可以直接取代 JdbcTemplate 调用。 它常常是简单的写 DAO方法,简单地调用一个方法 JdbcTemplate 直接(相对于 封装一个查询作为一种成熟的类)。

然而,如果你正变得可测量的值从使用RDBMS 操作类,继续使用这些类。

14.6.1A SqlQuery

SqlQuery 是一个可重用的、线程安全的类 封装了一个SQL查询。 子类必须实现 newRowMapper(. .) 方法提供一个 RowMapper 可以创建一个实例 对象遍历每行中获得的 ResultSet 这是中创建 执行查询。 这个 SqlQuery 类是 很少直接使用,因为 MappingSqlQuery 子类提供了一个更方便的实现映射行 到Java类。 其他实现,扩展 SqlQuery 是 MappingSqlQueryWithParameters 和 UpdatableSqlQuery 。

14.6.2A MappingSqlQuery

MappingSqlQuery 是一个可重用的查询在吗 这具体的子类必须实现抽象 mapRow(. .) 方法将每一行的 提供 ResultSet 成一个对象的 指定的类型。 下面的示例显示了一个自定义查询,地图了 数据 t演员 关系的一个实例 演员 类。

```
public class ActorMappingQuery extends MappingSqlQuery<Actor> {

    public ActorMappingQuery(DataSource ds) {
        super(ds, "select id, first_name, last_name from t_actor where id = ?");
        super.declareParameter(new SqlParameter("id", Types.INTEGER));
        compile();
    }
```

```
    @Override
    protected Actor mapRow(ResultSet rs, int rowNumber) throws SQLException {
        Actor actor = new Actor();
        actor.setId(rs.getLong("id"));
        actor.setFirstName(rs.getString("first_name"));
        actor.setLastName(rs.getString("last_name"));
        return actor;
    }

}
```

这个类扩展 MappingSqlQuery 参数化与 演员 类型。 这个 构造函数,对于这个客户查询需要 数据源 作为唯一的参数。 在这个 构造函数调用构造器的你超类的 数据源 和SQL,应该 执行该查询检索的行。 这个SQL将用于 创建一个 PreparedStatement 所以它可能 包含占位符的任何参数中传递 执行你 必须声明每个参数使用吗 declareParameter 方法传递一个 SqlParameter 。 这个 SqlParameter 需要一个名称和JDBC类型作为 定义在 java sql类型 。 在你定义所有 参数,您调用 编译() 方法所以 语句可以准备和后执行。 这个类是线程安全的 编译后,所以只要这些实例 是由刀是初始化它们可以一直作为实例 变量和被重用。

```
private ActorMappingQuery actorMappingQuery;

@Autowired
public void setDataSource(DataSource dataSource) {
    this.actorMappingQuery = new ActorMappingQuery(dataSource);
}

public Customer getCustomer(Long id) {
    return actorMappingQuery.findObject(id);
}
```

该方法在本例中检索客户id, 是作为唯一的参数传递。 因为我们只需要一个对象 我们简单地调用返回的便利方法 findObject 与id作为参数。 如果我们有相反的查询返回一个列表 的对象,把额外的参数然后我们将使用其中一个 执行方法,它采用一组参数值传递 可变参数。

```
public List<Actor> searchForActors(int age, String namePattern) {
    List<Actor> actors = actorSearchMappingQuery.execute(age, namePattern);
    return actors;
}
```

14.6.3A SqlUpdate

这个 SqlUpdate 类封装了一个SQL 更新。 像一个查询,更新对象是可重用的,像所有 RdbmsOperation 类,一个更新可以有 参数和定义在SQL。 这个类提供了一些 更新(. .) 方法类似于 执行(. .) 查询对象的方法。 这个 SqlUpdate 类是混凝土。 它可以细分类,例如,添加一个自定义的更新方法,如在 以下代码片段,它只是叫 执行 。 然而, 你不需要继承 SqlUpdate 类 因为它可以很容易地通过设置参数化SQL和声明 参数。

```
import java.sql.Types;

import javax.sql.DataSource;

import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.SqlUpdate;

public class UpdateCreditRating extends SqlUpdate {

    public UpdateCreditRating(DataSource ds) {
        setDataSource(ds);
        setSql("update customer set credit_rating = ? where id = ?");
        declareParameter(new SqlParameter("creditRating", Types.NUMERIC));
        declareParameter(new SqlParameter("id", Types.NUMERIC));
        compile();
    }

    /**
     * @param id for the Customer to be updated
     * @param rating the new value for credit rating
     * @return number of rows updated
     */
    public int execute(int id, int rating) {
        return update(rating, id);
    }
}
```

14.6.4A StoredProcedure

这个 StoredProcedure 类是一个超类 对对象的抽象,RDBMS存储过程。 这个类是 文摘 和它的各种 执行(. .) 方法 保护 访问,防止使用其他比通过一个子类,它提供了更严格的 打字。

继承的 SQL 属性的名称将会成为 存储过程在RDBMS。

定义一个参数 StoredProcedure 类,你使用一个 SqlParameter 或它的一个子类。 你必须 指定参数名称和SQL类型的构造函数 下面的代码片段。 SQL类型被指定使用 java sql类型 常量。

```
new SqlParameter("in_id", Types.NUMERIC),
new SqlOutParameter("out_first_name", Types.VARCHAR),
```

第一线的 SqlParameter 声明了一个在参数。 在参数可以用于存储 过程调用和查询使用 SqlQuery 和它的子类中覆盖 下一节。

第二行用 SqlOutParameter 声明了一个 出 参数用于存储 过程调用。 还有一个 SqlInOutParameter 对于 我 nOut 参数,参数,提供一个 在 值的过程,也返回 值。

对于 我 n 参数,除了 名称和SQL类型,您可以指定一个量表数值数据或一个 类型名称为自定义数据库类型。 对于 出 参数你 可以提供一个 RowMapper 处理映射的行 返回从一个裁判光标。 另一个选项是指定一个 SqlReturnType 这使您能够定义 自定义处理的返回值。

下面的示例是一个简单的刀使用 StoredProcedure 要调用一个函数, sysdate() 配有任何Oracle数据库。 到 使用存储过程的功能你必须创建一个类 延伸 StoredProcedure 。 在这个例子中, StoredProcedure 类是一个内部类,但如果 你需要重用 StoredProcedure 你申报的 它作为一个顶级类。 这个例子没有输入参数,但一个 输出参数声明为日期类型使用类 SqlOutParameter 。 这个 execute() 法执行过程和提取返回的日期的 结果 地图 。 结果 地图 有一个入口为每个声明输出 参数,在这种情况下只有一个,使用参数名称 关键。

```
import java.sql.Types;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.object.StoredProcedure;

public class StoredProcedureDao {

    private GetSysdateProcedure getSysdate;

    @Autowired
    public void init(DataSource dataSource) {
        this.getSysdate = new GetSysdateProcedure(dataSource);
    }

    public Date getSysdate() {
        return getSysdate.execute();
    }

    private class GetSysdateProcedure extends StoredProcedure {

        private static final String SQL = "sysdate";

        public GetSysdateProcedure(DataSource dataSource) {
            setDataSource(dataSource);
            setFunction(true);
            setSql(SQL);
            declareParameter(new SqlOutParameter("date", Types.DATE));
            compile();
        }

        public Date execute() {
            // the 'sysdate' sproc has no input parameters, so an empty Map is supplied...
            Map<String, Object> results = execute(new HashMap<String, Object>());
            Date sysdate = (Date) results.get("date");
            return sysdate;
        }
    }

}
```

下面的示例 StoredProcedure 有两个输出参数(在本例中,甲骨文REF游标)。

```
import oracle.jdbc.OracleTypes;
import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.object.StoredProcedure;

import javax.sql.DataSource;
import java.util.HashMap;
```

```
import java.util.Map;

public class TitlesAndGenresStoredProcedure extends StoredProcedure {

    private static final String SPROC_NAME = "AllTitlesAndGenres";

    public TitlesAndGenresStoredProcedure(DataSource dataSource) {
        super(dataSource, SPROC_NAME);
        declareParameter(new SqlOutParameter("titles", OracleTypes.CURSOR, new TitleMapper()));
        declareParameter(new SqlOutParameter("genres", OracleTypes.CURSOR, new GenreMapper()));
        compile();
    }

    public Map<String, Object> execute() {
        // again, this sproc has no input parameters, so an empty Map is supplied
        return super.execute(new HashMap<String, Object>());
    }
}
```

注意,重载的变体 declareParameter(. .) 法中已使用 这个 TitlesAndGenresStoredProcedure 构造函数 传递 RowMapper 实现 实例;这是一个非常方便和强大的方法来重用现有的 功能。 代码的两个 RowMapper 实现提供 下面。

这个 TitleMapper 类地图一个 ResultSet 一个 标题 域对象中的每一行提供 ResultSet :

```
import org.springframework.jdbc.core.RowMapper;

import java.sql.ResultSet;
import java.sql.SQLException;

import com.foo.domain.Title;

public final class TitleMapper implements RowMapper<Title> {

    public Title mapRow(ResultSet rs, int rowNum) throws SQLException {
        Title title = new Title();
        title.setId(rs.getLong("id"));
        title.setName(rs.getString("name"));
        return title;
    }
}
```

这个 GenreMapper 类地图一个 ResultSet 一个 流派 域对象中的每一行提供 ResultSet 。

```
import org.springframework.jdbc.core.RowMapper;

import java.sql.ResultSet;
import java.sql.SQLException;

import com.foo.domain.Genre;

public final class GenreMapper implements RowMapper<Genre> {

    public Genre mapRow(ResultSet rs, int rowNum) throws SQLException {
        return new Genre(rs.getString("name"));
    }
}
```

将参数传递到一个存储过程,它具有一个或多个 输入参数在其定义RDBMS中,您可以编写一个强烈 类型 执行(. .) 方法,将代表 超类的无类型 执行(Map参数) 方法 (保护 访问); 示例:

```
import oracle.jdbc.OracleTypes;
import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.StoredProcedure;

import javax.sql.DataSource;

import java.sql.Types;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;

public class TitlesAfterDateStoredProcedure extends StoredProcedure {

    private static final String SPROC_NAME = "TitlesAfterDate";
    private static final String CUTOFF_DATE_PARAM = "cutoffDate";

    public TitlesAfterDateStoredProcedure(DataSource dataSource) {
        super(dataSource, SPROC_NAME);
        declareParameter(new SqlParameter(CUTOFF_DATE_PARAM, Types.DATE);
        declareParameter(new SqlOutParameter("titles", OracleTypes.CURSOR, new TitleMapper()));
        compile();
    }
```

```
    public Map<String, Object> execute(Date cutoffDate) {
        Map<String, Object> inputs = new HashMap<String, Object>();
        inputs.put(CUTOFF_DATE_PARAM, cutoffDate);
        return super.execute(inputs);
    }
}
```

14.7一个常见的问题与参数和数据值处理

常见问题与参数和数据值存在 不同的方法由Spring框架提供的JDBC。

14.7.1A提供SQL类型信息参数

春天通常决定了SQL类型的参数依据 传入参数的类型。 可以显式地提供 SQL类型被用来当设置参数值。 这是有时 需要正确地设置NULL值。

你可以提供SQL类型信息在几个方面:

- 许多更新和查询的方法 JdbcTemplate 采取额外的参数 的形式 int 数组。 该数组是用来 显示SQL类型的对应的参数使用常数 值 java sql类型 类。 提供 一个条目为每个参数。
- 您可以使用 SqlParameterValue 类 包装参数值,需要这些额外的信息。 创建 一个新实例为每个值和通过在SQL类型和参数 在构造函数中值。 你也可以提供一个可选的规模 参数的数值。
- 方法使用命名参数,使用 SqlParameterSource 类 BeanPropertySqlParameterSource 或 MapSqlParameterSource 。 他们都有方法 注册SQL类型为任何指定参数 值。

14.7.2A处理BLOB和CLOB对象

你可以存储图片,其他二进制对象,大块大块的 文本。 这些大型对象被称为BLOB和CLOB数据的二进制 字符数据。 在春天你可以处理这些大型对象通过使用 JdbcTemplate的直接和还在使用更高的抽象 RDBMS对象和提供的 SimpleJdbc 类。 所有 这些方法使用的一种实现 LobHandler 接口的实际管理 LOB数据。 这个 LobHandler 提供了访问 LobCreator 类,通过 getLobCreator 方法,用于创建新的LOB 对象被插入。

这个 LobCreator / LobHandler 提供 以下支持LOB的输入和输出:

- blob
  - 一个byte[]getBlobAsBytes和setBlobAsBytes
  - 一个InputStreamgetBlobAsBinaryStream和 setBlobAsBinaryStream
- CLOB
  - 一个getClobAsString字符串和setClobAsString
  - 一个InputStreamgetClobAsAsciiStream和 setClobAsAsciiStream
  - 读者getClobAsCharacterStream和 setClobAsCharacterStream

下一个例子显示了如何创建和插入一个BLOB。 以后你 将会看到如何从数据库读回。

这个例子使用一个 JdbcTemplate 和一个 实施 AbstractLobCreatingPreparedStatementCallbac K 。 它实现了一个方法, setValues 。 这种方法提供了一个 LobCreator 那你使用设置值的LOB 列在您的SQL insert语句。

在这个例子中,我们假设有一个变量, lobHandle R ,已经被设置为一个实例 的 DefaultLobHandler 。 你通常设置这个 价值通过依赖注入。

```
final File blobIn = new File("spring2004.jpg");
final InputStream blobIs = new FileInputStream(blobIn);
final File clobIn = new File("large.txt");
final InputStream clobIs = new FileInputStream(clobIn);
final InputStreamReader clobReader = new InputStreamReader(clobIs);
jdbcTemplate.execute(
  "INSERT INTO lob_table (id, a_clob, a_blob) VALUES (?, ?, ?)",
  new AbstractLobCreatingPreparedStatementCallback(lobHandler) {❶
      protected void setValues(PreparedStatement ps, LobCreator lobCreator)
          throws SQLException {
        ps.setLong(1, 1L);
        lobCreator.setClobAsCharacterStream(ps, 2, clobReader, (int)clobIn.length());❷
        lobCreator.setBlobAsBinaryStream(ps, 3, blobIs, (int)blobIn.length());❸
      }
  }
```

```
);
blobIs.close();
clobReader.close();
```

❶ 通过在lobHandler,在这个例子中是一个平原 DefaultLobHandler
❷ 使用方法 setClobAsCharacterStream ,通过在 内容的CLOB。
❸ 使用方法 setBlobAsBinaryStream ,通过在内容 的BLOB。

现在是时间去阅读LOB数据从数据库。 再一次,你 使用一个 JdbcTemplate 与相同的实例变量 l obHandler 和一个引用 DefaultLobHandler 。

```
List<Map<String, Object>> l = jdbcTemplate.query("select id, a_clob, a_blob from lob_table",
      new RowMapper<Map<String, Object>>() {
        public Map<String, Object> mapRow(ResultSet rs, int i) throws SQLException {
          Map<String, Object> results = new HashMap<String, Object>();
          String clobText = lobHandler.getClobAsString(rs, "a_clob");❶
          results.put("CLOB", clobText);
          byte[] blobBytes = lobHandler.getBlobAsBytes(rs, "a_blob");❷
          results.put("BLOB", blobBytes);
          return results;
        }
      });
```

❶ 使用方法 getClobAsString, 检索内容的CLOB。
❷ 使用方法 getBlobAsBytes, 检索内容的BLOB。

14.7.3A传入的值列表的条款

SQL标准允许选择行基于一个表达式 这包括一个变量的值列表。 一个典型的例子 select * from t演员,id(1、2、3) 。 这个变量 列表是不直接支持准备语句的JDBC 标准;你不能声明一个变量数量的占位符。 你需要 一个数量的变化与所需数量的占位符准备, 或者你需要动态生成的SQL字符串一旦你知道如何 许多占位符是必需的。 命名参数中提供支持 这个 NamedParameterJdbcTemplate 和 JdbcTemplate 以后者的方法。 传入值作为一个 java util列表 的 原始对象。 这个列表将被用来插入所需的 占位符和传入值在该声明 执行。

注意

小心当传递的价值观。 JDBC标准并 不保证你可以使用超过100个值为 在 表达式列表。 各种数据库超过这个数字, 但他们通常有一个硬限制多少值是允许的。 甲骨文的极限是1000。

除了原始值的值列表,你可以 创建一个 java util列表 的对象数组。 这 列表将支持多个表达式的定义 在 条款如 select * from t演员那里(id、last_name)在((1, “约翰逊”),(2,' Harrop ')) 。 当然,这要求你的 数据库支持这种语法。

14.7.4A处理复杂类型为存储过程调用

当您调用存储过程你可以有时使用复杂 特定于数据库的类型。 以适应这些类型,弹簧 提供了一个 SqlReturnType 为处理这些当 他们返回存储过程调用和 SqlTypeValue 当他们通过传递 参数的存储过程。

下面是一个示例返回一个Oracle的价值 Struct 用户声明类型的对象 项目类型 。 这个 SqlReturnType 接口有一个方法命名 getTypeValue 必须实现的。 这个接口是用来作为部分的 声明的 SqlOutParameter 。

```
final TestItem - new TestItem(123L, "A test item",
      new SimpleDateFormat("yyyy-M-d").parse("2010-12-31")););

declareParameter(new SqlOutParameter("item", OracleTypes.STRUCT, "ITEM_TYPE",
   new SqlReturnType() {
     public Object getTypeValue(CallableStatement cs, int colIndx, int sqlType, String typeName)
        throws SQLException {
      STRUCT struct = (STRUCT)cs.getObject(colIndx);
      Object[] attr = struct.getAttributes();
      TestItem item = new TestItem();
      item.setId(((Number) attr[0]).longValue());
      item.setDescription((String)attr[1]);
      item.setExpirationDate((java.util.Date)attr[2]);
      return item;
    }
   }));
```

你使用 SqlTypeValue 到 通过在一个Java对象的价值像 TestItem 到一个存储过程。 这个 SqlTypeValue 接口有一个方法命名 createTypeValue 你必须实现。 这个 通过活动连接,您可以使用它来创建 数据库对象(如 StructDescriptor 年代,见以下 例,或 ArrayDescriptor 年代。

```
final TestItem - new TestItem(123L, "A test item",
        new SimpleDateFormat("yyyy-M-d").parse("2010-12-31"););

SqlTypeValue value = new AbstractSqlTypeValue() {
  protected Object createTypeValue(Connection conn, int sqlType, String typeName) throws SQLException {
    StructDescriptor itemDescriptor = new StructDescriptor(typeName, conn);
    Struct item = new STRUCT(itemDescriptor, conn,
        new Object[] {
            testItem.getId(),
            testItem.getDescription(),
            new java.sql.Date(testItem.getExpirationDate().getTime())
        });
    return item;
  }
};
```

这 SqlTypeValue 现在可以被添加 到地图包含输入参数调用的执行 存储过程。

另一个使用的 SqlTypeValue 经过 在一个数组的值以Oracle存储过程。 甲骨文有自己的 内部 数组 类,必须用在这 情况下,你可以使用 SqlTypeValue 创建 甲骨文的一个实例 数组 和填充它 与价值观从Java 数组 。

```
final Long[] ids = new Long[] {1L, 2L};

SqlTypeValue value = new AbstractSqlTypeValue() {
  protected Object createTypeValue(Connection conn, int sqlType, String typeName) throws SQLException {
    ArrayDescriptor arrayDescriptor = new ArrayDescriptor(typeName, conn);
    ARRAY idArray = new ARRAY(arrayDescriptor, conn, ids);
    return idArray;
  }
};
```

14.8一个嵌入式数据库支持

这个 org.springframework.jdbc.datasource.embedded 包提供了支持嵌入式Java数据库引擎。 支持 HSQL , h2 ,和 Derby 本机提供。 你 也可以使用一个可扩展的API来插入新的嵌入式数据库类型和 数据源 实现。

14.8.1A为什么使用嵌入式数据库吗?

嵌入式数据库在开发阶段是有用的 项目由于其轻量级的性质。 福利包括易 配置,快速启动时间、可测试性和能力 在开发过程中快速发展的SQL。

14.8.2A创建嵌入式数据库实例使用Spring的XML

如果你想揭露嵌入式数据库实例作为bean的一个 春天ApplicationContext,使用嵌入式数据库的标签 spring jdbc命名空间:

```
<jdbc:embedded-database id="dataSource">
    <jdbc:script location="classpath:schema.sql"/>
    <jdbc:script location="classpath:test-data.sql"/>
</jdbc:embedded-database>
```

前面的配置创建一个嵌入式HSQL数据库 填充SQL从模式。 sql和testdata。 sql资源 类路径。 数据库实例提供了春天 集装箱作为一个bean类型 javax.sql.DataSource 。 这个bean可以被注入到数据访问对象 需要。

14.8.3A以编程方式创建一个嵌入式数据库实例

这个 EmbeddedDatabaseBuilder 类提供了 一口流利的API来创建一个嵌入式数据库编程。 使用 这当你需要创建一个嵌入式数据库实例在一个 独立的环境,比如数据访问对象的单元测试:

```
EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
EmbeddedDatabase db = builder.setType(H2).addScript("my-schema.sql").addScript("my-test-data.sql").build();
// do stuff against the db (EmbeddedDatabase extends javax.sql.DataSource)
db.shutdown()
```

14.8.4A扩展嵌入式数据库支持

Spring JDBC嵌入式数据库支持可扩展的两种方法:

1. 实现 EmbeddedDatabaseConfigurer 支持一个新的嵌入式数据库类型,例如Apache 德比。
2. 实现 DataSourceFactory 到 支持一个新的数据源的实现,比如连接 池,管理嵌入式数据库连接。

你是鼓励捐献回来扩展Spring 社区 jira.springframework.org 。

14.8.5A HSQL使用

Spring支持HSQL 1 8 0以上。 HSQL是默认的嵌入式 如果没有指定数据库类型是明确的。 指定HSQL明确, 设置 类型 属性的嵌入式数据库 标签 HSQL 。 如果 你是使用builder API调用 setType(EmbeddedDatabaseType) 方法 EmbeddedDatabaseType.HSQL 。

14.8.6A使用H2

Spring支持数据库以及H2。 启用H2,设置 类型 属性的 嵌入式数据库 标签 h2 。 如果 你是使用builder API调用 setType(EmbeddedDatabaseType) 方法 EmbeddedDatabaseType.H2 。

14.8.7A使用Derby

春天也支持Apache Derby 10.5及以上。 让德比, 设置 类型 属性的 嵌入式数据库 标签 Derby 。 如果 使用构建器API,调用 setType(EmbeddedDatabaseType) 方法 EmbeddedDatabaseType.Derby 。

14.8.8A测试数据访问逻辑与嵌入式数据库

嵌入式数据库提供一个轻量级的方法来测试数据访问 代码。 以下是数据访问单元测试模板,使用一个 嵌入式数据库:

```
public class DataAccessUnitTestTemplate {

    private EmbeddedDatabase db;

    @Before
    public void setUp() {
        // creates an HSQL in-memory database populated from default scripts
        // classpath:schema.sql and classpath:data.sql
        db = new EmbeddedDatabaseBuilder().addDefaultScripts().build();
    }

    @Test
    public void testDataAccess() {
        JdbcTemplate template = new JdbcTemplate(db);
        template.query(...);
    }

    @After
    public void tearDown() {
        db.shutdown();
    }

}
```

14.9一个初始化一个数据源

这个 org.springframework.jdbc.datasource.init 包提供了支持初始化一个现有的 数据源 。 嵌入式数据库支持提供了 一个选项来创建和初始化 数据源 对于一个应用程序,但是有时你 需要初始化一个实例运行在某个服务器上。

14.9.1A初始化数据库实例使用Spring的XML

如果你想要初始化数据库,您可以提供一个 引用一个DataSource bean,使用 初始化数据库 标记在 spring jdbc 命名空间:

```
<jdbc:initialize-database data-source="dataSource">
  <jdbc:script location="classpath:com/foo/sql/db-schema.sql"/>
  <jdbc:script location="classpath:com/foo/sql/db-test-data.sql"/>
</jdbc:initialize-database>
```

上面的示例脚本指定运行两种反对 数据库:第一个脚本是创建一个模式,第二个是一个 测试数据集插入。 脚本的位置也可以模式 通配符在平时的蚂蚁风格用于资源在弹簧(如。 classpath *:/ com/foo/ * * / sql / *数据的sql)。 如果一个模式是 用脚本执行在词汇的顺序URL或 文件名。

的默认行为数据库初始化器是 无条件地执行脚本提供。 这不会永远 你想要的,例如,如果对一个现有的数据库,运行 已经有测试数据在它。 不小心删除的可能性 数据是减少最常见的模式(如上图),创建了 表的第一个,然后再插入数据,第一步将会失败如果已经存在的表。

但是,为了获得更多的控制创建和删除 现有的数据,XML命名空间提供了一个两个选项。 这个 首先是标志开关初始化和关闭。 这可以设置 根据环境(如拉一个布尔值系统 属性或一个环境bean),例如。

```
<jdbc:initialize-database data-source="dataSource"
   enabled="#{systemProperties.INITIALIZE_DATABASE}">
  <jdbc:script location="..."/>
</jdbc:initialize-database>
```

第二个选项来控制与现有数据所发生的是 更能容忍失败。 为此你可以控制的能力 初始化器忽略某些错误的SQL执行的 脚本,例如。

```
<jdbc:initialize-database data-source="dataSource" ignore-failures="DROPS">
  <jdbc:script location="..."/>
</jdbc:initialize-database>
```

在这个例子中我们 说我们希望有时候脚本将运行在一个空的 数据库和有一些下降语句的脚本将 因此失败。 所以失败的SQL滴 语句将 忽视,但其他的失败将导致异常。 这是有用的,如果 你的SQL方言不支持 滴…… 如果存在 (或 类似的)但你要无条件删除所有之前的测试数据 重新创建它。 在这种情况下,第一个脚本通常是一组下降, 其次是一套 创建 语句。

这个 忽略失败 选项可以设置为 没有 (默认的), 滴 (忽略失败 滴)或 所有 (忽略所有失败)。

如果你需要更多的控制比你得到的XML名称空间,你 可以简单地使用 DataSourceInitializer 直接,它定义一个组件在应用程序中。

初始化的其他组件的依赖 数据库

一个大的类的应用程序可以使用这个数据库 初始化器没有进一步的并发症:那些不使用 数据库直到Spring上下文已经开始。 如果你 应用程序是 不 其中的一个然后你可能 需要阅读这一节的其余部分。

数据库初始化取决于数据源实例和 运行脚本提供了在其初始化的回调(观点。 init方法 在XML bean定义或 InitializingBean)。 如果其他bean依赖于相同的数据 源,也使用数据源在一个初始化的回调之后 可能有一个问题,因为数据尚未 初始化。 一个常见的例子是一个缓存初始化 急切地从数据库加载数据在应用程序 启动。

绕过这个问题你两个选择:改变你的缓存 初始化策略,后面的阶段,或确保数据库 初始化器初始化第一。

第一个选项可能容易如果应用程序在你的 控制,而不是其他。 一些建议如何实现这个 是

- 使缓存初始化懒洋洋地在第一次使用, 改善应用程序的启动时间
- 有你的缓存或一个单独的组件初始化 缓存实现 生命周期 或 SmartLifecycle 。 当应用程序上下文开始 了 SmartLifecycle 可以自动启动 它的 autoStartup 国旗是集,和一个 生命周期 可以开始手动调用吗 ConfigurableApplicationContext.start() 在 封闭的上下文。
- 使用弹簧 ApplicationEvent 或类似 自定义观察者机制引发缓存的初始化。 ContextRefreshedEvent 总是发表的吗 上下文时准备好使用(毕竟豆子已经 初始化),所以,通常是一个有用的钩(这是怎么了 SmartLifecycle 通过默认)。

第二个选项也可以很容易。 一些建议 实现这个是

- 依靠弹簧BeanFactory默认行为,这是 豆子是初始化,在注册订单。 你可以很容易地 安排采用惯例一组 <进口/ >元素,订单应用程序模块, 和确保数据库和数据库初始化 上市第一
- 单独的数据源和业务组件 用它和控制他们的启动顺序将它们 单独的ApplicationContext实例(如父母有 数据源和儿童有业务组件)。 这 结构是常见的在Spring web应用程序,但可以更多 一般应用。
- 使用一个模块运行时像SpringSource dm Server和 单独的数据源和组件都依赖于它。 如指定bundle启动顺序为 datasource - > 初始化器- >业务组件。

15。 一个对象关系映射(ORM)数据访问

15.1一个介绍ORM和春天

Spring框架支持集成 使用Hibernate,Java持久化API(JPA),Java数据对象(JDO)和 iBATIS SQL映射为资源管理、数据访问对象(DAO) 实现,和交易策略。 例如,对于Hibernate 有一流的支持与几个方便奥委会特性 解决了许多典型的Hibernate集成问题。 您可以配置 所有的支持功能,为O / R映射工具(对象关系)通过 依赖注入。 他们可以参加春季的资源 和事务管理,他们符合春天的通用 事务和DAO异常层次结构。 推荐的集成 风格是代码DAOs反对纯Hibernate,JPA,JDO api。 这个 使用Spring的旧式风格的刀模板不再推荐; 然而,这种风格的报道中可以找到 SectionA一个。 1、一个ORM usagea经典 在附录。

春天增加显著增强ORM层你的选择 当您创建数据访问应用程序。 你可以利用尽可能多的 集成支持如你所愿,你应该比较这种集成 努力的成本和风险建立一个类似的基础设施 内部。 您可以使用许多ORM支持像一个图书馆, 无论技术,因为所有东西都设

计成一组 可重用的javabean。 ORM在Spring IoC容器促进 配置和部署。 因此大多数的例子在这一节中显示 配置Spring容器内。

好处,使用Spring框架来创建你的ORM DAOs 包括:

- 更容易测试。 Spring的IoC方法使得 它容易交换实现和配置的位置 hibernate SessionFactory 情况下, JDBC 数据源 情况下,事务 经理和映射对象实现(如果需要)。 这反过来使其 更容易测试每一块持续相关代码 隔离。
- 常见的数据访问异常。 弹簧可以 从你的ORM工具包装异常,将他们从专有 (潜在的检查),一个共同的运行时异常 DataAccessException层次结构。 这个特性允许您处理大多数 持久性的异常,这是不可恢复的,只有在 适当的层,没有恼人的样板捕获,抛出, 异常声明。 你仍然可以陷阱和处理异常 必要的。 记住,JDBC异常(包括数据库特定的 方言)也转换为相同的层次结构,这意味着你 可以执行一些操作与JDBC在一个一致的编程吗 模型。
- 通用资源管理。 春天 应用程序上下文可以处理位置和配置 hibernate SessionFactory 实例,JPA 会 实例,JDBC 数据源 实例,iBATIS SQL映射 配置对象,和其他相关资源。 这使得这些 值容易管理和改变。 Spring提供了高效、简单 安全操作的持久性资源。 例如,相关代码 使用Hibernate通常需要使用相同的冬眠 会话 以确保效率和适当的 事务处理。 春天使它易于创建和绑定 会话 当前线程 透明的, 暴露出当前 会话 通过 hibernate SessionFactory 。 因此春天 解决了许多慢性问题的典型Hibernate使用,任何地方 或JTA事务环境。
- 综合事务管理。 你可以 包装你的ORM代码与一个声明式的、面向方面的编程 (AOP)风格方法拦截器通过 transactional 注释或 显式配置事务AOP的建议在一个XML 配置文件。 在这两种情况下,事务语义和异常 处理(回滚等)将为您处理。 如前所 下面,在 资源和事务 管理 ,你也可以交换各种事务经理, 不影响你的orm相关代码。 例如,您可以互换 本地事务和JTA之间,相同的全服务(这样的 作为声明性事务)可在这两个场景。 此外,jdbc相关代码完全整合事务性的 你使用的代码做ORM。 这是有用的数据访问 不适合ORM,比如批处理和BLOB流, 仍需要 共享公共事务与ORM操作。

待办事项: 提供链接到当前的样本

15.2一般ORM集成的考虑

本节重点介绍注意事项,适用于所有ORM 技术。 这个 SectionA 15.3,一个Hibernatea 一节中提供了更多的 细节和也显示这些功能和配置在一个具体的 上下文。

春天的主要目标是清楚的ORM集成应用程序 分层,与任何数据访问和事务的技术,和宽松的 耦合的应用程序对象。 没有更多的业务服务依赖关系 数据访问或事务策略,没有更多的硬编码的资源 查找,没有更多的,没有更多的单件前些年定制服务 注册中心。 一个简单的和一致的方法来连接应用程序 对象,使他们成为可重用和自由从集装箱依赖关系 可能的。 所有的个人数据访问特性是有用的在他们自己的 但良好的集成Spring应用程序上下文的概念,提供 基于xml的配置和交叉引用的普通JavaBean实例 那不需要弹簧意识。 在一个典型的Spring应用程序,许多 重要的对象是JavaBeans:数据访问模板、数据访问 对象、事务经理、业务服务,使用数据访问 对象和事务管理器,web视图解析器、web控制器 使用业务服务,等等。

15.2.1A资源和事务管理

典型的业务应用程序是堆满了重复 资源管理代码。 许多项目试图发明他们自己的 解决方案,有时牺牲正确处理失败的 编程方便。 提倡简单的解决方案为适当的春天 资源处理,即国际奥委会通过模板的例子 JDBC和AOP应用拦截器的ORM技术。

基础设施提供了适当的资源处理和 适当的转换的特定API的一个未经检查的异常 基础设施异常层次结构。 春天 介绍了一个DAO异常层次结构,适用于任何数据访问 策略。 为直接JDBC, JdbcTemplate 类 在上一节中提到的处理和适当提供连接 转换的 SQLException 到 DataAccessException 层次结构,包括 翻译的特定于数据库的SQL错误代码到有意义的异常 类。 为ORM技术,详见下一节如何得到 相同的异常转换效益。

当涉及到事务管理, JdbcTemplate 在春季班钩子 事务支持,同时支持JTA和JDBC事务,通过 各自的弹簧事务管理器。 为支持ORM 技术弹簧提供了Hibernate,JPA和JDO支持通过 Hibernate,JPA,JDO事务经理以及JTA支持。 对于 事务支持的细节,请参阅 ChapterA 12, 事务管理 一章。

15.2.2A异常翻译

当你使用Hibernate,JPA或JDO在刀,你必须决定如何 处理持久性技术的原生异常类。 DAO 抛出一个子类的一个 HibernateException , PersistenceException 或 JDOException 根据不同的技术。 这些异常都是运行时异常和不需要 宣布或捕获。 您可能还需要处理 IllegalArgumentException 和 IllegalStateException 。 这意味着调用者 只能治疗异常通常致命的,除非他们想靠吗 在持久性技术自身的异常结构。 捕捉 具体原因如一个乐观锁定失败是不可能的 无需调用者实现策略。 取消这交易 可能接受的应用程序和/或强烈orm 不需要任何特殊的例外处理。 然而,春天使 异常翻译应用透明地通过 @Repository 注释:

```
@Repository
public class ProductDaoImpl implements ProductDao {

    // class body here...

}
```

```
<beans>

  <!-- Exception translation bean post processor -->
  <bean class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"/>

  <bean id="myProductDao" class="product.ProductDaoImpl"/>

</beans>
```

后处理程序自动查找所有例外 翻译(实现的 PersistenceExceptionTranslator 接口) 和建议所有bean标记着 @Repository 注释,以便 发现译者可以拦截和应用适当的 翻译在抛出的异常。

总之:你可以实现DAOs基于朴素的持久性 技术的API和注解,同时仍然受益 spring管理事务,依赖注入和透明 异常转换(如果需要)春天的自定义异常 层次结构。

15.3一个Hibernate

我们将首先报道 Hibernate 3 在一个春天 环境中,用它来说明这一方法,弹簧需要 对O / R映射器集成。 本部分将涵盖许多问题细节和显示不同的DAO实现和 事务界定。 大多数这些模式可以直接翻译 所有其他支持ORM工具。 以下部分在这一章 将覆盖其他ORM技术,显示更简短的例子吗 那里。

注意

Spring 3.0的,弹簧需要Hibernate 3.2或更高版本。

15.3.1A SessionFactory 设置在一个春天 容器

为了避免将应用程序对象硬编码的资源查找, 你可以定义资源,如JDBC 数据源 或Hibernate SessionFactory 在春天像豆子 集装箱。 需要访问资源的应用程序对象接收 引用这样的预定义的实例通过bean引用 示的刀定义在接下来的部分。

以下摘自一个XML应用程序上下文定义 显示了如何设置一个JDBC 数据源 和一个 hibernate SessionFactory 上 它:

```
<beans>

  <bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
    <property name="url" value="jdbc:hsqldb:hsql://localhost:9001"/>
    <property name="username" value="sa"/>
    <property name="password" value=""/>
  </bean>

  <bean id="mySessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="myDataSource"/>
    <property name="mappingResources">
      <list>
        <value>product.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <value>
        hibernate.dialect=org.hibernate.dialect.HSQLDialect
      </value>
    </property>
  </bean>

</beans>
```

开关从本地Jakarta Commons DBCP BasicDataSource 到一个jndi位于 数据源 (通常是管理的 应用程序服务器)只是一个物质的配置:

```
<beans>

  <jee:jndi-lookup id="myDataSource" jndi-name="java:comp/env/jdbc/myds"/>

</beans>
```

你也可以访问jndi坐落 SessionFactory ,使用Spring的 JndiObjectFactoryBean / < jee:jndi查找> 检索并将其公开。 然而,这

通常是不常见的EJB上下文之外。

15.3.2A实现DAOs基于普通Hibernate 3 API

Hibernate 3有一个特性称为上下文会话,其中 Hibernate本身管理一个电流 会话 每笔交易。 这是大约 相当于一个Hibernate的春天的同步 会话 每笔交易。 相应的 就像下面的例子DAO实现基于平原 Hibernate API:

```
public class ProductDaoImpl implements ProductDao {

    private SessionFactory sessionFactory;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    public Collection loadProductsByCategory(String category) {
        return this.sessionFactory.getCurrentSession()
                .createQuery("from test.Product product where product.category=?")
                .setParameter(0, category)
                .list();
    }
}
```

这种风格是类似于Hibernate参考 文档和例子,除了抱着 SessionFactory 在一个实例变量。 我们强烈推荐这样一个基于实例的设置在老派 静态 HibernateUtil 类 从Hibernate的CaveatEmptor示例应用程序。 (一般来说,不 保持任何资源 静态 变量,除非绝对 必要的。)

上面的刀是依赖注入模式:它适合 很好地融入一个Spring IoC容器,就像如果编码的反对 春天的 hibernatetemplate 。 当然,这样的刀 也可以设置在普通的Java(例如,在单元测试)。 简单 实例化并调用 setSessionFactory(. .) 与所需的工厂参考。 作为一个Spring bean定义,DAO 就像下面一样:

```
<beans>

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>

</beans>
```

这把刀的主要优势是,它取决于风格 Hibernate API只;没有进口弹簧类是必需的。 这是 当然吸引人从一个具有非侵袭性的角度来看,和将没有 怀疑感觉更自然的Hibernate开发者。

然而,DAO抛出平原 HibernateException (这是遏制,那么 不是必须声明或捕捉),这意味着调用者只能 治疗异常通常致命的——除非他们想依赖 Hibernate的异常层次结构。 捕捉特定原因如一个 乐观锁定失败就不可能不把调用者 实现策略。 取消这交易可能是可以接受的 应用程序基于hibernate和/或强烈不需要任何 特殊例外处理。

幸运的是,春天的 LocalSessionFactoryBean 支持Hibernate的 SessionFactory.getCurrentSession() 方法 任何Spring事务策略,返回当前Spring管理 事务性 会话 即使 HibernateTransactionManager 。 当然, 标准的方法的行为仍然是当前的回归 会话 有关正在进行的JTA 事务,如果任何。 这种行为适用不管你 使用Spring的 JtaTransactionManager ,EJB 容器管理的事务(cmt),或JTA。

总之:你可以实现DAOs基于平原Hibernate 3 API,同时仍然能够参与spring管理 事务。

15.3.3A声明式事务划分

我们建议你使用Spring的声明式事务 支持,这使您能够代替显式事务界定 API调用在你的Java代码与AOP事务拦截器。 这 事务拦截器可以配置在Spring容器使用 要么Java注释或XML。 这个声明式事务功能允许你 保持业务服务的重复事务界定。自由代码和关注添加业务逻辑,这是真正的价值 您的应用程序。

注意

继续之前,你是 强烈 鼓励阅读 SectionA 12.5,一个managementa声明式事务 如果你 并没有这样做。

此外,像传播行为和事务语义 隔离级别可以改变在一个配置文件,不影响 业务服务实现。

下面的例子展示了如何配置一个AOP 事务拦截器,使用XML,因为一个简单的服务类:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:aop="http://www.springframework.org/schema/aop"
      xmlns:tx="http://www.springframework.org/schema/tx"
      xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/tx
      http://www.springframework.org/schema/tx/spring-tx.xsd
      http://www.springframework.org/schema/aop
      http://www.springframework.org/schema/aop/spring-aop.xsd">

  <!-- SessionFactory, DataSource, etc. omitted -->

  <bean id="transactionManager"
          class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory"/>
  </bean>

  <aop:config>
    <aop:pointcut id="productServiceMethods"
          expression="execution(* product.ProductService.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="productServiceMethods"/>
  </aop:config>

  <tx:advice id="txAdvice" transaction-manager="myTxManager">
    <tx:attributes>
      <tx:method name="increasePrice*" propagation="REQUIRED"/>
      <tx:method name="someOtherBusinessMethod" propagation="REQUIRES_NEW"/>
      <tx:method name="*" propagation="SUPPORTS" read-only="true"/>
    </tx:attributes>
  </tx:advice>

  <bean id="myProductService" class="product.SimpleProductService">
    <property name="productDao" ref="myProductDao"/>
  </bean>

</beans>
```

这是服务类,建议:

```
public class ProductServiceImpl implements ProductService {

    private ProductDao productDao;

    public void setProductDao(ProductDao productDao) {
        this.productDao = productDao;
    }

    // notice the absence of transaction demarcation code in this method
    // Spring's declarative transaction infrastructure will be demarcating
    // transactions on your behalf
    public void increasePriceOfAllProductsInCategory(final String category) {
        List productsToChange = this.productDao.loadProductsByCategory(category);
        // ...
    }
}
```

我们还显示一个属性支持基于配置,在 下面的例子。 你 @ transactional注释服务层与注释和指导 Spring容器来找到这些注释和提供事务 语义这些注释的方法。

```
public class ProductServiceImpl implements ProductService {

    private ProductDao productDao;

    public void setProductDao(ProductDao productDao) {
        this.productDao = productDao;
    }

    @Transactional
    public void increasePriceOfAllProductsInCategory(final String category) {
        List productsToChange = this.productDao.loadProductsByCategory(category);
        // ...
    }

    @Transactional(readOnly = true)
    public List<Product> findAllProducts() {
        return this.productDao.findAllProducts();
    }

}
```

正如您可以看到的从以下配置示例中, 配置是大大简化,而XML的例子中, 同时还提供相同的功能由注释 在服务层代码。 你需要提供的 TransactionManager实现和一个 “< tx:注解驱动/ >” 条目。

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop.xsd">

  <!-- SessionFactory, DataSource, etc. omitted -->

  <bean id="transactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory"/>
  </bean>

  <tx:annotation-driven/>

  <bean id="myProductService" class="product.SimpleProductService">
    <property name="productDao" ref="myProductDao"/>
  </bean>

</beans>
```

15.3.4A程序性事务界定

你可以限定交易在一个更高的水平的 应用程序的基础上,这样的底层数据访问服务跨越 任何数量的操作。 也不存在的限制 实现周围的业务服务;它只需要一个 春天 PlatformTransactionManager 。 再一次, 后者可以来自任何地方,但最好是作为一个bean引用 通过 setTransactionManager(. .) 方法, 就像 productDAO 应该设定的吗 setProductDao(. .) 法。 以下 片段显示一个事务管理器和一个业务服务定义 一个Spring应用程序上下文,和一个示例的一个业务方法 实现:

```xml
<beans>

  <bean id="myTxManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>

  <bean id="myProductService" class="product.ProductServiceImpl">
    <property name="transactionManager" ref="myTxManager"/>
    <property name="productDao" ref="myProductDao"/>
  </bean>

</beans>
```

```java
public class ProductServiceImpl implements ProductService {

    private TransactionTemplate transactionTemplate;
    private ProductDao productDao;

    public void setTransactionManager(PlatformTransactionManager transactionManager) {
        this.transactionTemplate = new TransactionTemplate(transactionManager);
    }

    public void setProductDao(ProductDao productDao) {
        this.productDao = productDao;
    }

    public void increasePriceOfAllProductsInCategory(final String category) {
        this.transactionTemplate.execute(new TransactionCallbackWithoutResult() {

            public void doInTransactionWithoutResult(TransactionStatus status) {
                List productsToChange = this.productDao.loadProductsByCategory(category);
                // do the price increase...
            }
        }
        );
    }
}
```

春天的 TransactionInterceptor 允许任何 检查应用程序异常被抛出的回调代码,虽然 TransactionTemplate 仅限于未经检查的 例外在回调。 TransactionTemplate 在案件触发回滚 一个未经检查的应用程序异常,或如果事务被标记 回滚只有由应用程序(通过 TransactionStatus)。 TransactionInterceptor 同样的行为 违约但允许可配置回滚策略/方法。

15.3.5A事务管理策略

两 TransactionTemplate 和 TransactionInterceptor 代表实际的 事务处理到一个 PlatformTransactionManager 实例,可以

一个 HibernateTransactionManager (对于一个 hibernate SessionFactory ,使用一个 ThreadLocal 会话 下罩)或一个 JtaTransactionManager (委托给了JTA 子系统的容器),Hibernate应用程序。 你甚至可以使用 一个自定义 PlatformTransactionManager 实现。 开关从本地Hibernate事务管理 对JTA,比如当面对分布式事务的要求 某些部署您的应用程序,只是一种 配置。 简单地取代Hibernate事务经理 春天的JTA事务实现。 两个事务界定 和数据访问代码将工作没有变化,因为他们只是利用 一般事务管理api。

对于分布式事务跨多个Hibernate会话 工厂,干脆把 JtaTransactionManager 作为一个交易策略与多个 LocalSessionFactoryBean 定义。 每个刀 然后得到一个特定的 SessionFactory 引用传递到其相应的bean属性。 如果所有的潜在 JDBC数据源是事务性的容器,一个业务服务 可以限定交易在任何数量的dao和任意数量的吗 会话工厂没有特殊的方面,只要是使用 JtaTransactionManager 作为战略。

```
<beans>

  <jee:jndi-lookup id="dataSource1" jndi-name="java:comp/env/jdbc/myds1"/>

  <jee:jndi-lookup id="dataSource2" jndi-name="java:comp/env/jdbc/myds2"/>

  <bean id="mySessionFactory1"
        class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="myDataSource1"/>
    <property name="mappingResources">
      <list>
        <value>product.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <value>
        hibernate.dialect=org.hibernate.dialect.MySQLDialect
        hibernate.show_sql=true
      </value>
    </property>
  </bean>

  <bean id="mySessionFactory2"
        class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="myDataSource2"/>
    <property name="mappingResources">
      <list>
        <value>inventory.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <value>
        hibernate.dialect=org.hibernate.dialect.OracleDialect
      </value>
    </property>
  </bean>

  <bean id="myTxManager" class="org.springframework.transaction.jta.JtaTransactionManager"/>

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="sessionFactory" ref="mySessionFactory1"/>
  </bean>

  <bean id="myInventoryDao" class="product.InventoryDaoImpl">
    <property name="sessionFactory" ref="mySessionFactory2"/>
  </bean>

  <bean id="myProductService" class="product.ProductServiceImpl">
    <property name="productDao" ref="myProductDao"/>
    <property name="inventoryDao" ref="myInventoryDao"/>
  </bean>

  <aop:config>
    <aop:pointcut id="productServiceMethods"
              expression="execution(* product.ProductService.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="productServiceMethods"/>
  </aop:config>

  <tx:advice id="txAdvice" transaction-manager="myTxManager">
    <tx:attributes>
      <tx:method name="increasePrice*" propagation="REQUIRED"/>
      <tx:method name="someOtherBusinessMethod" propagation="REQUIRES_NEW"/>
      <tx:method name="*" propagation="SUPPORTS" read-only="true"/>
    </tx:attributes>
  </tx:advice>

</beans>
```

两 HibernateTransactionManager 和 JtaTransactionManager 允许适当的jvm级别 与Hibernate缓存处理,没有特定容器的事务 经理查找或JCA连接器(如果你不使用EJB来发起 事务)。

HibernateTransactionManager 可以导出 Hibernate JDBC 连接 对纯JDBC 访问代码,为一个特定的 数据源 。 此功能允许高级事务界定与混合 Hibernate和JDBC数据访问完全没有JTA,如果你 只有一个数据库访问。 HibernateTransactionManager 自动暴露 Hibernate事务作为一个JDBC事务如果你有设置 传入 SessionFactory 与 数据源 通过 数据源 财产的 LocalSessionFactoryBean 类。 或者,你 可以指定明确的 数据源 对于 该交易被认为是暴露通过吗 数据源 财产的 HibernateTransactionManager 类。

15.3.6A比较容器管理和本地定义的资源

你可以切换一个容器管理的JNDI SessionFactory 和一个 局部定义的一个,而无需改变一行 应用程序代码。 是否保持资源定义在容器 或在本地应用程序中主要是一个重要的事务 策略,您使用。 而春天定义的地方 SessionFactory ,一个手动注册 JNDI SessionFactory 不提供任何 福利。 部署 SessionFactory 通过Hibernate的JCA连接器提供的附加价值 参与Java EE服务器的管理基础设施,但是 不添加实际价值超过。

春天的事务支持不是绑定到一个容器。 与任何其他战略配置比JTA、事务支持也 工作在一个独立的或测试环境。 特别是在典型的 单数据库升级到企业级的情况下交易,弹簧的单资源的地方 事务支持是 一个轻量级的和强大的替代JTA。 当您使用本地 EJB 无状态会话bean来驱动交易,你主要取决于一个EJB 容器和JTA,即使你只访问一个数据库,只有 使用无状态会话bean来提供声明性事务通过 容器管理的事务。 同时, 直接使用JTA编程方式需要一个Java EE环境 好。 JTA并不只涉及集装箱依赖从JTA 本身和JNDI 数据源 实例。 对于非弹簧,jta驱动,你要冬眠交易使用 Hibernate JCA连接器,或额外的Hibernate事务代码 TransactionManagerLookup 配置为 适当的jvm级别缓存。

台弹力事务工作与局部定义的 hibernate SessionFactory 当他们做 当地一个JDBC 数据源 如果他们是 访问一个数据库。 因此你只需要使用Spring的JTA 事务策略当你有分布式事务的需求。 JCA连接器需要特定容器部署步骤, 显然JCA支持放在第一位。 这个配置需要 更多的工作比一个简单的web应用程序的部署与当地资源 定义和台弹力事务。 还有,你经常需要 企业版你的容器如果你正在使用,例如, WebLogic表达,它不提供JCA。 一个Spring应用程序与 当地的资源和交易跨越一个单一的数据库工作 任何Java EE web容器(没有JTA,JCA或EJB)如Tomcat, 树脂、甚至普通码头。 此外,您可以很容易地重用这样的 中间层在桌面应用程序或测试套件。

考虑到所有的事情,如果你不使用ejb,坚持当地 SessionFactory 设置和弹簧的 HibernateTransactionManager 或 JtaTransactionManager 。 你把所有的 好处,包括适当的事务的jvm级别缓存和 分布式事务,没有不便的容器 部署。 JNDI注册一个Hibernate SessionFactory 通过JCA连接器 只有增加价值如果结合ejb。

15.3.7A伪应用服务器与Hibernate的警告

在一些JTA环境具有非常严格的 XADataSource 实现——目前 只有一些WebLogic Server和WebSphere版本——当 Hibernate是 配置不考虑JTA PlatformTransactionManager 对象 这种环境下,它是可能的假警报或例外 显示在应用程序服务器日志。 这些警告或异常 表明连接访问不再有效,或JDBC 访问不再是有效的,可能是因为事务不再 活跃的。 作为一个例子,这是一个实际的异常从服务器:

```
java.sql.SQLException: The transaction is no longer active - status: 'Committed'.
   No further JDBC access is allowed within this transaction.
```

你解决这个警告通过简单地使Hibernate意识到 JTA PlatformTransactionManager 实例, 它将同步(连同弹簧)。 你有两个选择 这样做:

- 如果在您的应用程序上下文你已经直接 获得JTA PlatformTransactionManager 对象 (大概从JNDI通过 JndiObjectFactoryBean 或 < jee:jndi查找>) 和喂养它,例如,春天的 JtaTransactionManager ,那么最简单的方法 是指定的引用bean定义这个JTA吗 PlatformTransactionManager 实例作为 的价值 JtaTransactionManager 财产 对于 LocalSessionFactoryBean。 弹簧然后让 对象可以冬眠。
- 更有可能你不已经JTA PlatformTransactionManager 实例, 因为春天的 JtaTransactionManager 可以 发现它本身。 因此 您需要配置Hibernate来查找JTA PlatformTransactionManager 直接。 这可以通过配置应用服务器,具体的 TransactionManagerLookup 类在Hibernate 配置中所描述的一样,Hibernate手册。

本节的其余部分描述的事件顺序 有和没有发生Hibernate的意识的JTA PlatformTransactionManager 。

当Hibernate配置不与任何意识的JTA PlatformTransactionManager ,以下 事件发生在当一个JTA事务提交:

1. JTA事务提交。
2. 春天的 JtaTransactionManager 是 同步到JTA事务,所以它被称为回通过 afterCompletion 回调的JTA事务 经理。
3. 在其他活动中,这种同步可以 触发一个回调到春天冬眠,通过Hibernate的 afterTransactionCompletion 回调(使用 清除

Hibernate缓存),其次是显式的 close() 调用Hibernate会话,它 使Hibernate来尝试 close() JDBC 连接。
4. 在某些环境中,这个 连接关闭() 叫然后触发 警告或错误,应用程序服务器不再考虑 连接 可用的,因为 事务已经提交。

当Hibernate配置了JTA的意识 PlatformTransactionManager ,以下 事件发生在当一个JTA事务提交:

1. JTA事务准备提交。
2. 春天的 JtaTransactionManager 是 同步到JTA事务,事务被称为 回来通过 beforeCompletion 回调的 JTA事务管理器。
3. 春天是知道Hibernate本身是同步的 JTA事务,比前面的行为不同 场景。 假设Hibernate 会话 需要关闭, 春天将关闭现在。
4. JTA事务提交。
5. Hibernate是同步的JTA事务,所以 事务被称为回通过 afterCompletion 回调的JTA事务 经理,可以正确地明确其缓存。

JDO 15.4

Spring支持标准JDO 2.0和2.1 api作为数据访问 策略,遵循同样的风格为Hibernate支持。 这个 相应的集成类位于 org.springframework.orm.jdo 包。

15.4.1A PersistenceManagerFactory 设置

弹簧提供了一个 LocalPersistenceManagerFactoryBean 类, 允许您定义一个本地JDO PersistenceManagerFactory 在春天 应用程序上下文:

```
<beans>

 <bean id="myPmf" class="org.springframework.orm.jdo.LocalPersistenceManagerFactoryBean">
  <property name="configLocation" value="classpath:kodo.properties"/>
 </bean>

</beans>
```

或者,您可以设置一个 PersistenceManagerFactory 通过直接 实例化一个 PersistenceManagerFactory 实现 类。 一个JDO PersistenceManagerFactory 实现类遵循javabean模式,就像一个JDBC 数据源 的实现类,它是 一个自然的适合一个配置,使用弹簧。 这个设置风格 通常支持spring定义JDBC 数据源 ,传递到 ConnectionFactory 财产。 例如,对于 开源JDO实现 DataNucleus(原JPOX)(http://www.datanucleus.org/), 这是XML配置 PersistenceManagerFactory 实现:

```
<beans>

 <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
  <property name="driverClassName" value="${jdbc.driverClassName}"/>
  <property name="url" value="${jdbc.url}"/>
  <property name="username" value="${jdbc.username}"/>
  <property name="password" value="${jdbc.password}"/>
 </bean>

 <bean id="myPmf" class="org.datanucleus.jdo.JDOPersistenceManagerFactory" destroy-method="close">
  <property name="connectionFactory" ref="dataSource"/>
  <property name="nontransactionalRead" value="true"/>
 </bean>

</beans>
```

你也可以设置JDO PersistenceManagerFactory 在JNDI 环境的Java EE应用服务器,通常通过JCA 连接器提供了特定的JDO实现。 春天的 标准 JndiObjectFactoryBean 或 < jee:jndi查找> 可以用来 检索和揭露这样一个 PersistenceManagerFactory 。 然而, 一个EJB上下文之外,没有真正的利益存在着 PersistenceManagerFactory 在JNDI:只有 选择这样一个设置为一个很好的理由。 看到 SectionA 15 3 6,一个比较容器管理和本地定义的resourcesa 对于一个讨论;参数 也有适用于JDO。

15.4.2A实现DAOs基于JDO API的平原

DAOs可以直接写对平原JDO API,没有 任何Spring依赖项,通过使用一个注射 PersistenceManagerFactory 。 以下 是一个例子,一个相应的DAO实现:

```
public class ProductDaoImpl implements ProductDao {

   private PersistenceManagerFactory persistenceManagerFactory;

   public void setPersistenceManagerFactory(PersistenceManagerFactory pmf) {
      this.persistenceManagerFactory = pmf;
   }

   public Collection loadProductsByCategory(String category) {
      PersistenceManager pm = this.persistenceManagerFactory.getPersistenceManager();
```

```
        try {
            Query query = pm.newQuery(Product.class, "category = pCategory");
            query.declareParameters("String pCategory");
            return query.execute(category);
        }
        finally {
          pm.close();
        }
    }
}
```

因为上面的刀是依赖注入模式,它 正好符合Spring容器,就像如果编码的反对 春天的 JdoTemplate :

```
<beans>

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="persistenceManagerFactory" ref="myPmf"/>
  </bean>

</beans>
```

主要的问题是,他们总是这样DAOs得到一个新的 PersistenceManager 从工厂。 到 访问spring管理事务 PersistenceManager ,定义一个 TransactionAwarePersistenceManagerFactoryProxy (包括在弹簧)放在你的目标 PersistenceManagerFactory ,然后通过 引用代理加入到dao像下面的例子:

```
<beans>

  <bean id="myPmfProxy"
      class="org.springframework.orm.jdo.TransactionAwarePersistenceManagerFactoryProxy">
    <property name="targetPersistenceManagerFactory" ref="myPmf"/>
  </bean>

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="persistenceManagerFactory" ref="myPmfProxy"/>
  </bean>

</beans>
```

你的数据访问代码将收到一个事务 PersistenceManager (如果有的话) PersistenceManagerFactory.getPersistenceManager() 它调用的方法。 后者方法调用会通过代理, 首先检查当前的事务吗 PersistenceManager 还没找到新 一个从工厂。 任何 close() 号召 PersistenceManager 被忽略的情况下 一个事务 PersistenceManager 。

如果你的数据访问代码总是运行在一个活跃的交易 (或至少在活动事务同步),它是安全的 省略了 persistencemanager关闭() 打电话 因此整个 最后 块,你可能做的 保持你的DAO实现简洁:

```
public class ProductDaoImpl implements ProductDao {

    private PersistenceManagerFactory persistenceManagerFactory;

    public void setPersistenceManagerFactory(PersistenceManagerFactory pmf) {
        this.persistenceManagerFactory = pmf;
    }

    public Collection loadProductsByCategory(String category) {
        PersistenceManager pm = this.persistenceManagerFactory.getPersistenceManager();
        Query query = pm.newQuery(Product.class, "category = pCategory");
        query.declareParameters("String pCategory");
        return query.execute(category);
    }
}
```

这样的dao,依靠活跃的交易,这是推荐的 你执行活动事务通过关闭 TransactionAwarePersistenceManagerFactoryProxy ' s allowCreate 国旗:

```
<beans>

  <bean id="myPmfProxy"
      class="org.springframework.orm.jdo.TransactionAwarePersistenceManagerFactoryProxy">
    <property name="targetPersistenceManagerFactory" ref="myPmf"/>
    <property name="allowCreate" value="false"/>
  </bean>

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="persistenceManagerFactory" ref="myPmfProxy"/>
  </bean>

</beans>
```

这把刀的主要优势是,它取决于风格JDO API 只有,没有进口弹簧类是必需的。 这是当然 从一个角度吸引具有非侵袭性,可能会觉得更 自然的JDO开发人员。

然而,DAO抛出平原 JDOException (这是遏制,那么 不是必须声明或捕捉),这意味着调用者只能 治疗异常致命的,除非你想依靠JDO的 异常结构。 捕捉特定原因如乐观 锁定失败就不可能不把调用者了 实现策略。 取消这交易可能是可以接受的 应用程序和/或基于jdo强烈不需要任何特殊的 异常处理。

总之,你可以根据平原DAOs JDO API,他们可以 还参与spring管理事务。 这种策略可能 吸引你如果你已经熟悉JDO。 然而,这样的dao 扔平原 JDOException ,你会 必须显式地转换为春天的吗 DataAccessException (如果需要的话)。

15.4.3A事务管理

注意

你是 强烈 鼓励阅读 SectionA 12.5,一个managementa声明式事务 如果你还没有这么做,以获得一个 更详细的报道Spring的声明式事务 支持。

在交易执行服务操作,您可以使用 春天的共同声明式事务设施。 例如:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
      xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:aop="http://www.springframework.org/schema/aop"
      xmlns:tx="http://www.springframework.org/schema/tx"
      xsi:schemaLocation="
  http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/tx
  http://www.springframework.org/schema/tx/spring-tx.xsd
  http://www.springframework.org/schema/aop
  http://www.springframework.org/schema/aop/spring-aop.xsd">

  <bean id="myTxManager" class="org.springframework.orm.jdo.JdoTransactionManager">
    <property name="persistenceManagerFactory" ref="myPmf"/>
  </bean>

  <bean id="myProductService" class="product.ProductServiceImpl">
    <property name="productDao" ref="myProductDao"/>
  </bean>

  <tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
      <tx:method name="increasePrice*" propagation="REQUIRED"/>
      <tx:method name="someOtherBusinessMethod" propagation="REQUIRES_NEW"/>
      <tx:method name="*" propagation="SUPPORTS" read-only="true"/>
    </tx:attributes>
  </tx:advice>

  <aop:config>
    <aop:pointcut id="productServiceMethods"
          expression="execution(* product.ProductService.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="productServiceMethods"/>
  </aop:config>

</beans>
```

JDO需要一个活跃的事务修改持久化对象。 非事务性冲洗概念不存在在JDO,相比之下 冬眠。 由于这个原因,您需要设置选择JDO 实现一个特定的环境。 具体地说,您需要设置 它显式地对JTA同步,来检测一个活跃的JTA 事务本身。 这是没有必要为本地事务 由弹簧的 JdoTransactionManager ,但 有必要参与JTA事务,是否由 春天的 JtaTransactionManager 或者通过EJB CMT 和 平原JTA。

JdoTransactionManager 能够 露出一个JDO事务JDBC访问代码访问相同的 JDBC 数据源 ,只要 注册 JdoDialect 支持检索的底层JDBC 连接 。 这是 对于基于jdbc的JDO 2.0实现默认情况下。

15.4.4A JdoDialect

作为一个高级功能,既 JdoTemplate 和 JdoTransactionManager 支持自定义 JdoDialect 可以传递到 JdoDialect bean属性。 在这个场景中,dao将 没有收到一个 PersistenceManagerFactory 参考而是一个完整的 JdoTemplate 实例 (例如,传递到 JdoTemplate 财产的 JdoDaoSupport)。 使用 JdoDialect 实现,您可以启用 高级功能由弹簧,通常是在一个特定于供应商的方式:

- 运用特定的事务语义如定制 隔离级别或事务超时

- 检索事务JDBC 连接 因接触到基于jdbc的 DAOs
- 应用查询超时,它会自动计算 从spring管理事务超时
- 急切地冲洗一 使用PersistenceManager, 让 可见,基于jdbc事务更改数据访问代码
- 高级翻译的 JDOExceptions 到 春天 DataAccessExceptions

看到 JdoDialect Javadoc为更多的细节 在其运作和如何使用它们在春天的JDO 支持。

15.5一个JPA

春天JPA,可用在 org.springframework.orm.jpa 包,提供 全面支持 Java 持久性API 以类似的方式的集成 Hibernate或JDO,虽然意识到底层实现 以提供额外的功能。

15.5.1A三个选项设置在一个春天的JPA环境

春天的JPA支持提供了三种方法建立JPA 会 需要使用的 应用程序获得一个实体管理器。

LocalEntityManagerFactoryBean

注意

只使用这个选项在简单的部署环境如 独立的应用和集成测试。

这个 LocalEntityManagerFactoryBean 创建 一个 会 适合 简单的部署环境中,应用程序只使用JPA 数据访问。 工厂bean使用JPA PersistenceProvider 自动 机制(根据JPA的Java SE引导),在大多数 情况下,只需要您指定持久性单元名称:

```
<beans>

  <bean id="myEmf" class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
    <property name="persistenceUnitName" value="myPersistenceUnit"/>
  </bean>

</beans>
```

这种形式的JPA部署是最简单和最 有限的。 你不能指 现有的JDBC 数据源 bean 定义和不支持全局事务的存在。 此外, 编织(字节码转换)的持久化类是 特定于提供程序,通常需要一个特定的JVM指定代理 在启动。 这个选项是足够的只有独立 应用程序和测试环境,JPA规范 设计。

获得一个 会 从 JNDI

注意

使用这个选项当部署一个Java EE 5服务器。 检查 你的服务器的文档如何部署一个自定义JPA提供者 到你的服务器,允许不同的提供商比 服务器的默认。

获得一个 会 从JNDI(例如在一个Java EE 5环境),是一个简单的问题 改变XML配置:

```
<beans>

   <jee:jndi-lookup id="myEmf" jndi-name="persistence/myPersistenceUnit"/>

</beans>
```

这一行动承担标准Java EE 5引导:Java EE服务器autodetects持久性单元(实际上, meta - inf / persistence . xml 在应用程序jar文件) 和 持久化单元ref Java EE中的条目 部署描述符(例如, web . xml)和 定义环境命名上下文位置对于那些坚持 单位。

在这种情况下,整个持久性单元的部署, 包括织造(字节码转换)的持久性 类是Java EE服务器。 JDBC 数据源 通过JNDI定义 位置 meta - inf / persistence . xml 文件; EntityManager事务是集成了服务器的JTA 子系统。 春天仅仅使用获得的 会 ,将其传给应用程序对象通过依赖注入和管理 交易的持久化单元, 通常通过 JtaTransactionManager 。

如果多个持久性单元用于相同的应用程序, 该bean的jndi检索等名称应与持久性单元 持久性单元名称,应用程序使用引用它们, 例如,在 @PersistenceUnit 和 persistencecontext 注释。

LocalContainerEntityManagerFactoryBean

注意

使用这个选项在一个基于spring的完整的JPA功能 应用程序环境。 这包括诸如Tomcat web容器 以及独立的应用和集成测试 复杂的持久性的需求。

这个 LocalContainerEntityManagerFactoryBean 给 完全控制 会 配置和适合的环境中细粒度 定制是必需的。 这个 LocalContainerEntityManagerFactoryBean 创建 一个 PersistenceUnitInfo 实例的基础 在 persistence . xml 文件,提供的 dataSourceLookup 策略,指定的 LoadTimeWeaver 。 可能是这样处理 自定义数据源的JNDI和控制外的编织 过程。 下面的示例显示了一个典型的bean定义 LocalContainerEntityManagerFactoryBean :

```
<beans>

 <bean id="myEmf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="someDataSource"/>
  <property name="loadTimeWeaver">
    <bean class="org.springframework.instrument.classloading.InstrumentationLoadTimeWeaver"/>
  </property>
 </bean>

</beans>
```

下面的示例显示了一个典型 persistence . xml 文件:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">

  <persistence-unit name="myUnit" transaction-type="RESOURCE_LOCAL">
    <mapping-file>META-INF/orm.xml</mapping-file>
    <exclude-unlisted-classes/>
  </persistence-unit>

</persistence>
```

注意

这个 排除未上市类 元素总是 表明 没有 扫描带注释的实体 类应该发生,为了支持 <排除未上市类/ > 快捷方式。 这是在 线与JPA规范,这表明捷径,但 不幸的是在冲突与JPA XSD,这意味着 假 的快捷方式。 因此, <排除未上市类>假 < /排除未上市类/ > 不支持。 简单 省略了 排除未上市类 元素如果你想 实体类扫描发生。

使用 LocalContainerEntityManagerFactoryBean 是 最强大的JPA的设置选项,允许灵活的地方 配置在应用程序中。 它支持链接到现有的 JDBC 数据源 ,同时支持本地 和全球事务,等等。 然而,它也增加了 要求在运行时环境中,比如可用性的 一个编织能力类装入器如果持久性提供者的要求 字节码转换。

这个选项可能冲突与内置的JPA功能 Java EE 5服务器。 在一个完整的Java EE 5的环境,考虑获得 你 会 从JNDI。 另外,指定一个自定义 persistenceXmlLocation 在你的 LocalContainerEntityManagerFactoryBean 定义,例如,meta - inf /我的持久性。 xml,并且仅包含 一个描述符与这个名字在你的应用程序jar文件。 因为 Java EE 5服务器只有寻找违约 meta - inf / persistence . xml 文件,它忽略了这种 自定义持久性单元,从而避免冲突与一个 台弹力JPA设置前期。 (这适用于树脂3.1 例子)。

这个 LoadTimeWeaver 接口是一个 spring提供了类,允许JPA ClassTransformer 实例 插在一个特定的方式,这取决于环境是一个 web容器或应用程序服务器。 挂钩 ClassTransformers 通过一个Java 5 代理 通常是没有效率的。 代理工作反对 整个虚拟机 和检查 每一 类加载的,通常是 不良在生产服务器环境。

Spring提供了许多 LoadTimeWeaver 实现 不同的环境,从而使 ClassTransformer 实例 仅应用 每个类装入器 并不是每 VM。

当加载时编织是必需的吗?

并不是所有的JPA提供者需要JVM剂;Hibernate 是一个 例子,一个没有。 如果你的供应商不需要一个 剂或你有其他选择,比如应用增强 在构建时通过一个自定义的编译器或一个ant任务, 装载时编织器 **不应该** 是 使用。

指 一个章节弹簧configurationa 在AOP章要了解更多有关 LoadTimeWeaver 实现和他们的设置,一般性或定制 不同的平台(比如Tomcat,WebLogic,OC4J,GlassFish、树脂和JBoss)。

在上述描述的部分,您可以配置一个上下文宽 LoadTimeWeaver 使用 @EnableLoadTimeWeaving 注释的 背景:加载时间织布XML元素。 这样一个全球韦弗捡到所有JPA LocalContainerEntityManagerFactoryBeans 自动。 这是最便捷的方式建立一个装载时编织器,自动提供平台 (WebLogic,OC4J,GlassFish、Tomcat、树脂、JBoss或VM剂)和自动传播的韦弗所有韦弗知道豆:

```
<context:load-time-weaver/>
<bean id="emf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  ...
</bean>
```

然而,如果需要,可以手动指定一个专门的韦弗通过 LoadTimeWeaver 属性:

```
<bean id="emf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
 <property name="loadTimeWeaver">
  <bean class="org.springframework.instrument.classloading.ReflectiveLoadTimeWeaver"/>
 </property>
</bean>
```

无论怎样LTW配置,使用这种技术,JPA应用程序依赖 仪表可以运行在目标平台(例:Tomcat)不需要代理。 这是重要的,特别是当托管应用程序依赖于不同的JPA实现 因为JPA变压器应用只在类装入器级别,从而 彼此隔绝。

处理多个持久性单元

应用程序依赖于多个持久性单元 位置,存储在不同的罐子在类路径中,例如, 弹簧提供了 PersistenceUnitManager 作为一个 中央存储库,避免持久性单元的发现 过程,它可以是昂贵的。 默认的实现允许 多个位置的指定解析和后来恢复 通过持久性单元名称。 (默认情况下,类路径是 寻找 meta - inf / persistence . xml 文件。)

```
<bean id="pum" class="org.springframework.orm.jpa.persistenceunit.DefaultPersistenceUnitManager">
 <property name="persistenceXmlLocations">
  <list>
   <value>org/springframework/orm/jpa/domain/persistence-multi.xml</value>
   <value>classpath:/my/package/**/custom-persistence.xml</value>
   <value>classpath*:META-INF/persistence.xml</value>
  </list>
 </property>
 <property name="dataSources">
  <map>
   <entry key="localDataSource" value-ref="local-db"/>
   <entry key="remoteDataSource" value-ref="remote-db"/>
  </map>
 </property>
 <!-- if no datasource is specified, use this one -->
 <property name="defaultDataSource" ref="remoteDataSource"/>
</bean>

<bean id="emf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
 <property name="persistenceUnitManager" ref="pum"/>
 <property name="persistenceUnitName" value="myCustomUnit"/>
</bean>
```

默认的实现允许定制的 PersistenceUnitInfo 情况下, 前美联储到JPA提供者,通过声明通过它 属性, 影响 所有 主办单位,或 通过编程,通过 PersistenceUnitPostProcessor ,这 允许持久化单元的选择。 如果没有 PersistenceUnitManager 是指定的, 一是创建和内部使用 LocalContainerEntityManagerFactoryBean 。

15.5.2A实现DAOs基于普通JPA

注意

虽然 会 实例是线程安全的, EntityManager 实例不。 这个 注入JPA EntityManager 像 一个 EntityManager 获取从一个 应用程序服务器的JNDI环境所定义的那样,JPA 规范。 它代表的所有调用当前事务 EntityManager ,如果任何;否则,它 回落到一个新创建的 EntityManager 每个操作,实际上 使其使用 线程安全的。

可以编写代码对平原JPA没有任何 Spring依赖项,通过使用一个注射 会 或 EntityManager 。 弹簧可以理解 @PersistenceUnit 和 persistencecontext 注释都在 如果一个字段和方法级别 PersistenceAnnotationBeanPostProcessor 是 启用。 一个普通的JPA DAO实现使用 @PersistenceUnit 注释可能 看起来像这样:

```
public class ProductDaoImpl implements ProductDao {

   private EntityManagerFactory emf;

   @PersistenceUnit
   public void setEntityManagerFactory(EntityManagerFactory emf) {
      this.emf = emf;
   }

   public Collection loadProductsByCategory(String category) {
      EntityManager em = this.emf.createEntityManager();
      try {
```

```
            Query query = em.createQuery("from Product as p where p.category = ?1");
            query.setParameter(1, category);
            return query.getResultList();
        }
        finally {
            if (em != null) {
                em.close();
            }
        }
    }
}
```

上面的刀也不依赖春天和仍然很合身 成一个Spring应用程序上下文。 此外,DAO利用 注释需要注入的默认 会 :

```
<beans>

  <!-- bean post-processor for JPA annotations -->
  <bean class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor"/>

  <bean id="myProductDao" class="product.ProductDaoImpl"/>

</beans>
```

作为一种替代方法来定义 PersistenceAnnotationBeanPostProcessor 明确,考虑使用弹簧 背景:注释配置 XML元素在你 应用程序上下文配置。 这样做将自动注册所有 弹簧标准后处理器对基于注解的配置, 包括 CommonAnnotationBeanPostProcessor 和 等等。

```
<beans>

  <!-- post-processors for all standard config annotations -->
  <context:annotation-config/>

  <bean id="myProductDao" class="product.ProductDaoImpl"/>

</beans>
```

主要的问题是,它这样一个刀总是创建一个新的 EntityManager 通过工厂。 你 可以避免这一点,请求事务吗 EntityManager (也称为 “共享 EntityManager “因为它是一个共享的,线程安全的代理实际 事务性EntityManager)注射而不是 工厂:

```
public class ProductDaoImpl implements ProductDao {

    @PersistenceContext
    private EntityManager em;

    public Collection loadProductsByCategory(String category) {
        Query query = em.createQuery("from Product as p where p.category = :category");
        query.setParameter("category", category);
        return query.getResultList();
    }
}
```

这个 persistencecontext 注释有 可选属性 类型 ,该属性的默认值 PersistenceContextType.TRANSACTION 。 这个默认是你需要获得一个共享EntityManager代理。 另一种, PersistenceContextType.EXTENDED ,是一个完全 不同的事情:这个结果在一个所谓的扩展EntityManager, 这是 不是线程安全的 ,因此不能使用 在并发访问的组件(比如spring管理单例 bean。 扩展 EntityManagers只应该用于有状态 组件,例如,驻留在一个会话,与生命周期的 EntityManager没有绑定到当前事务而是被 完全依赖于应用程序。

注入的 EntityManager 是 spring管理(意识到正在进行的事务)。 它是重要的 注意,尽管新DAO实现使用方法级别 注入一个 EntityManager 而不是 一个 会 ,没有变化 需要在应用程序上下文XML由于注释的使用。

这把刀的主要优势是,它只风格取决于 Java持久化API;没有进口弹簧类是必需的。 此外,正如JPA注释是理解,这种注射 应用自动由Spring容器。 这是吸引人的从一个 具有非侵袭性的角度来看,和可能会感觉更自然的JPA 开发人员。

方法和字段级注入

注释表明依赖注入(如 @PersistenceUnit 和 persistencecontext)可以应用于现场或 方法在类的内部,因此表达式 方法级 注入 和 字段级注入 。 字段级注释是简洁和更容易使用,同时方法级允许进一步加工的注入依赖项。 在这两种情况下成员可见性(公共的、受保护的、私有的)做 并不重要。

类级注释呢?

在Java EE 5平台,它们用于依赖 声明,而不是资源注入。

15.5.3A事务管理

注意

你是 强烈 鼓励阅读 SectionA 12.5,一个managementa声明式事务 如果你还没有这么做,以获得一个 更详细的报道Spring的声明式事务 支持。

在交易执行服务操作,您可以使用 春天的共同声明式事务设施。 例如:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop.xsd">

  <bean id="myTxManager" class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="myEmf"/>
  </bean>

  <bean id="myProductService" class="product.ProductServiceImpl">
    <property name="productDao" ref="myProductDao"/>
  </bean>

  <aop:config>
    <aop:pointcut id="productServiceMethods" expression="execution(* product.ProductService.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="productServiceMethods"/>
  </aop:config>

  <tx:advice id="txAdvice" transaction-manager="myTxManager">
    <tx:attributes>
      <tx:method name="increasePrice*" propagation="REQUIRED"/>
      <tx:method name="someOtherBusinessMethod" propagation="REQUIRES_NEW"/>
      <tx:method name="*" propagation="SUPPORTS" read-only="true"/>
    </tx:attributes>
  </tx:advice>

</beans>
```

春天JPA允许配置 JpaTransactionManager 揭露一个JPA事务 JDBC访问代码访问相同的JDBC 数据源 ,只要注册 JpaDialect 支持检索的 底层JDBC 连接 。 出 箱,Spring提供了方言的Toplink,Hibernate和OpenJPA JPA 实现。 见下一节详细讨论 JpaDialect 机制。

15.5.4A JpaDialect

作为一个高级功能 JpaTemplate , JpaTransactionManager 和子类的 AbstractEntityManagerFactoryBean 支持自定义 JpaDialect ,被传递到 JpaDialect bean属性。 在这种情况下, DAOs没有得到 会 参考而是 一个完整的 JpaTemplate 实例(例如, 通过 到 JpaTemplate 财产 的 JpaDaoSupport)。 一个 JpaDialect 实现可以使一些高级功能支持的春天, 通常在一个特定于供应商的方式:

- 运用特定的事务语义如定制 隔离级别或事务超时)
- 检索事务JDBC 连接 因接触到基于jdbc的 DAOs)
- 高级翻译的 PersistenceExceptions 春天 DataAccessExceptions

这是特别有价值的特殊事务语义 和高级翻译的异常。 默认实现 使用(DefaultJpaDialect)不提供任何 特殊的功能,如果上面的功能是必需的,你必须 指定适当的方言。

看到 JpaDialect Javadoc更多 详细的操作和怎样使用它们在春天的JPA 支持。

15.6一个iBATIS SQL映射

iBATIS支持Spring框架多像JDBC 支持,它支持相同的模板样式编程,和作为 与JDBC和其他ORM技术支持工作的iBATIS 春天的异常层次结构,让您享受Spring的IoC 特性。

事务管理可以通过Spring的标准处理 设施。 没有特殊的事务策略是必要的,为iBATIS, 因为没有特别的事务性资源涉及其他比JDBC 连接 。 因此,春天的标准JDBC DataSourceTransactionManager 或 JtaTransactionManager 非常 足够的。

注意

Spring支持iBATIS 2. x。 iBATIS 1。 x支持类是没有 不再提供。

15.6.1A设置 SqlMapClient

使用iBATIS SQL地图涉及创建SqlMap配置文件 包含语句和结果的地图。 春天负责装货 那些使用 SqlMapClientFactoryBean 。 为 的例子,我们将使用以下 帐户 类:

```
public class Account {

    private String name;
    private String email;

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return this.email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

到地图这 帐户 类和iBATIS 2。 x我们 需要创建以下SQL映射 帐户xml :

```
<sqlMap namespace="Account">

  <resultMap id="result" class="examples.Account">
    <result property="name" column="NAME" columnIndex="1"/>
    <result property="email" column="EMAIL" columnIndex="2"/>
  </resultMap>

  <select id="getAccountByEmail" resultMap="result">
    select ACCOUNT.NAME, ACCOUNT.EMAIL
    from ACCOUNT
    where ACCOUNT.EMAIL = #value#
  </select>

  <insert id="insertAccount">
    insert into ACCOUNT (NAME, EMAIL) values (#name#, #email#)
  </insert>

</sqlMap>
```

配置文件为iBATIS 2看起来像这样:

```
<sqlMapConfig>

  <sqlMap resource="example/Account.xml"/>

</sqlMapConfig>
```

记住,iBATIS加载资源从类路径,所以要 确定添加 帐户xml 文件类 路径。

我们可以使用 SqlMapClientFactoryBean 在 Spring容器。 请注意,使用iBATIS SQL地图2。 x,JDBC 数据源 通常是指定的吗 SqlMapClientFactoryBean ,使懒惰 加载。 这是配置需要这些bean 定义:

```
<beans>

  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
  </bean>

  <bean id="sqlMapClient" class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
    <property name="configLocation" value="WEB-INF/sqlmap-config.xml"/>
    <property name="dataSource" ref="dataSource"/>
  </bean>

</beans>
```

15.6.2A使用 SqlMapClientTemplate 和 SqlMapClientDaoSupport

这个 SqlMapClientDaoSupport 类提供了一个 支持类相似 SqlMapDaoSupport 。 我们把它实现我们的道:

```
public class SqlMapAccountDao extends SqlMapClientDaoSupport implements AccountDao {

    public Account getAccount(String email) throws DataAccessException {
        return (Account) getSqlMapClientTemplate().queryForObject("getAccountByEmail", email);
    }

    public void insertAccount(Account account) throws DataAccessException {
        getSqlMapClientTemplate().update("insertAccount", account);
    }
}
```

在DAO,我们使用预配置的 SqlMapClientTemplate 执行查询, 在设置好 SqlMapAccountDao 在 应用程序上下文和布线与我们的 SqlMapClient 实例:

```
<beans>

  <bean id="accountDao" class="example.SqlMapAccountDao">
    <property name="sqlMapClient" ref="sqlMapClient"/>
  </bean>

</beans>
```

一个 SqlMapTemplate 实例还可以 手动创建,传递 SqlMapClient 作为 构造函数参数。 这个 SqlMapClientDaoSupport 基地 preinitializes类只是一个 SqlMapClientTemplate 实例对我们。

这个 SqlMapClientTemplate 提供了一种通用的 执行 法,把一个自定义的 SqlMapClientCallback 实现作为参数。 这 可以,例如,用于批处理:

```
public class SqlMapAccountDao extends SqlMapClientDaoSupport implements AccountDao {

    public void insertAccount(Account account) throws DataAccessException {
        getSqlMapClientTemplate().execute(new SqlMapClientCallback() {
            public Object doInSqlMapClient(SqlMapExecutor executor) throws SQLException {
                executor.startBatch();
                executor.update("insertAccount", account);
                executor.update("insertAddress", account.getAddress());
                executor.executeBatch();
            }
        });
    }
}
```

一般来说,任何操作组合所提供的本地 SqlMapExecutor API可以用于这样一个回调。 任何抛出 SQLException 自动转换 Spring的通用 DataAccessException 层次结构。

15.6.3A实现DAOs基于普通iBATIS API

DAOs可以编写对平原iBATIS API,没有任何 Spring依赖项,直接使用注射 SqlMapClient 。 下面的示例显示了一个 相应的DAO 实现:

```
public class SqlMapAccountDao implements AccountDao {

    private SqlMapClient sqlMapClient;

    public void setSqlMapClient(SqlMapClient sqlMapClient) {
        this.sqlMapClient = sqlMapClient;
    }

    public Account getAccount(String email) {
        try {
            return (Account) this.sqlMapClient.queryForObject("getAccountByEmail", email);
        }
        catch (SQLException ex) {
            throw new MyDaoException(ex);
        }
    }

    public void insertAccount(Account account) throws DataAccessException {
        try {
            this.sqlMapClient.update("insertAccount", account);
        }
        catch (SQLException ex) {
            throw new MyDaoException(ex);
        }
    }
}
```

在这个场景中,您需要处理 SQLException 抛出的iBATIS API定制 时尚,通常通过包装它在自己的特定于应用程序的刀 例外。

连接在应用程序上下文仍然看起来像它 并在示例的吗 SqlMapClientDaoSupport , 由于这样的事实,即普通刀仍然遵循了ibatis的基础 依赖注入模式:

```
<beans>

  <bean id="accountDao" class="example.SqlMapAccountDao">
    <property name="sqlMapClient" ref="sqlMapClient"/>
  </bean>

</beans>
```

16。 一个编组XML使用O / X映射器

16.1一个介绍

在这一章,我们将介绍Spring的对象/ XML映射支持。 对象/ XML映射,或O / X映射 短,是指将一个XML文档,从一个对象。 这个转换过程也 被称为XML编组或XML序列化。 本章使用这些术语可以互换使用。

该领域内的O / X映射,一个 Marshaller 负责序列化一个吗 对象(图)XML。 以类似的方式,一个 解组程序 反序列化XML的 对象图。 这个XML可以把形式的DOM文档,一个输入或输出流,或一个SAX处理程序。

一些使用Spring的好处对你的O / X映射需求:

易于配置一个 春天的bean工厂很容易marshallers配置,而不需要构建JAXB上下文, JiBX绑定工厂,等marshallers可以配置为在您的应用程序的任何其他bean 上下文。 此外,XML的基于配置可用于许多marshallers,使 配置更加简单。

一致的接口 Spring的O / X映射操作通过两个全球接口: Marshaller 和 解组程序 接口。 这些抽象允许你切换O / X映射 框架相对轻松地,很少或根本没有变化需要在类的 编组。 这种方法还有其他的好处使我们能够完成XML编组与 一个混搭的方法(例如一些编组进行,其他使用JAXB使用XMLBeans) 非侵入性的方式,利用每个技术的力量。

一致的异常层次结构一个 Spring提供了一个转换从异常从底层O / X映射工具来自己的异常 层次与 XmlMappingException 作为根异常。 可以预期, 这些运行时异常包装原始异常所以不丢失信息。

16.2一个信号员和解组程序

所述的介绍,一个 Marshaller 序列化一个对象到XML,一个 解组程序 反序列化XML流到一个对象。 在本节中,我们将描述 这两个弹簧接口用于此目的。

16.2.1A Marshaller

春天所有编组操作背后的抽象 org.springframework.oxm.Marshaller 接口,主要的方法 下面列出的是。

```
public interface Marshaller {

  /**
   * Marshals the object graph with the given root into the provided Result.
   */
  void marshal(Object graph, Result result)
      throws XmlMappingException, IOException;
}
```

这个 Marshaller 接口有一个主要方法,统帅给定的 对象到一个给定的 javax.xml.transform.Result 。 结果是一个标签 界面,基本上代表了一个XML输出抽象:混凝土实现包装各种XML 表示,这显示在下表中。

结果 实现	封装XML表示
DOMResult	org w3c dom节点
SAXResult	org xml sax contenthandler
StreamResult	java输入输出文件 , java io outputstream ,或 java io作家

注意

虽然 元帅() 方法接受一个普通对象作为它的第一个 参数,最 Marshaller 实现不能处理任意 对象。 相反,一个对象类必须映射到一个映射文件,标有一个注释, 注册信号员,或有一个共同的基类。 参考进一步部分 在这

一章来确定你选择的O / X技术管理这个。

16.2.2A解组程序

类似 Marshaller ,有 org.springframework.oxm.Unmarshaller 接口。

```
public interface Unmarshaller {

    /**
     * Unmarshals the given provided Source into an object graph.
     */
    Object unmarshal(Source source)
        throws XmlMappingException, IOException;
}
```

这个界面也有一个方法,该方法从给定的读取 javax.xml.transform.Source (一个XML输入抽象),并返回 对象读取。 与结果,源是一个标记接口,有三个具体的实现。 每个 包装不同的XML表示,显示在下表中。

源 实现	封装XML表示
DOMSource	org w3c dom节点
SAXSource	org xml sax inputsource ,和 org xml sax xmlreader
StreamSource	java输入输出文件 , java io inputstream ,或 java io读者

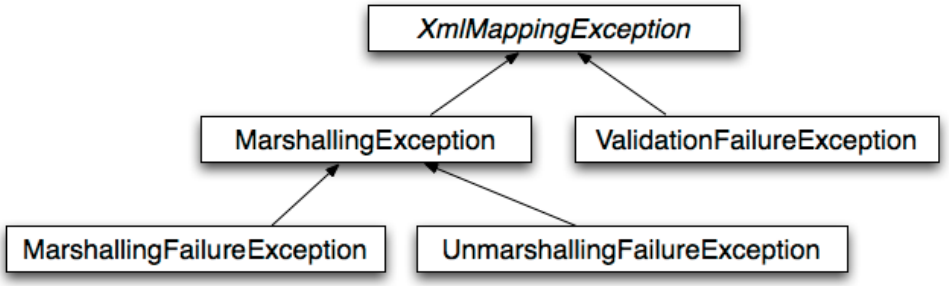
尽管有两个单独的编组接口(Marshaller 和 解组程序),所有实现发现在春天ws实现在 一个类。 这意味着您可以连接一个编组器类和引用它既是marshaller和一个 解组程序在你的 中 。

16.2.3A XmlMappingException

春天将异常从底层O / X映射工具自身的异常层次结构 XmlMappingException 作为根异常。 可以预期,这些运行时 异常包装原始异常所以不会丢失任何信息。

此外, MarshallingFailureException 和 UnmarshallingFailureException 提供区分编组和 解组操作,即使底层O / X映射工具并不这样做。

O / X映射显示异常层次结构如下图所示:



O / X映射异常层次结构

16.3一个使用信号员和解组程序

春天的OXM可以用于各种各样的情况。 在接下来的例子中,我们将使用它 元帅的设置一个spring管理应用程序作为一个XML文件。 我们将使用一个简单的JavaBean来 代表设置:

```
public class Settings {
    private boolean fooEnabled;

    public boolean isFooEnabled() {
        return fooEnabled;
    }

    public void setFooEnabled(boolean fooEnabled) {
        this.fooEnabled = fooEnabled;
    }
}
```

应用程序类使用这个bean来存储它的设置。 除了一个主要方法,类有两个 方法: 储存设定() 保存设置bean到文件命名 设置xml

,和 loadSettings() 再次加载这些设置。 一个 main() 方法构造一个Spring应用程序上下文,并调用这两个方法。

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.oxm.Marshaller;
import org.springframework.oxm.Unmarshaller;

public class Application {
   private static final String FILE_NAME = "settings.xml";
   private Settings settings = new Settings();
   private Marshaller marshaller;
   private Unmarshaller unmarshaller;

   public void setMarshaller(Marshaller marshaller) {
      this.marshaller = marshaller;
   }

   public void setUnmarshaller(Unmarshaller unmarshaller) {
      this.unmarshaller = unmarshaller;
   }

   public void saveSettings() throws IOException {
      FileOutputStream os = null;
      try {
         os = new FileOutputStream(FILE_NAME);
         this.marshaller.marshal(settings, new StreamResult(os));
      } finally {
         if (os != null) {
            os.close();
         }
      }
   }

   public void loadSettings() throws IOException {
      FileInputStream is = null;
      try {
         is = new FileInputStream(FILE_NAME);
         this.settings = (Settings) this.unmarshaller.unmarshal(new StreamSource(is));
      } finally {
         if (is != null) {
            is.close();
         }
      }
   }

   public static void main(String[] args) throws IOException {
      ApplicationContext appContext =
         new ClassPathXmlApplicationContext("applicationContext.xml");
      Application application = (Application) appContext.getBean("application");
      application.saveSettings();
      application.loadSettings();
   }
}
```

这个 应用 同时需要 Marshaller 和 解组程序 属性设置。我们可以使用以下 中 :

```
<beans>
   <bean id="application" class="Application">
      <property name="marshaller" ref="castorMarshaller" />
      <property name="unmarshaller" ref="castorMarshaller" />
   </bean>
   <bean id="castorMarshaller" class="org.springframework.oxm.castor.CastorMarshaller"/>
</beans>
```

这个应用程序上下文使用Castor,但是我们可以使用任何其他marshaller实例描述 在本章后面。 注意,不需要任何进一步的Castor配置默认值,所以bean 的定义是相当简单的。 还请注意, 使用CastorMarshaller 实现两 Marshaller 和 解组程序 ,所以我们可以参考 到 使用CastorMarshaller bean都 Marshaller 和 解组程序 应用程序的属性。

这个示例应用程序将生成以下 设置xml 文件:

```
<?xml version="1.0" encoding="UTF-8"?>
<settings foo-enabled="false"/>
```

16.4基于XML配置

Marshallers可以配置更简明地使用标签从OXM名称空间。 让这些标签可用,适当的模式必须首先在导言引用的XML配置文

件。 注意 “oxm相关文本如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:oxm="http://www.springframework.org/schema/oxm"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans.xsd
   http://www.springframework.org/schema/oxm
   http://www.springframework.org/schema/oxm/spring-oxm.xsd">
```

目前,下列标签可用:

- jaxb2-marshaller
- xmlbeans marshaller
- 蓖麻marshaller
- jibx marshaller

每个标签将被解释在其各自的信号员的部分。 作为一个例子虽然,这里就是 配置一个JAXB2 marshaller看起来像:

```
<oxm:jaxb2-marshaller id="marshaller" contextPath="org.springframework.ws.samples.airline.schema"/>
```

JAXB 16.5

JAXB绑定编译器转换一个W3C XML Schema为一个或多个Java类,一个 jaxb属性 文件,以及可能出现的一些资源文件。 JAXB也提供一个 的方法来产生一个带注释的Java类的模式。

Spring支持JAXB 2.0 API作为XML编组策略,遵循 Marshaller 和 解组程序 接口描述 SectionA 16.2,一个Marshaller和Unmarshallera 。 相应的集成 类居住在 org.springframework.oxm.jaxb 包。

16.5.1A Jaxb2Marshaller

这个 Jaxb2Marshaller 类实现两个春天 Marshaller 和 解组程序 接口。 它 需要一个上下文路径操作,您可以设置使用 contextPath 财产。 上下文路径是一个列表的冒号(:)分隔的Java包名称中包含模式 派生类。 它还提供了一个 classesToBeBound 属性,它允许您设置一个数组的 类是由marshaller。 执行模式验证通过指定一个或多个 模式的资源到bean,就像这样:

```
<beans>

   <bean id="jaxb2Marshaller" class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
      <property name="classesToBeBound">
         <list>
            <value>org.springframework.oxm.jaxb.Flight</value>
            <value>org.springframework.oxm.jaxb.Flights</value>
         </list>
      </property>
      <property name="schema" value="classpath:org/springframework/oxm/schema.xsd"/>
   </bean>
   ...

</beans>
```

XML的基于配置

这个 jaxb2-marshaller 标签配置 org.springframework.oxm.jaxb.Jaxb2Marshaller 。 这里是一个例子:

```
<oxm:jaxb2-marshaller id="marshaller" contextPath="org.springframework.ws.samples.airline.schema"/>
```

另外,列表的类来绑定可以提供给信号员通过 类一定 子标记:

```
<oxm:jaxb2-marshaller id="marshaller">
   <oxm:class-to-be-bound name="org.springframework.ws.samples.airline.schema.Airport"/>
   <oxm:class-to-be-bound name="org.springframework.ws.samples.airline.schema.Flight"/>
   ...
</oxm:jaxb2-marshaller>
```

可用的属性是:

属性	描述	需要

id	信号员的id	没有
contextPath	JAXB上下文路径	没有

16.6一个蓖麻

Castor的XML映射是一个开源的XML绑定框架。 它允许您将数据包含在 一个java对象模型到/从一个XML文档。 默认情况下,它不需要任何进一步的配置, 尽管一个映射文件可以用来更好地控制Castor的行为。

在Castor的更多信息,请参阅 蓖麻网站 。 Spring集成类位于 org.springframework.oxm.castor 包。

16.6.1A使用CastorMarshaller

与JAXB, 使用CastorMarshaller 实现两 Marshaller 和 解组程序 接口。 它可以连接如下:

```
<beans>

  <bean id="castorMarshaller" class="org.springframework.oxm.castor.CastorMarshaller" />
  ...

</beans>
```

16.6.2A映射

尽管可以依靠Castor的默认编组的行为,它可能是必要的 更多的控制它。 这可以通过使用Castor映射文件。 要了解更多信息,请参考 到 Castor的XML映射 。

映射可以设置使用 mappingLocation 资源属性,指示 下面用一个类路径的资源。

```
<beans>
  <bean id="castorMarshaller" class="org.springframework.oxm.castor.CastorMarshaller" >
    <property name="mappingLocation" value="classpath:mapping.xml" />
  </bean>
</beans>
```

XML的基于配置

这个 蓖麻marshaller 标签配置 org.springframework.oxm.castor.CastorMarshaller 。 这里是一个例子:

```
<oxm:castor-marshaller id="marshaller" mapping-location="classpath:org/springframework/oxm/castor/mapping.xml"/>
```

marshaller实例可以被配置在两个方面,通过指定的位置,要么 一个映射文件(通过 映射位置 属性),或通过 识别Java pojo(通过 目标类 或 目标包 属性),存在相应的 XML描述符类。 后者的方式通常是结合使用XML代码生成 从XML模式。

可用的属性是:

属性	描述	需要
id	信号员的id	没有
编码	编码用于从XML数据分解	没有
目标类	一个Java类名称为POJO的XML类描述符是可用的(如 通过生成代码生成)	没有
目标包	一个Java包名称,标识一个包,包含pojo和他们的 相应蓖麻 XML描述符类(通过代码生成生成从XML模式)	没有
映射位置	位置的XML映射文件Castor	没有

16.7一个XMLBeans

XMLBeans是XML绑定工具,有完整的XML模式的支持,并提供完整的XML Infoset 保真度。 它采用了不同的方法,大多数其他的O / X映射框架,在那 所有的类,生成一个XML模式都是来自 XmlObject ,包含XML绑定信息。

在XMLBeans的更多信息,请参阅 XMLBeans web站点 。 这个春天ws集成类驻留 在 org.springframework.oxm.xmlbeans 包。

16.7.1A XmlBeansMarshaller

这个 XmlBeansMarshaller 实现两 Marshaller 和 解组程序 接口。 它可以配置如下:

```
<beans>
  <bean id="xmlBeansMarshaller" class="org.springframework.oxm.xmlbeans.XmlBeansMarshaller" />
  ...
</beans>
```

注意

注意, XmlBeansMarshaller 只能元帅类型的对象吗 XmlObject , 并不是每个 java . lang . object 。

XML的基于配置

这个 xmlbeans marshaller 标签配置 org.springframework.oxm.xmlbeans.XmlBeansMarshaller 。 这里是一个例子:

```
<oxm:xmlbeans-marshaller id="marshaller"/>
```

可用的属性是:

属性	描述	需要
id	信号员的id	没有
选项	bean的名称XmlOptions也被用于这个信号员。 通常一个 XmlOptionsFactoryBean 定义	没有

JiBX 16.8

JiBX框架提供了一个解决方案类似JDO提供了ORM:绑定定义定义了 规则Java对象如何从XML转换为或。 在准备绑定和编译类,一个JiBX绑定编译器提高类文件,添加代码来处理转换的实例 从类或XML。

在JiBX的更多信息,请参阅 JiBX网站 。 Spring集成类位于 org.springframework.oxm.jibx 包。

16.8.1A JibxMarshaller

这个 JibxMarshaller 类实现两 Marshaller 和 解组程序 接口。 操作,它需要类的名称,在,你可以元帅设置使用 targetClass 财产。 可选地,您可以设置绑定名称使用 bindingName 财产。 在下一个示例,我们绑定 航班 类:

```
<beans>
  <bean id="jibxFlightsMarshaller" class="org.springframework.oxm.jibx.JibxMarshaller">
    <property name="targetClass">org.springframework.oxm.jibx.Flights</property>
  </bean>

  ...
```

一个 JibxMarshaller 是配置为一个类。 如果你想元帅 多个类,您必须配置多个 JibxMarshaller 年代 不同 targetClass 属性值。

XML的基于配置

这个 jibx marshaller 标签配置 org.springframework.oxm.jibx.JibxMarshaller 。 这里是一个例子:

```
<oxm:jibx-marshaller id="marshaller" target-class="org.springframework.ws.samples.airline.schema.Flight"/>
```

可用的属性是:

属性	描述	需要
id	信号员的id	没有
目标类	这个信号员的目标类	是的
bindingName	绑定使用的名称这信号员	没有

16.9一个XStream

是一个简单的图书馆XStream序列化对象到XML,然后再返回。 它不需要任何映射,和 生成干净的XML。

XStream的更多信息,请参阅 XStream网站 。 Spring集成类位于 org.springframework.oxm.xstream 包。

16.9.1A XStreamMarshaller

这个 XStreamMarshaller 不需要任何配置,可以配置吗 在一个应用程序上下文直接。 为了进一步定制XML,您可以设置一个 别名映射 ,包括字符串别名映射到类:

```
<beans>

  <bean id="xstreamMarshaller" class="org.springframework.oxm.xstream.XStreamMarshaller">
    <property name="aliases">
      <props>
        <prop key="Flight">org.springframework.oxm.xstream.Flight</prop>
      </props>
    </property>
  </bean>
  ...

</beans>
```

