

# Table of Contents

Introduction	1.1
Part I. Spring框架概览	1.2
1.Spring入门	1.2.1
2.Spring框架简介	1.2.2
2.1 依赖注入和控制反转	1.2.2.1
2.2 Modules	1.2.2.2
2.2.1 Core Container	1.2.2.2.1
2.2.2 AOP and Instrumentation	1.2.2.2.2
2.2.3 Messaging	1.2.2.2.3
2.2.4 Data Access\Integration 数据访问\集成	1.2.2.2.4
2.2.5 Web	1.2.2.2.5
2.2.6 Test	1.2.2.2.6
2.3 Usage scenarios 使用场景	1.2.2.3
2.3.1 Dependency Management and Naming Conventions 依赖关系管理和命名约定	1.2.2.3.1
2.3.2 Logging 日志	1.2.2.3.2
Part II. Core Technologies核心技术	1.3
3.The IoC container	1.3.1
3.1 Spring IoC容器和bean的简介	1.3.1.1
3.2 容器概述	1.3.1.2
3.2.1 配置元数据	1.3.1.2.1
3.2.2 实例化容器	1.3.1.2.2
3.2.3 使用容器	1.3.1.2.3
3.3 Bean概述	1.3.1.3
3.3.1 命名bean	1.3.1.3.1
3.3.2 Instantiating beans	1.3.1.3.2
3.4 Dependencies	1.3.1.4
3.4.1 Dependency Injection	1.3.1.4.1
3.4.2 Dependencies and configuration in detail	1.3.1.4.2
3.4.3 Using depends-on	1.3.1.4.3

3.4.4 Lazy-initialized beans	1.3.1.4.4
3.4.5 Autowiring collaborators	1.3.1.4.5
3.4.6 Method injection	1.3.1.4.6
3.5 bean作用域	1.3.1.5
3.5.1 单例作用域	1.3.1.5.1
3.5.2 The prototype 作用域	1.3.1.5.2
3.5.3 Singleton beans with prototype-bean dependencies	1.3.1.5.3
3.5.4 Request, session, application, and WebSocket scopes	1.3.1.5.4
3.5.5 自定义作用域	1.3.1.5.5
3.6 Customizing the nature of a bean	1.3.1.6
3.6.1 生命周期回调函数	1.3.1.6.1
3.6.2 ApplicationContextAware和BeanNameAware	1.3.1.6.2
3.6.3 Other Aware interfaces	1.3.1.6.3
3.7 Spring Bean的继承	1.3.1.7
3.8 容器扩展点	1.3.1.8
3.8.1 使用BeanPostProcessor定制bean	1.3.1.8.1
3.8.2 使用BeanFactoryPostProcessor定制配置元数据	1.3.1.8.2
3.8.3 使用FactoryBean定制实例化逻辑	1.3.1.8.3
3.9 基于注解的容器配置	1.3.1.9
3.9.1 @Required	1.3.1.9.1
3.9.2 @Autowired	1.3.1.9.2
3.9.3 使用@Primary微调基于注解的自动装配	1.3.1.9.3
3.9.4 使用@Qualifier限定符微调基于注解的自动装配	1.3.1.9.4
3.9.5 使用泛型作为自动注入的限定符	1.3.1.9.5
3.9.6 CustomAutowireConfigurer	1.3.1.9.6
3.9.7 @Resource	1.3.1.9.7
3.9.8 @PostConstruct and @PreDestroy	1.3.1.9.8
3.10 ClassPath扫描和管理组件	1.3.1.10
3.10.1 @Component和各代码层注解	1.3.1.10.1
3.10.2 Meta-annotations 元注解	1.3.1.10.2
3.10.3 自动检测类并注册bean定义	1.3.1.10.3
3.10.4 使用过滤器自定义扫描	1.3.1.10.4
3.10.5 在组件中定义Bean元数据	1.3.1.10.5
3.10.6 命名自动注册组件	1.3.1.10.6

3.10.7 为`component-scan`组件提供作用域	1.3.1.10.7
3.10.8 为注解提供标示符Qualifier	1.3.1.10.8
<b>3.11 使用JSR-330标准注解</b>	<b>1.3.1.11</b>
3.11.1 使用@Inject @Name依赖注入	1.3.1.11.1
3.11.2 @Named:相当于@Component	1.3.1.11.2
3.11.3 JSR-330标准注解的限制	1.3.1.11.3
<b>3.12 基于Java的容器配置</b>	<b>1.3.1.12</b>
3.12.1 基本概念：@Bean和@Configuration	1.3.1.12.1
3.12.2 使用AnnotationConfigApplicationContext实例化Spring容器	
3.12.3 使用@Bean注解	1.3.1.12.3 1.3.1.12.2
3.12.4 使用@Configuration注解	1.3.1.12.4
3.12.5 组合Java基本配置	1.3.1.12.5
<b>3.13 Environment抽象</b>	<b>1.3.1.13</b>
3.13.1 bean定义profiles	1.3.1.13.1
3.13.2 XML bean定义profile	1.3.1.13.2
3.13.3 PropertySource abstraction	1.3.1.13.3
3.13.4 @PropertySource	1.3.1.13.4
3.13.5 Placeholder resolution in statements	1.3.1.13.5
<b>3.14 注册LoadTimeWeaver</b>	<b>1.3.1.14</b>
<b>3.15 ApplicationContext 的附加功能</b>	<b>1.3.1.15</b>
3.15.1 Internationalization using MessageSource	1.3.1.15.1
3.15.2 Standard and Custom Events	1.3.1.15.2
3.15.3 通过便捷的方式访问底层资源	1.3.1.15.3
3.15.4 快速对web应用的`ApplicationContext`实例化	1.3.1.15.4
3.15.5 将Spring ApplicationContext作为JAVA EE RAR文件部署	
<b>3.16 The BeanFactory</b>	<b>1.3.1.16 1.3.1.15.5</b>
3.16.1 BeanFactory or ApplicationContext?	1.3.1.16.1
3.16.2 Glue code and the evil singleton	1.3.1.16.2
<b>4.Resources</b>	<b>1.3.2</b>
4.1 介绍	1.3.2.1
4.2 Resource 接口	1.3.2.2
4.3 内置的 Resource 实现	1.3.2.3
4.3.1 UrlResource	1.3.2.3.1

4.3.2 ClassPathResource	1.3.2.3.2
4.3.3 FileSystemResource	1.3.2.3.3
4.3.4 ServletContextResource	1.3.2.3.4
4.3.5 InputStreamResource	1.3.2.3.5
4.3.6 ByteArrayResource	1.3.2.3.6
4.4 ResourceLoader 接口	1.3.2.4
4.5 ResourceLoaderAware 接口	1.3.2.5
4.6 Resources as dependencies	1.3.2.6
4.7 应用上下文和资源路径	1.3.2.7
4.7.1 构造应用上下文	1.3.2.7.1
4.7.2 使用通配符构造应用上下文	1.3.2.7.2
4.7.3 FileSystemResource 警告	1.3.2.7.3
5. 数据校验、数据绑定和类型转换	1.3.3
5.1 简介	1.3.3.1
5.2 使用 Spring 的 Validator 接口来进行数据校验	1.3.3.2
5.3 通过错误编码得到错误信息	1.3.3.3
5.4 Bean 操作和 BeanWrapper	1.3.3.4
5.4.1 设置并获取基本和嵌套属性	1.3.3.4.1
5.4.2 内置 PropertyEditor 实现	1.3.3.4.2
5.5 Spring Type Conversion	1.3.3.5
5.6 Spring Field Formatting	1.3.3.6
5.7 Configuring a global date & time format	1.3.3.7
5.8 Spring Validation	1.3.3.8
6. Spring 表达式语言 (SpEL)	1.3.4
6.1 介绍	1.3.4.1
6.2 功能概述	1.3.4.2
6.3 使用 Spring 表达式接口的表达式运算操作	1.3.4.3
6.4 Expression support for defining bean definitions	1.3.4.4
6.5 Language Reference	1.3.4.5
6.6 Classes used in the examples	1.3.4.6
7. Aspect Oriented Programming with Spring	1.3.5
7.1 Introduction	1.3.5.1
7.2 @AspectJ support	1.3.5.2
7.3 Schema-based AOP support	1.3.5.3

7.4 Choosing which AOP declaration style to use	1.3.5.4
7.5 Mixing aspect types	1.3.5.5
7.6 Proxying mechanisms	1.3.5.6
7.7 Programmatic creation of @AspectJ Proxies	1.3.5.7
7.8 Using AspectJ with Spring applications	1.3.5.8
7.9 Further Resources	1.3.5.9
<b>8. Spring AOP APIs</b>	<b>1.3.6</b>
8.1 Introduction	1.3.6.1
8.2 Pointcut API in Spring	1.3.6.2
8.3 Advice API in Spring	1.3.6.3
<b>Part III. 测试</b>	<b>1.4</b>
<b>9. Spring 框架下的测试</b>	<b>1.4.1</b>
<b>10. 单元测试</b>	<b>1.4.2</b>
<b>11. 集成测试</b>	<b>1.4.3</b>
11.1 概述	1.4.3.1
11.2 集成测试的目标	1.4.3.2
11.3 JDBC 测试支持	1.4.3.3
11.4 注解	1.4.3.4
11.5 Spring TestContext Framework	1.4.3.5
11.6 Spring MVC Test Framework	1.4.3.6
11.6.1 Server-Side Tests	1.4.3.6.1
11.6.2 HtmlUnit Integration	1.4.3.6.2
11.6.3 Client-Side REST Tests	1.4.3.6.3
11.7 PetClinic Example	1.4.3.7
<b>12. Further Resources</b>	<b>1.4.4</b>
<b>Part IV. 数据访问</b>	<b>1.5</b>
<b>13. Transaction Management</b>	<b>1.5.1</b>
13.1 Introduction to Spring Framework transaction management	1.5.1.1
13.2 Advantages of the Spring Framework's transaction support model	
13.2.1 Global transactions	1.5.1.2.1
13.2.2 Local transactions	1.5.1.2.2
13.2.3 Spring Framework's consistent programming model	1.5.1.2.3
13.3 Understanding the Spring Framework transaction abstraction	1.5.1.3

13.4 Synchronizing resources with transactions	1.5.1.4
13.4.1 High-level synchronization approach	1.5.1.4.1
13.4.2 Low-level synchronization approach	1.5.1.4.2
13.4.3 TransactionAwareDataSourceProxy	1.5.1.4.3
13.5 Declarative transaction management	1.5.1.5
13.5.1 Understanding the Spring Framework's declarative transaction implementation	1.5.1.5.1
13.5.2 Example of declarative transaction implementation	1.5.1.5.2
13.5.3 Rolling back a declarative transaction	1.5.1.5.3
13.5.4 Configuring different transactional semantics for different beans	
13.5.5 settings	1.5.1.5.5 1.5.1.5.4
13.5.6 Using @Transactional	1.5.1.5.6
13.5.7 Transaction propagation	1.5.1.5.7
13.5.8 Advising transactional operations	1.5.1.5.8
13.5.9 Using @Transactional with AspectJ	1.5.1.5.9
13.6.2 Using the PlatformTransactionManager	1.5.1.5.10
13.6 Programmatic transaction management	1.5.1.6
13.6.1 Using the TransactionTemplate	1.5.1.6.1
13.6.2 Using the PlatformTransactionManager	1.5.1.6.2
13.7 Choosing between programmatic and declarative transaction management	1.5.1.7
13.8 Transaction bound event	1.5.1.8
13.9 Application server-specific integration	1.5.1.9
13.9.1 IBM WebSphere	1.5.1.9.1
13.9.2 Oracle WebLogic Server	1.5.1.9.2
13.10 Solutions to common problems	1.5.1.10
13.10.1 Use of the wrong transaction manager for a specific DataSource	
13.11 Further Resources	1.5.1.11 1.5.1.10.1
14. DAO support	1.5.2
14.1 Introduction	1.5.2.1
14.2 Consistent exception hierarchy	1.5.2.2
14.3 Annotations used for configuring DAO or Repository classes	1.5.2.3
15. 使用 JDBC 实现数据访问	1.5.3
15.1 介绍 Spring JDBC 框架	1.5.3.1

15.1.1 选择一种JDBC数据库访问方法	1.5.3.1.1
15.1.2 包层级	1.5.3.1.2
15.2 使用JDBC核心类控制基础的JDBC处理过程和异常处理机制	1.5.3.2
15.2.1 JdbcTemplate	1.5.3.2.1
15.2.2 NamedParameterJdbcTemplate	1.5.3.2.2
15.2.3 SQLExceptionTranslator	1.5.3.2.3
15.2.4 执行SQL语句	1.5.3.2.4
15.2.5 运行查询	1.5.3.2.5
15.2.6 更新数据库	1.5.3.2.6
15.2.7 获取自增Key	1.5.3.2.7
15.3 控制数据库连接	1.5.3.3
15.3.1 DataSource	1.5.3.3.1
15.3.2 DataSourceUtils	1.5.3.3.2
15.3.3 SmartDataSource	1.5.3.3.3
15.3.4 AbstractDataSource	1.5.3.3.4
15.3.5 SingleConnectionDataSource	1.5.3.3.5
15.3.6 DriverManagerDataSource	1.5.3.3.6
15.3.7 TransactionAwareDataSourceProxy	1.5.3.3.7
15.3.8 DataSourceTransactionManager	1.5.3.3.8
15.4 JDBC批量操作	1.5.3.4
15.4.1 使用JdbcTemplate来进行基础的批量操作	1.5.3.4.1
15.4.2 对象列表的批量处理	1.5.3.4.2
15.4.3 多个批处理操作	1.5.3.4.3
15.5 利用SimpleJdbc类简化JDBC操作	1.5.3.5
15.5.1 利用SimpleJdbcInsert插入数据	1.5.3.5.1
15.5.2 使用SimpleJdbcInsert获取自增Key	1.5.3.5.2
15.5.3 使用SimpleJdbcInsert指定列	1.5.3.5.3
15.5.4 使用SqlParameterSource 提供参数值	1.5.3.5.4
15.5.5 利用SimpleJdbcCall调用存储过程	1.5.3.5.5
15.5.6 为SimpleJdbcCall显式定义参数	1.5.3.5.6
15.5.7 如何定义SqlParameters	1.5.3.5.7
15.5.8 使用SimpleJdbcCall调用内置存储函数	1.5.3.5.8
15.5.9 从SimpleJdbcCall返回ResultSet/REF游标	1.5.3.5.9
15.6 像Java对象那样操作JDBC	1.5.3.6

15.6.1 SqlQuery	1.5.3.6.1
15.6.2 MappingSqlQuery	1.5.3.6.2
15.6.3 SqlUpdate	1.5.3.6.3
15.6.4 StoredProcedure	1.5.3.6.4
15.7 参数和数据处理的常见问题	1.5.3.7
15.7.1 为参数设置SQL的类型信息	1.5.3.7.1
15.7.2 处理BLOB和CLOB对象	1.5.3.7.2
15.7.3 传入IN语句的列表值	1.5.3.7.3
15.7.4 处理存储过程调用的复杂类型	1.5.3.7.4
15.8 内嵌数据库支持	1.5.3.8
15.8.1 为什么使用一个内嵌数据库？	1.5.3.8.1
15.8.2 使用Spring配置来创建内嵌数据库	1.5.3.8.2
15.8.3 使用编程方式创建内嵌数据库	1.5.3.8.3
15.8.4 选择内嵌数据库的类型	1.5.3.8.4
15.8.5 使用内嵌数据库测试数据访问层逻辑	1.5.3.8.5
15.8.6 生成内嵌数据库的唯一名字	1.5.3.8.6
15.8.7 内嵌数据库扩展支持	1.5.3.8.7
15.9 初始化Datasource	1.5.3.9
15.9.1 使用Spring XML来初始化数据库	1.5.3.9.1
16.ORM和数据访问	1.5.4
16.1 介绍一下Spring中的ORM	1.5.4.1
16.2 集成ORM的注意事项	1.5.4.2
16.2.1 资源和事务管理	1.5.4.2.1
16.2.2 异常转义	1.5.4.2.2
16.3 Hibernate	1.5.4.3
16.3.1 在Spring容器中配置SessionFactory	1.5.4.3.1
16.3.2 基于Hibernate API来实现DAO	1.5.4.3.2
16.3.3 声明式事务划分	1.5.4.3.3
16.3.4 编程式事务划分	1.5.4.3.4
16.3.5 事务管理策略	1.5.4.3.5
16.3.6 对比由容器管理的和本地定义的资源	1.5.4.3.6
16.3.7 Hibernate的虚假应用服务器警告	1.5.4.3.7
16.4 JPA	1.5.4.4

16.4.1 Spring中JPA配置的三个选项	1.5.4.4.1
16.4.2 基于JPA的EntityManagerFactory和EntityManager来实现DAO	
16.4.3 Spring驱动的JPA事务	1.5.4.4.3 1.5.4.4.2
16.4.4 JpaDialect和JpaVendorAdapter	1.5.4.4.4
16.4.5 为JPA配置JTA事务管理	1.5.4.4.5
17.Marshalling XML using O/X Mappers	1.5.5
17.1 Introduction	1.5.5.1
17.1.1 Ease of configuration	1.5.5.1.1
17.1.2 Consistent Interfaces	1.5.5.1.2
17.1.3 Consistent Exception Hierarchy	1.5.5.1.3
17.2 Marshaller and Unmarshaller	1.5.5.2
17.2.1 Marshaller	1.5.5.2.1
17.2.2 Unmarshaller	1.5.5.2.2
17.2.3 XmlMappingException	1.5.5.2.3
17.3 Using Marshaller and Unmarshaller	1.5.5.3
17.4 XML Schema-based Configuration	1.5.5.4
17.5 JAXB	1.5.5.5
17.5.1 Jaxb2Marshaller	1.5.5.5.1
17.6 Castor	1.5.5.6
17.6.1 CastorMarshaller	1.5.5.6.1
17.6.2 Mapping	1.5.5.6.2
17.7 JiBX	1.5.5.7
17.7.1 JibxMarshaller	1.5.5.7.1
17.8 XStream	1.5.5.8
17.8.1 XStreamMarshaller	1.5.5.8.1
Part V. The Web	1.6
18.Web MVC 框架	1.6.1
18.1 Spring Web MVC框架的介绍	1.6.1.1
18.1.1 Spring Web MVC的特点	1.6.1.1.1
18.1.2 其他MVC实现的可插拔性	1.6.1.1.2
18.2 DispatcherServlet	1.6.1.2
18.2.1 WebApplicationContext中的特殊Bean类型	1.6.1.2.1
18.2.2 默认DispatcherServlet 配置	1.6.1.2.2
18.2.3 DispatcherServlet 处理序列	1.6.1.2.3

18.3 实现 Controllers	1.6.1.3
18.3.1 使用 @Controller 定义控制器	1.6.1.3.1
18.3.2 使用 @RequestMapping 映射请求	1.6.1.3.2
18.3.3 定义 @RequestMapping 处理方法	1.6.1.3.3
18.3.4 异步请求处理	1.6.1.3.4
18.3.5 测试控制器	1.6.1.3.5
18.4 处理程序映射	1.6.1.4
18.4.1 用 HandlerInterceptor 拦截请求	1.6.1.4.1
18.5 解决观点	1.6.1.5
18.5.1 使用 ViewResolver 界面解析视图	1.6.1.5.1
18.5.2 链接视图解析器	1.6.1.5.2
18.5.3 重定向到视图	1.6.1.5.3
18.5.4 ContentNegotiatingViewResolver	1.6.1.5.4
18.6 使用 flash 属性	1.6.1.6
18.7 构建 URI	1.6.1.7
18.7.1 构建控制器和方法的 URI	1.6.1.7.1
18.7.2 构建来自视图的控制器和方法的 URI	1.6.1.7.2
18.8 使用区域设置	1.6.1.8
18.8.1 获取时区信息	1.6.1.8.1
18.8.2 AcceptHeaderLocaleResolver	1.6.1.8.2
18.8.3 CookieLocaleResolver	1.6.1.8.3
18.8.4 SessionLocaleResolver	1.6.1.8.4
18.8.5 LocaleChangeInterceptor	1.6.1.8.5
18.9 使用主题	1.6.1.9
18.9.1 主题概述	1.6.1.9.1
18.9.2 定义主题	1.6.1.9.2
18.9.3 主题解析器	1.6.1.9.3
18.10 Spring 的多部分（文件上传）支持	1.6.1.10
18.10.1 介绍	1.6.1.10.1
18.10.2 与 Commons FileUpload 一起使用 MultipartResolver	1.6.1.10.2
18.10.3 在 Servlet 3.0 中使用 MultipartResolver	1.6.1.10.3
18.10.4 处理表单中的文件上传	1.6.1.10.4
18.10.5 处理来自编程客户端的文件上传请求	1.6.1.10.5

18.11 处理异常	1.6.1.11
18.11.1 HandlerExceptionResolver	1.6.1.11.1
18.11.2 @ExceptionHandler	1.6.1.11.2
18.11.3 处理标准的Spring MVC异常	1.6.1.11.3
18.11.4 REST控制器异常处理	1.6.1.11.4
18.11.5 使用@ResponseStatus注解业务异常	1.6.1.11.5
18.11.6 自定义默认Servlet容器错误页面	1.6.1.11.6
18.12 网络安全	1.6.1.12
18.13 关于配置支持的公约	1.6.1.13
18.13.1 Controller ControllerClassNameHandlerMapping	1.6.1.13.1
18.13.2 Model ModelMap (ModelAndView)	1.6.1.13.2
18.13.3 View - RequestToViewNameTranslator	1.6.1.13.3
18.14 HTTP缓存支持	1.6.1.14
18.14.1 缓存控制HTTP头	1.6.1.14.1
18.14.2 HTTP缓存支持静态资源	1.6.1.14.2
18.14.3 支持控制器中的Cache-Control，ETag和Last-Modified响应头	
18.14.4 浅层ETag支持	1.6.1.14.4 1.6.1.14.3
18.15 基于代码的Servlet容器初始化	1.6.1.15
18.16 配置Spring MVC	1.6.1.16
18.16.1 启用MVC Java Config或MVC XML命名空间	1.6.1.16.1
18.16.2 定制提供的配置	1.6.1.16.2
18.16.3 转换和格式化	1.6.1.16.3
18.16.4 验证	1.6.1.16.4
18.16.5 拦截器	1.6.1.16.5
18.16.6 内容谈判	1.6.1.16.6
18.16.7 视图控制器	1.6.1.16.7
18.16.8 查看解析器	1.6.1.16.8
18.16.9 服务于资源	1.6.1.16.9
18.16.10 回到“Default”Servlet服务资源	1.6.1.16.10
18.16.11 路径匹配	1.6.1.16.11
18.16.12 消息转换器	1.6.1.16.12
18.16.13 使用MVC Java配置进行高级自定义	1.6.1.16.13
18.16.14 使用MVC命名空间进行高级自定义	1.6.1.16.14
19.视图技术	1.6.2

19.1 简介	1.6.2.1
19.2 Thymeleaf	1.6.2.2
19.3 Groovy Markup Templates	1.6.2.3
19.3.1 配置	1.6.2.3.1
19.3.2 例子	1.6.2.3.2
19.4 FreeMarker	1.6.2.4
19.4.1 依赖	1.6.2.4.1
19.4.2 上下文配置	1.6.2.4.2
19.4.3 创建模板	1.6.2.4.3
19.4.4 高级FreeMarker配置	1.6.2.4.4
19.4.5 绑定支持和表单处理	1.6.2.4.5
19.5 JSP & JSTL	1.6.2.5
19.5.1 视图解析	1.6.2.5.1
19.5.2 'Plain-old' JSPs versus JSTL	1.6.2.5.2
19.5.3 Additional tags facilitating development	1.6.2.5.3
19.5.4 使用Spring的表单标签库	1.6.2.5.4
19.6 Script templates	1.6.2.6
19.6.1 Dependencies	1.6.2.6.1
19.6.2 How to integrate script based templating	1.6.2.6.2
19.7 XML Marshalling View	1.6.2.7
19.8 Tiles	1.6.2.8
19.8.1 Dependencies	1.6.2.8.1
19.8.2 How to integrate Tiles	1.6.2.8.2
19.9 XSLT	1.6.2.9
19.9.1 My First Words	1.6.2.9.1
19.10 Document views (PDF/Excel)	1.6.2.10
19.10.1 Introduction	1.6.2.10.1
19.10.2 Configuration and setup	1.6.2.10.2
19.11 Feed Views	1.6.2.11
19.12 JSON Mapping View	1.6.2.12
19.13 XML Mapping View	1.6.2.13



Chinese translation of Spring Framework 5.x Reference Documentation. The current version of Spring Framework 5.x is 5.0.0.M4. There is also a GitBook version of the book:  
<https://muyinchen.gitbooks.io/spring-framework-5-0-0-m3/content/> Github Read the address : <https://github.com/muyinchen/Spring-Framework-5.0.0.M3-CN/blob/master/SUMMARY.md>

《Spring Framework 5.x参考文档》中文翻译（包含了官方文档以及其他文章）。至今为止，Spring Framework 的最新版本为 5.0.0.M4

翻译自：

<http://docs.spring.io/spring/docs/5.0.0.M4/spring-framework-reference/htmlsingle/>

本人会在文档内做版本间的对比，请关注里面的注即可，对比文档版本 4.3.0.RELEASE

## 2017.2.15 :

第四章 Resources 翻译完毕

## 2017.2.12 :

终于在十六将第三章搞完

## 2017.1.1 :

对5.0.0.M3做升级版本到5.0.0.M4

## Contact 联系作者:

Blog:<https://muyinchen.github.io/>

email: [fei6751803@163.com](mailto:fei6751803@163.com)

## Introduction

### Github :

<https://github.com/muyinchen>

### Github 本书阅读地址：

<https://github.com/muyinchen/Spring-Framework-5.0.0.M3-CN/blob/master/SUMMARY.md>

**Gitbook** 阅读地址：

<https://muyinchen.gitbooks.io/spring-framework-5-0-0-m3/content/>

# Part I. Spring框架概览

The Spring Framework is a lightweight solution and a potential one-stop-shop for building your enterprise-ready applications. However, Spring is modular, allowing you to use only those parts that you need, without having to bring in the rest. You can use the IoC container, with any web framework on top, but you can also use only the [Hibernate integration code](#) or the [JDBC abstraction layer](#). The Spring Framework supports declarative transaction management, remote access to your logic through RMI or web services, and various options for persisting your data. It offers a full-featured [MVC framework](#), and enables you to integrate [AOP](#) transparently into your software.

Spring Framework 是一种轻量级的解决方案，是构建你的企业级应用程序的潜在一站式解决方案。尽管如此， Spring 是模块化的，允许你只使用你需要的那些部分，而不必引入其他的。你可以使用IoC容器，任何 Web 框架在顶部(只是底层用Spring框架，比如ssh，中间那层用了 Spring )，但你也可以只使用 Hibernate 集成代码或 JDBC 抽象层。 Spring 框架支持声明式事务管理，通过RMI或Web服务远程访问你的逻辑，以及用于持久存储数据的各种选项。它提供了一个全功能的 MVC 框架，并使你能够将 AOP 透明地集成到你的软件中。

Spring is designed to be non-intrusive, meaning that your domain logic code generally has no dependencies on the framework itself. In your integration layer (such as the data access layer), some dependencies on the data access technology and the Spring libraries will exist. However, it should be easy to isolate these dependencies from the rest of your code base.

Spring 设计为非侵入式的，这意味着你所写的逻辑代码通常没有对框架本身的依赖。在你的集中处理层（例如数据访问层）中，将存在对数据访问技术和Spring库的一些依赖。但是，应该很容易将这些依赖关系与其余代码库隔离开。

This document is a reference guide to Spring Framework features. If you have any requests, comments, or questions on this document, please post them on the [user mailing list](#). Questions on the Framework itself should be asked on StackOverflow (see <https://spring.io/questions>).

本文档是 Spring Framework 特性的参考指南。如果你对本文档有任何要求，意见或问题，请将其张贴在用户邮件列表中。框架本身的问题应该在 StackOverflow （请参阅 <https://spring.io/questions>）。

This reference guide provides detailed information about the Spring Framework. It provides comprehensive documentation for all features, as well as some background about the underlying concepts (such as "*Dependency Injection*") that Spring has embraced.

本参考指南提供了有关 Spring 框架的详细信息。它提供了所有功能的全面文档，以及 Spring 所拥有的基本概念（例如“依赖注入”）的一些背景知识。

If you are just getting started with Spring, you may want to begin using the Spring Framework by creating a [Spring Boot](#) based application. Spring Boot provides a quick (and opinionated) way to create a production-ready Spring based application. It is based on the Spring Framework, favors convention over configuration, and is designed to get you up and running as quickly as possible.

如果你刚刚开始使用 Spring，你可能想要通过创建一个基于 Spring Boot 的应用程序来开始使用 Spring Framework。Spring Boot 提供了一种快速（和建议性）的方式来创建基于生产环境的 Spring 应用程序。它是基于 Spring 框架，喜欢约定的配置，并且旨在让你快速启动和运行。

You can use [start.spring.io](http://start.spring.io) to generate a basic project or follow one of the "[Getting Started](#)" guides like the [Getting Started Building a RESTful Web Service](#) one. As well as being easier to digest, these guides are very *task focused*, and most of them are based on Spring Boot. They also cover other projects from the Spring portfolio that you might want to consider when solving a particular problem.

你可以使用 start.spring.io 来生成基本项目，或者按照“入门”指南之一，就像开始构建 RESTful Web 服务一样。除了更容易理解消化，这些指南是非常专注于任务，其中大部分都是基于 Spring Boot。它们还涵盖了 Spring 解决特定问题时可能需要考虑的 Spring 组合中的其他项目。

The Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications. Spring handles the infrastructure so you can focus on your application.

Spring Framework 是一个 Java 平台框架，它为开发 Java 应用程序的全面的基础架构支持。Spring 处理基础架构，以便你可以专注于应用程序。

Spring enables you to build applications from "plain old Java objects" (POJOs) and to apply enterprise services non-invasively to POJOs. This capability applies to the Java SE programming model and to full and partial Java EE.

Spring 使你能够从“普通旧 Java 对象”（POJO）构建应用程序，并将企业服务非侵入性地应用于 POJO。此功能适用于 Java SE 编程模型 和完整和部分 Java EE。

Examples of how you, as an application developer, can benefit from the Spring platform:

- Make a Java method execute in a database transaction without having to deal with transaction APIs.
- Make a local Java method a remote procedure without having to deal with remote APIs.
- Make a local Java method a management operation without having to deal with JMX APIs.
- Make a local Java method a message handler without having to deal with JMS APIs.

作为应用程序开发人员，你可以如何从 Spring 平台中受益的示例：

- 在数据库事务中执行 Java 方法，而不必处理事务 API。
- 使本地 Java 方法成为远程过程，而不必处理远程 API。（意思就是无须管相关的远程调用 api，Spring 已经封装好了，你自己只需要专注自己的逻辑即可，下同，不再解释）
- 使本地 Java 方法成为管理操作，而不必处理 JMX API。
- 使本地 Java 方法成为消息处理器，而不必处理 JMS API。

A Java application — a loose term that runs the gamut from constrained, embedded applications to n-tier, server-side enterprise applications — typically consists of objects that collaborate to form the application proper. Thus the objects in an application have *dependencies* on each other.

Java应用程序 - 一个宽松的术语，其运行范围从受限的嵌入式应用程序到n层，服务器端企业应用程序 - 通常由协作形成应用程序的对象组成。因此，在一个应用程序中的对象具有关于彼此的依赖性。

Although the Java platform provides a wealth of application development functionality, it lacks the means to organize the basic building blocks into a coherent whole, leaving that task to architects and developers. Although you can use design patterns such as *Factory*, *Abstract Factory*, *Builder*, *Decorator*, and *Service Locator* to compose the various classes and object instances that make up an application, these patterns are simply that: best practices given a name, with a description of what the pattern does, where to apply it, the problems it addresses, and so forth. Patterns are formalized best practices that *you must implement yourself* in your application.

尽管 Java 平台提供了丰富的应用程序开发功能，但它缺乏将基本构建块组织成一个整体的手段，将该任务留给架构师和开发人员。虽然可以使用设计模式，例如工厂，抽象工厂，建筑模式，装饰器和服务定位器等设计模式来组成构成应用程序的各种类和对象实例，这些模式是：最佳实践给出一个名字，有什么模式的描述，如何应用它，它解决的问题，等等。模式是形式化的最佳实践，你必须在应用程序中实现自己。

The Spring Framework *Inversion of Control* (IoC) component addresses this concern by providing a formalized means of composing disparate components into a fully working application ready for use. The Spring Framework codifies formalized design patterns as first-class objects that you can integrate into your own application(s). Numerous organizations and institutions use the Spring Framework in this manner to engineer robust, *maintainable* applications.

Spring框架控制反转（IoC）组件通过提供一种形式化的手段将不同的组件组成一个完全可用的应用程序来解决这个问题。Spring框架将形式化的设计模式编译为可以集成到自己的应用程序中的第一类对象。许多组织和机构以这种方式使用Spring框架来设计强大的，可维护的应用程序。（就是将各种设计模式融合到框架中，而开发人员只需要关注自己的逻辑，在不知不觉中就已经用到了最佳的设计模式，自己的代码也得以健壮）

## 背景

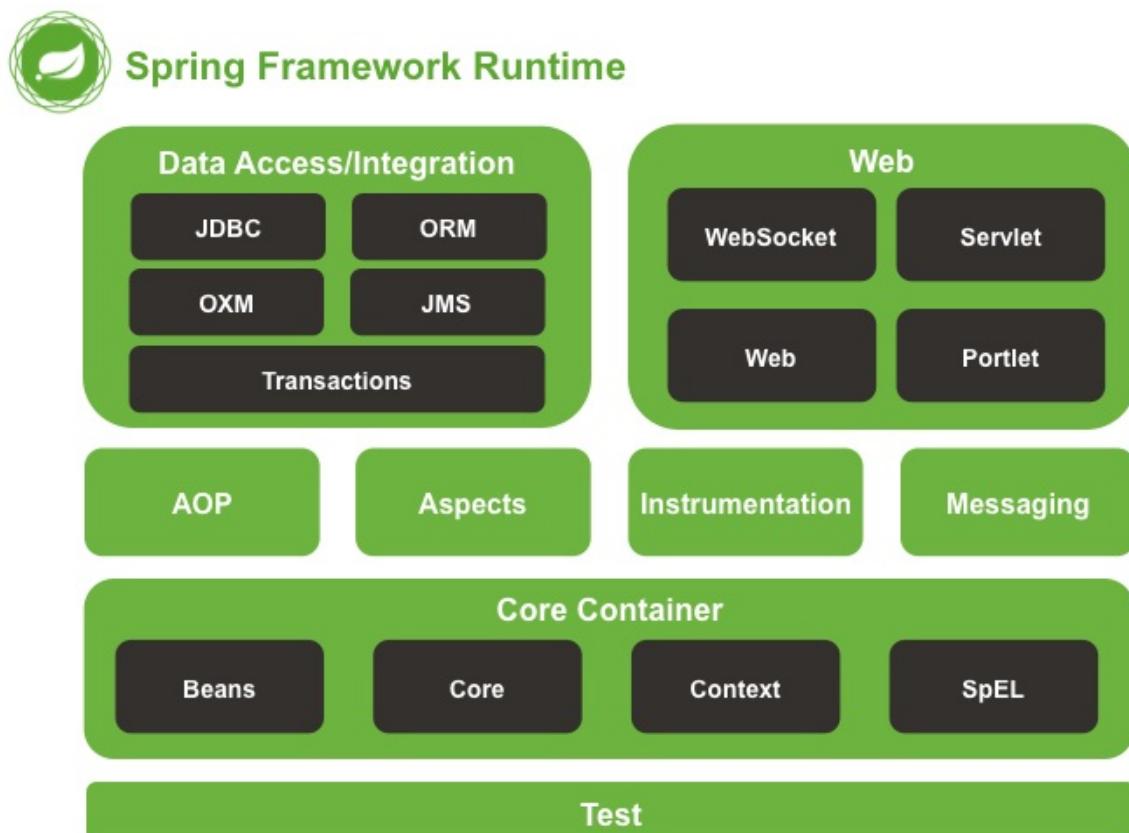
"*The question is, what aspect of control are [they] inverting?*" Martin Fowler posed this question about Inversion of Control (IoC) [on his site](#) in 2004. Fowler suggested renaming the principle to make it more self-explanatory and came up with *Dependency Injection*.

“现在的问题是，什么方面的控制是（他们）反转？”Martin Fowler在2004年在他的网站上提出了关于反转控制（IoC）的问题。Fowler建议重命名这个原则，使其更一目了然，并提出依赖注入。

The Spring Framework consists of features organized into about 20 modules. These modules are grouped into Core Container, Data Access/Integration, Web, AOP (Aspect Oriented Programming), Instrumentation, Messaging, and Test, as shown in the following diagram.

Spring框架由大约20个功能模块组成。这些模块分为核心容器，数据访问/集成，Web，AOP（面向方面的编程），仪器，消息传递和测试，如下图所示。

图2.1 Spring框架的概述



The following sections list the available modules for each feature along with their artifact names and the topics they cover. Artifact names correlate to *artifact IDs* used in [Dependency Management tools](#).

以下部分列出了每个功能的可用模块及其构件名称及其涵盖的主题。构件名称与构件关联id用于管理工具的依赖。

## 2.2.1 Core Container

The [Core Container](#) consists of the `spring-core`, `spring-beans`, `spring-context`, `spring-context-support`, and `spring-expression` (Spring Expression Language) modules.

The `spring-core` and `spring-beans` modules provide the fundamental parts of the framework, including the IoC and Dependency Injection features. The `BeanFactory` is a sophisticated implementation of the factory pattern. It removes the need for programmatic singletons and allows you to decouple the configuration and specification of dependencies from your actual program logic.

核心容器由 `spring-core`，`spring-beans`，`spring-context`，`spring-context-support` 和 `spring-expression` (Spring Expression Language) 模块 组成。

`spring-core` 和 `spring-beans` 模块提供了框架的基本部分，包括 IoC 和依赖注入功能。  
`BeanFactory` 是一个复杂的工厂模式的实现。它消除了编程单例的需要(程序员不必对单例亲力亲为)，并允许你从实际的程序逻辑中分离依赖性的配置和规范。

The [Context](#) (`spring-context`) module builds on the solid base provided by the [Core and Beans](#) modules: it is a means to access objects in a framework-style manner that is similar to a JNDI registry. The Context module inherits its features from the Beans module and adds support for internationalization (using, for example, resource bundles), event propagation, resource loading, and the transparent creation of contexts by, for example, a Servlet container. The Context module also supports Java EE features such as EJB, JMX, and basic remoting. The `ApplicationContext` interface is the focal point of the Context module.

`spring-context-support` provides support for integrating common third-party libraries into a Spring application context, in particular for caching (EhCache, JCache) and scheduling (CommonJ, Quartz).

`Context` (`spring-context`) 模块建立在 `Core` 和 `Beans` 模块 提供的实体基础之上：它是一种以类似于 `JNDI` 注册表 的框架样式方式访问对象的手段。`Context` 模块 从 `Beans` 模块 继承其特性，并增加了对国际化（例如使用资源束），事件传播，资源加载以及通过例如 `Servlet` 容器 的透明创建上下文的支持。`Context` 模块 还支持 Java EE 功能，如 `EJB`，`JMX` 和基本远程处理。`ApplicationContext` 接口是 `Context` 模块 的焦点。`spring-context-support` 提供支持将常见的第三方库集成到 Spring 应用程序上下文中，特别是用于缓存（`EhCache`，`JCache`）和定时任务等（`CommonJ`，`Quartz`）。

The `spring-expression` module provides a powerful [Expression Language](#) for querying and manipulating an object graph at runtime. It is an extension of the unified expression language (unified EL) as specified in the JSP 2.1 specification. The language supports setting and getting property values, property assignment, method invocation, accessing the

content of arrays, collections and indexers, logical and arithmetic operators, named variables, and retrieval of objects by name from Spring's IoC container. It also supports list projection and selection as well as common list aggregations.

`spring-expression` 模块提供了一个强大的表达式语言，用于在运行时查询和操作对象图。它是 JSP 2.1 规范中规定的统一表达式语言（统一 EL）的扩展。该语言支持设置和获取属性值，属性赋值，方法调用，访问数组的内容，集合和索引器，逻辑和算术运算符，命名变量，以及通过 `Spring IOC` 容器中的名称检索对象。它还支持列表投影和选择以及公共列表聚合。

The `spring-aop` module provides an [AOP](#) Alliance-compliant aspect-oriented programming implementation allowing you to define, for example, method interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated. Using source-level metadata functionality, you can also incorporate behavioral information into your code, in a manner similar to that of .NET attributes.

The separate `spring-aspects` module provides integration with AspectJ.

The `spring-instrument` module provides class instrumentation support and classloader implementations to be used in certain application servers. The `spring-instrument-tomcat` module contains Spring's instrumentation agent for Tomcat.

`spring-aop` 模块提供了一个符合 [AOP](#) Alliance-compliant 的面向方面的编程实现，允许你定义例如方法拦截器和切入点来干净地解耦实现应该分离的功能的代码。使用源代码级元数据功能，你还可以以类似于 [.NET](#) 属性的方式将行为信息合并到代码中。

单独的 `spring-aspects` 模块提供与 [AspectJ](#) 的集成。

`spring-instrument-tomcat` 模块提供类仪器支持和类加载器实现以在某些应用服务器中使用。`spring-instrument-tomcat` 模块包含 Spring 的 Tomcat 的工具代理。

Spring Framework 4 includes a `spring-messaging` module with key abstractions from the *Spring Integration* project such as `Message`, `MessageChannel`, `MessageHandler`, and others to serve as a foundation for messaging-based applications. The module also includes a set of annotations for mapping messages to methods, similar to the Spring MVC annotation based programming model.

Spring Framework 4 包括一个 Spring 消息传递模块，它具有来自 Spring Integration 项目的关键抽象，例如 `Message`，`MessageChannel`，`MessageHandler` 和其他，用作基于消息传递的应用程序的基础。该模块还包括一组用于将消息映射到方法的注解，类似于基于 Spring MVC 注解的编程模型。

## 2.2.4 Data Access/Integration 数据访问/集成

The *Data Access/Integration* layer consists of the JDBC, ORM, OXM, JMS, and Transaction modules.

The `spring-jdbc` module provides a JDBC abstraction layer that removes the need to do tedious JDBC coding and parsing of database-vendor specific error codes.

The `spring-tx` module supports *programmatic and declarative transaction* management for classes that implement special interfaces and for *all your POJOs (Plain Old Java Objects)*.

The `spring-orm` module provides integration layers for popular *object-relational mapping* APIs, including `JPA` and `Hibernate`. Using the `spring-orm` module you can use these O/R-mapping frameworks in combination with all of the other features Spring offers, such as the simple declarative transaction management feature mentioned previously.

The `spring-oxm` module provides an abstraction layer that supports *Object/XML mapping* implementations such as JAXB, Castor, JiBX and XStream.

The `spring-jms` module (*Java Messaging Service*) contains features for producing and consuming messages. Since Spring Framework 4.1, it provides integration with the `spring-messaging` module.

数据访问/集成层由 `JDBC`，`ORM`，`OXM`，`JMS` 和 `Transaction` 模块组成。

`spring-jdbc` 模块提供了一个 `JDBC` 抽象层，消除了对繁琐的 `JDBC` 编码和解析数据库供应商特定的错误代码的需要。

`spring-tx` 模块支持实现特殊接口的类以及所有 `POJO`（普通 Java 对象）的编程和声明事务管理。

`spring-orm` 模块为流行的对象关系映射 API 提供集成层，包括 `JPA` 和 `Hibernate`。使用 `spring-orm` 模块，您可以使用这些 O / R 映射框架结合 Spring 提供的所有其他功能，例如前面提到的简单声明式事务管理功能。

`spring-oxm` 模块提供了一个支持 `对象/ XML` 映射实现的抽象层，例如JAXB，Castor，JiBX和XStream。

`spring-jms` 模块（*Java消息服务*）包含用于生成和使用消息的功能。从 Spring Framework 4.1 开始，它提供了与 `spring-messaging` 模块的集成。

## 2.2.5 Web

The *Web* layer consists of the `spring-web`, `spring-webmvc` and `spring-websocket` modules.

The `spring-web` module provides basic web-oriented integration features such as multipart file upload functionality and the initialization of the IoC container using Servlet listeners and a web-oriented application context. It also contains an HTTP client and the web-related parts of Spring's remoting support.

The `spring-webmvc` module (also known as the *Web-Servlet* module) contains Spring's model-view-controller ([MVC](#)) and REST Web Services implementation for web applications. Spring's MVC framework provides a clean separation between domain model code and web forms and integrates with all of the other features of the Spring Framework.

Web 层由 `spring-web`，`spring-webmvc` 和 `spring-websocket` 模块组成。（注：这里和4的文档比少了 `spring-webmvc-portlet` 模块）

`spring-web` 模块提供基本的面向 Web 的集成功能，例如多部分文件上传功能和使用 `Servlet` 倾听器和面向 Web 的应用程序上下文来初始化 `IoC` 容器。它还包含一个 `HTTP` 客户端和 `Web` 的相关部分的 `Spring` 的远程支持。

`spring-webmvc` 模块（也称为 *Web-Servlet* 模块）包含用于 `Web` 应用程序的 `Spring` 的模型视图控制器 ([MVC](#)) 和 REST Web 服务实现。`Spring` 的 `MVC` 框架提供了 `domain model` (领域模型) 代码和 `Web` 表单之间的清晰分离，并且集成了 `Spring Framework` 所有的其他功能。

## 2.2.6 Test

The `spring-test` module supports the [unit testing](#) and [integration testing](#) of Spring components with JUnit or TestNG. It provides consistent [loading](#) of Spring `ApplicationContext`s and [caching](#) of those contexts. It also provides [mock objects](#) that you can use to test your code in isolation.

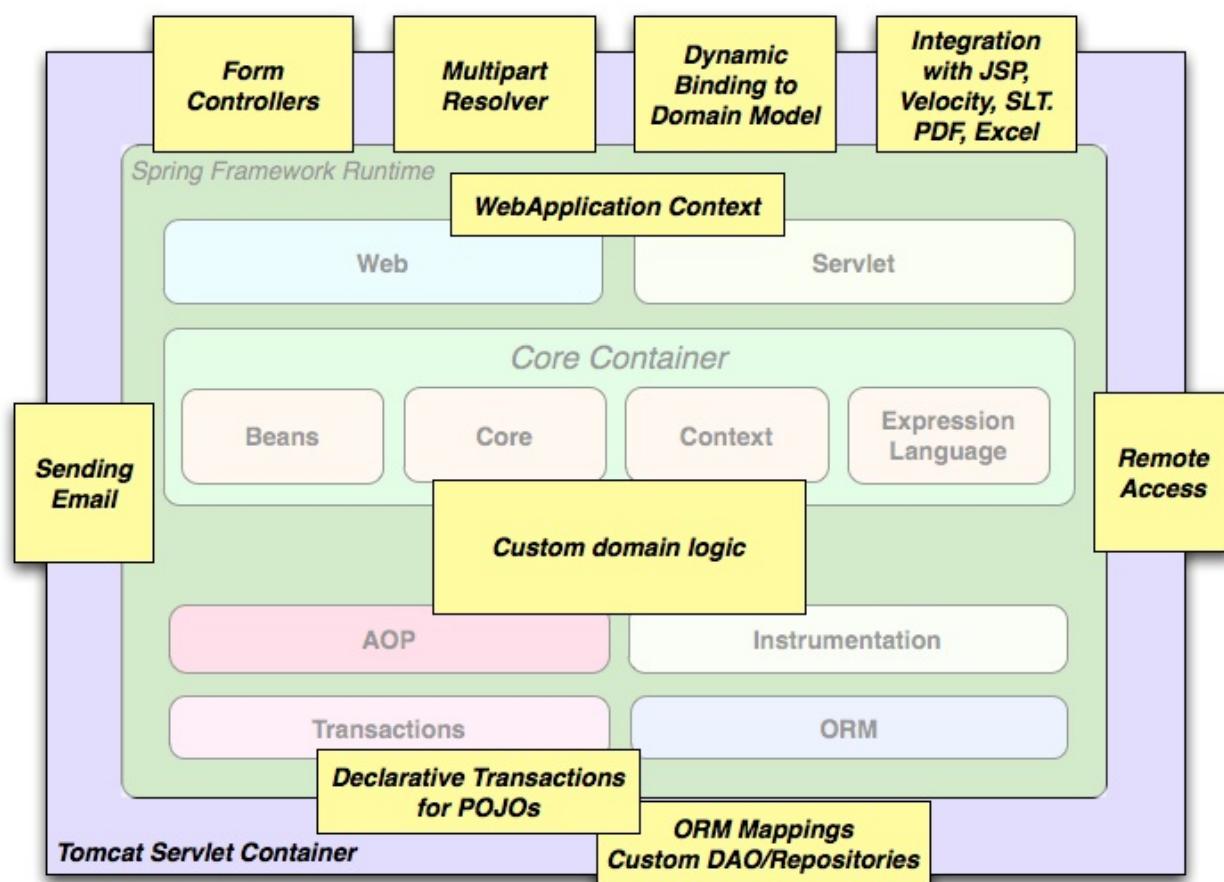
`spring-test` 模块支持使用 `JUnit` 或 `TestNG` 对 Spring 组件进行单元测试和集成测试。它提供了 `SpringApplicationContexts` 的一致加载和这些上下文的缓存。它还提供了 [mock objects](#)(模拟对象)，您可以使用它来单独测试您的代码。

## 2.3 Usage scenarios 使用场景

The building blocks described previously make Spring a logical choice in many scenarios, from embedded applications that run on resource-constrained devices to full-fledged enterprise applications that use Spring's transaction management functionality and web framework integration.

之前描述的构建块使 Spring 成为许多场景中的逻辑(也就是下意识合理的)选择，从资源受限的嵌入式程序到成熟的企业应用程序都可以使用 Spring 事务管理功能和 web 框架集成。

图2.2。典型的完整Spring Web应用程序

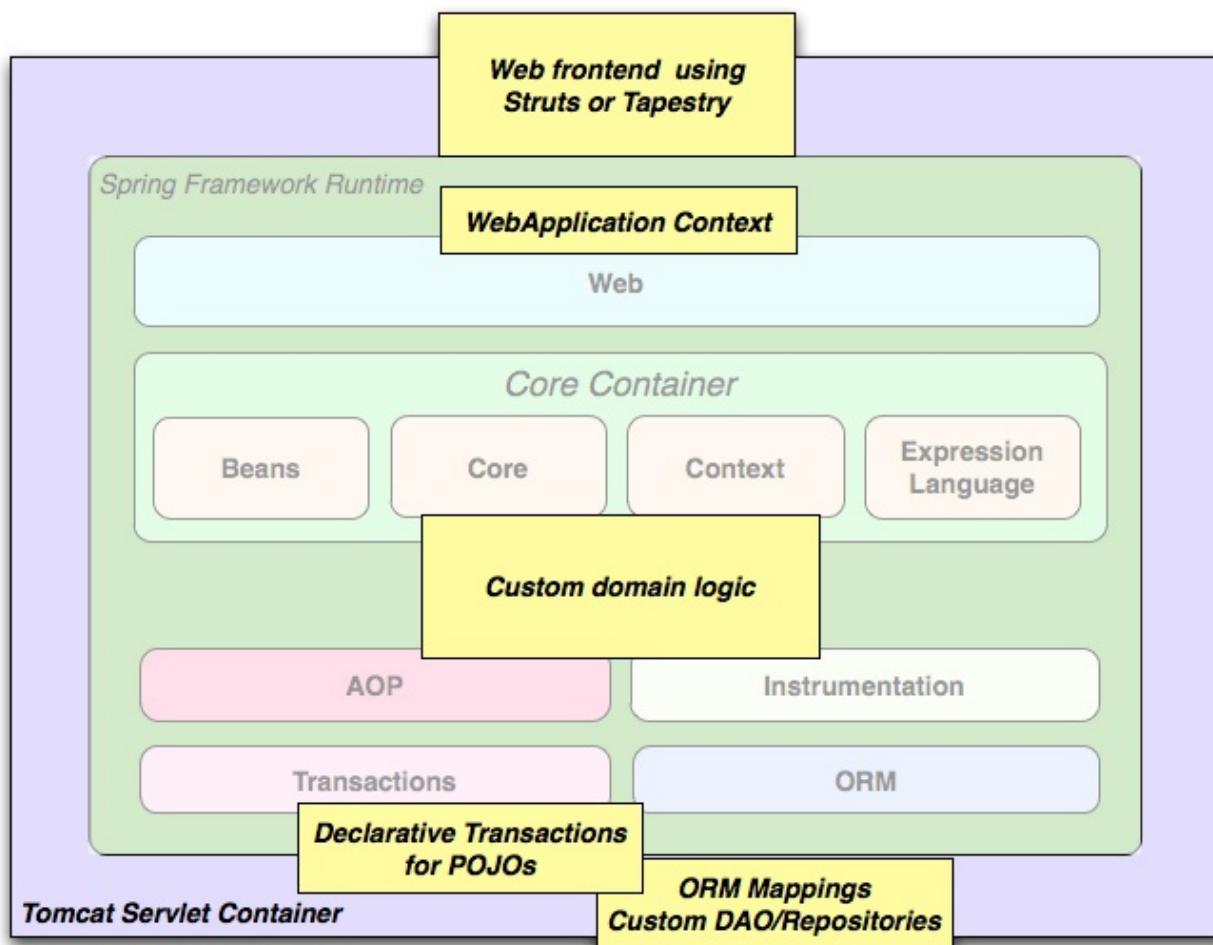


Spring's [declarative transaction management features](#) make the web application fully transactional, just as it would be if you used EJB container-managed transactions. All your custom business logic can be implemented with simple POJOs and managed by Spring's IoC container. Additional services include support for sending email and validation that is independent of the web layer, which lets you choose where to execute validation rules. Spring's ORM support is integrated with JPA and Hibernate; for example, when using Hibernate, you can continue to use your existing mapping files and standard Hibernate

`SessionFactory` configuration. Form controllers seamlessly integrate the web-layer with the domain model, removing the need for `ActionForms` or other classes that transform HTTP parameters to values for your domain model.

Spring 的声明式事务管理特性使 Web 应用程序具有完全事务性，就像使用 EJB 容器管理的事务一样。所有的定制业务逻辑都可以通过简单的 POJO 实现，并由 Spring 的 IOC 容器管理。其他服务包括独立于 Web 层的发送电子邮件和验证的支持，允许你选择执行验证规则的位置。Spring 的 ORM 支持集成了 JPA 和 Hibernate；(注：和 4 的文档比少了 JDO) 例如，当使用 Hibernate 时，可以继续使用现有的映射文件和标准的 Hibernate SessionFactory 配置。表单控制器将 Web 层与域模型无缝集成，从而无需使用 ActionForms 或其他将 HTTP 参数转换为域模型值的类。(注：也就是这些参数无须单独创建 POJO 来表达)

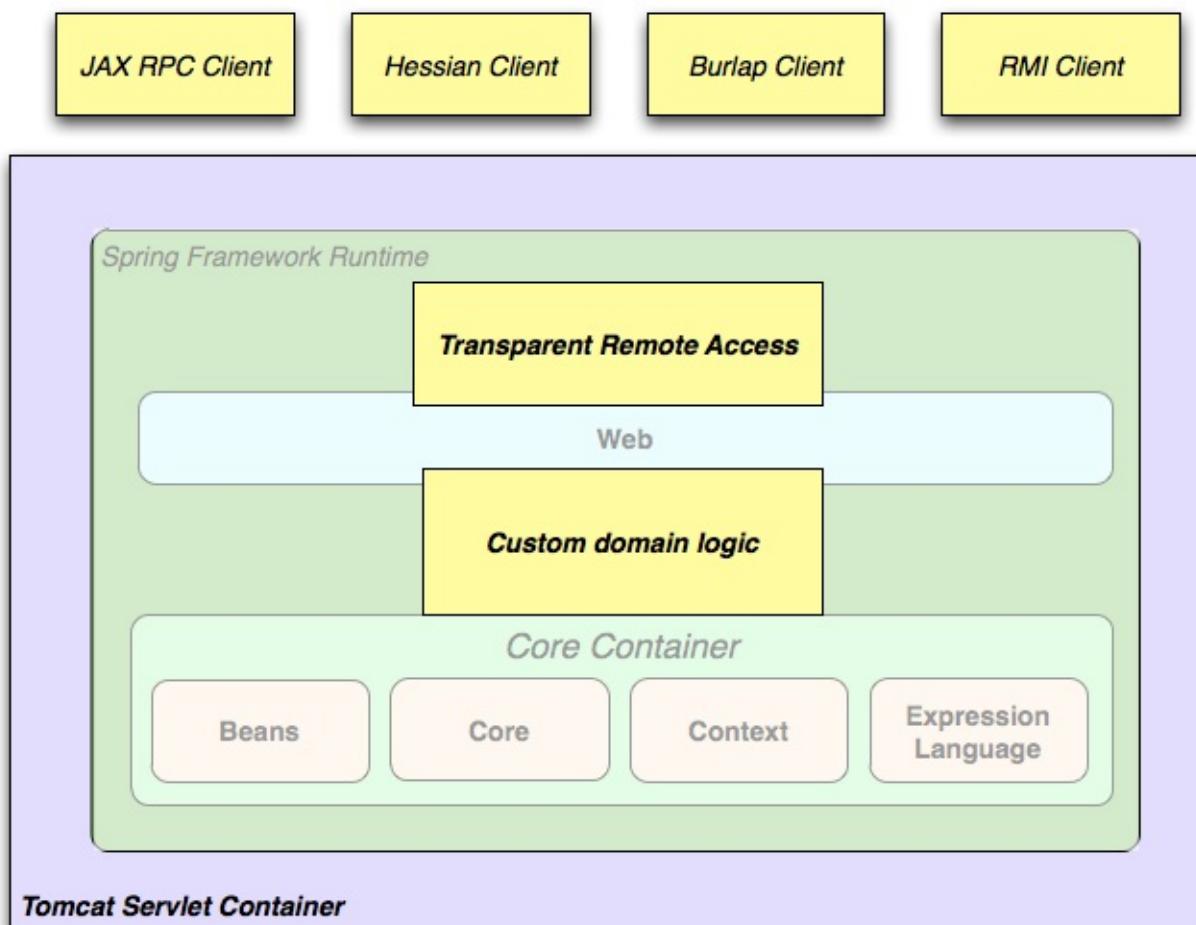
图 2.3。Spring 中间层使用第三方 Web 框架



Sometimes circumstances do not allow you to completely switch to a different framework. The Spring Framework does *not* force you to use everything within it; it is not an *all-or-nothing* solution. Existing front-ends built with Struts, Tapestry, JSF or other UI frameworks can be integrated with a Spring-based middle-tier, which allows you to use Spring transaction features. You simply need to wire up your business logic using an `ApplicationContext` and use a `webApplicationContext` to integrate your web layer.

有时情况不允许你完全切换到一个不同的框架。 Spring 框架不强迫你使用它里面的一切；它不是一个全有或全无的解决方案。使用 Struts , Tapestry , JSF 或其他 UI 框架构建的现有前端可以与基于 Spring 的中间层集成，它允许你使用 Spring 事务功能。你只需要使用 ApplicationContext 连接你的业务逻辑，并使用 WebApplicationContext 来集成你的 web 层。

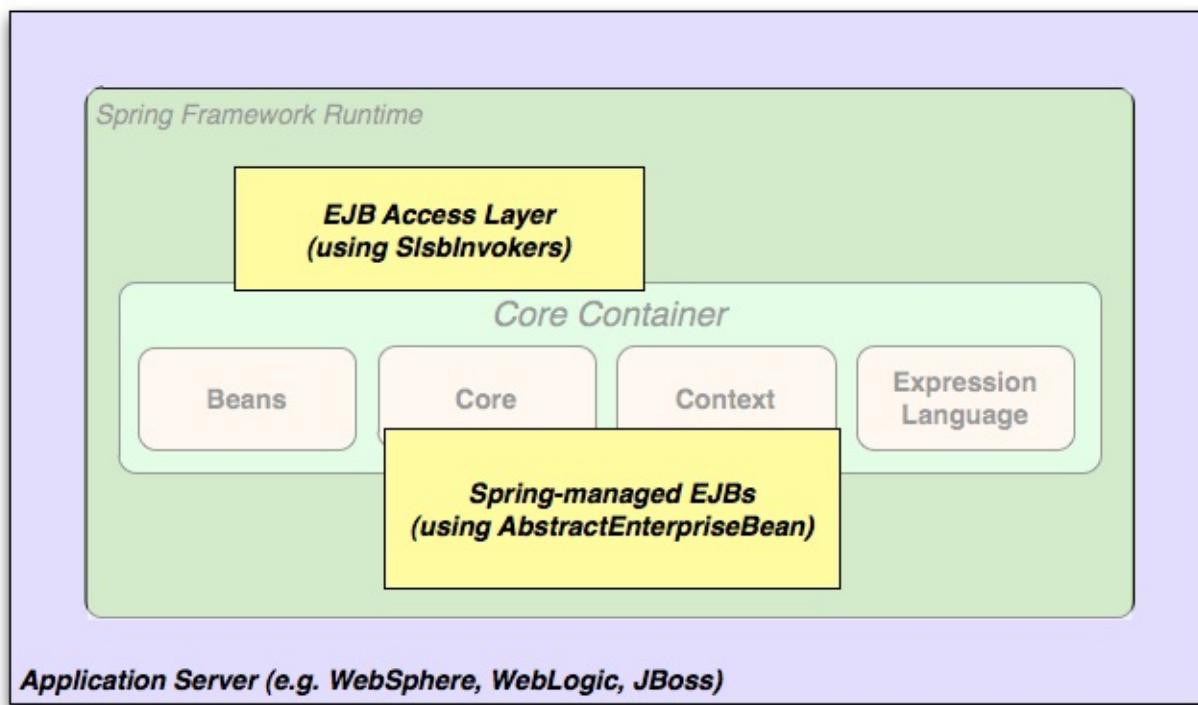
图 2.4。 远程使用场景



When you need to access existing code through web services, you can use Spring's Hessian- , Rmi- OR HttpInvokerProxyFactoryBean classes. Enabling remote access to existing applications is not difficult.

当您需要通过 Web 服务访问现有代码时，可以使用 Spring 的 Hessian- , Rmi- 或 HttpInvokerProxyFactoryBean 类(注：此处和4的文档对比，4使用的是 JaxRpcProxyFactory 类)。启用对现有应用程序的远程访问并不困难。

图 2.5。 EJB - 包装现有POJO



The Spring Framework also provides an [access and abstraction layer](#) for Enterprise JavaBeans, enabling you to reuse your existing POJOs and wrap them in stateless session beans for use in scalable, fail-safe web applications that might need declarative security.

Spring Framework 还为 Enterprise JavaBeans 提供了一个 [访问和抽象层](#)，使您可以重用现有的 `POJOs`，并将其封装在无状态会话 `bean` 中，以用于可能需要声明性安全性的扩展即有安全故障的 `Web` 应用程序中。

## 2.3.1 Dependency Management and Naming Conventions

### 依赖关系管理和命名约定

Dependency management and dependency injection are different things. To get those nice features of Spring into your application (like dependency injection) you need to assemble all the libraries needed (jar files) and get them onto your classpath at runtime, and possibly at compile time. These dependencies are not virtual components that are injected, but physical resources in a file system (typically). The process of dependency management involves locating those resources, storing them and adding them to classpaths. Dependencies can be direct (e.g. my application depends on Spring at runtime), or indirect (e.g. my application depends on `commons-dbcp` which depends on `commons-pool`). The indirect dependencies are also known as "transitive" and it is those dependencies that are hardest to identify and manage.

If you are going to use Spring you need to get a copy of the jar libraries that comprise the pieces of Spring that you need. To make this easier Spring is packaged as a set of modules that separate the dependencies as much as possible, so for example if you don't want to write a web application you don't need the spring-web modules. To refer to Spring library modules in this guide we use a shorthand naming convention `spring-*` or `spring-*.jar`, where `*` represents the short name for the module (e.g. `spring-core` , `spring-webmvc` , `spring-jms` , etc.). The actual jar file name that you use is normally the module name concatenated with the version number (e.g. `spring-core-5.0.0.M4.jar`).

Each release of the Spring Framework will publish artifacts to the following places:

- Maven Central, which is the default repository that Maven queries, and does not require any special configuration to use. Many of the common libraries that Spring depends on also are available from Maven Central and a large section of the Spring community uses Maven for dependency management, so this is convenient for them. The names of the jars here are in the form `spring-*.jar` and the Maven groupId is `org.springframework` .
- In a public Maven repository hosted specifically for Spring. In addition to the final GA releases, this repository also hosts development snapshots and milestones. The jar file names are in the same form as Maven Central, so this is a useful place to get development versions of Spring to use with other libraries deployed in Maven Central. This repository also contains a bundle distribution zip file that contains all Spring jars bundled together for easy download.

So the first thing you need to decide is how to manage your dependencies: we generally recommend the use of an automated system like Maven, Gradle or Ivy, but you can also do it manually by downloading all the jars yourself.

You will find below the list of Spring artifacts. For a more complete description of each modules, see [Section 2.2, “Modules”](#).

依赖管理和依赖注入是不同的。为了让 Spring 的这些不错的功能运用到运用程序中(比如依赖注入)，你需要导入所有需要的库 (jar文件) ，并且在编译、运行的时候将它们放到你的类路径中。这些依赖关系不是注入的虚拟组件，而是文件系统中的物理资源 (通常)。依赖关系管理的过程包括定位这些资源，存储它们并将它们添加到类路径。依赖可以是直接的 (例如我的应用程序依赖于 Spring 在运行时) 或间接 (例如我的应用程序依赖于 commons-dbcp ，这取决于 commons-pool ) 。间接依赖性也称为“传递性”，它是那些最难识别和管理的依赖性。

如果你要使用 Spring ，你需要得到一个包含你需要的 Spring 的 jar 库的 jar 文件。为了使这过程更简单， Spring 被打包为一组模块，尽可能地分离依赖，所以假如你不想编写一个 web 应用程序，你不需要 spring-web 模块。为了在本指南中引用 Spring 库模块，我们使用一个简写命名约定 spring-\* 或 spring-\* .jar ，其中\*表示模块的短名称 (例如spring-core , spring-webmvc , spring-jms 等) 。 ) 。你使用的实际jar文件名通常是与版本号连接的模块名 (例如 spring-core-5.0.0.M4.jar ) 。

Spring Framework 的每个版本都会将工件发布到以下位置：

- Maven Central ，它是 Maven 查询的默认存储库，不需要使用任何特殊配置。 Spring 依赖的许多通用库也可以从 Maven Central 获得， Spring 社区的一大部分使用 Maven 进行依赖关系管理，所以这对他们很方便。这里的jars的名字是在 spring-\* .jar 和 Maven groupId 是 org.springframework 。
- 在一个专门为 Spring 托管的公共 Maven 仓库中。除了最终 GA 版本，此存储库还托管开发快照和里程碑。 jar 文件名的格式与 Maven Central 相同，因此这是一个有用的地方，可以让 Spring 的开发版本与 Maven Central 中部署的其他库一起使用。此存储库还包含捆绑包分发 zip 文件，其中包含捆绑在一起的所有 Spring jar 以便于下载。

所以你需要决定的第一件事是如何管理你的依赖：我们一般建议使用一个自动化系统，如 Maven , Gradle 或 Ivy ，但你也可以手动下载所有的 jar 本身。

你会发现下面的 Spring 工件列表。有关每个模块的更完整的描述，请参见第 2.2 节“模块”。

**表2.1。 Spring 框架组件**

GroupId	ArtifactId	Description
org.springframework	spring-aop	Proxy-based AOP support 基于代理的AOP支持
org.springframework	spring-aspects	AspectJ based aspects 基于AspectJ的切面
org.springframework	spring-beans	Beans support, including Groovy Bean支持，包括Groovy
org.springframework	spring-	Application context runtime, including scheduling and remoting abstractions 应用程序上下文运行

	context	时，包括调度和远程抽象
org.springframework	spring-context-support	Support classes for integrating common third-party libraries into a Spring application context 支持将常见的第三方库集成到Spring应用程序上 下文中的类
org.springframework	spring-core	Core utilities, used by many other Spring modules 核心应用程序，由许多其他Spring模块使用
org.springframework	spring-expression	Spring Expression Language (SpEL)
org.springframework	spring-instrument	Instrumentation agent for JVM bootstrapping JVM引导的工具代理
org.springframework	spring-instrument-tomcat	Instrumentation agent for Tomcat Tomcat的工具代理
org.springframework	spring-jdbc	JDBC support package, including DataSource setup and JDBC access support JDBC支持包， 包括DataSource设置和JDBC访问支持
org.springframework	spring-jms	JMS support package, including helper classes to send and receive JMS messages JMS支持包， 包括用于发送和接收JMS消息的助手类
org.springframework	spring-messaging	Support for messaging architectures and protocols 支持消息架构和协议
org.springframework	spring-orm	Object/Relational Mapping, including JPA and Hibernate support 对象/关系映射，包括JPA和 Hibernate支持
org.springframework	spring-oxm	Object/XML Mapping 对象/ XML映射
org.springframework	spring-test	Support for unit testing and integration testing Spring components 支持单元测试和集成测试的 Spring组件
org.springframework	spring-tx	Transaction infrastructure, including DAO support and JCA integration 事务基础设施，包括 DAO支持和集成制定
org.springframework	spring-web	Web support packages, including client and web remoting Web支持包，包括客户端和Web远程处理
org.springframework	spring-webmvc	REST Web Services and model-view-controller implementation for web applications Web应用程序的REST Web服务和模型 - 视图 - 控制器实现
org.springframework	springwebsocket	WebSocket and SockJS implementations, including STOMP support WebSocket和SockJS实现，包括STOMP支持

## Spring Dependencies and Depending on Spring Spring 依赖管理

Although Spring provides integration and support for a huge range of enterprise and other external tools, it intentionally keeps its mandatory dependencies to an absolute minimum: you shouldn't have to locate and download (even automatically) a large number of jar libraries in order to use Spring for simple use cases. For basic dependency injection there is only one mandatory external dependency, and that is for logging (see below for a more detailed description of logging options).

Next we outline the basic steps needed to configure an application that depends on Spring, first with Maven and then with Gradle and finally using Ivy. In all cases, if anything is unclear, refer to the documentation of your dependency management system, or look at some sample code - Spring itself uses Gradle to manage dependencies when it is building, and our samples mostly use Gradle or Maven.

虽然 Spring 提供了大量对企业和其他外部工具的集成和支持，它有意地将其强制依赖保持在绝对最低限度：你无需查找和下载（甚至自动）大量的jar库，以便使用 Spring 的简单用例。对于基本依赖注入，只有一个强制性的外部依赖，这是用于日志记录的（有关日志选项的更详细描述，请参见下文）。

接下来，我们概述配置依赖于 Spring 的应用程序所需的基本步骤，首先使用 Maven，然后使用 Gradle，最后使用 Ivy。在所有情况下，如果有什么不清楚，请参考依赖管理系统的文档，或者查看一些示例代码 - Spring 本身使用 Gradle 在构建时管理依赖关系，我们的示例主要使用 Gradle 或 Maven。

## Maven Dependency Management Maven 依赖管理

If you are using [Maven](#) for dependency management you don't even need to supply the logging dependency explicitly. For example, to create an application context and use dependency injection to configure an application, your Maven dependencies will look like this:

如果你使用 Maven 进行依赖关系管理，你甚至不需要显式提供日志记录依赖关系。例如，要创建应用程序上下文并使用依赖注入来配置应用程序，你的 Maven 依赖关系将如下所示：

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.0.0.M4</version>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

That's it. Note the scope can be declared as runtime if you don't need to compile against Spring APIs, which is typically the case for basic dependency injection use cases.

The example above works with the Maven Central repository. To use the Spring Maven repository (e.g. for milestones or developer snapshots), you need to specify the repository location in your Maven configuration. For full releases:

就是这样。注意，如果不需要针对 Spring API 进行编译，范围( scope )可以声明为运行时( runtime )，这通常是基本依赖注入使用用例。

上面的示例使用 Maven Central 存储库。要使用 Spring Maven 存储库（例如，用于里程碑或开发人员快照），你需要在 Maven 配置中指定存储库位置。完整版本：

```
<repositories>
  <repository>
    <id>io.spring.repo.maven.release</id>
    <url>http://repo.spring.io/release/</url>
    <snapshots><enabled>false</enabled></snapshots>
  </repository>
</repositories>
```

里程碑：

```
<repositories>
  <repository>
    <id>io.spring.repo.maven.milestone</id>
    <url>http://repo.spring.io/milestone/</url>
    <snapshots><enabled>false</enabled></snapshots>
  </repository>
</repositories>
```

And for snapshots: 以及快照：

```
<repositories>
  <repository>
    <id>io.spring.repo.maven.snapshot</id>
    <url>http://repo.spring.io/snapshot/</url>
    <snapshots><enabled>true</enabled></snapshots>
  </repository>
</repositories>
```

## Maven "Bill Of Materials" Dependency

It is possible to accidentally mix different versions of Spring JARs when using Maven. For example, you may find that a third-party library, or another Spring project, pulls in a transitive dependency to an older release. If you forget to explicitly declare a direct dependency yourself, all sorts of unexpected issues can arise.

To overcome such problems Maven supports the concept of a "bill of materials" (BOM) dependency. You can import the `spring-framework-bom` in your `dependencyManagement` section to ensure that all spring dependencies (both direct and transitive) are at the same version.

在使用 `Maven` 时，可能会意外混合不同版本的 `Spring JAR`。例如，你可能会发现第三方库或另一个 `Spring` 项目将传递依赖项拉入旧版本。如果你忘记自己显式声明一个直接依赖，可能会出现各种意想不到的问题。

为了克服这种问题，`Maven` 支持“物料清单”（`BOM`）依赖的概念。你可以在 `dependencyManagement` 部分中导入 `spring-framework-bom`，以确保所有 `spring` 依赖项（直接和可传递）具有相同的版本。

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-framework-bom</artifactId>
      <version>5.0.0.M4</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

An added benefit of using the BOM is that you no longer need to specify the `<version>` attribute when depending on Spring Framework artifacts:

使用 `BOM` 后，当依赖 `Spring Framework` 组件后，无需再指定 `<version>` 属性

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
  </dependency>
</dependencies>
```

## Gradle Dependency Management 通过Gradle 做依赖管理

To use the Spring repository with the [Gradle](#) build system, include the appropriate URL in the `repositories` section:

要将 Spring 存储库与 Gradle 构建系统一起使用，请在 `repositories` 部分中包含相应的 URL：

```
repositories {
    mavenCentral()
    // and optionally...
    maven { url "http://repo.spring.io/release" }
}
```

You can change the `repositories` URL from `/release` to `/milestone` or `/snapshot` as appropriate. Once a repository has been configured, you can declare dependencies in the usual Gradle way:

你可以根据需要将 `repositories` 中URL从 `/ release` 更改为 `/ milestone` 或 `/ snapshot`。一旦配置了 `repositories`，就可以按照通常的 Gradle 方式声明依赖关系：

```
dependencies {
    compile("org.springframework:spring-context:5.0.0.M4")
    testCompile("org.springframework:spring-test:5.0.0.M4")
}
```

## Ivy Dependency Management Ivy 依赖管理

If you prefer to use [Ivy](#) to manage dependencies then there are similar configuration options.

To configure Ivy to point to the Spring repository add the following resolver to your `ivysettings.xml` :

如果你喜欢使用 [Ivy](#) 来管理依赖，那么有类似的配置选项。

要配置 Ivy 指向 Spring 存储库，请将以下解析器添加到你的 `ivysettings.xml`：

```
<resolvers>
    <ibiblio name="io.spring.repo.maven.release"
        m2compatible="true"
        root="http://repo.spring.io/release/" />
</resolvers>
```

You can change the `root` URL from `/release/` to `/milestone/` or `/snapshot/` as appropriate.

Once configured, you can add dependencies in the usual way. For example (in `ivy.xml`):

你可以根据需要将 `root URL` 从 `/release/` 更改为 `/milestone/` 或 `/snapshot/`。

配置后，你可以按照通常的方式添加依赖关系。例如（在 `ivy.xml` 中）：

```
<dependency org="org.springframework"
    name="spring-core" rev="5.0.0.M4" conf="compile->runtime"/>
```

## Distribution Zip Files

Although using a build system that supports dependency management is the recommended way to obtain the Spring Framework, it is still possible to download a distribution zip file.

Distribution zips are published to the Spring Maven Repository (this is just for our convenience, you don't need Maven or any other build system in order to download them).

To download a distribution zip open a web browser to

<http://repo.spring.io/release/org/springframework/spring> and select the appropriate subfolder for the version that you want. Distribution files end `-dist.zip`, for example `spring-framework-{spring-version}-RELEASE-dist.zip`. Distributions are also published for `milestones` and `snapshots`.

尽管使用支持依赖性管理的构建系统是获取 `Spring Framework` 的推荐方式，但仍然可以下载分发 `zip` 文件。

分发 `zip` 是发布到 `Spring Maven` 仓库（这只是为了我们的方便，你不需要 `Maven` 或任何其他构建系统为了下载它们）。

要下载分发 `zip`，请打开 Web 浏览器到

<http://repo.spring.io/release/org/springframework/spring>，然后为所需的版本选择适当的子文件夹。分发文件结尾是 `-dist.zip`，例如 `spring-framework-{spring-version}-RELEASE-dist.zip`。还分发了里程碑和快照的分发。

## 2.3.2 Logging 日志

Logging is a very important dependency for Spring because *a)* it is the only mandatory external dependency, *b)* everyone likes to see some output from the tools they are using, and *c)* Spring integrates with lots of other tools all of which have also made a choice of logging dependency. One of the goals of an application developer is often to have unified logging configured in a central place for the whole application, including all external components. This is more difficult than it might have been since there are so many choices of logging framework.

The mandatory logging dependency in Spring is the Jakarta Commons Logging API (JCL). We compile against JCL and we also make JCL `Log` objects visible for classes that extend the Spring Framework. It's important to users that all versions of Spring use the same logging library: migration is easy because backwards compatibility is preserved even with applications that extend Spring. The way we do this is to make one of the modules in Spring depend explicitly on `commons-logging` (the canonical implementation of JCL), and then make all the other modules depend on that at compile time. If you are using Maven for example, and wondering where you picked up the dependency on `commons-logging`, then it is from Spring and specifically from the central module called `spring-core`.

The nice thing about `commons-logging` is that you don't need anything else to make your application work. It has a runtime discovery algorithm that looks for other logging frameworks in well known places on the classpath and uses one that it thinks is appropriate (or you can tell it which one if you need to). If nothing else is available you get pretty nice looking logs just from the JDK (`java.util.logging` or JUL for short). You should find that your Spring application works and logs happily to the console out of the box in most situations, and that's important.

日志记录是 Spring 的一个非常重要的依赖，因为 *a)* 它是唯一的强制性的外部依赖，*b)* 每个人都喜欢从他们使用的工具看到一些输出，和 *c)* Spring 集成了许多其他工具，日志依赖性的选择。应用程序开发人员的目标之一通常是在整个应用程序的中心位置配置统一日志记录，包括所有外部组件。这就更加困难，因为它可能已经有太多选择的日志框架。

Spring 中的强制性日志依赖性是 Jakarta Commons Logging API (JCL)。我们编译 JCL，我们也使 JCL Log 对象对于扩展 Spring 框架的类可见。对于用户来说，所有版本的 Spring 都使用相同的日志库很重要：迁移很容易，因为即使使用扩展 Spring 的应用程序也保持向后兼容性。我们这样做的方式是使 Spring 中的一个模块显式地依赖 commons-logging (JCL 的规范实现)，然后使所有其他模块在编译时依赖它。如果你使用 Maven 为例，并想知道你在哪里选择对 commons-logging 的依赖，那么它是从 Spring，特别是从中央模块称为 spring-core ( 关于此处，理解就好，翻译的不到位 )。

关于 commons-logging 的好处是，你不需要任何其他东西来就能让你的应用程序工作。它有一个运行时发现算法，该算法在众所周知的 classpath 路径下寻找其他日志框架，并使用它认为是合适的（或者你可以告诉它，如果你需要）。如果没有其他可用的，你可以从 JDK（java.util.logging 或简称 JUL）获得漂亮的查看日志。你应该会发现，你的 Spring 应用程序在大多数情况下可以很好地工作和记录到控制台，这很重要。

## Not Using Commons Logging 不使用 Commons Logging

Unfortunately, the runtime discovery algorithm in commons-logging , while convenient for the end-user, is problematic. If we could turn back the clock and start Spring now as a new project it would use a different logging dependency. The first choice would probably be the Simple Logging Facade for Java ( [SLF4J](#)), which is also used by a lot of other tools that people use with Spring inside their applications.

There are basically two ways to switch off commons-logging :

1. Exclude the dependency from the spring-core module (as it is the only module that explicitly depends on commons-logging )
2. Depend on a special commons-logging dependency that replaces the library with an empty jar (more details can be found in the [SLF4J FAQ](#))

To exclude commons-logging, add the following to your dependencyManagement section:

不幸的是，公共日志中的运行时发现算法虽然对于终端用户方便，但是是有问题的。如果我们可以时光倒流并启动 Spring 作为一个新项目，它将使用不同的日志依赖关系。第一个选择可能是用于 Java 的简单日志外观 ([SLF4J](#))，它也被许多其他工具用于 Spring 在其应用程序中使用。

基本上有两种方法关闭 commons-logging :

1.从 spring-core 模块中排除依赖性（因为它是唯一显式依赖 commons-logging 的模块） 2.依赖于一个特殊的 commons-logging 依赖，用一个空jar替换库（更多细节可以在 [SLF4J FAQ](#) 中找到）

要排除 commons-logging ，请将以下内容添加到你的 dependencyManagement 部分：

```

<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>5.0.0.M4</version>
        <exclusions>
            <exclusion>
                <groupId>commons-logging</groupId>
                <artifactId>commons-logging</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
</dependencies>

```

Now this application is probably broken because there is no implementation of the JCL API on the classpath, so to fix it a new one has to be provided. In the next section we show you how to provide an alternative implementation of JCL using SLF4J as an example.

现在这个应用程序运行不了，因为没有在类路径上实现 JCL API，所以要解决它需要提供一个新的。在下一节中，我们将向你展示如何使用 SLF4J 提供 JCL 的替代实现。

## Using SLF4J 使用 SLF4J

SLF4J is a cleaner dependency and more efficient at runtime than commons-logging because it uses compile-time bindings instead of runtime discovery of the other logging frameworks it integrates. This also means that you have to be more explicit about what you want to happen at runtime, and declare it or configure it accordingly. SLF4J provides bindings to many common logging frameworks, so you can usually choose one that you already use, and bind to that for configuration and management.

SLF4J provides bindings to many common logging frameworks, including JCL, and it also does the reverse: bridges between other logging frameworks and itself. So to use SLF4J with Spring you need to replace the commons-logging dependency with the SLF4J-JCL bridge. Once you have done that then logging calls from within Spring will be translated into logging calls to the SLF4J API, so if other libraries in your application use that API, then you have a single place to configure and manage logging.

A common choice might be to bridge Spring to SLF4J, and then provide explicit binding from SLF4J to Log4J. You need to supply 4 dependencies (and exclude the existing commons-logging): the bridge, the SLF4J API, the binding to Log4J, and the Log4J implementation itself. In Maven you would do that like this

`SLF4J` 是一个更简洁的依赖，在运行时比 `commons-logging` 更高效，因为它使用编译时绑定，而不是其集成的其他日志框架的运行时发现。这也意味着你必须更明确地了解你想在运行时发生什么，并声明它或相应地配置它。`SLF4J` 提供了对许多常见日志框架的绑定，因此你通常可以选择一个已经使用的绑定，并绑定到配置和管理。

`SLF4J` 提供对许多常见日志框架（包括 `JCL`）的绑定，并且它也做了反向工作：充当其他日志框架与其自身之间的桥梁。因此，要在 `Spring` 中使用 `SLF4J`，需要使用 `SLF4J-JCL` 桥替换 `commons-logging` 依赖关系。一旦你这样做，那么来自 `Spring` 内部的日志调用将被转换为对 `SLF4J API` 的日志调用，因此如果应用程序中的其他库使用该 API，那么你有一个地方可以配置和管理日志记录。

常见的选择可能是将 `Spring` 桥接到 `SLF4J`，然后提供从 `SLF4J` 到 `Log4J` 的显式绑定。你需要提供 4 个依赖（并排除现有的 `commons-logging`）：桥梁，`SLF4J API`，绑定到 `Log4J` 和 `Log4J` 实现本身。在 `Maven`，你会这样做

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>5.0.0.M4</version>
        <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>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>1.5.8</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>1.5.8</version>
    </dependency>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.14</version>
    </dependency>
</dependencies>
```

That might seem like a lot of dependencies just to get some logging. Well it is, but it *is* optional, and it should behave better than the vanilla `commons-logging` with respect to classloader issues, notably if you are in a strict container like an OSGi platform. Allegedly there is also a performance benefit because the bindings are at compile-time not runtime.

A more common choice amongst SLF4J users, which uses fewer steps and generates fewer dependencies, is to bind directly to [Logback](#). This removes the extra binding step because Logback implements SLF4J directly, so you only need to depend on two libraries not four (`jcl-over-slf4j` and `logback`). If you do that you might also need to exclude the `slf4j-api` dependency from other external dependencies (not Spring), because you only want one version of that API on the classpath.

这可能看起来像很多依赖只是为了获得一些日志。它的确如此，但它是可选的，它在关于类加载器的问题上应该比 `commons-logging` 表现的更加的好，特别是当它运行在一个严格的容器中像 `osgi` 平台。据称，还有一个性能优势，因为绑定是在编译时，而不是运行时。

SLF4J 用户中更常见的选择是使用较少的步骤和生成较少的依赖关系，它是直接绑定到 `Logback`。这消除了额外的绑定步骤，因为 `Logback` 直接实现 `SLF4J`，所以你只需要依赖于两个不是四个库（`jcl-over-slf4j` 和 `logback`）。如果你这样做，你可能还需要从其他外部依赖（不是 `Spring`）中排除 `slf4j-api` 依赖，因为你只需要在类路径上有一个版本的 `API`。

## Using Log4J

Many people use [Log4j](#) as a logging framework for configuration and management purposes. It's efficient and well-established, and in fact it's what we use at runtime when we build and test Spring. Spring also provides some utilities for configuring and initializing Log4j, so it has an optional compile-time dependency on Log4j in some modules.

To make Log4j work with the default JCL dependency (`commons-logging`) all you need to do is put Log4j on the classpath, and provide it with a configuration file (`log4j.properties` or `log4j.xml` in the root of the classpath). So for Maven users this is your dependency declaration:

许多人使用 `Log4j` 作为日志框架用于配置和管理目的。它是高效的和成熟的，事实上，这是我们在运行时使用时，我们构建和测试 `Spring`。`Spring` 还提供了一些用于配置和初始化 `Log4j` 的实用程序，所以它在一些模块中对 `Log4j` 有一个可选的编译时依赖。

要使 `Log4j` 使用默认的 `JCL` 依赖 (`commons-logging`)，所有你需要做的是将 `Log4j` 放在类路径上，并为它提供一个配置文件（在类路径的根目录下 `log4j.properties` 或 `log4j.xml`）。所以对于 `Maven` 用户，这是你的依赖性声明：

```

<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>5.0.0.M4</version>
    </dependency>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.14</version>
    </dependency>
</dependencies>

```

And here's a sample log4j.properties for logging to the console:

下面是一个简单的 log4j.properties 的实例，用于将日志打印到控制台：

```

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

```

## Runtime Containers with Native JCL

Many people run their Spring applications in a container that itself provides an implementation of JCL. IBM Websphere Application Server (WAS) is the archetype. This often causes problems, and unfortunately there is no silver bullet solution; simply excluding commons-logging from your application is not enough in most situations.