警告

默认情况下,XStream允许任意类来将它们分散,从而导致安全 漏洞。 因此,建议设置 supportedClasses 产权 XStreamMarshaller ,就像这样:

```
<bean id="xstreamMarshaller" class="org.springframework.oxm.xstream.XStreamMarshaller">
  <property name="supportedClasses" value="org.springframework.oxm.xstream.Flight"/>
  ...
</bean>
```

这将确保只有注册类资格解组。

此外,您可以注册 自定义转换器 确保只有您可以将它们分散支持类。

注意

注意,XStream是一个XML序列化库,而不是一个数据绑定库。 因此,它有 名称空间支持有限。 因此,它是相当不适合使用在Web服务。

PartA V。 一个Web

这部分的参考文档涵盖了春天 框架的支持表示层(和特别 基于网络的表示层)。

Spring框架的web框架, Spring Web MVC 覆盖在第一夫妇的 章。 一个数量的剩余的章节的部分 参考文档涉及Spring框架的集成与其他web技术,例如 Struts 和 JSF (名字,但两个)。

这部分包括覆盖Spring的MVC portlet框架 。

- ChapterA 17, Web MVC框架
- ChapterA 18, 视图技术
- ChapterA 19, 与其他web框架集成
- ChapterA 20, Portlet MVC框架

17。 Web MVC框架

17.1一个介绍Spring Web MVC框架

Spring的Web模型-视图-控制器(MVC)框架设计 围绕一个 DispatcherServlet 分派请求, 到处理程序,可配置的处理程序映射、视图解析、语言环境 和主题解析以及支持上传文件。 默认 处理程序是基于 controller 和 @RequestMapping 注释,提供一个

广泛的灵活的处理方法。 通过引入弹簧 3.0, controller 机制还允许 你创建RESTful Web站点和应用程序,通过 @PathVariable 注释和其他 特性。

在Spring Web MVC您可以使用任何对象作为命令或 形式支持对象;您不需要实现一个特定于框架的 接口或基类。 春天的数据绑定是高度灵活的: 对 的例子,它将作为验证错误类型不匹配,可以 评估应用程序,而不是系统错误。 因此你不需要 复制你的业务对象的属性作为简单、无类型的字符串 你的表单对象仅仅处理无效提交,或转换 正确的字符串。 相反,它通常比直接结合到你的 业务对象。

春天的视图解析是非常灵活的。 一个 控制器 通常负责 准备模型 地图 与数据和选择 视图名称但它也可以直接写响应流和 完成请求。 视图名称解析是高度可配置的通过 文件扩展名或接受标题内容类型的谈判,通过bean 的名字,一个属性文件,甚至是一个自定义的 ViewResolver 实现。 模型(米在MVC)是一个 地图 接口,它允许 对于完整的抽象视图技术。 你可以整合基于直接与模板渲染技术如JSP、速度 和Freemarker,或直接生成XML、JSON、原子,和许多其他类型 的内容。 该模型 地图 只是 变成了一个适当的格式,比如JSP请求属性,一个 Velocity模板模型。

一个 对扩展开放..... 一个

一个关键的设计原则在Spring Web MVC和在春天一般 是 一个 对扩展开放、关闭 修改 一个原理。

一些方法在Spring Web MVC核心类的标记 最后 。 作为一名开发人员你不能覆盖这些 方法来提供自己的行为。 这个还没有做任意, 但是具体该原理在心里。

这一原则的解释,请参考 专家 Spring Web MVC和网络流量 Ladd和其他由赛斯; 特别是看到节”一看设计,”在117页的 第一个版本。 另外,看到

1. 鲍勃 马丁,开放闭合原则(PDF)

你不能添加建议最后的方法当你使用Spring MVC。 例如,您不能添加建议 AbstractController.setSynchronizeOnSession() 法。 指 SectionA 9 6 1,一个理解AOP proxiesa 更多 信息在AOP代理和为什么你不能添加建议最后 方法。

17.1.1A特性的Spring Web MVC

Spring的web模块包括许多独特的网络支持 特点:

- 明确分工 。 一个每个角色 控制器,验证器,命令对象,形成对象,模型对象, DispatcherServlet ,处理程序映射视图 解析器,等等一个能否实现由一个专门的 对象。
- 强大的和简单的配置两个 框架和应用程序类作为JavaBeans 。 这 配置功能包括简单引用跨语境, 如从web控制器到业务对象 验证器。
- 适应性、非侵入性、和 的灵活性。 定义任何控制器方法签名你 需要,可能使用一个参数注释(如 @RequestParam,@RequestHeader,@PathVariable,和更多)对于一个给定的 场景。
- 可重用的业务代码 , 不需要 对于重复 。 使用现有的业务对象作为命令 或形成对象而不是镜像他们扩展特定的 框架的基类。
- 定制绑定和验证 。 类型 应用程序级验证不匹配是错误,保持 冒犯价值,本地化的日期和号码绑定,等等 而不是字符串只有形式对象与手工解析和 转换为业务对象。
- 可定制的处理程序映射和视图 决议 。 处理程序映射和视图解析 策略的范围从简单的基于url的配置,来 复杂的,专门解决战略。 春天是多web MVC框架,要灵活授权一个特定的 技术。
- 灵活的模型转移 。 模型转移 使用一个名称/值 地图 支持容易 集成与任何视图技术。
- 可定制的语言环境和主题的分辨率,支持 有或没有春天的jsp标记库,支持JSTL, 支持速度而不需要额外的桥梁,所以 在。
- 一个简单而强大的JSP标记库称为 Spring标记库提供支持功能,如数据 绑定和主题 。 自定义标签允许最大 灵活性方面的标记代码。 标签上的信息 库描述符,请参见附录资格 AppendixA G, 弹簧tld
- 一个JSP表单标记库,介绍了Spring 2.0中, 使得编写形式在JSP页面更容易。 对于 标记库描述符的信息,请参阅附录资格 AppendixA H, 弹簧形式tld
- bean的生命周期的范围仅仅是当前HTTP 请求或HTTP 会话 。 这不是一个特定的功能Spring MVC的本身,而是 这个 WebApplicationContext 容器(s),使用Spring MVC。 这些bean范围描述 在 SectionA 5 5 4,一个请求、会话和scopesa 全球会议

Spring Web Flow

Spring Web Flow(SWF)的目标是成为最好的解决方案的管理 web应用程序的页面流。

SWF集成了现有的框架(如Spring MVC,Struts, JSF,在两个servlet和portlet的环境。 如果你有一个业务 过程(或进程),将受益于一个对话模型 不是一个纯粹的请求模型,然后SWF可能解决方案。

SWF允许您捕获逻辑页面流作为独立的模块 在不同的情况下,是可重用的,因此是理想的建筑web应用程序模块,指导用户通过导航控制 驱动的业务流程。

主权财富基金的更多信息,请参考 Spring Web Flow网站 。

插拔性17.1.2A其他MVC实现

实现比非spring MVC为一些项目。 许多团队希望利用他们现有的投资技巧和 工具。 大量的知识和经验存在Struts 框架。 如果你可以忍受Struts的体系结构的缺陷,它可以是一个 可行的选择web层;同样的适用于网络系统和其他 web MVC框架。

如果你不想使用Spring的web MVC,但打算利用 其他的解决方案,弹簧提供,您可以将web MVC 您所选择的框架与弹簧容易。

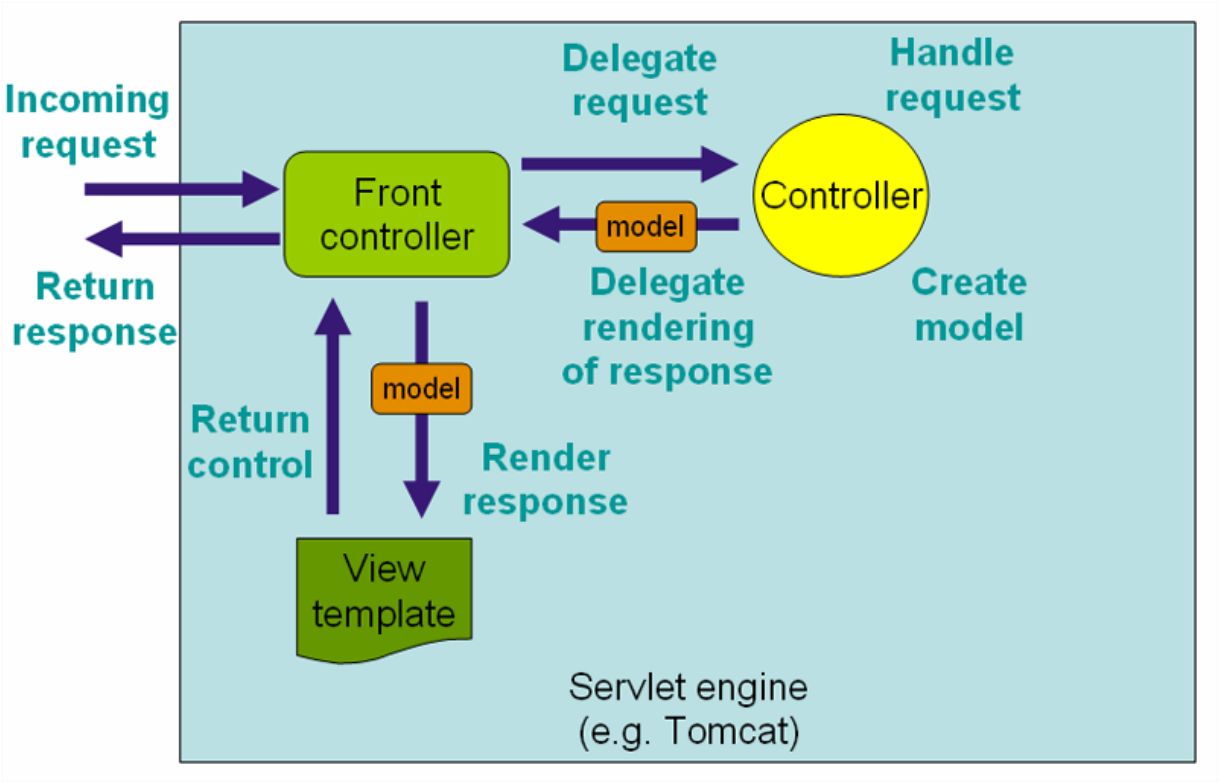
简单地启动一个春天 根应用程序上下文通过它 ContextLoaderListener ,来访问它 它的 ServletContext 属性(或弹簧的 各自的助手方法)在Struts或网络系统动作。 没有 “插件”是相关的,所以没有专门的集成是必要的。 从 web层的角度来看,您只需使用弹簧作为一个图书馆, 根应用程序上下文实例作为入口点。

你注册的bean和Spring的服务可以在你 指尖即使没有春天的Web MVC。 春天不竞争 Struts或网络系统在这种情况下。 它只是解决了许多地区 那纯web MVC框架不,从bean配置数据 访问和事务处理。 所以你可以丰富你的应用程序 一个春天的中间层和/或数据访问层,即使你只是想 使用,例如,事务抽象与JDBC或 Hibernate。

17.2一个的 DispatcherServlet

Spring的web MVC框架是,像许多其他web MVC框架, 请求驱动,围绕一个中心Servlet,分派请求 对控制器和提供其他功能,促进了 开发web应用程序。 春天的 DispatcherServlet 然而,不仅仅是 那。 它完全集成Spring IoC容器和作为 这样允许您使用其他功能,春天。

请求处理工作流的Spring Web MVC DispatcherServlet 见下面的吗 图。 读者会认得的模式精明的 DispatcherServlet 是一个表达的吗 一个 前端控制器 一个 设计模式(这是一个模式 Spring Web MVC股票与其他许多领先的Web框架)。



请求处理工作流在Spring Web MVC(高 级别)

这个 DispatcherServlet 是一个实际的 Servlet (它继承自 HttpServlet 基类),因此是宣布 这个 web . xml 您的web应用程序。 你需要地图 请求你要 DispatcherServlet 到 处理,通过使用URL映射在相同的 web . xml 文件。 这是标准Java EE Servlet配置; 下面的例子 显示了这样一个 DispatcherServlet 声明和 映射:

```
<web-app>

  <servlet>
    <servlet-name>example</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>example</servlet-name>
    <url-pattern>/example/*</url-pattern>
  </servlet-mapping>

</web-app>
```

在前面的示例中,所有请求开始 /例子 会处理吗 DispatcherServlet 实例命名 例子 。 在一个Servlet 3.0 +环境,你也有 选项配置Servlet容器以编程方式。 下面是代码 相当于上述基础 web . xml 示例:

```
public class MyWebApplicationInitializer implements WebApplicationInitializer {
```

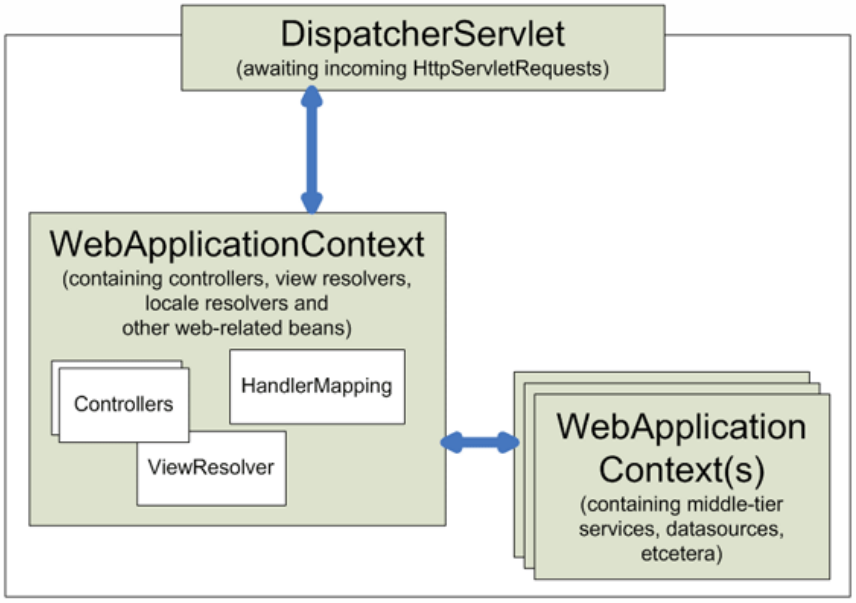
```
    @Override
    public void onStartup(ServletContext container) {
        ServletRegistration.Dynamic registration = container.addServlet("dispatcher", new DispatcherServlet());
        registration.setLoadOnStartup(1);
        registration.addMapping("/example/*");
    }

}
```

WebApplicationInitializer 是一个接口 Spring MVC提供确保你的基于代码的配置是发现和 自动使用初始化任何Servlet 3容器。 一个抽象的基类 实现这interace命名 AbstractDispatcherServletInitializer 使得它更容易 登记 DispatcherServlet 通过简单地指定 它的servlet映射。 看到 基于代码的Servlet容器初始化 为更多的细节。

以上仅是第一步在设置 Spring Web MVC。 你 现在需要配置各种bean使用的Spring Web MVC 框架(超越 DispatcherServlet 本身)。

详细的 SectionA 5.14,一个附加的功能 ApplicationContext 一个 , ApplicationContext 实例在春天可以 是作用域。 在Web MVC框架,每个 DispatcherServlet 有自己的 WebApplicationContext ,它继承了所有 已经定义的bean在根 WebApplicationContext 。 这些遗传 豆子可以覆盖在servlet特定的范围,您可以定义 新范围特定bean本地到给定的Servlet 实例。



上下文层次在Spring Web MVC

在初始化的 DispatcherServlet , Spring MVC查找一个文件命名 [servlet - name]servlet xml 在 - inf 您的web应用程序的目录并创建 定义的bean,覆盖任何bean定义的定义 具有相同名称的在全球范围内。

考虑以下 DispatcherServlet Servlet配置(在 web . xml 文件):

```
<web-app>

    <servlet>
        <servlet-name>golfing</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>golfing</servlet-name>
        <url-pattern>/golfing/*</url-pattern>
    </servlet-mapping>

</web-app>
```

与上面的Servlet配置到位,你 需要一个文件叫什么 / - inf / **高尔夫** servlet xml 在你的 应用程序,该文件将包含您所有的Spring Web具体的mvc 组件(bean)。 你可以改变的确切位置 配置文件通过一个Servlet初始化参数(见下文 详情)。

这个 WebApplicationContext 是一个 扩展的平原 ApplicationContext 这有一些额外的功能所需的web应用程序。 它不同从一个正常的 ApplicationContext 在这 有能力解决主题(见吗 SectionA 17.9,一个themesa使用), 和,它知道这Servlet相关(通

过一个链接 这个 ServletContext)。 这个 WebApplicationContext 绑定在 ServletContext ,通过使用静态方法 在 RequestContextUtils 类你可以总是 查找 WebApplicationContext 如果你 需要访问它。

17.2.1A特殊Bean类型 WebApplicationContext

春天 DispatcherServlet 使用特殊的 bean处理请求并呈现适当的视图。 这些bean Spring MVC的一部分。 你可以选择哪些特殊bean来使用 通过简单的配置一个或更多的人在 WebApplicationContext 。 然而,你不需要做,最初因为Spring MVC 维护一个列表的默认bean使用如果你没有配置任何。 将在下一节进行详细介绍。 首先看下面的表格 清单的特殊bean类型 DispatcherServlet 依赖。

17.1为多。 一个特殊的bean类型 WebApplicationContext

Bean类型	解释
HandlerMapping	地图传入请求传递给处理程序和一个列表的 前置和后处理器(处理程序拦截器)基于一些标准的细节有所不同 HandlerMapping 实现。 最受欢迎的实现支持 但其他实现注释的控制器也存在。
HandlerAdapter	帮助 DispatcherServlet 到 调用处理程序映射到一个请求无论处理程序 实际上是调用。 例如,调用一个带注释的控制器 需要解决各种注释。 因此主要目的 的 HandlerAdapter 是保护 DispatcherServlet 从这些细节。
HandlerExceptionResolver	地图视图也允许例外更多 复杂的异常处理代码。
ViewResolver	解决逻辑基于字符串的视图名称到实际 视图 类型。
LocaleResolver	解决地区客户使用, 为了能够提供国际化的观点
ThemeResolver	解决主题您的web应用程序可以使用, 的例子,提供个性化的布局
MultipartResolver	解析多部分请求例如支持处理 从HTML表单文件上传。
FlashMapManager	存储和检索“输入”和“输出” FlashMap 这可以用于传递属性 从一个请求到另一个,通常在一个重定向。

DispatcherServlet 17.2.2A默认配置

正如上一节中提到的每个特殊bean 这个 DispatcherServlet 维护一个列表 使用默认的实现。 这个信息是 保存在文件 DispatcherServlet.properties 在包 org.springframework.web.servlet 。

所有特殊bean有一些合理的默认值 他们自己的。 虽然迟早你会需要定制 一个或更多这些bean的属性提供。 例如它是十分常见的配置 一个 InternalResourceViewResolver 设置其 前缀 财产 父视图文件的位置。

无论细节,这个重要的概念 理解是,一旦 你配置一个特殊bean如一个 InternalResourceViewResolver 在你的 WebApplicationContext ,你 有效地覆盖列表的默认实现 这将一直使用否则这些特别的bean 类型。 例如如果你配置一个 InternalResourceViewResolver , 默认列表的 ViewResolver 实现被忽略。

在 SectionA 17.15,一个配置弹簧MVCa 您将了解 其他选项配置Spring MVC包括 MVC Java配置和MVC XML名称空间两者提供 一个简单的起点和假设小知识 Spring MVC的作品如何。 不管您如何选择 配置您的应用程序中,概念的解释这个 部分是基本应该对你有所帮助。

17.2.3A DispatcherServlet处理顺序

在你设置一个 DispatcherServlet ,和一个 请求到来时,特定的 DispatcherServlet , DispatcherServlet 开始处理请求 如下:

1. 这个 WebApplicationContext 是 寻找和绑定在请求作为一个属性的 控制器和其他元素在这个过程中可以使用。 它 默认绑定在钥匙吗 DispatcherServlet.WEB_APPLICATION_CONTEXT_ATTRIBUTE 。
2. 语言环境解析器绑定到请求使元素 在这个过程中解决地区使用在处理 请求(渲染视图、准备数据,等等)。 如果你不 需要语言环境解决,你不需要它。
3. 主题解析器绑定到请求让元素等 作为观点确定哪些主题使用。 如果你不使用的主题,你 可以忽略它。
4. 如果你指定一个多部分文件解析器,请求 multiparts检查;如果multiparts被发现后,请求 裹在

MultipartHttpServletRequest 对于 进一步处理的其他过程中的元素。 看到 SectionA 17.10,一个弹簧的多部分(文件上传)supporta 为进一步的信息关于多部分 处理。

5. 一个适当的处理程序是寻找。 如果一个处理程序被发现, 执行链相关处理程序(预处理器, postprocessors和控制器)是按顺序执行的准备 模型或渲染。
6. 如果一个模型被返回,视图呈现。 如果没有模型 回来的时候,(可能是由于一个预处理或后处理程序拦截 请求,也许出于安全原因),没有视图呈现, 因为请求可能已经完成。

处理异常的解析器中声明的 WebApplicationContext 接异常 这期间会抛出请求的处理。 使用这些异常 解析器允许您定义自定义行为来解决 例外。

春天 DispatcherServlet 还支持 返回的 最后修改日期 ,如 指定的Servlet API。 确定最后的过程 修改日期为一个特定的请求很简单: DispatcherServlet 查找一个合适的处理程序 映射和测试处理程序是否发现实现了 lastModified 接口。 如果是这样,这个值的 长 getLastModified(请求) 方法 lastModified 接口返回给 客户端。

你可以定制个人 DispatcherServlet 实例通过添加Servlet 初始化参数(初始 元素) Servlet声明在 web . xml 文件。 看到 下表为支持的参数列表。

为多17 2 DispatcherServlet 初始化 参数

参数	解释
contextClass	类,实现了 WebApplicationContext ,这 实例化这个Servlet所使用的上下文。 默认情况下, XmlWebApplicationContext 是使用。
contextConfigLocation	字符串传递到上下文实例(指定的 contextClass),表明上下文(年代) 被发现。 字符串包含潜在的多个字符串 (使用逗号作为分隔符)来支持多个上下文。 在 有多个上下文位置与bean定义 两次,最新的位置优先。
名称空间	名称空间的 WebApplicationContext 。 默认为 [servlet - name]servlet 。

17.3一个实现控制器

控制器提供访问应用程序的行为 通常通过服务接口定义。 控制器 解释用户的输入并将其转换为一个模型,是代表来 用户通过视图。 弹簧实现控制器在一个非常抽象 的方法,它使您能够创建一个广泛的控制器。

Spring 2.5引入了一个基于注解的编程模型为MVC 控制器,使用等注释 @RequestMapping , @RequestParam , @ModelAttribute ,等等。 这个注释 支持可用于Servlet MVC和Portlet MVC。 控制器 实现在这个风格没有扩展特定的基类或 实现特定的接口。 此外,他们通常没有 直接依赖于Servlet或Portlet api,尽管您可以轻松 配置访问Servlet或Portlet设施。

提示

可用的 样品 库 ,许多web应用程序利用注释 这一节中描述的支持包括 MvcShowcase , MvcAjax , MvcBasic , 宠物诊所 , 生产商 ,和其他人。

```
@Controller
public class HelloWorldController {

    @RequestMapping("/helloWorld")
    public String helloWorld(Model model) {
        model.addAttribute("message", "Hello World!");
        return "helloWorld";
    }
}
```

正如您可以看到的, controller 和 @RequestMapping 注释允许灵活 方法名称和签名。 在这个特殊的例子中,方法接受 一个模型 并返回一个视图的名字作为一个 字符串 ,但其他各种方法参数和 返回值可以作为后面解释这部分。 controller 和 @RequestMapping 和许多其他的 注释形式Spring MVC的基础实施。 本节 这些注释文件以及它们如何使用最为普遍的一种Servlet环境。

17.3.1A定义一个控制器 controller

这个 controller 注释 表明一个特定的类服务的角色 控制器 。 春天不需要你延长 任何控制器的基类或参考Servlet API。 然而,你可以 还是参考servlet特定功能如果你需要。

这个 controller 注释作为 一个原型的带注释的类,说明它的作用。 这个 调度程序扫描如此注释的类映射方法和检测 @RequestMapping 注释(见下 部分)。

你可以定义注释控制器豆子明确,使用 标准的Spring bean定义在调度员的上下文。 然而, 这个 controller 原型还允许 为自动,符合春天一般支持检测 组件类在类路径中,自动注册的bean定义 为他们。

启用自动如此注释的控制器,您添加 组件扫描到您的配置。 使用 spring上下文 模式见下面的XML 代码片段:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:p="http://www.springframework.org/schema/p"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd">

   <context:component-scan base-package="org.springframework.samples.petclinic.web"/>

   <!-- ... -->

</beans>
```

17.3.2A映射请求 @RequestMapping

你使用 @RequestMapping 注释url映射如 /预约 到 整个类或一个特定的处理程序方法。 通常 类级别注释映射一个特定请求路径(或路径模式) 在一个窗体控制器,与额外的方法级注释 缩小主要映射为一个特定的HTTP方法请求方法 (“得到” , “后” 等)或一个HTTP请求参数条件。

下面的例子从 生产商 样品 显示了一个控制器在Spring MVC应用程序,它使用这个 注释:

```java
@Controller
@RequestMapping("/appointments")
public class AppointmentsController {

   private final AppointmentBook appointmentBook;

   @Autowired
   public AppointmentsController(AppointmentBook appointmentBook) {
      this.appointmentBook = appointmentBook;
   }

   @RequestMapping(method = RequestMethod.GET)
   public Map<String, Appointment> get() {
      return appointmentBook.getAppointmentsForToday();
   }

   @RequestMapping(value="/{day}", method = RequestMethod.GET)
   public Map<String, Appointment> getForDay(@PathVariable @DateTimeFormat(iso=ISO.DATE) Date day, Model model) {
      return appointmentBook.getAppointmentsForDay(day);
   }

   @RequestMapping(value="/new", method = RequestMethod.GET)
   public AppointmentForm getNewForm() {
      return new AppointmentForm();
   }

   @RequestMapping(method = RequestMethod.POST)
   public String add(@Valid AppointmentForm appointment, BindingResult result) {
      if (result.hasErrors()) {
         return "appointments/new";
      }
      appointmentBook.addAppointment(appointment);
      return "redirect:/appointments";
   }
}
```

在这个示例中, @RequestMapping 是用在许多地方。 第一次使用是在类型(类) 水平,这表明所有的处理方法在这个控制器 相对于 /预约 路径。 这个 get() 方法有进一步 @RequestMapping 细化:它只 接受GET请求,这意味着为一个HTTP GET /预约 调用这个方法。 这个 post() 有一个类似的细化, getNewForm() 结合了HTTP的定义 方法和路径为一个,这样得到的请求 约会/新 由该方法。

这个 getForDay() 方法显示了另一个 使用 @RequestMapping :URI模板。 (见 接下来的 部分)。

一个 @RequestMapping 在类 水平不是必需的。 没有它,所有路径只是绝对的, 不相对。 下面的例子从 宠物诊所 示例应用程序展示了一个治愈率更高 控制器使用 @RequestMapping :

```
@Controller
public class ClinicController {

    private final Clinic clinic;

    @Autowired
    public ClinicController(Clinic clinic) {
        this.clinic = clinic;
    }

    @RequestMapping("/")
    public void welcomeHandler() {
    }

    @RequestMapping("/vets")
    public ModelMap vetsHandler() {
        return new ModelMap(this.clinic.getVets());
    }

}
```

使用 @RequestMapping 在 接口方法

一个常见的陷阱在处理注释的控制器类 发生在应用功能,需要创建一个代理 控制器对象(如。 transactional 方法)。 通常你 将引入的接口控制器为了使用JDK吗 动态代理。 做这项工作你必须移动 @RequestMapping 注释,以及 任何其他类型和方法级注释(如。 @ModelAttribute , @InitBinder) 接口以及映射机制只能“看” 接口公开的代理。 或者,你可以激活 代理目标类= " true " 在配置 功能应用于控制器(在我们的事务场景 在 < tx:注解驱动/ >)。 这样做表明 基于cglib代理,子类应该用来代替 基于接口的JDK代理。 为更多的信息在不同的代理 机制看 SectionA 9.6,一个mechanismsa代理 。

但是要注意,方法参数注释,例如。 @RequestParam ,必须出现在 方法签名的控制器类。

新的支持类 @RequestMapping 方法在Spring MVC 3.1

Spring 3.1引入了一套新的支持类 @RequestMapping 方法称为 RequestMappingHandlerMapping 和 RequestMappingHandlerAdapter 分别。 他们建议使用,甚至需要利用 新功能在Spring MVC 3.1和前进。 新的支持 类是默认启用的MVC名称空间和MVC Java 配置但必须配置明确如果使用既不。 本节描述一些 重要的区别旧的和新的支持类。

Spring 3.1之前,类型和方法级的请求映射 在两个独立的阶段检查——一个控制器被选第一个 的 DefaultAnnotationHandlerMapping 和 实际的方法来调用是缩小了的第二 这个 AnnotationMethodHandlerAdapter 。

新的支持类在Spring 3.1中, RequestMappingHandlerMapping 是唯一的地方 在决定哪些方法应该处理请求。 认为作为一个集合的控制器方法独特的端点 为每个方法和映射来自类型和方法级 @RequestMapping 信息。

这使一些新的可能性。 一次一个 HandlerInterceptor 或 HandlerExceptionResolver 现在可以预计, 基于对象的处理程序是一个 HandlerMethod , 它允许他们检查的具体方法,其参数和 相关的注释。 加工为URL不再需要 被划分到不同的控制器。

还有几件事情不再可能的:

- 选择一个控制器首先用 SimpleUrlHandlerMapping 或 BeanNameUrlHandlerMapping 然后狭窄 该方法基于 @RequestMapping 注释。
- 依靠方法名称作为补救机制 之间消除歧义两 @RequestMapping 方法 这没有一个明确的路径映射URL路径但否则 匹配同样,如通过HTTP方法。 在新的支持类 @RequestMapping 方法必须被映射 独特的。
- 有一个默认的方法(没有一个明确的吗 路径映射),请求处理如果没有其他 控制器方法匹配更具体。 在新的支持 如果一个匹配的方法类没有找到一个404错误 是提高。

上述特征仍支持与现有的支持 类。 然而利用新的Spring MVC 3.1特性 你需要使用新的支持类。

URI模板模式

URI模板 可以用于方便吗 访问URL的选定部分在一个 @RequestMapping 法。

类URI URI模板是一个字符串,其中包含一个或多个 变量名。 当你为这些变量替代值, 模板变成一个URI。 这个 提出 RFC 为 URI模板定义了一个URI是参数化的。 对于 示例中,URI模板 http://www.example.com/users/ { userId } 包含 变量 userId 。 赋值 弗雷德 对变量的收益率 http://www.example.com/users/fred 。

在Spring MVC可以使用 @PathVariable 注释方法 参数的值绑定到一个URI模板变量:

```
@RequestMapping(value="/owners/{ownerId}", method=RequestMethod.GET)
public String findOwner(@PathVariable String ownerId, Model model) {
  Owner owner = ownerService.findOwner(ownerId);
  model.addAttribute("owner", owner);
  return "displayOwner";
}
```

URI模板" /业主/ { ownerId } " 指定变量名 ownerId 。 当 控制器处理这个请求,值 ownerId 将发现的价值 适当的部分的URI。 例如,当一个请求到来时 /业主/弗雷德 的价值, ownerId 是 弗雷德 。

提示

处理@PathVariable注释,Spring MVC需要 找到匹配的URI模板变量的名字。 你可以指定它 在注释:

```
@RequestMapping(value="/owners/{ownerId}", method=RequestMethod.GET)
public String findOwner(@PathVariable("ownerId") String theOwner, Model model) {
  // implementation omitted
}
```

或者如果URI模板变量名称匹配方法 参数名称可以省略这些细节。 只要你的代码是不 编译没有调试信息,Spring MVC将匹配 方法参数名称到URI模板变量名称:

```
@RequestMapping(value="/owners/{ownerId}", method=RequestMethod.GET)
public String findOwner(@PathVariable String ownerId, Model model) {
  // implementation omitted
}
```

一个方法可以有任意数量的 @PathVariable 注释:

```
@RequestMapping(value="/owners/{ownerId}/pets/{petId}", method=RequestMethod.GET)
public String findPet(@PathVariable String ownerId, @PathVariable String petId, Model model) {
  Owner owner = ownerService.findOwner(ownerId);
  Pet pet = owner.getPet(petId);
  model.addAttribute("pet", pet);
  return "displayPet";
}
```

当一个 @PathVariable 注释是 用在 Map < String,String > 参数, 地图是填充所有URI模板变量。

URI模板可以组装的类型和路径水平 @RequestMapping 注释。 作为一个结果, findPet() 方法可以调用一个URL 如 /业主/42 /宠物/ 21 。

```
@Controller
@RequestMapping("/owners/{ownerId}")
public class RelativePathUriTemplateController {

  @RequestMapping("/pets/{petId}")
  public void findPet(@PathVariable String ownerId, @PathVariable String petId, Model model) {
    // implementation omitted
  }
}
```

一个 @PathVariable 参数可以 的 **任何简单的类型** 比如int、long 日期等。弹簧自动转换为适当的类型或 抛出一个TypeMismatchException 如果不能 这样做。 你也可以注册支持解析额外的数据 类型。 看到 一个章节方法参数和类型Conversiona 和 一个章节定制 WebDataBinder initializationa 。

URI模板用正则表达式模式

有时你需要更精确定义URI模板 变量。 考虑到URL "/ spring web / spring web 3 0 5瓶" 。 你怎么把它吗 分成多个部分?

这个 @RequestMapping 注释 支持使用正则表达式在URI模板变量。 这个 语法是 { varName:regex } 在第一部分定义了吗变量名和第二——表达式定期 示例:

```
@RequestMapping("/spring-web/{symbolicName:[a-z-]+}-{version:\\d\\.\\d\\.\\d}{extension:\\.[a-z]+}")
  public void handle(@PathVariable String version, @PathVariable String extension) {
    // ...
  }
}
```

路径模式

除了URI模板, @RequestMapping 注释也 支持ant是基于路径的模式(例如, / myPath / *而已)。 结合URI模板和 ant是基于团还支持(例如, /业主/ * /宠物/ { petId })。

模式包含占位符

模式 @RequestMapping 注释 支持$ {... }占位符与当地的属性和/或系统属性 和环境变量。 这可能是有用的情况下路径一个控制器是映射到可能需要通过配置定制。 更多信息见Javadoc的占位符 来完成 。

矩阵变量

URI规范 RFC 3986 定义包括名称-值对的可能性路径段内。 没有特定的术语的规范。 一般的“URI路径参数”可以应用虽然更加独特 “矩阵uri” ，源自一个古老的职位由Tim berners - lee,也经常使用 和广为人知。 在这些被称为Spring MVC 作为矩阵变量。

矩阵变量可以出现在任何路径段,每个矩阵变量 用“;”分隔(分号)。 例如: “/汽车;颜色=红;年= 2012” 。 多个值可以是“,”(逗号)分隔 “颜色=红、绿、蓝” 或变量名称可以重复 “颜色=红;颜色=绿色;颜色=蓝” 。

如果一个URL将包含矩阵变量,请求映射 模式必须代表他们与URI模板。 这样可以确保请求可以匹配正确不管 矩阵变量存在与否,他们是按照什么顺序 提供了。

下面是一个例子,提取矩阵变量“问”：

```
// GET /pets/42;q=11;r=22

@RequestMapping(value = "/pets/{petId}", method = RequestMethod.GET)
public void findPet(@PathVariable String petId, @MatrixVariable int q) {

  // petId == 42
  // q == 11

}
```

因为所有的路径段可能包含矩阵变量,在某些情况下 你需要更具体的识别在变量预计将:

```
// GET /owners/42;q=11/pets/21;q=22

@RequestMapping(value = "/owners/{ownerId}/pets/{petId}", method = RequestMethod.GET)
public void findPet(
    @MatrixVariable(value="q", pathVar="ownerId") int q1,
    @MatrixVariable(value="q", pathVar="petId") int q2) {

  // q1 == 11
  // q2 == 22

}
```

一个矩阵变量可以定义为可选的,默认值指定:

```
// GET /pets/42

@RequestMapping(value = "/pets/{petId}", method = RequestMethod.GET)
  public void findPet(@MatrixVariable(required=true, defaultValue="1") int q) {

    // q == 1

  }
```

所有的矩阵变量可能得到一个地图:

```
// GET /owners/42;q=11;r=12/pets/21;q=22;s=23

@RequestMapping(value = "/owners/{ownerId}/pets/{petId}", method = RequestMethod.GET)
  public void findPet(
        @MatrixVariable Map<String, String> matrixVars,
        @MatrixVariable(pathVar="petId"") Map<String, String> petMatrixVars) {

    // matrixVars: ["q" : [11,22], "r" : 12, "s" : 23]
    // petMatrixVars: ["q" : 11, "s" : 23]

  }
```

注意,使使用矩阵变量,你必须设置 removeSemicolonContent 财产的 RequestMappingHandlerMapping 到 假 。 默认情况下,它被设置为 真正的 除了 MVC名称空间和MVC的Java配置这两种自动启用 矩阵变量的使用。

消耗品媒体类型

你可以缩小主要映射通过指定的列表 消耗品媒体类型。 请求将被匹配只有在 内容类型 请求头匹配指定的 媒体类型。 例如:

```
@Controller
@RequestMapping(value = "/pets", method = RequestMethod.POST, consumes="application/json")
public void addPet(@RequestBody Pet pet, Model model) {
    // implementation omitted
}
```

消耗品媒体类型表达式也可以否定在 !文本/平原 匹配所有请求其他比 那些 内容类型 的 文本/平原 。

提示

这个 消耗 条件支持 的类型和方法级别上。 不像大多数其他的条件,当 使用类型水平,方法级消费品类型覆盖 而不是扩展类型级别消耗品类型。

可生产的媒体类型

你可以缩小主要映射通过指定的列表 可延长的媒体类型。 请求将被匹配只有在 接受 请求头匹配其中的一个 值。 此外,使用 产生 条件保证了实际内容类型用于生成 响应尊重媒体的类型中 产生 条件。 例如:

```
@Controller
@RequestMapping(value = "/pets/{petId}", method = RequestMethod.GET, produces="application/json")
@ResponseBody
public Pet getPet(@PathVariable String petId, Model model) {
    // implementation omitted
}
```

就像 消耗 ,可生产的媒体 类型的表达式可以否定在 !文本/平原 匹配所有请求其他比有一个 接受 头值 文本/平原 。

提示

这个 产生 条件支持 的类型和方法级别上。 不像大多数其他的条件,当 使用类型水平,方法级可生产的类型覆盖 而不是扩展类型级别可生产的类型。

请求参数和头的值

你可以缩小请求匹配通过请求参数 条件如 “myParam” ,” ! myParam” ,或 “myParam = myValue” 。 前两个测试的要求 参数存在缺失和第三/一个特定的参数 值。 下面是一个示例请求参数值 条件:

```
@Controller
@RequestMapping("/owners/{ownerId}")
public class RelativePathUriTemplateController {

  @RequestMapping(value = "/pets/{petId}", method = RequestMethod.GET, params="myParam=myValue")
  public void findPet(@PathVariable String ownerId, @PathVariable String petId, Model model) {
    // implementation omitted
  }
}
```

同样可以做测试请求头的存在/缺席 或以匹配根据特定请求报头值:

```
@Controller
@RequestMapping("/owners/{ownerId}")
public class RelativePathUriTemplateController {

@RequestMapping(value = "/pets", method = RequestMethod.GET, headers="myHeader=myValue")
  public void findPet(@PathVariable String ownerId, @PathVariable String petId, Model model) {
    // implementation omitted
  }
}
```

提示

虽然您可以匹配 内容类型 和 接受 头值使用媒体类型野生 卡(例如 “content - type = text / *” 将 匹配 “text / plain” 和 “text / html”),建议使用 消耗 和 产生 条件下分别相反。 他们的目的是专门为 这一目的。

17.3.3A定义 @RequestMapping 处理程序 方法

一个 @RequestMapping 处理器方法可以有 一个非常灵活的签名。 支持的方法参数和返回 值将在下面部分。 大多数参数可以 用在任意订单的唯一例外 BindingResult 参数。 这是描述的 下一节。

> **注意**
>
> Spring 3.1引入了一套新的支持类 @RequestMapping 方法称为 RequestMappingHandlerMapping 和 RequestMappingHandlerAdapter 分别。 他们建议使用,甚至需要利用 新功能在Spring MVC 3.1和前进。 新的支持类是默认启用从MVC名称空间和 与使用MVC Java配置但必须 明确如果使用既不配置。

支持方法参数类型

以下是支持的方法参数:

- 请求或响应对象(Servlet API)。 选择任何 特定的请求或响应类型,例如 ServletRequest 或 HttpServletRequest 。
- 会话对象(Servlet API):类型 HttpSession 。 一个论点的 类型强制存在相应的会话。 作为一个 因此,这样的一个论点是永远 空 。

  > **注意**
  >
  > 会话的访问可能不是线程安全的,特别是在 一个Servlet环境。 考虑设置 RequestMappingHandlerAdapter ' s “synchronizeOnSession”标记为“true”如果多个请求 允许并发访问会话。

- org.springframework.web.context.request.WebRequest 或 org.springframework.web.context.request.NativeWebRequest 。 允许通用请求参数访问以及 请求/会话属性访问,没有联系到本地 Servlet / Portlet API。
- java util地区 对于当前 请求场所,由最具体的语言环境解析器 可用的,实际上,配置的 LocaleResolver 在一个Servlet 环境。
- java io inputstream / java io读者 通往 请求的内容。 这个值是原始InputStream /读者 Servlet API公开的。
- java io outputstream / java io作家 对于产生 响应的内容。 这个值是原始的OutputStream /作家 Servlet API公开的。
- java安全主要 包含当前认证用户。
- @PathVariable 注释参数 获取URI模板变量。 看到 一个章节URI模板Patternsa 。
- @MatrixVariable 注释参数 获取名称-值对位于URI路径片断。 看到 一个章节Variablesa矩阵 。
- @RequestParam 注释参数 获取特定的Servlet请求参数。 参数 值转换为声明的方法参数类型。 看到 一个章节绑定请求参数方法参数 @RequestParam 一个 。
- @RequestHeader 注释 参数获取特定的Servlet请求的HTTP标头。 参数值转换为声明的方法参数 类型。
- @RequestBody 注释 参数获取HTTP请求体。 参数值 转换为声明的方法参数类型使用吗 HttpMessageConverter 年代。看到 一个章节映射与@RequestBody请求主体 annotationa 。
- @RequestPart 注释 参数获取内容的一个“多部分/格式数据” 请求部分。 看到 SectionA 17 10 5,一个处理文件上传请求从编程clientsa 和 SectionA 17.10,一个弹簧的多部分(文件上传)supporta 。
- HttpEntity < ? > 参数 访问Servlet请求的HTTP标题和内容。 这个 请求流将被转换成实体使用 HttpMessageConverter 年代。看到 一个章节使用 HttpEntity < ? > 一个 。
- java util地图 / org.springframework.ui.Model / org.springframework.ui.ModelMap 对于 丰富的隐式模型,接触到网络 视图。
- org.springframework.web.servlet.mvc.support.RedirectAttributes 指定确切的属性集使用以防 重定向和也添加闪光属性(属性存储 暂时在服务器端,使它们可用 请求重定向后)。 RedirectAttributes 被用于代替吗 隐式模型如果方法返回一个“重定向:”前缀的视图 名称或 RedirectView 。
- 命令或表单对象绑定请求参数bean 属性(通过setter)或直接向田地,同 可定制的类型转换,取决于 @InitBinder 方法和/或 HandlerAdapter配置。 看到 WebBindingInitializer 属性 RequestMappingHandlerAdapter 。 这样 命令对象以及它们的验证结果将 公开为模型属性默认情况下,使用命令类 类的名字——例如模型属性”orderAddress”命令 “some.package.OrderAddress”类型的对象。 这个 ModelAttribute 注释可用于 一个方法参数来定制模型属性名称 使用。
- org.springframework.validation.Errors / org.springframework.validation.BindingResult 验证结果为前面的命令对象(或形式 紧接方法参数)。
- org.springframework.web.bind.support.SessionStatus 状态处理标记形式处理完成, 触发器的清理会话属性 指定的 @SessionAttributes 注释处理程序的类型级别。
- org.springframework.web.util.UriComponentsBuilder 一个建筑工人准备一个URL相对于当前请求的 主机、端口、方案、上下文路径和文字部分 servlet映射。

这个 错误 或 BindingResult 参数必须遵循 模型对象被绑定立即方法 签名可能更多,一个模型对象和弹簧将创建 一个单独的 BindingResult 实例 他们中的每一个所以以下样品不工作:

ExampleA 17.1。 一个无效的BindingResult和@ModelAttribute订购

```
@RequestMapping(method = RequestMethod.POST)
public String processSubmit(@ModelAttribute("pet") Pet pet,
    Model model, BindingResult result) { ... }
```

注意,有一个 模型 参数之间的 宠物 和 BindingResult 。 得到这个工作 你必须重新排序参数如下:

```
@RequestMapping(method = RequestMethod.POST)
public String processSubmit(@ModelAttribute("pet") Pet pet,
    BindingResult result, Model model) { ... }
```

支持方法返回类型

以下是支持的返回类型:

- 一个 ModelAndView 对象, 模型隐式富含命令对象和结果 的 @ModelAttribute 注释的参考数据 访问器方法。
- 一个 模型 对象, 视图名称隐式确定通过 RequestToViewNameTranslator 和 该模型隐式富含命令对象和 结果 @ModelAttribute 注释 参考数据访问方法。
- 一个 地图 对象揭露 模型、视图与名字隐式确定通过 RequestToViewNameTranslator 和 该模型隐式富含命令对象和 结果 @ModelAttribute 注释 参考数据访问方法。
- 一个 视图 对象, 模型隐式确定通过命令对象和 @ModelAttribute 注释的参考数据 访问器方法。 处理程序方法也可以通过编程的方式 丰富的模型通过宣布 模型 参数(见上图)。
- 一个 字符串 值解释 作为逻辑视图名称,与模型隐式确定 通过命令对象和 @ModelAttribute 带注释的参考数据访问方法。 处理程序方法 也可以通过编程的方式丰富模型通过声明一个吗 模型 参数(见 以上)。
- 无效 如果该方法处理响应 本身(通过编写响应内容直接,声明一个 类型的参数, ServletResponse / HttpServletResponse 这 目的),或者如果该视图名称应该是含蓄的 决定通过一个 RequestToViewNameTranslator (不 声明一个响应参数的处理程序方法 签名)。
- 如果该方法标注 @ResponseBody ,返回类型是 写入响应HTTP的身体。 返回值将 转换为声明的方法参数类型使用 HttpMessageConverter 年代。看到 一个章节映射响应的身体 @ResponseBody annotationa 。
- 一个 HttpEntity < ? > 或 ResponseEntity < ? > 对象提供 访问Servlet响应HTTP标题和内容。 这个 实体的身体可以转化为响应流使用 HttpMessageConverter 年代。看到 一个章节使用 HttpEntity < ? > 一个 。
- 一个 可调用< ? > 可以 时返回应用程序想要产生回报 价值管理的异步线程中的Spring MVC。
- 一个 DeferredResult < ? > 可以 时返回应用程序想要产生回报 值从一个线程自己的选择。
- 其他任何返回类型被认为是一个单一的模型 属性会接触到视图,使用属性名称 指定通过 @ModelAttribute 在 方法级(或默认的属性名称基于返回 类型的类名)。 该模型是含蓄地富含命令 对象和结果的 @ModelAttribute 带注释的参考数据访问方法。

绑定请求参数方法参数 @RequestParam

使用 @RequestParam 注释来绑定 请求参数方法参数在你的控制器。

下面的代码片段显示了使用方法:

```
@Controller
@RequestMapping("/pets")
@SessionAttributes("pet")
public class EditPetForm {

    // ...

    @RequestMapping(method = RequestMethod.GET)
    public String setupForm(@RequestParam("petId") int petId, ModelMap model) {
        Pet pet = this.clinic.loadPet(petId);
        model.addAttribute("pet", pet);
        return "petForm";
    }

    // ...
```

参数使用这个注释默认是必需的,但 你可以指定一个参数是可选的设置 @RequestParam ' s 需要 属性 假 (例如,

@RequestParam(value = " id ", 要求= false))。

类型转换是自动应用如果目标方法 参数类型不 字符串 。 看到 一个章节方法参数和类型Conversiona 。

映射与@RequestBody请求主体 注释

这个 @RequestBody 方法参数 注释表明一个方法参数应该绑定到 价值的HTTP请求体。 例如:

```
@RequestMapping(value = "/something", method = RequestMethod.PUT)
public void handle(@RequestBody String body, Writer writer) throws IOException {
  writer.write(body);
}
```

你把请求体方法参数使用 HttpMessageConverter 。 HttpMessageConverter 负责 从HTTP请求消息转换到一个对象和转换从一个对象到HTTP响应体。 这个 RequestMappingHandlerAdapter 支持 @RequestBody 注释与以下 默认 HttpMessageConverters :

- ByteArrayHttpMessageConverter 将字节数组。
- StringHttpMessageConverter 转换 字符串。
- FormHttpMessageConverter 转换 表单数据到/从MultiValueMap <字符串,字符串>。
- SourceHttpMessageConverter 转换 /从一个javax.xml.transform.Source。

在这些转换器的更多信息,请参阅 消息转换器 。 还要注意 ,如果使用MVC名称空间或MVC Java配置,更广泛 范围的消息转换器默认注册。 看到 启用MVC Java配置或 MVC XML名称空间 为更多的信息。

如果你打算读和写XML,您将需要配置 这个 MarshallingHttpMessageConverter 与 特定 Marshaller 和一个 解组程序 实现从 org.springframework.oxm 包。 这个例子 下面显示了如何做,直接在你的配置但如果 配置您的应用程序是通过MVC名称空间或 MVC Java配置见 启用 MVC Java配置或MVC XML名称空间 相反。

```
<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">
   <property name="messageConverters">
     <util:list id="beanList">
       <ref bean="stringHttpMessageConverter"/>
       <ref bean="marshallingHttpMessageConverter"/>
     </util:list>
   </property
</bean>

<bean id="stringHttpMessageConverter"
      class="org.springframework.http.converter.StringHttpMessageConverter"/>

<bean id="marshallingHttpMessageConverter"
     class="org.springframework.http.converter.xml.MarshallingHttpMessageConverter">
  <property name="marshaller" ref="castorMarshaller" />
  <property name="unmarshaller" ref="castorMarshaller" />
</bean>

<bean id="castorMarshaller" class="org.springframework.oxm.castor.CastorMarshaller"/>
```

一个 @RequestBody 方法参数可以 标注 @Valid ,在这种情况下,这将是 验证使用配置 验证器 实例。 当使用MVC名称空间或MVC Java配置,一个jsr - 303验证器 配置自动假设一个jsr - 303实现吗 可以在类路径中。

就像 @ModelAttribute 参数, 一个 错误 参数可以用来检查错误。 如果这种观点并不宣称,一个 MethodArgumentNotValidException 将提高。 异常处理 DefaultHandlerExceptionResolver ,发送 一个 400年 错误返回到客户机。

注意

也看到 启用MVC Java配置或MVC XML名称空间 对于 配置消息信息转换器和一个验证器 通过MVC名称空间或MVC Java配置。

映射响应的身体 @ResponseBody 注释

这个 @ResponseBody 注释是 类似 @RequestBody 。 这 注释可以戴上一个方法和表明,返回类型 应该直接写到HTTP响应正文(而不是放置吗 在一个模型,或解释为一个视图名称)。 例如:

```
@RequestMapping(value = "/something", method = RequestMethod.PUT)
@ResponseBody
public String helloWorld() {
```

```
  return "Hello World";
}
```

上面的例子将导致文本 你好 世界 被写入HTTP响应流。

与 @RequestBody ,春天 将返回的对象转换成一个响应体通过使用一个 HttpMessageConverter 。 更多 这些转换器的信息,请参见前一节和 消息转换器 。

使用 HttpEntity < ? >

这个 HttpEntity 类似于 @RequestBody 和 @ResponseBody 。 除了获得 请求和响应的身体, HttpEntity (和响应具体的子类 ResponseEntity)还允许访问 请求和响应头,就像这样:

```
@RequestMapping("/something")
public ResponseEntity<String> handle(HttpEntity<byte[]> requestEntity) throws UnsupportedEncodingException {
  String requestHeader = requestEntity.getHeaders().getFirst("MyRequestHeader"));
  byte[] requestBody = requestEntity.getBody();
  // do something with request header and body

  HttpHeaders responseHeaders = new HttpHeaders();
  responseHeaders.set("MyResponseHeader", "MyValue");
  return new ResponseEntity<String>("Hello World", responseHeaders, HttpStatus.CREATED);
}
```

上面的例子中得到的价值 MyRequestHeader 请求头,并读取的身体 作为一个字节数组。 它添加了 MyResponseHeader 到响应,写 你好世界 响应 流,并设置响应状态代码201(创建)。

与 @RequestBody 和 @ResponseBody ,弹簧使用 HttpMessageConverter 将从 并请求和响应流。 为更多的信息关于这些转换器,请参见前一节和 消息转换器 。

使用 @ModelAttribute 在一个 方法

这个 @ModelAttribute 注释 可以用在方法或方法参数。 本节解释 它的使用方法而下一节解释了它的使用 方法参数。

一个 @ModelAttribute 在一个方法 表明该方法的目的是将一个或多个模型 属性。 这种方法支持相同的参数类型 @RequestMapping 方法但是不能 直接映射到请求。 相反 @ModelAttribute 方法在一个控制器 之前被调用 @RequestMapping 方法,在相同的控制器。 一对夫妇的例子:

```
// Add one attribute
// The return value of the method is added to the model under the name "account"
// You can customize the name via @ModelAttribute("myAccount")

@ModelAttribute
public Account addAccount(@RequestParam String number) {
   return accountManager.findAccount(number);
}

// Add multiple attributes

@ModelAttribute
public void populateModel(@RequestParam String number, Model model) {
   model.addAttribute(accountManager.findAccount(number));
   // add more ...
}
```

@ModelAttribute 使用方法 填充模型与常用属性例如 填充下拉与国家或与宠物类型,或检索 命令对象像帐户为了使用它来表示数据 在一个HTML表单。 后者是进一步讨论在未来 部分。

请注意这两个风格的 @ModelAttribute 方法。 在第一个, 隐式方法添加一个属性通过返回它。 在 其次,该方法接受一个 模型并添加任何 数量的模型属性来它。 你可以选择两个 风格。根据您的需要

一个控制器可以有任意数量的 @ModelAttribute 方法。 所有这些 方法之前被调用 @RequestMapping 方法相同的 控制器。

@ModelAttribute 方法也可以 定义在一个 @ControllerAdvice 注释 类和这样的方法适用于所有的控制器。 这个 @ControllerAdvice 注释是 一个组件注释允许实现类时自动被检测 通过类路径扫描。

提示

会发生什么,如果一个模型属性名称不明确 指定的吗? 在这种情况下一个默认的名字是分配给模型 属性基于其类型。 例如,如果这个方法返回一个 类型的对象 帐户 ,使用默认的名称 “账户” 。 你可以改变,通过价值

的 @ModelAttribute 注释。 如果添加 属性直接 模型 ,使用 适当的过载 addAttribute(. .) 方法—— 即。 ,有或没有一个属性的名字。

这个 @ModelAttribute 注释 可以用在 @RequestMapping 方法 为好。 在这种情况下的返回值 @RequestMapping 方法 解释 作为一个模型属性而不是视图名称。 视图的名称是 来自视图名称惯例相反很像的方法 返回一个无效看到 SectionA 17 12 3,一个视图- RequestToViewNameTranslator 一个 。

使用 @ModelAttribute 在一个 方法参数

如上一节所介绍的 @ModelAttribute 可以用在方法 或方法参数。 本节解释了其使用方法 参数。

一个 @ModelAttribute 在一个方法 参数表明参数应该被检索模型。 如果 没有出现在模型中,参数应该被实例化第一 然后添加到模型中。 一旦出现在模型中,参数的 字段应该填充所有请求参数, 匹配的名称。 这称为数据绑定在Spring MVC,非常 有用的机制,使您不必解析每个表单字段 单独。

```
@RequestMapping(value="/owners/{ownerId}/pets/{petId}/edit", method = RequestMethod.POST)
public String processSubmit(@ModelAttribute Pet pet) {

}
```

鉴于上面的例子,可以宠物实例从何而来? 有几种选择:

- 它也许已经在模型由于使用 @SessionAttributes 一个看到 一个章节使用 @SessionAttributes 存储模型 属性在HTTP会话requestsa之间 。
- 它也许已经在模型由于一个 @ModelAttribute 方法在相同的 控制器一个如上一节所介绍的。
- 它可能被检索基于URI模板变量和 类型转换器(更详细地解释下)。
- 它可能被实例化使用其默认构造函数。

一个 @ModelAttribute 方法是一个 常见的方法来获取一个属性的数据库,这可能 有选择地存储在请求之间通过使用 @SessionAttributes 。 在某些情况下, 可以方便的检索属性通过使用一个URI模板吗 变量和类型转换器。 这里是一个例子:

```
@RequestMapping(value="/accounts/{account}", method = RequestMethod.PUT)
public String save(@ModelAttribute("account") Account account) {

}
```

在这个示例模型属性的名称(即。 “账户”) 的名字相匹配的一个URI模板变量。 如果你注册 转换器<字符串,帐户> 可以把 这个 字符串 帐户值转换为 帐户 实例,然后上面的示例 工作不需要一个 @ModelAttribute 法。

下一步是数据绑定。 这个 WebDataBinder 类匹配请求参数 名字一个包括查询字符串参数和表单字段一个模型 属性字段的名字。 匹配的字段类型后填充 转换(从字符串到目标字段类型)已经应用 在必要时。 数据绑定和验证都包含在 ChapterA 7, 验证、数据绑定、类型转换 。 定制数据绑定过程为一个 控制器级别覆盖着 一个章节定制 WebDataBinder initializationa 。

由于数据绑定可能有错误如失踪 必需的字段或类型转换错误。 检查这些错误 添加一个 BindingResult 论点立即 后 @ModelAttribute 论点:

```
@RequestMapping(value="/owners/{ownerId}/pets/{petId}/edit", method = RequestMethod.POST)
public String processSubmit(@ModelAttribute("pet") Pet pet, BindingResult result) {

    if (result.hasErrors()) {
        return "petForm";
    }

    // ...

}
```

与 BindingResult 你可以检查是否 错误被发现在这种情况下,它的普遍呈现相同的形式 错误的地方可以证明借助弹簧的吗 <错误> 表单标记。

除了数据绑定也可以调用验证使用 您自己的自定义验证器传递相同的 BindingResult 这是用来记录数据 绑定错误。 允许数据绑定和验证错误 是在一个地方积累和随后的反馈 用户:

```
@RequestMapping(value="/owners/{ownerId}/pets/{petId}/edit", method = RequestMethod.POST)
public String processSubmit(@ModelAttribute("pet") Pet pet, BindingResult result) {

    new PetValidator().validate(pet, result);
```

```
    if (result.hasErrors()) {
        return "petForm";
    }

    // ...
}
```

或者你可以通过添加验证时自动调用 jsr - 303 @Valid 注释:

```
@RequestMapping(value="/owners/{ownerId}/pets/{petId}/edit", method = RequestMethod.POST)
public String processSubmit(@Valid @ModelAttribute("pet") Pet pet, BindingResult result) {

    if (result.hasErrors()) {
        return "petForm";
    }

    // ...
}
```

看到 SectionA 7.8,一个弹簧3 Validationa 和 ChapterA 7, 验证、数据绑定、类型转换 有关如何配置和使用 验证。

使用 @SessionAttributes 存储模型 属性在HTTP会话请求之间

这个类型水平 @SessionAttributes 注释声明使用会话属性通过一个特定处理器。 这通常会列出名字的模型属性或类型的 模型属性应该透明地存储在会话中 或一些会话存储,作为形式支持bean之间 后续的请求。

下面的代码片段显示了使用该注释, 指定模型属性名称:

```
@Controller
@RequestMapping("/editPet.do")
@SessionAttributes("pet")
public class EditPetForm {
    // ...
}
```

注意

当使用控制器接口(如。 ,对AOP代理), 确保始终把 所有 你的映射 注释,如 @RequestMapping 和 @SessionAttributes —— 控制器 接口 而不是 实现类。

重定向和flash属性指定

默认情况下所有模型属性被认为是公开为 URI模板变量在重定向URL。 剩下的 属性那些原始类型或集合/阵列 基本类型是自动附加作为查询参数。

在注释的控制器模型可能包含但 添加额外的属性最初呈现的目的(如。 下拉字段值)。 获得精确的控制属性 用于重定向的场景中,一个 @RequestMapping 方法可以声明一个 类型的参数, RedirectAttributes 和 用它来添加属性用于 RedirectView 。 如果控制器方法执行 重定向的内容 RedirectAttributes 是使用。 否则 默认的内容 模型 是 使用。

这个 RequestMappingHandlerAdapter 提供 一个国旗称为 “ignoreDefaultModelOnRedirect” 这 可以用来显示内容的默认吗 模型 永远不要使用如果 控制器方法重定向。 相反,控制器方法应该 声明一个属性的类型 RedirectAttributes 或者如果它不做 所以没有属性应该传给 RedirectView 。 两个MVC名称空间和 MVC Java配置保持这个标志设置为 假 为了保持 向后兼容性。 然而,对于新的应用程序我们推荐 设置它 真正的

这个 RedirectAttributes 接口 也可以用来添加闪光属性。 不像其他的重定向 属性,最终在目标重定向URL,flash属性 保存在HTTP会话(因此不出现在URL)。 该模型的控制器提供目标重定向URL 自动接收这些闪光的属性之后,他们是 将从该会话。 看到 SectionA 17.6,一个attributesa使用闪光灯 概述的一般支持flash属性在春天 MVC。

处理 “应用程序/ x-www-form-urlencoded” 数据

前一节介绍了使用 @ModelAttribute 支持形式 提交请求从浏览器客户端。 相同的注释是 推荐使用非浏览器请求从客户。 然而有一个明显的差异与工作的时候 HTTP PUT请求。 浏览器可以提交表单数据通过HTTP GET或HTTP 邮报。 非浏览器客户端也可以通过HTTP提交表单把。 这 提出了一个挑战,因为Servlet规范要求 ServletRequest.getParameter *() 家族的方法 支持表单字段访问只有HTTP POST,而不是HTTP把。

支持HTTP PUT和补丁请求, spring web 模块提供过滤器 HttpPutFormContentFilter ,可以 配置在 web . xml :

```
<filter>
```

```
  <filter-name>httpPutFormFilter</filter-name>
  <filter-class>org.springframework.web.filter.HttpPutFormContentFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>httpPutFormFilter</filter-name>
  <servlet-name>dispatcherServlet</servlet-name>
</filter-mapping>

<servlet>
  <servlet-name>dispatcherServlet</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>
```

上述过滤器拦截HTTP PUT和补丁与内容类型的请求 应用程序/ x-www-form-urlencoded ,读取表单 数据从请求的主体,和包装了 ServletRequest 为了使表单数据 可以通过 ServletRequest.getParameter *() 家族的 方法。

注意

作为 HttpPutFormContentFilter 消费的主体 请求,它不应该被配置为将补丁或url,依靠其他 转换器为 应用程序/ x-www-form-urlencoded 。 这包括 @RequestBody MultiValueMap <字符串,字符串> 和 HttpEntity < MultiValueMap <字符串,字符串> > 。

映射cookie的值与@CookieValue注释

这个 @CookieValue 注释 允许一个方法参数赋值为一个HTTP 饼干。

让我们考虑下面的饼干已经收到 一个http请求:

```
JSESSIONID=415A4AC178C59DACE0B2C9CA727CDD84
```

下面的代码示例演示了如何获取值的 这个 jsessionId 饼干:

```
@RequestMapping("/displayHeaderInfo.do")
public void displayHeaderInfo(@CookieValue("JSESSIONID") String cookie) {

  //...

}
```

类型转换是自动应用如果目标方法 参数类型不 字符串 。 看到 一个章节方法参数和类型Conversiona 。

这个注释支持注释处理程序方法 Servlet和Portlet的环境。

映射属性与@RequestHeader请求头 注释

这个 @RequestHeader 注释 允许一个方法参数绑定到一个请求头。

下面是一个示例请求头:

```
Host                    localhost:8080
Accept                  text/html,application/xhtml+xml,application/xml;q=0.9
Accept-Language         fr,en-gb;q=0.7,en;q=0.3
Accept-Encoding         gzip,deflate
Accept-Charset          ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive              300
```

下面的代码示例演示了如何获取值的 这个 接受编码 和 维生 标题:

```
@RequestMapping("/displayHeaderInfo.do")
public void displayHeaderInfo(@RequestHeader("Accept-Encoding") String encoding,
                              @RequestHeader("Keep-Alive") long keepAlive) {

  //...

}
```

类型转换是自动应用如果方法参数 不是 字符串 。 看到 一个章节方法参数和类型Conversiona 。

提示

可用的内置支持转换为一个以逗号分隔的 字符串数组/收集的字符串或其他类型已知的 类型转换系统。 例如一个方法参数注释 与 @RequestHeader(“接受”) 可能的类型 字符串 但也 String[] 或 列表<字符串>

。

这个注释支持注释处理程序方法 Servlet和Portlet的环境。

方法参数和类型转换

基于字符串的值从请求中提取包括请求 参数、路径变量,请求头和cookie值可能 需要被转换成目标类型的方法参数或 字段(如。 ,绑定一个请求参数中一个字段 @ModelAttribute 参数)他们 一定会。 如果目标类型是不 字符串 , 弹簧自动转换为适当的类型。 所有简单 类型如int、long、日期等,都受支持。 你可以进一步 自定义转换过程中 WebDataBinder (见 一个章节定制 WebDataBinder initializationa)或通过注册 格式化器 与 FormattingConversionService (见 SectionA 7.6,一个弹簧3场 Formattinga)。

定制 WebDataBinder 初始化

定制绑定与PropertyEditors请求参数 通过弹簧的 WebDataBinder ,你可以使用 @InitBinder 带注释的方法在 你的控制器, @InitBinder 方法 在一个 @ControllerAdvice 类, 或提供一个自定义 WebBindingInitializer 。

定制数据绑定和 @InitBinder

注释控制器方法 @InitBinder 允许您配置 web数据绑定直接在你的控制器类。 @InitBinder 识别方法, 初始化 WebDataBinder 那将是 用于填充命令和表单对象参数的注释 处理程序方法。

这样的init粘合剂的方法支持所有参数 @RequestMapping 支持,除了 命令/表单对象和相应的验证结果对象。 init粘合剂的方法必须没有返回值。 因此,他们是 通常声明为 无效 。 典型参数 包括 WebDataBinder 结合 WebRequest 或 java util地区 , 允许代码登记 上下文相关的编辑。

下面的例子演示了如何使用 @InitBinder 配置一个 CustomDateEditor 对于所有 java util日期 表单属性。

```
@Controller
public class MyFormController {

    @InitBinder
    public void initBinder(WebDataBinder binder) {
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        dateFormat.setLenient(false);
        binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, false));
    }

    // ...
}
```

配置一个自定义 WebBindingInitializer

客观数据绑定的初始化,你可以提供一个 定制实现的 WebBindingInitializer 界面, 然后通过提供一个自定义的bean启用配置一个 AnnotationMethodHandlerAdapter ,因此 覆盖默认配置。

下面的例子从PetClinic应用程序很好地展示了一个 配置使用一个自定义实现的 WebBindingInitializer 界面, org.springframework.samples.petclinic.web.ClinicBindingInitializer , 这配置所需的PropertyEditors几个 宠物诊所控制器。

```
<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">
    <property name="cacheSeconds" value="0" />
    <property name="webBindingInitializer">
        <bean class="org.springframework.samples.petclinic.web.ClinicBindingInitializer" />
    </property>
</bean>
```

定制数据绑定的外化 @InitBinder 方法

@InitBinder 方法也可以 定义在一个 @ControllerAdvice 注释 类在这种情况下它们适用于所有的控制器。 这提供了一个 替代使用 WebBindingInitializer 。

这个 @ControllerAdvice 注释是 一个组件注释允许实现类时自动被检测 通过类路径扫描。

支持“last - modified”响应头方便 内容缓存

一个 @RequestMapping 方法可能 希望支持 “last - modified” HTTP请求, 合同中定义的Servlet API的 getLastModified

方法,以促进内容 缓存。 这涉及到计算lastModified 长 值对于一个给定的要求,比较它 反对 “如果以后修改的 请求头 值,并可能返回一个响应状态码304(不是 修改)。 一个带注释的方法可以实现,作为控制器 如下:

```
@RequestMapping
public String myHandleMethod(WebRequest webRequest, Model model) {

   long lastModified = // 1. application-specific calculation

   if (request.checkNotModified(lastModified)) {
      // 2. shortcut exit - no further processing necessary
      return null;
    }

   // 3. or otherwise further request processing, actually preparing content
   model.addAttribute(...);
   return "myViewName";
}
```

有两个关键元素来注意:调用 request.checkNotModified(lastModified) 并返回 空 。 前者设置响应状态到304 它返回之前 真正的 。 后者,在组合 前者,使Spring MVC做任何进一步的处理 请求。

17.3.4A异步请求处理

Spring MVC 3.2引入Servlet 3基于异步请求 处理。 而不是返回一个值,像往常一样,一个控制器的方法 现在可以返回一个 java util并发调用 和生产返回值从一个单独的线程。 与此同时,主要的Servlet 集装箱的线程被释放并允许处理其他请求。 Spring MVC调用 可调用的 在一个 单独的线程的帮助下 TaskExecutor 当 可调用的 返回时, 请求被派回Servlet容器的简历 处理返回的值 可调用的 。 下面是一个示例控制器方法:

```
@RequestMapping(method=RequestMethod.POST)
public Callable<String> processUpload(final MultipartFile file) {

  return new Callable<String>() {
    public Object call() throws Exception {
      // ...
      return "someView";
    }
  };
}
```

第二个选择是控制器返回的一个实例 DeferredResult 。 在这种情况下,返回值 也将产生一个单独的线程。 然而,该线程不是已知的Spring MVC。 例如结果可能产生的反应 一些外部的事件,比如一个JMS消息,一个任务计划,等等。 下面是一个示例控制器方法:

```
@RequestMapping("/quotes")
@ResponseBody
public DeferredResult<String> quotes() {
  DeferredResult<String> deferredResult = new DeferredResult<String>();
  // Save the deferredResult in in-memory queue ...
  return deferredResult;
}

// In some other thread...
deferredResult.setResult(data);
```

这可能是很难理解没有任何知识的 异步处理功能,Servlet 3 它肯定会帮助去阅读它。 在一个非常最小考虑以下基本事实:

- 一个 ServletRequest 在异步模式下可以将通过调用吗 request.startAsync() 。 这样做的主要影响是 Servlet,以及任何过滤器,可以退出但响应 将继续开放允许其他线程来完成加工。
- 调用 request.startAsync() 返回一个 AsyncContext ,可用于 进一步控制异步处理。 例如它提供 该方法 调度 ,它可以被称为从一个 应用程序线程为了 “调度” 请求回 Servlet容器。 一个异步调度是类似于一个前锋 除了它是由一个线程(应用程序)到另一个 (Servlet容器)线程而向前发生同步 在相同的(Servlet容器)线程。
- ServletRequest 提供访问 当前 DispatcherType ,这 可以用来区分如果 Servlet 或 一个 滤波器 是处理 最初的请求处理线程,当它处理 一个异步调度。

与上面的想法,以下是序列 为异步请求的事件处理 可调用的 : (1)控制器返回 可调用的 ,(2)Spring MVC开始异步处理 并提交 可调用的 一个 TaskExecutor 在一个单独的线程处理,(3) DispatcherServlet 和所有过滤器的退出请求处理线程但响应 仍然是开放的,(4) 可调用的 产生的结果 和Spring MVC将请求返回Servlet容器, (5) DispatcherServlet 也会被再次调用和处理 简历与异步产生的结果 可调用的 。 确切的测序(2), (3)和(4)可能取决于执行的速度 并发线程。

事件的顺序为异步请求处理 DeferredResult 是相同的在主除了吗 它由应用程序产生异步源于一些线程: (1)控制器返回 DeferredResult 并将它保存 在一些内存中的队列或列表,可以访问它, (2)Spring MVC开始异步处理,(3) DispatcherServlet 和

所有已配置的过滤器的退出请求处理线程但响应 仍然是开放的,(4)应用程序设置 DeferredResult 从一些线程和Spring MVC将请求返回Servlet容器, (5) DispatcherServlet 也会被再次调用和处理 简历与异步产生结果。

解释为异步请求处理的动机和原因和使用它 超出了本文的范围。 为进一步的信息你可能想读 这篇博客系列 。

异常处理异步请求

会发生什么如果 可调用的 返回 从控制器方法引发一个异常而被执行? 效果类似于时所发生的任何控制器方法提出 一个例外。 它是由一个匹配 @ExceptionHandler 方法在相同的 控制器或通过一个配置的 HandlerExceptionResolver 实例。

> **注意**
>
> 在后台,当一个 可调用的 引发一个异常,Spring MVC仍然派遣到Servlet 集装箱恢复处理。 唯一的区别是,结果的执行 可调用的 是一个 异常 必须处理 与配置的 HandlerExceptionResolver 实例。

当使用一个 DeferredResult ,你有 选择调用它 setErrorResult(对象) 方法 并提供一个 异常 或任何其他对象 你想使用的结果。 如果结果是一个 异常 ,它将处理一个 匹配 @ExceptionHandler 方法 相同的控制器或任何配置 HandlerExceptionResolver 实例。

拦截异步请求

现有的 HandlerInterceptor 可以 实现 AsyncHandlerInterceptor ,这 提供了一个额外的方法 afterConcurrentHandlingStarted 。 这是调用异步处理开始后,当最初的 请求处理线程正在退出。 看到Javadoc的 AsyncHandlerInterceptor 为更多的细节 在那。

进一步选择生命周期回调是异步请求 直接提供 DeferredResult , 的方法 onTimeout(Runnable) 和 onCompletion(Runnable) 。 那些被称为当 异步请求是关于超时或分别完成了。 超时事件可以由设置 DeferredResult 一些值。 完成回调不过是最终的,结果不能 再设置。

类似的回调也可以用 可调用的 。 然而,您将需要包装 这个 可调用的 在一个实例的 WebAsyncTask 然后使用该注册 超时和完成回调。 就像 DeferredResult ,超时事件可以 处理和一个值可以返回而完成事件是终局的。

你也可以注册一个 CallableProcessingInterceptor 或 DeferredResultProcessingInterceptor 在全球范围内通过MVC Java配置或MVC名称空间。 那些拦截器提供一套完整的回调函数,并应用每一个 时间一个 可调用的 或 DeferredResult 是使用。

配置为异步请求处理

Servlet 3 Async配置

使用异步请求处理Servlet 3,您需要更新 web . xml 到3.0版本:

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  version="3.0">

  ...

 </web-app>
```

这个 DispatcherServlet 和任何 滤波器 配置需要 这个 <异步支持> < /异步支持>真实 子元件。 此外,任何 滤波器 这也需要参与异步分派也应该配置 支持异步分派器类型。 注意,它是安全的 要启用异步分派器类型提供的所有过滤器 Spring框架的,因为它们不会卷入async 除非需要分派。

如果使用Servlet 3,基于Java的配置,例如通过 WebApplicationInitializer ,你会 还需要设置 “asyncSupported “国旗以及 异步分派器类型就像 web . xml 。 为了简化所有这些配置,可以考虑 扩展 AbstractDispatcherServletInitializer 或 AbstractAnnotationConfigDispatcherServletInitializer , 自动设置这些选项,使它很容易注册吗 滤波器 实例。

Spring MVC Async配置

MVC Java配置和MVC名称空间都提供选项 配置异步请求处理。 WebMvcConfigurer 有方法 configureAsyncSupport 而<mvc:注解驱动的> 有一个<异步支持>子元素。

那些允许您配置默认的超时时间值使用 异步请求,如果没有设置取决于底层的Servlet 容器(例如。10秒在Tomcat)。 您还可以配置一个 AsyncTaskExecutor 用于执行 可调用的 实例返回 控制器方法。 强烈推荐配置该财产 因为默认情况下使用Spring MVC SimpleAsyncTaskExecutor 。 MVC Java配置 和MVC名称空间还允许您注册 CallableProcessingInterceptor 和 DeferredResultProcessingInterceptor 实例。

如果你需要覆盖默认的超时时间值 特定 DeferredResult ,您可以通过使用 适当的类的构造函数。 类似地,对于一个 可调用的 ,你可以把它封装在一个 WebAsyncTask 并使用适当的类 构造函数自定义超时值。 类的构造函数 WebAsyncTask 还允许提供一个 AsyncTaskExecutor 。

17.3.5A测试控制器

这个 弹簧测试 模块提供一流的支持 带注释的控制器进行测试。 看到 SectionA 11 3 6,一个测试FrameworkaSpring MVC 。

17.4处理程序映射

在早期版本的春天,用户需要定义一个或 更多的 HandlerMapping 豆子在web 应用程序上下文映射传入web请求到适当的处理程序。 通过引入注释控制器,您通常不需要 做,因为 RequestMappingHandlerMapping 自动查找 @RequestMapping 注释所有 controller 豆子。 然而,记住所有 HandlerMapping 类扩展从 AbstractHandlerMapping 有 以下属性,您可以使用自定义他们的 行为:

拦截器

拦截器使用的列表。 HandlerInterceptor 年代了 SectionA 17 4 1,一个拦截请求 HandlerInterceptor 一个 。

defaultHandler

默认的处理程序来使用,当这个处理程序映射不 结果在一个匹配的处理程序。

秩序

基于价值的顺序属性(请参阅 org.springframework.core.Ordered 接口), 弹簧类处理程序映射所有可用的上下文和 适用于第一个匹配的处理程序。

alwaysUseFullPath

如果 真正的 ,弹簧使用完整路径在 当前Servlet上下文找到一个适当的处理程序。 如果 假 (默认),在当前的路径 使用Servlet映射。 例如,如果一个Servlet映射使用 /测试/ * 和 alwaysUseFullPath 属性设置为true, /测试/:viewpage html 使用,而如果 属性设置为false, /:viewpage html 是 使用。

urlDecode

默认为 真正的 ,在Spring 2.5。 如果你 喜欢比较编码路径,设置此标志 假 。 然而, HttpServletRequest 总是暴露了Servlet路径在解码形式。 要知道Servlet路径将 不匹配相比,编码的路径。

下面的例子展示了如何配置一个拦截器:

```
<beans>
 <bean id="handlerMapping" class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping">
  <property name="interceptors">
   <bean class="example.MyInterceptor"/>
  </property>
 </bean>

<beans>
```

17.4.1A拦截请求 HandlerInterceptor

春天的处理程序映射机制包括处理程序拦截器, 这是有用的,当你想应用特定的功能吗 一定的要求,例如,检查主。

拦截器位于处理程序映射必须实现 HandlerInterceptor 从 org.springframework.web.servlet 包。 这 接口定义了三个方法: preHandle(. .) 是 称为 之前 实际的处理程序被执行; postHandle(. .) 叫做 在 处理程序被执行;和 afterCompletion(. .) 是 称为 在完整的请求完成 。 这三个方法应该提供足够的灵活性,可以做各种各样的 预处理和后处理。

这个 preHandle(. .) 方法返回一个布尔 值。 您可以使用这个方法来打破或继续加工 执行链。 当这个方法返回 真正的 , 处理程序执行链将继续;当它返回false, DispatcherServlet 假定拦截器本身 有照顾的请求(例如,并呈现一个合适的吗 视图)和不继续

执行其他拦截器和 实际处理程序在执行链。

拦截器可以配置使用 拦截器 房地产,这是目前在所有 HandlerMapping 类扩展从 AbstractHandlerMapping 。 这是显示在下面的例子:

```
<beans>
  <bean id="handlerMapping"
      class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping">
    <property name="interceptors">
      <list>
        <ref bean="officeHoursInterceptor"/>
      </list>
    </property>
  </bean>

  <bean id="officeHoursInterceptor"
      class="samples.TimeBasedAccessInterceptor">
    <property name="openingTime" value="9"/>
    <property name="closingTime" value="18"/>
  </bean>
<beans>
```

```
package samples;

public class TimeBasedAccessInterceptor extends HandlerInterceptorAdapter {

  private int openingTime;
  private int closingTime;

  public void setOpeningTime(int openingTime) {
    this.openingTime = openingTime;
  }

  public void setClosingTime(int closingTime) {
    this.closingTime = closingTime;
  }

  public boolean preHandle(
      HttpServletRequest request,
      HttpServletResponse response,
      Object handler) throws Exception {

    Calendar cal = Calendar.getInstance();
    int hour = cal.get(HOUR_OF_DAY);
    if (openingTime <= hour && hour < closingTime) {
      return true;
    } else {
      response.sendRedirect("http://host.com/outsideOfficeHours.html");
      return false;
    }
  }
}
```

任何请求由这种映射是拦截了 TimeBasedAccessInterceptor 。 如果当前时间 是办公时间外,用户被重定向到一个静态的HTML文件 说,例如,您只能访问该网站在办公室 小时。

注意

当使用 RequestMappingHandlerMapping 实际的处理程序 是一个实例的 HandlerMethod 哪一个 识别特定的控制器方法,将被调用。

正如您可以看到的,春天适配器类 HandlerInterceptorAdapter 让它更容易 扩展 HandlerInterceptor 接口。

提示

在上面的例子中,配置拦截器将适用 处理所有请求注释的控制器方法。 如果你想 缩小URL路径,拦截器应用,你可以使用 MVC名称空间或MVC Java配置,或声明bean实例 类型的 MappedInterceptor 那样做。 看到 启用MVC Java配置或MVC XML名称空间 。

17.5解决的观点

所有的MVC框架为web应用程序提供一种方法来解决 视图。 Spring提供了视图解析器,它使您能够渲染模型 在一个浏览器没有束缚你到一个特定的视图技术。 出 箱,春天使您能够使用jsp,Velocity模板和XSLT的观点, 例如。 看到 ChapterA 18, 视图技术 讨论如何 集成和使用一系列不同的视图技术。

这两个接口,是重要的弹簧处理 观点 ViewResolver 和 视图 。 这个 ViewResolver 提供之间的映射 视图名称和实际的观点。 这个 视图 接口地址准备请求和手请求 到一个视图技术。

17.5.1A解决意见的 ViewResolver 接口

作为讨论 SectionA 17.3,一个Controllersa实施 ,所有处理程序 方法在Spring Web MVC控制器必须解决一个逻辑视图 的名字,或者显式(如。 通过返回一个 字符串 , 视图 ,或 ModelAndView)或 隐式(即。 根据公约)。 在春天是解决意见 通过一个逻辑视图名称,由一个视图解析器解析。 春天到来 与相当多的视图解析器。 此表列出了他们中的大多数,一对夫妇 遵循的例子。

17.3为多。 一个视图解析器

ViewResolver	描述
AbstractCachingViewResolver	摘要视图解析器缓存视图。 经常意见 之前需要一些准备可以用它们;延长这一观点 解析器提供缓存。
XmlViewResolver	实施 ViewResolver 接受一个 配置文件写在XML和相同的DTD作为春天的 XML bean工厂。 默认的配置文件 / - inf /视图的xml 。
ResourceBundleViewResolver	实施 ViewResolver 使用bean 定义在一个 ResourceBundle , 指定的绑定包的名称。 通常你定义 包在一个属性文件,位于类路径中。 这个 默认文件名是 视图属性 。
UrlBasedViewResolver	简单实现的 ViewResolver 接口, 影响直接解决逻辑视图名称,网址, 没有一个明确的映射定义。 这是适当的,如果 你的逻辑名称匹配的名称查看资源在一个 直截了当的方式,不需要任意 映射。
InternalResourceViewResolver	方便子类的 UrlBasedViewResolver 支持 InternalResourceView (实际上,servlet 和jsp)和子类如 JstlView 和 TilesView 。 你可以指定视图 类的所有视图生成这个解析器使用 setViewClass(. .) 。 看到的Javadocs UrlBasedViewResolver 类 细节。
VelocityViewResolver / FreeMarkerViewResolver	方便子类的 UrlBasedViewResolver 支持 VelocityView (实际上,速度 模板)或 FreeMarkerView 分别和自定义子类的他们。
ContentNegotiatingViewResolver	实施 ViewResolver 接口, 解决一个视图基于请求文件名称或 接受 头。 看到 SectionA 17 5 4,一个 ContentNegotiatingViewResolver 一个 。

作为一个例子,使用JSP技术作为一个视图,您可以使用 UrlBasedViewResolver 。 这个视图解析器 翻译一个视图名称到一个URL和手请求到 RequestDispatcher呈现视图。

```
<bean id="viewResolver"
    class="org.springframework.web.servlet.view.UrlBasedViewResolver">
  <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>
  <property name="prefix" value="/WEB-INF/jsp/"/>
  <property name="suffix" value=".jsp"/>
</bean>
```

当返回 测试 作为一个逻辑视图名称, 这个视图解析器将请求转发到 RequestDispatcher ,将会发送请求 / - inf / jsp /测试jsp 。

当你把不同的视图技术在一个web应用程序, 您可以使用 ResourceBundleViewResolver :

```
<bean id="viewResolver"
    class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
  <property name="basename" value="views"/>
  <property name="defaultParentView" value="parentView"/>
</bean>
```

这个 ResourceBundleViewResolver 检查 ResourceBundle 被basename和 为每个视图它应该解决,它使用的价值 财产 [viewname]。(类) 作为视图类和 属性值 [viewname]url 作为视图的url。 的例子可以发现在第二章涵盖视图 技术。 正如您可以看到的,你可以确定一个父视图,从哪 所有视图的属性文件 一个 扩展 一个 。 这样你可以 指定一个默认的视图类,例如。

注意

子类的 AbstractCachingViewResolver 缓存视图实例,它们解决。 缓存可以提高性能 特定的视图技术。 可以关掉缓存的 设置 缓存 财产 假 。 此外,如果你必须刷新一定 查看在运行时(例如,当速度模板修改), 您可以使用 removeFromCache(String viewName,语言环境 loc) 法。

ViewResolvers 17.5.2A链接

Spring支持多个视图解析器。 因此你可以链 解析器,例如,覆盖特定视图在某些 情况下。 你链视图解析器通过添加不止一个解析器 你的应用程序上下文,如有必要,通过设置 秩序 属性来指定排序。 记住, 更高的顺序属性,后来被定位在视图解析器 链条。

在以下示例中,链的视图解析器由 两个解析器,一个 InternalResourceViewResolver , 这始终是自动定位为最后解析器在吗 链,一个 XmlViewResolver 用于指定 Excel的观点。 Excel的观点是不支持的 InternalResourceViewResolver 。

```
<bean id="jspViewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>
  <property name="prefix" value="/WEB-INF/jsp/"/>
  <property name="suffix" value=".jsp"/>
</bean>

<bean id="excelViewResolver" class="org.springframework.web.servlet.view.XmlViewResolver">
  <property name="order" value="1"/>
  <property name="location" value="/WEB-INF/views.xml"/>
</bean>

<!-- in views.xml -->

<beans>
  <bean name="report" class="org.springframework.example.ReportExcelView"/>
</beans>
```

如果一个特定的视图解析器不会导致一个视图,春天 检查其他视图解析器的上下文。 如果额外的视图 解析器存在,春天继续检查他们,直到一个视图 解决了。 如果没有视图解析器返回一个视图,春天会抛出一个 ServletException 。

合同的一个视图解析器指定一个视图解析器 可以 返回null指示视图无法 发现。 并不是所有的视图解析器做这个,但是,因为在某些情况下, 这个解析器不能检测是否存在的视图。 对于 的例子, InternalResourceViewResolver 使用 这个 RequestDispatcher 在内部,和调度 是唯一的方法来找出如果JSP是存在的,但这一行动只能 执行一次。 这同样适用于这个 VelocityViewResolver 和一些其他人。 检查 Javadoc视图解析器,查看它是否报告不存在的 视图。 因此,把一个 InternalResourceViewResolver 在链在 其他地方比过去,结果在连锁不充分 检查,因为 InternalResourceViewResolver 将 总是 返回一个视图!

17.5.3A重定向到视图

正如前面提到的,一个控制器通常返回一个逻辑 视图名称,一个视图解析器解析为一个特定的视图 技术。 对于视图jsp等技术处理 通过Servlet或JSP引擎,这个分辨率通常是处理 通过联合 InternalResourceViewResolver 和 InternalResourceView ,这问题一个内部 向前或包括通过Servlet API的 RequestDispatcher.forward(. .) 方法或 RequestDispatcher.include() 法。 对于其他视图 技术,如速度、XSLT,等等,视图本身写道 内容直接到响应流。

它有时需要发送一个HTTP重定向回 客户端,在视图呈现。 这是可取的,例如, 当一个控制器一直打电话 邮报 艾德数据, 和反应实际上是一个代表团到另一个控制器(用于 例成功表单提交)。 在这种情况下,一个正常的 内部的前锋将意味着其他控制器还可以看到 相同 邮报 数据,这可能是有问题的 它可以与其他预期数据混淆。 另一个理由去执行 重定向在显示结果的可能性,是消除用户提交表单数据多次。 在这个场景中, 浏览器会首先发送一个初始 邮报 ,它将 收到一个响应,重定向到一个不同的URL;最后是 浏览器将执行一个后续 得到 URL 命名在重定向响应。 因此,从的角度来看 浏览器,当前页面不反映的结果 邮报 而是一个 得到 。 最后 效果是没有办法用户可以不小心 Re - 邮报 相同的数据通过执行刷新。 这个 刷新部队一个 得到 结果的页面,而不是重新发送 最初的 邮报 数据。

RedirectView

力的一个方法一个重定向的结果作为一个控制器 响应是控制器创建和返回的一个实例 春天的 RedirectView 。 在这种情况下, DispatcherServlet 不使用正常的观点吗 解决机制。 而因为这被(定向) 视图已经, DispatcherServlet 简单 指示视图来做它的工作。

这个 RedirectView 问题一个 HttpServletResponse.sendRedirect() 称之为 返回到客户端浏览器作为一个HTTP重定向。 默认情况下,所有 模型属性被认为是公开为URI模板 变量在重定向URL。 剩下的那些属性 是基本类型或集合/数组的基本类型是 自动附加作为查询参数。

附加原始类型属性作为查询参数可能 期望的结果如果一个模型实例是专门为准备 这个重定向。 然而,在注释的控制器模型可能

包含 添加额外的属性(如拉呈现的目的 字段值)。 为了避免可能发生的有这样的属性 出现在URL带注释的控制器可以声明的一个参数 类型 RedirectAttributes 和使用它来 指定准确的属性来提供 RedirectView 。 如果控制器方法决定 重定向的内容 RedirectAttributes 是使用。 否则 模型的内容是使用。

注意,URI模板变量从当前请求 自动可用一个重定向URL时扩大和不 需要添加显式既不通过 模型 也 RedirectAttributes 。 例如:

```
@RequestMapping(value = "/files/{path}", method = RequestMethod.POST)
public String upload(...) {
   // ...
   return "redirect:files/{path}";
}
```

如果你使用 RedirectView 和观点是 由控制器本身,建议你配置 这个重定向URL可以注入到控制器,它不是 烤到控制器但中配置上下文以及 视图名称。 下一节将讨论这个过程。

这个 重定向: 前缀

而使用 RedirectView 做工精细, 如果控制器本身创建 RedirectView ,没有回避这个事实 这个控制器是意识到一个重定向是发生。 这是 真正的理想和夫妻的事情太紧。 控制器 不应该关心响应得到处理。 一般来说 它应该只操作视图名称而言,注射 到它。

特殊的 重定向: 前缀允许你 完成这个。 如果一个视图的名字是返回的前缀 重定向: , UrlBasedViewResolver (和所有的子类) 认出这是一个特殊的指示,一个重定向是必要的。 这个 剩下的视图名称将被视为 “重定向URL。

净效果是一样的,如果控制器已经回来一个 RedirectView ,但现在控制器本身可以 简单的操作方面的逻辑视图名称。 一个逻辑视图名称 如 重定向:/ /一些/ myapp资源 将重定向 相对于当前Servlet上下文,而一个名称如 重定向:http://myhost.com/some/arbitrary/path 将 重定向到一个绝对URL。

这个 转发: 前缀

也可以使用一个特殊的 转发: 前缀为视图名称,最终解决 UrlBasedViewResolver 和子类。 这 创建一个 InternalResourceView (说到底,一个 RequestDispatcher.forward()) 在其余的视图名称,这被认为是一个URL。 因此,这个前缀并不有用 InternalResourceViewResolver 和 InternalResourceView (用于jsp例如)。 但前缀可以帮助当你主要是使用另一个 视图技术,但仍想强迫一个向前的一个资源 由Servlet / JSP引擎。 (注意,您也可以链 多个视图解析器,相反。)

与 重定向: 前缀,如果视图 名称与 转发: 前缀是注入 控制器,该控制器不检测任何特别的是 发生在条款处理响应。

17.5.4A ContentNegotiatingViewResolver

这个 ContentNegotiatingViewResolver 不 解决的观点本身而是代表其他视图解析器, 选择视图,类似于表示请求的 客户端。 存在两种策略对客户端请求一个表示 从服务器:

- 使用一个不同的URI为每个资源,通常是通过使用一个 不同的文件扩展名的URI。 例如,URI http://www.example.com/users/fred.pdf 请求一个PDF 代表用户弗雷德和 http://www.example.com/users/fred.xml 请求一个 XML表示。
- 使用相同的URI为客户机来定位资源,但是 设置 接受 HTTP请求报头列出 媒体 类型 让它能够理解。 例如,一个HTTP请求 http://www.example.com/users/fred 与一个 接受 标题设置为 应用程序/ pdf 请求一个PDF代表用户弗雷德,虽然 http://www.example.com/users/fred 与一个 接受 标题设置为 text / xml 请求一个XML表示。 这种策略被称为 内容谈判 。

  注意

  一个问题 接受 头是它 是不可能把它在一个web浏览器在HTML。 例如,在 Firefox,它是固定的:

  ```
  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
  ```

  因为这个原因,它被普遍地认为使用一个不同的URI 对于每个表示当开发基于浏览器的web 应用程序。

支持多种表示形式的资源,Spring提供了 这个 ContentNegotiatingViewResolver 解决 视图基于文件扩展名或 接受 头的 HTTP请求。 ContentNegotiatingViewResolver 不执行视图决议本身而是代表一个吗 列表的视图解析器,您指定通过bean属

性 ViewResolvers 。

这个 ContentNegotiatingViewResolver 选择 一个适当的 视图 来处理请求 比较请求媒体类型(s)与媒体类型(也称为 内容类型)支持的 视图 它的每个相关 ViewResolvers 。 第一 视图 在列表中,有一个兼容的 内容类型 返回表示 客户端。 如果一个兼容视图不能提供的 ViewResolver 链,然后视图列表 指定通过 DefaultViews 属性将 咨询了。 后一种选择是适合单例 视图 这可以使一个合适的 表示当前资源不管逻辑视图 的名字。 这个 接受 头可能包含通配符, 例子 text / * ,在这种情况下一个 视图 其内容类型是 text / xml 是一个兼容匹配。

支持这一决议的一个视图基于文件扩展名,使用 这个 ContentNegotiatingViewResolver bean属性 媒体类型 指定一个映射的文件扩展名来 媒体类型。 更多信息算法用于确定 请求媒体类型,参考的API文档 ContentNegotiatingViewResolver 。

下面是一个示例配置 ContentNegotiatingViewResolver:

```
<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
 <property name="mediaTypes">
  <map>
   <entry key="atom" value="application/atom+xml"/>
   <entry key="html" value="text/html"/>
   <entry key="json" value="application/json"/>
  </map>
 </property>
 <property name="viewResolvers">
  <list>
   <bean class="org.springframework.web.servlet.view.BeanNameViewResolver"/>
   <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <property name="suffix" value=".jsp"/>
   </bean>
  </list>
 </property>
 <property name="defaultViews">
  <list>
   <bean class="org.springframework.web.servlet.view.json.MappingJackson2JsonView" />
  </list>
 </property>
</bean>


<bean id="content" class="com.springsource.samples.rest.SampleContentAtomView"/>
```

这个 InternalResourceViewResolver 处理 翻译的视图名称和JSP页面,而 BeanNameViewResolver 返回一个视图基于 bean 的名称。 (见 “ 解决视图和 ViewResolver接口 “为更多的细节在Spring如何查找 和实例化视图)。 在这个例子中, 内容 豆是一个类,继承 AbstractAtomFeedView ,该方法将返回一个Atom RSS 饲料。 更多的信息创建一个Atom提要表示,看到 原子的部分观点。

在上面的配置,如果请求是由一个 . html 扩展,视图解析器寻找一个视图 匹配的 text / html 媒体类型。 这个 InternalResourceViewResolver 提供 匹配视图 text / html 。 如果请求 该文件的扩展 原子 ,视图解析器 查找匹配的一个视图 应用程序/ atom + xml 媒体类型。 这种观点是 提供的 BeanNameViewResolver 这地图 这个 SampleContentAtomView 如果视图名称 返回 内容 。 如果请求是由 文件扩展 json , MappingJackson2JsonView 实例从 DefaultViews 列表将会被选择不管 视图名称。 另外,客户端请求可以没有一个文件 扩展但 接受 标题设置为 首选媒体类型,相同的分辨率要求视图 发生。

注意

如果 ContentNegotiatingViewResolver 的列表 ViewResolvers的配置不明确,它会自动使用 任何 ViewResolvers定义在应用程序上下文。

对应的控制器代码,返回一个原子的RSS提要 对于一个URI的形式 http://localhost/content.atom 或 http://localhost/content 与一个 接受 头的应用程序/原子+ xml显示 下面。

```
@Controller
public class ContentController {

   private List<SampleContent> contentList = new ArrayList<SampleContent>();

   @RequestMapping(value="/content", method=RequestMethod.GET)
   public ModelAndView getContent() {
      ModelAndView mav = new ModelAndView();
      mav.setViewName("content");
      mav.addObject("sampleContentList", contentList);
      return mav;
   }

}
```

17.6一个使用flash属性

Flash属性提供了一种方法为一个请求来存储属性 用于另一个。 这是最常用的重新定向时 一个例如, Post /重定向/得到 模式。 Flash 属性保存在重定向(通常是暂时的 会话)可用请求重定向和后 立即删除。

Spring MVC有两个主要的抽象支持flash的属性。 FlashMap 是用来保存flash属性而 FlashMapManager 用于存储、检索、和管理 FlashMap 实例。

Flash属性支持总是和不需要启用 如果不使用显式虽然,不会引起HTTP会话创建。 在 每个请求都有一个“输入” FlashMap 与属性通过从先前的请求(如果有的话)和一个“输出” FlashMap 用属性来保存为一个后续 请求。 两 FlashMap 实例都可以访问 从任何地方在Spring MVC通过静态方法在 RequestContextUtils 。

带注释的控制器通常不需要使用 FlashMap 直接。 相反一个 @RequestMapping 方法可以接受一个 类型的参数, RedirectAttributes 和使用 它添加闪光属性一个重定向的场景。 Flash属性添加 通过 RedirectAttributes 自动 传播到 FlashMap“输出”。 同样的重定向后 属性从“输入” FlashMap 是 自动添加到 模型 的 控制器为目标URL。

匹配请求flash属性

flash属性的概念存在于许多其他网络 框架和已被证明是暴露有时并发问题。 这是因为根据定义flash属性存储到 下一个请求。 然而非常“下一步”请求可能不是 但另一个异步请求接收者(如轮询或 资源请求)在这种情况下,flash属性是除去了 早期的。

减少这类问题的可能性, RedirectView 自动“邮票” FlashMap 实例与路径和查询 参数的目标URL重定向。 反过来,违约 FlashMapManager 匹配信息 传入的请求当查找“输入” FlashMap 。

这并不排除并发问题的可能性 但是它大大减少完全与信息 中已有的重定向URL。 因此使用flash 属性是推荐主要用于重定向的场景。

17.7建筑 URI 年代

Spring MVC提供一个机制来构建和编码URI 使用 UriComponentsBuilder 和 UriComponents 。

例如你可以扩大和编码URI模板字符串:

```
UriComponents uriComponents =
    UriComponentsBuilder.fromUriString("http://example.com/hotels/{hotel}/bookings/{booking}").build();

URI uri = uriComponents.expand("42", "21").encode().toUri();
```

注意, UriComponents 是不可变的, 这个 扩大() 和 encode() 操作返回的新实例,如果必要的。

你也可以扩大和编码使用单个URI组件:

```
UriComponents uriComponents =
    UriComponentsBuilder.newInstance()
      .scheme("http").host("example.com").path("/hotels/{hotel}/bookings/{booking}").build()
      .expand("42", "21")
      .encode();
```

在一个Servlet环境了 ServletUriComponentsBuilder 子类提供了 静态工厂方法可以复制URL信息 Servlet请求:

```
HttpServletRequest request = ...

// Re-use host, scheme, port, path and query string
// Replace the "accountId" query param

ServletUriComponentsBuilder ucb =
    ServletUriComponentsBuilder.fromRequest(request).replaceQueryParam("accountId", "{id}").build()
      .expand("123")
      .encode();
```

或者,你可以选择复制一个子集的可用 信息,包括上下文路径:

```
// Re-use host, port and context path
// Append "/accounts" to the path

ServletUriComponentsBuilder ucb =
      ServletUriComponentsBuilder.fromContextPath(request).path("/accounts").build()
```

或在某些情况下 DispatcherServlet 映射 的名字(例如。 /主/ *),你也可以有文字部分 servlet映射的包括:

```
// Re-use host, port, context path
// Append the literal part of the servlet mapping to the path
// Append "/accounts" to the path

ServletUriComponentsBuilder ucb =
      ServletUriComponentsBuilder.fromServletMapping(request).path("/accounts").build()
```

17.8一个使用场所

春天的大部分地区的架构支持国际化, 就像Spring web MVC框架执行。 DispatcherServlet 使您能够自动 解决信息使用客户的语言环境。 这是做了 LocaleResolver 对象。

当一个请求出现时, DispatcherServlet 寻找一个locale解析器, 如果找到一个试图用它来设置语言环境。 使用 RequestContext.getLocale() 法,你总是可以的 检索地区,是解决区域分解器。

除了自动区域分辨率,也可以附加一个 拦截器来处理程序映射(请参阅 SectionA 17 4 1,一个拦截请求 HandlerInterceptor 一个 更多信息 处理程序映射拦截器)来改变在特定的语言环境 情况下,例如,基于参数在请求。

语言环境解析器和拦截器中定义 org.springframework.web.servlet.i18n 包, 在你的应用程序上下文配置以正常的方式。 这是一个 选择的区域包括在春天解析器。

17.8.1A AcceptHeaderLocaleResolver

此区域设置解析器检查 接收语言 头在请求发送 由客户端(例如。 ,一个web浏览器)。 通常这头字段包含 现场客户的操作系统。

17.8.2A CookieLocaleResolver

此区域设置解析器检查 饼干 这 可能存在于客户端,看看一个地区被指定。 如果是这样,它 使用指定的语言环境。 使用这个语言环境的属性解析器 你可以指定cookie的名称以及最高年龄。 找到 下面的例子定义了一个 CookieLocaleResolver 。

```
<bean id="localeResolver" class="org.springframework.web.servlet.i18n.CookieLocaleResolver">

   <property name="cookieName" value="clientlanguage"/>

   <!-- in seconds. If set to -1, the cookie is not persisted (deleted when browser shuts down) -->
   <property name="cookieMaxAge" value="100000">

</bean>
```

为多17 4 a CookieLocaleResolver 属性

财产	默认	描述
cookieName	类名+地区	cookie的名称
cookieMaxAge	整数马克斯int	最大时间一个cookie将保持持久的 客户端。 如果指定1,cookie将持久存储;它 将只提供到客户端关闭他或她吗 浏览器。
cookiePath	/	限制了能见度的cookie的某一部分 你的网站。 当cookiePath被指定,cookie只会 可见,它下面路径和路径。

17.8.3A SessionLocaleResolver

这个 SessionLocaleResolver 允许你 从会话中检索地区可能相关 用户的请求。

17.8.4A LocaleChangeInterceptor

你可以改变通过添加不同的地区 LocaleChangeInterceptor 的处理程序 映射(见 SectionA 17.4,一个处理程序mappingsa)。 它将检测 参数在请求和改变语言环境。 它调用 setLocale() 在 LocaleResolver 这也存在 上下文。 下面的例子显示,调用所有 *视图 资源包含一个参数命名的 siteLanguage 现在将改变语言环境。 因此,对于 例,对以下URL的请求, http://www.sf.net/home.view?siteLanguage=nl 将 修改网站的语言来荷兰。

```
<bean id="localeChangeInterceptor"
    class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
  <property name="paramName" value="siteLanguage"/>
</bean>

<bean id="localeResolver"
    class="org.springframework.web.servlet.i18n.CookieLocaleResolver"/>

<bean id="urlMapping"
    class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
  <property name="interceptors">
    <list>
      <ref bean="localeChangeInterceptor"/>
    </list>
  </property>
  <property name="mappings">
    <value>/**/*.view=someController</value>
  </property>
</bean>
```

17.9一个使用主题

17.9.1A主题的概述

你可以使用Spring Web MVC框架主题设置总体 您的应用程序的外观和感觉,从而增强用户体验。 一个 主题是一组静态资源,典型的样式表和 图片,影响视觉风格的应用程序。

17.9.2A定义主题

使用主题在你的web应用程序,您必须设置一个 实施 org.springframework.ui.context.ThemeSource 接口。 这个 WebApplicationContext 接口扩展 themeSource 但 代表它的职责,一个专用的实现。 默认情况下 代表将一个 org.springframework.ui.context.support.ResourceBundleThemeSource 实现,加载属性文件的根 类路径。 使用一个自定义 themeSource 实现或配置基本名称的前缀 ResourceBundleThemeSource ,你可以注册一个 在应用程序上下文bean与保留的名字 themeSource 。 web应用程序上下文 自动检测一个bean叫这个名字的,使用它。

当使用 ResourceBundleThemeSource ,一个 主题是一个简单的属性文件中定义的。 这个 资源属性文件列出了构成主题。 这是一个 示例:

```
styleSheet=/themes/cool/style.css
background=/themes/cool/img/coolBg.jpg
```

的键属性的名称,参考主题 元素从视图代码。 对于一个JSP,你通常使用 春天:主题 自定义标签,这是非常相似的 春天:消息 标签。 以下JSP片段使用 主题定义在前面的示例自定义外观和 感觉:

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<html>
  <head>
    <link rel="stylesheet" href="<spring:theme code='styleSheet'/>" type="text/css"/>
  </head>
  <body style="background=<spring:theme code='background'/>">
    ...
  </body>
</html>
```

默认情况下, ResourceBundleThemeSource 使用一个空的基本名称前缀。 因此,属性文件 从根的加载类路径。 因此你会把酷属性 主题定义在一个目录 根的类路径,例如,在 / - inf /类 。 这个 ResourceBundleThemeSource 使用标准Java 资源包加载机制,允许完整 国际化的主题。 例如,我们可以有一个 / - inf /类/酷问属性 引用一个 特殊的背景图像与荷兰文本它。

17.9.3A主题解析器

在你定义的主题,就像在前一节,你决定 这主题来使用。 这个 DispatcherServlet 将 寻找一个bean命名 ThemeResolver 找出哪一个 ThemeResolver 实现 使用。 一个主题解析器工作在同样的方式作为一个 LocaleResolver 。 它检测到的主题 使用一个特定的请求,还可以更改请求的主题。 这个 以下主题解析器提供了春天:

为多17 5 ThemeResolver 实现

类	描述
FixedThemeResolver	选择一个固定的主题,设置使用 defaultThemeName 财产。
SessionThemeResolver	主题是保存在用户的HTTP会话。 它 只需要设置一次,每个会话,但不坚持 在会话之间。
CookieThemeResolver	选定的主题是存储在一个cookie的 客户端。

春天还提供了一个 ThemeChangeInterceptor 这允许主题变化 在每个请求与一个简单的请求参数。

17.10一个弹簧的多部分(文件上传)支持

17.10.1A介绍

春天的内置多部分支持处理文件上传在网络 应用程序。 你使这几部分的支持可插入的 MultipartResolver 对象中定义的 org.springframework.web.multipart 包。 春天 提供了一个 MultipartResolver 实现使用 Commons FileUpload 和另一个使用Servlet 3.0 多部分请求解析。

默认情况下,弹簧没有几部分的处理,因为一些 开发人员要处理multiparts本身。 你使弹簧 多部分处理通过添加一个多部分解析器到web 应用程序的上下文。 每个请求检查,看它是否包含一个 多部分。 如果没有多部分是发现,继续按照预期的要求。 如果一个多部分是发现在请求, MultipartResolver 这已被宣布在你 上下文使用。 在那之后,在你的请求多部分的属性 像对待任何其他属性。

17.10.2A使用 MultipartResolver 与 Commons FileUpload

下面的例子显示了如何使用 CommonsMultipartResolver :

```
<bean id="multipartResolver"
   class="org.springframework.web.multipart.commons.CommonsMultipartResolver">

   <!-- one of the properties available; the maximum file size in bytes -->
   <property name="maxUploadSize" value="100000"/>
</bean>
```

当然你也需要把合适的罐子在你 类路径的多部分解析器工作。 在的情况下 CommonsMultipartResolver ,你需要使用 commons fileupload jar 。

当春天 DispatcherServlet 检测到一个 多部分请求,解析器,它可以激活已宣布进入 你的上下文并移交请求。 这个解析器然后包装了 电流 HttpServletRequest 成 MultipartHttpServletRequest 支持 多部分文件上传。 使用 MultipartHttpServletRequest ,你可以得到 multiparts信息包含在这个请求和实际 获得了多部分文件本身在你的控制器。

17.10.3A使用 MultipartResolver 与 Servlet 3.0

为了使用基于Servlet 3.0几部分的解析,你需要 标志着 DispatcherServlet 与 “多部分配置” 部分 web . xml ,或与一个 javax.servlet.MultipartConfigElement 在 编程Servlet登记,或在案件的一个自定义的Servlet类 可能与 javax.servlet.annotation.MultipartConfig 注释您的Servlet类。 配置设置如最大 大小或存储位置需要被应用在Servlet 登记水平作为Servlet 3.0不允许这些设置 从MultipartResolver完成。

一旦Servlet 3.0多部分解析已经使在其中的一个 上述方法可以添加 StandardServletMultipartResolver 你的春天 配置:

```
<bean id="multipartResolver"
   class="org.springframework.web.multipart.support.StandardServletMultipartResolver">
</bean>
```

17.10.4A处理一个文件上传表单中

后 MultipartResolver 完成了 工作,请求处理像任何其他。 首先,创建一个表格 一个文件输入,将允许用户上传表单。 编码 属性(enctype = "多部分/格式数据")让 浏览器知道如何编码表单作为多部分要求:

```
<html>
  <head>
    <title>Upload a file please</title>
  </head>
  <body>
    <h1>Please upload a file</h1>
    <form method="post" action="/form" enctype="multipart/form-data">
      <input type="text" name="name"/>
      <input type="file" name="file"/>
      <input type="submit"/>
    </form>
  </body>
</html>
```

下一步是创建一个控制器处理文件 上传。 这个控制器是非常类似 正常带注释的 controller ,不过我们用的 MultipartHttpServletRequest 或 MultipartFile 在方法参数:

```
@Controller
public class FileUploadController {

  @RequestMapping(value = "/form", method = RequestMethod.POST)
  public String handleFormUpload(@RequestParam("name") String name,
    @RequestParam("file") MultipartFile file) {

    if (!file.isEmpty()) {
      byte[] bytes = file.getBytes();
      // store the bytes somewhere
      return "redirect:uploadSuccess";
    } else {
      return "redirect:uploadFailure";
    }
  }

}
```

请注意如何 @RequestParam 方法 参数映射到输入元素中声明的形式。 在这个 的例子,没有完成了 byte[] ,但在 练习你可以将它保存在一个数据库,它存储在文件系统, 等等。

当使用Servlet 3.0多部分解析您还可以使用 javax.servlet.http.Part 的方法 参数:

```
@Controller
public class FileUploadController {

  @RequestMapping(value = "/form", method = RequestMethod.POST)
  public String handleFormUpload(@RequestParam("name") String name,
    @RequestParam("file") Part file) {

    InputStream inputStream = file.getInputStream();
    // store bytes from uploaded file somewhere

    return "redirect:uploadSuccess";
  }

}
```

17.10.5A处理文件上传请求从编程的客户

多部分的请求也可以从非浏览器提交客户 在RESTful服务的场景。 上面的例子和 配置应用在这里。 然而,与浏览器 一般提交文件和简单的表单字段,一个程序化的客户机可以 还发送更复杂的数据的一个特定内容类型一个例如一个 多部分的请求与文件和第二部分与JSON格式的数据:

```
POST /someUrl
Content-Type: multipart/mixed

--edt7Tfrdusa7r3lNQc79vXuhIIMlatb7PQg7Vp
Content-Disposition: form-data; name="meta-data"
Content-Type: application/json; charset=UTF-8
Content-Transfer-Encoding: 8bit

{
  "name": "value"
}
--edt7Tfrdusa7r3lNQc79vXuhIIMlatb7PQg7Vp
Content-Disposition: form-data; name="file-data"; filename="file.properties"
```

```
Content-Type: text/xml
Content-Transfer-Encoding: 8bit
... File Data ...
```

你可以访问部分命名为“元数据” @RequestParam(“元数据”)的字符串 元数据 控制器方法参数。 然而,你将 可能更愿意接受一个强类型对象的初始化 JSON格式的数据在请求的主体部分,非常相似 方式 @RequestBody 皈依者的身体 非多部分请求到目标对象的帮助 HttpMessageConverter 。

您可以使用 @RequestPart 注释而不是 @RequestParam 注释为这个目的。 它允许你的内容 具体多部分通过一个 HttpMessageConverter 考虑 这个 ” 内容类型的 头的多部分:

```
@RequestMapping(value="/someUrl", method = RequestMethod.POST)
public String onSubmit(@RequestPart("meta-data") MetaData metadata,
                       @RequestPart("file-data") MultipartFile file) {
   // ...