To be clear about this: the problems reported are usually not with JCL per se, or even with commons-logging : rather they are to do with binding commons-logging to another framework (often Log4J). This can fail because commons-logging changed the way they do the runtime discovery in between the older versions (1.0) found in some containers and the modern versions that most people use now (1.1). Spring does not use any unusual parts of the JCL API, so nothing breaks there, but as soon as Spring or your application tries to do any logging you can find that the bindings to Log4J are not working.

In such cases with WAS the easiest thing to do is to invert the class loader hierarchy (IBM calls it "parent last") so that the application controls the JCL dependency, not the container. That option isn't always open, but there are plenty of other suggestions in the public domain for alternative approaches, and your mileage may vary depending on the exact version and feature set of the container.

许多人在一个容器中运行他们的 Spring 应用程序，该容器本身提供了 JCL 的实现。 IBM Websphere 应用服务器（WAS）就是一个例子。这常常导致问题，不幸的是没有一个一劳永逸解决方案；在大多数情况下，只是从你的应用程序中排除 commons-logging 是不够的。

要明确这一点：报告的问题通常不是 JCL 本身，或者甚至与 commons-logging：而是他们要绑定到另一个框架（通常 Log4J）的公共日志。这可能会失败，因为 commons-logging 改变了在一些容器中发现的旧版本（1.0）和大多数人现在使用的现代版本（1.1）之间执行运行时发现的方式。 Spring 不使用 JCL API 的任何不常见的模块，所以没有什么问题出现，但一旦 Spring 或你的应用程序试图做任何日志记录，你可以发现绑定的 Log4J 不工作了。

在这种情况下，使用 WAS，最容易做的事情是反转类加载器层次结构（IBM 称之为“父最后一个”），以便应用程序控制 JCL 依赖关系，而不是容器。该选项并不总是开放的，但在公共领域有许多其他建议替代方法，你的里程可能会根据容器的确切版本和功能集而有所不同。

# Part II. Core Technologies 核心技术

This part of the reference documentation covers all of those technologies that are absolutely integral to the Spring Framework.

Foremost amongst these is the Spring Framework's Inversion of Control (IoC) container. A thorough treatment of the Spring Framework's IoC container is closely followed by comprehensive coverage of Spring's Aspect-Oriented Programming (AOP) technologies. The Spring Framework has its own AOP framework, which is conceptually easy to understand, and which successfully addresses the 80% sweet spot of AOP requirements in Java enterprise programming.

Coverage of Spring's integration with AspectJ (currently the richest - in terms of features - and certainly most mature AOP implementation in the Java enterprise space) is also provided.

- [Chapter 3, The IoC container](#)
- [Chapter 4, Resources](#)
- [Chapter 5, Validation, Data Binding, and Type Conversion](#)
- [Chapter 6, Spring Expression Language \(SpEL\)](#)
- [Chapter 7, Aspect Oriented Programming with Spring](#)
- [Chapter 8, Spring AOP APIs](#)

这部分参考文档涵盖了所有那些绝对必要的 spring 框架的技术。

其中最重要的是 Spring 框架的控制反转（ IoC ）容器。对 Spring 框架的 IoC 容器的全面处理,紧跟其后是全面覆盖 Spring 的面向方面的编程（ AOP ）技术。

(关于 AOP 的理解，自己写了篇博文[谈 AOP 的通俗理解](#))

spring 框架有自己的 AOP 框架，它在概念上很容易理解，而且成功解决了 Java企业级应用 中 AOP 需求80%的核心要素。

Spring 还提供了与 AspectJ （目前是最丰富的 - 在功能方面 - 而且也无疑是Java企业领域最成熟的AOP实现 ）的集成。

- 第3章， IoC 容器
- 第4章，资源
- 第5章，验证，数据绑定和类型转换
- 第6章，Spring表达式语言（ SpEL ）
- 第7章，使用Spring的面向方面的编程
- 第8章， Spring AOP API



### 3. The IoC container IoC容器

## 3.1 Spring IoC容器和bean的简介

This chapter covers the Spring Framework implementation of the Inversion of Control (IoC) [1] principle. IoC is also known as *dependency injection* (DI). It is a process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. The container then *injects* those dependencies when it creates the bean. This process is fundamentally the inverse, hence the name *Inversion of Control* (IoC), of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes, or a mechanism such as the *Service Locator* pattern.

The `org.springframework.beans` and `org.springframework.context` packages are the basis for Spring Framework's IoC container. The `BeanFactory` interface provides an advanced configuration mechanism capable of managing any type of object. `ApplicationContext` is a sub-interface of `BeanFactory`. It adds easier integration with Spring's AOP features; message resource handling (for use in internationalization), event publication; and application-layer specific contexts such as the `WebApplicationContext` for use in web applications.

In short, the `BeanFactory` provides the configuration framework and basic functionality, and the `ApplicationContext` adds more enterprise-specific functionality. The `ApplicationContext` is a complete superset of the `BeanFactory`, and is used exclusively in this chapter in descriptions of Spring's IoC container. For more information on using the `BeanFactory` instead of the `ApplicationContext`, refer to [Section 3.16, “The BeanFactory”](#).

In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC *container* are called *beans*. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. Otherwise, a bean is simply one of many objects in your application. Beans, and the *dependencies* among them, are reflected in the *configuration metadata* used by a container.

本章介绍了 Spring 框架实现的控制反转（`Ioc`）[1]原理。 `Ioc` 也称为依赖注入（`DI`）。它是一个过程，对象通过构造函数参数，工厂方法的参数或在对象实例构造或从工厂方法返回后在对象实例上设置的属性来定义它们的依赖关系，即它们一起合作的其他对象。然后容器在创建bean时注入那些依赖。这个过程基本上是相反的，因此称为控制反转（`Ioc`），通过使用类的直接构造或诸如服务定位器模式的机制来控制其依赖性的实例化或位置的bean自身的名称。

`org.springframework.beans` 和 `org.springframework.context` 包是 Spring Framework 的 IoC 容器的基础。`BeanFactory` 接口提供了一种能够管理任何类型的对象的高级配置机制。

`ApplicationContext` 是 `BeanFactory` 的子接口。它增加了与 Spring 的 AOP 特性的更容易的集成到一起的实现;消息资源处理(用于国际化),事件发布;和应用程序层特定上下文(如 `WebApplicationContext`)以用于 Web 应用程序。

简而言之, `BeanFactory` 提供了配置框架和基本功能, `ApplicationContext` 添加了更多的企业特定功能。`ApplicationContext` 是 `BeanFactory` 的完整超集, 并且在本章中专门用于描述 Spring 的 IoC 容器。有关使用 `BeanFactory` 而不是 `ApplicationContext` 的更多信息, 请参见第3.16节“`BeanFactory`”。

在 Spring 中, 构成应用程序主干并由 Spring IoC 容器管理的对象称为 bean。bean 是由 Spring IoC 容器实例化, 组装和以其他方式管理的对象。此外, bean 只是应用程序中许多对象之一。Bean 及其之间的依赖关系被容器所使用的配置元数据所反射。

## 3.2 容器概述

The interface `org.springframework.context.ApplicationContext` represents the Spring IoC container and is responsible for instantiating, configuring, and assembling the aforementioned beans. The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata. The configuration metadata is represented in XML, Java annotations, or Java code. It allows you to express the objects that compose your application and the rich interdependencies between such objects.

Several implementations of the `ApplicationContext` interface are supplied out-of-the-box with Spring. In standalone applications it is common to create an instance of `ClassPathXmlApplicationContext` or `FileSystemXmlApplicationContext`. While XML has been the traditional format for defining configuration metadata you can instruct the container to use Java annotations or code as the metadata format by providing a small amount of XML configuration to declaratively enable support for these additional metadata formats.

In most application scenarios, explicit user code is not required to instantiate one or more instances of a Spring IoC container. For example, in a web application scenario, a simple eight (or so) lines of boilerplate web descriptor XML in the `web.xml` file of the application will typically suffice (see [Section 3.15.4, “Convenient ApplicationContext instantiation for web applications”](#)). If you are using the [Spring Tool Suite](#) Eclipse-powered development environment this boilerplate configuration can be easily created with few mouse clicks or keystrokes.

The following diagram is a high-level view of how Spring works. Your application classes are combined with configuration metadata so that after the `ApplicationContext` is created and initialized, you have a fully configured and executable system or application.

接口 `org.springframework.context.ApplicationContext` 表示 Spring IoC 容器，并负责实例化，配置和组合上述 `bean`。容器通过读取配置元数据获取关于要实例化，配置和组合的对象的指令。配置元数据以 `XML`，`Java注释` 或 `Java代码` 表示。它允许你表达组成你的应用程序的对象和这些对象之间丰富的相互依赖。

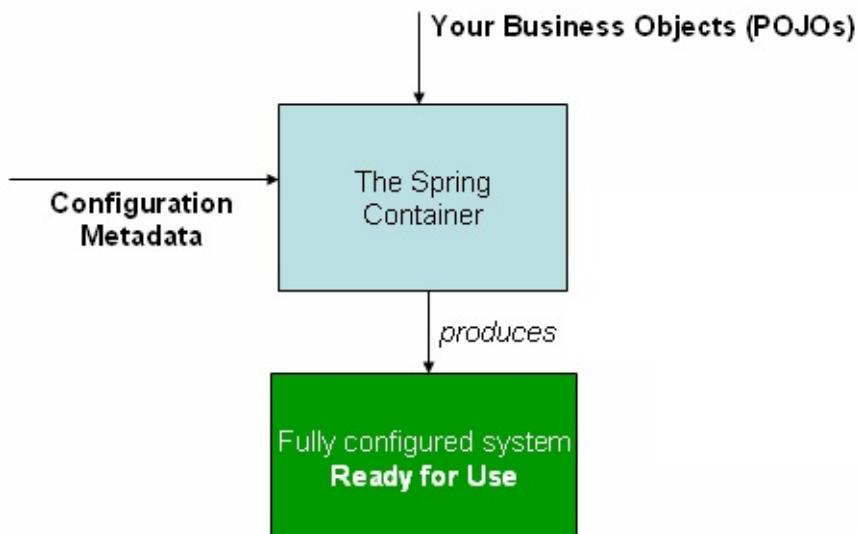
`ApplicationContext` 接口的几个实现是 Spring 提供的开箱即用的。在独立应用程序中，通常创建一个 `ClassPathXmlApplicationContext` 或 `FileSystemXmlApplicationContext` 的实例。尽管 `XML` 是定义配置元数据的传统格式，但您可以通过提供少量的 `XML配置` 来声明性地支持这些额外的元数据格式，从而指示容器使用 `Java注释` 或 `代码` 作为元数据格式。

在大多数应用场景中，不需要显式用户代码来实例化 `Spring IoC` 容器的一个或多个实例。例如，在 `Web` 应用程序场景中，应用程序的 `web.xml` 文件中的一个简单的样板网站的 `XML` 描述符通常就足够了（请参见第3.15.4节“便捷的 `ApplicationContext` 实例化 Web 应用程序”）。如果

您使用的是 Eclipse 工具套件 Eclipse 开发环境，那么只需点击鼠标或敲击即可轻松创建此样板配置。

下图是 Spring 如何工作的高级视图。您的应用程序类与配置元数据相结合，以便在创建和初始化 ApplicationContext 之后，您具有完全配置和可执行的系统或应用程序。

**Figure 3.1.Spring IoC 容器**



### 3.2.1 配置元数据

As the preceding diagram shows, the Spring IoC container consumes a form of *configuration metadata*; this configuration metadata represents how you as an application developer tell the Spring container to instantiate, configure, and assemble the objects in your application.

Configuration metadata is traditionally supplied in a simple and intuitive XML format, which is what most of this chapter uses to convey key concepts and features of the Spring IoC container.



XML-based metadata is *not* the only allowed form of configuration metadata. The Spring IoC container itself is *totally* decoupled from the format in which this configuration metadata is actually written. These days many developers choose [Java-based configuration](#) for their Spring applications.

For information about using other forms of metadata with the Spring container, see:

- [Annotation-based configuration](#): Spring 2.5 introduced support for annotation-based configuration metadata.
- [Java-based configuration](#): Starting with Spring 3.0, many features provided by the Spring JavaConfig project became part of the core Spring Framework. Thus you can define beans external to your application classes by using Java rather than XML files. To use these new features, see the `@Configuration`, `@Bean`, `@Import` and `@DependsOn` annotations.

Spring configuration consists of at least one and typically more than one bean definition that the container must manage. XML-based configuration metadata shows these beans configured as `elements inside a top-level element`. Java configuration typically uses `@Bean` annotated methods within a `@Configuration class`.

These bean definitions correspond to the actual objects that make up your application. Typically you define service layer objects, data access objects (DAOs), presentation objects such as Struts `Action` instances, infrastructure objects such as Hibernate `SessionFactories`, JMS `Queues`, and so forth. Typically one does not configure fine-grained domain objects in the container, because it is usually the responsibility of DAOs and business logic to create and load domain objects. However, you can use Spring's integration with AspectJ to configure objects that have been created outside the control of an IoC container. See [Using AspectJ to dependency-inject domain objects with Spring](#).

The following example shows the basic structure of XML-based configuration metadata:

如上图所示， Spring IoC 容器使用一种形式的配置元数据；此配置元数据表示你作为应用程序开发人员如何告诉 Spring 容器 如何实例化，配置和组合应用程序中的对象。

配置元数据传统上以简单和直观的 XML 格式 提供，这是本章的大部分用来传达 Spring IoC 容器的关键概念和特性。



基于 XML 的元数据不是配置元数据唯一允许的形式。Spring IoC 容器本身完全与实际写入配置元数据的格式解耦。现在，许多开发人员为他们的 Spring 应用程序选择基于 Java 的配置。

有关在 Spring 容器中使用其他形式的元数据的信息，请参阅：

- 基于注释的配置： Spring 2.5 引入了对基于注解的配置元数据的支持。
- 基于 Java 的配置：从 Spring 3.0 开始，Spring JavaConfig 项目提供的许多功能成为 Spring Framework 核心的一部分。因此，您可以使用 Java 而不是 XML 文件来定义应用程序类外部的 bean。要使用这些新功能，请参阅 @Configuration ， @Bean ， @Importand 和 @DependsOn 注释。

Spring 配置包括容器必须管理的至少一个，通常多个 bean 定义。基于 XML 的配置元数据显示这些 bean 配置为顶级元素内的元素。Java 配置通常在 @ConfigurationClass 中使用 @Bean 注解方法。

这些 bean 定义对应于组成应用程序的实际对象。通常，您定义服务层对象，数据访问对象（DAO），演示对象（如 Struts Action 实例），基础结构对象（如 Hibernate SessionFactories），JMS 队列等。通常， 不会在容器中配置细粒度域对象，因为通常由 DAO 和业务逻辑负责创建和加载域对象。但是，您可以使用 Spring 与 AspectJ 的集成来配置在 IoC 容器 控制之外创建的对象。请参阅 使用 AspectJ 通过 Spring 来依赖注入域对象。

以下示例显示了基于 XML 的配置元数据的基本结构：

```
<?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>
```

The `id` attribute is a string that you use to identify the individual bean definition. The `class` attribute defines the type of the bean and uses the fully qualified classname. The value of the `id` attribute refers to collaborating objects. The XML for referring to collaborating objects is not shown in this example; see [Dependencies](#) for more information.

`id` 属性 是一个字符串，用于标识 `单个bean` 定义。 `class` 属性 定义 `bean` 的类型，并 使用完全限定的类名 。 `id` 属性 的值指 协作对象 (即容器内此 `bean` 的名称)。在此示例中未显示用于引用协作对象的 `XML`；有关详细信息，请参阅依赖关系。

### 3.2.2 实例化容器

Instantiating a Spring IoC container is straightforward. The location path or paths supplied to an `ApplicationContext` constructor are actually resource strings that allow the container to load configuration metadata from a variety of external resources such as the local file system, from the Java `CLASSPATH`, and so on.

实例化Spring IoC容器很简单。提供给一个 `ApplicationContext` 构造函数的位置路径实际上是允许容器从各种外部资源（如本地文件系统，从Java `CLASSPATH` 等）加载配置元数据的资源字符串。

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



在你学习Spring的IoC容器之后，你可能想更多地了解Spring的Resource资源抽象，如[第4章，资源\\*](#)，它提供了从URI语法中定义的位置读取InputStream的方便机制。特别是，Resource路径用于构造应用程序上下文，如[第4.7节“应用程序上下文和资源路径”](#)

以下示例显示服务层对象（`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.PetStore
ServiceImpl">
        <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>
```

以下示例显示数据访问对象 `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.jpa.JpaAccountDao">
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>

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

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

```

◀ ▶

```

</beans>

```

In the preceding example, the service layer consists of the class `PetStoreServiceImpl`, and two data access objects of the type `JpaAccountDao` and `JpaItemDao` (based on the JPA Object/Relational mapping standard). The `property name` element refers to the name of the JavaBean property, and the `ref` element refers to the name of another bean definition. This linkage between `id` and `ref` elements expresses the dependency between collaborating objects. For details of configuring an object's dependencies, see [Dependencies](#). 在前面的例子中，服务层由类 `PetStoreServiceImpl` 和 `JpaAccountDao` 和 `JpaItemDao` 类型的两个数据访问对象（基于JPA对象/关系映射标准）组成。`property name` 元素是指JavaBean属性的名称，`ref` 元素是指另一个bean定义的名称。`id` 和 `ref` 元素之间的链接表示协作对象之间的依赖关系。有关配置对象依赖性的详细信息，请参阅 [Dependencies](#).

## 撰写基于XML的配置元数据

It can be useful to have bean definitions span multiple XML files. Often each individual XML configuration file represents a logical layer or module in your architecture.

You can use the application context constructor to load bean definitions from all these XML fragments. This constructor takes multiple `Resource` locations, as was shown in the previous section. Alternatively, use one or more occurrences of the `<`>` element to load bean definitions from another file or files. For example: 使bean定义跨越多个XML文件可能很有用。通常每个单独的XML配置文件表示您的架构中的逻辑层或模块。

您可以使用应用程序上下文构造函数从所有这些XML片段加载bean定义。这个构造函数需要多个Resource位置，如上一节所示。或者，使用一个或多个出现的`元素从其他文件加载bean定义。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>
```

In the preceding example, external bean definitions are loaded from three files:

`services.xml`, `messageSource.xml`, and `themeSource.xml`. All location paths are relative to the definition file doing the importing, so `services.xml` must be in the same directory or classpath location as the file doing the importing, while `messageSource.xml` and `themeSource.xml` must be in a `resources` location below the location of the importing file. As you can see, a leading slash is ignored, but given that these paths are relative, it is better form not to use the slash at all. The contents of the files being imported, including the top level `元素, must be valid XML bean definitions according to the Spring Schema.

在前面的示例中，外部bean定义从三个文件加

载：`services.xml`, `messageSource.xml` 和 `themeSource.xml`。所有位置路径都与执行导入的定义文件相关，因此 `services.xml` 必须与执行导入的文件位于相同的目录或类路径位置，而 `messageSource.xml` 和 `themeSource.xml` 必须在位于导入文件位置下方的 `resources` 位置。如你所见，前导斜杠被忽略，但是鉴于这些路径是相对的，所以最好不要使用斜杠。被导入的文件的内容，包括顶层`元素，必须是根据Spring Schema的有效XML bean定义。



使用相对“`..`”路径引用父目录中的文件是可能的，但不推荐。这样做会对当前应用程序之外的文件创建依赖关系。特别地，不推荐对“`classpath : URL`”（例如，“`classpath : .. / services.xml`”）使用此引用，其中运行时解析进程选择“最近”类路径根，然后查看其父目录。类路径配置更改可能导致选择不同的不正确目录。您可以始终使用完全限定资源位置而不是相对路径：例如，“`file : C : /config/services.xml`”或“`classpath : / config / services.xml`”。但是，请注意，您正在将应用程序的配置耦合到特定的绝对位置。通常优选为这样的绝对位置保留间接，例如通过在运行时针对JVM系统属性解析的“`$ {...}`”占位符。

### 3.2.3 使用容器

The `ApplicationContext` is the interface for an advanced factory capable of maintaining a registry of different beans and their dependencies. Using the method `T getBean(String name, Class requiredType)` you can retrieve instances of your beans.

The `ApplicationContext` enables you to read bean definitions and access them as follows:

`ApplicationContext` 是一个高级工厂的接口，能够维护不同bean及其依赖关系的注册表。使用方法 `T getBean (String name, Class requiredType)` 你可以检索bean的实例。

`ApplicationContext` 允许你读取bean定义并访问它们，如下所示：

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

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

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

You use `getBean()` to retrieve instances of your beans. The `ApplicationContext` interface has a few other methods for retrieving beans, but ideally your application code should never use them. Indeed, your application code should have no calls to the `getBean()` method at all, and thus no dependency on Spring APIs at all. For example, Spring's integration with web frameworks provides for dependency injection for various web framework classes such as controllers and JSF-managed beans.

你使用 `getBean()` 来检索bean的实例。`ApplicationContext` 接口有一些其他方法来检索bean，但理想情况下你的应用程序代码不应该使用它们。事实上，你的应用程序代码应该根本没有调用 `getBean()` 方法，因此根本没有依赖于Spring API。例如，Spring与Web框架的集成并为各种Web框架类（如控制器和JSF管理的bean）提供了依赖注入。

## 3.3 Bean概述

A Spring IoC container manages one or more *beans*. These beans are created with the configuration metadata that you supply to the container, for example, in the form of XML ``definitions.

Within the container itself, these bean definitions are represented as `BeanDefinition` objects, which contain (among other information) the following metadata:

- A *package-qualified class name*: typically the actual implementation class of the bean being defined.
- Bean behavioral configuration elements, which state how the bean should behave in the container (scope, lifecycle callbacks, and so forth).
- References to other beans that are needed for the bean to do its work; these references are also called *collaborators* or *dependencies*.
- Other configuration settings to set in the newly created object, for example, the number of connections to use in a bean that manages a connection pool, or the size limit of the pool.

This metadata translates to a set of properties that make up each bean definition.

Spring IoC容器管理一个或多个 *beans*。这些bean是使用您提供给容器的配置元数据创建的，例如，以XML“定义”的形式。

在容器本身内，这些bean定义表示为 `BeanDefinition` 对象，它包含（除其他信息之外）以下元数据：

- 包限定类名：通常是定义的bean的实际实现类。
- Bean行为配置元素，它说明了bean在容器中的行为（范围，生命周期回调等等）。
- 引用其他bean的bean需要做的工作；这些引用也称为协作者或依赖性。
- 在新创建的对象中设置的其他配置设置，例如，在管理连接池的bean中使用的连接数，或者池的大小限制。

此元数据转换为构成每个bean定义的一组属性。

<b>Table 3.1. The bean definition   Property   Explained in...     -----</b>
-----   -----    class   <a href="#">Section 3.3.2, “Instantiating beans”</a>    name
<a href="#">Section 3.3.1, “Naming beans”</a>    scope   <a href="#">Section 3.5, “Bean scopes”</a>    constructor
arguments   <a href="#">Section 3.4.1, “Dependency Injection”</a>    properties   <a href="#">Section 3.4.1, “Dependency Injection”</a>
autowiring mode   <a href="#">Section 3.4.5, “Autowiring collaborators”</a>

lazy-initialization mode | [Section 3.4.4, “Lazy-initialized beans”](#) | | initialization method | the section called “Initialization callbacks” | | destruction method | the section called “Destruction callbacks” |

In addition to bean definitions that contain information on how to create a specific bean, the `ApplicationContext` implementations also permit the registration of existing objects that are created outside the container, by users. This is done by accessing the `ApplicationContext`'s `BeanFactory` via the method `getBeanFactory()` which returns the `BeanFactory` implementation `DefaultListableBeanFactory`. `DefaultListableBeanFactory` supports this registration through the methods `registerSingleton(..)` and `registerBeanDefinition(..)`. However, typical applications work solely with beans defined through metadata bean definitions.

除了包含如何创建特定bean的bean定义之外，`ApplicationContext`实现还允许注册由用户在容器外部创建的现有对象。这是通过访问`ApplicationContext`的`BeanFactory`通过方法 `getBeanFactory()` 来实现的，它返回`BeanFactory`实现 `DefaultListableBeanFactory`。  
`DefaultListableBeanFactory` 通过方法 `registerSingleton(..)` 和 `registerBeanDefinition(..)` 支持这种注册。然而，典型的应用程序只能通过元数据定义的 bean 来定义。



Bean元数据和手动提供的单例实例需要尽早注册，以便容器在自动装配和其他自检步骤期间正确地进行推理。虽然在某种程度上支持覆盖现有元数据和现有单例实例，但是在运行时（与动态访问工厂同时）对新bean的注册未被官方支持，并且可能导致并发访问异常和/或bean容器中的不一致状态。

### 3.3.1 命名bean

Every bean has one or more identifiers. These identifiers must be unique within the container that hosts the bean. A bean usually has only one identifier, but if it requires more than one, the extra ones can be considered aliases.

In XML-based configuration metadata, you use the `id` and/or `name` attributes to specify the bean identifier(s). The `id` attribute allows you to specify exactly one id. Conventionally these names are alphanumeric ('myBean', 'fooService', etc.), but may contain special characters as well. If you want to introduce other aliases to the bean, you can also specify them in the `name` attribute, separated by a comma ( , ), semicolon ( ; ), or white space. As a historical note, in versions prior to Spring 3.1, the `id` attribute was defined as an `xsd:ID` type, which constrained possible characters. As of 3.1, it is defined as an `xsd:string` type. Note that bean `id` uniqueness is still enforced by the container, though no longer by XML parsers.

You are not required to supply a name or id for a bean. If no name or id is supplied explicitly, the container generates a unique name for that bean. However, if you want to refer to that bean by name, through the use of the `ref` element or [Service Locator](#) style lookup, you must provide a name. Motivations for not supplying a name are related to using [inner beans](#) and [autowiring collaborators](#).

每个 bean 都有一个或多个标识符。这些标识符在托管bean的容器中必须是唯一的。一个 bean通常只有一个标识符，但是如果它需要多个标识符，那么额外的标识符可以被认为是别名。

在基于XML的配置元数据中，您使用 `id` 和/或 `name` 属性来指定bean标识符。`id` 属性允许你指定一个id。通常这些名称是字母数字的('myBean', 'fooService'等)，但也可能包含特殊字符。如果要向bean引入其他别名，还可以在 `name` 属性中指定它们，用逗号( , )，分号( ; )或空格分隔。作为一个历史记录，在Spring 3.1之前的版本中，'id'属性被定义为一个 `xsd : ID` 类型，它限制了可能的字符。从3.1开始，它被定义为一个 `xsd : string`` 类型。注意bean'id'唯一性仍然由容器强制执行，虽然不再由XML解析器。

您不需要为bean提供名称或ID。如果没有明确提供名称或ID，容器将为该bean生成一个唯一的名称。但是，如果你想通过名称引用那个bean，通过使用 `ref` 元素或[Service Locator](#)样式查找，您必须提供一个名称。不提供名称的动机与使用[内部bean](#)和[自动装配协作者](#)相关。

#### Bean命名约定

The convention is to use the standard Java convention for instance field names when naming beans. That is, bean names start with a lowercase letter, and are camel-cased from then on. Examples of such names would be (without quotes) `'accountManager'` , `'accountService'` , `'userDao'` , `'loginController'` , and so forth.

Naming beans consistently makes your configuration easier to read and understand, and if you are using Spring AOP it helps a lot when applying advice to a set of beans related by name.

约定是在命名bean时使用标准Java约定作为实例字段名称。也就是说，bean名称以小写字母开头，从那时开始是驼峰式的。这样的名称的示例将是（无引号）`accountManager`，`accountService`，`userDao`，`loginController`等。

命名Bean一致地使您的配置更容易阅读和理解，如果您使用Spring AOP，它对按照名称相关的一组bean应用建议时会有很多帮助。



With component scanning in the classpath, Spring generates bean names for unnamed components, following the rules above: essentially, taking the simple class name and turning its initial character to lower-case. However, in the (unusual) special case when there is more than one character and both the first and second characters are upper case, the original casing gets preserved. These are the same rules as defined by `java.beans.Introspector.decapitalize` (which Spring is using here). 通过类路径中的组件扫描，Spring根据上面的规则生成未命名组件的bean名称：基本上，取简单的类名称并将其初始字符转换为小写。然而，在（异常）特殊情况下，当存在多个字符并且第一和第二字符都是大写字母时，原始形式被保留。这些是由 `java.beans.Introspector.decapitalize` (Spring在这里使用) 定义的相同规则

## Aliasing a bean outside the bean definition bean 的别名

In a bean definition itself, you can supply more than one name for the bean, by using a combination of up to one name specified by the `id` attribute, and any number of other names in the `name` attribute. These names can be equivalent aliases to the same bean, and are useful for some situations, such as allowing each component in an application to refer to a common dependency by using a bean name that is specific to that component itself.

Specifying all aliases where the bean is actually defined is not always adequate, however. It is sometimes desirable to introduce an alias for a bean that is defined elsewhere. This is commonly the case in large systems where configuration is split amongst each subsystem, each subsystem having its own set of object definitions. In XML-based configuration metadata, you can use the `<alias/>` element to accomplish this.

在对 bean 定义时，可以通过使用由 `id` 属性指定指定一个唯一的名称外，为了提供多个名称，需要通过 `name` 属性加以指定，所有这个名称都指向同一个bean，在某些情况下提供别名非常有用，例如为了让应用每一个组件都能更容易的对公共组件进行引用。

然而，指定bean实际定义的所有别名并不总是足够的。有时需要为在其他地方定义的bean引入别名。在大型系统中通常是这种情况，其中配置在每个子系统之间分割，每个子系统具有其自己的对象定义集合。在基于XML的配置元数据中，您可以使用 `<alias/>` 元素来实现这一

点。

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

In this case, a bean in the same container which is named `fromName`, may also, after the use of this alias definition, be referred to as `toName`.

For example, the configuration metadata for subsystem A may refer to a DataSource via the name `subsystemA-dataSource`. The configuration metadata for subsystem B may refer to a DataSource via the name `subsystemB-dataSource`. When composing the main application that uses both these subsystems the main application refers to the DataSource via the name `myApp-dataSource`. To have all three names refer to the same object you add to the MyApp configuration metadata the following aliases definitions:

在这种情况下，名为 `fromName` 的同一容器中的bean也可以在使用此别名定义之后称为 `toName`。

例如，子系统A的配置元数据可以通过名称“`subsystemA-dataSource`”引用DataSource。子系统B的配置元数据可以通过名称“`subsystemB-dataSource`”引用DataSource。当编译使用这两个子系统的主应用程序时，主应用程序通过名称 `myApp-dataSource` 引用DataSource。要使所有三个名称引用添加到MyApp配置元数据中的同一对象，以下别名定义：

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

Now each component and the main application can refer to the dataSource through a name that is unique and guaranteed not to clash with any other definition (effectively creating a namespace), yet they refer to the same bean.

现在每个组件和主应用程序可以通过唯一的名称引用dataSource，并且保证不与任何其他定义（有效地创建命名空间）冲突，但它们引用同一个bean。

### 基于 Java 的配置

如果你使用Java配置，`@ Bean` 注解可以用来提供别名，详细信息请看[Section 3.12.3, “Using the @Bean annotation”](#)。

### 3.3.2 Instantiating beans 实例化bean

A bean definition essentially is a recipe for creating one or more objects. The container looks at the recipe for a named bean when asked, and uses the configuration metadata encapsulated by that bean definition to create (or acquire) an actual object.

If you use XML-based configuration metadata, you specify the type (or class) of object that is to be instantiated in the `class` attribute of the `<bean/>` element. This `class` attribute, which internally is a `Class` property on a `BeanDefinition` instance, is usually mandatory. (For exceptions, see the section called “[Instantiation using an instance factory method](#)” and [Section 3.7, “Bean definition inheritance”](#).) You use the `class` property in one of two ways:

- Typically, to specify the bean class to be constructed in the case where the container itself directly creates the bean by calling its constructor reflectively, somewhat equivalent to Java code using the `new` operator.
- To specify the actual class containing the `static` factory method that will be invoked to create the object, in the less common case where the container invokes a `static factory` method on a class to create the bean. The object type returned from the invocation of the `static` factory method may be the same class or another class entirely.

bean定义本质上是创建一个或多个对象的方法。容器在询问时查看命名bean的配方，并使用由该bean定义封装的配置元数据来创建（或获取）实际对象。

如果使用基于XML的配置元数据，则指定要在 `<bean />` 元素的 `class` 属性中实例化的对象的类型（或类）。这个 `class` 属性，在内部是一个 `BeanDefinition` 实例的 `Class` 属性，通常是强制的。（对于异常，请参见“[使用实例工厂方法实例化”一节](#)和[第3.7节“Bean定义继承”](#)。）使用 `class` 属性有两种方法之一：

- 通常，在容器本身通过反射调用其构造函数直接创建bean的情况下指定要构造的bean类，某种程度上等同于使用“new”运算符的Java代码。
- 要指定包含将被调用来创建对象的“static”工厂方法的实际类，类中包含静态方法。从 `static` 工厂方法的调用返回的对象类型可以是完全相同的类或另一个类。

**内部类名.** If you want to configure a bean definition for a `static` nested class, you have to use the *binary* name of the nested class.

For example, if you have a class called `Foo` in the `com.example` package, and this `Foo` class has a `static` nested class called `Bar`, the value of the ‘`class`’ attribute on a bean definition would be...

**内部类名.** 如果你想为一个'static'嵌套类配置bean定义，你必须使用嵌套类的 *binary* 名字。

例如，如果你在 com.example 包中有一个名为 Foo 的类，并且这个 Foo 类有一个叫 Bar 的 static 嵌套类，`class` attribute 的值一个 bean 的定义是...

```
com.example.Foo$Bar
```

Notice the use of the \$ character in the name to separate the nested class name from the outer class name. 注意在名称中使用 \$ 字符来分隔嵌套类名和外部类名。

## Instantiation with a constructor 通过构造函数实例化

When you create a bean by the constructor approach, all normal classes are usable by and compatible with Spring. That is, the class being developed does not need to implement any specific interfaces or to be coded in a specific fashion. Simply specifying the bean class should suffice. However, depending on what type of IoC you use for that specific bean, you may need a default (empty) constructor.

The Spring IoC container can manage virtually *any* class you want it to manage; it is not limited to managing true JavaBeans. Most Spring users prefer actual JavaBeans with only a default (no-argument) constructor and appropriate setters and getters modeled after the properties in the container. You can also have more exotic non-bean-style classes in your container. If, for example, you need to use a legacy connection pool that absolutely does not adhere to the JavaBean specification, Spring can manage it as well.

With XML-based configuration metadata you can specify your bean class as follows:

当你通过构造函数方法创建一个 bean 时，所有正常的类都可以被 Spring 使用并兼容。也就是说，正在开发的类不需要实现任何特定接口或者以特定方式编码。只需指定 bean 类就足够了。但是，根据您用于该特定 bean 的 IoC 的类型，您可能需要一个默认（空）构造函数。

Spring IoC 容器可以管理你想要管理的虚拟任何类；它不限于管理真正的 JavaBean。大多数 Spring 用户喜欢实际的 JavaBeans 只有一个默认的（无参数）构造函数和适当的 setters 和 getters 在容器中的属性之后建模。你也可以在你的容器中有更多异常的非 bean 风格的类。例如，如果您需要使用绝对不遵守 JavaBean 规范的旧连接池，Spring 也可以管理它。

使用基于 XML 的配置元数据，您可以按如下所示指定 bean 类：

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

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

有关向构造函数指定参数（如果需要）和在构建对象后设置对象实例属性的机制的详细信息，请参见 [依赖注入](#)。

## Instantiation with a static factory method 使用静态工厂方法实例化

When defining a bean that you create with a static factory method, you use the `class` attribute to specify the class containing the `static` factory method and an attribute named `factory-method` to specify the name of the factory method itself. You should be able to call this method (with optional arguments as described later) and return a live object, which subsequently is treated as if it had been created through a constructor. One use for such a bean definition is to call `static` factories in legacy code.

The following bean definition specifies that the bean will be created by calling a factory-method. The definition does not specify the type (class) of the returned object, only the class containing the factory method. In this example, the `createInstance()` method must be a `static` method.

当定义一个使用静态工厂方法创建的bean时，除了需要指定 `class` 属性外，还需要通过 `factory-method` 属性来指定创建 bean 实例的工厂方法。Spring 将调用此方法(其可选参数接下来介绍)返回实例对象，就此而言，跟通过普通构造器创建类实例没什么两样。

以下bean定义指定将通过调用factory-method创建bean。该定义不指定返回对象的类型（类），只指定包含工厂方法的类。在这个例子中，`createInstance()` 方法必须是 `static` 方法。

```
<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;
    }
}
```

有关向工厂方法提供（可选）参数和在工厂返回对象后设置对象实例属性的机制的详细信息，请参阅[依赖和配置详解](#)。

## 使用实例工厂方法实例化

Similar to instantiation through a [static factory method](#), instantiation with an instance factory method invokes a non-static method of an existing bean from the container to create a new bean. To use this mechanism, leave the `class` attribute empty, and in the `factory-bean` attribute, specify the name of a bean in the current (or parent/ancestor) container that contains the instance method that is to be invoked to create the object. Set the name of the factory method itself with the `factory-method` attribute.

类似于通过[静态工厂方法](#)实例化，使用实例工厂方法的实例化从容器调用现有bean的非静态方法以创建新bean。要使用此机制，将 `class` 属性保留为空，并在 `factory-bean` 属性中，指定当前（或父/祖先）容器中包含要调用的实例方法的bean的名称 创建对象。使用 `factory-method` 属性设置工厂方法本身的名字。

```
<!-- 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;
    }
}
```

One factory class can also hold more than one factory method as shown here: 一个工厂类也可以有多个工厂方法，如下代码所示：

```
<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;  
    }  
  
}
```

This approach shows that the factory bean itself can be managed and configured through dependency injection (DI). See [Dependencies and configuration in detail](#).



在Spring文档中，*factory bean* 是指在Spring容器中配置的bean，它将通过instance 或static 工厂方法创建对象。相比之下，“FactoryBean”（注意大写）指的是Spring特有的 FactoryBean.

## 3.4 Dependencies 依赖

A typical enterprise application does not consist of a single object (or bean in the Spring parlance). Even the simplest application has a few objects that work together to present what the end-user sees as a coherent application. This next section explains how you go from defining a number of bean definitions that stand alone to a fully realized application where objects collaborate to achieve a goal.

典型的企业应用程序不只包括一个对象（或 spring 语法中的 bean ）。即使是最简单的应用程序也有一些对象，一起工作并最终呈现用户看到的一致性应用程序。下一节将介绍如何从定义多个独立的bean定义到一个完全实现的应用程序，其中对象通过协作来实现一个目标。

### 3.4.1 Dependency Injection 依赖注入

*Dependency injection* (DI) is a process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. The container then *injects* those dependencies when it creates the bean. This process is fundamentally the inverse, hence the name *Inversion of Control* (IoC), of the bean itself controlling the instantiation or location of its dependencies on its own by using direct construction of classes, or the *Service Locator* pattern.

Code is cleaner with the DI principle and decoupling is more effective when objects are provided with their dependencies. The object does not look up its dependencies, and does not know the location or class of the dependencies. As such, your classes become easier to test, in particular when the dependencies are on interfaces or abstract base classes, which allow for stub or mock implementations to be used in unit tests.

DI exists in two major variants, [Constructor-based dependency injection](#) and [Setter-based dependency injection](#).

依赖注入（DI）是一个过程，对象通过构造函数参数，工厂方法的参数或在构造对象实例后设置的属性来定义它们的依赖关系，即它们所处理的其他对象或从工厂方法返回。容器然后在创建bean时注入这些依赖。这个过程基本上是相反的，因此名称为 *Inversion of Control* (IoC)，通过使用类的直接构造或 *Service Locator* 模式来控制其自身的依赖性来实例化或通过调用的bean自身的名称(比如xml里的那些配置，通过名字直接调用到)。

代码与DI原理更清洁，当对象提供其依赖性时，解耦更有效。该对象不查找其依赖关系，并且不知道依赖关系的位置或类。因此，您的类变得更容易测试，特别是当依赖关系在接口或抽象基类上时，这允许在单元测试中使用存根或模拟实现。

DI存在两个主要形式，[基于构造函数的依赖注入](#) 和 [基于Setter的依赖注入](#)。

#### 基于构造方法的依赖注入

*Constructor-based* DI is accomplished by the container invoking a constructor with a number of arguments, each representing a dependency. Calling a `static` factory method with specific arguments to construct the bean is nearly equivalent, and this discussion treats arguments to a constructor and to a `static` factory method similarly. The following example shows a class that can only be dependency-injected with constructor injection. Notice that there is nothing *special* about this class, it is a POJO that has no dependencies on container specific interfaces, base classes or annotations.

基于构造函数的 DI由容器调用具有多个参数的构造函数完成，每个参数表示依赖。调用具有特定参数的 static 工厂方法来构造 bean 几乎是等效的，并且这个结论同样对将参数传递给构造函数和静态工厂方法有效。以下示例显示只能使用构造函数注入进行依赖关系注入的类。注意，这个类没有什么特别的\*，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...

}
```

## 基于**Setter**的依赖注入

*Setter-based DI* is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

The following example shows a class that can only be dependency-injected using pure setter injection. This class is conventional Java. It is a POJO that has no dependencies on container specific interfaces, base classes or annotations.

基于**Setter**的 DI是通过在调用一个无参构造函数或无参数“static”工厂方法来实例化你的bean之后，通过容器调用你的bean上的**setter**方法来完成的。

以下示例显示只能使用纯**setter**注入进行依赖关系注入的类。这个类是常规的Java。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 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...
}

}

```

The `ApplicationContext` supports constructor-based and setter-based DI for the beans it manages. It also supports setter-based DI after some dependencies have already been injected through the constructor approach. You configure the dependencies in the form of a `BeanDefinition`, which you use in conjunction with `PropertyEditor` instances to convert properties from one format to another. However, most Spring users do not work with these classes directly (i.e., programmatically) but rather with XML `bean` definitions, annotated components (i.e., classes annotated with `@Component`, `@Controller`, etc.), or `@Bean` methods in Java-based `@Configuration` classes. These sources are then converted internally into instances of `BeanDefinition` and used to load an entire Spring IoC container instance.

`ApplicationContext` 支持其管理的`bean`的基于构造函数和基于`setter`的DI。它还支持基于`setter`的DI，在一些依赖关系已经通过构造函数方法注入之后。您以一个`BeanDefinition`的形式配置依赖关系，你可以与`PropertyEditor`实例结合使用将属性从一种格式转换为另一种格式。然而，大多数Spring用户不直接使用这些类（即，以编程方式），而是使用XML `bean` 定义，或注解组件（即用`@ Component`，`@ Controller`等注解的类）`@ Bean` 方法在基于Java的`@ Configuration`类中定义。然后通过这些配置在内部转换为`BeanDefinition`的实例，并用于加载整个Spring IoC容器实例。`

### 基于构造函数或基于`setter`的DI？

Since you can mix constructor-based and setter-based DI, it is a good rule of thumb to use constructors for *mandatory dependencies* and setter methods or configuration methods for *optional dependencies*. Note that use of the `@Required` annotation on a setter method can be used to make the property a required dependency.

The Spring team generally advocates constructor injection as it enables one to implement application components as *immutable objects* and to ensure that required dependencies are not `null`. Furthermore constructor-injected components are always returned to client

(calling) code in a fully initialized state. As a side note, a large number of constructor arguments is a *bad code smell*, implying that the class likely has too many responsibilities and should be refactored to better address proper separation of concerns.

Setter injection should primarily only be used for optional dependencies that can be assigned reasonable default values within the class. Otherwise, not-null checks must be performed everywhere the code uses the dependency. One benefit of setter injection is that setter methods make objects of that class amenable to reconfiguration or re-injection later. Management through [JMX MBeans](#) is therefore a compelling use case for setter injection.

Use the DI style that makes the most sense for a particular class. Sometimes, when dealing with third-party classes for which you do not have the source, the choice is made for you. For example, if a third-party class does not expose any setter methods, then constructor injection may be the only available form of DI.

因为你可以混合基于构造函数和基于setter的DI，使用构造函数强制依赖 和setter方法或可选依赖的配置方法是一个好的经验法则。请注意，在设置方法上使用 [@Required](#) 注解可以用于使属性成为必需的依赖关系。

Spring 团队通常倡导构造函数注入，因为它能够将应用程序组件实现为 *immutable* 对象，并确保所需的依赖项不是“null”。此外，构造器注入的组件总是返回到完全初始化状态下的客户端（调用）代码。需要多说一点，大量的构造函数参数是一个坏的代码气味，暗示该类可能有太多的责任，应该重构，以更好地解耦。

Setter注入应主要仅用于可以在类中分配合理默认值的可选依赖项。否则，必须在代码使用依赖性的任何地方执行非空检查。setter注入的一个好处是setter方法使该类的对象可以重新配置或稍后重新注入。因此，通过[JMX MBeans](#)的管理是setter注入的一个让人眼前一亮的例子。

使用对特定类最有意义的DI样式。有时候，当你处理你没有源码的第三方类时，选择你所能做到的。例如，如果第三方类没有公开任何setter方法，则构造函数注入可能是DI的唯一可用形式。

## 依赖解析过程

The container performs bean dependency resolution as follows:

- The `ApplicationContext` is created and initialized with configuration metadata that describes all the beans. Configuration metadata can be specified via XML, Java code, or annotations.
- For each bean, its dependencies are expressed in the form of properties, constructor arguments, or arguments to the static-factory method if you are using that instead of a normal constructor. These dependencies are provided to the bean, *when the bean is actually created*.

- Each property or constructor argument is an actual definition of the value to set, or a reference to another bean in the container.
- Each property or constructor argument which is a value is converted from its specified format to the actual type of that property or constructor argument. By default Spring can convert a value supplied in string format to all built-in types, such as `int`, `long`, `String`, `boolean`, etc.

The Spring container validates the configuration of each bean as the container is created. However, the bean properties themselves are not set until the bean *is actually created*. Beans that are singleton-scoped and set to be pre-instantiated (the default) are created when the container is created. Scopes are defined in [Section 3.5, “Bean scopes”](#). Otherwise, the bean is created only when it is requested. Creation of a bean potentially causes a graph of beans to be created, as the bean’s dependencies and its dependencies’ dependencies (and so on) are created and assigned. Note that resolution mismatches among those dependencies may show up late, i.e. on first creation of the affected bean.

容器对bean依赖性解析过程如下：

- 使用描述所有bean的配置元数据创建和初始化ApplicationContext。配置元数据可以通过XML，Java代码或注解指定。
- 对于每个bean，它的依赖关系以属性，构造函数参数或静态工厂方法的参数的形式表示，如果你使用而不是一个正常的构造函数。这些依赖关系提供给bean，实际创建bean时调用。
- 每个属性或构造函数参数是要设置的值的实际定义，或对容器中另一个bean的引用。
- 作为值的每个属性或构造函数参数从其指定的格式转换为该属性或构造函数参数的实际类型。默认情况下，Spring可以将以字符串格式提供的值转换为所有内置类型，例如 `int`，`long`，`String`，`boolean` 等。

Spring容器在创建容器时验证每个bean的配置。但是，bean属性本身不会设置，直到实际创建的时候调用。在创建容器时创建单例范围并设置为预实例化的Bean（默认值）。范围在[第3.5节“Bean scopes”](#)中定义。否则，仅当请求时才创建bean。创建bean可能导致创建bean的图形，因为bean的依赖关系及其依赖关系（依此类推）被创建和分配。注意，那些依赖关系之间的分辨率不匹配可能迟到，即首次创建受影响的bean时显示。

循环依赖

If you use predominantly constructor injection, it is possible to create an unresolvable circular dependency scenario.

For example: Class A requires an instance of class B through constructor injection, and class B requires an instance of class A through constructor injection. If you configure beans for classes A and B to be injected into each other, the Spring IoC container detects this circular reference at runtime, and throws a `BeanCurrentlyInCreationException`.

One possible solution is to edit the source code of some classes to be configured by setters rather than constructors. Alternatively, avoid constructor injection and use setter injection only. In other words, although it is not recommended, you can configure circular dependencies with setter injection.

Unlike the *typical* case (with no circular dependencies), a circular dependency between bean A and bean B forces one of the beans to be injected into the other prior to being fully initialized itself (a classic chicken/egg scenario).

You can generally trust Spring to do the right thing. It detects configuration problems, such as references to non-existent beans and circular dependencies, at container load-time. Spring sets properties and resolves dependencies as late as possible, when the bean is actually created. This means that a Spring container which has loaded correctly can later generate an exception when you request an object if there is a problem creating that object or one of its dependencies. For example, the bean throws an exception as a result of a missing or invalid property. This potentially delayed visibility of some configuration issues is why `ApplicationContext` implementations by default pre-instantiate singleton beans. At the cost of some upfront time and memory to create these beans before they are actually needed, you discover configuration issues when the `ApplicationContext` is created, not later. You can still override this default behavior so that singleton beans will lazy-initialize, rather than be pre-instantiated.

If no circular dependencies exist, when one or more collaborating beans are being injected into a dependent bean, each collaborating bean is *totally* configured prior to being injected into the dependent bean. This means that if bean A has a dependency on bean B, the Spring IoC container completely configures bean B prior to invoking the setter method on bean A. In other words, the bean is instantiated (if not a pre-instantiated singleton), its dependencies are set, and the relevant lifecycle methods (such as a [configured init method](#) or the [InitializingBean callback method](#)) are invoked.

如果主要使用构造函数注入，可以创建一个不可解析的循环依赖场景。

例如：A类通过构造函数注入需要B类的实例，B类通过构造函数注入需要A类的实例。如果将A和B类的bean配置为彼此注入，则Spring IoC容器在运行时检测到此循环引用，并抛出一个 `BeanCurrentlyInCreationException`。

一个可行的解决方案是编辑要由**setter**而不是构造函数配置的一些类的源代码。或者，避免构造函数注入，并仅使用**setter**注入。换句话说，虽然不推荐，可以使用**setter**注入配置循环依赖性。

与典型情况（没有循环依赖）不同，bean A和bean B之间的循环依赖性迫使一个bean在被完全初始化之前被注入另一个bean（一个经典的鸡/鸡蛋场景）。

你一般可以信任Spring做正确的事情。它在容器加载时检测配置问题，例如引用不存在的bean和循环依赖性。Spring在实际创建bean时尽可能晚地设置属性并解析依赖关系。这意味着，如果在创建该对象或其某个依赖关系时出现问题，则在请求对象时，已正确加载的Spring容器可能稍后会生成异常。例如，bean由于缺少或无效的属性而抛出异常。这可能延迟一些配置问题的可见性是因为ApplicationContext实现这默认情况下预实例化单例bean。以实际需要之前创建这些bean的一些预先时间和内存为代价，您在创建ApplicationContext时不会发现配置问题。您仍然可以覆盖此默认行为，以便单例bean将延迟初始化，而不是预先实例化(其实就是懒加载，真正用到的时候才会发生，提高了系统的性能)。

如果不存在循环依赖性，则当一个或多个协作bean被注入到依赖bean中时，每个协作bean在被注入到依赖bean之前被完全配置。这意味着如果bean A对bean B有依赖性，则Spring IoC容器在调用bean A上的setter方法之前完全配置bean B.换句话说，bean被实例化（如果不是实例化的单例）设置相关性，并调用相关的生命周期方法（如配置的init方法或InitializingBean回调方法）。

## 依赖注入示例

The following example uses XML-based configuration metadata for setter-based DI. A small part of a Spring XML configuration file specifies some bean definitions: 以下示例使用基于XML的配置基于setter的DI。Spring XML配置文件的一小部分指定了一些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;
    }

}

```

In the preceding example, setters are declared to match against the properties specified in the XML file. The following example uses constructor-based DI:

在前面的示例中，**setters**被声明为与XML文件中指定的属性匹配。以下示例使用基于构造函数的DI：

```

<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;
    }

}

```

The constructor arguments specified in the bean definition will be used as arguments to the constructor of the `ExampleBean`.

Now consider a variant of this example, where instead of using a constructor, Spring is told to call a `static` factory method to return an instance of the object:

在bean定义中指定的构造函数参数将被用作 `ExampleBean` 构造函数的参数。

现在考虑这个例子的一个变体，在这里不使用构造函数，Spring被告知要调用一个 `static` 工厂方法来返回一个对象的实例：

```

<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;
    }

}
```

Arguments to the `static` factory method are supplied via undefined `elements`, exactly the same as if a constructor had actually been used. The type of the class being returned by the factory method does not have to be of the same type as the class that contains the `static` factory method, although in this example it is. An instance (non-static) factory method would be used in an essentially identical fashion (aside from the use of the `factory-bean` attribute instead of the `class` attribute), so details will not be discussed here.

`static` 工厂方法的参数是通过未定义的 `<constructor-arg>` 元素提供的，完全和实际使用的构造函数一样。由工厂方法返回的类的类型不必与包含“`static`”工厂方法的类的类型相同，尽管在本例中它是。一个实例（非静态）工厂方法将以基本相同的方式使用（除了使用“`factory-bean`”属性而不是“`class`”属性），因此这里不再讨论细节。

### 3.4.2 Dependencies and configuration in detail 依赖和配置的种种细节

As mentioned in the previous section, you can define bean properties and constructor arguments as references to other managed beans (collaborators), or as values defined inline. Spring's XML-based configuration metadata supports sub-element types within its `<property/>` and `<constructor-arg/>` elements for this purpose.

如上一节所述，您可以将bean属性和构造函数参数定义为对其他托管bean（协作者）的引用，或者作为内联定义的值。Spring通过配置基于XML的元数据来支持它的`<property />`和`<constructor-arg />`元素中的子元素类型。

### Straight values (primitives, Strings, and so on) 直值（基本类型，字符串等）

The `value` attribute of the `<property/>` element specifies a property or constructor argument as a human-readable string representation. Spring's [conversion service](#) is used to convert these values from a `String` to the actual type of the property or argument.

`<property />` 元素的 `value` 属性指定一个属性或构造函数参数作为一个人为可读的字符串表示。Spring的[转换服务](#)用于将这些值从一个 `String` 转换到属性或参数的实际类型。

```
<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>
```

The preceding XML is more succinct; however, typos are discovered at runtime rather than design time, unless you use an IDE such as [IntelliJ IDEA](#) or the [Spring Tool Suite](#) (STS) that support automatic property completion when you create bean definitions. Such IDE assistance is highly recommended.

You can also configure a `java.util.Properties` instance as:

前面的XML更简洁；但是，除非您使用[IntelliJ IDEA](#)或[Spring Tool Suite \(STS\)](#)等IDE，否则会在运行时而不是设计时间(也就是在敲代码的时候)发现打字错误，在创建bean定义时支持自动属性完成。强烈推荐此类IDE帮助。

您还可以将 `java.util.Properties` 实例配置为：

```

<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>

```

The Spring container converts the text inside the `<value/>` element into a `java.util.Properties` instance by using the JavaBeans `PropertyEditor` mechanism. This is a nice shortcut, and is one of a few places where the Spring team do favor the use of the nested `<value/>` element over the `value` attribute style.

Spring容器通过使用JavaBeans的PropertyEditor机制将`<value />`元素内的文本转换为`java.util.Properties`实例。这是一个很好的快捷方式，并且是Spring团队喜欢在“value”属性样式上使用嵌套的`<value />`元素的几个地方之一。

### The `idref` element `idref`元素

The `idref` element is simply an error-proof way to pass the `id` (string value - not a reference) of another bean in the container to a `<constructor-arg/>` or `<property/>` element.

`idref` 元素只是一种将容器中另一个bean的 `id` (字符串值 - 不是引用) 传递给 `<constructor-arg />` 或 `<property />` 元素。

```

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

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

```

The above bean definition snippet is *exactly* equivalent (at runtime) to the following snippet:  
上面的bean定义代码片段与下面的代码片段 完全相同（在运行时）：

```

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

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

```

The first form is preferable to the second, because using the `idref` tag allows the container to validate *at deployment time* that the referenced, named bean actually exists. In the second variation, no validation is performed on the value that is passed to the `targetName` property of the `client` bean. Typos are only discovered (with most likely fatal results) when the `client` bean is actually instantiated. If the `client` bean is a `prototype` bean, this typo and the resulting exception may only be discovered long after the container is deployed.



The `local` attribute on the `idref` element is no longer supported in the 4.0 beans xsd since it does not provide value over a regular `bean` reference anymore. Simply change your existing `idref local` references to `idref bean` when upgrading to the 4.0 schema.

A common place (at least in versions earlier than Spring 2.0) where the `<idref/>` element brings value is in the configuration of [AOP interceptors](#) in a `ProxyFactoryBean` bean definition. Using `<idref/>` elements when you specify the interceptor names prevents you from misspelling an interceptor id.

第一种形式优于第二种形式，因为使用 `idref` 标签允许容器 在部署时 验证被引用的命名bean 实际存在。在第二个变体中，不对传递给 `client` bean 的 `targetName` 属性的值执行验证。当 `client` bean 实际被实例化时，只有发现了 typos (而且可能是致命的结果)。如果 `client` bean 是一个 `prototype` bean，这个打印错误所生成的异常只能在容器部署后很久 才被发现。

#### ! [Note]

`idref` 元素上的 `local` 属性在 4.0 bean xsd 中不再支持，因为它不再提供超过正则 `bean` 引用的值。在升级到 4.0 模式时，只需将现有的“`idref local`”引用更改为“`idref bean`”。

`<idref />` 元素带来的一个常见的价值（至少在 Spring 2.0 之前的版本中）是在 [AOP 拦截器](#) 的配置中。在一个 `ProxyFactoryBean` bean 定义中，当指定拦截器名称时，使用 `<idref />` 元素 可防止拼写拦截器 ID。

## References to other beans (collaborators)

引用其他bean (协作者)

The `ref` element is the final element inside a `<constructor-arg/>` or `<property/>` definition element. Here you set the value of the specified property of a bean to be a reference to another bean (a collaborator) managed by the container. The referenced bean is a dependency of the bean whose property will be set, and it is initialized on demand as needed before the property is set. (If the collaborator is a singleton bean, it may be initialized already by the container.) All references are ultimately a reference to another object. Scoping and validation depend on whether you specify the `id/name` of the other object through the `bean`, `local`, or `parent` attributes.

Specifying the target bean through the `bean` attribute of the `<ref/>` tag is the most general form, and allows creation of a reference to any bean in the same container or parent container, regardless of whether it is in the same XML file. The value of the `bean` attribute may be the same as the `id` attribute of the target bean, or as one of the values in the `name` attribute of the target bean.

`ref` 元素是 `<constructor-arg/>` 或 `<property/>` 定义元素中的最后一个元素。在这里，你将 `bean` 的指定属性的值设置为对容器管理的另一个 `bean`（协作者）的引用。引用的 `bean` 是将设置其属性的 `bean` 的依赖关系，并且在设置属性之前根据需要对其进行初始化。（如果协作者是单例 `bean`，它可能已经由容器初始化。）所有引用最终都是对另一个对象的引用。范围和验证取决于您是通过“`bean`”，“`local`”还是“`parent`”属性指定其他对象的 `id / name`。

通过 `<ref/>` 标签的 `bean` 属性指定目标 `bean` 是最通用的形式，允许创建对同一容器或父容器中的任何 `bean` 的引用，而不管它是否在同一个 XML 文件。“`bean`”属性的值可以与目标 `bean` 的“`id`”属性相同，也可以与目标 `bean` 的“`name`”属性中的值之一相同。

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

Specifying the target bean through the `parent` attribute creates a reference to a bean that is in a parent container of the current container. The value of the `parent` attribute may be the same as either the `id` attribute of the target bean, or one of the values in the `name` attribute of the target bean, and the target bean must be in a parent container of the current one. You use this bean reference variant mainly when you have a hierarchy of containers and you want to wrap an existing bean in a parent container with a proxy that will have the same name as the parent bean .

通过 `parent` 属性指定目标 `bean` 将创建对当前容器的父容器中的 `bean` 的引用。“`parent`”属性的值可以与目标 `bean` 的“`id`”属性相同，也可以与目标 `bean` 的“`name`”属性中的一个值相同，并且目标 `bean` 必须位于父容器中的当前一个。你使用这个 `bean` 引用变体主要是当你有一个容器的层次结构，并且你想用一个与父 `bean` 同名的代理在一个父容器中包装一个现有的 `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>
```



'ref' 元素的 local 属性在 4.0 beans xsd 中不再支持，因为它不再提供超过正则范围 bean 引用的值。在升级到 4.0 模式时，只需将现有的 ref local 引用更改为 ref bean`。

## Inner beans

A `<bean/>` element inside the `<property/>` or `<constructor-arg/>` elements defines a so-called *inner bean*. 元素内的 `<property />` 或 `<constructor-arg />` 元素里面定义了一个所谓的 *inner(内部)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>
```

An inner bean definition does not require a defined id or name; if specified, the container does not use such a value as an identifier. The container also ignores the `scope` flag on creation: Inner beans are *always* anonymous and they are *always* created with the outer bean. It is *not* possible to inject inner beans into collaborating beans other than into the enclosing bean or to access them independently.

As a corner case, it is possible to receive destruction callbacks from a custom scope, e.g. for a request-scoped inner bean contained within a singleton bean: The creation of the inner bean instance will be tied to its containing bean, but destruction callbacks allow it to participate in the request scope's lifecycle. This is not a common scenario; inner beans typically simply share their containing bean's scope.

内部bean定义不需要定义的id或名称; 如果指定, 容器不使用这样的值作为标识符。容器在创建时也忽略 scope 标志: 内部bean 总是匿名的, 它们总是用外部bean创建。将内部bean注入到协作bean中而不是注入到封闭bean中或者独立访问它们是不可能的。

作为一种角落情况, 可以从自定义范围接收销毁回调, 例如。对于包含在单例bean内的请求范围内部bean : 内部bean实例的创建将绑定到其包含的bean , 但销毁回调允许它参与请求范围的生命周期。这不是一个常见的情况; 内部bean通常简单地共享它们被包含bean的范围。

## Collections 集合

In the `<list/>` , `<set/>` , `<map/>` , and `<props/>` elements, you set the properties and arguments of the Java Collection types `List` , `Set` , `Map` , and `Properties` , respectively. 在 `<list/>` , `<set/>` , `<map/>` 和 `<props/>` 元素中, 设置Java Collection 类型 `List` , `Set` , `Map` 和 `Properties` 的属性和参数。

```

<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>

```

The value of a map key or value, or a set value, can also again be any of the following elements:

一个map键或值或set值也可以是以下任何元素：

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

## Collection merging 集合合并

The Spring container also supports the *merging* of collections. An application developer can define a parent-style `<list/>`, `<set/>`, `<map/>` or `<props/>` element, and have child-style `<list/>`, `<set/>`, `<map/>` or `<props/>` elements inherit and override values from the parent collection. That is, the child collection's values are the result of merging the elements of the parent and child collections, with the child's collection elements overriding values specified in the parent collection.

This section on merging discusses the parent-child bean mechanism. Readers unfamiliar with parent and child bean definitions may wish to read the relevant section before continuing.

The following example demonstrates collection merging: Spring 容器还支持集合的合并。一个应用程序开发人员可以定义一个父类型 `<list/>`, `<set/>`, `<map/>` 或者 `<props/>`, `<list/>`<set/>`, `<map/>` 或 `<props/>` 元素继承并覆盖父集合中的值。也就是说，子集合的值是合并父和子集合的元素的结果，子集合元素覆盖父集合中指定的值。

\*本节关于合并讨论了父子bean的机制。不熟悉父和子bean定义的读者可能希望在继续之前阅读相关章节。

以下示例演示集合合并：

```

<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>

```

Notice the use of the `merge=true` attribute on the `<props/>` element of the `adminEmails` property of the `child` bean definition. When the `child` bean is resolved and instantiated by the container, the resulting instance has an `adminEmails`Properties` collection that contains the result of the merging of the child's `adminEmails` collection with the parent's `adminEmails` collection.

注意在 `child` bean 定义的 `adminEmails` 属性的 `<props />` 元素上使用 `merge = true` 属性。当 `child` bean 被容器解析和实例化时，生成的实例会有一个 `adminEmails`Properties` 集合，其中包含子集 `adminEmails` collection 与父集合'adminEmails` 集合的合并结果。

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

The child `Properties` collection's value set inherits all property elements from the parent `<props/>`, and the child's value for the `support` value overrides the value in the parent collection.

This merging behavior applies similarly to the `<list/>`, `<map/>`, and `<set/>` collection types. In the specific case of the `<list/>` element, the semantics associated with the `List` collection type, that is, the notion of an `ordered` collection of values, is maintained; the parent's values precede all of the child list's values. In the case of the `Map`, `Set`, and `Properties` collection types, no ordering exists. Hence no ordering semantics are in effect for the collection types that underlie the associated `Map`, `Set`, and `Properties` implementation types that the container uses internally.

子属性 `Properties` 集合的值集合从父 `<props/>` 继承所有属性元素，`support` 值的子值将覆盖父集合中的值。

这种合并行为类似地适用于 `<list/>`, `<map/>` 和 `<set/>` 集合类型。在 `<list/>` 元素的特定情况下，与 `List` 集合类型相关联的语义，即 `ordered` 集合的值的概念被维护；父级的值在所有子级列表的值之前。在 `Map`, `Set` 和 `Properties` 集合类型的情况下，不存在排序。因此没有排序语义对集合类型有效，这些类型是容器在内部使用的相关联的 `Map`, `Set` 和 `Properties` 实现类型的基础。

### Limitations of collection merging 集合合并的限制

You cannot merge different collection types (such as a `Map` and a `List`), and if you do attempt to do so an appropriate `Exception` is thrown. The `merge` attribute must be specified on the lower, inherited, child definition; specifying the `merge` attribute on a parent collection definition is redundant and will not result in the desired merging.

你不能合并不同的集合类型（例如一个 `Map` 和一个 `List`），如果你试图这样做，一个适当的 `Exception` 被抛出。`merge` 属性必须在下层，继承，子定义上指定；在父集合定义上指定 `merge` 属性是多余的，并且不会导致所需的合并。

### Strongly-typed collection 强类型集合

With the introduction of generic types in Java 5, you can use strongly typed collections. That is, it is possible to declare a `Collection` type such that it can only contain `String` elements (for example). If you are using Spring to dependency-inject a strongly-typed `Collection` into

a bean, you can take advantage of Spring's type-conversion support such that the elements of your strongly-typed `Collection` instances are converted to the appropriate type prior to being added to the `Collection`.

随着在Java 5中引入泛型类型，可以使用强类型集合。也就是说，可以声明一个 `collection` 类型，使得它只能包含 `String` 元素（例如）。如果你使用Spring来将一个强类型的“Collection”依赖注入到一个bean中，你可以利用Spring的类型转换支持，这样强类型的“Collection”实例的元素被转换为适当的类型被添加到 `collection`。

```
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>
```

When the `accounts` property of the `foo` bean is prepared for injection, the generics information about the element type of the strongly-typed `Map` is available by reflection. Thus Spring's type conversion infrastructure recognizes the various value elements as being of type `Float`, and the string values `9.99`, `2.75`, and `3.99` are converted into an actual `Float` type.

当 `foo` bean 的 `accounts` 属性准备注入时，强类型 `Map` 的元素类型的泛型信息可以通过反射来获得。因此，Spring 的类型转换基础设施将各种值元素识别为“`Float`”类型，并且字符串值“`9.99`”、“`2.75`”和“`3.99`”被转换为实际的“`Float`”类型。

## Null and empty string values Null 和空字符串

Spring treats empty arguments for properties and the like as empty `strings`. The following XML-based configuration metadata snippet sets the `email` property to the empty `string` value ("").

Spring将空参数的属性等作为空的字符串处理。以下基于XML的配置元数据片段将email属性设置为空的“String”值（“”）。

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

以上配置等同于以下Java代码：

```
exampleBean.setEmail("")
```

`<null/>` 元素用来处理null值。例如：

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

以上配置等同于以下Java代码：

```
exampleBean.setEmail(null)
```

## XML shortcut with the p-namespace XML使用p命名空间进行简化操作

The p-namespace enables you to use the `bean` element's attributes, instead of nested `<property/>` elements, to describe your property values and/or collaborating beans.

Spring supports extensible configuration formats [with namespaces](#), which are based on an XML Schema definition. The `beans` configuration format discussed in this chapter is defined in an XML Schema document. However, the p-namespace is not defined in an XSD file and exists only in the core of Spring.

The following example shows two XML snippets that resolve to the same result: The first uses standard XML format and the second uses the p-namespace.

p-namespace(命名空间p)使您能够使用 `bean` 元素的属性，而不是嵌套的 `<property/>` 元素来描述属性值和/或协作bean。

Spring支持基于XML的可扩展配置格式[带命名空间](#) 模式定义。本章讨论的 `beans` 配置格式在 XML Schema文档中定义。但是，p命名空间不是在XSD文件中定义的，只存在于Spring的核心。

以下示例显示解析为相同结果的两个XML片段：第一个使用标准XML格式，第二个使用p命名空间。

```
<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>
```

The example shows an attribute in the p-namespace called email in the bean definition. This tells Spring to include a property declaration. As previously mentioned, the p-namespace does not have a schema definition, so you can set the name of the attribute to the property name.

This next example includes two more bean definitions that both have a reference to another bean:

该示例显示了bean定义中名为email的p-namespace(p命名空间)中的属性。这告诉Spring包含一个属性声明。如前所述，p命名空间没有模式定义，因此你可以设置属性的名字作为bean的property的名字。

下一个示例包括两个具有对另一个bean的引用的bean定义：

```

<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>

```

As you can see, this example includes not only a property value using the p-namespace, but also uses a special format to declare property references. Whereas the first bean definition uses `<property name="spouse" ref="jane"/>` to create a reference from bean `john` to bean `jane`, the second bean definition uses `p:spouse-ref="jane"` as an attribute to do the exact same thing. In this case `spouse` is the property name, whereas the `-ref` part indicates that this is not a straight value but rather a reference to another bean.

如您所见，此示例不仅包含使用 p-namespace(p命名空间)的属性值，还使用特殊格式声明属性引用。第一个bean定义使用 `<property name = "spouse" ref = "jane"/>` 创建一个john bean 对jane bean的引用,第二个bean的定义使用了 `p:spouse-ref="jane"` 作为一个属性来做同样的事情。在这种情况下，`spouse` 是属性名，而 `-ref` 部分表示这不是一个直接的值，而是一个对另一个bean的引用。



p-namespace(p命名空间)不如标准XML格式灵活。例如，声明属性引用的格式与以“Ref”结尾的属性冲突(声明属性的引用是以Ref结尾的，采用p命名空间将会产生冲突)，而标准XML格式不会。我们建议您仔细选择您的方法，并将其传达给您的团队成员，以避免生成同时使用所有这三种方法的XML文档。

## XML shortcut with the c-namespace

Similar to the [the section called “XML shortcut with the p-namespace”](#), the `c-namespace`, newly introduced in Spring 3.1, allows usage of inlined attributes for configuring the constructor arguments rather than nested `constructor-arg` elements.

Let's review the examples from the section called "Constructor-based dependency injection" with the `c:` namespace:

类似于“XML shortcut with the p-namespace”这一节，在Spring 3.1中新引入的 `c-namespace` 允许使用内联属性来配置构造函数参数，而不是嵌套 `constructor-arg` 的元素。

让我们回顾一下“基于构造函数的依赖注入”一节中的例子并顺带使用 `c:` 命名空间进行改造：

```

<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.co
m"/>

</beans>

```

The `c:` namespace uses the same conventions as the `p:` one (trailing `-ref` for bean references) for setting the constructor arguments by their names. And just as well, it needs to be declared even though it is not defined in an XSD schema (but it exists inside the Spring core).

For the rare cases where the constructor argument names are not available (usually if the bytecode was compiled without debugging information), one can use fallback to the argument indexes:

`c:` 命名空间使用与 `p:` 一样的约定（用于bean引用的尾部“`-ref`”），用于通过其名称设置构造函数参数。同样，它需要被声明，即使它没有在XSD模式中定义（但它存在于Spring核心内部）。

对于少数情况下构造函数参数名称不可用（通常如果字节码没有调试信息编译），可以使用 fallback 到参数索引：

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



由于XML语法，索引符号需要存在前导 `_`，因为XML属性名称不能以数字开头（即使某些IDE允许它）。

In practice, the constructor resolution [mechanism](#) is quite efficient in matching arguments so unless one really needs to, we recommend using the name notation through-out your configuration.

实际上，构造函数的解析[机制](#)在匹配参数方面是相当高效的，除非一个真正需要，我们建议通过使用名称符号来进行你的配置。

## Compound property names 组合属性名称

You can use compound or nested property names when you set bean properties, as long as all components of the path except the final property name are not `null`. Consider the following bean definition.

您可以在设置bean属性时使用复合或嵌套属性名称，只要路径的所有组件（最终属性名称除外）不为`null`。考虑下面的bean定义：

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

The `foo` bean has a `fred` property, which has a `bob` property, which has a `sammy` property, and that final `sammy` property is being set to the value `123`. In order for this to work, the `fred` property of `foo`, and the `bob` property of `fred` must not be `null` after the bean is constructed, or a `NullPointerException` is thrown.

`foo` bean有一个`fred`属性，`fred`又有一个`bob`属性，`bob`又有一个`sammy`属性，最后的`sammy`属性被设置为值`123`。为了使这个工作，`foo`的`fred`属性和`fred`的`bob`属性在构造bean之后不能是`null`，否则`NullPointerException`会被抛出。

### 3.4.3 Using depends-on

If a bean is a dependency of another that usually means that one bean is set as a property of another. Typically you accomplish this with the `<ref/>` element in XML-based configuration metadata. However, sometimes dependencies between beans are less direct; for example, a static initializer in a class needs to be triggered, such as database driver registration. The `depends-on` attribute can explicitly force one or more beans to be initialized before the bean using this element is initialized. The following example uses the `depends-on` attribute to express a dependency on a single bean:

如果bean是另一个的依赖，通常意味着一个bean被设置为另一个的属性。通常你用`<ref />`元素在XML中配置依赖。然而，有时bean之间的依赖不直接；例如，类的静态块初始化，又比如数据库驱动程序注册。`depends-on`属性可以在使用此元素的bean初始化之前明确强制一个或多个bean被初始化。以下示例使用`depends-on`属性来表示对单个bean的依赖关系：

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

为了实现多个bean的依赖，你可以在`depends-on`中将指定的多个bean名字用分隔符进行分隔，分隔符可以是逗号，空格以及分号等。

```
<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" />
```

 bean定义中的`depends-on`属性可以指定初始化时间依赖性，在`singleton`的情况下只有bean，一个相应的销毁时间依赖。在给定的bean本身被销毁之前，首先销毁定义与给定bean的依赖关系的依赖bean。因此，依赖也可以控制销毁的顺序。

### 3.4.4 Lazy-initialized beans 延迟初始化bean

By default, `ApplicationContext` implementations eagerly create and configure all `singleton` beans as part of the initialization process. Generally, this pre-instantiation is desirable, because errors in the configuration or surrounding environment are discovered immediately, as opposed to hours or even days later. When this behavior is *not* desirable, you can prevent pre-instantiation of a singleton bean by marking the bean definition as lazy-initialized. A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup.

In XML, this behavior is controlled by the `lazy-init` attribute on the `<bean>` element; for example:

默认情况下，`ApplicationContext` 实现急切地创建和配置所有 `singleton` bean 作为初始化过程的一部分。通常，这种预实例化是期望的，因为配置或周围环境中的错误被立即发现，而不是数小时或甚至数天之后。当这种行为不是所期望的，你可以通过将 `bean` 定义标记为延迟初始化来阻止单例 `bean` 的预实例化。一个延迟初始化的 `bean` 告诉 IoC 容器在第一次请求时而不是在启动时创建一个 `bean` 实例。

在 XML 中，此行为由 `<bean/>` 元素上的 `lazy-init` 属性控制；例如：

```
<bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-init="true"/>
<bean name="not.lazy" class="com.foo.AnotherBean"/>
```

When the preceding configuration is consumed by an `ApplicationContext`, the bean named `lazy` is not eagerly pre-instantiated when the `ApplicationContext` is starting up, whereas the `not.lazy` bean is eagerly pre-instantiated.

However, when a lazy-initialized bean is a dependency of a singleton bean that is *not* lazy-initialized, the `ApplicationContext` creates the lazy-initialized bean at startup, because it must satisfy the singleton's dependencies. The lazy-initialized bean is injected into a singleton bean elsewhere that is not lazy-initialized.

You can also control lazy-initialization at the container level by using the `default-lazy-init` attribute on the `<beans/>` element; for example:

当前面的配置被一个 `ApplicationContext` 消费时，名为 `lazy` 的 `bean` 在 `ApplicationContext` 启动时不会被预先实例化，而 `not.lazy` `bean` 被预先实例化。

但是，当一个延迟初始化的 `bean` 是单例 `bean` 的依赖，而这个单例 `bean` 又不是 `lazy` 初始化时，`ApplicationContext` 在启动时创建延迟初始化的 `bean`，因为它必须满足 `singleton` 的依赖。因此延迟加载的 `bean` 会被注入单例 `bean`。

您还可以通过使用 `<bean />` 元素上的 `default-lazy-init` 属性在容器级别控制延迟初始化；例如：

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

### 3.4.5 Autowiring collaborators 自动装配协作者

The Spring container can *autowire* relationships between collaborating beans. You can allow Spring to resolve collaborators (other beans) automatically for your bean by inspecting the contents of the `ApplicationContext`. Autowiring has the following advantages:

- Autowiring can significantly reduce the need to specify properties or constructor arguments. (Other mechanisms such as a bean template [discussed elsewhere in this chapter](#) are also valuable in this regard.)
- Autowiring can update a configuration as your objects evolve. For example, if you need to add a dependency to a class, that dependency can be satisfied automatically without you needing to modify the configuration. Thus autowiring can be especially useful during development, without negating the option of switching to explicit wiring when the code base becomes more stable.

When using XML-based configuration metadata [2], you specify autowire mode for a bean definition with the `autowire` attribute of the `<bean/>` element. The autowiring functionality has four modes. You specify autowiring *per bean* and thus can choose which ones to autowire.

Spring容器可以自动装配协作bean之间的关系。你可以允许Spring通过检查`ApplicationContext`的内容来自动为你的bean解析协作者（其他bean）。自动装配有以下优点：

- 自动装配可以显着减少指定属性或构造函数参数的需求。（其他机制，如bean模板[在本章其他地方讨论](#)在这方面也很有价值。）
- 自动装配可以在您的对象发生变化时更新配置。例如，如果您需要向类添加依赖关系，则可以自动满足该依赖关系，而无需修改配置。因此，自动依赖在开发期间可以是特别有用的，当系统趋于稳定时改为显式装配。

当使用基于 XML 的配置元数据[2]时，您指定autowire“属性的bean定义的自动装配模式。自动装配功能有四种模式。您指定自动装配每个bean，因此可以选择哪些自动装配。

**Table 3.2. Autowiring modes**

Mode	Explanation 模式解释
no	(默认) 无自动装配。Bean引用必须通过 <code>ref</code> 元素定义。对于较大型部署，不建议更改默认设置，因为明确指定协作者会提供更好控制和清晰度。在一定程度上，它记录了系统的结构。
byName	按属性名称自动装配。Spring查找与需要自动注入的属性同名的bean。例如，如果bean定义设置为 <code>autowire by name</code> ，并且它包含 <code>master</code> 属性（即它有一个 <code>setMaster(..)</code> 方法），Spring会查找名为 <code>master</code> 的bean 定义，并使用它来设置属性。
byType	允许属性在属性类型中只有一个bean存在于容器中时自动连接。如果存在多个，则会抛出致命异常，这表示您不能对该bean使用 <code>byType</code> autowiring。如果没有匹配的bean，什么都不发生；该属性未设置。
constructor	类似于 <code>byType</code> ，但适用于构造函数参数。如果在容器中没有找到与构造器参数类型一致的bean，则会抛出异常。

With `byType` or `constructor` autowiring mode, you can wire arrays and typed-collections. In such cases *all* autowire candidates within the container that match the expected type are provided to satisfy the dependency. You can autowire strongly-typed Maps if the expected key type is `String`. An autowired Maps values will consist of all bean instances that match the expected type, and the Maps keys will contain the corresponding bean names.

You can combine autowire behavior with dependency checking, which is performed after autowiring completes.

使用 `byType` 或构造函数自动装配模式，可以应用于数组和类型集合。在这种情况下，提供容器中所有匹配的自动装配对象将被应用于满足各种依赖。如果预期的键类型为“String”，则可以自动注入强类型的Map。一个自动装配的Map value值将包含与预期类型匹配的所有Bean实例。

你可以结合自动装配和依赖检查，后者将会在自动装配完成之后进行。

In most application scenarios, most beans in the container are [singletons](#). When a singleton bean needs to collaborate with another singleton bean, or a non-singleton bean needs to collaborate with another non-singleton bean, you typically handle the dependency by defining one bean as a property of the other. A problem arises when the bean lifecycles are different. Suppose singleton bean A needs to use non-singleton (prototype) bean B, perhaps on each method invocation on A. The container only creates the singleton bean A once, and thus only gets one opportunity to set the properties. The container cannot provide bean A with a new instance of bean B every time one is needed.

A solution is to forego some inversion of control. You can [make bean A aware of the container](#) by implementing the `ApplicationContextAware` interface, and by [making a `getBean\("B"\)` call to the container](#) ask for (a typically new) bean B instance every time bean A needs it. The following is an example of this approach:

在大多数应用场景中，容器中的大多数bean都是 [singletons](#)。当单例bean需要与另一个单例bean协作或非单例bean需要与另一个非单例bean协作时，通常通过将一个bean定义为另一个的属性来处理依赖关系。不过对于具有不同生命周期的bean来说这样做就会出现问题。假设单例bean A需要使用非单例（原型）bean B，也许在A上的每个方法调用上。容器仅创建单例bean A一次，因此只有一次机会来设置属性。这样就没办法在需要的时候每次让容器为bean A提供一个新的bean B实例。

解决方案是放弃一些控制的反转。您可以通过实现以下操作来让 `bean A` 知道容器 `ApplicationContextAware` 接口，并通过[对容器调用`getBean \("B"\)`](#) 在每次bean A需要它时请求（一个通常是新的）bean B实例。以下是此方法的示例：

```
// 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;
    }
}
```

The preceding is not desirable, because the business code is aware of and coupled to the Spring Framework. Method Injection, a somewhat advanced feature of the Spring IoC container, allows this use case to be handled in a clean fashion.

You can read more about the motivation for Method Injection in [this blog entry](#).

前面是不可取的，，因为业务代码和Spring框架产生的耦合。方法注入，作为Spring IoC容器的高级特性，，允许以干净的方式处理这个用例。

你可以在这个博客查看阅读更多关于方法注入的动机

## Lookup method injection

查找方法注入具有使容器覆盖受容器管理的bean 上的方法的能力，从而可以返回容器中另一个命名bean的查找结果。查找方法注入适用于原型bean，如前一节中所述的场景。Spring框架通过使用从CGLIB库生成的字节码来动态生成覆盖该方法的子类来实现此方法注入。



为了使这个动态子类化起作用，Spring bean容器将继承的类不能是 `final`，被重写的方法也不能是 `final`。对一个具有 `abstract` 方法的类进行单元测试需要你为其写一个子类并提供该抽象方法的实现。`Concrete(具体)`方法对于组件扫描也是必要的，这需要具体的类来拾取。另一个关键的限制是，查找方法将不能使用工厂方法，特别是在配置类中不使用 `@Bean` 方法，因为容器不负责在这种情况下创建实例，因此不能即时创建运行时生成的子类。

看看前面代码片段中的 `CommandManager` 类，你会看到Spring容器将动态覆盖 `createCommand()` 方法的实现。你的 `CommandManager` 类不会有任何Spring依赖，在下面重写的例子中可以看出：

```
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();
}
```

在包含要注入的方法（在这种情况下为“`CommandManager`”）的客户端类中，要注入的方法需要具有以下形式的签名：

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

如果方法是“抽象的”，动态生成的子类实现该方法。否则，动态生成的子类将覆盖原始类中定义的具体方法。例如：

```

<!-- a stateful bean deployed as a prototype (non-singleton) -->
<bean id="myCommand" 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="myCommand"/>
</bean>

```

标识为 `commandManager` 的bean在需要 `myCommand` bean的新实例时，将调用自己  
的 `createCommand()` 方法。将 `myCommand` bean 设置成`prototype`，如果这是实际上需要的话，  
一定要谨慎处理。如果它是一个 `singleton`，那么每次将返回相同的 `myCommand` bean。

或者，在基于注解的组件模型中，您可以通过“`@Lookup`”注释声明一个查找方法：

```

public abstract class CommandManager {

    public Object process(Object commandState) {
        Command command = createCommand();
        command.setState(commandState);
        return command.execute();
    }

    @Lookup("myCommand")
    protected abstract Command createCommand();
}

```

或者，更常用点，你可以依赖于目标bean根据查找方法的声明返回类型得到解决：

```

public abstract class CommandManager {

    public Object process(Object commandState) {
        MyCommand command = createCommand();
        command.setState(commandState);
        return command.execute();
    }

    @Lookup
    protected abstract MyCommand createCommand();
}

```

注意，您通常会使用具体的子类实现来声明这种带注解的查找方法，以使它们与Spring的组件  
扫描规则兼容，默认情况下会忽略抽象类。此限制不适用于显式注册或显式导入的bean类的  
情况。



另一种访问不同范围的目标bean的方法是 `objectFactory / Provider` 注入点。查看[the section called “Scoped beans as dependencies”](#)。感兴趣的读者也可以找到 `ServiceLocatorFactoryBean`（在 `org.springframework.beans.factory.config` 包中）来用一用。

## 任意方法替换

与查找方法注入相比，很少有用的方法注入形式是使用另一个方法实现替换托管bean中的任意方法的能力。用户可以安全地跳过本节的其余部分，直到实际需要该功能。

使用基于XML的配置元数据，您可以使用“`replaced-method`”元素将现有的方法实现替换为另一个已部署的bean。考虑下面的类，使用方法`computeValue`，我们要覆盖：

```
public class MyValueCalculator {

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

    // some other methods...

}
```

实现 `org.springframework.beans.factory.support.MethodReplacer` 接口的类提供了新的方法定义。

```
/**
 * 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 ...;
    }
}
```

下面的bean定义中指定了要配置的原始类和将要重写的方法：

```
<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"/>
```

您可以在 `<replaced-method/>` 元素中使用一个或多个包含的 `<arg-type/>` 元素来指示要覆盖的方法的方法签名。仅当方法重载并且类中存在多个变量时，参数的签名才是必需的。为了方便起见，参数的类型字符串可以是完全限定类型名称的子字符串。例如，以下所有匹配 `java.lang.String`：

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

因为参数的数量通常足以区分每个可能的选择，这个简写方式可以节省大量的输入，允许你只需要输入最短字符串就可匹配参数类型。

## 3.5 bean作用域

当创建bean定义时，您将创建一个 *recipe* 用于创建由该bean所定义的类的实际实例。bean 定义是一个配方(模型)的想法很重要，因为它意味着，和一个类一样，您可以从单个配方(模型)创建多个对象实例。

你不仅可以控制要插入到从特定bean定义创建的对象的各种依赖性和配置值，还可以控制从特定bean定义创建的对象的 *scope*。这种方法是强大和灵活的，因为您可以选择通过配置创建的对象的作用域，而无需在代码层面去控制。bean可以被定义为部署在多个作用域之一：开箱即用，Spring框架支持六个作用域，其中五个作用域(这里就和之前的文档有区别了，可以做相应的对比)仅在使用Web感知的 ApplicationContext 时可用。

以下作用域是开箱即用的。您还可以创建[自定义作用域](#)

**Table 3.3. Bean scopes**

Scope	Description
singleton	默认的。一个bean定义，在一个IoC容器内只会产生一个对象。
prototype	一个bean定义会产生多个对象实例
request	一个bean定义产生的bean生命周期为一个HTTP请求；也就是，每一个HTTP请求都会根据bean定义产生一个对象实例。该作用域只有在Spring web ApplicationContext 上下文环境中才有效。
session	产生的bean生命周期在HTTP 会话期间。该作用域只有在Spring web ApplicationContext 上下文环境中才有效
application	将单个bean定义范围限定为ServletContext的生命周期。该作用域只有在Spring web ApplicationContext 上下文环境中才有效
websocket	将单个bean定义范围限定为WebSocket的生命周期。该作用域只有在Spring web ApplicationContext 上下文环境中才有效



从Spring 3.0开始，线程范围可用，但默认情况下未注册。有关更多信息，请参阅 [SimpleThreadScope](#) 的文档。有关如何注册此或任何其他自定义范围的说明，请参阅[使用自定义范围](#)一节。

### 3.5.1 单例作用域

单例bean只会产生一个实例，并且所有对具有与该bean定义匹配的id或name的bean的请求Spring容器都只会返回一个实例。

换句话说，当你定义一个bean定义，并且作为一个singleton，Spring IoC容器创建该bean所定义的对象的一个实例。这个单个实例存储在这样的单例bean的缓存中，所有后续的请求和引用返回缓存的这个对象。□

Spring的单例bean的概念不同于Singleton模式，和在Gang of Four (GoF) 模式的书中定义的不同地方。GoF Singleton硬编码对象的范围，使得每一个类加载器内会产生单例类的一个实例。Spring单例的作用域恰如其名：一个容器对应一个bean。这意味着如果你在一个Spring容器中为一个特定的类定义了一个bean，那么Spring容器将创建一个实例，并且只有一个由该bean定义定义的类的实例。*singleton scope* (单例作用域)是Spring 中的默认配置。要将bean定义为XML中的单例，您可以编写，例如：

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

<!-- 和下面的写法等价，因为单例作用域是默认的 -->
<bean id="accountService" class="com.foo.DefaultAccountService" scope="singleton"/>
```

### 3.5.2 The prototype scope

设置bean作用域为prototype，就是非单例，bean部署的prototype scope导致每次对该特定bean的请求时都会创建新的bean实例。也就是说，bean被注入到另一个bean中，或者通过容器上的getBean()方法调用来请求它。通常，对所有有状态bean使用prototype scope，对无状态bean使用singleton scope。

下图说明了Spring prototype scope。数据访问对象（DAO）通常不被配置为prototype scope，因为通常DAO不持有任何会话状态；

接下来看看如何在XML中定义prototype bean：

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

与其他作用域相比，Spring不管理prototype bean的完整生命周期：容器实例化，配置和其他方式组装原型对象，并将其传递给客户端，没有原型实例的进一步记录。因此，尽管初始化生命周期回调方法在所有对象上被调用，不管范围如何，在prototype的情况下，配置的销毁生命周期回调被不调用。客户端代码必须清理prototype作用域对象，并释放prototype bean持有的昂贵资源。要使Spring容器释放prototype作用域的bean所拥有的资源，请尝试使用自定义bean post-processor，它持有需要被清理bean的引用。

在某些方面，Spring容器在prototype作用域bean中的作用是Java“new”运算符的替代。经由该点的所实例化的bean的所有生命周期管理必须由客户端处理。（有关Spring容器中bean的生命周期的详细信息，请参见[第3.6.1节“生命周期回调”](#)。）

### 3.5.3 Singleton beans 与 prototype-bean 依赖关系

当单例类依赖了原型类，要知道依赖在单例类初始化的时候就已经注入好了。因此，如果您将一个原型范围的bean依赖注入到一个单例范围的bean中，一个新的原型bean被实例化，然后依赖注入到单例bean中。原型实例是曾经提供给单例作用域bean的唯一实例。

但是，假设您希望单例范围的bean在运行时重复获取原型作用域bean的新实例。您不能将原型作用域bean依赖注入到单例bean中，因为当Spring容器实例化单例bean并解析和注入其依赖性时，该注入只发生一次。如果您在运行时需要多次获取新的原型bean实例，请参见第[3.4.6节“方法注入”](#)

### 3.5.4 Request, session, application, and WebSocket scopes

request,session,global session作用域，只有在spring web ApplicationContext的实现中(比如 XmlWebApplicationContext)才会起作用，若在常规Spring IoC容器中使用，比如 ClassPathXmlApplicationContext中，就会收到一个异常IllegalStateException来告诉你不能识别的bean作用域

#### 初始化web配置

为了支持 request , session , application 和 websocket 级别（web-scoped bean）的 bean的作用域，在定义bean之前需要一些小的初始配置。（Spring标准作用域，包括 singleton 和 prototype ，这个初始设置不是必须要配置的。）

如何配置要根据具体的 Servlet 环境

如果你在Spring Web MVC中访问作用域bean，实际上，在Spring DispatcherServlet 处理的请求中，则不需要特殊的设置：DispatcherServlet 已经暴露了所有相关的信息。

如果你使用一个Servlet 2.5 web容器，请求在Spring的 dispatcherServlet 之外处理（例如，当使用JSF或Struts时），你需要注

册 org.springframework.web.context.request.RequestContextListener ``ServletRequestListener 。对于Servlet 3.0+，这可以通过 WebApplicationInitializer 接口以编程方式完成。或者，对于较旧的容器，将以下声明添加到Web应用程序的 web.xml 文件中：

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

或者，如果你的listener设置有问题，请考虑使用Spring的 RequestContextFilter 。过滤器映射取决于周围的Web应用程序配置，因此必须根据需要进行更改。

```
<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`, and `RequestContextFilter` all do exactly the same thing, namely bind the HTTP request object to the `Thread` that is servicing that request. This makes beans that are request- and session-scoped available further down the call chain. `DispatcherServlet`, `RequestContextListener` 和 `RequestContextFilter` 都做同样的事情，即将HTTP请求对象绑定到正在为该请求提供服务的 `Thread` 线程中。这使得相关bean可以共享一个相同请求和会话作用域，并在调用链中进一步可用。

## Request scope

考虑下面这种bean定义：

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

Spring容器通过对每个HTTP请求使用 `loginAction` bean定义来创建一个 `LoginAction` bean的新实例。也就是说，`loginAction` bean的作用域是在HTTP请求级别。您可以根据需要更改创建的实例的内部状态，因为根据此 `loginAciton` bean定义创建的其他bean实例并不会看到这些状态的改变；他们为各自的request拥有。当request完成处理，request作用的bean就被丢弃了。

当使用注解驱动组件或Java Config时，`@RequestScope` 注解可以用于将一个组件分配给 request 范围。

```
@RequestScope
@Component
public class LoginAction {
    // ...
}
```

## Session scope

考虑下面这种bean的xml配置定义：

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

Spring容器通过对单个HTTP会话的生命周期使用 userPreferences bean 定义来创建 userPreferences bean 的新实例。换句话说，userPreferences bean 在 HTTP Session 级别有效地作用域。和 request-scoped bean 相类似，可以改变 bean 实例的内部状态，不管 bean 创建了多少实例都可以，要知道，使用相同的 userPreferences 定义创建的其他的 bean 实例看不到这些状态的改变，因为他们都是为各自的 HTTP Session 服务的。当 HTTP Session 最终被丢弃时，被限定为该特定 HTTP Session 的 bean 也被丢弃。

当使用注解驱动组件或 Java Config 时，@SessionScope 注解可用于将一个组件分配给 session 范围。

```
@SessionScope  
@Component  
public class UserPreferences {  
    // ...  
}
```

## Application scope

考虑下面这种bean定义：

```
<bean id="appPreferences" class="com.foo.AppPreferences" scope="application"/>
```

Spring容器通过对整个web应用程序使用 appPreferences bean 定义来创建一个 AppPreferences bean 的新实例。也就是说，appPreferences bean 是在 ServletContext 级别定义的，存储为一个常规的 ServletContext 属性。这有点类似于 Spring 单例 bean，但在两个重要方面有所不同：1、他是每一个 ServletContext 一个实例，而不是 Spring ApplicationContext 范围。2、它是直接暴露的，作为 servletContext 属性，因此可见。

当使用注解驱动组件或 Java Config 时，@ApplicationScope 注解可用于将一个组件分配给 application 作用域。

```
@ApplicationScope  
@Component  
public class AppPreferences {  
    // ...  
}
```

## 不同级别作用域bean之间依赖

Spring IoC容器不仅管理对象（bean）的实例化，还管理协作者（或依赖关系）。如果要将（例如）一个HTTP请求作用域bean注入到较长期作用域的另一个bean中，您可以选择注入一个AOP代理来代替该作用域bean。也就是说，您需要注入一个代理对象，该对象暴露与作用域对象相同的公共接口，但也可以从相关作用域（例如HTTP请求）查找实际目标对象，并将方法调用委托给真实对象。

你也可以在定义为“singleton”的bean之间使用 `<aop:scoped-proxy/>`，然后通过引用一个可序列化的中间代理，因此能够在反序列化时重新获得目标单例bean。

当对 `prototype scope` 的bean 声明 `<aop:scoped-proxy/>` 时，共享代理上的每个方法调用都将导致创建一个新的目标实例，然后将该调用转发给该目标实例。

此外，`scoped`代理不是以生命周期安全的方式从较小作用域访问bean的唯一方法。您也可以简单地将注入点（即构造函数/`setter`参数或自动注入字段）声明

为 `ObjectFactory<MyTargetBean>`，允许一个 `getObject()` 调用在需要时根据所需查找当前实例 - 作为一个扩展的变体，你可以声明 `ObjectProvider`，它提供了几个附加的访问变量，包括 `getIfAvailable` 和 `getIfUnique`。这个JSR-330变体被称为 `Provider`，`Provider` 声明和对应的 `get()` 调用每次检索尝试。有关JSR-330整体的更多详细信息，请参见 [here](#)。

在下面的例子中的配置只有一行，但重要的是要了解背后的“为什么”以及“如何”。

NTOE:加入本人说明(当你访问userService可以不需要访问userPreferences，而你要访问userPreferences必须要先访问userService, 使用了的AOP包装，在cglib生成userPreferences代理的时候先来一层AOP的包装)

```

<?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>

```

要创建这样的代理，您需要将一个子元素 `<aop:scoped-proxy/>` 插入到一个有范围的bean定义中(参见[选择要创建的代理类型](#)和[第38章，基于XML模式的配置部分](#))为什么

在 `request`，`session` 和自定义scope级别限定bean的定义需要 `<aop:scoped-proxy/>` 元素？让我们检查下面的单例bean定义，并将其与您需要为上述scope定义的内容进行对比（注意，下面的 `userPreferences` bean定义是不完整的）。

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

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

在前面的例子中，单例bean `userManager` 被注入对HTTP Session 作用域的bean `userPreferences` 的引用。这里的要点是 `userManager` bean是一个单例：它将被实例化每个容器一次，它的依赖（在这种情况下只有一个，`userPreferences` bean）也只注入一次。这意味着 `userManager` bean将只对完全相同的 `userPreferences` 对象操作，也就是说，它最初注入的对象。

这是不将较短寿命的作用域bean注入到较长寿命的作用域bean时所需的行为，例如将一个HTTP Session 作用域的协作bean作为依赖注入到单例bean中。相反，你需要一个单独的 `userManager` 对象，并且对于HTTP Session 的生命周期，你需要一个特定于所述HTTP会话的 `userPreferences` 对象。因此，容器创建一个对象，暴露与 `UserPreferences` 类完全相同的公共接口（理想情况下，一个对象，是一个 `UserPreferences` 实例），它可以从作用域机制（HTTP请求，`Session` 等）。容器将这个代理对象注入到 `userManager` bean中，它不知道这个 `UserPreferences` 引用是一个代理。在这个例子中，当一个 `UserManager` 实例调用一个依赖注入 `UserPreferences` 对象的方法时，它实际上是在代理上调用一个方法。代理然后从（在这种情况下）HTTP Session 提取真正的 `UserPreferences` 对象，并将方法调用委托给检索的真正的 `UserPreferences` 对象。

因此，当将 `request-` 和 `session-scoped` bean注入协作对象时，您需要以下，正确和完整的配置：

```
<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>
```

### 选择创建代理类型

默认情况下，当Spring容器为标记了 `<aop:scoped-proxy/>` 元素的bean创建一个代理时，将创建一个基于[CGLIB](#)的类代理。



CGLIB代理只拦截公共方法调用！不要在这样的代理上调用非公共方法；它们不会被委派给实际的作用域目标对象。

或者，您可以将Spring容器配置成(分开念)为这些作用域bean创建标准的基于JDK接口的代理，通过为 `<aop:scoped-proxy/>` 的“proxy-target-class”属性的值指定 `false` 元素。使用基于JDK接口的代理意味着在应用程序类路径中不需要其他库来实现此类代理。然而，这也意味着作用域bean的类必须实现至少一个接口，并且注入bean作用域的所有协作者 必须通过它的一个接口引用该bean。

```
<!-- 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>
```

有关选择基于类或基于接口的代理的更多详细信息，请参见[第7.6节“代理机制”](#)。

### 3.5.5 自定义作用域

bean作用域机制是可扩展的；你可以定义自己的作用域，甚至重新定义现有的作用域，虽然后者被认为是不推荐的做法，并且，你不能覆盖内置的 singleton 和 prototype 作用域。

#### 创建自定义作用域

要将自定义作用域集成到Spring容器中，您需要实现本部分中描述的 `org.springframework.beans.factory.config.Scope` 接口。有关如何实现自己的作用域的想法，请参阅Spring框架本身和 [Scope javadocs](#)，它解释了您需要更详细实现的方法。

`Scope` 接口有四种方法来从作用域中获取对象，将它们从作用域中删除，并允许它们被销毁。

下面的方法作用是返回作用域中对象。比如，`session` 作用域的实现，该方法返回 `session-scoped` 会话作用域bean(若不存在，方法创建该bean的实例，并绑定到`session`会话中，用于引用，然后返回该对象)

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

下面的方法作用是从作用域中删除对象。以 `session` 作用域实现为例，方法内删除对象后，会返回该对象，但是若找不到指定对象，则会返回 `null`

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

下面的方法作用是注册销毁回调函数，销毁是指对象销毁或者是作用域内对象销毁。销毁回调的详情请参看[javadocs](#)或者Spring 作用域实现。

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

下面的方法，用于获取作用域会话标识。每个作用域的标识都不一样。比如，`session` 作用域的实现中，标识就是 `session` 标识（应该是指`sessionId`）

```
String getConversationId()
```

#### 使用自定义作用域

在你编写和测试一个或多个自定义 `Scope` 实现之后，你需要让Spring容器意识到你的新作用域。下面的方法是使用Spring容器注册一个新的 `Scope` 的核心方法：

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

这个方法在 `ConfigurableBeanFactory` 接口上声明，在大多数具体的 `ApplicationContext` 实现中都可以使用，它通过 `BeanFactory` 属性与 Spring 一起提供。

`registerScope(..)` 方法的第一个参数是与作用域相关联的唯一名称；Spring 容器本身中这样的名称的例子是 `singleton` 和 `prototype`。`registerScope(..)` 方法的第二个参数是你想要注册和使用的自定义 `Scope` 实现的一个实际实例。

假设你编写了你的自定义 `Scope` 实现，然后像下面那样注册它。



下面的示例使用 Spring Simple 中包含的 `SimpleThreadScope`，但默认情况下未注册。这些指令对于您自己的自定义 `scope` 实现是一样的。

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

然后，创建符合您的自定义 `Scope` 的作用域规则的 `bean` 定义：

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

使用自定义 `Scope` 实现，你不限于作用域的编程(即代码)注册。你也可以使用 `CustomScopeConfigurer` 类来声明性地注册 `Scope`：

```

<?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>

```

当你在 FactoryBean 实现中放置 `<aop:scoped-proxy/>` 时，它是工厂 bean 本身的作用域，而不是从 `getObject()` 返回的对象。

## 3.6 Customizing the nature of a bean

### 3.6.1 生命周期回调函数

要与容器的bean生命周期管理进行交互，可以实现

`Spring InitializingBean` 和 `DisposableBean` 接口。容器为前者调用 `beforePropertiesSet()`，后者调用 `destroy()`，以允许bean在初始化和销毁bean时执行某些操作。



JSR-250 `@PostConstruct` 和 `@PreDestroy` 注解通常被认为是在现代Spring应用程序中接收生命周期回调的最佳实践。使用这些注解意味着你的bean不耦合到Spring特定的接口。有关详细信息，请参见第3.9.8节 `@PostConstruct` 和 `@PreDestroy`。如果你不想使用JSR-250注解，但是你仍然希望删除耦合考虑使用`init-method`和`destroy-method`对象定义元数据。

在内部，Spring框架使用 `BeanPostProcessor` 实现来处理它可以找到并调用适当方法的任何回调接口。如果你需要自定义特性或其他生命周期行为但Spring不提供开箱即用的，你可以自己实现一个 `BeanPostProcessor`。有关详细信息，请参见第3.8节 容器扩展点。

除了初始化和销毁回调之外，Spring管理的对象还可以实现 `Lifecycle` 接口，使得这些对象可以参与由容器自身生命周期驱动的启动和关闭过程。

本节介绍了生命周期回调接口。

#### 初始化回调

`org.springframework.beans.factory.InitializingBean` 接口允许bean在容器已经设置好bean的所有必需属性之后执行初始化工作。`InitializingBean` 接口指定了一个单一的方法：

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

建议不要使用 `InitializingBean` 接口，因为它不必要的将代码耦合到Spring。或者，使用 `@PostConstruct` 注解或指定POJO初始化方法。在基于XML的配置元数据的情况下，你使用 `init-method` 属性来指定具有`void`无参数签名的方法的名称。使用Java配置，你使用 `@Bean` 的 `initMethod` 属性，参见 [接收生命周期回调 一节](#)。例如，以下：

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

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

...和下面作用是完全一样的...

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

```
public class AnotherExampleBean implements InitializingBean {  
  
    public void afterPropertiesSet() {  
        // do some initialization work  
    }  
  
}
```

但不会将代码耦合到Spring。

## 销毁回调

实现 `org.springframework.beans.factory.DisposableBean` 接口允许bean在包含它的容器被销毁时获得回调。`DisposableBean` 接口指定了一个单一的方法:

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

建议你不要使用 `DisposableBean` 回调接口，因为它不必要地将代码耦合到Spring。或者，使用 `@PreDestroy` 注解或指定bean定义支持的通用方法。使用基于XML的配置元数据，你在 `<bean />` 上使用 `destroy-method` 属性。使用Java配置，你使用 `@Bean` 的 `destroyMethod` 属性，参见 [接收生命周期回调一节](#)。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)
    }

}
```

和下面作用是完全一样的：

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

```
public class AnotherExampleBean implements DisposableBean {

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

}
```

但不会将代码耦合到Spring。

□

一个 `<bean>` 元素的 `destroy-method` 属性可以被赋予一个特殊的(推断)值，该值指示 Spring 自动检测特定 bean 类上的一个 `public close` 或 `shutdown` 方法实现 (`java.lang.AutoCloseable` 或 `java.io.Closeable` 因此匹配)。这个特殊的(推断)值也可以在 `<beans>` 元素的 `default-destroy-method` 属性上设置，以将这个行为应用到整个 bean 集合中(参见 [默认初始化 销毁方法](#))注意，这是 Java 配置的默认行为。

## 默认的初始化函数和销毁函数

当你编写不使用 Spring 特定的 `InitializeBean` 和 `DisposableBean` 回调接口的初始化和销毁方法回调时，你通常使用诸如 `init()`，`initialize()`，`dispose()`，等等。理想情况下，此类生命周期回调方法的名称在整个项目中标准化，以便所有开发人员使用相同的方法名称并确保一致性。

你可以配置 Spring 容器来查找命名初始化，并在每个 bean 上销毁回调方法名。这意味着，作为应用程序开发人员，你可以编写应用程序类并使用称为 `init()` 的初始化回调，而无需为每个 bean 定义配置一个 `init-method = "init"` 属性。Spring IoC 容器在创建 bean 时(并且根据前面描述的标准生命周期回调契约)调用该方法。此功能还对初始化和 `destroy` 方法回调强制执行一致的命名约定。

假设你的初始化回调方法名为 `init()`，而 `destroy` 回调方法名为 `destroy()`。你的类将类似于以下示例中的类。

```

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>

```

在顶层 `<beans/>` 元素属性上的 `default-init-method` 属性的存在导致 Spring IoC 容器识别一个名为 `init` 在 `bean` 中的方法作为初始化方法回调。当 `bean` 被创建和组装时，如果 `bean` 类有这样的方法，它会在适当的时间被调用。

你可以使用顶层的 `<bean/>` 元素上的 `default-destroy-method` 属性来类似地配置 `destroy` 方法回调(在 XML 中)。

如果现有的 `bean` 类已经具有与常规方式不同的回调方法，则可以使用 `init-method` 和 `destroy-method` 属性指定(在 XML 中)方法名称来覆盖默认值 `<bean/>` 本身。

Spring 容器保证在提供了所有依赖关系的 `bean` 之后立即调用配置的初始化回调。因此，对原始 `bean` 引用调用初始化回调，这意味着 AOP 拦截器等尚未应用于 `bean`。目标 `bean` 是完全创建的首先，然后一个 AOP 代理(例如)应用它的拦截器链。如果目标 `bean` 和代理是分开定义的，你的代码甚至可以与原始目标 `bean` 交互，绕过代理。因此，将拦截器应用于 `init` 方法是不一致的，因为这样做会将目标 `bean` 的生命周期与其代理/拦截器耦合，并在代码直接与原始目标 `bean` 交互时留下奇怪的语义。

## 结合生命周期机制

从Spring 2.5开始，你有三个选项来控制bean生命周期行为 `InitializingBean` 和 `DisposableBean` 回调接口；自定义 `init()` 和 `destroy()` 方法；和 `@PostConstruct` 和 `@PreDestroy` 注解。你可以组合这些机制来控制给定的bean。



如果为bean配置了多个生命周期机制，并且每个机制都配置了不同的方法名称，则按照下面列出的顺序执行每个配置的方法。但是，如果配置了相同的方法名，例如，对于多个这些生命周期机制，初始化方法的 `init()`，则该方法将执行一次，如前一节所述。

为同一个bean配置的多个生命周期机制，具有不同的初始化方法，调用顺序如下：

- 使用 `@PostConstruct` 注解的方法
- `InitializingBean` 回调接口定义的 `afterPropertiesSet()`
- 自定义配置的 `init()` 方法

销毁方法以相同的顺序调用：

- 用 `@PreDestroy` 注解的方法
- `destroy()` 由 `DisposableBean` 回调接口定义
- 自定义配置的 `destroy()` 方法

## 启动和关闭回调

`Lifecycle` 接口定义了任何具有自己生命周期需求的对象的基本方法(例如启动和停止一些后台进程)：

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

任何Spring管理的对象可以实现该接口。然后，当ApplicationContext本身接收到开始和停止信号时，例如，对于在运行时的停止/重新启动情形，它将级联这些调用到在该上下文中定义的所有 `Lifecycle` 实现。它通过委托给一个 `LifecycleProcessor` 来实现：

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

注意，`LifecycleProcessor` 本身是 `Lifecycle` 接口的扩展。它还添加了两个其他方法来对正在刷新和关闭的上下文做出反应。

 注意，`org.springframework.context.Lifecycle` 接口定义规则只是明确了启动/停止通知的定义，并不意味着在上下文刷新时自动启动。考虑实现 `org.springframework.context.SmartLifecycle` 而不是细粒度控制特定bean的自动启动（包括启动阶段）。此外，请注意，停止通知不能保证在销毁之前：在正常关闭时，所有 `Lifecycle` bean 将首先在传播一般销毁回调之前收到停止通知；然而，在上下文的生命周期中的热刷新或中止的刷新尝试，只有 `destroy` 方法将被调用。

启动和关闭调用的顺序很重要。如果任何两个对象之间存在 `依赖` 关系，则依赖方会在依赖启动之后启动，会在依赖停止之前停止。然而，有时直接依赖性是未知的。你可能只知道某种类型的对象应该在另一种类型的对象之前开始。在这些情况下，`SmartLifecycle` 接口定义了另一个选项，即在它的超级接口 `Phased` 上定义的 `getPhase()` 方法。

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

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

启动时，具有最低层次的对象首先开始，并且在停止时，遵循相反的顺序。因此，实现 `SmartLifecycle` 并且其 `getPhase()` 方法返回 `Integer.MIN_VALUE` 的对象将首先开始，最后停止。若是返回了 `Integer.MAX_VALUE`，那么该方法最后启动最先停止（因为该对象依赖其他 bean 才能运行）。关于 `phase` 的值，常规的并未实现 `SmartLifecycle` 接口的 `Lifecycle` 对象，其值默认为 0。因此，负 `phase` 值表示要在常规 `Lifecycle` 对象之前启动（在常规 `Lifecycle` 对象之后停止），使用正值则恰恰相反。

正如你可以看到由SmartLifecycle定义的stop方法接受一个回调。任何实现必须在该实现的关闭过程完成后调用该回调的 run() 方法。这使得能够在必要时实现异步关闭，因为 LifecycleProcessor interface DefaultLifecycleProcessor 的默认实现将等待到每个阶段内的对象组的超时值以调用该回调。默认每阶段超时为30秒。你可以通过在上下文中定义名为 lifecycleProcessor 的bean来覆盖默认生命周期处理器实例。如果只想修改超时，则定义以下内容就足够了：

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

如上所述， LifecycleProcessor 接口定义了用于刷新和关闭上下文的回调方法。后者将简单地驱动关闭进程，就像显式地调用了 stop() 一样，但是当上下文关闭时，它会发生。另一方面， 刷新 回调启用了 SmartLifecycle bean的另一个功能。当上下文被刷新时(在所有对象被实例化和初始化之后)，该回调将被调用，并且在那时，默认生命周期处理器将检查每个 SmartLifecycle 对象的 isAutoStartup() 方法返回的布尔值。如果 true ，那么该对象将在该点开始，而不是等待上下文或其自己的 start() 方法的显式调用(这和容器刷新不同，标准的容器实现启动不会自动发生)。 phase 值以及任何 depends-on 关系将以与上述相同的方式来确定启动顺序。

## 在非Web应用程序中正常关闭Spring IoC容器

本节仅适用于非Web应用程序。Spring的基于Web的 ApplicationContext 实现已经有代码在相关Web应用程序关闭时优雅地关闭Spring IoC容器。

如果你在非Web应用程序环境中使用Spring的IoC容器；例如，在富客户端桌面环境中；你使用JVM注册关闭挂接。这样做可确保正常关闭并调用单元bean上的相关销毁方法，以便释放所有资源。当然，你仍然必须正确配置和实现这些destroy回调。

要注册一个 shutdown hook，你需要调用在ConfigurableApplicationContext接口上声明的 registerShutdownHook() 方法：

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

public final class Boot {

    public static void main(final String[] args) throws Exception {

        ConfigurableApplicationContext 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...

    }
}
```

## 3.6.2 ApplicationContextAware和BeanNameAware

当一个 ApplicationContext 创建一个实现 org.springframework.context.ApplicationContextAware 接口的对象实例时，该实例提供了对 ApplicationContext 的引用。

```
public interface ApplicationContextAware {
    void setApplicationContext(ApplicationContext applicationContext) throws BeansException;
}
```

因此，bean可以通过 ApplicationContext 接口，通过编程方式操作创建它们的 ApplicationContext ，或者通过转换对这个接口的已知子类的引用，例如 ConfigurableApplicationContext ，这暴露了额外的功能。一个用途是其他bean的程序检索。有时这种能力是有用的;然而，通常你应该避免它，因为它将代码耦合到Spring，并不遵循控制反转风格，其中协作者作为属性提供给bean。 ApplicationContext 的其他方法提供对文件资源的访问，发布应用程序事件和访问 MessageSource 。这些附加功能在[第3.15节，“ApplicationContext的附加功能”](#)

从Spring 2.5开始，autowiring是另一个替代方法来获取对ApplicationContext的引用。“传统的”构造函数和byType的自动装配模式([如第3.4.5节“自动装配协作者”中所述](#))可以分别为构造函数参数或setter方法参数提供类型ApplicationContext的依赖。为了更灵活，包括自动连接字段和多个参数方法的能力，使用新的基于注解的自动装配功能。如果你这样做，ApplicationContext被自动装配到一个字段，构造函数参数或方法参数中，如果相关的字段，构造函数或方法带有@ Autowired注解，那么应该是ApplicationContext类型。有关详细信息，请参见[第3.9.2节“@Autowired”](#)。

当一个 ApplicationContext 创建一个实现 org.springframework.beans.factory.BeanNameAware 接口的类时，该类将提供对其关联对象定义中定义的名称的引用。

```
public interface BeanNameAware {
    void setBeanName(String name) throws BeansException;
}
```

The callback is invoked after population of normal bean properties but before an initialization callback such as InitializingBean afterPropertiesSet or a custom init-method. 回调在普通bean属性的设置之后但在初始化回调之前被调用(本人注:其实这里我也有点懵，估计是在后面

例如里的方法之前优先调用), 例如 `InitializingBean afterPropertiesSet` 或一个自定义init方法。

### 3.6.3 Other Aware interfaces

除了上面讨论的 `ApplicationContextAware` 和 `BeanNameAware`，Spring 提供了一系列 `Aware` 接口，允许 bean 向容器指示他们需要一个基础结构(*Spring API*)依赖。最重要的 `Aware` 接口总结如下 - 作为一般规则，看名字就能知道依赖类型：

**Table 3.4. Aware interfaces**

名称	注入依赖	详情...
ApplicationContextAware	声明 ApplicationContext	Section 3.6.2, “ApplicationContextAware and BeanNameAware”
ApplicationEventPublisherAware	Event publisher of the enclosing ApplicationContext	Section 3.15, “Additional Capabilities of the ApplicationContext”
BeanClassLoaderAware	加载bean的类加载器	Section 3.3.2, “Instantiating beans”
BeanFactoryAware	Declaring BeanFactory	Section 3.6.2, “ApplicationContextAware and BeanNameAware”
BeanNameAware	Name of the declaring bean	Section 3.6.2, “ApplicationContextAware and BeanNameAware”
BootstrapContextAware	Resource adapter BootstrapContext the container runs in. Typically available only in JCA aware Application Contexts	Chapter 28, JCA CCI
LoadTimeWeaverAware	Defined weaver for processing class definition at load time	Section 7.8.4, “Load-time weaving with AspectJ in the Spring Framework”
MessageSourceAware	配置的解决消息的策略（支持参数化和国际化）	Section 3.15, “Additional Capabilities of the ApplicationContext”
NotificationPublisherAware	Spring JMX notification publisher	Section 27.7, “Notifications”
ResourceLoaderAware	Configured loader for low-level access to resources	Chapter 4, Resources
ServletConfigAware	Current ServletConfig the container runs in. Valid only in a web-aware Spring Application Context	Chapter 18, Web MVC framework
ServletContextAware	Current ServletContext the container runs in. Valid only in a web-aware Spring Application Context	Chapter 18, Web MVC framework

再次注意，这些接口的使用将您的代码绑定到Spring API，而不遵循控制反转风格。因此，建议除非有需求的基础bean才使用编程式访问容器。

## 3.7 Spring Bean的继承

bean定义可以包含大量配置信息，包括构造函数参数，属性值和特定于容器的信息，例如初始化方法，静态工厂方法名称等。子bean定义从父定义继承配置数据。子定义可以根据需要覆盖一些值，或添加其他值。使用父和子bean定义可以节省大量的输入。这是模板的一种形式，讲究的就是效率。

如果你以编程方式使用一个 `ApplicationContext` 接口，子bean定义由 `ChildBeanDefinition` 类来表示。大多数用户不在这个级别上使用它们，而是在类 `ClassPathXmlApplicationContext` 中声明性地配置bean定义。当您使用基于XML的配置元数据时，您通过使用 `parent` 属性指定子bean定义，并指定父bean作为此属性的值。

```
<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>
```

子bean定义使用父定义中的bean类(如果没有指定)，但也可以覆盖它。在后一种情况下，子bean类必须与父兼容，也就是说，它必须接受父的属性值。

子bean定义从父级继承作用域，构造函数参数值，属性值和方法覆盖，并具有添加新值的选项。您指定的任何作用域，初始化方法，销毁方法和/或 `static` 工厂方法设置将覆盖相应的父设置。

剩下的设置是 `always` 取自子定义：`depends on`，`autowire`模式，依赖性检查，`singleton`，`lazy init`。

前面的示例通过使用 `abstract` 属性将父bean定义显式标记为抽象。如果父定义没有指定类，则需要将父bean定义明确标记为'abstract'，如下所示：

```
<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bean不能自己实例化，因为它是不完整的，它也被显式标记为“抽象”。当定义是这样的“抽象”时，它只能用作纯模板bean定义，用来作为子定义的父定义。试图使用这样的抽象父bean，通过引用它作为另一个bean的ref属性或使用父bean id执行显式的getBean()调用，返回一个错误。同样，容器的内部 preInstantiateSingletons() 方法忽略定义为抽象的bean定义。

ApplicationContext 默认实例化所有单例。因此，重要的是(至少对于单例bean)如果你有一个(父)bean定义，你打算只使用作为一个模板，并且这个定义指定一个类，你必须确保设置 abstract 属性为 true ，否则应用程序上下文将实际(尝试)预实例化 abstract bean 。

## 3.8 容器扩展点

通常，应用程序开发人员不需要子类化 `ApplicationContext` 实现类。相反，Spring IoC容器可以通过插入特殊集成接口的实现来扩展。接下来的几个部分描述这些集成接口。

### 3.8.1 使用 BeanPostProcessor 定制 bean

`BeanPostProcessor` 接口定义了 `回调方法`，你可以实现它来提供你自己的(或重载容器的默认)实例化逻辑，依赖解析逻辑等等。如果你想在Spring容器完成实例化，配置和初始化bean之后实现一些自定义逻辑，可以插入一个或多个`BeanPostProcessor`实现。

你可以配置多个 `BeanPostProcessor` 实例，你可以通过设置 `order` 属性来控制这些 `BeanPostProcessor` 的执行顺序。只有当 `BeanPostProcessor` 实现了“`Ordered`”接口时，才可以设置此属性；如果你写自己的`BeanPostProcessor`，你应该考虑实现“`Ordered`”接口。有关更多详细信息，请参阅“`BeanPostProcessor`”和“有序”接口的 javadoc。另见下面关于 [“`BeanPostProcessor`的程序化注册”的注解注册 Beanpost 处理器](#)。

`BeanPostProcessor` 操作 `bean`(或对象) 实例；也就是说，Spring IoC 容器实例化一个 `bean` 实例，然后 `BeanPostProcessor` 做他们的工作。`BeanPostProcessor` 的作用域是每个容器。这仅在使用容器层次结构时才会用到这个。如果你在一个容器中定义一个 `BeanPostProcessor`，它将只对该容器中的 `bean` 进行后处理。换句话说，在一个容器中定义的 `bean` 不会被另一个容器中定义的 `BeanPostProcessor` 进行后处理，即使两个容器都是同一层次结构的一部分。要改变实际的 `bean` 定义(即 `blueprint` 定义 `bean`,译者注：应该是指各种配置元数据，比如 `xml`、注解等)，你需要使用一个 `BeanFactoryPostProcessor`，如第3.8.2节“[使用 `BeanFactoryPostProcessor` 定制配置元数据](#)”。

`org.springframework.beans.factory.config.BeanPostProcessor` 接口恰好包含两个回调方法。当这样的类在容器内注册为 `post-processor`，对于由容器创建的每个 `bean` 实例，容器创建所有 `bean` 在容器初始化方法(比如 `InitializingBean` 的 `afterPropertiesSet()` 方法和其他所有的声明的 `init` 方法)和所有 `bean` 初始化回调之前，运行 `post-processor` 回调。`post-processor` 可以对 `bean` 实例采取任何操作，包括完全忽略回调。`bean post-processor` 通常检查回调接口或者可以用代理包装 `bean`。一些 Spring AOP 基础结构类被实现为 `bean post-processor`，以便提供代理包装逻辑。

一个 `ApplicationContext` 自动检测在配置元数据中定义的实现 `BeanPostProcessor` 接口的任何 `bean`。`ApplicationContext` 将这些 `bean` 注册为后处理器，以便稍后在创建 `bean` 时调用它们。`Bean` 后处理器可以像任何其他 `bean` 一样部署在容器中。

注意，当在配置类上使用 `@Bean` 工厂方法声明 `BeanPostProcessor` 时，工厂方法的返回类型应该是实现类本身或至少是 `org.springframework.beans.factory.config.BeanPostProcessor` 接口，清楚地表明该 `bean` 的后处理器性质。否则，`ApplicationContext` 将不能在完全创建它之前通过类型自动检测它。因为 `BeanPostProcessor` 需要尽早的实例化，以便应用于上下文中其他 `bean` 的初始化，所以这种尽早的类型检测是至关重要的。

虽然 BeanPostProcessor 注册的推荐方法是通过 ApplicationContext 自动检测(如上所述)，但是也可以使用 addBeanPostProcessor 方法在 ConfigurableBeanFactory 上注册它们。当需要在注册之前评估条件逻辑，或者甚至在层次结构中跨上下文复制 bean post processors 时，这可能是有用的。注意， BeanPostProcessor s 以编程方式添加 不遵守 Ordered 接口。注册的顺序就是执行的次序。还要注意，“BeanPostProcessor”以编程方式注册，总是在通过自动检测注册之前处理，而不管任何显式排序。

实现 BeanPostProcessor 接口的类是 special 的，并且容器的处理方式不同。所有 BeanPostProcessor 和他们直接引用的 bean 在启动时被实例化，作为 ApplicationContext 的特殊启动阶段的一部分。接下来，所有的 BeanPostProcessors 都会按照次序注册到容器中，在其他 bean 使用 BeanPostProcessors 处理时也会使用此顺序。因为 AOP 的 auto-proxying 自动代理是 BeanPostProcessor 的默认实现，它既不引用 BeanPostProcessors 也不引用其他 bean，不会发生 auto-proxying 自动代理，因此不会有切面织入。对于任何这样的 bean，你应该看到一个信息日志消息：“ Bean foo 不能由所有的 BeanPostProcessor 接口处理(例如：不符合自动代理) ” 注意，如果你有 bean 连接到你的 BeanPostProcessor 使用 autowiring 或 @Resource (可能回退到自动装配)，Spring 可能会在搜索类型匹配依赖关系候选时访问意外的 bean，因此使它们不适合自动代理或他们是其他种类的 bean post-processing 。例如，如果你有一个使用 @Resource 注解的依赖，其中 field/ setter 名称不直接对应于 bean 的声明名称，并且没有使用 name 属性，那么 Spring 将访问其他 bean 以按类型匹配它们。

以下示例显示如何在 ApplicationContext 中写入，注册和使用 BeanPostProcessor 。

## Example: Hello World, BeanPostProcessor-style

第一个例子说明了基本用法。该示例显示了一个自定义的 BeanPostProcessor 实现，它调用每个 bean 的 `toString()` 方法，因为它是由容器创建的，并将结果字符串打印到系统控制台。

下面找到自定义的“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.gr
oovy">
        <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` 是如何定义的。它甚至没有名称，并且因为它是一个bean，它可以像任何其他bean一样依赖注入。(前面的配置也定义了一个由Groovy脚本支持的bean。Spring动态语言支持在标题为[第31章， 动态语言支持](#)。)

以下简单的Java应用程序执行上述代码和配置：

```

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);
    }

}

```

上面应用程序的输出类似于以下内容:

```

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

```

## 示例:RequiredAnnotationBeanPostProcessor

使用回调接口或注解结合自定义的“BeanPostProcessor”实现是扩展Spring IoC容器的常见手段。。。例如Spring的 RequiredAnnotationBeanPostProcessor，是个 BeanPostProcessor 实现类，spring内置，作用是确保Spring bean定义上的带注解的JavaBean属性确实被注入了值。

## 3.8.2 使用 BeanFactoryPostProcessor 定制配置元数据

我们将要看到的下一个扩展点

是“org.springframework.beans.factory.config.BeanFactoryPostProcessor”。这个接口的语义类似于 BeanPostProcessor，其中一个主要区别是：BeanFactoryPostProcessor 对 bean 配置元数据操作；也就是说，Spring IoC 容器允许 BeanFactoryPostProcessor 读取配置元数据，并可能在容器实例化（除 BeanFactoryPostProcessor 之外的任何）bean 之前改变它。

你可以配置多个 BeanFactoryPostProcessor s，你可以通过设置 order property 来控制这些 BeanFactoryPostProcessor 的执行顺序。但是，如果 BeanFactoryPostProcessor 实现了 Ordered 接口，则只能设置此属性。如果你写自己的“BeanFactoryPostProcessor”，你应该考虑实现 Ordered 接口。有关更多详细信息，请参阅 BeanFactoryPostProcessor 和 Ordered 接口的 javadoc。

如果你想改变实际的 bean 实例（即从配置元数据创建的对象），那么你需要使用一个 BeanPostProcessor（如上所述第3.8.1节“BeanPostProcessor”）。虽然在技术上可以使用 BeanFactoryPostProcessor（例如，使用 BeanFactory.getBean() 中的 bean 实例，这样做会导致提前的 bean 实例化，违反标准容器生命周期。这可能导致负面的副作用，如绕过 bean 后处理。此外，BeanFactoryPostProcessor s 的作用域也是在各自的容器内。这仅在使用容器层次结构时才相关。如果你在一个容器中定义一个 BeanFactoryPostProcessor，它只会应用于该容器中的 bean 定义。一个容器中的 Bean 定义不会被另一个容器中的 BeanFactoryPostProcessor 进行后置处理，即使这两个容器都是同一层次结构的一部分。）

当在“ApplicationContext”中声明一个 bean 工厂后置处理器时，它会自动执行，以便将定义容器的配置元数据进行的更改应用。Spring 包括一些预定义的 bean 工厂后置处理器，例如 PropertyOverrideConfigurer 和 PropertyPlaceholderConfigurer。例如，定制的 BeanFactoryPostProcessor 也可以用来注册自定义属性编辑器。

一个 ApplicationContext 自动检测到它里面实现 BeanFactoryPostProcessor 接口的任何 bean 并部署。它使用这些 bean 作为 bean 工厂后置处理器，在适当的时间。你可以像部署其他 bean 一样部署这些后置处理器 bean。（即 ApplicationContext 自动探测 BeanFactoryPostProcessor 接口的实现类。容器使用这些 bean 作为 bean 工厂 post-processors。可以像其他 bean 那样将 post-processor 部署在容器内。）

和 BeanPostProcessor s 一样，你通常不想为延迟初始化配置 BeanFactoryPostProcessor s。如果没有其他 bean 引用 Bean(Factory)PostProcessor，后置处理器将不会被实例化。因此，标记它的延迟初始化将被忽略，并且 Bean(Factory)PostProcessor 将被及时实例化，即使你在 <beans/> 元素的声明上将 default-lazy-init 属性设置为 true。也不行

## Example: the Class name substitution PropertyPlaceholderConfigurer

你使用 `PropertyPlaceholderConfigurer` 将bean的属性值使用标准的Java Properties格式定义在一个单独的文件中。这样可以将应用的自定义环境配置属性隔离出来，例如数据库URL和密码，而不需要修改容器的主XML定义文件或Java 代码文件的复杂性或风险。

考虑以下基于XML的配置元数据片段，其中定义了具有占位符值的“DataSource”。该示例显示从外部 `Properties` 文件配置的属性。在运行时，将一个 `PropertyPlaceholderConfigurer` 应用于将替换DataSource的某些属性的元数据。要替换的值被指定为遵循Ant / log4j / JSP EL 样式的  `${property-name}` 形式的 *placeholder*。

```
<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>
```

在标准java Properties格式文件中实际的值：

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

因此，字符串  `${jdbc.username}` 在运行时被替换为值'sa'，这同样适用于与属性文件中的键匹配的其他占位符值。`PropertyPlaceholderConfigurer` 检查bean定义的大多数属性和属性中的占位符。此外，可以自定义占位符前缀和后缀。

使用Spring 2.5中引入的 `context` 命名空间，可以使用专用配置元素配置属性占位符。一个或多个位置可以在 `location` 属性中以逗号分隔的列表形式提供。

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

`PropertyPlaceholderConfigurer` 不仅在你指定的 `Properties` 文件中寻找属性。默认情况下，如果在指定的属性文件中找不到属性，它还会检查Java“System”属性。你可以通过将 `configurer` 的 `systemPropertiesMode` 属性设置为以下三个支持的整数值之一来定制此行为：

- - never \* (0): 不检查系统属性
- - fallback \* (1): 如果在指定的属性文件中不可解析，请检查系统属性。这是默认值。
- - override \* (2): 首先检查系统属性，然后尝试指定的属性文件。这允许系统属性覆盖任何其他属性源。

有关更多信息，请参阅 `PropertyPlaceholderConfigurer` javadocs。



你可以使用 `PropertyPlaceholderConfigurer` 来替换类名，当你必须在运行时选择一个特定的实现类时，这是有用的。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}" />
```

若类在运行时期间不能解析为合法类，`ApplicationContext` 创建非延迟初始化bean的 `preInstantiateSingletons()` 期间抛错误

## Example: the PropertyOverrideConfigurer

“`PropertyOverrideConfigurer`”是另一个bean工厂后置处理器，类似于`PropertyPlaceholderConfigurer`，但是与后者不同，原始定义可以设置默认值或者根本不设置值。如果重写的“Properties”文件没有特定bean属性的条目，那么就使用默认的上下文定义。

注意，bean定义并不知道它会被重写，所以使用了重写配置在XML配置中并不直观。在多个`PropertyOverrideConfigurer` 实例为同一个bean属性定义不同的值的情况下，最后一个实例配置会生效。

`Properties`文件配置格式如下：

```
beanName.property=value
```

例如：

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

此示例文件可用于包含名为 `dataSource` 的 bean 的容器定义，该 bean 具有 `driver` 和 `url` 属性。

还支持复合属性名称，只要路径(除了覆盖的最终属性)的每个组件都已为 非空 (可能由构造函数初始化)。在此示例中...

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

1. foo bean 的 fred 属性的 bob 属性的 sammy 属性设置为标量值 123。



指定的重写值都是字面值；它们不会转换为 bean 引用。当 XML bean 定义中的原始值指定一个 bean 引用时，此约定也适用。

使用 Spring 2.5 中引入的 `context` 命名空间，可以使用专用配置元素配置属性覆盖：

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

### 3.8.3 使用FactoryBean定制实例化逻辑

对象实现 `org.springframework.beans.factory.FactoryBean` 接口，则成为它本身的工厂。

`FactoryBean` 接口是 Spring IoC 容器实例化逻辑的扩展点。假如初始化代码非常复杂，此时使用 Java 编码比使用 XML 配置更容易表达。这种场景中，你可以创建自己的 `FactoryBean`，在该类中编写复杂的初始化程序，然后将你的自定义 `FactoryBean` 插入到容器。

`FactoryBean` 接口提供了三种方法：

- `Object getObject()`：返回此工厂创建的对象的实例。实例可以共享，这取决于这个工厂是返回单例还是原型。
- `boolean isSingleton()`：如果这个“`FactoryBean`”返回单例，则返回 `true`，否则返回 `false`。
- `Class<?> getObjectType()`：返回由 `getObject()` 方法或 `null` 返回的对象类型，如果类型不是预先知道的。

`FactoryBean` 概念和接口在 Spring 框架中的许多地方使用；Spring 内置的有超过 50 个实现。

当你需要向容器请求一个实际的 `FactoryBean` 实例本身而不是它生成的 bean 时，在调用 `ApplicationContext` 的 `getBean()` 方法时，用符号（`&`）作为前缀。所以对于给定的 `FactoryBean`，`id` 为 `myBean`，在容器上调用 `getBean("myBean")` 会返回 `FactoryBean` 所产生的 bean；而调用 `getBean("&myBean")` 返回 `FactoryBean` 实例本身。

## 3.9 基于注解的容器配置

### 注解是否比XML更好地配置Spring？

引入基于注解的配置进而引发了这种方法是否比XML更好的问题。简短的答案是得看情况。长的答案是，每种方法都有其优缺点，通常由开发人员决定哪种策略更适合他们。由于它们的定义方式，注解在它们的声明中提供了很多上下文，导致更短和更简洁的配置。但是，XML擅长在不接触源码或者无需反编译的情况下组装组件。一些开发人员更喜欢在源码上使用注解配置，而另一些人认为注解类不再是POJO，此外，配置变得分散，更难以控制。

无论选择什么，Spring都可以适应这两种风格，甚至可以将它们混合在一起。值得指出的是，Spring通过其[JavaConfig](#)选项，允许注解以无侵入方式使用，无需接触目标组件源代码，而且在工具方面，[Spring Tool Suite](#)支持所有配置样式。

XML设置的替代方法是基于注解的配置，它依赖于字节码元数据来连接组件替代XML声明。开发人员通过使用相关类，方法或字段声明上的注解来将配置移动到组件类本身中，而不是使用XML描述bean的配置。如在[“Example:RequiredAnnotationBeanPostProcessor”一节中提到的](#)一起使用BeanPostProcessor与注解是扩展Spring IoC容器的常用手段。例如，Spring 2.0引入了使用[@Required](#)注解。Spring 2.5使得有可能遵循同样的一般方法来驱动Spring的依赖注入。本质上，[@Autowired](#)注解提供了相同的能力，在这一章有详解[Section 3.4.5, “Autowiring collaborators”](#)，但具有更细粒度的控制和更广泛的适用性。Spring 2.5还添加了对JSR-250注解的支持，例如[@PostConstruct](#) 和 [@PreDestroy](#)。Spring 3.0增加了对[javax.inject](#)包中包含的JSR-330(Java的依赖注入)注解的支持，例如[@Inject](#) 和 [@Named](#)。有关这些注解的详细信息，请参见[相关部分](#)。

注解注入在 XML注入之前执行，因此同时使用这两种方式注入时，XML配置会覆盖注解配置。

和之前一样，你可以将它们注册为单独的bean定义，但也可以通过在基于XML的Spring配置中包含以下标记来进行隐式注册(注意包含“context”命名空间):

```
<?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>
```

(隐式注册的后置处理器包括 `AutowiredAnnotationBeanPostProcessor` , `CommonAnnotationBeanPostProcessor` , `PersistenceAnnotationBeanPostProcessor` , , 以及 上述 `RequiredAnnotationBeanPostProcessor` 。)

`<context:annotation-config/>` 只在定义bean的相同应用程序上下文中查找bean上的注解。这意味着，如果你在 `DispatcherServlet` 的 `WebApplicationContext` 中插入 `<context:annotation-config/>` ，它只会检查你的控制器中的 `@Autowired` bean，而不是你的服务。有关详细信息，请参见第18.2节“`DispatcherServlet`”

### 3.9.1 @Required

`@Required` 注解适用于bean的setter方法，如以下示例所示：

```
public class SimpleMovieLister {  
  
    private MovieFinder movieFinder;  
  
    @Required  
    public void setMovieFinder(MovieFinder movieFinder) {  
        this.movieFinder = movieFinder;  
    }  
  
    // ...  
}
```

此注解仅表示受影响的bean属性必须在配置时通过bean定义中的显式赋值或通过自动注入。如果受影响的bean属性尚未指定值，容器将抛出异常；这导致及时明确的失败，避免后面的 `NullPointerException` 或类似异常。仍然建议你将断言放入bean类本身，例如，放入init方法。这样做强制执行那些必需的引用和值，即使你在容器外使用这个类。

## 3.9.2 @Autowired



JSR 330的 `@Inject` 注解可以用来代替Spring的 `@Autowired` 注解。有关更多详细信息，请参阅[here](#)。

你可以将`@Autowired` 注解应用于构造函数：

```
public class MovieRecommender {

    private final CustomerPreferenceDao customerPreferenceDao;

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

    // ...
}
```



As of Spring Framework 4.3, the `@Autowired` constructor is no longer necessary if the target bean only defines one constructor. If several constructors are available, at least one must be annotated to teach the container which one it has to use. 从Spring Framework 4.3开始，如果目标bean仅定义一个构造函数，则不再需要`@Autowired`构造函数。如果有几个构造函数可用，至少有一个必须注解以让容器知道它必须使用哪一个。

正如所料，你还可以将 `@Autowired` 注解应用于“传统”**setter**方法：

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

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

    // ...
}
```

你还可以将注解应用于具有任意名称和/或多个参数的方法：

```

public class MovieRecommender {

    private MovieCatalog movieCatalog;

    private CustomerPreferenceDao customerPreferenceDao;

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

    // ...

}

```

你可以将 `@Autowired` 应用于字段，甚至可以与构造函数混合使用：

```

public class MovieRecommender {

    private final CustomerPreferenceDao customerPreferenceDao;

    @Autowired
    private MovieCatalog movieCatalog;

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

    // ...

}

```

也可以用在数组上，注解标注于属性或者方法上，数组的类型是 `ApplicationContext` 中定义的bean的类型。

```

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;
    }

    // ...

}

```



如果你想要数组元素或者集合元素被排序成一个特定的顺序，你的bean可以实现 `org.springframework.core.Ordered` 接口，或者使用 `@Order` 或者标准 `@Priority` 注解。

只要期望的键类型是String，那么Map类型可以自动注入。Map值将包含所有类型的bean，并且键将包含相应的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，当没有候选bean可用的时候自动注入会失败；默认是将带有注解的方法，构造函数和字段视为指示 `required dependencies`。这种行为设置为非必须的，如下所示。

```

public class SimpleMovieLister {

    private MovieFinder movieFinder;

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

    // ...

}

```



只有一个带注解的构造函数 `per-class` 可以标记为 `required`，但是可以注解多个非必需的构造函数。在这种情况下，每个被考虑的候选人和 Spring 使用的 `greediest` 构造函数的依赖可以满足，这是具有最大数量的参数的构造函数。`@Autowired` 的 `required` 属性被推荐在 `@Required` 注解。`required` 属性指示该属性不是自动注入目的所必需的，如果不能自动注入，则忽略该属性。`@Required`，另一方面，需要强调的是，它强制通过容器支持的任何方式来设置的属性。如果没有注入值，则引发相应的异常。

你也可以使用 `@Autowired` 作为常见的可解析依赖关系的接口(只要该接口有类实现，所以才说可解析依赖关系的接口，也就是面向接口编程，一个接口可以有多种实现，也就有多种配置了，不改变接代码的前提下可以做到灵活迭代): `BeanFactory`，`ApplicationContext`，`Environment`，`ResourceLoader`，`ApplicationEventPublisher` 和 `MessageSource`。这些接口及其扩展接口(如 `ConfigurableApplicationContext` 或 `ResourcePatternResolver`)会自动解析，无需特殊设置。

```

public class MovieRecommender {

    @Autowired
    private ApplicationContext context;

    public MovieRecommender() {
    }

    // ...
}

```



`@Autowired`，`@Inject`，`@Resource`，和 `@Value` 注解由 Spring 的 `BeanPostProcessor` 实现处理，也就是说你不能使用自定义的 `BeanPostProcessor` 或者自定义 `BeanFactoryPostProcessor` 应用这些注解。这些类型的组装，必须明确的由 XML 或者使用 Spring `@Bean` 方法完成。



### 3.9.3 使用@Primary微调基于注解的自动装配

因为按类型的自动注入可能导致多个候选者，所以通常需要对选择过程具有更多的控制。实现这一点的一种方法是使用Spring的 `@Primary` 注解。它表示如果存在多个候选者且另一个bean只需要一个特定类型的bean依赖时，就使用标记了`@Primary`注解的那个依赖。如果候选中只有一个“Primary”bean，那么它将是自动注入的值。

让我们假设我们有以下配置，将 `firstMovieCatalog` 定义为 `primary MovieCatalog`。

```
@Configuration
public class MovieConfiguration {

    @Bean
    @Primary
    public MovieCatalog firstMovieCatalog() { ... }

    @Bean
    public MovieCatalog secondMovieCatalog() { ... }

    // ...
}
```

With such configuration, the following `MovieRecommender` will be autowired with the `firstMovieCatalog`. 有了这样的配置，下面的 `MovieRecommender` 将被 `firstMovieCatalog` 自动注入。

```
public class MovieRecommender {

    @Autowired
    private MovieCatalog movieCatalog;

    // ...
}
```

相应的bean定义如下所示。

```
<?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" primary="true">
        <!-- inject any dependencies required by this bean -->
    </bean>

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

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

</beans>
```

### 3.9.4 使用@Qualifier限定符微调基于注解的自动装配

`@Primary` 是一种有效的方法，当一个主要候选项可以被确定时，使用具有多个实例的类型的自动注入。当需要对选择过程做更多的控制时，可以使用Spring的 `@Qualifier` 注解。你可以为指定的参数绑定一个限定的值，缩小类型匹配集，以便为每个参数选择特定的bean。在最简单的情况下，这可以是一个简单的描述性值：

```
public class MovieRecommender {  
  
    @Autowired  
    @Qualifier("main")  
    private MovieCatalog movieCatalog;  
  
    // ...  
  
}
```

`@Qualifier` 注解也可以在单独的构造函数参数或方法参数上指定：

```
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。带有限定符“main”的bean会被装配到拥有同样值的构造方法参数上。

```

<?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>

```

bean的name会作为备用 qualifier 值。因此你可以定义bean的id为 main 替代内嵌的 qualifier 元素，这将同样会匹配上。但是，虽然你可以使用这个约定来按名称引用特定的bean，但是 @Autowired 默认是使用带有限定符的类型驱动的注入的。这就意味着 qualifier 值，甚至是bean 的name作为备选项，只是为了缩小类型匹配的范围；他们并不能作为引用的bean的唯一标示符。好的 qualifier 值是 main , EMEA , persistent ,能表达具体的组件的特性,这些qualifier独立于bean的 id ，因为id可能是匿名bean自动生成的，如前面的例子。

限定符也适用于类型集合，如上所述，例如，“Set”。在这种情况下，根据声明的限定符 qualifiers 所匹配的bean都会被注入到集合内。这意味着限定符qualifiers并不是唯一的;它们只是构成过滤标准。例如，你可以定义多个具有相同限定的值“action”的 MovieCatalog bean，所有这些都将注入到带有 @Qualifier("action") 注解的 Set 中。

如果你打算通过name名字来驱动注解注入，不要主要使用 @Autowired，即使在技术上能够通过 @Qualifier 值引用一个bean名字。相反，使用JSR-250 @Resource 注解，该注解在语义上定义为通过其唯一名称标识特定目标组件，其中声明的类型与匹配进程无关。

@Autowired 具有相当不同的语义：在按类型选择候选bean之后，指定的String限定的值将仅在这些类型选择的候选中被考虑。匹配一个“account”限定符与标记有相同限定符标签的bean。对于自身定义为集合/map或数组类型的bean，@Resource 是一个很好的解决方案，通过唯一名称引用特定的集合或数组bean。也就是说，从4.3开始，只要元素类型信息保存在 @Bean 返回类型签名或集合继承层次结构中，collection / map和数组类型也可以通过Spring的 @Autowired 类型匹配算法进行匹配。在这种情况下，可以使用限定的值来选择相同类型的集合，如上一段所述。在4.3中，@Autowired也考虑了注入的自引用，即引用当前注入的bean。自引用只是一种后备选项；还是优先使用正常的依赖注入其它的bean。在这个意义上，自引用不参与到正常的候选者选择中，并且从来都不是主要的，相反，它们总是有最低的优先级。在实践中，使用自引用作为最后的手段，例如。通过bean的事务代理在同一实例上调用其他方法：在这种情况下，考虑将受影响的方法分解为单独的委托bean。或者，使用 @Resource，它可以通过其唯一名称获取代理回到当前bean。@Autowired 应用于字段，构造函数和多参数方法，允许在参数上使用 qualifier 限定符注解缩小取值范围。相比之下，只有具有单个参数的字段和bean属性setter方法才支持 @Resource。因此，如果注入目标是构造函数或多参数方法，请使用 qualifiers 限定符。

你可以创建自己的自定义限定符注解。只需定义一个注解，并在定义中提供 @Qualifier 注解：

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

    String value();
}
```

然后，你可以为自动注入的字段和参数提供自定义限定符：

```
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，提供候选bean定义的信息。你可以添加 `<qualifier />` 标签作为 `<bean />` 标签的子元素，然后指定`type`类型和`value`值来匹配自定义的 `qualifier` 注解。`type` 是自定义注解的权限定类名(包路径+类名)。如果没有重名的注解，那么可以使用类名(不含包路径)。这两种方法都在以下示例中演示。

```

<?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[第3.10节“类路径扫描和组件管理”](#)，你将看到一个基于注解的替代方法，以在XML中提供`qualifier`限定符元数据。具体来说，请参见[第3.10.8节“使用注解提供限定符元数据”](#)。

在某些情况下，使用没有值的注解可能就足够了。当注解用于更通用的目的并且可以应用于几种不同类型的依赖上时，这可能是有用的。例如，你可以提供在没有Internet连接时搜索的 `offline` 目录。首先定义简单注解：

```

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

```

然后将注解添加到要自动注入的字段或属性：

```
public class MovieRecommender {

    @Autowired
    @Offline
    private MovieCatalog offlineCatalog;

    // ...

}
```

现在bean的定义只需要一个限定符 type :

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

你还可以为自定义限定名qualifier注解增加属性，用于替代简单的value属性。如果在要自动注入的字段或参数上指定了多个属性值，则bean的定义必须全部匹配这些属性值才能被视为自动注入候选者。作为示例，请考虑以下注解定义：

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

    String genre();

    Format format();

}
```

在这种情况下，Format 是一个枚举：

```
public enum Format {
    VHS, DVD, BLURAY
}
```

要自动注入的字段使用自定义限定符注解，并包括两个属性的值：genre 和 format。

```
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，**bean**定义应该包含匹配的限定的值。此示例还演示了可以使用**bean meta**属性而不是 **<qualifier/>** 子元素。如果可用，**<qualifier/>** 及其属性优先，但是如果没有这样的限定符，自动注入机制会使用 **<meta/>** 标签中提供的值，如在最后两个**bean**定义中下面的例子。

```
<?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>
```

### 3.9.5 使用泛型作为自动注入的限定符

除了 `@Qualifier` 注解，还可以使用 Java 通用类型作为限定的隐式形式。例如，假设你具有以下配置：

```
@Configuration
public class MyConfiguration {

    @Bean
    public StringStore stringStore() {
        return new StringStore();
    }

    @Bean
    public IntegerStore integerStore() {
        return new IntegerStore();
    }

}
```

假设上面的 bean 实现一个通用接口，即 `Store` 和 `StringStore`，你可以 `@Autowired` 注解 `Store` 接口，泛型作为限定符 `qualifier`：

```
@Autowired
private Store<String> s1; // <String> qualifier, injects the stringStore bean

@Autowired
private Store<Integer> s2; // <Integer> qualifier, injects the integerStore bean
```

自动注入 `Lists, Maps and Arrays` 时，通用限定符也适用：

```
// Inject all Store beans as long as they have an <Integer> generic
// Store<String> beans will not appear in this list
@Autowired
private List<Store<Integer>> s;
```

### 3.9.6 CustomAutowireConfigurer

`CustomAutowireConfigurer` 是一个 `BeanFactoryPostProcessor`，它允许你注册自己的自定义 `qualifier` 注解类型，无需指定 `@Qualifier` 注解：

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

`AutowireCandidateResolver` 通过以下方式确定自动注入候选者：

- 每个 `bean` 定义的“`autowire-candidate`”值
- 在 `<beans/>` 元素上可用的任何 `default-autowire-candidates` 模式
- 存在 `@Qualifier` 注解和注册到 `CustomAutowireConfigurer` 的任何自定义注解

当多个 `bean` 有资格作为自动注入候选时，“首要 `bean`”的取决于：如果候选项中的一个 `bean` 定义的“`primary`”属性设置为“`true`”，则将选择它。

### 3.9.7 @Resource

Spring还支持对字段或bean属性设置方法使用JSR-250 `@Resource` 注解。这是Java EE 5和6中的常见模式，例如在JSF 1.2托管bean或JAX-WS 2.0端点中。Spring也支持Spring管理对象的这种模式。

`@Resource` 接受一个`name`属性，默认情况下，Spring将该值解释为要注入的bean名称。换句话说，它遵循 *by-name* 语义，如下例所示：

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

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

}
```

如果没有明确指定名称，则默认名称派生自字段名称或setter方法。在字段的情况下，它采用字段名称；在setter方法的情况下，它接受bean属性名称。因此，以下示例将使名为“movieFinder”的bean注入到其setter方法中：

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

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

}
```

□

若使用了 `CommonAnnotationBeanPostProcessor`，注解提供的`name`名字将被解析为bean的`name`名字。如果你配置了Spring的 `SimpleJndiBeanFactory`，这些`name`名字可以通过JNDI解析。但是，推荐你使用默认的行为，简单的使用Spring的 `JNDI`，这样可以保持逻辑引用，而不是直接引用。

`@Resource` 没有明确指定`name`时，类似于 `@Autowired`，和 `@Autowired` 相似，对于特定 bean(Spring API 内的 bean)，`@Resource` 会以类型匹配方式替代 bean name 名字匹配方式，比如：`BeanFactory`，`ApplicationContext`，`ResourceLoader`，`ApplicationEventPublisher` 和 `Me`

ssageSource 接口。

因此，在下面的示例中，`customerPreferenceDao` 字段首先查找名为`customersPreferenceDao` 的bean，然若未找到，则会使用类型匹配 `CustomerPreferenceDao` 类的实例。`context field`域将会注入 `ApplicationContext`：

```
public class MovieRecommender {  
  
    @Resource  
    private CustomerPreferenceDao customerPreferenceDao;  
  
    @Resource  
    private ApplicationContext context;  
  
    public MovieRecommender() {  
    }  
  
    // ...  
}
```

### 3.9.8 @PostConstruct and @PreDestroy

`CommonAnnotationBeanPostProcessor` 不仅识别 `@Resource` 注解，还识别 JSR-250 生命周期注解。在 Spring 2.5 中引入，对这些注解的支持提供了另一个替代 `initialization callbacks` 和 `destroy callbacks`。如果 `CommonAnnotationBeanPostProcessor` 在 `Spring ApplicationContext` 中注册，它会在相应的 Spring bean 生命周期调用相应的方法，就像是 Spring 生命周期接口方法，或者是明确声明的回调函数。在下面的示例中，缓存将在初始化时预填充，并在销毁时清除。

```
public class CachingMovieLister {

    @PostConstruct
    public void populateMovieCache() {
        // populates the movie cache upon initialization...
    }

    @PreDestroy
    public void clearMovieCache() {
        // clears the movie cache upon destruction...
    }

}
```



有关组合各种生命周期机制的影响的详细信息，请参见“[组合生命周期机制](#)”一节。

## 3.10 ClassPath扫描和管理组件

本章中的大多数示例使用XML来指定在Spring容器中生成每个BeanDefinition的配置元数据。上一节([第3.9节“基于注解的容器配置”](#))演示了如何通过源代码注解提供大量的配置元数据。然而，即使在这些示例中，注解也仅仅用于驱动依赖注入，“base”bean依然是明确的在XML文件中定义。本节介绍通过扫描类路径隐式检测候选组件的选项。候选者组件是class类，这些类经过过滤匹配，由Spring容器注册注册的bean定义，成为Spring bean。这消除了使用XML执行bean注册的需要(也就是没有XML什么事儿了);而是可以使用注解(例如 @Component)，AspectJ类型表达式或你自己的自定义过滤条件来选择哪些类将在容器中注册bean定义。

从Spring 3.0开始，Spring JavaConfig项目提供的许多功能都是Spring Framework核心的一部分。这允许你使用Java而不是使用传统的XML文件来定义bean。可以看[看 @Configuration](#)，[@Bean](#)，[@Import](#) 和 [@DependsOn](#) 注解，以了解如何使用这些新特性的例子。

### 3.10.1 @Component和各代码层注解

`@Repository` 注解是满足存储库(也称为数据访问对象或DAO)的角色。此标记的使用之一使用此注解会自动转换异常，如[第16.2.2节“Exception translation”](#)所述。

Spring提供了更多的构造型注解: `@Component` , `@Service` 和 `@Controller` 。 `@Component` 可用于管理任何Spring的组件. `@Repository` , `@Service` 和 `@Controller` 是 `@Component` 用于指定用例的特殊形式，用于更具体的用例，例如，服务和表现层。因此，你可以使用 `@Component` 注解你的组件类，但是通过使用 `@Repository` , `@Service` 或 `@Controller` 来注解它们，能够让你的类更易于被合适的工具处理或与切面(aspect)关联。例如，这些注解可以使目标组件成为切入点。在Spring框架的未来版本中，`@Repository` , `@Service` ,和 `@Controller` 也可能带有附加的语义。因此，如果你在使用 `@Component` 或 `@Service` 来选择服务层，`@Service` 显然是更好的选择。同理，在持久化层要选择 `@Repository` ,它能自动转换异常。

### 3.10.2 Meta-annotations 元注解

Spring提供的许多注解可以在你自己的代码中用作元注解。元注解即一种可用于别的注解之上简单的注解。例如，上面提到的 `@Service` 注解 `@Component` 的元注解：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component // Spring will see this and treat @Service in the same way as @Component
public @interface Service {

    ...
}
```

元注解也可以组合创建组合注解。例如，来自Spring MVC的 `@RestController` 注解是 `@Controller` 和 `@ResponseBody` 的组成的。

此外，组合注解也可以重新定义来自元注解的属性。这在只想暴露元注解的部分属性值的时候非常有用。例如，Spring的 `@SessionScope` 注解将它的作用域硬编码为 `session`，但仍允许自定义 `proxyMode`。

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Scope(WebApplicationContext.SCOPE_SESSION)
public @interface SessionScope {

    /**
     * Alias for {@link Scope#proxyMode}.
     * <p>Defaults to {@link ScopedProxyMode#TARGET_CLASS} .
     */
    @AliasFor(annotation = Scope.class)
    ScopedProxyMode proxyMode() default ScopedProxyMode.TARGET_CLASS;

}
```

`@SessionScope`然后就可以使用了，而且不需要提供`proxyMode`，如下：

```
@Service
@SessionScope
public class SessionScopedService {
    ...
}
```

或者重写`proxyMode`的值，如下所示：

```
@Service  
 @SessionScope(proxyMode = ScopedProxyMode.INTERFACES)  
 public class SessionScopedUserService implements UserService {  
     // ...  
 }
```

有关详细信息，请参阅[Spring Annotation Programming Model](#).

### 3.10.3 自动检测类并注册bean定义

Spring可以自动检测各代码层的被注解的类，并使用 ApplicationContext 内注册相应的 BeanDefinition 。例如，以下两个类就可以被自动探测：

```
@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
}
```

要自动检测这些类并注册相应的bean，需要在@Configuration配置中添加 @ComponentScan ，其中 basePackages 属性是两个类的常用父包。(或者，你可以指定包含每个类的父包的逗号/分号/空格分隔列表。)

```
@Configuration
@ComponentScan(basePackages = "org.example")
public class AppConfig {
    ...
}
```

□
---

为了简洁，上面可能使用注解的 value 属性，即 @ComponentScan("org.example")

也可以使用XML形式的配置：

```

<?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: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>

```



使用 `<context:component-scan>` 隐式地启用 `<context:annotation-config>` 的功能。当使用 `<context:component-scan>` 时，通常不需要包含 `<context:annotation-config>`。



类路径扫描的包必须保证这些包出现在classpath中。当使用Ant构建JAR时，请确保你没有激活JAR任务的纯文件开关。此外在某些环境装因为安全策略，classpath目录也许不能访问。比如，JDK 1.7.0\_45及更高版本的独立应用程序(需要在你的manifests中设置信任类库“Trusted-Library”;请参见 <http://stackoverflow.com/questions/19394570/java-jre-7u45-breaks-classloader-getresources> )

此外，当使用component-scan元素

时，“AutowiredAnnotationBeanPostProcessor”和“CommonAnnotationBeanPostProcessor”都会隐式包含。意味着这两个组件也是自动探测和注入的--所有这些都不需要XML配置。



通过设置 annotation-config 属性值为 false 即禁用 AutowiredAnnotationBeanPostProcessor 和 CommonAnnotationBeanPostProcessor 的注册。

### 3.10.4 使用过滤器自定义扫描

默认情况下，使用 `@Component`，`@Repository`，`@Service`，`@Controller` 注解的类或者注解为 `@Component` 的自定义注解的类才能被检测为候选组件。但是，你可以通过应用自定义过滤器来修改和扩展此行为。将它们添加为 `@ComponentScan` 注解的 `includeFilters` 或 `excludeFilters` 参数(或作为 `component-scan` 元素的 `include-filter` 或 `exclude-filter` 子元素)。每个过滤器元素需要 `type` 和 `expression` 属性。下表介绍了过滤选项。

**Table 3.5.** 过滤器类型

过滤器类型	表达式示例	描述
annotation (default)	<code>org.example.SomeAnnotation</code>	目标组件类级别的注解
assignable	<code>org.example.SomeClass</code>	目标组件继承或实现的类或接口
aspectj	<code>org.example..*Service+</code>	用于匹配目标组件的AspectJ类型表达式
regex	<code>org\example\Default.*</code>	用于匹配目标组件类名的正则表达式
custom	<code>org.example.MyTypeFilter</code>	<code>org.springframework.core.type.TypeFilter</code> 接口的自定义实现

以下示例显示了忽略所有 `@Repository` 注解，并使用带有“stub”的`Repository`代替：

```

@Configuration
@ComponentScan(basePackages = "org.example",
               includeFilters = @Filter(type = FilterType.REGEX, pattern = ".*Stub.*Repository"),
               excludeFilters = @Filter(Repository.class))
public class AppConfig {
    ...
}

```

或者使用XML形式配置：

```

<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>

```



你还可以通过在注解上设置 `useDefaultFilters = false` 或通过 `use-default-filters = "false"` 作为 `<component-scan />` 元素的属性来禁用默认过滤器。这将不会自动检测带有 `@Component` , `@Repository` , `@Service` , `@Controller` , or `@Configuration` 注解的类.

### 3.10.5 在组件中定义Bean元数据

Spring组件也可以向容器提供bean定义元数据。在 @Configuration 注解的类中使用 @Bean 注解定义bean元数据(也就是Spring bean)。这里有一个简单的例子:

```
@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Spring组件有个方法 doWork() 。然而，它还有一个工厂方法 publicInstance() ，用于产生一个bean定义。 @Bean 注解了工厂方法，还设置了其他的bean定义的属性，例如通过 @Qualifier 注解的qualifier值。可以指定的其他方法级别注解是 @Scope , @Lazy 以及自定义 qualifier注解。

除了用于组件初始化的角色之外， @Lazy 注解也可以在 @Autowired 或者 @Inject 处使用。这种情况下，该注入将会变成延迟注入代理lazy-resolution proxy(也就是懒加载)。

自动注入的字段和方法也可以像前面讨论的一样被支持，也可以支持@Bean方法的自动注入：

```

@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(spouse);
        tb.setCountry(country);
        return tb;
    }

    @Bean
    private TestBean privateInstance() {
        return new TestBean("privateInstance", i++);
    }

    @Bean
    @RequestScope
    public TestBean requestScopedInstance() {
        return new TestBean("requestScopedInstance", 3);
    }

}

```

样例中，自动注入的方法参数，类型 `String`，名称为 `country`，将会被设置为另一个实例 `privateInstance` 的 `Age` 属性。Spring Expression Language 元素通过符号 `# {<expression>}` 定义属性的值。对于 `@Value` 注解，表达式解析器在解析表达式后，会查找 bean 的名字并设置 value。

从 Spring Framework 4.3 开始，你还可以声明一个类型为“`InjectionPoint`”（或其更具体的子类“`DependencyDescriptor`”）的工厂方法参数，以访问触发创建当前 bean 的请求注入点。请注意，这仅适用于实际创建 bean 实例，而不适用于注入现有实例。因此，这个特性对 `prototype` scope 的 bean 最有意义。对于其他 scope，工厂方法将只能看到触发在给定 scope 中创建新 bean 实例的注入点：例如，触发创建一个延迟单例 bean 的依赖。在这种情况下，使用提供的注入点元数据具有语义关怀（为程序员考虑提供便利）。

```

@Component
public class FactoryMethodComponent {

    @Bean @Scope("prototype")
    public TestBean prototypeInstance(InjectionPoint injectionPoint) {
        return new TestBean("prototypeInstance for " + injectionPoint.getMember());
    }
}

```

在 Spring component 中处理 @Bean 和在 @Configuration 中处理是不一样的。区别在于，在 @Component 中，不会使用 CGLIB 增强去拦截方法和属性的调用。在 @Configuration 注解的类中，@Bean 注解的方法创建的 bean 对象的方法和属性的调用，是使用 CGLIB 代理。方法的调用不是常规的 Java 语法，而是通过容器来提供通常的生命周期管理和代理 Spring bean，甚至在通过编程地方式调用 @Bean 方法时也会形成对其他 bean 的引用。相比之下，在一个简单的“@Component”类中调用 @Bean 方法中的方法或字段具有标准 Java 语义，没有应用特殊的 CGLIB 处理或其他约束。

你可以将 @Bean 方法声明为 static，允许在不将其包含的配置类作为实例的情况下调用它们。这在定义后置处理器 bean 时是特别有意义的。比如 BeanFactoryPostProcessor 或 BeanPostProcessor，因为这类 bean 会在容器的生命周期前期被初始化，而不会触发其它部分的配置。对静态 @Bean 方法的调用永远不会被容器拦截，即使在 @Configuration 类 内部。这是由于技术限制 CGLIB 的子类代理只会重写非静态方法。因此，对另一个 @Bean 方法的直接调用只有标准的 Java 语法，只会从工厂方法本身直接返回一个独立的实例。

由于 Java 语言的可见性，@Bean 方法并不一定会对容器中的 bean 有效。你可能很随意的在非 @Configuration 类中定义或定义为静态方法。然而，在 @Configuration 类中的正常的 @Bean 方法都需要被重写的，因此，它们不应该定义为 private 或 final。@Bean 方法也可以在父类中被发现，同样适用于 Java 8 中接口的默认方法。这使得组建复杂的配置时能具有更好的灵活性，甚至可能通过 Java 8 的默认方法实现多重继承，这在 Spring 4.2 开始支持。

最后，请注意，单个类可以为同一个 bean 保存多个 @Bean 方法，如根据运行时可用的依赖关系选择合适的工厂方法。使用的算法时选择“最贪婪”的构造方法，一些场景可能会按如下方法选择相应的工厂方法：满足最多依赖的会被选择，这与使用 @Autowired 时选择多个构造方法时类似。

### 3.10.6 命名自动注册组件

扫描处理过程其中一步就是自动探测组件，扫描器使用 `BeanNameGenerator` 对探测到的组件命名。默认情况下，各代码层注解(`@Component, @Repository, @Service, @Controller`)所包含的 `name` 值，将会作为相应的bean定义的名字。

如果这些注解没有 `name` 值，或者是其他一些被探测到的组件（比如使用自定义过滤器探测到的），默认的bean name生成器生成，以小写类名作为bean名字。例如，如果检测到以下两个组件，则名称将是 `myMovieLister` 和 `movieFinderImpl`：

```
@Service("myMovieLister")
public class SimpleMovieLister {
    // ...
}
```

```
@Repository
public class MovieFinderImpl implements MovieFinder {
    // ...
}
```

如果你不想依赖默认的bean命名策略，你可以提供一个自定义bean命名策略。首先，实现 `BeanNameGenerator` 接口，并确保包括一个默认的无参构造函数。然后，在配置扫描程序时提供完全限定类名：

```
@Configuration
@ComponentScan(basePackages = "org.example", nameGenerator = MyNameGenerator.class)
public class AppConfig {
    ...
}
```

```
<beans>
    <context:component-scan base-package="org.example"
        name-generator="org.example.MyNameGenerator" />
</beans>
```

生成规则应当如下，考虑和注解一起生成name，便于其他组件明确的引用。另一方面，当容器负责组装时，自动生成的名称就足够了。

### 3.10.7 为 component-scan 组件提供作用域

与Spring管理的组件一样，默认的最常见的作用域是单例singleton。然而，有时你需要其他的作用域，可以通过 @Scope 注解来指定。只需在注解中提供作用域的名称：

```
@Scope("prototype")
@Repository
public class MovieFinderImpl implements MovieFinder {
    // ...
}
```

有关特定于Web的作用域的详细信息，请参见第3.5.4节“请求，会话，应用程序和WebSocket作用域”。

想要自己提供自定义作用域解析的策略，而不是依赖于基于注解的方法，得实现 `ScopeMetadataResolver` 接口，并确保包含一个默认的无参数构造函数。然后，在配置扫描程序时提供完全限定类名：

```
@Configuration
@ComponentScan(basePackages = "org.example", scopeResolver = MyScopeResolver.class)
public class AppConfig {
    ...
}
```

```
<beans>
    <context:component-scan base-package="org.example"
        scope-resolver="org.example.MyScopeResolver" />
</beans>
```

当使用某个非单例作用域时，为作用域对象生成代理也许非常必要。原因参看“[作为依赖关系的作用域bean](#)”一节中的描述。`component-scan` 元素中有一个 `scope-proxy` 属性，即可实现此目的。它的值有三个选项：`no, interfaces, and targetClass`，比如下面的配置会生成标准的JDK动态代理：

```
@Configuration
@ComponentScan(basePackages = "org.example", scopedProxy = ScopedProxyMode.INTERFACES)
public class AppConfig {
    ...
}
```

```
<beans>
    <context:component-scan base-package="org.example"
        scoped-proxy="interfaces" />
</beans>
```

### 3.10.8 为注解提供标示符 Qualifier

“@Qualifier”注解在[Section 3.9.4, “Fine-tuning annotation-based autowiring with qualifiers”](#)讨论过。该部分中的示例演示了在解析自动注入候选者时使用 `@Qualifier` 注解和自定义限定符注解以提供细粒度控制。因为这些示例基于XML bean定义，所以使用XML中的 `bean` 元素的 `qualifier` 或 `meta` 子元素在候选bean定义上提供了限定符元数据。当依靠classpath扫描并自动检测组件时，你可以在候选类上提供具有类型级别注解的限定符元数据(you provide the qualifier metadata with type-level annotations on the candidate class.)。以下三个示例演示了此技术：

```
@Component
@Qualifier("Action")
public class ActionMovieCatalog implements MovieCatalog {
    // ...
}
```

```
@Component
@Genre("Action")
public class ActionMovieCatalog implements MovieCatalog {
    // ...
}
```

```
@Component
@Offline
public class CachingMovieCatalog implements MovieCatalog {
    // ...
}
```

□

与大多数基于注解的替代方法一样，请注意，注解元数据绑定到类定义本身，而使用xml的时候，允许同一类型的beans在qualifieer元数据中提供变量，因为元数据是依据实例而不是类来提供的。

## 3.11 使用JSR-330标准注解

从Spring 3.0开始，Spring提供了对JSR-330标准注解(依赖注入)的支持。这些注解以Spring注解相同的方式被扫描。你只需要在你的classpath路径中引入相关的jar。



如果你使用Maven，`javax.inject artifact`可以在标准的Maven仓库中找到(<http://repo1.maven.org/maven2/javax/inject/javax.inject/1/>)。你可以向文件pom.xml中添加以下依赖关系：

```
<dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId>
    <version>1</version>
</dependency>
```

### 3.11.1 使用@.Inject @Name依赖注入

`@javax.inject.Inject` 可以使用下面的方式来替代 `@Autowired` :

```
import javax.inject.Inject;

public class SimpleMovieLister {

    private MovieFinder movieFinder;

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

    public void listMovies() {
        this.movieFinder.findMovies(...);
        ...
    }
}
```

与 `@Autowired` 一样，可以在字段级别，方法级别和构造函数参数级别使用 `@Inject` 。此外，你可以将注入点声明为 `Provider` ，允许按需访问作用域较小的bean或通过 `Provider.get()` 调用对其他bean的延迟访问。在上述示例基础上做如下改变:

```
import javax.inject.Inject;
import javax.inject.Provider;

public class SimpleMovieLister {

    private Provider<MovieFinder> movieFinder;

    @Inject
    public void setMovieFinder(Provider<MovieFinder> movieFinder) {
        this.movieFinder = movieFinder;
    }

    public void listMovies() {
        this.movieFinder.get().findMovies(...);
        ...
    }
}
```

如果你想要为注入的依赖项使用限定名称，则应该使用 `@Named` ，如下所示:

```
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;
    }

    // ...
}
```

### 3.11.2 @Named:相当于@Component

`@javax.inject.Named` 或 `javax.annotation.ManagedBean` 可以使用下面的方式来替代 `@Component`：

```
import javax.inject.Inject;
import javax.inject.Named;

@Named("movieListener") // @ManagedBean("movieListener") could be used as well
public class SimpleMovieLister {

    private MovieFinder movieFinder;

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

    // ...
}
```

`@Component` 通常不指定组件名字。`@Named` 也能这么用：

```
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` 或 `@ManagedBean` 时，可以使用与使用Spring注解完全相同的方式使用 component-scanning 组件扫描：

```
@Configuration  
@ComponentScan(basePackages = "org.example")  
public class AppConfig {  
    ...  
}
```



与 `@Component` 相反，JSR-330 `@Named` 和 JSR-250 `ManagedBean` 注解是不可组合使用的。  
请使用 Spring 的 stereotype model 来构建自定义组件注解。

### 3.11.3 JSR-330标准注解的限制

使用标准注解时，重要的是要知道哪些重要功能不可用，如下表所示：

**Table 3.6. Spring组件模型元素vs. JSR-330变体**

Spring	javax.inject.*	javax.inject的局限性
@Autowired	@Inject	@Inject没有require属性，可以使用Java 8的Optional代替。
@Component	@Named / @ManagedBean	JSR-330没有提供组合模型，仅仅只是一种标识组件的方式
@Scope("singleton")	@Singleton	JSR-330默认的作用域类似于Spring的prototype。然而，为了与Spring一般的配置的默认值保持一致，JSR-330配置的bean在Spring中默认为singleton。为了使用singleton以外的作用域，必须使用Spring的@Scope注解。javax.inject也提供了一个@Scope注解，不过这仅仅被用于创建自己的注解。
@Qualifier	@Qualifier / @Named	javax.inject.Qualifier仅使用创建自定义的限定符。可以通过 javax.inject.Named 创建与Spring中@Qualifier一样的限定符
@Value	-	无
@Required	-	无
@Lazy	-	无
ObjectFactory	Provider	javax.inject.Provider是对Spring的ObjectFactory的直接替代，仅仅使用简短的get()方法即可。它也可以与Spring的@Autowired或无注解的构造方法和setter方法一起使用。

## 3.12 基于**Java**的容器配置

### 3.12.1 基本概念：@Bean和@Configuration

Spring中基于Java的配置的核心内容是 @Configuration 注解的类和 @Bean 注解的方法。

@Bean 注解用于表明一个方法将会实例化、配置和初始化一个由Spring IoC容器管理的新对象。这就像在XML中元素中元素一样。你可以在任何Spring @Component 中使用 @Bean 注解方法，但大多数情况下，@Bean 是配合 @Configuration 使用的。

使用 @Configuration 注解类表明这个类的目的就是作为bean定义的地方。此外， @Configuration 类内部的bean可以调用本类中定义的其它bean作为依赖。最简单的配置如下所示：

```
@Configuration
public class AppConfig {

    @Bean
    public MyService myService() {
        return new MyServiceImpl();
    }

}
```

上面的 AppConfig 类下面的XML形式是等价的

```
<beans>
    <bean id="myService" class="com.acme.services.MyServiceImpl"/>
</beans>
```

#### Full @Configuration vs 'lite' @Beans mode?

当 @Bean 方法在没有用 @Configuration 注解的类中声明时，它们被称为以 lite 模式处理。例如，在 @Component 或甚至在 plain old class 中声明的bean方法将被认为是'lite'。

与完整的 @Configuration 不同，lite @Bean 方法不能轻易地声明bean之间的依赖关系。通常一个 @Bean 方法在 lite 模式下操作时不应该调用另一个 @Bean 方法。

只有在 @Configuration 类中使用 @Bean 方法是一种推荐的方法，可以确保始终使用 full 模式。这将防止相同的 @Bean 方法被意外地多次调用，并且有助于减少在 lite 模式下操作时难以跟踪的微小错误。

@Bean 和 @Configuration 注解将在下面的章节中深入讨论。首先，我们将介绍使用基于Java代码的配置来创建一个spring容器的各种方法。



### 3.12.2 使用AnnotationConfigApplicationContext实例化Spring容器

下Spring的 AnnotationConfigApplicationContext 部分，是Spring3.0中新增的。这是一个强大的(译注原文中是多才多艺的*versatile*) ApplicationContext 实现,不仅能解析 @Configuration 注解类，也能解析 @Componnet 注解的类和使用 JSR-330 注解的类。

当使用 @Configuration 类作为输入时， @Configuration 类本身被注册为一个bean定义，类中所有声明的 @Bean 方法也被注册为bean定义。

当提供 @Component 和JSR-330类时，它们被注册为bean定义，并且假定在必要时在这些类中使用DI元数据，例如 @Autowired 或 @Inject 。

#### 简单结构

Spring以XML作为配置元数据实例化一个 ClassPathXmlApplicationContext，以 @Configuration 类作为配置元数据时，Spring以差不多的方式，实例化一个 AnnotationConfigApplicationContext 。因此，Spring 容器可以实现零XML配置。

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

如上所述， AnnotationConfigApplicationContext 不限于只使用 @Configuration 类任何 @Component或JSR-330注解的类都能被提供给这个构造方法。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();
}
```

上面假设 MyServiceImpl , Dependency1 和 Dependency2 使用Spring依赖注入注解，如 @Autowired 。

#### 使用register(Class<?>...)编程式构造Spring容器

An `AnnotationConfigApplicationContext` may be instantiated using a no-arg constructor and then configured using the `register()` method. This approach is particularly useful when programmatically building an `AnnotationConfigApplicationContext`.