}
```

注意 MultipartFile 方法参数 可以通过 @RequestParam 或 与 @RequestPart 可以互换使用。 然而, @RequestPart(“元数据”)的元数据 方法参数在本例中是解读为JSON内容基于它 ” 内容类型的 头和转换的帮助 这个 MappingJackson2HttpMessageConverter 。

17.11一个处理异常

17.11.1A HandlerExceptionResolver

春天 HandlerExceptionResolver 实现 处理意想不到的过程中发生的异常控制器执行。 一个 HandlerExceptionResolver 有点类似于 异常映射可以定义在web应用程序描述符 web . xml 。 然而,他们提供一个更灵活的方式 这样做。 例如他们提供信息处理程序是 执行当异常抛出。 此外,一种编程方式 处理异常的给你更多的选择回应 适当地在请求被转发到另一个URL(相同的 最终的结果是当你使用Servlet特定异常 映射)。

除了实现 HandlerExceptionResolver 接口,它 仅仅是一种实现吗 resolveException(异常,处理程序) 方法和 返回一个 ModelAndView ,您还可以使用提供的 SimpleMappingExceptionResolver 或创建 @ExceptionHandler 方法。 这个 SimpleMappingExceptionResolver 使您能够把类名的任何异常,可能被抛出 并将其映射到一个视图名称。 这是功能相当于异常映射特性从Servlet API,但是也有可能 实现更细的粒度映射的异常来自不同 处理程序。 这个 @ExceptionHandler 注释另一方面可以用在方法应该调用来处理一个 例外。 这种方法可能是在一个局部定义的 controller 或者可以申请在全球所有 @RequestMapping 方法内定义的时候 一个 @ControllerAdvice 类。 下面更详细地解释这个。

17.11.2A @ExceptionHandler

这个 HandlerExceptionResolver 接口 和 SimpleMappingExceptionResolver 实现 允许你地图例外声明以及一些特定视图可选的Java逻辑转发之前那些观点。 然而,在某些情况下, 特别是当依靠 @ResponseBody 方法 而不是视图解析,它会更方便直接设置 地位的反应和可选地写错误内容的主体 响应。

你可以通过 @ExceptionHandler 方法。 当宣布在一个控制器等方法应用到异常 提出 @RequestMapping 方法的 contoroller(或它的任何子类)。 你也可以声明一个 @ExceptionHandler 方法在一个 @ControllerAdvice 在这种情况下,这类处理异常从 @RequestMapping 方法从任何控制器。 这个 @ControllerAdvice 注释是 一个组件的注解,可以使用类路径扫描。 这是 当使用MVC自动启用名称空间和MVC的Java配置, 或者取决于 ExceptionHandlerExceptionResolver 配置或不是。 下面是一个例子,一个控制器局部 @ExceptionHandler 方法:

```
@Controller
public class SimpleController {


  // @RequestMapping methods omitted ...


  @ExceptionHandler(IOException.class)
  public ResponseEntity<String> handleIOException(IOException ex) {

    // prepare responseEntity

    return responseEntity;
  }

}
```

这个 @ExceptionHandler 值可以设置为 异常类型的数组。 如果抛出一个异常,匹配一个 这个类型的列表,然后注释的方法与匹配 @ExceptionHandler 将被调用。 如果 注释值没有设置然后异常类型列为法 使用的参数。

就像标准控制器的方法一个 @RequestMapping 注释,方法参数 和返回值的 @ExceptionHandler 方法 可以灵活。 例如, HttpServletRequest 在Servlet可以访问吗 环境和 PortletRequest 在Portlet 环境。 返回类型可以是一个 字符串 , 这是解释为一个视图名称、一个吗 ModelAndView 对象, ResponseEntity 或者你也可以添加 @ResponseBody 有返回值的方法 转化为消息转换器和写入响应流。

17.11.3A处理标准的Spring MVC例外

Spring MVC可能提高很多异常而处理 一个请求。 这个 SimpleMappingExceptionResolver 可以很容易地 地图任何例外一个默认的错误观点所需要的。 然而,在处理客户解释反应在一个自动化的 你会想要设置特定的响应状态代码。 根据 异常提出了状态代码可能表明客户端错误(4 xx)或一个 服务器错误(5 xx)。

这个 DefaultHandlerExceptionResolver 翻译 Spring MVC例外特定的错误状态码。 这是注册 默认名称空间与MVC,MVC Java配置和 这个 DispatcherServlet (即不使用MVC 名称空间或Java配置)。 下面列出的是一些异常处理 这个解析器和相应的状态代码:

异常	HTTP状态代码
BindException	400(错误请求)
ConversionNotSupportedException	500(内部服务器错误)
HttpMediaTypeNotAcceptableException	406(不接受)
HttpMediaTypeNotSupportedException	415(不支持的媒体类型)
HttpMessageNotReadableException	400(错误请求)
HttpMessageNotWritableException	500(内部服务器错误)
HttpRequestMethodNotSupportedException	405(法不允许)
MethodArgumentNotValidException	400(错误请求)
MissingServletRequestParameterException	400(错误请求)
MissingServletRequestPartException	400(错误请求)
NoSuchRequestHandlingMethodException	404(未找到)
TypeMismatchException	400(错误请求)

这个 DefaultHandlerExceptionResolver 作品 透明地通过设置状态的反应。 但是,物权法 写任何错误内容的主体,而你的反应应用程序可能需要添加内容到每个错误的开发者 反应例如当提供一个REST API。 你可以准备一个 ModelAndView 和 渲染错误内容通过视图解析——即通过配置一个 ContentNeogitatingViewResolver , MappingJacksonJsonView ,等等。 然而,你可能更喜欢使用 @ExceptionHandler 方法相反。

如果你喜欢写错误内容通过 @ExceptionHandler 方法可以扩展 ResponseEntityExceptionHandler 相反。 这是一个方便的基地 @ControllerAdvice 类提供一个 @ExceptionHandler 方法来处理标准的Spring MVC异常和返回 ResponseEntity 。 这允许您定制响应 和写错误内容与消息转换器。 看到Javadoc的 ResponseEntityExceptionHandler 为更多的细节。

17.11.4A注释与业务异常 @ResponseStatus

业务异常可以标注 @ResponseStatus 。 当发生异常, 这个 ResponseStatusExceptionResolver 处理它的 设置相应的状态响应。 默认情况下, DispatcherServlet 寄存器 这个 ResponseStatusExceptionResolver 和它是 可供使用。

17.11.5A定制错误页面的默认Servlet容器

当响应的状态被设置为一个错误状态码 和身体的反应是空的,Servlet容器通常呈现 一个HTML格式的错误页面。 自定义错误页面默认的容器,你可以 声明一个 <页面偷走了> 元素 web . xml 。 直到Servlet 3,该元素必须 被映射到一个特定的状态代码或异常类型。 从 Servlet 3一个错误页面不需要映射,有效 意味着指定位置定制默认Servlet容器 错误页面。

```
<error-page>
  <location>/error</location>
```

```
</error-page>
```

请注意,实际的位置错误页面可以是一个 JSP页面或其他一些URL在容器内包括一个处理 通过一个 controller 方法:

当编写错误信息,状态代码和错误消息 设置 HttpServletResponse 可以 通过请求属性在一个控制器:

```
@Controller
public class ErrorController {

    @RequestMapping(value="/error", produces="application/json")
    @ResponseBody
    public Map<String, Object> handle(HttpServletRequest request) {

        Map<String, Object> map = new HashMap<String, Object>();
        map.put("status", request.getAttribute("javax.servlet.error.status_code"));
        map.put("reason", request.getAttribute("javax.servlet.error.message"));

        return map;
    }

}
```

或在一个JSP:

```
<%@ page contentType="application/json" pageEncoding="UTF-8"%>
{
  status:<%=request.getAttribute("javax.servlet.error.status_code") %>,
  reason:<%=request.getAttribute("javax.servlet.error.message") %>
}
```

17.12一个约定优于配置支持

对于许多项目,坚持既定惯例和 有合理的默认值正是他们(项目)的需要,和 Spring Web MVC现在有明确的支持 约定优于 配置 。 这意味着如果你建立一个集 的命名约定和其他类似的东西,你可以 大幅 减少数量的配置 需要设置处理程序映射视图解析器, ModelAndView 实例,等等。这是一个巨大的好处 至于快速成型,也可以借一个程度的(总是 好有)一个代码库的一致性你应该选择 推进到生产。

约定优于配置支持地址三个核心领域 MVC:模型、视图和控制器。

17.12.1A控制器 ControllerClassNameHandlerMapping

这个 ControllerClassNameHandlerMapping 类 是 HandlerMapping 实现 使用约定来确定请求url之间的映射和 控制器 实例来处理 这些请求。

考虑以下简单的 控制器 实现。 要特别 注意的 名称 的类。

```
public class ViewShoppingCartController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) {
        // the implementation is not hugely important for this example...
    }
}
```

这里是一个片段从相应的Spring Web MVC 配置文件:

```
<bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping"/>

<bean id="viewShoppingCart" class="x.y.z.ViewShoppingCartController">
    <!-- inject dependencies as required... -->
</bean>
```

这个 ControllerClassNameHandlerMapping 发现 所有的不同的处理程序(或 控制器)豆定义在它 应用程序上下文和条 控制器 了 的名字来定义其处理程序映射。 因此, ViewShoppingCartController 映射到 / viewshoppingcart * 请求的URL。

让我们看看一些例子,以便中心思想变得 熟悉。 (注意所有小写的url,相比 骆驼下套管 控制器 类 的名字。)

- WelcomeController 映射到 /欢迎* 请求URL
- HomeController 映射到 / home * 请求URL
- IndexController 映射到 /指数* 请求URL
- RegisterController 映射到 /寄存器* 请求URL

对于 MultiActionController 处理程序类,产生的映射是稍微复杂。 这个 控制器 名称在以下 例子被认为是 MultiActionController 实现:

- AdminController 映射到 / admin / * 请求URL
- CatalogController 映射到 /目录/ * 请求URL

如果你遵循约定的命名你的 控制器 实现作为 XXX **控制器** , ControllerClassNameHandlerMapping 保存你 单调的定义和维护一个潜在的 了 SimpleUrlHandlerMapping (或岗位)。

这个 ControllerClassNameHandlerMapping 类 扩展了 AbstractHandlerMapping 基类所以 您可以定义 HandlerInterceptor 实例和其他一切就像许多其他 HandlerMapping 实现。

17.12.2A模型 ModelMap (ModelAndView)

这个 ModelMap 类本质上是一个 荣耀 地图 这可以使添加 对象将显示在(或在)一个 视图 坚持一个共同的命名 公约。 考虑以下 控制器 实现;请注意 对象添加到 ModelAndView 没有任何 指定相关的名称。

```
public class DisplayShoppingCartController implements Controller {

   public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) {

      List cartItems = // get a List of CartItem objects
      User user = // get the User doing the shopping

      ModelAndView mav = new ModelAndView("displayShoppingCart"); <-- the logical view name

      mav.addObject(cartItems); <-- look ma, no name, just the object
      mav.addObject(user); <-- and again ma!

      return mav;
   }
}
```

这个 ModelAndView 类使用一个 ModelMap 类,是一种自定义 地图 实现自动 生成一个键为一个对象,当一个对象被添加到它。 这个 战略确定的名字添加对象的例子 一个标量对象如 用户 ,使用短 类名称的对象的类。 下面的例子是名称 产生的标量对象放到一个吗 ModelMap 实例。

- 一个 x y用户 实例添加将 这个名字 用户 生成的。
- 一个 x y登记 实例添加将 你的名字 登记 生成的。
- 一个 x y foo 实例添加将有 名称 foo 生成的。
- 一个 java . util . hashmap 实例添加将 你的名字 HashMap 生成的。 你可能 要明确这个名字在这种情况下,因为 HashMap 小于直观。
- 添加 空 将导致一个 IllegalArgumentException 被扔。 如果 对象(或对象),你可以添加 空 ,那你还会想要明确的 这个名字。

这个策略来生成一个名称后添加 集 或 列表 是窥集合,带吗 简短的类名称的第一个集合中对象,并使用它 与 列表 附加到的名字。 这同样适用于 与 数组的数组虽然没有必要了解数组 内容。 几个例子将名字的语义生成的集合更清晰:

- 一个 x y用户[] 数组和零个或更多 x y用户 添加元素将有名字 userList 生成的。
- 一个 x y foo[] 数组和零个或更多 x y用户 添加元素将有名字 fooList 生成的。
- 一个 java util arraylist 与一个或多个 x y用户 添加元素将有名字 userList 生成的。
- 一个 java util hashset 与一个或多个 x y foo 添加元素将有名字 fooList 生成的。
- 一个 **空** java util arraylist 将不会被添加 (实际上, addObject(. .) 调用将 基本上不做任何操作)。

> **什么,没有自动多元化?**
>
> 约定优于配置Spring Web MVC的支持并不 支持自动多元化。 也就是说,您不能添加 列表 的 人 对象 ModelAndView 和有 生成的名称是 人们 。
>
> 这个决定是经过一番讨论, 一个 最少意外原则 一个 赢得了 结束。

17.12.3A视图- RequestToViewNameTranslator

这个 RequestToViewNameTranslator 接口决定了一个逻辑 视图 名称 当没有这样的逻辑视图的名字是显式地提供。 它只有一个 的实现, DefaultRequestToViewNameTranslator 类。

这个 DefaultRequestToViewNameTranslator 地图 请求url逻辑视图名称,与这个例子:

```
public class RegistrationController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) {
        // process the request...
        ModelAndView mav = new ModelAndView();
        // add data as necessary to the model...
        return mav;
        // notice that no View or logical view name has been set
    }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- this bean with the well known name generates view names for us -->
    <bean id="viewNameTranslator"
        class="org.springframework.web.servlet.view.DefaultRequestToViewNameTranslator"/>

    <bean class="x.y.RegistrationController">
        <!-- inject dependencies as necessary -->
    </bean>

    <!-- maps request URLs to Controller names -->
    <bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping"/>

    <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <property name="suffix" value=".jsp"/>
    </bean>

</beans>
```

注意在实施 handleRequest(. .) 方法没有 视图 或逻辑视图的名字是曾经上设置 这个 ModelAndView 返回。 这个 DefaultRequestToViewNameTranslator 肩负着 生成 逻辑视图名称 从的URL 请求。 对于上面的 RegistrationController , 用于 结合 ControllerClassNameHandlerMapping ,一个请求的URL 的 http://localhost/registration.html 结果在一个 逻辑视图名称的 登记 产生了 这个 DefaultRequestToViewNameTranslator 。 这 逻辑视图的名字是然后解决到 / - inf / jsp /注册jsp 视图的 InternalResourceViewResolver bean。

提示

你不需要定义一个 DefaultRequestToViewNameTranslator bean 明确。 如果你喜欢的默认设置 DefaultRequestToViewNameTranslator ,你可以 依赖于Spring Web MVC DispatcherServlet 到 实例化 这个类的一个实例,如果一个没有明确 配置。

当然,如果你需要更改默认设置,那么你做的 需要配置自己的 DefaultRequestToViewNameTranslator bean 明确。 咨询的综合Javadoc DefaultRequestToViewNameTranslator 类 详细的各种属性,这些属性可以配置。

17.13一个ETag支持

一个 Etag (实体标记)是一个HTTP响应头返回的HTTP / 1.1兼容 web服务器用来确定变化在给定的URL的内容。 它可以 被认为是更复杂的继承者 last - modified 头。 当一个服务器返回一个 交涉的ETag头,客户端可以使用这个头 后续会,在一个 没有头。 如果 内容没有改变,服务器返回 304:不 修改 。

支持ETags是由Servlet过滤器 ShallowEtagHeaderFilter 。 这是一个普通的Servlet 过滤器,因此可用于任何web框架的结合。 这个 ShallowEtagHeaderFilter 过滤器创建所谓的 浅ETags(相对于深ETags,稍后详细说明), 过滤器缓存内容呈现的JSP(或其他内容), 生成MD5散列,并返回,作为ETag头的 响应。 下次一个客户端发送一个请求相同的资源,它 使用散列的 没有 值。 过滤器检测到这,再次呈现视图,并比较这两个散列。 如果他们 都是平等的, 304年 返回。 这个过滤器将不会保存 处理能力,作为视图仍呈现。 它唯一保存 是带宽,呈现响应不是送过去吗 线。

你配置 ShallowEtagHeaderFilter 在 web . xml :

```
<filter>
  <filter-name>etagFilter</filter-name>
    <filter-class>org.springframework.web.filter.ShallowEtagHeaderFilter</filter-class>
</filter>

<filter-mapping>
```

```
  <filter-name>etagFilter</filter-name>
  <servlet-name>petclinic</servlet-name>
</filter-mapping>
```

17.14基于Servlet容器初始化

在一个Servlet 3.0 +环境中,您可以选择配置 Servlet容器通过编程作为一种替代或结合 一个 web . xml 文件。 下面是一个例子,注册一个 DispatcherServlet :

```
import org.springframework.web.WebApplicationInitializer;

public class MyWebApplicationInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext container) {
        XmlWebApplicationContext appContext = new XmlWebApplicationContext();
        appContext.setConfigLocation("/WEB-INF/spring/dispatcher-config.xml");

        ServletRegistration.Dynamic registration = container.addServlet("dispatcher", new DispatcherServlet(appContext));
        registration.setLoadOnStartup(1);
        registration.addMapping("/");
    }

}
```

WebApplicationInitializer 是一个接口 Spring MVC提供确保你实现检测和 自动使用初始化任何Servlet 3容器。 一个抽象的基 类实现的 WebApplicationInitializer 命名 AbstractDispatcherServletInitializer 使它甚至 容易注册 DispatcherServlet 通过简单地覆盖 方法来指定servlet映射和位置的 DispatcherServlet 配置:

```
public class MyWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return null;
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[] { MyWebConfig.class };
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }

}
```

上面的例子是一个应用程序,它使用基于java的春天 配置。 如果使用基于xml的Spring配置,扩大直接从 AbstractDispatcherServletInitializer :

```
public class MyWebAppInitializer extends AbstractDispatcherServletInitializer {

    @Override
    protected WebApplicationContext createRootApplicationContext() {
        return null;
    }

    @Override
    protected WebApplicationContext createServletApplicationContext() {
        XmlWebApplicationContext cxt = new XmlWebApplicationContext();
        cxt.setConfigLocation("/WEB-INF/spring/dispatcher-config.xml");
        return cxt;
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }

}
```

AbstractDispatcherServletInitializer 还提供了 一个方便的方式来添加 滤波器 实例 ,让他们自动映射到 DispatcherServlet :

```
public class MyWebAppInitializer extends AbstractDispatcherServletInitializer {

    // ...

    @Override
    protected Filter[] getServletFilters() {
```

```
        return new Filter[] { new HiddenHttpMethodFilter(), new CharacterEncodingFilter() };
    }

}
```

每个过滤器添加缺省名称根据其具体类型和 自动映射到 DispatcherServlet 。

这个 isAsyncSupported 保护方法的 AbstractDispatcherServletInitializer 提供 一个地方,使异步支持 DispatcherServlet 和 所有过滤器映射到它。 默认情况下这个标志被设置为 真正的 。

17.15一个配置Spring MVC

SectionA 17 2 1,一个特殊Bean类型 WebApplicationContext 一个 和 SectionA 17 2 2,一个默认DispatcherServlet Configurationa 解释 Spring MVC的特别的豆子和默认的实现 使用的 DispatcherServlet 。 在本节中,您将了解两个额外的方法 配置Spring MVC。 即MVC Java配置和 MVC XML名称空间。

MVC Java配置和MVC命名空间提供 类似的默认配置,覆盖 这个 DispatcherServlet 违约。 目标是多余的大多数应用程序不必必须创建相同的配置和也 提供高级的结构配置 Spring MVC,作为一个简单的起点和 几乎不需要先验知识的基础 配置。

你可以选择MVC Java配置或 MVC名称空间取决于你的偏好。 也正如你 会看到进一步的下面,与MVC Java配置吗 容易看到底层的配置以及 使细粒度直接定制 创建了Spring MVC豆子。 但让我们从头开始。

17.15.1A启用MVC Java配置或MVC XML名称空间

启用MVC Java配置添加注释 @EnableWebMvc 你的 @ configuration 类:

```
@Configuration
@EnableWebMvc
public class WebConfig {

}
```

达到相同的XML使用 mvc:注解驱动的 元素:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:mvc="http://www.springframework.org/schema/mvc"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/mvc
      http://www.springframework.org/schema/mvc/spring-mvc.xsd">

   <mvc:annotation-driven />

</beans>
```

上面的注册一个 RequestMappingHandlerMapping ,一个 RequestMappingHandlerAdapter ,和一个 ExceptionHandlerExceptionResolver (等等) 在处理请求的支持与注释的控制器方法使用 注释(如 @RequestMapping , @ExceptionHandler ,和其他人。

它还支持以下:

1. 弹簧3风格类型转换通过 conversionService 实例 除了JavaBeans PropertyEditors用于数据绑定。
2. 支持 格式化 号码 字段使用 @NumberFormat 注释通过 conversionService 。
3. 支持 格式化 日期, 日历,长,Joda时间字段使用 @DateTimeFormat 注释。
4. 支持 验证 controller 输入与 @Valid , 如果一个jsr - 303提供者是出现在类路径中。
5. HttpMessageConverter支持 @RequestBody 方法 参数和 @ResponseBody 方法返回值 @RequestMapping 或 @ExceptionHandler 方法。
   这是HttpMessageConverters的完整列表的设置 mvc:注解驱动:
   - ByteArrayHttpMessageConverter 将字节数组。
   - StringHttpMessageConverter 转换字符串。
   - ResourceHttpMessageConverter 皈依/从 org.springframework.core.io.Resource 所有的媒体类型。
   - SourceHttpMessageConverter 皈依/从一个 javax.xml.transform.Source 。
   - FormHttpMessageConverter 将表单数据到/从一个 MultiValueMap <字符串, 字符串> 。
   - Jaxb2RootElementHttpMessageConverter 将Java对象来/从XML补充说如果JAXB2是 出现在类路径中。

- MappingJackson2HttpMessageConverter (或 MappingJacksonHttpMessageConverter) 转换为一个/从JSON补充说如果杰克逊2(杰克逊)存在 在类路径中。
- AtomFeedHttpMessageConverter 将Atom提要一个补充说如果罗马上是否存在 类路径。
- RssChannelHttpMessageConverter 将RSS提要一个补充说如果罗马上是否存在 类路径。

17.15.2A定制提供的配置

自定义默认配置在Java中你只是 实现 WebMvcConfigurer 接口或更有可能的扩展类 WebMvcConfigurerAdapter 和覆盖 你需要的方法。 下面是一个例子,一些可用的 方法覆盖。 看到 WebMvcConifgurer 对于 一个列表的所有方法和Javadoc详情:

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

  @Override
  protected void addFormatters(FormatterRegistry registry) {
    // Add formatters and/or converters
  }

  @Override
  public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {
    // Configure the list of HttpMessageConverters to use
  }

}
```

自定义的缺省配置 < mvc:注解驱动/ > 检查什么 属性和子元素它支持。 您可以查看 Spring MVC的XML模式 或使用代码完成功能的IDE来发现 什么属性和子元素都是可用的。 下面的示例显示了一个子集,什么是可用的:

```
<mvc:annotation-driven conversion-service="conversionService">
    <mvc:message-converters>
        <bean class="org.example.MyHttpMessageConverter"/>
        <bean class="org.example.MyOtherHttpMessageConverter"/>
    </mvc:message-converters>
</mvc:annotation-driven>

<bean id="conversionService" class="org.springframework.format.support.FormattingConversionServiceFactoryBean">
    <property name="formatters">
        <list>
            <bean class="org.example.MyFormatter"/>
            <bean class="org.example.MyOtherFormatter"/>
        </list>
    </property>
</bean>
```

17.15.3A配置拦截器

您可以配置 HandlerInterceptors 或 WebRequestInterceptors 应用 所有传入的请求或限制到特定的URL路径模式。

拦截器的一个示例Java注册:

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

  @Override
  public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(new LocaleInterceptor());
    registry.addInterceptor(new ThemeInterceptor()).addPathPatterns("/**").excludePathPatterns("/admin/**");
    registry.addInterceptor(new SecurityInterceptor()).addPathPatterns("/secure/*");
  }

}
```

在XML使用 < mvc:拦截器> 元素:

```
<mvc:interceptors>
    <bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor" />
    <mvc:interceptor>
        <mapping path="/**"/>
        <exclude-mapping path="/admin/**"/>
        <bean class="org.springframework.web.servlet.theme.ThemeChangeInterceptor" />
    </mvc:interceptor>
    <mvc:interceptor>
        <mapping path="/secure/*"/>
        <bean class="org.example.SecurityInterceptor" />
    </mvc:interceptor>
</mvc:interceptors>
```

17.15.4A配置内容协商

与Spring框架盯着3.2,您可以配置Spring MVC如何 决定了从客户端请求媒体类型为请求映射 以及对内容协商的目的。 可用的选项是 检查文件的扩展名在请求URI, “接受” 头, 一个请求参数,以及回到默认内容类型。 默认情况下,文件扩展名在请求URI首先被检查和 “接受” 头检查下。

对于文件扩展名在请求URI,MVC Java配置和 MVC名称空间,自动注册这样的扩展 json , . xml , . rss ,和 原子 如果 相应的依赖关系如杰克逊,JAXB2或罗马 出现在类路径中。 额外的扩展可能是不需要的 注册明确如果他们可以发现通过 ServletContext.getMimeType(字符串) 或 Java激活框架 (见 javax.activation.MimetypesFileTypeMap)。

下面是一个例子,定制内容协商 选择通过MVC Java配置:

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

  @Override
  public void configureContentNegotiation(ContentNegotiationConfigurer configurer) {
    configurer.favorPathExtension(false).favorParameter(true);
  }
}
```

在MVC名称空间, < mvc:注解驱动的> 元素 有 内容协商经理 属性,它预计 ContentNegotiationManager 这反过来又可以被创建 与 ContentNegotiationManagerFactoryBean :

```
<mvc:annotation-driven content-negotiation-manager="contentNegotiationManager" />

<bean id="contentNegotiationManager" class="org.springframework.web.accept.ContentNegotiationManagerFactoryBean">
   <property name="favorPathExtension" value="false" />
   <property name="favorParameter" value="true" />
   <property name="mediaTypes" >
      <value>
         json=application/json
         xml=application/xml
      </value>
   </property>
</bean>
```

如果不使用MVC Java配置或MVC名称空间,你需要 创建一个实例 ContentNegotiationManager 和用它来配置 RequestMappingHandlerMapping 为请求映射 的目的, RequestMappingHandlerAdapter 和 ExceptionHandlerExceptionResolver 对于 内容协商的目的。

注意, ContentNegotiatingViewResolver 现在还可以配置一个 ContentNegotiatingViewResolver , 所以你可以使用一个实例在Spring MVC。

在更先进的情况下,它可能是有用的配置 多个 ContentNegotiationManager 实例 这反过来可能包含自定义 ContentNegotiationStrategy 实现。 例如你可以配置 ExceptionHandlerExceptionResolver 与 一个 ContentNegotiationManager 总是 解析请求的媒体类型 “application / json” 。 或者你可能想塞一个自定义策略,有一些逻辑选择 一个默认内容类型(如XML或JSON)如果没有内容 类型被要求。

17.15.5A配置视图控制器

这是一个快捷方式定义 ParameterizableViewController ,立即 当调用转发到一个视图。 使用它在静态情况下没有 Java控制器逻辑视图生成之前执行 响应。

一个例子的转发请求 “/” 一个视图称为 “家” 在Java中:

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

  @Override
  public void addViewControllers(ViewControllerRegistry registry) {
    registry.addViewController("/").setViewName("home");
  }

}
```

和相同的XML使用 < mvc:视图控制器> 元素:

```
<mvc:view-controller path="/" view-name="home"/>
```

17.15.6A配置服务资源

这个选项允许静态资源请求的遵循某一个特定的 URL模式可以服务 ResourceHttpRequestHandler 从任何的名单 资源 位置。 这提供了一个方便的 方法为静态资源从地点以外的网络 应用程序根,包括定位在类路径中。 这个 缓存时间 财产可以用于设置远的未来 过期标头(1年是推荐的优化工具 如页面速度和YSlow),这样他们就会更有效率 利用客户端。 正确的处理程序也评估 last - modified 标题(如果存在),这样一个 304年 将返回状态代码为合适,避免 对资源的不必要的开销已经缓存了 客户端。 例如,服务资源请求的URL模式 /资源/ * * 从 公共资源 在web应用程序的根目录中您将使用:

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

  @Override
  public void addResourceHandlers(ResourceHandlerRegistry registry) {
    registry.addResourceHandler("/resources/**").addResourceLocations("/public-resources/");
  }

}
```

和相同的XML:

```
<mvc:resources mapping="/resources/**" location="/public-resources/"/>
```

为这些资源与未来一年到期保证 最大限度的使用浏览器缓存和减少HTTP请求了 由浏览器:

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

  @Override
  public void addResourceHandlers(ResourceHandlerRegistry registry) {
    registry.addResourceHandler("/resources/**").addResourceLocations("/public-resources/").setCachePeriod(31556926);
  }

}
```

和XML:

```
<mvc:resources mapping="/resources/**" location="/public-resources/" cache-period="31556926"/>
```

这个 映射 属性必须是一个蚂蚁模式,可以 使用 SimpleUrlHandlerMapping ,和 位置 属性必须指定一个或多个有效的资源 目录的位置。 可以指定多个资源位置使用 一个逗号分隔的值列表。 指定的地点将 在指定的订单检查存在的资源为任何 给定的请求。 例如,为了使服务资源的 web应用程序的根和从一个已知的道路 / meta - inf /公共web资源/ 在任何jar的 类路径使用:

```
@EnableWebMvc
@Configuration
public class WebConfig extends WebMvcConfigurerAdapter {

  @Override
  public void addResourceHandlers(ResourceHandlerRegistry registry) {
    registry.addResourceHandler("/resources/**")
      .addResourceLocations("/", "classpath:/META-INF/public-web-resources/");
  }

}
```

和XML:

```
<mvc:resources mapping="/resources/**" location="/, classpath:/META-INF/public-web-resources/"/>
```

当服务资源可能会改变当一个新版本的 应用程序部署,建议您将一个 版本字符串映射模式用于请求资源, 所以,你可能会迫使客户请求新部署的版本的 你的应用程序的资源。 这样一个版本字符串可以参数化的 和访问使用?可以轻易地在一个单一的 地方当部署新版本。

作为一个例子,我们考虑一个应用程序,它使用一个 性能优化的自定义构建(推荐)的Dojo JavaScript库在生产,构建一般 部署在web应用程序在一个路径 /公共资源/ dojo /设置dojo . js 。 因为不同地区的 Dojo可能被纳入定制构建为每个新版本的 应用程序,客户机web浏览器需要被迫 下载,定制的 应该 资源任何时候一个 新版本的应用程序部署。 一个简单的方法来实现这个 会来管理版本的应用程序在一个属性文件, 如:

```
application.version=1.0.0
```

然后让这个属性文件的值访问? 作为一个bean使用 util:属性 标签:

```
<util:properties id="applicationProps" location="/WEB-INF/spring/application.properties"/>
```

应用程序现在可以通过版本?,我们可以 合并到使用 资源 标签:

```
<mvc:resources mapping="/resources-#{applicationProps['application.version']}/**" location="/public-resources/"/>
```

在Java中,您可以使用 @PropertySouce 注释,然后注入 环境 抽象来访问所有定义的属性:

```
@Configuration
@EnableWebMvc
@PropertySource("/WEB-INF/spring/application.properties")
public class WebConfig extends WebMvcConfigurerAdapter {

  @Inject Environment env;

  @Override
  public void addResourceHandlers(ResourceHandlerRegistry registry) {
    registry.addResourceHandler("/resources-" + env.getProperty("application.version") + "/**")
      .addResourceLocations("/public-resources/");
  }

}
```

最后,请求资源与适当的URL,我们可以 利用弹簧JSP标签:

```
<spring:eval expression="@applicationProps['application.version']" var="applicationVersion"/>

<spring:url value="/resources-{applicationVersion}" var="resourceUrl">
    <spring:param name="applicationVersion" value="${applicationVersion}"/>
</spring:url>

<script src="${resourceUrl}/dojo/dojo.js" type="text/javascript"> </script>
```

17.15.7A mvc:默认servlet处理程序

该标签允许映射 DispatcherServlet 到 “/” (因此覆盖映射的容器的默认Servlet), 同时仍然允许静态资源请求处理 容器的默认Servlet。 它配置一个 DefaultServletHttpRequestHandler 与一个URL映射 “/ * *” 和优先级最低的相对于其他URL映射。

这个处理程序将所有请求转发到默认Servlet。 因此它是重要的,它仍然是去年在订单的所有其他 url HandlerMappings 。 这将是这样的如果你使用 < mvc:注解驱动的> 或者如果你是 设置您自己的自定义 HandlerMapping 实例是 确保将其 秩序 财产价值低于 的 DefaultServletHttpRequestHandler ,这是 integer . max_value之间 。

启用这个特性使用默认设置使用:

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

  @Override
  public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {
    configurer.enable();
  }

}
```

或在XML:

```
<mvc:default-servlet-handler/>
```

这个警告覆盖” / “Servlet映射是 RequestDispatcher 为默认Servlet必须检索 的名字而不是路径。 这个 DefaultServletHttpRequestHandler 将尝试 自动检测默认Servlet容器在启动时,使用 一组已知的名字对大多数主要的Servlet容器 (包括Tomcat、Jetty,GlassFish、JBoss、树脂、WebLogic、和 WebSphere)。 如果默认Servlet被自定义配置一个 不同的名称,或者如果一个不同的Servlet容器是用在哪里的 默认Servlet的名字是未知的,那么默认Servlet的名称 必须显式地提供像下面的例子:

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

  @Override
  public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {
```

```
    configurer.enable("myCustomDefaultServlet");
  }

}
```

或在XML:

```
<mvc:default-servlet-handler default-servlet-name="myCustomDefaultServlet"/>
```

17.15.8A Spring Web MVC资源更多

看到下面的链接和更多资源的指针对Spring Web MVC:

- 有许多优秀的文章和教程,展示如何 与Spring MVC构建web应用程序。 读他们的 春天 文档 页面。
- 一个 专家Spring Web MVC和网络流量 一个 由赛斯Ladd 和其他人(由Apress出版出版的)是一个出色的硬拷贝的来源 Spring Web MVC善良。

17.15.9A高级定制与MVC Java配置

正如你可以看到从上面的例子,MVC Java配置和 MVC命名空间提供更高级别的构造,不 需要深入了解底层的豆子为您创建。 相反,它可以帮助你专注于你的应用程序的需要。 不过,在某些时候你可能需要更细粒度的控制 或者你可能只是想了解底层的配置。

第一步更细粒度的控制是看到 潜在的豆子为您创建。 在MVC Java配置你可以 看到Javadoc和 @ bean 方法在 WebMvcConfigurationSupport 。 在这个类的配置自动导入 通过 @EnableWebMvc 注释。 事实上如果你打开 @EnableWebMvc 你可以 看到 @ import 语句。

接下来的一步是更细粒度的控制 自定义属性在一个bean创建 WebMvcConfigurationSupport 或者 提供自己的实例。 这需要两件事情—— 删除 @EnableWebMvc 注释以防止进口然后 扩展直接从 WebMvcConfigurationSupport 。 这里是一个例子:

```
@Configuration
public class WebConfig extends WebMvcConfigurationSupport {

  @Override
  public void addInterceptors(InterceptorRegistry registry){
    // ...
  }

  @Override
  @Bean
  public RequestMappingHandlerAdapter requestMappingHandlerAdapter() {

      // Create or let "super" create the adapter
      // Then customize one of its properties
  }

}
```

注意,修改豆子这样并不妨碍 你使用任何高级的结构在前面显示 本节。

17.15.10A高级定制与MVC名称空间

细粒度控制配置创建 你有点难度,与MVC名称空间。

如果你需要这样做,而不是复制 配置它提供,考虑配置一个 BeanPostProcessor 检测到 你要定制的bean类型,然后修改它 属性是必要的。 例如:

```
@Component
public class MyPostProcessor implements BeanPostProcessor {

  public Object postProcessBeforeInitialization(Object bean, String name) throws BeansException {
    if (bean instanceof RequestMappingHandlerAdapter) {
      // Modify properties of the adapter
    }
  }

}
```

注意, MyPostProcessor 需要 包含在一个 <组件扫描/ > 为了让它被发现或如果你喜欢你可以声明它 显式地包含XML bean声明。

18。 一个视图技术

18.1一个介绍

的领域之一是春天擅长分离视图 技术从其余的MVC框架。 例如,决定 使用速度或XSLT代替现有的JSP是主要的问题 配置。 本章涵盖了主要的视图技术工作 与春天,触动简要如何添加新的。 本章 假设您已经熟悉 SectionA 17.5,一个解决viewsa 涵盖的基本观点通常是连接到MVC吗 框架。

18.2一个JSP & JSTL

Spring提供了开箱即用的解决方案的几个JSP和 JSTL的观点。 使用JSP或JSTL是通过使用一个正常定义视图解析器 在 WebApplicationContext 。 此外, 当然你需要编写一些jsp,实际上会呈现 视图。

注意

设置应用程序以使用JSTL是一个常见的错误, 主要是由不同的servlet规范混乱、JSP和JSTL 版本号,它们意味着什么,以及如何正确地声明标记库。 本文 如何引用和使用JSTL在您的Web应用程序吗 提供了一个 有用的指南常见的陷阱和如何避免它们。 注意,在 Spring 3.0,最低支持servlet版本是2.4(JSP 2.0和JSTL 1.1), 这样可以减少的范围有所混淆。

18.2.1A视图解析器

就像任何其他视图技术你结合 春天,你将需要一个jsp视图解析器,将解决你 视图。 最常用的视图解析器在开发与jsp 是 InternalResourceViewResolver 和 ResourceBundleViewResolver 。 两者都是宣布在 这个 WebApplicationContext :

```
<!-- the ResourceBundleViewResolver -->
<bean id="viewResolver" class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
  <property name="basename" value="views"/>
</bean>

# And a sample properties file is uses (views.properties in WEB-INF/classes):
welcome.(class)=org.springframework.web.servlet.view.JstlView
welcome.url=/WEB-INF/jsp/welcome.jsp

productList.(class)=org.springframework.web.servlet.view.JstlView
productList.url=/WEB-INF/jsp/productlist.jsp
```

正如您可以看到的, ResourceBundleViewResolver 需要一个属性 文件定义视图名称映射到1)类和2)一个URL。 与 ResourceBundleViewResolver 你可以混合不同的 类型的视图只使用一个解析器。

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>
  <property name="prefix" value="/WEB-INF/jsp/"/>
  <property name="suffix" value=".jsp"/>
</bean>
```

这个 InternalResourceBundleViewResolver 可以 被配置为使用jsp如上所述。 作为最佳实践,我们 强烈建议将JSP文件目录下的一个目录 ' - inf' 目录,所以可以 是没有直接访问客户。

18.2.2A '普通' jsp和JSTL

当使用Java标准标记库,您必须使用一个特殊的 视图类, JstlView ,JSTL需要一些 准备在诸如I18N特性将工作。

18.2.3A附加标记促进发展

Spring提供了数据绑定的请求参数命令 在早些章节描述的对象。 开发方便 JSP页面在结合这些数据绑定功能,弹簧 提供了一些标记,可以使事情变得更简单。 所有的标签都有春天 HTML转义 特性来启用或禁用 转义字符。

标记库描述符(TLD)包含在 spring-webmvc.jar 。 进一步的信息关于个人标签中可以找到 附录题为 AppendixA G, 弹簧tld 。

18.2.4A使用Spring标记库的形式

自版本2.0起,Spring提供了一组全面的数据 绑定知道标签处理表单元素当使用JSP和弹簧 Web MVC。 每个标签提供了支持的属性集 相应的HTML标记对应,使标签熟悉的和 直观的使用。 这个标签生成的HTML是HTML 4.01 / XHTML 1.0 兼容的。

与其他形式/输入标记库,弹簧的形式标记库 结合Spring Web MVC,给标签访问命令吗 对象和参考数据控制器处理。 正如你将看到的 下面的例子,使jsp表单标签更易于开发,阅读 和维护。

让我们浏览一下形式标记和看一个例子,每个 标签是使用。 我们已经包括生成的HTML片段,某些标签 需要进一步的评论。

配置

表单标记库是捆绑在 spring-webmvc.jar 。 库描述符被称为 弹簧形式tld 。

使用标签从这个库,添加以下指令 顶你的JSP页面:

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

... 在 形式 是标记名称前缀吗 想要使用这个库的标签。

这个 形式 标签

这个标签呈现HTML的形式” 标签,并公开一个绑定路径 到内标签绑定。 它把命令对象 PageContext 因此,命令对象可以 访问内部标签。 所有其他的标签在这个图书馆 嵌套标签的吗 形式 标签 。

让我们假定我们有一个域对象称为 用户 。 这是一个JavaBean属性如 FirstName 和 lastName 。 我们将 把它作为我们的形式支持对象形式控制器返回 形式jsp 。 下面是一个例子 形式jsp 将看起来像:

```
<form:form>
    <table>
      <tr>
        <td>First Name:</td>
        <td><form:input path="firstName" /></td>
      </tr>
      <tr>
        <td>Last Name:</td>
        <td><form:input path="lastName" /></td>
      </tr>
      <tr>
        <td colspan="2">
          <input type="submit" value="Save Changes" />
        </td>
      </tr>
    </table>
 </form:form>
```

这个 FirstName 和 lastName 值检索从命令对象放置在 PageContext 通过页面控制器。 请继续阅读,看到更复杂的例子使用的内部标签 与 形式 标签。

生成的HTML看上去像一个标准形式:

```
<form method="POST">
    <table>
     <tr>
        <td>First Name:</td>
        <td><input name="firstName" type="text" value="Harry"/></td>
     </tr>
     <tr>
        <td>Last Name:</td>
        <td><input name="lastName" type="text" value="Potter"/></td>
     </tr>
     <tr>
        <td colspan="2">
         <input type="submit" value="Save Changes" />
        </td>
     </tr>
    </table>
 </form>
```

前面的JSP假设变量名的形式 支持对象是 “命令” 。 如果你有把 形式支持对象到模型的另一个名称(绝对下 最佳实践),那么您可以绑定到命名的变量的形式像 所以:

```
<form:form commandName="user">
    <table>
      <tr>
        <td>First Name:</td>
        <td><form:input path="firstName" /></td>
      </tr>
      <tr>
        <td>Last Name:</td>
        <td><form:input path="lastName" /></td>
```

```
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
</form:form>
```

这个 输入 标签

这个标签呈现HTML的输入的标签使用绑定值 和类型= '文本'默认情况下。 例如,这个标签,看到 一个章节的 形式 taga 。 从 Spring 3.1 您可以使用其他类型这样的html5特定类型如“电子邮件”，“电话”、“日期”,和其他人。

这个 复选框 标签

这个标签呈现HTML的输入的标记类型 “复选框”。

让我们假设我们的 用户 有偏好 如通迅订阅和一系列的爱好。 下面是一个 的例子 偏好 类:

```
public class Preferences {

    private boolean receiveNewsletter;

    private String[] interests;

    private String favouriteWord;

    public boolean isReceiveNewsletter() {
        return receiveNewsletter;
    }

    public void setReceiveNewsletter(boolean receiveNewsletter) {
        this.receiveNewsletter = receiveNewsletter;
    }

    public String[] getInterests() {
        return interests;
    }

    public void setInterests(String[] interests) {
        this.interests = interests;
    }

    public String getFavouriteWord() {
        return favouriteWord;
    }

    public void setFavouriteWord(String favouriteWord) {
        this.favouriteWord = favouriteWord;
    }
}
```

这个 形式jsp 将看起来像:

```
<form:form>
    <table>
        <tr>
            <td>Subscribe to newsletter?:</td>
            <%-- Approach 1: Property is of type java.lang.Boolean --%>
            <td><form:checkbox path="preferences.receiveNewsletter"/></td>
        </tr>

        <tr>
            <td>Interests:</td>
            <td>
                <%-- Approach 2: Property is of an array or of type java.util.Collection --%>
                Quidditch: <form:checkbox path="preferences.interests" value="Quidditch"/>
                Herbology: <form:checkbox path="preferences.interests" value="Herbology"/>
                Defence Against the Dark Arts: <form:checkbox path="preferences.interests"
                    value="Defence Against the Dark Arts"/>
            </td>
        </tr>
        <tr>
            <td>Favourite Word:</td>
            <td>
                <%-- Approach 3: Property is of type java.lang.Object --%>
                Magic: <form:checkbox path="preferences.favouriteWord" value="Magic"/>
            </td>
        </tr>
    </table>
</form:form>
```

有三个方法 复选框 标签 这应该满足你所有需求。复选框

- 方法一,当绑定值的类型 java朗布尔 , 输入(复选框) 被标记为'检查'如果 绑定值是 真正的 。 这个 价值 属性对应的解决 价值的 setValue(对象) 价值 财产。
- 方法2 -当绑定值的类型 数组 或 java util收集 , 输入(复选框) 被标记为'检查'如果 配置 setValue(对象) 价值存在于 绑定 收集 。
- 方法3 -任何其他绑定值类型, 输入(复选框) 被标记为'检查'如果 配置 setValue(对象) 等于 绑定值。

注意,无论方法,相同的HTML结构 生成的。 下面是一个HTML片段的复选框:

```
<tr>
    <td>Interests:</td>
    <td>
       Quidditch: <input name="preferences.interests" type="checkbox" value="Quidditch"/>
       <input type="hidden" value="1" name="_preferences.interests"/>
       Herbology: <input name="preferences.interests" type="checkbox" value="Herbology"/>
       <input type="hidden" value="1" name="_preferences.interests"/>
       Defence Against the Dark Arts: <input name="preferences.interests" type="checkbox"
          value="Defence Against the Dark Arts"/>
       <input type="hidden" value="1" name="_preferences.interests"/>
    </td>
 </tr>
```

你可能不希望看到的是额外的隐藏字段 在每个复选框。 当一个复选框在一个HTML页面 不 检查,它的值将不会被发送到 服务器作为HTTP请求的一部分参数一旦表单 提交,所以我们需要一个解决了这个怪癖在HTML中为了 弹簧形式数据绑定工作。 这个 复选框 标签 遵循现有的春季大会包括一个隐藏的参数 前缀为下划线(“_”)对于每个复选框。 通过这样做,你 实际上是告诉春天吗 一个 复选框是可见的形式,我希望我的对象 该表单数据将被绑定到反映状态的 复选框不管什么 一个 。

这个 复选框 标签

这个标签呈现多个HTML标记 “输入” 与类型 “复选框” 。

建筑的例子从以前的 复选框 标记部分。 有时候你不喜欢 要列出所有可能的爱好在JSP页面。 你会 而在运行时提供一个列表的选择和传递 在标记中。 这是目的 复选框 标签。 你传入一个 数组 ,一个 列表 或 地图 包含可用的选项 “项” 属性。 通常绑定属性是一个集合所以它 可以容纳多个值由用户选择。 下面是一个例子 JSP使用这个标签:

```
<form:form>
    <table>
       <tr>
          <td>Interests:</td>
          <td>
             <%-- Property is of an array or of type java.util.Collection --%>
             <form:checkboxes path="preferences.interests" items="${interestList}"/>
          </td>
       </tr>
    </table>
 </form:form>
```

这个例子假设” interestList” 是一个 列表 可以作为一个模型属性包含 字符串的值来进行选择的。 如果使用一个 地图,地图输入键将被用作值和地图入口的 值将被用作标签显示。 你也可以使用 自定义对象,您可以提供属性名称的值 使用 “itemValue” 和标签使用” itemLabel” 。

这个 radiobutton 标签

这个标签呈现HTML的输入 “标签” 广播” 型。

一个典型的使用模式将涉及多个标签实例 绑定到同一财产但是具有不同的值。

```
<tr>
    <td>Sex:</td>
    <td>Male: <form:radiobutton path="sex" value="M"/> <br/>
       Female: <form:radiobutton path="sex" value="F"/> </td>
 </tr>
```

这个 radiobuttons 标签

这个标签呈现多个HTML标记 “输入” 与类型 “广播” 。

就像 复选框 标签上面,你 可能想通过在可用的选项是一个运行时变量。 对于 您将使用这种用法 radiobuttons 标签。 你传入一个 数组 ,一个 列表 或 地图 包含 可用的选项在 “物品” 属性。 在这种情况下你 使用一个地图,地图输入键将被用作值和地图

输入的值将被用作标签显示。 你也可以 使用一个自定义的对象,你可以提供财产的名称 值使用“itemValue”和标签使用”itemLabel”。

```
<tr>
    <td>Sex:</td>
    <td><form:radiobuttons path="sex" items="${sexOptions}"/></td>
 </tr>
```

这个 密码 标签

这个标签呈现HTML的输入的标记类型“密码”使用 绑定值。

```
<tr>
    <td>Password:</td>
    <td>
       <form:password path="password" />
    </td>
 </tr>
```

请注意,默认情况下,密码值 不 显示。 如果你想要的密码值 示,然后将值设置的 ” showPassword” 属性为true,就像这样。

```
<tr>
    <td>Password:</td>
    <td>
       <form:password path="password" value="^76525bvHGq" showPassword="true" />
    </td>
 </tr>
```

这个 选择 标签

这个标签呈现HTML”选择“元素。 它支持数据 绑定到选择的选择以及使用嵌套 选项 和 选项 标签。

让我们假设一个 用户 列出了 技能。

```
<tr>
    <td>Skills:</td>
    <td><form:select path="skills" items="${skills}"/></td>
 </tr>
```

如果 用户的 技能在草药学, HTML源的“技巧”一行将看起来像:

```
<tr>
    <td>Skills:</td>
    <td><select name="skills" multiple="true">
       <option value="Potions">Potions</option>
       <option value="Herbology" selected="selected">Herbology</option>
       <option value="Quidditch">Quidditch</option></select>
    </td>
 </tr>
```

这个 选项 标签

这个标签呈现HTML '选项'。 它集“选择” 适当的基于绑定值。

```
<tr>
    <td>House:</td>
    <td>
       <form:select path="house">
          <form:option value="Gryffindor"/>
          <form:option value="Hufflepuff"/>
          <form:option value="Ravenclaw"/>
          <form:option value="Slytherin"/>
       </form:select>
    </td>
 </tr>
```

如果 用户的 房子是在格兰芬多, HTML源的“房子”的争吵会看起来像:

```
<tr>
    <td>House:</td>
    <td>
       <select name="house">
          <option value="Gryffindor" selected="selected">Gryffindor</option>
          <option value="Hufflepuff">Hufflepuff</option>
          <option value="Ravenclaw">Ravenclaw</option>
          <option value="Slytherin">Slytherin</option>
       </select>
```

```
    </td>
  </tr>
```

这个 选项 标签

这个标签呈现的HTML列表”选项”标签。 它集 “选定的属性作为适当的基于绑定值。

```
<tr>
    <td>Country:</td>
    <td>
      <form:select path="country">
        <form:option value="-" label="--Please Select"/>
        <form:options items="${countryList}" itemValue="code" itemLabel="name"/>
      </form:select>
    </td>
  </tr>
```

如果 用户 住在英国,HTML 源的“国家”一行将看起来像:

```
<tr>
    <td>Country:</td>
    <td>
      <select name="country">
        <option value="-">--Please Select</option>
        <option value="AT">Austria</option>
        <option value="UK" selected="selected">United Kingdom</option>
        <option value="US">United States</option>
      </select>
    </td>
  </tr>
```

如示例所示,合并后的用途的一个 选项 标记 选项 标签 产生相同的标准的HTML,但允许您显式地指定 一个值在JSP,是只显示(属于它等) 默认字符串的例子:“——请选择”。

这个 物品 属性通常密集 与一个收集体或者数组对象的项目。 itemValue 和 itemLabel 简单 引用bean属性的条目对象,如果指定; 否则,该项目将stringified对象本身。 或者,你可以指定一个 地图 的项目,在 这种情况下,映射键是解释为选项值和地图吗值对应选项标签。 如果 itemValue 和/或 itemLabel 碰巧被指定为好, 项目价值属性将应用于地图键和物品标签 属性将应用于地图的价值。

这个 Textarea 标签

这个标签呈现HTML的文本区”。

```
<tr>
    <td>Notes:</td>
    <td><form:textarea path="notes" rows="3" cols="20" /></td>
    <td><form:errors path="notes" /></td>
  </tr>
```

这个 隐藏 标签

这个标签呈现HTML的输入的标记类型“隐藏”的使用 绑定值。 提交一个解开隐藏值,使用HTML 输入 标记隐含型。

```
<form:hidden path="house" />
```

如果我们选择提交“房子”值作为一个隐藏的人, HTML会看起来像:

```
<input name="house" type="hidden" value="Gryffindor"/>
```

这个 错误 标签

这个标签显示字段错误在HTML的跨度”标签。 它提供了 访问错误出现在您的控制器或那些 由任何验证器与控制器关联。

让我们假设我们想要显示所有的错误消息 FirstName 和 lastName 字段 一旦我们提交表单。 我们有一个验证器的实例 用户类称为 UserValidator 。

```
public class UserValidator implements Validator {

    public boolean supports(Class candidate) {
        return User.class.isAssignableFrom(candidate);
    }
```

```
    public void validate(Object obj, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName", "required", "Field is required.");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "lastName", "required", "Field is required.");
    }
}
```

这个 形式jsp 将看起来像:

```
<form:form>
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName" /></td>
            <%-- Show errors for firstName field --%>
            <td><form:errors path="firstName" /></td>
        </tr>

        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName" /></td>
            <%-- Show errors for lastName field --%>
            <td><form:errors path="lastName"  /></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
</form:form>
```

如果我们提交一个表单与空值 FirstName 和 lastName 字段, 这就是HTML将会看起来像:

```
<form method="POST">
    <table>
        <tr>
            <td>First Name:</td>
            <td><input name="firstName" type="text" value=""/></td>
            <%-- Associated errors to firstName field displayed --%>
            <td><span name="firstName.errors">Field is required.</span></td>
        </tr>

        <tr>
            <td>Last Name:</td>
            <td><input name="lastName" type="text" value=""/></td>
            <%-- Associated errors to lastName field displayed --%>
            <td><span name="lastName.errors">Field is required.</span></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
</form>
```

如果我们想要显示的整个列表错误对于一个给定的 页面吗? 下面的例子显示了 错误 标签 它还支持一些基本功能。

- 路径= “*” -显示所有错误
- 路径= “姓” -显示所有错误 相关 lastName 领域
- 如果 路径 省略了——对象只显示错误吗

下面的示例将显示一个错误清单的顶部 页面,其次是专业的实战错误旁边的字段:

```
<form:form>
    <form:errors path="*" cssClass="errorBox" />
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName" /></td>
            <td><form:errors path="firstName" /></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName" /></td>
            <td><form:errors path="lastName"  /></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
```

```
    </table>
  </form:form>
```

HTML会看起来像:

```
<form method="POST">
    <span name="*.errors" class="errorBox">Field is required.<br/>Field is required.</span>
    <table>
      <tr>
        <td>First Name:</td>
        <td><input name="firstName" type="text" value=""/></td>
        <td><span name="firstName.errors">Field is required.</span></td>
      </tr>

      <tr>
        <td>Last Name:</td>
        <td><input name="lastName" type="text" value=""/></td>
        <td><span name="lastName.errors">Field is required.</span></td>
      </tr>
      <tr>
        <td colspan="3">
          <input type="submit" value="Save Changes" />
        </td>
      </tr>
  </form>
```

HTTP方法转换

剩下的一个重要原则是使用了统一的接口。 这意味着所有资源(url)可以操作使用相同的 四个HTTP方法:获取、放置、发布和删除。 对于每个方法, HTTP规范定义了确切的语义。 例如,GET 应该是一个安全的操作,这意味着是没有边 效果,和一把或删除应该是幂等的,也就是说你 可以重复这些操作一遍又一遍,但最终的结果吗 应该是相同的。 虽然HTTP定义了这四个方法,HTML只有 支持两种:GET和POST。 幸运的是,有两个可能 解决方案:您可以使用JavaScript来做你的把或删除, 或者只是做一篇与“真实”的方法作为一个额外的参数 (建模为一个隐藏在一个HTML表单输入字段)。 这个小窍门 就是春天的HiddenHttpMethodFilter 确实。 这个过滤器是一个普通的Servlet过滤器,因此它可以用于 结合任何web框架(不仅仅是Spring MVC)。 只需添加 这个过滤器到您的网页。 xml和一篇文章和一个隐藏的方法 参数将被转换成相应的HTTP方法 请求。

支持HTTP方法转换的Spring MVC形式标记 更新,支持设置HTTP方法。 例如,下面的 片段取自更新的宠物诊所样本

```
<form:form method="delete">
    <p class="submit"><input type="submit" value="Delete Pet"/></p>
</form:form>
```

这实际上会执行一个HTTP POST,“真正的”删除 法隐藏在一个请求参数,是捡到 HiddenHttpMethodFilter ,在web . xml中定义:

```
<filter>
   <filter-name>httpMethodFilter</filter-name>
   <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>
</filter>

<filter-mapping>
   <filter-name>httpMethodFilter</filter-name>
   <servlet-name>petclinic</servlet-name>
</filter-mapping>
```

相应的controller方法 如下所示:

```
@RequestMapping(method = RequestMethod.DELETE)
public String deletePet(@PathVariable int ownerId, @PathVariable int petId) {
  this.clinic.deletePet(petId);
  return "redirect:/owners/" + ownerId;
}
```

HTML5标签

从弹簧3,弹簧形式标记库允许进入 动态属性,这意味着您可以输入任何HTML5的特定属性。

在Spring 3.1中,表单输入标签支持进入一个type属性 除了“文本”。 这是为了让呈现新的HTML5具体 输入类型如“电子邮件”、“日期”、“范围”,和其他人。 注意, 输入类型=“文本”是不需要因为“文本”是默认的类型。

18.3一个瓷砖

它是可能的集成瓷砖——就像任何其他视图 技术——在使用Spring的web应用程序。 下面描述了在 一个宽路如何做到这一点。

注意: 本节的重点是Spring的支持 对瓷砖2(单机版的瓷砖,需要Java 5 +) org.springframework.web.servlet.view.tiles2 包作为 以及瓷砖3 org.springframework.web.servlet.view.tiles3 包。 春天也继续支持瓷砖1。 x(也称为。 “Struts Tiles”,如 附带Struts 1.1 +,兼容Java 1.4)在原始的 org.springframework.web.servlet.view.tiles 包。

18.3.1A依赖性

能够使用瓷砖,你必须有几个额外的 包括在项目依赖关系。 以下是列表的 你需要的依赖关系。

- 瓷砖或更高版本2.1.2
- Commons BeanUtils
- Commons蒸煮器
- 通用日志

18.3.2A如何集成瓷砖

能够使用瓷砖,你必须使用文件配置它 包含定义(基本信息定义和其他 瓷砖的概念,请看看 http://tiles.apache.org)。 在春季这样做是使用 TilesConfigurer 。 看看下面的吗 块示例ApplicationContext配置:

```
<bean id="tilesConfigurer" class="org.springframework.web.servlet.view.tiles2.TilesConfigurer">
 <property name="definitions">
  <list>
   <value>/WEB-INF/defs/general.xml</value>
   <value>/WEB-INF/defs/widgets.xml</value>
   <value>/WEB-INF/defs/administrator.xml</value>
   <value>/WEB-INF/defs/customer.xml</value>
   <value>/WEB-INF/defs/templates.xml</value>
  </list>
 </property>
</bean>
```

正如你可以看到,有五个文件,包含定义 都是位于 “- inf / defs” 目录。 在初始化 的 WebApplicationContext ,文件 将加载并初始化定义工厂将。 在 已经完成,瓷砖包括在定义文件可以 作为视图在Spring web应用程序。 能够使用 你必须要有一个观点 ViewResolver 就像任何其他视图技术用于弹簧。 下面你可以 发现两种可能性, UrlBasedViewResolver 和 ResourceBundleViewResolver 。

UrlBasedViewResolver

这个 UrlBasedViewResolver 实例化 鉴于 viewClass 它为每个视图 解决。

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.UrlBasedViewResolver">
 <property name="viewClass" value="org.springframework.web.servlet.view.tiles2.TilesView"/>
</bean>
```

ResourceBundleViewResolver

这个 ResourceBundleViewResolver 必须 提供了一个属性文件包含的viewnames和viewclasses 解析器可以使用:

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
 <property name="basename" value="views"/>
</bean>
```

```
...
welcomeView.(class)=org.springframework.web.servlet.view.tiles2.TilesView
welcomeView.url=welcome (this is the name of a Tiles definition)

vetsView.(class)=org.springframework.web.servlet.view.tiles2.TilesView
vetsView.url=vetsView (again, this is the name of a Tiles definition)

findOwnersForm.(class)=org.springframework.web.servlet.view.JstlView
findOwnersForm.url=/WEB-INF/jsp/findOwners.jsp
...
```

正如您可以看到的,当使用 ResourceBundleViewResolver ,你可以很容易地混合 不同的视图技术。

注意, TilesView 类瓷砖2 支持JSTL(JSP标准标记库)的盒子,而 有一个单独的 TilesJstlView 子类在 瓷砖1。 x的支持。

SimpleSpringPreparerFactory 和 SpringBeanPreparerFactory

作为一个高级功能,春天也支持两种特殊瓦片2 PreparerFactory 实现。 检查 出瓷砖的文档了解如何使用 ViewPreparer 引用你的瓷砖 定义文件。

指定 SimpleSpringPreparerFactory 到 自动装配ViewPreparer实例基于指定的填表人类, Spring的容器应用回调以及应用配置 春天BeanPostProcessors。 如果春天的上下文宽注释配置 已被激活,注释在ViewPreparer类会吗 自动检测和应用。 注意,这个预计填表人 类 在瓷砖定义文件,就像 默认 PreparerFactory 确实。

指定 SpringBeanPreparerFactory 到 操作指定填表人 名称 而不是 类,获得相应的Spring bean DispatcherServlet的应用程序上下文。 完整的bean创建 过程将在Spring应用程序上下文的控制在 这种情况下,允许使用显式的依赖注入 配置,作用域豆类等。请注意,您需要定义一个 Spring bean定义(使用每填表人的名字在你的瓷砖 定义)。

```
<bean id="tilesConfigurer" class="org.springframework.web.servlet.view.tiles2.TilesConfigurer">
 <property name="definitions">
  <list>
   <value>/WEB-INF/defs/general.xml</value>
   <value>/WEB-INF/defs/widgets.xml</value>
   <value>/WEB-INF/defs/administrator.xml</value>
   <value>/WEB-INF/defs/customer.xml</value>
   <value>/WEB-INF/defs/templates.xml</value>
  </list>
 </property>

 <!-- resolving preparer names as Spring bean definition names -->
 <property name="preparerFactoryClass"
    value="org.springframework.web.servlet.view.tiles2.SpringBeanPreparerFactory"/>

</bean>
```

18.4一个速度& FreeMarker

速度 和 freemarker 是两个模板 语言,可以用作在Spring MVC的视图技术 应用程序。 语言非常相似,为类似的需求和 所以在这部分放在一起考虑。 为语义和句法 这两种语言之间的差异,请参阅 freemarker web站点。

18.4.1A依赖性

您的web应用程序将需要包括 速度- 1. - x x jar 或 freemarker - 2. - x jar 为了处理 速度或FreeMarker分别和 commons集合jar 需要 速度。 他们通常是包含在 - inf / lib 文件夹,他们是有保证的 发现了一个Java EE服务器和添加到您的应用程序的类路径。 当然,假设你已经有了 spring-webmvc.jar 在你的 “- inf / lib” 目录太! 如果你使用 春天的“dateToolAttribute”或“numberToolAttribute '在你的速度 视图,您还将需要包括 速度-工具-通用- 1. - x jar

18.4.2A上下文配置

一个合适的配置是通过添加相关的初始化 configurer bean定义你 ‘* servlet xml” 如下所示:

```
<!--
 This bean sets up the Velocity environment for us based on a root path for templates.
 Optionally, a properties file can be specified for more control over the Velocity
 environment, but the defaults are pretty sane for file based template loading.
-->
<bean id="velocityConfig" class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">
 <property name="resourceLoaderPath" value="/WEB-INF/velocity/"/>
</bean>

<!--

 View resolvers can also be configured with ResourceBundles or XML files. If you need
 different view resolving based on Locale, you have to use the resource bundle resolver.

-->
<bean id="viewResolver" class="org.springframework.web.servlet.view.velocity.VelocityViewResolver">
 <property name="cache" value="true"/>
 <property name="prefix" value=""/>
 <property name="suffix" value=".vm"/>
</bean>
```

```
<!-- freemarker config -->
<bean id="freemarkerConfig" class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">
 <property name="templateLoaderPath" value="/WEB-INF/freemarker/"/>
</bean>

<!--

 View resolvers can also be configured with ResourceBundles or XML files. If you need
 different view resolving based on Locale, you have to use the resource bundle resolver.
```

```
-->
<bean id="viewResolver" class="org.springframework.web.servlet.view.freemarker.FreeMarkerViewResolver">
  <property name="cache" value="true"/>
  <property name="prefix" value=""/>
  <property name="suffix" value=".ftl"/>
</bean>
```

注意

对于非web应用程序添加一个 VelocityConfigurationFactoryBean 或 FreeMarkerConfigurationFactoryBean 你 应用程序上下文定义文件。

18.4.3A创建模板

你的模板需要存储在指定的目录 * Configurer 豆如上所示。 这个文档不 详细介绍创建模板的两种语言——请参阅 他们的相关网站信息。 如果你使用视图解析器 高亮显示,然后逻辑视图名称与模板文件 名字在类似的风格 InternalResourceViewResolver 对于JSP的。 所以如果 你的控制器返回一个包含视图名称ModelAndView对象的 “欢迎” 然后解析器将寻找 / - inf / freemarker /欢迎ftl 或 / - inf /速度/欢迎vm 模板作为 适当的。

18.4.4A高级配置

突出的基本配置将适合 大多数应用程序的需求,但是额外的配置选项 可用于在不寻常的或先进的需求决定。

速度属性

这个文件是完全可选的,但如果指定,包含 值传递到速度运行时,以便配置 速度本身。 只需要先进的配置,如果你 需要这个文件,指定它的地理位置 VelocityConfigurer 上面的bean定义。

```
<bean id="velocityConfig" class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">
  <property name="configLocation" value="/WEB-INF/velocity.properties"/>
</bean>
```

或者,您可以指定速度属性直接在 该bean定义为速度配置bean取代了 “configLocation” 属性与下面的内联属性。

```
<bean id="velocityConfig" class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">
  <property name="velocityProperties">
    <props>
      <prop key="resource.loader">file</prop>
      <prop key="file.resource.loader.class">
       org.apache.velocity.runtime.resource.loader.FileResourceLoader
      </prop>
      <prop key="file.resource.loader.path">${webapp.root}/WEB-INF/velocity</prop>
      <prop key="file.resource.loader.cache">false</prop>
    </props>
  </property>
</bean>
```

参考 API 文档 对于Spring配置的速度,或 速度文档的例子和定义 '速度属性” 文件本身。

freemarker

FreeMarker “设置” 和 “SharedVariables” 可以被传递 直接FreeMarker 配置 对象 春季管理通过设置适当的bean属性 FreeMarkerConfigurer bean。 这个 freemarkerSettings 财产需要 java util属性 对象和 freemarkerVariables 财产需要 java util地图 。

```
<bean id="freemarkerConfig" class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">
  <property name="templateLoaderPath" value="/WEB-INF/freemarker/"/>
  <property name="freemarkerVariables">
    <map>
      <entry key="xml_escape" value-ref="fmXmlEscape"/>
    </map>
  </property>
</bean>

<bean id="fmXmlEscape" class="freemarker.template.utility.XmlEscape"/>
```

看到FreeMarker文档设置和细节 变量作为他们申请 配置 对象。

18.4.5A绑定支持和形式处理

Spring提供了一个用于JSP标记库的包含 (还有其他一些东西) <春天:bind / > 标签。 这个标签主要是使形式显示值形式的支

持 对象和显示结果的失败验证从 验证器 在web或业务层。 从版本 1.1,春天现在支持相同的功能在两个速度 和FreeMarker,额外的便利宏生成表单 输入元素本身。

bind宏

一组标准的宏保存在 spring-webmvc.jar 文件为两种语言,因此他们 总是可以一个适当配置的应用程序。

一些宏定义在Spring库 考虑内部(私有)但没有这样的范围存在于宏观 定义使所有宏调用代码和用户可见 模板。 以下部分只集中在你的宏 需要直接调用在你的模板。 如果你想 直接查看宏代码,文件被称为弹簧。 vm / 春天。 ftl和在包 org.springframework.web.servlet.view.velocity 或 org.springframework.web.servlet.view.freemarker 分别。

简单绑定

在你的html表单(vm / ftl模板),作为 “formView” 弹簧形式控制器,你可以使用代码类似 以下绑定到字段值,并显示错误消息 每个输入字段以类似的方式对JSP等效。 注意, 命令对象的名称是 “命令” 默认情况下,但可以 覆盖在你的MVC配置通过设置 “commandName的bean 房地产在窗体上控制器。 示例代码如下所示的 personFormV 和 personFormF 前面配置的观点;

```
<!-- velocity macros are automatically available -->
<html>
...
<form action="" method="POST">
  Name:
  #springBind( "command.name" )
  <input type="text"
    name="${status.expression}"
    value="$!status.value" /><br>
  #foreach($error in $status.errorMessages) <b>$error</b> <br> #end
  <br>
  ...
  <input type="submit" value="submit"/>
</form>
...
</html>
```

```
<!-- freemarker macros have to be imported into a namespace.  We strongly
recommend sticking to 'spring' -->
<#import "/spring.ftl" as spring />
<html>
...
<form action="" method="POST">
  Name:
  <@spring.bind "command.name" />
  <input type="text"
    name="${spring.status.expression}"
    value="${spring.status.value?default("")}" /><br>
  <#list spring.status.errorMessages as error> <b>${error}</b> <br> </#list>
  <br>
  ...
  <input type="submit" value="submit"/>
</form>
...
</html>
```

springBind / < @spring.bind > 需要一个 “路径” 的说法 它包括你的命令对象的名字(这将是吗 “命令”,除非你改变它在你的FormController属性) 其次是一个时期和字段的名字在命令对象 您希望绑定到。 也可以使用嵌套的字段如 “命令地址街” 。 这个 绑定 宏观假设 默认指定的HTML转义行为ServletContext 参数 defaultHtmlEscape 在web . xml

可选的形式的宏调用 # springBindEscaped / < @spring.bindEscaped > 需要第二个参数 和显式地指定是否应该使用HTML转义在 状态错误消息或价值观。 设置为真或假的要求。 额外的形式处理宏简化使用HTML转义和 这些宏应该尽可能使用。 他们解释 下一节。

表单输入生成宏

额外的便利宏两种语言简化两个 绑定和表单生成(包括验证错误显示)。 它 是没有必要使用这些宏生成表单输入字段, 而且他们可以混合和匹配与简单的HTML或电话直接去 春天绑定宏强调之前。

下面的表显示了VTL的可用宏和FTL 定义和参数列表,每个需要。

18.1为多。 一个表的宏定义

宏	VTL定义	FTL定义
消息 (输出一个 字符串从一个资源包基于代码 参数)	# springMessage(码)	< @spring.message 代码/ >
messageText (输出一个 字符串从一个资源包基于编码参数, 回值下降的默认参数)	# springMessageText($代码 $文本)	< @spring。 messageText代码, 文本/ >
url (前缀相对 URL与应用程序的上下文根)	# springUrl($ relativeUrl)	< @spring.url relativeUrl / >
formInput (标准 输入字段用于收集用户输入)	# springFormInput($ path $属性)	< @spring。 formInput路径、属性 fieldType / >
formHiddenInput * (隐藏的输入字段提交非用户输入)	# springFormHiddenInput($ path $属性)	< @spring。 formHiddenInput路径, 属性/ >
formPasswordInput * (标准输入字段用于收集密码。 请注意,没有 值会被填充在这种类型的字段)	# springFormPasswordInput($ path $属性)	< @spring。 formPasswordInput路径, 属性/ >
formTextarea (大 文本字段用于收集长,自由的文本输入)	# springFormTextarea($ path $属性)	< @spring。 formTextarea路径, 属性/ >
formSingleSelect (下降 下拉框选项允许一个单一的要求值 选定)	# springFormSingleSelect($ path选项美元 $属性)	< @spring。 formSingleSelect路径,选择, 属性/ >
formMultiSelect (列表框的选项允许用户选择0或多个 值)	# springFormMultiSelect($ path选项美元 $属性)	< @spring。 formMultiSelect路径,选择, 属性/ >
formRadioButtons (组单选按钮允许一个单一的选择是 从可用选项)	# springFormRadioButtons($ path选项美元 $ $属性)分隔符	< @spring。 formRadioButtons路径,选择分离器、属性/ >
formCheckboxes (一组 的复选框允许0或多个值选定)	# springFormCheckboxes($ path选项美元 $ $属性)分隔符	< @spring。 formCheckboxes路径,选择, 分离器、属性/ >
formCheckbox (单个 复选框)	# springFormCheckbox($ path $属性)	< @spring。 formCheckbox路径, 属性/ >
showErrors (简化 显示验证错误的绑定字段)	# springShowErrors($分离器 classOrStyle美元)	< @spring。 showErrors分离器, classOrStyle / >

*在FTL(FreeMarker),这两个宏并不实际 需要您可以使用正常 formInput 宏, 指定的 隐藏 ” 或 ” 密码 ” 的值 fieldType 参数。

参数上面的任意宏有一致的 含义:

- 路径:字段名称绑定到(ie “命令名称”)
- 选项:地图的所有可用的值,可以 选定的输入字段。 地图的关键代表 的值将从形式回传,并绑定到 命令对象。 地图对象存储对密钥标签 显示在表单上的用户和可能是不同的 相应的值的形式回传。 通常这样一个地图 提供参考数据的控制器。 任何地图 实现可以根据需要使用行为。 对于 严格分类地图,一个 SortedMap 如 TreeMap 用一个合适的比较器可以使用 和任意地图,应该返回值在插入 订单,使用 LinkedHashMap 或 LinkedMap 从共享集合。
- 分隔符:在多个选项可用作为谨慎的 元素(单选按钮或复选框),序列的字符 用于分隔列表中的每一个人(即 “< br >”)。
- 属性:一个附加字符串的任意标签或文本 是包含在HTML标签本身。 这个字符串是呼应 字面上的宏。 例如,在一场你可能。文本区 供应属性为 “行= " 5 " cols = " 60 " '或者你可以通过风格 信息,如 “风格= "边境:1 px固体银” 。
- classOrStyle:showErrors宏,CSS的名称 类跨度标签包装每个错误将使用。 如果没有 信息提供(或值是空的),那么这个错误将包裹在< b > < / b >标签。

宏的例子下面列出了一些在FTL和一些 在可变阈值逻辑。 在使用这两种语言之间存在的差异,他们呢 在笔记中解释。

输入字段

```
<!-- the Name field example from above using form macros in VTL -->
...
  Name:
  #springFormInput("command.name" "")<br>
  #springShowErrors("<br>" "")<br>
```

formInput的宏观需要的路径参数(命令名称) 和一个额外的属性参数是空的例子 以上。 宏,连同所有其他表单生成宏, 执行隐式弹簧约束的路径参数。 绑定 有效期至一个新的绑定发生所以showErrors宏 不需要传递的路径参数——它只是操作。再 在哪个领域创建一个绑定是最后。

showErrors的宏观需要分离器参数(字符,用于单独的多个错误在一个给定的 场),还接受第二个参数,这个时候一个类名 或样式属性。 注意,FreeMarker能够指定默认 值的属性参数,与速度,和这两个 宏调用以上可以表达如下在FTL:

```
<@spring.formInput "command.name"/>
<@spring.showErrors "<br>"/>
```

输出如下所示的表单片段生成的名字 场,并显示验证错误后的形式 提交没有值的字段。 验证是通过 春天的验证框架。

生成的HTML看起来像这样:

```
Name:
 <input type="text" name="name" value=""
>
<br>
 <b>required</b>
<br>
<br>
```

formTextarea宏的作品formInput一样 宏和接受相同的参数列表。 通常,第二 参数(属性)将用于通过样式信息或 行和cols属性的文本区。

选择字段

四个选择域宏可以用来生成常见的UI 价值选择的输入在你的HTML表单。

- formSingleSelect
- formMultiSelect
- formRadioButtons
- formCheckboxes

每四个宏接受一个地图的选项包含 表单字段的值,和标签,对应 值。 和标签的值可以是相同的。

单选按钮的一个例子在FTL低于。 表单支持 对象指定一个默认值“伦敦”这个字段,所以 没有验证是必要的。 当形式呈现,整个城市的列表选择提供参考数据 模型根据名称“cityMap’ 。

```
...
 Town:
 <@spring.formRadioButtons "command.address.town", cityMap, "" /><br><br>
```

这使得一行单选按钮,一个用于每个值 cityMap 使用分隔符 “。 没有额外的 属性提供的(最后一个参数的宏 失踪)。 cityMap使用相同的字符串的每个键-值对 在地图上。 地图的钥匙是什么形式的实际提交作为 发布请求参数,映射值标签,用户 看到。 在上面的例子中,如果列出三个众所周知的城市 和一个默认值的形式支持对象,HTML会 是

```
Town:
<input type="radio" name="address.town" value="London"

>
London
<input type="radio" name="address.town" value="Paris"
 checked="checked"
>
Paris
<input type="radio" name="address.town" value="New York"

>
New York
```

如果应用程序希望处理城市内部编码 例如,地图的代码将创建合适的钥匙 像下面的例子。

```
protected Map referenceData(HttpServletRequest request) throws Exception {
  Map cityMap = new LinkedHashMap();
```

```
  cityMap.put("LDN", "London");
  cityMap.put("PRS", "Paris");
  cityMap.put("NYC", "New York");

  Map m = new HashMap();
  m.put("cityMap", cityMap);
  return m;
}
```

现在的代码会产生输出,收音机值 相关的代码,但用户仍然看到了更多的用户友好 城市的名字。

```
Town:
<input type="radio" name="address.town" value="LDN"

>
London
<input type="radio" name="address.town" value="PRS"
 checked="checked"
>
Paris
<input type="radio" name="address.town" value="NYC"

>
New York
```

HTML转义和XHTML合规

默认使用表单宏将导致HTML标签之上 这是HTML 4.01兼容,使用默认值为HTML 定义在web逃离。 使用Spring的xml绑定支持。 在 为了使标签XHTML兼容或覆盖默认的HTML 逃值,您可以指定两个变量在你的模板(或在 你的模型,他们可以看见你的模板)。 这个 利用指定他们的模板是,他们可以 更改为不同的值在稍后的模板处理 为不同领域提供不同的行为在你的表格。

切换到XHTML合规为你的标签,指定一个值 “真正的” 为一个模型/上下文变量命名xhtmlCompliant:

```
## for Velocity..
#set($springXhtmlCompliant = true)

<#-- for FreeMarker -->
<#assign xhtmlCompliant = true in spring>
```

任何标记生成的春宏现在将XHTML 兼容处理后这个指令。

以类似的方式,可以指定HTML转义每 字段:

```
<#-- until this point, default HTML escaping is used -->

<#assign htmlEscape = true in spring>
<#-- next field will use HTML escaping -->
<@spring.formInput "command.name" />

<#assign htmlEscape = false in spring>
<#-- all future fields will be bound with HTML escaping off -->
```

18.5一个XSLT

XSLT是一种转换语言,XML和流行作为一个视图 在web应用程序的技术。 XSLT可以是一个不错的选择作为一个视图 技术如果应用程序自然地处理XML,或者如果你的模型 可以很容易地转换成XML。 下面的小节展示如何制作 一个XML文档作为模型数据并把它与XSLT转换在一个 Spring Web MVC应用程序。

18.5.1A我第一句话

这个例子是一个简单的Spring应用程序,创建一个列表 的话在 控制器 并将它们添加 模型图。 返回地图连同我们的视图名称 XSLT视图。 看到 SectionA 17.3,一个Controllersa实施 对于细节的Spring Web MVC的 控制器 接口。 XSLT视图将 把单词列表变成一个简单的XML文档准备 转换。

Bean定义

配置是标准的一个简单的Spring应用程序。 这个 dispatcher servlet配置文件包含一个引用 ViewResolver 、URL映射和单一控制器豆.....

```
<bean id="homeController"class="xslt.HomeController"/>
```

... 我们的单词生成逻辑封装。

标准MVC控制器代码

控制器逻辑被封装在的一个子类 基类AbstractController 的处理程序方法 被定义的就像这样...

```
protected ModelAndView handleRequestInternal(
    HttpServletRequest request,
    HttpServletResponse response) throws Exception {

    Map map = new HashMap();
    List wordList = new ArrayList();

    wordList.add("hello");
    wordList.add("world");

    map.put("wordList", wordList);

    return new ModelAndView("home", map);
}
```

到目前为止,我们什么也没做,这是XSLT具体。 模型数据 已经建立在同样的方式你会为任何其他Spring MVC吗 应用程序。 根据配置的应用程序现在, 这单词列表可以显示JSP / JSTL让他们补充说 作为请求属性,或者他们可能是由速度增加 对象的VelocityContext 。 为了 有XSLT渲染他们,他们当然必须被转换成一个XML 文件不知何故。 有可用的软件包,将 自动的domify的对象图,但在春天,你有 完整的灵活性来创建DOM从你的模型以任何方式你 选择。 这可以防止转换XML玩太大 参与你的模型数据结构,这是一个危险当使用 工具来管理domification过程。

模型数据转换为XML

为了创建一个DOM文档从我们的单词列表或任何 其他的模型数据,我们必须继承(提供) org.springframework.web.servlet.view.xslt.AbstractXsltView 类。 这样做,我们也必须实现抽象的典型 方法 createXsltSource(. .) 法。 第一 参数传递给这个方法是我们的模型图。 下面是完整的 清单的 主页 类在我们的琐碎 词的应用程序:

```
package xslt;

// imports omitted for brevity

public class HomePage extends AbstractXsltView {

    protected Source createXsltSource(Map model, String rootName, HttpServletRequest
        request, HttpServletResponse response) throws Exception {

        Document document = DocumentBuilderFactory.newInstance().newDocumentBuilder().newDocument();
        Element root = document.createElement(rootName);

        List words = (List) model.get("wordList");
        for (Iterator it = words.iterator(); it.hasNext();) {
            String nextWord = (String) it.next();
            Element wordNode = document.createElement("word");
            Text textNode = document.createTextNode(nextWord);
            wordNode.appendChild(textNode);
            root.appendChild(wordNode);
        }
        return new DOMSource(root);
    }

}
```

一系列的参数名称/值对可以被定义 通过你的子类将被添加到变换对象。 这个 参数名称必须匹配你的XSLT模板中定义的 宣布与 < xsl:param name = " myParam" defaultValue > < / xsl:param > 。 指定 参数,覆盖 getParameters() 方法 AbstractXsltView 类和返回 一个 地图 名称/值对。 如果你 参数需要获得信息从当前请求,你 可以覆盖 getParameters(HttpServletRequest 请求) 方法相反。

定义视图属性

视图。 属性文件(或等价的xml定义如果 你使用的是一个基于XML视图解析器像我们那样的速度 上述的例子)看起来像这样的一个视图应用程序 "我的第一个词:

```
home.(class)=xslt.HomePage
home.stylesheetLocation=/WEB-INF/xsl/home.xslt
home.root=words
```

在这里,你可以看到视图是联系在一起的 主页 类只是书面的处理 模型domification在第一个属性 " 。(类)" 。 这个 " stylesheetLocation" 属性指向 XSLT文件将XML转换成HTML处理对我们 和最后的财产 "根" 名称, 将被用作XML文档的根。 这被传递到 主页 级以上在第二个参数 这个 createXsltSource(. .) 方法(年代)。

文档转换

最后,我们有XSLT代码用于转换上面的 文档。 如上图所示 ” 视图属性” 文件、样式表被称为 '家里xslt的 和它生活在war文件中 “- inf / xsl” 目录。

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" omit-xml-declaration="yes"/>

  <xsl:template match="/">
    <html>
      <head><title>Hello!</title></head>
      <body>
        <h1>My First Words</h1>
        <xsl:apply-templates/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="word">
    <xsl:value-of select="."/><br/>
  </xsl:template>

</xsl:stylesheet>
```

18.5.2A总结

一个概要文件的讨论和他们的位置在战争 文件显示在简化结构下面的战争。

```
ProjectRoot
 |
 +- WebContent
    |
    +- WEB-INF
       |
       +- classes
       |   |
       |   +- xslt
       |   |   |
       |   |   +- HomePageController.class
       |   |   +- HomePage.class
       |   |
       |   +- views.properties
       |
       +- lib
       |   |
       |   +- spring-*.jar
       |
       +- xsl
       |   |
       |   +- home.xslt
       |
       +- frontcontroller-servlet.xml
```

您还需要确保一个XML解析器和XSLT引擎 可以在类路径中。 JDK 1.4提供了他们在默认情况下, 大多数Java EE容器也会使它们可用默认情况下,但它的 一个可能的误差源的需要注意的。

18.6文档视图(PDF / Excel)

18.6.1A介绍

返回一个HTML页面并不是最佳的方式为用户 视图模型输出,春天使它简单生成PDF 文档或电子表格动态模型数据。 这个 文档视图和从服务器传输的 正确的内容类型(希望)启用客户端PC运行他们 电子表格或PDF查看器应用程序的响应。

为了使用Excel视图,您需要添加 “山芋” 图书馆 你的类路径,并生成PDF,iText库。

18.6.2A配置和设置

基于文档视图处理在一个几乎相同的时尚 XSLT的观点,以下部分建立在之前的一个 演示了相同的控制器用于XSLT例子 呈现相同的模型调用既是一个PDF文档和一个Excel 电子表格(也可以被查看和操纵在开放 办公室)。

文档视图定义

首先,让我们修改意见。 属性文件(或xml 当量)和添加一个简单的视图定义为两个文档类型。 整个文件现在看起来像这样与

XSLT视图显示 早:

```
home.(class)=xslt.HomePage
home.stylesheetLocation=/WEB-INF/xsl/home.xslt
home.root=words

xl.(class)=excel.HomePage

pdf.(class)=pdf.HomePage
```

如果你想从一个 模板表格或一个fillable PDF表单来添加您的模型数据,指定位置 随着“url”属性在视图定义中

控制器代码

我们将会使用的控制器代码仍然是完全相同的 XSLT例子早期其他比改变视图的名称使用。 当然,你可以聪明,有这种选择基于URL 参数或其他逻辑——证明春天真的很好 在解耦中的视图控制器!