`AnnotationConfigApplicationContext` 可以通过无参构造函数实例化然后，然后调用 `register()` 方法进行配置。这种方法在以编程方式构建一个 `AnnotationConfigApplicationContext` 时特别有用。

```
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();
}
```

## 使用 `scan(String...)` 扫描组件

要启用组件扫描，只要像下面这样配置`@Configuration`类即可：

```
@Configuration
@ComponentScan(basePackages = "com.acme")
public class AppConfig {
    ...
}
```



有经验的用户可能更熟悉使用等价的XML形式配置

```
<beans>
    <context:component-scan base-package="com.acme"/>
</beans>
```

上面的例子中，`com.acme` 包会被扫描，只要是使用了 `@Component` 注解的类，都会被注册进容器中。同样地，`AnnotationConfigApplicationContext` 暴露的 `scan(String...)` 方法也允许扫描类，完成相同的功能：

```

public static void main(String[] args) {
    AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
    ctx.scan("com.acme");
    ctx.refresh();
    MyService myService = ctx.getBean(MyService.class);
}

```



请记住，`@Configuration` 类是被`@Component`元注解`meta-annotated` 注解的类 所以它们也会被扫描到。上面的例子中，假设`AppConfig`定义在`com.acme`包中（或更深的包中），调用`scan()`时它也会被扫描到，并且它里面配置的所有`@Bean`方法会在`refresh()`的时候被注册到容器中。

## 使用 AnnotationConfigWebApplicationContext 支持 web 应用

一个`WebApplicationContext`与`AnnotationConfigApplicationContext`的变种是`AnnotationConfigWebApplicationContext`。这个实现可以用于配置`Spring ContextLoaderListener` servlet 监听器，`Spring MVC` 的 `DispatcherServlet` 等时使用。下面是一个 `web.xml` 代码片段，用于配置典型的`Spring MVC Web`应用程序。注意 `contextClass` 类的`context-param`和`init-param`:

```

<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.AnnotationConfigWebApplicationCont
ext
        </param-value>
    </context-param>

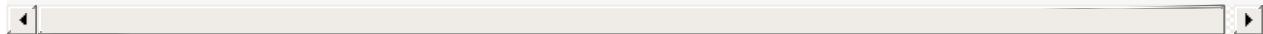
    <!-- Configuration locations must consist of one or more comma- or space-delimited
         fully-qualified @Configuration classes. Fully-qualified packages may also be
         specified for component-scanning -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>com.acme.AppConfig</param-value>
    </context-param>

    <!-- Bootstrap the root application context as usual using ContextLoaderListener -
    ->
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listene
r-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.AnnotationConfigWebApplication
Context
        </param-value>
    </init-param>
    <!-- Again, config locations must consist of one or more comma- or space-delim
ited
        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>
```



### 3.12.3 使用@Bean注解

`@Bean` 是一个方法级的注解，它与XML中的类似。注解支持 `<bean/>` 提供的一些属性，例如`init-method`, `destroy-method`, `autowiring`和 `name`。

可以在`@Configuration`或`@Component`注解的类中使用`@Bean`注解。

#### 声明bean

要声明一个bean，只需使用`@Bean`注解一个方法。使用此方法，将会在`ApplicationContext`内注册一个bean，bean的类型是方法的返回值类型。默认情况下，bean名称将与方法名称相同。下面是一个简单的例子：`@Bean`方法声明：

```
@Configuration
public class AppConfig {

    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl();
    }

}
```

上面的配置完全等同于以下Spring XML：

```
<beans>
    <bean id="transferService" class="com.acme.TransferServiceImpl"/>
</beans>
```

两种方式都定义了一个名字为`transferService`的bean，且绑定了`TransferServiceImpl`的实例：

```
transferService -> com.acme.TransferServiceImpl
```

#### Bean之间的依赖

一个`@Bean`注解的方法可以有任意数量的参数描述构建该bean所需的依赖。例如，如果我们的`TransferService`需要一个`AccountRepository`，我们可以通过一个方法参数来实现该依赖：

```
@Configuration
public class AppConfig {

    @Bean
    public TransferService transferService(AccountRepository accountRepository) {
        return new TransferServiceImpl(accountRepository);
    }

}
```

解析机制与基于构造函数的依赖注入非常相似，参见[相关部分](#)。

## 接收生命周期回调

使用 `@Bean` 注解定义的任何类都支持常规的生命周期回调，并且可以使用JSR-250 的 `@PostConstruct` 和 `@PreDestroy` 注解，参见[JSR-250注解](#)。

完全支持常规Spring `lifecycle`回调。如果bean实现了 `InitializingBean`, `DisposableBean`, or `Lifecycle`，它们各自的方法由容器调用。

同样地，标准的 `*Aware` 接口，如 `BeanFactoryAware`, `BeanNameAware`, `MessageSourceAware`, `ApplicationContextAware` 等等都完全支持。

`@Bean` 注解支持指定任意初始化和销毁回调方法，就像Spring XML的 `bean` 元素的 `init-method` 和 `destroy-method` 属性：

```

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();
    }
}

```



默认情况下，使用java config 定义的bean中 `close` 方法或者 `shutdown` 方法，会作为销毁回调自动调用。

若bean中有 `close` , `shutdown` 方法，又不是销毁回调，通过设置 `@Bean(destroyMethod="")` ，即可关闭该默认的自动匹配销毁回调模式。

你可能希望对通过JNDI获取的资源执行此操作，因为它的生命周期是在应用程序外部管理的。尤其是，使用 `DataSource` 时一定要关闭它，不然会有问题。

```

@Bean(destroyMethod="")
public DataSource dataSource() throws NamingException {
    return (DataSource) jndiTemplate.lookup("MyDS");
}

```

同样地，，使用 `@Bean` 方法，通常会选择使用程序化的JNDI查找:使用Spring的 `JndiTemplate` / `JndiLocatorDelegate` 帮助类或直接使用JNDI的`InitialContext`但是不要使用 `JndiObjectFactoryBean` 变体，因为它会强制你去声明一个返回类型作为 `FactoryBean`的类型代替实际的目标类型，这使得交叉引用变得很困难。

当然，上面 `Foo` 的例子中，也可以在构造函数中调用 `init()` 方法，和上面例子中的效果相同；

```
@Configuration
public class AppConfig {
    @Bean
    public Foo foo() {
        Foo foo = new Foo();
        foo.init();
        return foo;
    }

    // ...
}
```



如果是直接使用java，对于对象，你想怎么搞就怎么搞，并不总需要依赖容器生命周期！

## 指定bean的作用域

使用`@Scope`注解

可以使用任何标准的方式为 `@Bean` 注解的bean指定一个作用域。你可以使用[Bean Scopes](#)章节中的任意标准作用域。

默认范围是 `singleton`，但是你可以使用 `@Scope` 注解来覆盖它：

```
@Configuration
public class MyConfiguration {

    @Bean
    @Scope("prototype")
    public Encryptor encryptor() {
        // ...
    }

}
```

## `@Scope` 和作用域代理

Spring通过[scoped代理](#)提供了一种方便的方式完成作用域bean依赖关系。在使用XML配置时创建这种代理的最简单的方法是 `<aop:scoped-proxy/>` 元素。若是在Java代码中配置bean,有一种等价的做法，使用 `@Scope` 注解并配置其 `proxyMode` 属性。默认是没有代理 (`ScopedProxyMode.NO`)，但你可以指定 `ScopedProxyMode.TARGET_CLASS` 或 `ScopedProxyMode.INTERFACES`。

如果将XML格式的作用域代理示例转换成Java中使用 `@Bean`，差不多是这样：

```
// an HTTP Session-scoped bean exposed as a proxy
@Bean
@SessionScope
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;
}
```

## 自定义bean名字

默认情况下，配置类中，使用 `@Bean` 的方法名作为返回bean的名字。但是，使用 `name` 属性可以覆盖此功能。

```
@Configuration
public class AppConfig {

    @Bean(name = "myFoo")
    public Foo foo() {
        return new Foo();
    }
}
```

## bean别名

如第3.3.1节“命名bean”中所讨论的有时候需要给一个bean指定多个name。`@Bean` 注解的 `name` 属性就是干这个用的，该属性接收一个字符串数组。

```
@Configuration
public class AppConfig {

    @Bean(name = { "dataSource", "subsystemA-dataSource", "subsystemB-dataSource" })
    public DataSource dataSource() {
        // instantiate, configure and return DataSource bean...
    }
}
```

## bean描述

有时，提供bean的更详细的文本描述是有帮助的。用于监视目的(通过JMX)的时候，这可能特别有用。

要向一个 @Bean 添加一个描述，可以使用 @Description 注解：

```
@Configuration
public class AppConfig {

    @Bean
    @Description("Provides a basic example of a bean")
    public Foo foo() {
        return new Foo();
    }

}
```

### 3.12.4 使用@Configuration注解

`@Configuration` 是一个类级别的注解，表明该类将作为bean定义的配置元数据。

`@Configuration` 类通过`public @Bean` 注解方法声明bean。在`@Configuration` 类上调用`@Bean` 方法也可以用于定义bean间依赖关系。参见第3.12.1节“基本概念:@Bean和@Configuration”里的介绍。

#### 注入内部bean依赖

当`@Bean` 彼此有依赖关系时，表示依赖关系就像调用另一个bean方法一样简单：

```
@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通过构造函数注入接收对`bar` 的引用。

这种方式仅仅适用于在`@Configuration` 内部定义的`@Bean`方法。在普通的`@Component`类中不能声明内部依赖。

#### 查找方法注入

如前所述，`lookup method injection` 是一种高级功能，你应该很少使用。。但是，在一个单例bean依赖原型作用域bean的场景中，就非常有用了。Java中，提供了很友好的api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，你可以创建一个 `CommandManager` 的子类，其中抽象 `createCommand()` 方法被覆盖，这样就可以让它查找到新的原型`command`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();
        }
    }
}

```

## 更多关于Java配置内部工作的信息

下面示例中，展示了 `@Bean` 注解的方法被调用了2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()` 在 `clientService1()` 中调用一次，在 `clientService2()` 中调用一次。由于这个方法创建一个新的 `ClientDaoImpl` 实例并返回它，你通常期望有两个实例(每个服务一个实例)。这肯定会有问题:在Spring中，实例化的bean默认情况下有一个 `singleton scope`。这就是它的神奇之处:所有 `@Configuration` 类在启动时都是通过 `CGLIB` 创建一个子类。在子类中，在调用父类的方法并创建一个新的实例之前，子类中的方法首先检查是否缓存过该bean实例。注意，从Spring 3.2开始，不再需要将CGLIB添加到classpath中，因为CGLIB类已经在`org.springframework.cglib`下重新打包并直接包含在spring-core JAR中，可以直接使用。



这种行为可以根据bean的作用域而变化，我们这里只讨论单例。



实际上还会有一些限制，因为CGLIB是在启动的时候动态地添加这些特性，所以配置的类不能是final的。但是，从4.3开始，允许在配置类上使用任何构造函数，包括使用 `@Autowired` 或一个非默认构造函数声明作为默认注入。如果想避免任何CGLIB带来的限制，请考虑声明非 `@Configuration` 类的 `@Bean` 方法，例如在纯的 `@Component` 类。这样在 `@Bean` 方法之间的交叉方法调用将不会被拦截，因此你必须在构造函数或方法级别上依赖于依赖注入。

### 3.12.5 组合Java基本配置

#### 使用@Import注解

与Spring XML文件中使用 `<import/>` 元素来帮助进行模块化配置一样，`@Import` 注解允许从另一个配置类加载 `@Bean` 定义：

```
@Configuration
public class ConfigA {

    @Bean
    public A a() {
        return new A();
    }

}

@Configuration
@Import(ConfigA.class)
public class ConfigB {

    @Bean
    public B b() {
        return new B();
    }

}
```

现在，不需要在实例化上下文时同时指定 `ConfigA.class` 和 `ConfigB.class`，而是仅需要提供 `ConfigB` 即可：

```
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，而不是要求开发人员在构建期间记住一个潜在的大量的 `@Configuration` 类。



从Spring Framework 4.2开始，`@Import` 也可以支持对普通组件类的引用了，类似于 `AnnotationConfigApplicationContext.register` 方法。如果你想避免组件扫描，使用几个配置类作为明确定义你所有组件的入口点，这是非常有用的。

#### 在导入的**@Bean**定义上注入依赖

上面的例子可以很好运行，但是太简单了。在大多数实际情况下，bean将在配置类之间相互依赖。当使用XML时，这本身不是一个问题，因为没有涉及编译器，可以简单地声明 `ref = "someBean"` 并且相信Spring将在容器初始化期间可以很好地处理它。当然，当使用 `@Configuration` 类时，Java编译器会有一些限制，符合Java的语法。

幸运的是，解决这个问题很简单。因为我们已经讨论过，`@Bean` 方法可以有任意的参数用于描述其依赖。让我们考虑一个更加真实的场景，使用了多个 `@Configuration` 类，每个配置都依赖其他配置中是bean声明：

```

@Configuration
public class ServiceConfig {

    @Bean
    public TransferService transferService(AccountRepository accountRepository) {
        return new TransferServiceImpl(accountRepository);
    }

}

@Configuration
public class RepositoryConfig {

    @Bean
    public AccountRepository accountRepository(DataSource dataSource) {
        return new JdbcAccountRepository(dataSource);
    }

}

@Configuration
@Import({ServiceConfig.class, RepositoryConfig.class})
public class SystemTestConfig {

    @Bean
    public 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");
}

```

还有另一种方法来实现相同的结果。记住 `@Configuration` 类也是容器中的一个bean:这意味着他们可以像任何其他bean一样使用 `@Autowired` 和 `@Value` 注解！



确保你注入的依赖关系是最简单的类型。`@Configuration` 类会在上下文初始化的早期被处理，所以它的依赖会在更早期被初始化。如果可能的话，请像上面这样使用参数化注入。同样地，对于通过 `@Bean` 声明的 `BeanPostProcessor` 和 `BeanFactoryPostProcessor` 请谨慎对待。这些通常应该声明为“静态的 `@Bean`”方法，不会触发包含它们的配置类的实例化。否则，`@Autowired` 和 `@Value` 在配置类本身上是不起作用的，因为它们太早被实例化了。

```
@Configuration
public class ServiceConfig {

    @Autowired
    private AccountRepository accountRepository;

    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl(accountRepository);
    }

}

@Configuration
public class RepositoryConfig {

    private final DataSource dataSource;

    @Autowired
    public RepositoryConfig(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    @Bean
    public AccountRepository accountRepository() {
        return new JdbcAccountRepository(dataSource);
    }

}

@Configuration
@Import({ServiceConfig.class, RepositoryConfig.class})
public class SystemTestConfig {

    @Bean
    public 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");
}
```



在 `@Configuration` 类中的构造函数注入只在 Spring 4.3 以后才支持。还要注意，如果目标 `bean` 只定义一个构造函数，则不需要指定 `@Autowired`；在上面的例子中，`@Autowired` 在 `RepositoryConfig` 构造函数中不是必需的。

在上面的场景中，`@Autowired` 可以很好的工作，使设计更具模块化，但是自动注入的是哪个 `bean` 依然有些模糊不清。例如，作为一个开发者查看 `ServiceConfig` 类时，你怎么知道 `@Autowired AccountRepository` `bean` 在哪定义的呢？代码中并未明确指出，还好，[Spring Tool Suite](#) 提供了工具，可以展示 `bean` 之间是如何装配的 - 这可能是你需要的。此外，你的 Java IDE 可以轻松找到所有的定义和 `AccountRepository` 类型引用的地方，并可以快速地找出 `@Bean` 方法定义的地方。

万一需求不允许这种模糊的装配，并且你要在 IDE 内从 `Configuration` 类直接定位到依赖类 `bean`，考虑使用硬编码，即由依赖类本身定位：

```
@Configuration
public class ServiceConfig {

    @Autowired
    private RepositoryConfig repositoryConfig;

    @Bean
    public TransferService transferService() {
        // navigate 'through' the config class to the @Bean method!
        return new TransferServiceImpl(repositoryConfig.accountRepository());
    }

}
```

在上面的情况下，`AccountRepository` 的定义就很明确了。然而，`ServiceConfig` 现在紧紧耦合到 `RepositoryConfig`；这是一种折衷的方法。这种紧耦合某种程度上可以通过接口或抽象解决，如下：

```
@Configuration
public class ServiceConfig {

    @Autowired
    private RepositoryConfig repositoryConfig;

    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl(repositoryConfig.accountRepository());
    }
}

@Configuration
public interface RepositoryConfig {

    @Bean
    AccountRepository accountRepository();

}

@Configuration
public class DefaultRepositoryConfig implements RepositoryConfig {

    @Bean
    public AccountRepository accountRepository() {
        return new JdbcAccountRepository(...);
    }
}

@Configuration
@Import({ServiceConfig.class, DefaultRepositoryConfig.class}) // import the concrete config!
public class SystemTestConfig {

    @Bean
    public 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` 就与具体的 `DefaultRepositoryConfig` 松散耦合了，内置的IDE工具仍然有用：开发人员很容易得到一个 `RepositoryConfig` 实现类的继承体系。以这种方式，操纵 `@Configuration` 类及其依赖关系与操作基于接口的代码的过程没有什么区别。

## 有条件地包含`@Configuration`类或`@Bean`方法

根据特定的系统状态，开启或者关闭一个 `@Configuration` 类，甚至是针对个别 `@Bean` 方法开启或者关闭，通常很有用。一个常见的例子是使用 `@Profile` 注解来激活bean，只有在 `Spring Environment` 中启用了一个特定的配置文件才有效(参见第3.13.1节“Bean定义配置文件”)。

`@Profile` 注解实际上是使用一个更灵活的注解 `@Conditional` 实现的。`@Conditional` 注解表示特定的 `org.springframework.context.annotation.Condition` 实现，表明一个`@Bean`被注册之前会先询问`@Condition`。

`Condition` 接口的实现只提供一个返回 `true` 或 `false` 的 `matches(...)` 方法。例如，这里是用于 `@Profile` 的一个具体的 `Condition` 实现：

```

@Override
public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
    if (context.getEnvironment() != null) {
        // Read the @Profile annotation attributes
        MultiValueMap<String, Object> attrs = metadata.getAllAnnotationAttributes(Profile.class.getName());
        if (attrs != null) {
            for (Object value : attrs.get("value")) {
                if (context.getEnvironment().acceptsProfiles(((String[]) value))) {
                    return true;
                }
            }
            return false;
        }
    }
    return true;
}

```

有关更多信息，请参阅 `@Conditional` javadocs。

## 绑定Java与XML配置

Spring的 `@Configuration` 类并不能100%地替代XML配置。一些情况下使用XML的命名空间仍然是最理想的方式来配置容器。在某些场景下，XML是很方便或必要的，你可以选择：使用例如`ClassPathXmlApplicationContext`，或者以Java为主使用`AnnotationConfigApplicationContext`并在需要的时候使用`@ImportResource`注解导入XML配置

## 基于XML混合使用@Configuration类

更受人喜爱的方法是从XML启动容器并包含 `@Configuration` 类。例如，在使用Spring的现有系统中，大量使用的是Spring XML配置的，所以很容易根据需要创建 `@Configuration` 类，并包含他们到XML文件中。接下来看看此场景。

请记住，`@Configuration` 类最终也只是容器中的bean定义。在这个例子中，我们创建一个名为 `AppConfig` 的 `@Configuration` 类，并将它包含在 `system-test-config.xml` 中作为 `<bean />` 定义。因为 `<context:annotation-config />` 被打开，容器将识别 `@Configuration` 注解，并正确处理在`AppConfig`中声明的 `@Bean` 方法。

```
@Configuration
public class AppConfig {

    @Autowired
    private DataSource dataSource;

    @Bean
    public AccountRepository accountRepository() {
        return new JdbcAccountRepository(dataSource);
    }

    @Bean
    public 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:hsq://localhost/xdb
jdbc.username=sa
jdbc.password=

```

```

public static void main(String[] args) {
    ApplicationContext ctx = new ClassPathXmlApplicationContext("classpath:/com/acme/s
ystem-test-config.xml");
    TransferService transferService = ctx.getBean(TransferService.class);
    // ...
}

```



在上面的 `system-test-config.xml` 中，`AppConfig` 的 `<bean/>` 并没有声明一个 `id` 元素。虽然这样做是可以接受的，但没有其他 `bean` 会引用它，并且不太可能通过名称从容器中显式提取它。与 `DataSource` `bean` 类似，它只能通过类型自动注入，所以一个显式 `bean id` 并不是严格要求的。

因为 `@Configuration` 是被元注解 `@Component` 注解的，所以 `@Configuration` 注解的类也可以被自动扫描。使用与上面相同的场景，我们可以重新定义 `system-test-config.xml` 以使用组件扫描。注意，在这种情况下，我们不需要显式声明 `<context:annotation-config/>`，因为 `<context:component-scan/>` 启用相同的功能。

### 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>

```

### 基于`@Configuration`混合使用`xml`配置

在 `@Configuration` 类为配置容器的主要方式的应用程序中，也需要使用一些`XML`配置。在这些情况下，只需使用 `@ImportResource`，并只定义所需的`XML`。这样做实现了一种“以`Java`为主”的方法来配置容器并将尽可能少的使用`XML`。

```

@Configuration
@ImportResource("classpath:/com/acme/properties-config.xml")
public class AppConfig {

    @Value("${jdbc.url}")
    private String url;

    @Value("${jdbc.username}")
    private String username;

    @Value("${jdbc.password}")
    private String password;

    @Bean
    public 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:mysql: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);
    // ...
}

```

## 3.13 Environment抽象

`Environment` 是集成在容器中的抽象，他包含了两个方面: `profiles` 和 `properties`。

`profile` 配置是一个被命名的，`bean`定义的逻辑组，这些`bean`只有在给定的`profile`配置激活时才会注册到容器。无论是以XML还是通过注解定义，`Bean`都可以分配给配置文件。

`Environment` 对象在`profile`中的角色是判断哪一个`profile`应该在当前激活和哪一个`profile`应该在默认情况下激活。

在所有的应用中，`Properties`属性扮演一个非常重要的角色，可能来源于一下源码变量:`properties`文件，JVM系统属性，系统环境变量，JNDI，`servlet`上下文参数，点对点的属性对象，Maps等。`Environment` 对象的作用，对于`properties`来说，是提供给用户方便的服务接口，方便撰写配置、方便解析配置。

### 3.13.1 bean定义profiles

bean定义profiles是核心容器内的一种机制，该机制能在不同环境中注册不同的bean。环境的意思是，为不同的用户做不同的事儿，该功能在很多场景中都非常有用，包括：

- 开发期使用内存数据源，在QA或者产品上则使用来自JNDI的相同的数据源
- 开发期使用监控组件，当部署以后则关闭监控组件，是应用更高效
- 为用户各自注册自定义bean实现

考虑一个实际应用中的场景，现在需要一个 `DataSource`。在测试环境中，这样配置：

```
@Bean
public DataSource dataSource() {
    return new EmbeddedDatabaseBuilder()
        .setType(EmbeddedDatabaseType.HSQL)
        .addScript("my-schema.sql")
        .addScript("my-test-data.sql")
        .build();
}
```

现在让我们考虑如何将此应用程序部署到QA或生产环境中，假设应用程序的数据源将注册到生产应用程序服务器的JNDI目录。我们的 `dataSource` bean现在看起来像这样：

```
@Bean(destroyMethod="")
public DataSource dataSource() throws Exception {
    Context ctx = new InitialContext();
    return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");
}
```

问题是，在当前环境中如何切换这两个配置。随着时间推移，Spring用户设计了很多种方式完成此切换，通常使用系统环境变量和XML `<import/>` 绑定，`<import/>` 元素包含一个  `${placeholder}`  符号，使用环境变量来设置  `${placeholder}`  符号所代表的值，从而达到切换正确配置文件的目的。bean定义profiles是核心容器功能，提供针对子问题的解决方案。

概括一下上面的场景：环境决定bean定义，最后发现，我们需要在某些上下文环境中使用某些bean，在其他环境中则不用这些bean。你也许会说，你需要在场景A中注册一组bean定义，在场景B中注册另外一组。先看看我们如何修改配置来完成此需求。

## @Profile

`@Profile` 注解用于当一个或多个配置文件激活的时候，用来指定组件是否有资格注册。使用上面的例子，我们可以重写 `dataSource` 配置如下：

```

@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(destroyMethod="")
    public DataSource dataSource() throws Exception {
        Context ctx = new InitialContext();
        return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");
    }
}

```



如前所述，使用 `@Bean` 方法，通常会选择使用JNDI查找:使用Spring的 `JndiTemplate` / `JndiLocatorDelegate helper`或上面显示的直接JNDI `InitialContext` 用法，但不是 `JndiObjectFactoryBean` 这将使你声明返回类型必须为 `FactoryBean` 类型

`@Profile` 可以用作**meta-annotation**，用于创建自定义组合注解。下面的例子定义了一个自定义 `@Production` 注解，该注解用于替换 `@Profile("production")`：

```

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Profile("production")
public @interface Production {
}

```

`@Profile` 也能注解方法，用于配置一个配置类中的指定bean:

```

@Configuration
public class AppConfig {

    @Bean
    @Profile("dev")
    public DataSource devDataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.HSQL)
            .addScript("classpath:com/bank/config/sql/schema.sql")
            .addScript("classpath:com/bank/config/sql/test-data.sql")
            .build();
    }

    @Bean
    @Profile("production")
    public DataSource productionDataSource() throws Exception {
        Context ctx = new InitialContext();
        return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");
    }
}

```



如果 `@Configuration` 类标有 `@Profile`，类中所有 `@Bean` 和 `@Import` 注解相关的类都将被忽略，除非该profile被激活。如果一个 `@Component` 或 `@Configuration` 类被标记为 `@Profile({"p1", "p2"})`，那么除非profile'p1'和/或'p2'已被激活。否则该类将不会注册/处理。如果给定的配置文件以NOT运算符( ! )为前缀，如果配置文件为 **not active**，则注册的元素将被注册。例如，给定 `@Profile({"p1", "! p2"})`，如果配置文件'p1'激活或配置文件'p2'没有激活，则会注册。

### 3.13.2 XML bean 定义 profile

XML 中的 `<beans>` 元素有一个 `profile` 属性。上面的示例配置可以重写为两个 XML 文件，如下所示：

```
<beans profile="dev"
    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="...">

    <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"
    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="...">

    <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>
</beans>
```

也可以不用分开 2 个文件，在同一个 XML 中配置 2 个 `<beans/>`，`<beans/>` 元素也有 `profile` 属性：

```

<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="...">>

    <!-- other bean definitions -->

    <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>

```

The `spring-bean.xsd` has been constrained to allow such elements only as the last ones in the file. This should help provide flexibility without incurring clutter in the XML files. `spring-bean.xsd` 强制允许将 `profile` 元素定义在文件的最后面，这有助于在 XML 文件中提供灵活的方式而又不引起混乱。

## 启用 `profile`

现在我们已经更新了配置，我们仍然需要指定要激活哪个配置文件。如果我们现在启动我们的示例程序，我们将看到一个 `NoSuchBeanDefinitionException` 抛出，因为容器找不到名为 `dataSource` 的 Spring bean。

激活配置文件可以通过几种方式完成，但最直接的是通过 `ApplicationContext` 以编程方式来处理'Environment'API:

```

AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
ctx.getEnvironment(). setActiveProfiles("dev");
ctx.register(SomeConfig.class, StandaloneDataConfig.class, JndiDataConfig.class);
ctx.refresh();

```

此外，配置文件也可以通过 `spring.profiles.active` 属性声明性地激活，可以通过系统环境变量，JVM 系统属性，`web.xml` 中的 `servle` 上下文参数指定，甚至作为 JNDI 中的一个条目来设置，参见第3.13.3节“PropertySource抽象”。在集成测试中，可以通过 `spring-test` 模块中的 `@ActiveProfiles` 注解来声明活动配置文件(参见“使用环境配置文件的上下文配置”一节)。

注意，配置文件不是“二选一”的；你可以一次激活多个配置文件。以编程方式，只需要在 `setActiveProfiles()` 方法提供多个配置文件的名字即可，这里接收的 `String...` 可变参数：

```
ctx.getEnvironment().setActiveProfiles("profile1", "profile2");
```

声明式的使用 `spring.profiles.active`，值可以为逗号分隔的配置文件名列表：

```
-Dspring.profiles.active="profile1,profile2"
```

## 默认profile配置

`default` 配置文件表示默认开启的profile配置。考虑以下：

```
@Configuration
@Profile("default")
public class DefaultDataConfig {

    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.HSQL)
            .addScript("classpath:com/bank/config/sql/schema.sql")
            .build();
    }
}
```

如果没有配置文件激活，上面的 `dataSource` 就会被创建。这提供了一种默认的方式。如果有任何一个配置文件启用，`default` 配置就不会生效。

默认配置文件的名字(`default`)可以通过 `Environment` 的 `setDefaultProfiles` 方法或者 `spring.profiles.default` 属性修改。

### 3.13.3 PropertySource abstraction

Spring的 `Environment` 抽象提供用于一系列的property sources属性配置文件的搜索操作。要充分说明，请考虑以下内容：

```
ApplicationContext ctx = new GenericApplicationContext();
Environment env = ctx.getEnvironment();
boolean containsFoo = env.containsProperty("foo");
System.out.println("Does my environment contain the 'foo' property? " + containsFoo);
```

在上面的代码段中，我们看到一个高级别的方法来询问Spring是否为当前环境定义了 `foo` 属性。为了回答这个问题，`Environment` 对象对一组 `PropertySource` 对象进行搜索。

`PropertySource` 是对任何键值对的简单抽象，Spring的 `StandardEnvironment` 配置有两个 `PropertySource` 对象 - 一个表示JVM系统属性(*a la* `System.getProperties()`)，一个表示系统环境变量(*a la* `System.getenv()`)。

这些默认property源在“StandardEnvironment”中存在，用于独立应用程序。  
`StandardServletEnvironment` 用默认的property配置源填充，默认配置源包括servlet配置和servlet上下文参数。它可以选择启用 `JndiPropertySource`。有关详细信息，请参阅 javadocs.

具体来说，当使用 `StandardEnvironment` 时，如果在运行时存在 `foo` 系统属性或 `foo` 环境变量，`env.containsProperty("foo")` 的调用将返回true。

执行的搜索是分层的。默认情况下，系统属性优先于环境变量，因此如果在调用 `env.getProperty("foo")` 时，两个地方都设置了 `foo` 属性，系统属性值返回优先于环境变量。注意，属性值不会被合并，而是被前面的条目完全覆盖。对于公共的“`StandardServletEnvironment`”，完整的层次结构如下，最高优先级条目在顶部:`ServletConfig`参数(如果适用，例如`case ServletContext参数(web.xml上下文参数条目)`)`JNDI`环境变量(“`java:comp / env`”条目)`JVM`系统属性(“`-D`命令行参数)`JVM`系统环境系统环境变量)

最重要的是，整个机制是可配置的。也许你需要一个自定义的properties源，并将该源整合到这个检索层级中。没问题 - 只需简单实现和实例化你自己的`PropertySource`，并将其添加到当前“`Environment`”的“`PropertySources`”集中：

```
ConfigurableApplicationContext ctx = new GenericApplicationContext();
MutablePropertySources sources = ctx.getEnvironment().getPropertySources();
sources.addFirst(new MyPropertySource());
```

在上面的代码中，“MyPropertySource”在搜索中以最高优先级添加。如果它包含一个 `foo` 属性，，它将会被探测并返回，优先于其他 `PropertySource` 中的 `foo` property属性。`MutablePropertySources` API暴露了很多方法，允许你精准的操作property属性源。

### 3.13.4 @PropertySource

`@PropertySource` 注解提供了一个方便的方式，用于增加一个 `PropertySource` 到 Spring 的“Environment”中。

给定一个包含键/值对“`testbean.name = myTestBean`”的文件“`app.properties`”，以下 `@Configuration` 类使用 `@PropertySource`，调用 `testBean.getName()` 返回“`myTestBean`”。

```
@Configuration
@PropertySource("classpath:/com/myco/app.properties")
public class AppConfig {
    @Autowired
    Environment env;

    @Bean
    public TestBean testBean() {
        TestBean testBean = new TestBean();
        testBean.setName(env.getProperty("testbean.name"));
        return testBean;
    }
}
```

任何的存在于 `@PropertySource` 中的  `${...}`  占位符，将被解析为定义在环境中的属性配置文件中的属性值：

```
@Configuration
@PropertySource("classpath:/com/${my.placeholder:default/path}/app.properties")
public class AppConfig {
    @Autowired
    Environment env;

    @Bean
    public TestBean testBean() {
        TestBean testBean = new TestBean();
        testBean.setName(env.getProperty("testbean.name"));
        return testBean;
    }
}
```

假设“`my.placeholder`”存在于已经注册的一个属性源中，例如 系统属性或环境变量，占位符将被解析为相应的值。如果没有，那么 `default/path` 将被用作默认值。如果未指定默认值，那么 `property` 将不能解析，，则将抛出 `IllegalArgumentException`。

### 3.13.5 Placeholder resolution in statements

以前，元素中占位符的值只能针对JVM系统属性或环境变量进行解析。现在不再是这种情况。因为环境抽象集成在整个容器中，所以很容易通过它来对占位符进行解析。这意味着你可以以任何你喜欢的方式来配置这个解析过程：可以改变是优先查找系统properties或者是有限查找环境变量，或者删除它们；增加自定义property源，使之成为更合适的。

具体来说，无论“自定义”属性定义在何处，以下语句都会工作，只要它在“Environment”中可用：

```
<beans>
    <import resource="com/bank/service/${customer}-config.xml"/>
</beans>
```

## 3.14 注册LoadTimeWeaver

LoadTimeWeaver被Spring用来在将类加载到Java虚拟机(JVM)中时动态地转换类。

若要开启加载时织入，得在 `@Configuration` 类中增加 `@EnableLoadTimeWeaving`：

```
@Configuration  
@EnableLoadTimeWeaving  
public class AppConfig {  
  
}
```

或者对于XML配置使用 `context:load-time-weaver` 元素：

```
<beans>  
    <context:load-time-weaver/>  
</beans>
```

一旦配置为 `ApplicationContext`。该`ApplicationContext`中的任何bean都可以实现 `LoadTimeWeaverAware`，从而接收对load-time weaver实例的引用。这特别适用于[Spring的JPA支持](#)，其中load-time weaving加载织入对JPA类转换非常必要。参考 `LocalContainerEntityManagerFactoryBean` javadocs更多的细节。有关AspectJ加载时编织的更多信息，请参见[第7.8.4节“Spring框架中使用AspectJ加载时编织”](#)。

## 3.15 ApplicationContext 的附加功能

正如在章节介绍中讨论的，`org.springframework.beans.factory` 包提供了用于管理和操作 bean 的基本功能，包括以编程方式。`org.springframework.context` 包增加了 `ApplicationContext` 接口，它继承了 `BeanFactory` 接口，除了扩展其他接口外还提供更多的 *application framework-oriented style*(面向应用程序框架的风格)的附加功能。许多人通过完全声明的方式使用 `ApplicationContext`，甚至不是以编程方式创建它，而是依赖于诸如 `ContextLoader` 这样的支持类来自动实例化 `ApplicationContext`，并将它作为 Java EE web 应用程序的普通启动过程的一部分。

为了以更加面向框架的方式增强 `BeanFactory` 的功能，上下文包还提供了以下功能：

- 通过 `MessageSource` 接口访问 `i18n-style` 中的消息。
- 通过 `ResourceLoader` 接口访问资源，例如 URL 和文件。
- 事件发布，即通过使用 `ApplicationEventPublisher` 接口给实现了 `ApplicationListener` 接口的 bean 发布事件。
- 通过 `HierarchicalBeanFactory` 接口，加载多级 contexts，允许关注某一层级 context，比如应用的 web 层。

### 3.15.1 Internationalization using MessageSource

`ApplicationContext` 接口扩展了一个名为 `MessageSource` 的接口，因此提供了国际化(i18n)功能。Spring还提供了接口 `HierarchicalMessageSource`，它可以分层解析消息。这些接口一起提供了Spring对消息解析的基础。在这些接口上定义的方法包括：

- `String getMessage(String code, Object [] args, String default, Locale loc)` :用于从“`MessageSource`”中检索消息的基本方法。当找不到指定区域设置的消息时，将使用默认消息。任何传递的参数都将成为替换值，使用标准库提供的 `MessageFormat` 功能。
- `String getMessage(String code, Object [] args, Locale loc)` :本质上与上一个方法相同，但有一个区别：不能指定默认消息；如果找不到消息，则抛出 `NoSuchMessageException`。
- `String getMessage(MessageSourceResolvable resolvable, Locale locale)` :在前面的方法中使用的所有属性也被包装在一个名为 `MessageSourceResolvable` 的类中，你可以使用这个方法。

当一个 `ApplicationContext` 被加载时，它会自动搜索在上下文中定义的一个 `MessageSource` bean。bean必须有名称 `messageSource`。如果找到这样的bean，则将对前面方法的所有调用委派给消息源。如果没有找到消息源，`ApplicationContext` 会尝试找到一个包含同名bean的父对象。如果是，它使用那个bean作为 `MessageSource`。如果 `ApplicationContext` 找不到消息的任何源，一个空的 `DelegatingMessageSource` 被实例化，以便能够接受对上面定义的方法的调用。

Spring提供了两个 `MessageSource` 实现，`ResourceBundleMessageSource` 和 `StaticMessageSource`。两者实现 `HierarchicalMessageSource` 为了做嵌套消息。“`StaticMessageSource`”很少使用，但提供了程序化的方式来添加消息到源。`ResourceBundleMessageSource` 如下例所示：

```
<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>
```

在示例中，假设你在类路径中定义了三个资源包，称为“format”，“exceptions”和“windows”。任何解析消息的请求都将以JDK标准方式处理，通过ResourceBundles解析消息。为了示例的目的，假设上述两个资源束文件的内容是...

```
# in format.properties
message=Alligators rock!
```

```
# in exceptions.properties
argument.required=The {0} argument is required.
```

A program to execute the `MessageSource` functionality is shown in the next example.

Remember that all `ApplicationContext` implementations are also `MessageSource` implementations and so can be cast to the `MessageSource` interface. 执行 `MessageSource` 功能的程序如下例所示。记住所有的 `ApplicationContext` 实现也是 `MessageSource` 实现，因此可以被转换到 `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);
}
```

上述程序产生的输出将是...

```
Alligators rock!
```

总而言之，`MessageSource` 在一个名为 `beans.xml` 的文件中定义，该文件存在于类路径的根目录下。`messageSource` bean 定义通过其 `basenames` 属性引用了大量的资源束。在列表中传递到 `basenames` 属性的三个文件作为文件存在于类路径的根目录，分别称为 `format.properties`，`exceptions.properties` 和 `windows.properties`。

下一个示例显示传递给消息查询的参数；这些参数将被转换为字符串并插入查找消息中的占位符。

```

<beans>

    <!-- this MessageSource is being used in a web application -->
    <bean id="messageSource" class="org.springframework.context.support.ResourceBundle
MessageSource">
        <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);
    }

}

```

调用 `execute()` 方法产生的输出将是...

```
The userDao argument is required.
```

关于国际化(i18n)，Spring的各种“MessageSource”实现遵循与标准JDK ResourceBundle 相同的区域设置解析和回滚规则。简而言之，继续前面定义的“messageSource”示例，如果要针对英国( en-GB )语言环境解析消息，你将创建名

为 `format_en_GB.properties`，`exceptions_en_GB.properties` 的文件，  
`windows_en_GB.properties`。

通常，区域设置解析由应用程序的环境配置管理。在此示例中，手动指定要解析(英国)邮件的区域设置。

```
# 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);
}
```

运行上述程序的输出将是...

```
Ebagum lad, the 'userDao' argument is required, I say, required.
```

你还可以使用 `MessageSourceAware` 接口来获取对已定义的任何 `MessageSource` 的引用。在创建和配置bean时，在实现`MessageSourceAware`接口的 `ApplicationContext` 中定义的任何 bean都会注入应用程序上下文的“`MessageSource`”。

作为 `ResourceBundleMessageSource` 的替代，`Spring` 提供了一个 `ReloadableResourceBundleMessageSource` 类。这个变体支持相同的 `bundle` 文件格式，但是比标准的基于 `JDK` 的 `ResourceBundleMessageSource` 实现更灵活。特别是，它允许从任何 `Spring` 资源位置(不仅仅是从类路径)读取文件，并支持 `bundle` 重新加载 `bundle` 属性文件 同时有效地将它们缓存存在其间)。查看 `ReloadableResourceBundleMessageSource` javadocs 可以进一步了解详细信息。

### 3.15.2 Standard and Custom Events

ApplicationContext 中的事件处理通过 ApplicationEvent 类和 ApplicationListener 接口来提供。如果实现 ApplicationListener 接口的bean被部署到context上下文中，那么每当一个 ApplicationEvent 被发布到 ApplicationContext 时，该bean就会收到通知。其实，这是一个标准的的观察者模式。



As of Spring 4.2, the event infrastructure has been significantly improved and offer an [annotation-based model](#) as well as the ability to publish any arbitrary event, that is an object that does not necessarily extend from `ApplicationEvent`. When such an object is published we wrap it in an event for you.

从4.2开始，事件基础结构已经得到显着改进，并提供了一个[基于注解的模型](#)使其有发布任意事件的能力，即便这个对象不一定是从ApplicationEvent扩展的。当这样的对象被发布时，我们会将它包装在一个事件中。

Spring提供以下标准事件：

**Table 3.7. Built-in Events**

Event	Explanation
ContextRefreshedEvent	当 ApplicationContext 初始化或者刷新时发布，比如，使用 ConfigurableApplicationContext 接口的 refresh() 方法。这里“初始化”的意思是指，所有的bean已经被加载、post-processor后处理bean已经被探测到并激活，单例bean已经 pre-instantiated 预先初始化，并且 ApplicationContext 对象已经可用。只要 context 上下文未关闭，可以多次触发刷新动作，某些 ApplicationContext 支持“热”刷新。比如， XmlWebApplicationContext 支持热刷新， GenericApplicationContext 就不支持。
ContextStartedEvent	当 ApplicationContext 启动时候发布，使用 ConfigurableApplicationContext 接口的 start() 方法。这里的“启动”意思是指，所有的 Lifecycle 生命周期 bean 接收到了明确的启动信号。通常，这个信号用来在明确的“停止”指令之后重启 beans，不过也可能是使用了启动组件，该组件并未配置自动启动，比如：组件在初始化的时候并未启动。
ContextStoppedEvent	当 ApplicationContext 停止时发布，使用 ConfigurableApplicationContext 接口的 stop() 方法。这里的“停止”的意思是指所有的 Lifecycle 生命周期 bean 接收到了明确的停止信号。一个停止了的 context 上下文可以通过 start() 调用重启。
ContextClosedEvent	当 ApplicationContext 关闭时候发布，使用 ConfigurableApplicationContext 接口的 close() 方法。这里“关闭”的意思是所有的单例 bean 已经销毁。一个关闭的 context 上下文达到的生命周期的重点。不能刷新，不能重启。
RequestHandledEvent	是一个 web 专用事件，告诉所有的 beans：一个 HTTP request 正在处理。这个时间在 request 处理完成之后发布。该事件仅适用于使用 Spring 的 DispatcherServlet 的 web 应用。

您还可以创建和发布自己的自定义事件。这个例子演示了一个简单的类，它继承了 Spring 的 ApplicationEvent 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...
}
```

要发布一个自定义的 `ApplicationEvent`，在 `ApplicationEventPublisher` 上调用 `publishEvent()` 方法。通常这是通过创建一个实现 `ApplicationEventPublisherAware` 并将它注册为 Spring bean 的类来实现的。看样例：

```
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...
    }

}
```

在配置期间，Spring 容器将检测到 `EmailService` 实现了 `ApplicationEventPublisherAware`，并将自动调用 `setApplicationEventPublisher()`。实际上，传入的参数将是 Spring 容器本身；你只需通过 `ApplicationEventPublisher` 接口就可以很简单的在应用程序上下文进行交互。

要接收自定义的 `ApplicationEvent`，首先创建一个实现 `ApplicationListener` 的类然后注册为一个 Spring bean。看例子：

```
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` 一般是用你自定义事件的类型 `BlackListEvent` 作为范型类参数化的。这意味着 `onApplicationEvent()` 方法可以保持类型安全，无需向下转型。`event listener` 事件监听想注册多少就注册多少，但请注意，默认情况下事件侦听器同步接收事件。这意味着 `publishEvent()` 方法阻塞，直到所有侦听器都完成处理事件。这种同步和单线程方法的一个优点是，当侦听器接收到事件时，如果事务上下文可用，它在发布者的事务上下文中操作。如果需要其他的事件发布策略，请参考[JavaDoc for Spring](#)的 `ApplicationEventMulticaster` 接口。

下面的样例展示了之前提到过的类如何定义bean并注册、配置：

```
<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>
```

综合起来，当 `emailService` bean 的 `sendEmail()` 方法被调用时，如果有任何电子邮件应该被列入黑名单，则会发布一个 `BlackListEvent` 类型的自定义事件。`blackListNotifier` bean 被注册为一个 `ApplicationListener`，因此接收 `BlackListEvent`，此时它可以通知适当的对象。

Spring的事件机制被设计用于在同一应用程序上下文中的Spring bean之间进行简单的通信。然而，对于更复杂的企业集成需求，有单独维护的[Spring Integration](#) 项目提供了完整的支持并可用于构建轻量级，[pattern-oriented\(面向模式\)](#)，依赖Spring编程模型的事件驱动架构

## Annotation-based Event Listeners

从Spring 4.2开始，事件监听器可以通过 `EventListener` 注解在托管bean的任何公共方法上注册。`BlackListNotifier` 可以改写如下：

```

public class BlackListNotifier {

    private String notificationAddress;

    public void setNotificationAddress(String notificationAddress) {
        this.notificationAddress = notificationAddress;
    }

    @EventListener
    public void processBlackListEvent(BlackListEvent event) {
        // notify appropriate parties via notificationAddress...
    }

}

```

As you can see above, the method signature actually *infer* which even type it listens to. This also works for nested generics as long as the actual event resolves the generics parameter you would filter on.

If your method should listen to several events or if you want to define it with no parameter at all, the event type(s) can also be specified on the annotation itself:

正如你上面所看到的，方法签名实际上用来推断它所侦听的类型。这也适用于嵌套泛型，只要实际事件解析了你将过滤的泛型参数。

如果你的方法应该监听几个事件，或者如果你想定义的这个方法没有任何参数，也可以在注解本身上指定事件类型：

```

@EventListener({ContextStartedEvent.class, ContextRefreshedEvent.class})
public void handleContextStart() {
}

```

It is also possible to add additional runtime filtering via the `condition` attribute of the annotation that defines a `SpEL expression` that should match to actually invoke the method for a particular event.

For instance, our notifier can be rewritten to be only invoked if the `test` attribute of the event is equal to `foo` :

也可以通过定义一个 `SpEL 表达式` 的注解的 `condition` 属性添加额外的运行时过滤，它们应该匹配以实际调用特定事件的方法。

例如，如果事件的 `test` 属性等于 `foo`，我们的通知器可以重写为仅被调用：

```

@EventListener(condition = "#b1Event.test == 'foo'")
public void processBlackListEvent(BlackListEvent b1Event) {
    // notify appropriate parties via notificationAddress...
}

```

Each SpEL expression evaluates again a dedicated context. The next table lists the items made available to the context so one can use them for conditional event processing:

每个“SpEL”表达式再次评估一个专用上下文。下一个表列出了可用于上下文的项目，因此可以使用它们进行条件事件处理：

**Table 3.8. Event SpEL available metadata**

Name	Location	Description	Example
Event	root object	The actual ApplicationEvent	#root.event
Arguments array	root object	The arguments (as array) used for invoking the target	#root.args[0]
Argument name	evaluation context	Name of any of the method arguments. If for some reason the names are not available (e.g. no debug information), the argument names are also available under the #a<#arg> where #arg stands for the argument index (starting from 0).	#b1Event or #a0 (one can also use #p0 or #p<#arg> notation as an alias).

注意，#root.event 允许你访问底层事件，即使你的方法签名实际上引用了一个被发布的任意对象。

如果你需要发布一个事件作为处理另一个事件的结果，只需更改方法签名返回应该发布的事件(即更改方法返回类型)，如：

```

@EventListener
public ListUpdateEvent handleBlackListEvent(BlackListEvent event) {
    // notify appropriate parties via notificationAddress and
    // then publish a ListUpdateEvent...
}

```

 异步监听器不支持此功能。

这将通过上述方法处理每个“BlackListEvent”并发布一个新的“ListUpdateEvent”。如果你需要发布几个事件，只需返回一个 Collection 的事件。

## Asynchronous Listeners

If you want a particular listener to process events asynchronously, simply reuse the regular `@Async` support:

如果你想要一个特定的监听器异步处理事件，只需重复使用 regular `@Async` support:

```
@EventListener
@Async
public void processBlackListEvent(BlackListEvent event) {
    // BlackListEvent is processed in a separate thread
}
```

使用异步事件时请注意以下限制:

如果事件监听器抛出一个 `Exception`，它不会传播给调用者，请检查 `AsyncUncaughtExceptionHandler` 以获取更多细节。2.此类事件监听器无法发送回复。如果你需要将处理的结果发送另一个事件，注入 `ApplicationEventPublisher` 来手动发送此事件。

## Ordering Listeners

如果你需要在另一个监听器之前调用某个监听器，只需在方法声明中添加 `@Order` 注解:

```
@EventListener
@Order(42)
public void processBlackListEvent(BlackListEvent event) {
    // notify appropriate parties via notificationAddress...
}
```

## Generic Events

你还可以使用泛型来进一步定义事件的结构。考虑一个 `EntityCreatedEvent`，其中 `T` 是创建的实际实体的类型。你可以创建以下监听器定义，以便只接收 `Person` 的 `EntityCreatedEvent`:

```
@EventListener
public void onPersonCreated(EntityCreatedEvent<Person> event) {
    ...
}
```

由于泛型擦除，只有此事件符合事件监听器所过滤的通用参数条件，那么才会触发相应的处理事件 (有点类似于 `class PersonCreatedEvent extends EntityCreatedEvent { ... }`)

在某些情况下，如果所有事件遵循相同的结构(如上述事件的情况)，这可能变得相当乏味。在这种情况下，你可以实现 `ResolvableTypeProvider` 来引导框架超出所提供的运行时环境范围：

```
public class EntityCreatedEvent<T>
    extends ApplicationEvent implements ResolvableTypeProvider {

    public EntityCreatedEvent(T entity) {
        super(entity);
    }

    @Override
    public ResolvableType getResolvableType() {
        return ResolvableType.forClassWithGenerics(getClass(),
            ResolvableType.forInstancegetSource()));
    }
}
```



这不仅适用于 `ApplicationEvent`，而且可以作为一个事件发送任何一个任意对象。

### 3.15.3 通过便捷的方式访问底层资源

为了最佳使用和理解应用程序上下文，推荐大家通过Spring的Resource abstraction资源抽象熟悉他们，如第4章，资源一节所述。

应用程序上下文是一个 ResourceLoader，可以用来加载 Resources。一个 Resource 本质上是 JDK类 java.net.URL 的一个扩展，实际上 Resource 的实现类中大多含有 java.net.URL 的实例。 Resource 几乎能从任何地方透明的获取底层资源，可以是classpath类路径、文件系统、标准的URL资源及变种URL资源。如果资源定位字串是简单的路径，没有任何特殊前缀，就适合于实际应用上下文类型。

可以配置一个bean部署到应用上下文中，用以实现特殊的回调接口， ResourceLoaderAware，它会在初始化期间自动回调。可以暴露 Resource 的type属性，这样就可以访问静态资源；静态资源可以像其他properties那样被注入 Resource。可以使用简单的字串路径指定资源，这要依赖于特殊的JavaBean PropertyEditor，该类是通过context自动注册，当bean部署时候它将转换资源中的字串为实际的资源对象。

提供给 ApplicationContext 构造函数的一个或多个位置路径实际上是资源字符串，并且以简单形式对特定上下文实现进行适当处理。 ClassPathXmlApplicationContext 将一个简单的定位路径视为类路径位置。你还可以使用带有特殊前缀的定位路径，这样就可以强制从classpath 或者URL定义加载路径，而不用考虑实际的上下文类型。

### 3.15.4 快速对web应用的 ApplicationContext 实例化

你可以通过使用一个 `ContextLoader` 来声明性地创建 `ApplicationContext` 实例。当然你也可以通过使用 `ApplicationContext` 的一个实现来编程实现 `ApplicationContext`。

你可以使用 `ContextLoaderListener` 注册一个 `ApplicationContext`，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-cl
ass>
</listener>
```

监听器检查 `contextConfigLocation` 参数。如果参数不存在，则监听器使用 `/WEB-INF /applicationContext.xml` 作为默认值。如果参数存在，监听器通过使用预定义的分隔符(逗号，分号和空格)分隔String，并使用分割后的值作为将搜索应用程序上下文的位置。也支持Ant样式路径风格。比如，`WEB-INF/*Context.xml`，将会匹配"WEB-INF"目录下所有以"Context.xml"结尾的file文件，`/WEB-INF/**/*Context.xml`，将会匹配"WEB-INF"下所有层级子目录的 `Context.xml` 结尾的文件。

### 3.15.5 将 Spring ApplicationContext 作为 Java EE RAR 文件部署

可以将 Spring ApplicationContext 部署为 RAR 文件，将上下文及其所有必需的 bean 类和库 JAR 封装在 Java EE RAR 部署单元中。这相当于引导一个独立的 ApplicationContext，只是托管在 Java EE 环境中，能够访问 Java EE 服务器设施。在部署无头 WAR 文件（实际上，没有任何 HTTP 入口点，仅用于在 Java EE 环境中引导 Spring ApplicationContext 的 WAR 文件）的情况下 RAR 部署是更自然的替代方案。

RAR 部署非常适合不需要 HTTP 入口点但仅由消息端点和调度作业组成的应用程序上下文。在这种情况下，Bean 可以使用应用程序服务器资源，例如 JTA 事务管理器和 JNDI 绑定的 JDBC DataSources 和 JMS ConnectionFactory 实例，并且还可以通过 Spring 的标准事务管理和 JNDI 和 JMX 支持设施向平台的 JMX 服务器注册。应用程序组件还可以通过 Spring 的 TaskExecutor 抽象实现与应用程序服务器的 JCA WorkManager 交互。

通过查看 [SpringContextResourceAdapter](#) 类的 JavaDoc，可以知道用于 RAR 部署中涉及的配置详细信息。

对于 Spring ApplicationContext 作为 Java EE RAR 文件的简单部署：将所有应用程序类打包到 RAR 文件中，这是具有不同文件扩展名的标准 JAR 文件。将所有必需的库 JAR 添加到 RAR 归档的根目录中。添加一个“META-INF / ra.xml”部署描述符（如 SpringContextResourceAdapter 的 JavaDoc 中所示）和相应的 Spring XML bean 定义文件（通常为“META-INF / applicationContext.xml”），导致 RAR 文件进入应用程序服务器的部署目录。

这种 RAR 部署单元通常是独立的；它们不会将组件暴露给外界，甚至不会暴露给同一应用程序的其他模块。与基于 RAR 的 ApplicationContext 的交互通常通过发生在与其他模块共享的 JMS 目标的情况下。基于 RAR 的 ApplicationContext 还会在其他情况下使用，例如调度一些作业，对文件系统中的新文件（等等）作出反应。如果需要允许从外部同步访问，它可以做到如导出 RMI 端点，然后很自然的可以由同一机器上的其他应用模块使用。

## 3.16 The BeanFactory

`BeanFactory` 为 Spring 的 IoC 功能提供了基础支撑，但是在集成第三方框架时只能直接使用，现在已经成为历史。`BeanFactory` 和相关接口，如 `BeanFactoryAware`，`InitializingBean`，`DisposableBean`，仍然存在于 Spring 中，目的是向下兼容与 Spring 集成的大量第三方框架。通常第三方组件不能使用更现代的等同物，例如 `@PostConstruct` 或 `@PreDestroy`，以便与 JDK 1.4 保持兼容或避免对 JSR-250 的依赖。

本部分主要讲解有关 `BeanFactory` 和 `ApplicationContext` 之间的不同的背景区别，以及如何通过经典的单例查找直接访问 IoC 容器。

## 3.16.1 BeanFactory or ApplicationContext?

优先使用 `ApplicationContext`，除非你有一个很好的理由不这样做。

因为 `ApplicationContext` 包括 `BeanFactory` 的所有功能，和 `BeanFactory` 相比更值得推荐，除了一些特定的场景，比如，在资源受限的设备上运行的内嵌的应用，这些设备非常关注内存消耗。无论如何，对于大多数的企业级应用和系统，`ApplicationContext` 都是首选。Spring 使用了大量的 `BeanPostProcessor` 扩展点 (以实现代理等)。如果你只使用一个简单的 `BeanFactory`，大量的功能将失效，比如 `transactions` 和 `AOP`，至少得多一些额外的处理。这种情况可能会令人困惑，因为没有什么是实际上错误的配置。

下表列出了“`BeanFactory`”和“`ApplicationContext`”接口和实现提供的功能。

**Table 3.9. Feature Matrix**

Feature	BeanFactory	ApplicationContext
bean实例化和组装	Yes	Yes
自动注册 <code>BeanPostProcessor</code>	No	Yes
自动注册 <code>BeanFactoryPostProcessor</code>	No	Yes
便利的消息资源访问(用于i18n)	No	Yes
<code>ApplicationEvent</code> 的发布	No	Yes

要使用 `BeanFactory` 实现显式地注册 bean 后置处理器 post-processor，你得需要编写如下代码：

```
DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
// populate the factory with bean definitions

// now register any needed BeanPostProcessor instances
MyBeanPostProcessor postProcessor = new MyBeanPostProcessor();
factory.addBeanPostProcessor(postProcessor);

// now start using the factory
```

要在使用 `BeanFactory` 实现时显式地注册一个 `BeanFactoryPostProcessor`，你必须编写如下代码：

```
DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(factory);
reader.loadBeanDefinitions(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);
```

在这两种情况下显示注册步骤非常不便，这就是为什么推荐使用 `ApplicationContext` 的原因之一，就是为了方便的使用 `BeanFactoryPostProcessors` 和 `BeanPostProcessors`。这些机制实现了一些非常重要的功能，例如属性占位符替换和AOP。

### 3.16.2 Glue code and the evil singleton

在应用中，推荐使用依赖注入的方式编写，使用Spring IoC容器管理的代码，当需要创建实例时，从容器获取他的依赖关系，并且对象对容器将对其一无所知。然而，对于有时需要将其他代码绑定在一起的小代码层，也许会需要与其他层、组件、bean互相协作，可以以单例（准单例）方式使用Spring IoC容器。例如，第三方组件可能尝试直接构造新对象（`Class.forName()` style），而无法从Spring IoC容器中获取这些对象如果第三方组件创建的对象是stub或者proxy代理，然后这些对象以单例风格从loc容器获取真正的对象加以委派，此时控制反转完成主要工作（对象将脱离于容器管理）。此时，大部分代码不需知道容器、也不需知道如何访问容器，好处就是代码解耦。EJBs也可以使用这种方式，代理的方式委派给简单的java实现对象，java对象从loc容器检出。

在服务定位器样式中查找应用程序上下文有时是访问共享Spring管理的组件的唯一途径，例如在EJB 2.1环境中，或者当你希望跨WAR文件将WebContextContext作为父对象共享到WebApplicationContexts时。在这种情况下，你应该使用实用程序类 `ContextSingletonBeanFactoryLocator` 定位器，在此[Spring team blog entry](#)有描述。

---

[1] 见 [Background](#)

[2] 见 [Section 3.4.1, “Dependency Injection”](#)



## 4.1 介绍

仅仅使用 java 标准 `java.net.URL` 和针对不同 `URL` 前缀的标准处理器并不能满足我们对各种底层资源的访问，比如：我们就不能通过 `URL` 的标准实现来访问相对类路径或者相对 `ServletContext` 的各种资源。虽然我们可以针对特定的 `url` 前缀来注册一个新的处理 `handler`（和现有的针对各种特定前缀的处理器类似，比如 `http:`），然而这往往是一件比较麻烦的事情(要求了解 `url` 的实现机制等)，而且 `url` 接口也缺少了部分基本的方法，如检查当前资源是否存在方法。

## 4.2 Resource 接口

相对标准 url 访问机制，spring 的 `Resource` 接口对抽象底层资源的访问提供了一套更好的机制。

```
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;

}
```

`Resource` 接口里的最重要的几个方法：

- `getInputStream()`：定位并且打开当前资源，返回当前资源的 `InputStream`。预计每一次调用都会返回一个新的 `InputStream`，因此关闭当前输出流就成为了调用者的责任。
- `exists()`：返回一个 `boolean`，表示当前资源是否真的存在。
- `isOpen()`：返回一个 `boolean`，表示当前资源是否一个已打开的输入流。如果结果为 `true`，返回的 `InputStream` 不能多次读取，只能是一次性读取之后，就关闭 `InputStream`，以防止内存泄漏。除了 `InputStreamResource`，其他常用 `Resource` 实现都会返回 `false`。
- `getDescription()`：返回当前资源的描述，当处理资源出错时，资源的描述会用于错误信息的输出。一般来说，资源的描述是一个完全限定的文件名称，或者是当前资源的真实 `url`。

Resource 接口里的其他方法可以让你获得代表当前资源的 URL 或 File 对象（前提是底层实现可兼容的，也支持该功能）。

在 spring 里，Resource 抽象有着相当广泛的使用，比如，当需要一个资源时，Resource 可以作为方法签名里的一个参数类型。在 spring api 中，有些方法（如各种 ApplicationContext 实现的构造函数）会直接采用普通格式的 string 路径来创建合适的 Resource，调用者也可以通过在路径里带上指定的前缀来创建特定 Resource 实现。

Resource 接口（实现）不仅可以被 spring 大量的应用，其也非常适合作为你编程中访问资源的辅助工具类。当你仅需要使用到 Resource 接口实现时，可以直接忽略 spring 的其余部分。单独使用 Rsoucece 实现，会造成代码与 spring 的部分耦合，可也仅耦合了其中一小部分辅助类，而且你可以将 Reource 实现作为 URL 的一种访问底层更为有效的替代，与你引入其他库来达到这种目的是一样的。

需要注意的是 Resource 实现并没有去重新发明轮子，而是尽可能地采用封装。举个例子，UrlResource 里就封装了一个 URL 对象，在其内的逻辑就是通过封装的 URL 对象 来完成的。

## 4.3 内置的 **Resource** 实现

spring 直接提供了多种开箱即用的 `Resource` 实现。

### 4.3.1 UrlResource

`UrlResource` 封装了一个 `java.net.URL` 对象，用来访问 URL 可以正常访问的任意对象，比如文件、an HTTP target, an FTP target, 等等。所有的 URL 都可以用一个标准化的字符串来表示。如通过正确的标准化前缀，可以用来表示当前 URL 的类型，当中就包括用于访问文件系统路径的 `file:` ; 通过 `http` 协议 访问资源的 `http:` , 通过 `ftp` 协议 访问资源的 `ftp:` , 还有很多.....

可以显式化地使用 `UrlResource` 构造函数来创建一个 `UrlResource`，不过通常我们可以在调用一个 api 方法是，使用一个代表路径的 `String` 参数来隐式创建一个 `UrlResource`。对于后一种情况，会由一个 `javabean` 的 `PropertyEditor` 来决定创建哪一种 `Resource`。如果路径里包含某一个通用的前缀（如 `classpath:` ）, `PropertyEditor` 会根据这个通用的前缀来创建恰当的 `Resource`；反之，如果 `PropertyEditor` 无法识别这个前缀，会把这个路径作为一个标准的 URL 来创建一个 `UrlResource`。

### 4.3.2 ClassPathResource ClassPathResource

可以从类路径上加载资源，其可以使用线程上下文加载器、指定加载器或指定的 class 类型中的任意一个来加载资源。

当类路径上资源存于文件系统中，ClassPathResource 支持以 `java.io.File` 的形式访问，可当类路径上的资源存于尚未解压(没有被Servlet 引擎或其他可解压的环境解压)的 jar 包中，ClassPathResource 就不再支持以 `java.io.File` 的形式访问。鉴于上面所说这个问题，spring 中各式 Resource 实现都支持以 `java.net.URL` 的形式访问。

可以显式使用 `ClassPathResource` 构造函数来创建一个 `ClassPathResource`，不过通常我们可以在调用一个 api 方法时，使用一个代表路径的 `String` 参数来隐式创建一个 `ClassPathResource`。对于后一种情况，会由一个 javabean 的 `PropertyEditor` 来识别路径中 `classpath:` 前缀，从而创建一个 `ClassPathResource`。

### 4.3.3 FileSystemResource

这是针对 `java.io.File` 提供的 `Resource` 实现。显然，我们可以使用 `FileSystemResource` 的 `getFile()` 函数获取 `File` 对象，使用 `getURL()` 获取 `URL` 对象。

### 4.3.4 ServletContextResource

这是为了获取 web 根路径的 ServletContext 资源而提供的 Resource 实现。

ServletContextResource 完全支持以流和 URL 的方式访问，可只有当 web 项目是已解压的（不是以 war 等压缩包形式存在）且该 ServletContext 资源存于文件系统里，ServletContextResource 才支持以 `java.io.File` 的方式访问。至于说到，我们的 web 项目是否已解压和相关的 ServletContext 资源是否会存于文件系统里，这个取决于我们所使用的 Servlet 容器。若 Servlet 容器没有解压 web 项目，我们可以直接以 JAR 的形式的访问，或者其他可以想到的方式（如访问数据库）等。

### 4.3.5 InputStreamResource

这是针对 `InputStream` 提供的 `Resource` 实现。建议，在确实没有找到其他合适的 `Resource` 实现时，才使用 `InputStreamResource`。如果可以，尽量选择 `ByteArrayResource` 或其他基于文件的 `Resource` 实现来代替。

与其他 `Resource` 实现一比较，`InputStreamResource` 倒像一个已打开资源的描述符，因此，调用 `isOpen()` 方法会返回 `true`。除了在需要获取资源的描述符或需要从输入流多次读取时，都不要使用 `InputStreamResource` 来读取资源。

## 4.3.6 ByteArrayResource

原文应该有误(贴出如下): This is a `Resource` implementation for a given byte array. It creates a `ByteArrayInputStream` for the given byte array.

It's useful for loading content from any given byte array, without having to resort to a single-use `InputStreamResource`. 然后附上 `ByteArrayResource` 源码,看个人注释

```
//  
// Source code recreated from a .class file by IntelliJ IDEA  
// (powered by Fernflower decompiler)  
  
  
package org.springframework.core.io;  
  
import java.io.ByteArrayInputStream;  
import java.io.IOException;  
import java.io.InputStream;  
import java.util.Arrays;  
import org.springframework.core.io.AbstractResource;  
  
public class ByteArrayResource extends AbstractResource {  
    //通过一个字节数组来创建 `ByteArrayResource`。  
    private final byte[] byteArray;  
    private final String description;  
    //通过一个字节数组来创建 `ByteArrayResource`。  
    public ByteArrayResource(byte[] byteArray) {  
        this(byteArray, "resource loaded from byte array");  
    }  
  
    public ByteArrayResource(byte[] byteArray, String description) {  
        if(byteArray == null) {  
            throw new IllegalArgumentException("Byte array must not be null");  
        } else {  
            this.byteArray = byteArray;  
            this.description = description != null?description:"";  
        }  
    }  
  
    public final byte[] getByteArray() {  
        return this.byteArray;  
    }  
  
    public boolean exists() {  
        return true;  
    }  
  
    public long contentLength() {  
        return (long)this.byteArray.length;  
    }
```

```
public InputStream getInputStream() throws IOException {
    return new ByteArrayInputStream(this.byteArray);
}

public String getDescription() {
    return "Byte array resource [" + this.description + "]";
}

public boolean equals(Object obj) {
    return obj == this || obj instanceof ByteArrayResource && Arrays.equals(((ByteArrayResource)obj).byteArray, this.byteArray);
}

public int hashCode() {
    return byte[].class.hashCode() * 29 * this.byteArray.length;
}
}
```

这是针对字节数组提供的 Resource 实现。可以通过一个字节数组来创建 ByteArrayResource。

当需要从字节数组加载内容时，ByteArrayResource 是一个不错的选择，使用 ByteArrayResource 可以不用求助于 InputStreamResource。

## 4.4 ResourceLoader 接口

`ResourceLoader` 接口是用来加载 `Resource` 对象 的，换句话说，就是当一个对象需要获取 `Resource` 实例 时，可以选择实现 `ResourceLoader` 接口。

```
public interface ResourceLoader {
    Resource getResource(String location);
}
```

`spring` 里所有的应用上下文都是实现了 `ResourceLoader` 接口，因此，所有应用上下文都可以通过 `getResource()` 方法获取 `Resource` 实例。

当你在指定应用上下文调用 `getResource()` 方法时，而指定的位置路径又没有包含特定的前缀，`spring` 会根据当前应用上下文来决定返回哪一种类型 `Resource`。举个例子，假设下面的代码片段是通过 `ClassPathXmlApplicationContext` 实例来调用的：

```
Resource template = ctx.getResource("some/resource/path/myTemplate.txt");
```

那 `spring` 会返回一个 `ClassPathResource` 对象；类似的，如果是通过一个 `FileSystemXmlApplicationContext` 实例调用的，返回的是一个 `FileSystemResource` 对象；如果是通过 `WebApplicationContext` 实例调用的，返回的是一个 `ServletContextResource` 对象.....

如上所说，你就可以在指定的应用上下文中使用 `Resource` 实例来加载当前应用上下文的资源。

还有另外一种场景里，如在其他应用上下文里，你可能会强制需要获取一个 `ClassPathResource` 对象，这个时候，你可以通过加上指定的前缀来实现这一需求，如：

```
Resource template = ctx.getResource("classpath:some/resource/path/myTemplate.txt");
```

类似的，你可以通过其他任意的 `url` 前缀 来强制获取 `UrlResource` 对象：

```
Resource template = ctx.getResource("file:///some/resource/path/myTemplate.txt");
```

```
Resource template = ctx.getResource("http://myhost.com/resource/path/myTemplate.txt");
```

下面，给出一个表格来总结一下 `spring` 根据各种位置路径加载资源的策略：

**Table 4.1. Resource strings**

前缀	样例	说明
classpath:	classpath:com/myapp/config.xml	从类路径加载
file:	file:///data/config.xml	将其作为 URL 对象，从文件系统加载 [1]
http:	http://myserver/logo.png	将其作为 URL 对象加载
(none)	/data/config.xml	取决于底层的 ApplicationContext
[1] 另请参阅:[Section 4.7.3, “FileSystemResource 警告”		

## 4.5 ResourceLoaderAware 接口

`ResourceLoaderAware` 是一个特殊的标记接口，用来标记提供 `ResourceLoader` 引用的对象。

```
public interface ResourceLoaderAware {
    void setResourceLoader(ResourceLoader resourceLoader);
}
```

当将一个 `ResourceLoaderAware` 接口的实现类部署到应用上下文时(此类会作为一个 `spring` 管理的 `bean`)，应用上下文会识别出此为一个 `ResourceLoaderAware` 对象，并将自身作为一个参数来调用 `setResourceLoader()` 函数，如此，该实现类便可使用 `ResourceLoader` 获取 `Resource` 实例来加载你所需要的资源。(附：为什么能将应用上下文作为一个参数来调用 `setResourceLoader()` 函数呢？不要忘了，在前文有谈过，`spring` 的所有上下文都实现了 `ResourceLoader` 接口)。

当然了，一个 `bean` 若想加载指定路径下的资源，除了刚才提到的实现 `ResourcesLoaderAware` 接口之外（将 `ApplicationContext` 作为一个 `ResourceLoader` 对象注入），`bean` 也可以实现 `ApplicationContextAware` 接口，这样可以直接使用应用上下文来加载资源。但总的来说，在需求满足都满足的情况下，最好是使用的专用 `ResourceLoader` 接口，因为这样代码只会与接口耦合，而不会与整个 `spring ApplicationContext` 耦合。与 `ResourceLoader` 接口耦合，抛开 `spring` 来看，就是 提供了一个加载资源的工具类接口。

从 `spring 2.5` 开始，除了实现 `ResourceLoaderAware` 接口，也可采取另外一种替代方案——依赖于 `ResourceLoader` 的自动装配。“传统”的 `constructor` 和 `bytype` 自动装配模式都支持 `ResourceLoader` 的装配（可参阅 [Section 3.4.5, “自动装配协作者”](#)）——前者以构造参数的形式装配，后者以 `setter` 方法中参数装配。若为了获得更大的灵活性(包括属性注入的能力和多参方法)，可以考虑使用基于注解的新注入方式。使用注解 `@Autowired` 标记 `ResourceLoader` 变量，便可将其注入到成员属性、构造参数或方法参数中(`@Autowired` 详细的使用方法可参考[Section 3.9.2, “@Autowired”](#).)。

## 4.6 Resources as dependencies

如果bean本身要通过某种动态过程来确定和提供资源路径，那么bean使用ResourceLoader接口来加载资源就变得有意义了。假如加载某种类型的模板，其中所需的特定资源取决于用户的角色。如果资源是静态的，那么完全可以舍弃不用 ResourceLoader 接口，只需让bean暴露它需要的 Resource 属性，并按照预期注入属性即可。

是什么使得注入这些属性变得如此简单，是因为所有应用程序上下文注册和使用一个特殊的 JavaBean 的 PropertyEditor，它可以将 String paths 转换为 Resource 对象。因此，如果 myBean 有一个类型为 Resource 的模板属性，它可以用一个简单的字符串配置该资源，如下所示：

```
<bean id="myBean" class="...">
    <property name="template" value="some/resource/path/myTemplate.txt"/>
</bean>
```

需要注意的是，资源路径没有前缀，因为应用程序上下文本身将被作为 ResourceLoader 来使用，资源本身将通过 classPathResource，FileSystemResource 或 ServletContextResource（视情况而定）来加载，这取决于上下文的确切类型。

如果需要强制使用特定的 Resource 类型，则可以使用前缀。以下两个示例显示如何强制使用 classPathResource 和 UrlResource（后者用于访问文件系统文件）。

```
<property name="template" value="classpath:some/resource/path/myTemplate.txt">

<property name="template" value="file:///some/resource/path/myTemplate.txt"/>
```



## 4.7.1 构造应用上下文

(某一特定) 应用上下文的构造器通常可以使用字符串或字符串数组所指代的(多个)资源(如 xml 文件)来构造当前上下文。

当指定的位置路径没有带前缀时，那从指定位置路径创建的 Resource 类型(用于后续加载 bean 定义),取决于所使用应用上下文。举个例子，如下所创建的

ClassPathXmlApplicationContext :

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("conf/appContext.xml");
```

会从类路径加载 bean 的定义，因为所创建的 Resource 实例是 ClassPathResource .但所创建的是 FileSystemXmlApplicationContext 时，

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("conf/appContext.xml");
```

则会从文件系统加载 bean 的定义，这种情况下，资源路径是相对工作目录而言的。

注意：若位置路径带有 classpath 前缀或 URL 前缀，会覆盖默认创建的用于加载 bean 定义的 Resource 类型，比如这种情况下的 FileSystemXmlApplicationContext

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("classpath:conf/appContext.xml");
```

，实际是从类路径下加载了 bean 的定义。可是，这个上下文仍然是 FileSystemXmlApplicationContext ，而不是 ClassPathXmlApplicationContext ，在后续作为 ResourceLoader 来使用时，不带前缀的路径仍然会从文件系统中加载。

## 构造 ClassPathXmlApplicationContext 实例 - 快捷方式

ClassPathXmlApplicationContext 提供了多个构造函数，以利于快捷创建 ClassPathXmlApplicationContext 的实例。最好莫过于使用只包含多个 xml 文件名 (不带路径信息) 的字符串数组和一个 Class 参数的构造器，所省略路径信息

ClassPathXmlApplicationContext 会从 Class 参数获取：

下面的这个例子，可以让你对个构造器有比较清晰的认识。试想一个如下类似的目录结构：

```
com/
  foo/
    services.xml
    daos.xml
    MessengerService.class
```

由 `services.xml` 和 `daos.xml` 中 bean 所组成的 `ClassPathXmlApplicationContext`，可以这样来初始化：

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

欲要知道 `ClassPathXmlApplicationContext` 更多不同类型的构造器，请查阅 [Javadocs](#) 文档。

## 4.7.2 使用通配符构造应用上下文

从前文可知，应用上下文构造器的中的资源路径可以是单一的路径（即一对一地映射到目标资源）；另外资源路径也可以使用高效的通配符——可包含 `classpath*`: 前缀或 **ant** 风格的正则表达式（使用 **spring** 的 `PathMatcher` 来匹配）。

通配符机制的其中一种应用可以用来组装组件式的应用程序。应用程序里所有组件都可以在一个共知的位置路径发布自定义的上下文片段，则最终应用上下文可使用 `classpath*`: 在同一路径前缀(前面的共知路径)下创建，这时所有组件上下文的片段都会被自动组装。

谨记，路径中的通配符特定用于应用上下文的构造器，只会在应用构造时有效，与其 `Resource` 自身类型没有任何关系。不可以使用 `classpath*`: 来构造任一真实的 `Resource`，因为一个资源点一次只可以指向一个资源。（如果直接使用 `PathMatcher` 的工具类，也可以在路径中使用通配符）

### Ant 风格模式

以下是一些使用了 Ant 风格的位置路径：

```
/WEB-INF/*-context.xml
com/mycompany/**/applicationContext.xml
file:C:/some/path/*-context.xml
classpath:com/mycompany/**/applicationContext.xml
```

- 当位置路径使用了 **ant** 风格，解析器会遵循一套复杂且预定义的逻辑来解释这些位置路径。解释器会先从位置路径里获取最靠前的不带通配符的路径片段，并使用这个路径片段来创建一个 `Resource`，并从 `Resource` 里获取其 URL，若所获取到 URL 前缀并不是 "jar:", 或其他特殊容器产生的特殊前缀（如 **WebLogic** 的 `zip:` , **WebSphere** 的 `wsjar` ），则从 `Resource` 里获取 `java.io.File` 对象，并通过其遍历文件系统。进而解决位置路径里通配符；若获取的是 "jar:" 的 URL，解析器会从其获取一个 `java.net.JarURLConnection` 或手动解析此 URL，并遍历 jar 文件的内容进而解决位置路径的通配符。

#### 可移植性所带来的影响

如果指定的路径已经是文件URL(不管是显式地或隐式地)，这是因为首先，基础“`ResourceLoader`”就是一文件系统，其次，通配符保证以完全可移植的方式进行工作。

如果指定的路径是类路径位置，则解析器必须通过 `ClassLoader.getResource()` 调用获取最后一个非通配符路径段URL。因为这只是路径的一个节点（而不是末尾的文件），它实际上是未定义的（在“`ClassLoader`”javadocs 中可查看细节），在这种情况下并不能确定返回什么样

的URL。实际上，它始终用一个 `java.io.File` 解析目录，其中类路径资源解析到文件系统位置，或某种类型的jar URL，其中类路径资源解析为jar位置。但是，这个操作还有一个可移植性问题。

如果获取了最后一个非通配符段的jar URL，解析器必须能够从中获取 `java.net.JarURLConnection`，或者手动解析jar URL，以便能够遍历jar的内容，并解析通配符。这适用于大多数工作环境，但在某些其他特定环境中会有问题，然后导致解析失败，强烈建议在特定环境中彻底测试来自jar的资源的通配符解析，测试成功之后再对其做依赖使用。

## classpath\*: 的可移植性

当构造基于 xml 文件的应用上下文时，位置路径可以使用 `classpath*:` 前缀：

```
ApplicationContext ctx =
    new ClassPathXmlApplicationContext("classpath*:conf/appContext.xml");
```

`classpath*:` 的使用表示类路径下所有匹配文件名称的资源都会被获取(本质上就是调用了 `ClassLoader.getResources(...)` 方法)，接着将获取到的资源组装成最终的应用上下文。

通配符路径依赖了底层 `classloader` 的 `getResources()` 方法。可是现在大多数应用服务器提供了自身的 `classloader` 实现，其处理 jar 文件的形式可能各有不同。要在指定服务器测试 `classpath*:` 是否有效，简单点可以使用 `getClass().getClassLoader().getResources("<someFileInsideTheJar>")` 去加载类路径 jar 包里的一个文件。尝试在两个不同的路径加载名称相同的文件，如果返回的结果不一致，就需要查看一下此服务器中与 `classloader` 行为设置相关的文档。

在位置路径的其余部分，`classpath*:` 前缀可以与 `PathMatcher` 结合使用，如：" `classpath*:META-INF/*-beans.xml`"。这种情况的解析策略非常简单：取位置路径最靠前的无通配符片段，调用 `ClassLoader.getResources()` 获取所有匹配的类层次加载器可加载的资源，随后将 `PathMatcher` 的策略应用于每一个获得的资源（起过滤作用）。

## 通配符的补充说明

除非所有目标资源都存于文件系统，否则 `classpath*:` 和 ant 风格模式的结合使用，都只能在至少有一个确定根包路径的情况下，才能达到预期的效果。换句话说，就是像 `classpath*:.xml` 这样的 pattern 不能从根目录的 jar 文件中获取资源，只能从根目录的扩展目录获取资源。此问题的造成源于 jdk 的 `ClassLoader.getResources()` 方法的局限性——当向 `ClassLoader.getResources()` 传入空串时(表示搜索潜在的根目录)，只能获取的文件系统的文件位置路径，即获取不了 jar 中文件的位置路径。

如果在多个类路径上存在所搜索的根包，那使用 `classpath:` 和 `ant` 风格模式一起指定的资源不保证找到匹配的资源。因为使用如下的 `pattern`

```
classpath:com/mycompany/**/service-context.xml
```

去搜索只在某一个路径存在的指定资源

```
com/mycompany/package1/service-context.xml
```

时，解析器只会对 `getResource("com/mycompany")` 返回的(第一个) URL 进行遍历和解释，则当在多个类路径存在基础包节点 "com/mycompany" 时(如在多个 jar 存在这个基础节点)，解析器就不一定会找到指定资源。因此，这种情况下建议结合使用 `classpath*:` 和 `ant` 风格模式，`classpath*:` 会让解析器去搜索所有包含基础包节点的类路径。

### 4.7.3 FileSystemResource 警告

`FileSystemResource` 并没有依附 `FileSystemApplicationContext`，因为 `FileSystemApplicationContext` 并不是一个真正的 `ResourceLoader`。`FileSystemResource` 并没有按约定规则来处理绝对和相对路径。相对路径是相对与当前工作而言，而绝对路径则是相对文件系统的根目录而言。

然而为了向后兼容，当 `FileSystemApplicationContext` 是一个 `ResourceLoader` 实例，我们做了一些改变——不管 `FileSystemResource` 实例的位置路径是否以 / 开头，`FileSystemApplicationContext` 都强制将其作为相对路径来处理。事实上，这意味着以下例子等效：

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("conf/context.xml");
```

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("/conf/context.xml");
```

还有：（即使它们的意义不一样——一个是相对路径，另一个是绝对路径。）

```
FileSystemXmlApplicationContext ctx = ...;
ctx.getResource("some/resource/path/myTemplate.txt");
```

```
FileSystemXmlApplicationContext ctx = ...;
ctx.getResource("/some/resource/path/myTemplate.txt");
```

实践中，如果确实需要使用绝对路径，建议放弃 `FileSystemResource` / `FileSystemXmlApplicationContext` 在绝对路径的使用，而强制使用 `file:` 的 `UrlResource`。

```
// Resource 只会是 UrlResource，与上下文的真实类型无关
ctx.getResource("file:///some/resource/path/myTemplate.txt");
// 强制 FileSystemXmlApplicationContext 通过 UrlResource 加载资源
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("file:///conf/context.xml");
```

## 5. 数据校验、数据绑定和类型转换

## 5.1 简介

### JSR-303/JSR-349 Bean Validation

Spring 4.0 默认即支持Bean Validation 1.0 (JSR-303)和 Bean Validation 1.1(JSR-349)校验规范，同时也能适配Spring的 `validator` 校验接口。

应用既可以像[Section 5.8, “Spring Validation”](#)所述一次性开启全局的Bean Validation，也可以在你需要验证的地方一一开启。

应用可以像[Section 5.8.3, “Configuring a DataBinder”](#)所述在每个 `DataBinder` 实例中注册多个自定义的Spring `Validator` 实例，这对那些不愿意使用注解来实现插件式的校验逻辑来说非常有用。

在业务逻辑中考虑数据校验利弊参半，Spring 提供的校验(和数据绑定)方案也未能解决这个问题。能明确的是数据校验不应该被限定在web层使用，它应该能很方便的执行本地化，并且能在任何需要 数据校验的场合以插件的形式提供服务。基于以上考虑，Spring 设计了一个既基本又方便使用且能在所有层使用的 `Validator` 接口。

Spring 提供了我们称作 `DataBinder` 的对象来处理数据绑定，所谓的数据绑定就是将用户的输入自动的绑定到我们的领域模型(或者说用来处理用户所输入的对象)。Spring 的 `validator` 和 `DataBinder` 构成了 `validation` 包，这个包主要被Spring MVC框架使用，但绝不限于在该框架使用。

在Spring中 `BeanWrapper` 是一个很基本的概念，在很多地方都有使用到它。但是，你可能从来都没有直接使用到它。鉴于这是一份参考文档，我们认为很有必要对 `BeanWrapper` 进行必要的解释。在这一章中我们将解释 `BeanWrapper`，在你尝试将数据绑定到对象时一定会使用到它。

Spring的数据绑定和较低级别的`BeanWrapper`都会使用`PropertyEditors`来进行转换和格式化。`PropertyEditor` 是JavaBeans规范的一部分，在这一章中我们将进行探讨。Spring3引入了"core.convert"这个包来提供通用 的类型转换工具和高级"format"包来格式化UI显示；这两个包提供的工具可以用作`PropertyEditors` 的替代品，我们也将在这章对它们展开讨论。

## 5.2 使用 Spring 的 Validator 接口来进行数据校验

Spring 提供了 `validator` 接口用来进行对象的数据校验。 `Validator` 接口在进行数据校验的时候会要求传入一个 `Errors` 对象，当有错误产生时会将错误信息放入该 `Errors` 对象。

我们假设有这么一个数据对象：

```
public class Person {

    private String name;
    private int age;

    // 省略getters和setters...
}
```

为了给 `Person` 类提供校验行为我们可以通过实现 `org.springframework.validation.Validator` 这个接口的两个方法来实现：

- `supports(Class)` - 判断该 `Validator` 是否能校验提供的 `class` 的实例？
- `validate(Object, org.springframework.validation.Errors)` - 校验给定的对象，如果有校验失败信息，将其放入 `Errors` 对象

实现一个校验器是相当简单的，尤其是当你知道 `spring` 已经提供了一个 `ValidationUtils` 工具类时。

```
public class PersonValidator implements Validator {

    /**
     * 这个校验器*仅仅*只校验Person实例
     */
    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");
        }
    }
}
```

如同你看到的，`ValidationUtils` 中的静态方法 `rejectIfEmpty(..)` 用来拒绝 'name' 这个属性当它为 `null` 或空字符串时。你可以看看 `ValidationUtils` 的 javadocs，提前了解下除了例子中展示的功能外还有哪些好用的方法。

当校验一个复杂的对象时，自定义一个校验器类（封装嵌套对象的校验器类）比把校验逻辑分散到各个嵌套对象会更方便管理。比如：现在有一个 `Customer` 复杂对象，它有两个 `String` 类型的属性 (`first and second name`)，以及一个 `Address` 对象；这个 `Address` 对象和 `Customer` 对象是毫无关系的，它还实现了 `AddressValidator` 这样一个校验器。如果你想在 `Customer` 校验器类中重用 `Address` 校验器的功能（这种重用不是通过简单的代码拷贝），你可以将 `Address` 校验器的实例通过依赖注入的方式注入到 `Customer` 校验器中。像下面所描述的这样：

```
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;  
    }  
  
    /**  
     * 这个校验器校验Customer实例，同时也会校验Customer的子类实例  
     */  
    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());  
        } finally {  
            errors.popNestedPath();  
        }  
    }  
}
```

校验错误都会向作为参数传入的 `Errors` 对象进行报告。如果你使用的是 Spring Web MVC，你可以使用 `<spring:bind/>` 标签来提取校验错误信息，当然你也可以通过自己的方式来提取错误信息，这些方式可以通过阅读 javadocs 来获取更多的帮助。

## 5.3 通过错误编码得到错误信息

前面我们谈到数据绑定和数据校验。如何拿到校验错误信息是我们最后需要讨论的一个问题。在上面的例子中，我们拒绝了 `name` 和 `age` 属性。如果我们想要输出校验错误的提示信息，就要用到校验失败时设置的错误编码(本例中就是 `name` 和 `age` )。当你调用 `Errors` 接口中的 `rejectValue` 方法或者它的任何一个方法，它的优先实现不仅仅会注册作为参数传入进来的错误编码，还会注册一些遵循一定规则的错误编码。注册哪些规则的错误编码取决于你使用的 `MessageCodesResolver` 。当我们使用默认的 `DefaultMessageCodesResolver` 时，除了会将错误信息注册到你指定的错误编码上之外，这些错误信息还会注册到包含属性名的错误编码上。假如你调用这样一个方法 `rejectValue("age", "too.darn.old")` ，Spring除了会注册 `too.darn.old` 这个错误编码外，还会注册 `too.darn.old.age` 和 `too.darn.old.age.int` 这两个错误编码（即一个是包含属性名，另外一个既包含属性名还包含类型）；这在Spring中作为一种约定，这样所有的开发者都能按照这种约定来定位错误信息了。

想要获取更多有关 `MessageCodesResolver` 和默认的策略，可以通过下面的在线文档获取：

[MessageCodesResolver](#) [DefaultMessageCodesResolver](#) ,

## 5.4 Bean操作和BeanWrapper

`org.springframework.beans` 包遵循Oracle提供的JavaBeans标准。一个JavaBean只是一个包含默认无参构造器的类，它遵循一个命名约定(通过一个例子)：一个名为 `bingoMadness` 属性将有一个设置方法 `setBingoMadness(..)` 和一个获取方法 `getBingoMadness(..)`。有关JavaBeans和其规范的更多信息，请参考Oracle的网站([javabeans](#))。

`beans`包里一个非常重要的类是 `BeanWrapper` 接口和它的相应实现(`BeanWrapperImpl`)。引用自java文档，`BeanWrapper` 提供了设置和获取属性值(单独或批量)、获取属性描述符以及查询属性以确定它们是可读还是可写的功能。`BeanWrapper` 还提供对嵌套属性的支持，能够不受嵌套深度的限制启用子属性的属性设置。然后，`BeanWrapper` 提供了无需目标类代码的支持就能够添加标准JavaBeans的 `PropertyChangeListeners` 和 `VetoableChangeListeners` 的能力。最后然而并非最不重要的是，`BeanWrapper` 提供了对索引属性设置的支持。`BeanWrapper` 通常不会被应用程序的代码直接使用，而是由 `DataBinder` 和 `BeanFactory` 使用。

`BeanWrapper` 的名字已经部分暗示了它的工作方式：它包装一个bean以对其执行操作，比如设置和获取属性。

## 5.4.1 设置并获取基本和嵌套属性

使用 `setProperty(s)` 和 `getProperty(s)` 可以设置并获取属性，两者都带有几个重载方法。在 Spring 自带的 Java 文档中对它们有更详细的描述。重要的是要知道对象属性指示的几个约定。几个例子：

表 5.1. 属性示例

表达式	说明
<code>name</code>	表示属性 <code>name</code> 与方法 <code>getName()</code> 或 <code>isName()</code> 和 <code>setName()</code> 相对应
<code>account.name</code>	表示属性 <code>account</code> 的嵌套属性 <code>name</code> 与方法 <code>getAccount().setName()</code> 或 <code>getAccount().getName()</code> 相对应
<code>account[2]</code>	表示索引属性 <code>account</code> 的第三个元素。索引属性可以是 <code>array</code> 、 <code>list</code> 或其他自然排序的集合
<code>account[COMPANYNAME]</code>	表示映射属性 <code>account</code> 被键 <code>COMPANYNAME</code> 索引到的映射项的值

下面你会发现一些使用 `BeanWrapper` 来获取和设置属性的例子。

(如果你不打算直接使用 `BeanWrapper`，那么下一部分对你来说并不重要。如果你仅使用 `DataBinder` 和 `BeanFactory` 以及它们开箱即用的实现，你应该跳到关于 `PropertyEditor` 部分的开头)。

考虑下面两个类：

```

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;
    }
}

```

以下的代码片段展示了如何检索和操纵实例化的 Companies 和 Employees 的某些属性：

```
BeanWrapper company = new 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 = new 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");
```

## 5.4.2 Built-in PropertyEditor implementations

Spring uses the concept of `PropertyEditors` to effect the conversion between an `Object` and a `String`. If you think about it, it sometimes might be handy to be able to represent properties in a different way than the object itself. For example, a `Date` can be represented in a human readable way (as the `String '2007-14-09'`), while we're still able to convert the human readable form back to the original date (or even better: convert any date entered in a human readable form, back to `Date` objects). This behavior can be achieved by *registering custom editors*, of type `java.beans.PropertyEditor`. Registering custom editors on a `BeanWrapper` or alternately in a specific IoC container as mentioned in the previous chapter, gives it the knowledge of how to convert properties to the desired type. Read more about `PropertyEditors` in the javadocs of the `java.beans` package provided by Oracle.