子类化Excel的观点

到底是为XSLT示例,我们将子类合适 抽象类来实现自定义的行为产生 我们的输出文件。 Excel,这涉及到写作的一个子类 org.springframework.web.servlet.view.document.AbstractExcelView (Excel文件生成的POI)或 org.springframework.web.servlet.view.document.AbstractJExcelView (JExcelApi-generated Excel文件)和实施 buildExcelDocument() 法。

下面是完整的清单为我们的观点,即芋泥Excel 显示单词列表从模型映射在连续的行 一个新的电子表格的第一列:

```
package excel;

// imports omitted for brevity

public class HomePage extends AbstractExcelView {

    protected void buildExcelDocument(
        Map model,
        HSSFWorkbook wb,
        HttpServletRequest req,
        HttpServletResponse resp)
        throws Exception {

        HSSFSheet sheet;
        HSSFRow sheetRow;
        HSSFCell cell;

        // Go to the first sheet
        // getSheetAt: only if wb is created from an existing document
        // sheet = wb.getSheetAt(0);
        sheet = wb.createSheet("Spring");
        sheet.setDefaultColumnWidth((short) 12);

        // write a text at A1
        cell = getCell(sheet, 0, 0);
        setText(cell, "Spring-Excel test");

        List words = (List) model.get("wordList");
        for (int i=0; i < words.size(); i++) {
            cell = getCell(sheet, 2+i, 0);
            setText(cell, (String) words.get(i));

        }
    }
}
```

和下面的是一个视图生成相同的Excel文件,现在使用 JExcelApi:

```
package excel;

// imports omitted for brevity

public class HomePage extends AbstractJExcelView {

    protected void buildExcelDocument(Map model,
        WritableWorkbook wb,
        HttpServletRequest request,
        HttpServletResponse response)
    throws Exception {

        WritableSheet sheet = wb.createSheet("Spring", 0);

        sheet.addCell(new Label(0, 0, "Spring-Excel test"));
```

```
        List words = (List) model.get("wordList");
        for (int i = 0; i < words.size(); i++) {
            sheet.addCell(new Label(2+i, 0, (String) words.get(i)));
        }
    }
}
```

注意之间的差异。api 我们已经发现了 JExcelApi有点更直观,而且,JExcelApi已经 稍微更好的图像处理功能。 有记忆 问题与大型Excel文件在使用JExcelApi不过。

如果你现在修改控制器,这样它返回 XL 视图的名称(返回新 ModelAndView(“xl”,地图);),并再次运行应用程序, 你应该发现Excel电子表格创建并下载 当你请求自动同一页面之前。

子类化对于PDF的观点

PDF版本的单词列表更加简单。 这一次, 类扩展 org.springframework.web.servlet.view.document.AbstractPdfView 并实现了 buildPdfDocument() 方法 如下:

```
package pdf;

// imports omitted for brevity

public class PDFPage extends AbstractPdfView {

    protected void buildPdfDocument(
        Map model,
        Document doc,
        PdfWriter writer,
        HttpServletRequest req,
        HttpServletResponse resp)
        throws Exception {

        List words = (List) model.get("wordList");

        for (int i=0; i<words.size(); i++)
            doc.add( new Paragraph((String) words.get(i)));

    }
}
```

再一次,修改控制器返回 PDF 视图与 返回新 ModelAndView(“pdf”,地图); ,并重新加载URL在你 应用程序。 这个时间应该会出现一个PDF文件清单每个 这些词在模型图。

18.7一个JasperReports

JasperReports(http://jasperreports.sourceforge.net)是一个强大的 开源的报表引擎,支持创建报告的设计 使用一个容易理解XML文件格式。 JasperReports能够 渲染报道在四个不同的格式:CSV,Excel,HTML和 pdf。

18.7.1A依赖性

您的应用程序将需要包含的最新版本 JasperReports,写这篇文章的时候是0 6 1。 jasperreports 本身取决于以下项目:

- BeanShell
- Commons BeanUtils
- Commons集合
- Commons蒸煮器
- 通用日志
- iText
- 芋泥

JasperReports还需要JAXP兼容的XML解析器。

18.7.2A配置

配置在Spring容器JasperReports观点你 配置您需要定义一个 ViewResolver 地图视图的名称 适当的视图类根据您想要的格式你的报告 中呈现的。

配置 ViewResolver

通常,您将使用 ResourceBundleViewResolver 映射视图名称 视图类和文件在一个属性文件。

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
  <property name="basename" value="views"/>
</bean>
```

这里我们已经配置的一个实例 ResourceBundleViewResolver 类,它将看起来 为视图映射在资源包与基地的名字 视图 。 (此文件的内容描述 下一节)。

配置 视图 年代

Spring框架包含了五个不同的 视图 实现的话, 四相对应的四个输出格式支持 通过JasperReports,允许格式待定 在运行时:

18 2为多JasperReports 视图 类

类名	呈现格式
JasperReportsCsvView	CSV
JasperReportsHtmlView	HTML
JasperReportsPdfView	PDF
JasperReportsXlsView	Microsoft Excel
JasperReportsMultiFormatView	视图是 决定 在运行时

其中一个类的映射视图名称和报告文件 一种添加适当的条目在资源包 在前一节中配置如下所示:

```
simpleReport.(class)=org.springframework.web.servlet.view.jasperreports.JasperReportsPdfView
simpleReport.url=/WEB-INF/reports/DataSourceReport.jasper
```

在这里你可以看到视图名称 simpleReport 映射到 JasperReportsPdfView 类,导致输出 本报告以PDF格式呈现。 这个 url 属性的视图设置的位置 底层的报告文件。

关于报告文件

有两个不同类型的JasperReports报告文件:设计 文件,其中有一个 jrxml 扩展, 编制报告的文件,它有一个 贾斯珀 扩展。 通常,您可以使用Ant任务JasperReports编译 你 jrxml 设计文件到一个 贾斯珀 文件在部署到你的 应用程序。 与Spring框架你可以映射这两种 文件到你的报告文件,框架将照顾 编译 jrxml 文件为你飞翔。 你 应该注意,在吗 jrxml 文件编译 Spring框架,编制报告将缓存的终身 的应用程序。 因此,要改变这个文件你需要 重新启动您的应用程序。

使用 JasperReportsMultiFormatView

这个 JasperReportsMultiFormatView 允许 报告的格式是在运行时指定。 实际呈现的 这份报告是委托给其他JasperReports视图类 —— JasperReportsMultiFormatView 类简单 添加一个包装器层,允许精确的实现 在运行时指定。

这个 JasperReportsMultiFormatView 类 介绍两个概念:格式键和鉴别器关键。 这个 JasperReportsMultiFormatView 类使用 映射键来查找视图的实际实现类,它使用 格式查找关键的映射键。 从编写代码的角度来看 你添加一个条目到您的模型格式的键和关键 映射键值,例如:

```
public ModelAndView handleSimpleReportMulti(HttpServletRequest request,
HttpServletResponse response) throws Exception {

  String uri = request.getRequestURI();
  String format = uri.substring(uri.lastIndexOf(".") + 1);

  Map model = getModel();
  model.put("format", format);

  return new ModelAndView("simpleReportMulti", model);
}
```

在这个例子中,映射键是确定的 扩展的请求URI和添加到模型根据 默认格式的关键: 格式 。 如果你想使用一个 不同的格式,那么你可以配置此键使用 formatKey 财产的 JasperReportsMultiFormatView 类。

默认情况下以下映射键映射中配置的 JasperReportsMultiFormatView：

为多18 3 JasperReportsMultiFormatView 默认 映射键映射

映射键	视图类
CSV	JasperReportsCsvView
HTML	JasperReportsHtmlView
PDF	JasperReportsPdfView
XLS	JasperReportsXlsView

所以在上面的例子中请求URI / foo / myReport。 pdf将 被映射到 JasperReportsPdfView 类。 你 可以覆盖映射视图类映射的关键使用吗 formatMappings 财产的 JasperReportsMultiFormatView 。

18.7.3A填充 ModelAndView

为了使你的报告正确的格式你有 选择,你必须提供弹簧与所有的数据需要填充 你的报告。 这意味着你必须对JasperReports传递所有报告 参数随着报告数据源。 报告参数 简单的名称/值对,可以添加到 地图 为您的模型作为你会添加任何 名称/值对。

当添加数据源模型你有两种方法 选择。 第一种方法是添加的一个实例 JRDataSource 或 收集 类型模型 地图 在任意键。 弹簧将 然后在模型中找到这个对象并将其作为该报告 数据源。 例如,你可能会这样想:填充你的模型

```
private Map getModel() {
  Map model = new HashMap();
  Collection beanData = getBeanData();
  model.put("myBeanData", beanData);
  return model;
}
```

第二种方法是添加的实例 JRDataSource 或 收集 在一个 特定的键,然后配置这个键使用 reportDataKey 属性的视图类。 在这两个 春天将包装的案例实例 收集 在一个 JRBeanCollectionDataSource 实例。 对于 示例:

```
private Map getModel() {
  Map model = new HashMap();
  Collection beanData = getBeanData();
  Collection someData = getSomeData();
  model.put("myBeanData", beanData);
  model.put("someData", someData);
  return model;
}
```

在这里你可以看到两个 收集 实例 被添加到模型中。 确保正确的使用,我们 简单地修改我们的视图配置适当的:

```
simpleReport.(class)=org.springframework.web.servlet.view.jasperreports.JasperReportsPdfView
simpleReport.url=/WEB-INF/reports/DataSourceReport.jasper
simpleReport.reportDataKey=myBeanData
```

注意,当使用第一种方法,春天将使用 第一个实例的 JRDataSource 或 收集 时遇到的。 如果你需要的地方 的多个实例 JRDataSource 或 收集 到模型你需要使用 第二种方法。

18.7.4A处理子报告

JasperReports支持嵌入式子报告内 你的主报告文件。 有各种各样的机制 包括子报告在你的报告文件。 最简单的方式就是努力 代码报告路径和SQL查询的子报告到你的 设计文件。 这种方法的弊端是显而易见的:值 硬编码到你的报告文件,使其减少可重用性 难以修改和更新报告的设计。 要克服这一点,你可以 配置子报告声明,你可以包含额外的数据 对于这些子报告直接从您的控制器。

配置子报告文件

为了控制子报告文件都包含在一个主 报告使用弹簧,你的报告文件必须配置为接受 子报告从外部源。 要做到这一点,你声明一个 你的报告参数文件,类似于:

```
<parameter name="ProductsSubReport" class="net.sf.jasperreports.engine.JasperReport"/>
```

然后,你定义你的子报告使用这个子报告 参数:

```
<subreport>
  <reportElement isPrintRepeatedValues="false" x="5" y="25" width="325"
    height="20" isRemoveLineWhenBlank="true" backcolor="#ffcc99"/>
  <subreportParameter name="City">
    <subreportParameterExpression><![CDATA[$F{city}]]></subreportParameterExpression>
  </subreportParameter>
  <dataSourceExpression><![CDATA[$P{SubReportData}]]></dataSourceExpression>
  <subreportExpression class="net.sf.jasperreports.engine.JasperReport">
          <![CDATA[$P{ProductsSubReport}]]></subreportExpression>
</subreport>
```

这定义了一个主报告文件,预计子报告来 被传递的实例 净科幻jasperreports引擎jasperreports 在 参数 ProductsSubReport 。 当配置你的 碧玉视图类,您可以指示弹簧加载一个报告文件和 将它传递到JasperReports引擎作为子报告使用 subReportUrls 属性:

```
<property name="subReportUrls">
  <map>
    <entry key="ProductsSubReport" value="/WEB-INF/reports/subReportChild.jrxml"/>
  </map>
</property>
```

在这里,关键的 地图 对应于子报告的名称参数报告 设计文件和条目URL的报告文件。 弹簧将 加载这个报告文件,编译,如果有必要,并将其传递到 JasperReports引擎在给定的关键。

配置子报告数据来源

这一步是完全可选当使用Spring配置 子报告。 如果你愿意,你仍然可以配置数据源 你的子报告使用静态查询。 然而,如果你想要的春天 转换返回数据在你的 ModelAndView 进 实例的 JRDataSource 然后你需要指定 这在你的参数 ModelAndView 春天 应该转换。 要做到这一点,配置参数名称的列表使用 这个 subReportDataKeys 属性的选择 视图类:

```
<property name="subReportDataKeys" value="SubReportData"/>
```

在这里,关键你供应 **必须** 对应使用的键都在你 ModelAndView 和使用的键在你的报表设计文件。

18.7.5A出口国参数配置

如果你有特殊的要求配置——出口国 也许你想要一个特定的页面大小为您的PDF报告——你可以 配置这些参数在Spring的声明式出口国 配置文件使用 exporterParameters 属性的视图类。 这个 exporterParameters 属性类型的作为 地图 。 在你的配置关键的入口应该的全名 一个静态字段,该字段包含出口国参数定义,和 一个条目的价值应该是你想要的值分配给 参数。 这样的一个示例如下所示:

```
<bean id="htmlReport" class="org.springframework.web.servlet.view.jasperreports.JasperReportsHtmlView">
 <property name="url" value="/WEB-INF/reports/simpleReport.jrxml"/>
 <property name="exporterParameters">
  <map>
   <entry key="net.sf.jasperreports.engine.export.JRHtmlExporterParameter.HTML_FOOTER">
    <value>Footer by Spring!
     &lt;/td&gt;&lt;td width="50%"&gt;&amp;nbsp; &lt;/td&gt;&lt;/tr&gt;
     &lt;/table&gt;&lt;/body&gt;&lt;/html&gt;
    </value>
   </entry>
  </map>
 </property>
</bean>
```

在这里你可以看到 JasperReportsHtmlView 配置了一个 出口国参数 net.sf.jasperreports.engine.export.JRHtmlExporterParameter.HTML_FOOTER 它将输出一个页脚在生成的HTML。

18.8一个提要的观点

两 AbstractAtomFeedView 和 AbstractRssFeedView 从基类继承 AbstractFeedView 和用于提供原子和 RSS提要的观点恭敬地。 他们是基于java.net的 罗马 项目和位于 包 org.springframework.web.servlet.view.feed 。

AbstractAtomFeedView 需要你 实现 buildFeedEntries() 方法和 选择性地覆盖 buildFeedMetadata() 方法 (默认的实现是空的),如下所示。

```
public class SampleContentAtomView extends AbstractAtomFeedView {

  @Override
  protected void buildFeedMetadata(Map<String, Object> model, Feed feed,
      HttpServletRequest request) {
    // implementation omitted
  }

  @Override
  protected List<Entry> buildFeedEntries(Map<String, Object> model,
      HttpServletRequest request, HttpServletResponse response)
      throws Exception {

    // implementation omitted
  }
}
```

类似的需求申请执行 AbstractRssFeedView ,如下所示。

```
public class SampleContentAtomView extends AbstractRssFeedView {

  @Override
  protected void buildFeedMetadata(Map<String, Object> model, Channel feed,
                                   HttpServletRequest request) {
    // implementation omitted
  }

  @Override
  protected List<Item> buildFeedItems(Map<String, Object> model,
      HttpServletRequest request, HttpServletResponse response)
      throws Exception {
    // implementation omitted
  }

}
```

这个 buildFeedItems() 和 buildFeedEntires() 方法通过在HTTP请求的情况下 你需要访问该地区。 传入的HTTP响应只为 设置的饼干或其他HTTP头。 提要将自动 写入响应对象的方法返回后。

例如,创建一个Atom视图Alef请参考 Arendsen SpringSource的团队博客 条目 。

18.9一个XML编组视图

这个 MarhsallingView 使用XML Marshaller 定义在 org.springframework.oxm 包来呈现 作为XML响应内容。 要数据编组的对象可以直接设置 使用 MarhsallingView ' s modelKey bean属性。 另外,视图将 遍历所有模型属性和元帅只有那些类型支持的 Marshaller 。 更多 信息的功能 org.springframework.oxm 包指的 章 编组XML使用O / X 映射器 。

18.10一个JSON映射视图

这个 MappingJackson2JsonView (或 MappingJacksonJsonView 根据 杰克逊版本你有)使用了杰克逊 图书馆的 ObjectMapper 呈现响应内容 作为JSON。 默认情况下,模型的整个内容(除了地图 特定于框架的类)将被编码成JSON。 为的情况下 内容的地图需要过滤,用户可以指定一个特定的组 模型属性编码通过 RenderedAttributes 财产。 这个 extractValueFromSingleKeyModel 财产 还可用于有价值在单一关键模型提取和 序列化的直接而不是一种映射模型的属性。

JSON映射可以根据需要定制通过使用提供的杰克逊 注释。 当进一步控制是必要的,一个自定义的 ObjectMapper 可以被注入通过吗 ObjectMapper 财产的情况自定义JSON 序列化器/反序列化器需要提供特定的类型。

19。 一个集成其他的web框架

19.1一个介绍

这一章详细Spring的集成与第三方网站 框架如 JSF , Struts , 网络系统 ,和 Tapestry 。

的一个核心价值主张的Spring框架是, 启用 选择 。 一般来说,春天确实 不强制使用或购买到任何特定的体系结构、技术、 或方法(尽管它当然推荐一些对他人)。 这 自由选择架构,技术,或方法 这是最相关开发人员和他或她的开发团队 最明显的网络区域,Spring提供了自己的网络 框架(Spring MVC),而在同一时间 提供集成与一些流行的第三方web框架。 这允许一个继续利用任何和所有的技能一个可能 获得了在一个特定的web框架(如

Spring Web Flow

Spring Web Flow(SWF)的目标是成为最好的解决方案的管理 web应用程序的页面流。

SWF集成了现有的框架(如Spring MVC,Struts, JSF,在两个servlet和portlet的环境。 如果你有

Struts,同时吗 同时能够享受春天带来的好处所提供在其他 等领域的数据访问、声明式事务管理, 灵活的配置和应用程序组装。

有省掉了长毛的销售模式(头脑前面 段),本章的余下部分将集中在肉的 你最喜欢的细节web框架集成与弹簧。 有一件事 这通常是由开发人员来评论在Java从其他 语言是看似超级丰富的web框架可用 java。 确实有大量的Java web框架 空间;事实上有太多盖的表象 细节在单个章节。 这一章因此挑选的四更 流行的web框架在Java中,从Spring配置 这是共同的所有受支持的web框架,然后详细介绍 特定的集成选项为每个支持的web框架。

一个业务 过程(或进程),将受益于一个对话模型 不是一个纯粹的请求模型,然后SWF可能解决方案。

SWF允许您捕获逻辑页面流作为独立的模块 在不同的情况下,是可重用的,因此是理想的建筑web应用程序模块,指导用户通过导航控制 驱动的业务流程。

主权财富基金的更多信息,请参考 Spring Web Flow网站 。

注意

请注意,本章并不试图解释 如何使用任何受支持的web框架。 例如,如果你想 使用Struts的表示层web应用程序的,假设您已经熟悉Struts。 如果你需要 更多细节关于任何受支持的web框架本身, 请做参考 SectionA 19.7,一个进一步Resourcesa 最后 本章的。

19.2公共配置

在深入集成的细节每种受支持的web 框架,让我们首先看一看,Spring配置 不 针对任何一个web框架。 (本节 同样适用于春天的web框架,Spring MVC。)

一个概念(因缺乏一个更好的词)支持 (弹簧)的轻量级应用程序模型是分层的 体系结构。 请记住,在一个“经典”的分层架构,网络 层只是许多层,它作为入口点 到一个服务器端应用程序和它代表的服务对象 (门面)中定义的服务层来满足业务特定的(和 表示技术不可知论者)的用例。 在春天,这些服务 对象,任何其他特定于业务对象和数据访问对象,等等。 存在于一个不同的“业务上下文”,其中包含 没有 web或表示层对象(表示 对象(如Spring MVC控制器通常配置的 不同的“表示上下文”)。 本节详细介绍如何配置 Spring容器(一个 WebApplicationContext), 包含所有的“商业豆”的应用程序。

在细节:所有需要做的就是声明一个 ContextLoaderListener 在标准的Java EE servlet web . xml 文件的网页 应用程序,并添加一个 contextConfigLocation <上下文参数/ >部分(在同一个文件中),定义了哪些设置 Spring XML配置文件的加载。

下面找到<侦听器/ >配置:

```
<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

下面找到<上下文参数/ >配置:

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/applicationContext*.xml</param-value>
</context-param>
```

如果你不指定 contextConfigLocation 上下文参数, ContextLoaderListener 将 寻找一个文件叫做 / - inf /中 加载。 一旦上下文文件加载,Spring创建一个 WebApplicationContext 对象根据bean定义并将其存储在 ServletContext web应用程序的。

所有的Java web框架的基础上构建Servlet API,所以 一个可以使用下面的代码片段来获得这种“业务 上下文的ApplicationContext 创造的 ContextLoaderListener 。

```
WebApplicationContext ctx = WebApplicationContextUtils.getWebApplicationContext(servletContext);
```

这个 WebApplicationContextUtils 类是为了方便,所以你不必记住这个名字的 ServletContext 属性。 它的getWebApplicationContext() 方法将返回 空 如果一个对象不存在根据WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE 关键。 而不是风险越来越nullpointerexception 在你的应用程序,它的更好的使用 getRequiredWebApplicationContext() 法。 这种方法 抛出一个异常当 ApplicationContext 是 失踪的。

一旦你有了一个参考 WebApplicationContext ,您可以检索bean通过 他们的名称或类型。 大多数开发人员检索bean的名字,然后把他们 到他们的一个实现接口。

幸运的是,大多数的框架在这一节中有简单的 查找bean的方法。 它们不仅会很容易让豆子 Spring容器,但他们还允许您在使用依赖注入 他们的控制器。 每个web框架部分有更多的细节在其 特定的集成策略。

19.3一个JavaServer Faces 1.1和1.2

JavaServer Faces(JSF)是基于组件的JCP标准, 事件驱动的web用户界面框架。 在Java EE 5,这是一个 官方的部分Java EE的伞。

JSF运行时为流行和受欢迎的JSF组件 库、检查 Apache MyFaces项目 。 项目还提供了常见的MyFaces JSF 扩展如 MyFaces乐团 :一个 基于spring的JSF扩展提供了丰富的谈话范围 支持。

注意

Spring Web Flow 2.0提供了丰富的JSF支持通过其新 建立了弹簧面临模块,包括jsf为中心的使用(如 这一节中描述的)和弹簧中心使用(使用JSF视图 在一个Spring MVC调度员)。 检查 Spring Web Flow 网站 详情!

关键的元素在春天的JSF集成JSF 1.1 VariableResolver 机制。 在JSF 1.2中,弹簧 支持 ELResolver 机制作为一个 新一代版本的JSF EL集成。

19.3.1A DelegatingVariableResolver(JSF 1.1/1.2)

最简单的方法是将一个中间层的春天 JSF web层是使用 DelegatingVariableResolver 类。 到 这个变量解析器配置的应用程序,你将需要 编辑人的 faces上下文xml 文件。 后 开放 <面临config / > 元素,添加一个 <应用程序/ > 元素和一个 <变量解析器/ > 元素在它。 这个 值的变量解析器应该参考弹簧的 DelegatingVariableResolver ;例如:

```
<faces-config>
  <application>
    <variable-resolver>org.springframework.web.jsf.DelegatingVariableResolver</variable-resolver>
    <locale-config>
      <default-locale>en</default-locale>
      <supported-locale>en</supported-locale>
      <supported-locale>es</supported-locale>
    </locale-config>
    <message-bundle>messages</message-bundle>
  </application>
</faces-config>
```

这个 DelegatingVariableResolver 将首先 代表值查找默认解析器的底层JSF 实现,然后春天的 “业务上下文的WebApplicationContext 。 这允许一个人很容易 将依赖项注入到一个的jsf托管bean。

托管bean的定义 面临配置xml 文件。 找到一个例子,下面 # { userManager } 是一个bean,获取 春天的业务上下文” 。

```
<managed-bean>
  <managed-bean-name>userList</managed-bean-name>
  <managed-bean-class>com.whatever.jsf.UserList</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>userManager</property-name>
    <value>#{userManager}</value>
  </managed-property>
</managed-bean>
```

19.3.2A SpringBeanVariableResolver(JSF 1.1/1.2)

SpringBeanVariableResolver 是一个变种的 DelegatingVariableResolver 。 它代表 春天的 “业务上下文的WebApplicationContext 第一 然后到默认的解析器 潜在的JSF实现。 这是有用的特别是当使用 请求/会话范围内的bean与特殊弹簧解析规则,例如。 春天 FactoryBean 实现。

配置聪明,简单的定义 SpringBeanVariableResolver 在你的 faces上下文xml 文件:

```
<faces-config>
  <application>
    <variable-resolver>org.springframework.web.jsf.SpringBeanVariableResolver</variable-resolver>
    ...
  </application>
</faces-config>
```

19.3.3A SpringBeanFacesELResolver(JSF 1.2 +)

SpringBeanFacesELResolver 是一个JSF 1.2 兼容 ELResolver 实现,整合 与标准统一EL使用JSF 1.2和JSP 2.1。 像 SpringBeanVariableResolver ,它代表 春天的 “业务上下文的 WebApplicationContext 第一 ,然后到默认的解析器 潜在的 JSF实现。

配置聪明,简单的定义 SpringBeanFacesELResolver 在JSF 1.2 faces上下文xml 文件:

```
<faces-config>
  <application>
    <el-resolver>org.springframework.web.jsf.el.SpringBeanFacesELResolver</el-resolver>
    ...
  </application>
</faces-config>
```

19.3.4A FacesContextUtils

一个自定义 VariableResolver 作品 当一个属性映射的豆子 面临配置xml ,但有时你可能需要抓住 一个bean明确。 这个 FacesContextUtils 类使这项任务变得很容易。 它类似于 WebApplicationContextUtils , 除了需要一个 FacesContext 参数 而不是 ServletContext 参数。

```
ApplicationContext ctx = FacesContextUtils.getWebApplicationContext(FacesContext.getCurrentInstance());
```

19.4一个Apache Struts 1。 倍和2.倍

Struts 曾经是 事实上的 Java应用程序的web框架,主要 因为它是第一个被释放(2001年6月)。 它已经被重命名为 Struts 1 (与 Struts 2)。许多应用程序仍在使用它。 发明的克雷格·麦克拉纳罕,Struts是一个开源项目托管的Apache 软件基础。 当时,它大大简化了JSP / Servlet 编程范型和赢得了许多开发人员使用 专有的框架。 它简化了编程模型,它是开放的 源(因此免费的啤酒),它有一个很大的社区,它 允许项目增长,成为深受Java web 开发人员。

> **注意**
>
> 以下部分将讨论Struts 1即。 “Struts 经典” 。
>
> Struts 2实际上是一个不同的产品——一个继承人 网络系统(详见2.2 SectionA 19.5,一个xa网络系统2.),携带 Struts品牌现在。 查看Struts 2 春天 插件 对于内置的Spring integration附带Struts 2。 一般来说,Struts 2是接近网络系统2.2比Struts 1 Spring integration方面影响。

整合你的Struts 1。 x应用程序与春天,你有两个 选项:

- 配置弹簧来管理你的动作像豆子,使用 ContextLoaderPlugin ,并设置它们的依赖项 在Spring上下文文件。
- 子类春天的 ActionSupport 类 和拿起你的spring管理bean明确使用 getWebApplicationContext() 法。

19.4.1A ContextLoaderPlugin

这个 ContextLoaderPlugin 是一个Struts 1.1 +插件加载Spring上下文文件为Struts吗 ActionServlet 。 这里指的是根 WebApplicationContext (加载 ContextLoaderListener)作为它的父类。 默认 名称的上下文文件映射servlet的名称,再加上 servlet xml 。 如果 ActionServlet 定义在web。 xml作为 < servlet - name > < / servlet - name >行动 , 默认是 / - inf / 行动servlet xml 。

配置该插件,添加以下XML插件 部分的底部附近 struts - config . xml 文件:

```
<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn"/>
```

上下文的位置配置文件可以被定制 使用' contextConfigLocation 的财产。

```
<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">
  <set-property property="contextConfigLocation"
      value="/WEB-INF/action-servlet.xml,/WEB-INF/applicationContext.xml"/>
</plug-in>
```

可以使用这个插件加载你所有的上下文文件, 它可能是有用的在使用测试工具像StrutsTestCase。 StrutsTestCase的 MockStrutsTestCase 不会 在启动时初始化听众因此,把你的上下文文件 插件是一个解决方案。 (bug已经立案 对于这个问题,但是已经关闭 “不会 修复的)。

配置这个插件后在 struts - config . xml ,您可以配置您的 行动 能够管理弹簧。 弹簧(1.1.3 +) 提供了两个方法:

- 覆盖Struts的默认 几乎 与春天的 DelegatingRequestProcessor 。
- 使用 DelegatingActionProxy 类 这个 类型 属性的 <操作映射> 。

这两种方法让你管理你的行动和他们的 依赖性的 行动servlet xml 文件。 这个 行动之间的桥梁 struts - config . xml 和 行动servlet xml 是建立与 操作映射的“路径”和“名字”的bean。 如果你有以下 在你的 struts - config . xml 文件:

```
<action path="/users" .../>
```

你必须定义的bean,行动与“/用户”的名字 行动servlet xml :

```
<bean name="/users" .../>
```

DelegatingRequestProcessor

配置 DelegatingRequestProcessor 在你的 struts - config . xml 文件,覆盖 “processorClass”属性的<控制器>元素。 这些线按照<操作映射>元素。

```
<controller>
  <set-property property="processorClass"
     value="org.springframework.web.struts.DelegatingRequestProcessor"/>
</controller>
```

在添加此设置,你的行动将会被自动 抬头在春天的上下文文件,不管什么类型。 事实上, 你甚至不需要指定一个类型。 下面两个片段 将工作:

```
<action path="/user" type="com.whatever.struts.UserAction"/>
<action path="/user"/>
```

如果你使用Struts的 模块 功能, 您的bean名称必须包含模块的前缀。 例如,一个动作 定义为 <行动路径= " /用户" / > 与模块前缀“admin”需要一个bean名字 < bean name = " / admin /用户" / > 。

注意

如果您使用的是瓷砖在Struts应用程序,你必须 配置您的<控制器>与 DelegatingTilesRequestProcessor 相反。

DelegatingActionProxy

如果你有一个自定义的 几乎 和 不能使用 DelegatingRequestProcessor 或 DelegatingTilesRequestProcessor 方法,你 可以使用 DelegatingActionProxy 作为输入 你的操作映射。

```
<action path="/user" type="org.springframework.web.struts.DelegatingActionProxy"
   name="userForm" scope="request" validate="false" parameter="method">
  <forward name="list" path="/userList.jsp"/>
  <forward name="edit" path="/userForm.jsp"/>
</action>
```

该bean定义在 行动servlet xml 是一样的,不管你是使用一个自定义的 几乎 或 DelegatingActionProxy 。

如果你定义你的 行动 在一个上下文 文件,完整的特性集Spring的bean容器将 可用于:依赖注入以及选择 实例化一个新的 行动实例为每个 请求。 激活后,添加 范围= “原型” 你的行动的bean 定义。

```
<bean name="/user" scope="prototype" autowire="byName"
   class="org.example.web.UserAction"/>
```

19.4.2A ActionSupport类

如前所述,您可以检索 WebApplicationContext 从 ServletContext 使用 WebApplicationContextUtils 类。 一个更简单的方法 是扩展Spring的吗 行动 Struts类。 例如,而不是子类化Struts” 行动 类,你可以子类春天的 ActionSupport 类。

这个 ActionSupport 类提供了额外的 方便的方法,如 getWebApplicationContext() 。 下面是一个例子 您将如何使用该在一个动作:

```
public class UserAction extends DispatchActionSupport {

   public ActionForward execute(ActionMapping mapping,
```

```
                        ActionForm form,
                        HttpServletRequest request,
                        HttpServletResponse response) throws Exception {
        if (log.isDebugEnabled()) {
            log.debug("entering 'delete' method...");
        }
        WebApplicationContext ctx = getWebApplicationContext();
        UserManager mgr = (UserManager) ctx.getBean("userManager");
        // talk to manager for business logic
        return mapping.findForward("success");
    }
}
```

弹簧包括所有的子类Struts动作标准 ——春天版本仅仅有 支持 附加 名字:

- ActionSupport ,
- DispatchActionSupport ,
- LookupDispatchActionSupport 和
- MappingDispatchActionSupport 。

推荐的策略是使用最适合的方法 你的项目。 子类化使代码更可读,你知道 你究竟是如何解析依赖项。 相比之下,使用ContextLoaderPlugin 允许您方便地添加新的 在你的上下文依赖XML文件。 无论哪种方式,弹簧提供了一些 好选择集成Struts。

19.5网络系统2. x

从 网络系统 主页 :

一个 网络系统是一个Java web应用程序开发框架。 这是 与开发人员的生产力和专代码简单 介意,提供强劲的支持构建可重用的UI模板,这样 作为表单控件,UI主题、国际化、动态形式 参数映射到javabean,健壮的客户端和服务器端 验证,以及更多。 一个

网络工作的体系结构和概念很容易 理解,和框架还有一个广泛的标记库以及 很好地解耦验证。

一个关键的推动者在网络系统的技术堆栈 一个 IoC容器 管理网络系统操作,处理 “线路” 业务对象等。网络系统2.2版本之前,网络系统使用自己的 专有的IoC容器(并提供集成点,这样一个 可以整合一个IoC容器如春天的混进去)。 然而, 在网络系统2.2版本,默认的IoC容器内使用 网络系统 是 春天。 这显然是伟大的新闻如果 一个是一个春天的开发人员,因为它意味着,一个是立即熟悉基本的IoC配置、习语等在 网络系统。

现在的利益坚持干(不要重复你自己) 原则,那将是愚蠢的弹簧网络系统集成的文档 鉴于这一事实,网络系统团队已经写这样一个那样。 请查阅 弹簧网络系统 集成页面 在 网络系统wiki 对于完整的真相。

注意,弹簧网络系统集成代码开发(和 继续保持和改进)的网络系统开发人员 他们自己。 所以请参考第一到网络系统网站和论坛如果你 有问题的集成。 但随时 发表评论和查询关于弹簧网络系统集成的 春天 支持论坛 ,太。

19.6一个Tapestry 3。 倍和4.倍

从 Tapestry 主页 :

一个 Tapestry是一个开源框架来创建动态, 健壮的、高度可伸缩的web应用程序的Java。 Tapestry补充 并基于标准的Java Servlet API,所以它的作品在任何 servlet容器或应用程序服务器。 一个

虽然春天有它自己的 强大的web 层 ,有许多独特的优势,构建一个 企业Java应用程序结合使用Tapestry的web 用户界面和Spring容器的低层次。 本节 网络集成一章试图详细一些最佳实践 结合这两个框架。

一个 典型 分层的企业级Java应用程序 建立与Tapestry和弹簧将包括一个顶级用户界面(UI) 层由Tapestry,和大量的低层次,所有连接到一起 由一个或多个弹簧容器。 Tapestry的参考文档 包含以下代码片段的最佳实践建议。 (文本, 今年春天的作者部分增加了包含在 [] 括号内。)

一个 一个非常成功的设计模式在Tapestry是保持页面 和组件非常简单,和 **代表** 尽可能多的逻辑了 HiveMind[或弹簧,或者无论]服务。 侦听器方法应该 理想情况下做小超过元帅一起正确的信息 并将其传递到一个服务。 一个

关键的问题就是:如何供应与Tapestry页面 协作服务? 答案,在理想的情况下,是一个会想 依赖项注入这些服务直接进人的Tapestry页面。 在 Tapestry,一个人可以影响这个依赖项注入由多种 意味着。 本节只会列举依赖注入 意味着春天所提供。 真正的美丽的其余部分 春天tapestry集成是优雅而灵活的设计 Tapestry本身让做这个依赖项注入spring管理的 豆子一个有把握

的事情。 (另一个好处是,这个春天挂毯 集成代码编写,并继续保持-的 Tapestry的创造者 霍华德·M。 刘易斯船 ,所以向他致敬,什么是真的有些柔滑 平滑集成)。

19.6.1A注入spring管理bean

假设我们有以下简单的Spring容器定义 (在无处不在的XML格式):

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:jee="http://www.springframework.org/schema/jee"
      xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee.xsd">

<beans>
   <!-- the DataSource -->
   <jee:jndi-lookup id="dataSource" jndi-name="java:DefaultDS"/>

   <bean id="hibSessionFactory"
       class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
      <property name="dataSource" ref="dataSource"/>
   </bean>

   <bean id="transactionManager"
       class="org.springframework.transaction.jta.JtaTransactionManager"/>

   <bean id="mapper"
       class="com.whatever.dataaccess.mapper.hibernate.MapperImpl">
      <property name="sessionFactory" ref="hibSessionFactory"/>
   </bean>

   <!-- (transactional) AuthenticationService -->
   <bean id="authenticationService"
       class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
      <property name="transactionManager" ref="transactionManager"/>
      <property name="target">
         <bean class="com.whatever.services.service.user.AuthenticationServiceImpl">
            <property name="mapper" ref="mapper"/>
         </bean>
      </property>
      <property name="proxyInterfacesOnly" value="true"/>
      <property name="transactionAttributes">
         <value>
            *=PROPAGATION_REQUIRED
         </value>
      </property>
   </bean>

   <!-- (transactional) UserService -->
   <bean id="userService"
       class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
      <property name="transactionManager" ref="transactionManager"/>
      <property name="target">
         <bean class="com.whatever.services.service.user.UserServiceImpl">
            <property name="mapper" ref="mapper"/>
         </bean>
      </property>
      <property name="proxyInterfacesOnly" value="true"/>
      <property name="transactionAttributes">
         <value>
            *=PROPAGATION_REQUIRED
         </value>
      </property>
   </bean>

 </beans>
```

在Tapestry应用程序,上面的bean定义需要 是 加载到一个春天 容器 和任何相关的Tapestry页面需要提供 (注射) authenticationService 和 userService 豆类,它实现了 authenticationService 和 userService 接口, 分别。

在这一点上,应用程序上下文可用web 应用程序通过调用Spring的静态的效用函数 WebApplicationContextUtils.getApplicationContext(servletContext) , 在servletContext是标准的吗 ServletContext 从 Java EE Servlet 规范。 因此,一个简单的机制来得到一个页面 实例的 userService ,例如, 将代码如:

```
WebApplicationContext appContext = WebApplicationContextUtils.getApplicationContext(
   getRequestCycle().getRequestContext().getServlet().getServletContext());
UserService userService = (UserService) appContext.getBean("userService");
// ... some code which uses UserService
```

这种机制并工作。 已经说过,它可以制作了很多 更少的冗长的大部分功能通过封装在一个方法中 的基类或组件的页面。 然而,在某些方面它 违背国际奥委会原则;理想情况下你想页面没有 要问为一个特定的上下文bean的名字,事实上, 页面将理想不知道

上下文在所有。

幸运的是,有一种机制,允许这个。 我们依赖的事实 Tapestry已经有一个机制来声明添加属性 一个页面,它实际上是首选的方法来管理所有 属性在一个页面在这个声明式的方式,所以,Tapestry可以 妥善管理自己的生命周期的一部分页面和组件 生命周期。

> **注意**
>
> 这在下一节适用于Tapestry 3. x。 如果你是 使用Tapestry版本4。 x,请咨询一节 一个章节依赖Spring bean注入到Tapestry页面- Tapestry 4。 x即将 。

依赖注入Spring bean到Tapestry页面

首先我们需要使 ApplicationContext 可用到Tapestry 页面或组件,不需要有 ServletContext ,这是因为在舞台上 页面的/组件的生命周期,当我们需要访问 ApplicationContext , ServletContext 不会是容易的 页面,因此我们不能使用 WebApplicationContextUtils.getApplicationContext(servletContext) 直接。 一种方法是通过定义一个定制版本的挂毯 IEngine 它公开了这 我们:

```
package com.whatever.web.xportal;

// import ...

public class MyEngine extends org.apache.tapestry.engine.BaseEngine {

    public static final String APPLICATION_CONTEXT_KEY = "appContext";

    /**
     * @see org.apache.tapestry.engine.AbstractEngine#setupForRequest(org.apache.tapestry.request.RequestContext)
     */
    protected void setupForRequest(RequestContext context) {
        super.setupForRequest(context);

        // insert ApplicationContext in global, if not there
        Map global = (Map) getGlobal();
        ApplicationContext ac = (ApplicationContext) global.get(APPLICATION_CONTEXT_KEY);
        if (ac == null) {
            ac = WebApplicationContextUtils.getWebApplicationContext(
                context.getServlet().getServletContext()
            );
            global.put(APPLICATION_CONTEXT_KEY, ac);
        }
    }
}
```

这个引擎类地方的Spring应用程序上下文作为 属性称为 “appContext” 在这个Tapestry应用程序的 “全球” 对象。 确保注册这特殊的IEngine实例 应该用于这Tapestry应用程序,一个条目在吗 Tapestry应用程序定义文件。 例如:

```
file: xportal.application:
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC
    "-//Apache Software Foundation//Tapestry Specification 3.0//EN"
    "http://jakarta.apache.org/tapestry/dtd/Tapestry_3_0.dtd">
<application
    name="Whatever xPortal"
    engine-class="com.whatever.web.xportal.MyEngine">
</application>
```

组件定义文件

现在在我们的页面或组件定义文件(*。 页面或* .jwc), 我们只是添加属性规范元素获取bean我们 需要的 ApplicationContext , 和 创建页面或组件属性对于他们。 例如:

```
    <property-specification name="userService"
                    type="com.whatever.services.service.user.UserService">
        global.appContext.getBean("userService")
    </property-specification>
    <property-specification name="authenticationService"
                    type="com.whatever.services.service.user.AuthenticationService">
        global.appContext.getBean("authenticationService")
    </property-specification>
```

OGNL表达式内的财产规范指定了 初始值的属性,作为一个bean获得的 上下文。 整个页面的定义看起来像这样:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE page-specification PUBLIC
```

```
    "-//Apache Software Foundation//Tapestry Specification 3.0//EN"
    "http://jakarta.apache.org/tapestry/dtd/Tapestry_3_0.dtd">

<page-specification class="com.whatever.web.xportal.pages.Login">

    <property-specification name="username" type="java.lang.String"/>
    <property-specification name="password" type="java.lang.String"/>
    <property-specification name="error" type="java.lang.String"/>
    <property-specification name="callback" type="org.apache.tapestry.callback.ICallback" persistent="yes"/>
    <property-specification name="userService"
                    type="com.whatever.services.service.user.UserService">
      global.appContext.getBean("userService")
    </property-specification>
    <property-specification name="authenticationService"
                    type="com.whatever.services.service.user.AuthenticationService">
      global.appContext.getBean("authenticationService")
    </property-specification>

    <bean name="delegate" class="com.whatever.web.xportal.PortalValidationDelegate"/>

    <bean name="validator" class="org.apache.tapestry.valid.StringValidator" lifecycle="page">
      <set-property name="required" expression="true"/>
      <set-property name="clientScriptingEnabled" expression="true"/>
    </bean>

    <component id="inputUsername" type="ValidField">
      <static-binding name="displayName" value="Username"/>
      <binding name="value" expression="username"/>
      <binding name="validator" expression="beans.validator"/>
    </component>

    <component id="inputPassword" type="ValidField">
      <binding name="value" expression="password"/>
     <binding name="validator" expression="beans.validator"/>
     <static-binding name="displayName" value="Password"/>
     <binding name="hidden" expression="true"/>
    </component>

</page-specification>
```

添加抽象访问器

现在在Java类定义为页面或组件 本身,所有我们需要做的就是添加一个抽象的getter方法 我们已经定义属性(为了能够访问 属性)。

```
// our UserService implementation; will come from page definition
public abstract UserService getUserService();
// our AuthenticationService implementation; will come from page definition
public abstract AuthenticationService getAuthenticationService();
```

为了完整性,整个Java类,对于一个登录 页面在这个示例中,看起来像这样:

```
package com.whatever.web.xportal.pages;

/**
 *  Allows the user to login, by providing username and password.
 *  After successfully logging in, a cookie is placed on the client browser
 *  that provides the default username for future logins (the cookie
 *  persists for a week).
 */
public abstract class Login extends BasePage implements ErrorProperty, PageRenderListener {

    /** the key under which the authenticated user object is stored in the visit as */
    public static final String USER_KEY = "user";

    /** The name of the cookie that identifies a user **/
    private static final String COOKIE_NAME = Login.class.getName() + ".username";
    private final static int ONE_WEEK = 7 * 24 * 60 * 60;

    public abstract String getUsername();
    public abstract void setUsername(String username);

    public abstract String getPassword();
    public abstract void setPassword(String password);

    public abstract ICallback getCallback();
    public abstract void setCallback(ICallback value);

    public abstract UserService getUserService();
    public abstract AuthenticationService getAuthenticationService();

    protected IValidationDelegate getValidationDelegate() {
        return (IValidationDelegate) getBeans().getBean("delegate");
    }

    protected void setErrorField(String componentId, String message) {
```

```
        IFormComponent field = (IFormComponent) getComponent(componentId);
        IValidationDelegate delegate = getValidationDelegate();
        delegate.setFormComponent(field);
        delegate.record(new ValidatorException(message));
    }

    /**
     *  Attempts to login.
     * <p>
     *  If the user name is not known, or the password is invalid, then an error
     *  message is displayed.
     **/
    public void attemptLogin(IRequestCycle cycle) {

        String password = getPassword();

        // Do a little extra work to clear out the password.
        setPassword(null);
        IValidationDelegate delegate = getValidationDelegate();

        delegate.setFormComponent((IFormComponent) getComponent("inputPassword"));
        delegate.recordFieldInputValue(null);

        // An error, from a validation field, may already have occurred.
        if (delegate.getHasErrors()) {
            return;
        }

        try {
            User user = getAuthenticationService().login(getUsername(), getPassword());
           loginUser(user, cycle);
        }
        catch (FailedLoginException ex) {
            this.setError("Login failed: " + ex.getMessage());
            return;
        }
    }

    /**
     *  Sets up the {@link User} as the logged in user, creates
     *  a cookie for their username (for subsequent logins),
     *  and redirects to the appropriate page, or
     *  a specified page).
     **/
    public void loginUser(User user, IRequestCycle cycle) {

        String username = user.getUsername();

        // Get the visit object; this will likely force the
        // creation of the visit object and an HttpSession
        Map visit = (Map) getVisit();
        visit.put(USER_KEY, user);

        // After logging in, go to the MyLibrary page, unless otherwise specified
        ICallback callback = getCallback();

        if (callback == null) {
            cycle.activate("Home");
        }
        else {
            callback.performCallback(cycle);
        }

        IEngine engine = getEngine();
        Cookie cookie = new Cookie(COOKIE_NAME, username);
        cookie.setPath(engine.getServletPath());
        cookie.setMaxAge(ONE_WEEK);

        // Record the user's username in a cookie
        cycle.getRequestContext().addCookie(cookie);
        engine.forgetPage(getPageName());
    }

    public void pageBeginRender(PageEvent event) {
        if (getUsername() == null) {
            setUsername(getRequestCycle().getRequestContext().getCookieValue(COOKIE_NAME));
        }
    }
}
```

依赖注入Spring bean到Tapestry页面- Tapestry 4。 x风格

实现依赖注入spring管理的bean到 Tapestry页面在Tapestry版本4。 x是 所以 多 更简单的。 所需要的是一个单 附加 图书馆 ,和一些(小)金额(实质上是样板) 配置。 简单的打包和部署这个库的任何 其他的库的)需要你的web应用程序(通常在 - inf / lib)。

然后,您将需要创建和公开Spring容器 使用 方法详细 以前 。 然后你可以注入spring管理bean到 Tapestry很容易;如果我们使

用Java 5,考虑 登录 页面从上面:我们只需要 注释相应的getter方法,以便依赖注入 spring管理的 userService 和 authenticationService 对象(很多类 定义为清晰起见省略)。

```
package com.whatever.web.xportal.pages;

public abstract class Login extends BasePage implements ErrorProperty, PageRenderListener {

    @InjectObject("spring:userService")
    public abstract UserService getUserService();

    @InjectObject("spring:authenticationService")
    public abstract AuthenticationService getAuthenticationService();

}
```

我们已经基本完成。 剩下的只有HiveMind 配置,公开Spring容器中存储的 ServletContext 作为一个HiveMind服务; 例如:

```
<?xml version="1.0"?>
<module id="com.javaforge.tapestry.spring" version="0.1.1">

    <service-point id="SpringApplicationInitializer"
       interface="org.apache.tapestry.services.ApplicationInitializer"
       visibility="private">
       <invoke-factory>
          <construct class="com.javaforge.tapestry.spring.SpringApplicationInitializer">
             <set-object property="beanFactoryHolder"
                value="service:hivemind.lib.DefaultSpringBeanFactoryHolder" />
          </construct>
       </invoke-factory>
    </service-point>

    <!-- Hook the Spring setup into the overall application initialization. -->
    <contribution
       configuration-id="tapestry.init.ApplicationInitializers">
       <command id="spring-context"
          object="service:SpringApplicationInitializer" />
    </contribution>

</module>
```

如果您使用的是Java 5(从而获得注释), 那真的是它。

如果你不使用Java 5,那么你显然没有注释 一个是Tapestry页面类注释;相反,一个简单的使用 老式的XML声明依赖注入,因为 例,在 页面 或 .jwc 文件 登录 页面 (或组件):

```
<inject property="userService" object="spring:userService"/>
<inject property="authenticationService" object="spring:authenticationService"/>
```

在这个例子中,我们设法让服务定义的bean Spring容器提供到Tapestry页面声明 时尚。 页面类不知道服务的实现 来自,事实上很容易滑倒在另一个的实现, 例如,在测试过程中。 这个控制反转是最好的 目标和利益的Spring框架,我们设法延长 它在整个堆栈在这个Tapestry应用程序。

19.7进一步资源

以下链接资源的各种网络进一步 本章中描述的框架。

- 这个 JSF 主页
- 这个 Struts 主页
- 这个 网络系统 主页
- 这个 Tapestry 主页

20。 一个Portlet的MVC框架

20.1一个介绍

除了支持传统的(基于servlet的Web开发, 春天也支持jsr - 168 Portlet开发。 尽可能的 Portlet MVC框架的镜像Web MVC框架,也 使用相同的底层视图抽象和集成技术。 所以,是 确定审核章题为 ChapterA 17, Web MVC框架 和 ChapterA 18, 视图技术 然后再继续这一章。

jsr - 168规范的Java Portlet

对于更一般的信息关于portlet开发,请 回顾一个白皮书从太阳资格 “介绍JSR 168” , 当然, jsr - 168规范 本身。

注意

记住,虽然Spring MVC的概念的 在春天Portlet MVC相同,有一些显著的差异 由独特的工作流的jsr - 168 portlet。

主要的方式不同于servlet portlet工作流 工作流是请求的portlet可以有两种截然不同的 阶段:操作阶段和渲染阶段。 在操作阶段是 只执行一次,就是任何“后端”更改或行为发生, 如在数据库进行更改。 在呈现阶段然后生产 什么是显示给用户每次刷新显示。 这里的关键点是,对于一个总体要求,动作 阶段是只执行一次,但呈现阶段可以执行 多次。 这提供了(并且需要)一个干净的分离 这个活动持续状态修改您的系统和 这些活动会产生什么是显示给用户。

双阶段的portlet请求是一个真正的优势 jsr - 168规范的。 例如,动态搜索结果可以 定期更新显示没有用户显式地运行 搜索。 大多数其他portlet的MVC框架试图完全 隐藏两个阶段从开发人员和使它看起来很像 传统servlet开发——我们认为这是可能的 方法删除的一个主要好处使用portlet。 所以, 分离的两个阶段是保存整个春天的Portlet MVC框架。 这种方法的主要表现是,在 servlet版本的MVC类将有一个方法,交易 请求,portlet版本的MVC类将有两个 处理请求的方法:一个用于在操作阶段和一个用于 渲染阶段。 例如,servlet版本的 基类AbstractController 有handleRequestInternal(. .) 法,portlet 版本的 基类AbstractController 已经 handleActionRequestInternal(. .) 和handleRenderRequestInternal(. .) 方法。

这个框架被设计在一个 DispatcherPortlet 分派请求,对 处理程序,可配置的处理程序映射和视图解析,只是 随着 DispatcherServlet 在web框架 确实。 文件上传也是以同样的方式支持。

Spring Web Flow

Spring Web Flow(SWF)的目标是成为最好的解决方案的管理 web应用程序的页面流。

SWF集成了现有的框架(如Spring MVC,Struts, JSF,在两个servlet和portlet的环境。 如果你有一个业务 过程(或进程),将受益于一个对话模型不是一个纯粹的请求模型,然后SWF可能解决方案。

SWF允许您捕获逻辑页面流作为独立的模块 在不同的情况下,是可重用的,因此是理想的建筑web应用程序模块,指导用户通过导航控制 驱动的业务流程。

主权财富基金的更多信息,请参考 Spring Web Flow网站 。

语言环境的分辨率和主题不支持决议 这些地区的Portlet MVC -在的权限门户/ portlet容器和不适当的在春天水平。 然而,所有的机制在春天,依赖语言环境(如 国际化的信息)将仍然正常运作,因为DispatcherPortlet 暴露在当前语言环境 一样 DispatcherServlet 。

20.1.1A控制器- C在MVC

默认的处理程序仍然是一个非常简单的 控制器 接口,提供两个 方法:

- 无效handleActionRequest(请求、响应)
- handleRenderRequest ModelAndView(请求、响应)

该框架还包括大多数相同的控制器 实现层次,如 基类AbstractController , SimpleFormController ,等等。 数据绑定, 使用命令对象,模型处理和视图解析都是 在servlet框架一样。

20.1.2A视图- V在MVC

所有的视图呈现功能的servlet框架 通过一个特殊的桥直接使用servlet命名 ViewRendererServlet 。 通过使用这个servlet, portlet请求转换为servlet请求和视图可以 使用完整呈现正常的servlet的基础设施。 这意味着所有 现有的渲染器,比如JSP、速度等,仍然是可以使用的 在portlet。

20.1.3A web范围豆

春天Portlet支持bean的生命周期的MVC的作用范围是整个 当前HTTP请求或HTTP 会话 (两 正常和全球)。 这不是春天的具体特性Portlet MVC 本身,而是的 WebApplicationContext 容器(s),弹簧Portlet MVC使用。 这些bean范围描述 详细 SectionA 5 5 4,一个请求、会话和scopesa全球会议

20.2一个的 DispatcherPortlet

MVC是一个Portlet请求驱动web MVC框架,围绕 一个portlet,分派请求控制器和提供其他 功能促进开发portlet应用程序。 春天的 DispatcherPortlet 然而,做更多 就比。 它完全集成了春天 ApplicationContext 和允许您使用 其他功能春天。

像普通的portlet, DispatcherPortlet 是宣布在 portlet . xml 文件的web应用程序:

```
<portlet>
  <portlet-name>sample</portlet-name>
  <portlet-class>org.springframework.web.portlet.DispatcherPortlet</portlet-class>
  <supports>
    <mime-type>text/html</mime-type>
    <portlet-mode>view</portlet-mode>
  </supports>
  <portlet-info>
    <title>Sample Portlet</title>
  </portlet-info>
</portlet>
```

这个 DispatcherPortlet 现在需要 配置。

在Portlet的MVC框架,每个 DispatcherPortlet 有自己的 WebApplicationContext ,它继承了所有 已经定义的bean在根 WebApplicationContext 。 这些遗传 豆子可以覆盖在portlet特定的范围,和新 可以定义范围特定bean本地一个给定的 portlet实例。

该框架将,在初始化的一个 DispatcherPortlet ,找一个文件命名 [portlet名称]portlet xml 在 - inf 您的web应用程序的目录并 创建bean定义 (覆盖任何bean定义的定义具有相同的名字 全球范围)。

配置使用的位置 DispatcherPortlet 可以修改通过吗 portlet的初始化参数(见以下内容)。

春天 DispatcherPortlet 有几个 特别的豆子它使用,为了能够处理请求和 呈现适当的视图。 这些bean是包含在春天 框架和可配置的 WebApplicationContext ,就像任何其他 bean将配置。 每个bean描述更多 下面详细。 现在,我们将只提到他们,只是让你知道 他们的存在,使我们继续谈论 DispatcherPortlet 。 对于大多数的豆子, 违约提供,所以你不必担心配置 他们。

20.1为多。 一个特别的豆子在 WebApplicationContext

表达式	解释
处理程序映射(年代)	(SectionA 20.5,一个处理程序mappingsa) 列表的前和后处理器和控制器 将被执行,如果他们匹配某些标准(实例匹配指定的portlet模式 控制器)
控制器(s)	(SectionA 20.4,一个Controllersa)豆子 提供实际的功能(或者至少,访问 功能)的一部分,MVC三和弦
视图解析器	(SectionA 20.6,一个观点和解决主题)能够 解决视图名称以视图定义
几部分的解析器	(SectionA 20.7,一个扇形(文件上传)supporta)提供 文件上传功能过程从HTML 形式
处理程序异常解析器	(SectionA 20.8,一个处理exceptionsa) 提供功能映射视图或例外 实现其他更复杂的异常处理 代码

当一个 DispatcherPortlet 是设置为使用 和一个请求到来时,特定的 DispatcherPortlet ,它开始处理 请求。 下面的列表描述了完整的处理一个请求去 如果由一个通过 DispatcherPortlet :

1. 地区返回的 PortletRequest.getLocale() 被绑定到 请求让过程中的元素解决地区使用 当处理请求(渲染视图、准备数据,等等)。
2. 如果一个多部分解析器是指定的,这是一个 ActionRequest ,请求是 multiparts检查,如果他们发现,那是裹着 MultipartActionRequest 为进一步 处理其他过程中的元素。 (见 SectionA 20.7,一个扇形(文件上传)supporta 为进一步的信息关于 多部分处理)。
3. 一个适当的处理程序是寻找。 如果一个处理程序 是发现,执行链相关处理程序吗 (预处理器,后处理器、控制器)将按顺序执行的 准备一个模型。
4. 如果一个模型被返回,视图呈现,使用 视图解析器已经配置了 WebApplicationContext 。 如果没有模型 返回(这可能是由于一个预处理或后处理 拦截请求,例如,出于安全原因),没有 视图呈现,因为请求可能已经 实现。

期间会抛出的异常处理的请求 得到了任何处理程序异常,解析器 宣布在 WebApplicationContext 。 使用这些异常解析器可以定义定制的行为在案例 这种例外得到抛出。

你可以定制春天的 DispatcherPortlet 通过添加上下文参数 portlet . xml 文件或 portlet init参数。 下面列出的可能性。

20 2为多 DispatcherPortlet 初始化参数

参数	解释
contextClass	类,实现了 WebApplicationContext , 这将被用来实例化所使用的上下文 这个portlet。 如果这个参数不是指定, XmlPortletApplicationContext 将 被使用。
contextConfigLocation	字符串传递到上下文实例 (指定的 contextClass) 标明上下文(s)可以被发现。 字符串是 可能被分为多个字符串(使用 逗号作为分隔符)来支持多个上下文(在 有多个上下文位置,豆类,定义了两次,最新的优先)。
名称空间	名称空间的 WebApplicationContext 。 默认为 [portlet名称]portlet 。
viewRendererUrl	URL, DispatcherPortlet 可以访问一个 实例的 ViewRendererServlet (见 SectionA 20.3, 一个了 ViewRendererServlet 一个)。

20.3一个的 ViewRendererServlet

在Portlet的呈现过程MVC有点更复杂的比 Web MVC。 为了重用所有 视图技术 从Spring Web MVC,我们必须转换 PortletRequest / PortletResponse 到 HttpServletRequest / HttpServletResponse 然后调用 渲染 方法 视图 。 要做到这一点, DispatcherPortlet 使用一个特定的servlet, 存在的目的只有一个: ViewRendererServlet 。

为了 DispatcherPortlet 渲染到 工作,你必须声明的一个实例 ViewRendererServlet 在 web . xml 文件在您的web应用程序一样 如下:

```
<servlet>
   <servlet-name>ViewRendererServlet</servlet-name>
   <servlet-class>org.springframework.web.servlet.ViewRendererServlet</servlet-class>
</servlet>

<servlet-mapping>
   <servlet-name>ViewRendererServlet</servlet-name>
   <url-pattern>/WEB-INF/servlet/view</url-pattern>
</servlet-mapping>
```

执行实际的渲染, DispatcherPortlet 执行以下操作:

1. 将 WebApplicationContext 请求 作为一个属性在相同 web应用程序上下文属性 关键, DispatcherServlet 使用。
2. 将 模型 和 视图 对象请求做出 他们可用的 ViewRendererServlet 。
3. 构造一个 PortletRequestDispatcher 和执行 一个 包括 使用 / WEB - 步/ servlet /视图 URL映射到 ViewRendererServlet 。

这个 ViewRendererServlet 才能够 调用 渲染 方法 视图 与适当的 参数。

实际的URL ViewRendererServlet 可以改变使用 DispatcherPortlet 一个年代 viewRendererUrl 配置参数。

20.4一个控制器

控制器在Portlet MVC非常类似于Web MVC 控制器,和移植代码从一个到另应 简单的。

Portlet的基础架构是MVC控制器 org.springframework.web.portlet.mvc.Controller 接口,这是下面列出的。

```
public interface Controller {

   /**
    * Process the render request and return a ModelAndView object which the
    * DispatcherPortlet will render.
    */
   ModelAndView handleRenderRequest(RenderRequest request, RenderResponse response)
      throws Exception;

   /**
    * Process the action request. There is nothing to return.
    */
   void handleActionRequest(ActionRequest request, ActionResponse response)
      throws Exception;
}
```

正如您可以看到的,该Portlet 控制器 接口需要两种方法 这两个阶段的处理portlet请求:操作请求和 渲染的请求。 在操作阶段

应该能够处理一个 操作请求,呈现阶段应该有能力处理 呈现请求并返回一个适当的模型和视图。 而 控制器 接口是相当抽象的,Spring MVC提供了几种控制器Portlet已经包含一个 很多功能你可能需要;它们中的大多数都是非常相似的 从Spring Web MVC控制器。 这个 控制器 接口只定义了 最常见的功能要求每个控制器:处理一个 操作请求,处理一个请求,并返回一个渲染模型和一个 视图。

20.4.1A 基类AbstractController 和 PortletContentGenerator

当然,仅仅一个 控制器 接口是不够的。 提供一个基本的基础设施,所有 弹簧Portlet MVC的 控制器 年代 继承 基类AbstractController ,一个类 提供访问Spring的 ApplicationContext 和控制 缓存。

20.3为多。 提供的一个特性 基类AbstractController

参数	解释
requireSession	指示是否这个 控制器 需要 会话来完成工作。 这个功能是提供给 所有的控制器。 如果一个会话不存在当 这样一个控制器接收请求,用户 通知使用 SessionRequiredException 。
synchronizeSession	使用这个如果你想处理这个 控制器是同步用户的会话。 更具体的,控制器将扩展 覆盖handleRenderRequestInternal(. .) 和 handleActionRequestInternal(. .) 方法,这将是 同步的usera年代会话如果你指定 这个变量。
renderWhenMinimized	如果你想让你的控制器实际 渲染视图当portlet是在最小化 状态,设置为true。 默认情况下,这个值设置为 假,以便portlet,在最小化状态 小姐t显示任何内容。
cacheSeconds	当你想要一个控制器来覆盖 默认的缓存过期定义portlet, 在这里指定一个正整数。 默认情况下它是 设置为 1 ,这并没有改变 默认的缓存。 设置它 0 将确保结果是从来没有缓存。

这个 requireSession 和 cacheSeconds 属性的声明 PortletContentGenerator 类,它是 超类的 基类AbstractController)但包括这里的完整性。

当使用 基类AbstractController 作为一个 你的控制器基类(不建议使用,因为 也有很多其他控制器可能已经做这份工作吗 你)你只需要覆盖要么 handleActionRequestInternal(ActionRequest, ActionResponse) 法或者handleRenderRequestInternal(RenderRequest, 由于RenderResponse) 方法(或两者),实现您的逻辑, 并返回一个ModelAndView 对象(在案件 的 handleRenderRequestInternal)。

默认的实现两个 handleActionRequestInternal(. .) 和 handleRenderRequestInternal(. .) 抛出一个 PortletException 。 这是符合 的行为, GenericPortlet 从JSR - 168规范API。 所以你只需要覆盖的方法 你的控制器是为了处理。

这是短的例子包括一个类和一个声明 在web应用程序上下文。

```
package samples;

import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;

import org.springframework.web.portlet.mvc.AbstractController;
import org.springframework.web.portlet.ModelAndView;

public class SampleController extends AbstractController {

    public ModelAndView handleRenderRequestInternal(RenderRequest request, RenderResponse response) {
        ModelAndView mav = new ModelAndView("foo");
        mav.addObject("message", "Hello World!");
        return mav;
    }
}

<bean id="sampleController" class="samples.SampleController">
    <property name="cacheSeconds" value="120"/>
</bean>
```

上述类和web应用程序中声明 上下文是所有你需要另外建立一个处理程序映射(请参阅 SectionA 20.5,一个处理程序mappingsa)得到这非常简单 控制器工作。

20.4.2A其他简单的控制器

虽然您可以扩展 基类AbstractController , Spring MVC提供了许多Portlet提供具体的实现 功能一般用于简单的MVC应用程序。

这个 ParameterizableViewController 是 基本上一样的例子,这个事实以外 你可以指定视图名称,它将返回在web 应用程序上下文(不需要硬编码视图的名称)。

这个 PortletModeNameViewController 使用 当前模式portlet的视图名称的。 所以,如果你的 portlet在视图模式(即。 PortletMode.VIEW) 然后它使用“视图”作为视图名称。

20.4.3A命令控制器

Spring MVC有完全相同的Portlet的层次 命令控制器 作为Spring Web MVC。 他们 提供一个相互交流的方式的数据对象和动态绑定 参数从 PortletRequest 到 指定的数据对象。 你的数据对象不需要 实现一个特定于框架的接口,所以你可以直接 操纵你的持久对象如果你愿望。 让我们检查一下什么 命令控制器可用,让概述你能做什么 与他们:

- AbstractCommandController ——一个命令控制器可以使用创建你自己的命令 控制器,能够将请求参数绑定到数据 对象你指定。 这个类没有提供形式 功能,它不过提供验证功能和 允许您指定在控制器本身如何处理 命令对象,一直充满了参数请求。
- AbstractFormController —— 一个抽象的控制器提供表单提交的支持。 使用 该控制器可以模型形式和填充它们使用 命令对象检索在控制器。 在一个用户已经 充满了形式, AbstractFormController 绑定的字段,验证,和手对象回 控制器采取适当的行动。 支持的功能有: 无效的表单提交(重新提交),验证,和正常的 表格工作流。 你实现的方法来确定哪些视图 用于形式表示和成功。 使用这个控制器 如果你需要形式,但不想指定什么看法你 要给用户在应用程序 上下文。
- SimpleFormController ——一个 混凝土 AbstractFormController 这 甚至提供了更多的支持在创建一个窗体,带有一个相应的命令对象。 这个 SimpleFormController 允许您指定一个 命令对象,为形式viewname viewname为页面你 要显示用户在表单提交成功了, 更多的。
- AbstractWizardFormController 一个 一个具体的 AbstractFormController 这 提供了一个向导式界面进行编辑的内容命令对象跨多个显示页面。 支持多个 用户操作:完成、取消或页面的变化,所有这些都 容易地指定请求参数 视图。

这些命令控制器是很强大,但是他们做的 需要详细了解他们的运作方式,以便使用 他们有效地。 仔细地评估整个Javadocs 层次结构,然后查看一些示例实现在你 开始使用它们。

20.4.4A PortletWrappingController

而不是开发新的控制器,它可以使用 现有的portlet和请求映射到他们从一个 DispatcherPortlet 。 使用 PortletWrappingController ,你可以 实例化一个现有的 Portlet 作为一个 控制器 如下:

```
<bean id="myPortlet" class="org.springframework.web.portlet.mvc.PortletWrappingController">
   <property name="portletClass" value="sample.MyPortlet"/>
   <property name="portletName" value="my-portlet"/>
   <property name="initParameters">
      <value>config=/WEB-INF/my-portlet-config.xml</value>
   </property>
</bean>
```

这可能非常有价值的因为你可以使用拦截器 预处理和后处理,将这些portlet请求。 因为jsr - 168不支持任何类型的过滤机制,这是 非常方便。 例如,这可以用来将Hibernate OpenSessionInViewInterceptor 在MyFaces JSF Portlet。

20.5处理程序映射

使用处理程序映射可以映射输入portlet请求 适当的处理程序。 有一些处理程序映射可以使用了 的框中,例如, PortletModeHandlerMapping ,但是让我们先 检查的一般概念 HandlerMapping 。

注意:我们是故意使用术语一个Handlera这里相反 一个Controllera。 DispatcherPortlet 设计 可以使用其他方法来处理请求只是春天的Portlet MVCa年代的控制器。 一个处理程序可以处理portlet的任何对象 请求。 控制器是一个例子的处理程序,和他们的 课程默认。 使用其他一些框架与 DispatcherPortlet ,相应的实现 的 HandlerAdapter 是所需要的。

功能的一个基本 HandlerMapping 提供的交付 的 HandlerExecutionChain ,它必须包含 处理程序匹配传入的请求,也可能包含一个 列表的处理程序拦截器应用于请求。 当一个 请求出现时, DispatcherPortlet 将手 它处理程序映射到让它检查请求和上来 用一个合适 HandlerExecutionChain 。 然后 这个 DispatcherPortlet 将执行处理程序 和拦截器链(如果有的话)。 这些概念都是准确的 同样的在Spring Web MVC。

可配置的处理程序映射的概念,可以选择 包含拦截器(之前或之后执行实际的处理程序是 执行,或两者)是极其强大的。 很多的支

持 功能可以建立一个自定义 HandlerMapping 。 想到一个自定义处理程序 映射,选择处理程序不仅基于portlet模式 请求进入,但还在一个特定状态的会话 与请求相关联。

在Spring Web MVC,处理程序映射通常基于url。 因为确实没有这种东西在一个Portlet的URL,我们必须 使用其他机制来控制映射。 两个最常见的 portlet模式和一个请求参数,但可用的东西 可以使用portlet请求在一个自定义处理程序映射。

本节的其余部分描述了三个Portlet MVC的春天 最常用的处理程序映射。 他们所有的扩展 AbstractHandlerMapping 和分享以下 属性:

- 拦截器 的列表 拦截器使用。 HandlerInterceptor 年代了 SectionA 20 5 4,一个添加 HandlerInterceptor sa 。
- defaultHandler :默认 处理程序来使用,当这个处理程序映射并不导致 匹配的处理程序。
- 秩序 :基于价值的 顺序属性(请参阅 org.springframework.core.Ordered 接口),弹簧将排序所有处理程序映射中可用 上下文和应用第一个匹配的处理程序。
- lazyInitHandlers :允许懒惰 初始化的单处理器(原型处理程序总是 懒洋洋地初始化)。 默认值是错误的。 这个属性是 直接在三个具体实现 处理程序。

20.5.1A PortletModeHandlerMapping

这是一个简单的处理程序映射,映射传入的请求 基于当前模式的portlet(如viewa,edita, 一个helpa)。 一个例子:

```
<bean class="org.springframework.web.portlet.handler.PortletModeHandlerMapping">
  <property name="portletModeMap">
    <map>
      <entry key="view" value-ref="viewHandler"/>
      <entry key="edit" value-ref="editHandler"/>
      <entry key="help" value-ref="helpHandler"/>
    </map>
  </property>
</bean>
```

20.5.2A ParameterHandlerMapping

如果我们需要浏览到多个控制器没有 更改portlet模式,最简单的方法就是用一个请求 参数,用作键来控制映射。

ParameterHandlerMapping 使用价值 一个特定的请求参数来控制映射。 默认 名称的参数 '行动' ,但可以改变 使用 ” parameterName” 财产。

bean配置这种映射的样子 这样的:

```
<bean class="org.springframework.web.portlet.handler.ParameterHandlerMapping” >
  <property name="parameterMap">
    <map>
      <entry key="add" value-ref="addItemHandler"/>
      <entry key="edit" value-ref="editItemHandler"/>
      <entry key="delete" value-ref="deleteItemHandler"/>
    </map>
  </property>
</bean>
```

20.5.3A PortletModeParameterHandlerMapping

最强大的内置处理程序映射, PortletModeParameterHandlerMapping 结合 前两个的功能,允许不同的 导航在每个portlet模式。

再一次的默认名字参数是 “行动” ,但却可以 被改变使用 parameterName 财产。

默认情况下,相同的参数值可以用于两个 不同的portlet模式。 这是如此,如果门户本身 更改portlet模式,请求将不再有效 映射。 这种行为可以改变通过设置 allowDupParameters 属性为true。 然而, 这是不推荐的。

bean配置这种映射的样子 这样的:

```
<bean class="org.springframework.web.portlet.handler.PortletModeParameterHandlerMapping">
  <property name="portletModeParameterMap">
    <map>
      <entry key="view"> <!-- 'view' portlet mode -->
        <map>
          <entry key="add" value-ref="addItemHandler"/>
          <entry key="edit" value-ref="editItemHandler"/>
          <entry key="delete" value-ref="deleteItemHandler"/>
        </map>
```

```
        </entry>
        <entry key="edit"> <!-- 'edit' portlet mode -->
          <map>
            <entry key="prefs" value-ref="prefsHandler"/>
            <entry key="resetPrefs" value-ref="resetPrefsHandler"/>
          </map>
        </entry>
      </map>
    </property>
  </bean>
```

这种映射可以链接之前 PortletModeHandlerMapping ,这样就能提供 默认值为每个模式和一个整体违约以及。

20.5.4A添加 HandlerInterceptor 年代

春天的处理程序映射机制有一个处理程序的概念 拦截器,它可以是非常有用的,当你想应用 对某些特定功能要求,例如,检查 对于一个校长。 再一次春天Portlet MVC实现这些概念 在同样的方式作为Web MVC。

拦截器位于处理程序映射必须实现 HandlerInterceptor 从 org.springframework.web.portlet 包。 只是 像servlet版本,这个接口定义了三个方法:一个 这将被称为在实际处理程序将被执行 (preHandle),一个被称为后 处理程序执行(postHandle),一个是 命名为完整的请求完成 (afterCompletion)。 这三个方法应该 提供足够的灵活性来做各种各样的前置和后 处理。

这个 preHandle 方法返回一个布尔 值。 您可以使用这个方法来打破或继续加工 执行的链。 当这个方法返回 真正的 ,处理程序执行链将继续。 当它返回 假 , DispatcherPortlet 假定拦截 本身已经照顾的请求(例如,并呈现一个 适当的视图)和不继续执行其他 拦截器和实际处理程序在执行链。

这个 postHandle 方法只是呼吁一个 RenderRequest 。 这个 preHandle 和 afterCompletion 方法被称为两一个 ActionRequest 和一个 RenderRequest 。 如果你需要 执行逻辑在这些方法只是一种类型的请求,一定 检查什么样的请求是在处理它。

20.5.5A HandlerInterceptorAdapter

与servlet包,portlet包有一个 具体实施的 HandlerInterceptor 称为 HandlerInterceptorAdapter 。 这类 空的版本的所有方法,这样您就可以继承这个 类和实施只是一个或两个方法当那就是你 需要。

20.5.6A ParameterMappingInterceptor

portlet包也有一个具体的拦截器命名 ParameterMappingInterceptor 这是为了 用于直接与 ParameterHandlerMapping 和 PortletModeParameterHandlerMapping 。 这 拦截器将导致参数,用于控制 映射是转发的 ActionRequest 到后来的 RenderRequest 。 这将帮助确保 , RenderRequest 映射到 相同的处理 ActionRequest 。 这是做的 preHandle 拦截器的方法,这样你就可以 还是修改参数值在你处理程序来改变的 RenderRequest 将映射。

注意,这个拦截器调用 setRenderParameter 在 ActionResponse ,这意味着你 不能叫 sendRedirect 在你的处理程序时 使用这个拦截器。 如果你需要做外部重定向之后 你要么需要向前映射参数或手动 写一个不同的拦截器来为你处理的。

20.6一个观点和解决他们

如前所述,春天Portlet MVC直接重用所有 视图技术从Spring Web MVC。 这不仅包括 各种 视图 实现自己, 但也 ViewResolver 实现。 要了解更多信息,请参考 ChapterA 18, 视图技术 和 SectionA 17.5,一个解决viewsa 分别。

一些物品在使用现有的 视图 和 ViewResolver 实现是值得一提:

- 大部分门户网站期望结果的呈现 portlet是一个HTML片段。 所以,事情就像JSP / JSTL,速度, FreeMarker,XSLT将非常有意义。 但这是不可能的观点 这返回其他文档类型将任何意义在一个portlet 上下文。
- 不存在所谓的HTTP重定向的 在一个portlet(sendRedirect(. .) 方法 ActionResponse 不能 被用来保持在门户)。 所以, RedirectView 和使用 “重定向: 前缀将 不 正常工作在Portlet MVC。
- 它可能会使用 “向前: “ 前缀从 在Portlet的MVC。 然而,请记住,因为你在 portlet,您不知道当前URL看起来像。 这 意味着你不能使用相对URL来访问其他资源 您的web应用程序,你将不得不使用一个绝对 url。

同样,对于JSP开发,新的春天Taglib和新 弹簧形式都在portlet视图标签的工作方式完全相同 他们工作在servlet的观点。

20.7一个多部分(文件上传)支持

春天已经内置多部分支持Portlet MVC处理文件 上传在portlet应用程序,就像Web MVC并。 设计 多部分的支持是通过可插 PortletMultipartResolver 对象,定义 在 org.springframework.web.portlet.multipart 包。 弹簧提供了一个 PortletMultipartResolver 使用 Commons FileUpload 。 如何上传文件是支持将被描述在本节的其余部分。

默认情况下,没有几部分的处理将由弹簧Portlet MVC,有些开发人员会想处理multiparts本身。 你 将会使它自己添加一个多部分解析器对吗 web应用程序的上下文。 在你已经做了, DispatcherPortlet 将检查每个请求吗 看它是否包含一个多部分。 如果没有多部分是发现,请求 如预期将继续。 然而,如果一个多部分是发现的 请求, PortletMultipartResolver 这已被宣布将使用在你的上下文。 在那之后, 在你的请求将多部分属性被当作任何其他 属性。

注意

任何配置 PortletMultipartResolver bean 必须 有以下id(或名称): “ PortletMultipartResolver ” 。 如果你已经定义了你的 PortletMultipartResolver 与任何其他的名字, 然后 DispatcherPortlet 将 不 找到你的 PortletMultipartResolver ,和 因此没有几部分的支持将会生效。

20.7.1A使用 PortletMultipartResolver

下面的例子显示了如何使用 CommonsPortletMultipartResolver :

```
<bean id="portletMultipartResolver"
      class="org.springframework.web.portlet.multipart.CommonsPortletMultipartResolver">

   <!-- one of the properties available; the maximum file size in bytes -->
   <property name="maxUploadSize" value="100000"/>
</bean>
```

当然你也需要把合适的罐子在你 类路径的多部分解析器工作。 在的情况下 CommonsMultipartResolver ,你需要使用 commons fileupload jar 。 一定要使用至少 下议院的1.1版本作为早期版本并不FileUpload 支持jsr - 168 Portlet应用程序。

现在您已经看到了如何设置Portlet MVC为处理 多部分的请求,让我们谈谈如何真正使用它。 当 DispatcherPortlet 检测到一个多部分 请求,解析器,它可以激活已被宣布在你 上下文并移交请求。 这个解析器然后做什么 包装当前 ActionRequest 在一个 MultipartActionRequest 这已经 支持多部分文件上传。 使用 MultipartActionRequest 你可以 multiparts信息包含在这个请求和 实际获得的多部分文件本身在你 控制器。

注意,你只能得到几部分的文件上传部分 的 ActionRequest ,而不是作为一个 RenderRequest 。

20.7.2A处理一个文件上传表单中

后 PortletMultipartResolver 完成 做它的工作,请求将被处理像任何其他。 使用 这个 PortletMultipartResolver ,创建 一个窗体,带有一个上传字段(参见下面的例子), 然后让春天绑定文件到您的表单(支持对象)。 到 其实让用户上传一个文件,我们必须创建一个(JSP / HTML) 形式:

```
<h1>Please upload a file</h1>
<form method="post" action="<portlet:actionURL/>" enctype="multipart/form-data">
   <input type="file" name="file"/>
   <input type="submit"/>
</form>
```

正如您可以看到的,我们已经创建了一个字段命名一个filea相匹配的 属性包含的bean byte[] 数组。 而且我们已经添加了编码属性 (enctype = "多部分/格式数据”),这是 有必要让浏览器知道如何编码多部分字段 (不要忘了这个!)。

就像任何其他财产,不是自动的 可转换为字符串或原始类型,能够把二进制 数据在你的对象,你必须注册一个自定义编辑器的 PortletRequestDataBinder 。 有几个 可用的编辑处理文件和设置上的结果 一个对象。 有一个 StringMultipartFileEditor 能够 转换文件的字符串(使用用户定义的字符集), 有一个 ByteArrayMultipartFileEditor 哪一个 转换文件到字节数组。 他们功能类似于 CustomDateEditor 。

因此,可以使用一个表单上传文件,声明 解析器,一个映射到一个控制器,将过程bean, 控制器本身。

```
<bean id="portletMultipartResolver"
      class="org.springframework.web.portlet.multipart.CommonsPortletMultipartResolver"/>

<bean class="org.springframework.web.portlet.handler.PortletModeHandlerMapping">
   <property name="portletModeMap">
      <map>
         <entry key="view" value-ref="fileUploadController"/>
      </map>
   </property>
```

```
</bean>

<bean id="fileUploadController" class="examples.FileUploadController">
   <property name="commandClass" value="examples.FileUploadBean"/>
   <property name="formView" value="fileuploadform"/>
   <property name="successView" value="confirmation"/>
</bean>
```

在那之后,创建控制器和实际的类来保存 文件属性。

```
public class FileUploadController extends SimpleFormController {

   public void onSubmitAction(ActionRequest request, ActionResponse response,
         Object command, BindException errors) throws Exception {

      // cast the bean
      FileUploadBean bean = (FileUploadBean) command;

      // let's see if there's content there
      byte[] file = bean.getFile();
      if (file == null) {
         // hmm, that's strange, the user did not upload anything
      }

      // do something with the file here
   }

   protected void initBinder(
         PortletRequest request, PortletRequestDataBinder binder) throws Exception {
      // to actually be able to convert Multipart instance to byte[]
      // we have to register a custom editor
      binder.registerCustomEditor(byte[].class, new ByteArrayMultipartFileEditor());
      // now Spring knows how to handle multipart object and convert
   }
}

public class FileUploadBean {

   private byte[] file;

   public void setFile(byte[] file) {
      this.file = file;
   }

   public byte[] getFile() {
      return file;
   }
}
```

正如您可以看到的, FileUploadBean 已经 一个属性的类型 byte[] 保存该文件。 这个 注册一个自定义编辑器控制器让春天知道如何 其实把扇形对象解析器已经发现 属性指定的bean。 在这个例子中,什么也不做 与 byte[] 财产的bean本身,但是 在实践中你可以做任何你想要的(将它保存在一个数据库, 邮寄给某人,等等)。

一个等价的示例文件绑定直 字符串类型的属性表单上的支持对象可能看起来像 这个:

```
public class FileUploadController extends SimpleFormController {

   public void onSubmitAction(ActionRequest request, ActionResponse response,
         Object command, BindException errors) throws Exception {

      // cast the bean
      FileUploadBean bean = (FileUploadBean) command;

      // let's see if there's content there
      String file = bean.getFile();
      if (file == null) {
         // hmm, that's strange, the user did not upload anything
      }

      // do something with the file here
   }

   protected void initBinder(
      PortletRequest request, PortletRequestDataBinder binder) throws Exception {

      // to actually be able to convert Multipart instance to a String
      // we have to register a custom editor
      binder.registerCustomEditor(String.class,
         new StringMultipartFileEditor());
      // now Spring knows how to handle multipart objects and convert
   }
}

public class FileUploadBean {

   private String file;
```

```
    public void setFile(String file) {
        this.file = file;
    }

    public String getFile() {
        return file;
    }
}
```

当然,最后这个例子只会让(逻辑)的感觉 上下文的上传一个纯文本文件(它不会工作在 上传的情况下一个图像文件)。

第三部分(最后)选项是一个直接绑定 MultipartFile 财产上声明 (形式支持)对象的类。 在这种情况下不需要 注册任何自定义属性编辑器因为没有类型 要执行转换。

```
public class FileUploadController extends SimpleFormController {

    public void onSubmitAction(ActionRequest request, ActionResponse response,
            Object command, BindException errors) throws Exception {

        // cast the bean
        FileUploadBean bean = (FileUploadBean) command;

        // let's see if there's content there
        MultipartFile file = bean.getFile();
        if (file == null) {
            // hmm, that's strange, the user did not upload anything
        }

        // do something with the file here
    }
}

public class FileUploadBean {

    private MultipartFile file;

    public void setFile(MultipartFile file) {
        this.file = file;
    }

    public MultipartFile getFile() {
        return file;
    }
}
```

20.8一个处理异常

就像Servlet MVC、Portlet MVC提供了 HandlerExceptionResolver 年代以缓解 痛苦的意外发生的异常而你的请求被 由处理程序处理的,匹配请求。 Portlet MVC也 提供了一个portlet特定的、具体的 SimpleMappingExceptionResolver 这使您 采取类名称的任何异常,可能会抛出并将其映射 一个视图名称。

20.9基于注解的控制器配置

Spring 2.5引入了一个基于注解的编程模型为MVC 控制器,使用等注释 @RequestMapping , @RequestParam , @ModelAttribute 等等,这个注释 支持可用于Servlet MVC和Portlet MVC。 控制器 实现在这个风格没有扩展特定的基类或实现特定的接口。 此外,他们通常没有 直接依赖于Servlet或Portlet API的,虽然他们很容易 可以访问Servlet或Portlet设施如果需要。

以下部分文档这些注释和它们是如何 最常用的在一个Portlet的环境。

20.9.1A设置调度程序注释支持

@RequestMapping 只会被处理 如果一个相应的 HandlerMapping (类型级别注释) 和/或 HandlerAdapter (在方法级别注释) 出现在调度员。 这是默认的情况下在两个 DispatcherServlet 和 DispatcherPortlet 。

然而,如果你是定义定制 HandlerMappings 或 HandlerAdapters ,那么您需要确保一个 相应的自定义 DefaultAnnotationHandlerMapping 和/或 AnnotationMethodHandlerAdapter 被定义为好 ——只要你打算使用 @RequestMapping 。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
    http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean class="org.springframework.web.portlet.mvc.annotation.DefaultAnnotationHandlerMapping"/>

  <bean class="org.springframework.web.portlet.mvc.annotation.AnnotationMethodHandlerAdapter"/>

  // ... (controller bean definitions) ...

</beans>
```

定义一个 DefaultAnnotationHandlerMapping 和/或 AnnotationMethodHandlerAdapter 明确 也是有意义的,如果你想自定义映射策略,例如。 指定一个自定义 WebBindingInitializer (见下文)。

20.9.2A定义一个控制器 controller

这个 controller 注释说明 这一个特定的类服务的角色 控制器 。 不需要任何控制器扩展基类或引用 Portlet API。 你当然还可以引用portlet特定的 功能如果您需要。

基本的目的 controller 注释是作为原型的带注释的类,指示 它的作用。 分配器将扫描如此注释的类映射 方法,检测 @RequestMapping 注释(参见下一节)。

带注释的控制器bean可以定义明确, 使用一个标准的Spring bean定义在调度员的上下文。 然而, controller 原型还 允许自动, 符合Spring 2.5的总支持 检测组件类在类路径中,自动注册的bean 定义的。

启用自动控制器这样的注释,你必须添加 组件扫描到您的配置。 这是很容易通过使用 这个 spring上下文 模式如图所示在下面 XML片段:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:p="http://www.springframework.org/schema/p"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">

  <context:component-scan base-package="org.springframework.samples.petportal.portlet"/>

  // ...

</beans>
```

20.9.3A映射请求 @RequestMapping

这个 @RequestMapping 注释用于 portlet模式映射像“视图”/“编辑”到整个类或一个特定的 处理程序方法。 典型的类型级别注释映射一个特定的模式 (或模式加上参数条件)到一个表单控制器,使用额外的 方法级注释“缩小”主要映射为特定的 portlet请求参数。

提示

@RequestMapping 在类型 水平可用于普通的实现 控制器 接口。 在这种情况下,请求处理代码将遵循 传统 处理(行动|渲染)请求 签名, 而控制器的映射将被表示通过 @RequestMapping 注释。 这种方法适用于预构建 控制器 基类,如 SimpleFormController , 太。

在接下来的讨论中,我们将关注控制器 这是基于注释处理程序方法。

以下是一个例子,一个表单控制器从 PetPortal示例应用程序使用此注释:

```
@Controller
@RequestMapping("EDIT")
@SessionAttributes("site")
public class PetSitesEditController {

  private Properties petSites;

  public void setPetSites(Properties petSites) {
    this.petSites = petSites;
  }

  @ModelAttribute("petSites")
  public Properties getPetSites() {
    return this.petSites;
```

```
    }

    @RequestMapping  // default (action=list)
    public String showPetSites() {
        return "petSitesEdit";
    }

    @RequestMapping(params = "action=add")  // render phase
    public String showSiteForm(Model model) {
        // Used for the initial form as well as for redisplaying with errors.
        if (!model.containsAttribute("site")) {
            model.addAttribute("site", new PetSite());
        }
        return "petSitesAdd";
    }

    @RequestMapping(params = "action=add")  // action phase
    public void populateSite(
            @ModelAttribute("site") PetSite petSite, BindingResult result,
            SessionStatus status, ActionResponse response) {

        new PetSiteValidator().validate(petSite, result);
        if (!result.hasErrors()) {
            this.petSites.put(petSite.getName(), petSite.getUrl());
            status.setComplete();
            response.setRenderParameter("action", "list");
        }
    }

    @RequestMapping(params = "action=delete")
    public void removeSite(@RequestParam("site") String site, ActionResponse response) {
        this.petSites.remove(site);
        response.setRenderParameter("action", "list");
    }
}
```

20.9.4A支持处理程序方法参数

处理程序方法进行注释 @RequestMapping 可以很灵活吗 签名。 他们可能有以下类型的参数,在任意的 订单(除了验证结果,需要遵循正确的后 相应的命令对象,如果需要):

- 请求和响应对象(Portlet API)。 你可以选择任何特定的请求/响应类型,例如。PortletRequest / ActionRequest / RenderRequest。 一个显式声明行动/渲染 理由也用于映射到一个特定的请求处理程序的类型 方法(如果没有其他信息鉴于区分 行动之间,呈现请求)。
- 会话对象(Portlet API):PortletSession类型的。 一个论点 这种类型将执行相应的会话的存在。 因此,这样的一个论点将永远不会 空 。
- org.springframework.web.context.request.WebRequest 或 org.springframework.web.context.request.NativeWebRequest 。 允许通用请求参数访问以及请求/会话 属性访问,没有联系到本地Servlet / Portlet API。
- java util地区 当前请求 语言环境(门户地区在一个Portlet环境)。
- java io inputstream / java io读者 访问请求的内容。 这将是原始InputStream /读者公开的Portlet API。
- java io outputstream / java io作家 生成响应内容。 这是原始的OutputStream /作家作为公开的Portlet API。
- @RequestParam 注释参数 获取特定的Portlet请求参数。 参数值 可以转化为方法参数类型声明。
- java util地图 / org.springframework.ui.Model / org.springframework.ui.ModelMap 对于 丰富的隐式模型,将暴露在网络视图。
- 命令/表单对象绑定参数:作为bean 属性或字段,可定制的类型转换,视 在 @InitBinder 方法和/或 HandlerAdapter配置——看到 “ WebBindingInitializer ” 属性 AnnotationMethodHandlerAdapter 。 这样 命令对象以及它们的验证结果将 公开为模型属性,默认情况下使用对确立 命令类名称的属性表示法(比如。 “orderAddress” 式” mypackage.OrderAddress”)。 指定一个参数水平 ModelAttribute 注释来声明一个 具体的模型属性名称。
- org.springframework.validation.Errors / org.springframework.validation.BindingResult 验证结果为前面的命令/表单对象(前面的论点)立即。
- org.springframework.web.bind.support.SessionStatus 状态处理标记形式处理完成(触发 清理的会话属性指定的 @SessionAttributes 注释在 处理器类型级别)。

下面的返回类型都支持处理方法:

- 一个 ModelAndView 对象,与模型隐式 富含命令对象和结果的 @ModelAttribute 带注释的参考数据访问方法。
- 一个 模型 对象,与视图的名字隐式 决定通过一个 RequestToViewNameTranslator 和模型隐式富含命令对象和结果的 @ModelAttribute 带注释的参考数据访问方法。
- 一个 地图 对象暴露模型,视图的名称 含蓄地决定通过一个 RequestToViewNameTranslator 和模型隐式富含命令对象和

结果的 @ModelAttribute 带注释的参考数据访问方法。

- 一个 视图 对象,与模型隐式 确定通过命令对象和 @ModelAttribute 带注释的参考数据访问方法。 处理程序方法也可以以编程方式丰富模型通过宣布 模型 参数(见上图)。
- 一个 字符串 价值理解为视图名称, 与模型隐式确定通过命令对象和 @ModelAttribute 带注释的参考数据访问方法。 处理程序方法也可以通过编程的方式丰富模型通过宣布 模型 参数(见上图)。
- 无效 如果该方法处理响应本身 (如通过写响应内容直接)。
- 其他任何返回类型将被视为一个单一的模型属性 会接触到视图,使用指定的属性名称通过 @ModelAttribute 在方法级别(或默认 属性名称基于返回类型的类名否则)。 该模型 将隐式地富含命令对象和结果的 @ModelAttribute 带注释的参考数据访问方法。

20.9.5A绑定请求参数方法参数 @RequestParam

这个 @RequestParam 注释用于 绑定请求参数方法参数在你的控制器。

下面的代码片段从PetPortal示例应用程序 显示的用法:

```
@Controller
@RequestMapping("EDIT")
@SessionAttributes("site")
public class PetSitesEditController {

    // ...

    public void removeSite(@RequestParam("site") String site, ActionResponse response) {
        this.petSites.remove(site);
        response.setRenderParameter("action", "list");
    }

    // ...
}
```

使用该注释参数默认是必需的,但你 可以指定一个参数是可选的设置吗 @RequestParam ' s 需要 属性 假 (例如, @RequestParam(value = " id ",要求= false))。

20.9.6A提供一个链接的数据模型 @ModelAttribute

@ModelAttribute 有两个使用场景吗 控制器。 当放置在方法参数, @ModelAttribute 是用来映射模型属性 到具体,带注释的方法的参数(请参阅 populateSite() 下面的方法)。 这是如何 控制器将获取一个对象的引用控股在输入的数据 表单。 此外,该参数可以被声明为特定的 类型的表单支持对象而不是作为一个通用的 java . lang . object ,从而增加类型 安全。

@ModelAttribute 也用在方法 水平提供 参考数据 对模型(见 这个 getPetSites() 下面的方法)。 对于这种用法 方法签名可以包含相同的类型作为记录上面 这个 @RequestMapping 注释。

注意: @ModelAttribute 带注释的方法将被执行 之前 这个 选择 @RequestMapping 注释处理程序方法。 他们有效地查找与特定属性的隐式模型, 经常从数据库加载。 这样一个属性可以已经 通过 @ModelAttribute 注释 处理程序方法参数的选择处理程序方法,潜在的 与绑定和验证应用于它。

下面的代码片段显示了这两种用法的 注释:

```
@Controller
@RequestMapping("EDIT")
@SessionAttributes("site")
public class PetSitesEditController {

    // ...