A couple of examples where property editing is used in Spring:

- *setting properties on beans* is done using `PropertyEditors`. When mentioning `java.lang.String` as the value of a property of some bean you're declaring in XML file, Spring will (if the setter of the corresponding property has a `Class`-parameter) use the `ClassEditor` to try to resolve the parameter to a `Class` object.
- *parsing HTTP request parameters* in Spring's MVC framework is done using all kinds of `PropertyEditors` that you can manually bind in all subclasses of the `CommandController`.

Spring has a number of built-in `PropertyEditors` to make life easy. Each of those is listed below and they are all located in the `org.springframework.beans.propertyeditors` package. Most, but not all (as indicated below), are registered by default by `BeanWrapperImpl`. Where the property editor is configurable in some fashion, you can of course still register your own variant to override the default one:

**Table 5.2. Built-in PropertyEditors**

Class	Explanation
<code>ByteArrayPropertyEditor</code>	Editor for byte arrays. Strings will simply be converted to their corresponding byte representations. Registered by default by <code>BeanWrapperImpl</code> .
<code>ClassEditor</code>	Parses Strings representing classes to actual classes and the other way around. When a class is not found, an <code>IllegalArgumentException</code> is thrown. Registered by default by <code>BeanWrapperImpl</code> .
<code>CustomBooleanEditor</code>	Customizable property editor for <code>Boolean</code> properties. Registered by default by <code>BeanWrapperImpl</code> , but, can be overridden by registering custom instance of it as custom editor.

CustomCollectionEditor	Property editor for Collections, converting any source Collection to a given target Collection type.
CustomDateEditor	Customizable property editor for java.util.Date, supporting a custom DateFormat. NOT registered by default. Must be user registered as needed with appropriate format.
CustomNumberEditor	Customizable property editor for any Number subclass like Integer , Long , Float , Double . Registered by default by BeanWrapperImpl , but can be overridden by registering custom instance of it as a custom editor.
FileEditor	Capable of resolving Strings to java.io.File objects. Registered by default by BeanWrapperImpl .
InputStreamEditor	One-way property editor, capable of taking a text string and producing (via an intermediate ResourceEditor and Resource ) an InputStream , so InputStream properties may be directly set as Strings. Note that the default usage will not close the InputStream for you! Registered by default by BeanWrapperImpl .
LocaleEditor	Capable of resolving Strings to Locale objects and vice versa (the String format is [country][variant], which is the same thing the toString() method of Locale provides). Registered by default by BeanWrapperImpl .
PatternEditor	Capable of resolving Strings to java.util.regex.Pattern objects and vice versa.
PropertiesEditor	Capable of converting Strings (formatted using the format as defined in the javadocs of the java.util.Properties class) to Properties objects. Registered by default by BeanWrapperImpl .
StringTrimmerEditor	Property editor that trims Strings. Optionally allows transforming an empty string into a null value. NOT registered by default; must be user registered as needed.
URLEditor	Capable of resolving a String representation of a URL to an actual URL object. Registered by default by BeanWrapperImpl .

Spring uses the `java.beans.PropertyEditorManager` to set the search path for property editors that might be needed. The search path also includes `sun.bean.editors` , which includes `PropertyEditor` implementations for types such as `Font` , `Color` , and most of the primitive types. Note also that the standard JavaBeans infrastructure will automatically discover `PropertyEditor` classes (without you having to register them explicitly) if they are in the same package as the class they handle, and have the same name as that class, with '`Editor`' appended; for example, one could have the following class and package structure, which would be sufficient for the `FooEditor` class to be recognized and used as the `PropertyEditor` for `Foo`-typed properties.

```

com
chank
pop
    Foo
        FooEditor // the PropertyEditor for the Foo class

```

Note that you can also use the standard `BeanInfo` JavaBeans mechanism here as well (described [in not-amazing-detail here](#)). Find below an example of using the `BeanInfo` mechanism for explicitly registering one or more `PropertyEditor` instances with the properties of an associated class.

```

com
chank
pop
    Foo
        FooBeanInfo // the BeanInfo for the Foo class

```

Here is the Java source code for the referenced `FooBeanInfo` class. This would associate a `CustomNumberEditor` with the `age` property of the `Foo` class.

```

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());
        }
    }
}

```

## Registering additional custom PropertyEditors

When setting bean properties as a string value, a Spring IoC container ultimately uses standard JavaBeans `PropertyEditors` to convert these Strings to the complex type of the property. Spring pre-registers a number of custom `PropertyEditors` (for example, to convert

a classname expressed as a string into a real `Class` object). Additionally, Java's standard JavaBeans `PropertyEditor` lookup mechanism allows a `PropertyEditor` for a class simply to be named appropriately and placed in the same package as the class it provides support for, to be found automatically.

If there is a need to register other custom `PropertyEditors`, there are several mechanisms available. The most manual approach, which is not normally convenient or recommended, is to simply use the `registerCustomEditor()` method of the `ConfigurableBeanFactory` interface, assuming you have a `BeanFactory` reference. Another, slightly more convenient, mechanism is to use a special bean factory post-processor called `CustomEditorConfigurer`. Although bean factory post-processors can be used with `BeanFactory` implementations, the `CustomEditorConfigurer` has a nested property setup, so it is strongly recommended that it is used with the `ApplicationContext`, where it may be deployed in similar fashion to any other bean, and automatically detected and applied.

Note that all bean factories and application contexts automatically use a number of built-in property editors, through their use of something called a `BeanWrapper` to handle property conversions. The standard property editors that the `BeanWrapper` registers are listed in [the previous section](#). Additionally, `ApplicationContexts` also override or add an additional number of editors to handle resource lookups in a manner appropriate to the specific application context type.

Standard JavaBeans `PropertyEditor` instances are used to convert property values expressed as strings to the actual complex type of the property. `CustomEditorConfigurer`, a bean factory post-processor, may be used to conveniently add support for additional `PropertyEditor` instances to an `ApplicationContext`.

Consider a user class `ExoticType`, and another class `DependsOnExoticType` which needs `ExoticType` set as a property:

```

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;
    }
}

```

When things are properly set up, we want to be able to assign the type property as a string, which a `PropertyEditor` will behind the scenes convert into an actual `ExoticType` instance:

```

<bean id="sample" class="example.DependsOnExoticType">
    <property name="type" value="aNameForExoticType"/>
</bean>

```

The `PropertyEditor` implementation could look similar to this:

```

// converts string representation to ExoticType object
package example;

public class ExoticTypeEditor extends PropertyEditorSupport {

    public void setAsText(String text) {
        setValue(new ExoticType(text.toUpperCase()));
    }
}

```

Finally, we use `CustomEditorConfigurer` to register the new `PropertyEditor` with the `ApplicationContext`, which will then be able to use it as needed:

```

<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="customEditors">
        <map>
            <entry key="example.ExoticType" value="example.ExoticTypeEditor"/>
        </map>
    </property>
</bean>

```

## Using PropertyEditorRegistrars

Another mechanism for registering property editors with the Spring container is to create and use a `PropertyEditorRegistrar`. This interface is particularly useful when you need to use the same set of property editors in several different situations: write a corresponding registrar and reuse that in each case. `PropertyEditorRegistrars` work in conjunction with an interface called `PropertyEditorRegistry`, an interface that is implemented by the Spring `BeanWrapper` (and `DataBinder`). `PropertyEditorRegistrars` are particularly convenient when used in conjunction with the `CustomEditorConfigurer` (introduced [here](#)), which exposes a property called `setPropertyEditorRegistrars(..)`: `PropertyEditorRegistrars` added to a `CustomEditorConfigurer` in this fashion can easily be shared with `DataBinder` and Spring MVC Controllers. Furthermore, it avoids the need for synchronization on custom editors: a `PropertyEditorRegistrar` is expected to create fresh `PropertyEditor` instances for each bean creation attempt.

Using a `PropertyEditorRegistrar` is perhaps best illustrated with an example. First off, you need to create your own `PropertyEditorRegistrar` implementation:

```

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...
    }
}

```

See also the `org.springframework.beans.support.ResourceEditorRegistrar` for an example `PropertyEditorRegistrar` implementation. Notice how in its implementation of the `registerCustomEditors(..)` method it creates new instances of each property editor.

Next we configure a `CustomEditorConfigurer` and inject an instance of our `CustomPropertyEditorRegistrar` into it:

```

<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"/>

```

Finally, and in a bit of a departure from the focus of this chapter, for those of you using [Spring's MVC web framework](#), using `PropertyEditorRegistrars` in conjunction with data-binding `Controllers` (such as `SimpleFormController`) can be very convenient. Find below an example of using a `PropertyEditorRegistrar` in the implementation of an `initBinder(..)` method:

```

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
}

```

This style of `PropertyEditor` registration can lead to concise code (the implementation of `initBinder(..)` is just one line long!), and allows common `PropertyEditor` registration code to be encapsulated in a class and then shared amongst as many `Controllers` as needed.

## 5.5 Spring Type Conversion

Spring 3 introduces a `core.convert` package that provides a general type conversion system. The system defines an SPI to implement type conversion logic, as well as an API to execute type conversions at runtime. Within a Spring container, this system can be used as an alternative to PropertyEditors to convert externalized bean property value strings to required property types. The public API may also be used anywhere in your application where type conversion is needed.

### 5.5.1 Converter SPI

The SPI to implement type conversion logic is simple and strongly typed:

```
package org.springframework.core.convert.converter;

public interface Converter<S, T> {

    T convert(S source);

}
```

To create your own converter, simply implement the interface above. Parameterize `s` as the type you are converting from, and `T` as the type you are converting to. Such a converter can also be applied transparently if a collection or array of `s` needs to be converted to an array or collection of `T`, provided that a delegating array/collection converter has been registered as well (which `DefaultConversionService` does by default).

For each call to `convert(s)`, the source argument is guaranteed to be NOT null. Your Converter may throw any unchecked exception if conversion fails; specifically, an `IllegalArgumentException` should be thrown to report an invalid source value. Take care to ensure that your `Converter` implementation is thread-safe.

Several converter implementations are provided in the `core.convert.support` package as a convenience. These include converters from Strings to Numbers and other common types. Consider `StringToInteger` as an example for a typical `Converter` implementation:

```
package org.springframework.core.convert.support;

final class StringToInteger implements Converter<String, Integer> {

    public Integer convert(String source) {
        return Integer.valueOf(source);
    }

}
```

## 5.5.2ConverterFactory

When you need to centralize the conversion logic for an entire class hierarchy, for example, when converting from String to java.lang.Enum objects, implement `ConverterFactory` :

```
package org.springframework.core.convert.converter;

public interfaceConverterFactory<S, R> {

    <T extends R> Converter<S, T> getConverter(Class<T> targetType);

}
```

Parameterize S to be the type you are converting from and R to be the base type defining the *range* of classes you can convert to. Then implement `getConverter(Class)`, where T is a subclass of R.

Consider the `stringToEnum` `ConverterFactory` as an example:

```

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());
        }
    }
}

```

### 5.5.3 GenericConverter

When you require a sophisticated Converter implementation, consider the GenericConverter interface. With a more flexible but less strongly typed signature, a GenericConverter supports converting between multiple source and target types. In addition, a GenericConverter makes available source and target field context you can use when implementing your conversion logic. Such context allows a type conversion to be driven by a field annotation, or generic information declared on a field signature.

```

package org.springframework.core.convert.converter;

public interface GenericConverter {

    public Set<ConvertiblePair> getConvertibleTypes();

    Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType)
;
}

```

To implement a GenericConverter, have getConvertibleTypes() return the supported source→target type pairs. Then implement convert(Object, TypeDescriptor, TypeDescriptor) to implement your conversion logic. The source TypeDescriptor provides access to the

source field holding the value being converted. The target TypeDescriptor provides access to the target field where the converted value will be set.

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.



Because GenericConverter is a more complex SPI interface, only use it when you need it. Favor Converter orConverterFactory 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 a `static valueOf` method, is defined on the target class. `ConditionalGenericConverter` is the union of the `GenericConverter` and `ConditionalConverter` interfaces that allows you to define such custom matching criteria:

```
public interface ConditionalGenericConverter
    extends GenericConverter, ConditionalConverter {

    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)`.

### 5.5.4 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.

## 5.5.5 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.

If no `ConversionService` is registered with Spring, the original `PropertyEditor`-based system is used.

To register a default `ConversionService` with Spring, add the following bean definition with id `conversionService`:

```

<bean id="conversionService"
      class="org.springframework.context.support.ConversionServiceFactoryBean"/>

```

A default `ConversionService` can convert between strings, numbers, enums, collections, maps, and other common types. To supplement or override the default converters with your own custom converter(s), set the `converters` property. Property values may implement either of the `Converter`, `ConverterFactory`, or `GenericConverter` interfaces.

```
<bean id="conversionService"
      class="org.springframework.context.support.ConversionServiceFactoryBean">
    <property name="converters">
      <set>
        <bean class="example.MyCustomConverter"/>
      </set>
    </property>
</bean>
```

It is also common to use a `ConversionService` within a Spring MVC application. See [Section 18.16.3, “Conversion and Formatting”](#) in the Spring MVC chapter.

In certain situations you may wish to apply formatting during conversion. See [Section 5.6.3, “FormatterRegistry SPI”](#) for details on using `FormattingConversionServiceFactoryBean`.

## 5.5.6 Using a `ConversionService` programmatically

To work with a `ConversionService` instance programmatically, simply inject a reference to it like you would for any other bean:

```
@Service
public class MyService {

    @Autowired
    public MyService(ConversionService conversionService) {
        this.conversionService = conversionService;
    }

    public void doit() {
        this.conversionService.convert(...);
    }
}
```

For most use cases, the `convert` method specifying the `targetType` can be used but it will not work with more complex types such as a collection of a parameterized element. If you want to convert a `List` of `Integer` to a `List` of `String` programmatically, for instance, you need to provide a formal definition of the source and target types.

Fortunately, `TypeDescriptor` provides various options to make that straightforward:

```
DefaultConversionService cs = new DefaultConversionService();

List<Integer> input = ....
cs.convert(input,
    TypeDescriptor.forObject(input), // List<Integer> type descriptor
    TypeDescriptor.collection(List.class, TypeDescriptor.valueOf(String.class)));
```

Note that `DefaultConversionService` registers converters automatically which are appropriate for most environments. This includes collection converters, scalar converters, and also basic `Object` to `String` converters. The same converters can be registered with any `ConverterRegistry` using the `static addDefaultConverters` method on the `DefaultConversionService` class.

Converters for value types will be reused for arrays and collections, so there is no need to create a specific converter to convert from a `Collection` of `s` to a `Collection` of `T`, assuming that standard collection handling is appropriate.

## 5.6 Spring Field Formatting

As discussed in the previous section, `core.convert` is a general-purpose type conversion system. It provides a unified `ConversionService` API as well as a strongly-typed `Converter` SPI for implementing conversion logic from one type to another. A Spring Container uses this system to bind bean property values. In addition, both the Spring Expression Language (SpEL) and DataBinder use this system to bind field values. For example, when SpEL needs to coerce a `Short` to a `Long` to complete an `expression.setValue(Object bean, Object value)` attempt, the `core.convert` system performs the coercion.

Now consider the type conversion requirements of a typical client environment such as a web or desktop application. In such environments, you typically convert *from String* to support the client postback process, as well as back *to String* to support the view rendering process. In addition, you often need to localize String values. The more general `core.convert` Converter SPI does not address such *formatting* requirements directly. To directly address them, Spring 3 introduces a convenient `Formatter` SPI that provides a simple and robust alternative to `PropertyEditors` for client environments.

In general, use the `Converter` SPI when you need to implement general-purpose type conversion logic; for example, for converting between a `java.util.Date` and a `java.lang.Long`. Use the `Formatter` SPI when you're working in a client environment, such as a web application, and need to parse and print localized field values. The `ConversionService` provides a unified type conversion API for both SPIs.

### 5.6.1 Formatter SPI

The `Formatter` SPI to implement field formatting logic is simple and strongly typed:

```
package org.springframework.format;

public interface Formatter<T> extends Printer<T>, Parser<T> {
```

Where `Formatter` extends from the `Printer` and `Parser` building-block interfaces:

```
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;
}
```

To create your own Formatter, simply implement the Formatter interface above.

Parameterize T to be the type of object you wish to format, for example, `java.util.Date`.

Implement the `print()` operation to print an instance of T for display in the client locale.

Implement the `parse()` operation to parse an instance of T from the formatted

representation returned from the client locale. Your Formatter should throw a

`ParseException` or `IllegalArgumentException` if a parse attempt fails. Take care to ensure  
your Formatter implementation is thread-safe.

Several Formatter implementations are provided in `format` subpackages as a convenience.

The `number` package provides a `NumberFormatter`, `CurrencyFormatter`, and

`PercentFormatter` to format `java.lang.Number` objects using a `java.text.NumberFormat`.

The `datetime` package provides a `DateFormatter` to format `java.util.Date` objects with a

`java.text.DateFormat`. The `datetime.joda` package provides comprehensive datetime

formatting support based on the [Joda Time library](#).

Consider `DateFormatter` as an example `Formatter` implementation:

```
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;
    }

}
```

The Spring team welcomes community-driven `Formatter` contributions; see [jira.spring.io](https://jira.spring.io) to contribute.

## 5.6.2 Annotation-driven Formatting

As you will see, field formatting can be configured by field type or annotation. To bind an Annotation to a formatter, implement `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);

}
```

Parameterize A to be the field annotationType you wish to associate formatting logic with, for example `org.springframework.format.annotation.DateTimeFormat`. Have `getFieldTypes()` return the types of fields the annotation may be used on. Have `getPrinter()` return a Printer to print the value of an annotated field. Have `getParser()` return a Parser to parse a clientValue for an annotated field.

The example AnnotationFormatterFactory implementation below binds the `@NumberFormat` Annotation to a formatter. This annotation allows either a number style or pattern to be specified:

```

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();
            }
        }
    }
}

```

To trigger formatting, simply annotate fields with `@NumberFormat`:

```

public class MyModel {

    @NumberFormat(style=Style.CURRENCY)
    private BigDecimal decimal;

}

```

## Format Annotation API

A portable format annotation API exists in the `org.springframework.format.annotation` package. Use `@NumberFormat` to format `java.lang.Number` fields. Use `@DateTimeFormat` to format `java.util.Date`, `java.util.Calendar`, `java.util.Long`, or Joda Time fields.

The example below uses `@DateTimeFormat` to format a `java.util.Date` as a ISO Date (yyyy-MM-dd):

```
public class MyModel {  
  
    @DateTimeFormat(iso=ISO.DATE)  
    private Date date;  
  
}
```

### 5.6.3 FormatterRegistry SPI

The `FormatterRegistry` is an SPI for registering formatters and converters.

`FormattingConversionService` is an implementation of `FormatterRegistry` suitable for most environments. This implementation may be configured programmatically or declaratively as a Spring bean using `FormattingConversionServiceFactoryBean`. Because this implementation also implements `ConversionService`, it can be directly configured for use with Spring's `DataBinder` and the Spring Expression Language (SpEL).

Review the `FormatterRegistry` SPI below:

```
package org.springframework.format;  
  
public interface FormatterRegistry extends ConverterRegistry {  
  
    void addFormatterForFieldType(Class<?> fieldType, Printer<?> printer, Parser<?> pa  
rser);  
  
    void addFormatterForFieldType(Class<?> fieldType, Formatter<?> formatter);  
  
    void addFormatterForFieldType(Formatter<?> formatter);  
  
    void addFormatterForAnnotation(AnnotationFormatterFactory<?, ?> factory);  
  
}
```

As shown above, Formatters can be registered by `fieldType` or annotation.

The `FormatterRegistry` SPI allows you to configure Formatting rules centrally, instead of duplicating such configuration across your Controllers. For example, you might want to enforce that all Date fields are formatted a certain way, or fields with a specific annotation are formatted in a certain way. With a shared `FormatterRegistry`, you define these rules once and they are applied whenever formatting is needed.

### 5.6.4 FormatterRegistrar SPI

The FormatterRegistrar is an SPI for registering formatters and converters through the FormatterRegistry:

```
package org.springframework.format;

public interface FormatterRegistrar {

    void registerFormatters(FormatterRegistry registry);

}
```

A FormatterRegistrar is useful when registering multiple related converters and formatters for a given formatting category, such as Date formatting. It can also be useful where declarative registration is insufficient. For example when a formatter needs to be indexed under a specific field type different from its own or when registering a Printer/Parser pair. The next section provides more information on converter and formatter registration.

## 5.6.5 Configuring Formatting in Spring MVC

See [Section 18.16.3, “Conversion and Formatting”](#) in the Spring MVC chapter.

## 5.7 Configuring a global date & time format

By default, date and time fields that are not annotated with `@DateTimeFormat` are converted from strings using the `DateFormat.SHORT` style. If you prefer, you can change this by defining your own global format.

You will need to ensure that Spring does not register default formatters, and instead you should register all formatters manually. Use

the `org.springframework.format.datetime.joda.JodaTimeFormatterRegistrar` or `org.springframework.format.datetime.DateFormatterRegistrar` class depending on whether you use the Joda Time library.

For example, the following Java configuration will register a global ' `yyyyMMdd`' format. This example does not depend on the Joda Time library:

```
@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;
    }
}
```

If you prefer XML based configuration you can use a

`FormattingConversionServiceFactoryBean`. Here is the same example, this time using Joda Time:

```

<?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.FormattingC
onversionServiceFactoryBean">
        <property name="registerDefaultFormatters" value="false" />
        <property name="formatters">
            <set>
                <bean class="org.springframework.format.number.NumberFormatAnnotationF
ormatterFactory" />
                </set>
            </property>
            <property name="formatterRegistrars">
                <set>
                    <bean class="org.springframework.format.datetime.joda.JodaTimeFormatte
rRegistrar">
                        <property name="dateFormatter">
                            <bean class="org.springframework.format.datetime.joda.DateTime
FormatterFactoryBean">
                                <property name="pattern" value="yyyyMMdd"/>
                            </bean>
                        </property>
                    </bean>
                </set>
            </property>
        </bean>
    </beans>

```



Joda Time provides separate distinct types to represent `date`, `time` and `date-time` values. The `dateFormatter`, `timeFormatter` and `dateTimeFormatter` properties of the `JodaTimeFormatterRegistrar` should be used to configure the different formats for each type. The `DateTimeFormatterFactoryBean` provides a convenient way to create formatters.

If you are using Spring MVC remember to explicitly configure the conversion service that is used. For Java based `@Configuration` this means extending the `WebMvcConfigurationSupport` class and overriding the `mvcConversionService()` method. For XML you should use the '`conversion-service`' attribute of the `mvc:annotation-driven` element. See [Section 18.16.3, “Conversion and Formatting”](#) for details.

## 5.8 Spring Validation

Spring 3 introduces several enhancements to its validation support. First, the JSR-303 Bean Validation API is now fully supported. Second, when used programmatically, Spring's DataBinder can now validate objects as well as bind to them. Third, Spring MVC now has support for declaratively validating `@Controller` inputs.

### 5.8.1 Overview of the JSR-303 Bean Validation API

JSR-303 standardizes validation constraint declaration and metadata for the Java platform. Using this API, you annotate domain model properties with declarative validation constraints and the runtime enforces them. There are a number of built-in constraints you can take advantage of. You may also define your own custom constraints.

To illustrate, consider a simple PersonForm model with two properties:

```
public class PersonForm {  
    private String name;  
    private int age;  
}
```

JSR-303 allows you to define declarative validation constraints against such properties:

```
public class PersonForm {  
  
    @NotNull  
    @Size(max=64)  
    private String name;  
  
    @Min(0)  
    private int age;  
}
```

When an instance of this class is validated by a JSR-303 Validator, these constraints will be enforced.

For general information on JSR-303/JSR-349, see the [Bean Validation website](#). For information on the specific capabilities of the default reference implementation, see the [Hibernate Validator](#) documentation. To learn how to setup a Bean Validation provider as a Spring bean, keep reading.

## 5.8.2 Configuring a Bean Validation Provider

Spring provides full support for the Bean Validation API. This includes convenient support for bootstrapping a JSR-303/JSR-349 Bean Validation provider as a Spring bean. This allows for a `javax.validation.ValidatorFactory` or `javax.validation.Validator` to be injected wherever validation is needed in your application.

Use the `LocalValidatorFactoryBean` to configure a default Validator as a Spring bean:

```
<bean id="validator"
      class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"/>
```

The basic configuration above will trigger Bean Validation to initialize using its default bootstrap mechanism. A JSR-303/JSR-349 provider, such as Hibernate Validator, is expected to be present in the classpath and will be detected automatically.

## Injecting a Validator

`LocalValidatorFactoryBean` implements both `javax.validation.ValidatorFactory` and `javax.validation.Validator`, as well as Spring's `org.springframework.validation.Validator`. You may inject a reference to either of these interfaces into beans that need to invoke validation logic.

Inject a reference to `javax.validation.Validator` if you prefer to work with the Bean Validation API directly:

```
import javax.validation.Validator;

@Service
public class MyService {

    @Autowired
    private Validator validator;
```

Inject a reference to `org.springframework.validation.Validator` if your bean requires the Spring Validation API:

```

import org.springframework.validation.Validator;

@Service
public class MyService {

    @Autowired
    private Validator validator;

}

```

## Configuring Custom Constraints

Each Bean Validation constraint consists of two parts. First, a `@Constraint` annotation that declares the constraint and its configurable properties. Second, an implementation of the `javax.validation.ConstraintValidator` interface that implements the constraint's behavior. To associate a declaration with an implementation, each `@Constraint` annotation references a corresponding `ValidationConstraint` implementation class. At runtime, a `ConstraintValidatorFactory` instantiates the referenced implementation when the constraint annotation is encountered in your domain model.

By default, the `LocalValidatorFactoryBean` configures a `SpringConstraintValidatorFactory` that uses Spring to create `ConstraintValidator` instances. This allows your custom `ConstraintValidators` to benefit from dependency injection like any other Spring bean.

Shown below is an example of a custom `@Constraint` declaration, followed by an associated `ConstraintValidator` implementation that uses Spring for dependency injection:

```

@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;

    ...
}

```

As you can see, a ConstraintValidator implementation may have its dependencies @Autowired like any other Spring bean.

## Spring-driven Method Validation

The method validation feature supported by Bean Validation 1.1, and as a custom extension also by Hibernate Validator 4.3, can be integrated into a Spring context through a `MethodValidationPostProcessor` bean definition:

```
<bean class="org.springframework.validation.beanvalidation.MethodValidationPostProcess  
or"/>
```

In order to be eligible for Spring-driven method validation, all target classes need to be annotated with Spring's `@Validated` annotation, optionally declaring the validation groups to use. Check out the `MethodValidationPostProcessor` javadocs for setup details with Hibernate Validator and Bean Validation 1.1 providers.

## Additional Configuration Options

The default `LocalValidatorFactoryBean` configuration should prove sufficient for most cases. There are a number of configuration options for various Bean Validation constructs, from message interpolation to traversal resolution. See the `LocalValidatorFactoryBean` javadocs for more information on these options.

### 5.8.3 Configuring a DataBinder

Since Spring 3, a DataBinder instance can be configured with a Validator. Once configured, the Validator may be invoked by calling `binder.validate()`. Any validation Errors are automatically added to the binder's BindingResult.

When working with the DataBinder programmatically, this can be used to invoke validation logic after binding to a target object:

```
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();
```

A DataBinder can also be configured with multiple `validator` instances via

`dataBinder.addValidators` and `dataBinder.replaceValidators`. This is useful when combining globally configured Bean Validation with a Spring `validator` configured locally on a DataBinder instance. See [???](#).

## 5.8.4 Spring MVC 3 Validation

See Section 18.16.4, “Validation” in the Spring MVC chapter.



## 6.1 介绍

The Spring Expression Language（简称SpEL）是一种强大的支持在运行时查询和操作对象图的表达式语言。语言语法类似于Unified EL，但提供了额外的功能，特别是方法调用和基本字符串模板功能。

尽管还有其他几种Java表达式语言，比如OGNL，MVEL和JBoss EL，但SpEL只是为了向Spring社区提供一种支持良好的表达式语言，你可以在所有使用Spring框架的产品中使用SpEL。其语言特性是由使用Spring框架项目的需求所驱动的，包括Eclipse中基础Spring工具套件中的代码完成支持功能的工具要求。也就是说，SpEL基于一种抽象实现的技术API，允许在需要时集成其他表达式语言来实现。

虽然SpEL作为Spring产品组合中的表达式运算操作的基础，但它并不直接与Spring有关，可以独立使用。为了自包含，本章中的很多例子都使用SpEL，就像它是一种独立的表达语言。这就需要创建一些引导作用的基础实现类，如解析器。大多数Spring用户将不需要处理这种基础实现类，并且只会将表达式字符串作为运算操作。这个典型用途的一个例子是将Spel集成到创建XML或基于注释的bean定义中，如表达式支持定义bean的定义所示。

本章将介绍表达式语言的特点及其API及其语言语法。在好几个地方，使用Inventor和Inventor's Society类作为表达式运算操作的目标对象。这些类声明和用于填充它们的数据在本章末尾列出。

## 6.2 功能概述

表达式语言支持以下功能

- Literal expressions(文字表达)
- Boolean and relational operators(布尔和关系运算)
- Regular expressions(正则表达式)
- Class expressions(类表达式)
- Accessing(访问) properties, arrays, lists, maps
- Method invocation(方法调用)
- Relational operators(关系运算)
- Assignment(分配)
- Calling constructors(调用构造函数)
- Bean references(Bean引用)
- Array construction(数组构造)
- Inline lists(内联集合)
- Inline maps(内联映射)
- Ternary operator(三元操作)
- Variables(变量)
- User defined functions(用户定义的功能)
- Collection projection(集合投影)
- Collection selection(集合选择)
- Templated expressions(模板表达式)

## 6.3 使用Spring表达式接口的表达式运算操作

本节介绍了SpEL接口及其表达式语言的简单使用。完整的语言参考可以在“语言参考”一节中找到。

以下代码介绍了SpEL API来运算操作文字字符串表达式“Hello World”。

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("'Hello World'");
String message = (String) exp.getValue();
```

变量message的值只是 'Hello World'.

您最有可能使用的SpEL类和接口位于包 `org.springframework.expression` 及其子包和包 `spel.support`

接口 `ExpressionParser` 负责解析表达式字符串。在此示例中，表达式字符串是由周围的单引号表示的字符串文字。接口 `Expression` 负责运算操作先前定义的表达式字符串。当分别调用 `parser.parseExpression` 和 `exp.getValue` 时，有两个可以抛出的异常，`ParseException` 和 `EvaluationException`。

SpEL支持各种功能，如调用方法，访问属性和调用构造函数。

作为方法调用的一个例子，我们在字符串文字中调用 `concat` 方法。

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("'Hello World'.concat('!')");
String message = (String) exp.getValue();
```

变量message的值现在是 'Hello World!'.

作为调用JavaBean属性的示例，可以调用 `String` 属性 `getBytes`，如下所示。

```
ExpressionParser parser = new SpelExpressionParser();

// invokes 'getBytes()'
Expression exp = parser.parseExpression("'Hello World'.getBytes");
byte[] bytes = (byte[]) exp.getValue();
```

SpEL还支持嵌套属性，使用标准点符号，即`prop1.prop2.prop3`链式写法和属性值的设置

Public fields may also be accessed.

```
ExpressionParser parser = new SpelExpressionParser();

// invokes 'getBytes().length'
Expression exp = parser.parseExpression("'Hello World'.bytes.length");
int length = (Integer) exp.getValue();
```

可以调用 `String` 的构造函数，而不是使用字符串文字。

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("new String('hello world').toUpperCase()");
String message = exp.getValue(String.class);
```

注意使用通用方法 `public (T) T getValue (Class <T> desiredResultType)`。使用此方法不需要将表达式的值转换为所需的结果类型。如果该值不能转换为类型 `T` 或使用注册的类型转换器转换，则将抛出 `EvaluationException`。

`Spel` 的更常见的用法是提供一个针对特定对象实例（称为根对象）进行运算操作的表达式字符串。这里有两个选项，并且选择哪个由反对当前被验证的表达式的对象是否在每次调用后而改变再验证表达式来决定。在以下示例中，我们从 `Inventor` 类的实例中检索 `name` 属性。

```
// 创建并对日历对象设值
GregorianCalendar c = new GregorianCalendar();
c.set(1856, 7, 9);

// 构造函数的参数是姓名，生日和国籍。
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);
```

在最后一行，字符串变量 `name` 的值将被设置为“`Nikola Tesla`”。

`StandardEvaluationContext` 类可以指定哪个对象的“`name`”属性将被运算操作。如果根对象不太可能改变，这是使用的机制，可以在运算操作上下文中简单地设置一次。如果根对象可能会重复更改，则可以在每次调用 `getValue` 时提供该对象，如下例所示：

```
// 创建并对日历对象设值
GregorianCalendar c = new GregorianCalendar();
c.set(1856, 7, 9);

// 构造函数的参数是姓名，生日和国籍。
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，目标对象 `tesla` 已经直接提供给 `getValue`，表达式运算操作的基础动作即在内部创建和管理默认运算操作的上下文 - 它不需要被额外提供。

`StandardEvaluationContext` 构建起来代价相对昂贵，并在重复使用时构建高速缓存的状态，使得能够更快地执行后续的表达式运算操作。因此，最好在可能的情况下缓存和重新使用它们，而不是为每个表达式运算操作构建一个新的。

在某些情况下，可能需要使用配置的运算操作上下文，但在每次调用 `getValue` 时仍会提供不同的根对象。`getValue` 允许在同一个调用中指定两者。在这些情况下，传递给该调用的根对象被认为覆盖在任何（可能为`null`）指定的运算操作的上下文中。



在独立使用SpEL中，需要创建解析器，解析表达式，并可能提供运算操作上下文和根上下文对象。但是，更常见的用法是仅将SpEL表达式字符串作为配置文件的一部分，例如 Spring bean或Spring Web Flow定义。在这种情况下，解析器，运算操作上下文，根对象和任何预定义的变量都是隐式设置的，要求用户除表达式之外不要指定。

作为最后的介绍性示例，使用上一个示例中的`Inventor`对象来显示布尔运算符的使用。

```
Expression exp = parser.parseExpression("name == 'Nikola Tesla'");
boolean result = exp.getValue(context, Boolean.class); // evaluates to true
```

### 6.3.1 EvaluationContext 接口

运算表达式以解析属性，方法，字段并帮助执行类型转换时使用 `EvaluationContext` 接口。开箱即用的实现 `StandardEvaluationContext`，使用反射来操纵对象，并缓存 `java.lang.reflect.Method`，`java.lang.reflect.Field` 和 `java.lang.reflect.Constructor` 实例以提高性能。

`StandardEvaluationContext` 是您可以通过方法 `setRootObject()` 或将根对象传递到构造函数中来指定要对其进行运算操作的根对象。您还可以使用 `setVariable()` 和 `registerFunction()` 方法指定将在表达式中使用的变量和函数。变量

和函数的使用在语言参考部分变量和函数中有所描述。 `StandardEvaluationContext` 还可以注册自定义 `ConstructorResolvers`，`MethodResolvers` 和 `PropertyAccessors`，以扩展SpEL如何运算操作表达式。请参考这些类的JavaDoc了解更多详细信息。

## 类型转换

默认情况下，SpEL使用Spring核

心 (`org.springframework.core.convert.ConversionService`) 中提供的转换服务。此转换服务附带许多转换器，内置于常用转换，但也可完全扩展，因此可以添加类型之间的自定义转换。此外，它具有泛型感知的关键功能。这意味着在使用表达式中的泛型类型时，SpEL将尝试转换以维护遇到的任何对象的类型正确性。

这在实践中意味着什么？假设使用 `setValue()` 的赋值被用于设置 `List` 属性。属性的类型实际上是 `List <Boolean>`。Spel将会认识到列表的元素需要在被放置在其中之前被转换为布尔值。一个简单的例子：

```
class Simple {
    public List<Boolean> booleanList = new ArrayList<Boolean>();
}

Simple simple = new Simple();

simple.booleanList.add(true);

StandardEvaluationContext simpleContext = new StandardEvaluationContext(simple);

// false允许作为字符串， SpEL和转换服务将正确识别它需要是一个布尔值并对其进行转换
parser.parseExpression("booleanList[0]").setValue(simpleContext, "false");

// b为布尔值false
Boolean b = simple.booleanList.get(0);
```

### 6.3.2 解析器配置

可以使用解析器配置对象 (`org.springframework.expression.spel.SpelParserConfiguration`) 来配置SpEL表达式解析器。该配置对象控制一些表达式组件的行为。例如，如果索引到数组或集合中，并且指定索引处的元素为空，其将自动创建元素。当使用由一组属性引用组成的表达式时，这是非常有用的。如果索引到数组或列表中，并指定超出数组或列表的当前大小的结尾的索引，其将自动提高数组或列表大小以适应该索引。

```

class Demo {
    public List<String> list;
}

// 开启如下功能：
// - 空引用自动初始化
// - 集合大小自动增长
SpelParserConfiguration config = new SpelParserConfiguration(true,true);

ExpressionParser parser = new SpelExpressionParser(config);

Expression expression = parser.parseExpression("list[3]");

Demo demo = new Demo();

Object o = expression.getValue(demo);

// demo.list现在被实例化为拥有四个元素的集合
// 每个元素都是一个新的空字符串

```

还可以配置SpEL表达式编译器的行为。

### 6.3.3 SpEL编译

Spring Framework 4.1包含一个基本的表达式编译器。表达式通常因为在运算操作过程中提供了大量的动态灵活性被解释，但不能提供最佳性能。对于偶尔的表达式使用这是很好的，但是当被其他并不真正需要动态灵活性的组件（如Spring Integration）使用时，性能可能非常重要。

新的SpEL编译器旨在满足这一需求。编译器将在体现了表达行为的运算操作期间即时生成一个真正的Java类，并使用它来实现更快的表达式求值。由于缺少对表达式按类型归类，编译器在执行编译时会使用在表达式解释运算期间收集的信息来编译。例如，它不仅仅是从表达式中知道属性引用的类型，而是在第一个解释运算过程中会发现它是什么。当然，如果各种表达式元素的类型随着时间的推移而变化，那么基于此信息的编译可能会导致问题产生。因此，编译最适合于重复运算操作时类型信息不会改变的表达式。

对于这样的基本表达式：

```
someArray[0].someProperty.someOtherProperty < 0.1
```

这涉及到数组访问，某些属性的取消和数字操作，性能增益可以非常显着。在50000次迭代的微型基准运行示例中，只使用解释器需要75ms，而使用编译版本的表达式仅需3ms。

### 编译器配置

默认情况下，编译器未打开，但有两种方法可以打开它。可以使用先前讨论的解析器配置过程或者当将SpEL使用嵌入到另一个组件中时通过系统属性打开它。本节讨论这两个选项。

重要的是要明白，编译器可以运行几种模式，在枚举 (`org.springframework.expression.spel.SpelCompilerMode`) 中捕获。模式如下：

- `OFF` - 编译器关闭；这是默认值。
- `IMMEDIATE` - 在即时模式下，表达式将尽快编译。这通常是在第一次解释运算之后。如果编译的表达式失败（通常是由于类型更改，如上所述）引起的，则表达式运算操作的调用者将收到异常。
- `MIXED` - 在混合模式下，表达式随着时间的推移在解释模式和编译模式之间静默地切换。经过一些解释运行后，它们将切换到编译模式，如果编译后的表单出现问题（如上所述改变类型），表达式将自动重新切换回解释模式。稍后，它可能生成另一个编译表单并切换到它。基本上，用户进入即时模式的异常是内部处理的。

存在`IMMEDIATE`模式，因为混合模式可能会导致具有副作用的表达式的问题。如果一个编译的表达式在部分成功之后崩掉，它可能已经完成了影响系统状态的事情。如果发生这种情况，调用者可能不希望它在解释模式下静默地重新运行，因为表达式的一部分可能运行两次。

选择模式后，使用 `SpelParserConfiguration` 配置解析器：

```
SpelParserConfiguration config = new SpelParserConfiguration(SpelCompilerMode.IMMEDIATE,
    this.getClass().getClassLoader());

SpelExpressionParser parser = new SpelExpressionParser(config);

Expression expr = parser.parseExpression("payload");

MyMessage message = new MyMessage();

Object payload = expr.getValue(message);
```

当指定编译器模式时，也可以指定一个类加载器（允许传递`null`）。编译表达式将在任何提供的子类加载器中被定义。重要的是确保是否指定了类加载器，它可以看到表达式运算操作过程中涉及的所有类型。如果没有指定，那么将使用默认的类加载器（通常是在表达式计算期间运行的线程的上下文类加载器）。

配置编译器的第二种方法是将SpEL嵌入其他组件内部使用，并且可能无法通过配置对象进行配置。在这些情况下，可以使用系统属性。属性 `spring.expression.compiler.mode` 可以设置为 `SpelCompilerMode` 枚举值之一（关闭，即时或混合）。

## 编译器限制

虽然Spring Framework 4.1的基本编译框架已经存在，但是，框架还不支持编译各种表达式。最初的重点是在可能在性能要求高的关键环境中使用的常见表达式。这些表达方式目前无法编译：

- expressions involving assignment(涉及转让的表达)
- expressions relying on the conversion service(依赖转换服务的表达式)
- expressions using custom resolvers or accessors(使用自定义解析器或访问器的表达式)
- expressions using selection or projection(使用选择或投影的表达式)

越来越多的类型的表达式将在未来可编译。

## 6.4 具体bean定义的表达式支持

Spel表达式可以与XML或基于注释的配置元数据一起使用，用于定义 BeanDefinitions。在这两种情况下，定义表达式的语法格式为 `#{<expression string>}`。

### 6.4.1 基于XML的配置

可以使用如下所示的表达式设置属性或构造函数参数arg的值。

```
<bean id="numberGuess" class="org.springframework.samples.NumberGuess">
    <property name="randomNumber" value="#{ T(java.lang.Math).random() * 100.0 }"/>

    <!-- other properties -->
</bean>
```

变量 `systemProperties` 是预定义的，因此可以在您的表达式中使用它，如下所示。请注意，您不必在此上下文中使用 `#` 符号作为预定义变量的前缀。

```
<bean id="taxCalculator" class="org.springframework.samples.TaxCalculator">
    <property name="defaultLocale" value="#{ systemProperties['user.region'] }"/>

    <!-- other properties -->
</bean>
```

您也可以通过名称引用其他bean属性。

```
<bean id="numberGuess" class="org.springframework.samples.NumberGuess">
    <property name="randomNumber" value="#{ T(java.lang.Math).random() * 100.0 }"/>

    <!-- other properties -->
</bean>

<bean id="shapeGuess" class="org.springframework.samples.ShapeGuess">
    <property name="initialShapeSeed" value="#{ numberGuess.randomNumber }"/>

    <!-- other properties -->
</bean>
```

### 6.4.2 基于注释的配置

`@Value` 注释可以放置在字段，方法和方法/构造函数参数上以指定默认值。

这是一个设置字段变量默认值的示例。

```
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;
    }

}
```

等效的属性设置方法如下所示。

```
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;
    }

}
```

使用 `@Autowired` 注解的方法和构造函数也可以使用 `@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;
    }

    // ...
}
```

## 6.5 语言参考

### 6.5.1 字面常量表达式

支持的字面常量表达式的类型是字符串，数值（int，real，hex），boolean和null。字符串由单引号分隔。要将一个单引号本身放在字符串中，请使用两个单引号。

以下列表显示了字面常量的简单用法。通常，它们不会像这样使用，而是作为更复杂表达式的一部分，例如在逻辑比较运算符的一侧使用字面常量。

```
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();
```

数字支持使用负号，指数符号和小数点。默认情况下，使用Double.parseDouble（）解析实数。

### 6.5.2 Properties, Arrays, Lists, Maps, Indexers

使用属性引用进行导航很简单：只需使用句点来指示嵌套的属性值。

Inventor 类，pupin 和 tesla 的实例使用示例中使用的类中列出的数据进行填充。为了导航“down”，并获得Tesla的出生年份和Pupin的出生城市，使用以下表达式。

```
// evals to 1856
int year = (Integer) parser.parseExpression("Birthdate.Year + 1900").getValue(context);

String city = (String) parser.parseExpression("placeOfBirth.City").getValue(context);
```

属性名称的第一个字母不区分大小写。数组和列表的内容使用方括号表示法获得。

```

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);

```

map映射的内容是通过在括号内指定字面常量键值得到的。在这种情况下，因为Officers映射的键是字符串，我们可以指定字符串字面常量。

```

// 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");

```

### 6.5.3 内联列表

List列表可以使用 {} 表示法直接在表达式中表示。

```

// 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);

```

`{}` 本身就是一个空列表。出于性能原因，如果列表本身完全由固定字面常量组成，则会创建一个常量列表来表示表达式，而不是在每个运算操作上构建一个新列表。

## 6.5.4 内联映射

也可以使用 `{key : value}` 表示法直接在表达式中表示Map映射。

```
// evaluates to a Java map containing the two entries
Map inventorInfo = (Map) parser.parseExpression("{name:'Nikola',dob:'10-July-1856'}").getValue(context);

Map mapOfMaps = (Map) parser.parseExpression("{name:{first:'Nikola',last:'Tesla'},dob:{day:10,month:'July',year:1856}}").getValue(context);
```

`{:}` 本身就是一个空的Map映射。出于性能原因，如果Map映射本身由固定字面常量或其他嵌套常量结构（List列表或Map映射）组成，则会创建一个常量Map映射来表示表达式，而不是在每个运算操作上构建一个新的Map映射。引用Map映射键是可选的，上面的示例不使用引用的键。

## 6.5.5 阵列构造

可以使用熟悉的Java语法构建数组，可选择提供一个初始化器，以便在构建时填充数组。

```
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)
;
```

当构造多维数组时，它当前不允许提供初始化器。

## 6.5.6 方法

使用典型的Java编程语法调用方法。您也可以调用字面常量的方法。

```
// 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);
```

## 6.5.7 运算符

### 关系运算符

关系运算符 小于，小于等于，大于，大于等于，等于，不等于，使用标准运算符符号表示。

```
// 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);
```



大于/小于与`null`的比较遵循一个简单的规则：`null`在这里被视为没有（不是0）。因此，任何其他值始终大于`null`（`X > null`始终为真），并且没有其他值比它小（`X < null`始终为`false`）。如果您更喜欢数字比较，请避免基于数字的`null`比较，这有利于与零比较（例如`X > 0`或`X < 0`）。

除标准关系运算符外，SpEL还支持`instanceof` 和基于正则表达式的匹配运算符。

```
// evaluates to false
boolean falseValue = parser.parseExpression(
    "'xyz' instanceof T(Integer)").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);
```



请谨慎使用原始类型，因为它们会立即包装到包装器类型中，因此，如果预期的话，`1 instanceof T(int)` 将计算为`false`，而`1 instanceof T(Integer)` 的计算结果为`true`。

每个符号操作符也可以被指定为纯粹的字母等价物。这避免了所使用的符号对嵌入表达式的文档类型（例如XML文档）具有特殊含义的问题。文本等价物如下所示：`lt (<)`，`gt (>)`，`le (≤)`，`ge (≥)`，`eq (==)`，`ne (!=)`，`div (/)`，`mod (%)` `not (!)`。这些不区分大小写。

## 逻辑运算符

支持的逻辑运算符是and, or, and not。它们的用途如下所示。

```
// -- 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);
```

## 算术运算符

加法运算符可以用于数字和字符串。减法，乘法和除法只能用于数字。支持的其他算术运算符是模数（%）和指数幂（^）。执行标准运算符优先级。这些操作符将在下面展示。

```

// 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

```

## 6.5.8 赋值

通过使用赋值运算符来完成属性的设置。这通常在调用 `setValue` 之前完成，但也可以在对 `getValue` 的调用中完成。

```

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);

```

## 6.5.9 类型运算符

特殊的 T 运算符可用于指定 `java.lang.Class` (类型) 的实例。也可以使用此运算符调用静态方法。`TheStandardEvaluationContext` 使用 `TypeLocator` 来查找类型，并且可以使用对 `java.lang` 包的理解来构建 `StandardTypeLocator` (可以被替换)。这意味着对 `java.lang` 中的类型的引用 `T()` 不需要是完全限定的，但是所有其他类型引用必须是。

```
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);
```

## 6.5.10 构造函数

可以使用新的运算符调用构造函数。除了原始类型和字符串 (可以使用 `int`, `float` 等) 之外，所有标准类名称都应该被使用。

```
Inventor einstein = p.parseExpression(
    "new org.springframework.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.springframework.samples.spel.inventor.Inventor(
        'Albert Einstein', 'German'))").getValue(societyContext);
```

## 6.5.11 变量

可以使用语法 `#variableName` 在表达式中引用变量。使用 `StandardEvaluationContext` 上的 `setVariable` 方法设置变量。

```
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"
```

## #this 和 #root 变量

变量`#this`始终被定义，并且引用当前的运算操作对象（针对被解析的那个非限定引用）。变量`#root`始终被定义，并引用根上下文对象。虽然`#this`可能因为表达式的组件被运算操作而变化，但是`#root`总是引用根。

```
// 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);
```

## 6.5.12 功能

您可以通过注册能够在表达式字符串中调用的用户定义的函数来扩展SpEL。该功能通过方法使用 `StandardEvaluationContext` 进行注册。

```
public void registerFunction(String name, Method m)
```

对Java方法的引用提供了该函数的实现。例如，一个反转字符串的实用方法如下所示。

```
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();
}
```

然后将该方法注册到运算操作的上下文中，并可在表达式字符串中使用。

```
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);
```

### 6.5.13 Bean引用

如果使用bean解析器配置了运算操作上下文，则可以使用（@）符号从表达式中查找bean。

```
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);
```

要访问工厂bean本身，bean名称应改为带有（&）符号的前缀。

```
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);
```

### 6.5.14 三元运算符 (If-Then-Else)

您可以使用三元运算符在表达式中执行if-then-else条件逻辑。一个最小的例子是：

```
String falseString = parser.parseExpression(
    "false ? 'trueExp' : 'falseExp'").getValue(String.class);
```

在这种情况下，布尔值false会返回字符串值“falseExp”。一个更现实的例子如下所示。

```

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 + ' Soci-
    ety'";
String queryResultString = parser.parseExpression(expression)
    .getValue(societyContext, String.class);
// queryResultString = "Nikola Tesla is a member of the IEEE Society"

```

另请参阅Elvis操作员的下一部分，为三元运算符提供更短的语法。

## 6.5.15 Elvis操作符

Elvis操作符缩短了三元操作符语法，并以Groovy语言使用。通过三元运算符语法，您通常必须重复一次变量两次，例如：

```

String name = "Elvis Presley";
String displayName = name != null ? name : "Unknown";

```

相反，您可以使用Elvis操作符，命名与Elvis的发型相似。

```

ExpressionParser parser = new SpelExpressionParser();

String name = parser.parseExpression("name?:'Unknown'").getValue(String.class);

System.out.println(name); // 'Unknown'

```

这里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); // Nikola Tesla

tesla.setName(null);

name = parser.parseExpression("Name?:'Elvis Presley'").getValue(context, String.class);

System.out.println(name); // Elvis Presley

```

## 6.5.16 安全导航运算符

安全导航运算符用于避免 `NullPointerException` 并来自 Groovy 语言。通常当您对对象的引用时，您可能需要在访问对象的方法或属性之前验证它不为空。为了避免这种情况，安全导航运算符将简单地返回 `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!!!

```



`Elvis` 操作符可用于在表达式中设置默认值，例如在 `@Value` 表达式中：`@Value("# {systemProperties['pop3.port'] ?: 25}")` 这将注入系统属性 `pop3.port`（如果已定义）或 `25`（如果未定义）。

## 6.5.17 集合选择

选择是强大的表达式语言功能，允许您通过从其条目中选择将一些源集合转换为另一个。

选择使用语法 `.?[selectionExpression]`。这将过滤收集并返回一个包含原始元素子集的新集合。例如，选择将使我们能够轻松得到Serbian发明家名单：

```
List<Inventor> list = (List<Inventor>) parser.parseExpression(
    "Members.?[Nationality == 'Serbian']").getValue(societyContext);
```

`list`列表和`map`映射都可以进行选择。在前一种情况下，根据每个单独列表元素，同时针对`map`映射，根据每个`map`映射条目（Java类型`Map.Entry`的对象）运算操作作为选择标准。`map`映射条目的键和值可以作为选择中使用的属性访问。

此表达式将返回一个由原始`map`映射的元素组成的新`map`映射，其中条目值小于27。

```
Map newMap = parser.parseExpression("map.?[value<27]").getValue();
```

除了返回所有选定的元素之外，还可以检索第一个或最后一个值。要获得与选择匹配的第一个条目，语法为`^[...]`，同时获取最后匹配的选择，语法为`$[...]`。

## 6.5.18 集合投影

投影允许集合驱动子表达式的运算操作，结果是一个新的集合。投影的语法是`![projectionExpression]`。最容易理解的例子，假设我们有一个发明家的名单，但希望得到他们出生的城市的名单。有效地，我们要对发明人列表中的每个条目进行“`placeOfBirth.city`”运算操作。使用投影：

```
// returns ['Smiljan', 'Idvor']
List placesOfBirth = (List)parser.parseExpression("Members.! [placeOfBirth.city]");
```

`map`映射也可以用于驱动投影，在这种情况下，投影表达式将针对`map`映射中的每个条目进行运算操作（表示为Java `Map.Entry`）。跨`map`映射投影的结果是由对每个`map`映射条目的投影表达式的运算操作组成的列表。

## 6.5.19 表达式模板

表达式模板允许文字文本与一个或多个运算操作块进行混合。每个运算操作块都用您可以定义的前缀和后缀字符进行分隔，常用的选择是使用`#{}` 作为分隔符。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"
```

字符串通过连接文本文本'random number is'与运算操作#{}分隔符中的表达式的结果进行运算操作，在这种情况下是调用random()方法的结果。`parseExpression()`方法的第二个参数是ParserContext类型。`ParserContext`接口用于决定表达式如何被解析以支持表达式模板功能。`TemplateParserContext`的定义如下所示。

```
public class TemplateParserContext implements ParserContext {

    public String getExpressionPrefix() {
        return "#{";
    }

    public String getExpressionSuffix() {
        return "}";
    }

    public boolean isTemplate() {
        return true;
    }
}
```

## 6.6 示例中使用的类

### Inventor.java

```
package org.springframework.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;
    }
}
```

## 6.6 Classes used in the examples

---

```
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.springframework.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;
    }
}
```

Society.java

```
package org.springframework.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) {
        for (Inventor inventor : members) {
            if (inventor.getName().equals(name)) {
                return true;
            }
        }
        return false;
    }

}
```



## 7.1 Introduction

*Aspect-Oriented Programming* (AOP) complements Object-Oriented Programming (OOP) by providing another way of thinking about program structure. The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the *aspect*. Aspects enable the modularization of concerns such as transaction management that cut across multiple types and objects. (Such concerns are often termed *crosscutting* concerns in AOP literature.)

One of the key components of Spring is the *AOP framework*. While the Spring IoC container does not depend on AOP, meaning you do not need to use AOP if you don't want to, AOP complements Spring IoC to provide a very capable middleware solution.

### Spring 2.0 AOP

Spring 2.0 introduces a simpler and more powerful way of writing custom aspects using either a [schema-based approach](#) or the [@AspectJ annotation style](#). Both of these styles offer fully typed advice and use of the AspectJ pointcut language, while still using Spring AOP for weaving.

The Spring 2.0 schema- and @AspectJ-based AOP support is discussed in this chapter. Spring 2.0 AOP remains fully backwards compatible with Spring 1.2 AOP, and the lower-level AOP support offered by the Spring 1.2 APIs is discussed in [the following chapter](#).

AOP is used in the Spring Framework to...

- ... provide declarative enterprise services, especially as a replacement for EJB declarative services. The most important such service is [declarative transaction management](#).
- ... allow users to implement custom aspects, complementing their use of OOP with AOP.



If you are interested only in generic declarative services or other pre-packaged declarative middleware services such as pooling, you do not need to work directly with Spring AOP, and can skip most of this chapter.

### 7.1.1 AOP concepts

Let us begin by defining some central AOP concepts and terminology. These terms are not Spring-specific... unfortunately, AOP terminology is not particularly intuitive; however, it would be even more confusing if Spring used its own terminology.

- **Aspect:** a modularization of a concern that cuts across multiple classes. Transaction management is a good example of a crosscutting concern in enterprise Java applications. In Spring AOP, aspects are implemented using regular classes (the [schema-based approach](#)) or regular classes annotated with the `@Aspect` annotation (the [@AspectJ style](#)).
- **Join point:** a point during the execution of a program, such as the execution of a method or the handling of an exception. In Spring AOP, a join point *always* represents a method execution.
- **Advice:** action taken by an aspect at a particular join point. Different types of advice include "around," "before" and "after" advice. (Advice types are discussed below.) Many AOP frameworks, including Spring, model an advice as an *interceptor*, maintaining a chain of interceptors *around* the join point.
- **Pointcut:** a predicate that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (for example, the execution of a method with a certain name). The concept of join points as matched by pointcut expressions is central to AOP, and Spring uses the AspectJ pointcut expression language by default.
- **Introduction:** declaring additional methods or fields on behalf of a type. Spring AOP allows you to introduce new interfaces (and a corresponding implementation) to any advised object. For example, you could use an introduction to make a bean implement an `IsModified` interface, to simplify caching. (An introduction is known as an inter-type declaration in the AspectJ community.)
- **Target object:** object being advised by one or more aspects. Also referred to as the *advised* object. Since Spring AOP is implemented using runtime proxies, this object will always be a *proxied* object.
- **AOP proxy:** an object created by the AOP framework in order to implement the aspect contracts (advise method executions and so on). In the Spring Framework, an AOP proxy will be a JDK dynamic proxy or a CGLIB proxy.
- **Weaving:** linking aspects with other application types or objects to create an advised object. This can be done at compile time (using the AspectJ compiler, for example), load time, or at runtime. Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime.

Types of advice:

- **Before advice:** Advice that executes before a join point, but which does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).
- **After returning advice:** Advice to be executed after a join point completes normally: for example, if a method returns without throwing an exception.
- **After throwing advice:** Advice to be executed if a method exits by throwing an exception.

- *After (finally) advice*: Advice to be executed regardless of the means by which a join point exits (normal or exceptional return).
- *Around advice*: Advice that surrounds a join point such as a method invocation. This is the most powerful kind of advice. Around advice can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception.

Around advice is the most general kind of advice. Since Spring AOP, like AspectJ, provides a full range of advice types, we recommend that you use the least powerful advice type that can implement the required behavior. For example, if you need only to update a cache with the return value of a method, you are better off implementing an after returning advice than an around advice, although an around advice can accomplish the same thing. Using the most specific advice type provides a simpler programming model with less potential for errors. For example, you do not need to invoke the `proceed()` method on the `JoinPoint` used for around advice, and hence cannot fail to invoke it.

In Spring 2.0, all advice parameters are statically typed, so that you work with advice parameters of the appropriate type (the type of the return value from a method execution for example) rather than `Object` arrays.

The concept of join points, matched by pointcuts, is the key to AOP which distinguishes it from older technologies offering only interception. Pointcuts enable advice to be targeted independently of the Object-Oriented hierarchy. For example, an around advice providing declarative transaction management can be applied to a set of methods spanning multiple objects (such as all business operations in the service layer).

## 7.1.2 Spring AOP capabilities and goals

Spring AOP is implemented in pure Java. There is no need for a special compilation process. Spring AOP does not need to control the class loader hierarchy, and is thus suitable for use in a Servlet container or application server.

Spring AOP currently supports only method execution join points (advising the execution of methods on Spring beans). Field interception is not implemented, although support for field interception could be added without breaking the core Spring AOP APIs. If you need to advise field access and update join points, consider a language such as AspectJ.

Spring AOP's approach to AOP differs from that of most other AOP frameworks. The aim is not to provide the most complete AOP implementation (although Spring AOP is quite capable); it is rather to provide a close integration between AOP implementation and Spring IoC to help solve common problems in enterprise applications.

Thus, for example, the Spring Framework's AOP functionality is normally used in conjunction with the Spring IoC container. Aspects are configured using normal bean definition syntax (although this allows powerful "autoproxying" capabilities): this is a crucial difference from other AOP implementations. There are some things you cannot do easily or efficiently with Spring AOP, such as advise very fine-grained objects (such as domain objects typically): AspectJ is the best choice in such cases. However, our experience is that Spring AOP provides an excellent solution to most problems in enterprise Java applications that are amenable to AOP.

Spring AOP will never strive to compete with AspectJ to provide a comprehensive AOP solution. We believe that both proxy-based frameworks like Spring AOP and full-blown frameworks such as AspectJ are valuable, and that they are complementary, rather than in competition. Spring seamlessly integrates Spring AOP and IoC with AspectJ, to enable all uses of AOP to be catered for within a consistent Spring-based application architecture. This integration does not affect the Spring AOP API or the AOP Alliance API: Spring AOP remains backward-compatible. See [the following chapter](#) for a discussion of the Spring AOP APIs.



One of the central tenets of the Spring Framework is that of *non-invasiveness*; this is the idea that you should not be forced to introduce framework-specific classes and interfaces into your business/domain model. However, in some places the Spring Framework does give you the option to introduce Spring Framework-specific dependencies into your codebase: the rationale in giving you such options is because in certain scenarios it might be just plain easier to read or code some specific piece of functionality in such a way. The Spring Framework (almost) always offers you the choice though: you have the freedom to make an informed decision as to which option best suits your particular use case or scenario. One such choice that is relevant to this chapter is that of which AOP framework (and which AOP style) to choose. You have the choice of AspectJ and/or Spring AOP, and you also have the choice of either the @AspectJ annotation-style approach or the Spring XML configuration-style approach. The fact that this chapter chooses to introduce the @AspectJ-style approach first should not be taken as an indication that the Spring team favors the @AspectJ annotation-style approach over the Spring XML configuration-style. See [Section 7.4, “Choosing which AOP declaration style to use”](#) for a more complete discussion of the whys and wherefores of each style.

### 7.1.3 AOP Proxies

Spring AOP defaults to using standard JDK *dynamic proxies* for AOP proxies. This enables any interface (or set of interfaces) to be proxied.

Spring AOP can also use CGLIB proxies. This is necessary to proxy classes rather than interfaces. CGLIB is used by default if a business object does not implement an interface. As it is good practice to program to interfaces rather than classes; business classes normally

will implement one or more business interfaces. It is possible to [force the use of CGLIB](#), in those (hopefully rare) cases where you need to advise a method that is not declared on an interface, or where you need to pass a proxied object to a method as a concrete type.

It is important to grasp the fact that Spring AOP is *proxy-based*. See [Section 7.6.1, “Understanding AOP proxies”](#) for a thorough examination of exactly what this implementation detail actually means.

## 7.2 @AspectJ support

@AspectJ refers to a style of declaring aspects as regular Java classes annotated with annotations. The @AspectJ style was introduced by the [AspectJ project](#) as part of the AspectJ 5 release. Spring interprets the same annotations as AspectJ 5, using a library supplied by AspectJ for pointcut parsing and matching. The AOP runtime is still pure Spring AOP though, and there is no dependency on the AspectJ compiler or weaver.

Using the AspectJ compiler and weaver enables use of the full AspectJ language, and is discussed in [Section 7.8, “Using AspectJ with Spring applications”](#).

### 7.2.1 Enabling @AspectJ Support

To use @AspectJ aspects in a Spring configuration you need to enable Spring support for configuring Spring AOP based on @AspectJ aspects, and *autoproxying* beans based on whether or not they are advised by those aspects. By autoproxying we mean that if Spring determines that a bean is advised by one or more aspects, it will automatically generate a proxy for that bean to intercept method invocations and ensure that advice is executed as needed.

The @AspectJ support can be enabled with XML or Java style configuration. In either case you will also need to ensure that AspectJ’s `aspectjweaver.jar` library is on the classpath of your application (version 1.6.8 or later). This library is available in the `'lib'` directory of an AspectJ distribution or via the Maven Central repository.

### Enabling @AspectJ Support with Java configuration

To enable @AspectJ support with Java `@Configuration` add the `@EnableAspectJAutoProxy` annotation:

```
@Configuration  
@EnableAspectJAutoProxy  
public class AppConfig {  
}
```

### Enabling @AspectJ Support with XML configuration

To enable @AspectJ support with XML based configuration use the `aop:aspectj-autoproxy` element:

```
<aop:aspectj-autoproxy/>
```

This assumes that you are using schema support as described in [Chapter 38, XML Schema-based configuration](#). See [Section 38.2.7, “the aop schema”](#) for how to import the tags in the `aop` namespace.

## 7.2.2 Declaring an aspect

With the @AspectJ support enabled, any bean defined in your application context with a class that is an @AspectJ aspect (has the `@Aspect` annotation) will be automatically detected by Spring and used to configure Spring AOP. The following example shows the minimal definition required for a not-very-useful aspect:

A regular bean definition in the application context, pointing to a bean class that has the `@Aspect` annotation:

```
<bean id="myAspect" class="org.xyz.NotVeryUsefulAspect">
    <!-- configure properties of aspect here as normal -->
</bean>
```

And the `NotVeryUsefulAspect` class definition, annotated with `org.aspectj.lang.annotation.Aspect` annotation:

```
package org.xyz;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class NotVeryUsefulAspect {
```

Aspects (classes annotated with `@Aspect`) may have methods and fields just like any other class. They may also contain pointcut, advice, and introduction (inter-type) declarations.

You may register aspect classes as regular beans in your Spring XML configuration, or autodetect them through classpath scanning - just like any other Spring-managed bean. However, note that the `@Aspect` annotation is *not* sufficient for autodetection in the classpath: For that purpose, you need to add a separate `@Component` annotation (or alternatively a custom stereotype annotation that qualifies, as per the rules of Spring's component scanner).

In Spring AOP, it is *not* possible to have aspects themselves be the target of advice from other aspects. The `@Aspect` annotation on a class marks it as an aspect, and hence excludes it from auto-proxying.

### 7.2.3 Declaring a pointcut

Recall that pointcuts determine join points of interest, and thus enable us to control when advice executes. *Spring AOP only supports method execution join points for Spring beans*, so you can think of a pointcut as matching the execution of methods on Spring beans. A pointcut declaration has two parts: a signature comprising a name and any parameters, and a pointcut expression that determines *exactly* which method executions we are interested in. In the `@AspectJ` annotation-style of AOP, a pointcut signature is provided by a regular method definition, and the pointcut expression is indicated using the `@Pointcut` annotation (the method serving as the pointcut signature *must* have a `void` return type).

An example will help make this distinction between a pointcut signature and a pointcut expression clear. The following example defines a pointcut named `'anyOldTransfer'` that will match the execution of any method named `'transfer'`:

```
@Pointcut("execution(* transfer(..))")// the pointcut expression
private void anyOldTransfer() {}// the pointcut signature
```

The pointcut expression that forms the value of the `@Pointcut` annotation is a regular AspectJ 5 pointcut expression. For a full discussion of AspectJ's pointcut language, see the [AspectJ Programming Guide](#) (and for extensions, the [AspectJ 5 Developers Notebook](#)) or one of the books on AspectJ such as "Eclipse AspectJ" by Colyer et. al. or "AspectJ in Action" by Ramnivas Laddad.

## Supported Pointcut Designators

Spring AOP supports the following AspectJ pointcut designators (PCD) for use in pointcut expressions:

## Other pointcut types

The full AspectJ pointcut language supports additional pointcut designators that are not supported in Spring. These are: `call`, `get`, `set`, `preinitialization`, `staticinitialization`, `initialization`, `handler`, `adviceexecution`, `withincode`, `cflow`, `cflowbelow`, `if`, `@this`, and `@withincode`. Use of these pointcut designators in pointcut expressions interpreted by Spring AOP will result in an `IllegalArgumentException` being thrown.

The set of pointcut designators supported by Spring AOP may be extended in future releases to support more of the AspectJ pointcut designators.

- `execution` - for matching method execution join points, this is the primary pointcut designator you will use when working with Spring AOP
- `within` - limits matching to join points within certain types (simply the execution of a method declared within a matching type when using Spring AOP)
- `this` - limits matching to join points (the execution of methods when using Spring AOP) where the bean reference (Spring AOP proxy) is an instance of the given type
- `target` - limits matching to join points (the execution of methods when using Spring AOP) where the target object (application object being proxied) is an instance of the given type
- `args` - limits matching to join points (the execution of methods when using Spring AOP) where the arguments are instances of the given types
- `@target` - limits matching to join points (the execution of methods when using Spring AOP) where the class of the executing object has an annotation of the given type
- `@args` - limits matching to join points (the execution of methods when using Spring AOP) where the runtime type of the actual arguments passed have annotations of the given type(s)
- `@within` - limits matching to join points within types that have the given annotation (the execution of methods declared in types with the given annotation when using Spring AOP)
- `@annotation` - limits matching to join points where the subject of the join point (method being executed in Spring AOP) has the given annotation

Because Spring AOP limits matching to only method execution join points, the discussion of the pointcut designators above gives a narrower definition than you will find in the AspectJ programming guide. In addition, AspectJ itself has type-based semantics and at an execution join point both `this` and `target` refer to the same object - the object executing the method. Spring AOP is a proxy-based system and differentiates between the proxy object itself (bound to `this`) and the target object behind the proxy (bound to `target`).



Due to the proxy-based nature of Spring's AOP framework, protected methods are by definition *not* intercepted, neither for JDK proxies (where this isn't applicable) nor for CGLIB proxies (where this is technically possible but not recommendable for AOP purposes). As a consequence, any given pointcut will be matched against *public methods only!* If your interception needs include protected/private methods or even constructors, consider the use of Spring-driven [native AspectJ weaving](#) instead of Spring's proxy-based AOP framework. This constitutes a different mode of AOP usage with different characteristics, so be sure to make yourself familiar with weaving first before making a decision.

Spring AOP also supports an additional PCD named `bean`. This PCD allows you to limit the matching of join points to a particular named Spring bean, or to a set of named Spring beans (when using wildcards). The `bean` PCD has the following form:

```
bean(idOrNameOfBean)
```

The `idOrNameOfBean` token can be the name of any Spring bean: limited wildcard support using the `*` character is provided, so if you establish some naming conventions for your Spring beans you can quite easily write a `bean` PCD expression to pick them out. As is the case with other pointcut designators, the `bean` PCD can be `&&`'ed, `||`'ed, and `!` (negated) too.



Please note that the `bean` PCD is *only* supported in Spring AOP - and *not* in native AspectJ weaving. It is a Spring-specific extension to the standard PCDs that AspectJ defines and therefore not available for aspects declared in the `@Aspect` model. The `bean` PCD operates at the *instance* level (building on the Spring bean name concept) rather than at the type level only (which is what weaving-based AOP is limited to). Instance-based pointcut designators are a special capability of Spring's proxy-based AOP framework and its close integration with the Spring bean factory, where it is natural and straightforward to identify specific beans by name.

## Combining pointcut expressions

Pointcut expressions can be combined using '`&&`', '`||`' and '`!`'. It is also possible to refer to pointcut expressions by name. The following example shows three pointcut expressions:

`anyPublicOperation` (which matches if a method execution join point represents the execution of any public method); `inTrading` (which matches if a method execution is in the trading module), and `tradingoperation` (which matches if a method execution represents any public method in the trading module).

```

@Pointcut("execution(public * *(..))")
private void anyPublicOperation() {}

@Pointcut("within(com.xyz.someapp.trading..*)")
private void inTrading() {}

@Pointcut("anyPublicOperation() && inTrading()")
private void tradingOperation() {}

```

It is a best practice to build more complex pointcut expressions out of smaller named components as shown above. When referring to pointcuts by name, normal Java visibility rules apply (you can see private pointcuts in the same type, protected pointcuts in the hierarchy, public pointcuts anywhere and so on). Visibility does not affect pointcut *matching*.

## Sharing common pointcut definitions

When working with enterprise applications, you often want to refer to modules of the application and particular sets of operations from within several aspects. We recommend defining a "SystemArchitecture" aspect that captures common pointcut expressions for this purpose. A typical such aspect would look as follows:

```

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.someapp.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() {}

}

```

The pointcuts defined in such an aspect can be referred to anywhere that you need a pointcut expression. For example, to make the service layer transactional, you could write:

```

<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>

```

The `<aop:config>` and `<aop:advisor>` elements are discussed in [Section 7.3, “Schema-based AOP support”](#). The transaction elements are discussed in [Chapter 13, Transaction Management](#).

## Examples

Spring AOP users are likely to use the `execution` pointcut designator the most often. The format of an execution expression is:

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?name-pattern(para
m-pattern)
           throws-pattern?)
```

All parts except the returning type pattern (ret-type-pattern in the snippet above), name pattern, and parameters pattern are optional. The returning type pattern determines what the return type of the method must be in order for a join point to be matched. Most frequently you will use `*` as the returning type pattern, which matches any return type. A fully-qualified type name will match only when the method returns the given type. The name pattern matches the method name. You can use the `*` wildcard as all or part of a name pattern. If specifying a declaring type pattern then include a trailing `.` to join it to the name pattern component. The parameters pattern is slightly more complex: `()` matches a method that takes no parameters, whereas `(...)` matches any number of parameters (zero or more). The pattern `(*)` matches a method taking one parameter of any type, `(*, String)` matches a method taking two parameters, the first can be of any type, the second must be a String. Consult the [Language Semantics](#) section of the AspectJ Programming Guide for more information.

Some examples of common pointcut expressions are given below.

- the execution of any public method:

```
execution(public * *(...))
```

- the execution of any method with a name beginning with "set":

```
execution(* set*(...))
```

- the execution of any method defined by the `AccountService` interface:

```
execution(* com.xyz.service.AccountService.*(...))
```

- the execution of any method defined in the service package:

```
execution(* com.xyz.service.*.*(...))
```

- the execution of any method defined in the service package or a sub-package:

```
execution(* com.xyz.service..*.*(..))
```

- any join point (method execution only in Spring AOP) within the service package:

```
within(com.xyz.service.*)
```

- any join point (method execution only in Spring AOP) within the service package or a sub-package:

```
within(com.xyz.service..*)
```

- any join point (method execution only in Spring AOP) where the proxy implements the `AccountService` interface:

```
this(com.xyz.service.AccountService)
```



'this' is more commonly used in a binding form :- see the following section on advice for how to make the proxy object available in the advice body.

- any join point (method execution only in Spring AOP) where the target object implements the `AccountService` interface:

```
target(com.xyz.service.AccountService)
```



'target' is more commonly used in a binding form :- see the following section on advice for how to make the target object available in the advice body.

- any join point (method execution only in Spring AOP) which takes a single parameter, and where the argument passed at runtime is `Serializable` :

```
args(java.io.Serializable)
```



'args' is more commonly used in a binding form :- see the following section on advice for how to make the method arguments available in the advice body.

Note that the pointcut given in this example is different to `execution(* * (java.io.Serializable))` : the args version matches if the argument passed at runtime is Serializable, the execution version matches if the method signature declares a single parameter of type `Serializable`.

- any join point (method execution only in Spring AOP) where the target object has an `@Transactional` annotation:

```
@target(org.springframework.transaction.annotation.Transactional)
```



'@target' can also be used in a binding form :- see the following section on advice for how to make the annotation object available in the advice body.

- any join point (method execution only in Spring AOP) where the declared type of the target object has an `@Transactional` annotation:

```
@within(org.springframework.transaction.annotation.Transactional)
```



'@within' can also be used in a binding form :- see the following section on advice for how to make the annotation object available in the advice body.

- any join point (method execution only in Spring AOP) where the executing method has an `@Transactional` annotation:

```
@annotation(org.springframework.transaction.annotation.Transactional)
```



'@annotation' can also be used in a binding form :- see the following section on advice for how to make the annotation object available in the advice body.

- any join point (method execution only in Spring AOP) which takes a single parameter, and where the runtime type of the argument passed has the `@Classified` annotation:

```
@args(com.xyz.security.Classified)
```



'@args' can also be used in a binding form :- see the following section on advice for how to make the annotation object(s) available in the advice body.

- any join point (method execution only in Spring AOP) on a Spring bean named

```
tradeService :
```

```
bean(tradeService)
```

- any join point (method execution only in Spring AOP) on Spring beans having names that match the wildcard expression `*Service` :

```
bean(*Service)
```

## Writing good pointcuts

During compilation, AspectJ processes pointcuts in order to try and optimize matching performance. Examining code and determining if each join point matches (statically or dynamically) a given pointcut is a costly process. (A dynamic match means the match cannot be fully determined from static analysis and a test will be placed in the code to determine if there is an actual match when the code is running). On first encountering a pointcut declaration, AspectJ will rewrite it into an optimal form for the matching process. What does this mean? Basically pointcuts are rewritten in DNF (Disjunctive Normal Form) and the components of the pointcut are sorted such that those components that are cheaper to evaluate are checked first. This means you do not have to worry about understanding the performance of various pointcut designators and may supply them in any order in a pointcut declaration.

However, AspectJ can only work with what it is told, and for optimal performance of matching you should think about what they are trying to achieve and narrow the search space for matches as much as possible in the definition. The existing designators naturally fall into one of three groups: kinded, scoping and context:

- Kinded designators are those which select a particular kind of join point. For example: execution, get, set, call, handler
- Scoping designators are those which select a group of join points of interest (of probably many kinds). For example: within, withincode
- Contextual designators are those that match (and optionally bind) based on context. For example: this, target, @annotation

A well written pointcut should try and include at least the first two types (kinded and scoping), whilst the contextual designators may be included if wishing to match based on join point context, or bind that context for use in the advice. Supplying either just a kinded designator or just a contextual designator will work but could affect weaving performance (time and memory used) due to all the extra processing and analysis. Scoping designators

are very fast to match and their usage means AspectJ can very quickly dismiss groups of join points that should not be further processed - that is why a good pointcut should always include one if possible.

## 7.2.4 Declaring advice

Advice is associated with a pointcut expression, and runs before, after, or around method executions matched by the pointcut. The pointcut expression may be either a simple reference to a named pointcut, or a pointcut expression declared in place.

### Before advice

Before advice is declared in an aspect using the `@Before` annotation:

```
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() {
        // ...
    }

}
```

If using an in-place pointcut expression we could rewrite the above example as:

```
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() {
        // ...
    }

}
```

### After returning advice

After returning advice runs when a matched method execution returns normally. It is declared using the `@AfterReturning` annotation:

```
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() {
        // ...
    }

}
```



Note: it is of course possible to have multiple advice declarations, and other members as well, all inside the same aspect. We're just showing a single advice declaration in these examples to focus on the issue under discussion at the time.

Sometimes you need access in the advice body to the actual value that was returned. You can use the form of `@AfterReturning` that binds the return value for this:

```
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) {
        // ...
    }

}
```

The name used in the `returning` attribute must correspond to the name of a parameter in the advice method. When a method execution returns, the return value will be passed to the advice method as the corresponding argument value. A `returning` clause also restricts matching to only those method executions that return a value of the specified type (`Object` in this case, which will match any return value).

Please note that it is *not* possible to return a totally different reference when using after-returning advice.

## After throwing advice

After throwing advice runs when a matched method execution exits by throwing an exception. It is declared using the `@AfterThrowing` annotation:

```
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() {
        // ...
    }

}
```

Often you want the advice to run only when exceptions of a given type are thrown, and you also often need access to the thrown exception in the advice body. Use the `throwing` attribute to both restrict matching (if desired, use `Throwable` as the exception type otherwise) and bind the thrown exception to an advice parameter.

```
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) {
        // ...
    }

}
```

The name used in the `throwing` attribute must correspond to the name of a parameter in the advice method. When a method execution exits by throwing an exception, the exception will be passed to the advice method as the corresponding argument value. A `throwing` clause also restricts matching to only those method executions that throw an exception of the specified type (`DataAccessException` in this case).

## After (finally) advice

After (finally) advice runs however a matched method execution exits. It is declared using the `@After` annotation. After advice must be prepared to handle both normal and exception return conditions. It is typically used for releasing resources, etc.

```
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 advice

The final kind of advice is around advice. Around advice runs "around" a matched method execution. It has the opportunity to do work both before and after the method executes, and to determine when, how, and even if, the method actually gets to execute at all. Around advice is often used if you need to share state before and after a method execution in a thread-safe manner (starting and stopping a timer for example). Always use the least powerful form of advice that meets your requirements (i.e. don't use around advice if simple before advice would do).

Around advice is declared using the `@Around` annotation. The first parameter of the advice method must be of type `ProceedingJoinPoint`. Within the body of the advice, calling `proceed()` on the `ProceedingJoinPoint` causes the underlying method to execute. The `proceed` method may also be called passing in an `Object[]` - the values in the array will be used as the arguments to the method execution when it proceeds.