    @ModelAttribute("petSites")
    public Properties getPetSites() {
        return this.petSites;
    }

    @RequestMapping(params = "action=add")  // action phase
    public void populateSite(
            @ModelAttribute("site") PetSite petSite, BindingResult result,
            SessionStatus status, ActionResponse response) {

        new PetSiteValidator().validate(petSite, result);
        if (!result.hasErrors()) {
            this.petSites.put(petSite.getName(), petSite.getUrl());
            status.setComplete();
            response.setRenderParameter("action", "list");
        }
    }
```

```
}
```

20.9.7A指定属性存储在会话 @SessionAttributes

这个类型水平 @SessionAttributes 注释声明使用会话属性通过一个特定处理器。 这通常会列出名字的模型属性或类型的 模型属性应 透明地存储在会话或一些会话存储, 作为后续请求之间的形式支持bean。

下面的代码片段显示了使用这个 注释:

```
@Controller
@RequestMapping("EDIT")
@SessionAttributes("site")
public class PetSitesEditController {
  // ...
}
```

20.9.8A定制 WebDataBinder 初始化

定制绑定请求参数与PropertyEditors等等。 通过弹簧的 WebDataBinder ,你可以使用 @InitBinder 带注释的方法在你的 控制器或外部化配置提供一个自定义的 WebBindingInitializer 。

定制数据绑定和 @InitBinder

注释控制器方法 @InitBinder 允许您配置网络 数据绑定直接在你的控制器类。 @InitBinder 识别方法 初始化 WebDataBinder 将使用 填充命令和表单对象参数的注释处理程序 方法。

这样的init粘合剂的方法支持所有参数 @RequestMapping 支持,除了 命令/表单对象和相应的验证结果对象。 init粘合剂的方法必须没有返回值。 因此,他们是 通常声明为 无效 。 典型参数包括 WebDataBinder 结合 WebRequest 或 java util地区 ,允许代码登记 上下文相关的编辑。

下面的例子演示了如何使用 @InitBinder 配置一个 CustomDateEditor 对于所有 java util日期 表单属性。

```
@Controller
public class MyFormController {

    @InitBinder
    public void initBinder(WebDataBinder binder) {
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        dateFormat.setLenient(false);
        binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, false));
    }

    // ...
}
```

配置一个自定义 WebBindingInitializer

客观数据绑定的初始化,你可以提供一个 定制实现的 WebBindingInitializer 接口,它 然后启用通过提供一个自定义的bean配置一个 AnnotationMethodHandlerAdapter ,因此覆盖 默认的配置。

20.10一个Portlet应用程序的部署

这个过程的部署Portlet应用程序是没有春天的MVC 不同于部署任何jsr - 168 Portlet应用程序。 然而,这 面积是一般足以让人迷惑了,值得在这里讨论 短暂的。

一般来说,门户/ portlet容器运行在一个web应用程序在您的 servlet容器和portlet运行在另一个应用在你的 servlet容器。 为了使portlet容器webapp使 调用portlet webapp它必须使交叉上下文调用 著名的servlet,它提供了访问portlet服务定义 在你的 portlet . xml 文件。

jsr - 168规范没有指定具体如何应该 发生,所以每个portlet容器有自己的机制,这,这 通常涉及某种部署processa让修改 portlet web应用程序本身,然后注册portlet在 portlet容器。

最少 web . xml 文件在您的portlet 网络应用正在修改注入著名的servlet,portlet 容器将调用。 在某些情况下一个servlet将服务所有 portlet的web应用程序,在其他情况下将会有一个实例 servlet将为每个portlet。

一些portlet容器也将注入库和/或 配置文件到web应用程序一样。 portlet容器必须 也使其实现Portlet的JSP标记库可用 你的网络应用程序。

底线是,重要的是要理解的 部署需要你的目标门户和确保他们得到满足 (通常由以下自动化部署过程它提供)。 一定要仔细审查文档从门户对于这个 过程。

一旦你已经部署portlet,评审结果 web . xml 文件为理智。 一些老的门户网站有 众所周知,腐败的定义 ViewRendererServlet ,从而打破了渲染 您的portlet。

PartA六世一个集成

这部分的参考文档涵盖了春天 框架的集成与一些Java EE(及相关) 技术。

- ChapterA 21日 使用Spring Remoting和web服务
- ChapterA 22, Enterprise JavaBeans(EJB)集成
- ChapterA 23, JMS(Java消息服务)
- ChapterA 24, JMX
- ChapterA 25, JCA CCI
- ChapterA 26日 邮件
- ChapterA 27日 任务执行和调度
- ChapterA 28日 动态语言支持
- ChapterA 29日 缓存抽象

21。 一个使用Spring Remoting和web服务

21.1一个介绍

弹簧特性集成类远程支持使用 各种技术。 远程支持简化了开发 远程启用服务,实现你的平常(弹簧)pojo。 目前,Spring支持以下远程技术:

- 远程方法调用(RMI) 。 通过 使用 RmiProxyFactoryBean 和 RmiServiceExporter 弹簧同时支持 传统的RMI(java rmi远程 接口和 java rmi remoteexception)和 透明的远程调用器通过RMI(与任何Java 接口)。
- Spring的HTTP调用程序 。 弹簧提供了一个 特殊的远程战略,允许Java序列化通过 HTTP,支持任何Java接口(就像RMI调用程序)。 这个 相应支持类 HttpInvokerProxyFactoryBean 和 HttpInvokerServiceExporter 。
- 黑森 。 通过使用Spring的 HessianProxyFactoryBean 和 HessianServiceExporter 你可以透明地 暴露你的服务使用轻量级二进制基于http的 Caucho提供的协议。
- 麻袋 。 麻袋是Caucho的基于xml的 替代黑森。 Spring提供了支持类等 BurlapProxyFactoryBean 和 BurlapServiceExporter 。
- jax - rpc 。 弹簧提供远程支持 web服务通过jax - rpc(J2EE 1.4的web服务API)。
- jax - ws 。 弹簧提供远程支持 通过jax - ws web服务(继任者的jax - rpc,介绍 在Java EE 5和Java 6)。
- jms 。 Remoting使用JMS作为底层 协议是支持通过 JmsInvokerServiceExporter 和 JmsInvokerProxyFactoryBean 类。

在讨论远程处理功能的春天,我们将使用 以下领域模型和相应的服务:

```
public class Account implements Serializable{

    private String name;

    public String getName(){
        return name;
    }

    public void setName(String name) {
      this.name = name;
    }
}
```

```
public interface AccountService {

    public void insertAccount(Account account);

    public List<Account> getAccounts(String name);
}
```

```
public interface RemoteAccountService extends Remote {

    public void insertAccount(Account account) throws RemoteException;

    public List<Account> getAccounts(String name) throws RemoteException;
}
```

```
// the implementation doing nothing at the moment
public class AccountServiceImpl implements AccountService {

    public void insertAccount(Account acc) {
        // do something...
    }

    public List<Account> getAccounts(String name) {
        // do something...
    }
}
```

我们将开始公开服务到一个远程客户端通过使用RMI 和谈论一些缺点的使用RMI。 然后我们会继续 显示一个示例使用麻绳作为协议。

21.2一个公开服务使用RMI

使用Spring的支持RMI,可以透明地公开 服务通过RMI的基础设施。 经过这个设置,你 主要有一个配置远程ejb类似,除了事实 没有标准支持安全上下文传播或 远程事务传播。 春天确实提供挂钩等 额外的调用上下文当使用RMI调用程序,因此您可以为 例插入安全框架或自定义的安全凭据 在这里。

21.2.1A出口服务使用 RmiServiceExporter

使用 RmiServiceExporter ,我们可以公开 接口的对象作为RMI对象AccountService。 接口 可以通过使用吗 RmiProxyFactoryBean ,或 通过纯RMI对于传统RMI服务。 这个 RmiServiceExporter 明确支持 揭露任何非RMI服务通过RMI调用。

当然,我们首先必须建立我们的服务在春季 容器:

```
<bean id="accountService" class="example.AccountServiceImpl">
    <!-- any additional properties, maybe a DAO? -->
</bean>
```

接下来,我们将不得不公开我们的服务使用 RmiServiceExporter :

```
<bean class="org.springframework.remoting.rmi.RmiServiceExporter">
    <!-- does not necessarily have to be the same name as the bean to be exported -->
    <property name="serviceName" value="AccountService"/>
    <property name="service" ref="accountService"/>
    <property name="serviceInterface" value="example.AccountService"/>
    <!-- defaults to 1099 -->
    <property name="registryPort" value="1199"/>
</bean>
```

正如您可以看到的,我们覆盖港口RMI注册表。 通常,应用程序服务器还维护一个RMI注册表和它是 明智的,不会干扰。 此外,服务的名字是 用于绑定服务在。 所以现在,该服务将被绑定 在 “rmi:/ /主机:1199 / AccountService” 。 我们将使用 URL之后要链接的服务在客户端。

注意

这个 servicePort 财产被省略了(它 默认值为0)。这意味着,一个匿名的端口将被用于 沟通与服务。

21.2.2A链接在服务客户端

我们的客户是一个简单的对象使用 AccountService 管理账户:

```
public class SimpleObject {

    private AccountService accountService;

    public void setAccountService(AccountService accountService) {
        this.accountService = accountService;
    }

    // additional methods using the accountService
```

```
}
```

链接在服务的客户,我们将创建一个单独的 Spring容器,包含简单的对象和服务链接 配置位:

```
<bean class="example.SimpleObject">
  <property name="accountService" ref="accountService"/>
</bean>

<bean id="accountService" class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
  <property name="serviceUrl" value="rmi://HOST:1199/AccountService"/>
  <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

这是我们需要做的,支持远程帐户服务 客户端。 春天将透明地创建一个调用程序和远程 启用帐户服务通过 RmiServiceExporter 。 在客户端我们链接 它在使用 RmiProxyFactoryBean 。

21.3一个使用麻绳或麻袋远程调用服务通过HTTP

提供了一个基于http的黑森二进制远程协议。 这是 Caucho开发和更多信息本身可以发现麻绳 在 http://www.caucho.com 。

21.3.1A连接的 DispatcherServlet 对于 黑森和co。

黑森沟通通过HTTP和是使用一个自定义的servlet。 使用Spring的 DispatcherServlet 的原则, 知道从Spring Web MVC用法, 你可以很容易地连接这样一个servlet 暴露你的服务。 首先我们要创建一个新的servlet在你 应用程序(这是摘录 “web . xml”):

```
<servlet>
  <servlet-name>remoting</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>remoting</servlet-name>
  <url-pattern>/remoting/*</url-pattern>
</servlet-mapping>
```

你可能熟悉Spring的 DispatcherServlet 原则和如果是这样,你知道的 那现在你要创建一个Spring容器配置资源 命名 “远程处理servlet xml” (这个名字后的 你的servlet) ' - inf’ 目录。 应用程序上下文将用于未来 部分。

或者,考虑使用Spring的简单 HttpRequestHandlerServlet 。 这允许您 嵌入远程出口国定义应用程序上下文根 (默认情况下在 “- inf /中”), 与单个servlet定义指向特定出口国豆子。 每个servlet名称必须匹配它的bean名称目标出口国 这种情况下。

21.3.2A暴露您的bean使用 HessianServiceExporter

在新创建的应用程序上下文称为 remoting servlet xml ,我们将创建一个 HessianServiceExporter 导出你的 服务:

```
<bean id="accountService" class="example.AccountServiceImpl">
  <!-- any additional properties, maybe a DAO? -->
</bean>

<bean name="/AccountService" class="org.springframework.remoting.caucho.HessianServiceExporter">
  <property name="service" ref="accountService"/>
  <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

现在我们准备环节的服务客户。 没有明确的 处理程序映射是指定的,请求url映射到服务,所以 BeanNameUrlHandlerMapping 将被使用:因此, 该服务将被导出的URL表示通过bean的名字 在包含 DispatcherServlet ' s映射 (如上所述): “http://HOST:8080 /远程/ AccountService” 。

另外,创建一个 HessianServiceExporter 在你的根应用 上下文(如在 “- inf /中”):

```
<bean name="accountExporter" class="org.springframework.remoting.caucho.HessianServiceExporter">
  <property name="service" ref="accountService"/>
  <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

在后一种情况下,定义相应的servlet对于这个 出口国在 “web . xml” ,相同的最终结果: 出口商得到映射到请求路径 /远程/ AccountService 。 注意,该servlet名称 需要匹配的bean名称目标出口国。

```
<servlet>
   <servlet-name>accountExporter</servlet-name>
   <servlet-class>org.springframework.web.context.support.HttpRequestHandlerServlet</servlet-class>
</servlet>

<servlet-mapping>
   <servlet-name>accountExporter</servlet-name>
   <url-pattern>/remoting/AccountService</url-pattern>
</servlet-mapping>
```

21.3.3A链接服务的客户端

使用 HessianProxyFactoryBean 我们可以 链接的服务客户。 同样的原理与应用 RMI的例子。 我们将创建一个单独的bean工厂或应用程序上下文 和提到下面的豆类, SimpleObject 是使用 AccountService 管理账户:

```
<bean class="example.SimpleObject">
   <property name="accountService" ref="accountService"/>
</bean>

<bean id="accountService" class="org.springframework.remoting.caucho.HessianProxyFactoryBean">
   <property name="serviceUrl" value="http://remotehost:8080/remoting/AccountService"/>
   <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

21.3.4A使用麻袋

我们不会讨论麻袋,基于xml的粗麻布,相当于在 细节在这里,因为它是配置,建立以完全相同的方式 随着黑森变体上面解释。 只是替换词 黑森 与 麻袋 和你所有 组去。

21.3.5A应用HTTP基本身份验证服务暴露通过 黑森或麻袋

的优点之一是,我们和麻袋黑森可以轻松 应用HTTP基本身份验证,因为这两个协议都是基于HTTP的。 你的正常的HTTP服务器安全机制可以很容易地应用通过 使用 web . xml 安全特性,例如。 通常,你不使用用户的安全凭据在这里,而是 定义共享凭证 黑森/ BurlapProxyFactoryBean 水平(类似于一个 JDBC 数据源)。

```
<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping">
   <property name="interceptors" ref="authorizationInterceptor"/>
</bean>

<bean id="authorizationInterceptor"
     class="org.springframework.web.servlet.handler.UserRoleAuthorizationInterceptor">
   <property name="authorizedRoles" value="administrator,operator"/>
</bean>
```

这是一个例子,我们明确提及 BeanNameUrlHandlerMapping 并设置一个拦截器 只允许管理员和操作员叫豆子提到 在这个应用程序上下文。

注意

当然,这个例子不显示一个灵活的类型的安全 基础设施。 更多选项至于安全而言,有 看看春季安全项目 http://static.springsource.org/spring-security/site/ 。

21.4一个公开服务使用HTTP调用器

相对于麻袋、麻绳,这都是轻量级的 协议使用他们自己的苗条的序列化机制,春天HTTP 调用器使用标准的Java序列化机制公开服务 通过HTTP。 这有一个巨大的优势,如果你的参数和返回类型 是复杂的类型,不能被序列化使用序列化 机制和麻袋用麻绳(参见下一节 考虑当选择远程技术)。

下罩,弹簧使用要么标准提供的设施 通过执行HTTP调用或J2SE下议院 HttpClient 。 使用后者如果你需要更多 先进的和易于使用的功能。 指 jakarta.apache.org/commons/httpclient 为更多的信息。

21.4.1A公开服务对象

设置HTTP调用程序的基础架构服务对象 就像你会做仔细地用麻绳或麻袋相同。 正如黑森支持提供了 HessianServiceExporter ,春天的HttpInvoker 支持提供了 org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter 。

揭露了 AccountService (上面提到的) 在Spring Web MVC DispatcherServlet , 以下配置需要到位的调度员的 应用程序上下

文:

```
<bean name="/AccountService" class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
  <property name="service" ref="accountService"/>
  <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

这样一个出口国定义将暴露通过 DispatcherServlet 的标准映射的设施, 解释部分在黑森。

另外,创建一个 HttpInvokerServiceExporter 在你的根 应用程序上下文(例如在 “- inf /中”):

```
<bean name="accountExporter" class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
  <property name="service" ref="accountService"/>
  <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

此外,定义相应的servlet这出口国 “web . xml” ,servlet名称匹配的bean 目标名称出口国:

```
<servlet>
  <servlet-name>accountExporter</servlet-name>
  <servlet-class>org.springframework.web.context.support.HttpRequestHandlerServlet</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>accountExporter</servlet-name>
  <url-pattern>/remoting/AccountService</url-pattern>
</servlet-mapping>
```

如果您正在运行一个servlet容器之外,正在使用 Sun的Java 6,那么您可以使用内置的HTTP服务器的实现。 您可以配置 SimpleHttpServerFactoryBean 连同 SimpleHttpInvokerServiceExporter 如图所示的例子:

```
<bean name="accountExporter"
  class="org.springframework.remoting.httpinvoker.SimpleHttpInvokerServiceExporter">
  <property name="service" ref="accountService"/>
  <property name="serviceInterface" value="example.AccountService"/>
</bean>

<bean id="httpServer"
     class="org.springframework.remoting.support.SimpleHttpServerFactoryBean">
  <property name="contexts">
    <util:map>
      <entry key="/remoting/AccountService" value-ref="accountExporter"/>
    </util:map>
  </property>
  <property name="port" value="8080" />
</bean>
```

21.4.2A链接在服务客户端

再次,链接在服务从客户的十分相似 你会做它当使用麻绳或麻袋。 使用代理,春天 能把你的调用HTTP POST请求的URL 指向出口服务。

```
<bean id="httpInvokerProxy" class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
  <property name="serviceUrl" value="http://remotehost:8080/remoting/AccountService"/>
  <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

正如前面提到过的,你可以选择你想要什么HTTP客户端 使用。 默认情况下, HttpInvokerProxy 使用 J2SE HTTP功能,但您还可以使用共享 HttpClient 通过设置 httpInvokerRequestExecutor 属性:

```
<property name="httpInvokerRequestExecutor">
  <bean class="org.springframework.remoting.httpinvoker.CommonsHttpInvokerRequestExecutor"/>
</property>
```

21.5 Web服务

Spring提供了完全支持标准的Java web服务 api:

- 公开web服务使用jax - rpc
- 访问web服务使用jax - rpc
- 使用jax - ws web服务公开
- 访问web服务使用jax - ws

注意

为什么两个标准Java web服务api？

jax - rpc 1.1是标准的web服务API在J2EE 1.4。 作为其 的名字所表明的,它侧重于对RPC绑定,这成为越来越 不受欢迎的在过去几年。 因此,它一直 jax - ws 2.0取代在Java EE 5、更灵活的条款 绑定也被大量注释。 jax - ws 2.1也 包括在Java 6(或更具体地说,在Sun的JDK 1 6 0 04和 以上;先前Sun JDK 1.6.0版本包含jax - ws 2.0)、集成 JDK的内置的HTTP服务器。

春天可以处理两个标准Java web服务api。 在Java Java EE 5 / 6,显而易见的选择是jax - ws。 在J2EE 1.4环境 ,运行在Java 5中,您可能会选择插入一个jax - ws 提供者;检查你的Java EE服务器的文档。

除了支持jax - rpc和jax - ws股票在春天的核心, 弹簧组合也特性 Spring Web 服务 ,一个解决方案,文档驱动的web,契约优先的服务——强烈推荐给建筑现代化、面向未来的web 服务。

21.5.1A基于servlet的web服务公开使用jax - rpc

Spring提供了一个方便的基类jax - rpc servlet 端点实现—— ServletEndpointSupport 。 暴露我们的 AccountService 我们把春天的 ServletEndpointSupport 类和实现我们的 这里的业务逻辑,通常委托调用业务 层。

```
/**
 * JAX-RPC compliant RemoteAccountService implementation that simply delegates
 * to the AccountService implementation in the root web application context.
 *
 * This wrapper class is necessary because JAX-RPC requires working with dedicated
 * endpoint classes. If an existing service needs to be exported, a wrapper that
 * extends ServletEndpointSupport for simple application context access is
 * the simplest JAX-RPC compliant way.
 *
 * This is the class registered with the server-side JAX-RPC implementation.
 * In the case of Axis, this happens in "server-config.wsdd" respectively via
 * deployment calls. The web service engine manages the lifecycle of instances
 * of this class: A Spring application context can just be accessed here.
 */import org.springframework.remoting.jaxrpc.ServletEndpointSupport;

public class AccountServiceEndpoint extends ServletEndpointSupport implements RemoteAccountService {

    private AccountService biz;

    protected void onInit() {
        this.biz = (AccountService) getWebApplicationContext().getBean("accountService");
    }

    public void insertAccount(Account acc) throws RemoteException {
        biz.insertAccount(acc);
    }

    public Account[] getAccounts(String name) throws RemoteException {
        return biz.getAccounts(name);
    }
}
```

我们 AccountServletEndpoint 需要运行在 相同的web应用程序作为Spring上下文允许访问 春天的设施。 以防轴,复制 AxisServlet 定义成你的 “web . xml” ,并设置端点在 “server-config.wsdd” (或使用部署工具)。 看到 样例应用程序 JPetStore那里 OrderService 是作为web服务公开吗 使用轴。

21.5.2A访问web服务使用jax - rpc

Spring提供了两个工厂bean创建jax - rpc的web服务 代理,即 LocalJaxRpcServiceFactoryBean 和 JaxRpcPortProxyFactoryBean 。 前者只能 返回一个jax - rpc服务类同我们合作。 后者是 成熟的版本,可以返回一个代理,它实现了我们的 业务服务接口。 在这个示例中,我们使用后者创建 一个代理的 AccountService 端点 我们暴露在前面的部分中。 你会发现春天已经大 支持web服务需要小编码努力——大部分的 设置在Spring配置文件像往常一样:

```
<bean id="accountWebService" class="org.springframework.remoting.jaxrpc.JaxRpcPortProxyFactoryBean">
    <property name="serviceInterface" value="example.RemoteAccountService"/>
    <property name="wsdlDocumentUrl" value="http://localhost:8080/account/services/accountService?WSDL"/>
    <property name="namespaceUri" value="http://localhost:8080/account/services/accountService"/>
    <property name="serviceName" value="AccountService"/>
    <property name="portName" value="AccountPort"/>
</bean>
```

在 serviceInterface 是我们远程业务 接口的客户将使用。 wsdlDocumentUrl 是 WSDL文件的URL。 弹簧需要这个在启动时创建 jax - rpc服务。 namespaceUri 对应 targetNamespace在。 wsdl文件。 名 对应的服务名称。 wsdl文件。 portName

对应的端口名. wsdl 文件。

访问web服务是现在很容易当我们有一个bean 工厂,将公开为 RemoteAccountService 接口。 我们可以连接这个了 在春天:

```
<bean id="client" class="example.AccountClientImpl">
  ...
  <property name="service" ref="accountWebService"/>
</bean>
```

从客户端代码我们可以访问web服务一样 是一个普通的类,除了它抛出了吗 RemoteException 。

```
public class AccountClientImpl {

  private RemoteAccountService service;

  public void setService(RemoteAccountService service) {
    this.service = service;
  }

  public void foo() {
    try {
      service.insertAccount(...);
    }
    catch (RemoteException ex) {
      // ouch
    }
  }
}
```

我们可以摆脱托运 RemoteException 因为Spring支持 自动转换为其对应的无节制的 RemoteException 。 这就要求我们 提供一个非rmi接口也。 我们的配置是现在:

```
<bean id="accountWebService" class="org.springframework.remoting.jaxrpc.JaxRpcPortProxyFactoryBean">
  <property name="serviceInterface" value="example.AccountService"/>
  <property name="portInterface" value="example.RemoteAccountService"/>
  ...
</bean>
```

在 serviceInterface 改为我们的不 RMI接口。 我们的RMI接口现在使用属性定义 portInterface 。 我们的客户端代码现在可以避免处理 java rmi remoteexception :

```
public class AccountClientImpl {

  private AccountService service;

  public void setService(AccountService service) {
    this.service = service;
  }

  public void foo() {
    service.insertAccount(...);
  }
}
```

注意,你也可以把“portInterface”部分并指定一个 普通业务接口为“serviceInterface”。 在这种情况下, JaxRpcPortProxyFactoryBean 将自动 切换到jax - rpc“动态调用接口”,执行动态 没有一个固定的端口调用存根。 的优点是,你不 甚至需要一个Java rmi兼容的接口(例如,在港口周围 情况下的非目标web服务),所有你需要的是一个匹配的 业务接口。 看看 JaxRpcPortProxyFactoryBean 对细节的javadoc 在运行时影响。

jax - rpc映射21.5.3A Bean注册

复杂的对象转移在钢丝等 帐户 我们必须注册的bean映射的 客户端。

注意

在服务器端使用轴注册的bean映射 通常做在 “server-config.wsdd” 文件。

我们将使用轴登记在客户端bean映射。 到 这样做,我们需要注册这个bean映射以编程方式:

```
public class AxisPortProxyFactoryBean extends JaxRpcPortProxyFactoryBean {

  protected void postProcessJaxRpcService(Service service) {
    TypeMappingRegistry registry = service.getTypeMappingRegistry();
    TypeMapping mapping = registry.createTypeMapping();
    registerBeanMapping(mapping, Account.class, "Account");
```

```
        registry.register("http://schemas.xmlsoap.org/soap/encoding/", mapping);
    }

    protected void registerBeanMapping(TypeMapping mapping, Class type, String name) {
        QName qName = new QName("http://localhost:8080/account/services/accountService", name);
        mapping.register(type, qName,
                new BeanSerializerFactory(type, qName),
                new BeanDeserializerFactory(type, qName));
    }
}
```

21.5.4A注册你自己的jax - rpc处理程序

在这一节中,我们将注册我们自己的 javax.rpc.xml.handler.Handler 网络 服务代理,我们可以做定制代码在SOAP消息 送丝。这个 处理程序 是 回调接口。 有一个基类中提供了便利 jaxrpc.jar ,即 javax.rpc.xml.handler.GenericHandler 我们将 扩展:

```
public class AccountHandler extends GenericHandler {

    public QName[] getHeaders() {
        return null;
    }

    public boolean handleRequest(MessageContext context) {
        SOAPMessageContext smc = (SOAPMessageContext) context;
        SOAPMessage msg = smc.getMessage();
        try {
            SOAPEnvelope envelope = msg.getSOAPPart().getEnvelope();
            SOAPHeader header = envelope.getHeader();
            ...
        }
        catch (SOAPException ex) {
            throw new JAXRPCException(ex);
        }
        return true;
    }
}
```

现在我们需要做的就是注册我们的AccountHandler jax - rpc服务,它将调用 handleRequest(. .) 在消息被发送 在钢丝。 春天已经在这次的写作没有声明支持 注册处理程序,所以我们必须使用可编程的方法。 但是春天已经非常方便我们做这个我们可以覆盖 postProcessJaxRpcService(. .) 方法,是专为这个:

```
public class AccountHandlerJaxRpcPortProxyFactoryBean extends JaxRpcPortProxyFactoryBean {

    protected void postProcessJaxRpcService(Service service) {
        QName port = new QName(this.getNamespaceUri(), this.getPortName());
        List list = service.getHandlerRegistry().getHandlerChain(port);
        list.add(new HandlerInfo(AccountHandler.class, null, null));
        logger.info("Registered JAX-RPC AccountHandler on port " + port);
    }
}
```

最后,我们必须记得要做的是改变弹簧 我们工厂使用的配置bean:

```
<bean id="accountWebService" class="example.AccountHandlerJaxRpcPortProxyFactoryBean">
    ...
</bean>
```

21.5.5A基于servlet的web服务公开使用jax - ws

Spring提供了一个方便的基类jax - ws servlet 端点实现—— SpringBeanAutowiringSupport 。 暴露我们的 AccountService 我们把春天的 SpringBeanAutowiringSupport 类和实现 我们这里的业务逻辑,通常委托调用业务 层。 我们将简单地使用Spring 2.5的 @ autowired 注释来表达这类依赖于spring管理 豆子。

```
/**
 * JAX-WS compliant AccountService implementation that simply delegates
 * to the AccountService implementation in the root web application context.
 *
 * This wrapper class is necessary because JAX-WS requires working with dedicated
 * endpoint classes. If an existing service needs to be exported, a wrapper that
 * extends SpringBeanAutowiringSupport for simple Spring bean autowiring (through
 * the @Autowired annotation) is the simplest JAX-WS compliant way.
 *
 * This is the class registered with the server-side JAX-WS implementation.
 * In the case of a Java EE 5 server, this would simply be defined as a servlet
 * in web.xml, with the server detecting that this is a JAX-WS endpoint and reacting
 * accordingly. The servlet name usually needs to match the specified WS service name.
 *
 * The web service engine manages the lifecycle of instances of this class.
 * Spring bean references will just be wired in here.
```

```
 */
import org.springframework.web.context.support.SpringBeanAutowiringSupport;

@WebService(serviceName="AccountService")
public class AccountServiceEndpoint extends SpringBeanAutowiringSupport {

    @Autowired
    private AccountService biz;

    @WebMethod
    public void insertAccount(Account acc) {
      biz.insertAccount(acc);
    }

    @WebMethod
    public Account[] getAccounts(String name) {
      return biz.getAccounts(name);
    }
}
```

我们 AccountServletEndpoint 需要运行在 相同的web应用程序作为Spring上下文允许访问 春天的设施。 这是默认的情况下在Java EE 5 环境中,使用标准的合同为jax - ws servlet端点 部署。 看到Java EE 5 web服务教程的细节。

21.5.6A出口独立的web服务使用jax - ws

内置的jax - ws提供者是与Sun的JDK 1.6 支持web服务暴露使用内置的HTTP服务器的 包括在JDK 1.6。 春天的SimpleJaxWsServiceExporter 检测所有 webservice 带注释的豆子在Spring应用程序 背景下,出口通过默认的jax - ws服务器(JDK 1.6 HTTP服务器)。

在这个场景中,端点实例定义和管理 作为Spring bean本身;他们将注册与jax - ws 但他们的生命周期将发动机的Spring应用程序上下文。 这意味着春天的功能像显式依赖注入 可能会被应用到端点实例。 当然,注解驱动的 注入通过 @ autowired 将作为好。

```
<bean class="org.springframework.remoting.jaxws.SimpleJaxWsServiceExporter">
   <property name="baseAddress" value="http://localhost:8080/"/>
</bean>

<bean id="accountServiceEndpoint" class="example.AccountServiceEndpoint">
   ...
</bean>

...
```

这个 AccountServiceEndpoint 可能来自 春天的 SpringBeanAutowiringSupport 但不 必须自端点是一个完全spring管理bean这里。 这 意味着端点实现可能看起来像如下,没有 任何父类声明和弹簧的 @ autowired 配置注释仍然被授予:

```
@WebService(serviceName="AccountService")
public class AccountServiceEndpoint {

    @Autowired
    private AccountService biz;

    @WebMethod
    public void insertAccount(Account acc) {
      biz.insertAccount(acc);
    }

    @WebMethod
    public List<Account> getAccounts(String name) {
      return biz.getAccounts(name);
    }
}
```

21.5.7A导出web服务使用jax - ws国际扶轮的春天 支持

太阳的jax - ws RI、开发项目的一部分,船只GlassFish 弹簧支架作为它的jax - ws共享项目。 这允许 定义jax - ws端点作为spring管理bean,类似于 独立模式在前一节中讨论的,但这一次在一个 Servlet环境。 注意,这不是便携式在Java EE 5环境;主要用于非EE环境如 Tomcat,嵌入jax - ws扶轮作为web 应用程序。

不同标准的风格的出口基于servlet 端点是生命周期的端点实例本身 将管理这里的春天,会有只有一个jax - ws吗 servlet定义在web . xml 。 在标准的Java EE 5个风格(如上图),你会有一个servlet定义/ 服务端点,每个端点通常委托给弹簧 豆类(通过使用 @ autowired ,如图所示 以上)。

看看 https://jax-ws-commons.dev.java.net/spring/ 详细的设置和使用的风格。

21.5.8A访问web服务使用jax - ws

类似于jax - rpc支持,Spring提供了两个工厂 bean创建jax - ws web服务代理,即 LocalJaxWsServiceFactoryBean 和 JaxWsPortProxyFactoryBean 。 前者只能 返回一个jax - ws服务类同我们合作。 后者是 成熟的版本,可以返回一个代理,它实现了我们的 业务服务接口。 在这个示例中,我们使用后者创建 一个代理的 AccountService 端点 (再一次):

```
<bean id="accountWebService" class="org.springframework.remoting.jaxws.JaxWsPortProxyFactoryBean">
  <property name="serviceInterface" value="example.AccountService"/>
  <property name="wsdlDocumentUrl" value="http://localhost:8888/AccountServiceEndpoint?WSDL"/>
  <property name="namespaceUri" value="http://example/"/>
  <property name="serviceName" value="AccountService"/>
  <property name="portName" value="AccountServiceEndpointPort"/>
</bean>
```

在 serviceInterface 是我们的业务 接口的客户将使用。 wsdlDocumentUrl 是 WSDL文件的URL。 弹簧需要这一个启动时间来创建 jax - ws服务的。 namespaceUri 对应 targetNamespace在。 wsdl文件。 名 对应的服务名称。 wsdl文件。 portName 对应的端口名. wsdl 文件。

访问web服务是现在很容易当我们有一个bean 工厂,将公开为 AccountService 接口。 我们可以在春天连接这个:

```
<bean id="client" class="example.AccountClientImpl">
  ...
  <property name="service" ref="accountWebService"/>
</bean>
```

从客户端代码我们可以访问web服务一样 是一个正常的类:

```
public class AccountClientImpl {

  private AccountService service;

  public void setService(AccountService service) {
    this.service = service;
  }

  public void foo() {
    service.insertAccount(...);
  }
}
```

注意: 上面的是稍微简化在 jax - ws的端点接口和实现类需要来 被标注 webservice , @SOAPBinding 等注释。 这意味着你不能(容易)使用纯Java接口和实现类 jax - ws端点工件;您需要添加注释相应的第一。 检查jax - ws文档的细节要求。

21.6一个JMS

也可以透明地使用JMS作为公开的服务 底层通信协议。 JMS远程支持 Spring框架是相当基本的,它发送和接收的 同一线程 和相同 事务性 会话 ,和 像这样的吞吐量将非常依赖于实现的。 注意, 这些单线程和事务性的约束只适用于 Spring的JMS **remoting** 支持。 看到 ChapterA 23, JMS(Java消息服务) 关于春天的丰富的支持 基于jms的 **消息** 。

下面的接口是用于两个服务器和客户端 侧。

```
package com.foo;

public interface CheckingAccountService {

  public void cancelAccount(Long accountId);
}
```

以下简单的实现上述接口使用 在服务器端。

```
package com.foo;

public class SimpleCheckingAccountService implements CheckingAccountService {

  public void cancelAccount(Long accountId) {
    System.out.println("Cancelling account [" + accountId + "]");
  }
}
```

这个配置文件包含jms基础设施豆类, 都是共享的客户机和服务器。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
```

```
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="connectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL" value="tcp://ep-t43:61616"/>
  </bean>

  <bean id="queue" class="org.apache.activemq.command.ActiveMQQueue">
      <constructor-arg value="mmm"/>
  </bean>

</beans>
```

21.6.1A服务器端配置

在服务器上,你只需要公开服务对象使用 这个 JmsInvokerServiceExporter 。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="checkingAccountService"
      class="org.springframework.jms.remoting.JmsInvokerServiceExporter">
    <property name="serviceInterface" value="com.foo.CheckingAccountService"/>
    <property name="service">
      <bean class="com.foo.SimpleCheckingAccountService"/>
    </property>
  </bean>

  <bean class="org.springframework.jms.listener.SimpleMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="destination" ref="queue"/>
    <property name="concurrentConsumers" value="3"/>
    <property name="messageListener" ref="checkingAccountService"/>
  </bean>

</beans>
```

```
package com.foo;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Server {

  public static void main(String[] args) throws Exception {
    new ClassPathXmlApplicationContext(new String[]{"com/foo/server.xml", "com/foo/jms.xml"});
  }
}
```

21.6.2A客户端配置

客户端只需要创建一个客户端代理,将 实现约定的接口（CheckingAccountService）。 由此产生的 对象创建了后面的bean定义后可以 注入到其他客户端对象和代理会照顾 转发到服务器端对象的调用通过JMS。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="checkingAccountService"
      class="org.springframework.jms.remoting.JmsInvokerProxyFactoryBean">
    <property name="serviceInterface" value="com.foo.CheckingAccountService"/>
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="queue" ref="queue"/>
  </bean>

</beans>
```

```
package com.foo;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Client {

  public static void main(String[] args) throws Exception {
    ApplicationContext ctx = new ClassPathXmlApplicationContext(
        new String[] {"com/foo/client.xml", "com/foo/jms.xml"});
    CheckingAccountService service = (CheckingAccountService) ctx.getBean("checkingAccountService");
```

```
        service.cancelAccount(new Long(10));
    }
}
```

你也希望研究提供支持 行话 项目(引用 首页广告) 一个 ... 是一个轻量级基于POJO的远程和消息吗 图书馆基于Spring框架的远程库中 扩展以支持JMS。 一个

21.7一个自动检测是没有实现远程接口

主要的原因为什么实现接口的自动识别并 不会出现远程接口是避免打开太多的大门 远程调用。 目标对象可能实现内部回调 接口像 InitializingBean 或 DisposableBean 哪一个不希望吗 暴露于呼叫者。

提供代理与所有接口实现的目标 通常不重要在当地的情况。 但当出口远程 服务,你应该公开特定的服务接口,与特定的 操作用于远程使用。 除了内部调 接口,目标可能实现多个业务接口,与 只是其中的一个用于远程曝光。 由于这些原因,我们 需要 这样一个服务接口 指定的。

这是一个平衡配置方便和风险 的意外接触的内部方法。 总是指定一个服务 接口是不太多的精力,让你安全起见有关 控制曝光的具体方法。

21.8一个考虑当选择技术

这里给出每个技术都有它的缺点。 你 应该仔细考虑您的需求,服务中暴露和 你将要发送的对象在钢丝在选择技术。

当使用RMI,这是不可能的访问对象通过 HTTP协议,除非你穿隧RMI交通。 RMI是一个相当 协议,它支持重量级完全对象序列化的 是重要的在使用复杂的数据模型需要序列化结束了吗 这个线。 然而,rmi jrmp与Java客户:它是一个Java Java 远程解决方案。

春天的HTTP调用程序是一个不错的选择如果你需要基于HTTP的 remoting也依赖Java序列化。 它的股票基本 基础设施与RMI调用器,只是使用HTTP作为传输。 注意, HTTP调用器不仅限于java java远程也 春天在客户端和服务器端。 (后者也适用于 春天的RMI调用程序对非RMI接口)。

黑森和/或麻袋可能提供重要的价值当操作 在异构环境中,因为他们明确允许非java 客户。 然而,非java支持仍然是有限的。 已知问题包括 Hibernate对象序列化的结合 懒洋洋地初始化集合。 如果你有这样一个数据模型,考虑 使用RMI或HTTP调用器代替黑森。

JMS可以用于提供集群的服务和允许 JMS代理照顾负载平衡、发现和 自动故障转移。 默认情况下:Java序列化使用JMS时使用remoting但是JMS提供者可以使用不同的机制对铁丝 格式化,比如XStream允许服务器来实现在其他 技术。

最后但并非最不重要,EJB胜过RMI,它 支持标准的基于角色的身份验证和授权和远程 事务传播。 它可能会得到RMI调用器或HTTP 调用器来支持安全上下文传播,虽然这是 没有提供核心弹簧:只有适当的钩子堵塞 在第三方或定制的解决方案在这里。

21.9一个访问RESTful服务的客户端

这个 RestTemplate 是核心类 客户端访问RESTful服务。 它在概念上类似于 其他模板类在春天,如 JdbcTemplate 和 JmsTemplate 和其他模板类发现在其他弹簧组合项目。 RestTemplate 的行为是定制提供 回调方法和配置 HttpMessageConverter 用来元帅 对象到HTTP请求体和数据编出任何响应返回 成一个对象。 因为它是常见的使用XML作为消息格式,春天 提供了一个 MarshallingHttpMessageConverter 这 使用对象到xml框架的一部分 org.springframework.oxm 包。 这给你一个 广泛的选择的XML对象映射技术来选择 从。

本节描述如何使用 RestTemplate 和它相关的 HttpMessageConverters 。

21.9.1A RestTemplate

在Java调用RESTful服务通常是通过使用一个助手 类比如Jakarta Commons HttpClient 。 对于 常见这种方法太REST操作低层次如图所示 下面。

```
String uri = "http://example.com/hotels/1/bookings";

PostMethod post = new PostMethod(uri);
String request = // create booking request content
post.setRequestEntity(new StringRequestEntity(request));

httpClient.executeMethod(post);
```

```
if (HttpStatus.SC_CREATED == post.getStatusCode()) {
  Header location = post.getRequestHeader("Location");
  if (location != null) {
    System.out.println("Created new booking at :" + location.getValue());
  }
}
```

RestTemplate提供更高级别的方法,对应于每个 六个主要的HTTP方法,使许多RESTful服务调用一个 行程序和执行休息的最佳实践。

21.1为多。 一个概述RestTemplate方法

HTTP方法	RestTemplate 方法
删除	删除
得到	getForObject
	getForEntity
头	headForHeaders(字符串 url,Stringa¦urlVariables)
选项	optionsForAllow(字符串 url,Stringa¦urlVariables)
邮报	postForLocation(字符串 url、对象请求,Stringa¦urlVariables)
	postForObject(字符串 url、对象请求、类< T > responseType,Stringa¦ uriVariables)
把	把(字符串 url、对象请求,Stringa¦urlVariables)

的名字 RestTemplate 方法遵循一个 命名约定,第一部分显示HTTP方法是什么 调用和第二部分表明什么是返回。 例如, 方法 getForObject() 将执行一个GET,转换吗 HTTP响应为一个对象类型的选择和返回, 对象。 该方法 postForLocation() 将做一个后,将给定的对象为一个HTTP请求,并返回 响应HTTP报头位置新创建的对象可以 发现。 如果发生了异常处理HTTP请求,一个例外 类型的 RestClientException 将 扔,这种行为可以改变堵在另一个 ResponseErrorHandler 实现进 RestTemplate 。

对象传递到和这些方法返回的转换为 和从HTTP消息 HttpMessageConverter 实例。 转换器主要mime类型默认注册,但是你也可以编写自己的转换器和注册它通过吗 messageConverters() bean属性。 默认 转换器实例注册模板 ByteArrayHttpMessageConverter , StringHttpMessageConverter , FormHttpMessageConverter 和 SourceHttpMessageConverter 。 你可以覆盖 这些默认值使用 messageConverters() bean 房地产需要如果使用 MarshallingHttpMessageConverter 或 MappingJackson2HttpMessageConverter 。

每个方法使用URI模板参数的两种形式,无论是作为一个 字符串 可变长度参数或一个 Map < String,String > 。 例如,

```
String result = restTemplate.getForObject("http://example.com/hotels/{hotel}/bookings/{booking}",
                                          String.class,"42", "21");
```

使用可变长度参数和

```
Map<String, String> vars = Collections.singletonMap("hotel", "42");
String result =
  restTemplate.getForObject("http://example.com/hotels/{hotel}/rooms/{hotel}", String.class, vars);
```

使用 Map < String,String > 。

创建一个实例 RestTemplate 你可以 简单地调用默认的无参数构造函数。 这将使用标准的Java 类从 Java.net 包作为底层 实现创建HTTP请求。 这可以覆盖 指定的一个实现 ClientHttpRequestFactory 。 Spring提供了 实现 HttpComponentsClientHttpRequestFactory 使用 Apache HttpComponents HttpClient 创建请求。 HttpComponentsClientHttpRequestFactory 配置 使用的一个实例 org apache httpclient http客户端 哪一个 可以反过来被配置成凭证信息或连接吗 池功能。

提示

注意, Java.net 实现 HTTP请求可能会抛出一个异常当访问响应的状态 这是一个错误(例如,401)。 如果这是一个问题,切换到 HttpComponentsClientHttpRequestFactory 相反。

前面的示例使用Apache HttpComponents HttpClient 直接重写使用 RestTemplate 如下所示

```
uri = "http://example.com/hotels/{id}/bookings";

RestTemplate template = new RestTemplate();

Booking booking = // create booking object

URI location = template.postForLocation(uri, booking, "1");
```

一般的回调接口 RequestCallback 和时调用 执行方法被调用。

```
public <T> T execute(String url, HttpMethod method, RequestCallback requestCallback,
                     ResponseExtractor<T> responseExtractor,
                     String... urlVariables)


// also has an overload with urlVariables as a Map<String, String>.
```

这个 RequestCallback 接口是 定义为

```
public interface RequestCallback {
 void doWithRequest(ClientHttpRequest request) throws IOException;
}
```

和允许您操作请求头和写入 请求主体。 当使用execute方法你不必担心 关于任何资源管理,模板将永远关闭 请求和处理任何错误。 请查阅API文档了解更多 信息使用execute方法和其他的意义 方法参数。

使用URI

对于每个主要的HTTP方法, RestTemplate 提供了变异或者拍URI或字符串 java净uri 作为第一个参数。

字符串URI模板参数作为一个变体接受 字符串变量长度参数或作为一个 Map < String,String > 。 他们还认为URL字符串不是编码和需要编码。 例如以下:

```
restTemplate.getForObject("http://example.com/hotel list", String.class);
```

将执行一个上车的吗 http://example.com/hotel%20list 。 这意味着如果输入URL字符串已经编码的,它将被编码的两次——即。 http://example.com/hotel%20list 将成为 http://example.com/hotel%2520list 。 如果这不是预定效果,使用 java净uri 方法的变体,它假定 URL已经编码也通常有用如果你想 重用单个(完全展开) URI 多次。

这个 UriComponentsBuilder 类可以使用 建立和编码 URI 包括支持 为URI模板。 例如你可以从一个URL字符串:

```
UriComponents uriComponents =
        UriComponentsBuilder.fromUriString("http://example.com/hotels/{hotel}/bookings/{booking}").build()
            .expand("42", "21")
            .encode();

URI uri = uriComponents.toUri();
```

或指定每个URI组件分别:

```
UriComponents uriComponents =
        UriComponentsBuilder.newInstance()
            .scheme("http").host("example.com").path("/hotels/{hotel}/bookings/{booking}").build()
            .expand("42", "21")
            .encode();

URI uri = uriComponents.toUri();
```

处理请求和响应头

除了上面描述的方法, RestTemplate 也有 交换() 方法,它能 用于任意HTTP方法执行基于 HttpEntity 类。

或许最重要的是, 交换() 方法可以用来添加请求头和响应头读。 例如:

```
HttpHeaders requestHeaders = new HttpHeaders();
requestHeaders.set("MyRequestHeader", "MyValue");
HttpEntity<?> requestEntity = new HttpEntity(requestHeaders);

HttpEntity<String> response = template.exchange("http://example.com/hotels/{hotel}",
  HttpMethod.GET, requestEntity, String.class, "42");
```

```
String responseHeader = response.getHeaders().getFirst("MyResponseHeader");
String body = response.getBody();
```

在上面的示例中,我们首先准备一个请求实体包含 MyRequestHeader 头。 然后我们获得响应,和 阅读 MyResponseHeader 和身体。

21.9.2A HTTP消息转换

对象传递给和方法返回 getForObject() , postForLocation() ,和 put() 被转换成HTTP请求和来自哪里 HTTP响应的 HttpMessageConverters 。 这个 HttpMessageConverter 接口是 下面给你一个更好的感觉,它的功能

```
public interface HttpMessageConverter<T> {

    // Indicate whether the given class and media type can be read by this converter.
    boolean canRead(Class<?> clazz, MediaType mediaType);

    // Indicate whether the given class and media type can be written by this converter.
    boolean canWrite(Class<?> clazz, MediaType mediaType);

    // Return the list of MediaType objects supported by this converter.
    List<MediaType> getSupportedMediaTypes();

    // Read an object of the given type from the given input message, and returns it.
    T read(Class<T> clazz, HttpInputMessage inputMessage) throws IOException,
                                                        HttpMessageNotReadableException;

    // Write an given object to the given output message.
    void write(T t, HttpOutputMessage outputMessage) throws IOException,
                                                    HttpMessageNotWritableException;

  }
```

具体实现为主要媒体(mime)类型 中提供了框架和默认注册的 RestTemplate 在客户端和 AnnotationMethodHandlerAdapter 在 服务器端。

的实现 HttpMessageConverter 年代中描述 以下部分。 对于所有转换器一个默认的媒体类型是使用但 可以被设置吗 supportedMediaTypes bean属性

StringHttpMessageConverter

一个 HttpMessageConverter 实现,可以读取和写入字符串从HTTP请求 和响应。 默认情况下,这个转换器支持所有文本媒体类型 (text / *),并写有 内容类型 的 文本/平原 。

FormHttpMessageConverter

一个 HttpMessageConverter 实现,可以读取和写入表单数据来自HTTP请求 和响应。 默认情况下,这个转换器的读和写媒体类型 应用程序/ x-www-form-urlencoded 。 表单数据 是读和写进一个吗 MultiValueMap <字符串, 字符串> 。

ByteArrayHttpMessageConverter

一个 HttpMessageConverter 实现,可以读取和写入字节数组从HTTP 请求和响应。 默认情况下,这个转换器支持所有媒体 类型(* / *),并写有 内容类型 的 应用程序/八位元流 。 这可以覆盖 设置 supportedMediaTypes 财产, 覆盖 getContentType(byte[]) 。

MarshallingHttpMessageConverter

一个 HttpMessageConverter 实现,可以读取和写入XML使用Spring的 Marshaller 和 解组程序 抽象的 org.springframework.oxm 包。 这个转换器 需要 Marshaller 和 解组程序 才可以使用。 这些可以通过构造函数注入或bean属性。 默认情况下 这个转换器支持(text / xml)和 (application / xml)。

MappingJackson2HttpMessageConverter(或MappingJacksonHttpMessageConverter与杰克逊1. x)

一个 HttpMessageConverter 实现,可以读取和写入JSON使用杰克逊 ObjectMapper 。 JSON映射可以 根据需要定制通过使用杰克逊提供注释。 当 进一步控制是必要的,一个自定义的 ObjectMapper 可以通过注射 这个 ObjectMapper 财产的情况定制 JSON序列化器/反序列化器需要提供特定的类型。 默认情况下这个转换器支持(application / json)。

SourceHttpMessageConverter

一个 HttpMessageConverter 实现,可以读和写 javax.xml.transform.Source 从HTTP 请求和响应。 只有 DOMSource ,

SAXSource ,和 StreamSource 都受支持。 默认情况下,这个 转换器支持(text / xml)和 (application / xml)。

BufferedImageHttpMessageConverter

一个 HttpMessageConverter 实现,可以读和写 java.awt.image.BufferedImage 从HTTP 请求和响应。 这个转换器的读和写的媒体类型 支持的Java I / O API。

22。 一个Enterprise JavaBeans(EJB)集成

22.1一个介绍

作为一个轻量级容器,春天通常被认为是一个EJB 替换。 我们相信,对于许多(如果不是大多数应用程序和使用 情况下,弹簧作为一个容器,结合其丰富的支持 功能领域的事务,ORM和JDBC访问,是一个更好的 选择比实现等效的功能通过EJB容器和 ejb。

然而,重要的是要注意,使用弹簧不阻止 你使用ejb。 事实上,春天使它更容易访问ejb和 实现ejb和内部功能。 此外,使用Spring访问ejb提供的服务允许实现这些服务的 到后来透明本地EJB之间切换,远程EJB或者POJO (普通旧式Java对象)的变体,没有客户机代码必须 被改变。

在这一章,我们看看春天可以帮助你访问和 实现ejb。 Spring提供了特定的价值当访问无状态 会话bean(SLSBs),因此我们将首先讨论这个。

22.2一个ejb访问

22.2.1A概念

调用一个方法在本地或远程无状态会话bean, 客户机代码必须正常执行JNDI查找获得(本地或 远程)EJB本地对象,然后使用“创建”上的方法调用该对象 获得实际的(本地或远程)EJB对象。 一个或多个方法 然后调用EJB。

为了避免重复低级代码,许多EJB应用程序使用 服务定位器和业务代表模式。 这些都比 喷涂JNDI查找在客户端代码,但是他们平常的 实现有重大缺陷。 例如:

- 通常使用ejb代码取决于服务定位器或 业务代表单例对象,使它很难测试。
- 对于使用服务定位器模式没有 业务代表,应用程序代码仍然最终不得不调用 create()方法在EJB家里,和处理结果 例外。 因此它仍是绑定到EJB API和复杂性 EJB编程模型的。
- 实现业务代表模式通常的结果 在重要的代码重复,我们需要编写无数 方法,简单地调用相同的方法在EJB。

春天的方法是允许创建和使用代理对象, 通常配置Spring容器内,充当无编码的 业务代表。 你不需要写另一个服务定位器,另一个 JNDI查找,或重复的方法在一个手工编码的业务代表,除非 你实际上是在这样的代码添加实际价值。

22.2.2A访问当地SLSBs

假设我们有一个web控制器,需要使用一个地方 EJB。 我们会遵循最佳实践和使用EJB业务方法 接口模式,这样EJBa年代本地接口扩展了一个非 ejb特定业务方法的接口。 莉塔年代称之为业务 方法接口 MyComponent 。

```
public interface MyComponent {
    ...
}
```

的一个主要原因,使用业务接口模式的方法 是确保同步方法签名之间在本地吗 接口和bean实现类是自动的。 另一个原因是 后来,使我们能更容易地切换到一个POJO(普通的旧 Java对象)实现的服务如果这样做很有意义。 当然我们会也需要实现本地home接口和 提供一个实现类实现 SessionBean 和 MyComponent 业务方法的接口。 现在 只有Java编码我们会需要做勾搭我们的web层控制器到 EJB实现是揭露一个setter方法的类型 MyComponent 在控制器。 这将保存引用作为一个实例变量 控制器:

```
private MyComponent myComponent;

public void setMyComponent(MyComponent myComponent) {
    this.myComponent = myComponent;
}
```

我们可以随后使用该实例变量在任何业务 方法的控制器。 现在假设我们获取我们的控制器 对象从一个Spring容器,我们可以(在同一个上下文)配置 LocalStatelessSessionProxyFactoryBean 实例,它 将EJB代理对象。 代理的配置和设置的 这个

MyComponent 财产的控制器已经完成 与一个配置条目如:

```
<bean id="myComponent"
    class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">
  <property name="jndiName" value="ejb/myBean"/>
  <property name="businessInterface" value="com.mycom.MyComponent"/>
</bean>

<bean id="myController" class="com.mycom.myController">
  <property name="myComponent" ref="myComponent"/>
</bean>
```

还有年代很多工作发生在幕后,礼貌的 Spring AOP框架,虽然你舞台t被迫使用AOP 概念来享受结果。 这个 MyComponent bean 定义创建一个代理为EJB,它实现了业务 方法接口。 EJB本地缓存的家是在启动时,所以还有年代 只有一个JNDI查找。 每次调用的EJB,代理 调用 classname 方法在本地EJB和 调用相应的业务方法的EJB。

这个 myController bean定义设置 MyComponent 财产的控制器类的 EJB代理。

或者(最好的情况下许多这样的代理定义), 考虑使用 < jee:local-slsb > 配置元素在春天的“哎呀”命名空间:

```
<jee:local-slsb id="myComponent" jndi-name="ejb/myBean"
    business-interface="com.mycom.MyComponent"/>

<bean id="myController" class="com.mycom.myController">
  <property name="myComponent" ref="myComponent"/>
</bean>
```

这个EJB访问机制提供了巨大的简化 应用程序代码:web层代码(或其他EJB客户机代码)没有 依赖于使用的EJB。 如果我们想要取代这个EJB引用 使用POJO或一个mock对象或其他测试存根,我们可以简单地改变 这个 MyComponent bean定义没有丝毫改变 行Java代码。 此外,我们havenat不得不写一行 JNDI查找或其他EJB管道代码的一部分,我们的应用程序。

在真实的应用程序基准和经验表明 这种方法的性能开销(包括反射 调用目标EJB)是最小的,并且通常是无法觉察的 在典型的使用。 记住,我们小姐t想进行细粒度的要求 对EJB的不管怎样,还有年代关联的成本与EJB的基础设施 在应用程序服务器。

有一个警告关于JNDI查找。 在一个bean 容器,这类通常是最好作为一个单例(根本 没有理由让它一个原型)。 然而,如果bean容器 前实例化(如做不同的单例对象XML ApplicationContext 变体) 你可能有一个问题如果bean容器之前加载EJB 集装箱装载目标EJB。 这是因为JNDI查找将 中执行 init() 这个类的方法,然后 缓存,但是EJB不会被束缚在目标位置然而。 解决方案是不预实例化这个工厂对象,但允许它 在第一次使用创建。 在XML的容器,这是通过控制 这个 懒init 属性。

虽然这不会感兴趣的大多数的春天 用户,那些做编程AOP使用ejb可能想看看 LocalSlsbInvokerInterceptor 。

22.2.3A访问远程SLSBs

访问远程ejb本质上是相同的,访问当地 ejb,除了 SimpleRemoteStatelessSessionProxyFactoryBean 或 < jee:remote-slsb > 配置元素使用。 当然,有或没有春天,远程调用语义应用;一个 调用一个对象的方法在另一个VM在另一个电脑做 有时必须区别对待从使用场景和 故障处理。

春天的EJB客户端支持更增加了一个优势 非弹簧的方法。 通常它是有问题的对EJB客户端代码 很容易之间来回切换调用ejb本地或 远程。 这是因为远程接口方法必须声明 他们把 RemoteException ,客户端代码必须处理 用这个的,本地接口方法不。 客户机代码 写给本地ejb需要被转移到远程ejb 通常需要修改添加处理远程异常, 和客户端代码的编写远程ejb需要被转移到本地ejb,要么保持不变但做很多不必要的处理 远程异常,或需要修改删除代码。 与 春天远程EJB代理,可以不申报任何抛出 RemoteException 在你的业务方法接口和 实现EJB代码,有一个远程接口是相同的除了 它把 RemoteException 和依赖 代理动态治疗的两个接口好像他们是相同的。 那是,客户端代码不需要处理托运 RemoteException 类。 任何实际 RemoteException 这是扔在EJB 调用将被重新抛出作为非检查 RemoteAccessException 类,它是的一个子类 RuntimeException 。 目标服务就可以 之间随意切换本地EJB或远程EJB(甚至普通的Java 对象)的实现,没有客户端代码知道或关心。 的 当然,这是可选的;没有什么阻止你声明 remoteexception 在你的业务接口。

22.2.4A访问EJB 2。 x SLSBs与EJB 3 SLSBs

访问EJB 2。 x会话bean和EJB 3会话bean通过Spring 很大程度上是透明的。 春天的EJB访问器,包括 < jee:local-slsb > 和 < jee:remote-slsb > 设施、透明地适应实际的组件在运行时。 他们处理一个home接口如果发现(EJB 2。 x风格),或直接执行 如果没有主接口组件调用可用(EJB 3风格)。

注意:对于EJB 3会话bean,您可以有效地使用 JndiObjectFactoryBean / < jee:jndi查找> ,因为完全可用组件引用都暴露在平原JNDI查找那里。 定义明确的 < jee:local-slsb > / < jee:remote-slsb > 查找简单地提供 一致的和更明确的EJB访问配置。

22.3一个使用Spring的EJB实现支持类

22.3.1A EJB 2。 x基类

Spring提供了方便的类来帮助您实现ejb。 这些都是为了鼓励良好的实践把业务 ejb在pojo背后的逻辑,使得ejb事务负责 界定和(可选)远程处理。

实现一个无状态或有状态会话bean,或者一个消息驱动的 豆,你只需要得到您的实现类 AbstractStatelessSessionBean , AbstractStatefulSessionBean ,和 AbstractMessageDrivenBean / AbstractJmsMessageDrivenBean , 分别。

考虑一个例子,实际上代表无状态会话bean 实现一个普通的java服务对象。 我们有业务 接口:

```
public interface MyComponent {
    public void myMethod(...);
    ...
}
```

我们也有普通的Java实现对象:

```
public class MyComponentImpl implements MyComponent {
    public String myMethod(...) {
        ...
    }
    ...
}
```

和最后的无状态会话Bean本身:

```
public class MyFacadeEJB extends AbstractStatelessSessionBean
        implements MyFacadeLocal {

    private MyComponent myComp;

    /**
     * Obtain our POJO service object from the BeanFactory/ApplicationContext
     * @see org.springframework.ejb.support.AbstractStatelessSessionBean#onEjbCreate()
     */
    protected void onEjbCreate() throws CreateException {
        myComp = (MyComponent) getBeanFactory().getBean(
            ServicesConstants.CONTEXT_MYCOMP_ID);
    }

    // for business method, delegate to POJO service impl.
    public String myFacadeMethod(...) {
        return myComp.myMethod(...);
    }
    ...
}
```

春天EJB支持基类将在默认情况下创建和加载 一个Spring IoC容器作为他们部分的生命周期,然后可用 到EJB(例如,使用在上面的代码中获取POJO 服务对象)。 装运是通过一个策略的一个子类对象 BeanFactoryLocator 。 的实际实现 BeanFactoryLocator 默认使用的是 ContextJndiBeanFactoryLocator 创建 ApplicationContext从资源位置指定为一个JNDI 环境变量(如EJB类,在 java:comp / env / ejb / BeanFactoryPath)。 如果有需要 改变BeanFactory / ApplicationContext加载策略,默认 BeanFactoryLocator 实现使用可能被覆盖 通过调用 setBeanFactoryLocator() 法,要么 在 setSessionContext() , 或在实际的构造函数 EJB。 请参阅Javadocs为更多的细节。

javadoc中所描述的一样,期待着有状态会话bean 钝化和重新激活他们的生命周期的一部分,并使用 非可序列化的容器实例(这是正常的情况下)会 手动调用 unloadBeanFactory() 和 loadBeanFactory() 从 认为() 和 ejbActivate() 分别卸载和重新加载 BeanFactory钝化和激活,因为它不能被救赎 EJB容器。

的默认行为 ContextJndiBeanFactoryLocator 类加载 ApplicationContext 使用EJB, 足够的对于某些情况。 然而,这是有问题的时候 ApplicationContext 是加载大量的豆子, 或初始化这些豆子是耗时或内存 强化(如Hibernate SessionFactory 初始化, 例如),因为每个EJB都有自己的拷贝。 在这种情况下,用户可能想要覆盖默认的 ContextJndiBeanFactoryLocator 使用和使用另一个 BeanFactoryLocator 变体,如 ContextSingletonBeanFactoryLocator 可以负载 并使用一个共享的容器被多个ejb或其他客户。 这样做是相对简单的,通过添加代码类似于此的 EJB:

```
    /**
     * Override default BeanFactoryLocator implementation
     * @see javax.ejb.SessionBean#setSessionContext(javax.ejb.SessionContext)
     */
    public void setSessionContext(SessionContext sessionContext) {
```

```
        super.setSessionContext(sessionContext);
        setBeanFactoryLocator(ContextSingletonBeanFactoryLocator.getInstance());
        setBeanFactoryLocatorKey(ServicesConstants.PRIMARY_CONTEXT_ID);
    }
```

你将需要创建一个bean定义文件命名 beanRefContext.xml 。 这个文件定义了所有bean工厂(通常是在应用程序上下文的形式),可以使用 在EJB。 在许多情况下,这个文件将只包含一个这样的bean定义(在那里 businessApplicationContext.xml 包含bean定义所有业务 服务pojo):

```
<beans>
   <bean id="businessBeanFactory" class="org.springframework.context.support.ClassPathXmlApplicationContext">
      <constructor-arg value="businessApplicationContext.xml" />
   </bean>
</beans>
```

在上面的例子中, ServicesConstants.PRIMARY_CONTEXT_ID 常数 将定义如下:

```
public static final String ServicesConstants.PRIMARY_CONTEXT_ID = "businessBeanFactory";
```

请参阅各自的javadoc BeanFactoryLocator 和 ContextSingletonBeanFactoryLocator 类的更多信息 他们的用法。

22.3.2A EJB 3注入拦截器

对于EJB 3会话bean和消息驱动bean,Spring提供了方便 拦截器,解决了Spring 2.5的 @ autowired 注释 在EJB组件类: org.springframework.ejb.interceptor.SpringBeanAutowiringInterceptor 。 这个拦截器可以应用通过一个 @Interceptors 注释 在EJB组件类,或通过一个 拦截器绑定 XML元素在EJB部署描述符。

```
@Stateless
@Interceptors(SpringBeanAutowiringInterceptor.class)
public class MyFacadeEJB implements MyFacadeLocal {

   // automatically injected with a matching Spring bean
   @Autowired
   private MyComponent myComp;

   // for business method, delegate to POJO service impl.
   public String myFacadeMethod(...) {
      return myComp.myMethod(...);
   }
   ...
}
```

SpringBeanAutowiringInterceptor 默认情况下得到目标 豆子从 ContextSingletonBeanFactoryLocator , 上下文bean定义文件中定义一个命名 beanRefContext.xml 。 默认情况下,单个上下文定义,期望得到类型相当 比的名字。 然而,如果你需要选择多个上下文定义, 一个特定的定位关键是必需的。 这个定位键(即上下文的名称 定义在 beanRefContext.xml)可以显式地指定 通过覆盖 getBeanFactoryLocatorKey 方法 在一个自定义 SpringBeanAutowiringInterceptor 子类。

或者,考虑覆盖 SpringBeanAutowiringInterceptor ' s getBeanFactory 方法,如获得共享 ApplicationContext 从一个定制架类。

23。 一个JMS(Java消息服务)

23.1一个介绍

Spring提供了JMS集成框架,简化了使用 JMS API的很像Spring的集成并为JDBC API。

JMS大致可以分为两个方面的功能,即 生产和消费的信息。 这个 JmsTemplate 类是用于信息的生产 和同步消息接收。 对于异步接收类似 Java EE的消息驱动bean的风格,Spring提供了大量的信息 侦听器容器,用于创建消息驱动pojo (MDPs)。

包 org.springframework.jms.core 提供 核心功能使用JMS。 它包含JMS模板类 这简化了使用JMS处理创建和释放 资源,就像 JdbcTemplate 并对 JDBC。 常见的设计原则是为春天的模板类 助手方法来执行常见的操作和更复杂的 使用,委托加工的本质任务用户实现 回调接口。 JMS模板遵循相同的设计。 类 提供各种方便的方法来发送消息,消耗一个 信息同步,并暴露了JMS会话和消息生产者 给用户。

包 org.springframework.jms.support 提供 JMSException 翻译功能。 翻译将检查 JMSException 一个镜像的层次结构层次的未经检查的异常。 如果有 任何供应商具体子类的检查 javax.jms.JMSException ,这个异常是包装 在不 UncategorizedJmsException 。

包 org.springframework.jms.support.converter 提供了一个 MessageConverter 抽象转化 在Java对象和JMS消息。

包 org.springframework.jms.support.destination 提供 各种策略来管理JMS目的地,比如提供 服务定位器为目的地存储在JNDI。

最后,包 org.springframework.jms.connection 提供了一个 实施 ConnectionFactory 合适 在独立应用程序中使用。 它还包含一个实现 春天的 PlatformTransactionManager 对于JMS (巧妙命名 JmsTransactionManager)。 这 允许无缝集成JMS作为事务性资源投入 春天的事务管理机制。

23.2一个使用Spring JMS

23.2.1A JmsTemplate

这个 JmsTemplate 类是中央类 在JMS核心包。 它简化了使用JMS自它处理 创建和发布的资源在发送或同步 接收消息。

代码,使用 JmsTemplate 只需要 要实现回调接口给他们一个明确定义的高 水平的合同。 这个 MessageCreator 回调 接口创建一个消息给一个 会话 在提供的调用代码 JmsTemplate 。 为了允许更复杂 使用JMS API,回调 SessionCallback 为用户提供了JMS 会话和回调 ProducerCallback 暴露 一个 会话 和 MessageProducer 一对。

JMS API公开两种类型的发送方法,一个带 交货方式、优先级和生存时间作为服务质量(QOS) 参数和一个不带参数使用默认的QOS 值。 因为有许多的发送方法 JmsTemplate 、QOS参数的设置 已经被公开为bean属性来避免重复的号码吗 的发送方法。 同样,超时值同步接收 调用设置使用属性 setReceiveTimeout 。

一些JMS提供者允许设置默认值的QOS 通过配置管理的ConnectionFactory。 这个有效果,调用 MessageProducer 的发送方法 发送(目的地目的地、消息消息) 将使用不同的QOS默认值比指定的JMS 规范。 为了提供一致的QOS管理价值观, 这个JmsTemplate 因此必须特别 能使用自己的QOS值通过设置布尔属性 isExplicitQosEnabled 到 真正的 。

> **注意**
>
> 实例的 JmsTemplate 类 线程安全一旦配置 。 这是重要的 因为它意味着您可以配置的单独一个实例 JmsTemplate 然后安全地注入这 共享 引用到多个合作者。 到 要明确, JmsTemplate 是有状态的,在那 它维护了一个引用 ConnectionFactory ,但这个状态 不 会话状态。

23.2.2A连接

这个 JmsTemplate 需要一个引用 ConnectionFactory 。 这个 ConnectionFactory 是部分的JMS 规范和作为工作的切入点和JMS。 这是 使用客户机应用程序作为一个工厂来创建连接 JMS提供者和封装了各种配置参数,许多 这是针对供应商如SSL配置选项。

内部使用JMS时EJB,供应商提供的实现 的JMS接口,以便他们可以参与声明 事务管理和执行池的连接和会话。 为了使用这种实现,Java EE容器一般 需要您声明一个JMS连接工厂 resource - ref 在EJB或servlet部署 描述符。 确保使用这些特性的JmsTemplate 在EJB中,客户端应用程序 应该确保它引用托管实现的吗 ConnectionFactory 。

缓存消息传递资源

标准的API包括创建许多中间对象。 到 发送一个消息下面执行“API”走

```
ConnectionFactory->Connection->Session->MessageProducer->send
```

ConnectionFactory之间和发送操作有 三个中间对象的创建和销毁。 优化 资源使用和提高性能的两个实现IConnectionFactory被提供。

SingleConnectionFactory

Spring提供了一个实现的 ConnectionFactory 界面, SingleConnectionFactory ,这将返回 相同 连接 在所有 createConnection() 电话和忽略调用 close() 。 这是用于测试和 独立的环境,这样相同的连接可用于 多个 JmsTemplate 调用,可以跨越任何 事务的数量。 SingleConnectionFactory 需要一个参考标准 ConnectionFactory 这将会出现 从JNDI。

CachingConnectionFactory

这个 CachingConnectionFactory 扩展了 功能的 SingleConnectionFactory 和 添加缓存的会

话,MessageProducers,MessageConsumers。 最初的缓存大小设置为1,使用属性 SessionCacheSize 增加数量的缓存 会话。 请注意,实际的数量会更多。会话缓存 比这个数字作为会话缓存的基于他们的承认 模式,所以可以有多达4缓存的会话实例当 SessionCacheSize 设置为一个,一个用于每个 AcknowledgementMode。 MessageProducers和MessageConsumers被缓存 在他们拥有会话,也考虑到独特的 属性的生产者和消费者当缓存。 MessageProducers缓存在基于他们的目的地。 MessageConsumers缓存在基于关键组成的 目的地,选择器,noLocal交付国旗,和耐用 订阅名称(如果创建持久的消费者)。

23.2.3A目的地管理

目的地,像ConnectionFactories,JMS管理 对象可以存储和检索在JNDI。 当配置一个 您可以使用Spring应用程序上下文的 JNDI工厂类 JndiObjectFactoryBean / < jee:jndi查找> 执行依赖 喷射在你的对象的引用,JMS目的地。 然而, 通常这种策略是繁琐的如果有大量的 目的地在应用程序或如果有先进的目的地 管理功能独特的JMS提供者。 这样的例子 先进的目的地管理将创建动态 目的地或支持一个层次名称空间的目的地。 这个 JmsTemplate 代表的分辨率 目的地名称到一个JMS目的地对象来实现的 接口 DestinationResolver 。 DynamicDestinationResolver 是默认 实现使用 JmsTemplate 和 提供解决动态目的地。 一个 JndiDestinationResolver 也提供了 作为一个服务定位器为目的地包含在JNDI和 可选地落回行为包含在 DynamicDestinationResolver 。

经常使用的目的地在JMS应用程序只 运行时已知,因此不能管理时创建的 应用程序部署。 这往往是因为有共享 应用程序逻辑系统组件之间的交互,创建 目的地在运行时根据一个众所周知的命名约定。 即使创建动态目的地不是部分的JMS 规范,大多数供应商已经提供了这个功能。 动态 目的地都创建一个用户定义的名称 区分他们从临时目的地和通常不是 注册在JNDI。 API用于创建动态目的地不同 从供应商到供应商自相关特性 目的地是针对供应商的。 然而,一个简单的实现选择 ,有时是由供应商是漠视的警告 JMS规范和使用 TopicSession 方法 createTopic(String topicName) 或 QueueSession 方法 createQueue(字符串要使用queueName) 创建一个新的 目的地与缺省目标属性。 根据供应商 的实现, DynamicDestinationResolver 可能 然后还创建一个物理目的地而不是只解决 一。

布尔型属性 pubSubDomain 用于 配置 JmsTemplate 用知识的 JMS域正在被使用。 默认情况下,此属性的值 假,表明点到点域,队列,将被使用。 使用此属性 JmsTemplate 决定了 行为的动态目的地通过实现的决议 DestinationResolver 接口。

您还可以配置 JmsTemplate 与 通过属性缺省目标 defaultDestination 。 默认的目的地将是 用于发送和接收操作,不指一个特定的 目的地。

23.2.4A消息侦听器容器

最常见的用途之一JMS消息在EJB世界是 驱动消息驱动bean(mdb)。 Spring提供了一个解决方案来创建 消息驱动pojo(MDPs),不会将一个用户一个EJB 集装箱。 (见 SectionA 23 4 2,一个异步接收消息驱动POJOsa—— 详细报道Spring的MDP支持。)

一个消息侦听器容器是用来接收消息从一个 JMS消息队列和驱动MessageListener,注入 它。 侦听器容器负责所有线程的消息接收和派遣到侦听器进行处理。 一个消息 侦听器容器是中间人MDP和消息传递 供应商,并负责登记接收消息, 参与交易,资源获取和释放, 异常转换和诸如此类的。 这允许你作为一个应用程序 开发人员编写(可能是复杂的)业务逻辑关联 接收消息(和可能的反应),和代表 样板JMS基础设施问题的框架。

有两个标准JMS消息侦听器容器包装 与春天,每个都有其专门的功能集。

SimpleMessageListenerContainer

这消息侦听器容器是简单的两个 标准的口味。 它创建一个固定数量的JMS会话和 消费者在启动、注册侦听器使用标准JMS MessageConsumer.setMessageListener() 方法, 和让JMS提供者执行侦听器回调。 这变种不允许动态适应运行时要求或 参与外部管理事务。 兼容性明智, 它保持非常密切的精神独立JMS规范 ,但一般不兼容Java EE的JMS的限制。

DefaultMessageListenerContainer

这消息侦听器容器中使用的大多数情况下。 相比之下, SimpleMessageListenerContainer , 这个集装箱变种确实允许动态适应运行时 要求和能够参与外部管理事务。 每个收到的消息是注册时XA事务 配置一个 JtaTransactionManager ,所以 处理可能利用XA事务语义。 这 侦听器容器罢工之间取得良好的平衡低要求 JMS提供者,先进的功能,如事务 参与,并且兼容Java EE环境中。

23.2.5A事务管理

弹簧提供了一个 JmsTransactionManager 管理事务一个JMS ConnectionFactory 。 这允许JMS应用程序 利用管理的事务特性中描述的春天 ChapterA 12, 事务管理 。 这个 JmsTransactionManager 执行本地资源 事务,绑定一个JMS连接/会话对从

指定的 ConnectionFactory 到线程。 JmsTemplate 自动检测这样的 事务性资源和操作进行相应的。

在一个Java EE环境中, ConnectionFactory 将池连接和 会话,所以这些资源有效地重用在交易。 在一个独立的环境,使用Spring的 SingleConnectionFactory 将会导致一个共享 jms 连接 ,每个事务拥有它的 自己独立的 会话 。 或者,考虑 使用一个特定于提供程序的池适配器如ActiveMQ的 PooledConnectionFactory 类。

JmsTemplate 还可以用吗 JtaTransactionManager 和一个具有xa能力JMS ConnectionFactory 用于执行分布式 事务。 注意,这需要使用JTA事务 经理以及一个正确配置xa ConnectionFactory ! (检查 你的Java EE服务器的/ JMS提供者的文档)。

之间的代码重用一个托管和非托管事务 环境可以被混淆在使用JMS API来创建一个 会话 从 连接 。 这是因为JMS API只有一个工厂方法创建一个 会话 和它需要的值 事务和应答模式。 在托管环境,设置 这些值的责任是环境的事务 基础设施,所以这些值忽略供应商的包装器 JMS连接。 当使用 JmsTemplate 在 一个非托管的环境你可以指定这些值通过使用 属性 sessionTransacted 和 sessionAcknowledgeMode 。 当使用一个 PlatformTransactionManager 与 JmsTemplate ,模板将永远是给定一个 事务性JMS 会话 。

23.3一个发送一个 消息

这个 JmsTemplate 包含许多便利 方法来发送消息。 有发送方法,指定 目的地使用 destination 对象 和那些指定目的地使用字符串中使用的JNDI 查找。 send方法,没有目的地参数使用 缺省目标。 下面是一个示例,它将消息发送到队列 使用1.0.2中实现。

```
import javax.jms.ConnectionFactory;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Queue;
import javax.jms.Session;

import org.springframework.jms.core.MessageCreator;
import org.springframework.jms.core.JmsTemplate;

public class JmsQueueSender {

    private JmsTemplate jmsTemplate;
    private Queue queue;

    public void setConnectionFactory(ConnectionFactory cf) {
        this.jmsTemplate = new JmsTemplate(cf);
    }

    public void setQueue(Queue queue) {
        this.queue = queue;
    }

    public void simpleSend() {
        this.jmsTemplate.send(this.queue, new MessageCreator() {
            public Message createMessage(Session session) throws JMSException {
              return session.createTextMessage("hello queue world");
            }
        });
    }
}
```

下面的例子使用了 MessageCreator 回调 创建一个文本消息从提供的 会话 对象。 这个 JmsTemplate 由传递 引用 ConnectionFactory 。 作为替代, 一个零参数的构造函数和 ConnectionFactory 并给出了可用于构建实例在JavaBean样式吗(使用BeanFactory或普通的Java代码)。 或者,考虑派生 从春天的 JmsGatewaySupport 便利的基类, 它提供了预构建JMS配置bean属性。

该方法 destinationName MessageCreator发送(String, 创造者) 让你发送消息的字符串名称的使用 目的地。 如果这些名字被注册在JNDI,你应该设置 DestinationResolver 属性模板的一个 实例的 JndiDestinationResolver 。

如果你创建了 JmsTemplate 和指定的 一个默认的目的地 发送(MessageCreator c) 发送一个消息到达目的地。

23.3.1A使用消息转换器

为了方便发送的域模型对象, JmsTemplate 有不同的发送方法,拿一个吗 Java对象作为一个参数为一个消息的数据内容。 重载方法 convertAndSend() 和 receiveAndConvert() 在 JmsTemplate 代表一个转换过程 实例的 MessageConverter 接口。 这接口定义了一个简单的合同将Java对象之间, JMS消息。 默认实现 SimpleMessageConverter 支持转换 之间 字符串 和 TextMessage , byte[] 和 BytesMesssage ,和 java util地图 和 MapMessage 。 通过使用转换器,你和你的 应用程序代码就可以专注于业务对象正在发送或 收到通过JMS和不关心细节的它是如何 表示为一个JMS消息。

目前包括一个沙箱 MapMessageConverter 它使用反射来 转换一个JavaBean和一之间 MapMessage 。 其他受欢迎的实现选择你可能实现自己 转换器,使用现有的XML编组方案,如JAXB、 蓖麻,XMLBeans,或XStream,创建一个 TextMessage 代表 对象。

适应的设置一个消息的属性、头部和 身体不能一般地封装在一个转换器类, 这个 MessagePostProcessor 接口给 你访问消息后转换,但是在它 发送。 下面的例子演示了如何修改一个消息头和 一个财产后 java util地图 是 转换为一个消息。

```
public void sendWithConversion() {
    Map map = new HashMap();
    map.put("Name", "Mark");
    map.put("Age", new Integer(47));
    jmsTemplate.convertAndSend("testQueue", map, new MessagePostProcessor() {
        public Message postProcessMessage(Message message) throws JMSException {
            message.setIntProperty("AccountID", 1234);
            message.setJMSCorrelationID("123-00001");
            return message;
        }
    });
}
```

这个结果在一个消息的形式:

```
MapMessage={
    Header={
        ... standard headers ...
        CorrelationID={123-00001}
    }
    Properties={
        AccountID={Integer:1234}
    }
    Fields={
        Name={String:Mark}
        Age={Integer:47}
    }
}
```

23.3.2A SessionCallback 和 ProducerCallback

而发送业务涵盖许多常见的使用场景,那里 一些情况下,您想要执行多个操作在一个JMS 会话 或 MessageProducer 。 这个 SessionCallback 和 ProducerCallback 暴露JMS 会话 和 会话 / MessageProducer 分别对。 这个 execute() 方法 JmsTemplate 执行这些回调 方法。

23.4一个接收消息

23.4.1A同步接收

而JMS是通常关联着异步处理,它 可以使用消息同步。 重载 收到(. .) 方法提供此功能。 在同步接收、调用线程阻塞,直到一个消息 变得可用。 这是一个危险的操作自调用 线程可能被阻塞。无限期 房地产 receiveTimeout 指定接收者多久 放弃之前应该等待等待一个消息。

23.4.2A异步接收消息驱动pojo,

在一个时尚类似于一个消息驱动的Bean(MDB)在EJB 世界,消息驱动POJO(MDP)作为接收机为JMS 消息。 一个限制(参见下文的讨论 这个 MessageListenerAdapter 类)MDP 它必须实现 javax.jms.MessageListener 接口。 也请注意,如果您将收到的POJO 信息在多个线程,重要的是要确保你的 实现线程安全的。

下面是一个简单的实现MDP:

```
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

public class ExampleListener implements MessageListener {

    public void onMessage(Message message) {
        if (message instanceof TextMessage) {
            try {
                System.out.println(((TextMessage) message).getText());
            }
            catch (JMSException ex) {
                throw new RuntimeException(ex);
            }
```

```
        }
        else {
            throw new IllegalArgumentException("Message must be of type TextMessage");
        }
    }
}
```

一旦你实现了你的 MessageListener ,我们需要创建一个 消息侦听器容器。

找到下面的示例演示如何定义和配置的 消息侦听器容器,附带弹簧(在这种情况下 DefaultMessageListenerContainer)。

```
<!-- this is the Message Driven POJO (MDP) -->
<bean id="messageListener" class="jmsexample.ExampleListener" />

<!-- and this is the message listener container -->
<bean id="jmsContainer" class="org.springframework.jms.listener.DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="destination" ref="destination"/>
    <property name="messageListener" ref="messageListener" />
</bean>
```

请参考弹簧Javadoc的各种消息侦听器 容器中的所有描述所支持的特性每个 实现。

23.4.3A了 SessionAwareMessageListener 接口

这个 SessionAwareMessageListener 接口是一个spring特定接口,它提供了一个类似的 合同到JMS MessageListener 接口,还提供了消息处理方法访问 JMS 会话 的 消息 是收到了。

```
package org.springframework.jms.listener;

public interface SessionAwareMessageListener {

    void onMessage(Message message, Session session) throws JMSException;
}
```

你可以选择你的MDPs实现这个接口(在 偏好标准JMS MessageListener 如果你想要的接口) 你的MDPs能够应对任何收到的信息(使用 会话 中提供 onMessage(消息、会话) 方法)。 所有的 消息侦听器容器实现船舶与弹簧有 支持MDPs实现要么 MessageListener 或 SessionAwareMessageListener 接口。 类,实现了 SessionAwareMessageListener 来的 警告,然后与弹簧通过接口。 这个 选择是否使用它是完全交给你一个 应用程序开发人员或架构师。

请注意, “onMessage(. .)” 方法 这个 SessionAwareMessageListener 接口 抛出 JMSException 。 与标准 jms MessageListener 界面,当使用 这个 SessionAwareMessageListener 接口,它的职责是处理任何客户端代码 抛出的异常。

23.4.4A了 MessageListenerAdapter

这个 MessageListenerAdapter 类是 最后的组件在春天的异步消息传递的支持:在一个 简而言之,它允许你暴露几乎 任何 类 作为一个MDP(当然有一些限制)。

考虑下面的接口定义。 注意,尽管 接口扩展了没有 MessageListener 也 SessionAwareMessageListener 接口, 它仍然可以被用作MDP通过使用 MessageListenerAdapter 类。 还要注意如何 各种信息处理方法是强类型的根据 内容 的各种 消息 类型, 他们可以接收和 处理。

```
public interface MessageDelegate {

    void handleMessage(String message);

    void handleMessage(Map message);

    void handleMessage(byte[] message);

    void handleMessage(Serializable message);
}
```

```
public class DefaultMessageDelegate implements MessageDelegate {
    // implementation elided for clarity...
}
```

特别是,注意上面的实现 MessageDelegate 接口(以上 DefaultMessageDelegate 类) 没有 JMS依赖性在所有。 它真的是一个 POJO, 我们将做成一个MDP通过以下配置。

```
<!-- this is the Message Driven POJO (MDP) -->
<bean id="messageListener" class="org.springframework.jms.listener.adapter.MessageListenerAdapter">
```

```
    <constructor-arg>
        <bean class="jmsexample.DefaultMessageDelegate"/>
    </constructor-arg>
</bean>

<!-- and this is the message listener container... -->
<bean id="jmsContainer" class="org.springframework.jms.listener.DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="destination" ref="destination"/>
    <property name="messageListener" ref="messageListener" />
</bean>
```

下面是一个例子,另一个MDP,只能处理 接收JMS TextMessage 消息。 注意消息处理方法实际上是叫 “接收” (名称的信息处理方法 一个 MessageListenerAdapter 默认为 “handleMessage”),但它是可配置的(正如你将 见下文)。 还要注意如何 “接收(. .)” 方法 是强类型的接收和回复只是JMS TextMessage 消息。

```
public interface TextMessageDelegate {

    void receive(TextMessage message);
}
```

```
public class DefaultTextMessageDelegate implements TextMessageDelegate {
    // implementation elided for clarity...
}
```

配置的服务员 MessageListenerAdapter 看起来就像 这个:

```
<bean id="messageListener" class="org.springframework.jms.listener.adapter.MessageListenerAdapter">
    <constructor-arg>
        <bean class="jmsexample.DefaultTextMessageDelegate"/>
    </constructor-arg>
    <property name="defaultListenerMethod" value="receive"/>
    <!-- we don't want automatic message context extraction -->
    <property name="messageConverter">
        <null/>
    </property>
</bean>
```

请注意,如果上面的 “messageListener的 接收JMS 消息 类型的其他 比 TextMessage ,一个 IllegalStateException 将会抛出(和 随后吞下)。 另一个的能力 MessageListenerAdapter 类的能力 自动发回一个响应 消息 如果一个处理程序方法返回一个非void的价值。 考虑到接口和类:

```
public interface ResponsiveTextMessageDelegate {

    // notice the return type...
    String receive(TextMessage message);
}
```

```
public class DefaultResponsiveTextMessageDelegate implements ResponsiveTextMessageDelegate {
    // implementation elided for clarity...
}
```

如果上面的 DefaultResponsiveTextMessageDelegate 用于 联同一个 MessageListenerAdapter 然后 任何非空值,返回执行 “接收(. .)” 方法将(在默认 配置)被转化为一个 TextMessage 。 由此产生的 TextMessage 会被发送到吗 目的地 (如果有的话)中定义 JMS应答财产的原始 消息 ,或默认 目的地 设置 MessageListenerAdapter (如果已 配置);如果没有 目的地 是发现 然后一个 InvalidDestinationException 将 扔(和请注意这个异常 将 不 被吞下, 将 向上传递 调用堆栈)。

在交易23.4.5A处理消息

消息监听器调用事务内只需要 重新配置监听器容器。

当地资源交易可以被激活通过 sessionTransacted 国旗在侦听器容器 定义。 每个消息侦听器调用将操作在一个 活跃的JMS事务,消息接收回滚的情况下 侦听器执行失败。 发送一条响应消息(通过 SessionAwareMessageListener)将部分 相同的本地事务,但任何其他资源操作(这样的 作为数据库访问)将独立运作。 这通常需要 重复的信息检测的侦听器实现,覆盖了 数据库处理情况已承诺但消息处理 未能提交。

```
<bean id="jmsContainer" class="org.springframework.jms.listener.DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="destination" ref="destination"/>
    <property name="messageListener" ref="messageListener"/>
    <property name="sessionTransacted" value="true"/>
</bean>
```

参加外部管理的事务,你会 需要配置一个事务管理器,并使用一个侦听器容器 支持外部管理事务:通常 DefaultMessageListenerContainer 。

配置消息侦听器容器XA事务 参与,要配置一个 JtaTransactionManager (默认情况下, 代表Java EE服务器的事务子系统)。 注意, 潜在的JMS ConnectionFactory需要具有xa能力和正确 注册你的JTA事务协调员! (检查你的Java EE 服务器的配置JNDI资源。) 这允许消息接收 以及如数据库访问参与同一事务(统一提交语义,以牺牲XA事务日志 开销)。

```
<bean id="transactionManager" class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

那么你只需要将它添加到我们前面的集装箱 配置。 这个集装箱将会照料剩下的一切。

```
<bean id="jmsContainer" class="org.springframework.jms.listener.DefaultMessageListenerContainer">
  <property name="connectionFactory" ref="connectionFactory"/>
  <property name="destination" ref="destination"/>
  <property name="messageListener" ref="messageListener"/>
  <property name="transactionManager" ref="transactionManager"/>
</bean>
```

JCA 23.5支持消息端点

从2.5版本开始,春天还提供了支持 基于jca MessageListener 集装箱。 这个 JmsMessageEndpointManager 将尝试 自动确定 ActivationSpec 类的名字从提供者的 ResourceAdapter 类名。 因此,它 通常可能只是提供Spring的通用吗 JmsActivationSpecConfig 按照以下所示 的例子。

```
<bean class="org.springframework.jms.listener.endpoint.JmsMessageEndpointManager">
  <property name="resourceAdapter" ref="resourceAdapter"/>
  <property name="activationSpecConfig">
    <bean class="org.springframework.jms.listener.endpoint.JmsActivationSpecConfig">
      <property name="destinationName" value="myQueue"/>
    </bean>
  </property>
  <property name="messageListener" ref="myMessageListener"/>
</bean>
```

或者,您可以设置 JmsMessageEndpointManager 与一个给定 ActivationSpec 对象。 这个 ActivationSpec 对象也可能来自一个 JNDI查找(使用 < jee:jndi查找>)。

```
<bean class="org.springframework.jms.listener.endpoint.JmsMessageEndpointManager">
  <property name="resourceAdapter" ref="resourceAdapter"/>
  <property name="activationSpec">
    <bean class="org.apache.activemq.ra.ActiveMQActivationSpec">
      <property name="destination" value="myQueue"/>
      <property name="destinationType" value="javax.jms.Queue"/>
    </bean>
  </property>
  <property name="messageListener" ref="myMessageListener"/>
</bean>
```

使用Spring的 ResourceAdapterFactoryBean , 目标 ResourceAdapter 可能是 本地配置中所描绘的一样下面的例子。

```
<bean id="resourceAdapter" class="org.springframework.jca.support.ResourceAdapterFactoryBean">
  <property name="resourceAdapter">
    <bean class="org.apache.activemq.ra.ActiveMQResourceAdapter">
      <property name="serverUrl" value="tcp://localhost:61616"/>
    </bean>
  </property>
  <property name="workManager">
    <bean class="org.springframework.jca.work.SimpleTaskWorkManager"/>
  </property>
</bean>
```

指定的 WorkManager 也可能 指向一个特定于环境的线程池——通常通过 SimpleTaskWorkManager的 “asyncTaskExecutor” 财产。 考虑定义共享的线程池为你所有 ResourceAdapter 如果你碰巧实例 使用多个适配器。

在某些环境中(例如。WebLogic 9或以上),整个 ResourceAdapter 对象也可从 JNDI相反(使用 < jee:jndi查找>)。 这个 基于spring的消息监听器可以与之交互,服务器托管 ResourceAdapter ,也使用服务器的 内置 WorkManager 。

请咨询的JavaDoc JmsMessageEndpointManager , JmsActivationSpecConfig ,和 ResourceAdapterFactoryBean 为更多的细节。

春天也提供了一个通用的JCA消息端点管理器这是 没有绑定到JMS: org.springframework.jca.endpoint.GenericMessageEndpointManager 。 该组件允许使用任何消息侦听器类型(如CCI

MessageListener)和任何特定于提供程序的ActivationSpec对象。 检查 你的JCA提供者的文档以了解实际的 功能的连接器,并咨询 GenericMessageEndpointManager “年代的JavaDoc spring特定配置的细节。

注意

基于jca消息端点管理非常类似于EJB 2.1 消息驱动bean,它使用相同的底层资源提供者 合同。 喜欢与EJB 2.1 mdb,任何消息侦听器接口 的支持,您可以使用JCA提供者在Spring上下文中的作为 好。 春天却提供了明确的“便利”支持 JMS,仅仅是因为JMS是最常见的端点API一起使用 JCA端点管理合同。

23.6一个JMS支持名称空间

Spring 2.5引入了XML名称空间简化JMS 配置。 使用JMS名称空间元素,你将需要 参考JMS模式:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jms="http://www.springframework.org/schema/jms"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/jms http://www.springframework.org/schema/jms/spring-jms.xsd">

<!-- <bean/> definitions here -->

</beans>
```

命名空间包含两个顶级元素: <侦听器容器/ > 和 < jca侦听器容器/ > 这两种可能 包含一个或多个 <侦听器/ > 子元素。 下面的示例是一个基本的配置了两个听众。

```
<jms:listener-container>

  <jms:listener destination="queue.orders" ref="orderService" method="placeOrder"/>

  <jms:listener destination="queue.confirmations" ref="confirmationLogger" method="log"/>

</jms:listener-container>
```

上面的示例创建两个不同的等价于侦听器 容器和两个截然不同的bean定义 MessageListenerAdapter bean定义为 演示了在 SectionA 23 4 4,一个了 MessageListenerAdapter 一个 。 除了上面所示的属性, 侦听器 元素 可能包含几个可选的。 下表描述了所有可用的 属性:

23.1为多。 一个属性的JMS <侦听器> 元素

属性	描述
id	一个bean名称托管侦听器容器。 如果 未指定,一个bean名称将自动 生成的。
目的地(必需)	目的地的名字为这个侦听器,解决了 通过 DestinationResolver 策略。
Ref(必需)	bean名称的处理程序对象。
方法	处理程序方法的名称来调用。 如果 Ref 指向一个 MessageListener 或弹簧 SessionAwareMessageListener , 这 属性可以省略。
响应目的地	名称的默认响应目的地发送 响应消息。 这将是应用于案例的一个请求 消息,不带“JMSReplyTo”字段。 这个类型的这 目的地将取决于侦听器容器的 “目的地型”属性。 注意:这只适用于一个 侦听器方法与一个返回值,而每个结果对象 将被转换成一个响应消息。
订阅	持久订阅的名称,如果 任何。
选择器	一个可选的消息选择器对于这个 侦听器。

这个 <侦听器容器/ > 元素也 接受一些可选的属性。 这允许定制的 不同的策略(例如, TaskExecutor 和 DestinationResolver)以及基本JMS设置 和资源引用。 利用这些属性,可以定义 高度定制的侦听器容器同时仍然受益 便利的名称空间。

```
<jms:listener-container connection-factory="myConnectionFactory"
                        task-executor="myTaskExecutor"
                        destination-resolver="myDestinationResolver"
                        transaction-manager="myTransactionManager"
                        concurrency="10">

    <jms:listener destination="queue.orders" ref="orderService" method="placeOrder"/>

    <jms:listener destination="queue.confirmations" ref="confirmationLogger" method="log"/>

</jms:listener-container>
```

下表描述了所有可用的属性。 咨询 类级别的Javadoc AbstractMessageListenerContainer 和它的具体 子类的详情请看个人属性。 Javadoc也 提供了一个讨论交易的选择和消息返还 场景。

23.2为多。 一个属性的JMS <侦听器容器> 元素

属性	描述
容器类型	这个监听器容器的类型。 可用 选项有: 默认 , 简单 , default102 ,或 simple102 (默认值是 default ')。
连接工厂	一个引用到JMS ConnectionFactory bean(默认 bean的名字是 的connectionFactory的)。
任务执行人	指春天 TaskExecutor 对于JMS侦听器 调用器。
目标解析器	一个参考 DestinationResolver 策略 解决JMS 目的地 。
消息转换器	一个参考 MessageConverter 策略 将JMS消息侦听器方法参数。 默认是一个 SimpleMessageConverter 。
目的地型	JMS目的地类型为这个监听器: 队列 , 主题 或 durableTopic 。 默认的是 队列 。
客户机id	JMS客户机id对于这个侦听器容器。 需要 指定当使用持久订阅。
缓存	缓存级别为JMS资源: 没有 , 连接 , 会话 , 消费者 或 汽车 。 在默认情况下(汽车), 缓存级别将有效地 “消费” , 除非外部 事务管理器指定了-在这种情况下 有效违约将 没有 (假设 事务管理,风格ee的Java给定 ConnectionFactory是一个包括遵从扩充体系结构标准池)。
承认	本地JMS承认模式: 汽车 , 客户 , dups-ok 或 事务 。 一个 价值 事务 激活本地 事务 会话 。 作为一个 选择,指定 事务管理器 下面描述的属性。 默认是 汽车 。

事务管理器	引用一个外部 PlatformTransactionManager (通常是一个事务协调员,例如基于xa。春天的 JtaTransactionManager)。 如果不指定, 将使用本机承认(见“承认” 属性)。
并发	并发会话的数量/消费者开始 对于每个侦听器。 可以是一个简单的数字,表示吗 最大数量(如。 “5”)或一系列 指示下也 作为上限(如。 “3 - 5”)。 注意,是一个特定的最低 只是一个提示,可能在运行时被忽略。 默认是1;保持 并发限于1对于主题听众或如果队列 排序是非常重要的;考虑提高一般 队列。
预取	最大数量的信息加载到一个单一的 会话。 注意,提高这个数字可能导致饥饿的 并发的消费者!

配置一个基于jca容器的听众“jms”模式 支持是非常相似的。

```
<jms:jca-listener-container resource-adapter="myResourceAdapter"
                            destination-resolver="myDestinationResolver"
                            transaction-manager="myTransactionManager"
                            concurrency="10">

    <jms:listener destination="queue.orders" ref="myMessageListener"/>

</jms:jca-listener-container>
```

可用的配置选项JCA变体是 描述如下表:

23.3为多。 一个属性的JMS < jca侦听器容器/ > 元素

属性	描述
资源适配器	一个参考JCA ResourceAdapter bean(默认 bean的名字是 “resourceAdapter”)。
激活规范工厂	一个参考 JmsActivationSpecFactory 。 这个 默认是自动侦测JMS提供者及其 ActivationSpec 类(参见 DefaultJmsActivationSpecFactory)
目标解析器	一个参考 DestinationResolver 策略 解决JMS 目的地 。
消息转换器	一个参考 MessageConverter 策略 将JMS消息侦听器方法参数。 默认是一个 SimpleMessageConverter 。
目的地型	JMS目的地类型为这个监听器: 队列 , 主题 或 durableTopic 。 默认的是 队列 。
客户机id	JMS客户机id对于这个侦听器容器。 需要 指定当使用持久订阅。
承认	本地JMS承认模式: 汽车 , 客户 , dups-ok 或 事务 。 一个 价值 事务 激活本地 事务 会话 。 作为一个 选择,指定 事务管理器 下面描述的属性。 默认是 汽车 。
事务管理器	引用一个春天 JtaTransactionManager 或 javax.transaction.TransactionManager 为启动XA事务为每个传入消息。 如果 未指定,将使用本机承认(参见 “承认” 属性)。
并发	并发会话的数量/消费者开始 对于每个侦听器。 可以是一个简单的数字,表示吗 最大数量(如。 “5”)或一系列 指示下也 作为上限(如。 “3 - 5”)。 注意,是一个特定的最低 只是一个提示,通常会被忽略在运行时使用 JCA 侦听器容器。 默认值是1。
预取	最大数量的信息加载到一个单一的 会话。 注意,提高这个数字可能导致饥饿的 并发的消费者!

24.一个JMX

24.1一个介绍

在春天的JMX支持提供的功能很容易 和透明地集成Spring应用程序到一个JMX 基础设施。

具体来说,Spring的JMX支持提供了四个核心 特点:

- 自动注册的 任何 春天 bean作为JMX MBean
- 一种灵活的机制来控制的管理界面 你的豆子
- 声明暴露在遥远的mbean,jsr - 160 连接器
- 简单的代理本地和远程MBean 资源

这些特性是为了工作而不耦合你 应用程序组件要么弹簧或JMX接口和类。事实上,大部分应用程序类不需要知道 要么弹簧或JMX为了利用Spring JMX 特性。

> **JMX吗?**
>
> 这一章是没有介绍JMX..... 它不会尝试 解释为什么一个人的动机可能想要使用JMX(或实际上是什么 字母JMX实际上代表)。 如果你是新到JMX,请查看 SectionA 24.8,一个进一步Resourcesa 本章末尾的。

24.2一个出口你的豆子,JMX

核心类在Spring的JMX框架 MBeanExporter 。 这个类是负责 你的Spring bean,并注册他们使用JMX MBeanServer 。 例如,考虑以下 类:

```
package org.springframework.jmx;

public class JmxTestBean implements IJmxTestBean {

    private String name;
    private int age;
    private boolean isSuperman;

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public int add(int x, int y) {
        return x + y;
    }

    public void dontExposeMe() {
        throw new RuntimeException();
    }
}
```

揭露了这个bean的属性和方法作为属性和 操作的MBean只需配置的一个实例 MBeanExporter 在你的配置文件和类 通过在bean如下所示:

```
<beans>

  <!-- this bean must not be lazily initialized if the exporting is to happen -->
  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter" lazy-init="false">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean1" value-ref="testBean"/>
      </map>
    </property>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

</beans>
```

相关的bean定义从上面的配置代码片段 是 出口国 bean。 这个 bean 属性告诉 MBeanExporter 到底的 您的bean必须出口 JMX MBeanServer 。 在默认配置中,关键的中的每个条目 bean 地图 用作 ObjectName bean的引用 相应的输入值。 这种行为可以改变所描述的那样 SectionA 24.4,一个控制 ObjectName 对你的beansas 。

在这个配置中, testBean bean是 公开为下一个MBean ObjectName 豆:name = testBean1 。 默认情况下,所有 公共 bean 的属性被公开为 属性和所有 公共 方法(酒吧那些 继承了 对象 类)被公开为 操作。

24.2.1A创建一个 MBeanServer

上面的配置假设应用程序的运行 一个环境,有一个(且只有一个) MBeanServer 已经运行。 在这种情况下,Spring 将试图找到跑步吗 MBeanServer 并注册您的bean与该服务器(如果有的话)。 这种行为是 当你的应用程序运行很有用在容器如 Tomcat或 IBM WebSphere,有它自己的 MBeanServer 。

然而,这种方法是无用的在一个独立的环境, 或者在一个容器中运行时,并没有提供一个 MBeanServer 。 为了解决这个您可以创建一个 MBeanServer 通过添加一个实例声明 实例的 org.springframework.jmx.support.MBeanServerFactoryBean 类到您的配置。 你也可以保证一个特定的 MBeanServer 被设置的值 MBeanExporter ' s 服务器 财产 MBeanServer 值返回 MBeanServerFactoryBean ;例如:

```
<beans>

 <bean id="mbeanServer" class="org.springframework.jmx.support.MBeanServerFactoryBean"/>

 <!--
  this bean needs to be eagerly pre-instantiated in order for the exporting to occur;
  this means that it must not be marked as lazily initialized
 -->
 <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
  <property name="beans">
   <map>
    <entry key="bean:name=testBean1" value-ref="testBean"/>
   </map>
  </property>
  <property name="server" ref="mbeanServer"/>
 </bean>

 <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
  <property name="name" value="TEST"/>
  <property name="age" value="100"/>
 </bean>

</beans>
```

这里的一个实例 MBeanServer 创建 的 MBeanServerFactoryBean 和供给 这个 MBeanExporter 通过服务器属性。 当 你提供自己的 MBeanServer 实例, MBeanExporter 不会试图找到一个吗 运行 MBeanServer 和将使用提供的 MBeanServer 实例。 为它正常工作, 你必须(当然)有一个JMX实现在你的类路径。

24.2.2A重用现有的 MBeanServer

如果没有指定服务器, MBeanExporter 试图自动检测运行 MBeanServer 。 这个作品在大多数环境中,只有一个 MBeanServer 使用实例,然而当多个 实例存在,出口商可能选错了服务器。 在这样的 情况下,一个人应该使用 MBeanServer agentId 说明哪些实例被使用:

```
<beans>
 <bean id="mbeanServer" class="org.springframework.jmx.support.MBeanServerFactoryBean">
  <!-- indicate to first look for a server -->
  <property name="locateExistingServerIfPossible" value="true"/>
  <!-- search for the MBeanServer instance with the given agentId -->
  <property name="agentId" value="<MBeanServer instance agentId>"/>
 </bean>

 <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
  <property name="server" ref="mbeanServer"/>
 ...
 </bean>
</beans>
```

平台/情况下现有的 MBeanServer 有一个动态(或未知) agentId 这是通过查找检索 方法,一个人应该使用 工厂方法 :

```
<beans>
 <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
  <property name="server">
   <!-- Custom MBeanServerLocator -->
   <bean class="platform.package.MBeanServerLocator" factory-method="locateMBeanServer"/>
  </property>
```

```
    <!-- other beans here -->

  </bean>
</beans>
```

24.2.3A延迟初始化的mbean

如果你配置一个bean与 MBeanExporter 这也是配置为懒惰 初始化,那么 MBeanExporter 将 **不** 打破这个合同,将避免 实例化bean。 相反,它将与注册一个代理 这个 MBeanServer 并将推迟获得bean 从容器中,直到第一次调用代理的发生。

24.2.4A自动注册的mbean

任何豆类出口通过 MBeanExporter 和已经有效的mbean 注册原有的 MBeanServer 没有 从春天进一步干预。 mbean可以自动检测 的 MBeanExporter 通过设置 自动检测 财产 真正的 :

```
<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
 <property name="autodetect" value="true"/>
</bean>

<bean name="spring:mbean=true" class="org.springframework.jmx.export.TestDynamicMBean"/>
```

在这里,豆叫 春天:mbean = true 是 已经一个有效的JMX MBean和将自动注册 春天。 默认情况下,豆类,个对JMX的登记 有他们的bean名称用作吗 ObjectName 。 这 行为可以覆盖所详述的 SectionA 24.4,一个控制 ObjectName 对你的beansas 。

24.2.5A控制登记行为

考虑的场景:一个春天 MBeanExporter 尝试注册一个 MBean 与一个 MBeanServer 使用 ObjectName “豆:name = testBean1” 。 如果一个 MBean 实例已经被注册在 同样的 ObjectName ,默认行为是 失败(和抛出 InstanceAlreadyExistsException)。

可以控制的行为,到底会发生什么事 当一个 MBean 是注册一个 MBeanServer 。 Spring的JMX支持允许 三种不同的登记行为控制登记 行为当注册过程发现一个 MBean 已经被注册在相同吗 ObjectName ,这些登记行为 在总结如下表:

24.1为多。 一个注册行为

登记行为	解释
注册失败在现有	这是默认的注册行为。 如果一个 MBean 实例已经 注册在相同 ObjectName , 这个 MBean 正在注册将 不允许注册和一个 InstanceAlreadyExistsException 将 扔。 现有的 MBean 是 不受影响。
登记忽视现有	如果一个 MBean 实例有 已经注册在相同 ObjectName , MBean 正在注册将 不 被注册。 现有的 MBean 未受影响,不是吗 异常 将会抛出。 这是有用的在设置在多个应用程序 想要分享一个共同的 MBean 在一个 共享 MBeanServer 。
登记取代现有	如果一个 MBean 实例有 已经注册在相同 ObjectName ,现有的 MBean 那是以前注册的 将未注册和新的吗 MBean 将注册在其位置(新吗 MBean 有效地代替了以前 实例)。

上面的值被定义为常量的 MBeanRegistrationSupport 类(MBeanExporter 类来源于这 超类)。 如果你想要改变默认的注册行为, 你只需要设置 registrationBehaviorName 财产在你 MBeanExporter 定义其中的一个 值。

下面的例子说明了如何效应的一个转变 默认的注册行为 登记取代现有 行为:

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean1" value-ref="testBean"/>
      </map>
    </property>
    <property name="registrationBehaviorName" value="REGISTRATION_REPLACE_EXISTING"/>
  </bean>
```

```
  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

</beans>
```

24.3一个控制管理接口的豆子

在前面的示例中,您没有控制管理 您的bean接口; 所有 的 公共 属性和方法的每个导出的bean 被曝光的JMX分别属性和操作。 锻炼 更细粒度的控制哪些属性和方法的 出口豆类实际上是公开为JMX的属性和操作, Spring JMX提供了一个全面的和可扩展的机制 控制管理接口的豆子。

24.3.1A了 MBeanInfoAssembler 接口

在幕后, MBeanExporter 代表的一个实现 org.springframework.jmx.export.assembler.MBeanInfoAssembler 接口负责定义的管理界面 每个bean被暴露。 默认实现, org.springframework.jmx.export.assembler.SimpleReflectiveMBeanInfoAssembler , 只是定义了一个管理界面,暴露所有公共属性 和方法(如你之前看到的例子)。 弹簧提供了两个 附加的实现 MBeanInfoAssembler 接口,允许 你来控制生成的管理界面使用要么 源代码级的元数据或任意界面。

24.3.2A使用源代码级别的元数据(JDK 5.0注释)

使用 MetadataMBeanInfoAssembler 你 可以定义为您的bean管理接口使用源水平 元数据。 阅读的元数据被封装的 org.springframework.jmx.export.metadata.JmxAttributeSource 接口。 Spring JMX提供了一个默认实现使用JDK 5.0注释, 即 org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource 。 这个 MetadataMBeanInfoAssembler 必须 配置一个实现实例 的 JmxAttributeSource 界面 函数正确(有 没有 默认)。

标记一个bean出口到JMX,你应该标注bean 类 ManagedResource 注释。 每个 方法你想暴露作为一个操作必须注明 ManagedOperation 注释和每个属性你 希望揭露必须注明 ManagedAttribute 注释。 当标记 属性可以省略要么注释的 getter或 setter来创建一个只写或只读属性 分别。

下面的例子显示了带注释的版本的 JmxTestBean 类,你之前看到的:

```
package org.springframework.jmx;

import org.springframework.jmx.export.annotation.ManagedResource;
import org.springframework.jmx.export.annotation.ManagedOperation;
import org.springframework.jmx.export.annotation.ManagedAttribute;

@ManagedResource(objectName="bean:name=testBean4", description="My Managed Bean", log=true,
   logFile="jmx.log", currencyTimeLimit=15, persistPolicy="OnUpdate", persistPeriod=200,
   persistLocation="foo", persistName="bar")
public class AnnotationTestBean implements IJmxTestBean {

  private String name;
  private int age;

  @ManagedAttribute(description="The Age Attribute", currencyTimeLimit=15)
  public int getAge() {
    return age;
  }

  public void setAge(int age) {
    this.age = age;
  }

  @ManagedAttribute(description="The Name Attribute",
     currencyTimeLimit=20,
     defaultValue="bar",
     persistPolicy="OnUpdate")
  public void setName(String name) {
    this.name = name;
  }

  @ManagedAttribute(defaultValue="foo", persistPeriod=300)
  public String getName() {
    return name;
  }

  @ManagedOperation(description="Add two numbers")
  @ManagedOperationParameters({
    @ManagedOperationParameter(name = "x", description = "The first number"),
    @ManagedOperationParameter(name = "y", description = "The second number")})
  public int add(int x, int y) {
    return x + y;
```

```
  }

  public void dontExposeMe() {
    throw new RuntimeException();
  }
}
```

在这里你可以看到 JmxTestBean 类 标记着 ManagedResource 注释和 这 ManagedResource 注释配置 与一组属性。 这些属性可以用来配置 各种方面的MBean所生成 MBeanExporter 和进一步的解释 稍后在节 SectionA 24 3 3,一个源代码级别的元数据Typesa 。

您还将注意到双方 年龄 和 名称 属性的注释 ManagedAttribute 注释,但对于 这个 年龄 财产,只有getter被标记。 这 会导致这两种属性是包含在管理吗 接口作为属性,但是 年龄 属性将 是只读的。

最后,你会发现 添加(int,int) 方法是标记着 ManagedOperation 属性而 dontExposeMe() 方法不。 这将导致管理接口只包含一个操作, 添加(int,int) ,当使用 MetadataMBeanInfoAssembler 。

下面的配置显示了如何配置 MBeanExporter 使用 MetadataMBeanInfoAssembler :

```
<beans>
  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="assembler" ref="assembler"/>
    <property name="namingStrategy" ref="namingStrategy"/>
    <property name="autodetect" value="true"/>
  </bean>

  <bean id="jmxAttributeSource"
      class="org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource"/>

  <!-- will create management interface using annotation metadata -->
  <bean id="assembler"
      class="org.springframework.jmx.export.assembler.MetadataMBeanInfoAssembler">
    <property name="attributeSource" ref="jmxAttributeSource"/>
  </bean>

  <!-- will pick up the ObjectName from the annotation -->
  <bean id="namingStrategy"
      class="org.springframework.jmx.export.naming.MetadataNamingStrategy">
    <property name="attributeSource" ref="jmxAttributeSource"/>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.AnnotationTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>
</beans>
```

在这里你可以看到一个 MetadataMBeanInfoAssembler bean已经 配置的一个实例 AnnotationJmxAttributeSource 类和传递给 这个 MBeanExporter 通过汇编程序属性。 这是所有被要求利用元数据驱动的 管理接口你春天暴露的mbean。

24.3.3A源代码级别的元数据类型

下面的源代码级元数据类型都可以使用的 Spring JMX:

24.2为多。 一个源代码级别的元数据类型

目的	注释	注释类型
马克的所有实例 类 作为 JMX管理的资源	@ManagedResource	类
标志着作为一个JMX操作方法	@ManagedOperation	方法
马克一个getter或setter作为JMX的一半 属性	@ManagedAttribute	方法(只有getter和setter)
定义描述操作参数	@ManagedOperationParameter 和 @ManagedOperationParameters	方法

下列配置参数都可以使用 这些源代码级别的元数据类型:

24.3为多。 一个源代码级别的元数据参数

参数	描述	适用于
ObjectName	使用 MetadataNamingStrategy 确定 ObjectName 的 托管资源	ManagedResource
描述	集描述资源的友好, 属性或操作	ManagedResource , ManagedAttribute , ManagedOperation , ManagedOperationParameter
currencyTimeLimit	设置的值 currencyTimeLimit 描述符字段	ManagedResource , ManagedAttribute
defaultvalue	设置的值 defaultvalue 描述符字段	ManagedAttribute
日志	设置的值 日志 描述符 领域	ManagedResource
日志文件	设置的值 日志文件 描述符字段	ManagedResource
persistPolicy	设置的值 persistPolicy 描述符字段	ManagedResource
persistPeriod	设置的值 persistPeriod 描述符字段	ManagedResource
persistLocation	设置的值 persistLocation 描述符字段	ManagedResource
persistName	设置的值 persistName 描述符字段	ManagedResource
名称	设置显示名称的操作参数	ManagedOperationParameter
指数	集索引的操作参数	ManagedOperationParameter

24.3.4A了 AutodetectCapableMBeanInfoAssembler 接口

为了简化配置甚至进一步,春天介绍了 AutodetectCapableMBeanInfoAssembler 接口 它扩展 MBeanInfoAssembler 界面添加支持自动MBean资源。 如果你 配置 MBeanExporter 的一个实例 AutodetectCapableMBeanInfoAssembler 然后它是 允许“投票”包含bean因接触到JMX。

盒子之外,只有实现的 AutodetectCapableMBeanInfo 接口是 MetadataMBeanInfoAssembler 将投票 包括任何bean这标记着 ManagedResource 属性。 默认的方法 在这种情况下是使用bean的名称 ObjectName 这导致一个配置怎么样 这个:

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <!-- notice how no 'beans' are explicitly configured here -->
    <property name="autodetect" value="true"/>
    <property name="assembler" ref="assembler"/>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

  <bean id="assembler" class="org.springframework.jmx.export.assembler.MetadataMBeanInfoAssembler">
    <property name="attributeSource">
        <bean class="org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource"/>
    </property>
  </bean>

</beans>
```

注意,在这个配置没有豆子被传递到 MBeanExporter ,然而, JmxTestBean 仍将注册因为它是吗 注明 ManagedResource 属性和 MetadataMBeanInfoAssembler 检测这和选票 包括它。 唯一的问题在于,这个名字的 这个 JmxTestBean 现在的业务含义。 你可以 解决这个问题通过更改默认行为 ObjectName 创作中定义 SectionA 24.4,一个控制 ObjectName 对你的beansas 。

24.3.5A定义管理接口使用Java接口

除了 MetadataMBeanInfoAssembler ,春天还包括 这个 InterfaceBasedMBeanInfoAssembler 允许 你来约束方法和属性暴露基于 一组方法定义在接口的集合。

虽然标准的机制是使用mbean暴露 接口和一个简单的命名方案, InterfaceBasedMBeanInfoAssembler 扩展这个 功能通过消除需要命名约定,允许你 使用多个接口和删除需要您的bean MBean接口实现。

考虑这个接口,用于定义一个管理 界面 JmxTestBean 类,你看到 早:

```
public interface IJmxTestBean {

  public int add(int x, int y);

  public long myOperation();

  public int getAge();

  public void setAge(int age);

  public void setName(String name);

  public String getName();
}
```

该接口定义的方法和属性,将 公开为操作和属性在JMX MBean。 下面的代码 展示了如何配置Spring JMX使用这个接口 定义的管理界面:

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean5" value-ref="testBean"/>
      </map>
    </property>
    <property name="assembler">
      <bean class="org.springframework.jmx.export.assembler.InterfaceBasedMBeanInfoAssembler">
        <property name="managedInterfaces">
          <value>org.springframework.jmx.IJmxTestBean</value>
        </property>
      </bean>
    </property>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

</beans>
```

在这里你可以看到 InterfaceBasedMBeanInfoAssembler 配置为 使用 IJmxTestBean 接口当 构建管理界面对于任何bean。 它是重要的 明白bean处理 InterfaceBasedMBeanInfoAssembler 是 不 必须实现该接口用来 生成JMX管理界面。

在上述的情况下, IJmxTestBean 接口是用来构建所有管理界面为所有bean。 在很多情况下这是不希望出现的行为,你可能想要使用 不同的接口不同的豆子。 在这种情况下,您可以通过 InterfaceBasedMBeanInfoAssembler 一个 属性 实例通过 interfaceMappings 物业,其中关键的每个 入口是bean名称和值的每个条目是一个以逗号分隔的 列表的接口名称使用的 bean。

如果没有管理界面是通过指定 managedInterfaces 或 interfaceMappings 属性,那么 InterfaceBasedMBeanInfoAssembler 将反映在 bean和使用所有的接口实现,bean 创建管理界面。

24.3.6A使用 MethodNameBasedMBeanInfoAssembler

这个 MethodNameBasedMBeanInfoAssembler 允许你指定一个方法名的清单,将会暴露在JMX 作为属性和操作。 下面的代码显示了一个示例 配置:

```
<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
  <property name="beans">
    <map>
      <entry key="bean:name=testBean5" value-ref="testBean"/>
    </map>
  </property>
  <property name="assembler">
    <bean class="org.springframework.jmx.export.assembler.MethodNameBasedMBeanInfoAssembler">
      <property name="managedMethods">
        <value>add,myOperation,getName,setName,getAge</value>
      </property>
    </bean>
  </property>
</bean>
```

在这里你可以看到这个方法 添加 和 myOperation 将公开为JMX操作和 getName() , setName(字符串) 和 getAge() 将公开为适当的一半的一个吗 JMX属性。 在上面的代码中,该方法适用于bean映射 这是暴露于JMX。 控制方法暴露在一个bean 基础上,使用 methodMappings 财产的 MethodNameMBeanInfoAssembler bean名称映射 列表的方法名称。

24.4一个控制 ObjectName 年代为你豆

在幕后, MBeanExporter 代表的一个实现 ObjectNamingStrategy 获得 ObjectName 年代的每个bean是 注册。 默认实现, KeyNamingStrategy ,将缺省使用的关键 这个 bean 地图 随着 ObjectName 。 此外, KeyNamingStrategy 可以映射的关键吗 bean 地图 一个条目在 属性 文件(或文件)来解决 ObjectName 。 除了 KeyNamingStrategy ,弹簧提供了两个额外的 ObjectNamingStrategy 实现: IdentityNamingStrategy 建立一个 ObjectName 基于JVM的身份的bean 和 MetadataNamingStrategy 使用源 级元数据来获得 ObjectName 。

24.4.1A阅读 ObjectName 年代从 属性

您可以配置自己的 KeyNamingStrategy 实例和配置它 读 ObjectName 年代从一个 属性 使用bean实例,而不是关键。 这个 KeyNamingStrategy 将试图找到一个条目吗 在 属性 与一个键对应 bean的关键。 如果没有这样的条目存在或如果 属性 实例空 然后bean关键本身是使用。

下面的代码显示了一个示例配置 KeyNamingStrategy :

```
<beans>

 <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
  <property name="beans">
   <map>
    <entry key="testBean" value-ref="testBean"/>
   </map>
  </property>
  <property name="namingStrategy" ref="namingStrategy"/>
 </bean>

 <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
  <property name="name" value="TEST"/>
  <property name="age" value="100"/>
 </bean>

 <bean id="namingStrategy" class="org.springframework.jmx.export.naming.KeyNamingStrategy">
  <property name="mappings">
   <props>
    <prop key="testBean">bean:name=testBean1</prop>
   </props>
  </property>
  <property name="mappingLocations">
   <value>names1.properties,names2.properties</value>
  </property>
 </bean

</beans>
```

这里的一个实例 KeyNamingStrategy 是 配置一个 属性 实例 合并从 属性 实例定义为 映射属性和属性文件位于路径 属性定义的映射。 在这个配置中, testBean bean将获得 ObjectName 豆:name = testBean1 因为这是中的条目 属性 实例有一个键对应于bean关键。

如果没有条目 属性 实例可以 被发现然后bean键名用作 ObjectName 。

24.4.2A使用 MetadataNamingStrategy

这个 MetadataNamingStrategy 使用 这个 ObjectName 财产的 ManagedResource 属性在每个bean来创建 这个 ObjectName 。 下面的代码显示了 配置 MetadataNamingStrategy :

```
<beans>

 <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
  <property name="beans">
   <map>
    <entry key="testBean" value-ref="testBean"/>
   </map>
  </property>
  <property name="namingStrategy" ref="namingStrategy"/>
 </bean>

 <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
  <property name="name" value="TEST"/>
  <property name="age" value="100"/>
```

```
  </bean>

  <bean id="namingStrategy" class="org.springframework.jmx.export.naming.MetadataNamingStrategy">
    <property name="attributeSource" ref="attributeSource"/>
  </bean>

  <bean id="attributeSource"
      class="org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource"/>

</beans>
```

如果没有 ObjectName 提供了 这个 ManagedResource 属性,然后一个 ObjectName 将创建的吗 以下格式: (完全合格的包名称]:type =[简短名称)、名称=[bean名称] 。 例如,生成的 ObjectName 为 以下bean将: com foo:type = = myBean MyClass、名称 。

```
<bean id="myBean" class="com.foo.MyClass"/>
```

24.4.3A配置基于注解的MBean的出口

如果你喜欢使用 基于注解的 方法 定义你的管理接口,然后一个便利的子类 MBeanExporter 是可用的: AnnotationMBeanExporter 。 在定义这个子类的一个实例, namingStrategy , 汇编 ,和 attributeSource 配置不再需要,因为它总是使用标准的Java 基于注解的元数据(自动总是启用)。 事实上, 而不是定义一个 MBeanExporter 豆,一个甚至 简单的语法支持 @EnableMBeanExport @ configuration 注释。

```
@Configuration
@EnableMBeanExport
public class AppConfig {

}
```

如果你更喜欢基于XML配置 “ 背景:mbean出口” 元素具有同样目的。

```
<context:mbean-export/>
```

你可以提供一个参考特定的MBean 服务器 如果 必要的, defaultDomain 属性 (一个属性的 AnnotationMBeanExporter) 接受一个备用价值生成的MBean ObjectNames “域。 这将使用 在地方的完全限定包名称中所描述的一样 在前一节 MetadataNamingStrategy 。

```
@EnableMBeanExport(server="myMBeanServer", defaultDomain="myDomain")
@Configuration
ContextConfiguration {

}
```

```
<context:mbean-export server="myMBeanServer" default-domain="myDomain"/>
```

注意

不要使用基于接口的AOP代理结合自动吗 JMX注释你的bean类。 基于接口的代理 “隐藏” 目标类, 这也隐藏了JMX管理资源注释。 因此,使用目标类代理 在这种情况下:通过设置 “代理目标类的国旗 < aop:config / > , < tx:注解驱动/ > 等。否则,你的JMX豆子 可能是在启动时悄悄地忽略.....

24.5一个jsr - 160连接器

对于远程访问,Spring JMX模块提供两种 FactoryBean 实现在 org.springframework.jmx.support 包创建 客户端和服务器端连接器。

24.5.1A端连接器

有Spring JMX创建、启动和揭露一个jsr - 160 JMXConnectorServer 使用以下 配置:

```
<bean id="serverConnector" class="org.springframework.jmx.support.ConnectorServerFactoryBean"/>
```

默认情况下 ConnectorServerFactoryBean 创建一个 JMXConnectorServer 绑定到 “服务:jmx:jmxmp:/ / localhost:9875” 。 这个 serverConnector 豆因此暴露了地方 MBeanServer 客户通过JMXMP协议 在本地主机、端口9875。 注意,JMXMP协议被标记为 可选的JSR 160规范:目前,主要的开源 JMX实现,MX4J,提供J2SE 5.0做的 不 JMXMP支持。

指定另一个URL和注册 JMXConnectorServer 本身与 MBeanServer 使用 serviceUrl 和 ObjectName 属性分别为:

```
<bean id="serverConnector"
    class="org.springframework.jmx.support.ConnectorServerFactoryBean">
  <property name="objectName" value="connector:name=rmi"/>
  <property name="serviceUrl"
        value="service:jmx:rmi://localhost/jndi/rmi://localhost:1099/myconnector"/>
</bean>
```

如果 ObjectName 属性设置弹簧 将自动注册您的连接器与 MBeanServer 在这 ObjectName 。 下面的例子显示了全套 参数可以传递的 ConnectorServerFactoryBean 当创建一个 JMXConnector:

```
<bean id="serverConnector"
    class="org.springframework.jmx.support.ConnectorServerFactoryBean">
  <property name="objectName" value="connector:name=iiop"/>
  <property name="serviceUrl"
          value="service:jmx:iiop://localhost/jndi/iiop://localhost:900/myconnector"/>
  <property name="threaded" value="true"/>
  <property name="daemon" value="true"/>
  <property name="environment">
    <map>
      <entry key="someKey" value="someValue"/>
    </map>
  </property>
</bean>
```

注意,当使用一个基于rmi连接器你需要查找 服务(tnameserv或rmiregistry)要开始为了这个名字 注册完成。 如果您使用的是弹簧出口远程 为你服务通过RMI,然后春天就已经构建了一个 RMI注册表。 如果不是,你可以很容易地开始一个注册表使用 以下代码片段的配置:

```
<bean id="registry" class="org.springframework.remoting.rmi.RmiRegistryFactoryBean">
  <property name="port" value="1099"/>
</bean>
```

24.5.2A端连接器

创建一个 MBeanServerConnection发起 一个 启用远程jsr - 160 MBeanServer 使用 MBeanServerConnectionFactoryBean 如图所示 下图:

```
<bean id="clientConnector" class="org.springframework.jmx.support.MBeanServerConnectionFactoryBean">
  <property name="serviceUrl" value="service:jmx:rmi://localhost/jndi/rmi://localhost:1099/jmxrmi"/>
</bean>
```

24.5.3A JMX在麻袋/黑森/ SOAP

jsr - 160允许扩展通信方式 完成客户端和服务器之间。 上面的示例是使用 强制性要求的基于rmi实现jsr - 160规范 (IIOP和JRMP)和(可选)JMXMP。 通过使用其他供应商或 JMX实现(如 MX4J)您可以利用 的协议,如肥皂、粗麻布,麻袋在简单的HTTP或SSL和 其他:

```
<bean id="serverConnector" class="org.springframework.jmx.support.ConnectorServerFactoryBean">
  <property name="objectName" value="connector:name=burlap"/>
  <property name="serviceUrl" value="service:jmx:burlap://localhost:9874"/>
</bean>
```

对于上面的示例中,使用MX4J 3.0.0;看到 官方MX4J文档了解更多信息。

24.6通过代理访问mbean

Spring JMX允许您创建代理,对光调用 mbean注册在本地或远程 MBeanServer 。 这些代理为您提供一个标准的Java接口,通过该你 可以与你进行mbean。 下面的代码展示了如何配置一个 代表一个MBean运行在一个地方 MBeanServer :

```
<bean id="proxy" class="org.springframework.jmx.access.MBeanProxyFactoryBean">
   <property name="objectName" value="bean:name=testBean"/>
   <property name="proxyInterface" value="org.springframework.jmx.IJmxTestBean"/>
</bean>
```

在这里你可以看到,创建一个代理注册的MBean 在 ObjectName : 豆:name = testBean 。 接口的集合 代理将执行的控制 proxyInterfaces 属性和规则映射 方法和属性在这些接口来操作和属性 MBean是相同的使用规则 InterfaceBasedMBeanInfoAssembler 。

这个 MBeanProxyFactoryBean 可以创建一个代理 任何MBean可以通过 MBeanServerConnection发起 。 默认情况下,当地的 MBeanServer 的位置和使用,但是你可以吗 覆盖这个和提供一个 MBeanServerConnection发起 指向远程 MBeanServer 以满足 代理指向远程mbean:

```
<bean id="clientConnector"
    class="org.springframework.jmx.support.MBeanServerConnectionFactoryBean">
  <property name="serviceUrl" value="service:jmx:rmi://remotehost:9875"/>
</bean>

<bean id="proxy" class="org.springframework.jmx.access.MBeanProxyFactoryBean">
  <property name="objectName" value="bean:name=testBean"/>
  <property name="proxyInterface" value="org.springframework.jmx.IJmxTestBean"/>
  <property name="server" ref="clientConnector"/>
</bean>
```

在这里你可以看到,我们创建一个 MBeanServerConnection发起 指向一个远程机器 使用 MBeanServerConnectionFactoryBean 。 这 MBeanServerConnection发起 随后被传递到吗 MBeanProxyFactoryBean 通过 服务器 财产。 代理创建将前进 所有调用 MBeanServer 通过这 MBeanServerConnection发起 。

24.7一个通知

Spring的JMX提供全面支持JMX 通知。

24.7.1A登记侦听器进行通知

Spring的JMX支持使得它很容易注册任意数量的 NotificationListeners 与任何数量的mbean (这包括由弹簧的mbean出口 MBeanExporter 并通过一些注册的mbean 其他的机制)。 通过一个例子,考虑一个场景 想被告知(通过一个吗 通知) 每个时间属性的目标MBean的变化。

```
package com.example;

import javax.management.AttributeChangeNotification;
import javax.management.Notification;
import javax.management.NotificationFilter;
import javax.management.NotificationListener;

public class ConsoleLoggingNotificationListener
            implements NotificationListener, NotificationFilter {

    public void handleNotification(Notification notification, Object handback) {
        System.out.println(notification);
        System.out.println(handback);
    }

    public boolean isNotificationEnabled(Notification notification) {
        return AttributeChangeNotification.class.isAssignableFrom(notification.getClass());
    }
}
```

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean1" value-ref="testBean"/>
      </map>
    </property>
    <property name="notificationListenerMappings">
      <map>
        <entry key="bean:name=testBean1">
          <bean class="com.example.ConsoleLoggingNotificationListener"/>
        </entry>
      </map>
    </property>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

</beans>
```

与上面的配置到位,每次一个JMX 通知 是广播从目标MBean吗 (豆:name = testBean1), ConsoleLoggingNotificationListener bean 作为一个监听器注册通过 notificationListenerMappings 属性将 通知。 这个 ConsoleLoggingNotificationListener bean可以采取任何行动它认为适当的回应 这个 通知 。

你也可以直接使用bean名称之间的联系出口豆类 和听众:

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean1" value-ref="testBean"/>
      </map>
    </property>
    <property name="notificationListenerMappings">
      <map>
        <entry key="testBean">
          <bean class="com.example.ConsoleLoggingNotificationListener"/>
        </entry>
      </map>
    </property>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

</beans>
```

如果一个人想要注册一个 一个NotificationListener 实例的所有bean,封闭 MBeanExporter 出口,可以使用特殊的通配符 ‘*’ (不含引号) 作为关键的一个条目 notificationListenerMappings 属性映射,例如:

```
<property name="notificationListenerMappings">
  <map>
    <entry key="*">
      <bean class="com.example.ConsoleLoggingNotificationListener"/>
    </entry>
  </map>
</property>
```

如果一个人需要做逆(即注册一个数量的不同 听众对一个MBean),那么必须使用 NotificationListeners (在列表属性相反 偏好的 notificationListenerMappings 属性)。 这一次,而不是配置简单 一个NotificationListener 对于一个MBean,一个 配置 NotificationListenerBean 实例... 一个 NotificationListenerBean 封装了一个 一个NotificationListener 和 ObjectName (或 ObjectNames),它是对注册 在一个 MBeanServer 。 这个 NotificationListenerBean 也展示了一个 数量的其他性质如 NotificationFilter 和一个任意handback 对象,可以使用JMX通知场景在先进。

配置在使用 NotificationListenerBean 实例不是疯狂 不同的是先前展示:

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean1" value-ref="testBean"/>
      </map>
    </property>
    <property name="notificationListeners">
      <list>
        <bean class="org.springframework.jmx.export.NotificationListenerBean">
          <constructor-arg>
            <bean class="com.example.ConsoleLoggingNotificationListener"/>
          </constructor-arg>
          <property name="mappedObjectNames">
            <list>
              <value>bean:name=testBean1</value>
            </list>
          </property>
        </bean>
      </list>
    </property>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

</beans>
```

上面的例子是相当于第一个通知的例子。 假设我们想要那么给定handback对象每次 通知 提高,另外我们吗 要过滤掉无关 通知 通过 提供一个 NotificationFilter 。 (满一 讨论什么handback对象是,甚而一个 NotificationFilter 是,请做参考吗 部分的 JMX规范(1.2)资格 的JMX 通知模型”)。

```
<beans>

 <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
  <property name="beans">
   <map>
    <entry key="bean:name=testBean1" value-ref="testBean1"/>
    <entry key="bean:name=testBean2" value-ref="testBean2"/>
   </map>
  </property>
  <property name="notificationListeners">
    <list>
      <bean class="org.springframework.jmx.export.NotificationListenerBean">
        <constructor-arg ref="customerNotificationListener"/>
        <property name="mappedObjectNames">
          <list>
            <!-- handles notifications from two distinct MBeans -->
            <value>bean:name=testBean1</value>
            <value>bean:name=testBean2</value>
          </list>
        </property>
        <property name="handback">
          <bean class="java.lang.String">
            <constructor-arg value="This could be anything..."/>
          </bean>
        </property>
        <property name="notificationFilter" ref="customerNotificationListener"/>
      </bean>
    </list>
  </property>
 </bean>

 <!-- implements both the NotificationListener and NotificationFilter interfaces -->
 <bean id="customerNotificationListener" class="com.example.ConsoleLoggingNotificationListener"/>

 <bean id="testBean1" class="org.springframework.jmx.JmxTestBean">
  <property name="name" value="TEST"/>
  <property name="age" value="100"/>
 </bean>

 <bean id="testBean2" class="org.springframework.jmx.JmxTestBean">
  <property name="name" value="ANOTHER TEST"/>
  <property name="age" value="200"/>
 </bean>

</beans>
```

24.7.2A发布通知

春天不仅提供了支持注册接收 通知 ,也为出版 通知 。

注意

请注意,这部分是真正唯一有关的春天 托管bean,暴露出来时通过一个mbean MBeanExporter ,任何现有的、用户定义的 mbean应该使用标准的JMX api对通知的发布。

关键的接口在Spring的JMX通知发布支持 是 NotificationPublisher 接口(定义 在 org.springframework.jmx.export.notification 包)。 任何bean将被导出为一个MBean通过 MBeanExporter 实例可以实现相关 NotificationPublisherAware 接口获得 访问 NotificationPublisher 实例。 这个 NotificationPublisherAware 接口简单 供应的一个实例 NotificationPublisher 到实现bean通过一个简单的setter方法,该bean可以 然后使用发布 通知 。

所述的Javadoc NotificationPublisher 类,托管bean, 发布事件是通过 NotificationPublisher 机制是 不 负责国家管理的任何通知侦听器之类的..... Spring的JMX支持需要 负责处理所有的JMX基础设施问题。 所有人需要做 应用程序开发人员是实现 NotificationPublisherAware 界面,开始 使用提供的发布事件 NotificationPublisher 实例。 注意, NotificationPublisher 将被设置 在 托管bean已经注册了 MBeanServer 。

使用 NotificationPublisher 实例 很简单的... 一个简单的创建一个JMX 通知 实例(或的一个实例 适当 通知 子类),填充 通知与数据相关的事件的 发表,和一个然后调用 sendNotification(通知) 在 NotificationPublisher 实例,传递 通知 。

在下面发现一个简单的例子... 在这种情况下,出口 实例的 JmxTestBean 将要发布 一个 NotificationEvent 每次 添加(int,int) 操作被调用。

```
package org.springframework.jmx;

import org.springframework.jmx.export.notification.NotificationPublisherAware;
import org.springframework.jmx.export.notification.NotificationPublisher;
import javax.management.Notification;
```

```
public class JmxTestBean implements IJmxTestBean, NotificationPublisherAware {

    private String name;
    private int age;
    private boolean isSuperman;
    private NotificationPublisher publisher;

    // other getters and setters omitted for clarity

    public int add(int x, int y) {
        int answer = x + y;
        this.publisher.sendNotification(new Notification("add", this, 0));
        return answer;
    }

    public void dontExposeMe() {
        throw new RuntimeException();
    }

    public void setNotificationPublisher(NotificationPublisher notificationPublisher) {
        this.publisher = notificationPublisher;
    }
}
```

这个 NotificationPublisher 接口和 机械来把一切工作是一个很不错的功能Spring的JMX支持。 然而它确实有价格标签类的耦合两个弹簧和JMX; 总是,我们的建议就是要务实..... 如果你需要提供的功能 NotificationPublisher 和你可以接受两个弹簧的耦合 和JMX,那么这样做。

24.8进一步资源

这部分包含关于JMX其它资源的链接。

- 这个 JMX主页 在太阳
- 这个 JMX规范 (jsr - 000003)
- 这个 JMX远程API规范 (jsr - 000160)
- 这个 MX4J 主页 (一个开源的实现各种JMX 规格)
- 开始使用JMX ——一篇介绍性文章从太阳。

25。 JCA CCI

25.1一个介绍

Java EE规范提供了一个标准化访问企业 信息系统(EIS):JCA(J2EE连接器体系结构)。 这 规范分为几个不同的部分:

- SPI(服务提供者接口),连接器供应商 必须实现。 这些接口构成的资源适配器 可以部署在一个Java EE应用服务器。 在这种情况下, 服务器管理连接池、事务和安全(托管 模式)。 应用程序服务器还负责管理 配置,这是外面举行客户机应用程序。 一个 连接器可以不使用应用服务器;在这 情况下,应用程序必须配置它直接(非受管 模式)。
- CCI(常见的客户端接口),一个应用程序可以使用 与连接器,因此与EIS。 一个API 对当地事务界定也提供。

春天的目的是提供类CCI支持访问 CCI连接器在典型的春天的风格,利用Spring框架的 通用资源和事务管理设施。

> **注意**
>
> 客户端连接器并不总是使用CCI。 一些 连接器暴露自己的api,只提供JCA资源适配器来 使用系统合同的一个Java EE容器(连接池、全球 事务、安全)。 春天不提供特殊支持这样的 连接器特定api。

25.2一个配置CCI

25.2.1A连接器配置

基础资源使用JCA CCI是 ConnectionFactory 接口。 这个 连接器使用必须提供该接口的一个实现。

使用连接器,您可以将其部署在您的应用程序 服务器并获取 ConnectionFactory 从服务器的JNDI环境(管理模式)。 连接器必须包装成一个RAR文件(资源适配器存档),包含一个 ra xml 文件来描述它的部署 特征。 实际的名字是指定的资源,当你 部署它。 来访问它在春天,简单地使用Spring的 JndiObjectFactoryBean / < jee:jndi查找> 获取工厂的JNDI 的名字。

另一种使用连接器是嵌入在你的应用程序 (非受管模式),而不是使用一个应用服务器来部署和 配置它。 Spring提供了这种可能性来配置一个连接器作为 一个豆,通过提供的 FactoryBean (LocalConnectionFactoryBean)。 用这种方式,你 只需要连接器图书馆在类路径中(没有RAR文件也没有 ra xml 描述符需要)。 图书馆必须 提取的连接器的RAR文件,如果必要的。

一旦你有访问你 ConnectionFactory 实例,您可以 其注入你的组件。 这些组件可以是编码 针对平原CCI API或利用Spring的支持类,CCI 访问(例如。 CciTemplate)。

注意

当你使用一个连接器在非受管模式下,你不能使用 全球事务,因为资源是从未参军/退市 在当前全球事务的当前线程。 资源 只是不知道任何全球Java EE的事务,可能吗 运行。

25.2.2A ConnectionFactory 配置在 春天

为了使连接到EIS,您需要获得一个 ConnectionFactory 从应用程序 服务器如果你在托管模式下,或直接从春天如果你 在非受管模式。

在托管模式下,你访问 ConnectionFactory 从JNDI; 属性将应用程序服务器中配置的。

```
<jee:jndi-lookup id="eciConnectionFactory" jndi-name="eis/cicseci"/>
```

在非受管模式下,您必须配置 ConnectionFactory 你想要使用的 春天的配置作为一个JavaBean。 这个 LocalConnectionFactoryBean 类提供了这 设置风格,传递 ManagedConnectionFactory 实现你的 连接器,暴露出应用层CCI ConnectionFactory 。

```
<bean id="eciManagedConnectionFactory" class="com.ibm.connector2.cics.ECIManagedConnectionFactory">
  <property name="serverName" value="TXSERIES"/>
  <property name="connectionURL" value="tcp://localhost/"/>
  <property name="portNumber" value="2006"/>
</bean>

<bean id="eciConnectionFactory" class="org.springframework.jca.support.LocalConnectionFactoryBean">
  <property name="managedConnectionFactory" ref="eciManagedConnectionFactory"/>
</bean>
```

注意

你不能直接实例化一个特定的 ConnectionFactory 。 你需要去 通过相应的实施 ManagedConnectionFactory 界面 连接器。 这个接口是部分的JCA SPI规范。

25.2.3A CCI连接配置

JCA CCI允许开发人员配置连接 EIS使用 ConnectionSpec 实现你的连接器。 为了配置它的属性, 你需要把目标连接工厂和一个专用的适配器, ConnectionSpecConnectionFactoryAdapter 。 所以, 专用 ConnectionSpec 可以 配置属性 connectionSpec (作为一个 内心的bean)。

这个属性不是强制性的,因为CCI ConnectionFactory 接口定义了两个 不同的方法来获得一个CCI连接。 一些 connectionSpec 属性通常可以 应用程序服务器中配置的(在托管模式)或 相应的当地 ManagedConnectionFactory 实现。

```
public interface ConnectionFactory implements Serializable, Referenceable {
  ...
  Connection getConnection() throws ResourceException;
  Connection getConnection(ConnectionSpec connectionSpec) throws ResourceException;
  ...
}
```

弹簧提供了一个 ConnectionSpecConnectionFactoryAdapter 这 允许指定一个 connectionSpec 实例使用的所有操作在一个给定的工厂。 如果适配器的 connectionSpec 财产被指定,适配器 使用 getConnection 变体与 connectionSpec 参数,否则没有争论的变体。

```
<bean id="managedConnectionFactory"
   class="com.sun.connector.cciblackbox.CciLocalTxManagedConnectionFactory">
  <property name="connectionURL" value="jdbc:hsqldb:hsql://localhost:9001"/>
  <property name="driverName" value="org.hsqldb.jdbcDriver"/>
</bean>

<bean id="targetConnectionFactory"
```

```
  class="org.springframework.jca.support.LocalConnectionFactoryBean">
 <property name="managedConnectionFactory" ref="managedConnectionFactory"/>
</bean>

<bean id="connectionFactory"
  class="org.springframework.jca.cci.connection.ConnectionSpecConnectionFactoryAdapter">
 <property name="targetConnectionFactory" ref="targetConnectionFactory"/>
 <property name="connectionSpec">
  <bean class="com.sun.connector.cciblackbox.CciConnectionSpec">
   <property name="user" value="sa"/>
   <property name="password" value=""/>
  </bean>
 </property>
</bean>
```

25.2.4A使用单一CCI连接

如果你想使用一个单一的CCI连接,弹簧提供了 进一步 ConnectionFactory 适配器 管理这个。 这个 SingleConnectionFactory 适配器 类将打开一个连接懒洋洋地并关闭它当这个bean 在应用程序关闭被摧毁。 这个类将提供特殊的 连接 代理行为 因此,所有共享相同的底层物理连接。

```
<bean id="eciManagedConnectionFactory"
  class="com.ibm.connector2.cics.ECIManagedConnectionFactory">
 <property name="serverName" value="TEST"/>
 <property name="connectionURL" value="tcp://localhost/"/>
 <property name="portNumber" value="2006"/>
</bean>

<bean id="targetEciConnectionFactory"
  class="org.springframework.jca.support.LocalConnectionFactoryBean">
 <property name="managedConnectionFactory" ref="eciManagedConnectionFactory"/>
</bean>

<bean id="eciConnectionFactory"
  class="org.springframework.jca.cci.connection.SingleConnectionFactory">
 <property name="targetConnectionFactory" ref="targetEciConnectionFactory"/>
</bean>
```

注意

这 ConnectionFactory 适配器 不能直接被配置 connectionSpec 。 使用一个中介 ConnectionSpecConnectionFactoryAdapter , SingleConnectionFactory 如果你需要谈判 一个单一的连接为一个特定的 connectionSpec 。

25.3一个使用Spring的CCI访问支持

25.3.1A记录转换

的目标之一是提供JCA CCI支持方便 设施操纵CCI记录。 开发人员可以指定 策略创建记录和提取数据从记录、使用 春天的 CciTemplate 。 下面的接口 将配置策略使用输入和输出记录如果你不 愿与您的应用程序中直接记录。

为了创建一个输入 记录 , 开发人员可以使用一个专用的实施 RecordCreator 接口。

```
public interface RecordCreator {

  Record createRecord(RecordFactory recordFactory) throws ResourceException, DataAccessException;
}
```

正如您可以看到的, createRecord(. .) 方法 收到 RecordFactory 实例作为 参数,它对应于 RecordFactory 的 ConnectionFactory 使用。 这个参考 可以用来创建 IndexedRecord 或 MappedRecord 实例。 以下 示例显示了如何使用 RecordCreator 接口和索引/映射记录。

```
public class MyRecordCreator implements RecordCreator {

  public Record createRecord(RecordFactory recordFactory) throws ResourceException {
    IndexedRecord input = recordFactory.createIndexedRecord("input");
    input.add(new Integer(id));
    return input;
  }
}
```

一个输出 记录 可以用来 接收数据从EIS回来。 因此,一个特定的实现 RecordExtractor 接口可以传递 到春天的 CciTemplate 对提取数据从 输出 记录 。

```
public interface RecordExtractor {

  Object extractData(Record record) throws ResourceException, SQLException, DataAccessException;
}
```

下面的示例显示了如何使用 RecordExtractor 接口。

```
public class MyRecordExtractor implements RecordExtractor {

  public Object extractData(Record record) throws ResourceException {
    CommAreaRecord commAreaRecord = (CommAreaRecord) record;
    String str = new String(commAreaRecord.toByteArray());
    String field1 = string.substring(0,6);
    String field2 = string.substring(6,1);
    return new OutputObject(Long.parseLong(field1), field2);
  }
}
```

25.3.2A了 CciTemplate

这个 CciTemplate 是中央类的吗 核心CCI支持包 (org.springframework.jca.cci.core)。 它简化了 使用CCI因为它处理创建和释放资源。 这 有助于避免一些常见错误,如忘记总是关闭 连接。 它关心的生命周期连接和交互 对象,让应用程序代码关注生成输入记录 应用程序数据和应用程序数据提取从输出 记录。

JCA规范定义了两种截然不同的CCI方法调用 EIS的操作。 CCI的 交互 接口提供了两个执行方法签名:

```
public interface javax.resource.cci.Interaction {
  ...
  boolean execute(InteractionSpec spec, Record input, Record output) throws ResourceException;

  Record execute(InteractionSpec spec, Record input) throws ResourceException;
  ...
}
```

根据模板方法调用时, CciTemplate 会知道哪个 执行 方法调用交互。 在任何 情况下,一个正确地初始化 InteractionSpec 实例强制性的。

CciTemplate.execute(. .) 可用于两个吗 方法:

- 与直接 记录 参数。 在这种情况下,您只需要通过CCI输入记录, 返回的对象是对应的CCI输出记录。
- 与应用程序对象,使用记录的映射。 在这种情况下, 你需要提供相应的 RecordCreator 和 RecordExtractor 实例。

对于第一种方法,下列方法的模板 将被使用。 这些方法直接对应于那些 交互 接口。

```
public class CciTemplate implements CciOperations {

  public Record execute(InteractionSpec spec, Record inputRecord)
      throws DataAccessException { ... }

  public void execute(InteractionSpec spec, Record inputRecord, Record outputRecord)
      throws DataAccessException { ... }

}
```

对于第二种方法,我们需要指定记录创建 和记录提取策略作为参数。 接口的使用都是 那些在前一节中描述记录转换。 这个 相应 CciTemplate 方法是 以下:

```
public class CciTemplate implements CciOperations {

  public Record execute(InteractionSpec spec, RecordCreator inputCreator)
      throws DataAccessException { ... }

  public Object execute(InteractionSpec spec, Record inputRecord, RecordExtractor outputExtractor)
      throws DataAccessException { ... }

  public Object execute(InteractionSpec spec, RecordCreator creator, RecordExtractor extractor)
      throws DataAccessException { ... }

}
```

除非 outputRecordCreator 属性设置 在模板(参见下一节),每个方法调用 相应 执行 方法的CCI 交互 有两个参数: InteractionSpec 和输入 记录 ,接收一个输出 记录 作为返回值。

CciTemplate 还提供了方法来创建 IndexRecord 和 MappedRecord 外 RecordCreator 的实现, 通过其

createIndexRecord(. .) 和 createMappedRecord(. .) 方法。 这可以用 在DAO实现来创建 记录 实例传入 相应 CciTemplate.execute(. .) 方法。

```
public class CciTemplate implements CciOperations {

  public IndexedRecord createIndexedRecord(String name) throws DataAccessException { ... }

  public MappedRecord createMappedRecord(String name) throws DataAccessException { ... }

}
```

25.3.3A DAO支持

春天的CCI支持提供了一个抽象类,DAOs, 支持注射的 ConnectionFactory 或 CciTemplate 实例。 类的名称 CciDaoSupport :它提供了简单的 setConnectionFactory 和 setCciTemplate 方法。 在内部,这个类将 创建一个 CciTemplate 实例传入 ConnectionFactory ,使其暴露于 具体的数据访问在子类中实现。

```
public abstract class CciDaoSupport {

  public void setConnectionFactory(ConnectionFactory connectionFactory) { ... }
  public ConnectionFactory getConnectionFactory() { ... }

  public void setCciTemplate(CciTemplate cciTemplate) { ... }
  public CciTemplate getCciTemplate() { ... }

}
```

25.3.4A自动输出记录生成

如果连接器使用只支持 交互执行(. .) 法与输入和 输出记录作为参数(即,它要求所需的输出 记录能够传递,而不是返回一个合适的输出 记录),您可以设置 outputRecordCreator 财产 的 CciTemplate 自动生成一个 输出记录在填JCA连接器,如果响应 收到了。 这个记录将然后返回给调用者的 模板。

这个属性只是拥有一个实现的 RecordCreator 接口,用于, 目的。 这个 RecordCreator 接口有 已经讨论 SectionA 25 3 1,一个conversiona记录 。 这个 outputRecordCreator 财产必须直接 上指定的 CciTemplate 。 这可能是做 在应用程序代码中像这样:

```
cciTemplate.setOutputRecordCreator(new EciOutputRecordCreator());
```

或(推荐)在Spring配置,如果 CciTemplate 作为一个专门的bean配置 实例:

```
<bean id="eciOutputRecordCreator" class="eci.EciOutputRecordCreator"/>

<bean id="cciTemplate" class="org.springframework.jca.cci.core.CciTemplate">
  <property name="connectionFactory" ref="eciConnectionFactory"/>
  <property name="outputRecordCreator" ref="eciOutputRecordCreator"/>
</bean>
```

注意

随着 CciTemplate 类是线程安全的, 它通常会被配置为共享实例。

25.3.5A总结

下面的表总结了机制 CciTemplate 类和相应的方法 呼吁CCI 交互 接口:

25.1为多。 使用的 交互 执行 方法

CciTemplate方法签名	CciTemplate outputRecordCreator 财产	执行方法呼吁CCI 交互
记录执行(InteractionSpec 记录)	没有设置	记录执行(InteractionSpec 记录)
记录执行(InteractionSpec 记录)	集	布尔执行(InteractionSpec记录, 记录)
void execute(InteractionSpec记录, 记录)	没有设置	void execute(InteractionSpec

void execute(InteractionSpec记录, 记录)	没有设置	记录, 记录)
void execute(InteractionSpec记录, 记录)	集	void execute(InteractionSpec 记录, 记录)
记录执行(InteractionSpec RecordCreator)	没有设置	记录执行(InteractionSpec 记录)
记录执行(InteractionSpec RecordCreator)	集	void execute(InteractionSpec 记录, 记录)
记录执行(InteractionSpec记录, RecordExtractor)	没有设置	记录执行(InteractionSpec 记录)
记录执行(InteractionSpec记录, RecordExtractor)	集	void execute(InteractionSpec 记录, 记录)
记录执行(InteractionSpec RecordCreator,RecordExtractor)	没有设置	记录执行(InteractionSpec 记录)
记录执行(InteractionSpec RecordCreator,RecordExtractor)	集	void execute(InteractionSpec 记录, 记录)

25.3.6A使用CCI 连接 和 交互 直接

CciTemplate 还提供了可能 工作直接与CCI连接和交互,以同样的方式 作为 JdbcTemplate 和 JmsTemplate 。 这是有用的,当你想 执行多个操作CCI连接或交互,因为 的例子。

接口 ConnectionCallback 提供CCI 连接 作为参数, 为了执行自定义操作,加上CCI ConnectionFactory , 连接 诞生了。 后者可能有用例如得到一个有关 RecordFactory 实例和创建 索引/映射记录,例如。

```
public interface ConnectionCallback {

  Object doInConnection(Connection connection, ConnectionFactory connectionFactory)
      throws ResourceException, SQLException, DataAccessException;
}
```

接口 InteractionCallback 提供CCI 交互 ,为了 执行自定义操作,加上相应的CCI ConnectionFactory 。

```
public interface InteractionCallback {

  Object doInInteraction(Interaction interaction, ConnectionFactory connectionFactory)
      throws ResourceException, SQLException, DataAccessException;
}
```

注意

InteractionSpec 对象可以 可以在多个模板调用共享或新创建的 在每个回调方法。 这是完全的刀 实现。

25.3.7A例子 CciTemplate 使用

在本节中,使用 CciTemplate 将显示访问CICS ECI模式,与IBM CICS ECI连接器。

首先,一些初始化的CCI InteractionSpec 必须采取措施来指定 这CICS程序访问和如何与其交互。

```
ECIInteractionSpec interactionSpec = new ECIInteractionSpec();
interactionSpec.setFunctionName("MYPROG");
interactionSpec.setInteractionVerb(ECIInteractionSpec.SYNC_SEND_RECEIVE);
```

然后程序可以使用CCI通过弹簧的模板和指定 自定义对象之间的映射和CCI 记录 。

```
public class MyDaoImpl extends CciDaoSupport implements MyDao {

  public OutputObject getData(InputObject input) {
    ECIInteractionSpec interactionSpec = ...;

    OutputObject output = (ObjectOutput) getCciTemplate().execute(interactionSpec,
        new RecordCreator() {
          public Record createRecord(RecordFactory recordFactory) throws ResourceException {
```

```
            return new CommAreaRecord(input.toString().getBytes());
          }
        },
        new RecordExtractor() {
          public Object extractData(Record record) throws ResourceException {
            CommAreaRecord commAreaRecord = (CommAreaRecord)record;
            String str = new String(commAreaRecord.toByteArray());
            String field1 = string.substring(0,6);
            String field2 = string.substring(6,1);
            return new OutputObject(Long.parseLong(field1), field2);
          }
        });

    return output;
  }
}
```

正如前面所讨论的,回调函数可以用来直接操作 CCI连接或交互。

```
public class MyDaoImpl extends CciDaoSupport implements MyDao {

  public OutputObject getData(InputObject input) {
    ObjectOutput output = (ObjectOutput) getCciTemplate().execute(
        new ConnectionCallback() {
          public Object doInConnection(Connection connection, ConnectionFactory factory)
              throws ResourceException {

            // do something...
          }
        });
    }
    return output;
  }
}
```

注意

与 ConnectionCallback , 连接 将管理和使用 封闭的 CciTemplate ,但任何交互 上创建的连接必须由管理回调 实现。

对于一个更具体的回调,您可以实现一个 InteractionCallback 。 传入的 交互 将管理和关闭吗 这个 CciTemplate 在这种情况下。

```
public class MyDaoImpl extends CciDaoSupport implements MyDao {

  public String getData(String input) {
    ECIInteractionSpec interactionSpec = ...;

    String output = (String) getCciTemplate().execute(interactionSpec,
        new InteractionCallback() {
          public Object doInInteraction(Interaction interaction, ConnectionFactory factory)
              throws ResourceException {
            Record input = new CommAreaRecord(inputString.getBytes());
            Record output = new CommAreaRecord();
            interaction.execute(holder.getInteractionSpec(), input, output);
            return new String(output.toByteArray());
          }
        });

    return output;
  }
}
```

对于上述示例,相应的配置 涉及Spring bean可以在非受管模式如下:

```
<bean id="managedConnectionFactory" class="com.ibm.connector2.cics.ECIManagedConnectionFactory">
  <property name="serverName" value="TXSERIES"/>
  <property name="connectionURL" value="local:"/>
  <property name="userName" value="CICSUSER"/>
  <property name="password" value="CICS"/>
</bean>

<bean id="connectionFactory" class="org.springframework.jca.support.LocalConnectionFactoryBean">
  <property name="managedConnectionFactory" ref="managedConnectionFactory"/>
</bean>

<bean id="component" class="mypackage.MyDaoImpl">
  <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

在托管模式(即,在一个Java EE环境), 配置可能看起来如下:

```
<jee:jndi-lookup id="connectionFactory" jndi-name="eis/cicseci"/>

<bean id="component" class="MyDaoImpl">
  <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

25.4一个建模CCI访问操作对象

这个 org.springframework.jca.cci.object 包 包含的支持类,允许您访问EIS不同 风格:通过可重用的操作对象,类似于Spring JDBC 操作对象(参见JDBC章)。 这将通常封装了 CCI API:一个应用级的输入对象将被传递到操作 对象,因此它可以构建输入记录,然后把接收到的 数据记录到一个应用程序级输出对象并返回它。

注意 :这种方法是在内部根据 CciTemplate 类和 RecordCreator / RecordExtractor 接口,重用 机械的春天的核心CCI支持。

25.4.1A MappingRecordOperation

MappingRecordOperation 本质上执行 同样的工作 CciTemplate ,但代表一个 特定的、预先配置的操作作为一个对象。 它提供了两个 模板方法来指定如何将一个输入对象的输入 记录,以及如何将一个输出到输出对象记录(记录 映射):

- createInputRecord(. .) 指定如何 把一个输入对象来输入 记录
- extractOutputData(. .) 指定如何 提取一个输出对象从一个输出 记录

下面是这些方法的签名:

```
public abstract class MappingRecordOperation extends EisOperation {
  ...
  protected abstract Record createInputRecord(RecordFactory recordFactory, Object inputObject)
      throws ResourceException, DataAccessException { ... }

  protected abstract Object extractOutputData(Record outputRecord)
      throws ResourceException, SQLException, DataAccessException { ... }
  ...
}
```

此后,为了执行EIS,您需要使用操作 一个单一的执行方法,传入一个应用级的输入对象 和接收一个应用级的输出对象作为结果:

```
public abstract class MappingRecordOperation extends EisOperation {
  ...
  public Object execute(Object inputObject) throws DataAccessException {
  ...
}
```

正如您可以看到的,相反 CciTemplate 类,这 执行(. .) 方法没有 InteractionSpec 作为参数。 相反, InteractionSpec 是全球的操作。 下面的构造函数必须用来实例化 操作对象与一个特定的 InteractionSpec :

```
InteractionSpec spec = ...;
MyMappingRecordOperation eisOperation = new MyMappingRecordOperation(getConnectionFactory(), spec);
...
```

25.4.2A MappingCommAreaOperation

一些连接器使用记录基于COMMAREA的代表 字节数组包含参数发送到EIS和数据 由它返回。 Spring提供了一个特殊的操作类的工作 直接在COMMAREA而不是记录。 这个 MappingCommAreaOperation 类扩展了 MappingRecordOperation 类来提供这样 特殊COMMAREA的支持。 它含蓄地使用 CommAreaRecord 类作为输入和输出记录 类型,并提供了两个新的方法来转换输入对象到一个 输入和输出COMMAREA COMMAREA到一个输出对象。

```
public abstract class MappingCommAreaOperation extends MappingRecordOperation {
  ...
  protected abstract byte[] objectToBytes(Object inObject)
      throws IOException, DataAccessException;

  protected abstract Object bytesToObject(byte[] bytes)
      throws IOException, DataAccessException;
  ...
}
```

25.4.3A自动输出记录生成

因为每个 MappingRecordOperation 子类是 基于CciTemplate内部,同样的方法来自动生成 输出记录与 CciTemplate 是可用的。 每一个操作对象提供一个相应的 setOutputRecordCreator(. .) 法。 为进一步 信息,请参阅 SectionA 25 3 4,一个自动输

出记录generationa 。

25.4.4A总结

操作对象方法使用记录的方式一样 这个 CciTemplate 类。

25.2为多。 一个使用交互执行方法

MappingRecordOperation 方法签名	MappingRecordOperation outputRecordCreator 财产	执行方法呼吁CCI 交互
对象执行(对象)	没有设置	记录执行(InteractionSpec 记录)
对象执行(对象)	集	布尔执行(InteractionSpec记录, 记录)

25.4.5A例子 MappingRecordOperation 使用

在本节中,使用 MappingRecordOperation 将显示访问吗 数据库与黑箱CCI连接器。

注意

最初版本的连接器是由Java EE SDK(版本1.3),可以从太阳。

首先,一些初始化的CCI InteractionSpec 必须采取措施来指定 哪些SQL请求来执行。 在本例中,我们直接定义方式 将参数的请求记录的方法,CCI 将结果记录到一个实例CCI的 人 类。

```
public class PersonMappingOperation extends MappingRecordOperation {

  public PersonMappingOperation(ConnectionFactory connectionFactory) {
    setConnectionFactory(connectionFactory);
    CciInteractionSpec interactionSpec = new CciConnectionSpec();
    interactionSpec.setSql("select * from person where person_id=?");
    setInteractionSpec(interactionSpec);
  }

  protected Record createInputRecord(RecordFactory recordFactory, Object inputObject)
      throws ResourceException {
    Integer id = (Integer) inputObject;
    IndexedRecord input = recordFactory.createIndexedRecord("input");
    input.add(new Integer(id));
    return input;
  }

  protected Object extractOutputData(Record outputRecord)
      throws ResourceException, SQLException {
    ResultSet rs = (ResultSet) outputRecord;
    Person person = null;
    if (rs.next()) {
      Person person = new Person();
      person.setId(rs.getInt("person_id"));
      person.setLastName(rs.getString("person_last_name"));
      person.setFirstName(rs.getString("person_first_name"));
    }
    return person;
  }
}
```

然后应用程序可以执行操作对象, 个人标识符作为参数。 注意,操作对象可以被设置 为共享实例,因为它是线程安全的。

```
public class MyDaoImpl extends CciDaoSupport implements MyDao {

  public Person getPerson(int id) {
    PersonMappingOperation query = new PersonMappingOperation(getConnectionFactory());
    Person person = (Person) query.execute(new Integer(id));
    return person;
  }
}
```

相应的配置Spring bean可能看起来像 在非受管模式如下:

```
<bean id="managedConnectionFactory"
   class="com.sun.connector.cciblackbox.CciLocalTxManagedConnectionFactory">
  <property name="connectionURL" value="jdbc:hsqldb:hsql://localhost:9001"/>
  <property name="driverName" value="org.hsqldb.jdbcDriver"/>
</bean>

<bean id="targetConnectionFactory"
   class="org.springframework.jca.support.LocalConnectionFactoryBean">
  <property name="managedConnectionFactory" ref="managedConnectionFactory"/>
</bean>

<bean id="connectionFactory"
   class="org.springframework.jca.cci.connection.ConnectionSpecConnectionFactoryAdapter">
  <property name="targetConnectionFactory" ref="targetConnectionFactory"/>
  <property name="connectionSpec">
    <bean class="com.sun.connector.cciblackbox.CciConnectionSpec">
      <property name="user" value="sa"/>
      <property name="password" value=""/>
    </bean>
  </property>
</bean>

<bean id="component" class="MyDaoImpl">
  <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

在托管模式(即,在一个Java EE环境), 配置可能看起来如下:

```
<jee:jndi-lookup id="targetConnectionFactory" jndi-name="eis/blackbox"/>

<bean id="connectionFactory"
   class="org.springframework.jca.cci.connection.ConnectionSpecConnectionFactoryAdapter">
  <property name="targetConnectionFactory" ref="targetConnectionFactory"/>
  <property name="connectionSpec">
    <bean class="com.sun.connector.cciblackbox.CciConnectionSpec">
      <property name="user" value="sa"/>
      <property name="password" value=""/>
    </bean>
  </property>
</bean>

<bean id="component" class="MyDaoImpl">
  <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

25.4.6A例子 MappingCommAreaOperation 使用

在本节中,使用 MappingCommAreaOperation 将显示:访问吗 一个CICS ECI模式与IBM CICS ECI连接器。

首先,CCI InteractionSpec 需要被初始化为指定CICS程序访问和如何 与它进行交互。

```
public abstract class EciMappingOperation extends MappingCommAreaOperation {

  public EciMappingOperation(ConnectionFactory connectionFactory, String programName) {
    setConnectionFactory(connectionFactory);
    ECIInteractionSpec interactionSpec = new ECIInteractionSpec(),
    interactionSpec.setFunctionName(programName);
    interactionSpec.setInteractionVerb(ECIInteractionSpec.SYNC_SEND_RECEIVE);
    interactionSpec.setCommareaLength(30);
    setInteractionSpec(interactionSpec);
    setOutputRecordCreator(new EciOutputRecordCreator());
  }

  private static class EciOutputRecordCreator implements RecordCreator {
    public Record createRecord(RecordFactory recordFactory) throws ResourceException {
      return new CommAreaRecord();
    }
  }
}
```

抽象的 EciMappingOperation 类可以 然后再指定自定义对象之间的映射和 记录 。

```
public class MyDaoImpl extends CciDaoSupport implements MyDao {

  public OutputObject getData(Integer id) {
    EciMappingOperation query = new EciMappingOperation(getConnectionFactory(), "MYPROG") {
      protected abstract byte[] objectToBytes(Object inObject) throws IOException {
        Integer id = (Integer) inObject;
        return String.valueOf(id);
      }
      protected abstract Object bytesToObject(byte[] bytes) throws IOException;
        String str = new String(bytes);
        String field1 = str.substring(0,6);
        String field2 = str.substring(6,1);
```

```
        String field3 = str.substring(7,1);
        return new OutputObject(field1, field2, field3);
      }
    });

    return (OutputObject) query.execute(new Integer(id));
  }
}
```

相应的配置Spring bean可能看起来像 在非受管模式如下:

```
<bean id="managedConnectionFactory" class="com.ibm.connector2.cics.ECIManagedConnectionFactory">
  <property name="serverName" value="TXSERIES"/>
  <property name="connectionURL" value="local:"/>
  <property name="userName" value="CICSUSER"/>
  <property name="password" value="CICS"/>
</bean>

<bean id="connectionFactory" class="org.springframework.jca.support.LocalConnectionFactoryBean">
  <property name="managedConnectionFactory" ref="managedConnectionFactory"/>
</bean>

<bean id="component" class="MyDaoImpl">
  <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

在托管模式(即,在一个Java EE环境), 配置可能看起来如下:

```
<jee:jndi-lookup id="connectionFactory" jndi-name="eis/cicseci"/>

<bean id="component" class="MyDaoImpl">
  <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

25.5一个交易

JCA指定几个级别的事务支持资源 适配器。 这样的交易,你的资源适配器支持 指定在其 ra xml 文件。 本质上有 三个选项:没有(例如与CICS EPI连接器),当地的 事务(例如与CICS ECI连接器),全局事务 (例如与IMS connector)。

```
<connector>

  <resourceadapter>

    <!-- <transaction-support>NoTransaction</transaction-support> -->
    <!-- <transaction-support>LocalTransaction</transaction-support> -->
    <transaction-support>XATransaction</transaction-support>

  <resourceadapter>

<connector>
```

对全球事务,您可以使用Spring的一般事务 基础设施来限定交易, JtaTransactionManager 作为后端(委托给了 Java EE服务器的分布式事务处理协调器下面)。

对本地事务在一个单一的CCI ConnectionFactory ,弹簧提供了一个 具体事务管理策略对于CCI,类似于DataSourceTransactionManager 对于JDBC。 CCI API的 定义了一个本地事务对象和相应的本地事务 划分方法。 春天的CciLocalTransactionManager 执行这样的地方CCI 事务,完全符合春天的通用 PlatformTransactionManager 抽象。

```
<jee:jndi-lookup id="eciConnectionFactory" jndi-name="eis/cicseci"/>

<bean id="eciTransactionManager"
   class="org.springframework.jca.cci.connection.CciLocalTransactionManager">
  <property name="connectionFactory" ref="eciConnectionFactory"/>
</bean>
```

两个事务策略可以用于任何Spring的 事务界定设施,它声明或编程。 这是春天的一个后果是通用的PlatformTransactionManager 抽象, 这分离事务界定与实际执行 策略。 之间切换 JtaTransactionManager 和CciLocalTransactionManager 如有需要,保持你的 事务界定原样。

为更多的信息在春天的交易设施,看到 一章题为 ChapterA 12, 事务管理 。

26.一个电子邮件

26.1一个介绍

Spring框架提供了一个有用的实用工具库发送 电子邮件而使用户从具体的底层邮寄 系统和负责低水平的资源处理代表 客户端。

这个 org.springframework.mail 包是根水平方案 对于Spring框架的电子邮件支持。 中央接口来发送 邮件是 MailSender 接口,一个简单的值对象封装的属性等简单的邮件 从 和 到 (加上其他许多) SimpleMailMessage 类。 这个包还包含一个层次的受控异常提供 在更高层次的抽象程度较低,邮件系统异常 与根例外 MailException 。 请 参阅javadoc的更多信息丰富的邮件异常层次结构。

库依赖关系

以下额外的jar的类路径上你的 应用程序为了能够使用Spring框架的电子邮件库。

- 这个 javamail 邮件jar 图书馆
- 这个 JAF 激活jar 图书馆

所有这些库是在网络上免费下载。

这个 org.springframework.mail.javamail.JavaMailSender 界面添加了专门的 javamail 特性如MIME 消息支持 MailSender 接口 (从它继承)。 JavaMailSender 还提供了一个 回调接口,用于制备 JavaMail MIME消息,称 org.springframework.mail.javamail.MimeMessagePreparator

26.2使用

让我们假设有一个业务接口称为 OrderManager :

```
public interface OrderManager {

    void placeOrder(Order order);
}
```

让我们还假设有一个需求说明电子邮件消息 与订单号需要生成并发送到一个客户下单了 相关的订单。

26.2.1A基本 MailSender 和 SimpleMailMessage 使用

```
import org.springframework.mail.MailException;
import org.springframework.mail.MailSender;
import org.springframework.mail.SimpleMailMessage;

public class SimpleOrderManager implements OrderManager {

    private MailSender mailSender;
    private SimpleMailMessage templateMessage;

    public void setMailSender(MailSender mailSender) {
        this.mailSender = mailSender;
    }

    public void setTemplateMessage(SimpleMailMessage templateMessage) {
        this.templateMessage = templateMessage;
    }

    public void placeOrder(Order order) {

        // Do the business calculations...

        // Call the collaborators to persist the order...

        // Create a thread safe "copy" of the template message and customize it
        SimpleMailMessage msg = new SimpleMailMessage(this.templateMessage);
        msg.setTo(order.getCustomer().getEmailAddress());
        msg.setText(
            "Dear " + order.getCustomer().getFirstName()
                + order.getCustomer().getLastName()
                + ", thank you for placing order. Your order number is "
                + order.getOrderNumber());
        try{
            this.mailSender.send(msg);
        }
        catch(MailException ex) {
            // simply log it and go on...
            System.err.println(ex.getMessage());
        }
    }
}
```

下面找到bean定义上面的代码:

```
<bean id="mailSender" class="org.springframework.mail.javamail.JavaMailSenderImpl">
  <property name="host" value="mail.mycompany.com"/>
</bean>
```

```
<!-- this is a template message that we can pre-load with default state -->
<bean id="templateMessage" class="org.springframework.mail.SimpleMailMessage">
  <property name="from" value="customerservice@mycompany.com"/>
  <property name="subject" value="Your order"/>
</bean>

<bean id="orderManager" class="com.mycompany.businessapp.support.SimpleOrderManager">
  <property name="mailSender" ref="mailSender"/>
  <property name="templateMessage" ref="templateMessage"/>
</bean>
```

26.2.2A使用 JavaMailSender 和 MimeMessagePreparator

这里是另一个实现 OrderManager 使用 这个 MimeMessagePreparator 回调接口。 请注意 在这种情况下, MailSender 属性的类型是 JavaMailSender 所以,我们能够使用JavaMail MimeMessage 类:

```
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

import javax.mail.internet.MimeMessage;
import org.springframework.mail.MailException;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.mail.javamail.MimeMessagePreparator;

public class SimpleOrderManager implements OrderManager {

    private JavaMailSender mailSender;

    public void setMailSender(JavaMailSender mailSender) {
        this.mailSender = mailSender;
    }

    public void placeOrder(final Order order) {

        // Do the business calculations...

        // Call the collaborators to persist the order...

        MimeMessagePreparator preparator = new MimeMessagePreparator() {

            public void prepare(MimeMessage mimeMessage) throws Exception {

                mimeMessage.setRecipient(Message.RecipientType.TO,
                        new InternetAddress(order.getCustomer().getEmailAddress()));
                mimeMessage.setFrom(new InternetAddress("mail@mycompany.com"));
                mimeMessage.setText(
                    "Dear " + order.getCustomer().getFirstName() + " "
                        + order.getCustomer().getLastName()
                        + ", thank you for placing order. Your order number is "
                        + order.getOrderNumber());
            }
        };
        try {
            this.mailSender.send(preparator);
        }
        catch (MailException ex) {
            // simply log it and go on...
            System.err.println(ex.getMessage());
        }
    }
}
```

注意

邮件的代码是一个横切关注点,可能是一个候选人 重构到一个 自定义Spring AOP方面 , 然后可以执行适当的连接点在吗 OrderManager 目标。

Spring框架的邮件支持附带的标准JavaMail 实现。 请参考相关的JavaDocs获取更多信息。

26.3一个使用JavaMail MimeMessageHelper

一个类,它有很方便的在处理JavaMail消息 这个 org.springframework.mail.javamail.MimeMessageHelper 类, 这盾牌你不必使用JavaMail API的冗长。 使用 这个 MimeMessageHelper 它是相当容易的 创建一个 MimeMessage :

```
// of course you would use DI in any real-world cases
JavaMailSenderImpl sender = new JavaMailSenderImpl();
sender.setHost("mail.host.com");
```

```
MimeMessage message = sender.createMimeMessage();
MimeMessageHelper helper = new MimeMessageHelper(message);
helper.setTo("test@host.com");
helper.setText("Thank you for ordering!");

sender.send(message);
```

26.3.1A发送附件和内联资源

多部分邮件允许两个附件和内联资源。 例子的内联资源将图片或者您想要使用的样式表 在你的信息,但是,你不想显示为一个附件。

附件

以下示例向您展示如何使用 MimeMessageHelper 发送一个电子邮件连同一个 单一的JPEG图像附件。

```
JavaMailSenderImpl sender = new JavaMailSenderImpl();
sender.setHost("mail.host.com");

MimeMessage message = sender.createMimeMessage();

// use the true flag to indicate you need a multipart message
MimeMessageHelper helper = new MimeMessageHelper(message, true);
helper.setTo("test@host.com");

helper.setText("Check out this image!");

// let's attach the infamous windows Sample file (this time copied to c:/)
FileSystemResource file = new FileSystemResource(new File("c:/Sample.jpg"));
helper.addAttachment("CoolImage.jpg", file);

sender.send(message);
```

内联资源

以下示例向您展示如何使用 MimeMessageHelper 发送一个电子邮件以及一个 内联图像。

```
JavaMailSenderImpl sender = new JavaMailSenderImpl();
sender.setHost("mail.host.com");

MimeMessage message = sender.createMimeMessage();

// use the true flag to indicate you need a multipart message
MimeMessageHelper helper = new MimeMessageHelper(message, true);
helper.setTo("test@host.com");

// use the true flag to indicate the text included is HTML
helper.setText("<html><body><img src='cid:identifier1234'></body></html>", true);

// let's include the infamous windows Sample file (this time copied to c:/)
FileSystemResource res = new FileSystemResource(new File("c:/Sample.jpg"));
helper.addInline("identifier1234", res);

sender.send(message);
```

警告

内嵌资源添加到mime消息使用 指定 内容id（identifier1234 在上面的示例中)。 你的顺序添加文本和 资源是 **非常** 重要的。 一定要 第一次添加文本 那以后资源。 如果 你所做的是反过来的,它不会工作!

26.3.2A创建电子邮件内容使用模板库

在前面的例子的代码显式创建的 邮件的内容、使用方法等要求 消息settext(. .) 。 这是罚款 简单的情况下,它是可以在上下文中提到的 例子,目的是展示你最基本的API。

在典型的企业应用程序虽然,你不会 创建你的邮件的内容使用上述方法对一个号码 的原因。

- 创建基于html的电子邮件内容在Java代码中是单调乏味,而且容易出错
- 没有明确划分显示逻辑和业务逻辑
- 改变显示结构的电子邮件内容需要编写Java代码,重新编译和重新部署.....

通常采取的方法来解决这些问题是使用模板库 如FreeMarker或速度定义显示结构的电子邮件内容。 这让 你的代码只有在创建数据的任务,是要呈现在电子邮件 模板和发送电子邮件。 这绝对是一个最佳实践的时候 你的邮件的内容就比较复杂,用

Spring框架的支持类和速度变成FreeMarker 很容易做。 找到下面的示例使用Velocity模板库 创建电子邮件内容。

一个速度的基础例子

使用 速度 到 创建你的邮件模板(s),你将需要有速度库 可以在您的类路径中。 您还需要创建一个或多个Velocity模板 电子邮件的内容,您的应用程序的需要。 下面找到速度 模板,这个示例将使用。 如您所见,它是基于html的, 因为它是纯文本可以使用您喜欢的创建HTML 或文本编辑器。

```
# in the com/foo/package
<html>
<body>
<h3>Hi ${user.userName}, welcome to the Chipping Sodbury On-the-Hill message boards!</h3>

<div>
  Your email address is <a href="mailto:${user.emailAddress}">${user.emailAddress}</a>.
</div>
</body>

</html>
```

发现下面一些简单的代码和弹簧的XML配置, 利用上述的Velocity模板来创建电子邮件内容和 发送电子邮件(年代)。

```
package com.foo;

import org.apache.velocity.app.VelocityEngine;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.mail.javamail.MimeMessageHelper;
import org.springframework.mail.javamail.MimeMessagePreparator;
import org.springframework.ui.velocity.VelocityEngineUtils;

import javax.mail.internet.MimeMessage;
import java.util.HashMap;
import java.util.Map;

public class SimpleRegistrationService implements RegistrationService {

  private JavaMailSender mailSender;
  private VelocityEngine velocityEngine;

  public void setMailSender(JavaMailSender mailSender) {
    this.mailSender = mailSender;
  }

  public void setVelocityEngine(VelocityEngine velocityEngine) {
    this.velocityEngine = velocityEngine;
  }

  public void register(User user) {

    // Do the registration logic...

    sendConfirmationEmail(user);
  }

  private void sendConfirmationEmail(final User user) {
    MimeMessagePreparator preparator = new MimeMessagePreparator() {
      public void prepare(MimeMessage mimeMessage) throws Exception {
        MimeMessageHelper message = new MimeMessageHelper(mimeMessage);
        message.setTo(user.getEmailAddress());
        message.setFrom("webmaster@csonth.gov.uk"); // could be parameterized...
        Map model = new HashMap();
        model.put("user", user);
        String text = VelocityEngineUtils.mergeTemplateIntoString(
          velocityEngine, "com/dns/registration-confirmation.vm", model);
        message.setText(text, true);
      }
    };
    this.mailSender.send(preparator);
  }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="mailSender" class="org.springframework.mail.javamail.JavaMailSenderImpl">
    <property name="host" value="mail.csonth.gov.uk"/>
  </bean>

  <bean id="registrationService" class="com.foo.SimpleRegistrationService">
    <property name="mailSender" ref="mailSender"/>
    <property name="velocityEngine" ref="velocityEngine"/>
  </bean>
```

```
  <bean id="velocityEngine" class="org.springframework.ui.velocity.VelocityEngineFactoryBean">
    <property name="velocityProperties">
      <value>
       resource.loader=class
       class.resource.loader.class=org.apache.velocity.runtime.resource.loader.ClasspathResourceLoader
      </value>
    </property>
  </bean>

</beans>
```

27。 一个任务执行和调度

27.1一个介绍

Spring框架提供了抽象为异步 执行和调度的任务的 TaskExecutor 和 TaskScheduler 接口,分别。 春天也特性实现这些接口的支持 线程池或代表团CommonJ在应用服务器 环境。 最终使用这些实现后面的常见 接口抽象出Java SE 5之间的差别,Java SE 6和 Java EE环境中。

春天也功能集成类支持调度 与 定时器 ,自1.3 JDK的一部分, Quartz调度器(http://quartz-scheduler.org)。 这两个 调度器是设置使用 FactoryBean 可选引用 定时器 或 触发 实例,分别。 此外,一个 方便类的石英调度器和 定时器 是可用的,允许您调用吗 方法对现有目标对象(类似于正常 MethodInvokingFactoryBean 操作)。

27.2一个春天 TaskExecutor 抽象

Spring 2.0引入了一个新的抽象处理遗嘱执行人。 执行者是Java 5名线程池的概念。 这个 “执行人” 命名是由于这样的事实,即没有保证 底层的实现实际上是一个游泳池,一个执行者可能 单线程甚至同步。 Spring的抽象隐藏了 Java SE 1.4之间实现细节,Java SE和Java EE 5 环境。

春天的 TaskExecutor 接口是 与 java util并发执行器 接口。 事实上,其主要原因是抽象存在 需要Java 5当使用线程池。 这个接口有一个 方法 执行(可运行的任务) 接受一个任务 执行基于语义和配置的线程 池。

这个 TaskExecutor 最初是 给其他弹簧组件创建一个抽象为线程池 在需要的地方。 组件如 ApplicationEventMulticaster ,JMS的 AbstractMessageListenerContainer 和石英 集成所有使用 TaskExecutor 抽象池线程。 然而,如果你的豆子需要线程池 行为,就可能使用这种抽象为你自己的 的需要。

27.2.1A TaskExecutor 类型

有许多的预构建的实现 TaskExecutor 包含在春天 分布。 在所有的可能性,你不需要来实现 你自己的。

- SimpleAsyncTaskExecutor
  这个实现不重用任何线程,而是 启动一个新线程对于每个调用。 然而,它不支持 一个并发限制将阻止任何调用,结束了 直到一个槽的限制已经被释放。 如果你正在寻找真实 池,保持滚动页面下面进一步。
- SyncTaskExecutor
  这个实现不执行调用 异步。 相反,每次调用发生在调用 线程。 它主要用于多线程的情况 没有必要如简单的测试用例。
- ConcurrentTaskExecutor
  这个实现是一个包装一个Java 5 java util并发执行器 。 有一个 的选择, ThreadPoolTaskExecutor ,, 暴露了 执行人 配置参数 作为bean属性。 这是很少需要使用 ConcurrentTaskExecutor 但如果 ThreadPoolTaskExecutor 不是强劲足以满足你的需求, ConcurrentTaskExecutor 是一个 替代。
- SimpleThreadPoolTaskExecutor
  这个实现实际上是一个子类的石英的 SimpleThreadPool 倾听春天的吗 生命周期回调。 这通常用在当你有一个线程 池,可能需要共享两个石英和非石英 组件。
- ThreadPoolTaskExecutor
  这个实现只能用于在Java 5的环境 但也是最常用的一个环境。 它 公开了用于配置一个bean属性 java util并发threadpoolexecutor 和 封装在一个 TaskExecutor 。 如果你 需要一些先进的如 ScheduledThreadPoolExecutor ,它是 建议您使用一个 ConcurrentTaskExecutor 相反。

  > 它不可能使用任何补丁或替代版本 的 java . util . concurrent 包与 这个实现。 都是和 Dawid Kurzyniec Doug Lea的 实现使用不同的包结构, 防止他们正常工作。
- TimerTaskExecutor
  这个实现使用一个单一的 TimerTask 作为其支持的实现。 这是 不同 SyncTaskExecutor 在该方法调用是在独立的线程中

执行, 虽然他们是同步,线程。

- WorkManagerTaskExecutor
  这个实现使用CommonJ WorkManager作为它的 支持实现和中央便利类 设置在一个春天的CommonJ WorkManager参考上下文。 类似 SimpleThreadPoolTaskExecutor , 这个类实现WorkManager接口,因此可以 直接当作WorkManager一样。

> CommonJ是一组规范之间的联合开发 BEA和IBM。 这些规格不是Java EE标准,但是 在BEA的标准和IBM的应用服务器吗 实现。

27.2.2A使用 TaskExecutor

春天的 TaskExecutor 实现是作为简单的javabean。 在下面的例子中,我们 定义一个bean使用 ThreadPoolTaskExecutor 异步打印 出一组信息。

```
import org.springframework.core.task.TaskExecutor;

public class TaskExecutorExample {

  private class MessagePrinterTask implements Runnable {

    private String message;

    public MessagePrinterTask(String message) {
      this.message = message;
    }

    public void run() {
      System.out.println(message);
    }

  }

  private TaskExecutor taskExecutor;

  public TaskExecutorExample(TaskExecutor taskExecutor) {
    this.taskExecutor = taskExecutor;
  }

  public void printMessages() {
    for(int i = 0; i < 25; i++) {
      taskExecutor.execute(new MessagePrinterTask("Message" + i));
    }
  }
}
```

正如您可以看到的,而不是从池中检索一个线程和 执行自己,你添加你的 runnable 到 队列和 TaskExecutor 使用它 内部规则来决定当任务得到执行。

配置的规则 TaskExecutor 将使用,简单的bean 属性已经暴露。

```
<bean id="taskExecutor" class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor">
  <property name="corePoolSize" value="5" />
  <property name="maxPoolSize" value="10" />
  <property name="queueCapacity" value="25" />
</bean>

<bean id="taskExecutorExample" class="TaskExecutorExample">
  <constructor-arg ref="taskExecutor" />
</bean>
```

27.3一个春天 TaskScheduler 抽象

除了 TaskExecutor 抽象,Spring 3.0引入了一个 TaskScheduler 用各种方法来实现 调度任务运行在将来的某个时候。

```
public interface TaskScheduler {

    ScheduledFuture schedule(Runnable task, Trigger trigger);

    ScheduledFuture schedule(Runnable task, Date startTime);

    ScheduledFuture scheduleAtFixedRate(Runnable task, Date startTime, long period);

    ScheduledFuture scheduleAtFixedRate(Runnable task, long period);

    ScheduledFuture scheduleWithFixedDelay(Runnable task, Date startTime, long delay);

    ScheduledFuture scheduleWithFixedDelay(Runnable task, long delay);

}
```

最简单的方法是在一个名叫“时间表”,需要一个 runnable 和 日期 只有。 这将导致任务在指定的时间后运行一次。 所有 其他的方法能够调度任务重复运行。 固定利率和固定延迟的方法是简单的,定期执行, 但是这个方法,该方法接受一个触发器就灵活多了。

27.3.1A了 触发 接口

这个 触发 接口是 本质上是受jsr - 236,Spring 3.0的,还没有 被正式实施。 的基本想法 触发 是,执行时间可能吗 确定基于过去的执行结果甚至是任意的 条件。 如果这些决定做考虑的结果 前面的执行,信息可以在一个 TriggerContext 。 这个 触发 接口本身是相当 简单的:

```
public interface Trigger {
    Date nextExecutionTime(TriggerContext triggerContext);
}
```

正如您可以看到的, TriggerContext 是最重要的部分。 它封装了所有相关的数据, 对扩展开放,在未来如果有必要。 这个 TriggerContext 是一个接口(一个吗 SimpleTriggerContext 实现被 默认)。 在这里你可以看到什么方法可用 触发 实现。

```
public interface TriggerContext {
    Date lastScheduledExecutionTime();
    Date lastActualExecutionTime();
    Date lastCompletionTime();
}
```

27.3.2A 触发 实现

弹簧提供的两个实现 触发 接口。 最有趣的 一个是 CronTrigger 。 它使调度 基于cron表达式的任务。 例如下面的任务 被调度运行15分钟过去的每小时只有在 朝九晚五的“营业时间”在工作日。

```
scheduler.schedule(task, new CronTrigger("* 15 9-17 * * MON-FRI"));
```

其他的开箱即用的实现是一个 PeriodicTrigger 接受一个固定周期,一个 可选的初始延迟值,和一个布尔值,用于指示是否 期应被视为一个固定利率或一个固定的延迟。 自 TaskScheduler 接口已经定义了 方法对调度任务在一个固定利率或与一个固定的延迟, 这些方法应尽可能直接使用。 的价值 这个 PeriodicTrigger 实现,它可以 无法应用于组件的依赖 触发 抽象。 例如,它可能方便让周期性触发器,触发器,甚至基于cron 自定义触发器实现互换使用。 这样一个 组件可以利用依赖注入,这样这样 触发 可以配置 外部。

27.3.3A TaskScheduler 实现

像春天的 TaskExecutor 抽象,主要的好处 TaskScheduler 是,代码依赖 调度行为不需要耦合到一个特定的调度器 实现。 它提供的灵活性尤其相关 当在应用服务器环境中运行的线程应该 不会创建直接由应用程序本身。 对于这种情况, 弹簧提供了一个 TimerManagerTaskScheduler 这 代表一个CommonJ TimerManager实例,通常配置 一个jndi查找。

一个更简单的选择, ThreadPoolTaskScheduler ,立刻可以使用 外部线程管理不是一个要求。 在内部,它 代表一个 ScheduledExecutorService 实例。 ThreadPoolTaskScheduler 其实 实现了春天的 TaskExecutor 接口,因此,单个实例可用于异步执行 尽快 也 按照预定计划,和潜在的循环,执行死刑。

27.4一个注释支持调度和异步 执行

Spring提供了注解支持这两个任务调度和 异步方法执行。

27.4.1A使调度注释

使支持 @Scheduled 和 @Async 注释添加 @EnableScheduling 和 @EnableAsync 你的 @ configuration 类:

```
@Configuration
@EnableAsync
@EnableSCheduling
public class AppConfig {
}
```

你可以自由选择相关的注释 对于您的应用程序。 例如,如果您只需要支持 对于 @Scheduled ,简单的省略 @EnableAsync 。 更细粒度的 控制你可以另外实现 SchedulingConfigurer 和/或 AsyncConfigurer 接口。 看到 Javadoc详情。

如果你喜欢XML配置使用 <任务:注解驱动的> 元素。

```
<task:annotation-driven executor="myExecutor" scheduler="myScheduler"/>
<task:executor id="myExecutor" pool-size="5"/>
<task:scheduler id="myScheduler" pool-size="10"/>}
```

注意与上面的XML,一个执行器提供参考 处理这些任务,对应的方法 @Async 注释,调度程序 参考提供了管理这些方法注释 与 @Scheduled 。

27.4.2A @Scheduled的注释

@Scheduled的注释可以被添加到一个方法以及 触发元数据。 例如,下面的方法会被调用 每5秒用一个固定的延迟,这意味着,周期 从完成时间的测量每个前调用。

```
@Scheduled(fixedDelay=5000)
public void doSomething() {
    // something that should execute periodically
}
```

如果一个固定利率执行,是理想的,只需修改属性 名字中指定的注释。 以下将被执行 每5秒之间测量的开始时间的连续的每个 调用。

```
@Scheduled(fixedRate=5000)
public void doSomething() {
    // something that should execute periodically
}
```

对于固定的延迟和固定的任务,一个初始延迟可能 指示指定的毫秒数之前等待的第一 执行的方法。

```
@Scheduled(initialDelay=1000, fixedRate=5000)
public void doSomething() {
    // something that should execute periodically
}
```

如果简单的周期性调度并没有表现足够,那么一个 cron表达式可能被提供。 例如,以下只会 执行在工作日。

```
@Scheduled(cron="*/5 * * * * MON-FRI")
public void doSomething() {
    // something that should execute on weekdays only
}
```

注意,方法是计划必须有空隙的回报和 不要期望任何参数。 如果该方法需要互动 其他对象的应用程序上下文,然后那些通常 已经提供了通过依赖注入。

注意

确保你没有初始化的多个实例 同样的@Scheduled注释类在运行时,除非你想 安排回调每个这样的实例。 与此相关,确保 你不使用@Configurable bean类上的注释 与@Scheduled和注册为普通的Spring bean 容器:你会得到双倍的初始化,否则,一旦通过 容器和一旦通过@Configurable方面, 结果被调用方法的每个 @Scheduled两次。

27.4.3A @Async的注释

这个 @Async 注释可以 提供一个方法,以便调用该方法将发生 异步。 换句话说,调用者将返回后立即 调用和实际执行的方法将发生在一个任务 已提交的一个春天 TaskExecutor 。 在最简单的情况下, 注释可以被应用到一个 无效 受益者 法。

```
@Async
void doSomething() {
    // this will be executed asynchronously
}
```

不同的方法 @Scheduled 注释,这些方法可以 预计参数,因为它们将调用在 “正常” 的方式 在运行时调用者而不是从一个预定的任务要由 容器。 例如,下面是一个合法的应用 这个 @Async 注释。

```
@Async
```

```
void doSomething(String s) {
    // this will be executed asynchronously
}
```

即使方法返回值可以以异步方式调用。 然而,这些方法都必须有一个 未来 类型的返回值。 这仍然 提供异步执行的好处,以便调用者可以 执行其他任务之前调用 get() 在 未来的。

```
@Async
Future<String> returnSomething(int i) {
    // this will be executed asynchronously
}
```

@Async 不能用于 结合生命周期回调如 @PostConstruct 。 异步 初始化Spring bean,您现在使用一个单独的 初始化Spring bean调用 @Async 带注释的方法的目标 然后。

```
public class SampleBeanImpl implements SampleBean {

  @Async
  void doSomething() { ... }
}


public class SampleBeanInititalizer {

  private final SampleBean bean;

  public SampleBeanInitializer(SampleBean bean) {
    this.bean = bean;
  }

  @PostConstruct
  public void initialize() {
    bean.doSomething();
  }
}
```

27.4.4A遗嘱执行人资格与@Async

默认情况下,当指定 @Async 在 一个方法,将使用的执行器,是一个提供给 “注解驱动的元素如前所述。 然而, 价值 属性的 @Async 需要的时候可以使用注释 表明一个执行人除了默认时应使用 执行一个给定的方法。

```
@Async("otherExecutor")
void doSomething(String s) {
    // this will be executed asynchronously by "otherExecutor"
}
```

在这种情况下,“otherExecutor” 可能是任何的名字 执行人 豆在Spring容器,或 可能的名字吗 预选赛 产生任何 执行人 ,如作为指定的 <限定符> 元素或弹簧的 qualifier 注释。

27.5一个任务名称空间

Spring 3.0开始,有一个XML名称空间配置 TaskExecutor 和 TaskScheduler 实例。 它还提供了一个 方便的方式来配置任务调度与触发器。

27.5.1A “调度” 元素

以下元素将创建一个 ThreadPoolTaskScheduler 实例与 指定的线程池的大小。

```
<task:scheduler id="scheduler" pool-size="10"/>
```

提供的值为 “id” 属性将被用作 前缀为线程的名字在池中。 “调度” 元素 相对简单的。 如果你不能提供一个 “池大小的 属性,默认线程池将只有一个线程。 有 没有其他配置选项调度器。

27.5.2A “执行人” 元素

下面将创建一个 ThreadPoolTaskExecutor 实例:

```
<task:executor id="executor" pool-size="10"/>
```

与上面的调度器,价值提供 “id” 属性将被用作前缀为线程的名字在池中。 至于池大小而言,“执行人” 元素的支持 更多的配置

选项比“调度”元素。 首先, 线程池的一个 ThreadPoolTaskExecutor 是 本身更可配置。 而不仅仅是一个单一的大小,一个执行器的 线程池可能有不同的值 核心 和 马克斯 大小。 如果提供的是一个单值 然后执行程序将有一个固定大小的线程池(核心和马克斯 大小是相同的)。 然而,“执行人”元素的“池大小的 属性还接受一个范围在形式的“min马克斯”。

```
<task:executor id="executorWithPoolSizeRange"
            pool-size="5-25"
            queue-capacity="100"/>
```

正如您可以看到的从这个配置,一个“队列容量的值 也被提供。 线程池的配置也应该 被认为是在光的执行人的队列容量。 为了全面 描述之间的关系,池大小和队列容量, 参考文档 ThreadPoolExecutor 。 主要的思想是,当一个任务被提交,执行人将第一尝试使用一个免费的线程如果活动线程的数量是目前 不到核心的大小。 如果核心尺寸已经达到,那么 任务将会被添加到队列中,只要其能力尚未 达到了。 只有这样,如果队列的容量 已经 被达成,将执行程序创建一个新的线程之外的核心 大小。 如果最大的尺寸也已经达到了,那么遗嘱执行人将 拒绝任务。

默认情况下,该队列 无界 ,但这 很少是所需的配置,因为它会导致吗 outofmemoryerror错误 如果足够的任务添加到 虽然所有的队列池线程都忙。 此外,如果队列 无界,然后最大尺寸没有任何影响。 因为执行程序 总是试着队列创建一个新的线程之前超越核心 大小,一个队列必须有一个容量有限的线程池来成长 超出了核心大小(这是为什么 固定大小的 池 是唯一明智的案例当使用一个无界队列)。

在一个时刻,我们将回顾维生的影响设置 这又增加了另一个因素时要考虑提供一个池的大小 配置。 首先,让我们考虑这样一种情况,正如上面提到的,当 一个任务被拒绝。 默认情况下,当一个任务被拒绝,一个线程池 执行人将抛出一个 TaskRejectedException 。 然而,拒绝政策实际上是可配置的。 唯一的例外是 使用默认拒绝时抛出的政策 AbortPolicy 实现。 对于应用程序 哪里可以跳过某些任务负载较重的情况下,要么 DiscardPolicy 或 DiscardOldestPolicy 可以配置相反。 另一个选项,适用于应用程序需要节流 提交的任务负载较重的情况下,是 CallerRunsPolicy 。 而不是抛出 异常或丢弃的任务,政策只会迫使线程 这是调用提交方法运行任务本身。 这个想法是 这样一个调用者将是繁忙的在运行该任务和无法 立即提交其他任务。 因此,它提供了一个简单的方法 节流传入的负载,同时保持线程的极限 池和队列。 通常这允许执行程序“迎头赶上” 任务是处理,从而腾出一些能力在队列中,在 池,或两者兼而有之。 这些选项可以选择从 枚举值用于“拒绝策略的属性 “执行人”元素。

```
<task:executor id="executorWithCallerRunsPolicy"
            pool-size="5-25"
            queue-capacity="100"
            rejection-policy="CALLER_RUNS"/>
```

27.5.3A“计划任务”元素

最强大的特性是春天的任务名称空间 支持配置任务是安排在一个春天 应用程序上下文。 这是一个方法类似于其他 ” 方法调用器”在春天,例如JMS提供的名称空间 配置消息驱动pojo。 基本上一个“ref”属性可以 指向任何spring管理对象,和“方法”属性提供 这个名字将被调用的方法的物体上。 这是一个简单的 的例子。

```
<task:scheduled-tasks scheduler="myScheduler">
   <task:scheduled ref="beanA" method="methodA" fixed-delay="5000"/>
</task:scheduled-tasks>

<task:scheduler id="myScheduler" pool-size="10"/>
```

正如您可以看到的,该调度器是引用的外部元件, 和每个任务包括配置的触发器 元数据。 在前面的例子中,元数据定义了一个周期 触发器与一个固定的延迟显示等待的毫秒数 每个任务完成后执行。 另一个选择是“固定”, 显示的频率应该执行方法无论多久 以前的任何执行需要。 此外,对于固定的延迟和 固定的任务“初始延迟的参数可以指定的指示 等待的毫秒数之前的第一次执行 法。 更多的控制,一个“cron”属性可能被提供相反。 这里有一个例子证明这些其他的选择。

```
<task:scheduled-tasks scheduler="myScheduler">
   <task:scheduled ref="beanA" method="methodA" fixed-delay="5000" initial-delay="1000"/>
   <task:scheduled ref="beanB" method="methodB" fixed-rate="5000"/>
   <task:scheduled ref="beanC" method="methodC" cron="*/5 * * * * MON-FRI"/>
</task:scheduled-tasks>

<task:scheduler id="myScheduler" pool-size="10"/>
```

27.6一个使用石英调度器

石英使用 触发 , 工作 和 JobDetail 对象 实现调度的各种工作。 为背后的基本概念 石英,看一看 http://quartz-scheduler.org 。 为了方便 目的,Spring提供了两类,简化使用 石英在基于spring的应用。

使用JobDetailBean 27.6.1A

JobDetail 对象包含所有信息 需要运行工作。 Spring框架提供了一个 JobDetailBean 使 JobDetail 更实际的JavaBean和 合理的默认值。 让我们看一个例子:

```
<bean name="exampleJob" class="org.springframework.scheduling.quartz.JobDetailBean">
  <property name="jobClass" value="example.ExampleJob" />
  <property name="jobDataAsMap">
    <map>
      <entry key="timeout" value="5" />
    </map>
  </property>
</bean>
```

详细的工作的所有信息需要bean来运行工作（ExampleJob ）。 中指定的超时工作 数据地图。 工作地图数据是可以通过的 JobExecutionContext (传递给你在执行 时间),但是 JobDetailBean 也将 从工作性质数据映射到属性的实际工作。 所以在 这种情况下,如果 ExampleJob 包含一个属性 命名 超时 , JobDetailBean 将自动应用它:

```
package example;

public class ExampleJob extends QuartzJobBean {

  private int timeout;

  /**
   * Setter called after the ExampleJob is instantiated
   * with the value from the JobDetailBean (5)
   */
  public void setTimeout(int timeout) {
    this.timeout = timeout;
  }

  protected void executeInternal(JobExecutionContext ctx) throws JobExecutionException {
      // do the actual work
  }
}
```

所有额外的设置与工作细节bean是当然的 你也可用。

注意:使用 名称 和 集团 属性,您可以修改名称和 组的工作,分别。 默认情况下,工作的名称匹配 bean名称的工作细节bean(在上面的例子中,这是 ExampleJob)。

27.6.2A使用 MethodInvokingJobDetailFactoryBean

通常你只需要调用一个方法在一个特定的对象。 使用 这个 MethodInvokingJobDetailFactoryBean 你可以做 到底这个:

```
<bean id="jobDetail" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
  <property name="targetObject" ref="exampleBusinessObject" />
  <property name="targetMethod" value="doIt" />
</bean>
```

上面的例子将导致 doIt 方法被呼吁 exampleBusinessObject 方法(见下图):

```
public class ExampleBusinessObject {

  // properties and collaborators

  public void doIt() {
    // do the actual work
  }
}
```

```
<bean id="exampleBusinessObject" class="examples.ExampleBusinessObject"/>
```

使用 MethodInvokingJobDetailFactoryBean ,你不 需要创建一行工作,就是调用一个方法,你只 需要创建实际的业务对象和连接细节 对象。

默认情况下,石英工作是无状态的,结果 工作的可能性相互干扰。 如果你指定两个 触发器为同一 JobDetail ,它可能是 可能在第一份工作已经完成了,第二个将 开始。 如果 JobDetail 类都实现了 状态 接口,这不会发生。 第二个工作之前不会开始第一个完成。 让 工作产生的 MethodInvokingJobDetailFactoryBean 非并发,设置 并发 旗帜 假 。

```
<bean id="jobDetail" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
  <property name="targetObject" ref="exampleBusinessObject" />
  <property name="targetMethod" value="doIt" />
  <property name="concurrent" value="false" />
</bean>
```

注意

默认情况下,乔布斯将运行在一个并发的时尚。

27.6.3A布线工作使用触发器和 SchedulerFactoryBean

我们已经创建了工作细节和工作。 我们也回顾了 方便豆,允许您调用一个方法在一个特定的 对象。 当然,我们仍然需要安排自己的工作。 这 都是使用触发器和一个吗 SchedulerFactoryBean 。 几个触发器 可用在石英和弹簧提供了两种石英 FactoryBean 实现方便违约: CronTriggerFactoryBean 和 SimpleTriggerFactoryBean 。

触发器需要调度。 弹簧提供了一个 SchedulerFactoryBean 这暴露了触发器是 设置为属性。 SchedulerFactoryBean 时间表实际的工作,这些触发器。

找到下面几个例子:

```
<bean id="simpleTrigger" class="org.springframework.scheduling.quartz.SimpleTriggerFactoryBean">
   <!-- see the example of method invoking job above -->
   <property name="jobDetail" ref="jobDetail" />
   <!-- 10 seconds -->
   <property name="startDelay" value="10000" />
   <!-- repeat every 50 seconds -->
   <property name="repeatInterval" value="50000" />
</bean>

<bean id="cronTrigger" class="org.springframework.scheduling.quartz.CronTriggerFactoryBean">
   <property name="jobDetail" ref="exampleJob" />
   <!-- run every morning at 6 AM -->
   <property name="cronExpression" value="0 0 6 * * ?" />
</bean>
```

现在我们已经建立了两个触发器,一个运行每50秒一个 启动延迟10秒和一个每天早上6点。 完成 一切,我们需要设置 SchedulerFactoryBean :

```
<bean class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
   <property name="triggers">
      <list>
         <ref bean="cronTrigger" />
         <ref bean="simpleTrigger" />
      </list>
   </property>
</bean>
```

更多的属性是可用的 SchedulerFactoryBean 为你设置,例如 日历使用的工作细节,属性来定制石英, 等等。看一下 SchedulerFactoryBean Javadoc 为更多的信息。

28。 动态语言支持

28.1一个介绍

Spring 2.0引入了全面支持使用类和对象 定义使用动态语言(例如JRuby)和弹簧。 这个支持允许您编写任意数量的类支持动态语言, 和有Spring容器透明地实例化、配置和依赖注入 结果对象。

目前支持的动态语言是:

- JRuby 0.9/1.0
- Groovy 1.0/1.5
- BeanShell 2.0

完全工作示例的动态语言支持可以立即有用的 描述了 SectionA 28.4,一个Scenariosa 。

为什么只有这些语言吗?

支持的语言被选中是因为一个)的语言 有很多的牵引在Java企业社区,b)没有请求了 对于其他语言在Spring 2.0开发时间表, c)春季开发商最熟悉他们。

没有什么阻止包含进一步的语言虽然。 如果你想要 看到支持< 你最喜欢的动态语言在这里插入 >, 你总是可以提高在春天的一个问题 JIRA 页面(或实现这种支持自己)。

注意: 只有特定的版本是上面列出的支持 在Spring 2.5中。 特别是,JRuby 1.1(介绍了很多不兼容的API 变化)是 不 支持在这一点上的时间。

28.2一百一十一例

这大部分本章涉及描述动态语言支持 在细节。 在深入所有的前因后果动态语言支持, 让我们来看一个快速的示例bean定义在一个动态语言。 动态语言,这第一个bean是Groovy(本示例的基础 来自春天的测试套件,所以如果你想看到等效的例子吗 在任何其他受支持的语言,看看源代码)。

下面找到 信使 接口, Groovy bean将会实施,注意,这个接口定义 在普通的Java。 依赖对象,注射的引用 信使 不知道底层 实现是一个Groovy脚本。

```
package org.springframework.scripting;

public interface Messenger {

    String getMessage();
}
```

这是一个类的定义,有依赖 信使 接口。

```
package org.springframework.scripting;

public class DefaultBookingService implements BookingService {

    private Messenger messenger;

    public void setMessenger(Messenger messenger) {
        this.messenger = messenger;
    }

    public void processBooking() {
        // use the injected Messenger object...
    }
}
```

这是一个实现的 信使 接口 在Groovy中。

```
// from the file 'Messenger.groovy'
package org.springframework.scripting.groovy;

// import the Messenger interface (written in Java) that is to be implemented
import org.springframework.scripting.Messenger

// define the implementation in Groovy
class GroovyMessenger implements Messenger {

    String message
}
```

最后,这里是bean定义,将注射的效果 groovy定义 信使 实现进 的一个实例 DefaultBookingService 类。

注意

使用自定义动态语言标签定义动态语言支持bean, 你需要XML Schema序文顶部的Spring XML 配置文件。 您还需要使用一个春天 ApplicationContext 实现作为您的 IoC容器。 使用动态语言支持bean与平原 BeanFactory 实现是支持, 但是你必须管理管道内部的弹簧这样做。

基于配置的更多信息,请参阅 AppendixA E, XML的基于配置 。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:lang="http://www.springframework.org/schema/lang"
      xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/lang http://www.springframework.org/schema/lang/spring-lang.xsd">

    <!-- this is the bean definition for the Groovy-backed Messenger implementation -->
    <lang:groovy id="messenger" script-source="classpath:Messenger.groovy">
        <lang:property name="message" value="I Can Do The Frug" />
    </lang:groovy>

    <!-- an otherwise normal bean that will be injected by the Groovy-backed Messenger -->
    <bean id="bookingService" class="x.y.DefaultBookingService">
        <property name="messenger" ref="messenger" />
    </bean>

</beans>
```

这个 bookingService bean(一个 DefaultBookingService)现在可以使用其私有 信使 成员变量作为正常的因为 信使 实例,注射 进去 是 一个 信使 实例。 没有什么不同寻常的事情发生在这里,只是普通的Java和 平原Groovy。

希望上面的XML代码片段是不言而喻的,但别担心 过度如果不是。 保持阅读的深入细节令人费解的问题 诸多上面的配置。

28.3一个定义bean,动态语言支持

本节描述如何定义Spring托管bean中 任何支持的动态语言。

请注意,本章并不试图解释语法和 习语的动态语言支持。 例如,如果你想 使用Groovy编写特定的类在您的应用程序,那么 假设你已经知道Groovy。 如果你需要进一步的细节 关于动态语言本身,请查阅 SectionA 28.6,一个进一步Resourcesa 本章末尾的。

28.3.1A常见的概念

所涉及的步骤使用动态语言支持bean是如下:

1. 写测试的动态语言源代码(自然)
2. 然后 编写动态语言源代码本身:)
3. 定义你的动态语言支持bean使用适当的 < lang:语言/ > 元素在XML 配置(当然,你还可以定义这些bean以编程方式 使用Spring API——虽然你要咨询源 代码方向如何做这是这种类型的先进 配置是不包括在这一章)。 注意这是一个迭代 步骤。 您将需要至少一个bean定义/动态 语言源文件(虽然同样的动态语言源代码 文件当然是由多个bean定义引用)。

前两个步骤(测试和编写你的动态语言源文件) 超出了本章的范围。 指语言规范 和/或参考手册为选择的动态语言和裂纹与 发展你的动态语言源文件。 你 将 首先要阅读本章剩下的不过,Spring的动态语言 支持确实使一些(小)假设动态的内容 语言源文件。

这个 < lang:语言/ > 元素

最后一步包括定义动态语言支持bean的定义, 一个用于每个bean,您要配置(这是没有区别的 正常的JavaBean配置)。 然而,而不是指定 类的完全限定类名,是被实例化和 容器配置,您使用 < lang:语言/ > 元素定义了动态语言支持bean。

每个受支持的语言有一个对应的 < lang:语言/ > 元素:

- < lang:jruby / > (JRuby)
- < lang:groovy / > (Groovy)
- < lang:bsh / > (BeanShell)