The behavior of proceed when called with an Object[] is a little different than the behavior of proceed for around advice compiled by the AspectJ compiler. For around advice written using the traditional AspectJ language, the number of arguments passed to proceed must match the number of arguments passed to the around advice (not the number of arguments taken by the underlying join point), and the value passed to proceed in a given argument position supplants the original value at the join point for the entity the value was bound to (Don't worry if this doesn't make sense right now!). The approach taken by Spring is simpler and a better match to its proxy-based, execution only semantics. You only need to be aware of this difference if you are compiling @AspectJ aspects written for Spring and using proceed with arguments with the AspectJ compiler and weaver. There is a way to write such aspects that is 100% compatible across both Spring AOP and AspectJ, and this is discussed in the following section on advice parameters.

```
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;
    }
}
```

The value returned by the around advice will be the return value seen by the caller of the method. A simple caching aspect for example could return a value from a cache if it has one, and invoke proceed() if it does not. Note that proceed may be invoked once, many times, or not at all within the body of the around advice, all of these are quite legal.

## Advice parameters

Spring offers fully typed advice - meaning that you declare the parameters you need in the advice signature (as we saw for the returning and throwing examples above) rather than work with `Object[]` arrays all the time. We'll see how to make argument and other contextual values available to the advice body in a moment. First let's take a look at how to write generic advice that can find out about the method the advice is currently advising.

### Access to the current JoinPoint

Any advice method may declare as its first parameter, a parameter of type

`org.aspectj.lang.JoinPoint` (please note that around advice is *required* to declare a first parameter of type `ProceedingJoinPoint`, which is a subclass of `JoinPoint`). The `JoinPoint` interface provides a number of useful methods such as `getArgs()` (returns the method arguments), `getThis()` (returns the proxy object), `getTarget()` (returns the target object), `getSignature()` (returns a description of the method that is being advised) and `toString()` (prints a useful description of the method being advised). Please do consult the javadocs for full details.

## Passing parameters to advice

We've already seen how to bind the returned value or exception value (using after returning and after throwing advice). To make argument values available to the advice body, you can use the binding form of `args`. If a parameter name is used in place of a type name in an `args` expression, then the value of the corresponding argument will be passed as the parameter value when the advice is invoked. An example should make this clearer. Suppose you want to advise the execution of dao operations that take an `Account` object as the first parameter, and you need access to the account in the advice body. You could write the following:

```
@Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation() && args(account,...)")
public void validateAccount(Account account) {
    // ...
}
```

The `args(account,...)` part of the pointcut expression serves two purposes: firstly, it restricts matching to only those method executions where the method takes at least one parameter, and the argument passed to that parameter is an instance of `Account`; secondly, it makes the actual `Account` object available to the advice via the `account` parameter.

Another way of writing this is to declare a pointcut that "provides" the `Account` object value when it matches a join point, and then just refer to the named pointcut from the advice. This would look as follows:

```
@Pointcut("com.xyz.myapp.SystemArchitecture.dataAccessOperation() && args(account,...)")
private void accountDataAccessOperation(Account account) {}

@Before("accountDataAccessOperation(account)")
public void validateAccount(Account account) {
    // ...
}
```

The interested reader is once more referred to the AspectJ programming guide for more details.

The proxy object (`this`), target object (`target`), and annotations (`@within`, `@target`, `@annotation`, `@args`) can all be bound in a similar fashion. The following example shows how you could match the execution of methods annotated with an `@Auditable` annotation, and extract the audit code.

First the definition of the `@Auditable` annotation:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Auditable {
    AuditCode value();
}
```

And then the advice that matches the execution of `@Auditable` methods:

```
@Before("com.xyz.lib.Pointcuts.anyPublicMethod() && @annotation(auditable)")
public void audit(Auditable auditable) {
    AuditCode code = auditable.value();
    // ...
}
```

## Advice parameters and generics

Spring AOP can handle generics used in class declarations and method parameters. Suppose you have a generic type like this:

```
public interface Sample<T> {
    void sampleGenericMethod(T param);
    void sampleGenericCollectionMethod(Collection<T> param);
}
```

You can restrict interception of method types to certain parameter types by simply typing the advice parameter to the parameter type you want to intercept the method for:

```
@Before("execution(* ..Sample+.sampleGenericMethod(*)) && args(param)")
public void beforeSampleMethod(MyType param) {
    // Advice implementation
}
```

That this works is pretty obvious as we already discussed above. However, it's worth pointing out that this won't work for generic collections. So you cannot define a pointcut like this:

```
@Before("execution(* ..Sample+.sampleGenericCollectionMethod(*)) && args(param)")
public void beforeSampleMethod(Collection<MyType> param) {
    // Advice implementation
}
```

To make this work we would have to inspect every element of the collection, which is not reasonable as we also cannot decide how to treat `null` values in general. To achieve something similar to this you have to type the parameter to `collection<?>` and manually check the type of the elements.

## Determining argument names

The parameter binding in advice invocations relies on matching names used in pointcut expressions to declared parameter names in (advice and pointcut) method signatures. Parameter names are *not* available through Java reflection, so Spring AOP uses the following strategies to determine parameter names:

- If the parameter names have been specified by the user explicitly, then the specified parameter names are used: both the advice and the pointcut annotations have an optional "argNames" attribute which can be used to specify the argument names of the annotated method - these argument names are available at runtime. For example:

```
@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
}
```

If the first parameter is of the `JoinPoint`, `ProceedingJoinPoint`, or `JoinPoint.StaticPart` type, you may leave out the name of the parameter from the value of the "argNames" attribute. For example, if you modify the preceding advice to receive the join point object, the "argNames" attribute need not include it:

```

@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
}

```

The special treatment given to the first parameter of the `JoinPoint`, `ProceedingJoinPoint`, and `JoinPoint.StaticPart` types is particularly convenient for advice that do not collect any other join point context. In such situations, you may simply omit the "argNames" attribute. For example, the following advice need not declare the "argNames" attribute:

```

@Before("com.xyz.lib.Pointcuts.anyPublicMethod()")
public void audit(JoinPoint jp) {
    // ... use jp
}

```

- Using the `'argNames'` attribute is a little clumsy, so if the `'argNames'` attribute has not been specified, then Spring AOP will look at the debug information for the class and try to determine the parameter names from the local variable table. This information will be present as long as the classes have been compiled with debug information (`'-g:vars'` at a minimum). The consequences of compiling with this flag on are: (1) your code will be slightly easier to understand (reverse engineer), (2) the class file sizes will be very slightly bigger (typically inconsequential), (3) the optimization to remove unused local variables will not be applied by your compiler. In other words, you should encounter no difficulties building with this flag on.



If an `@AspectJ` aspect has been compiled by the AspectJ compiler (ajc) even without the debug information then there is no need to add the `argNames` attribute as the compiler will retain the needed information.

- If the code has been compiled without the necessary debug information, then Spring AOP will attempt to deduce the pairing of binding variables to parameters (for example, if only one variable is bound in the pointcut expression, and the advice method only takes one parameter, the pairing is obvious!). If the binding of variables is ambiguous given the available information, then an `AmbiguousBindingException` will be thrown.
- If all of the above strategies fail then an `IllegalArgumentException` will be thrown.

## Proceeding with arguments

We remarked earlier that we would describe how to write a proceed call *with arguments* that works consistently across Spring AOP and AspectJ. The solution is simply to ensure that the advice signature binds each of the method parameters in order. For example:

```
@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});
}
```

In many cases you will be doing this binding anyway (as in the example above).

## Advice ordering

What happens when multiple pieces of advice all want to run at the same join point? Spring AOP follows the same precedence rules as AspectJ to determine the order of advice execution. The highest precedence advice runs first "on the way in" (so given two pieces of before advice, the one with highest precedence runs first). "On the way out" from a join point, the highest precedence advice runs last (so given two pieces of after advice, the one with the highest precedence will run second).

When two pieces of advice defined in *different* aspects both need to run at the same join point, unless you specify otherwise the order of execution is undefined. You can control the order of execution by specifying precedence. This is done in the normal Spring way by either implementing the `org.springframework.core.Ordered` interface in the aspect class or annotating it with the `Order` annotation. Given two aspects, the aspect returning the lower value from `Ordered.getValue()` (or the annotation value) has the higher precedence.

When two pieces of advice defined in *the same* aspect both need to run at the same join point, the ordering is undefined (since there is no way to retrieve the declaration order via reflection for javac-compiled classes). Consider collapsing such advice methods into one advice method per join point in each aspect class, or refactor the pieces of advice into separate aspect classes - which can be ordered at the aspect level.

### 7.2.5 Introductions

Introductions (known as inter-type declarations in AspectJ) enable an aspect to declare that advised objects implement a given interface, and to provide an implementation of that interface on behalf of those objects.

An introduction is made using the `@DeclareParents` annotation. This annotation is used to declare that matching types have a new parent (hence the name). For example, given an interface `UsageTracked`, and an implementation of that interface `DefaultUsageTracked`, the following aspect declares that all implementors of service interfaces also implement the `UsageTracked` interface. (In order to expose statistics via JMX for example.)

```
@Aspect
public class UsageTracking {

    @DeclareParents(value="com.xyz.myapp.service.*+", defaultImpl=DefaultUsageTracked.class)
    public static UsageTracked mixin;

    @Before("com.xyz.myapp.SystemArchitecture.businessService() && this(usageTracked)")
    public void recordUsage(UsageTracked usageTracked) {
        usageTracked.incrementUseCount();
    }

}
```

The interface to be implemented is determined by the type of the annotated field. The `value` attribute of the `@DeclareParents` annotation is an AspectJ type pattern :- any bean of a matching type will implement the `UsageTracked` interface. Note that in the before advice of the above example, service beans can be directly used as implementations of the `UsageTracked` interface. If accessing a bean programmatically you would write the following:

```
UsageTracked usageTracked = (UsageTracked) context.getBean("myService");
```

## 7.2.6 Aspect instantiation models

(This is an advanced topic, so if you are just starting out with AOP you can safely skip it until later.)

By default there will be a single instance of each aspect within the application context. AspectJ calls this the singleton instantiation model. It is possible to define aspects with alternate lifecycles :- Spring supports AspectJ's `perthis` and `pertarget` instantiation models (`percflow`, `percflowbelow`, and `pertypewithin` are not currently supported).

A "perthis" aspect is declared by specifying a `perthis` clause in the `@Aspect` annotation. Let's look at an example, and then we'll explain how it works.

```

@Aspect("perthis(com.xyz.myapp.SystemArchitecture.businessService())")
public class MyAspect {

    private int someState;

    @Before(com.xyz.myapp.SystemArchitecture.businessService())
    public void recordServiceUsage() {
        // ...
    }

}

```

The effect of the '`perthis`' clause is that one aspect instance will be created for each unique service object executing a business service (each unique object bound to 'this' at join points matched by the pointcut expression). The aspect instance is created the first time that a method is invoked on the service object. The aspect goes out of scope when the service object goes out of scope. Before the aspect instance is created, none of the advice within it executes. As soon as the aspect instance has been created, the advice declared within it will execute at matched join points, but only when the service object is the one this aspect is associated with. See the AspectJ programming guide for more information on per-clauses.

The '`pertarget`' instantiation model works in exactly the same way as `perthis`, but creates one aspect instance for each unique target object at matched join points.

## 7.2.7 Example

Now that you have seen how all the constituent parts work, let's put them together to do something useful!

The execution of business services can sometimes fail due to concurrency issues (for example, deadlock loser). If the operation is retried, it is quite likely to succeed next time round. For business services where it is appropriate to retry in such conditions (idempotent operations that don't need to go back to the user for conflict resolution), we'd like to transparently retry the operation to avoid the client seeing a

`PessimisticLockingFailureException`. This is a requirement that clearly cuts across multiple services in the service layer, and hence is ideal for implementing via an aspect.

Because we want to retry the operation, we will need to use around advice so that we can call proceed multiple times. Here's how the basic aspect implementation looks:

```

@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;
    }

}

```

Note that the aspect implements the `Ordered` interface so we can set the precedence of the aspect higher than the transaction advice (we want a fresh transaction each time we retry). The `maxRetries` and `order` properties will both be configured by Spring. The main action happens in the `doConcurrentOperation` around advice. Notice that for the moment we're applying the retry logic to all `businessService()`s. We try to proceed, and if we fail with an `PessimisticLockingFailureException` we simply try again unless we have exhausted all of our retry attempts.

The corresponding Spring configuration is:

```
<aop:aspectj-autoproxy>

<bean id="concurrentOperationExecutor" class="com.xyz.myapp.service.impl.ConcurrentOperationExecutor">
    <property name="maxRetries" value="3"/>
    <property name="order" value="100"/>
</bean>
```

To refine the aspect so that it only retries idempotent operations, we might define an `Idempotent` annotation:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Idempotent {
    // marker annotation
}
```

and use the annotation to annotate the implementation of service operations. The change to the aspect to only retry idempotent operations simply involves refining the pointcut expression so that only `@Idempotent` operations match:

```
@Around("com.xyz.myapp.SystemArchitecture.businessService() && " +
        "@annotation(com.xyz.myapp.service.Idempotent)")
public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {
    ...
}
```

## 7.3 Schema-based AOP support

If you prefer an XML-based format, then Spring also offers support for defining aspects using the new "aop" namespace tags. The exact same pointcut expressions and advice kinds are supported as when using the @AspectJ style, hence in this section we will focus on the new syntax and refer the reader to the discussion in the previous section ([Section 7.2, “@AspectJ support”](#)) for an understanding of writing pointcut expressions and the binding of advice parameters.

To use the aop namespace tags described in this section, you need to import the `spring-aop` schema as described in [Chapter 38, XML Schema-based configuration](#). See [Section 38.2.7, “the aop schema”](#) for how to import the tags in the `aop` namespace.

Within your Spring configurations, all aspect and advisor elements must be placed within an `<aop:config>` element (you can have more than one `<aop:config>` element in an application context configuration). An `<aop:config>` element can contain pointcut, advisor, and aspect elements (note these must be declared in that order).

The `<aop:config>` style of configuration makes heavy use of Spring’s [auto-proxying](#) mechanism. This can cause issues (such as advice not being woven) if you are already using explicit auto-proxying via the use of `BeanNameAutoProxyCreator` or suchlike. The recommended usage pattern is to use either just the `<aop:config>` style, or just the `AutoProxyCreator` style.

### 7.3.1 Declaring an aspect

Using the schema support, an aspect is simply a regular Java object defined as a bean in your Spring application context. The state and behavior is captured in the fields and methods of the object, and the pointcut and advice information is captured in the XML.

An aspect is declared using the `element`, and the backing bean is referenced using the `ref` attribute:

```

<aop:config>
    <aop:aspect id="myAspect" ref="aBean">
        ...
        </aop:aspect>
    </aop:config>

    <bean id="aBean" class="...">
        ...
    </bean>

```

The bean backing the aspect (" `aBean` " in this case) can of course be configured and dependency injected just like any other Spring bean.

## 7.3.2 Declaring a pointcut

A named pointcut can be declared inside an element, enabling the pointcut definition to be shared across several aspects and advisors.

A pointcut representing the execution of any business service in the service layer could be defined as follows:

```

<aop:config>

    <aop:pointcut id="businessService"
        expression="execution(* com.xyz.myapp.service.*.*(..))"/>

</aop:config>

```

Note that the pointcut expression itself is using the same AspectJ pointcut expression language as described in [Section 7.2, “@AspectJ support”](#). If you are using the schema based declaration style, you can refer to named pointcuts defined in types (`@Aspects`) within the pointcut expression. Another way of defining the above pointcut would be:

```

<aop:config>

    <aop:pointcut id="businessService"
        expression="com.xyz.myapp.SystemArchitecture.businessService()"/>

</aop:config>

```

Assuming you have a `SystemArchitecture` aspect as described in [the section called “Sharing common pointcut definitions”](#).

Declaring a pointcut inside an aspect is very similar to declaring a top-level pointcut:

```

<aop:config>

    <aop:aspect id="myAspect" ref="aBean">

        <aop:pointcut id="businessService"
            expression="execution(* com.xyz.myapp.service.*.*(..))"/>

        ...

    </aop:aspect>

</aop:config>

```

Much the same way in an @AspectJ aspect, pointcuts declared using the schema based definition style may collect join point context. For example, the following pointcut collects the 'this' object as the join point context and passes it to advice:

```

<aop:config>

    <aop:aspect id="myAspect" ref="aBean">

        <aop:pointcut id="businessService"
            expression="execution(* com.xyz.myapp.service.*.*(..)) && this(service)"/>

        <aop:before pointcut-ref="businessService" method="monitor"/>

        ...

    </aop:aspect>

</aop:config>

```

The advice must be declared to receive the collected join point context by including parameters of the matching names:

```

public void monitor(Object service) {
    ...
}

```

When combining pointcut sub-expressions, '&&' is awkward within an XML document, and so the keywords 'and', 'or' and 'not' can be used in place of '&&', '||' and '!' respectively. For example, the previous pointcut may be better written as:

```

<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>

```

Note that pointcuts defined in this way are referred to by their XML id and cannot be used as named pointcuts to form composite pointcuts. The named pointcut support in the schema based definition style is thus more limited than that offered by the @AspectJ style.

### 7.3.3 Declaring advice

The same five advice kinds are supported as for the @AspectJ style, and they have exactly the same semantics.

#### Before advice

Before advice runs before a matched method execution. It is declared inside an `<aop:aspect>` using the `before` element.

```

<aop:aspect id="beforeExample" ref="aBean">

    <aop:before
        pointcut-ref="dataAccessOperation"
        method="doAccessCheck"/>

    ...

</aop:aspect>

```

Here `dataAccessOperation` is the id of a pointcut defined at the top (`<aop:config>`) level. To define the pointcut inline instead, replace the `pointcut-ref` attribute with a `pointcut` attribute:

```

<aop:aspect id="beforeExample" ref="aBean">

    <aop:before
        pointcut="execution(* com.xyz.myapp.dao.*.*(..))"
        method="doAccessCheck"/>

    ...

</aop:aspect>

```

As we noted in the discussion of the @AspectJ style, using named pointcuts can significantly improve the readability of your code.

The method attribute identifies a method (`doAccessCheck`) that provides the body of the advice. This method must be defined for the bean referenced by the aspect element containing the advice. Before a data access operation is executed (a method execution join point matched by the pointcut expression), the "doAccessCheck" method on the aspect bean will be invoked.

## After returning advice

After returning advice runs when a matched method execution completes normally. It is declared inside an `<aop:aspect>` in the same way as before advice. For example:

```

<aop:aspect id="afterReturningExample" ref="aBean">

    <aop:after-returning
        pointcut-ref="dataAccessOperation"
        method="doAccessCheck"/>

    ...

</aop:aspect>

```

Just as in the @AspectJ style, it is possible to get hold of the return value within the advice body. Use the returning attribute to specify the name of the parameter to which the return value should be passed:

```

<aop:aspect id="afterReturningExample" ref="aBean">

    <aop:after-returning
        pointcut-ref="dataAccessOperation"
        returning="retVal"
        method="doAccessCheck"/>

    ...

</aop:aspect>

```

The `doAccessCheck` method must declare a parameter named `retVal`. The type of this parameter constrains matching in the same way as described for `@AfterReturning`. For example, the method signature may be declared as:

```
public void doAccessCheck(Object retVal) { ... }
```

## After throwing advice

After throwing advice executes when a matched method execution exits by throwing an exception. It is declared inside an `<aop:aspect>` using the `after-throwing` element:

```

<aop:aspect id="afterThrowingExample" ref="aBean">

    <aop:after-throwing
        pointcut-ref="dataAccessOperation"
        method="doRecoveryActions"/>

    ...

</aop:aspect>

```

Just as in the `@AspectJ` style, it is possible to get hold of the thrown exception within the advice body. Use the `throwing` attribute to specify the name of the parameter to which the exception should be passed:

```

<aop:aspect id="afterThrowingExample" ref="aBean">

    <aop:after-throwing
        pointcut-ref="dataAccessOperation"
        throwing="dataAccessEx"
        method="doRecoveryActions"/>

    ...

</aop:aspect>

```

The `doRecoveryActions` method must declare a parameter named `dataAccessEx`. The type of this parameter constrains matching in the same way as described for `@AfterThrowing`. For example, the method signature may be declared as:

```
public void doRecoveryActions(DataAccessException dataAccessEx) { ... }
```

## After (finally) advice

After (finally) advice runs however a matched method execution exits. It is declared using the `after` element:

```

<aop:aspect id="afterFinallyExample" ref="aBean">

    <aop:after
        pointcut-ref="dataAccessOperation"
        method="doReleaseLock"/>

    ...

</aop:aspect>

```

## Around advice

The final kind of advice is around advice. Around advice runs "around" a matched method execution. It has the opportunity to do work both before and after the method executes, and to determine when, how, and even if, the method actually gets to execute at all. Around advice is often used if you need to share state before and after a method execution in a thread-safe manner (starting and stopping a timer for example). Always use the least powerful form of advice that meets your requirements; don't use around advice if simple before advice would do.

Around advice is declared using the `aop:around` element. The first parameter of the advice method must be of type `ProceedingJoinPoint`. Within the body of the advice, calling `proceed()` on the `ProceedingJoinPoint` causes the underlying method to execute. The `proceed` method may also be calling passing in an `Object[]` - the values in the array will be used as the arguments to the method execution when it proceeds. See [the section called “Around advice”](#) for notes on calling `proceed` with an `Object[]`.

```
<aop:aspect id="aroundExample" ref="aBean">

    <aop:around
        pointcut-ref="businessService"
        method="doBasicProfiling"/>

    ...

</aop:aspect>
```

The implementation of the `doBasicProfiling` advice would be exactly the same as in the [@AspectJ example](#) (minus the annotation of course):

```
public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {
    // start stopwatch
    Object retVal = pjp.proceed();
    // stop stopwatch
    return retVal;
}
```

## Advice parameters

The schema based declaration style supports fully typed advice in the same way as described for the [@AspectJ support](#) - by matching pointcut parameters by name against advice method parameters. See [the section called “Advice parameters”](#) for details. If you wish to explicitly specify argument names for the advice methods (not relying on the detection strategies previously described) then this is done using the `arg-names` attribute of the advice element, which is treated in the same manner to the "argNames" attribute in an advice annotation as described in [the section called “Determining argument names”](#). For example:

```
<aop:before
    pointcut="com.xyz.lib.Pointcuts.anyPublicMethod() and @annotation(auditable)"
    method="audit"
    arg-names="auditable"/>
```

The `arg-names` attribute accepts a comma-delimited list of parameter names.

Find below a slightly more involved example of the XSD-based approach that illustrates some around advice used in conjunction with a number of strongly typed parameters.

```
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);
    }
}
```

Next up is the aspect. Notice the fact that the `profile(..)` method accepts a number of strongly-typed parameters, the first of which happens to be the join point used to proceed with the method call: the presence of this parameter is an indication that the `profile(..)` is to be used as `around` advice:

```
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());
        }
    }
}
```

Finally, here is the XML configuration that is required to effect the execution of the above advice for a particular join point:

```

<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>

```

If we had the following driver script, we would get output something like this on standard output:

```

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)
```

## Advice ordering

When multiple advice needs to execute at the same join point (executing method) the ordering rules are as described in [the section called “Advice ordering”](#). The precedence between aspects is determined by either adding the `order` annotation to the bean backing the aspect or by having the bean implement the `Ordered` interface.

### 7.3.4 Introductions

Introductions (known as inter-type declarations in AspectJ) enable an aspect to declare that advised objects implement a given interface, and to provide an implementation of that interface on behalf of those objects.

An introduction is made using the `aop:declare-parents` element inside an `aop:aspect`. This element is used to declare that matching types have a new parent (hence the name). For example, given an interface `UsageTracked`, and an implementation of that interface `DefaultUsageTracked`, the following aspect declares that all implementors of service interfaces also implement the `UsageTracked` interface. (In order to expose statistics via JMX for example.)

```
<aop:aspect id="usageTrackerAspect" ref="usageTracking">

    <aop:declare-parents
        types-matching="com.xyz.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>
```

The class backing the `usageTracking` bean would contain the method:

```
public void recordUsage(UsageTracked usageTracked) {  
    usageTracked.incrementUseCount();  
}
```

The interface to be implemented is determined by `implement-interface` attribute. The value of the `types-matching` attribute is an AspectJ type pattern :- any bean of a matching type will implement the `UsageTracked` interface. Note that in the before advice of the above example, service beans can be directly used as implementations of the `UsageTracked` interface. If accessing a bean programmatically you would write the following:

```
UsageTracked usageTracked = (UsageTracked) context.getBean("myService");
```

### 7.3.5 Aspect instantiation models

The only supported instantiation model for schema-defined aspects is the singleton model. Other instantiation models may be supported in future releases.

### 7.3.6 Advisors

The concept of "advisors" is brought forward from the AOP support defined in Spring 1.2 and does not have a direct equivalent in AspectJ. An advisor is like a small self-contained aspect that has a single piece of advice. The advice itself is represented by a bean, and must implement one of the advice interfaces described in [Section 8.3.2, “Advice types in Spring”](#). Advisors can take advantage of AspectJ pointcut expressions though.

Spring supports the advisor concept with the `<aop:advisor>` element. You will most commonly see it used in conjunction with transactional advice, which also has its own namespace support in Spring. Here's how it looks:

```

<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>

```

As well as the `pointcut-ref` attribute used in the above example, you can also use the `pointcut` attribute to define a pointcut expression inline.

To define the precedence of an advisor so that the advice can participate in ordering, use the `order` attribute to define the `ordered` value of the advisor.

### 7.3.7 Example

Let's see how the concurrent locking failure retry example from [Section 7.2.7, “Example”](#) looks when rewritten using the schema support.

The execution of business services can sometimes fail due to concurrency issues (for example, deadlock loser). If the operation is retried, it is quite likely it will succeed next time round. For business services where it is appropriate to retry in such conditions (idempotent operations that don't need to go back to the user for conflict resolution), we'd like to transparently retry the operation to avoid the client seeing a

`PessimisticLockingFailureException`. This is a requirement that clearly cuts across multiple services in the service layer, and hence is ideal for implementing via an aspect.

Because we want to retry the operation, we'll need to use around advice so that we can call proceed multiple times. Here's how the basic aspect implementation looks (it's just a regular Java class using the schema support):

```

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;
    }

}

```

Note that the aspect implements the `Ordered` interface so we can set the precedence of the aspect higher than the transaction advice (we want a fresh transaction each time we retry). The `maxRetries` and `order` properties will both be configured by Spring. The main action happens in the `doConcurrentOperation` around advice method. We try to proceed, and if we fail with a `PessimisticLockingFailureException` we simply try again unless we have exhausted all of our retry attempts.

	<input type="checkbox"/>
	This class is identical to the one used in the @AspectJ example, but with the annotations removed.

The corresponding Spring configuration is:

```

<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>

```

Notice that for the time being we assume that all business services are idempotent. If this is not the case we can refine the aspect so that it only retries genuinely idempotent operations, by introducing an `Idempotent` annotation:

```

@Retention(RetentionPolicy.RUNTIME)
public @interface Idempotent {
    // marker annotation
}

```

and using the annotation to annotate the implementation of service operations. The change to the aspect to retry only idempotent operations simply involves refining the pointcut expression so that only `@Idempotent` operations match:

```

<aop:pointcut id="idempotentOperation"
    expression="execution(* com.xyz.myapp.service.*.*(..)) and
    @annotation(com.xyz.myapp.service.Idempotent)"/>

```

## 7.4 Choosing which AOP declaration style to use

Once you have decided that an aspect is the best approach for implementing a given requirement, how do you decide between using Spring AOP or AspectJ, and between the Aspect language (code) style, @AspectJ annotation style, or the Spring XML style? These decisions are influenced by a number of factors including application requirements, development tools, and team familiarity with AOP.

### 7.4.1 Spring AOP or full AspectJ?

Use the simplest thing that can work. Spring AOP is simpler than using full AspectJ as there is no requirement to introduce the AspectJ compiler / weaver into your development and build processes. If you only need to advise the execution of operations on Spring beans, then Spring AOP is the right choice. If you need to advise objects not managed by the Spring container (such as domain objects typically), then you will need to use AspectJ. You will also need to use AspectJ if you wish to advise join points other than simple method executions (for example, field get or set join points, and so on).

When using AspectJ, you have the choice of the AspectJ language syntax (also known as the "code style") or the @AspectJ annotation style. Clearly, if you are not using Java 5+ then the choice has been made for you... use the code style. If aspects play a large role in your design, and you are able to use the [AspectJ Development Tools \(AJDT\)](#) plugin for Eclipse, then the AspectJ language syntax is the preferred option: it is cleaner and simpler because the language was purposefully designed for writing aspects. If you are not using Eclipse, or have only a few aspects that do not play a major role in your application, then you may want to consider using the @AspectJ style and sticking with a regular Java compilation in your IDE, and adding an aspect weaving phase to your build script.

### 7.4.2 @AspectJ or XML for Spring AOP?

If you have chosen to use Spring AOP, then you have a choice of @AspectJ or XML style. There are various tradeoffs to consider.

The XML style will be most familiar to existing Spring users and it is backed by genuine POJOs. When using AOP as a tool to configure enterprise services then XML can be a good choice (a good test is whether you consider the pointcut expression to be a part of your configuration you might want to change independently). With the XML style arguably it is clearer from your configuration what aspects are present in the system.

The XML style has two disadvantages. Firstly it does not fully encapsulate the implementation of the requirement it addresses in a single place. The DRY principle says that there should be a single, unambiguous, authoritative representation of any piece of knowledge within a system. When using the XML style, the knowledge of *how* a requirement is implemented is split across the declaration of the backing bean class, and the XML in the configuration file. When using the @AspectJ style there is a single module - the aspect - in which this information is encapsulated. Secondly, the XML style is slightly more limited in what it can express than the @AspectJ style: only the "singleton" aspect instantiation model is supported, and it is not possible to combine named pointcuts declared in XML. For example, in the @AspectJ style you can write something like:

```
@Pointcut(execution(* get*()))
public void propertyAccess() {}

@Pointcut(execution(org.xyz.Account+ *(..)))
public void operationReturningAnAccount() {}

@Pointcut(propertyAccess() && operationReturningAnAccount())
public void accountPropertyAccess() {}
```

In the XML style I can declare the first two pointcuts:

```
<aop:pointcut id="propertyAccess"
    expression="execution(* get*())"/>

<aop:pointcut id="operationReturningAnAccount"
    expression="execution(org.xyz.Account+ *(..))"/>
```

The downside of the XML approach is that you cannot define the `accountPropertyAccess` pointcut by combining these definitions.

The @AspectJ style supports additional instantiation models, and richer pointcut composition. It has the advantage of keeping the aspect as a modular unit. It also has the advantage the @AspectJ aspects can be understood (and thus consumed) both by Spring AOP and by AspectJ - so if you later decide you need the capabilities of AspectJ to implement additional requirements then it is very easy to migrate to an AspectJ-based approach. On balance the Spring team prefer the @AspectJ style whenever you have aspects that do more than simple "configuration" of enterprise services.

## 7.5 Mixing aspect types

It is perfectly possible to mix @AspectJ style aspects using the autoproxying support, schema-defined `<aop:aspect>` aspects, `<aop:advisor>` declared advisors and even proxies and interceptors defined using the Spring 1.2 style in the same configuration. All of these are implemented using the same underlying support mechanism and will co-exist without any difficulty.

## 7.6 Proxying mechanisms

Spring AOP uses either JDK dynamic proxies or CGLIB to create the proxy for a given target object. (JDK dynamic proxies are preferred whenever you have a choice).

If the target object to be proxied implements at least one interface then a JDK dynamic proxy will be used. All of the interfaces implemented by the target type will be proxied. If the target object does not implement any interfaces then a CGLIB proxy will be created.

If you want to force the use of CGLIB proxying (for example, to proxy every method defined for the target object, not just those implemented by its interfaces) you can do so. However, there are some issues to consider:

- `final` methods cannot be advised, as they cannot be overridden.
- As of Spring 3.2, it is no longer necessary to add CGLIB to your project classpath, as CGLIB classes are repackaged under `org.springframework` and included directly in the `spring-core` JAR. This means that CGLIB-based proxy support 'just works' in the same way that JDK dynamic proxies always have.
- As of Spring 4.0, the constructor of your proxied object will NOT be called twice anymore since the CGLIB proxy instance will be created via Objenesis. Only if your JVM does not allow for constructor bypassing, you might see double invocations and corresponding debug log entries from Spring's AOP support.

To force the use of CGLIB proxies set the value of the `proxy-target-class` attribute of the `<aop:config>` element to true:

```
<aop:config proxy-target-class="true">
    <!-- other beans defined here... -->
</aop:config>
```

To force CGLIB proxying when using the `@AspectJ` autoproxy support, set the '`proxy-target-class`' attribute of the `<aop:aspectj-autoproxy>` element to `true` :

```
<aop:aspectj-autoproxy proxy-target-class="true"/>
```



Multiple `<aop:config>` sections are collapsed into a single unified auto-proxy creator at runtime, which applies the *strongest* proxy settings that any of the `<aop:config>` sections (typically from different XML bean definition files) specified. This also applies to the `<tx:annotation-driven>` and `<aop:aspectj-autoproxy>` elements. To be clear: using `proxy-target-class="true"` on `<tx:annotation-driven>`, `<aop:aspectj-autoproxy>` or `<aop:config>` elements will force the use of CGLIB proxies for all three of them.

## 7.6.1 Understanding AOP proxies

Spring AOP is *proxy-based*. It is vitally important that you grasp the semantics of what that last statement actually means before you write your own aspects or use any of the Spring AOP-based aspects supplied with the Spring Framework.

Consider first the scenario where you have a plain-vanilla, unproxied, nothing-special-about-it, straight object reference, as illustrated by the following code snippet.

```
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...
    }
}
```

If you invoke a method on an object reference, the method is invoked *directly* on that object reference, as can be seen below.



```
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();
    }
}
```

Things change slightly when the reference that client code has is a proxy. Consider the following diagram and code snippet.



```
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();  
    }  
}
```

The key thing to understand here is that the client code inside the `main(...)` of the `Main` class *has a reference to the proxy*. This means that method calls on that object reference will be calls on the proxy, and as such the proxy will be able to delegate to all of the interceptors (advice) that are relevant to that particular method call. However, once the call has finally reached the target object, the `SimplePojo` reference in this case, any method calls that it may make on itself, such as `this.bar()` or `this.foo()`, are going to be invoked against the *this* reference, and *not* the proxy. This has important implications. It means that self-invocation is *not* going to result in the advice associated with a method invocation getting a chance to execute.

Okay, so what is to be done about this? The best approach (the term best is used loosely here) is to refactor your code such that the self-invocation does not happen. For sure, this does entail some work on your part, but it is the best, least-invasive approach. The next approach is absolutely horrendous, and I am almost reticent to point it out precisely because it is so horrendous. You can (choke!) totally tie the logic within your class to Spring AOP by doing this:

```
public class SimplePojo implements Pojo {

    public void foo() {
        // this works, but... gah!
        ((Pojo) AopContext.currentProxy()).bar();
    }

    public void bar() {
        // some logic...
    }
}
```

This totally couples your code to Spring AOP, *and* it makes the class itself aware of the fact that it is being used in an AOP context, which flies in the face of AOP. It also requires some additional configuration when the proxy is being created:

```
public class Main {

    public static void main(String[] args) {

        ProxyFactory factory = new ProxyFactory(new SimplePojo());
        factory.addInterface(Pojo.class);
        factory.addAdvice(new RetryAdvice());
        factory.setExposeProxy(true);

        Pojo pojo = (Pojo) factory.getProxy();

        // this is a method call on the proxy!
        pojo.foo();
    }
}
```

Finally, it must be noted that AspectJ does not have this self-invocation issue because it is not a proxy-based AOP framework.

## 7.7 Programmatic creation of @AspectJ Proxies

In addition to declaring aspects in your configuration using either `<aop:config>` or `<aop:aspectj-autoproxy>`, it is also possible programmatically to create proxies that advise target objects. For the full details of Spring's AOP API, see the next chapter. Here we want to focus on the ability to automatically create proxies using @AspectJ aspects.

The class `org.springframework.aop.aspectj.annotation.AspectJProxyFactory` can be used to create a proxy for a target object that is advised by one or more @AspectJ aspects. Basic usage for this class is very simple, as illustrated below. See the javadocs for full information.

```
// 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();
```

## 7.8 Using AspectJ with Spring applications

Everything we've covered so far in this chapter is pure Spring AOP. In this section, we're going to look at how you can use the AspectJ compiler/weaver instead of, or in addition to, Spring AOP if your needs go beyond the facilities offered by Spring AOP alone.

Spring ships with a small AspectJ aspect library, which is available standalone in your distribution as `spring-aspects.jar`; you'll need to add this to your classpath in order to use the aspects in it. [Section 7.8.1, “Using AspectJ to dependency inject domain objects with Spring”](#) and [Section 7.8.2, “Other Spring aspects for AspectJ”](#) discuss the content of this library and how you can use it. [Section 7.8.3, “Configuring AspectJ aspects using Spring IoC”](#) discusses how to dependency inject AspectJ aspects that are woven using the AspectJ compiler. Finally, [Section 7.8.4, “Load-time weaving with AspectJ in the Spring Framework”](#) provides an introduction to load-time weaving for Spring applications using AspectJ.

### 7.8.1 Using AspectJ to dependency inject domain objects with Spring

The Spring container instantiates and configures beans defined in your application context. It is also possible to ask a bean factory to configure a *pre-existing* object given the name of a bean definition containing the configuration to be applied. The `spring-aspects.jar` contains an annotation-driven aspect that exploits this capability to allow dependency injection of *any object*. The support is intended to be used for objects created *outside of the control of any container*. Domain objects often fall into this category because they are often created programmatically using the `new` operator, or by an ORM tool as a result of a database query.

The `@Configurable` annotation marks a class as eligible for Spring-driven configuration. In the simplest case it can be used just as a marker annotation:

```
package com.xyz.myapp.domain;

import org.springframework.beans.factory.annotation.Configurable;

@Configuration
public class Account {
    // ...
}
```

When used as a marker interface in this way, Spring will configure new instances of the annotated type (`Account` in this case) using a bean definition (typically prototype-scoped) with the same name as the fully-qualified type name (`com.xyz.myapp.domain.Account`). Since the default name for a bean is the fully-qualified name of its type, a convenient way to declare the prototype definition is simply to omit the `id` attribute:

```
<bean class="com.xyz.myapp.domain.Account" scope="prototype">
    <property name="fundsTransferService" ref="fundsTransferService"/>
</bean>
```

If you want to explicitly specify the name of the prototype bean definition to use, you can do so directly in the annotation:

```
package com.xyz.myapp.domain;

import org.springframework.beans.factory.annotation.Configurable;

@Configuration("account")
public class Account {
    // ...
}
```

Spring will now look for a bean definition named "account" and use that as the definition to configure new `Account` instances.

You can also use autowiring to avoid having to specify a dedicated bean definition at all. To have Spring apply autowiring use the `autowire` property of the `@Configurable` annotation: specify either `@Configurable(autowire=Autowire.BY_TYPE)` or `@Configurable(autowire=Autowire.BY_NAME)` for autowiring by type or by name respectively. As an alternative, as of Spring 2.5 it is preferable to specify explicit, annotation-driven dependency injection for your `@Configurable` beans by using `@Autowired` or `@Inject` at the field or method level (see [Section 3.9, “Annotation-based container configuration”](#) for further details).

Finally you can enable Spring dependency checking for the object references in the newly created and configured object by using the `dependencyCheck` attribute (for example:

```
@Configurable(autowire=Autowire.BY_NAME, dependencyCheck=true) ). If this attribute is set to true, then Spring will validate after configuration that all properties (which are not primitives or collections) have been set.
```

Using the annotation on its own does nothing of course. It is the `AnnotationBeanConfigurerAspect` in `spring-aspects.jar` that acts on the presence of the annotation. In essence the aspect says "after returning from the initialization of a new object

of a type annotated with `@Configurable`, configure the newly created object using Spring in accordance with the properties of the annotation". In this context, *initialization* refers to newly instantiated objects (e.g., objects instantiated with the `new` operator) as well as to `Serializable` objects that are undergoing deserialization (e.g., via `readResolve()`).

One of the key phrases in the above paragraph is '*in essence*'. For most cases, the exact semantics of '*after returning from the initialization of a new object*' will be fine... in this context, '*after initialization*' means that the dependencies will be injected *after* the object has been constructed - this means that the dependencies will not be available for use in the constructor bodies of the class. If you want the dependencies to be injected *before* the constructor bodies execute, and thus be available for use in the body of the constructors, then you need to define this on the `@Configurable` declaration like so: `@Configurable(preConstruction=true)` You can find out more information about the language semantics of the various pointcut types in AspectJ [in this appendix](#) of the [AspectJ Programming Guide](#).

For this to work the annotated types must be woven with the AspectJ weaver - you can either use a build-time Ant or Maven task to do this (see for example the [AspectJ Development Environment Guide](#)) or load-time weaving (see [Section 7.8.4, “Load-time weaving with AspectJ in the Spring Framework”](#)). The `AnnotationBeanConfigurerAspect` itself needs configuring by Spring (in order to obtain a reference to the bean factory that is to be used to configure new objects). If you are using Java based configuration simply add `@EnableSpringConfigured` to any `@Configuration` class.

```
@Configuration
@EnableSpringConfigured
public class AppConfig {

}
```

If you prefer XML based configuration, the Spring `context` namespace defines a convenient `context:spring-configured` element:

```
<context:spring-configured/>
```

Instances of `@Configurable` objects created *before* the aspect has been configured will result in a message being issued to the debug log and no configuration of the object taking place. An example might be a bean in the Spring configuration that creates domain objects when it is initialized by Spring. In this case you can use the "depends-on" bean attribute to manually specify that the bean depends on the configuration aspect.

```

<bean id="myService"
      class="com.xzy.myapp.service.MyService"
      depends-on="org.springframework.beans.factory.aspectj.AnnotationBeanConfigurer
Aspect">

<! -- . . . -->

</bean>

```



Do not activate `@Configurable` processing through the bean configurer aspect unless you really mean to rely on its semantics at runtime. In particular, make sure that you do not use `@Configurable` on bean classes which are registered as regular Spring beans with the container: You would get double initialization otherwise, once through the container and once through the aspect.

## Unit testing @Configurable objects

One of the goals of the `@Configurable` support is to enable independent unit testing of domain objects without the difficulties associated with hard-coded lookups. If `@Configurable` types have not been woven by AspectJ then the annotation has no affect during unit testing, and you can simply set mock or stub property references in the object under test and proceed as normal. If `@Configurable` types *have* been woven by AspectJ then you can still unit test outside of the container as normal, but you will see a warning message each time that you construct an `@Configurable` object indicating that it has not been configured by Spring.

## Working with multiple application contexts

The `AnnotationBeanConfigurerAspect` used to implement the `@Configurable` support is an AspectJ singleton aspect. The scope of a singleton aspect is the same as the scope of `static` members, that is to say there is one aspect instance per classloader that defines the type. This means that if you define multiple application contexts within the same classloader hierarchy you need to consider where to define the `@EnableSpringConfigured` bean and where to place `spring-aspects.jar` on the classpath.

Consider a typical Spring web-app configuration with a shared parent application context defining common business services and everything needed to support them, and one child application context per servlet containing definitions particular to that servlet. All of these contexts will co-exist within the same classloader hierarchy, and so the

`AnnotationBeanConfigurerAspect` can only hold a reference to one of them. In this case we recommend defining the `@EnableSpringConfigured` bean in the shared (parent) application

context: this defines the services that you are likely to want to inject into domain objects. A consequence is that you cannot configure domain objects with references to beans defined in the child (servlet-specific) contexts using the `@Configurable` mechanism (probably not something you want to do anyway!).

When deploying multiple web-apps within the same container, ensure that each web-application loads the types in `spring-aspects.jar` using its own classloader (for example, by placing `spring-aspects.jar` in '`WEB-INF/lib`' ). If `spring-aspects.jar` is only added to the container wide classpath (and hence loaded by the shared parent classloader), all web applications will share the same aspect instance which is probably not what you want.

## 7.8.2 Other Spring aspects for AspectJ

In addition to the `@Configurable` aspect, `spring-aspects.jar` contains an AspectJ aspect that can be used to drive Spring's transaction management for types and methods annotated with the `@Transactional` annotation. This is primarily intended for users who want to use the Spring Framework's transaction support outside of the Spring container.

The aspect that interprets `@Transactional` annotations is the `AnnotationTransactionAspect`. When using this aspect, you must annotate the *implementation* class (and/or methods within that class), *not* the interface (if any) that the class implements. AspectJ follows Java's rule that annotations on interfaces are *not inherited*.

A `@Transactional` annotation on a class specifies the default transaction semantics for the execution of any *public* operation in the class.

A `@Transactional` annotation on a method within the class overrides the default transaction semantics given by the class annotation (if present). Methods of any visibility may be annotated, including private methods. Annotating non-public methods directly is the only way to get transaction demarcation for the execution of such methods.



Since Spring Framework 4.2, `spring-aspects` provides a similar aspect that offers the exact same features for the standard `javax.transaction.Transactional` annotation. Check `JtaAnnotationTransactionAspect` for more details.

For AspectJ programmers that want to use the Spring configuration and transaction management support but don't want to (or cannot) use annotations, `spring-aspects.jar` also contains `abstract` aspects you can extend to provide your own pointcut definitions. See the sources for the `AbstractBeanConfigurerAspect` and `AbstractTransactionAspect` aspects for more information. As an example, the following excerpt shows how you could write an aspect to configure all instances of objects defined in the domain model using prototype bean definitions that match the fully-qualified class names:

```

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);

}

```

### 7.8.3 Configuring AspectJ aspects using Spring IoC

When using AspectJ aspects with Spring applications, it is natural to both want and expect to be able to configure such aspects using Spring. The AspectJ runtime itself is responsible for aspect creation, and the means of configuring the AspectJ created aspects via Spring depends on the AspectJ instantiation model (the `per-xxx` clause) used by the aspect.

The majority of AspectJ aspects are *singleton* aspects. Configuration of these aspects is very easy: simply create a bean definition referencing the aspect type as normal, and include the bean attribute `'factory-method="aspectOf"`. This ensures that Spring obtains the aspect instance by asking AspectJ for it rather than trying to create an instance itself. For example:

```

<bean id="profiler" class="com.xyz.profiler.Profiler"
      factory-method="aspectOf">

    <property name="profilingStrategy" ref="jamonProfilingStrategy"/>
</bean>

```

Non-singleton aspects are harder to configure: however it is possible to do so by creating prototype bean definitions and using the `@Configurable` support from `spring-aspects.jar` to configure the aspect instances once they have been created by the AspectJ runtime.

If you have some `@AspectJ` aspects that you want to weave with AspectJ (for example, using load-time weaving for domain model types) and other `@AspectJ` aspects that you want to use with Spring AOP, and these aspects are all configured using Spring, then you will need to tell the Spring AOP `@AspectJ` autoproxying support which exact subset of the `@AspectJ` aspects defined in the configuration should be used for autoproxying. You can do this by using one or more `<include/>` elements inside the `<aop:aspectj-autoproxy/>`

declaration. Each `<include/>` element specifies a name pattern, and only beans with names matched by at least one of the patterns will be used for Spring AOP autoproxy configuration:

```
<aop:aspectj-autoproxy>
  <aop:include name="thisBean"/>
  <aop:include name="thatBean"/>
</aop:aspectj-autoproxy>
```

 Do not be misled by the name of the `<aop:aspectj-autoproxy/>` element: using it will result in the creation of *Spring AOP proxies*. The `@AspectJ` style of aspect declaration is just being used here, but the AspectJ runtime is *not* involved.

## 7.8.4 Load-time weaving with AspectJ in the Spring Framework

Load-time weaving (LTW) refers to the process of weaving AspectJ aspects into an application's class files as they are being loaded into the Java virtual machine (JVM). The focus of this section is on configuring and using LTW in the specific context of the Spring Framework: this section is not an introduction to LTW though. For full details on the specifics of LTW and configuring LTW with just AspectJ (with Spring not being involved at all), see the [LTW section of the AspectJ Development Environment Guide](#).

The value-add that the Spring Framework brings to AspectJ LTW is in enabling much finer-grained control over the weaving process. 'Vanilla' AspectJ LTW is effected using a Java (5+) agent, which is switched on by specifying a VM argument when starting up a JVM. It is thus a JVM-wide setting, which may be fine in some situations, but often is a little too coarse. Spring-enabled LTW enables you to switch on LTW on a *per-ClassLoader* basis, which obviously is more fine-grained and which can make more sense in a 'single-JVM-multiple-application' environment (such as is found in a typical application server environment).

Further, [in certain environments](#), this support enables load-time weaving *without making any modifications to the application server's launch script* that will be needed to add `-javaagent:path/to/aspectjweaver.jar` or (as we describe later in this section) `-javaagent:path/to/org.springframework.instrument-{version}.jar` (previously named `spring-agent.jar`). Developers simply modify one or more files that form the application context to enable load-time weaving instead of relying on administrators who typically are in charge of the deployment configuration such as the launch script.

Now that the sales pitch is over, let us first walk through a quick example of AspectJ LTW using Spring, followed by detailed specifics about elements introduced in the following example. For a complete example, please see the [Petclinic sample application](#).

## A first example

Let us assume that you are an application developer who has been tasked with diagnosing the cause of some performance problems in a system. Rather than break out a profiling tool, what we are going to do is switch on a simple profiling aspect that will enable us to very quickly get some performance metrics, so that we can then apply a finer-grained profiling tool to that specific area immediately afterwards.



The example presented here uses XML style configuration, it is also possible to configure and use [@AspectJ](#) with [Java Configuration](#). Specifically the `@EnableLoadTimeWeaving` annotation can be used as an alternative to `<context:load-time-weaver/>` (see [below](#) for details).

Here is the profiling aspect. Nothing too fancy, just a quick-and-dirty time-based profiler, using the [@AspectJ](#)-style of aspect declaration.

```

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(){}
}

```

We will also need to create an `META-INF/aop.xml` file, to inform the AspectJ weaver that we want to weave our `ProfilingAspect` into our classes. This file convention, namely the presence of a file (or files) on the Java classpath called `META-INF/aop.xml` is standard 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>

```

Now to the Spring-specific portion of the configuration. We need to configure a `LoadTimeWeaver` (all explained later, just take it on trust for now). This load-time weaver is the essential component responsible for weaving the aspect configuration in one or more `META-INF/aop.xml` files into the classes in your application. The good thing is that it does not require a lot of configuration, as can be seen below (there are some more options that you can specify, but these are detailed later).

```
<?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>
```

Now that all the required artifacts are in place - the aspect, the `META-INF/aop.xml` file, and the Spring configuration -, let us create a simple driver class with a `main(..)` method to demonstrate the LTW in action.

```
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();
    }
}
```

There is one last thing to do. The introduction to this section did say that one could switch on LTW selectively on a per- `ClassLoader` basis with Spring, and this is true. However, just for this example, we are going to use a Java agent (supplied with Spring) to switch on the LTW. This is the command line we will use to run the above `Main` class:

```
java -javaagent:C:/projects/foo/lib/global/spring-instrument.jar foo.Main
```

The `-javaagent` is a flag for specifying and enabling [agents to instrument programs running on the JVM](#). The Spring Framework ships with such an agent, the `InstrumentationSavingAgent`, which is packaged in the `spring-instrument.jar` that was supplied as the value of the `-javaagent` argument in the above example.

The output from the execution of the `Main` program will look something like that below. (I have introduced a `Thread.sleep(..)` statement into the `calculateEntitlement()` implementation so that the profiler actually captures something other than 0 milliseconds - the `01234` milliseconds is *not* an overhead introduced by the AOP :))

```
Calculating entitlement

Stopwatch 'ProfilingAspect': running time (millis) = 1234
-----
ms      %      Task name
-----
01234  100%  calculateEntitlement
```

Since this LTW is effected using full-blown AspectJ, we are not just limited to advising Spring beans; the following slight variation on the `Main` program will yield the same result.

```
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();
    }
}
```

Notice how in the above program we are simply bootstrapping the Spring container, and then creating a new instance of the `StubEntitlementCalculationService` totally outside the context of Spring... the profiling advice still gets woven in.

The example admittedly is simplistic... however the basics of the LTW support in Spring have all been introduced in the above example, and the rest of this section will explain the 'why' behind each bit of configuration and usage in detail.



The `ProfilingAspect` used in this example may be basic, but it is quite useful. It is a nice example of a development-time aspect that developers can use during development (of course), and then quite easily exclude from builds of the application being deployed into UAT or production.

## Aspects

The aspects that you use in LTW have to be AspectJ aspects. They can be written in either the AspectJ language itself or you can write your aspects in the @AspectJ-style. It means that your aspects are then both valid AspectJ *and* Spring AOP aspects. Furthermore, the compiled aspect classes need to be available on the classpath.

## 'META-INF/aop.xml'

The AspectJ LTW infrastructure is configured using one or more `META-INF/aop.xml` files, that are on the Java classpath (either directly, or more typically in jar files).

The structure and contents of this file is detailed in the main AspectJ reference documentation, and the interested reader is [referred to that resource](#). (I appreciate that this section is brief, but the `aop.xml` file is 100% AspectJ - there is no Spring-specific information or semantics that apply to it, and so there is no extra value that I can contribute either as a result), so rather than rehash the quite satisfactory section that the AspectJ developers wrote, I am just directing you there.)

## Required libraries (JARS)

At a minimum you will need the following libraries to use the Spring Framework's support for AspectJ LTW:

- `spring-aop.jar` (version 2.5 or later, plus all mandatory dependencies)
- `aspectjweaver.jar` (version 1.6.8 or later)

If you are using the [Spring-provided agent to enable instrumentation](#), you will also need:

- `spring-instrument.jar`

## Spring configuration

The key component in Spring's LTW support is the `LoadTimeWeaver` interface (in the `org.springframework.instrument.classloading` package), and the numerous implementations of it that ship with the Spring distribution. A `LoadTimeWeaver` is responsible for adding one or more `java.lang.instrument.ClassFileTransformers` to a `ClassLoader` at runtime, which opens the door to all manner of interesting applications, one of which happens to be the LTW of aspects.



If you are unfamiliar with the idea of runtime class file transformation, you are encouraged to read the javadoc API documentation for the `java.lang.instrument` package before continuing. This is not a huge chore because there is - rather annoyingly - precious little documentation there... the key interfaces and classes will at least be laid out in front of you for reference as you read through this section.

Configuring a `LoadTimeWeaver` for a particular `ApplicationContext` can be as easy as adding one line. (Please note that you almost certainly will need to be using an `ApplicationContext` as your Spring container - typically a `BeanFactory` will not be enough because the LTW support makes use of `BeanFactoryPostProcessors`.)

To enable the Spring Framework's LTW support, you need to configure a `LoadTimeWeaver`, which typically is done using the `@EnableLoadTimeWeaving` annotation.

```
@Configuration  
@EnableLoadTimeWeaving  
public class AppConfig {  
}
```

Alternatively, if you prefer XML based configuration, use the `<context:load-time-weaver/>` element. Note that the element is defined in the `context` namespace.

```

<?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>

```

The above configuration will define and register a number of LTW-specific infrastructure beans for you automatically, such as a `LoadTimeWeaver` and an `AspectJWeavingEnabler`. The default `LoadTimeWeaver` is the `DefaultContextLoadTimeWeaver` class, which attempts to decorate an automatically detected `LoadTimeWeaver`: the exact type of `LoadTimeWeaver` that will be 'automatically detected' is dependent upon your runtime environment (summarized in the following table).

**Table 7.1. DefaultContextLoadTimeWeaver LoadTimeWeavers**

Runtime Environment	LoadTimeWeaver implementation
Running in Oracle's <a href="#">WebLogic</a>	<code>WebLogicLoadTimeWeaver</code>
Running in Oracle's <a href="#">GlassFish</a>	<code>GlassFishLoadTimeWeaver</code>
Running in <a href="#">Apache Tomcat</a>	<code>TomcatLoadTimeWeaver</code>
Running in Red Hat's <a href="#">JBoss AS</a> or <a href="#">WildFly</a>	<code>JBossLoadTimeWeaver</code>
Running in IBM's <a href="#">WebSphere</a>	<code>WebSphereLoadTimeWeaver</code>
JVM started with Spring InstrumentationSavingAgent ( <code>java -javaagent:path/to/spring-instrument.jar</code> )	<code>InstrumentationLoadTimeWeaver</code>
Fallback, expecting the underlying ClassLoader to follow common conventions (e.g. applicable to <code>TomcatInstrumentableClassLoader</code> and <a href="#">Resin</a> )	<code>ReflectiveLoadTimeWeaver</code>

Note that these are just the `LoadTimeWeavers` that are autodetected when using the `DefaultContextLoadTimeWeaver`: it is of course possible to specify exactly which `LoadTimeWeaver` implementation that you wish to use.

To specify a specific `LoadTimeWeaver` with Java configuration implement the `LoadTimeWeavingConfigurer` interface and override the `getLoadTimeWeaver()` method:

```

@Configuration
@EnableLoadTimeWeaving
public class AppConfig implements LoadTimeWeavingConfigurer {

    @Override
    public LoadTimeWeaver getLoadTimeWeaver() {
        return new ReflectiveLoadTimeWeaver();
    }
}

```

If you are using XML based configuration you can specify the fully-qualified classname as the value of the `weaver-class` attribute on the `<context:load-time-weaver/>` element:

```

<?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.ReflectiveLoadTi
        meWeaver"/>

</beans>

```

The `LoadTimeWeaver` that is defined and registered by the configuration can be later retrieved from the Spring container using the well-known name `loadTimeweaver`. Remember that the `LoadTimeWeaver` exists just as a mechanism for Spring's LTW infrastructure to add one or more `ClassFileTransformers`. The actual `ClassFileTransformer` that does the LTW is the `ClassPreProcessorAgentAdapter` (from the `org.aspectj.weaver.loadtime` package) class. See the class-level javadocs of the `ClassPreProcessorAgentAdapter` class for further details, because the specifics of how the weaving is actually effected is beyond the scope of this section.

There is one final attribute of the configuration left to discuss: the `aspectjWeaving` attribute (or `aspectj-weaving` if you are using XML). This is a simple attribute that controls whether LTW is enabled or not; it is as simple as that. It accepts one of three possible values, summarized below, with the default value being `autodetect` if the attribute is not present.

**Table 7.2. AspectJ weaving attribute values**

Annotation Value	XML Value	Explanation
ENABLED	on	AspectJ weaving is on, and aspects will be woven at load-time as appropriate.
DISABLED	off	LTW is off... no aspect will be woven at load-time.
AUTODETECT	autodetect	If the Spring LTW infrastructure can find at least one <code>META-INF/aop.xml</code> file, then AspectJ weaving is on, else it is off. This is the default value.

## Environment-specific configuration

This last section contains any additional settings and configuration that you will need when using Spring's LTW support in environments such as application servers and web containers.

### Tomcat

Historically, [Apache Tomcat](#)'s default class loader did not support class transformation which is why Spring provides an enhanced implementation that addresses this need. Named `TomcatInstrumentableClassLoader`, the loader works on Tomcat 6.0 and above.

□

Do not define `TomcatInstrumentableClassLoader` anymore on Tomcat 8.0 and higher. Instead, let Spring automatically use Tomcat's new native `InstrumentableClassLoader` facility through the `TomcatLoadTimeWeaver` strategy.

If you still need to use `TomcatInstrumentableClassLoader`, it can be registered individually for each web application as follows:

- Copy `org.springframework.instrument.tomcat.jar` into `$CATALINA_HOME/lib`, where `$CATALINA_HOME` represents the root of the Tomcat installation)
- Instruct Tomcat to use the custom class loader (instead of the default) by editing the web application context file:

```
<Context path="/myWebApp" docBase="/my/webApp/location">
    <Loader
        loaderClass="org.springframework.instrument.classloading.tomcat.TomcatInstrumentableClassLoader"/>
</Context>
```

Apache Tomcat (6.0+) supports several context locations:

- server configuration file - `$CATALINA_HOME/conf/server.xml`

- default context configuration - `$CATALINA_HOME/conf/context.xml` - that affects all deployed web applications
- per-web application configuration which can be deployed either on the server-side at `$CATALINA_HOME/conf/[enginename]/[hostname]/[webapp]-context.xml` or embedded inside the web-app archive at `META-INF/context.xml`

For efficiency, the embedded per-web-app configuration style is recommended because it will impact only applications that use the custom class loader and does not require any changes to the server configuration. See the Tomcat 6.0.x [documentation](#) for more details about available context locations.

Alternatively, consider the use of the Spring-provided generic VM agent, to be specified in Tomcat's launch script (see above). This will make instrumentation available to all deployed web applications, no matter what ClassLoader they happen to run on.

### **WebLogic, WebSphere, Resin, GlassFish, JBoss**

Recent versions of WebLogic Server (version 10 and above), IBM WebSphere Application Server (version 7 and above), Resin (3.1 and above) and JBoss (6.x or above) provide a ClassLoader that is capable of local instrumentation. Spring's native LTW leverages such ClassLoaders to enable AspectJ weaving. You can enable LTW by simply activating load-time weaving as described earlier. Specifically, you do *not* need to modify the launch script to add `-javaagent:path/to/spring-instrument.jar`.

Note that GlassFish instrumentation-capable ClassLoader is available only in its EAR environment. For GlassFish web applications, follow the Tomcat setup instructions as outlined above.

Note that on JBoss 6.x, the app server scanning needs to be disabled to prevent it from loading the classes before the application actually starts. A quick workaround is to add to your artifact a file named `WEB-INF/jboss-scanning.xml` with the following content:

```
<scanning xmlns="urn:jboss:scanning:1.0"/>
```

### **Generic Java applications**

When class instrumentation is required in environments that do not support or are not supported by the existing `LoadTimeWeaver` implementations, a JDK agent can be the only solution. For such cases, Spring provides `InstrumentationLoadTimeWeaver`, which requires a Spring-specific (but very general) VM agent, `org.springframework.instrument-{version}.jar` (previously named `spring-agent.jar`).

To use it, you must start the virtual machine with the Spring agent, by supplying the following JVM options:

```
-javaagent:/path/to/org.springframework.instrument-{version}.jar
```

Note that this requires modification of the VM launch script which may prevent you from using this in application server environments (depending on your operation policies). Additionally, the JDK agent will instrument the *entire* VM which can prove expensive.

For performance reasons, it is recommended to use this configuration only if your target environment (such as [Jetty](#)) does not have (or does not support) a dedicated LTW.

## 7.9 Further Resources

More information on AspectJ can be found on the [AspectJ website](#).

The book *Eclipse AspectJ* by Adrian Colyer et. al. (Addison-Wesley, 2005) provides a comprehensive introduction and reference for the AspectJ language.

The book *AspectJ in Action, Second Edition* by Ramnivas Laddad (Manning, 2009) comes highly recommended; the focus of the book is on AspectJ, but a lot of general AOP themes are explored (in some depth).

## 8.1 Introduction

The previous chapter described the Spring's support for AOP using `@AspectJ` and schema-based aspect definitions. In this chapter we discuss the lower-level Spring AOP APIs and the AOP support used in Spring 1.2 applications. For new applications, we recommend the use of the Spring 2.0 and later AOP support described in the previous chapter, but when working with existing applications, or when reading books and articles, you may come across Spring 1.2 style examples. Spring 4.0 is backwards compatible with Spring 1.2 and everything described in this chapter is fully supported in Spring 4.0.

## 8.2 Pointcut API in Spring

Let's look at how Spring handles the crucial pointcut concept.

### 8.2.1 Concepts

Spring's pointcut model enables pointcut reuse independent of advice types. It's possible to target different advice using the same pointcut.

The `org.springframework.aop.Pointcut` interface is the central interface, used to target advices to particular classes and methods. The complete interface is shown below:

```
public interface Pointcut {  
  
    ClassFilter getClassFilter();  
  
    MethodMatcher getMethodMatcher();  
  
}
```

Splitting the `Pointcut` interface into two parts allows reuse of class and method matching parts, and fine-grained composition operations (such as performing a "union" with another method matcher).

The `ClassFilter` interface is used to restrict the pointcut to a given set of target classes. If the `matches()` method always returns true, all target classes will be matched:

```
public interface ClassFilter {  
  
    boolean matches(Class clazz);  
}
```

The `MethodMatcher` interface is normally more important. The complete interface is shown below:

```
public interface MethodMatcher {  
  
    boolean matches(Method m, Class targetClass);  
  
    boolean isRuntime();  
  
    boolean matches(Method m, Class targetClass, Object[] args);  
}
```

The `matches(Method, Class)` method is used to test whether this pointcut will ever match a given method on a target class. This evaluation can be performed when an AOP proxy is created, to avoid the need for a test on every method invocation. If the 2-argument `matches` method returns true for a given method, and the `isRuntime()` method for the `MethodMatcher` returns true, the 3-argument `matches` method will be invoked on every method invocation. This enables a pointcut to look at the arguments passed to the method invocation immediately before the target advice is to execute.

Most `MethodMatchers` are static, meaning that their `isRuntime()` method returns false. In this case, the 3-argument `matches` method will never be invoked.



If possible, try to make pointcuts static, allowing the AOP framework to cache the results of pointcut evaluation when an AOP proxy is created.

## 8.2.2 Operations on pointcuts

Spring supports operations on pointcuts: notably, *union* and *intersection*.

- Union means the methods that either pointcut matches.
- Intersection means the methods that both pointcuts match.
- Union is usually more useful.
- Pointcuts can be composed using the static methods in the `org.springframework.aop.support.Pointcuts` class, or using the `ComposablePointcut` class in the same package. However, using AspectJ pointcut expressions is usually a simpler approach.

## 8.2.3 AspectJ expression pointcuts

Since 2.0, the most important type of pointcut used by Spring is

`org.springframework.aop.aspectj.AspectJExpressionPointcut`. This is a pointcut that uses an AspectJ supplied library to parse an AspectJ pointcut expression string.

See the previous chapter for a discussion of supported AspectJ pointcut primitives.

## 8.2.4 Convenience pointcut implementations

Spring provides several convenient pointcut implementations. Some can be used out of the box; others are intended to be subclassed in application-specific pointcuts.

### Static pointcuts

Static pointcuts are based on method and target class, and cannot take into account the method's arguments. Static pointcuts are sufficient - *and best* - for most usages. It's possible for Spring to evaluate a static pointcut only once, when a method is first invoked: after that, there is no need to evaluate the pointcut again with each method invocation.

Let's consider some static pointcut implementations included with Spring.

### Regular expression pointcuts

One obvious way to specify static pointcuts is regular expressions. Several AOP frameworks besides Spring make this

possible. `org.springframework.aop.support.JdkRegexpMethodPointcut` is a generic regular expression pointcut, using the regular expression support in JDK 1.4+.

Using the `JdkRegexpMethodPointcut` class, you can provide a list of pattern Strings. If any of these is a match, the pointcut will evaluate to true. (So the result is effectively the union of these pointcuts.)

The usage is shown below:

```
<bean id="settersAndAbsquatulatePointcut"
      class="org.springframework.aop.support.JdkRegexpMethodPointcut">
  <property name="patterns">
    <list>
      <value>.*set.*</value>
      <value>.*absquatulate</value>
    </list>
  </property>
</bean>
```

Spring provides a convenience class, `RegexpMethodPointcutAdvisor`, that allows us to also reference an Advice (remember that an Advice can be an interceptor, before advice, throws advice etc.). Behind the scenes, Spring will use a `JdkRegexpMethodPointcut`. Using `RegexpMethodPointcutAdvisor` simplifies wiring, as the one bean encapsulates both pointcut and advice, as shown below:

```

<bean id="settersAndAbsquatulateAdvisor"
      class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice">
      <ref bean="beanNameOfAopAllianceInterceptor"/>
    </property>
    <property name="patterns">
      <list>
        <value>.*set.*</value>
        <value>.*absquatulate</value>
      </list>
    </property>
  </bean>

```

*RegexpMethodPointcutAdvisor* can be used with any Advice type.

### Attribute-driven pointcuts

An important type of static pointcut is a *metadata-driven* pointcut. This uses the values of metadata attributes: typically, source-level metadata.

## Dynamic pointcuts

Dynamic pointcuts are costlier to evaluate than static pointcuts. They take into account method *arguments*, as well as static information. This means that they must be evaluated with every method invocation; the result cannot be cached, as arguments will vary.

The main example is the `control flow` pointcut.

### Control flow pointcuts

Spring control flow pointcuts are conceptually similar to AspectJ *cflow* pointcuts, although less powerful. (There is currently no way to specify that a pointcut executes below a join point matched by another pointcut.) A control flow pointcut matches the current call stack. For example, it might fire if the join point was invoked by a method in the `com.mycompany.web` package, or by the `SomeCaller` class. Control flow pointcuts are specified using the `org.springframework.aop.support.ControlFlowPointcut` class.



Control flow pointcuts are significantly more expensive to evaluate at runtime than even other dynamic pointcuts. In Java 1.4, the cost is about 5 times that of other dynamic pointcuts.

### 8.2.5 Pointcut superclasses

Spring provides useful pointcut superclasses to help you to implement your own pointcuts.

Because static pointcuts are most useful, you'll probably subclass `StaticMethodMatcherPointcut`, as shown below. This requires implementing just one abstract method (although it's possible to override other methods to customize behavior):

```
class TestStaticPointcut extends StaticMethodMatcherPointcut {  
  
    public boolean matches(Method m, Class targetClass) {  
        // return true if custom criteria match  
    }  
}
```

There are also superclasses for dynamic pointcuts.

You can use custom pointcuts with any advice type in Spring 1.0 RC2 and above.

## 8.2.6 Custom pointcuts

Because pointcuts in Spring AOP are Java classes, rather than language features (as in AspectJ) it's possible to declare custom pointcuts, whether static or dynamic. Custom pointcuts in Spring can be arbitrarily complex. However, using the AspectJ pointcut expression language is recommended if possible.



Later versions of Spring may offer support for "semantic pointcuts" as offered by JAC: for example, "all methods that change instance variables in the target object."

## 8.3 Advice API in Spring

Let's now look at how Spring AOP handles advice.

### 8.3.1 Advice lifecycles

Each advice is a Spring bean. An advice instance can be shared across all advised objects, or unique to each advised object. This corresponds to *per-class* or *per-instance* advice.

Per-class advice is used most often. It is appropriate for generic advice such as transaction advisors. These do not depend on the state of the proxied object or add new state; they merely act on the method and arguments.

Per-instance advice is appropriate for introductions, to support mixins. In this case, the advice adds state to the proxied object.

It's possible to use a mix of shared and per-instance advice in the same AOP proxy.

### 8.3.2 Advice types in Spring

Spring provides several advice types out of the box, and is extensible to support arbitrary advice types. Let us look at the basic concepts and standard advice types.

#### Interception around advice

The most fundamental advice type in Spring is *interception around advice*.

Spring is compliant with the AOP Alliance interface for around advice using method interception. MethodInterceptors implementing around advice should implement the following interface:

```
public interface MethodInterceptor extends Interceptor {  
  
    Object invoke(MethodInvocation invocation) throws Throwable;  
}
```

The `MethodInvocation` argument to the `invoke()` method exposes the method being invoked; the target join point; the AOP proxy; and the arguments to the method. The `invoke()` method should return the invocation's result: the return value of the join point.

A simple `MethodInterceptor` implementation looks as follows:

```

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;
    }
}

```

Note the call to the `MethodInvocation`'s `proceed()` method. This proceeds down the interceptor chain towards the join point. Most interceptors will invoke this method, and return its return value. However, a `MethodInterceptor`, like any around advice, can return a different value or throw an exception rather than invoke the `proceed` method. However, you don't want to do this without good reason!



MethodInterceptors offer interoperability with other AOP Alliance-compliant AOP implementations. The other advice types discussed in the remainder of this section implement common AOP concepts, but in a Spring-specific way. While there is an advantage in using the most specific advice type, stick with `MethodInterceptor` around advice if you are likely to want to run the aspect in another AOP framework. Note that pointcuts are not currently interoperable between frameworks, and the AOP Alliance does not currently define pointcut interfaces.

## Before advice

A simpler advice type is a *before advice*. This does not need a `MethodInvocation` object, since it will only be called before entering the method.

The main advantage of a before advice is that there is no need to invoke the `proceed()` method, and therefore no possibility of inadvertently failing to proceed down the interceptor chain.

The `MethodBeforeAdvice` interface is shown below. (Spring's API design would allow for field before advice, although the usual objects apply to field interception and it's unlikely that Spring will ever implement it).

```

public interface MethodBeforeAdvice extends BeforeAdvice {

    void before(Method m, Object[] args, Object target) throws Throwable;
}

```

Note the return type is `void`. Before advice can insert custom behavior before the join point executes, but cannot change the return value. If a before advice throws an exception, this will abort further execution of the interceptor chain. The exception will propagate back up the interceptor chain. If it is unchecked, or on the signature of the invoked method, it will be passed directly to the client; otherwise it will be wrapped in an unchecked exception by the AOP proxy.

An example of a before advice in Spring, which counts all method invocations:

```
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;
    }
}
```



Before advice can be used with any pointcut.

## Throws advice

*Throws advice* is invoked after the return of the join point if the join point threw an exception. Spring offers typed throws advice. Note that this means that the `org.springframework.aop.ThrowsAdvice` interface does not contain any methods: It is a tag interface identifying that the given object implements one or more typed throws advice methods. These should be in the form of:

```
afterThrowing([Method, args, target], subclassOfThrowable)
```

Only the last argument is required. The method signatures may have either one or four arguments, depending on whether the advice method is interested in the method and arguments. The following classes are examples of throws advice.

The advice below is invoked if a `RemoteException` is thrown (including subclasses):

```
public class RemoteThrowsAdvice implements ThrowsAdvice {

    public void afterThrowing(RemoteException ex) throws Throwable {
        // Do something with remote exception
    }
}
```

The following advice is invoked if a `ServletException` is thrown. Unlike the above advice, it declares 4 arguments, so that it has access to the invoked method, method arguments and target object:

```
public class ServletThrowsAdviceWithArguments implements ThrowsAdvice {

    public void afterThrowing(Method m, Object[] args, Object target, ServletException
ex) {
        // Do something with all arguments
    }
}
```

The final example illustrates how these two methods could be used in a single class, which handles both `RemoteException` and `ServletException`. Any number of throws advice methods can be combined in a single class.

```
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
    }
}
```



If a throws-advice method throws an exception itself, it will override the original exception (i.e. change the exception thrown to the user). The overriding exception will typically be a `RuntimeException`; this is compatible with any method signature. However, if a throws-advice method throws a checked exception, it will have to match the declared exceptions of the target method and is hence to some degree coupled to specific target method signatures. *Do not throw an undeclared checked exception that is incompatible with the target method's signature!*



Throws advice can be used with any pointcut.

## After Returning advice

An after returning advice in Spring must implement the `org.springframework.aop.AfterReturningAdvice` interface, shown below:

```
public interface AfterReturningAdvice extends Advice {

    void afterReturning(Object returnValue, Method m, Object[] args, Object target)
        throws Throwable;
}
```

An after returning advice has access to the return value (which it cannot modify), invoked method, methods arguments and target.

The following after returning advice counts all successful method invocations that have not thrown exceptions:

```
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;
    }
}
```

This advice doesn't change the execution path. If it throws an exception, this will be thrown up the interceptor chain instead of the return value.



After returning advice can be used with any pointcut.

## Introduction advice

Spring treats introduction advice as a special kind of interception advice.

Introduction requires an `IntroductionAdvisor`, and an `IntroductionInterceptor`, implementing the following interface:

```
public interface IntroductionInterceptor extends MethodInterceptor {

    boolean implementsInterface(Class intf);
}
```

The `invoke()` method inherited from the AOP Alliance `MethodInterceptor` interface must implement the introduction: that is, if the invoked method is on an introduced interface, the introduction interceptor is responsible for handling the method call - it cannot invoke `proceed()`.

Introduction advice cannot be used with any pointcut, as it applies only at class, rather than method, level. You can only use introduction advice with the `IntroductionAdvisor`, which has the following methods:

```
public interface IntroductionAdvisor extends Advisor, IntroductionInfo {

    ClassFilter getClassFilter();

    void validateInterfaces() throws IllegalArgumentException;
}

public interface IntroductionInfo {

    Class[] getInterfaces();
}
```

There is no `MethodMatcher`, and hence no `Pointcut`, associated with introduction advice. Only class filtering is logical.

The `getInterfaces()` method returns the interfaces introduced by this advisor.

The `validateInterfaces()` method is used internally to see whether or not the introduced interfaces can be implemented by the configured `IntroductionInterceptor`.

Let's look at a simple example from the Spring test suite. Let's suppose we want to introduce the following interface to one or more objects:

```
public interface Lockable {
    void lock();
    void unlock();
    boolean locked();
}
```

This illustrates a *mixin*. We want to be able to cast advised objects to Lockable, whatever their type, and call lock and unlock methods. If we call the lock() method, we want all setter methods to throw a `LockedException`. Thus we can add an aspect that provides the ability to make objects immutable, without them having any knowledge of it: a good example of AOP.

Firstly, we'll need an `IntroductionInterceptor` that does the heavy lifting. In this case, we extend the `org.springframework.aop.support.DelegatingIntroductionInterceptor` convenience class. We could implement `IntroductionInterceptor` directly, but using `DelegatingIntroductionInterceptor` is best for most cases.

The `DelegatingIntroductionInterceptor` is designed to delegate an introduction to an actual implementation of the introduced interface(s), concealing the use of interception to do so. The delegate can be set to any object using a constructor argument; the default delegate (when the no-arg constructor is used) is this. Thus in the example below, the delegate is the `LockMixin` subclass of `DelegatingIntroductionInterceptor`. Given a delegate (by default itself), a `DelegatingIntroductionInterceptor` instance looks for all interfaces implemented by the delegate (other than `IntroductionInterceptor`), and will support introductions against any of them. It's possible for subclasses such as `LockMixin` to call the `suppressInterface(Class intf)` method to suppress interfaces that should not be exposed. However, no matter how many interfaces an `IntroductionInterceptor` is prepared to support, the `IntroductionAdvisor` used will control which interfaces are actually exposed. An introduced interface will conceal any implementation of the same interface by the target.

Thus `LockMixin` extends `DelegatingIntroductionInterceptor` and implements `Lockable` itself. The superclass automatically picks up that `Lockable` can be supported for introduction, so we don't need to specify that. We could introduce any number of interfaces in this way.

Note the use of the `locked` instance variable. This effectively adds additional state to that held in the target object.

```

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);
    }

}

```

Often it isn't necessary to override the `invoke()` method: the `DelegatingIntroductionInterceptor` implementation - which calls the delegate method if the method is introduced, otherwise proceeds towards the join point - is usually sufficient. In the present case, we need to add a check: no setter method can be invoked if in locked mode.

The introduction advisor required is simple. All it needs to do is hold a distinct `LockMixin` instance, and specify the introduced interfaces - in this case, just `Lockable`. A more complex example might take a reference to the introduction interceptor (which would be defined as a prototype): in this case, there's no configuration relevant for a `LockMixin`, so we simply create it using `new`.