确切的属性和子元素,可用 配置完全取决于哪些语言bean已经 定义在(特定于语言的章节提供完整的 真相在此)。

> **XML模式**
>
> 所有的配置例子在这一章利用 新的XML Schema支持Spring 2.0中添加。
>
> 有可能放弃使用XML Schema和坚持老式 基于你的春天的DTD验证XML文件,但是你失去了 在提供的方便 < lang:语言/ > 元素。 看到弹簧测试套件的旧式风格的例子 配置不需要XML的基于验证 (它是相当冗长,不隐藏任何潜在的春天实现从你)。

可刷新的豆子

(如果不是之一 这个 最引人注目的价值补充道。 动态语言支持在春 " 可刷新的bean的 特性。

一个可刷新的bean是一种动态语言支持bean与一个小 数量的配置,动态语言支持bean可以监视 改变它的底层源代码文件资源,然后重新加载本身 当动态语言源文件被改变(例如当一个 开发者编辑并保存更改文件系统上的文件)。

这允许开发人员部署任何数量的动态语言源代码 文件作为应用程序的一部分,配置Spring容器创建 bean支持动态语言源文件(使用机制 在本章中介绍),然后后来,当需求变更或 其他一些外部因素也发挥了作用,只需编辑一个动态语言 源文件和有任何改变他们反映在bean 动态语言支持的改变源文件。 没有必要 关闭正在运行的应用程序(或重新部署在一个web应用程序的情况下)。 动态语言支持bean的所以修改将采用新的状态 和逻辑的改变动态语言源文件。

> **注意**
>
> 请注意,此功能是 掉 默认情况下。

让我们来看一个例子,看看是多么容易开始使用 可刷新的豆子。 到 打开 可刷新的bean的 功能,您只需指定到底 一个 额外的属性 < lang:语言/ > 元素 您的bean定义。 所以如果我们坚持 这个例子 从早期 在这一章,我们将改变在Spring XML配置 来效果可刷新的豆:

```
<beans>
```

```
    <!-- this bean is now 'refreshable' due to the presence of the 'refresh-check-delay' attribute -->
    <lang:groovy id="messenger"
        refresh-check-delay="5000" <!-- switches refreshing on with 5 seconds between checks -->
        script-source="classpath:Messenger.groovy">
      <lang:property name="message" value="I Can Do The Frug" />
    </lang:groovy>

    <bean id="bookingService" class="x.y.DefaultBookingService">
      <property name="messenger" ref="messenger" />
    </bean>

  </beans>
```

这真的是你所要做的。 这个 “刷新检查延迟的 属性上定义 的“信使” bean定义 是的毫秒数之后,bean将刷新 任何更改底层动态语言源文件。 你可以关掉刷新行为通过分配一个负值 到 “刷新检查延迟的 属性。 记住,在默认情况下,刷新行为是禁用的。 如果你不 要刷新行为,然后根本不定义属性。

如果我们然后运行下面的程序我们可以锻炼可刷新的功能; 请原谅 “跳圈暂停执行” 恶作剧在下片代码。 这个 系统在阅读()电话是只有这样程序的执行过程中停顿而我(作者) 响和编辑底层动态语言源文件以便刷新会 触发器的动态语言支持bean当程序继续执行。

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.scripting.Messenger;

public final class Boot {

    public static void main(final String[] args) throws Exception {

        ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");
        Messenger messenger = (Messenger) ctx.getBean("messenger");
        System.out.println(messenger.getMessage());
        // pause execution while I go off and make changes to the source file...
        System.in.read();
        System.out.println(messenger.getMessage());
    }
}
```

让我们假设之后,对于本例,所有 调用 getMessage() 方法 信使 实现必须 改变了这样的消息被引用。 下面是更改,我(作者)对 信使groovy 源文件在执行时 程序暂停。

```
package org.springframework.scripting

class GroovyMessenger implements Messenger {

    private String message = "Bingo"

    public String getMessage() {
        // change the implementation to surround the message in quotes
        return "'" + this.message + "'"
    }

    public void setMessage(String message) {
        this.message = message
    }
}
```

当程序执行时,输出在输入暂停将 我可以做扭摆舞 。 变更后 源文件制作和保存,程序继续执行, 调用的结果 getMessage() 方法 动态语言支持 信使 实现 将 “我可以做扭摆舞” (注意 包含额外的引用)。

重要的是要理解,一个脚本将改变 不 触发一个刷新如果发生改变 在窗口的 “刷新检查延迟的 值。 它是同样重要的是去理解,修改脚本 不 其实 “捡” 到一个方法被调用 在动态语言支持bean。 只有当一个方法被调用在一个 动态语言支持bean,它检查,看看它的潜在的脚本 源已经改变了。 任何异常有关刷新脚本 (如遇到编译错误,或发现脚本 文件已被删除)将导致 致命 例外被传递到调用代码。

上面描述的可刷新的bean的行为确实 不 适用于动态语言源文件 定义使用 < lang:内联脚本/ > 元素 符号(见 一个章节内联动态语言filesa来源)。 此外,它 只有 适用于豆类, 改变底层的源文件可以被检测到; 例如,通过代码检查最后修改日期 动态语言源文件在文件系统上存在。

内联动态语言源文件

动态语言支持也可以满足动态语言 源文件中直接嵌入Spring bean定义。 更具体地说, < lang:内联脚本/ > 元素允许您定义动态语言源立即 在Spring配置文件。 一个例子将可能使 内联脚本功能清晰:

```
<lang:groovy id="messenger">
```

```
    <lang:inline-script>
package org.springframework.scripting.groovy;

import org.springframework.scripting.Messenger

class GroovyMessenger implements Messenger {

    String message
}
    </lang:inline-script>
    <lang:property name="message" value="I Can Do The Frug" />
</lang:groovy>
```

如果我们把一边的议题是否良好实践 定义动态语言源在Spring配置文件, < lang:内联脚本/ > 元素可以是有效的 一些场景。 例如,我们可能想要快速添加一个春天 验证器 实现一个Spring MVC 控制器 。 这不过是一个瞬间的工作 使用内联源。 (见 SectionA 28 4 2,一个Validatorsa照本宣科 对于这样一个例子。)

找到下面的一个例子的源代码定义一个基于jruby bean 直接在Spring XML配置文件使用 内联: 符号。 (注意使用& lt; 字符表示一个 ' < ' 字符。 在这种情况下 周围的内联源 < ![CDATA[]]> 地区可能更好。)

```
<lang:jruby id="messenger" script-interfaces="org.springframework.scripting.Messenger">
    <lang:inline-script>
require 'java'

include_class 'org.springframework.scripting.Messenger'

class RubyMessenger &lt; Messenger

 def setMessage(message)
  @@message = message
 end

 def getMessage
  @@message
 end

end
    </lang:inline-script>
    <lang:property name="message" value="Hello World!" />
</lang:jruby>
```

理解构造函数注入上下文中的动态语言支持bean

有一个 非常 重要的事情需要注意的 对于Spring的动态语言支持。 也就是说,它不是(目前) 可以供应构造函数参数,动态语言支持bean(因此 构造函数注入是不能用于动态语言支持bean)。 的利益使这个特殊处理构造函数和 属性100%清楚,下面的混合的代码和配置 将 不 工作。

```
// from the file 'Messenger.groovy'
package org.springframework.scripting.groovy;

import org.springframework.scripting.Messenger

class GroovyMessenger implements Messenger {

    GroovyMessenger() {}

    // this constructor is not available for Constructor Injection
    GroovyMessenger(String message) {
        this.message = message;
    }

    String message

    String anotherMessage
}
```

```
<lang:groovy id="badMessenger"
    script-source="classpath:Messenger.groovy">

    <!-- this next constructor argument will *not* be injected into the GroovyMessenger -->
    <!--     in fact, this isn't even allowed according to the schema -->
    <constructor-arg value="This will *not* work" />

    <!-- only property values are injected into the dynamic-language-backed object -->
    <lang:property name="anotherMessage" value="Passed straight through to the dynamic-language-backed object" />

</lang>
```

在实践中这种限制并不重要,因为它首次出现自 setter注入是注入风格深受绝大多数 无论如何(让我们的开发人员离开讨论是否如一个好的 事一天)。

28.3.2A JRuby豆

从JRuby主页...

一个 JRuby是一个100%纯java实现的Ruby编程语言。 一个

在保持与春天哲学提供选择,春天的 动态语言支持也支持bean定义在JRuby 语言。 JRuby语言是基于非常直观 Ruby语言,支持内联正则表达式,街区 (关闭),以及一大堆其他的特性,做出解决方案 对于一些域问题一大堆更容易发展。

JRuby库依赖关系

JRuby脚本支持在春天需要以下 图书馆是在您的应用程序的类路径中。

- jruby jar

JRuby的实现动态语言支持在春天 有趣的,这是会发生什么:春天创建一个JDK动态 代理执行所有的接口中指定的 的脚本接口的属性值的 < lang:ruby > 元素(这是为什么 你 必须 提供至少一个接口的值 的属性,(因此)程序接口使用 jruby支持bean)。

让我们来看一个完整的工作的例子,使用一个基于jruby bean。 这是 JRuby实现的 信使 接口定义是本章早些时候(为了方便它下面是重复)。

```
package org.springframework.scripting;

public interface Messenger {

    String getMessage();
}
```

```
require 'java'

class RubyMessenger
   include org.springframework.scripting.Messenger

   def setMessage(message)
      @@message = message
   end

   def getMessage
      @@message
   end
end

# this last line is not essential (but see below)
RubyMessenger.new
```

这是春天的XML定义的一个实例 RubyMessenger JRuby bean。

```
<lang:jruby id="messageService"
         script-interfaces="org.springframework.scripting.Messenger"
         script-source="classpath:RubyMessenger.rb">

   <lang:property name="message" value="Hello World!" />

</lang:jruby>
```

注意最后一行,JRuby源(“RubyMessenger.new”)。 当使用JRuby上下文中的Spring的动态语言支持,你是鼓励 实例化并返回一个新实例的JRuby类,您想要使用作为一个 动态语言支持bean作为执行的结果你的JRuby源。 你 可以实现通过简单的实例化一个新的实例的JRuby类去年吗 线的源文件,类似于:

```
require 'java'

include_class 'org.springframework.scripting.Messenger'

# class definition same as above...

# instantiate and return a new instance of the RubyMessenger class
RubyMessenger.new
```

如果你忘了这样做,这不是世界末日,这将然而导致 春天不得不拖网(反映地)通过类型表示你的JRuby类 寻找一个类来实例化。在大的计划来说这将是如此之快 你永远也不会注意到它,但它是可以避免的东西通过简单 有一个线如上图作为最后一行的JRuby脚本。 如果你不 供应这样的线,或如果弹簧无法找到一个JRuby脚本实例化类 然后一个不透明的ScriptCompilationException 将被立即执行后由JRuby来源 解释器。 关键的文本,认为这是一个的根源 例外可以立即发现下面(所以,如果你的Spring容器 把下面的异常在创建你的动态语言支持bean 和下面的文本是在相应的异常堆栈,这将有希望 让你确定,然后轻松地纠正这个问题):

org.springframework.scripting。 ScriptCompilationException:编译脚本返回的JRuby”

为了纠正这一点,只需实例化一个新的实例的任何类 你想暴露作为一个jruby动态语言支持bean(如上所示)。 请 还要注意,您可以定义为很多类和对象 当你想要在你的JRuby脚本;重要的是 源文件作为一个整体必须返回一个对象(Spring配置)。

看到 SectionA 28.4,一个Scenariosa 对于一些 场景,你可能想要使用jruby建立豆子。

28.3.3A Groovy bean

从Groovy主页...

一个 Groovy是一种敏捷的动态语言,Java 2平台 的许多特性,人们喜欢这么多的语言,像Python、Ruby 和Smalltalk,使它们可用于Java开发人员使用一个类似Java的语法。 一个

如果你读了这一章直接由上面,您就已经 看过一个例子 的 groovy的动态语言支持bean。 让我们看看另一个例子(再一次 使用一个例子从弹簧测试套件)。

Groovy库依赖关系

在春天的Groovy脚本支持需要以下 图书馆是在您的应用程序的类路径中。

- groovy 1 5 5罐
- asm 2 2 2罐
- antlr 2 7 6罐

```
package org.springframework.scripting;

public interface Calculator {

    int add(int x, int y);
}
```

这是一个实现的 计算器 接口在Groovy。

```
// from the file 'calculator.groovy'
package org.springframework.scripting.groovy

class GroovyCalculator implements Calculator {

    int add(int x, int y) {
        x + y
    }
}
```

```
<-- from the file 'beans.xml' -->
<beans>
    <lang:groovy id="calculator" script-source="classpath:calculator.groovy"/>
</beans>
```

最后,这里是一个小的应用程序来练习上面的配置。

```
package org.springframework.scripting;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void Main(String[] args) {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");
        Calculator calc = (Calculator) ctx.getBean("calculator");
        System.out.println(calc.add(2, 8));
    }
}
```

得到的输出结果运行上面的程序将 (意外) 10 。 (激动人心的例子,嗯? 记住,目的是说明 概念。 请查阅动态语言展示项目 更复杂的例子,或者事实上 SectionA 28.4,一个Scenariosa 在本章后面)。

,这是很重要的 不 定义一个以上的 类/ Groovy源代码文件。 虽然这是完全合法的在Groovy中,它 是一个不好的现象(毫无疑问):在利益一致的方法, 你应该(在意见的作者)尊重标准Java 约定一个(公共)类的每个源文件。

通过回调Groovy对象的定制

这个 GroovyObjectCustomizer 接口是一个回调函数,允许你钩附加 创建逻辑的过程中创建一个groovy支持bean。 例如,实现这个接口可以调用 任何所需的初始化方法(s),或设置一些默认属性 值,或指定一个自定义 元类 。

```
public interface GroovyObjectCustomizer {

  void customize(GroovyObject goo);
}
```

Spring框架将实例化一个实例,你的groovy支持 豆,然后通过创建的 GroovyObject 到指定的 GroovyObjectCustomizer 如果一个人被定义。 你可以做任何你喜欢的供应 GroovyObject 参考:这是预期 ,设置一个自定义的 元类 是大多数 人们会想做与这个回调,你可以看到一个示例 下面的这样做。

```
public final class SimpleMethodTracingCustomizer implements GroovyObjectCustomizer {

  public void customize(GroovyObject goo) {
    DelegatingMetaClass metaClass = new DelegatingMetaClass(goo.getMetaClass()) {

      public Object invokeMethod(Object object, String methodName, Object[] arguments) {
        System.out.println("Invoking '" + methodName + "'.");
        return super.invokeMethod(object, methodName, arguments);
      }
    };
    metaClass.initialize();
    goo.setMetaClass(metaClass);
  }
}
```

一个完整的讨论在Groovy元编程是范围以外的 Spring参考手册。 咨询相关的部分Groovy 参考手册,或者做一个在线搜索:有很多文章 关于这个话题。 实际上使用 GroovyObjectCustomizer 很容易如果你使用Spring 2.0名称空间支持。

```
<!-- define the GroovyObjectCustomizer just like any other bean -->
<bean id="tracingCustomizer" class="example.SimpleMethodTracingCustomizer" />

  <!-- ... and plug it into the desired Groovy bean via the 'customizer-ref' attribute -->
  <lang:groovy id="calculator"
    script-source="classpath:org/springframework/scripting/groovy/Calculator.groovy"
    customizer-ref="tracingCustomizer" />
```

如果你不使用Spring 2.0名称空间支持,你还可以 使用 GroovyObjectCustomizer 功能。

```
<bean id="calculator" class="org.springframework.scripting.groovy.GroovyScriptFactory">
    <constructor-arg value="classpath:org/springframework/scripting/groovy/Calculator.groovy"/>
    <!-- define the GroovyObjectCustomizer (as an inner bean) -->
    <constructor-arg>
      <bean id="tracingCustomizer" class="example.SimpleMethodTracingCustomizer" />
    </constructor-arg>
</bean>

<bean class="org.springframework.scripting.support.ScriptFactoryPostProcessor"/>
```

28.3.4A BeanShell豆

从BeanShell主页...

一个 BeanShell是一个小的,免费的,可嵌入的Java源代码解释器 用动态语言的特性,用Java编写的。 BeanShell动态 执行标准的Java语法和扩展了它与常见的脚本 的便利,例如松散类型、命令和方法闭包喜欢那些 在Perl和JavaScript。 一个

BeanShell库依赖关系的

在春天的BeanShell脚本支持需要以下 图书馆是在您的应用程序的类路径中。

- bsh - 2.0 - b4 jar

与Groovy,BeanShell-backed bean定义需要一些(小) 额外的配置。
BeanShell的实现动态语言 支持在春天是有趣的,这是会发生什么:弹簧产生 一个JDK动态代理实现的所有接口中指定的 的脚本接口的 属性值的 < lang:bsh > 元素(这是为什么 你 必须 提供至少一个接口的值 的属性,(因此)程序接口使用 BeanShell-backed bean)。 这就意味着,每一个BeanShell-backed上的方法调用 对象是经历JDK动态代理调用机制。

让我们来看一个完整的工作的例子,使用一个BeanShell-based bean 实现 信使 接口 这是定义在本章早些时候(下面再重复一遍方便)。

```
package org.springframework.scripting;

public interface Messenger {

    String getMessage();
}
```

这里是BeanShell “实现” (这个词用在这里是松散的 信使 接口。

```
String message;

String getMessage() {
    return message;
}
```

```
void setMessage(String aMessage) {
    message = aMessage;
}
```

这是春天的XML定义了一个“实例”上面的“类” (再一次,使用这个词非常松散这里)。

```
<lang:bsh id="messageService" script-source="classpath:BshMessenger.bsh"
    script-interfaces="org.springframework.scripting.Messenger">

    <lang:property name="message" value="Hello World!" />
</lang:bsh>
```

看到 SectionA 28.4,一个Scenariosa 对于一些 场景,你可能想要使用BeanShell-based豆子。

28.4一个场景

可能的场景定义Spring托管bean在脚本 语言将是有益的,当然,多种多样的。 本节 描述了两种可能的用例为动态语言支持在春天。

28.4.1A照本宣科的Spring MVC控制器

一组类,可能受益于使用动态语言支持 豆子是Spring MVC控制器。 在纯Spring MVC应用程序, 这个导航流过一个web应用程序是一个很大程度上 由代码封装在Spring MVC控制器。 随着导航流和其他表示层逻辑web 应用程序需要更新响应支持问题或改变 业务需求,它可能更容易影响任何这样的要求 通过编辑一个或更多的变化动态语言源文件和看到 这些变化被立即反映在一个运行的状态 应用程序。

记住,在轻量级架构模型支持项目 如春天,你通常旨在有真的 薄 表示层,所有的肉的业务 一个应用程序的逻辑被包含在域和服务层 类。 开发Spring MVC控制器作为动态语言支持bean 允许你改变表示层逻辑通过简单地编辑和保存 文本文件,任何改变这样的动态语言源文件将取决于 在配置)自动反映在bean的支持 通过动态语言源文件。

注意

为了效果这自动“皮卡”的任何变化 对动态语言支持bean时,您将不得不启用 “可刷新的豆子”功能。 看到 一个章节beansa可刷新 对于一个完整的治疗 这个功能的。

找到下面的一个例子 org.springframework.web.servlet.mvc.Controller 使用Groovy实现动态语言。

```
// from the file '/WEB-INF/groovy/FortuneController.groovy'
package org.springframework.showcase.fortune.web

import org.springframework.showcase.fortune.service.FortuneService
import org.springframework.showcase.fortune.domain.Fortune
import org.springframework.web.servlet.ModelAndView
import org.springframework.web.servlet.mvc.Controller

import javax.servlet.http.HttpServletRequest
import javax.servlet.http.HttpServletResponse

class FortuneController implements Controller {

    @Property FortuneService fortuneService

    ModelAndView handleRequest(
            HttpServletRequest request, HttpServletResponse httpServletResponse) {

        return new ModelAndView("tell", "fortune", this.fortuneService.tellFortune())
    }
}
```

```
<lang:groovy id="fortune"
            refresh-check-delay="3000"
            script-source="/WEB-INF/groovy/FortuneController.groovy">
    <lang:property name="fortuneService" ref="fortuneService"/>
</lang:groovy>
```

28.4.2A脚本验证器

另一个领域的应用开发与弹簧可能受益 从所提供的灵活性动态语言支持bean是 验证。 它 可能 更容易表达复杂的验证 逻辑使用松散类型动态语言(可能也有支持 对于内联正则表达式)相对于普通Java。

再次,发展为动态语言支持bean验证器允许你改变 验证逻辑通过简单地编辑和保存一个简单的文本文件,任何这样的 变化将(取

决于配置)自动反映 在执行过程中运行的应用程序,而且不需要重新启动 一个应用程序。

注意

请注意,为了效果自动“皮卡”的任何变化 对动态语言支持bean时,您将不得不启用 “可刷新的豆子”功能。 看到 一个章节beansa可刷新 对于一个完整的和 详细的治疗这个特性。

找到下面的一个例子。春天 org.springframework.validation.Validator 使用Groovy实现动态语言。 (见 SectionA 7.2,一个验证使用Spring的 验证器 interfacea 对于一个讨论 验证器 接口)。

```
import org.springframework.validation.Validator
import org.springframework.validation.Errors
import org.springframework.beans.TestBean

class TestBeanValidator implements Validator {

    boolean supports(Class clazz) {
        return TestBean.class.isAssignableFrom(clazz)
    }

    void validate(Object bean, Errors errors) {
        if(bean.name?.trim()?.size() > 0) {
            return
        }
        errors.reject("whitespace", "Cannot be composed wholly of whitespace.")
    }
}
```

28.5一个零零碎碎

最后这部分包含一些零零碎碎的相关动态语言 支持。

28.5.1A AOP——建议脚本bean

可以使用Spring AOP框架建议脚本bean。 Spring AOP框架实际上是没有意识到一个bean被 建议可能是一个脚本bean,所以所有的AOP用例和功能 ,你可能会使用或目的使用将使用脚本bean。 有 只有一个(小)的事情,你需要意识到当建议照本宣科 豆子..... 你不能使用基于类的代理,您必须使用 基于接口的代理 。

你当然不只是限于建议脚本bean..... 你可以 自己也写在一个支持方面动态语言和使用这种 豆子建议其他Spring bean。 这真的是一个先进的使用 动态语言支持虽然。

28.5.2A范围

如果它没有立即明显,脚本bean可以当然是作用域 就像任何其他bean。 这个 范围 属性 各种 < lang:语言/ > 元素允许您 控制范围的潜在脚本bean,就像它在一个 正规的bean。 (默认的范围 singleton ,就像它 与常规的bean)。

下面的示例使用找到的 范围 属性 定义一个Groovy bean作用域作为一个 原型 。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:lang="http://www.springframework.org/schema/lang"
      xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/lang http://www.springframework.org/schema/lang/spring-lang.xsd">

    <lang:groovy id="messenger" script-source="classpath:Messenger.groovy" scope="prototype">
        <lang:property name="message" value="I Can Do The RoboCop" />
    </lang:groovy>

    <bean id="bookingService" class="x.y.DefaultBookingService">
        <property name="messenger" ref="messenger" />
    </bean>

</beans>
```

看到 SectionA 5.5,一个scopesaBean 在 ChapterA 5, IoC容器 为更全面的讨论范围支持Spring框架。

28.6进一步资源

找到下面的其它资源的链接各种动态语言描述 在这一章。

- 这个 JRuby 主页
- 这个 groovy 主页
- 这个 BeanShell 主页

一些更活跃的社区成员春天还添加了支持 许多额外的动态语言,超越那些覆盖在这 一章。 虽然它是可能的,这样的第三方的贡献可能被添加到 支持的语言列表主要弹簧分布,你最好的选择,看到 如果你最喜欢的脚本语言是支持的 Spring模块项目 。

29。 一个缓存抽象

29.1一个介绍

从版本3.1,Spring框架提供了支持透明的 添加缓存到一个现有的Spring应用程序。 类似 事务 支持,缓存抽象允许使用不同的缓存一致性 解决方案与最小影响的代码。

29.2一个理解抽象的缓存

在其核心,抽象应用缓存来Java方法,减少因此处决人数的根据 信息存在于缓存。 那是,每次一个 针对性 方法被调用的抽象 将应用一个缓存行为检查是否已经执行的方法对给定的参数。 如果它有, 然后返回缓存的结果无需执行实际的方法;如果没有,那么,在执行方法 结果缓存和返回给用户,这样,下次该方法被调用时,缓存的结果返回。 这样,昂贵的方法(无论是CPU或IO绑定)可以只执行一次为一组给定的参数和结果 重用而不需要实际执行方法再次。 缓存逻辑应用透明没有任何干扰 到调用程序。

> **缓存与缓冲**
>
> 术语“缓冲”和“缓存”往往交替使用,但是要注意他们代表不同的东西。 一个缓冲区用于传统上作为一个中间临时存储数据的快速和慢速之间的实体。 作为一个 党必须 等待 对于其他影响性能,缓冲缓解通过 让整个数据块的一次移动,而不是在小块。 数据写和读只有一次从 缓冲。 此外,该缓冲区 可见 至少一方是意识到它。
>
> 一个缓存另一方面根据定义是隐藏的,任何一方都没有意识到缓存发生。 它提高了 性能但这是否允许相同的数据读取多个次快时尚。
>
> 进一步的解释之间的差异两个可以发现 这里 。

> **重要**
>
> 显然这种方法只能处理方法,保证能够返回相同的输出(结果)对于一个给定的输入 (或参数)不管有多少次被处决。

使用缓存的抽象,开发者需要照顾两个方面:

- 缓存宣言——识别方法需要缓存和他们的政策
- 缓存配置——支持缓存的数据存储和读取

注意,就像其他服务在Spring框架,缓存服务是一个抽象(不是一个缓存实现)和需要 使用一个实际的存储来存储缓存数据——即抽象使得开发人员不用写缓存 逻辑但不提供实际的商店。 有两个集成提供开箱即用的,对于JDK java util并发concurrentmap 和 EHCache ——看到 SectionA 29.6,一个插入不同的后端cachesa 更多信息插入其他缓存存储/提供者。

29.3声明基于注解的缓存

对于缓存宣言,抽象提供了两个Java注释: @Cacheable 和 @CacheEvict 允许方法 触发缓存人口或缓存驱逐。 让我们仔细看看每一个注释:

29.3.1A @Cacheable 注释

顾名思义, @Cacheable 是用来标定方法可缓存——也就是说,方法为谁结果存储到缓存吗 所以在随后的调用(有相同的参数),缓存中的值返回而不需要实际执行该方法。 在其最简单的形式, 注释声明需要缓存的名称相关的带注释的方法:

```
@Cacheable("books")
public Book findBook(ISBN isbn) {...}
```

在上面的代码片段中,该方法 findBook 伴随着缓存命名 书 。 每次调用该方法时,缓存 检查是否已经执行的调用,不需要重复。 虽然在大多数情况下,只有一个缓存是宣布,注释允许多个 指定名称,以便更多然后一个缓存使用。 在这种情况下,每个缓存将之前检查执行方法——如果至少一个缓存时, 然后相关的值将被返回:

> **注意**
>
> 所有其他的缓存,不包含该方法将更新为好,即使缓存的方法并不实际 执行。

```
@Cacheable({ "books", "isbns" })
public Book findBook(ISBN isbn) {...}
```

缺省密钥生成

因为缓存基本上是键值存储,每个调用的一个缓存的方法需要被翻译成合适的密钥缓存访问。 盒子之外,缓存使用一个简单的抽象 KeyGenerator 基于以下算法:

- 如果没有给出参数,返回0。
- 如果只有一个参数是给定的,返回该实例。
- 如果更多的人给出了参数,返回一个关键计算从散列的所有参数。

这种方法适用于对象 自然键 只要 hashCode() 反映了这一点。 如果不是这种情况,那么 对分布式或持久的环境中,战略需要被改变的对象是不会保留hashCode。 事实上,根据不同的JVM实现或运行条件下,相同的hashCode可以重用不同的对象,在同一个VM实例。

提供一个不同的 默认 键生成器,一个需要实现的 org.springframework.cache.KeyGenerator 接口。 一旦配置完成,发电机将用于每个声明这并不指定自己的密钥生成策略(见下文)。

自定义键生成申报

因为缓存是通用的,它很可能是目标的方法有各种各样的签名,不能简单地映射上的缓存结构。 这往往成为 很明显当目标方法有多个参数,其中只有一些适合缓存(其余只用于该方法的逻辑)。 例如:

```
@Cacheable("books")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed
```

乍一看,这两个 布尔 参数影响这本书发现,他们没有使用缓存。 而且如果只有一两个 是重要的,而后者是不?

对于这种情况, @Cacheable 注释允许用户指定如何生成的关键是通过它的 关键 属性。 开发人员可以使用 ? 选择感兴趣的参数(或他们的嵌套的属性),执行操作甚至调用任意方法没有 不必编写任何代码或实现任何接口。 这是推荐的方法 默认 发电机自方法往往是相当不同的签名作为代码基生长;默认的策略可能会为一些人工作的方法,它为所有的方法很少。

下面是一些例子的各种声明?——如果你不熟悉它,帮自己一个忙,读 ChapterA 8, 春天表达式语言(?) :

```
@Cacheable(value="books", key="#isbn"
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)


@Cacheable(value="books", key="#isbn.rawNumber")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)


@Cacheable(value="books", key="T(someType).hash(#isbn)")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)
```

上面的代码片段,展示了为什么很容易选择一个特定的参数,它的一个属性或甚至一个任意的(静态)方法。

条件缓存

有时,一种方法可能不适合缓存所有的时间(例如,它可能取决于给定的参数)。 缓存注释支持这样的功能 通过 条件 参数需要 ? 表达式的求值结果要么 真正的 或 假 。 如果 真正的 ,方法是缓存——如果不是,它表现为如果方法不缓存,这是由于时间执行每一个不管什么值是在缓存或什么 使用的参数。 一个简单的例子,下面的方法将被缓存,只有如果论点 名称 有一个长度较短的32:

```
@Cacheable(value="book", condition="#name.length < 32")
public Book findBook(String name)
```

此外, 条件 参数, 除非 参数可用于否决增加一个值到缓存。 不像 条件 , 除非 ? 表达式是evalulated 在 这个方法被调用。 扩大在前面的例子,也许我们只需要缓存的平装书:

```
@Cacheable(value="book", condition="#name.length < 32", unless="#result.hardback")
public Book findBook(String name)
```

可用缓存 ? 评估上下文

每个 ? 又一个专用的表达式计算 上下文 。 此外 到构建在参数,该框架提供了专门的缓存相关的元数据,比如参数名称。 下表列

出了项目可用的上下文 所以我们可以使用它们为关键和条件(参见下一节)计算:

29.1为多。 可用的元数据缓存?

名称	位置	描述	例子
methodName	根对象	被调用的方法名	#root.methodName
方法	根对象	调用的方法	#root.method.name
目标	根对象	目标对象被调用	#root.target
targetClass	根对象	这个类被调用的目标	#root.targetClass
args	根对象	(数组)的参数用于调用目标	#root.args[0]
缓存	根对象	收集的缓存对当前方法是执行	#root.caches[0].name
参数名称	评估上下文	名称的任何方法参数。 如果由于某种原因没有名称(例:没有调试信息), 参数名称也可根据 < #参数> 在 #参数 代表论点指数(从0开始)。	iban 或 a0 (还可以使用 p0 或 p < #参数> 符号作为一个别名)。
结果	评估上下文	方法调用的结果(缓存的值)。 只能在 " 除非 的表情和 " 缓存驱逐 " 表达式(当 beforeInvocation 是 假)。	#result

29.3.2A @CachePut 注释

对于缓存的情况下需要更新没有干扰的方法执行,一个可以使用 @CachePut 注释。 那是,该方法将永远 被执行和结果放在缓存中(根据 @CachePut 选项)。 它支持相同的选项 @Cacheable 和应该使用 对于缓存人口而不是方法流程优化。

注意,使用 @CachePut 和 @Cacheable 注释在相同的方法通常是沮丧,因为他们有不同的行为。 而后者 使方法执行要跳过使用缓存,前部队执行为了执行缓存更新。 这导致意想不到的行为,除了特定的 角情况下(如注释有条件,排除他们从对方),这样的声明应该被避免。

29.3.3A @CacheEvict 注释

缓存抽象允许不只是人口的一个缓存存储也驱逐。 这个过程有助于去除陈旧或未使用的数据从缓存中。 反对 @Cacheable 、注释 @CacheEvict 划定方法执行缓存 驱逐 ,这是方法,作为触发器 数据从缓存中移除。 就像它的兄弟姐妹, @CacheEvict 需要指定一个(或多个)缓存所影响的行动,允许 键或指定一个条件,但除此之外,有一个额外的参数 allEntries 这说明是否缓存宽驱逐需要执行吗 而不是只是一个条目(基于关键):

```
@CacheEvict(value = "books", allEntries=true)
public void loadBooks(InputStream batch)
```

这个选项方便当整个缓存区域需要清理出来——而非驱逐每个条目(这将花很长时间,因为它是低效的), 所有的条目在一个操作是删除如上所示。 注意,这个框架将忽略任何键指定在这个场景中,因为它不适用(整个缓存不仅驱逐 一个条目)。

一个还可以指示驱逐应该发生在(默认)或之前通过的方法执行 beforeInvocation 属性。 前者提供了相同的语义和其余的注释

——一旦方法成功完成,一个动作(在本例中驱逐)执行缓存。 如果这个方法不 执行(因为它可能会缓存)或者抛出一个异常,驱逐不发生。 后者(beforeInvocation = true)使拆迁发生总是,在方法 被调用——这是有用的情况下收回不需要绑定到该方法的结果。

重要的是要注意,void方法可以使用 @CacheEvict ——方法作为触发器、返回值被忽略(因为他们不相互作用 缓存)——这并不是这种情况 @Cacheable 添加/更新数据到缓存,因此需要一个结果。

29.3.4A @Caching 注释

有一些情况下,多个注释相同类型的,比如 @CacheEvict 或 @CachePut 需要指定,例如因为条件或关键 表达式是不同的在不同的缓存。 不幸的是Java不支持这样的声明但是有一个解决方案——使用一个 封闭 注释,在这种情况下, @Caching 。 @Caching 允许多个嵌套 @Cacheable , @CachePut 和 @CacheEvict 使用相同的方法:

```
@Caching(evict = { @CacheEvict("primary"), @CacheEvict(value = "secondary", key = "#p0") })
public Book importBooks(String deposit, Date date)
```

29.3.5A启用缓存注释

重要的是要注意,虽然宣布缓存注释并不会自动触发他们的行动——像很多东西在春天,特征是通过声明 使(这意味着如果你怀疑缓存是责怪,您可以禁用它通过消除只有一个配置行,而不是所有的注释你的代码)。

启用缓存注释添加注释 @EnableCaching 你的 @ configuration 类:

```
@Configuration
@EnableCaching
public class AppConfig {

}
```

另外对于XML配置使用 缓存:注解驱动的 元素:

```
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:cache="http://www.springframework.org/schema/cache"
  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
     http://www.springframework.org/schema/cache http://www.springframework.org/schema/cache/spring-cache.xsd">
  <cache:annotation-driven />
</beans>
```

两 缓存:注解驱动的 元素和 @EnableCaching 注释允许各种选项指定的方式影响 缓存行为添加到应用程序通过AOP。 配置是故意相似 与 transactional :

29.2为多。 一个缓存注释设置

XML属性	注释属性	默认	描述
缓存管理器	N / A(见CachingConfigurer Javadoc)	cacheManager	缓存管理器使用的名称。 只需要 如果缓存管理器的名称不是cacheManager ,例子 以上。
模式	模式	代理	默认的模式 “代理” 过程注释 豆子是代理使用Spring的AOP框架(以下 代理语义,正如上面所讨论的,申请方法调用 通过代理来只)。 另一种模式 “aspectj” 而不是编织类与春天的的影响 AspectJ缓存方面,修改目标类字节 代码,适用于任何类型的方法调用。 AspectJ编织 需要弹簧方面。 jar的类路径以及 装入时编织(或编译时编织)启用。 (见 一个章节弹簧configurationa 有关如何设置 装入时编织了。)
代理目标类	proxyTargetClass	假	只适用于代理模式。 控制什么类型的 缓存代理创建类标注 这个 @Cacheable 或 @CacheEvict 注释。 如果 代理目标类 属性设置到 真正的 ,然后基于类的代理 创建的。 如果 代理目标类 是 假 或者如果属性是省略了,然后 基于接口的代理创建标准JDK。 (见 SectionA 9.6,一个mechanismsa代理 对于一个详细的检查 不同的代理类型。)
			定义的命令缓存的建议 应用于bean注释吗 @Cacheable 或 @CacheEvict 。 (更多 相关规定信息订购AOP的建议, 看到 一个章

秩序	秩序	下令最低优先级	节orderinga建议)。 没有 指定顺序意味着AOP子系统决定 订单的建议。

注意

<缓存:注解驱动/ > 只希望 @Cacheable / @CacheEvict 在bean在相同的 应用程序上下文定义在。 这意味着,如果你把 <缓存:注解驱动/ > 在一个 WebApplicationContext 对于一个 DispatcherServlet ,它只检查 @Cacheable / @CacheEvict 豆子在你 控制器,而不是你的服务。 看到 SectionA 17.2,一个了 DispatcherServlet 一个 为更多的信息。

提示

春天建议你只标注具体类(和 方法的具体类) @Cache * 注释,而不是 来注解的接口。 你当然可以把 @Cache * 注释一个 接口(或一个接口方法),但这只能像你 期望它如果您使用的是基于接口的代理。 事实上, Java注释是 没有继承接口 意味着,如果您使用的是基于类的代理 (代理目标类= " true ")或编织的基础 方面(模式= " aspectj"),那么缓存 设置不认可的代理和编织 基础设施,和对象将不会被包裹在一个 缓存代理,这将是明显的 坏 。

方法可视性和 @Cacheable / @CachePut / @CacheEvict

当使用代理,你应该申请 @Cache * 注释只 方法 公共 可见性。 如果你 注释保护,私人或包可见方法与这些注释, 没有误差是提高了,但是带注释的方法不存在配置 缓存设置。 考虑使用AspectJ(见下文)如果你 需要标注非公有制方法它改变了字节码本身。

注意

在代理模式(默认),只有外部方法调用 通过代理来截获。 这意味着 自动调用,实际上,在目标对象的方法调用 另一种方法的目标对象,不会导致一个实际 缓存运行时即使调用方法是标记 @Cacheable ——考虑使用aspectj模式在这种情况下。

29.3.6A使用自定义注释

缓存抽象允许使用她自己的注解来识别方法触发缓存人口或驱逐。 这是非常方便作为模板机制,因为它消除了 需要复制缓存注释声明(特别有用如果键或条件指定)或者如果外国进口(org.springframework)不允许 在您的代码库。 类似于其余的 原型 注释,都 @Cacheable 和 @CacheEvict 可以用作元注释,注释,可以标注其他注释。 也就是说,让我们代替常见的 @Cacheable 用我们自己的宣言,定制 注释:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
@Cacheable(value= "books" , key="#isbn")
public @interface SlowService {
}
```

在上面,我们定义我们自己的 SlowService 注释这本身就是注释 @Cacheable ——现在我们可以替换下面的代码:

```
@Cacheable(value="books", key="#isbn")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)
```

与:

```
@SlowService
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)
```

尽管 @SlowService 不是一个Spring注释,容器会自动捡起它在运行时声明并理解意思。 请注意,作为 提到 以上 ,注解驱动的行为需要启用。

29.4基于xml的声明式缓存

如果注释不一个选项(没有访问来源或没有外部代码),一个可以使用XML声明性的缓存。 所以不要注释方法缓存,一个指定 目标方法和外部缓存指令(类似于声明式事务管理 建议)。 前面的示例 可以翻译成:

```
<!-- the service we want to make cacheable -->
```

```
<bean id="bookService" class="x.y.service.DefaultBookService"/>

<!-- cache definitions -->
<cache:advice id="cacheAdvice" cache-manager="cacheManager">
   <cache:caching cache="books">
      <cache:cacheable method="findBook" key="#isbn"/>
      <cache:cache-evict method="loadBooks" all-entries="true"/>
   </cache:caching>
</cache:advice>

<!-- apply the cacheable behavior to all BookService interfaces -->
<aop:config>
   <aop:advisor advice-ref="cacheAdvice" pointcut="execution(* x.y.BookService.*(..))"/>
</aop:config>
...
// cache manager definition omitted
```

在上面的配置, bookService 是可缓存的。 语义缓存被封装在应用 缓存:建议 定义这 指示方法 findBooks 用于将数据放入缓存而法 loadBooks 将数据。 两种定义都是工作着 书 缓存。

这个 aop:配置 定义应用缓存建议适当的点在程序中通过使用AspectJ切入点表达式(更多的信息是可用的 在 ChapterA 9, 面向方面的编程与弹簧)。 在上面的示例中,所有的方法 BookService 被认为是和缓存的建议适用于他们。

声明XML缓存支持所有的基于注解的模型,两者之间移动应该是相当容易的,而且两者都可以用在相同的应用程序。 基于XML的方法不能触摸目标代码但是它本质上是更详细的;在处理类的重载方法为目标的缓存,识别 适当的方法需要额外的努力自 方法参数不是一个好的鉴别器——在这些情况下,AspectJ切入点可以用来挑选目标 方法和应用适当的缓存功能。 然而通过XML,它是更容易应用一个包/组/接口宽缓存(再次由于AspectJ切入点)和创建 模版样的定义(就像我们在上面的例子中通过定义目标缓存通过 缓存:定义 缓存 属性)。

29.5一个配置缓存存储

盒子之外,缓存抽象提供了集成两个储存设备——一个在顶部的JDK ConcurrentMap 和一个 对于 EHCache 图书馆。 使用它们,需要简单地声明一个合适的 cacheManager ——一个实体,控制和管理 缓存 年代和可以用来检索这些储存。

29.5.1A JDK ConcurrentMap 的 缓存

在jdk的基础 缓存 实现驻留在 org.springframework.cache.concurrent 包。 它允许一个人使用 ConcurrentHashMap 作为一个支持 缓存 商店。

```
<!-- generic cache manager -->
<bean id="cacheManager" class="org.springframework.cache.support.SimpleCacheManager">
   <property name="caches">
   <set>
      <bean class="org.springframework.cache.concurrent.ConcurrentMapCacheFactoryBean" p:name="default"/>
      <bean class="org.springframework.cache.concurrent.ConcurrentMapCacheFactoryBean" p:name="books"/>
   </set>
   </property>
</bean>
```

上面的代码片段中使用 SimpleCacheManager 创建一个 cacheManager 为两个,嵌套 并发 缓存 实现命名 默认 和 书 。 注意,名字是直接配置每个缓存。

因为缓存创建的应用程序,它被绑定到它的生命周期,使它适合基本用例,测试或简单的应用程序。 缓存可伸缩性和非常快 但它没有提供任何管理或持久化功能也不收回合同。

29.5.2A ehcache基础 缓存

坐落在的EhCache实现 org.springframework.cache.ehcache 包。 再次,使用它,只需要声明合适的 cacheManager :

```
<bean id="cacheManager" class="org.springframework.cache.ehcache.EhCacheCacheManager" p:cache-manager-ref="ehcache"/>

<!-- EhCache library setup -->
<bean id="ehcache" class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean" p:config-location="ehcache.xml"/>
```

这个设置可以引导ehcache图书馆内(通过Spring IoC bean EHCache),然后连接到专用的 cacheManager 实现。 注意整个ehcache具体配置是读从资源 ehcache xml 。

29.5.3A gemfire基础 缓存

GemFire是面向内存/磁盘支持,弹性可伸缩的、用之不竭,活跃(内置基于模式的订阅通知), 在全球范围内复制数据库,提供全功能的边缘缓存。 为进一步的信息关于如何使用GemFire作为缓存管理器(以及更多),请参考 到 弹簧GemFire参考文档 。

29.5.4A处理缓存没有后备存储器

有时当切换环境或做测试,一个可能缓存声明没有实际支持缓存配置。 因为这是一个无效的配置,在运行一个 异常将通过自缓存基础设施是无法找到一个合适的商店。 在这种情况下,而不是删除缓存声明(可以证明乏味的), 一个可以在一个简单的线,虚拟缓存,不执行缓存,缓存的方法是,部队执行每一次:

```
<bean id="cacheManager" class="org.springframework.cache.support.CompositeCacheManager">
  <property name="cacheManagers"><list>
    <ref bean="jdkCache"/>
    <ref bean="gemfireCache"/>
  </list></property>
  <property name="fallbackToNoOpCache" value="true"/>
</bean>
```

这个 CompositeCacheManager 高于链多个 cacheManager 年代,此外,通过 fallbackToNoOpCache 旗,添加一个 没有op缓存,所有的定义不是由配置的缓存管理器。 那是,每一个缓存中没有定义要么 jdkCache 或 gemfireCache (配置以上)将由op的没有缓存,它不会存储任何信息导致目标方法执行每一次。

29.6一个插入不同的后端缓存

显然有很多缓存产品可以用作后备存储器。 填补他们在,一个需要提供一个 cacheManager 和 缓存 实现从不幸的是没有可用的标准,我们可以使用相反。 这听起来可能更难那么它就是因为在实践中, 类往往是简单的 适配器 年代,地图缓存抽象框架顶部的存储API作为 EHCache 类可以显示。 大多数 cacheManager 类可以使用类的 org.springframework.cache.support 包,如 AbstractCacheManager 这负责样板式代码只留下实际吗 映射 要完成。 我们希望在时间,图书馆,提供集成与弹簧 可以填写这小配置差距。

29.7一个我如何设置TTL /创科实业/驱逐政策/ XXX特性?

直接通过您的缓存提供者。 缓存抽象是…… 嗯,一个抽象不是一个缓存实现。 你正在使用的解决方案可能支持各种数据策略和不同的 拓扑,其他解决方案不(例如JDK ConcurrentHashMap)——暴露,在缓存中抽象将是无用的,因为简单 没有后盾支持。 这样的功能应控制直接通过支持缓存,当配置或通过其本地API。

PartA七世一个附录

AppendixA一个。 一个经典的弹簧的使用

这个附录讨论一些经典的弹簧使用模式作为一个 参考Spring应用程序开发者维护遗产。 这些用法 模式不再反映推荐的方式使用这些特性和 目前推荐使用覆盖在各自的部分 参考手册。

一个。 1一个经典的ORM用法

这部分文档经典的使用模式,您可能 遇到一个遗留Spring应用程序。 目前推荐的 使用模式,请参考 ChapterA 15, 对象关系映射(ORM)数据访问 一章。

A.1.1A冬眠

为当前推荐使用模式为Hibernate看到 SectionA 15.3,一个Hibernatea

这个 hibernatetemplate

基本的编程模型为模板如下,因为 方法,可以部分的任何自定义数据访问对象或业务 服务。 没有限制的实施 周围的对象,它只需要提供一个Hibernate SessionFactory 。 它可以让后者 从任何地方,但最好是作为从一个Spring IoC bean引用 容器——通过一个简单的 setSessionFactory(. .) bean属性setter。 下面的代码片段显示刀在Spring容器定义, 引用上面的定义 SessionFactory ,的一个实例 DAO方法实现。

```
<beans>

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>

</beans>
```

```
public class ProductDaoImpl implements ProductDao {

    private HibernateTemplate hibernateTemplate;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.hibernateTemplate = new HibernateTemplate(sessionFactory);
    }

    public Collection loadProductsByCategory(String category) throws DataAccessException {
        return this.hibernateTemplate.find("from test.Product product where product.category=?", category);
    }
}
```

这个 hibernatetemplate 类提供了许多 方法,镜所公开的方法在冬眠 会话 接口,除了一个 数量的便利方法如上面所示。 如果你需要 访问 会话 调用方法 这不是暴露的 hibernatetemplate , 你总是可以降到一个基于回调的方法一样。

```
public class ProductDaoImpl implements ProductDao {

    private HibernateTemplate hibernateTemplate;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.hibernateTemplate = new HibernateTemplate(sessionFactory);
    }

    public Collection loadProductsByCategory(final String category) throws DataAccessException {
        return this.hibernateTemplate.execute(new HibernateCallback() {

            public Object doInHibernate(Session session) {
                Criteria criteria = session.createCriteria(Product.class);
                criteria.add(Expression.eq("category", category));
                criteria.setMaxResults(6);
                return criteria.list();
            }
        };
    }
}
```

一个回调实现有效可用于任何 Hibernate的数据访问。 hibernatetemplate 将 确保 会话 实例 正常开启和关闭,并自动参与 事务。 模板实例是线程安全的和可重用的, 因此可以作为实例变量保持周围的类。 对于简单的单步行动像一个发现、负载、saveOrUpdate, 或删除电话, hibernatetemplate 提供 选择方便的方法,可以代替这种一线 回调实现。 此外,Spring提供了一个方便的 HibernateDaoSupport 基类,提供一个 setSessionFactory(. .) 方法接收一个 SessionFactory ,和 getSessionFactory() 和 getHibernateTemplate() 使用子类。 在组合,这样可以非常简单的DAO实现 典型的要求:

```
public class ProductDaoImpl extends HibernateDaoSupport implements ProductDao {

    public Collection loadProductsByCategory(String category) throws DataAccessException {
        return this.getHibernateTemplate().find(
            "from test.Product product where product.category=?", category);
    }
}
```

实现基于spring dao没有回调

作为替代使用Spring的 hibernatetemplate 实现DAOs,数据 访问代码也可以写在一个更传统的方式,没有 包装Hibernate访问代码在一个回调,同时仍然 尊重和参与Spring的通用 DataAccessException 层次结构。 这个 HibernateDaoSupport 基类提供的方法 访问当前事务 会话 和转换异常 这样一个场景,类似的方法也可以作为静态的帮手 在 SessionFactoryUtils 类。 注意,这样的代码通常会通过的 假 " 的价值 这个 getSession(. .) 方法 " allowCreate 的参数,执行内运行 事务(避免了需要关闭返回 会话 ,因为它的生命周期管理 事务)。

```
public class HibernateProductDao extends HibernateDaoSupport implements ProductDao {

    public Collection loadProductsByCategory(String category) throws DataAccessException, MyException {
        Session session = getSession(false);
        try {
            Query query = session.createQuery("from test.Product product where product.category=?");
            query.setString(0, category);
            List result = query.list();
            if (result == null) {
                throw new MyException("No search results.");
            }
```

```
            return result;
        }
        catch (HibernateException ex) {
            throw convertHibernateAccessException(ex);
        }
    }
}
```

这种直接的优势是,它冬眠访问代码 允许 任何 检查应用程序异常 扔在数据访问代码;相比 hibernatetemplate 类,它被限制为只扔未经检查的异常在回调。 请注意,您 可以经常推迟相应的检查和投掷的 应用程序异常,回调后,仍然允许 处理 hibernatetemplate 。 一般来说, hibernatetemplate 类的便利方法 更简单、更方便的对许多场景。

A.1.2A JDO

为当前推荐使用模式为JDO看到 SectionA 15.4,一个JDOa

JdoTemplate 和 JdoDaoSupport

每一刀将接收基于jdo的 PersistenceManagerFactory 通过 依赖注入。 这样一个DAO可以编码与平原JDO API, 处理给定的 PersistenceManagerFactory ,但会 通常,而被用于Spring框架的 JdoTemplate :

```
<beans>

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="persistenceManagerFactory" ref="myPmf"/>
  </bean>

</beans>
```

```
public class ProductDaoImpl implements ProductDao {

    private JdoTemplate jdoTemplate;

    public void setPersistenceManagerFactory(PersistenceManagerFactory pmf) {
        this.jdoTemplate = new JdoTemplate(pmf);
    }

    public Collection loadProductsByCategory(final String category) throws DataAccessException {
        return (Collection) this.jdoTemplate.execute(new JdoCallback() {
            public Object doInJdo(PersistenceManager pm) throws JDOException {
                Query query = pm.newQuery(Product.class, "category = pCategory");
                query.declareParameters("String pCategory");
                List result = query.execute(category);
                // do some further stuff with the result list
                return result;
            }
        });
    }
}
```

一个回调实现可以有效地用于任何JDO 数据访问。 JdoTemplate 将确保 PersistenceManager 年代正确打开了, 关闭,自动参与交易。 模板 实例是线程安全的和可重用的,因此可以保留 实例变量周围的类。 对于简单的单步 动作,比如单 找到 , 负载 , makePersistent ,或 删除 电话, JdoTemplate 提供了方便的方法,可以代替替代等一行 回调实现。 此外,Spring提供了一个方便的 JdoDaoSupport 基类,提供一个 setPersistenceManagerFactory(. .) 方法 接收 PersistenceManagerFactory ,和 getPersistenceManagerFactory() 和 getJdoTemplate() 使用子类。 在 组合,这样可以非常简单的DAO实现 典型的要求:

```
public class ProductDaoImpl extends JdoDaoSupport implements ProductDao {

    public Collection loadProductsByCategory(String category) throws DataAccessException {
        return getJdoTemplate().find(
            Product.class, "category = pCategory", "String category", new Object[] {category});
    }
}
```

作为替代使用Spring的 JdoTemplate ,你也可以基于spring的代码 DAOs在JDO API级别,显式地打开和关闭 PersistenceManager 。 阐释在 相应的Hibernate部分,主要利用这个 方法是,你的数据访问代码能够抛出检查 例外。 JdoDaoSupport 提供多种 支持方法对于这个场景,获取和释放 事务性 PersistenceManager 作为 以及转换为例外。

A.1.3A JPA

目前推荐使用模式的JPA看到 SectionA 15.5,一个JPAa

JpaTemplate 和 JpaDaoSupport

每一个jpa基于随后将会受到一个刀 会 通过依赖 注射。 这样一个DAO可以编码与平原JPA和处理 鉴于 会 或通过 春天的JpaTemplate :

```
<beans>

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="entityManagerFactory" ref="myEmf"/>
  </bean>

</beans>
```

```
public class JpaProductDao implements ProductDao {

    private JpaTemplate jpaTemplate;

    public void setEntityManagerFactory(EntityManagerFactory emf) {
        this.jpaTemplate = new JpaTemplate(emf);
    }

    public Collection loadProductsByCategory(final String category) throws DataAccessException {
        return (Collection) this.jpaTemplate.execute(new JpaCallback() {
            public Object doInJpa(EntityManager em) throws PersistenceException {
                Query query = em.createQuery("from Product as p where p.category = :category");
                query.setParameter("category", category);
                List result = query.getResultList();
                // do some further processing with the result list
                return result;
            }
        });
    }
}
```

这个 JpaCallback了 实现 允许任何类型的JPA数据访问。 这个 JpaTemplate 将确保 EntityManager 年代正确打开了, 关闭并自动参与交易。 此外, JpaTemplate 正确处理异常,使 确定资源清理和适当的事务滚 回来。 模板实例是线程安全的和可重用的,他们可以 被保留的封闭类的实例变量。 注意, JpaTemplate 提供单步动作,例如 发现、负载、合并等方法,以及替代便利 可以更换一个线回调实现。

此外,Spring提供了一个方便的 JpaDaoSupport 基类,提供了 get / setEntityManagerFactory 和 getJpaTemplate() 用于 子类:

```
public class ProductDaoImpl extends JpaDaoSupport implements ProductDao {

    public Collection loadProductsByCategory(String category) throws DataAccessException {
        Map<String, String> params = new HashMap<String, String>();
        params.put("category", category);
        return getJpaTemplate().findByNamedParams("from Product as p where p.category = :category", params);
    }
}
```

除了使用Spring的 JpaTemplate ,一个人也可以基于spring的代码 DAOs对JPA,做一个自己的显式 EntityManager 处理。 也阐述了在相应的Hibernate部分,主要的优势 这种方法的,你的数据访问代码能够抛出 已检查的异常。 JpaDaoSupport 提供了一个 各种各样的支持方法对于这个场景,用于检索和 释放事务 EntityManager , 以及转换为例外。

JpaTemplate的存在是一个主要的JdoTemplate兄弟 和HibernateTemplate,提供相同的风格,人们过去 它。

一个。 2一个经典的Spring MVC

...

一个。 3一个JMS使用

春天的一个好处是盾的JMS支持用户 从差异和1.1 api的JMS 1.0.2中。 (用于描述 两个api之间的差异在域统一参见侧栏)。 因为它是现在普遍遇到只有JMS 1.1 API使用 类,是基于JMS API已经弃用1.0.2在春天 3.0。 本节描述春天JMS支持JMS 1.0.2中弃用类。

A.3.1A JmsTemplate

位于包 org.springframework.jms.core 类 JmsTemplate102 提供的所有特性 这个 JmsTemplate 描述了JMS章,但是 基于JMS 1.0.2 API而不是JMS 1.1 API。 作为一个结果, 如果您使用的是JmsTemplate102您需要设置布尔属性 pubSubDomain 配置 JmsTemplate 用知识的JMS域 被使

域统一

有两个主要版本的JMS规范,1.0.2和 1.1。

JMS定义了两种类型的消息传递1.0.2域,点对点(队列)和发布/订阅(主题)。 反映出了这些的1.0.2 API 两个消息传递域通过提供一个并行类

用。 默认情况下这个属性的值是false,指示 的点对点的域,队列,将被使用。

A.3.2A异步消息接收

MessageListenerAdapter的 是结合使用Spring的吗 消息 侦听器容器 支持异步消息的接收 暴露出几乎任何类作为消息驱动POJO。 如果你正在使用 JMS 1.0.2 API,您会希望使用特定的类,比如1.0.2 MessageListenerAdapter102 , SimpleMessageListenerContainer102 , 和 DefaultMessageListenerContainer102 。 这些类 提供相同的功能作为JMS 1.1同行,不过基础 仅仅依赖于JMS 1.0.2 API。

A.3.3A连接

这个 ConnectionFactory 接口的一部分 JMS规范和作为入口点处理 JMS。 Spring提供了一个实现的 ConnectionFactory 界面, SingleConnectionFactory102 基于JMS 1.0.2 API,将返回相同的 连接 在所有 createConnection() 电话和忽略调用 close() 。 您将需要设置布尔财产 pubSubDomain 说明哪些消息 域作为 SingleConnectionFactory102 将 总是明确区分 javax.jms.QueueConnection 和一个 javax.jmsTopicConnection 。

层次结构对于每个 域。 因此,一个客户端应用程序成为特定领域的 使用JMS API。 JMS 1.1引入的概念域统一 最小化两个功能差异和客户端 API 这两个领域之间的差异。 作为一个示例的一个功能 差别将被删除,如果你使用一个JMS 1.1提供者可以 事务性消息从一种消费领域和产生一个消息 在其他使用相同的 会话 。

注意

JMS 1.1规范于2002年4月发布和 合并作为J2EE 1.4在2003年11月。 因此,常见的J2EE 1.3应用服务器仍在广泛使用(如 作为BEA WebLogic 8.1和IBM WebSphere 5.1)是基于JMS 1.0.2中。

A.3.4A事务管理

在JMS 1.0.2环境类 JmsTransactionManager102 提供支持 管理JMS事务一个连接工厂。 请参考 参考文档 JMS事务 管理 在这个功能的更多信息。

AppendixA B。 一个经典的Spring AOP使用

本附录中我们讨论 Spring AOP的低级api和AOP支持Spring 1.2的应用程序中使用。 为新的应用程序,我们建议使用AOP支持Spring 2.0的 所描述的 aop 章,但当使用现有的应用程序, 或当阅读书籍和文章,你可能遇到Spring 1.2风格的例子。 Spring 2.0是完全向后兼容Spring 1.2和描述的一切 在本附录是完全支持Spring 2.0中。

B。 1一个切入点API在春天

让我们看看如何处理关键切入点春季概念。

B.1.1A概念

春天的切入点模型支持独立的切入点的重用 建议类型。 可以使用同样的目标不同的建议 切入点。

这个 org.springframework.aop.Pointcut 接口 是中央接口,用于目标建议特定的类 和方法。 完整的界面如下所示:

```
public interface Pointcut {
    ClassFilter getClassFilter();
    MethodMatcher getMethodMatcher();
}
```

分裂 切入点 界面分为两部分 允许重用类和方法匹配部分,和细粒度 组成操作(如执行一个“联盟”与另一个方法 匹配器)。

这个 ClassFilter 接口是用来限制 切入点来一组给定的目标类。 如果 matches() 方法总是返回true,所有目标 类将匹配:

```
public interface ClassFilter {
    boolean matches(Class clazz);
}
```

这个 MethodMatcher 接口通常更 重要的。 完整的界面如下所示:

```
public interface MethodMatcher {
    boolean matches(Method m, Class targetClass);
```

```
    boolean isRuntime();

    boolean matches(Method m, Class targetClass, Object[] args);
}
```

这个 匹配(方法、类) 方法用于 测试这个切入点是否会匹配一个给定的方法对一个目标 类。 这种评价可以执行一个AOP代理创建时, 避免需要一个测试在每个方法调用。 如果2参数 匹配方法返回true对于一个给定的方法 isRuntime() MethodMatcher 方法返回 诚然,3参数匹配方法将在每个方法调用 调用。 这使得一个切入点来看看参数传递给 方法调用之前立即目标建议 执行。

最MethodMatchers是静态的,这意味着他们的 isRuntime() 方法返回false。 在这种情况下, 3参数匹配方法永远不会被调用。

提示

如果可能的话,尽量使切入点的静态,允许AOP 框架来缓存结果评价的切入点当AOP代理 被创建。

B.1.2A操作切入点

Spring支持操作切入点:值得注意的是, 联盟 和 路口 。

- 联盟意味着方法,要么切入点匹配。
- 十字路口意味着方法这两个切入点匹配。
- 联盟通常是更有用的。
- 切入点可以组合使用静态方法 org.springframework.aop.support.Pointcuts 类,或 使用 ComposablePointcut 类在同一个 包。 然而,使用AspectJ切入点表达式通常是一个 更简单的方法。

B.1.3A AspectJ切入点表达式

因为2.0,最重要类型的切入点用春天 org.springframework.aop.aspectj.AspectJExpressionPointcut 。 这是一个切入点,使用AspectJ提供的库来解析一个AspectJ 切入点表达式字符串。

看到前面一章讨论支持AspectJ切入点 原语。

B.1.4A便利切入点实现

Spring提供了一些方便的切入点实现。 一些 可以用现成的,其他人是打算再在吗 特定于应用程序的切入点。

静态切入点

静态的切入点是基于方法和目标类, 不能考虑到方法的参数。 静态的切入点是 足够—— 和最好的 ——对于大多数用途。 它是可能的春天 评估一个静态的切入点只有一次,当一个方法第一次被调用: 在那之后,就不需要再次评估与每一个切入点 方法调用。

让我们考虑一些静态的切入点实现包括 与春天。

正则表达式的切入点

一个明显的方式来指定静态的切入点是常规 表达式。 几个AOP框架除了弹簧使这个 可能的。 org.springframework.aop.support.Perl5RegexpMethodPointcut 正则表达式是一个通用的切入点,使用Perl 5正则吗 表达式语法。 这个 Perl5RegexpMethodPointcut 类依赖于雅加达奥罗的正则表达式匹配。 春天 还提供了 JdkRegexpMethodPointcut 类 使用正则表达式支持JDK 1.4 +。

使用 Perl5RegexpMethodPointcut 类, 你可以提供一个列表的模式字符串。 如果任何这些是一个 匹配,切入点将评估为 true。 (所以结果是 有效的结合这些切入点。)

使用如下所示:

```
<bean id="settersAndAbsquatulatePointcut"
  class="org.springframework.aop.support.Perl5RegexpMethodPointcut">
  <property name="patterns">
    <list>
      <value>.*set.*</value>
      <value>.*absquatulate</value>
    </list>
  </property>
```

```
</bean>
```

Spring提供了一个方便的类, RegexpMethodPointcutAdvisor ,这允许我们 还引用一个建议(请记住,一个建议可以是一个 拦截器,在建议,抛出建议等)。 在幕后, 弹簧将使用一个 JdkRegexpMethodPointcut 。 使用 RegexpMethodPointcutAdvisor 简化了布线, 作为一个bean封装两个切入点和通知,如图所示 下图:

```
<bean id="settersAndAbsquatulateAdvisor"
   class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
   <property name="advice">
      <ref local="beanNameOfAopAllianceInterceptor"/>
   </property>
   <property name="patterns">
      <list>
         <value>.*set.*</value>
         <value>.*absquatulate</value>
      </list>
   </property>
</bean>
```

RegexpMethodPointcutAdvisor 可以用 任何意见类型。

属性驱动的切入点

一个重要的类型的静态的切入点是一个 元数据驱动的 切入点。 这种使用价值 元数据属性:通常,源代码级别的元数据。

动态切入点

动态的切入点是昂贵的比静态的评价 切入点。 他们考虑的方法 参数 以及静态信息。 这 意味着他们必须评估每个方法调用;结果不能被缓存的,作为参数会有所不同。

主要的例子是 控制流 切入点。

控制流的切入点

弹簧控制流的切入点在概念上类似于 AspectJ cflow 切入点,虽然不 强大的。 (目前还没有方法能够指定一个切入点 执行以下一个连接点匹配另一个切入点。) 一个控制流切入点匹配 当前调用堆栈。 例如,它可能火如果连接点 调用一个方法吗 com mycompany web 包,或由 SomeCaller 类。 控制流 切入点指定使用 org.springframework.aop.support.ControlFlowPointcut 类。

> **注意**
>
> 控制流的切入点是更昂贵的 在运行时评估甚至比其他动态的切入点。 在Java 1.4中, 这个费用是5倍的其他动态的切入点。

B.1.5A切入点超类

Spring提供了有用的切入点超类来帮助你 实现自己的切入点。

因为静态的切入点是最有用的,你可能会子类 StaticMethodMatcherPointcut,如下所示。 这需要实现 只是一个抽象方法(尽管它可以覆盖其他 方法自定义行为):

```
class TestStaticPointcut extends StaticMethodMatcherPointcut {

   public boolean matches(Method m, Class targetClass) {
      // return true if custom criteria match
   }
}
```

还有超类动态的切入点。

您可以使用自定义切入点的任何意见类型在Spring 1.0 RC2及以上。

B.1.6A定制切入点

因为切入点在Spring AOP是Java类,而不是 语言特性(如AspectJ)可以声明自定义 切入点,无论是静态或动态。 自定义切入点在春天可以 任意复杂的。 然而,使用AspectJ切入点表达式 语言是建议如果可能的话。

> **注意**

后来版本的春天可能提供支持”语义 切入点”作为提供江淮:例如,“所有的方法,改变 实例变量在目标对象。”

B。 2一个API在春天。建议

现在让我们看看如何Spring AOP处理建议。

B.2.1A建议生命周期

每个建议是Spring bean。 一个实例可以共享的建议在所有 建议的对象,或独特的 每个建议的对象。 这对应于 每个类 或 每个建议。

每个类的建议是最常用的。 它是适合通用 建议如事务顾问。 这些不依赖于状态的 这个代理对象或添加新的状态;他们只是行为的方法和 参数。

每个建议适合的介绍,来支持 mixin。 在这种情况下,建议增加国家的代理 对象。

它可以使用一个混合的共享和每个建议 同样的AOP代理。

在春天B.2.2A建议类型

弹簧提供几个建议类型的盒子,是 可扩展支持任意类型的建议。 让我们看看基本 概念和标准建议类型。

拦截around通知

最基本的建议类型在春天 拦截around通知 。

春天是符合AOP联盟界面左右 建议使用方法拦截。 MethodInterceptors实施 在建议应该实现以下接口:

```
public interface MethodInterceptor extends Interceptor {

    Object invoke(MethodInvocation invocation) throws Throwable;
}
```

这个 MethodInvocation 参数 invoke() 方法调用的方法暴露; 目标连接点;AOP代理;和参数的方法。 这个 invoke() 方法应该返回 调用的结果:返回值的连接点。

一个简单的 MethodInterceptor 实现 看起来如下:

```
public class DebugInterceptor implements MethodInterceptor {

    public Object invoke(MethodInvocation invocation) throws Throwable {
        System.out.println("Before: invocation=[" + invocation + "]");
        Object rval = invocation.proceed();
        System.out.println("Invocation returned");
        return rval;
    }
}
```

注意调用MethodInvocation的 proceed() 法。 这个顺序从上到下 拦截器链向连接点。 大多数拦截器将调用 这种方法,并返回它的返回值。 然而,一个 MethodInterceptor,像任何around通知,可以返回一个不同的 值或者抛出一个异常,而不是调用proceed方法。 然而,你不想这样做没有好理由!

注意

与其他AOP MethodInterceptors提供互操作性 联盟兼容的AOP实现。 其他的建议类型 讨论在本节的其余部分实现常见的AOP 的概念,但在一个spring特定方式。 虽然是有利的 使用最具体的建议类型,坚持MethodInterceptor 在你可能会建议,如果想要运行在另一个方面 AOP框架。 注意,切入点目前不互操作框架之间,AOP联盟目前尚未定义 切入点的接口。

建议之前

一个简单的建议类型是 **之前 建议** 。 这并不需要一个 MethodInvocation 对象,因为它只会 进入方法之前调用。

主要的优势的建议是,之前是没有必要的 调用 proceed() 方法,因此也没有 可能无意中未能进行了拦截 链。

这个 MethodBeforeAdvice 界面显示 下面。 (春天的API设计将允许对字段之前的建议, 虽然通常的对象适用于现场拦截和它的 不太可能会实现它,春天)。

```
public interface MethodBeforeAdvice extends BeforeAdvice {

    void before(Method m, Object[] args, Object target) throws Throwable;
}
```

注意,返回类型 无效 。 之前 建议可以插入自定义行为在连接点之前执行,但是 不能改变返回值。 如果抛出一个之前的建议 例外,这将中止进一步执行拦截器链。 异常将传播支持拦截器链。 如果它是 无节制的,或在被调用的方法的签名,这将是 直接传递给客户,否则它将被包装在一个 未经检查的异常的AOP代理。

之前的一个例子建议在春天,它统计所有的方法 调用:

```
public class CountingBeforeAdvice implements MethodBeforeAdvice {

    private int count;

    public void before(Method m, Object[] args, Object target) throws Throwable {
        ++count;
    }

    public int getCount() {
        return count;
    }
}
```

提示

之前的建议可以被用于任何切入点。

抛出建议

抛出建议 之后才调用 返回的连接点如果连接点抛出一个异常。 弹簧提供类型抛出的建议。 注意,这意味着 org.springframework.aop.ThrowsAdvice 接口并 不包含任何方法:它是一个标记接口,确认了 给定对象实现一个或多个类型的抛出的建议方法。 这些 应该是在形式的:

```
afterThrowing([Method, args, target], subclassOfThrowable)
```

只有最后一个参数是必需的。 方法签名可能 要么一个或四个参数,取决于建议吗 方法是感兴趣的方法和参数。 以下 类的实例抛出的建议。

下面的建议是如果一个调用 RemoteException 扔(包括子类):

```
public class RemoteThrowsAdvice implements ThrowsAdvice {

    public void afterThrowing(RemoteException ex) throws Throwable {
        // Do something with remote exception
    }
}
```

以下的建议是如果一个调用 ServletException 抛出。 与上面的 建议,它声明4参数,以便它可以使用调用 方法,方法参数和目标对象:

```
public class ServletThrowsAdviceWithArguments implements ThrowsAdvice {

    public void afterThrowing(Method m, Object[] args, Object target, ServletException ex) {
        // Do something with all arguments
    }
}
```

最后的示例说明了这两种方法可以 用在一个类,它可以处理 RemoteException 和 ServletException 。 任何数量的抛出的建议方法可以合并成一个类。

```
public static class CombinedThrowsAdvice implements ThrowsAdvice {

    public void afterThrowing(RemoteException ex) throws Throwable {
        // Do something with remote exception
    }

    public void afterThrowing(Method m, Object[] args, Object target, ServletException ex) {
        // Do something with all arguments
```

```
    }
}
```

注意: 如果一个抛出建议方法抛出一个异常本身, 它将覆盖原来的异常(即改变抛出的异常给用户)。 压倒一切的异常通常是一个 RuntimeException;这是兼容的 任何方法签名。 然而,如果一个抛出建议方法会抛出一个异常,检查 它将匹配目标方法的声明异常,因此一些 度耦合到特定目标方法签名。 不抛出未申报吗 检查异常,不能与目标方法的签名!

提示

抛出的建议可以被用于任何切入点。

回国后的建议

回国后在春天的建议必须实现 org.springframework.aop.AfterReturningAdvice 界面,如下所示:

```
public interface AfterReturningAdvice extends Advice {

    void afterReturning(Object returnValue, Method m, Object[] args, Object target)
            throws Throwable;
}
```

回国后的建议已经获得返回值(它不能修改),调用方法,方法参数和 目标。

以下所有成功后返回的建议项 方法调用,没有抛出的异常:

```
public class CountingAfterReturningAdvice implements AfterReturningAdvice {

    private int count;

    public void afterReturning(Object returnValue, Method m, Object[] args, Object target)
            throws Throwable {
        ++count;
    }

    public int getCount() {
        return count;
    }
}
```

这个建议不改变执行路径。 如果它抛出一个 例外,这将被扔了拦截器链代替 返回值。

提示

回国后的建议可以被用于任何切入点。

介绍的建议

春天把介绍的建议当成一种特殊的 拦截的建议。

介绍需要一个 IntroductionAdvisor , 和一个 IntroductionInterceptor ,实现 以下界面:

```
public interface IntroductionInterceptor extends MethodInterceptor {

    boolean implementsInterface(Class intf);
}
```

这个 invoke() 方法继承了AOP 联盟 MethodInterceptor 接口必须实现 简介:也就是说,如果被调用的方法是在一个介绍 接口,介绍拦截器负责处理 方法调用——它不能调用 proceed() 。

介绍的建议不能用于任何切入点,因为它 仅适用于在类中,而不是方法,水平。 你只能使用 介绍与建议 IntroductionAdvisor , 这有以下方法:

```
public interface IntroductionAdvisor extends Advisor, IntroductionInfo {

    ClassFilter getClassFilter();

    void validateInterfaces() throws IllegalArgumentException;
}

public interface IntroductionInfo {

    Class[] getInterfaces();
```

```
}
```

没有 MethodMatcher ,因此没有 切入点 ,介绍相关的建议。 只有 类过滤是合乎逻辑的。

这个 getInterfaces() 方法返回 接口引入的这个顾问。

这个 validateInterfaces() 方法是在公司内部使用, 看是否可以实现介绍了接口的配置 IntroductionInterceptor 。

让我们看一个简单的例子从弹簧测试套件。 让我们 假设我们想要介绍以下接口与一个或多个 对象:

```
public interface Lockable {
    void lock();
    void unlock();
    boolean locked();
}
```

这说明了一个 **Mixin** 。 我们 希望能够投建议可锁定的对象,无论他们的 类型和调用方法锁定和解锁。 如果我们调用lock()方法, 我们希望所有的setter方法抛出 LockedException 。 因此我们可以添加一个方面 提供了能够使对象不变的,没有他们有 任何对它的认识:一个很好的例子,AOP。

首先,我们需要一个 IntroductionInterceptor 这确实沉重的 提升。 在本例中,我们扩展了 org.springframework.aop.support.DelegatingIntroductionInterceptor 方便的类。 我们可以实现 IntroductionInterceptor 直接,但使用 DelegatingIntroductionInterceptor 最适合最 情况下。

这个 DelegatingIntroductionInterceptor 是 用来代表一个介绍到实际实现的 引入界面(s)、隐匿使用拦截来做 所以。 委托可以设置为任何对象使用一个构造函数 参数,默认的代表(当使用不带参数的构造器) 这是。 因此在下面的例子中,代表了 LockMixin 子类的 DelegatingIntroductionInterceptor 。 给定一个代表 (默认情况下本身),一个 DelegatingIntroductionInterceptor 实例看起来 对于所有接口实现的代表(除了 IntroductionInterceptor),并将支持对任何介绍 他们的。 它的子类可能如 LockMixin 调用 suppressInterface(类intf) 方法抑制 接口不应该暴露。 然而,无论多少 接口一个 IntroductionInterceptor 准备 支持, IntroductionAdvisor 使用将 控制接口实际上是暴露。 一个介绍界面 将隐瞒任何实现相同的接口的 目标。

因此LockMixin子类 DelegatingIntroductionInterceptor 并实现了 可锁定的本身。 超类自动回升,可锁定的 可以支持的介绍,所以我们不需要指定。 我们可以引入任意数量的接口在这种方式。

注意使用 锁 实例变量。 这有效地增加了额外的状态来了在目标 对象。

```
public class LockMixin extends DelegatingIntroductionInterceptor
    implements Lockable {

    private boolean locked;

    public void lock() {
        this.locked = true;
    }

    public void unlock() {
        this.locked = false;
    }

    public boolean locked() {
        return this.locked;
    }

    public Object invoke(MethodInvocation invocation) throws Throwable {
        if (locked() && invocation.getMethod().getName().indexOf("set") == 0)
            throw new LockedException();
        return super.invoke(invocation);
    }

}
```

通常并不需要覆盖 invoke() 方法: DelegatingIntroductionInterceptor 实现——调用委托方法如果方法 介绍,否则向着连接点——通常是 足够的。 在目前的情况下,我们需要添加一个检查:没有setter 方法可以调用如果在锁定模式。

需要的是简单的介绍顾问。 所有需要做的 是举行一个截然不同的 LockMixin 实例,并指定 介绍了接口的——在这种情况下,只是 可封闭的 。 一个更复杂的例子可能需要 引用介绍拦截器(这将是定义为一个 原型):在这种情况下,没有配置相关的 LockMixin ,所以我们简单地创建它使用 新 。

```
public class LockMixinAdvisor extends DefaultIntroductionAdvisor {

    public LockMixinAdvisor() {
```

```
        super(new LockMixin(), Lockable.class);
    }
}
```

我们可以把这个顾问非常简单:它不需要 配置。 (然而,这 是 必要的: 不可能使用一个 IntroductionInterceptor 没有 IntroductionAdvisor)。 像往常一样同 介绍,该顾问必须每个,因为它是有状态的。 我们 需要一个不同的实例 LockMixinAdvisor ,和 因此 LockMixin ,对于每个建议的对象。 这个 顾问由部分建议对象的状态。

我们可以把这个顾问通过编程,使用 Advised.addAdvisor() 法,或(推荐的 在XML配置,就像任何其他顾问。 所有代理创建 选择下面讨论,包括“自动代理的创造者”,正确 处理介绍和有状态的混合。

B。 3一个顾问API在春天

在春天,一个顾问是一个方面,它只包含一个建议 与对象关联一个切入点表达式。

除了特殊情况的介绍,任何顾问可以 用于任何建议。 org.springframework.aop.support.DefaultPointcutAdvisor 是最常用的顾问类。 例如,它可以使用 一个 MethodInterceptor , BeforeAdvice 或 ThrowsAdvice 。

它是可能的混合顾问和咨询类型在春天在相同的 AOP代理。 例如,您可以使用一个拦截around通知,抛出 建议和在一个代理配置之前建议:春天将 自动创建必要的拦截器链。

B。 4一个使用ProxyFactoryBean创建AOP代理

如果你使用Spring IoC容器(一个ApplicationContext或 为你的业务对象BeanFactory)——和你应该! ——你会想要 使用Spring的AOP FactoryBeans之一。 (请记住,一个工厂bean 介绍了一个间接层,使它创建的对象 不同的类型)。

> **注意**
>
> Spring 2.0 AOP支持也使用工厂bean的被窝。

基本的方法来创建一个AOP代理在春天是使用 org.springframework.aop.framework.ProxyFactoryBean 。 这给完全控制切入点和通知,将申请, 和它们的顺序。 然而,有简单的选项是可取的 如果你不需要这样的控制。

B.4.1A基础知识

这个 ProxyFactoryBean 像其他弹簧 FactoryBean 实现,介绍一个级别的 间接寻址。 如果你定义一个 ProxyFactoryBean 与名称 foo ,什么对象引用 foo 看到的并不是 ProxyFactoryBean 实例本身,而是一个对象 创造的 ProxyFactoryBean 的实施 这个 getObject() 法。 该方法将创建一个 AOP代理包装一个目标对象。

最重要的优点之一,使用 ProxyFactoryBean 或另一个奥委会知道类来创建 AOP代理,是这意味着建议和切入点也可以 国际奥委会所管理的。 这是一个强大的功能,使某些方法 这是很难实现与其他AOP框架。 例如,一个 建议参考应用程序对象本身(除了目标, 这应该可以在任何AOP框架),受益于所有的吗 可插入性提供了依赖注入。

B.4.2A JavaBean属性

同大多数 FactoryBean 实现 提供弹簧, ProxyFactoryBean 类是 本身一个JavaBean。 它的属性是用来:

- 指定目标你想代理。
- 指定是否使用CGLIB(见下文和也 SectionA 10 5 3,一个JDK -和proxiesacglib的基础)。

一些关键属性是继承 org.springframework.aop.framework.ProxyConfig (超类所有AOP代理工厂在春天)。 这些关键属性包括:

- proxyTargetClass : 真正的 如果 目标类进行代理,而非目标类的接口。 如果这个属性值设置为 真正的 ,然后CGLIB代理 将被创建(参见下文 SectionA 10 5 3,一个JDK -和proxiesacglib的基础)。
- 优化 :控制是否积极 优化应用于代理 创建通过CGLIB 。 人们不应轻率地使用此设置,除非一个完全理解 有关AOP代理处理如何优化。 这是目前只使用 为CGLIB代理;它没有效应与JDK动态代理。
- 冻 :如果一个代理配置 冻 , 然后修改配置不再允许。 这是有用的作为 一个轻微的优化和对那些情况下当你不想打电话可以操纵代理(通过 建议 接口) 在代理已经创建。 此属性的默认值是 假 ,所以变化如添加额外的建议是允许的。
- exposeProxy :决定是否或不是当前 代理应该是公开的 ThreadLocal 所以, 它可以访问的目标。 如果一个目标需要获得 代理和 exposeProxy 属性设置为 真正的 ,目标可以使用 AopContext.currentProxy() 法。

- aopProxyFactory :实施 AopProxyFactory 使用。 提供了一种 定制是否使用动态代理,CGLIB或任何其他代理 策略。 默认的实现将选择动态代理或 CGLIB适当。 应该没有需要使用这个属性; 它的目的是允许添加新的代理类型在Spring 1.1。

其他属性具体到 ProxyFactoryBean 包括:

- proxyInterfaces :字符串数组,数组的接口 的名字。 如果这不是提供,CGLIB代理的目标类 将被使用(但亦见下面吗 SectionA 10 5 3,一个JDK -和proxiesacglib的基础)。
- interceptorNames :字符串数组的 顾问 ,拦截器或其他建议 名称适用。 的顺序是重要的,第一次先来服务 依据。 也就是说,在列表中第一个拦截器 将会是第一个可以截取调用。
  这个名字是bean名称在当前的工厂,包括 从祖先工厂bean名称。 你不能提到bean 这里引用自这样做会导致 ProxyFactoryBean 忽略singleton 设置的建议。
  你可以添加一个拦截器的名字与一个星号 (*)。 这将导致所有的应用 顾问bean与名称开始的前一部分的星号 应用。 使用这个特性的一个示例中可以找到 SectionA 10 5 6,一个使用“全球化”advisorsa 。
- 单例:工厂是否应该返回一个单一的 对象,不管多久 getObject() 方法被调用。 几个 FactoryBean 的实现提供了这样一个方法。 默认值是 真正的 。 如果你想要使用有状态的建议- 例如,对于有状态的混合——使用原型建议沿 与单值的 假 。

B.4.3A JDK -和cglib建立代理

这部分作为明确的文档上 ProxyFactoryBean 选择创建一个的 要么一个JDK -和基于cglib代理为特定目标对象 (即进行代理)。

> **注意**
>
> 的行为 ProxyFactoryBean 关于 创建JDK -或基于cglib代理之间更改版本1.2。 x和 2.0的春天。 这个 ProxyFactoryBean 现在 展品相似的语义对于自动接口 正如 TransactionProxyFactoryBean 类。

如果类的目标对象,进行代理(以下简称简单 被称为目标类)不实现任何接口,然后 一个基于cglib代理将被创建。 这是最简单的场景,因为 JDK代理是接口的基础,没有接口意味着JDK代理 甚至不是可能的。 一个简单的插头在目标bean,并指定 拦截器的列表通过 interceptorNames 财产。 请注意,代理将创建基于cglib即使 proxyTargetClass 财产的 ProxyFactoryBean 已经设置为 假 。 (显然这没有道理,最好是远离bean 定义,因为它是在最好的冗余,在最坏的混乱。)

如果目标类实现一个(或多个)接口,那么类型的 代理创建依赖于配置的 ProxyFactoryBean 。

如果 proxyTargetClass 财产的 ProxyFactoryBean 已经设置为 真正的 , 然后基于cglib代理将被创建。 这是有意义的,是在 保持最少意外原则。 即使 proxyInterfaces 财产的 ProxyFactoryBean 已经被设置为一个或多个 完全限定的接口名称,这一事实 proxyTargetClass 属性设置为 真正的 将 引起 基于cglib代理在效应。

如果 proxyInterfaces 财产的 ProxyFactoryBean 已经被设置为一个或多个 完全限定的接口名称,然后代理将被创建为基础的 jdk。 创建的代理将执行所有的接口,是指定的 在 proxyInterfaces 房地产;如果目标类 发生在实现一大堆多个接口中指定比 这个 proxyInterfaces 房地产,这是所有好和 不错,但这些额外的接口将不会实现的 返回的代理。

如果 proxyInterfaces 财产的 ProxyFactoryBean 已经 不 被 集,但目标类 并实现一个(或多个) 接口,那么 ProxyFactoryBean 将自动检测 事实上,目标类并实现至少一个接口, 和一个jdk建立代理将被创建。 接口,实际上是 代理将 所有 接口的目标 类实现;实际上,这是一样的只是提供一个列表 每一个接口,目标类实现的 proxyInterfaces 财产。 然而,这是大大减少 工作,并且更不容易输入错误。

B.4.4A代理接口

让我们看一个简单的例子 ProxyFactoryBean 在行动。 这个例子包括:

- 一个 目标bean 那将是代理。 这 是“personTarget“bean定义在下面的例子。
- 一个顾问和拦截器用来提供建议。
- AOP代理bean定义指定目标对象(personTarget bean)和接口代理,以及 建议申请。

```
<bean id="personTarget" class="com.mycompany.PersonImpl">
   <property name="name"><value>Tony</value></property>
   <property name="age"><value>51</value></property>
</bean>

<bean id="myAdvisor" class="com.mycompany.MyAdvisor">
   <property name="someProperty"><value>Custom string property value</value></property>
</bean>

<bean id="debugInterceptor" class="org.springframework.aop.interceptor.DebugInterceptor">
```

```
</bean>

<bean id="person"
  class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="proxyInterfaces"><value>com.mycompany.Person</value></property>

  <property name="target">><ref local="personTarget"/></property>
  <property name="interceptorNames">
    <list>
      <value>myAdvisor</value>
      <value>debugInterceptor</value>
    </list>
  </property>
</bean>
```

注意, interceptorNames 财产需要 字符串列表:bean的名称或顾问的拦截器 目前的工厂。 顾问,拦截器,前,后返回和 建议可以使用抛出对象。 顾问的排序是 显著的。

注意

你也许想知道为什么这个列表并不持有bean 引用。 原因是,如果ProxyFactoryBean的 单例属性设置为false,它必须能够返回 独立代理实例。 如果任何顾问本身就是一个 原型,一个独立的实例需要返回,所以它的 必要的能够获得的一个实例的原型 工厂;持有一个参考不是足够的。

“人” bean定义可以使用上面的地方 人的实现,如下所示:

```
Person person = (Person) factory.getBean("person");
```

在相同的奥委会其他bean上下文可以表达一个强类型 依赖它,像一个普通的Java对象:

```
<bean id="personUser" class="com.mycompany.PersonUser">
 <property name="person"><ref local="person" /></property>
</bean>
```

这个 PersonUser 类在这个例子会 揭露一个属性的类型的人。 至于它的关注,AOP 代理可以使用透明地在地方 “真实” 的人实现。 然而,它的类将是一个动态代理类。 它 可能丢给吗 建议 接口 (下面讨论)。

它可以隐藏目标和代理之间的区别 使用一个匿名 内在bean ,如下所示。 只有 ProxyFactoryBean 定义是不同的,建议 是只包括的完整性:

```
<bean id="myAdvisor" class="com.mycompany.MyAdvisor">
 <property name="someProperty">><value>Custom string property value</value></property>
</bean>

<bean id="debugInterceptor" class="org.springframework.aop.interceptor.DebugInterceptor"/>

<bean id="person" class="org.springframework.aop.framework.ProxyFactoryBean">
 <property name="proxyInterfaces">><value>com.mycompany.Person</value></property>
 <!-- Use inner bean, not local reference to target -->
 <property name="target">
  <bean class="com.mycompany.PersonImpl">
   <property name="name">><value>Tony</value></property>
   <property name="age">><value>51</value></property>
  </bean>
 </property>
 <property name="interceptorNames">
  <list>
   <value>myAdvisor</value>
   <value>debugInterceptor</value>
  </list>
 </property>
</bean>
```

这个的优点是只有一个类型的对象 人 :如果我们希望防止有用的用户 应用程序上下文获取一个引用联合国建议的对象,或 需要避免歧义与Spring IoC 自动装配 。 也可以说是一个优势 的定义是独立的。ProxyFactoryBean 然而,有 有时能够获得联合国建议的目标吗 工厂可能实际上是一个 优势 :对于 例,在特定的测试场景。

B.4.5A代理类

如果你需要代理类,而不是一个或更多 接口?

想象一下,在我们的例子中,没有 人 接口:我们需要建议一个类称为 人 没有实现任何业务接口。 在这种情况下,您可以配置弹簧使用CGLIB代理,而 比动态代理。 只要设置 proxyTargetClass 属性为true的上面ProxyFactoryBean。 虽然最好 程序接口,而不是类的能力,建议 实现接口的类不可能是有用的在处理 遗留代码。 (一般来说,春天不是规定性的。 虽然它使它 容易申请好实

践,它避免了迫使一个特定的 方法。)

如果你想,你可以强制使用CGLIB在任何情况下,即使 你有接口。

CGLIB代理作品通过生成目标类的一个子类 在运行时。 弹簧配置这个生成子类来委派方法 调用原始目标:子类是用来实现 装饰模式,编织的建议。

CGLIB代理一般应该是透明的,用户。 然而, 有一些问题需要考虑:

- 最后 方法不能被建议,因为他们 不能被覆盖。
- 在Spring 3.2不再需要添加CGLIB到你 项目类路径。 CGLIB类已经重新包装下 org。 springframework和包括直接在spring核心JAR。 这 既为用户方便以及避免潜在的冲突吗 与其他项目依赖不同版本的 CGLIB。

有小的性能区别,CGLIB代理 动态代理。 在Spring 1.0中,动态代理是稍快。 然而,这可能会改变在未来。 性能不应该 决定性的考虑在这种情况下。

B.4.6A使用“全球”顾问

通过附加一个星号,拦截器的名字,所有的顾问与 bean名称匹配前面的部分星号,将被添加到 顾问链。 这可以派上用场,如果你需要添加一组标准 “全球”顾问:

```
<bean id="proxy" class="org.springframework.aop.framework.ProxyFactoryBean">
 <property name="target" ref="service"/>
 <property name="interceptorNames">
  <list>
   <value>global*</value>
  </list>
 </property>
</bean>

<bean id="global_debug" class="org.springframework.aop.interceptor.DebugInterceptor"/>
<bean id="global_performance" class="org.springframework.aop.interceptor.PerformanceMonitorInterceptor"/>
```

B。 5一个简洁的代理定义

特别是当定义事务代理,你可能最终得到 许多类似的代理定义。 父母和孩子的使用bean 定义,以及内心的bean定义,可以导致更加简洁 和更简洁的代理定义。

首先一个家长, 模板 ,bean的定义是 创建的代理:

```
<bean id="txProxyTemplate" abstract="true"
      class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
 <property name="transactionManager" ref="transactionManager"/>
 <property name="transactionAttributes">
  <props>
   <prop key="*">PROPAGATION_REQUIRED</prop>
  </props>
 </property>
</bean>
```

这将永远不会被实例化本身,所以实际上可能 不完整的。 然后每个代理需要创建bean只是个孩子 定义,它将目标的代理作为一个内部bean 定义,因为目标永远不会被用在它自己的 不管怎样。

```
<bean id="myService" parent="txProxyTemplate">
 <property name="target">
  <bean class="org.springframework.samples.MyServiceImpl">
  </bean>
 </property>
</bean>
```

当然可能覆盖属性从父 模板,如在这种情况下,事务传播 设置:

```
<bean id="mySpecialService" parent="txProxyTemplate">
 <property name="target">
  <bean class="org.springframework.samples.MySpecialServiceImpl">
  </bean>
 </property>
 <property name="transactionAttributes">
  <props>
   <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>
   <prop key="find*">PROPAGATION_REQUIRED,readOnly</prop>
   <prop key="load*">PROPAGATION_REQUIRED,readOnly</prop>
   <prop key="store*">PROPAGATION_REQUIRED</prop>
```

```
    </props>
  </property>
</bean>
```

注意,在上面的例子中,我们有明确标志着父母 bean定义为 文摘 通过使用 文摘 属性,描述 以前 ,所以它可能 实际上不是曾经被实例化。 应用程序上下文(但不是简单的 bean工厂)将默认情况下预实例化所有单件。 因此 重要的(至少对单例bean),如果你有一个(父) bean定义你想只使用作为一个模板,这 定义指定了一个类,您必须确保设置 文摘 属性 真正的 , 否则,应用程序上下文实际上会尝试预实例化 它。

B。 6一个AOP代理以编程方式创建与ProxyFactory

它很容易使用Spring AOP代理以编程方式创建。 这 使您能够使用Spring AOP没有依赖Spring IoC。

以下清单显示了创建一个代理为目标对象, 与一个拦截器和一个顾问。 接口实现的 目标对象将自动代理:

```
ProxyFactory factory = new ProxyFactory(myBusinessInterfaceImpl);
factory.addInterceptor(myMethodInterceptor);
factory.addAdvisor(myAdvisor);
MyBusinessInterface tb = (MyBusinessInterface) factory.getProxy();
```

第一步是构建一个对象的类型 org.springframework.aop.framework.ProxyFactory 。 你可以 创建这个与目标对象,在上面的示例中,或指定 接口是在另一个构造函数,代理

您可以添加拦截器或顾问,和操纵他们 ProxyFactory生活。 如果你添加一个 IntroductionInterceptionAroundAdvisor可以导致代理来实现 额外的接口。

也有方便的方法在ProxyFactory(被继承 AdvisedSupport),这允许您添加其他建议类型 如之前和抛出的建议。 AdvisedSupport是超类的两个 ProxyFactory和ProxyFactoryBean。

提示

AOP代理创建集成与IoC框架是最好的 实践在大多数应用程序。 我们建议你具体化 配置从Java代码使用AOP,如一般。

B。 7一个操纵建议对象

然而你创建AOP代理,你可以操纵他们使用 org.springframework.aop.framework.Advised 接口。 任何AOP代理可以转换为这个接口,接口的任何其他 实现了。 这个接口包括以下方法:

```
Advisor[] getAdvisors();

void addAdvice(Advice advice) throws AopConfigException;

void addAdvice(int pos, Advice advice)
      throws AopConfigException;

void addAdvisor(Advisor advisor) throws AopConfigException;

void addAdvisor(int pos, Advisor advisor) throws AopConfigException;

int indexOf(Advisor advisor);

boolean removeAdvisor(Advisor advisor) throws AopConfigException;

void removeAdvisor(int index) throws AopConfigException;

boolean replaceAdvisor(Advisor a, Advisor b) throws AopConfigException;

boolean isFrozen();
```

这个 getAdvisors() 方法将返回一个顾问 对于每一个顾问,拦截器或其他建议,已添加到类型 工厂。 如果你添加了一个顾问,顾问在这个索引返回 将您添加的对象。 如果你添加了一个拦截器或其他 建议类型,弹簧将包裹这在一个顾问与切入点 总是返回true。 因此如果你添加了一个 MethodInterceptor ,返回这个索引advisor 将是一个 DefaultPointcutAdvisor 返回你的MethodInterceptor 和一个切入点匹配所有 类和方法。

这个 addAdvisor() 方法可以用来添加任何 顾问。 通常顾问控股切入点和通知将是 通用 DefaultPointcutAdvisor ,可以使用任何建议或切入点(但不是介绍)。

默认情况下,它可以添加或删除顾问或拦截器 即使已经创建一个代理。 唯一的限制是,它是 无法添加或删除一个介绍顾问,因为

现有的代理 从工厂将不会显示界面变化。 (你可以获得一个新的 代理从工厂来避免这个问题。)

一个简单的例子,铸造一个AOP代理 建议 接口和检查和处理 建议:

```
Advised advised = (Advised) myObject;
Advisor[] advisors = advised.getAdvisors();
int oldAdvisorCount = advisors.length;
System.out.println(oldAdvisorCount + " advisors");

// Add an advice like an interceptor without a pointcut
// Will match all proxied methods
// Can use for interceptors, before, after returning or throws advice
advised.addAdvice(new DebugInterceptor());

// Add selective advice using a pointcut
advised.addAdvisor(new DefaultPointcutAdvisor(mySpecialPointcut, myAdvice));

assertEquals("Added two advisors",
    oldAdvisorCount + 2, advised.getAdvisors().length);
```

注意

这是怀疑它是可取的(没有双关) 修改一个业务对象的建议在生产,虽然没有 怀疑合法使用案例。 然而,它可以是非常有用的 发展:例如,在测试。 我有时会发现它非常有用 能够添加测试代码的形式,一个拦截器或其他建议, 进入一个方法调用我想测试。 (例如, 建议可以在一个事务创建该方法:例如, 运行SQL检查数据库正确更新,在标记 事务的回滚)。

这取决于你如何创建代理,您通常可以设置一个 冻 旗,在这种情况下 建议 isFrozen() 方法将 返回true,任何试图修改建议通过添加或删除 将导致一个 AopConfigException 。 能够 冻结的状态建议对象有用的在某些情况下,对于 例,以防止调用代码删除安全拦截器。 它可能 也可用于Spring 1.1允许积极优化如果运行时 建议修改是不需要知道。

B。 8一个使用“火狐的一个插件”设施

到目前为止,我们已经被认为是显式创建的AOP代理使用 ProxyFactoryBean 或类似的工厂bean。

春天还允许我们使用“火狐的一个插件” bean定义,它可以 自动代理选定的bean定义。 这是建立在春天 “豆后置处理程序”基础设施,允许修改任何 作为容器加载bean定义。

在这个模型中,您将设置一些特殊bean定义XML bean定义文件来配置自动代理的基础设施。 这 允许您声明的目标自动代理资格:你 不需要使用 ProxyFactoryBean 。

有两种方法可以做到这一点:

- 使用一个火狐的一个插件的创造者,是指特定的豆子在 当前上下文。
- 火狐的一个插件创建的一个特例,值得 单独考虑,火狐的一个插件创建源代码级别的驱动 元数据属性。

B.8.1A火狐的一个插件bean定义

这个 org.springframework.aop.framework.autoproxy 包提供了以下标准火狐的一个插件的创造者。

BeanNameAutoProxyCreator

这个 BeanNameAutoProxyCreator 类是一个 BeanPostProcessor 自动创建AOP代理 对于bean与名称匹配的文字值或通配符。

```
<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
  <property name="beanNames"><value>jdk*,onlyJdk</value></property>
  <property name="interceptorNames">
    <list>
      <value>myInterceptor</value>
    </list>
  </property>
</bean>
```

与 ProxyFactoryBean ,有一个 interceptorNames 财产而不是一个列表的拦截器,允许 正确的行为为原型顾问。 命名为“拦截器”可以 顾问或任何类型的建议。

与汽车代理一般来说,主要的点使用 BeanNameAutoProxyCreator 是应用相同的吗 一直到多个对象的配置,以最小的 卷的配置。 这是一个受欢迎的选择应用 声明性事务到多个对象。

Bean定义的名字匹配,如“jdkMyBean”和 “onlyJdk”在上面的例子中,是与普通的旧bean定义 目标类。 AOP代理将被自动创建 BeanNameAutoProxyCreator 。 同样的建议 应用到所有匹配的豆子。 注意,如果使用顾问(相当 比拦截器在上面的例子中),可以申请的切入点 不同不同的豆子。

DefaultAdvisorAutoProxyCreator

一个更一般的和非常强大的汽车代理创造者 DefaultAdvisorAutoProxyCreator 。 这将 自动应用合格的顾问在当前背景下,没有 需要包括特定bean名称在火狐的一个插件顾问的 bean定义。 它提供了相同的价值一致的配置 和避免重复作为 BeanNameAutoProxyCreator 。

使用此机制包括:

- 指定 DefaultAdvisorAutoProxyCreator bean 定义。
- 指定任意数量的顾问在同一或相关 上下文。 注意,这些 必须 是顾问, 不只是拦截器或其他建议。 这是必要的,因为 必须有一个切入点来评估、检查合格证的 每个建议候选人bean定义。

这个 DefaultAdvisorAutoProxyCreator 将 自动评价包含在每个顾问的切入点,看看 什么(如果有的话)的建议应该适用于每个业务对象(如 “businessObject1”和“businessObject2”的例子)。

这意味着任何数量的顾问可以应用 自动为每个业务对象。 如果没有切入点的 顾问的任何方法相匹配的业务对象,该对象将不会是代理。 作为bean定义添加新的业务对象, 他们将会被自动代理如果必要的。

一般自动代理的优势使它 不可能获得一个调用者或依赖项联合国建议的对象。 调用getBean(“businessObject1”)在这个ApplicationContext将 返回一个AOP代理,而不是目标业务对象。 (“内在豆” 成语前面显示还提供这种好处。)

```
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>

<bean class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
  <property name="transactionInterceptor" ref="transactionInterceptor"/>
</bean>

<bean id="customAdvisor" class="com.mycompany.MyAdvisor"/>

<bean id="businessObject1" class="com.mycompany.BusinessObject1">
  <!-- Properties omitted -->
</bean>