```

public class LockMixinAdvisor extends DefaultIntroductionAdvisor {

    public LockMixinAdvisor() {
        super(new LockMixin(), Lockable.class);
    }
}

```

We can apply this advisor very simply: it requires no configuration. (However, it *is* necessary: It's impossible to use an `IntroductionInterceptor` without an *IntroductionAdvisor*.) As usual with introductions, the advisor must be per-instance, as it is stateful. We need a different

instance of `LockMixinAdvisor`, and hence `LockMixin`, for each advised object. The advisor comprises part of the advised object's state.

We can apply this advisor programmatically, using the `Advised.addAdvisor()` method, or (the recommended way) in XML configuration, like any other advisor. All proxy creation choices discussed below, including "auto proxy creators," correctly handle introductions and stateful mixins.

## Part III. 测试

作为 Spring 的开发团队，我们鼓励在开发活动中引入测试驱动开发（TDD，Test-Driven Development）的行为，因此本文档接下来将涵盖 Spring 框架对集成测试的支持（以及 Spring 下单元测试的最佳实践）。

Spring 开发团队发现对控制反转（IoC）的正确运用可以使针对代码编写单元测试与集成测试变得更为容易（`setter`方法的存在，以及类里面恰当的构造器，使得测试代码在无需类似服务工厂等辅助工具的前提下，也能够方便地对各个类进行调用）。我们希望通过整整一章对 Spring 框架下测试相关主题的讲解，让读者也能对我们的以上发现表示赞同。

## 9. Spring 框架下的测试

测试乃企业级软件开发的重要组成部分之一。本章专注于讲解采用 IoC 原则进行编码而给[单元测试](#)带来的好处，以及 Spring 框架对[集成测试](#)的支持如何为测试带来帮助。（对企业开发中如何进行代码测试的详尽讨论不在本文档范围之内）

# 10. 单元测试

比起传统的 Java EE 开发方式，依赖注入可以弱化你的代码对容器的依赖。在基于 JUnit 或 TestNG 的测试代码中，无需依赖于 Spring 或其他容器，你只需通过 `new` 操作符，便可以创建出组成你的应用程序的各种 POJO 对象。而通过 `mock 对象`（以及其它各种测试技术的综合运用），你可以将被测试的代码单独隔离开来进行测试。如果你在进行架构设计时遵循了 Spring 所推荐的模式，那么由此带来的诸如清晰的分层、组件化等等优点也会使你更容易地对你的代码进行单元测试。例如，通过 `mock DAO 层或 Repository 接口的实现`，针对服务层对象的单元测试代码并不需要真正地访问持久层的数据。

真正的单元测试代码运行的非常快，因为并不需要一整套的运行时架构去支持测试的运转。所以在开发中强调真正的单元测试必须成为编码方法论中的一部分将会极大地提高你的生产力。

也许没有这一节内容的帮助你也能写出高效的针对 IoC 应用的单元测试，但是以下将要提到的 Spring 所提供的 `mock 对象` 和 `测试支持类`，在一些特定的场景中会对单元测试很有帮助。

## 10.1 Mock 对象

### 10.1.1 环境

`org.springframework.mock.env` 包含了对抽象 `Environment` 和 `PropertySource` 的 Mock 实现（参考 [Section 3.13.1 1Bean 的定义文件](#) 和 [Section 3.13.3 PropertySource 抽象](#)）。`MockEnvironment` 和 `MockPropertySource` 对于编写针对依赖于环境相关属性的代码的，与容器无关的测试用例很有帮助。

### 10.1.2 JNDI

`org.springframework.mock.jndi` 包含了 JNDI SPI 的实现。这一实现可以让你为你的测试套件或独立应用配置起一个简单的 JNDI 环境。例如，如果测试代码和 Java EE 容器的 JDBC `DataSource` 都绑定到同一个 JNDI 名字上，你就可以在测试中直接复用应用程序的代码和配置而不需做任何改动。

### 10.1.3 Servlet API

The `org.springframework.mock.web` package contains a comprehensive set of Servlet API mock objects, which are useful for testing web contexts, controllers, and filters. These mock objects are targeted at usage with Spring's Web MVC framework and are generally more

convenient to use than dynamic mock objects such as [EasyMock](#) or alternative Servlet API mock objects such as [MockObjects](#). Since Spring Framework 4.0, the set of mocks in the `org.springframework.mock.web` package is based on the Servlet 3.0 API.

For thorough integration testing of your Spring MVC and REST `Controller`s in conjunction with your `WebApplicationContext` configuration for Spring MVC, see the [Spring MVC Test Framework](#).

`org.springframework.mock.web` 包含了十分全面的 Servlet API mock 对象。这些对象在测试 web 上下文，Controller 和 Filter 的时候很有用。由于这些 mock 对象是有针对性地为了与 Spring 的 Web MVC 框架共同使用而编写的，因此相比起诸如 EasyMock 这种动态mock 对象或 MockObjects 这种替代性的 Servlet API mock 对象，使用起来要更为方便。

假如是要把 Spring MVC 和 REST `Controller` 与 `WebApplicationContext` 配置结合起来进行一番彻底的集成测试，请使用 [Spring MVC 测试框架](#)。

## 10.2 单元测试支持类

### 10.2.1 通用支持工具

`org.springframework.test.util` 包含了一些供单元测试和集成测试中使用的通用工具。

`ReflectionTestUtils` 是一组基于反射的方法集合。在测试包含如下一些测试用例的应用时，开发人员可以使用这些工具方法应对诸如更改一个常量的值，赋值一个非公有字段，调用一个非公有 `setter` 方法，或调用一个非公有的 配置 或 生命周期 回调方法等测试场景：

- 譬如 JPA 和 Hibernate 等广泛采用 `private` 或 `protected` 访问方式而 `public` `setter` 方法来访问 domain entity 属性的 ORM (Object-Relational Mapping) 框架
- `@Autowired`，`@Inject` 和 `@Resource` 等用于对 `private` 或 `protected` 字段，`setter` 方法和配置方法进行依赖注入的 Spring 注解。
- `@PostConstruct` 和 `@PreDestroy` 等用在生命周期回调方法上的注解。

`AopTestUtils` 是一组 AOP 相关工具方法的集合。这些方法可以用于帮助获取隐藏于一重或多层 Spring 代理背后目标对象的引用。举个栗子，你使用 EasyMock 或 Mockito mock 了一个被包装在 Spring 代理之中的 bean，这时你可能需要对这个 mock 进行直接的访问，从而能够配置对此 mock 的期望行为并在稍后执行验证。关于 Spring 提供的核心 AOP 工具，请参考 `AopUtils` 和 `AopProxyUtils` 这两个类。

### 10.2.2 Spring MVC

The `org.springframework.test.web` package contains `ModelAndViewAssert`, which you can use in combination with JUnit, TestNG, or any other testing framework for unit tests dealing with Spring MVC  `ModelAndView` objects.

`org.springframework.test.web` 包含了 `ModelAndViewAssert` 类。你可以在用 Junit, TestNG 或任何测试框架编写的单元测试中使用这个类来帮助你跟 Spring MVC 框架的  `ModelAndView` 对象进行互动。

注：如果想要像测试 POJO 一样来测试你的 Spring MVC controller，你可以将 `ModelAndViewAssert` 与 `MockHttpServletRequest`、`MockHttpSession` 等来自 [Servlet API mocks](#) 的 Mock 类结合使用。而假如是要把 Spring MVC 和 REST controller 与 `WebApplicationContext` 配置结合起来进行一番彻底的集成测试，请使用 [Spring MVC 测试框架](#)。



# 11. 集成测试

## 11.1 概述

能够在不需要部署到应用服务器或连接到其它企业基础服务的前提下做一些集成测试是很重要的。这将使你能够测试以下内容：

- Spring IoC容器上下文的正确装配。
- 使用JDBC或其它ORM工具访问数据。这将包括SQL语句、Hibernate查询和JPA实体映射的正确性等等这些内容。

Spring Framework在 `spring-test` 模块中为集成测试提供了强有力的支持。该Jar包的实际名字可能会包含发布版本号而且可能是 `org.springframework.test` 这样长的形式，这取决于你是从哪获得的（请参阅[section on Dependency Management](#)中的解释）。这个库包括了 `org.springframework.test` 包，其中包含了使用Spring容器进行集成测试的重要的类。这个测试不依赖于应用服务器和其它的发布环境。这些测试会比单元测试稍慢但比同类型的Selenium测试或信赖于发布到应用服务器的远程测试要快得多。

在Spring2.5及之后的版本，单元和集成测试支持是以注解驱动[Spring TestContext框架](#)这样的形式提供的。TestContext框架对实际使用的测试框架是不可知的，因此可以使用包括JUnit, TestNG等等许多测试手段。

## 11.2 集成测试的目标

Spring的集成测试支持有以下几点主要目标：

- 管理各个测试执行之间的Spring IoC容器缓存
- 提供测试配置实例的依赖注入
- 提供适合集成测试的事务管理
- 提供辅助开发人员编写集成测试的具备Spring特性的基础类

下面几节将解释每个目标并提供实现和配置详情的链接。

### 11.2.1 上下文管理和缓存

Spring TestContext框架对Spring ApplicationContext 和 WebApplicationContext 提供一致性加载并对它们进行缓存。对加载的上下文进行缓存提供支持是很重要的，因为启动时间是个问题——不是因为Spring自己的开销，而是被Spring容器初始化的对象需要时间去初始化。比如，一个拥有50到100个Hibernate映射文件的项目可能花费10到20秒的时间去载这些映射文件，在运行每一个测试工具中的测试用例之前都会引发开销并导致总体测试运行变缓慢，降低开发效率。

测试类通常声明一批XML的资源路径或者Groovy的配置元数据——通常在类路径中——或者是一批用于配置应用程序的注解类。这些路径或者类跟在 web.xml 或者其它用于生产部署的配置文件中指定的一样的。

通常，一旦被加载过一次， ApplicationContext 就将被用于每个测试中。因此启动开销在一次测试集中将只会引发一次，随后执行的测试将会快得多。在这里，“测试集”的意思是在同一个JVM的所有测试——比如说，对给定项目或者模块的一次Ant、Maven或者Gradle构建运行的所有测试。在不太可能的情况下，一个测试会破坏应用上下文并引起重新加载——比如，修改一个bean定义或者应用程序对象的状态——TestContext框架将被设置为在开始下个测试之前重新加载配置并重建应用上下文。

参见Section 11.5.4, “Context management”和TestContext框架的the section called “Context caching”一节。

### 11.2.2 测试配置的信赖注入

当TestContext框架加载你的应用程序上下文的时候，它将通过信赖注入有选择性地配置测试实例。这为使用你的应用程序上下文中的预配置bean来建立测试配置提供了一个很方便的机制。这里有一个很大的便处就是你可以在不同的测试场景中重复使用应用程序上下文（比

如，配置基于Spring管理的对象图、事务代理、数据源等等），这样省去了为每个测试用例建立复杂的测试配置的必要。

举例说明，考虑这样一个场景，我们有一个类叫做 `HibernateTitleRepository`，它实现了 `Title` 领域实体的数据访问逻辑。我们想编写集成测试来测试以下方面：

- Spring配置：总的来说，就是与 `HibernateTitleRepository` 配置有关的一切是否正确和存在？
- Hibernate映射文件：是否所有映射都正确，并且延迟加载的设置是否准备就绪？
- `HibernateTitleRepository` 的逻辑：此类中的配置实例是否与预期一致？

查看使用[TestContext框架](#)进行测试配置的信赖注入。

### 11.2.3 事务管理

测试中访问一个真实数据库的一个常见的问题是在持久层存储状态付出的努力。即使你使用开发环境的数据库，改变相应状态也会影响将来的测试。并且，许多操作——插入或者改变持久层数据——也不能在事务之外执行（或者验证）。

`TestContext`框架解决了这个问题。默认行为下，这个框架将为每个测试创建并回滚一个事务。你只需简单的假定事务是存在的并写你的代码即可。如果你调用事务代理对象，他们也会根据它们配置的语义正确执行。而且，如果一个测试方法在运行相应事务时删除了选定表中的内容，事务默认情况下会进行回滚，数据库会回到测试执行前的那个状态。事务通过定义在应用程序上下文中的 `PlatformTransactionManager` bean 来得到支持。

如果你需要提交事务——通常不会这样做，但有时当你想用一个特定的测试来填充或者修改数据库时也会显得有用——`TestContext`框架将根据 `@Commit` 注解的指示对事务进行提交而不是回滚。

查看使用[TestContext框架](#)进行事务管理。

### 11.2.4 集成测试的支持类

Spring `TestContext`框架提供了一些支持来简化集成测试的编写。这些基础类为测试框架提供了定义良好的钩子，还有一些便利的实例变量和方法，使你能够访问：

- `ApplicationContext`，用于从整体上来进行显示的bean查找或者测试上下文的状态。
- `JdbcTemplate`，用于执行SQL语句来查询数据库。这些的查询可用于确认执行数据库相关的应用程序代码前后数据库的状态，并且Spring保证这些查询与应用程序代码在同一个事务作用域中执行。如果需要与ORM工具协同使用，请确保避免误报。

还有，你可能想用特定于你的项目的实例和方法来创建你自己自定义的，应用程序范围的超类。

查看 [TestContext](#) 框架的支持类。

## 11.3 JDBC测试支持

*Note that `AbstractTransactionalJUnit4SpringContextTests` and `AbstractTransactionalTestNGSpringContextTests` provide convenience methods which delegate to the aforementioned methods in `JdbcTestUtils`.*

The `spring-jdbc` module provides support for configuring and launching an embedded database which can be used in integration tests that interact with a database. For details, see [Section 15.8, “Embedded database support”](#) and [Section 15.8.5, “Testing data access logic with an embedded database”](#).

`org.springframework.test.jdbc` 是包含 `JdbcTestUtils` 的包，它是一个JDBC相关的工具方法集，意在简化标准数据库测试场景。特别地，`JdbcTestUtils` 提供以下静态工具方法：

- `countRowsInTable(..)` : 统计给定表的行数。
- `countRowsInTableWhere(..)` : 使用提供的 `where` 语句进行筛选统计给定表的行数。
- `deleteFromTables(..)` : 删除特定表的全部数据。
- `deleteFromTableWhere(..)` : 使用提供的 `where` 语句进行筛选并删除给定表的数据。
- `dropTables(..)` : 删除指定的表。

注

意 `AbstractTransactionalJUnit4SpringContextTests` 和 `AbstractTransactionalTestNGSpringContextTests` 提供了委托给前面所述的 `JdbcTestUtils` 中的方法的简便方法。

`spring-jdbc` 模块提供了配置和启动嵌入式数据库的支持，可用于与数据库交互的集成测试中。

详见[Section 15.8, “嵌入式数据库支持”](#)和[Section 15.8.5, “使用嵌入式数据库测试数据访问逻辑”](#)。

## 11.4 注解

### 11.4.1 Spring 测试注解

Spring 框架提供以下 Spring 特定的注解集合，你可以在单元和集成测试中协同 `TestContext` 框架使用它们。请参考相应的 Java 帮助文档作进一步了解，包括默认的属性，属性别名等等。

#### `@BootstrapWith`

`@BootstrapWith` 是一个用于配置 Spring `TestContext` 框架如何引导的类级别的注解。具体地说，`@BootstrapWith` 用于指定一个自定义的 `TestContextBootstrapper`。请查看[引导 `TestContext` 框架](#)作进一步了解。

#### `@ContextConfiguration`

`@ContextConfiguration` 定义了类级别的元数据来决定如何为集成测试来加载和配置应用程序上下文。具体地说，`@ContextConfiguration` 声明了用于加载上下文的应用程序上下文资源路径和注解类。

资源路径通常是类路径中的 XML 配置文件或者 Groovy 脚本；而注解类通常是使用 `@Configuration` 注解的类。但是，资源路径也可以指向文件系统中的文件和脚本，解决类也可能是组件类等等。

```
@ContextConfiguration("/test-config.xml")
public class XmlApplicationContextTests {
    // class body...
}
```

```
@ContextConfiguration(classes = TestConfig.class)
public class ConfigClassApplicationContextTests {
    // class body...
}
```

作为声明资源路径或注解类的替代方案或补充，`@ContextConfiguration` 可以用于声明 `ApplicationContextInitializer` 类。

```
@ContextConfiguration(initializers = CustomContextInitializer.class)
public class ContextInitializerTests {
    // class body...
}
```

`@ContextConfiguration` 偶尔也被用作声明 `ContextLoader` 策略。但注意，通常你不需要显示的配置加载器，因为默认的加载器已经支持资源路径或者注解类以及初始化器。

```
@ContextConfiguration(locations = "/test-context.xml", loader = CustomContextLoader.class)
public class CustomLoaderXmlApplicationContextTests {
    // class body...
}
```



`@ContextConfiguration` 默认对继承父类定义的资源路径或者配置类以及上下文初始化器提供支持。

参阅 [Section 11.5.4, 上下文管理](#) 和 `@ContextConfiguration` 帮助文档作进一步了解。

## @WebAppConfiguration

`@WebAppConfiguration` 是一个用于声明集成测试所加载的 `ApplicationContext` 须是 `WebApplicationContext` 的类级别的注解。测试类的 `@WebAppConfiguration` 注解只是为了保证用于测试的 `WebApplicationContext` 会被加载，它使用 `"file:src/main/webapp"` 路径默认值作为 web 应用的根路径（即，资源基路径）。资源基路径用于幕后创建一个 `MockServletContext` 作为测试的 `WebApplicationContext` 的 `ServletContext`。

```
@ContextConfiguration
@WebAppConfiguration
public class WebAppTests {
    // class body...
}
```

要覆盖默认值，请通过隐式值属性指定不同的基本资源路径。`classpath:` 和 `file:` 资源前缀都被支持。如果没有提供资源前缀，则假定路径是文件系统资源。

```
@ContextConfiguration
@WebAppConfiguration("classpath:test-web-resources")
public class WebAppTests {
    // class body...
}
```

注意 `@WebAppConfiguration` 必须和 `@ContextConfiguration` 一起使用，或者在同一个测试类，或者在测试类层次结构中。请参阅 `@WebAppConfiguration` 帮助文档作进一步了解。

## @ContextHierarchy

`@ContextHierarchy` 是一个用于为集成测试定义 `ApplicationContext` 层次结构的类级别的注解。`@ContextHierarchy` 应该声明一个或多个 `@ContextConfiguration` 实例列表，其中每一个定义上下文层次结构的一个层次。下面的例子展示了在同一个测试类中 `@ContextHierarchy` 的使用方法。但是，`@ContextHierarchy` 一样可以用于测试类的层次结构中。

```
@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...
}
```

如果你想合并或者覆盖一个测试类的层次结构中的应用程序上下文中指定层次的配置，你就必须在类层次中的每一个相应的层次通过为 `@ContextConfiguration` 的 `name` 属性提供与该层次相同的值的方式来显示地指定这个层次。请参阅[上下文层次关系](#)和 `@ContextHierarchy` 帮助文档来获得更多的示例。

## ② **@ActiveProfiles**

`@ActiveProfiles` 是一个用于当集成测试加载 `ApplicationContext` 的时候声明哪一个 *bean definition profiles* 被激活的类级别的注解。

```
@ContextConfiguration
@ActiveProfiles("dev")
public class DeveloperTests {
    // class body...
}
```

```
@ContextConfiguration
@ActiveProfiles({"dev", "integration"})
public class DeveloperIntegrationTests {
    // class body...
}
```



`@ActiveProfiles` 默认为继承激活的在超类声明的 `bean definition profiles` 提供支持。通过实现一个自定义的 `ActiveProfilesResolver` 并通过 `@ActiveProfiles` 的 `resolver` 属性来注册它的编程的方式来解决激活 `bean definition profiles` 问题也是可行的。

参阅[使用环境profiles来配置上下文](#)和[@ActiveProfiles](#) 帮助文档作进一步了解。

## @TestPropertySource

`@TestPropertySource` 是一个用于为集成测试加载 `ApplicationContext` 时配置属性文件的位置和增加到 `Environment` 中的 `PropertySources` 集中的内联属性的类级别的注解。

测试属性源比那些从系统环境或者 Java 系统属性以及通过 `@PropertySource` 或者编程方式声明方式增加的属性源具有更高的优先级。而且，内联属性比从资源路径加载的属性具有更高的优先级。

下面的例子展示了如何从类路径中声明属性文件。

```
@ContextConfiguration
@TestPropertySource("/test.properties")
public class MyIntegrationTests {
    // class body...
}
```

下面的例子展示了如何声明内联属性。

```
@ContextConfiguration
@TestPropertySource(properties = { "timezone = GMT", "port: 4242" })
public class MyIntegrationTests {
    // class body...
}
```

## @DirtiesContext

`@DirtiesContext` 指明测试执行期间该 Spring `ApplicationContext` 已经被弄脏（也就是说通过某种方式被更改或者破坏——比如，更改单例 bean 的状态）。当应用程序上下文被标为“脏”，它将从测试框架缓存中被移除并关闭。因此，Spring 容器将为随后需要同样配置元数据的测试而被重建。

`@DirtiesContext` 可以在同一个类或者类层次结构中的类级别和方法级别中使用。在这个场景下，`ApplicationContext` 将在任意此注解的方法之前或之后以及当前测试类之前或之后被标为“脏”，这取决于配置的 `methodMode` 和 `classMode`。

下面的例子解释了在多种配置场景下什么时候上下文会被标为“脏”。

- 当在一个类中声明并将类模式设为 BEFORE\_CLASS，则在当前测试类之前。

```
@DirtiesContext(classMode = BEFORE_CLASS)
public class FreshContextTests {
    // some tests that require a new Spring container
}
```

- 当在一个类中声明并将类模式设为 AFTER\_CLASS（也就是，默认的类模式），则在当前测试类之后。

```
@DirtiesContext
public class ContextDirtyingTests {
    // some tests that result in the Spring container being dirtied
}
```

- 当在一个类中声明并将类模式设为 BEFORE\_EACH\_TEST\_METHOD，则在当前测试类的每个方法之前。

```
@DirtiesContext(classMode = BEFORE_EACH_TEST_METHOD)
public class FreshContextTests {
    // some tests that require a new Spring container
}
```

- 当在一个类中声明并将类模式设为 AFTER\_EACH\_TEST\_METHOD，则在当前测试类的每个方法之后。

```
@DirtiesContext(classMode = AFTER_EACH_TEST_METHOD)
public class ContextDirtyingTests {
    // some tests that result in the Spring container being dirtied
}
```

- 当在一个方法中声明并将方法模式设为 BEFORE\_METHOD，则在当前方法之前。

```
@DirtiesContext(methodMode = BEFORE_METHOD)
@Test
public void testProcessWhichRequiresFreshAppCtx() {
    // some logic that requires a new Spring container
}
```

- 当在一个方法中声明并将方法模式设为 AFTER\_METHOD（也就是说，默认的方法模式），则在当前方法之后。

```
@DirtiesContext
@Test
public void testProcessWhichDirtiesAppCtx() {
    // some logic that results in the Spring container being dirtied
}
```

如果 `@DirtiesContext` 被用于上下文被配置为通过 `@ContextHierarchy` 定义的上下文层次中的一部分的测试中，则 `hierarchyMode` 标志可用于控制如何声明上下文缓存。默认将使用一个穷举算法用于清除包括不仅当前层次而且与当前测试拥有共同祖先的其它上下文层次的缓存。所有在拥有共同祖先上下文的子层次的应用程序上下文都会从上下文中被移除并关闭。如果穷举算法对于特定的使用场景显得有点威力过猛，那么你可以指定一个更简单的当前层算法来代替，如下所。

```
@ContextHierarchy({
    @ContextConfiguration("/parent-config.xml"),
    @ContextConfiguration("/child-config.xml")
})
public class BaseTests {
    // class body...
}

public class ExtendedTests extends BaseTests {

    @Test
    @DirtiesContext(hierarchyMode = CURRENT_LEVEL)
    public void test() {
        // some logic that results in the child context being dirtied
    }
}
```

参阅 `DirtiesContext.HierarchyMode` 帮助文档以获得 `EXHAUSTIVE` 和 `CURRENT_LEVEL` 算法更详细的了解。

## @TestExecutionListeners

`@TestExecutionListeners` 定义了一个类级别的元数据，用于配置需要用 `TestContextManager` 进行注册的 `TestExecutionListener` 实现。通常，`@TestExecutionListeners` 与 `@ContextConfiguration` 一起使用。

```
@ContextConfiguration
@TestExecutionListeners({CustomTestExecutionListener.class, AnotherTestExecutionListener.class})
public class CustomTestExecutionListenerTests {
    // class body...
}
```

`@TestExecutionListeners` 默认支持继承监听器。参阅帮助文档获得示例和更详细的了解。

## @Commit

`@Commit` 指定事务性的测试方法在测试方法执行完成后对事务进行提交。`@Commit` 可以用作 `@Rollback(false)` 的直接替代，以更好的传达代码的意图。和 `@Rollback` 一样，`@Commit` 可以在类层次或者方法层级声明。

```
@Commit
@Test
public void testProcessWithoutRollback() {
    // ...
}
```

## @Rollback

`@Rollback` 指明当测试方法执行完毕的时候是否对事务性方法中的事务进行回滚。如果为 `true`，则进行回滚；否则，则提交（请参加 `@Commit`）。在 Spring TestContext 框架中，集成测试默认的 `@Rollback` 语义为 `true`，即使你不显示的指定它。

当声明为类级别注解时，`@Rollback` 为测试类层次结构中的所有测试方法定义了默认回滚语义。当被声明为方法级别的注解，则 `@Rollback` 为特定的方法指定回滚语义，并覆盖类级别的 `@Rollback` 和 `@Commit` 语义。

```
@Rollback(false)
@Test
public void testProcessWithoutRollback() {
    // ...
}
```

## @BeforeTransaction

`@BeforeTransaction` 指明通过 Spring 的 `@Transactional` 注解配置为需要在事务中执行的测试方法在事务开始之前先执行注解的 `void` 方法。从 Spring 框架 4.3 版本起，`@BeforeTransaction` 方法不再需要为 `public` 并可能被声明为基于 Java 8 的接口的默认方法。

```
@BeforeTransaction
void beforeTransaction() {
    // logic to be executed before a transaction is started
}
```

## @AfterTransaction

`@AfterTransaction` 指明通过Spring的`@Transactional`注解配置为需要在事务中执行的测试方法在事务结束之后执行注解的`void`方法。从Spring框架4.3版本起，`@AfterTransaction`方法不再需要为`public`并可能被声明为基于Java8的接口的默认方法。

```
@AfterTransaction
void afterTransaction() {
    // logic to be executed after a transaction has ended
}
```

## @Sql

`@Sql`用于注解测试类或者测试方法，以让在集成测试过程中配置的SQL脚本能够在给定的数据库中得到执行。

```
@Test
@Sql({"/test-schema.sql", "/test-user-data.sql"})
public void userTest {
    // execute code that relies on the test schema and test data
}
```

请参阅[通过`@sql`声明执行的SQL脚本](#)作进一步了解。

## @SqlConfig

`@SqlConfig`定义了用于决定如何解析和执行通过`@Sql`注解配置的SQL脚本。

```
@Test
@Sql(
    scripts = "/test-user-data.sql",
    config = @SqlConfig(commentPrefix = "`", separator = "@@")
)
public void userTest {
    // execute code that relies on the test data
}
```

## @SqlGroup

`@SqlGroup`是一个用于聚合几个`@Sql`注解的容器注解。`@SqlGroup`可以直接使用，通过声明几个嵌套的`@Sql`注解，也可以与Java8的可重复注解支持协同使用，即简单地在同一个类或方法上声明几个`@Sql`注解，隐式地产生这个容器注解。

```

@Test
@SqlGroup({
    @Sql/scripts = "/test-schema.sql", config = @SqlConfig(commentPrefix = ``),
    @Sql("/test-user-data.sql")
})
public void userTest {
    // execute code that uses the test schema and test data
}

```

## 11.4.2 标准注解支持

以下注解为 Spring TestContext 框架所有的配置提供标准语义支持。注意这些注解不仅限于测试，可以用在 Spring 框架的任意地方。

- `@Autowired`
- `@Qualifier`
- `@Resource (javax.annotation)`如果JSR-250存在
- `@ManagedBean (javax.annotation)`如果JSR-250存在
- `@Inject (javax.inject)`如果JSR-330存在
- `@Named (javax.inject)`如果JSR-330存在
- `@PersistenceContext (javax.persistence)`如果JPA存在
- `@PersistenceUnit (javax.persistence)`如果JPA存在
- `@Required`
- `@Transactional`



在 Spring TestContext 框架中，`@PostConstruct` 和 `@PreDestroy` 可以通过标准语义在配置于 `ApplicationContext` 的任意应用程序组件中使用；但是，这些生命周期注解在实际测试类中只有很有限的作用。如果一个测试类的方法被注解为 `@PostConstruct`，这个方法将在 test 框架中的任何 `before` 方法（也就是被 JUnit 中的 `@Before` 注解方法）调用之前被执行，这个规则将被应用于测试类的每个方法。另一方面，如果一个测试类的方法被注解为 `@PreDestroy`，这个方法将永远不会被执行。因为建议在测试类中使用 test 框架的测试生命周期回调来代替使用 `@PostConstruct` 和 `@PreDestroy`。

## 11.4.3 Spring JUnit 4 测试注解

仅当与 [SpringRunner](#)，[Spring's JUnit rules](#) 或 [Spring's JUnit 4 support classes](#) 结合使用时，才支持以下注释。

### `@IfProfileValue`

`@IfProfileValue` 指明该测试只在特定的测试环境中被启用。如果配置的 `ProfileValueSource` 为所提供的 `name` 返回匹配的 `value`，这该测试将被启用。否则，该测试将被禁用并忽略。

`@IfProfileValue` 可以用在类级别、方法级别或者两个同时。使用类级别的 `@IfProfileValue` 注解优先于当前类或其子类的任意方法的使用方法级别的注解。有 `@IfProfileValue` 注解意味着则测试被隐式开启。这与JUnit4的 `@Ignore` 注解是相类似的，除了使用 `@Ignore` 注解是用于禁用测试的之外。

```
@IfProfileValue(name="java.vendor", value="Oracle Corporation")
@Test
public void testProcessWhichRunsOnlyOnOracleJvm() {
    // some logic that should run only on Java VMs from Oracle Corporation
}
```

或者，你可以配置 `@IfProfileValue` 使用 `values` 列表（或语义）来实现JUnit 4环境中的类似TestNG对测试组的支持。

```
@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

`@ProfileValueSourceConfiguration` 是类级别注解，用于当获取通过 `@IfProfileValue` 配置的 `profile` 值时指定使用什么样的 `ProfileValueSource` 类型。如果一个测试没有指定 `@ProfileValueSourceConfiguration`，那么默认使用 `SystemProfileValueSource`。

```
@ProfileValueSourceConfiguration(CustomProfileValueSource.class)
public class CustomProfileValueSourceTests {
    // class body...
}
```

## @Timed

`@Timed` 用于指明被注解的测试必须在指定的时限（毫秒）内结束。如果测试超过指定时限，就当作测试失败。

时限包括测试方法本身所耗费的时间，包括任何重复（请查看 `@Repeat`）及任意初始化和销毁所用的时间。

```
@Timed(millis=1000)
public void testProcessWithOneSecondTimeout() {
    // some logic that should not take longer than 1 second to execute
}
```

Spring的 `@Timed` 注解与JUnit 4的 `@Test(timeout=...)` 支持相比具有不同的语义。确切地说，由于在JUnit 4中处理方法执行超时的方式（也就是，在独立进程中执行该测试方法），如果一个测试方法执行时间太长，`@Test(timeout=...)` 将直接判定该测试失败。而Spring的 `@Timed` 则不直接判定失败而是等待测试完成。

## @Repeat

`@Repeat` 指明该测试方法需被重复执行。注解指定该测试方法被重复的次数。

重复的范围包括该测试方法自身也包括相应的初始化和销毁方法。

```
@Repeat(10)
@Test
public void testProcessRepeatedly() {
    // ...
}
```

### 11.4.4 Meta-Annotation Support for Testing

可以将大部分测试相关的注解当作meta-annotations使用，以创建自定义组合注解来减少测试集中的重复配置。

下面的每个都可以在 `TestContext` 框架中被当作meta-annotations使用。

- `@BootstrapWith`
- `@ContextConfiguration`
- `@ContextHierarchy`
- `@ActiveProfiles`
- `@TestPropertySource`
- `@DirtiesContext`
- `@WebAppConfiguration`
- `@TestExecutionListeners`
- `@Transactional`
- `@BeforeTransaction`
- `@AfterTransaction`
- `@Commit`
- `@Rollback`

- `@Sql`
- `@SqlConfig`
- `@SqlGroup`
- `@Repeat`
- `@Timed`
- `@IfProfileValue`
- `@ProfileValueSourceConfiguration`

例如，如果发现我们在基于JUnit 4的测试集中重复以下配置...

```
@RunWith(SpringRunner.class)
@ContextConfiguration({"app-config.xml", "test-data-access-config.xml"})
@ActiveProfiles("dev")
@Transactional
public class OrderRepositoryTests { }

@RunWith(SpringRunner.class)
@ContextConfiguration({"app-config.xml", "test-data-access-config.xml"})
@ActiveProfiles("dev")
@Transactional
public class UserRepositoryTests { }
```

我们可以通过一个自定义的组合注解来减少上述的重复量，将通用的测试配置集中起来，就像这样：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@ContextConfiguration({"app-config.xml", "test-data-access-config.xml"})
@ActiveProfiles("dev")
@Transactional
public @interface TransactionalDevTest { }
```

然后我们就可以像下面一样使用我们自定义的 `@TransactionalDevTest` 注解来简化每个类的配置：

```
@RunWith(SpringRunner.class)
@TransactionalDevTest
public class OrderRepositoryTests { }

@RunWith(SpringRunner.class)
@TransactionalDevTest
public class UserRepositoryTests { }
```

想获得详情，请查看[Spring注解编程模型](#)



# 11.5 Spring TestContext Framework

The *Spring TestContext Framework* (located in the `org.springframework.test.context` package) provides generic, annotation-driven unit and integration testing support that is agnostic of the testing framework in use. The TestContext framework also places a great deal of importance on *convention over configuration* with reasonable defaults that can be overridden through annotation-based configuration.

In addition to generic testing infrastructure, the TestContext framework provides explicit support for JUnit 4 and TestNG in the form of `abstract` support classes. For JUnit 4, Spring also provides a custom JUnit `Runner` and custom JUnit `Rules` that allow one to write so-called *POJO test classes*. POJO test classes are not required to extend a particular class hierarchy.

The following section provides an overview of the internals of the TestContext framework. If you are only interested in *using* the framework and not necessarily interested in *extending* it with your own custom listeners or custom loaders, feel free to go directly to the configuration ([context management](#), [dependency injection](#), [transaction management](#)), [support classes](#), and [annotation support](#) sections.

## 11.5.1 Key abstractions

The core of the framework consists of the `TestContextManager` class and the `TestContext`, `TestExecutionListener`, and `SmartContextLoader` interfaces. A `TestContextManager` is created per test class (e.g., for the execution of all test methods within a single test class in JUnit 4). The `TestContextManager` in turn manages a `TestContext` that holds the context of the current test. The `TestContextManager` also updates the state of the `TestContext` as the test progresses and delegates to `TestExecutionListener` implementations, which instrument the actual test execution by providing dependency injection, managing transactions, and so on. A `SmartContextLoader` is responsible for loading an `ApplicationContext` for a given test class. Consult the javadocs and the Spring test suite for further information and examples of various implementations.

## TestContext

`TestContext` encapsulates the context in which a test is executed, agnostic of the actual testing framework in use, and provides context management and caching support for the test instance for which it is responsible. The `TestContext` also delegates to a `SmartContextLoader` to load an `ApplicationContext` if requested.

## TestContextManager

`TestContextManager` is the main entry point into the *Spring TestContext Framework* and is responsible for managing a single `TestContext` and signaling events to each registered `TestExecutionListener` at well-defined test execution points:

- prior to any *before class* or *before all* methods of a particular testing framework
- test instance post-processing
- prior to any *before* or *before each* methods of a particular testing framework
- immediately before execution of the test method but after test setup
- immediately after execution of the test method but before test tear down
- after any *after* or *after each* methods of a particular testing framework
- after any *after class* or *after all* methods of a particular testing framework

## TestExecutionListener

`TestExecutionListener` defines the API for reacting to test execution events published by the `TestContextManager` with which the listener is registered. See [Section 11.5.3, “TestExecutionListener configuration”](#).

## Context Loaders

`ContextLoader` is a strategy interface that was introduced in Spring 2.5 for loading an `ApplicationContext` for an integration test managed by the Spring TestContext Framework. Implement `SmartContextLoader` instead of this interface in order to provide support for annotated classes, active bean definition profiles, test property sources, context hierarchies, and `WebApplicationContext` support.

`SmartContextLoader` is an extension of the `ContextLoader` interface introduced in Spring 3.1. The `SmartContextLoader` SPI supersedes the `ContextLoader` SPI that was introduced in Spring 2.5. Specifically, a `SmartContextLoader` can choose to process resource `locations`, annotated `classes`, or context `initializers`. Furthermore, a `SmartContextLoader` can set active bean definition profiles and test property sources in the context that it loads.

Spring provides the following implementations:

- `DelegatingSmartContextLoader` : one of two default loaders which delegates internally to an `AnnotationConfigContextLoader`, a `GenericXmlContextLoader`, or a `GenericGroovyXmlContextLoader` depending either on the configuration declared for the test class or on the presence of default locations or default configuration classes. Groovy support is only enabled if Groovy is on the classpath.
- `WebDelegatingSmartContextLoader` : one of two default loaders which delegates internally

to an `AnnotationConfigWebApplicationContext`, a `GenericXmlWebApplicationContext`, or a `GenericGroovyXmlWebApplicationContext` depending either on the configuration declared for the test class or on the presence of default locations or default configuration classes. A `webContextLoader` will only be used if `@WebAppConfiguration` is present on the test class. Groovy support is only enabled if Groovy is on the classpath.

- `AnnotationConfigContextLoader` : loads a standard `ApplicationContext` from *annotated classes*.
- `AnnotationConfigWebApplicationContext` : loads a `WebApplicationContext` from *annotated classes*.
- `GenericGroovyXmlContextLoader` : loads a standard `ApplicationContext` from *resource locations* that are either Groovy scripts or XML configuration files.
- `GenericGroovyXmlWebContextLoader` : loads a `WebApplicationContext` from *resource locations* that are either Groovy scripts or XML configuration files.
- `GenericXmlContextLoader` : loads a standard `ApplicationContext` from *XML resource locations*.
- `GenericXmlWebContextLoader` : loads a `WebApplicationContext` from *XML resource locations*.
- `GenericPropertiesContextLoader` : loads a standard `ApplicationContext` from Java Properties files.

## 11.5.2 Bootstrapping the TestContext framework

The default configuration for the internals of the Spring TestContext Framework is sufficient for all common use cases. However, there are times when a development team or third party framework would like to change the default `contextLoader`, implement a custom `TestContext` or `ContextCache`, augment the default sets of `ContextCustomizerFactory` and `TestExecutionListener` implementations, etc. For such low level control over how the `TestContext` framework operates, Spring provides a bootstrapping strategy.

`TestContextBootstrapper` defines the SPI for *bootstrapping* the `TestContext` framework. A `TestContextBootstrapper` is used by the `TestContextManager` to load the `TestExecutionListener` implementations for the current test and to build the `TestContext` that it manages. A custom bootstrapping strategy can be configured for a test class (or test class hierarchy) via `@BootstrapWith`, either directly or as a meta-annotation. If a bootstrapper is not explicitly configured via `@BootstrapWith`, either the `DefaultTestContextBootstrapper` or the `WebTestContextBootstrapper` will be used, depending on the presence of `@WebAppConfiguration`.

Since the `TestContextBootstrapper` SPI is likely to change in the future in order to accommodate new requirements, implementers are strongly encouraged not to implement this interface directly but rather to extend `AbstractTestContextBootstrapper` or one of its

concrete subclasses instead.

### 11.5.3 TestExecutionListener configuration

Spring provides the following `TestExecutionListener` implementations that are registered by default, exactly in this order.

- `ServletTestExecutionListener` : configures Servlet API mocks for a `WebApplicationContext`
- `DirtiesContextBeforeModesTestExecutionListener` : handles the `@DirtiesContext` annotation for *before* modes
- `DependencyInjectionTestExecutionListener` : provides dependency injection for the test instance
- `DirtiesContextTestExecutionListener` : handles the `@DirtiesContext` annotation for *after* modes
- `TransactionalTestExecutionListener` : provides transactional test execution with default rollback semantics
- `SqlScriptsTestExecutionListener` : executes SQL scripts configured via the `@Sql` annotation

## Registering custom TestExecutionListeners

Custom `TestExecutionListener`s can be registered for a test class and its subclasses via the `@TestExecutionListeners` annotation. See [annotation support](#) and the javadocs for `@TestExecutionListeners` for details and examples.

## Automatic discovery of default TestExecutionListeners

Registering custom `TestExecutionListener`s via `@TestExecutionListeners` is suitable for custom listeners that are used in limited testing scenarios; however, it can become cumbersome if a custom listener needs to be used across a test suite. Since Spring Framework 4.1, this issue is addressed via support for automatic discovery of *default* `TestExecutionListener` implementations via the `SpringFactoriesLoader` mechanism.

Specifically, the `spring-test` module declares all core default `TestExecutionListener`s under the `org.springframework.test.context.TestExecutionListener` key in its `META-INF/spring.factories` properties file. Third-party frameworks and developers can contribute their own `TestExecutionListener`s to the list of default listeners in the same manner via their own `META-INF/spring.factories` properties file.

## Ordering TestExecutionListeners

When the TestContext framework discovers default `TestExecutionListener`s via the aforementioned `SpringFactoriesLoader` mechanism, the instantiated listeners are sorted using Spring's `AnnotationAwareOrderComparator` which honors Spring's `Ordered` interface and `@order` annotation for ordering. `AbstractTestExecutionListener` and all default `TestExecutionListener`s provided by Spring implement `Ordered` with appropriate values. Third-party frameworks and developers should therefore make sure that their *default* `TestExecutionListener`s are registered in the proper order by implementing `Ordered` or declaring `@order`. Consult the javadocs for the `getOrder()` methods of the core default `TestExecutionListener`s for details on what values are assigned to each core listener.

## Merging `TestExecutionListeners`

If a custom `TestExecutionListener` is registered via `@TestExecutionListeners`, the *default* listeners will not be registered. In most common testing scenarios, this effectively forces the developer to manually declare all default listeners in addition to any custom listeners. The following listing demonstrates this style of configuration.

```
@ContextConfiguration
@TestExecutionListeners({
    MyCustomTestExecutionListener.class,
    ServletTestExecutionListener.class,
    DirtiesContextBeforeModesTestExecutionListener.class,
    DependencyInjectionTestExecutionListener.class,
    DirtiesContextTestExecutionListener.class,
    TransactionalTestExecutionListener.class,
    SqlScriptsTestExecutionListener.class
})
public class MyTest {
    // class body...
}
```

The challenge with this approach is that it requires that the developer know exactly which listeners are registered by default. Moreover, the set of default listeners can change from release to release — for example, `SqlScriptsTestExecutionListener` was introduced in Spring Framework 4.1, and `DirtiesContextBeforeModesTestExecutionListener` was introduced in Spring Framework 4.2. Furthermore, third-party frameworks like Spring Security register their own default `TestExecutionListener`s via the aforementioned [automatic discovery mechanism](#).

To avoid having to be aware of and re-declare **all default** listeners, the `mergeMode` attribute of `@TestExecutionListeners` can be set to `MergeMode.MERGE_WITH_DEFAULTS`. `MERGE_WITH_DEFAULTS` indicates that locally declared listeners should be merged with the default listeners. The merging algorithm ensures that duplicates are removed from the list

and that the resulting set of merged listeners is sorted according to the semantics of `AnnotationAwareOrderComparator` as described in the section called “Ordering `TestExecutionListeners`”. If a listener implements `Ordered` or is annotated with `@order` it can influence the position in which it is merged with the defaults; otherwise, locally declared listeners will simply be appended to the list of default listeners when merged.

For example, if the `MyCustomTestExecutionListener` class in the previous example configures its `order` value (for example, `500`) to be less than the order of the `ServletTestExecutionListener` (which happens to be `1000`), the `MyCustomTestExecutionListener` can then be automatically merged with the list of defaults *in front of* the `ServletTestExecutionListener`, and the previous example could be replaced with the following.

```
@ContextConfiguration
@TestExecutionListeners(
    listeners = MyCustomTestExecutionListener.class,
    mergeMode = MERGE_WITH_DEFAULTS
)
public class MyTest {
    // class body...
}
```

## 11.5.4 Context management

Each `TestContext` provides context management and caching support for the test instance it is responsible for. Test instances do not automatically receive access to the configured `ApplicationContext`. However, if a test class implements the `ApplicationContextAware` interface, a reference to the `ApplicationContext` is supplied to the test instance. Note that `AbstractJUnit4SpringContextTests` and `AbstractTestNGSpringContextTests` implement `ApplicationContextAware` and therefore provide access to the `ApplicationContext` automatically.

As an alternative to implementing the `ApplicationContextAware` interface, you can inject the application context for your test class through the `@Autowired` annotation on either a field or a setter method. For example:

```
*@RunWith(SpringRunner.class)***@ContextConfiguration*public
class MyTest { **@Autowired** private ApplicationContext applicationContext; // class
body...}
```

Similarly, if your test is configured to load a `WebApplicationContext`, you can inject the web application context into your test as follows:

```
*@RunWith(SpringRunner.class)***@WebAppConfiguration***@ContextConfiguration*publ
class MyWebAppTest { **@Autowired** private WebApplicationContext wac; // class
body...}
```

Dependency injection via `@Autowired` is provided by the `DependencyInjectionTestExecutionListener` which is configured by default (see [Section 11.5. “Dependency injection of test fixtures”](#)).

Test classes that use the `TestContext` framework do not need to extend any particular class or implement a specific interface to configure their application context. Instead, configuration is achieved simply by declaring the `@ContextConfiguration` annotation at the class level. If your test class does not explicitly declare application context resource `locations` or annotated `classes`, the configured `ContextLoader` determines how to load a context from a default location or default configuration classes. In addition to context resource `locations` and annotated `classes`, an application context can also be configured via application context `initializers`.

The following sections explain how to configure an `ApplicationContext` via XML configuration files, Groovy scripts, annotated classes (typically `@Configuration` classes), or context initializers using Spring's `@ContextConfiguration` annotation. Alternatively, you can implement and configure your own custom `SmartContextLoader` for advanced use cases.

## Context configuration with XML resources

To load an `ApplicationContext` for your tests using XML configuration files, annotate your test class with `@ContextConfiguration` and configure the `locations` attribute with an array that contains the resource locations of XML configuration metadata. A plain or relative path — for example `"context.xml"` — will be treated as a classpath resource that is relative to the package in which the test class is defined. A path starting with a slash is treated as an absolute classpath location, for example `"/org/example/config.xml"`. A path which represents a resource URL (i.e., a path prefixed with `classpath:`, `file:`, `http:`, etc.) will be used *as is*.

```
@RunWith(SpringRunner.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` supports an alias for the `locations` attribute through the standard Java `value` attribute. Thus, if you do not need to declare additional attributes in `@ContextConfiguration`, you can omit the declaration of the `locations` attribute name and declare the resource locations by using the shorthand format demonstrated in the following example.

```
@RunWith(SpringRunner.class)
@ContextConfiguration({"app-config.xml", "test-config.xml"})
public class MyTest {
    // class body...
}
```

If you omit both the `locations` and `value` attributes from the `@ContextConfiguration` annotation, the `TestContext` framework will attempt to detect a default XML resource location. Specifically, `GenericXmlContextLoader` and `GenericXmlWebContextLoader` detect a default location based on the name of the test class. If your class is named `com.example.MyTest`, `GenericXmlContextLoader` loads your application context from `"classpath:com/example/MyTest-context.xml"`.

```
package com.example;

@RunWith(SpringRunner.class)
// ApplicationContext will be loaded from
// "classpath:com/example/MyTest-context.xml"
@ContextConfiguration
public class MyTest {
    // class body...
}
```

## Context configuration with Groovy scripts

To load an `ApplicationContext` for your tests using Groovy scripts that utilize the [Groovy Bean Definition DSL](#), annotate your test class with `@ContextConfiguration` and configure the `locations` or `value` attribute with an array that contains the resource locations of Groovy scripts. Resource lookup semantics for Groovy scripts are the same as those described for [XML configuration files](#).

 Support for using Groovy scripts to load an `ApplicationContext` in the Spring `TestContext` Framework is enabled automatically if Groovy is on the classpath.

```
@RunWith(SpringRunner.class)
// ApplicationContext will be loaded from "/AppConfig.groovy" and
// "/TestConfig.groovy" in the root of the classpath
@ContextConfiguration({"AppConfig.groovy", "TestConfig.Groovy"})
public class MyTest {
    // class body...
}
```

If you omit both the `locations` and `value` attributes from the `@ContextConfiguration` annotation, the `TestContext` framework will attempt to detect a default Groovy script. Specifically, `GenericGroovyXmlContextLoader` and `GenericGroovyXmlWebContextLoader` detect a default location based on the name of the test class. If your class is named `com.example.MyTest`, the Groovy context loader will load your application context from `"classpath:com/example/MyTestContext.groovy"`.

```
package com.example;

@RunWith(SpringRunner.class)
// ApplicationContext will be loaded from
// "classpath:com/example/MyTestContext.groovy"
@Configuration
public class MyTest {
    // class body...
}
```

Both XML configuration files and Groovy scripts can be declared simultaneously via the `locations` or `value` attribute of `@ContextConfiguration`. If the path to a configured resource location ends with `.xml` it will be loaded using an `XmlBeanDefinitionReader`; otherwise it will be loaded using a `GroovyBeanDefinitionReader`. The following listing demonstrates how to combine both in an integration

```
test.*@RunWith(SpringRunner.class)*// ApplicationContext will be loaded from// "/app-
config.xml" and "/TestConfig.groovy"/*@ContextConfiguration({"/app-config.xml",
"/TestConfig.groovy" })*public class MyTest { // class body...}
```

## Context configuration with annotated classes

To load an `ApplicationContext` for your tests using *annotated classes* (see [Section 3.12, “Java-based container configuration”](#)), annotate your test class with `@ContextConfiguration` and configure the `classes` attribute with an array that contains references to annotated classes.

```
@RunWith(SpringRunner.class)
// ApplicationContext will be loaded from AppConfig and TestConfig
@Configuration(classes = {AppConfig.class, TestConfig.class})
public class MyTest {
    // class body...
}
```

The term ***annotated class*** can refer to any of the following. A class annotated with `@Configuration` A component (i.e., a class annotated with `@Component`, `@Service`, `@Repository`, etc.) A JSR-330 compliant class that is annotated with `javax.inject` annotations Any other class that contains `@Bean`-methods Consult the javadocs of `@Configuration` and `@Bean` for further information regarding the configuration and semantics of ***annotated classes***, paying special attention to the discussion of `@Bean Lite Mode`.

If you omit the `classes` attribute from the `@ContextConfiguration` annotation, the TestContext framework will attempt to detect the presence of default configuration classes. Specifically, `AnnotationConfigContextLoader` and `AnnotationConfigWebContextLoader` will detect all `static` nested classes of the test class that meet the requirements for configuration class implementations as specified in the `@Configuration` javadocs. In the following example, the `OrderServiceTest` class declares a `static` nested configuration class named `Config` that will be automatically used to load the `ApplicationContext` for the test class. Note that the name of the configuration class is arbitrary. In addition, a test class can contain more than one `static` nested configuration class if desired.

```

@RunWith(SpringRunner.class)
// ApplicationContext will be loaded from the
// static nested 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
    }
}

```

## Mixing XML, Groovy scripts, and annotated classes

It may sometimes be desirable to mix XML configuration files, Groovy scripts, and annotated classes (i.e., typically `@Configuration` classes) to configure an `ApplicationContext` for your tests. For example, if you use XML configuration in production, you may decide that you want to use `@Configuration` classes to configure specific Spring-managed components for your tests, or vice versa.

Furthermore, some third-party frameworks (like Spring Boot) provide first-class support for loading an `ApplicationContext` from different types of resources simultaneously (e.g., XML configuration files, Groovy scripts, and `@Configuration` classes). The Spring Framework historically has not supported this for standard deployments. Consequently, most of the

`SmartContextLoader` implementations that the Spring Framework delivers in the `spring-test` module support only one resource type per test context; however, this does not mean that you cannot use both. One exception to the general rule is that the

`GenericGroovyXmlContextLoader` and `GenericGroovyXmlWebContextLoader` support both XML configuration files and Groovy scripts simultaneously. Furthermore, third-party frameworks may choose to support the declaration of both `locations` and `classes` via `@ContextConfiguration`, and with the standard testing support in the `TestContext` framework, you have the following options.

If you want to use resource locations (e.g., XML or Groovy) and `@Configuration` classes to configure your tests, you will have to pick one as the *entry point*, and that one will have to include or import the other. For example, in XML or Groovy scripts you can include `@Configuration` classes via component scanning or define them as normal Spring beans; whereas, in a `@Configuration` class you can use `@ImportResource` to import XML configuration files or Groovy scripts. Note that this behavior is semantically equivalent to how you configure your application in production: in production configuration you will define either a set of XML or Groovy resource locations or a set of `@Configuration` classes that your production `ApplicationContext` will be loaded from, but you still have the freedom to include or import the other type of configuration.

## Context configuration with context initializers

To configure an `ApplicationContext` for your tests using context initializers, annotate your test class with `@ContextConfiguration` and configure the `initializers` attribute with an array that contains references to classes that implement `ApplicationContextInitializer`. The declared context initializers will then be used to initialize the

`ConfigurableApplicationContext` that is loaded for your tests. Note that the concrete `ConfigurableApplicationContext` type supported by each declared initializer must be compatible with the type of `ApplicationContext` created by the `SmartContextLoader` in use

(i.e., typically a `GenericApplicationContext`). Furthermore, the order in which the initializers are invoked depends on whether they implement Spring's `Ordered` interface or are annotated with Spring's `@Order` annotation or the standard `@Priority` annotation.

```
@RunWith(SpringRunner.class)
// ApplicationContext will be loaded from TestConfig
// and initialized by TestAppCtxInitializer
@ContextConfiguration(
    classes = TestConfig.class,
    initializers = TestAppCtxInitializer.class)
public class MyTest {
    // class body...
}
```

It is also possible to omit the declaration of XML configuration files, Groovy scripts, or annotated classes in `@ContextConfiguration` entirely and instead declare only `ApplicationContextInitializer` classes which are then responsible for registering beans in the context—for example, by programmatically loading bean definitions from XML files or configuration classes.

```
@RunWith(SpringRunner.class)
// ApplicationContext will be initialized by EntireAppInitializer
// which presumably registers beans in the context
@ContextConfiguration(initializers = EntireAppInitializer.class)
public class MyTest {
    // class body...
}
```

## Context configuration inheritance

`@ContextConfiguration` supports boolean `inheritLocations` and `inheritInitializers` attributes that denote whether resource locations or annotated classes and context initializers declared by superclasses should be *inherited*. The default value for both flags is `true`. This means that a test class inherits the resource locations or annotated classes as well as the context initializers declared by any superclasses. Specifically, the resource locations or annotated classes for a test class are appended to the list of resource locations or annotated classes declared by superclasses. Similarly, the initializers for a given test class will be added to the set of initializers defined by test superclasses. Thus, subclasses have the option of *extending* the resource locations, annotated classes, or context initializers.

If the `inheritLocations` or `inheritInitializers` attribute in `@ContextConfiguration` is set to `false`, the resource locations or annotated classes and the context initializers, respectively, for the test class *shadow* and effectively replace the configuration defined by superclasses.

In the following example that uses XML resource locations, the `ApplicationContext` for `ExtendedTest` will be loaded from "`base-config.xml`" and "`extended-config.xml`", in that order. Beans defined in "`extended-config.xml`" may therefore *override* (i.e., replace) those defined in "`base-config.xml`".

```
@RunWith(SpringRunner.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...
}
```

Similarly, in the following example that uses annotated classes, the `ApplicationContext` for `ExtendedTest` will be loaded from the `BaseConfig` and `ExtendedConfig` classes, in that order. Beans defined in `ExtendedConfig` may therefore override (i.e., replace) those defined in `BaseConfig`.

```
@RunWith(SpringRunner.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...
}
```

In the following example that uses context initializers, the `ApplicationContext` for `ExtendedTest` will be initialized using `BaseInitializer` and `ExtendedInitializer`. Note, however, that the order in which the initializers are invoked depends on whether they implement Spring's `Ordered` interface or are annotated with Spring's `@Order` annotation or the standard `@Priority` annotation.

```
@RunWith(SpringRunner.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...
}
```

## Context configuration with environment profiles

Spring 3.1 introduced first-class support in the framework for the notion of environments and profiles (a.k.a., *bean definition profiles*), and integration tests can be configured to activate particular bean definition profiles for various testing scenarios. This is achieved by annotating a test class with the `@ActiveProfiles` annotation and supplying a list of profiles that should be activated when loading the `ApplicationContext` for the test.

`@ActiveProfiles` may be used with any implementation of the new `SmartContextLoader` SPI, but `@ActiveProfiles` is not supported with implementations of the older `ContextLoader` SPI.

Let's take a look at some examples with XML configuration and `@Configuration` classes.

```

<!-- 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 profile="default">
        <jdbc:embedded-database id="dataSource">
            <jdbc:script
                location="classpath:com/bank/config/sql/schema.sql"/>
        </jdbc:embedded-database>
    </beans>

</beans>

```

```

package com.bank.service;

@RunWith(SpringRunner.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
    }
}

```

When `TransferServiceTest` is run, its `ApplicationContext` will be loaded from the `app-config.xml` configuration file in the root of the classpath. If you inspect `app-config.xml` you'll notice that the `accountRepository` bean has a dependency on a `dataSource` bean; however, `dataSource` is not defined as a top-level bean. Instead, `dataSource` is defined three times: in the *production* profile, the *dev* profile, and the *default* profile.

By annotating `TransferServiceTest` with `@ActiveProfiles("dev")` we instruct the Spring TestContext Framework to load the `ApplicationContext` with the active profiles set to `{"dev"}`. As a result, an embedded database will be created and populated with test data, and the `accountRepository` bean will be wired with a reference to the development `DataSource`. And that's likely what we want in an integration test.

It is sometimes useful to assign beans to a `default` profile. Beans within the default profile are only included when no other profile is specifically activated. This can be used to define *fallback* beans to be used in the application's default state. For example, you may explicitly provide a data source for `dev` and `production` profiles, but define an in-memory data source as a default when neither of these is active.

The following code listings demonstrate how to implement the same configuration and integration test but using `@Configuration` classes instead of 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(destroyMethod="")
    public DataSource dataSource() throws Exception {
        Context ctx = new InitialContext();
        return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");
    }
}

```

```

@Configuration
@Profile("default")
public class DefaultDataConfig {

    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.HSQL)
            .addScript("classpath:com/bank/config/sql/schema.sql")
            .build();
    }
}

```

```

@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(SpringRunner.class)
@ContextConfiguration(classes = {
    TransferServiceConfig.class,
    StandaloneDataConfig.class,
    JndiDataConfig.class,
    DefaultDataConfig.class})
@ActiveProfiles("dev")
public class TransferServiceTest {

    @Autowired
    private TransferService transferService;

    @Test
    public void testTransferService() {
        // test the transferService
    }
}

```

In this variation, we have split the XML configuration into four independent `@Configuration` classes:

- `TransferServiceConfig` : acquires a `dataSource` via dependency injection using `@Autowired`
- `StandaloneDataConfig` : defines a `dataSource` for an embedded database suitable for

### developer tests

- `JndiDataConfig` : defines a `dataSource` that is retrieved from JNDI in a production environment
- `DefaultDataConfig` : defines a `dataSource` for a default embedded database in case no profile is active

As with the XML-based configuration example, we still annotate `TransferServiceTest` with `@ActiveProfiles("dev")`, but this time we specify all four configuration classes via the `@ContextConfiguration` annotation. The body of the test class itself remains completely unchanged.

It is often the case that a single set of profiles is used across multiple test classes within a given project. Thus, to avoid duplicate declarations of the `@ActiveProfiles` annotation it is possible to declare `@ActiveProfiles` once on a base class, and subclasses will automatically inherit the `@ActiveProfiles` configuration from the base class. In the following example, the declaration of `@ActiveProfiles` (as well as other annotations) has been moved to an abstract superclass, `AbstractIntegrationTest`.

```
package com.bank.service;

@RunWith(SpringRunner.class)
@ContextConfiguration(classes = {
    TransferServiceConfig.class,
    StandaloneDataConfig.class,
    JndiDataConfig.class,
    DefaultDataConfig.class})
@ActiveProfiles("dev")
public abstract class AbstractIntegrationTest {
```

```
package com.bank.service;

// "dev" profile inherited from superclass
public class TransferServiceTest extends AbstractIntegrationTest {

    @Autowired
    private TransferService transferService;

    @Test
    public void testTransferService() {
        // test the transferService
    }
}
```

`@ActiveProfiles` also supports an `inheritProfiles` attribute that can be used to disable the inheritance of active profiles.

```

package com.bank.service;

// "dev" profile overridden with "production"
@ActiveProfiles(profiles = "production", inheritProfiles = false)
public class ProductionTransferServiceTest extends AbstractIntegrationTest {
    // test body
}

```

Furthermore, it is sometimes necessary to resolve active profiles for tests *programmatically* instead of declaratively — for example, based on:

- the current operating system
- whether tests are being executed on a continuous integration build server
- the presence of certain environment variables
- the presence of custom class-level annotations
- etc.

To resolve active bean definition profiles programmatically, simply implement a custom `ActiveProfilesResolver` and register it via the `resolver` attribute of `@ActiveProfiles`. The following example demonstrates how to implement and register a custom `OperatingSystemActiveProfilesResolver`. For further information, refer to the corresponding javadocs.

```

package com.bank.service;

// "dev" profile overridden programmatically via a custom resolver
@ActiveProfiles(
    resolver = OperatingSystemActiveProfilesResolver.class,
    inheritProfiles = false)
public class TransferServiceTest extends AbstractIntegrationTest {
    // test body
}

```

```

package com.bank.service.test;

public class OperatingSystemActiveProfilesResolver implements ActiveProfilesResolver {

    @Override
    String[] resolve(Class<?> testClass) {
        String profile = ...;
        // determine the value of profile based on the operating system
        return new String[] {profile};
    }
}

```

## Context configuration with test property sources

Spring 3.1 introduced first-class support in the framework for the notion of an environment with a hierarchy of *property sources*, and since Spring 4.1 integration tests can be configured with test-specific property sources. In contrast to the `@PropertySource` annotation used on `@Configuration` classes, the `@TestPropertySource` annotation can be declared on a test class to declare resource locations for test properties files or *inlined* properties. These test property sources will be added to the set of `PropertySources` in the `Environment` for the `ApplicationContext` loaded for the annotated integration test.

`@TestPropertySource` may be used with any implementation of the `SmartContextLoader` SPI, but `@TestPropertySource` is not supported with implementations of the older `contextLoader` SPI. Implementations of `SmartContextLoader` gain access to merged test property source values via the `getPropertySourceLocations()` and `getPropertySourceProperties()` methods in `MergedContextConfiguration`.

### Declaring test property sources

Test properties files can be configured via the `locations` or `value` attribute of `@TestPropertySource` as shown in the following example.

Both traditional and XML-based properties file formats are supported—for example,

```
"classpath:/com/example/test.properties" or "file:///path/to/file.xml".
```

Each path will be interpreted as a Spring `Resource`. A plain path—for example, `"test.properties"`—will be treated as a classpath resource that is *relative* to the package in which the test class is defined. A path starting with a slash will be treated as an *absolute* classpath resource, for example: `"/org/example/test.xml"`. A path which references a URL (e.g., a path prefixed with `classpath:`, `file:`, `http:`, etc.) will be loaded using the specified resource protocol. Resource location wildcards (e.g. `**/*.properties`) are not permitted: each location must evaluate to exactly one `.properties` or `.xml` resource.

```
@ContextConfiguration
@TestPropertySource("/test.properties")
public class MyIntegrationTests {
    // class body...
}
```

*Inlined* properties in the form of key-value pairs can be configured via the `properties` attribute of `@TestPropertySource` as shown in the following example. All key-value pairs will be added to the enclosing `Environment` as a single test `PropertySource` with the highest precedence.

The supported syntax for key-value pairs is the same as the syntax defined for entries in a Java properties file:

- "key=value"
- "key:value"
- "key value"

```
@ContextConfiguration
@TestPropertySource(properties = {"timezone = GMT", "port: 4242"})
public class MyIntegrationTests {
    // class body...
}
```

## Default properties file detection

If `@TestPropertySource` is declared as an empty annotation (i.e., without explicit values for the `locations` or `properties` attributes), an attempt will be made to detect a *default* properties file relative to the class that declared the annotation. For example, if the annotated test class is `com.example.MyTest`, the corresponding default properties file is `"classpath:com/example/MyTest.properties"`. If the default cannot be detected, an `IllegalStateException` will be thrown.

## Precedence

Test property sources have higher precedence than those loaded from the operating system's environment or Java system properties as well as property sources added by the application declaratively via `@PropertySource` or programmatically. Thus, test property sources can be used to selectively override properties defined in system and application property sources. Furthermore, inlined properties have higher precedence than properties loaded from resource locations.

In the following example, the `timezone` and `port` properties as well as any properties defined in `"/test.properties"` will override any properties of the same name that are defined in system and application property sources. Furthermore, if the `"/test.properties"` file defines entries for the `timezone` and `port` properties those will be overridden by the *inlined* properties declared via the `properties` attribute.

```
@ContextConfiguration
@TestPropertySource(
    locations = "/test.properties",
    properties = {"timezone = GMT", "port: 4242"})
)
public class MyIntegrationTests {
    // class body...
}
```

## Inheriting and overriding test property sources

`@TestPropertySource` supports boolean `inheritLocations` and `inheritProperties` attributes that denote whether resource locations for properties files and inlined properties declared by superclasses should be *inherited*. The default value for both flags is `true`. This means that a test class inherits the locations and inlined properties declared by any superclasses. Specifically, the locations and inlined properties for a test class are appended to the locations and inlined properties declared by superclasses. Thus, subclasses have the option of *extending* the locations and inlined properties. Note that properties that appear later will *shadow* (i.e., override) properties of the same name that appear earlier. In addition, the aforementioned precedence rules apply for inherited test property sources as well.

If the `inheritLocations` or `inheritProperties` attribute in `@TestPropertySource` is set to `false`, the locations or inlined properties, respectively, for the test class *shadow* and effectively replace the configuration defined by superclasses.

In the following example, the `ApplicationContext` for `BaseTest` will be loaded using only the `"base.properties"` file as a test property source. In contrast, the `ApplicationContext` for `ExtendedTest` will be loaded using the `"base.properties"` and `"extended.properties"` files as test property source locations.

```

@TestPropertySource("base.properties")
@ContextConfiguration
public class BaseTest {
    // ...
}

@TestPropertySource("extended.properties")
@ContextConfiguration
public class ExtendedTest extends BaseTest {
    // ...
}

```

In the following example, the `ApplicationContext` for `BaseTest` will be loaded using only the *inlined* `key1` property. In contrast, the `ApplicationContext` for `ExtendedTest` will be loaded using the *inlined* `key1` and `key2` properties.

```

@TestPropertySource(properties = "key1 = value1")
@ContextConfiguration
public class BaseTest {
    // ...
}

@TestPropertySource(properties = "key2 = value2")
@ContextConfiguration
public class ExtendedTest extends BaseTest {
    // ...
}

```

## Loading a WebApplicationContext

Spring 3.2 introduced support for loading a `WebApplicationContext` in integration tests. To instruct the TestContext framework to load a `WebApplicationContext` instead of a standard `ApplicationContext`, simply annotate the respective test class with `@webAppConfiguration`.

The presence of `@webAppConfiguration` on your test class instructs the TestContext framework (TCF) that a `WebApplicationContext` (WAC) should be loaded for your integration tests. In the background the TCF makes sure that a `MockServletContext` is created and supplied to your test's WAC. By default the base resource path for your `MockServletContext` will be set to "`src/main/webapp`". This is interpreted as a path relative to the root of your JVM (i.e., normally the path to your project). If you're familiar with the directory structure of a web application in a Maven project, you'll know that "`src/main/webapp`" is the default location for the root of your WAR. If you need to override this default, simply provide an alternate path to the `@WebAppConfiguration` annotation (e.g., `@WebAppConfiguration("src/test/webapp")`). If you wish to reference a base resource path from the classpath instead of the file system, just use Spring's `classpath:` prefix.

Please note that Spring's testing support for `WebApplicationContexts` is on par with its support for standard `ApplicationContexts`. When testing with a `WebApplicationContext` you are free to declare XML configuration files, Groovy scripts, or `@Configuration` classes via `@ContextConfiguration`. You are of course also free to use any other test annotations such as `@ActiveProfiles`, `@TestExecutionListeners`, `@Sql`, `@Rollback`, etc.

The following examples demonstrate some of the various configuration options for loading a `WebApplicationContext`.

### Conventions.

```

@RunWith(SpringRunner.class)

// defaults to "file:src/main/webapp"
@WebAppConfiguration

// detects "WacTests-context.xml" in same package
// or static nested @Configuration class
@ContextConfiguration

public class WacTests {
    //...
}

```

The above example demonstrates the TestContext framework's support for *convention over configuration*. If you annotate a test class with `@WebAppConfiguration` without specifying a resource base path, the resource path will effectively default to `"file:src/main/webapp"`. Similarly, if you declare `@ContextConfiguration` without specifying resource locations, annotated classes, or context initializers, Spring will attempt to detect the presence of your configuration using conventions (i.e., `"WacTests-context.xml"` in the same package as the `WacTests` class or static nested `@Configuration` classes).

### Default resource semantics.

```

@RunWith(SpringRunner.class)

// file system resource
@WebAppConfiguration("webapp")

// classpath resource
@ContextConfiguration("/spring/test-servlet-config.xml")

public class WacTests {
    //...
}

```

This example demonstrates how to explicitly declare a resource base path with `@WebAppConfiguration` and an XML resource location with `@ContextConfiguration`. The important thing to note here is the different semantics for paths with these two annotations. By default, `@WebAppConfiguration` resource paths are file system based; whereas, `@ContextConfiguration` resource locations are classpath based.

### Explicit resource semantics.

```
@RunWith(SpringRunner.class)

// classpath resource
@WebAppConfiguration("classpath:test-web-resources")

// file system resource
@ContextConfiguration("file:src/main/webapp/WEB-INF/servlet-config.xml")

public class WacTests {
    //...
}
```

In this third example, we see that we can override the default resource semantics for both annotations by specifying a Spring resource prefix. Contrast the comments in this example with the previous example.

To provide comprehensive web testing support, Spring 3.2 introduced a `ServletTestExecutionListener` that is enabled by default. When testing against a `WebApplicationContext` this `TestExecutionListener` sets up default thread-local state via Spring Web's `RequestContextHolder` before each test method and creates a `MockHttpServletRequest`, `MockHttpServletResponse`, and `ServletWebRequest` based on the base resource path configured via `@WebAppConfiguration`. `ServletTestExecutionListener` also ensures that the `MockHttpServletResponse` and `ServletWebRequest` can be injected into the test instance, and once the test is complete it cleans up thread-local state.

Once you have a `WebApplicationContext` loaded for your test you might find that you need to interact with the web mocks—for example, to set up your test fixture or to perform assertions after invoking your web component. The following example demonstrates which mocks can be autowired into your test instance. Note that the `WebApplicationContext` and `MockServletContext` are both cached across the test suite; whereas, the other mocks are managed per test method by the `ServletTestExecutionListener`.

## Injecting mocks.

```

@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;

    //...
}

```

## Context caching

Once the TestContext framework loads an `ApplicationContext` (or `WebApplicationContext`) for a test, that context will be cached and reused for *all* subsequent tests that declare the same unique context configuration within the same test suite. To understand how caching works, it is important to understand what is meant by *unique* and *test suite*.

An `ApplicationContext` can be *uniquely* identified by the combination of configuration parameters that are used to load it. Consequently, the unique combination of configuration parameters are used to generate a *key* under which the context is cached. The TestContext framework uses the following configuration parameters to build the context cache key:

- `locations` (*from `@ContextConfiguration`*)
- `classes` (*from `@ContextConfiguration`*)
- `contextInitializerClasses` (*from `@ContextConfiguration`*)
- `contextCustomizers` (*from `ContextCustomizerFactory`*)
- `contextLoader` (*from `@ContextConfiguration`*)
- `parent` (*from `@ContextHierarchy`*)
- `activeProfiles` (*from `@ActiveProfiles`*)
- `propertySourceLocations` (*from `@TestPropertySource`*)
- `propertySourceProperties` (*from `@TestPropertySource`*)

- `resourceBasePath` (*from @WebAppConfiguration*)

For example, if `TestClassA` specifies `{"app-config.xml", "test-config.xml"}` for the `locations` (or `value`) attribute of `@ContextConfiguration`, the `TestContext` framework will load the corresponding `ApplicationContext` and store it in a `static` context cache under a key that is based solely on those locations. So if `TestClassB` also defines `{"app-config.xml", "test-config.xml"}` for its locations (either explicitly or implicitly through inheritance) but does not define `@WebAppConfiguration`, a different `ContextLoader`, different active profiles, different context initializers, different test property sources, or a different parent context, then the same `ApplicationContext` will be shared by both test classes. This means that the setup cost for loading an application context is incurred only once (per test suite), and subsequent test execution is much faster.



The Spring `TestContext` framework stores application contexts in a *static* cache. This means that the context is literally stored in a `static` variable. In other words, if tests execute in separate processes the static cache will be cleared between each test execution, and this will effectively disable the caching mechanism. To benefit from the caching mechanism, all tests must run within the same process or test suite. This can be achieved by executing all tests as a group within an IDE. Similarly, when executing tests with a build framework such as Ant, Maven, or Gradle it is important to make sure that the build framework does not *fork* between tests. For example, if the `forkMode` for the Maven Surefire plug-in is set to `always` or `pertest`, the `TestContext` framework will not be able to cache application contexts between test classes and the build process will run significantly slower as a result.

Since Spring Framework 4.3, the size of the context cache is bounded with a default maximum size of 32. Whenever the maximum size is reached, a *least recently used*(LRU) eviction policy is used to evict and close stale contexts. The maximum size can be configured from the command line or a build script by setting a JVM system property named `spring.test.context.cache.maxSize`. As an alternative, the same property can be set programmatically via the `SpringProperties` API.

Since having a large number of application contexts loaded within a given test suite can cause the suite to take an unnecessarily long time to execute, it is often beneficial to know exactly how many contexts have been loaded and cached. To view the statistics for the underlying context cache, simply set the log level for the `org.springframework.test.context.cache` logging category to `DEBUG`.

In the unlikely case that a test corrupts the application context and requires reloading—for example, by modifying a bean definition or the state of an application object—you can annotate your test class or test method with `@DirtiesContext` (see the discussion of `@DirtiesContext` in [Section 11.4.1, “Spring Testing Annotations”](#)). This instructs Spring to remove the context from the cache and rebuild the application context before executing the

next test. Note that support for the `@DirtiesContext` annotation is provided by the `DirtiesContextBeforeModesTestExecutionListener` and the `DirtiesContextTestExecutionListener` which are enabled by default.

## Context hierarchies

When writing integration tests that rely on a loaded Spring `ApplicationContext`, it is often sufficient to test against a single context; however, there are times when it is beneficial or even necessary to test against a hierarchy of `ApplicationContext`s. For example, if you are developing a Spring MVC web application you will typically have a root

`WebApplicationContext` loaded via Spring's `ContextLoaderListener` and a child `WebApplicationContext` loaded via Spring's `DispatcherServlet`. This results in a parent-child context hierarchy where shared components and infrastructure configuration are declared in the root context and consumed in the child context by web-specific components. Another use case can be found in Spring Batch applications where you often have a parent context that provides configuration for shared batch infrastructure and a child context for the configuration of a specific batch job.

Since Spring Framework 3.2.2, it is possible to write integration tests that use context hierarchies by declaring context configuration via the `@ContextHierarchy` annotation, either on an individual test class or within a test class hierarchy. If a context hierarchy is declared on multiple classes within a test class hierarchy it is also possible to merge or override the context configuration for a specific, named level in the context hierarchy. When merging configuration for a given level in the hierarchy the configuration resource type (i.e., XML configuration files or annotated classes) must be consistent; otherwise, it is perfectly acceptable to have different levels in a context hierarchy configured using different resource types.

The following JUnit 4 based examples demonstrate common configuration scenarios for integration tests that require the use of context hierarchies.

`ControllerIntegrationTests` represents a typical integration testing scenario for a Spring MVC web application by declaring a context hierarchy consisting of two levels, one for the root `WebApplicationContext` (loaded using the `TestAppConfig @Configuration` class) and one for the `dispatcher servlet` `WebApplicationContext` (loaded using the `WebConfig @Configuration` class). The `WebApplicationContext` that is `autowired` into the test instance is the one for the child context (i.e., the lowest context in the hierarchy).

```

@RunWith(SpringRunner.class)
@WebAppConfiguration
@ContextHierarchy({
    @ContextConfiguration(classes = TestAppConfig.class),
    @ContextConfiguration(classes = WebConfig.class)
})
public class ControllerIntegrationTests {

    @Autowired
    private WebApplicationContext wac;

    // ...
}

```

The following test classes define a context hierarchy within a test class hierarchy.

`AbstractWebTests` declares the configuration for a root `webApplicationContext` in a Spring-powered web application. Note, however, that `AbstractWebTests` does not declare `@ContextHierarchy`; consequently, subclasses of `AbstractWebTests` can optionally participate in a context hierarchy or simply follow the standard semantics for `@ContextConfiguration`. `SapWebServiceTests` and `RestWebServiceTests` both extend `AbstractWebTests` and define a context hierarchy via `@ContextHierarchy`. The result is that three application contexts will be loaded (one for each declaration of `@ContextConfiguration`), and the application context loaded based on the configuration in `AbstractWebTests` will be set as the parent context for each of the contexts loaded for the concrete subclasses.

```

@RunWith(SpringRunner.class)
@WebAppConfiguration
@ContextConfiguration("file:src/main/webapp/WEB-INF/applicationContext.xml")
public abstract class AbstractWebTests {}

@ContextHierarchy(@ContextConfiguration("/spring/soap-ws-config.xml"))
public class SapWebServiceTests extends AbstractWebTests {}

@ContextHierarchy(@ContextConfiguration("/spring/rest-ws-config.xml"))
public class RestWebServiceTests extends AbstractWebTests {}

```

The following classes demonstrate the use of *named* hierarchy levels in order to *merge* the configuration for specific levels in a context hierarchy. `BaseTests` defines two levels in the hierarchy, `parent` and `child`. `ExtendedTests` extends `BaseTests` and instructs the Spring TestContext Framework to merge the context configuration for the `child` hierarchy level, simply by ensuring that the names declared via the `name` attribute in `@ContextConfiguration` are both "child". The result is that three application contexts will be loaded: one for `"/app-config.xml"`, one for `"/user-config.xml"`, and one for `{"/user-`

`config.xml", "/order-config.xml"}]`. As with the previous example, the application context loaded from `"/app-config.xml"` will be set as the parent context for the contexts loaded from `"/user-config.xml"` and `"/user-config.xml", "/order-config.xml"}]`.

```
@RunWith(SpringRunner.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 {}
```

In contrast to the previous example, this example demonstrates how to *override* the configuration for a given named level in a context hierarchy by setting the `inheritLocations` flag in `@ContextConfiguration` to `false`. Consequently, the application context for `ExtendedTests` will be loaded only from `"/test-user-config.xml"` and will have its parent set to the context loaded from `"/app-config.xml"`.

```
@RunWith(SpringRunner.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 {}
```

If `@DirtiesContext` is used in a test whose context is configured as part of a context hierarchy, the `hierarchyMode` flag can be used to control how the context cache is cleared. For further details consult the discussion of `@DirtiesContext` in [Spring Testing Annotations](#) and the `@DirtiesContext` javadocs.

## 11.5.5 Dependency injection of test fixtures

When you use the `DependencyInjectionTestExecutionListener` — which is configured by default — the dependencies of your test instances are *injected* from beans in the application context that you configured with `@ContextConfiguration`. You may use setter injection, field injection, or both, depending on which annotations you choose and whether you place them on setter methods or fields. For consistency with the annotation support introduced in Spring 2.5 and 3.0, you can use Spring's `@Autowired` annotation or the `@Inject` annotation from JSR 330.



The TestContext framework does not instrument the manner in which a test instance is instantiated. Thus the use of `@Autowired` or `@Inject` for constructors has no effect for test classes.

Because `@Autowired` is used to perform *autowiring by type*, if you have multiple bean definitions of the same type, you cannot rely on this approach for those particular beans. In that case, you can use `@Autowired` in conjunction with `@Qualifier`. As of Spring 3.0 you may also choose to use `@Inject` in conjunction with `@Named`. Alternatively, if your test class has access to its `ApplicationContext`, you can perform an explicit lookup by using (for example) a call to `applicationContext.getBean("titleRepository")`.

If you do not want dependency injection applied to your test instances, simply do not annotate fields or setter methods with `@Autowired` or `@Inject`. Alternatively, you can disable dependency injection altogether by explicitly configuring your class with

`@TestExecutionListeners` and omitting `DependencyInjectionTestExecutionListener.class` from the list of listeners.

Consider the scenario of testing a `HibernateTitleRepository` class, as outlined in the [Goals](#) section. The next two code listings demonstrate the use of `@Autowired` on fields and setter methods. The application context configuration is presented after all sample code listings.



The dependency injection behavior in the following code listings is not specific to JUnit 4. The same DI techniques can be used in conjunction with any testing framework. The following examples make calls to static assertion methods such as `assertNotNull()` but without prepending the call with `Assert`. In such cases, assume that the method was properly imported through an `import static` declaration that is not shown in the example.

The first code listing shows a JUnit 4 based implementation of the test class that uses `@Autowired` for field injection.

```

@RunWith(SpringRunner.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);
    }
}

```

Alternatively, you can configure the class to use `@Autowired` for setter injection as seen below.

```

@RunWith(SpringRunner.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);
    }
}

```

The preceding code listings use the same XML context file referenced by the `@ContextConfiguration` annotation (that is, `repository-config.xml`), which looks like this:

```

<?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.hibernate5.LocalSessionFactoryBean">
        <!-- configuration elided for brevity -->
    </bean>

</beans>

```



If you are extending from a Spring-provided test base class that happens to use `@Autowired` on one of its setter methods, you might have multiple beans of the affected type defined in your application context: for example, multiple `DataSource` beans. In such a case, you can override the setter method and use the `@Qualifier` annotation to indicate a specific target bean as follows, but make sure to delegate to the overridden method in the superclass as well.

```
// ... *@Autowired* *@Override* public void
setDataSource(**@Qualifier("myDataSource")** DataSource dataSource) {
    **super**.setDataSource(dataSource); } // ... The specified qualifier value indicates the
specific DataSource bean to inject, narrowing the set of type matches to a specific bean.
Its value is matched against <qualifier> declarations within the corresponding <bean>
definitions. The bean name is used as a fallback qualifier value, so you may effectively
also point to a specific bean by name there (as shown above, assuming that
"myDataSource" is the bean id).
```

## 11.5.6 Testing request and session scoped beans

[Request and session scoped beans](#) have been supported by Spring since the early years, and since Spring 3.2 it's a breeze to test your request-scoped and session-scoped beans by following these steps.

- Ensure that a `WebApplicationContext` is loaded for your test by annotating your test class with `@WebAppConfiguration`.
- Inject the mock request or session into your test instance and prepare your test fixture as appropriate.
- Invoke your web component that you retrieved from the configured `WebApplicationContext` (i.e., via dependency injection).
- Perform assertions against the mocks.

The following code snippet displays the XML configuration for a login use case. Note that the `userService` bean has a dependency on a request-scoped `loginAction` bean. Also, the `LoginAction` is instantiated using [SpEL expressions](#) that retrieve the username and password from the current HTTP request. In our test, we will want to configure these request parameters via the mock managed by the `TestContext` framework.

### Request-scoped bean configuration.

```
<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>
```

In `RequestScopedBeanTests` we inject both the `userService` (i.e., the subject under test) and the `MockHttpServletRequest` into our test instance. Within our `requestScope()` test method we set up our test fixture by setting request parameters in the provided `MockHttpServletRequest`. When the `loginUser()` method is invoked on our `userService` we are assured that the user service has access to the request-scoped `loginAction` for the current `MockHttpServletRequest` (i.e., the one we just set parameters in). We can then perform assertions against the results based on the known inputs for the username and password.

### Request-scoped bean test.

```

@RunWith(SpringRunner.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
    }
}

```

The following code snippet is similar to the one we saw above for a request-scoped bean; however, this time the `userService` bean has a dependency on a session-scoped `userPreferences` bean. Note that the `UserPreferences` bean is instantiated using a SpEL expression that retrieves the *theme* from the current HTTP session. In our test, we will need to configure a theme in the mock session managed by the TestContext framework.

### Session-scoped bean configuration.

```

<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>

```

In `sessionScopedBeanTests` we inject the `UserService` and the `MockHttpSession` into our test instance. Within our `sessionScope()` test method we set up our test fixture by setting the expected "theme" attribute in the provided `MockHttpSession`. When the `processUserPreferences()` method is invoked on our `userService` we are assured that the

user service has access to the session-scoped `userPreferences` for the current `MockHttpSession`, and we can perform assertions against the results based on the configured theme.

### Session-scoped bean test.

```
@RunWith(SpringRunner.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
    }
}
```

## 11.5.7 Transaction management

In the TestContext framework, transactions are managed by the `TransactionalTestExecutionListener` which is configured by default, even if you do not explicitly declare `@TestExecutionListeners` on your test class. To enable support for transactions, however, you must configure a `PlatformTransactionManager` bean in the `ApplicationContext` that is loaded via `@ContextConfiguration` semantics (further details are provided below). In addition, you must declare Spring's `@Transactional` annotation either at the class or method level for your tests.

## Test-managed transactions

*Test-managed transactions* are transactions that are managed *declaratively* via the `TransactionalTestExecutionListener` or *programmatically* via `TestTransaction` (see below). Such transactions should not be confused with *Spring-managed transactions* (i.e., those managed directly by Spring within the `ApplicationContext` loaded for tests) or *application-managed transactions* (i.e., those managed programmatically within application code that is invoked via tests). Spring-managed and application-managed transactions will typically participate in test-managed transactions; however, caution should be taken if Spring-

managed or application-managed transactions are configured with any *propagation* type other than `REQUIRED` or `SUPPORTS` (see the discussion on [transaction propagation](#) for details).

## Enabling and disabling transactions

Annotating a test method with `@Transactional` causes the test to be run within a transaction that will, by default, be automatically rolled back after completion of the test. If a test class is annotated with `@Transactional`, each test method within that class hierarchy will be run within a transaction. Test methods that are not annotated with `@Transactional` (at the class or method level) will not be run within a transaction. Furthermore, tests that are annotated with `@Transactional` but have the `propagation` type set to `NOT_SUPPORTED` will not be run within a transaction.

*Note that `AbstractTransactionalJUnit4SpringContextTests` and `AbstractTransactionalTestNGSpringContextTests` are preconfigured for transactional support at the class level.*

The following example demonstrates a common scenario for writing an integration test for a Hibernate-based `userRepository`. As explained in [the section called “Transaction rollback and commit behavior”](#), there is no need to clean up the database after the `createUser()` method is executed since any changes made to the database will be automatically rolled back by the `TransactionalTestExecutionListener`. See [Section 11.7, “PetClinic Example”](#) for an additional example.

```

@RunWith(SpringRunner.class)
@ContextConfiguration(classes = TestConfig.class)
@Transactional
public class HibernateUserRepositoryTests {

    @Autowired
    HibernateUserRepository repository;

    @Autowired
    SessionFactory sessionFactory;

    JdbcTemplate jdbcTemplate;

    @Autowired
    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    @Test
    public void createUser() {
        // track initial state in test database:
        final int count = countRowsInTable("user");

        User user = new User(...);
        repository.save(user);

        // Manual flush is required to avoid false positive in test
        sessionFactory.getCurrentSession().flush();
        assertNumUsers(count + 1);
    }

    protected int countRowsInTable(String tableName) {
        return Jdbc TestUtils.countRowsInTable(this.jdbcTemplate, tableName);
    }

    protected void assertNumUsers(int expected) {
        assertEquals("Number of rows in the [user] table.", expected, countRowsInTable
        ("user"));
    }
}

```

## Transaction rollback and commit behavior

By default, test transactions will be automatically rolled back after completion of the test; however, transactional commit and rollback behavior can be configured declaratively via the `@Commit` and `@Rollback` annotations. See the corresponding entries in the [annotation support](#) section for further details.

## Programmatic transaction management

Since Spring Framework 4.1, it is possible to interact with test-managed transactions *programmatically* via the static methods in `TestTransaction`. For example, `TestTransaction` may be used within `test` methods, `before` methods, and `after` methods to start or end the current test-managed transaction or to configure the current test-managed transaction for rollback or commit. Support for `TestTransaction` is automatically available whenever the `TransactionalTestExecutionListener` is enabled.

The following example demonstrates some of the features of `TestTransaction`. Consult the javadocs for `TestTransaction` for further details.

```
@ContextConfiguration(classes = TestConfig.class)
public class ProgrammaticTransactionManagementTests extends
    AbstractTransactionalJUnit4SpringContextTests {

    @Test
    public void transactionalTest() {
        // assert initial state in test database:
        assertEquals(2, countRowsInTable("user"));

        // changes to the database will be committed!
        TestTransaction.flagForCommit();
        TestTransaction.end();
        assertFalse(TestTransaction.isActive());
        assertEquals(0, countRowsInTable("user"));

        TestTransaction.start();
        // perform other actions against the database that will
        // be automatically rolled back after the test completes...
    }

    protected void assertEquals(int expected) {
        assertEquals("Number of rows in the [user] table.", expected, countRowsInTable("user"));
    }
}
```

## Executing code outside of a transaction

Occasionally you need to execute certain code before or after a transactional test method but outside the transactional context—for example, to verify the initial database state prior to execution of your test or to verify expected transactional commit behavior after test execution (if the test was configured to commit the transaction).

`TransactionalTestExecutionListener` supports the `@BeforeTransaction` and `@AfterTransaction` annotations exactly for such scenarios. Simply annotate any `void`

method in a test class or any `void` default method in a test interface with one of these annotations, and the `TransactionalTestExecutionListener` ensures that your *before transaction method* or *after transaction method* is executed at the appropriate time.



Any *before methods* (such as methods annotated with JUnit 4's `@Before`) and any *after methods* (such as methods annotated with JUnit 4's `@After`) are executed *within* a transaction. In addition, methods annotated with `@BeforeTransaction` or `@AfterTransaction` are naturally not executed for test methods that are not configured to run within a transaction.

## Configuring a transaction manager

`TransactionalTestExecutionListener` expects a `PlatformTransactionManager` bean to be defined in the Spring `ApplicationContext` for the test. In case there are multiple instances of `PlatformTransactionManager` within the test's `ApplicationContext`, a *qualifier* may be declared via `@Transactional("myTxMgr")` or `@Transactional(transactionManager = "myTxMgr")`, or `TransactionManagementConfigurer` can be implemented by an `@Configuration` class. Consult the javadocs for `TestContextTransactionUtils.retrieveTransactionManager()` for details on the algorithm used to look up a transaction manager in the test's `ApplicationContext`.

## Demonstration of all transaction-related annotations

The following JUnit 4 based example displays a fictitious integration testing scenario highlighting all transaction-related annotations. The example is **not** intended to demonstrate best practices but rather to demonstrate how these annotations can be used. Consult the [annotation support](#) section for further information and configuration examples. [Transaction management for `@sql`](#) contains an additional example using `@sql` for declarative SQL script execution with default transaction rollback semantics.

```
@RunWith(SpringRunner.class)
@ContextConfiguration
@Transactional(transactionManager = "txMgr")
@Commit
public class FictitiousTransactionalTest {

    @BeforeTransaction
    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 @Commit setting
    @Rollback
    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
    void verifyFinalDatabaseState() {
        // logic to verify the final state after transaction has rolled back
    }
}
```

When you test application code that manipulates the state of a Hibernate session or JPA persistence context, make sure to *flush* the underlying unit of work within test methods that execute that code. Failing to flush the underlying unit of work can produce *false positives*: your test may pass, but the same code throws an exception in a live, production environment. In the following Hibernate-based example test case, one method demonstrates a false positive, and the other method correctly exposes the results of flushing the session. Note that this applies to any ORM frameworks that maintain an in-memory *unit of work*.

```
// ... *@Autowired*SessionFactory sessionFactory; *@Transactional**@Test* // no expected exception!public void falsePositive() { updateEntityInHibernateSession(); // False positive: an exception will be thrown once the Hibernate // Session is finally flushed (i.e., in production code)} *@Transactional**@Test(expected = ...) *public void updateWithSessionFlush() { updateEntityInHibernateSession(); // Manual flush is required to avoid false positive in test sessionFactory.getCurrentSession().flush(); } // ... Or for JPA: // ... *@PersistenceContext*EntityManager entityManager; *@Transactional**@Test* // no expected exception!public void falsePositive() { updateEntityInJpaPersistenceContext(); // False positive: an exception will be thrown once the JPA // EntityManager is finally flushed (i.e., in production code)} *@Transactional**@Test(expected = ...) *public void updateWithEntityManagerFlush() { updateEntityInJpaPersistenceContext(); // Manual flush is required to avoid false positive in test entityManager.flush(); } // ...
```

## 11.5.8 Executing SQL scripts

When writing integration tests against a relational database, it is often beneficial to execute SQL scripts to modify the database schema or insert test data into tables. The `spring-jdbc` module provides support for *initializing* an embedded or existing database by executing SQL scripts when the Spring `ApplicationContext` is loaded. See [Section 15.8, “Embedded database support”](#) and [Section 15.8.5, “Testing data access logic with an embedded database”](#) for details.

Although it is very useful to initialize a database for testing *once* when the `ApplicationContext` is loaded, sometimes it is essential to be able to modify the database *during* integration tests. The following sections explain how to execute SQL scripts programmatically and declaratively during integration tests.

## Executing SQL scripts programmatically

Spring provides the following options for executing SQL scripts programmatically within integration test methods.

- `org.springframework.jdbc.datasource.init.ScriptUtils`
- `org.springframework.jdbc.datasource.init.ResourceDatabasePopulator`
- `org.springframework.test.context.junit4.AbstractTransactionalJUnit4SpringContextTests`
- `org.springframework.test.context.testng.AbstractTransactionalTestNGSpringContextTests`

`ScriptUtils` provides a collection of static utility methods for working with SQL scripts and is mainly intended for internal use within the framework. However, if you require full control over how SQL scripts are parsed and executed, `ScriptUtils` may suit your needs better than some of the other alternatives described below. Consult the javadocs for individual methods in `ScriptUtils` for further details.

`ResourceDatabasePopulator` provides a simple object-based API for programmatically populating, initializing, or cleaning up a database using SQL scripts defined in external resources. `ResourceDatabasePopulator` provides options for configuring the character encoding, statement separator, comment delimiters, and error handling flags used when parsing and executing the scripts, and each of the configuration options has a reasonable default value. Consult the javadocs for details on default values. To execute the scripts configured in a `ResourceDatabasePopulator`, you can invoke either the `populate(Connection)` method to execute the populator against a `java.sql.Connection` or the `execute(DataSource)` method to execute the populator against a `javax.sql.DataSource`. The following example specifies SQL scripts for a test schema and test data, sets the statement separator to `"@@"`, and then executes the scripts against a `DataSource`.

```
@Test
public void databaseTest {
    ResourceDatabasePopulator populator = new ResourceDatabasePopulator();
    populator.addScripts(
        new ClassPathResource("test-schema.sql"),
        new ClassPathResource("test-data.sql"));
    populator.setSeparator("@@");
    populator.execute(this.dataSource);
    // execute code that uses the test schema and data
}
```

Note that `ResourceDatabasePopulator` internally delegates to `ScriptUtils` for parsing and executing SQL scripts. Similarly, the `executeSqlScript(...)` methods in

`AbstractTransactionalJUnit4SpringContextTests` and `AbstractTransactionalTestNGSpringContextTests` internally use a `ResourceDatabasePopulator` for executing SQL scripts. Consult the javadocs for the various `executeSqlScript(...)` methods for further details.

## Executing SQL scripts declaratively with `@Sql`

In addition to the aforementioned mechanisms for executing SQL scripts *programmatically*, SQL scripts can also be configured *declaratively* in the Spring TestContext Framework. Specifically, the `@Sql` annotation can be declared on a test class or test method to configure the resource paths to SQL scripts that should be executed against a given

database either before or after an integration test method. Note that method-level declarations override class-level declarations and that support for `@Sql` is provided by the `SqlScriptsTestExecutionListener` which is enabled by default.

## Path resource semantics

Each path will be interpreted as a Spring `Resource`. A plain path—for example, `"schema.sql"`—will be treated as a classpath resource that is *relative* to the package in which the test class is defined. A path starting with a slash will be treated as an *absolute* classpath resource, for example: `"/org/example/schema.sql"`. A path which references a URL (e.g., a path prefixed with `classpath:`, `file:`, `http:`, etc.) will be loaded using the specified resource protocol.

The following example demonstrates how to use `@Sql` at the class level and at the method level within a JUnit 4 based integration test class.