<bean id="businessObject2" class="com.mycompany.BusinessObject2"/>
```

这个 DefaultAdvisorAutoProxyCreator 非常 有用的,如果你想应用相同的建议始终对许多 业务对象。 一旦基础设施定义就位, 你可以简单的添加新的业务对象不包括特定的 代理的配置。 你也可以减少在额外的方面非常 很容易——例如,跟踪或性能监控方面, 最小的变化来配置。

DefaultAdvisorAutoProxyCreator提供支持的过滤 (使用一个命名约定,以便只有特定的顾问 评估,允许使用多个不同的配置, AdvisorAutoProxyCreators在同一个工厂)和排序。 顾问 可以实现 org.springframework.core.Ordered 接口,以确保正确的订购,如果这是一个问题。 这个 TransactionAttributeSourceAdvisor用在上面的例子中有一个 可配置的秩序价值;默认设置是无序的。

AbstractAdvisorAutoProxyCreator

这是DefaultAdvisorAutoProxyCreator的超类。 你 可以创建自己的火狐的一个插件创建者通过子类化这类,在吗 这是不大可能发生的顾问定义提供不足 定制的行为的框架 DefaultAdvisorAutoProxyCreator 。

B.8.2A使用元数据驱动的汽车代理

一个特别重要的类型的自动代理是由 元数据。 这就形成了一个类似的编程模型来。net ServicedComponents 。 而不是使用XML部署 描述符如EJB,配置事务管理和 其他企业服务是关押在源代码级别的属性。

在本例中,您使用 DefaultAdvisorAutoProxyCreator ,结合 顾问,理解元数据属性。 元数据的细节 在切入点中举行的部分候选人顾问,而不是 火狐的一个插件创建类本身。

这是真正的一个特例 DefaultAdvisorAutoProxyCreator ,但是值得 考虑自己的。 (元数据知道代码是在切入点 中包含的顾问,而不是AOP框架本身。)

这个 /属性 JPetStore目录的 示例应用程序展示了使用属性驱动自动代理。 在 这种情况下,不需要使用

TransactionProxyFactoryBean 。 简单地定义 事务属性在业务对象是足够的,因为 使用元数据清楚的切入点。 该bean定义包括 下面的代码,在 / - inf / declarativeServices.xml 。 注意,这是通用的,可以用JPetStore外:

```
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>

<bean class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
 <property name="transactionInterceptor" ref="transactionInterceptor"/>
</bean>

<bean id="transactionInterceptor"
   class="org.springframework.transaction.interceptor.TransactionInterceptor">
 <property name="transactionManager" ref="transactionManager"/>
 <property name="transactionAttributeSource">
   <bean class="org.springframework.transaction.interceptor.AttributesTransactionAttributeSource">
     <property name="attributes" ref="attributes"/>
   </bean>
 </property>
</bean>

<bean id="attributes" class="org.springframework.metadata.commons.CommonsAttributes"/>
```

这个 DefaultAdvisorAutoProxyCreator bean 定义(这个名字并不重要,因此它甚至可以省略) 将接所有合格的切入点在当前应用程序上下文。 在这种情况下,“transactionAdvisor“bean定义,类型 TransactionAttributeSourceAdvisor ,将适用于 类或方法携带事务属性。 这个 TransactionAttributeSourceAdvisor取决于TransactionInterceptor, 通过构造函数依赖。 示例解决这个通过自动装配。 这个 AttributesTransactionAttributeSource 取决于 的实现 org.springframework.metadata.Attributes 接口。 在 这个片段,“属性” bean,用满足这个雅加达 共享属性的API来获取属性信息。 (应用程序 代码必须被编译使用共享属性编译 任务。)

这个 /注释 JPetStore目录的 示例应用程序包含了一个类似的例子,汽车的代理 由JDK 1.5 +注释。 下面的配置使 Spring的自动检测 事务性 注释,导致隐性代理bean包含这个 注释:

```
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>

<bean class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
 <property name="transactionInterceptor" ref="transactionInterceptor"/>
</bean>

<bean id="transactionInterceptor"
   class="org.springframework.transaction.interceptor.TransactionInterceptor">
 <property name="transactionManager" ref="transactionManager"/>
 <property name="transactionAttributeSource">
   <bean class="org.springframework.transaction.annotation.AnnotationTransactionAttributeSource"/>
 </property>
</bean>
```

这个 TransactionInterceptor 这里定义取决于 在一个 PlatformTransactionManager 定义,它是 不包括在这个通用的文件(尽管它可以),因为它将 特定于应用程序的事务需求(通常是 JTA,因为在这个例子中,或Hibernate,JDO或者JDBC):

```
<bean id="transactionManager"
   class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

提示

如果你只需要声明式事务管理、使用 这些通用的XML定义会导致弹簧自动 代理所有的类或方法与事务属性。 你不会 需要直接处理AOP,和编程模型是相似的 那的。 净ServicedComponents。

这种机制是可扩展的。 它可以做到自动代理 基于自定义属性。 你需要:

- 定义您的自定义属性。
- 指定一个顾问与必要的建议,包括 触发的切入点,存在的自定义属性 在一个类或方法。 你可以使用现有的建议, 只是实现一个静态的切入点,拿起自定义 属性。

有可能这样的顾问是不同的每个建议类 (例如,混合):他们只是需要被定义为原型, 而非单例,bean定义。 例如, LockMixin 介绍从春天拦截 测试套件,如上图所示,可以结合使用一个 属性驱动的切入点来目标mixin,如下所示。 我们使用 通用 DefaultPointcutAdvisor 、配置使用 JavaBean属性:

```
<bean id="lockMixin" class="org.springframework.aop.LockMixin"
   scope="prototype"/>

<bean id="lockableAdvisor" class="org.springframework.aop.support.DefaultPointcutAdvisor"
   scope="prototype">
 <property name="pointcut" ref="myAttributeAwarePointcut"/>
```

```
  <property name="advice" ref="lockMixin"/>
</bean>

<bean id="anyBean" class="anyclass" ...
```

如果属性清楚切入点匹配任何方法 anyBean 或其他bean定义,mixin将 应用。 请注意,这两个 LockMixin 和 lockableAdvisor 定义原型。 这个 myAttributeAwarePointcut 切入点可以是一个单例 定义,因为它不保存状态为个人建议 对象。

B。 9一个使用TargetSources

弹簧提供的概念 TargetSource , 表示在 org.springframework.aop.TargetSource 接口。 这个接口负责返回“目标对象” 实现连接点。 这个 TargetSource 实现要求目标实例每次AOP代理 处理一个方法调用。

开发人员使用Spring AOP通常不需要直接 与TargetSources,但是这提供了一个强有力的工具支持 池、热交换和其他复杂的目标。 例如,一个 池TargetSource可以返回一个不同的目标实例为每个 调用,使用池来管理实例。

如果你不指定一个TargetSource,一个默认实现 使用一个本地对象的。 相同的目标是为每一个返回 调用(如你所愿)。

让我们看一下标准目标来源提供弹簧,和 如何使用它们。

> **提示**
>
> 当使用一个自定义的目标源的,你的目标通常会需要 是一个原型,而不是一个单例bean定义。 这允许 春天来创建一个新的目标实例需要时。

B.9.1A热可切换目标来源

这个 org.springframework.aop.target.HotSwappableTargetSource 的存在是为了让目标的AOP代理要切换,同时允许 调用者保持他们对它的引用。

改变目标源的目标立即生效。 这个 HotSwappableTargetSource 是线程安全的。

你可以改变目标通过 交换() 方法 在HotSwappableTargetSource如下:

```
HotSwappableTargetSource swapper =
   (HotSwappableTargetSource) beanFactory.getBean("swapper");
Object oldTarget = swapper.swap(newTarget);
```

所需的XML定义如下所示:

```
<bean id="initialTarget" class="mycompany.OldTarget"/>

<bean id="swapper" class="org.springframework.aop.target.HotSwappableTargetSource">
  <constructor-arg ref="initialTarget"/>
</bean>

<bean id="swappable" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="targetSource" ref="swapper"/>
</bean>
```

上面的 交换() 调用目标的变化 可切换的bean。 客户持有一个引用bean将 不知道这种改变,但是会立即开始触及新 目标。

尽管这个例子并没有添加任何建议,它不是 需要添加的建议使用 TargetSource —— 课程任何 TargetSource 可以一起使用吗具有任意的建议。

B.9.2A池目标源

使用池目标源提供了一个类似的编程模型 与无状态会话ejb,池相同的实例 的维护,将免费对象方法调用的 池。

一个关键的区别SLSB春池,池 那个春天池可以应用到任何一个POJO。 与春天在 一般,这个服务可以被应用于一种非侵入性的方法。

Spring提供了开箱即用的支持Jakarta Commons池 1.3,它提供了一个相当有效的池实现。 你会 需要在您的应用程序共享池Jar的类路径中使用这个 特性。 它也可以子类 org.springframework.aop.target.AbstractPoolingTargetSource 支持任何其他池API。

示例配置如下所示:

```
<bean id="businessObjectTarget" class="com.mycompany.MyBusinessObject"
   scope="prototype">
 ... properties omitted
</bean>

<bean id="poolTargetSource" class="org.springframework.aop.target.CommonsPoolTargetSource">
  <property name="targetBeanName" value="businessObjectTarget"/>
  <property name="maxSize" value="25"/>
</bean>

<bean id="businessObject" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="targetSource" ref="poolTargetSource"/>
  <property name="interceptorNames" value="myInterceptor"/>
</bean>
```

请注意,目标对象——“businessObjectTarget” 例子—— 必须 是一个原型。 这允许 PoolingTargetSource 实现来创建新的 目标的实例来种植池是必要的。 看到javadoc 对于 AbstractPoolingTargetSource 和具体的 您希望使用子类的属性信息:“最大容量” 是最基本的,总是会出现。

在这种情况下,“myInterceptor” 是一个拦截器,该拦截器的名称 需要定义在相同的奥委会上下文。 然而,它不是 有必要指定拦截器使用池。 如果你只想要 池,没有其他建议,不要设置interceptorNames产权 所有。

可以配置弹簧以便能够施放 池对象的 org.springframework.aop.target.PoolingConfig 接口,它公开了信息和当前的配置 池的大小通过介绍。 您需要定义一个 顾问:

```
<bean id="poolConfigAdvisor" class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
  <property name="targetObject" ref="poolTargetSource"/>
  <property name="targetMethod" value="getPoolingConfigMixin"/>
</bean>
```

这个顾问是通过调用一个方便的方法获得的 AbstractPoolingTargetSource 类,因此使用 MethodInvokingFactoryBean。 这个顾问的名字(“poolConfigAdvisor” 这里)必须列表中的拦截器的名字在ProxyFactoryBean 暴露的合并对象。

演员将如下所示:

```
PoolingConfig conf = (PoolingConfig) beanFactory.getBean("businessObject");
System.out.println("Max pool size is " + conf.getMaxSize());
```

注意

池无状态的服务对象通常没有必要。 我们 不相信它应该是默认的选择,因为大多数无状态对象 自然是线程安全的,和实例池是有问题的 资源被缓存。

简单的连接池是可以使用自动代理。 这是可能的 设置TargetSources使用任何火狐的一个插件的创造者。

B.9.3A原型目标源

建立一个“原型” 目标源类似于一个池 TargetSource。 在这种情况下,一个新实例将创建的目标 在每个方法调用。 虽然在创建一个新对象的成本 不高在现代JVM,连接的成本的新对象 (满足其奥委会依赖性)可能更昂贵。 因此你 不应该用这种方法没有很好的理由。

要做到这一点,您可以修改 poolTargetSource 上面所示的定义如下。 (我也改变了名字,为了清晰。)

```
<bean id="prototypeTargetSource" class="org.springframework.aop.target.PrototypeTargetSource">
  <property name="targetBeanName" ref="businessObjectTarget"/>
</bean>
```

只有一个属性:目标bean的名称。 继承是用于TargetSource实现确保 一致的命名。 与池目标源、目标bean 必须是一个原型的bean定义。

B.9.4A ThreadLocal 目标来源

ThreadLocal 目标是有用的,如果你需要来源一个对象 创建为每个传入请求(每个线程)。 的概念 ThreadLocal 提供一个jdk宽设施 透明地存储资源与一个线程。 设置一个 ThreadLocalTargetSource 几乎一样的解释 其他类型的目标来源:

```
<bean id="threadlocalTargetSource" class="org.springframework.aop.target.ThreadLocalTargetSource">
  <property name="targetBeanName" value="businessObjectTarget"/>
</bean>
```

注意

有严重的问题(比如ThreadLocals)引用(可能 导致内存泄漏)当不正确的使用 多线程和多类加载器环境。 一个人应该总是 考虑在其他包装threadlocal类和从未直接 使用 ThreadLocal 本身(除了当然在包装类)。 同样,一个人应该永远记住要正确设置和复原(后者只需要调用 threadlocal集(空))资源 当地的线程。 复位应该做在任何情况下因为并不是 取消它可能导致问题行为。 Spring的ThreadLocal 这是否对你的支持,应该被认为是在忙 使用(比如ThreadLocals)引用没有其他合适的处理 代码。

B。 10一个定义新的 建议 类型

Spring AOP设计是可扩展的。 而拦截 实现策略是目前在内部使用,它是可能的 支持任意的建议类型除了开箱即用的拦截 around通知, 以前,抛出建议和回国后的建议。

这个 org.springframework.aop.framework.adapter 包是一个SPI包允许支持新的自定义类型的建议 被添加在不改变核心框架。 唯一约束在一个 定制 建议 类型是,它必须实现 org.aopalliance.aop.Advice 标记接口。

请参考 org.springframework.aop.framework.adapter 包的 Javadocs为进一步的信息。

B。 11一个进一步的资源

请参阅示例应用程序的Spring为进一步的例子 Spring AOP的:

- 的默认配置的JPetStore演示了使用 TransactionProxyFactoryBean 对于声明式事务 管理。
- 这个 /属性 JPetStore目录的 演示了使用声明式事务管理的属性驱动。

AppendixA C。 一个迁移到Spring Framework 3.1

本附录中我们讨论你的用户需要什么,知道什么时候升级到 Spring Framework 3.1。 对于一个功能概述,请参阅 ChapterA 3, 新特性和增强功能在Spring框架3.1

C。 1一个组件扫描对“org”基础包

Spring Framework 3.1引入了一个数量的 @ configuration 类比如 org.springframework.cache.annotation.ProxyCachingConfiguration 和 org.springframework.scheduling.annotation.ProxyAsyncConfiguration 。 因为 @ configuration 最终是元注释与弹簧的吗 component 注释,这些类将无意中被扫描 和处理的容器为任何组件扫描指令反对 不合格的“org”方案,例如:

```
<context:component-scan base-package="org"/>
```

因此,为了避免错误的报道 spr - 9843 , 任何此类指示应该更新至少一个资质等级的假设。

```
<context:component-scan base-package="org.xyz"/>
```

另外,一个 排他过滤器 可以使用。 看到 背景:组件扫描 文档获取详细信息。

AppendixA D。 一个迁移到Spring Framework 3.2

本附录中我们讨论你的用户需要什么,知道什么时候升级到 Spring Framework 3.2。 对于一个功能概述,请参阅 ChapterA 4, 新特性和增强功能在Spring框架3.2

d。 1新选的依赖性

某些国际米兰模块依赖关系现在 可选 在 Maven POM水平上,他们曾经要求。 例如, 弹簧tx 和它的依赖 spring上下文 。 这可能导致 ClassNotFoundErrors 或其他类似的问题,用户依赖 在传递依赖管理拉在下游的影响 弹簧- * 。 要解决这个问题,只需添加适当的缺失的jar到你的 构建配置。

d。 2一个EHCache支持搬到spring上下文的支持

随着春天的新JCache支持,EHCache支持类的 org.springframework.cache.ehcache 包离开 spring上下文 模块 spring上下文支持 。

d。 3一个内联的弹簧asm jar

在版本3.0和3.1,我们发表一个离散 弹簧asm 包含包装 org.objectweb.asm 3。 x来源。 像春天的 框架3.2,我们已经升级到 org.objectweb.asm 4.0和做 了独立的模块jar,支持内联直接在这些类 spring核心 。 这应该引起没有迁移问题对于大多数用户; 但只是碰碰运气,你有 弹簧asm 宣布直接 在你的项目的构建脚本,您需要删除它,当升级到 Spring Framework 3.2。

d。 4一个显式CGLIB不再需要依赖

在以前的版本中,用户Spring的AOP代理(例如基于子类通过 代理目标类= " true ")和 @ configuration 类支持需要声明一个显式的依赖CGLIB 2.2。 截至 Spring Framework 3.2,现在我们重新包装和内联最新CGLIB 3.0。

这意味着更大的方便用户,以及正确的功能 Java 7的用户创建子类的类型,包含代理 invokedynamic 字节码指令。 分装CGLIB内部保证 没有类路径冲突与其他第三方框架,可能依赖于其他 版本的CGLIB。

d。 5一个为OSGi用户

OSGi元数据不再是发表在个人Spring框架 jar清单文件。 MF文件。 请看到这 公告 为更多的信息关于用户可以得到osgi准备版本的春天 框架3.2罐。

d。 6一个MVC Java配置和MVC的名称空间

解释 SectionA 17 15 4,一个配置内容Negotiationa ,两个 MVC Java配置和MVC名称空间等注册扩展 json 和 . xml 如果 相应的类路径依赖关系存在。 这意味着控制器 方法现在可能返回JSON或XML格式的内容如果这些 扩展是出现在请求URI, 即使“接受”头不请求那些媒体类型。

新添加的支持矩阵变量加以解释 一个章节Variablesa矩阵 。 为了保持向后 兼容性,默认情况下,分号内容被删除从传入的 请求uri,因此 @MatrixVariable 不能用没有额外的配置。 然而,当使用 MVC Java配置和MVC的名称空间,分号内容是离开 在URI这样矩阵变量自动支持。 除去分号内容是控制通过 UrlPathHelper 财产的 RequestMappingHandlerMapping 。

d。 7一个解码的URI变量值

URI变量值现在得到解码当 UrlPathHelper.setUrlDecode 设置为 假 。 看到 spr - 9098 。

d。 8一个HTTP补丁方法

这个 DispatcherServlet 现在允许 HTTP方法,在以前它补丁没有。

d。 9一个瓷砖3

除了版本号变化,组瓷砖 依赖关系也发生了变化。 你需要有一个子集或全部 瓦片请求api , 瓷砖api , 瓦片核心 , 瓷砖servlet , 瓷砖jsp , 瓷砖el 。

d。 10一个Spring MVC测试独立项目

如果迁移从 弹簧测试mvc 独立项目 弹簧测试 模块 Spring Framework 3.2,您将需要调整根包 是 org.springframework.test.web.servlet 。

您将不再能够使用 MockMvcBuilders annotationConfigSetup 和 xmlConfigSetup 选项。 相反你会需要开关 使用 @WebAppConfiguration 支持 的 弹簧测试 加载Spring配置, 然后注入 WebApplicationContext 进 测试和使用它来创建一个 MockMvc 。 看到 SectionA 11 3 6,一个测试FrameworkaSpring MVC 详情。

d。 11一个弹簧测试依赖关系

这个 弹簧测试 模块已经升级到 依赖于JUnit 4.11(junit:junit),TestNG 6 5 2 (org testng:testng),和Hamcrest核心1.3 (org.hamcrest:hamcrest-core)。 每一种 依赖性是声明为一个 可选 依赖在 Maven POM。 此外,重要的是要注意,JUnit团队已经停止内联Hamcrest核心在吗 junit:junit Maven工件作为JUnit 4.11。 Hamcrest 核心现在是一个 需要 传递依赖的 JUnit ,用户可能因此需要删除任何 排除在 hamcrest-core 他们曾 配置为他们构建。

d。 12个一个公共API的变化

D.12.1A JDiff报告

选择JDiff报告现在被发布为用户提供一个方便的 意味着理解的版本之间的变化。 这些将是未来 每个小版本之间发表,例如,从3.1.3。 3.1.4版本。 释放; 最新的维护版本到最新的通用版本,例如。 3.1.3。 版本3 2 0释放 ,在每个里程碑之间 和/或RC为用户跟踪下一代的发展,例如。 3 2 0。 RC2,3 2 0释放 。

D.12.2A不

下面的包和类型已经全部或部分弃用 在Spring框架3.2和可能被删除在未来版本。 点击 到连接Javadoc对于每一项的具体细节。 也看见了 完整列表,不 在这个框架。

- org.springframework.orm.ibatis
- org.springframework.scheduling.backportconcurrent
- org.springframework.ejb.support
- org.springframework.http.converter.xml.XmlAwareFormHttpMessageConverter
- org.springframework.web.jsf.DelegatingVariableResolver
- org.springframework.web.jsf.SpringBeanVariableResolver
- org.springframework.ui.velocity.CommonsLoggingLogSystem
- org.springframework.ui.velocity.VelocityEngineUtils
- org.springframework.beans.factory.config.BeanReferenceFactoryBean
- org.springframework.beans.factory.config.CommonsLogFactoryBean
- org.springframework.beans.instrument.classloading.oc4j.OC4JLoadTimeWeaver
- org.springframework.transaction.jta.OC4JJtaTransactionManager
- org.springframework.web.util.ExpressionEvaluationUtils
- org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter
- org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerExceptionResolver
- org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping
- org.springframework.web.servlet.mvc.annotation.ServletAnnotationMappingUtils
- org.springframework.jmx.support.MBeanRegistrationSupport
- org.springframework.test.context.ContextConfigurationAttributes
- org.springframework.test.context.junit4.AbstractTransactionalJUnit4SpringContextTests : 使用 simpleJdbcTemplate 实例变量被弃用 支持新 JdbcTemplate 实例变量。
- org.springframework.test.context.testng.AbstractTransactionalTestNGSpringContextTests : 使用 simpleJdbcTemplate 实例变量被弃用 支持新 JdbcTemplate 实例变量。
- org.springframework.test.jdbc.SimpleJdbcTestUtils 已经弃用在 支持 JdbcTestUtils 而现在包含所有的吗 功能之前可在 SimpleJdbcTestUtils 。
- org.springframework.web.servlet.view.ContentNegotiatingViewResolver
- org.springframework.transaction.interceptor.TransactionAspectUtils
- org.springframework.http.HttpStatus
- org.springframework.web.util.UriUtils
- org.springframework.orm.jpa.vendor.TopLinkJpaDialect
- org.springframework.orm.jpa.vendor.TopLinkJpaVendorAdapter
- org.springframework.orm.util.CachingMapDecorator

AppendixA E。 一个XML的基于配置

E.1A介绍

这个附录详细介绍了基于XML的配置Spring 2.0和增强和扩展在Spring 2.5和3.0。

中央动机迁移到XML模式基于配置文件是 使Spring XML配置更加简单。 这个 “经典” 的 < bean / > 的方法是好的,但其一般性质来 有价格方面的配置开销。

DTD支持吗?

编写Spring配置文件使用老的DTD样式 仍然是完全支持。

从Spring IoC容器的观点来看, 一切 是一个bean。 这是个好消息对Spring IoC容器,因为如果一切 一个bean然后一切可以治疗在完全相同的时尚。 同样的,然而, 不是真正的从开发人员的角度看。 定义的对象在一个春天 XML配置文件不是所有通用、香草豆。 通常,每个bean需要 某种程度的具体配置。

没有什么会打破如果你放弃使用新的XML的基于 方法来编写Spring XML配置文件。 所有你失去 的是机会有更多的简洁和清晰的配置。 不论XML配置是DTD或基于架构,最后 这一切都归结到同一个对象模型在容器(即一个或 更多的BeanDefinition 实例)。

Spring 2.0的新XML的基于配置解决了这个问题。 这个 < bean / > 元素仍然存在,如果你 想,你可以继续写 完全相同的 风格的Spring XML配置只使用 < bean / > 元素。 新的XML的基于配置做,然而,使 Spring XML配置文件读。大大更清晰 此外,它允许 你表达的意图bean定义。

要记住的关键一点是,新的自定义标签的工作最好的基础设施 或集成豆类:例如,AOP,集合、事务、集成 3让框架如骡子,等等,而现有的bean标签是最适合的 特定于应用程序的豆类,如DAOs,服务层对象,验证器,等等。

下面的例子包括将有希望说服你,包容 XML Schema支持Spring 2.0中是一个好主意。 在社区的接待 一直在鼓励,同时,请注意这个事实,这个新配置机制 是完全可定制和可扩展的。 这意味着您可以编写您自己的特定于域的 配置标签能够更好代表应用程序的域;过程 参与这样做是覆盖在附录资格 AppendixA F, 可扩展的XML编辑 。

E。 2基于XML配置

E.2.1A引用模式

要切换从dtd样式到新的XML schema的风格,你需要 做出以下改变。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
   "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>

<!-- bean definitions here -->

</beans>
```

相当于在XML schema风格文件会.....

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

<!-- bean definitions here -->

</beans>
```

注意

这个 “xsi:schemaLocation” 片段实际上并不是必需的, 但可以包含引用一个本地副本的一个模式(可以是有用的 在开发期间)。

上面的Spring XML配置片段是样板,您可以复制和粘贴 (!),然后塞 < bean / > 定义为喜欢你总是 做的。 然而,整个点切换是 利用新的Spring 2.0的XML标记,因为它们的产品配置更加简单。 这个 节 SectionA e 2 2,的 util schemaa 演示了如何 立即开始使用一些更常见的实用工具标签。

剩下的这一章是致力于展示示例XML模式的新的春天 基于配置,至少一个例子为每一个新的标签。 格式如下 一个之前和之后的风格, 之前 XML片段显示 旧的(但仍有100%的法律和支持)的风格,然后立即 通过一个 在 示例显示了在新的XML的基于等效 风格。

E.2.2A了 util 模式

首先是报道的 util 标签。 正如它的名字 意味着, util 标签处理常见, 效用 配置问题,比如配置集合,引用常量, 和诸如此类的。

使用标记 util 模式,你需要 下面的序文顶部的Spring XML配置文件; 下面的代码片段中的文本引用正确的模式,这样 标签在 util 名称空间是可用的。

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:util="http://www.springframework.org/schema/util"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util.xsd">

<!-- bean definitions here -->

</beans>
```

< util:常数/ >

之前……

```
<bean id="..." class="...">
  <property name="isolation">
    <bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE"
    class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean" />
  </property>
</bean>
```

上面的配置使用一个春天 FactoryBean 的实现, FieldRetrievingFactoryBean , 设置的值 “隔离” 财产在bean 的价值 “java sql连接事务序列化” 常数。 这是很好,但是它是有点冗长和(不必要的) 春天的内部管道暴露给最终用户。

下面的XML的基于版本更简洁 和开发者的意图清楚地表达(” 注入这个常数 价值”),它只是更好读。

```
<bean id="..." class="...">
  <property name="isolation">
    <util:constant static-field="java.sql.Connection.TRANSACTION_SERIALIZABLE"/>
  </property>
</bean>
```

设置一个bean属性或构造函数参数从一个字段值

FieldRetrievingFactoryBean 是 FactoryBean (检索一个 静态 或非静态字段值。 它通常是 用于检索 公共 静态 最后 常量,这可能会被用来设置一个 属性值或构造函数的参数为另一个bean。

找到下面的示例展示了如何 静态 现场暴露,通过 使用 staticField 属性:

```
<bean id="myField"
      class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean">
  <property name="staticField" value="java.sql.Connection.TRANSACTION_SERIALIZABLE"/>
</bean>
```

还有一个方便使用的形式 静态 字段被指定为bean名称:

```
<bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE"
    class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean"/>
```

这确实意味着不再有任何选择在bean id(所以 任何其他bean,指的是它还将不得不使用这个长名称), 但这种形式是非常简洁的定义,非常方便使用 内心的bean自id没有指定的bean 参考:

```
<bean id="..." class="...">
  <property name="isolation">
    <bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE"
        class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean" />
  </property>
</bean>
```

它也可以访问非静态(实例)领域的另一个bean, 中描述的API文档 FieldRetrievingFactoryBean 类。

枚举值注入到bean属性或构造函数参数要么非常 容易在春天,在这你不必真的 做 任何事或了解弹簧内部(或甚至等类 随着 FieldRetrievingFactoryBean)。 让我们看一个例子 太容易注入一个枚举值,考虑这JDK 5枚举:

```
package javax.persistence;

public enum PersistenceContextType {

    TRANSACTION,
    EXTENDED

}
```

现在考虑一个setter的类型 PersistenceContextType :

```
package example;

public class Client {

    private PersistenceContextType persistenceContextType;

    public void setPersistenceContextType(PersistenceContextType type) {
        this.persistenceContextType = type;
    }
}
```

. . 和相应的bean定义:

```
<bean class="example.Client">
    <property name="persistenceContextType" value="TRANSACTION" />
</bean>
```

这种方法适用于经典类型安全仿真枚举(JDK 1.4和JDK 1.3); 春天会自动尝试匹配字符串属性值到一个常数 在enum类。

< util:属性路径/ >

之前……

```
<!-- target bean to be referenced by name -->
<bean id="testBean" class="org.springframework.beans.TestBean" scope="prototype">
  <property name="age" value="10"/>
  <property name="spouse">
    <bean class="org.springframework.beans.TestBean">
      <property name="age" value="11"/>
    </bean>
  </property>
</bean>

<!-- will result in 10, which is the value of property 'age' of bean 'testBean' -->
<bean id="testBean.age" class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>
```

上面的配置使用一个春天 FactoryBean 的实现, PropertyPathFactoryBean , 创建一个bean(类型 int)称为 “testBean.age” 这有一个值等于 “年龄” 财产的 “ testBean” bean。

之后...

```
<!-- target bean to be referenced by name -->
<bean id="testBean" class="org.springframework.beans.TestBean" scope="prototype">
  <property name="age" value="10"/>
  <property name="spouse">
    <bean class="org.springframework.beans.TestBean">
      <property name="age" value="11"/>
    </bean>
  </property>
</bean>

<!-- will result in 10, which is the value of property 'age' of bean 'testBean' -->
<util:property-path id="name" path="testBean.age"/>
```

的价值 “路径” 属性的 <属性路径/ > 标记遵循形式 “beanName.beanProperty” 。

使用 < util:属性路径/ > 设定一个bean属性或构造函数参数

PropertyPathFactoryBean 是 FactoryBean 评估属性路径在一个给定的 目标对象。 可以指定目标对象直接或通过一个bean的名字。 这个值可以被用于另一个bean定义为一个属性 值或构造函数参数。

下面是一个例子,一个路径是用来对付另一个bean的名字:

```
// target bean to be referenced by name
<bean id="person" class="org.springframework.beans.TestBean" scope="prototype">
  <property name="age" value="10"/>
  <property name="spouse">
    <bean class="org.springframework.beans.TestBean">
      <property name="age" value="11"/>
    </bean>
  </property>
</bean>

// will result in 11, which is the value of property 'spouse.age' of bean 'person'
<bean id="theAge"
    class="org.springframework.beans.factory.config.PropertyPathFactoryBean">
```

```
  <property name="targetBeanName" value="person"/>
  <property name="propertyPath" value="spouse.age"/>
</bean>
```

在这个例子中,一个路径评估对一个内在豆:

```
<!-- will result in 12, which is the value of property 'age' of the inner bean -->
<bean id="theAge"
    class="org.springframework.beans.factory.config.PropertyPathFactoryBean">
  <property name="targetObject">
    <bean class="org.springframework.beans.TestBean">
      <property name="age" value="12"/>
    </bean>
  </property>
  <property name="propertyPath" value="age"/>
</bean>
```

还有一个快捷形式,该bean的名字是属性的路径。

```
<!-- will result in 10, which is the value of property 'age' of bean 'person' -->
<bean id="person.age"
    class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>
```

这种形式确实意味着没有选择名称的bean。 任何引用还必须使用相同的id,这是路径。 当然,如果作为一种内在的bean,没有必要引用它 所有:

```
<bean id="..." class="...">
  <property name="age">
    <bean id="person.age"
        class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>
  </property>
</bean>
```

结果类型可能是专门设置的实际定义。 这是没有必要对大多数用例,但可以使用一些。 请参阅Javadocs更多信息在这个特性。

< util:属性/ >

之前……

```
<!-- creates a java.util.Properties instance with values loaded from the supplied location -->
<bean id="jdbcConfiguration" class="org.springframework.beans.factory.config.PropertiesFactoryBean">
  <property name="location" value="classpath:com/foo/jdbc-production.properties"/>
</bean>
```

上面的配置使用一个春天 FactoryBean 的实现, PropertiesFactoryBean , 实例化一个 java util属性 实例值加载 提供的 资源位置)。

之后...

```
<!-- creates a java.util.Properties instance with values loaded from the supplied location -->
<util:properties id="jdbcConfiguration" location="classpath:com/foo/jdbc-production.properties"/>
```

< util:列表/ >

之前……

```
<!-- creates a java.util.List instance with values loaded from the supplied 'sourceList' -->
<bean id="emails" class="org.springframework.beans.factory.config.ListFactoryBean">
  <property name="sourceList">
      <list>
        <value>pechorin@hero.org</value>
        <value>raskolnikov@slums.org</value>
        <value>stavrogin@gov.org</value>
        <value>porfiry@gov.org</value>
      </list>
  </property>
</bean>
```

上面的配置使用一个春天 FactoryBean 的实现, ListFactoryBean , 创建一个 java util列表 实例初始化 与值取自提供的 “sourceList” 。

之后...

```
<!-- creates a java.util.List instance with the supplied values -->
<util:list id="emails">
```

```
    <value>pechorin@hero.org</value>
    <value>raskolnikov@slums.org</value>
    <value>stavrogin@gov.org</value>
    <value>porfiry@gov.org</value>
</util:list>
```

你也可以显式地控制具体类型的 列表 这将通过使用化和填充的 ” 列表类” 属性 < util:列表/ > 元素。 例如,如果我们 真的需要一个 java util linkedlist 被实例化,我们可以 使用以下配置:

```
<util:list id="emails" list-class="java.util.LinkedList">
    <value>jackshaftoe@vagabond.org</value>
    <value>eliza@thinkingmanscrumpet.org</value>
    <value>vanhoek@pirate.org</value>
    <value>d'Arcachon@nemesis.org</value>
</util:list>
```

如果没有 ” 列表类” 属性提供的,一个 列表 实现将代为选择容器。

< util:地图/ >

之前……

```
<!-- creates a java.util.Map instance with values loaded from the supplied 'sourceMap' -->
<bean id="emails" class="org.springframework.beans.factory.config.MapFactoryBean">
  <property name="sourceMap">
      <map>
        <entry key="pechorin" value="pechorin@hero.org"/>
        <entry key="raskolnikov" value="raskolnikov@slums.org"/>
        <entry key="stavrogin" value="stavrogin@gov.org"/>
        <entry key="porfiry" value="porfiry@gov.org"/>
      </map>
  </property>
</bean>
```

上面的配置使用一个春天 FactoryBean 的实现, MapFactoryBean , 创建一个 java util地图 实例初始化 以键-值对取自提供的 “sourceMap” 。

之后...

```
<!-- creates a java.util.Map instance with the supplied key-value pairs -->
<util:map id="emails">
    <entry key="pechorin" value="pechorin@hero.org"/>
    <entry key="raskolnikov" value="raskolnikov@slums.org"/>
    <entry key="stavrogin" value="stavrogin@gov.org"/>
    <entry key="porfiry" value="porfiry@gov.org"/>
</util:map>
```

你也可以显式地控制具体类型的 地图 这将通过使用化和填充的 “地图类的 属性 < util:地图/ > 元素。 例如,如果我们 真的需要一个 java util treemap 被实例化,我们可以 使用以下配置:

```
<util:map id="emails" map-class="java.util.TreeMap">
    <entry key="pechorin" value="pechorin@hero.org"/>
    <entry key="raskolnikov" value="raskolnikov@slums.org"/>
    <entry key="stavrogin" value="stavrogin@gov.org"/>
    <entry key="porfiry" value="porfiry@gov.org"/>
</util:map>
```

如果没有 “地图类的 属性提供的,一个 地图 实现将代为选择容器。

< util:设置/ >

之前……

```
<!-- creates a java.util.Set instance with values loaded from the supplied 'sourceSet' -->
<bean id="emails" class="org.springframework.beans.factory.config.SetFactoryBean">
  <property name="sourceSet">
      <set>
        <value>pechorin@hero.org</value>
        <value>raskolnikov@slums.org</value>
        <value>stavrogin@gov.org</value>
        <value>porfiry@gov.org</value>
      </set>
  </property>
</bean>
```

上面的配置使用一个春天 FactoryBean 的实现, SetFactoryBean , 创建一个 java util集 实例初始化 与值取自提供的

” sourceSet” 。

之后...

```
<!-- creates a java.util.Set instance with the supplied values -->
<util:set id="emails">
  <value>pechorin@hero.org</value>
  <value>raskolnikov@slums.org</value>
  <value>stavrogin@gov.org</value>
  <value>porfiry@gov.org</value>
</util:set>
```

你也可以显式地控制具体类型的 集 这将通过使用化和填充的 ” 设置类的 属性 < util:设置/ > 元素。 例如,如果我们 真的需要一个 java util treeset 被实例化,我们可以 使用以下配置:

```
<util:set id="emails" set-class="java.util.TreeSet">
  <value>pechorin@hero.org</value>
  <value>raskolnikov@slums.org</value>
  <value>stavrogin@gov.org</value>
  <value>porfiry@gov.org</value>
</util:set>
```

如果没有 ” 设置类的 属性提供的,一个 集 实现将代为选择容器。

E.2.3A了 JEE 模式

这个 JEE 标签处理Java EE(Java企业版)相关 配置问题,如查找JNDI对象和定义EJB引用。

使用标记 JEE 模式,你需要 下面的序文顶部的Spring XML配置文件; 以下代码片段中的文本引用正确的模式,这样 标签在 JEE 名称空间是可用的。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee.xsd">

<!-- bean definitions here -->

</beans>
```

< jee:jndi查找/ > (简单的)

之前.....

```
<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName" value="jdbc/MyDataSource"/>
</bean>

<bean id="userDao" class="com.foo.JdbcUserDao">
  <!-- Spring will do the cast automatically (as usual) -->
  <property name="dataSource" ref="dataSource"/>
</bean>
```

之后...

```
<jee:jndi-lookup id="dataSource" jndi-name="jdbc/MyDataSource"/>

<bean id="userDao" class="com.foo.JdbcUserDao">
  <!-- Spring will do the cast automatically (as usual) -->
  <property name="dataSource" ref="dataSource"/>
</bean>
```

< jee:jndi查找/ > (与单JNDI环境设置)

之前.....

```
<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName" value="jdbc/MyDataSource"/>
  <property name="jndiEnvironment">
    <props>
      <prop key="foo">bar</prop>
    </props>
  </property>
```

```
</bean>
```

之后...

```
<jee:jndi-lookup id="simple" jndi-name="jdbc/MyDataSource">
  <jee:environment>foo=bar</jee:environment>
</jee:jndi-lookup>
```

< jee:jndi查找/ > (有多个JNDI环境设置)

之前.....

```
<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName" value="jdbc/MyDataSource"/>
  <property name="jndiEnvironment">
    <props>
      <prop key="foo">bar</prop>
      <prop key="ping">pong</prop>
    </props>
  </property>
</bean>
```

之后...

```
<jee:jndi-lookup id="simple" jndi-name="jdbc/MyDataSource">
  <!-- newline-separated, key-value pairs for the environment (standard Properties format) -->
  <jee:environment>
    foo=bar
    ping=pong
  </jee:environment>
</jee:jndi-lookup>
```

< jee:jndi查找/ > (复杂的)

之前.....

```
<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName" value="jdbc/MyDataSource"/>
  <property name="cache" value="true"/>
  <property name="resourceRef" value="true"/>
  <property name="lookupOnStartup" value="false"/>
  <property name="expectedType" value="com.myapp.DefaultFoo"/>
  <property name="proxyInterface" value="com.myapp.Foo"/>
</bean>
```

之后...

```
<jee:jndi-lookup id="simple"
        jndi-name="jdbc/MyDataSource"
        cache="true"
        resource-ref="true"
        lookup-on-startup="false"
        expected-type="com.myapp.DefaultFoo"
        proxy-interface="com.myapp.Foo"/>
```

< jee:local-slsb / > (简单的)

这个 < jee:local-slsb / > 标签配置 SessionBean EJB引用,就无状态。

之前.....

```
<bean id="simple"
    class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">
 <property name="jndiName" value="ejb/RentalServiceBean"/>
 <property name="businessInterface" value="com.foo.service.RentalService"/>
</bean>
```

之后...

```
<jee:local-slsb id="simpleSlsb" jndi-name="ejb/RentalServiceBean"
  business-interface="com.foo.service.RentalService"/>
```

< jee:local-slsb / > (复杂的)

```
<bean id="complexLocalEjb"
```

```
    class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">
  <property name="jndiName" value="ejb/RentalServiceBean"/>
  <property name="businessInterface" value="com.foo.service.RentalService"/>
  <property name="cacheHome" value="true"/>
  <property name="lookupHomeOnStartup" value="true"/>
  <property name="resourceRef" value="true"/>
</bean>
```

之后...

```
<jee:local-slsb id="complexLocalEjb"
   jndi-name="ejb/RentalServiceBean"
   business-interface="com.foo.service.RentalService"
   cache-home="true"
   lookup-home-on-startup="true"
   resource-ref="true">
```

< jee:remote-slsb / >

这个 < jee:remote-slsb / > 标签配置 引用 远程 无状态SessionBean EJB。

之前.....

```
<bean id="complexRemoteEjb"
    class="org.springframework.ejb.access.SimpleRemoteStatelessSessionProxyFactoryBean">
  <property name="jndiName" value="ejb/MyRemoteBean"/>
  <property name="businessInterface" value="com.foo.service.RentalService"/>
  <property name="cacheHome" value="true"/>
  <property name="lookupHomeOnStartup" value="true"/>
  <property name="resourceRef" value="true"/>
  <property name="homeInterface" value="com.foo.service.RentalService"/>
  <property name="refreshHomeOnConnectFailure" value="true"/>
</bean>
```

之后...

```
<jee:remote-slsb id="complexRemoteEjb"
   jndi-name="ejb/MyRemoteBean"
   business-interface="com.foo.service.RentalService"
   cache-home="true"
   lookup-home-on-startup="true"
   resource-ref="true"
   home-interface="com.foo.service.RentalService"
   refresh-home-on-connect-failure="true">
```

E.2.4A了 朗 模式

这个 朗 标签处理暴露对象 写在一个动态的语言如JRuby或Groovy bean在春季 集装箱。

这些标签(和动态语言支持)是全面覆盖 在这一章题为 ChapterA 28日 动态语言支持 。 请做参考, 章详细了解这种支持和 朗 标签 他们自己。

为了完整性,使用标记 朗 模式,你需要以下序文顶部的Spring XML 配置文件,文本在以下代码片段引用 正确的模式,以便标记 朗 名称空间是 可用。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:lang="http://www.springframework.org/schema/lang"
      xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/lang http://www.springframework.org/schema/lang/spring-lang.xsd">

<!-- bean definitions here -->

</beans>
```

E.2.5A了 jms 模式

这个 jms 标签处理配置jms相关 豆类如春天的 MessageListenerContainers 。 这些标签是详细的部分 JMS章 题为 SectionA 23.6,一个SupportaJMS名称空间 。 请做参考, 章详细了解这种支持和 jms 标签 他们自己。

为了完整性,使用标记 jms 模式,你需要以下序文顶部的Spring XML 配置文件,文本在以下代码片段引用 正确的模式,以便标记 jms 名称空间是 可用。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jms="http://www.springframework.org/schema/jms"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/jms http://www.springframework.org/schema/jms/spring-jms.xsd">

<!-- bean definitions here -->

</beans>
```

E.2.6A了 TX (事务)模式

这个 TX 标签处理配置所有这些 豆子在春天的全面支持事务。 这些标签是 在章节的题目为覆盖 ChapterA 12, 事务管理 。

提示

强烈建议您看看 “春天tx xsd” 文件附带的春天 分布。 这个文件是(当然),XML模式对春天的 事务配置,并涵盖所有的各种标记 TX 名称空间,包括属性默认值和 诸如此类的。 这个文件是记录内联,因此信息 这里不再重复的利益坚持干(不要重复 你自己)的原则。

为了完整性,使用标记 TX 模式,你需要以下序文顶部的Spring XML 配置文件,文本在以下代码片段引用 正确的模式,以便标记 TX 名称空间是 可用。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd">

<!-- bean definitions here -->

</beans>
```

注意

通常在使用标记 TX 您还将使用名称空间 标签的 aop 名称空间(因为在春天的声明式事务的支持实现使用 AOP)。 上面的XML代码片段包含相关的线路需要引用 aop 模式 因此,标记 aop 名称空间是可用的。

E.2.7A了 aop 模式

这个 aop 标签处理配置所有的事情 在Spring AOP:这包括Spring的基于代理的AOP框架和弹簧的 整合与AspectJ AOP框架。 这些标签是 全面覆盖在一章题为 ChapterA 9, 面向方面的编程与弹簧 。

为了完整性,使用标记 aop 模式,你需要以下序文顶部的Spring XML 配置文件,文本在以下代码片段引用 正确的模式,以便标记 aop 名称空间是 可用。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd">

<!-- bean definitions here -->

</beans>
```

E.2.8A了 上下文 模式

这个 上下文 标签处理 ApplicationContext 配置相关的管道——即通常不是豆子,是很重要的一个终端用户 而是豆子,做很多繁琐的工作在春天,如 BeanFactoryPostProcessors 。 下面的代码片段引用正确的模式,以便标记 上下文 名称空间是可用的。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
```

```
    xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">

<!-- bean definitions here -->

</beans>
```

注意

这个 上下文 模式仅仅是在Spring 2.5中引入的。

<属性占位符/ >

这个元素激活更换 $ {...} 占位符,解决 对指定的属性文件(作为一个 Spring资源位置)。 这个元素是一个方便的机制,建立了一个来完成 对于你,如果你需要更多的控制 来完成 ,只是 显式定义一个你自己。

<注释配置/ >

激活了弹簧基础设施为各种注释被检测到在bean类: 春天的 @ required 和 @ autowired ,以及 JSR 250的 @PostConstruct , @PreDestroy 和 @ resource (如果可用),和JPA的 persistencecontext 和 @PersistenceUnit (如果可用)。 或者,你可以选择激活个体 BeanPostProcessors 对于那些注释明确。

注意

此元素是 不 Spring的激活处理 transactional 注释。 使用 < tx:注解驱动/ > 元素 为此目的。

<组件扫描/ >

这个元素是详细的在 SectionA 5.9,一个configurationa基于注解的容器 。

<加载时间韦弗/ >

这个元素是详细的在 SectionA 9 8 4,一个装入时编织与AspectJ在春季Frameworka 。

< spring配置/ >

这个元素是详细的在 SectionA 9 8 1,一个使用AspectJ依赖注入与域对象 Springa 。

< mbean出口/ >

这个元素是详细的在 SectionA 24 4 3,一个配置MBean exporta基于注解的 。

E.2.9A了 工具 模式

这个 工具 标签是用于当你想添加 工具特定元数据定制的配置元素。 这个元数据 可以被工具,也意识到这个元数据和工具可以吗 然后做几乎任何他们想要使用它(验证等)。

这个 工具 标签没有被记录在这里公布的 弹簧由于他们目前正在审查。 如果你是一个第三方工具 供应商和你愿意贡献这个审查过程,然后做邮件 春天的邮件列表。 目前支持的 工具 标签中可以找到该文件 “弹簧工具xsd” 在 ' src / org/springframework/beans/factory/xml' 目录的 春源分布。

E.2.10A了 JDBC 模式

这个 JDBC 标记允许您快速配置一个 嵌入式数据库或初始化一个现有的数据源。 这些标签是 记录在 SectionA 14.8,一个supporta嵌入式数据库 和 SectionA 14.9,一个DataSourcea初始化 分别。

使用标记 JDBC 模式,你需要以下序文顶部的Spring XML 配置文件,文本在以下代码片段引用 正确的模式,以便标记 JDBC 名称空间是 可用。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/jdbc http://www.springframework.org/schema/jdbc/spring-jdbc.xsd">
```

```
<!-- bean definitions here -->

</beans>
```

E.2.11A了 缓存 模式

这个 缓存 标签可以用来支持Spring的 @CacheEvict , @CachePut 和 @Caching 注释。 它还支持声明性 基于xml的缓存。 看到 SectionA 29 3 5,一个annotationsa启用高速缓存 和 SectionA 29.4,一个声明性xml的cachinga 详情。

使用标记 缓存 模式,你需要以下序文顶部的Spring XML 配置文件,文本在以下代码片段引用 正确的模式,以便标记 缓存 名称空间是 可用。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jdbc="http://www.springframework.org/schema/cache"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/cache http://www.springframework.org/schema/jdbc/spring-cache.xsd">

<!-- bean definitions here -->

</beans>
```

E.2.12A了 bean 模式

最后但并非最不重要,我们有标记 bean 模式。 这些是相同的标签,已经在春天因为非常的黎明框架。 例子的各种标记 bean 模式是没有在这里显示 因为他们是有相当的全面覆盖 SectionA 5 4 2,一个依赖性和配置在detaila (实际上在整个 章)。

有一件事是新到bean标签本身在Spring 2.0是这个想法 元数据的任意bean。 在Spring 2.0中,现在可以添加零个或更多 键/值对 < bean / > XML定义。 什么,如果 什么,是做了这个额外的元数据是完全取决于您自己的自定义逻辑(和 所以一般只使用如果您正在编写自己的自定义标签中描述 附录题为 AppendixA F, 可扩展的XML编辑)。

找到下面的一个例子 < meta / > 标记上下文 周围的 < bean / > (请注意,没有任何逻辑 解释它的元数据是无用的原有)有效。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="foo" class="x.y.Foo">
    <meta key="cacheName" value="foo"/>
    <property name="name" value="Rick"/>
  </bean>

</beans>
```

对于上面的例子,你可能会认为,有一些 逻辑,使用bean定义和设置一些缓存基础设施 使用提供的元数据。

AppendixA F。 一个可扩展的XML编辑

F.1A介绍

从版本2.0,春天有特色的基于扩展机制 春天的基本的XML格式,定义和配置bean。 这部分是 致力于详细介绍如何对编写自己的自定义XML bean定义 解析器和集成这样的解析器到Spring IoC容器。

为了便于编辑配置文件使用支持模式的XML编辑器, 弹簧的可扩展的XML配置机制是基于XML模式。 如果你是 不熟悉Spring XML配置扩展当前的事物而来的 标准弹簧分布,请先阅读附录资格 AppendixA E, XML的基于配置 。

创建新的XML配置扩展可以通过以下(相对) 简单的步骤:

1. 创作 一个XML schema来描述您的自定义元素(年代)。
2. 编码 一个自定义 NamespaceHandler 实现(这是一个简单的一步,别担心)。
3. 编码 一个或多个 BeanDefinitionParser 实现(这就是真正的工作都完成了)。
4. 注册 上述构件与弹簧(这也是一个简单的步骤)。

下面是每个步骤的描述。 对于本例,我们将创建 XML扩展(一个自定义的XML元素),允许我们来配置对象的类型

SimpleDateFormat (从 java文本 包) 通过简单的方式。 当我们完成后,我们将能够定义bean定义的类型 SimpleDateFormat 这样的:

```
<myns:dateformat id="dateFormat"
   pattern="yyyy-MM-dd HH:mm"
   lenient="true"/>
```

(不要担心的事实,这个例子非常简单;更多 详细的例子遵循之后。 目的在这第一个简单示例是步行 你通过基本步骤。)

F。 2一个创作模式

创建XML配置扩展使用Spring的IoC容器 首先编写一个XML Schema来描述扩展。 下面 我们将会使用的模式来配置吗 SimpleDateFormat 对象。

```
<!-- myns.xsd (inside package org/springframework/samples/xml) -->

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.mycompany.com/schema/myns"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:beans="http://www.springframework.org/schema/beans"
   targetNamespace="http://www.mycompany.com/schema/myns"
   elementFormDefault="qualified"
   attributeFormDefault="unqualified">

  <xsd:import namespace="http://www.springframework.org/schema/beans"/>

  <xsd:element name="dateformat">
    <xsd:complexType>
      <xsd:complexContent>
        <xsd:extension base="beans:identifiedType">
          <xsd:attribute name="lenient" type="xsd:boolean"/>
          <xsd:attribute name="pattern" type="xsd:string" use="required"/>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>
  </xsd:element>

</xsd:schema>
```

(强调行包含一个扩展基地的所有标签 将可识别的(这意味着他们有一个吗 id 属性 这将作为bean标识符在容器)。 我们能够使用这个 属性,因为我们进口的弹簧提供 “豆子” 名称空间)。

上面的模式将被用来配置 SimpleDateFormat 对象,直接在XML应用程序上下文文件使用 < myns:dateformat / > 元素。

```
<myns:dateformat id="dateFormat"
   pattern="yyyy-MM-dd HH:mm"
   lenient="true"/>
```

请注意,在我们已经创建了基础设施类,上面的XML片段 基本上是一模一样的以下XML片段。 换句话说, 我们只是创建一个bean容器中,确定的名称 “dateFormat” 类型的 SimpleDateFormat ,一个 两个属性集。

```
<bean id="dateFormat" class="java.text.SimpleDateFormat">
   <constructor-arg value="yyyy-HH-dd HH:mm"/>
   <property name="lenient" value="true"/>
</bean>
```

注意

基于的方法来创建配置格式允许 紧密地与一个IDE的支持模式的XML编辑器。 使用正确 创作模式,您可以使用自动完成,用户选择几个 配置选项中定义的枚举。

F。 3一个编码一个 NamespaceHandler

除了模式,我们需要一个 NamespaceHandler 将解析所有元素的特定名称空间弹簧遇到 而解析配置文件。 这个 NamespaceHandler 应该在我们的情况下照顾的解析 myns:dateformat 元素。

这个 NamespaceHandler 界面很简单, 它功能只是三个方法:

- init() ——允许初始化 这个 NamespaceHandler 和将被弹簧 使用前处理程序
- BeanDefinition解析(元素,ParserContext) —— 称为当春天遇到一个顶级元素(不是嵌套在bean定义 或一个不同的名称空间)。 这个方法可以注册的bean定义本身和/或 返回一个bean定义。

- BeanDefinitionHolder装饰(节点,BeanDefinitionHolder,ParserContext) —— 称为当春天遇到属性或嵌套的元素的一个不同的名称空间。 装饰的一个或多个bean定义用于例子 开箱即用的范围Spring 2.0支持 。 我们将首先强调一个简单的例子,没有使用装饰,之后 我们将展示装饰在更高级的例子。

虽然它是完全有可能的代码你自己的 NamespaceHandler 对于整个名称空间 (和因此提供代码解析每个元素的名称空间), 这是常有的事,每个顶级XML元素在Spring XML 配置文件导致单个bean定义(如我们的 情况下,一个单一的 < myns:dateformat / > 元素 结果在一个单一的 SimpleDateFormat bean定义)。 弹簧特性许多方便的类,支持这个场景。 在这个例子中,我们将使用 NamespaceHandlerSupport 类:

```
package org.springframework.samples.xml;

import org.springframework.beans.factory.xml.NamespaceHandlerSupport;

public class MyNamespaceHandler extends NamespaceHandlerSupport {

    public void init() {
        registerBeanDefinitionParser("dateformat", new SimpleDateFormatBeanDefinitionParser());
    }
}
```

细心的读者会注意到,实际上没有一大堆的 在这类解析逻辑。 事实上..... 这个 NamespaceHandlerSupport 类有一个建在代表团的概念。 它支持的任何数量的登记 的 BeanDefinitionParser 情况下,它将代表 当它需要解析一个元素的名称空间。 这个干净的关注点分离 允许 NamespaceHandler 处理业务流程 解析的 所有 的自定义元素的名称空间, 而委托给 BeanDefinitionParsers 来做基础工作的 XML解析;这意味着每个 BeanDefinitionParser 将 只包含逻辑解析一个自定义的元素,如我们所见,在下一步

F。 4一个编码一个 BeanDefinitionParser

一个 BeanDefinitionParser 将会使用如果 NamespaceHandler 遇到一个XML元素的类型 被映射到特定的bean定义解析器 (这是 “dateFormat” 在这种情况下)。 换句话说, BeanDefinitionParser 是 负责解析 一个 不同的顶级XML元素中定义的 模式。 在解析器,我们将可以访问XML元素(因此它的子元素太) 这样我们可以解析我们的自定义XML内容,可以看到下面的例子:

```
package org.springframework.samples.xml;

import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.xml.AbstractSingleBeanDefinitionParser;
import org.springframework.util.StringUtils;
import org.w3c.dom.Element;

import java.text.SimpleDateFormat;

public class SimpleDateFormatBeanDefinitionParser extends AbstractSingleBeanDefinitionParser { ❶

  protected Class getBeanClass(Element element) {
    return SimpleDateFormat.class; ❷
  }

  protected void doParse(Element element, BeanDefinitionBuilder bean) {
    // this will never be null since the schema explicitly requires that a value be supplied
    String pattern = element.getAttribute("pattern");
    bean.addConstructorArg(pattern);

    // this however is an optional property
    String lenient = element.getAttribute("lenient");
    if (StringUtils.hasText(lenient)) {
      bean.addPropertyValue("lenient", Boolean.valueOf(lenient));
    }
  }
}
```

❶ 我们使用弹簧提供 AbstractSingleBeanDefinitionParser 处理很多基本的繁重工作的创建 单 BeanDefinition 。
❷ 我们提供 AbstractSingleBeanDefinitionParser 超类 与类型,我们的单 BeanDefinition 将代表。

在这个简单的例子中,这是所有我们需要做的。 创造我们的单 BeanDefinition 是由 AbstractSingleBeanDefinitionParser 超类,是提取和设置的bean定义的惟一标识符。

F。 5一个注册处理程序和模式

编码完成! 剩下要做的就是以某种方式使Spring XML 解析基础设施的意识到我们的定制元素;我们通过注册我们的习惯 NamespaceHandler 和定制的XSD文件在两个特殊目的 属性文件。 这些属性文件都放在一个 “meta - inf’ 目录在您的应用程序,可以 例,是分布式与二进制类的JAR文件。 Spring XML解析 基础设施将会自动捡起你的新扩展通过使用这些特殊的 属性文件,格式是详细的下面。

F.5.1A “meta - inf / spring处理程序”

properties文件称为 “弹簧处理程序” 包含一个映射 XML Schema名称空间的uri处理程序类。 所以对于我们的示例,我们需要写 以下:

```
http\://www.mycompany.com/schema/myns=org.springframework.samples.xml.MyNamespaceHandler
```

(” : “ 性格是一种有效的分隔符在Java属性的格式, 所以 ” : “ 字符在URI需要转义反斜线。)

第一部分(键)键-值对的是与您的自定义名称空间URI相关联 扩展和需要 完全匹配 的价值 的targetNamespace” 属性中指定自定义XSD schema。

F.5.2A “meta - inf / spring模式”

properties文件称为 “春天模式” 包含一个映射 XML Schema位置(指随着模式中声明的XML文件 使用模式的一部分 “xsi:schemaLocation” 属性) 到 类路径 资源。 这个文件是需要防止弹簧从 绝对不必使用默认的 EntityResolver 这需要 上网检索模式文件。 如果你指定映射在这个属性文件, 春天将搜索模式在类路径中(在这种情况下 “myns.xsd” 在 “org.springframework.samples.xml” 包):

```
http\://www.mycompany.com/schema/myns/myns.xsd=org/springframework/samples/xml/myns.xsd
```

这样的结果就是你被鼓励来部署您的XSD文件(s)旁边 这个 NamespaceHandler 和 BeanDefinitionParser 类在类路径中。

F。 6一个使用一个自定义的扩展在Spring的XML配置

使用一个自定义的扩展,你自己已经实现没有区别 使用其中一个 “自定义” 扩展Spring提供了直接从这个盒子。 找到下面 使用自定义的一个例子 < dateformat / > 元素开发的 前面的步骤在Spring XML配置文件。

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:myns="http://www.mycompany.com/schema/myns"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.mycompany.com/schema/myns http://www.mycompany.com/schema/myns/myns.xsd">

  <!-- as a top-level bean -->
  <myns:dateformat id="defaultDateFormat" pattern="yyyy-MM-dd HH:mm" lenient="true"/>

  <bean id="jobDetailTemplate" abstract="true">
    <property name="dateFormat">
      <!-- as an inner bean -->
      <myns:dateformat pattern="HH:mm MM-dd-yyyy"/>
    </property>
  </bean>

</beans>
```

F。 7一个丰满的例子

发现下面一些多定制XML扩展的例子更耐人寻味。

F.7.1A嵌套定义标签内自定义标记

这个例子演示了如何去写各种工件 需要满足一个目标以下配置:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:foo="http://www.foo.com/schema/component"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.foo.com/schema/component http://www.foo.com/schema/component/component.xsd">

  <foo:component id="bionic-family" name="Bionic-1">
    <foo:component name="Mother-1">
      <foo:component name="Karate-1"/>
      <foo:component name="Sport-1"/>
    </foo:component>
    <foo:component name="Rock-1"/>
  </foo:component>
```

```
</beans>
```

上面的配置其实巢自定义扩展在每个其他。 类 这实际上是由上述配置 < foo:组件/ > 元素是 组件 类(见下方)。 注意 如何 组件 类并 不 暴露 一个setter方法 “组件” 财产,这使得它很难 (或者说几乎不可能)配置的bean定义 组件 类使用setter注入。

```
package com.foo;

import java.util.ArrayList;
import java.util.List;

public class Component {

  private String name;
  private List<Component> components = new ArrayList<Component> ();

  // mmm, there is no setter method for the 'components'
  public void addComponent(Component component) {
    this.components.add(component);
  }

  public List<Component> getComponents() {
    return components;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
}
```

典型的解决此问题的方法是创建一个自定义的 FactoryBean 这暴露了一个setter属性 “组件” 财产。

```
package com.foo;

import org.springframework.beans.factory.FactoryBean;

import java.util.List;

public class ComponentFactoryBean implements FactoryBean<Component> {

  private Component parent;
  private List<Component> children;

  public void setParent(Component parent) {
    this.parent = parent;
  }

  public void setChildren(List<Component> children) {
    this.children = children;
  }

  public Component getObject() throws Exception {
    if (this.children != null && this.children.size() > 0) {
      for (Component child : children) {
        this.parent.addComponent(child);
      }
    }
    return this.parent;
  }

  public Class<Component> getObjectType() {
    return Component.class;
  }

  public boolean isSingleton() {
    return true;
  }
}
```

这是所有非常好,并很好地工作,但暴露了很多弹簧管道到 最终用户。 我们要做的,就是写一个自定义的扩展,隐藏掉这一切 春天的管道。 如果我们坚持 描述的步骤 以前 ,我们将首先创建XSD schema定义的结构 我们的定制标记。

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<xsd:schema xmlns="http://www.foo.com/schema/component"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://www.foo.com/schema/component"
      elementFormDefault="qualified"
      attributeFormDefault="unqualified">

  <xsd:element name="component">
    <xsd:complexType>
```

```
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element ref="component"/>
      </xsd:choice>
      <xsd:attribute name="id" type="xsd:ID"/>
      <xsd:attribute name="name" use="required" type="xsd:string"/>
    </xsd:complexType>
  </xsd:element>

</xsd:schema>
```

然后我们创建一个定制的 NamespaceHandler 。

```
package com.foo;

import org.springframework.beans.factory.xml.NamespaceHandlerSupport;

public class ComponentNamespaceHandler extends NamespaceHandlerSupport {

  public void init() {
    registerBeanDefinitionParser("component", new ComponentBeanDefinitionParser());
  }
}
```

接下来是定制的 BeanDefinitionParser 。 记得 ,我们正在打造的是一个 BeanDefinition 描述 一个 ComponentFactoryBean 。

```
package com.foo;

import org.springframework.beans.factory.config.BeanDefinition;
import org.springframework.beans.factory.support.AbstractBeanDefinition;
import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.support.ManagedList;
import org.springframework.beans.factory.xml.AbstractBeanDefinitionParser;
import org.springframework.beans.factory.xml.ParserContext;
import org.springframework.util.xml.DomUtils;
import org.w3c.dom.Element;

import java.util.List;

public class ComponentBeanDefinitionParser extends AbstractBeanDefinitionParser {

  protected AbstractBeanDefinition parseInternal(Element element, ParserContext parserContext) {
    return parseComponentElement(element);
  }

  private static AbstractBeanDefinition parseComponentElement(Element element) {
    BeanDefinitionBuilder factory = BeanDefinitionBuilder.rootBeanDefinition(ComponentFactoryBean.class);
    factory.addPropertyValue("parent", parseComponent(element));

    List<Element> childElements = DomUtils.getChildElementsByTagName(element, "component");
    if (childElements != null && childElements.size() > 0) {
      parseChildComponents(childElements, factory);
    }

    return factory.getBeanDefinition();
  }

  private static BeanDefinition parseComponent(Element element) {
    BeanDefinitionBuilder component = BeanDefinitionBuilder.rootBeanDefinition(Component.class);
    component.addPropertyValue("name", element.getAttribute("name"));
    return component.getBeanDefinition();
  }

  private static void parseChildComponents(List<Element> childElements, BeanDefinitionBuilder factory) {
    ManagedList<BeanDefinition> children = new ManagedList<BeanDefinition>(childElements.size());

    for (Element element : childElements) {
      children.add(parseComponentElement(element));
    }

    factory.addPropertyValue("children", children);
  }
}
```

最后,各种工件需要注册到Spring XML基础设施。

```
# in 'META-INF/spring.handlers'
http\://www.foo.com/schema/component=com.foo.ComponentNamespaceHandler
```

```
# in 'META-INF/spring.schemas'
http\://www.foo.com/schema/component/component.xsd=com/foo/component.xsd
```

F.7.2A定制属性在“正常”的元素

编写自己的自定义解析器和关联的工件并不困难,但有时它 不是正确的做法。 考虑场景,您需要添加元数据已 现有的bean定义。 在这种情况下,你肯定不想要去写 你自己的整个定制扩展;而你只是想添加一个额外的属性 现有的bean定义元素。

通过另一个示例,让我们说,你是服务类定义bean 定义为服务对象,将(未知的)被访问集群 JCache ,你想确保 指定的JCache实例是急切地开始在周围的集群:

```
<bean id="checkingAccountService" class="com.foo.DefaultCheckingAccountService"
    jcache:cache-name="checking.account">
  <!-- other dependencies here... -->
</bean>
```

我们要做的,这是创建另一个 BeanDefinition 当 “jcache:缓存名称” 属性是解析;这 BeanDefinition 将初始化指定JCache吗对于我们。 我们也将修改现有的 BeanDefinition 为 ” checkingAccountService” 所以,它将依赖于这个 新JCache-initializing BeanDefinition 。

```
package com.foo;

public class JCacheInitializer {

  private String name;

  public JCacheInitializer(String name) {
    this.name = name;
  }

  public void initialize() {
    // lots of JCache API calls to initialize the named cache...
  }
}
```

现在到自定义扩展。 首先,编写的XSD schema描述 自定义属性(很容易在这种情况下)。

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<xsd:schema xmlns="http://www.foo.com/schema/jcache"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.foo.com/schema/jcache"
        elementFormDefault="qualified">

  <xsd:attribute name="cache-name" type="xsd:string"/>

</xsd:schema>
```

其次,相关的 NamespaceHandler 。

```
package com.foo;

import org.springframework.beans.factory.xml.NamespaceHandlerSupport;

public class JCacheNamespaceHandler extends NamespaceHandlerSupport {

  public void init() {
    super.registerBeanDefinitionDecoratorForAttribute("cache-name",
        new JCacheInitializingBeanDefinitionDecorator());
  }
}
```

接下来,解析器。 注意,在这种情况下,因为我们将要解析XML 属性,我们写一个 BeanDefinitionDecorator 而不是 BeanDefinitionParser 。

```
package com.foo;

import org.springframework.beans.factory.config.BeanDefinitionHolder;
import org.springframework.beans.factory.support.AbstractBeanDefinition;
import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.xml.BeanDefinitionDecorator;
import org.springframework.beans.factory.xml.ParserContext;
import org.w3c.dom.Attr;
import org.w3c.dom.Node;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class JCacheInitializingBeanDefinitionDecorator implements BeanDefinitionDecorator {

  private static final String[] EMPTY_STRING_ARRAY = new String[0];

  public BeanDefinitionHolder decorate(
      Node source, BeanDefinitionHolder holder, ParserContext ctx) {
```

```
        String initializerBeanName = registerJCacheInitializer(source, ctx);
        createDependencyOnJCacheInitializer(holder, initializerBeanName);
        return holder;
    }

    private void createDependencyOnJCacheInitializer(BeanDefinitionHolder holder, String initializerBeanName) {
        AbstractBeanDefinition definition = ((AbstractBeanDefinition) holder.getBeanDefinition());
        String[] dependsOn = definition.getDependsOn();
        if (dependsOn == null) {
            dependsOn = new String[]{initializerBeanName};
        } else {
            List dependencies = new ArrayList(Arrays.asList(dependsOn));
            dependencies.add(initializerBeanName);
            dependsOn = (String[]) dependencies.toArray(EMPTY_STRING_ARRAY);
        }
        definition.setDependsOn(dependsOn);
    }

    private String registerJCacheInitializer(Node source, ParserContext ctx) {
        String cacheName = ((Attr) source).getValue();
        String beanName = cacheName + "-initializer";
        if (!ctx.getRegistry().containsBeanDefinition(beanName)) {
            BeanDefinitionBuilder initializer = BeanDefinitionBuilder.rootBeanDefinition(JCacheInitializer.class);
            initializer.addConstructorArg(cacheName);
            ctx.getRegistry().registerBeanDefinition(beanName, initializer.getBeanDefinition());
        }
        return beanName;
    }
}
```

最后,各种工件需要注册到Spring XML基础设施。

```
# in 'META-INF/spring.handlers'
http\://www.foo.com/schema/jcache=com.foo.JCacheNamespaceHandler
```

```
# in 'META-INF/spring.schemas'
http\://www.foo.com/schema/jcache/jcache.xsd=com/foo/jcache.xsd
```

F。 8一个进一步的资源

以下链接关于XML模式和进一步资源的可扩展的XML支持 在本章中介绍。

- 这个 XML Schema第1部分:结构第二版
- 这个 XML Schema第2部分:数据类型第二版

AppendixA g一个弹簧tld

G.1A介绍

一个视图的技术可以使用Spring框架 是Java服务器页面(jsp)。 帮助你实现视图使用Java服务器页面 Spring框架为您提供了一些标记为评估错误,设置 主题和输出国际化的消息。

请注意,各种标签产生的这种形式标记库 符合 xhtml - 1.0 -严格的规范 和服务员 dtd 。

本附录描述了 弹簧tld 标记库。

- SectionA G。 2、一个了 绑定 taga
- SectionA G。 3、一个了 escapeBody taga
- SectionA G。 4、一个了 hasBindErrors taga
- SectionA G。 5、一个了 htmlEscape taga
- SectionA G。 6、一个了 消息 taga
- SectionA G。 7、一个了 nestedPath taga
- SectionA G。 8、一个了 主题 taga
- SectionA G。 9日,一了 变换 taga
- SectionA G。 10、一个了 url taga
- SectionA G。 11日,的 eval taga

G.2A了 绑定 标签

提供BindStatus对象给定绑定路径。 HTML转义旗帜参与一个是页面范围或应用程序设置 (即通过HtmlEscapeTag

或“defaultHtmlEscape“上下文参数在web . xml)。

g 1一个属性为多

属性	要求?	运行时表达式?	描述
htmlEscape	假	真正的	设置HTML转义为这个标签,作为布尔值。 覆盖 产品在使用◆义设置当前页面。
ignoreNestedPath	假	真正的	设置是否忽略一个嵌套的路径,如果任何。 默认是不能忽视的。
路径	真正的	真正的	bean的路径或bean属性绑定状态 信息。 例如帐户名称、公司地址的邮政编码或者只是雇员。 状态对象将出口到页面范围, 专门为这个bean或bean属性

G.3A了 escapeBody 标签

逃其封闭的主体内容,利用HTML转义和/或JavaScript转义。 HTML转义旗帜参与一个是页面范围或应用程序设置 (即通过HtmlEscapeTag或“defaultHtmlEscape“上下文参数在web . xml)。

为多g 2一个属性

属性	要求?	运行时表达式?	描述
htmlEscape	假	真正的	设置HTML转义为这个标签,作为布尔值。 覆盖 默认的HTML转义设置当前页面。
javaScriptEscape	假	真正的	这个标签设置JavaScript逃跑,因为布尔值。 默认是假的。

G.4A了 hasBindErrors 标签

提供了错误的实例中绑定错误。 HTML转义旗帜参与一个是页面范围或应用程序设置 (即通过HtmlEscapeTag或“defaultHtmlEscape“上下文参数在web . xml)。

为多g 3一个属性

属性	要求?	运行时表达式?	描述
htmlEscape	假	真正的	设置HTML转义为这个标签,作为布尔值。 覆盖默认的HTML转义设置当前页面。
名称	真正的	真正的	bean的名称在请求,需要 检查错误。 如果错误是可利用为这个bean,他们 会绑定在 “错误的关键。

G.5A了 htmlEscape 标签

设置默认的HTML逃避当前页面的值。 覆盖一个“defaultHtmlEscape“上下文参数在web。 xml,如果任何。

为多g 4一个属性

属性	要求?	运行时表达式?	描述
defaultHtmlEscape	真正的	真正的	设置默认值为HTML转义,是把 到当前PageContext。

G.6A了 消息 标签

检索与给定的消息代码,或文本如果代码不是可解决的。 HTML转义旗帜参与一个是页面范围或应用程序设置 (即通过HtmlEscapeTag或“defaultHtmlEscape“上下文参数在web . xml)。

为多g 5一个属性

属性	要求?	运行时表达式?	描述
参数	假	真正的	设置可选的消息参数对于这个标记,如一个 (逗号)分隔的字符串(每个字符串参数可以包含JSP EL), 一个对象数组(用作参数数组),或一个对象(使用 作为单一参数)。
argumentSeparator	假	真正的	分隔符用于分裂 参数的字符串值,默认为“逗号”(',')。
代码	假	真正的	代码(关键)时使用查找消息。 如果代码不提供文本属性将被使用。
htmlEscape	假	真正的	设置HTML转义为这个标签,作为布尔值。 覆盖默认的HTML转义设置当前页面。
javaScriptEscape	假	真正的	这个标签设置JavaScript逃跑,因为布尔值。 默认是假的。
消息	假	真正的	一个MessageSourceResolvable参数(直接或通过JSP EL)。 完美契合一起使用时弹簧的验证错误 类,它们都实现了MessageSourceResolvable接口。 对于例,这允许您遍历所有的错误在一个表单, 通过每个错误(使用一个运行时表达式)的价值 “消息”属性,从而影响容易显示这样的错误 消息。
范围	假	真正的	当出口范围使用结果给一个变量。 这个属性只会用在当var也是集。可能的值是页面、请求、会话和应用程序。
文本	假	真正的	默认文本输出消息时给定的代码 不能被发现。 如果两个文本和代码没有设置,标签将 输出空。
var	假	真正的	字符串使用绑定的结果页面, 请求、会话或应用程序范围。 如果未指定,结果 得到输出至作者(即通常直接向JSP)。

G.7A了 nestedPath 标签

设置一个嵌套的路径使用bind标记的路径。

为多g 6一个属性

属性	要求?	运行时表达式?	描述
路径	真正的	真正的	设置路径,这个标签应当适用。 如。 ‘客户’ 允许绑定路径像 ‘地址。 街”而不是“客户地址街”。

G.8A了 主题 标签

检索主题信息与给定的代码,或文本如果代码不是可解决的。 HTML转义旗帜参与一个是页面范围或应用程序设置 (即通过HtmlEscapeTag或“defaultHtmlEscape“上下文参数在web . xml)。

为多g 7一个属性

属性	要求?	运行时表达式?	描述
参数	假	真正的	设置可选的消息参数对于这个标记,如一个 (逗号)分隔的字符串(每个字符串参数可以包含JSP EL), 一个对象数组(用作参数数组),或一个对象(使用 作为单一参数)。
argumentSeparator	假	真正的	分隔符用于分裂 参数的字符串值,默认为“逗号”(',')。
代码	假	真正的	代码(关键)时使用查找消息。 如果代码不提供文本属性将被使用。
htmlEscape	假	真正的	设置HTML转义为这个标签,作为布尔值。 覆盖默认的HTML转义设置当前页面。
javaScriptEscape	假	真正的	这个标签设置JavaScript逃跑,因为布尔值。 默认是假的。
消息	假	真正的	一个MessageSourceResolvable参数(直接或通过JSP EL)。
范围	假	真正的	当出口范围使用结果给一个变量。 这个属性只会用在当var也是集。可能的值是 页面、请求、会话和应用程序。
文本	假	真正的	默认文本输出消息时给定的代码 不能被发现。 如果两个文本和代码没有设置,标签将 输出空。
var	假	真正的	字符串使用绑定的结果页面, 请求、会话或应用程序范围。 如果未指定,结果得到输出至作者(即通常直接向JSP)。

G.9A了 变换 标签

提供了转换的字符串变量,使用一个合适的 自定义属性编辑器从BindTag(只能内部使用BindTag)。 HTML转义旗帜参与一个是页面范围或应用程序设置 (即通过HtmlEscapeTag或“defaultHtmlEscape的上下文参数在web . xml)。

为多g 8一个属性

属性	要求?	运行时表达式?	描述
htmlEscape	假	真正的	设置HTML转义为这个标签,作为布尔值。 覆盖 产品在使用◆义设置当前页面。
范围	假	真正的	当出口范围使用结果给一个变量。 这个属性只会用在当var也是集。可能的值是 页面、请求、会话和应用程序。
价值	真正的	真正的	值改变。 这是你想要的实际对象 有改变(例如一个日期)。 使用属性编辑器, 目前在用的“弹簧:绑定”标签。
var	假	真正的	字符串使用绑定的结果页面, 请求、会话或应用程序范围。 如果未指定,结果得到输出到作者(即通常直接向JSP)。

G.10A了 url 标签

创建了url,支持URI模板变量、HTML / XML和Javascript逃离逃离。 仿照了JSTL c:url标记和向后兼容性在心里。

为多g 9一个属性

属性	要求?	运行时表达式?	描述
url	真正的	真正的	URL来构建。 这个值可以包括模板{占位符} 这是替换URL编码值的命名参数。 参数 必须定义使用了param标记在这个标签的主体。
上下文	假	真正的	指定一个远程应用程序上下文路径。 默认的是 当前的应用程序上下文路径。
var	假	真正的	变量的名字出口URL值。 如果没有指定URL被编写为输出。
范围	假	真正的	var的范围。 “应用程序”、“会话”、“请求”和 “页面”范围的支持。 默认页面范围。 这个属性没有 效果,除非var属性定义。
htmlEscape	假	真正的	设置HTML转义为这个标签,作为一个布尔值。 覆盖 默认的HTML转义设置当前页面。
javaScriptEscape	假	真正的	这个标签设置JavaScript逃跑,一个布尔值。 默认是假的。

G.11A了 eval 标签

评估一个春天的表达式(?),或者打印结果或将它赋给一个变量。

为多g 10一个属性

属性	要求?	运行时表达式?	描述
表达式	真正的	真正的	要运算的表达式。
var	假	真正的	变量的名字出口评估的结果来。 如果没有指定评估的结果转换成字符串和书面作为输出。
范围	假	真正的	var的范围。 “应用程序”、“会话”、“请求”和 “页面”范围的支持。 默认页面范围。 这个属性没有 效果,除非var属性定义。
htmlEscape	假	真正的	设置HTML转义为这个标签,作为一个布尔值。 覆盖 默认的HTML转义设置当前页面。
javaScriptEscape	假	真正的	这个标签设置JavaScript逃跑,一个布尔值。 默认是假的。

AppendixA h一个弹簧形式tld

H.1A介绍

一个视图的技术可以使用Spring框架 是Java服务器页面(jsp)。 帮助你实现视图使用Java服务器页面 Spring框架为您提供了一些标记为评估错误,设置 主题和输出国际化的消息。

请注意,各种标签产生的这种形式标记库 符合 xhtml - 1.0 -严格的规范 和服务员 dtd 。

本附录描述了 弹簧形式tld 标记库。

- SectionA H。 2、一个了 复选框 taga
- SectionA H。 3、一个了 复选框 taga
- SectionA H。 4、一个了 错误 taga
- SectionA H。 5、一个了 形式 taga
- SectionA H。 6、一个了 隐藏 taga
- SectionA H。 7、一个了 输入 tag”
- Section H.8, “The label tag”
- Section H.9, “The option tag”
- Section H.10, “The options tag”
- Section H.11, “The password tag”
- Section H.12, “The radiobutton tag”
- Section H.13, “The radiobuttons tag”
- Section H.14, “The 选择 taga
- SectionA H。 15、一个了 Textarea taga

H.2A了 复选框 标签

呈现一个HTML的输入“标记”复选框”型。

h 1一个属性为多

属性	要求?	运行时表达式?	描述
accesskey	假	真正的	HTML标准属性
cssClass	假	真正的	相当于“类”——HTML可选属性
cssErrorClass	假	真正的	相当于“类”——HTML可选属性。 当绑定使用字段有错误。
cssStyle	假	真正的	相当于“风格”——HTML可选属性
dir	假	真正的	HTML标准属性
禁用	假	真正的	HTML可选属性。 设置此属性的值到“真正的”(没有引号)将禁用的HTML元素。
htmlEscape	假	真正的	启用/禁用呈现HTML转义的价值观。
id	假	真正的	HTML标准属性
标签	假	真正的	值显示为标签的一部分
朗	假	真正的	HTML标准属性
onblur	假	真正的	HTML事件属性
onchange	假	真正的	HTML事件属性
onclick	假	真正的	HTML事件属性
ondblclick	假	真正的	HTML事件属性
得到焦点	假	真正的	HTML事件属性
onkeydown	假	真正的	HTML事件属性
onkeypress	假	真正的	HTML事件属性
onkeyup	假	真正的	HTML事件属性
onmousedown	假	真正的	HTML事件属性
onmousemove	假	真正的	HTML事件属性
滑	假	真正的	HTML事件属性
onmouseover	假	真正的	HTML事件属性
onmouseup	假	真正的	HTML事件属性

路径	真正的	真正的	路径属性数据绑定
tabindex	假	真正的	HTML标准属性
标题	假	真正的	HTML标准属性
价值	假	真正的	HTML可选属性

H.3A了 复选框 标签

呈现多个HTML标记“输入”与“复选框”类型。

为多h 2一个属性

属性	要求?	运行时表达式?	描述
accesskey	假	真正的	HTML标准属性
cssClass	假	真正的	相当于“类”——HTML可选属性
cssErrorClass	假	真正的	相当于“类”——HTML可选属性。 当绑定使用字段有错误。
cssStyle	假	真正的	相当于“风格”——HTML可选属性
分隔符	假	真正的	分隔符使用每个标签之间的“输入”与类型”复选框”。 默认是没有分隔符。
dir	假	真正的	HTML标准属性
禁用	假	真正的	HTML可选属性。 设置此属性的值到“真正的”(没有引号)将禁用的HTML元素。
元素	假	真正的	指定的HTML元素,用于附上每个“输入”标记“复选框”型。 默认为“跨越”。
htmlEscape	假	真正的	启用/禁用呈现HTML转义的价值观。
id	假	真正的	HTML标准属性
itemLabel	假	真正的	值的一部分显示“输入”标签“复选框”型
物品	真正的	真正的	收集、地图或对象数组用于生成“输入”标签“复选框”型
itemValue	假	真正的	属性名称映射到“价值”属性的“输入”标签“复选框”型
朗	假	真正的	HTML标准属性
onblur	假	真正的	HTML事件属性
onchange	假	真正的	HTML事件属性
onclick	假	真正的	HTML事件属性
ondblclick	假	真正的	HTML事件属性
得到焦点	假	真正的	HTML事件属性
onkeydown	假	真正的	HTML事件属性
onkeypress	假	真正的	HTML事件属性
onkeyup	假	真正的	HTML事件属性
onmousedown	假	真正的	HTML事件属性
onmousemove	假	真正的	HTML事件属性
漫	假	真正的	HTML事件属性

滑	假	真正的	HTML事件属性
onmouseover	假	真正的	HTML事件属性
onmouseup	假	真正的	HTML事件属性
路径	真正的	真正的	路径属性数据绑定
tabindex	假	真正的	HTML标准属性
标题	假	真正的	HTML标准属性

H.4A了 错误 标签

在一个HTML呈现字段错误“跨”标签。

h 3一个属性为多

属性	要求?	运行时表达式?	描述
cssClass	假	真正的	相当于“类”——HTML可选属性
cssStyle	假	真正的	相当于“风格”——HTML可选属性
分隔符	假	真正的	分隔符显示多个错误消息。 默认为br标记。
dir	假	真正的	HTML标准属性
元素	假	真正的	指定的HTML元素,用于呈现封闭的错误。
htmlEscape	假	真正的	启用/禁用呈现HTML转义的价值观。
id	假	真正的	HTML标准属性
朗	假	真正的	HTML标准属性
onclick	假	真正的	HTML事件属性
ondblclick	假	真正的	HTML事件属性
onkeydown	假	真正的	HTML事件属性
onkeypress	假	真正的	HTML事件属性
onkeyup	假	真正的	HTML事件属性
onmousedown	假	真正的	HTML事件属性
onmousemove	假	真正的	HTML事件属性
滑	假	真正的	HTML事件属性
onmouseover	假	真正的	HTML事件属性
onmouseup	假	真正的	HTML事件属性
路径	假	真正的	路径错误对象数据绑定
tabindex	假	真正的	HTML标准属性
标题	假	真正的	HTML标准属性

H.5A了 形式 标签

呈现一个HTML的形式”标签,并公开一个绑定路径内标签绑定。

为多h 4一个属性

属性	要求?	运行时表达式?	描述
acceptCharset	假	真正的	指定字符编码的列表为输入数据,所接受的服务器处理这种形式。 该值是一个空间——和/或用逗号分隔的值列表的字符集。 客户端必须解释这个列表作为异列表,即。 ,服务器能够接受任何单一字符编码每个实体收到。
行动	假	真正的	需要HTML属性
commandName	假	真正的	模型属性的名称下表单对象暴露。 默认为“命令”。
cssClass	假	真正的	相当于“类”——HTML可选属性
cssStyle	假	真正的	相当于“风格”——HTML可选属性
dir	假	真正的	HTML标准属性
enctype	假	真正的	HTML可选属性
htmlEscape	假	真正的	启用/禁用呈现HTML转义的价值观。
id	假	真正的	HTML标准属性
朗	假	真正的	HTML标准属性
方法	假	真正的	HTML可选属性
ModelAttribute	假	真正的	模型属性的名称下表单对象暴露。 默认为“命令”。
名称	假	真正的	HTML标准属性——添加向后兼容情况下
onclick	假	真正的	HTML事件属性
ondblclick	假	真正的	HTML事件属性
onkeydown	假	真正的	HTML事件属性
onkeypress	假	真正的	HTML事件属性
onkeyup	假	真正的	HTML事件属性
onmousedown	假	真正的	HTML事件属性
onmousemove	假	真正的	HTML事件属性
滑	假	真正	HTML事件属性

		的	
onmouseover	假	真正的	HTML事件属性
onmouseup	假	真正的	HTML事件属性
onreset	假	真正的	HTML事件属性
onsubmit	假	真正的	HTML事件属性
目标	假	真正的	HTML可选属性
标题	假	真正的	HTML标准属性

H.6A了 隐藏 标签

呈现一个HTML的输入的标记类型隐含使用绑定值。

h 5为多属性

属性	要求?	运行时表达式?	描述
htmlEscape	假	真正的	启用/禁用呈现HTML转义的价值观。
id	假	真正的	HTML标准属性
路径	真正的	真正的	路径属性数据绑定

H.7A了 输入 标签

呈现一个HTML“输入”与“文本”的标签类型使用绑定值。

为多h 6一个属性

属性	要求?	运行时表达式?	描述
accesskey	假	真正的	HTML标准属性
alt	假	真正的	HTML可选属性
自动完成	假	真正的	常见的可选属性
cssClass	假	真正的	相当于“类”——HTML可选属性
cssErrorClass	假	真正的	相当于“类”——HTML可选属性。 当绑定使用字段有错误。
cssStyle	假	真正的	相当于“风格”——HTML可选属性
dir	假	真正的	HTML标准属性
禁用	假	真正的	HTML可选属性。 设置此属性的值到“真正的”(没有引号)将禁用的HTML元素。
htmlEscape	假	真正的	启用/禁用呈现HTML转义的价值观。
id	假	真正的	HTML标准属性

朗	假	真正的	HTML标准属性
最大长度	假	真正的	HTML可选属性
onblur	假	真正的	HTML事件属性
onchange	假	真正的	HTML事件属性
onclick	假	真正的	HTML事件属性
ondblclick	假	真正的	HTML事件属性
得到焦点	假	真正的	HTML事件属性
onkeydown	假	真正的	HTML事件属性
onkeypress	假	真正的	HTML事件属性
onkeyup	假	真正的	HTML事件属性
onmousedown	假	真正的	HTML事件属性
onmousemove	假	真正的	HTML事件属性
滑	假	真正的	HTML事件属性
onmouseover	假	真正的	HTML事件属性
onmouseup	假	真正的	HTML事件属性
onselect	假	真正的	HTML事件属性
路径	真正的	真正的	路径属性数据绑定
readOnly	假	真正的	HTML可选属性。 设置此属性的值到“真正的”(没有引号)将使HTML元素只读的。
大小	假	真正的	HTML可选属性
tabindex	假	真正的	HTML标准属性
标题	假	真正的	HTML标准属性

H.8A了 标签 标签

呈现一个表单字段标签在一个HTML标签的标签。

为多h 7一个属性

属性	要求?	运行时表达式?	描述
cssClass	假	真正的	相当于“类”——HTML可选属性。
cssErrorClass	假	真正的	相当于“类”——HTML可选属性。 只用当错误出现。
cssStyle	假	真正的	相当于“风格”——HTML可选属性
dir	假	真正的	HTML标准属性
对于	假	真正的	HTML标准属性
htmlEscape	假	真正的	启用/禁用呈现HTML转义的价值观。
id	假	真正的	HTML标准属性
朗	假	真正的	HTML标准属性
onclick	假	真正的	HTML事件属性
ondblclick	假	真正的	HTML事件属性
onkeydown	假	真正的	HTML事件属性

onkeydown	假	真正的	HTML事件属性
onkeypress	假	真正的	HTML事件属性
onkeyup	假	真正的	HTML事件属性
onmousedown	假	真正的	HTML事件属性
onmousemove	假	真正的	HTML事件属性
滑	假	真正的	HTML事件属性
onmouseover	假	真正的	HTML事件属性
onmouseup	假	真正的	HTML事件属性
路径	真正的	真正的	路径错误对象数据绑定
tabindex	假	真正的	HTML标准属性
标题	假	真正的	HTML标准属性

H.9A了 选项 标签

呈现一个HTML '选项'。 设置”选择“在适当的基于绑定值。

为多h 8一个属性

属性	要求?	运行时表达式?	描述
cssClass	假	真正的	相当于“类”——HTML可选属性
cssErrorClass	假	真正的	相当于“类”——HTML可选属性。 当绑定使用字段有错误。
cssStyle	假	真正的	相当于“风格”——HTML可选属性
dir	假	真正的	HTML标准属性
禁用	假	真正的	HTML可选属性。 设置此属性的值到“真正的”(没有引号)将禁用的HTML元素。
htmlEscape	假	真正的	启用/禁用呈现HTML转义的价值观。
id	假	真正的	HTML标准属性
标签	假	真正的	HTML可选属性
朗	假	真正的	HTML标准属性
onclick	假	真正的	HTML事件属性
ondblclick	假	真正的	HTML事件属性
onkeydown	假	真正的	HTML事件属性
onkeypress	假	真正的	HTML事件属性
onkeyup	假	真正的	HTML事件属性
onmousedown	假	真正的	HTML事件属性
onmousemove	假	真正的	HTML事件属性
滑	假	真正的	HTML事件属性
onmouseover	假	真正的	HTML事件属性
onmouseup	假	真正的	HTML事件属性
tabindex	假	真正的	HTML标准属性
标题	假	真正的	HTML标准属性

价值	真正的	真正的	HTML可选属性

H.10A了 选项 标签

呈现的HTML列表”选项”标签。 设置”选择“在适当的基于绑定值。

为多h 9一个属性

属性	要求?	运行时表达式?	描述
cssClass	假	真正的	相当于“类”——HTML可选属性
cssErrorClass	假	真正的	相当于“类”——HTML可选属性。 当绑定使用字段有错误。
cssStyle	假	真正的	相当于“风格”——HTML可选属性
dir	假	真正的	HTML标准属性
禁用	假	真正的	HTML可选属性。 设置此属性的值到“真正的”(没有引号)将禁用的HTML元素。
htmlEscape	假	真正的	启用/禁用呈现HTML转义的价值观。
id	假	真正的	HTML标准属性
itemLabel	假	真正的	属性名称映射到内部文本的“选项”标签
物品	真正的	真正的	收集、地图或对象数组用于生成内部“选项”标签
itemValue	假	真正的	属性名称映射到“价值”属性的“选项”标签
朗	假	真正的	HTML标准属性
onclick	假	真正的	HTML事件属性
ondblclick	假	真正的	HTML事件属性
onkeydown	假	真正的	HTML事件属性
onkeypress	假	真正的	HTML事件属性
onkeyup	假	真正的	HTML事件属性
onmousedown	假	真正的	HTML事件属性
onmousemove	假	真正的	HTML事件属性
滑	假	真正的	HTML事件属性
onmouseover	假	真正的	HTML事件属性
onmouseup	假	真正的	HTML事件属性
tabindex	假	真正的	HTML标准属性
标题	假	真正的	HTML标准属性

H.11A了 密码 标签

呈现一个HTML的输入的标记类型“密码”,使用绑定值。

为多h 10一个属性

属性	要求?	运行时表达式?	描述
accesskey	假	真正的	HTML标准属性
alt	假	真正的	HTML可选属性
自动完成	假	真正的	常见的可选属性
cssClass	假	真正的	相当于“类”——HTML可选属性
cssErrorClass	假	真正的	相当于“类”——HTML可选属性。 当绑定使用字段有错误。
cssStyle	假	真正的	相当于“风格”——HTML可选属性
dir	假	真正的	HTML标准属性
禁用	假	真正的	HTML可选属性。 设置此属性的值到“真正的”(没有引号)将禁用的HTML元素。
htmlEscape	假	真正的	启用/禁用呈现HTML转义的价值观。
id	假	真正的	HTML标准属性
朗	假	真正的	HTML标准属性
最大长度	假	真正的	HTML可选属性
onblur	假	真正的	HTML事件属性
onchange	假	真正的	HTML事件属性
onclick	假	真正的	HTML事件属性
ondblclick	假	真正的	HTML事件属性
得到焦点	假	真正的	HTML事件属性
onkeydown	假	真正的	HTML事件属性
onkeypress	假	真正的	HTML事件属性
onkeyup	假	真正的	HTML事件属性
onmousedown	假	真正的	HTML事件属性
onmousemove	假	真正的	HTML事件属性
滑	假	真正的	HTML事件属性
onmouseover	假	真正的	HTML事件属性
onmouseup	假	真正的	HTML事件属性
onselect	假	真正的	HTML事件属性
路径	真正的	真正的	路径属性数据绑定
readOnly	假	真正的	HTML可选属性。 设置此属性的值到“真正的”(没有引号)将使HTML元素只读的。
showPassword	假	真正的	是密码值被显示吗? 默认值为假。
大小	假	真正的	HTML可选属性
tabindex	假	真正的	HTML标准属性
标题	假	真正的	HTML标准属性

H.12A了 radiobutton 标签

呈现一个HTML的输入“标签”广播”型。

为多h 11一个属性

属性	要求?	运行时表达式?	描述
accesskey	假	真正的	HTML标准属性
cssClass	假	真正的	相当于“类”——HTML可选属性
cssErrorClass	假	真正的	相当于“类”——HTML可选属性。 当绑定使用字段有错误。
cssStyle	假	真正的	相当于“风格”——HTML可选属性
dir	假	真正的	HTML标准属性
禁用	假	真正的	HTML可选属性。 设置此属性的值到“真正的”(没有引号)将禁用的HTML元素。
htmlEscape	假	真正的	启用/禁用呈现HTML转义的价值观。
id	假	真正的	HTML标准属性
标签	假	真正的	值显示为标签的一部分
朗	假	真正的	HTML标准属性
onblur	假	真正的	HTML事件属性
onchange	假	真正的	HTML事件属性
onclick	假	真正的	HTML事件属性
ondblclick	假	真正的	HTML事件属性
得到焦点	假	真正的	HTML事件属性
onkeydown	假	真正的	HTML事件属性
onkeypress	假	真正的	HTML事件属性
onkeyup	假	真正的	HTML事件属性
onmousedown	假	真正的	HTML事件属性
onmousemove	假	真正的	HTML事件属性
滑	假	真正的	HTML事件属性
onmouseover	假	真正的	HTML事件属性
onmouseup	假	真正的	HTML事件属性
路径	真正的	真正的	路径属性数据绑定
tabindex	假	真正的	HTML标准属性
标题	假	真正的	HTML标准属性
价值	假	真正的	HTML可选属性

H.13A了 radiobuttons 标签

呈现多个HTML标记“输入”与“无线电”类型。

12 h为多属性

属性	要求?	运行时表达式?	描述
accesskey	假	真正的	HTML标准属性

cssClass	假	真正的	相当于“类”——HTML可选属性
cssErrorClass	假	真正的	相当于“类”——HTML可选属性。 当绑定使用字段有错误。
cssStyle	假	真正的	相当于“风格”——HTML可选属性
分隔符	假	真正的	分隔符使用每个标签之间的“输入”与“无线电”类型。 默认是没有分隔符。
dir	假	真正的	HTML标准属性
禁用	假	真正的	HTML可选属性。 设置此属性的值到“真正的”(没有引号)将禁用的HTML元素。
元素	假	真正的	指定的HTML元素,用于附上每个“输入”标签的无线电”型。 默认为“跨越”。
htmlEscape	假	真正的	启用/禁用呈现HTML转义的价值观。
id	假	真正的	HTML标准属性
itemLabel	假	真正的	值的一部分显示“输入”与“无线电”标签类型
物品	真正的	真正的	收集、地图或对象数组用于生成“输入”与“无线电”标签类型
itemValue	假	真正的	属性名称映射到“价值”属性的“输入”与“无线电”标签类型
朗	假	真正的	HTML标准属性
onblur	假	真正的	HTML事件属性
onchange	假	真正的	HTML事件属性
onclick	假	真正的	HTML事件属性
ondblclick	假	真正的	HTML事件属性
得到焦点	假	真正的	HTML事件属性
onkeydown	假	真正的	HTML事件属性
onkeypress	假	真正的	HTML事件属性
onkeyup	假	真正的	HTML事件属性
onmousedown	假	真正的	HTML事件属性
onmousemove	假	真正的	HTML事件属性
滑	假	真正的	HTML事件属性
onmouseover	假	真正的	HTML事件属性
onmouseup	假	真正的	HTML事件属性
路径	真正的	真正的	路径属性数据绑定
tabindex	假	真正的	HTML标准属性
标题	假	真正的	HTML标准属性

H.14A了 选择 标签

呈现一个HTML”选择“元素。 支持数据绑定到所选择的选项。

为多h 13一个属性

属性	要求?	运行时表达	描述

		式?	
accesskey	假	真正的	HTML标准属性
cssClass	假	真正的	相当于“类”——HTML可选属性
cssErrorClass	假	真正的	相当于“类”——HTML可选属性。 当绑定使用字段有错误。
cssStyle	假	真正的	相当于“风格”——HTML可选属性
dir	假	真正的	HTML标准属性
禁用	假	真正的	HTML可选属性。 设置此属性的值到“真正的”(没有引号)将禁用的HTML元素。
htmlEscape	假	真正的	启用/禁用呈现HTML转义的价值观。
id	假	真正的	HTML标准属性
itemLabel	假	真正的	属性名称映射到内部文本的“选项”标签
物品	假	真正的	收集、地图或对象数组用于生成内部“选项”标签
itemValue	假	真正的	属性名称映射到“价值”属性的“选项”标签
朗	假	真正的	HTML标准属性
多个	假	真正的	HTML可选属性
onblur	假	真正的	HTML事件属性
onchange	假	真正的	HTML事件属性
onclick	假	真正的	HTML事件属性
ondblclick	假	真正的	HTML事件属性
得到焦点	假	真正的	HTML事件属性
onkeydown	假	真正的	HTML事件属性
onkeypress	假	真正的	HTML事件属性
onkeyup	假	真正的	HTML事件属性
onmousedown	假	真正的	HTML事件属性
onmousemove	假	真正的	HTML事件属性
滑	假	真正的	HTML事件属性
onmouseover	假	真正的	HTML事件属性
onmouseup	假	真正的	HTML事件属性
路径	真正的	真正的	路径属性数据绑定
大小	假	真正的	HTML可选属性
tabindex	假	真正的	HTML标准属性
标题	假	真正的	HTML标准属性

H.15A了 Textarea 标签

呈现一个HTML的文本区”。

为多h 14一个属性

属性	要求?	运行时表达	描述

		式?	
accesskey	假	真正的	HTML标准属性
关口	假	真正的	需要HTML属性
cssClass	假	真正的	相当于“类”——HTML可选属性
cssErrorClass	假	真正的	相当于“类”——HTML可选属性。 当绑定使用字段有错误。
cssStyle	假	真正的	相当于“风格”——HTML可选属性
dir	假	真正的	HTML标准属性
禁用	假	真正的	HTML可选属性。 设置此属性的值到“真正的”(没有引号)将禁用的HTML元素。
htmlEscape	假	真正的	启用/禁用呈现HTML转义的价值观。
id	假	真正的	HTML标准属性
朗	假	真正的	HTML标准属性
onblur	假	真正的	HTML事件属性
onchange	假	真正的	HTML事件属性
onclick	假	真正的	HTML事件属性
ondblclick	假	真正的	HTML事件属性
得到焦点	假	真正的	HTML事件属性
onkeydown	假	真正的	HTML事件属性
onkeypress	假	真正的	HTML事件属性
onkeyup	假	真正的	HTML事件属性
onmousedown	假	真正的	HTML事件属性
onmousemove	假	真正的	HTML事件属性
滑	假	真正的	HTML事件属性
onmouseover	假	真正的	HTML事件属性
onmouseup	假	真正的	HTML事件属性
onselect	假	真正的	HTML事件属性
路径	真正的	真正的	路径属性数据绑定
readOnly	假	真正的	HTML可选属性。 设置此属性的值到“真正的”(没有引号)将使HTML元素只读的。
行	假	真正的	需要HTML属性
tabindex	假	真正的	HTML标准属性
标题	假	真正的	HTML标准属性