```
@RunWith(SpringRunner.class)
@ContextConfiguration
@Sql("/test-schema.sql")
public class DatabaseTests {

    @Test
    public void emptySchemaTest {
        // execute code that uses the test schema without any test data
    }

    @Test
    @Sql({"/test-schema.sql", "/test-user-data.sql"})
    public void userTest {
        // execute code that uses the test schema and test data
    }
}
```

## Default script detection

If no SQL scripts are specified, an attempt will be made to detect a `default` script depending on where `@Sql` is declared. If a default cannot be detected, an `IllegalStateException` will be thrown.

- *class-level declaration*: if the annotated test class is `com.example.MyTest`, the corresponding default script is `"classpath:com/example/MyTest.sql"`.
- *method-level declaration*: if the annotated test method is named `testMethod()` and is defined in the class `com.example.MyTest`, the corresponding default script is `"classpath:com/example/MyTest.testMethod.sql"`.

## Declaring multiple `@Sql` sets

If multiple sets of SQL scripts need to be configured for a given test class or test method but with different syntax configuration, different error handling rules, or different execution phases per set, it is possible to declare multiple instances of `@Sql`. With Java 8, `@Sql` can be used as a *repeatable* annotation. Otherwise, the `@SqlGroup` annotation can be used as an explicit container for declaring multiple instances of `@Sql`.

The following example demonstrates the use of `@Sql` as a repeatable annotation using Java 8. In this scenario the `test-schema.sql` script uses a different syntax for single-line comments.

```
@Test
@Sql/scripts = "/test-schema.sql", config = @SqlConfig(commentPrefix = "'")
@Sql("/test-user-data.sql")
public void userTest {
    // execute code that uses the test schema and test data
}
```

The following example is identical to the above except that the `@Sql` declarations are grouped together within `@SqlGroup` for compatibility with Java 6 and Java 7.

```
@Test
@SqlGroup({
    @Sql/scripts = "/test-schema.sql", config = @SqlConfig(commentPrefix = "'"),
    @Sql("/test-user-data.sql")
})
public void userTest {
    // execute code that uses the test schema and test data
}
```

## Script execution phases

By default, SQL scripts will be executed *before* the corresponding test method. However, if a particular set of scripts needs to be executed *after* the test method — for example, to clean up database state — the `executionPhase` attribute in `@Sql` can be used as seen in the following example. Note that `ISOLATED` and `AFTER_TEST_METHOD` are statically imported from `Sql.TransactionMode` and `Sql.ExecutionPhase` respectively.

```

@Test
@Sql(
    scripts = "create-test-data.sql",
    config = @SqlConfig(transactionMode = ISOLATED)
)
@Sql(
    scripts = "delete-test-data.sql",
    config = @SqlConfig(transactionMode = ISOLATED),
    executionPhase = AFTER_TEST_METHOD
)
public void userTest {
    // execute code that needs the test data to be committed
    // to the database outside of the test's transaction
}

```

## Script configuration with `@SqlConfig`

Configuration for script parsing and error handling can be configured via the `@SqlConfig` annotation. When declared as a class-level annotation on an integration test class, `@SqlConfig` serves as *global* configuration for all SQL scripts within the test class hierarchy. When declared directly via the `config` attribute of the `@Sql` annotation, `@SqlConfig` serves as *local* configuration for the SQL scripts declared within the enclosing `@Sql` annotation. Every attribute in `@SqlConfig` has an implicit default value which is documented in the javadocs of the corresponding attribute. Due to the rules defined for annotation attributes in the Java Language Specification, it is unfortunately not possible to assign a value of `null` to an annotation attribute. Thus, in order to support overrides of inherited global configuration, `@SqlConfig` attributes have an explicit default value of either `""` for Strings or `DEFAULT` for Enums. This approach allows local declarations of `@SqlConfig` to selectively override individual attributes from global declarations of `@SqlConfig` by providing a value other than `""` or `DEFAULT`. Global `@SqlConfig` attributes are inherited whenever local `@SqlConfig` attributes do not supply an explicit value other than `""` or `DEFAULT`. Explicit *local* configuration therefore overrides *global* configuration.

The configuration options provided by `@Sql` and `@SqlConfig` are equivalent to those supported by `ScriptUtils` and `ResourceDatabasePopulator` but are a superset of those provided by the `<jdbc:initialize-database/>` XML namespace element. Consult the javadocs of individual attributes in `@Sql` and `@SqlConfig` for details.

## Transaction management for `@Sql`

By default, the `SqlScriptsTestExecutionListener` will infer the desired transaction semantics for scripts configured via `@Sql`. Specifically, SQL scripts will be executed without a transaction, within an existing Spring-managed transaction—for example, a transaction managed by the `TransactionalTestExecutionListener` for a test annotated with `@Transactional`—or within an isolated transaction, depending on the configured value of

the `transactionMode` attribute in `@SqlConfig` and the presence of a `PlatformTransactionManager` in the test's `ApplicationContext`. As a bare minimum however, a `javax.sql.DataSource` must be present in the test's `ApplicationContext`.

If the algorithms used by `SqlScriptsTestExecutionListener` to detect a `DataSource` and `PlatformTransactionManager` and infer the transaction semantics do not suit your needs, you may specify explicit names via the `dataSource` and `transactionManager` attributes of `@SqlConfig`. Furthermore, the transaction propagation behavior can be controlled via the `transactionMode` attribute of `@SqlConfig` — for example, if scripts should be executed in an isolated transaction. Although a thorough discussion of all supported options for transaction management with `@Sql` is beyond the scope of this reference manual, the javadocs for `@SqlConfig` and `SqlScriptsTestExecutionListener` provide detailed information, and the following example demonstrates a typical testing scenario using JUnit 4 and transactional tests with `@Sql`. Note that there is no need to clean up the database after the `usersTest()` method is executed since any changes made to the database (either within the test method or within the `/test-data.sql` script) will be automatically rolled back by the `TransactionalTestExecutionListener` (see [transaction management](#) for details).

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = TestDatabaseConfig.class)
@Transactional
public class TransactionalSqlScriptsTests {

    protected JdbcTemplate jdbcTemplate;

    @Autowired
    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    @Test
    @Sql("/test-data.sql")
    public void usersTest() {
        // verify state in test database:
        assertEquals(2, countRowsInTable("user"));
        // execute code that uses the test data...
    }

    protected int countRowsInTable(String tableName) {
        return Jdbc TestUtils.countRowsInTable(this.jdbcTemplate, tableName);
    }

    protected void assertEquals(int expected) {
        assertEquals("Number of rows in the [user] table.", expected, countRowsInTable("user"));
    }
}
```

## 11.5.9 Parallel test execution

Spring Framework 5.0 introduces basic support for executing tests in parallel within a single JVM when using the *Spring TestContext Framework*. In general this means that most test classes or test methods can be executed in parallel without any changes to test code or configuration.



For details on how to set up parallel test execution, consult the documentation for your testing framework, build tool, or IDE.

Keep in mind that the introduction of concurrency into your test suite can result in unexpected side effects, strange runtime behavior, and tests that only fail intermittently or seemingly randomly. The Spring Team therefore provides the following general guidelines for when *not* to execute tests in parallel.

*Do not execute tests in parallel if:*

- Tests make use of Spring's `@DirtiesContext` support.
- Tests make use of JUnit 4's `@FixMethodOrder` support or any testing framework feature that is designed to ensure that test methods execute in a particular order. Note, however, that this does not apply if entire test classes are executed in parallel.
- Tests change the state of shared services or systems such as a database, message broker, filesystem, etc. This applies to both in-memory and external systems.



If parallel test execution fails with an exception stating that the `ApplicationContext` for the current test is no longer active, this typically means that the `ApplicationContext` was removed from the `ContextCache` in a different thread. This may be due to the use of `@DirtiesContext` or due to automatic eviction from the `ContextCache`. If `@DirtiesContext` is the culprit, you will either need to find a way to avoid using `@DirtiesContext` or exclude such tests from parallel execution. If the maximum size of the `ContextCache` has been exceeded, you can increase the maximum size of the cache. See the discussion on [context caching](#) for details.



Parallel test execution in the Spring TestContext Framework is only possible if the underlying `TestContext` implementation provides a *copy constructor* as explained in the javadocs for `TestContext`. The `DefaultTestContext` used in Spring provides such a constructor; however, if you use a third-party library that provides a custom `TestContext` implementation, you will need to verify if it is suitable for parallel test execution.

## 11.5.10 TestContext Framework support classes

## Spring JUnit 4 Runner

The *Spring TestContext Framework* offers full integration with JUnit 4 through a custom runner (supported on JUnit 4.12 or higher). By annotating test classes with `@RunWith(SpringJUnit4ClassRunner.class)` or the shorter `@RunWith(SpringRunner.class)` variant, developers can implement standard JUnit 4 based unit and integration tests and simultaneously reap the benefits of the TestContext framework such as support for loading application contexts, dependency injection of test instances, transactional test method execution, and so on. If you would like to use the Spring TestContext Framework with an alternative runner such as JUnit 4's `Parameterized` or third-party runners such as the `MockitoJUnitRunner`, you may optionally use [Spring's support for JUnit rules](#) instead.

The following code listing displays the minimal requirements for configuring a test class to run with the custom Spring `Runner`. `@TestExecutionListeners` is configured with an empty list in order to disable the default listeners, which otherwise would require an `ApplicationContext` to be configured through `@ContextConfiguration`.

```
@RunWith(SpringRunner.class)
@TestExecutionListeners({})
public class SimpleTest {

    @Test
    public void testMethod() {
        // execute test logic...
    }
}
```

## Spring JUnit 4 Rules

The `org.springframework.test.context.junit4.rules` package provides the following JUnit 4 rules (supported on JUnit 4.12 or higher).

- `SpringClassRule`
- `SpringMethodRule`

`SpringClassRule` is a JUnit `TestRule` that supports *class-level* features of the *Spring TestContext Framework*; whereas, `SpringMethodRule` is a JUnit `MethodRule` that supports instance-level and method-level features of the *Spring TestContext Framework*.

In contrast to the `SpringRunner`, Spring's rule-based JUnit support has the advantage that it is independent of any `org.junit.runner.Runner` implementation and can therefore be combined with existing alternative runners like JUnit 4's `Parameterized` or third-party runners such as the `MockitoJUnitRunner`.

In order to support the full functionality of the TestContext framework, a `SpringClassRule` must be combined with a `SpringMethodRule`. The following example demonstrates the proper way to declare these rules in an integration test.

```
// Optionally specify a non-Spring Runner via @RunWith(...)
@ContextConfiguration
public class IntegrationTest {

    @ClassRule
    public static final SpringClassRule SPRING_CLASS_RULE = new SpringClassRule();

    @Rule
    public final SpringMethodRule springMethodRule = new SpringMethodRule();

    @Test
    public void testMethod() {
        // execute test logic...
    }
}
```

## JUnit 4 support classes

The `org.springframework.test.context.junit4` package provides the following support classes for JUnit 4 based test cases (supported on JUnit 4.12 or higher).

- `AbstractJUnit4SpringContextTests`
- `AbstractTransactionalJUnit4SpringContextTests`

`AbstractJUnit4SpringContextTests` is an abstract base test class that integrates the *Spring TestContext Framework* with explicit `ApplicationContext` testing support in a JUnit 4 environment. When you extend `AbstractJUnit4SpringContextTests`, you can access a `protected ApplicationContext` instance variable that can be used to perform explicit bean lookups or to test the state of the context as a whole.

`AbstractTransactionalJUnit4SpringContextTests` is an abstract *transactional* extension of `AbstractJUnit4SpringContextTests` that adds some convenience functionality for JDBC access. This class expects a `javax.sql.DataSource` bean and a `PlatformTransactionManager` bean to be defined in the `ApplicationContext`. When you extend `AbstractTransactionalJUnit4SpringContextTests` you can access a `protected JdbcTemplate` instance variable that can be used to execute SQL statements to query the database. Such queries can be used to confirm database state both *prior to* and *after* execution of database-related application code, and Spring ensures that such queries run in the scope of the same transaction as the application code. When used in conjunction with an ORM tool, be sure to avoid [false positives](#). As mentioned in [Section 11.3, “JDBC Testing Support”](#), `AbstractTransactionalJUnit4SpringContextTests` also provides convenience

methods which delegate to methods in `JdbcTestUtils` using the aforementioned `jdbcTemplate`. Furthermore, `AbstractTransactionalJUnit4SpringContextTests` provides an `executeSqlScript(..)` method for executing SQL scripts against the configured `DataSource`.



These classes are a convenience for extension. If you do not want your test classes to be tied to a Spring-specific class hierarchy, you can configure your own custom test classes by using `@RunWith(SpringRunner.class)` or [Spring's JUnit rules](#).

## TestNG support classes

The `org.springframework.test.context.testng` package provides the following support classes for TestNG based test cases.

- `AbstractTestNGSpringContextTests`
- `AbstractTransactionalTestNGSpringContextTests`

`AbstractTestNGSpringContextTests` is an abstract base test class that integrates the *Spring TestContext Framework* with explicit `ApplicationContext` testing support in a TestNG environment. When you extend `AbstractTestNGSpringContextTests`, you can access a `protected ApplicationContext` instance variable that can be used to perform explicit bean lookups or to test the state of the context as a whole.

`AbstractTransactionalTestNGSpringContextTests` is an abstract *transactional* extension of `AbstractTestNGSpringContextTests` that adds some convenience functionality for JDBC access. This class expects a `javax.sql.DataSource` bean and a `PlatformTransactionManager` bean to be defined in the `ApplicationContext`. When you extend `AbstractTransactionalTestNGSpringContextTests` you can access a `protected JdbcTemplate` instance variable that can be used to execute SQL statements to query the database. Such queries can be used to confirm database state both *prior to* and *after* execution of database-related application code, and Spring ensures that such queries run in the scope of the same transaction as the application code. When used in conjunction with an ORM tool, be sure to avoid [false positives](#). As mentioned in [Section 11.3, “JDBC Testing Support”](#), `AbstractTransactionalTestNGSpringContextTests` also provides convenience methods which delegate to methods in `JdbcTestUtils` using the aforementioned `jdbcTemplate`. Furthermore, `AbstractTransactionalTestNGSpringContextTests` provides an `executeSqlScript(..)` method for executing SQL scripts against the configured `DataSource`.



These classes are a convenience for extension. If you do not want your test classes to be tied to a Spring-specific class hierarchy, you can configure your own custom test classes by using `@ContextConfiguration`, `@TestExecutionListeners`, and so on, and by manually instrumenting your test class with a `TestContextManager`. See the source code of `AbstractTestNGSpringContextTests` for an example of how to instrument your test class.

The `_Spring MVC Test framework` provides first class support for testing Spring MVC code using a fluent API that can be used with JUnit, TestNG, or any other testing framework. It's built on the [Servlet API mock objects](#) from the `spring-test` module and hence does not use a running Servlet container. It uses the `DispatcherServlet` to provide full Spring MVC runtime behavior and provides support for loading actual Spring configuration with the `_TestContext framework` in addition to a standalone mode in which controllers may be instantiated manually and tested one at a time.

`_Spring MVC Test` also provides client-side support for testing code that uses the `RestTemplate`. Client-side tests mock the server responses and also do not use a running server.



Spring Boot provides an option to write full, end-to-end integration tests that include a running server. If this is your goal please have a look at the [Spring Boot reference page](#). For more information on the differences between out-of-container and end-to-end integration tests, see [the section called “Differences between Out-of-Container and End-to-End Integration Tests”](#).

## 11.6.1 Server-Side Tests

It's easy to write a plain unit test for a Spring MVC controller using JUnit or TestNG: simply instantiate the controller, inject it with mocked or stubbed dependencies, and call its methods passing `MockHttpServletRequest`, `MockHttpServletResponse`, etc., as necessary. However, when writing such a unit test, much remains untested: for example, request mappings, data binding, type conversion, validation, and much more. Furthermore, other controller methods such as `@InitBinder`, `@ModelAttribute`, and `@ExceptionHandler` may also be invoked as part of the request processing lifecycle.

The goal of `_Spring MVC Test_` is to provide an effective way for testing controllers by performing requests and generating responses through the actual `DispatcherServlet`.

`_Spring MVC Test_` builds on the familiar "mock" implementations of the Servlet API available in the `spring-test` module. This allows performing requests and generating responses without the need for running in a Servlet container. For the most part everything should work as it does at runtime with a few notable exceptions as explained in the section called "[Differences between Out-of-Container and End-to-End Integration Tests](#)". Here is a JUnit 4 based example of using Spring MVC Test:

```
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@RunWith(SpringRunner.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(MediaType.parseMediaType(
"application/json; charset=UTF-8"
)))
.andExpect(status().isOk())
.andExpect(content().contentType(
"application/json"
))
.andExpect(jsonPath(
"$.name"
).value(
"Lee"
));
}

}

```

The above test relies on the `WebApplicationContext` support of the `_TestContext` framework for loading Spring configuration from an XML configuration file located in the same package as the test class, but Java-based and Groovy-based configuration are also supported. See these[sample tests](#).

The `MockMvc` instance is used to perform a `GET` request to `"/accounts/1"` and verify that the resulting response has status 200, the content type is `"application/json"`, and the response body has a JSON property called `"name"` with the value `"Lee"`. The `jsonPath` syntax is supported through the Jayway[JsonPath project](#). There are lots of other options for verifying the result of the performed request that will be discussed below.

## Static Imports

The fluent API in the example above requires a few static imports such as `MockMvcRequestBuilders.*`, `MockMvcResultMatchers.*`, and `MockMvcBuilders.*`. An easy way to find these classes is to search for types matching "`MockMvc*`". If using Eclipse, be sure to add them as "favorite static members" in the Eclipse preferences under `Java → Editor → Content Assist → Favorites`. That will allow use of content assist after typing the first character of the static method name. Other IDEs (e.g. IntelliJ) may not require any additional configuration. Just check the support for code completion on static members.

## Setup Choices

There are two main options for creating an instance of `MockMvc`. The first is to load Spring MVC configuration through the `TestContext framework`, which loads the Spring configuration and injects a `WebApplicationContext` into the test to use to build a `MockMvc` instance:

```
@RunWith(SpringRunner.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();
    }

    // ...
}
```

The second is to simply create a controller instance manually without loading Spring configuration. Instead basic default configuration, roughly comparable to that of the MVC JavaConfig or the MVC namespace, is automatically created and can be customized to a degree:

```
public  
class  
MyWebTests {  
  
    private  
    MockMvc mockMvc;  
  
    @Before  
    public  
    void  
    setup() {  
  
        this  
        .mockMvc = MockMvcBuilders.standaloneSetup(  
new  
        AccountController()).build();  
    }  
  
    // ...  
  
}
```

Which setup option should you use?

The \_"webAppContextSetup"\_ loads your actual Spring MVC configuration resulting in a more complete integration test. Since the \_TestContext framework\_ caches the loaded Spring configuration, it helps keep tests running fast, even as you introduce more tests in your test suite. Furthermore, you can inject mock services into controllers through Spring configuration in order to remain focused on testing the web layer. Here is an example of declaring a mock service with Mockito:

```
<
bean
id
=
"accountService"
class
=
"org.mockito.Mockito"
factory-method
=
"mock"
>
<
constructor-arg
value
=
"org.example.AccountService"
/
>
<
/bean
>
```

You can then inject the mock service into the test in order set up and verify expectations:

```

@RunWith(SpringRunner.class)
@WebAppConfiguration
@ContextConfiguration("test-servlet-context.xml")
public
class
AccountTests {

    @Autowired
    private
    WebApplicationContext wac;

    private
    MockMvc mockMvc;

    @Autowired
    private
    AccountService accountService;

    // ...

}

```

The `_standaloneSetup`\_ on the other hand is a little closer to a unit test. It tests one controller at a time: the controller can be injected with mock dependencies manually, and it doesn't involve loading Spring configuration. Such tests are more focused on style and make it easier to see which controller is being tested, whether any specific Spring MVC configuration is required to work, and so on. The `"standaloneSetup"` is also a very convenient way to write ad-hoc tests to verify specific behavior or to debug an issue.

Just like with any "integration vs. unit testing" debate, there is no right or wrong answer. However, using the `"standaloneSetup"` does imply the need for additional `"webApplicationContextSetup"` tests in order to verify your Spring MVC configuration. Alternatively, you may choose to write all tests with `"webApplicationContextSetup"` in order to always test against your actual Spring MVC configuration.

## Setup Features

No matter which `MockMvc` builder you use all `MockMvcBuilder` implementations provide some common and very useful features. For example you can declare an `Accept` header for all requests and expect a status of 200 as well as a `Content-Type` header in all responses as follows:

```
// static import of MockMvcBuilders.standaloneSetup

MockMvc mockMvc = standaloneSetup(
    new
        MusicController()
            .defaultRequest(get(
                "/"
            ).accept(MediaType.APPLICATION_JSON))
            .alwaysExpect(status().isOk())
            .alwaysExpect(content().contentType(
                "application/json; charset=UTF-8"
            ))
        .build());

```

In addition 3rd party frameworks (and applications) may pre-package setup instructions like the ones through a `MockMvcConfigurer`. The Spring Framework has one such built-in implementation that helps to save and re-use the HTTP session across requests. It can be used as follows:

```
// static import of SharedHttpSessionConfigurer.sharedHttpSession

MockMvc mockMvc = MockMvcBuilders.standaloneSetup(
    new
        TestController()
            .apply(sharedHttpSession())
        .build());

// Use mockMvc to perform requests...

```

See `ConfigurableMockMvcBuilder` for a list of all MockMvc builder features or use the IDE to explore the available options.

## Performing Requests

It's easy to perform requests using any HTTP method:

```
mockMvc.perform(post(
    "/hotels/{id}"
    ,
    42
).accept(MediaType.APPLICATION_JSON));
```

You can also perform file upload requests that internally use `MockMultipartHttpServletRequest` so that there is no actual parsing of a multipart request but rather you have to set it up:

```
mockMvc.perform(multipart(  
    "/doc"  
).file(  
    "a1"  
,  
    "ABC"  
.getBytes(  
    "UTF-8"  
)));
```

You can specify query parameters in URI template style:

```
mockMvc.perform(get(  
    "/hotels?foo={foo}"  
,  
    "bar"  
));
```

Or you can add Servlet request parameters representing either query or form parameters:

```
mockMvc.perform(get(  
    "/hotels"  
).param(  
    "foo"  
,  
    "bar"  
));
```

If application code relies on Servlet request parameters and doesn't check the query string explicitly (as is most often the case) then it doesn't matter which option you use. Keep in mind however that query params provided with the URI template will be decoded while request parameters provided through the `param(...)` method are expected to already be decoded.

In most cases it's preferable to leave out the context path and the Servlet path from the request URI. If you must test with the full request URI, be sure to set the `contextPath` and `servletPath` accordingly so that request mappings will work:

```
mockMvc.perform(get()
    "/app/main/hotels/{id}"
).contextPath(
    "/app"
).servletPath(
    "/main"
))
```

Looking at the above example, it would be cumbersome to set the contextPath and servletPath with every performed request. Instead you can set up default request properties:

```
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();
    }
}
```

The above properties will affect every request performed through the `MockMvc` instance. If the same property is also specified on a given request, it overrides the default value. That is why the HTTP method and URI in the default request don't matter since they must be specified on every request.

## Defining Expectations

Expectations can be defined by appending one or more `.andExpect(..)` calls after performing a request:

```
mockMvc.perform(get()
    "/accounts/1"
).andExpect(status().isOk());
```

`MockMvcResultMatchers.*` provides a number of expectations, some of which are further nested with more detailed expectations.

Expectations fall in two general categories. The first category of assertions verifies properties of the response: for example, the response status, headers, and content. These are the most important results to assert.

The second category of assertions goes beyond the response. These assertions allow one to inspect Spring MVC specific aspects such as which controller method processed the request, whether an exception was raised and handled, what the content of the model is, what view was selected, what flash attributes were added, and so on. They also allow one to inspect Servlet specific aspects such as request and session attributes.

The following test asserts that binding or validation failed:

```
mockMvc.perform(post(
    "/persons"
))
    .andExpect(status().isOk())
    .andExpect(model().attributeHasErrors(
        "person"
));
```

Many times when writing tests, it's useful to \_dump\_ the results of the performed request. This can be done as follows, where `print()` is a static import from `MockMvcResultHandlers`:

```
mockMvc.perform(post(
    "/persons"
))
    .andDo(print())
    .andExpect(status().isOk())
    .andExpect(model().attributeHasErrors(
        "person"
));
```

As long as request processing does not cause an unhandled exception, the `print()` method will print all the available result data to `System.out`. Spring Framework 4.2 introduced a `log()` method and two additional variants of the `print()` method, one that accepts an `OutputStream` and one that accepts a `Writer`. For example, invoking `print(System.err)` will print the result data to `System.err`; while

invoking `print(myWriter)` will print the result data to a custom writer. If you would like to have the `resultData_logged` instead of printed, simply invoke the `log()` method which will log the result data as a single `DEBUG` message under the `org.springframework.test.web.servlet.result` logging category.

In some cases, you may want to get direct access to the result and verify something that cannot be verified otherwise. This can be achieved by appending `.andReturn()` after all other expectations:

```
MvcResult mvcResult = mockMvc.perform(post(
    "/persons"
)).andExpect(status().isOk()).andReturn();

// ...
```

If all tests repeat the same expectations you can set up common expectations once when building the `MockMvc` instance:

```
standaloneSetup(
    new
    SimpleController()
        .alwaysExpect(status().isOk())
        .alwaysExpect(content().contentType(
            "application/json; charset=UTF-8"
        ))
        .build()
```

Note that common expectations are `_always_` applied and cannot be overridden without creating a separate `MockMvc` instance.

When JSON response content contains hypermedia links created with [Spring HATEOAS](#), the resulting links can be verified using JsonPath expressions:

```
mockMvc.perform(get(
    "/people"
).accept(MediaType.APPLICATION_JSON))
    .andExpect(jsonPath(
        "$.links[?(@.rel == 'self')].href"
    ).value(
        "http://localhost:8080/people"
));
```

When XML response content contains hypermedia links created with [Spring HATEOAS](#), the resulting links can be verified using XPath expressions:

```

Map<
String, String
>
ns = Collections.singletonMap(
"ns"
,
"http://www.w3.org/2005/Atom"
);
mockMvc.perform(get(
"/handle"
).accept(MediaType.APPLICATION_XML))
.andExpect(xpath(
"/person
s:link[@rel='self']/@href"
, ns).string(
"http://localhost:8080/people"
));

```

## Filter Registrations

When setting up a `MockMvc` instance, you can register one or more Servlet `Filter` instances:

```

mockMvc = standaloneSetup(
new
PersonController()).addFilters(
new
CharacterEncodingFilter()).build();

```

Registered filters will be invoked through via the `MockFilterChain` from `spring-test`, and the last filter will delegate to the `DispatcherServlet`.

## Differences between Out-of-Container and End-to-End Integration Tests

As mentioned earlier, Spring MVC Test is built on the Servlet API mock objects from the `spring-test` module and does not use a running Servlet container. Therefore there are some important differences compared to full end-to-end integration tests with an actual client and server running.

The easiest way to think about this is starting with a blank `MockHttpServletRequest`. Whatever you add to it is what the request will be. Things that may catch you by surprise are that there is no context path by default, no `jsessionid` cookie, no forwarding, error, or async dispatches, and therefore no actual JSP rendering. Instead, "forwarded" and "redirected" URLs are saved in the `MockHttpServletResponse` and can be asserted with expectations.

This means if you are using JSPs you can verify the JSP page to which the request was forwarded, but there won't be any HTML rendered. In other words, the JSP will not be invoked. Note however that all other rendering technologies which don't rely on forwarding such as Thymeleaf and Freemarker will render HTML to the response body as expected. The same is true for rendering JSON, XML, and other formats via `@ResponseBody` methods.

Alternatively you may consider the full end-to-end integration testing support from Spring Boot via `@WebIntegrationTest`. See the [Spring Boot reference](#).

There are pros and cons for each approach. The options provided in `_Spring MVC Test_` are different stops on the scale from classic unit testing to full integration testing. To be certain, none of the options in Spring MVC Test fall under the category of classic unit testing, but they are a little closer to it. For example, you can isolate the web layer by injecting mocked services into controllers, in which case you're testing the web layer only through the `DispatcherServlet` but with actual Spring configuration, just like you might test the data access layer in isolation from the layers above. Or you can use the standalone setup focusing on one controller at a time and manually providing the configuration required to make it work.

Another important distinction when using `_Spring MVC Test_` is that conceptually such tests are on the `_inside_` of the server-side so you can check what handler was used, if an exception was handled with a `HandlerExceptionResolver`, what the content of the model is, what binding errors there were, etc. That means it's easier to write expectations since the server is not a black box as it is when testing it through an actual HTTP client. This is generally an advantage of classic unit testing, that it's easier to write, reason about, and debug but does not replace the need for full integration tests. At the same time it's important not to lose sight of the fact that the response is the most important thing to check. In short, there is room here for multiple styles and strategies of testing even within the same project.

## Further Server-Side Test Examples

The framework's own tests include [many sample tests](#) intended to demonstrate how to use Spring MVC Test. Browse these examples for further ideas. Also the [spring-mvc-showcase](#) has full test coverage based on Spring MVC Test.

## 11.6.2 HtmlUnit Integration

Spring provides integration between [MockMvc](#) and [HtmlUnit](#). This simplifies performing end-to-end testing when using HTML based views. This integration enables developers to:

- Easily test HTML pages using tools such as [HtmlUnit](#), [WebDriver](#), & [Geb](#) without the need to deploy to a Servlet container
- Test JavaScript within pages
- Optionally test using mock services to speed up testing
- Share logic between in-container end-to-end tests and out-of-container integration tests



`MockMvc` works with templating technologies that do not rely on a Servlet Container (e.g., Thymeleaf, FreeMarker, etc.), but it does not work with JSPs since they rely on the Servlet container.

## Why HtmlUnit Integration?

The most obvious question that comes to mind is, "Why do I need this?". The answer is best found by exploring a very basic sample application. Assume you have a Spring MVC web application that supports CRUD operations on a `Message` object. The application also supports paging through all messages. How would you go about testing it?

With Spring MVC Test, we can easily test if we are able to create a `Message`.

```
MockHttpServletRequestBuilder createMessage = post("/messages/")
    .param("summary", "Spring Rocks")
    .param("text", "In case you didn't know, Spring Rocks!");

mockMvc.perform(createMessage)
    .andExpect(status().is3xxRedirection())
    .andExpect(redirectedUrl("/messages/123"));
```

What if we want to test our form view that allows us to create the message? For example, assume our form looks like the following snippet:

```

<form id="messageForm" action="/messages/" method="post">
    <div class="pull-right"><a href="/messages/">Messages</a></div>

    <label for="summary">Summary</label>
    <input type="text" class="required" id="summary" name="summary" value="" />

    <label for="text">Message</label>
    <textarea id="text" name="text"></textarea>

    <div class="form-actions">
        <input type="submit" value="Create" />
    </div>
</form>

```

How do we ensure that our form will produce the correct request to create a new message?  
A naive attempt would look like this:

```

mockMvc.perform(get("/messages/form"))
    .andExpect(xpath("//input[@name='summary']").exists())
    .andExpect(xpath("//textarea[@name='text']").exists());

```

This test has some obvious drawbacks. If we update our controller to use the parameter `message` instead of `text`, our form test would continue to pass even though the HTML form is out of sync with the controller. To resolve this we can combine our two tests.

```

String summaryParamName = "summary";
String textParamName = "text";
mockMvc.perform(get("/messages/form"))
    .andExpect(xpath("//input[@name='" + summaryParamName + "']").exists())
    .andExpect(xpath("//textarea[@name='" + textParamName + "']").exists());

MockHttpServletRequestBuilder createMessage = post("/messages/")
    .param(summaryParamName, "Spring Rocks")
    .param(textParamName, "In case you didn't know, Spring Rocks!");

mockMvc.perform(createMessage)
    .andExpect(status().is3xxRedirection())
    .andExpect(redirectedUrl("/messages/123"));

```

This would reduce the risk of our test incorrectly passing, but there are still some problems.

- What if we have multiple forms on our page? Admittedly we could update our xpath expressions, but they get more complicated the more factors we take into account (Are the fields the correct type? Are the fields enabled? etc.).

- Another issue is that we are doing double the work we would expect. We must first verify the view, and then we submit the view with the same parameters we just verified. Ideally this could be done all at once.
- Finally, there are some things that we still cannot account for. For example, what if the form has JavaScript validation that we wish to test as well?

The overall problem is that testing a web page does not involve a single interaction. Instead, it is a combination of how the user interacts with a web page and how that web page interacts with other resources. For example, the result of a form view is used as the input to a user for creating a message. In addition, our form view may potentially utilize additional resources which impact the behavior of the page, such as JavaScript validation.

### **Integration testing to the rescue?**

To resolve the issues above we could perform end-to-end integration testing, but this has some obvious drawbacks. Consider testing the view that allows us to page through the messages. We might need the following tests.

- Does our page display a notification to the user indicating that no results are available when the messages are empty?
- Does our page properly display a single message?
- Does our page properly support paging?

To set up these tests, we would need to ensure our database contained the proper messages in it. This leads to a number of additional challenges.

- Ensuring the proper messages are in the database can be tedious; consider foreign key constraints.
- Testing can become slow since each test would need to ensure that the database is in the correct state.
- Since our database needs to be in a specific state, we cannot run tests in parallel.
- Performing assertions on things like auto-generated ids, timestamps, etc. can be difficult.

These challenges do not mean that we should abandon end-to-end integration testing altogether. Instead, we can reduce the number of end-to-end integration tests by refactoring our detailed tests to use mock services which will execute much faster, more reliably, and without side effects. We can then implement a small number of true end-to-end integration tests that validate simple workflows to ensure that everything works together properly.

## Enter HtmlUnit Integration

So how can we achieve a balance between testing the interactions of our pages and still retain good performance within our test suite? The answer is: "By integrating MockMvc with HtmlUnit."

### HtmlUnit Integration Options

There are a number of ways to integrate `MockMvc` with HtmlUnit.

- [MockMvc and HtmlUnit](#): Use this option if you want to use the raw HtmlUnit libraries.
- [MockMvc and WebDriver](#): Use this option to ease development and reuse code between integration and end-to-end testing.
- [MockMvc and Geb](#): Use this option if you would like to use Groovy for testing, ease development, and reuse code between integration and end-to-end testing.

## MockMvc and HtmlUnit

This section describes how to integrate `MockMvc` and HtmlUnit. Use this option if you want to use the raw HtmlUnit libraries.

### MockMvc and HtmlUnit Setup

First, make sure that you have included a test dependency on `net.sourceforge.htmlunit:htmlunit`. In order to use HtmlUnit with Apache HttpComponents 4.5+, you will need to use HtmlUnit 2.18 or higher.

We can easily create an HtmlUnit `webClient` that integrates with `MockMvc` using the `MockMvcWebClientBuilder` as follows.

```
@Autowired  
WebApplicationContext context;  
  
WebClient webClient;  
  
@Before  
public void setup() {  
    webClient = MockMvcWebClientBuilder  
        .webAppContextSetup(context)  
        .build();  
}
```



This is a simple example of using `MockMvcWebClientBuilder`. For advanced usage see the section called “[Advanced MockMvcWebClientBuilder](#)”

This will ensure that any URL referencing `localhost` as the server will be directed to our `MockMvc` instance without the need for a real HTTP connection. Any other URL will be requested using a network connection as normal. This allows us to easily test the use of CDNs.

## MockMvc and HtmlUnit Usage

Now we can use `HtmlUnit` as we normally would, but without the need to deploy our application to a Servlet container. For example, we can request the view to create a message with the following.

```
HtmlPage createMsgFormPage = webClient.getPage("http://localhost/messages/form");
```



The default context path is `""`. Alternatively, we can specify the context path as illustrated in the section called “[Advanced MockMvcWebClientBuilder](#)”.

Once we have a reference to the `HtmlPage`, we can then fill out the form and submit it to create a message.

```
HtmlForm form = createMsgFormPage.getHtmlElementById("messageForm");
HtmlTextInput summaryInput = createMsgFormPage.getHtmlElementById("summary");
summaryInput.setValueAttribute("Spring Rocks");
HtmlTextArea textInput = createMsgFormPage.getHtmlElementById("text");
textInput.setText("In case you didn't know, Spring Rocks!");
HtmlSubmitInput submit = form.getOneHtmlElementByAttribute("input", "type", "submit");
HtmlPage newMessagePage = submit.click();
```

Finally, we can verify that a new message was created successfully. The following assertions use the [AssertJ](#) library.

```
assertThat(newMessagePage.getUrl().toString()).endsWith("/messages/123");
String id = newMessagePage.getHtmlElementById("id").getTextContent();
assertThat(id).isEqualTo("123");
String summary = newMessagePage.getHtmlElementById("summary").getTextContent();
assertThat(summary).isEqualTo("Spring Rocks");
String text = newMessagePage.getHtmlElementById("text").getTextContent();
assertThat(text).isEqualTo("In case you didn't know, Spring Rocks!");
```

This improves on our [MockMvc test](#) in a number of ways. First we no longer have to explicitly verify our form and then create a request that looks like the form. Instead, we request the form, fill it out, and submit it, thereby significantly reducing the overhead.

Another important factor is that [HtmlUnit uses the Mozilla Rhino engine](#) to evaluate JavaScript. This means that we can test the behavior of JavaScript within our pages as well!

Refer to the [HtmlUnit documentation](#) for additional information about using HtmlUnit.

## Advanced MockMvcWebClientBuilder

In the examples so far, we have used `MockMvcWebClientBuilder` in the simplest way possible, by building a `WebClient` based on the `WebApplicationContext` loaded for us by the Spring TestContext Framework. This approach is repeated here.

```
@Autowired
WebApplicationContext context;

WebClient webClient;

@Before
public void setup() {
    webClient = MockMvcWebClientBuilder
        .webAppContextSetup(context)
        .build();
}
```

We can also specify additional configuration options.

```
WebClient webClient;

@Before
public void setup() {
    webClient = MockMvcWebClientBuilder
        // demonstrates applying a MockMvcConfigurer (Spring Security)
        .webAppContextSetup(context, springSecurity())
        // for illustration only - defaults to ""
        .contextPath("")
        // By default MockMvc is used for localhost only;
        // the following will use MockMvc for example.com and example.org as well
        .useMockMvcForHosts("example.com", "example.org")
        .build();
}
```

As an alternative, we can perform the exact same setup by configuring the `MockMvc` instance separately and supplying it to the `MockMvcWebClientBuilder` as follows.

```

MockMvc mockMvc = MockMvcBuilders
    .webAppContextSetup(context)
    .apply(springSecurity())
    .build();

webClient = MockMvcWebClientBuilder
    .mockMvcSetup(mockMvc)
    // for illustration only - defaults to ""
    .contextPath("//")
    // By default MockMvc is used for localhost only;
    // the following will use MockMvc for example.com and example.org as well
    .useMockMvcForHosts("example.com", "example.org")
    .build();

```

This is more verbose, but by building the `webClient` with a `MockMvc` instance we have the full power of `MockMvc` at our fingertips.

For additional information on creating a `MockMvc` instance refer to the section called “[Setup Choices](#)”.

## MockMvc and WebDriver

In the previous sections, we have seen how to use `MockMvc` in conjunction with the raw `HtmlUnit` APIs. In this section, we will leverage additional abstractions within the Selenium [WebDriver](#) to make things even easier.

### Why WebDriver and MockMvc?

We can already use `HtmlUnit` and `MockMvc`, so why would we want to use `WebDriver`? The Selenium `WebDriver` provides a very elegant API that allows us to easily organize our code. To better understand, let's explore an example.



Despite being a part of [Selenium](#), `WebDriver` does not require a Selenium Server to run your tests.

Suppose we need to ensure that a message is created properly. The tests involve finding the HTML form input elements, filling them out, and making various assertions.

This approach results in numerous, separate tests because we want to test error conditions as well. For example, we want to ensure that we get an error if we fill out only part of the form. If we fill out the entire form, the newly created message should be displayed afterwards.

If one of the fields were named "summary", then we might have something like the following repeated in multiple places within our tests.

```
HtmlTextInput summaryInput = currentPage.getHtmlElementById("summary");
summaryInput.setValueAttribute(summary);
```

So what happens if we change the `id` to "smmry"? Doing so would force us to update all of our tests to incorporate this change! Of course, this violates the *DRY Principle*; so we should ideally extract this code into its own method as follows.

```
public HtmlPage createMessage(HtmlPage currentPage, String summary, String text) {
    setSummary(currentPage, summary);
    // ...
}

public void setSummary(HtmlPage currentPage, String summary) {
    HtmlTextInput summaryInput = currentPage.getHtmlElementById("summary");
    summaryInput.setValueAttribute(summary);
}
```

This ensures that we do not have to update all of our tests if we change the UI.

We might even take this a step further and place this logic within an Object that represents the `HtmlPage` we are currently on.

```

public class CreateMessagePage {

    final HtmlPage currentPage;

    final HtmlTextInput summaryInput;

    final HtmlSubmitInput submit;

    public CreateMessagePage(HtmlPage currentPage) {
        this.currentPage = currentPage;
        this.summaryInput = currentPage.getHtmlElementById("summary");
        this.submit = currentPage.getHtmlElementById("submit");
    }

    public <T> T createMessage(String summary, String text) throws Exception {
        setSummary(summary);

        HtmlPage result = submit.click();
        boolean error = CreateMessagePage.at(result);

        return (T) (error ? new CreateMessagePage(result) : new ViewMessagePage(result));
    }

    public void setSummary(String summary) throws Exception {
        summaryInput.setValueAttribute(summary);
    }

    public static boolean at(HtmlPage page) {
        return "Create Message".equals(page.getTitleText());
    }
}

```

Formerly, this pattern is known as the [Page Object Pattern](#). While we can certainly do this with HtmlUnit, WebDriver provides some tools that we will explore in the following sections to make this pattern much easier to implement.

## MockMvc and WebDriver Setup

To use Selenium WebDriver with the Spring MVC Test framework, make sure that your project includes a test dependency on `org.seleniumhq.selenium:selenium-htmlunit-driver`.

We can easily create a Selenium WebDriver that integrates with `MockMvc` using the `MockMvcHtmlUnitDriverBuilder` as follows.

```

@Autowired
WebApplicationContext context;

WebDriver driver;

@Before
public void setup() {
    driver = MockMvcHtmlUnitDriverBuilder
        .webAppContextSetup(context)
        .build();
}

```



This is a simple example of using `MockMvcHtmlUnitDriverBuilder`. For more advanced usage, refer to [the section called “Advanced MockMvcHtmlUnitDriverBuilder”](#)

This will ensure that any URL referencing `localhost` as the server will be directed to our `MockMvc` instance without the need for a real HTTP connection. Any other URL will be requested using a network connection as normal. This allows us to easily test the use of CDNs.

## MockMvc and WebDriver Usage

Now we can use WebDriver as we normally would, but without the need to deploy our application to a Servlet container. For example, we can request the view to create a message with the following.

```
CreateMessagePage page = CreateMessagePage.to(driver);
```

We can then fill out the form and submit it to create a message.

```
ViewMessagePage viewMessagePage =
    page.createMessage(ViewMessagePage.class, expectedSummary, expectedText);
```

This improves on the design of our `HtmlUnit test` by leveraging the `Page Object Pattern`. As we mentioned in [the section called “Why WebDriver and MockMvc?”](#), we can use the Page Object Pattern with HtmlUnit, but it is much easier with WebDriver. Let’s take a look at our new `CreateMessagePage` implementation.

```

public class CreateMessagePage extends AbstractPage {

    private WebElement summary;
    private WebElement text;

    @FindBy(css = "input[type=submit]")
    private WebElement submit;

    public CreateMessagePage(WebDriver driver) {
        super(driver);
    }

    public <T> T createMessage(Class<T> resultPage, String summary, String details) {
        this.summary.sendKeys(summary);
        this.text.sendKeys(details);
        this.submit.click();
        return PageFactory.initElements(driver, resultPage);
    }

    public static CreateMessagePage to(WebDriver driver) {
        driver.get("http://localhost:9990/mail/messages/form");
        return PageFactory.initElements(driver, CreateMessagePage.class);
    }
}

```

①

The first thing you will notice is that `CreateMessagePage` extends the `AbstractPage`. We won't go over the details of `AbstractPage`, but in summary it contains common functionality for all of our pages. For example, if our application has a navigational bar, global error messages, etc., this logic can be placed in a shared location.

The next thing you will notice is that we have a member variable for each of the parts of the HTML page that we are interested in. These are of type `WebElement`. `WebDriver`'s `PageFactory` allows us to remove a lot of code from the

② `HtmlUnit` version of `CreateMessagePage` by automatically resolving each `webElement`. The `PageFactory#initElements(WebDriver, Class<T>)` method will automatically resolve each `webElement` by using the field name and looking it up by the `id` or `name` of the element within the HTML page.

③ We can use the `@FindBy annotation` to override the default lookup behavior. Our example demonstrates how to use the `@FindBy` annotation to look up our submit button using a css selector, `input[type=submit]`.

Finally, we can verify that a new message was created successfully. The following assertions use the `FEST assertion library`.

```
assertThat(viewMessagePage.getMessage()).isEqualTo(expectedMessage);
assertThat(viewMessagePage.getSuccess()).isEqualTo("Successfully created a new message");
};
```

We can see that our `ViewMessagePage` allows us to interact with our custom domain model. For example, it exposes a method that returns a `Message` object.

```
public Message getMessage() throws ParseException {
    Message message = new Message();
    message.setId(getId());
    message.setCreated(getCreated());
    message.setSummary(getSummary());
    message.setText(getText());
    return message;
}
```

We can then leverage the rich domain objects in our assertions.

Lastly, don't forget to \_close\_ the `WebDriver` instance when the test is complete.

```
@After
public void destroy() {
    if (driver != null) {
        driver.close();
    }
}
```

For additional information on using WebDriver, refer to the Selenium[WebDriver documentation](#).

## Advanced MockMvcHtmlUnitDriverBuilder

In the examples so far, we have used `MockMvcHtmlUnitDriverBuilder` in the simplest way possible, by building a `WebDriver` based on the `WebApplicationContext` loaded for us by the Spring TestContext Framework. This approach is repeated here.

```

@Autowired
WebApplicationContext context;

WebDriver driver;

@Before
public void setup() {
    driver = MockMvcHtmlUnitDriverBuilder
        .webAppContextSetup(context)
        .build();
}

```

We can also specify additional configuration options.

```

WebDriver driver;

@Before
public void setup() {
    driver = MockMvcHtmlUnitDriverBuilder
        // demonstrates applying a MockMvcConfigurer (Spring Security)
        .webAppContextSetup(context, springSecurity())
        // for illustration only - defaults to ""
        .contextPath("")
        // By default MockMvc is used for localhost only;
        // the following will use MockMvc for example.com and example.org as well
        .useMockMvcForHosts("example.com", "example.org")
        .build();
}

```

As an alternative, we can perform the exact same setup by configuring the `MockMvc` instance separately and supplying it to the `MockMvcHtmlUnitDriverBuilder` as follows.

```

MockMvc mockMvc = MockMvcBuilders
    .webAppContextSetup(context)
    .apply(springSecurity())
    .build();

driver = MockMvcHtmlUnitDriverBuilder
    .mockMvcSetup(mockMvc)
    // for illustration only - defaults to ""
    .contextPath("")
    // By default MockMvc is used for localhost only;
    // the following will use MockMvc for example.com and example.org as well
    .useMockMvcForHosts("example.com", "example.org")
    .build();

```

This is more verbose, but by building the `WebDriver` with a `MockMvc` instance we have the full power of `MockMvc` at our fingertips.



For additional information on creating a `MockMvc` instance refer to [the section called “Setup Choices”](#).

## MockMvc and Geb

In the previous section, we saw how to use `MockMvc` with `WebDriver`. In this section, we will use [Geb](#) to make our tests even Groovy-er.

### Why Geb and MockMvc?

Geb is backed by WebDriver, so it offers many of the [same benefits](#) that we get from WebDriver. However, Geb makes things even easier by taking care of some of the boilerplate code for us.

### MockMvc and Geb Setup

We can easily initialize a Geb `Browser` with a Selenium `WebDriver` that uses `MockMvc` as follows.

```
def setup() {
    browser.driver = MockMvcHtmlUnitDriverBuilder
        .webAppContextSetup(context)
        .build()
}
```



This is a simple example of using `MockMvcHtmlUnitDriverBuilder`. For more advanced usage, refer to [the section called “Advanced MockMvcHtmlUnitDriverBuilder”](#)

This will ensure that any URL referencing `localhost` as the server will be directed to our `MockMvc` instance without the need for a real HTTP connection. Any other URL will be requested using a network connection as normal. This allows us to easily test the use of CDNs.

### MockMvc and Geb Usage

Now we can use Geb as we normally would, but without the need to deploy our application to a Servlet container. For example, we can request the view to create a message with the following:

```
to CreateMessagePage
```

We can then fill out the form and submit it to create a message.

```
when:
form.summary = expectedSummary
form.text = expectedMessage
submit.click(ViewMessagePage)
```

Any unrecognized method calls or property accesses/references that are not found will be forwarded to the current page object. This removes a lot of the boilerplate code we needed when using WebDriver directly.

As with direct WebDriver usage, this improves on the design of our [HtmlUnit test](#) by leveraging the *Page Object Pattern*. As mentioned previously, we can use the Page Object Pattern with HtmlUnit and WebDriver, but it is even easier with Geb. Let's take a look at our new Groovy-based `CreateMessagePage` implementation.

```
class CreateMessagePage extends Page {
    static url = 'messages/form'
    static at = { assert title == 'Messages : Create'; true }
    static content = {
        submit { $('input[type=submit]') }
        form { $('form') }
        errors(required:false) { $('label.error, .alert-error')?.text() }
    }
}
```

The first thing you will notice is that our `CreateMessagePage` extends `Page`. We won't go over the details of `Page`, but in summary it contains common functionality for all of our pages. The next thing you will notice is that we define a URL in which this page can be found. This allows us to navigate to the page as follows.

```
to CreateMessagePage
```

We also have an `at` closure that determines if we are at the specified page. It should return `true` if we are on the correct page. This is why we can assert that we are on the correct page as follows.

```
then:
at CreateMessagePage
errors.contains(
    'This field is required.')
)
```



We use an assertion in the closure, so that we can determine where things went wrong if we were at the wrong page.

Next we create a `content` closure that specifies all the areas of interest within the page. We can use [a jQuery-ish Navigator API](#) to select the content we are interested in.

Finally, we can verify that a new message was created successfully.

```
then:  
at ViewMessagePage  
success ==  
'Successfully created a new message'  
  
id  
date  
summary == expectedSummary  
message == expectedMessage
```

For further details on how to get the most out of Geb, consult [The Book of Geb](#) user's manual.

### 11.6.3 Client-Side REST Tests

Client-side tests can be used to test code that internally uses the `RestTemplate`. The idea is to declare expected requests and to provide "stub" responses so that you can focus on testing the code in isolation, i.e. without running a server. Here is an example:

```
RestTemplate restTemplate = new RestTemplate();

MockRestServiceServer mockServer = MockRestServiceServer.bindTo(restTemplate).build();
mockServer.expect(requestTo("/greeting")).andRespond(withSuccess());

// Test code that uses the above RestTemplate ...

mockServer.verify();
```

In the above example, `MockRestServiceServer`, the central class for client-side REST tests, configures the `RestTemplate` with a custom `clientHttpRequestFactory` that asserts actual requests against expectations and returns "stub" responses. In this case we expect a request to "/greeting" and want to return a 200 response with "text/plain" content. We could define as additional expected requests and stub responses as needed. When expected requests and stub responses are defined, the `RestTemplate` can be used in client-side code as usual. At the end of testing `mockServer.verify()` can be used to verify that all expectations have been satisfied.

By default requests are expected in the order in which expectations were declared. You can set the `ignoreExpectOrder` option when building the server in which case all expectations are checked (in order) to find a match for a given request. That means requests are allowed to come in any order. Here is an example:

```
server = MockRestServiceServer.bindTo(restTemplate).ignoreExpectOrder(true).build();
```

Even with unordered requests by default each request is allowed to execute once only. The `expect` method provides an overloaded variant that accepts an `ExpectedCount` argument that specifies a count range, e.g. `once`, `manyTimes`, `max`, `min`, `between`, and so on. Here is an example:

```
RestTemplate restTemplate = new RestTemplate();

MockRestServiceServer mockServer = MockRestServiceServer.bindTo(restTemplate).build();
mockServer.expect(times(2), requestTo("/foo")).andRespond(withSuccess());
mockServer.expect(times(3), requestTo("/bar")).andRespond(withSuccess());

// ...

mockServer.verify();
```

Note that when `ignoreExpectOrder` is not set (the default), and therefore requests are expected in order of declaration, then that order only applies to the first of any expected request. For example if "/foo" is expected 2 times followed by "/bar" 3 times, then there should be a request to "/foo" before there is a request to "/bar" but aside from that subsequent "/foo" and "/bar" requests can come at any time.

As an alternative to all of the above the client-side test support also provides a `ClientHttpRequestFactory` implementation that can be configured into a `RestTemplate` to bind it to a `MockMvc` instance. That allows processing requests using actual server-side logic but without running a server. Here is an example:

```
MockMvc mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).build();
this.restTemplate = new RestTemplate(new MockMvcClientHttpRequestFactory(mockMvc));

// Test code that uses the above RestTemplate ...

mockServer.verify();
```

## Static Imports

Just like with server-side tests, the fluent API for client-side tests requires a few static imports. Those are easy to find by searching "MockRest\*". Eclipse users should add "MockRestRequestMatchers.\*" and "MockRestResponseCreators.\*" as "favorite static members" in the Eclipse preferences under *Java → Editor → Content Assist → Favorites*. That allows using content assist after typing the first character of the static method name. Other IDEs (e.g. IntelliJ) may not require any additional configuration. Just check the support for code completion on static members.

## Further Examples of Client-side REST Tests

Spring MVC Test's own tests include [example tests](#) of client-side REST tests.



## 11.7 PetClinic Example

The PetClinic application, available on [GitHub](#), illustrates several features of the `_Spring TestContext Framework_` in a JUnit 4 environment. Most test functionality is included in the `AbstractClinicTests`, for which a partial listing is shown below:

```
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());
        // ...
    }

    // ...
}
```

Notes:

- This test case extends the `AbstractTransactionalJUnit4SpringContextTests` class, from which it inherits configuration for Dependency Injection (through the `DependencyInjectionTestExecutionListener`) and transactional behavior (through the `TransactionalTestExecutionListener`).
- The `clinic` instance variable — the application object being tested — is set by Dependency Injection through `@Autowired` semantics.
- The `getVets()` method illustrates how you can use the inherited `countRowsInTable()` method to easily verify the number of rows in a given table, thus verifying correct behavior of the application code being tested. This allows for

stronger tests and lessens dependency on the exact test data. For example, you can add additional rows in the database without breaking tests.

- Like many integration tests that use a database, most of the tests in `AbstractClinicTests` depend on a minimum amount of data already in the database before the test cases run. Alternatively, you might choose to populate the database within the test fixture set up of your test cases — again, within the same transaction as the tests.

The PetClinic application supports three data access technologies: JDBC, Hibernate, and JPA. By declaring `@ContextConfiguration` without any specific resource locations, the `AbstractClinicTests` class will have its application context loaded from the default location, `AbstractClinicTests-context.xml`, which declares a common `DataSource`. Subclasses specify additional context locations that must declare a `PlatformTransactionManager` and a concrete implementation of `Clinic`.

For example, the Hibernate implementation of the PetClinic tests contains the following implementation. For this example, `HibernateClinicTests` does not contain a single line of code: we only need to declare `@ContextConfiguration`, and the tests are inherited from `AbstractClinicTests`. Because `@ContextConfiguration` is declared without any specific resource locations, the `_Spring TestContext Framework_` loads an application context from all the beans defined in `AbstractClinicTests-context.xml` (i.e., the inherited locations) and `HibernateClinicTests-context.xml`, with `HibernateClinicTests-context.xml` possibly overriding beans defined in `AbstractClinicTests-context.xml`.

```
@ContextConfiguration
public class HibernateClinicTests extends AbstractClinicTests { }
```

In a large-scale application, the Spring configuration is often split across multiple files. Consequently, configuration locations are typically specified in a common base class for all application-specific integration tests. Such a base class may also add useful instance variables — populated by Dependency Injection, naturally — such as a `SessionFactory` in the case of an application using Hibernate.

As far as possible, you should have exactly the same Spring configuration files in your integration tests as in the deployed environment. One likely point of difference concerns database connection pooling and transaction infrastructure. If you are deploying to a full-blown application server, you will probably use its connection pool (available through JNDI) and JTA implementation. Thus in production you will use

a `JndiObjectFactoryBean` or `<jee:jndi-lookup>` for the `DataSource` and `JtaTransactionManager`. JNDI and JTA will not be available in out-of-container integration tests, so you should use a combination like the Commons

DBCP `BasicDataSource` and `DataSourceTransactionManager` or `HibernateTransactionManager` for them. You can factor out this variant behavior into a single XML file, having the choice between application server and a 'local' configuration separated from all other configuration, which will not vary between the test and production environments. In addition, it is advisable to use properties files for connection settings. See the PetClinic application for an example.

Consult the following resources for more information about testing:

- [JUnit](#): "A *programmer-oriented testing framework for Java*". Used by the Spring Framework in its test suite.
- [TestNG](#): A testing framework inspired by JUnit with added support for annotations, test groups, data-driven testing, distributed testing, etc.
- [AssertJ](#): "*Fluent assertions for Java*" including support for Java 8 lambdas, streams, etc.
- [Mock Objects](#): Article in Wikipedia.
- [MockObjects.com](#): Web site dedicated to mock objects, a technique for improving the design of code within test-driven development.
- [Mockito](#): Java mock library based on the [test spy](#) pattern.
- [EasyMock](#): Java library "*that provides Mock Objects for interfaces (and objects through the class extension) by generating them on the fly using Java's proxy mechanism.*" Used by the Spring Framework in its test suite.
- [JMock](#): Library that supports test-driven development of Java code with mock objects.
- [DbUnit](#): JUnit extension (also usable with Ant and Maven) targeted for database-driven projects that, among other things, puts your database into a known state between test runs.
- [The Grinder](#): Java load testing framework.

参考文档的这一部分涉及数据访问和数据访问层与业务或服务层之间的交互。

Spring的全面的事务管理支持得到了详细的介绍，随后全面介绍了Spring Framework集成的各种数据访问框架和技术。

- [Chapter 13,Transaction Management](#)
- [Chapter 14,DAO support](#)
- [Chapter 15,Data access with JDBC](#)
- [Chapter 16,Object Relational Mapping \(ORM\) Data Access](#)
- [Chapter 17,Marshalling XML using O/X Mappers](#)



# 13.1 Introduction to Spring Framework transaction management

Comprehensive transaction support is among the most compelling reasons to use the Spring Framework. The Spring Framework provides a consistent abstraction for transaction management that delivers the following benefits:

- Consistent programming model across different transaction APIs such as Java Transaction API (JTA), JDBC, Hibernate, and Java Persistence API (JPA).
- Support for [declarative transaction management](#).
- Simpler API for [programmatic transaction management](#) than complex transaction APIs such as JTA.
- Excellent integration with Spring's data access abstractions.

The following sections describe the Spring Framework's transaction value-adds and technologies. (The chapter also includes discussions of best practices, application server integration, and solutions to common problems.)

- [Advantages of the Spring Framework's transaction support model](#) describes why you would use the Spring Framework's transaction abstraction instead of EJB Container-Managed Transactions (CMT) or choosing to drive local transactions through a proprietary API such as Hibernate.
- [Understanding the Spring Framework transaction abstraction](#) outlines the core classes and describes how to configure and obtain `DataSource` instances from a variety of sources.
- [Synchronizing resources with transactions](#) describes how the application code ensures that resources are created, reused, and cleaned up properly.
- [Declarative transaction management](#) describes support for declarative transaction management.
- [Programmatic transaction management](#) covers support for programmatic (that is, explicitly coded) transaction management.
- [Transaction bound event](#) describes how you could use application events within a transaction.



Traditionally, Java EE developers have had two choices for transaction management: `_global_or_local_transactions`, both of which have profound limitations. Global and local transaction management is reviewed in the next two sections, followed by a discussion of how the Spring Framework's transaction management support addresses the limitations of the global and local transaction models.

### 13.2.1 Global transactions

Global transactions enable you to work with multiple transactional resources, typically relational databases and message queues. The application server manages global transactions through the JTA, which is a cumbersome API to use (partly due to its exception model). Furthermore, a JTA `UserTransaction` normally needs to be sourced from JNDI, meaning that `you_also_need` to use JNDI in order to use JTA. Obviously the use of global transactions would limit any potential reuse of application code, as JTA is normally only available in an application server environment.

Previously, the preferred way to use global transactions was via EJBCMT(*Container Managed Transaction*): CMT is a form of *declarative transaction management*(as distinguished from *programmatic transaction management*). EJB CMT removes the need for transaction-related JNDI lookups, although of course the use of EJB itself necessitates the use of JNDI. It removes most but not all of the need to write Java code to control transactions. The significant downside is that CMT is tied to JTA and an application server environment. Also, it is only available if one chooses to implement business logic in EJBs, or at least behind a transactional EJB facade. The negatives of EJB in general are so great that this is not an attractive proposition, especially in the face of compelling alternatives for declarative transaction management.

## 13.2.2 Local transactions

Local transactions are resource-specific, such as a transaction associated with a JDBC connection. Local transactions may be easier to use, but have significant disadvantages: they cannot work across multiple transactional resources. For example, code that manages transactions using a JDBC connection cannot run within a global JTA transaction. Because the application server is not involved in transaction management, it cannot help ensure correctness across multiple resources. (It is worth noting that most applications use a single transaction resource.) Another downside is that local transactions are invasive to the programming model.

### 13.2.3 Spring Framework's consistent programming model

Spring resolves the disadvantages of global and local transactions. It enables application developers to use a *consistent\_programming\_model\_in\_any\_environment*. You write your code once, and it can benefit from different transaction management strategies in different environments. The Spring Framework provides both declarative and programmatic transaction management. Most users prefer declarative transaction management, which is recommended in most cases.

With programmatic transaction management, developers work with the Spring Framework transaction abstraction, which can run over any underlying transaction infrastructure. With the preferred declarative model, developers typically write little or no code related to transaction management, and hence do not depend on the Spring Framework transaction API, or any other transaction API.

#### Do you need an application server for transaction management?

The Spring Framework's transaction management support changes traditional rules as to when an enterprise Java application requires an application server.

In particular, you do not need an application server simply for declarative transactions through EJBs. In fact, even if your application server has powerful JTA capabilities, you may decide that the Spring Framework's declarative transactions offer more power and a more productive programming model than EJB CMT.

Typically you need an application server's JTA capability only if your application needs to handle transactions across multiple resources, which is not a requirement for many applications. Many high-end applications use a single, highly scalable database (such as Oracle RAC) instead. Standalone transaction managers such as [Atomikos Transactions](#) and [JOTM](#) are other options. Of course, you may need other application server capabilities such as Java Message Service (JMS) and Java EE Connector Architecture (JCA).

The Spring Framework gives you the choice of when to scale your application to a fully loaded application server. Gone are the days when the only alternative to using EJB CMT or JTA was to write code with local transactions such as those on JDBC connections, and face a hefty rework if you need that code to run within global, container-managed transactions. With the Spring Framework, only some of the bean definitions in your configuration file, rather than your code, need to change.

## 13.3 Understanding the Spring Framework transaction abstraction

The key to the Spring transaction abstraction is the notion of *a transaction strategy*. A transaction strategy is defined by

the `org.springframework.transaction.PlatformTransactionManager` interface:

```
public interface PlatformTransactionManager {

    TransactionStatus getTransaction(
        TransactionDefinition definition) throws TransactionException;

    void commit(TransactionStatus status) throws TransactionException;

    void rollback(TransactionStatus status) throws TransactionException;
}
```

This is primarily a service provider interface (SPI), although it can be used [programmatically](#) from your application code. Because `PlatformTransactionManager` is an *interface*, it can be easily mocked or stubbed as necessary. It is not tied to a lookup strategy such as JNDI. `PlatformTransactionManager` implementations are defined like any other object (or bean) in the Spring Framework IoC container. This benefit alone makes Spring Framework transactions a worthwhile abstraction even when you work with JTA. Transactional code can be tested much more easily than if it used JTA directly.

Again in keeping with Spring's philosophy, the `TransactionException` that can be thrown by any of the `PlatformTransactionManager` interface's methods is [unchecked](#) (that is, it extends the `java.lang.RuntimeException` class). Transaction infrastructure failures are almost invariably fatal. In rare cases where application code can actually recover from a transaction failure, the application developer can still choose to catch and handle `TransactionException`. The salient point is that developers are [not forced to](#) do so.

The `getTransaction(..)` method returns a `TransactionStatus` object, depending on a `TransactionDefinition` parameter. The returned `TransactionStatus` might represent a new transaction, or can represent an existing transaction if a matching transaction exists in the current call stack. The implication in this latter case is that, as with Java EE transaction contexts, a `TransactionStatus` is associated with a [thread of execution](#).

The `TransactionDefinition` interface specifies:

- *Isolation* : The degree to which this transaction is isolated from the work of other transactions. For example, can this transaction see uncommitted writes from other

transactions?

- *Propagation* : Typically, all code executed within a transaction scope will run in that transaction. However, you have the option of specifying the behavior in the event that a transactional method is executed when a transaction context already exists. For example, code can continue running in the existing transaction (the common case); or the existing transaction can be suspended and a new transaction created. *Spring offers all of the transaction propagation options familiar from EJB CMT*. To read about the semantics of transaction propagation in Spring, see [Section 13.5.7, “Transaction propagation”](#).
- *Timeout* : How long this transaction runs before timing out and being rolled back automatically by the underlying transaction infrastructure.
- *Read-only status* : A read-only transaction can be used when your code reads but does not modify data. Read-only transactions can be a useful optimization in some cases, such as when you are using Hibernate.

These settings reflect standard transactional concepts. If necessary, refer to resources that discuss transaction isolation levels and other core transaction concepts. Understanding these concepts is essential to using the Spring Framework or any transaction management solution.

The `TransactionStatus` interface provides a simple way for transactional code to control transaction execution and query transaction status. The concepts should be familiar, as they are common to all transaction APIs:

```
public interface TransactionStatus extends SavepointManager {

    boolean isNewTransaction();

    boolean hasSavepoint();

    void setRollbackOnly();

    boolean isRollbackOnly();

    void flush();

    boolean isCompleted();

}
```

Regardless of whether you opt for declarative or programmatic transaction management in Spring, defining the correct `PlatformTransactionManager` implementation is absolutely essential. You typically define this implementation through dependency injection.

`PlatformTransactionManager` implementations normally require knowledge of the environment in which they work: JDBC, JTA, Hibernate, and so on. The following examples show how you can define a local `PlatformTransactionManager` implementation. (This example works with plain JDBC.)

You define a JDBC `DataSource`

```
<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>
```

The related `PlatformTransactionManager` bean definition will then have a reference to the `DataSource` definition. It will look like this:

```
<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

If you use JTA in a Java EE container then you use a container `DataSource`, obtained through JNDI, in conjunction with Spring's `JtaTransactionManager`. This is what the JTA and JNDI lookup version would look like:

```
<?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>
```

The `JtaTransactionManager` does not need to know about the `dataSource`, or any other specific resources, because it uses the container's global transaction management infrastructure.



The above definition of the `dataSource` bean uses the `<jndi-lookup/>` tag from the `jee` namespace. For more information on schema-based configuration, see [Chapter 38, XML Schema-based configuration](#), and for more information on the `<jee/>` tags see the section entitled [Section 38.2.3, “the jee schema”](#).

You can also use Hibernate local transactions easily, as shown in the following examples. In this case, you need to define a Hibernate `LocalSessionFactoryBean`, which your application code will use to obtain Hibernate `Session` instances.

The `DataSource` bean definition will be similar to the local JDBC example shown previously and thus is not shown in the following example.



If the `DataSource`, used by any non-JTA transaction manager, is looked up via JNDI and managed by a Java EE container, then it should be non-transactional because the Spring Framework, rather than the Java EE container, will manage the transactions.

The `txManager` bean in this case is of the `HibernateTransactionManager` type. In the same way as the `DataSourceTransactionManager` needs a reference to the `DataSource`, the `HibernateTransactionManager` needs a reference to the `SessionFactory`.

```

<bean id="sessionFactory" class="org.springframework.orm.hibernate5.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.hibernate5.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>

```

If you are using Hibernate and Java EE container-managed JTA transactions, then you should simply use the same `JtaTransactionManager` as in the previous JTA example for JDBC.

```
<bean id="txManager" class="org.springframework.transaction.jta.JtaTransactionManager"
/>
```



If you use JTA , then your transaction manager definition will look the same regardless of what data access technology you use, be it JDBC, Hibernate JPA or any other supported technology. This is due to the fact that JTA transactions are global transactions, which can enlist any transactional resource.

In all these cases, application code does not need to change. You can change how transactions are managed merely by changing configuration, even if that change means moving from local to global transactions or vice versa.

It should now be clear how you create different transaction managers, and how they are linked to related resources that need to be synchronized to transactions (for example `DataSourceTransactionManager` to a JDBC `DataSource`, `HibernateTransactionManager` to a Hibernate `SessionFactory`, and so forth). This section describes how the application code, directly or indirectly using a persistence API such as JDBC, Hibernate, or JPA, ensures that these resources are created, reused, and cleaned up properly. The section also discusses how transaction synchronization is triggered (optionally) through the relevant `PlatformTransactionManager`.

### 13.4.1 High-level synchronization approach

The preferred approach is to use Spring's highest level template based persistence integration APIs or to use native ORM APIs with transaction-aware factory beans or proxies for managing the native resource factories. These transaction-aware solutions internally handle resource creation and reuse, cleanup, optional transaction synchronization of the resources, and exception mapping. Thus user data access code does not have to address these tasks, but can be focused purely on non-boilerplate persistence logic. Generally, you use the native ORM API or take a\_template\_approach for JDBC access by using the `JdbcTemplate`. These solutions are detailed in subsequent chapters of this reference documentation.

## 13.4.2 Low-level synchronization approach

Classes such as `DataSourceUtils` (for JDBC), `EntityManagerFactoryUtils` (for JPA), `SessionFactoryUtils` (for Hibernate), and so on exist at a lower level. When you want the application code to deal directly with the resource types of the native persistence APIs, you use these classes to ensure that proper Spring Framework-managed instances are obtained, transactions are (optionally) synchronized, and exceptions that occur in the process are properly mapped to a consistent API.

For example, in the case of JDBC, instead of the traditional JDBC approach of calling the `getConnection()` method on the `DataSource`, you instead use Spring's `org.springframework.jdbc.datasource.DataSourceUtils` class as follows:

```
Connection conn = DataSourceUtils.getConnection(dataSource);
```

If an existing transaction already has a connection synchronized (linked) to it, that instance is returned. Otherwise, the method call triggers the creation of a new connection, which is (optionally) synchronized to any existing transaction, and made available for subsequent reuse in that same transaction. As mentioned, any `SQLException` is wrapped in a Spring Framework `CannotGetJdbcConnectionException`, one of the Spring Framework's hierarchy of unchecked `DataAccessExceptions`. This approach gives you more information than can be obtained easily from the `SQLException`, and ensures portability across databases, even across different persistence technologies.

This approach also works without Spring transaction management (transaction synchronization is optional), so you can use it whether or not you are using Spring for transaction management.

Of course, once you have used Spring's JDBC support, JPA support or Hibernate support, you will generally prefer not to use `DataSourceutils` or the other helper classes, because you will be much happier working through the Spring abstraction than directly with the relevant APIs. For example, if you use the Spring `JdbcTemplate` or `jdbc.object` package to simplify your use of JDBC, correct connection retrieval occurs behind the scenes and you won't need to write any special code.

### 13.4.3 TransactionAwareDataSourceProxy

At the very lowest level exists the `TransactionAwareDataSourceProxy` class. This is a proxy for a target `DataSource`, which wraps the target `DataSource` to add awareness of Spring-managed transactions. In this respect, it is similar to a transactional JNDI `DataSource` as provided by a Java EE server.

It should almost never be necessary or desirable to use this class, except when existing code must be called and passed a standard JDBC `DataSource` interface implementation. In that case, it is possible that this code is usable, but participating in Spring managed transactions. It is preferable to write your new code by using the higher level abstractions mentioned above.

## 13.5 Declarative transaction management



Most Spring Framework users choose declarative transaction management. This option has the least impact on application code, and hence is most consistent with the ideals of a\_non-invasive\_lightweight container.

The Spring Framework's declarative transaction management is made possible with Spring aspect-oriented programming (AOP), although, as the transactional aspects code comes with the Spring Framework distribution and may be used in a boilerplate fashion, AOP concepts do not generally have to be understood to make effective use of this code.

The Spring Framework's declarative transaction management is similar to EJB CMT in that you can specify transaction behavior (or lack of it) down to individual method level. It is possible to make a `setRollbackOnly()` call within a transaction context if necessary. The differences between the two types of transaction management are:

- Unlike EJB CMT, which is tied to JTA, the Spring Framework's declarative transaction management works in any environment. It can work with JTA transactions or local transactions using JDBC, JPA or Hibernate by simply adjusting the configuration files.
- You can apply the Spring Framework declarative transaction management to any class, not merely special classes such as EJBs.
- The Spring Framework offers declarative *rollback rules*, a feature with no EJB equivalent. Both programmatic and declarative support for rollback rules is provided.
- The Spring Framework enables you to customize transactional behavior, by using AOP. For example, you can insert custom behavior in the case of transaction rollback. You can also add arbitrary advice, along with the transactional advice. With EJB CMT, you cannot influence the container's transaction management except with `setRollbackOnly()`.
- The Spring Framework does not support propagation of transaction contexts across remote calls, as do high-end application servers. If you need this feature, we recommend that you use EJB. However, consider carefully before using such a feature, because normally, one does not want transactions to span remote calls.

### Where is TransactionProxyFactoryBean?

Declarative transaction configuration in versions of Spring 2.0 and above differs considerably from previous versions of Spring. The main difference is that there is no longer any need to configure `TransactionProxyFactoryBean` beans.

The pre-Spring 2.0 configuration style is still 100% valid configuration; think of the new `as` simply defining `TransactionProxyFactoryBean`beans` on your behalf.

The concept of rollback rules is important: they enable you to specify which exceptions (and throwables) should cause automatic rollback. You specify this declaratively, in configuration, not in Java code. So, although you can still call `setRollbackOnly()` on the `TransactionStatus` object to roll back the current transaction back, most often you can specify a rule that `MyApplicationException` must always result in rollback. The significant advantage to this option is that business objects do not depend on the transaction infrastructure. For example, they typically do not need to import Spring transaction APIs or other Spring APIs.

Although EJB container default behavior automatically rolls back the transaction on *asystem exception*(usually a runtime exception), EJB CMT does not roll back the transaction automatically on *anapplication exception*(that is, a checked exception other than `java.rmi.RemoteException` ). While the Spring default behavior for declarative transaction management follows EJB convention (roll back is automatic only on unchecked exceptions), it is often useful to customize this behavior.

## 13.5.1 Understanding the Spring Framework's declarative transaction implementation

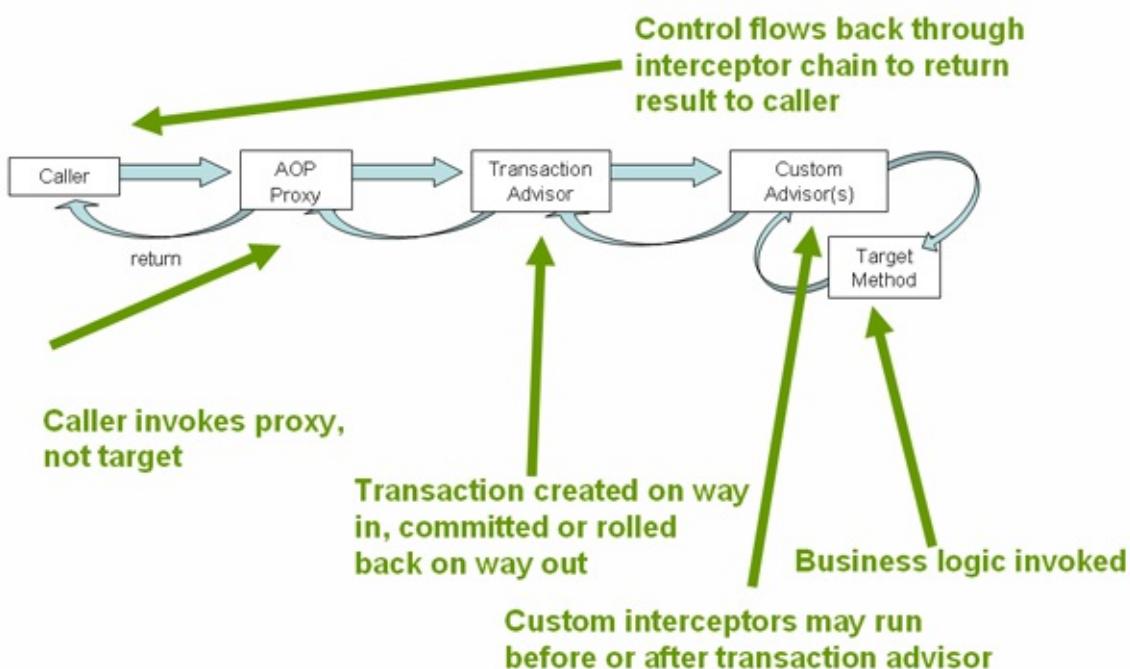
It is not sufficient to tell you simply to annotate your classes with the `@Transactional` annotation, add `@EnableTransactionManagement` to your configuration, and then expect you to understand how it all works. This section explains the inner workings of the Spring Framework's declarative transaction infrastructure in the event of transaction-related issues.

The most important concepts to grasp with regard to the Spring Framework's declarative transaction support are that this support is enabled *via AOP proxies*, and that the transactional advice is driven by *metadata* (currently XML- or annotation-based). The combination of AOP with transactional metadata yields an AOP proxy that uses a `TransactionInterceptor` in conjunction with an appropriate `PlatformTransactionManager` implementation to drive transactions *around method invocations*.



Spring AOP is covered in [Chapter 7, Aspect Oriented Programming with Spring](#).

Conceptually, calling a method on a transactional proxy looks like this...



## 13.5.2 Example of declarative transaction implementation

Consider the following interface, and its attendant implementation. This example uses `Foo` and `Bar` classes as placeholders so that you can concentrate on the transaction usage without focusing on a particular domain model. For the purposes of this example, the fact that the `DefaultFooService` class throws `UnsupportedOperationException` instances in the body of each implemented method is good; it allows you to see transactions created and then rolled back in response to the `UnsupportedOperationException` instance.

```
// 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();
    }

}
```

Assume that the first two methods of

the `FooService` interface, `getFoo(String)` and `getFoo(String, String)`, must execute in the context of a transaction with read-only semantics, and that the other methods, `insertFoo(Foo)` and `updateFoo(Foo)`, must execute in the context of a transaction with read-write semantics. The following configuration is explained in detail in the next few paragraphs.

```
<!-- 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.Foo
Service.*(..))"/>
    <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-meth
od="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.DataSourceTransact
ionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>

<!-- other <bean/> definitions here -->

</beans>

```

Examine the preceding configuration. You want to make a service object, the `fooService` bean, transactional. The transaction semantics to apply are encapsulated in the `tx:definition` tag. The definition reads as "... *all methods on starting with 'get' are to execute in the context of a read-only transaction, and all other methods are to execute with the default transaction semantics*". The `transaction-manager` attribute of the `tx:tag` is set to the name of the `PlatformTransactionManager` bean that is going to \*drive\* the transactions, in this case, the `txManager`bean.`



You can omit the `transaction-manager` attribute in the transactional advice ( ` ) if the bean name of the `PlatformTransactionManager` that you want to wire in has the name `transactionManager` . If the `PlatformTransactionManager` bean that you want to wire in has any other name, then you must use the `transaction-manager`attribute explicitly, as in the preceding example.`

The `<aop:config/>` definition ensures that the transactional advice defined by the `txAdvice` bean executes at the appropriate points in the program. First you define a pointcut that matches the execution of any operation defined in the `FooService` interface ( `fooServiceOperation` ). Then you associate the pointcut with the `txAdvice` using an advisor. The result indicates that at the execution of a `fooServiceOperation` , the advice defined by `txAdvice` will be run.

The expression defined within the `<aop:pointcut/>` element is an AspectJ pointcut expression; see[Chapter 7,Aspect Oriented Programming with Spring](#)for more details on pointcut expressions in Spring.

A common requirement is to make an entire service layer transactional. The best way to do this is simply to change the pointcut expression to match any operation in your service layer. For example:

```
<aop:config>
    <aop:pointcut id="fooServiceMethods" expression="execution(* x.y.service.*.*(..))" />
    <aop:advisor advice-ref="txAdvice" pointcut-ref="fooServiceMethods"/>
</aop:config>
```



*In this example it is assumed that all your service interfaces are defined in the `x.y.service` package; see[Chapter 7,Aspect Oriented Programming with Spring](#)for more details.*

Now that we've analyzed the configuration, you may be asking yourself, "Okay... but what does all this configuration actually do?".

The above configuration will be used to create a transactional proxy around the object that is created from the `fooService` bean definition. The proxy will be configured with the transactional advice, so that when an appropriate method is invoked on the proxy, a transaction is started, suspended, marked as read-only, and so on, depending on the transaction configuration associated with that method. Consider the following program that test drives the above configuration:

```

public final class Boot {

    public static void main(final String[] args) throws Exception {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("context.xml", Boo
t.class);
        FooService fooService = (FooService) ctx.getBean("fooService");
        fooService.insertFoo (new Foo());
    }
}

```

The output from running the preceding program will resemble the following. (The Log4J output and the stack trace from the `UnsupportedOperationException` thrown by the `insertFoo(..)` method of the `DefaultFooService` class have been truncated for clarity.)

```

<!-- 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.FooSe
rvice.insertFoo]
[DataSourceTransactionManager] - Acquired Connection [org.apache.commons.dbcp.Poolable
Connection@a53de4] for JDBC transaction

<!-- the insertFoo(..) method from DefaultFooService throws an exception... -->
[RuleBasedTransactionAttribute] - Applying rules to determine whether transaction shou
ld 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.apac
he.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.Def
aultFooService.insertFoo(DefaultFooService.java:14)
<!-- AOP infrastructure stack trace elements removed for clarity -->
at $Proxy0.insertFoo(Unknown Source)
at Boot.main(Boot.java:11)

```



### 13.5.3 Rolling back a declarative transaction

The previous section outlined the basics of how to specify transactional settings for classes, typically service layer classes, declaratively in your application. This section describes how you can control the rollback of transactions in a simple declarative fashion.

The recommended way to indicate to the Spring Framework's transaction infrastructure that a transaction's work is to be rolled back is to throw an `Exception` from code that is currently executing in the context of a transaction. The Spring Framework's transaction infrastructure code will catch any unhandled `Exception` as it bubbles up the call stack, and make a determination whether to mark the transaction for rollback.

In its default configuration, the Spring Framework's transaction infrastructure `code_only_marks` a transaction for rollback in the case of runtime, unchecked exceptions; that is, when the thrown exception is an instance or subclass of `RuntimeException`. (`Error` `s` will also - by default - result in a rollback). Checked exceptions that are thrown from a transactional method `do_not_result` in rollback in the default configuration.

You can configure exactly which `Exception` types mark a transaction for rollback, including checked exceptions. The following XML snippet demonstrates how you configure rollback for a checked, application-specific `Exception` type.

```
<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>
```

You can also specify 'no rollback rules', if you `do_not_want` a transaction rolled back when an exception is thrown. The following example tells the Spring Framework's transaction infrastructure to commit the attendant transaction even in the face of an unhandled `InstrumentNotFoundException`.

```
<tx:advice id="txAdvice">
    <tx:attributes>
        <tx:method name="updateStock" no-rollback-for="InstrumentNotFoundException"/>
        <tx:method name="*"/>
    </tx:attributes>
</tx:advice>
```

When the Spring Framework's transaction infrastructure catches an exception and is consults configured rollback rules to determine whether to mark the transaction for rollback, the \_strongest\_matching rule wins. So in the case of the following configuration, any exception other than an `InstrumentNotFoundException` results in a rollback of the attendant transaction.

```
<tx:advice id="txAdvice">
    <tx:attributes>
        <tx:method name="*" rollback-for="Throwable" no-rollback-for="InstrumentNotFoundException"/>
    </tx:attributes>
</tx:advice>
```

You can also indicate a required rollback *programmatically*. Although very simple, this process is quite invasive, and tightly couples your code to the Spring Framework's transaction infrastructure:

```
public void resolvePosition() {
    try {
        // some business logic...
    } catch (NoProductInStockException ex) {
        // trigger rollback programmatically
        TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();
    }
}
```

You are strongly encouraged to use the declarative approach to rollback if at all possible. Programmatic rollback is available should you absolutely need it, but its usage flies in the face of achieving a clean POJO-based architecture.

### 13.5.4 Configuring different transactional semantics for different beans

Consider the scenario where you have a number of service layer objects, and you want to apply a\_totally\_different\_transactional configuration to each of them. You do this by defining distinct `elements with differing pointcut and advice-ref attribute values.

As a point of comparison, first assume that all of your service layer classes are defined in a root `x.y.service` package. To make all beans that are instances of classes defined in that package (or in subpackages) and that have names ending in `Service` have the default transactional configuration, you would write the following:

```

<?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>

```

The following example shows how to configure two distinct beans with totally different transactional settings.

```

<?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>

```



## 13.5.5<tx:advice/>settings

This section summarizes the various transactional settings that can be specified using the `tx:tag`. The default settings are:

- Propagation setting is REQUIRED.
- Isolation level is DEFAULT.
- Transaction is read/write.
- Transaction timeout defaults to the default timeout of the underlying transaction system, or none if timeouts are not supported.
- Any `RuntimeException` triggers rollback, and any checked `Exception` does not.

You can change these default settings; the various attributes of the `tx:tags` that are nested within `<tx:advice/>` tags are summarized below:

**Table 13.1. settings**

Attribute	Required?	Default	Description
<code>name</code>	Yes		Method name(s) with which the transaction to be associated. The wildcard () character to associate the same transaction attribute a number of methods; for example, `get`, `handle_`, `on_Event`, and
<code>propagation</code>	No	REQUIRED	Transaction propagation behavior.
<code>isolation</code>	No	DEFAULT	Transaction isolation level.
<code>timeout</code>	No	-1	Transaction timeout value (in seconds).
<code>read-only</code>	No	false	Is this transaction read-only?
<code>rollback-for</code>	No		Exception(s) that trigger rollback; comma-separated. For example, <code>com.foo.MyBusinessException, ServiceUnavailable</code> .
<code>no-rollback-for</code>	No		Exception(s) that do_not_trigger rollback; delimited. For example, <code>com.foo.MyBusinessException, ServiceUnavailable</code> .

## 13.5.6 Using @Transactional

In addition to the XML-based declarative approach to transaction configuration, you can use an annotation-based approach. Declaring transaction semantics directly in the Java source code puts the declarations much closer to the affected code. There is not much danger of undue coupling, because code that is meant to be used transactionally is almost always deployed that way anyway.



The standard `javax.transaction.Transactional` annotation is also supported as a drop-in replacement to Spring's own annotation. Please refer to JTA 1.2 documentation for more details.

The ease-of-use afforded by the use of the `@Transactional` annotation is best illustrated with an example, which is explained in the text that follows. Consider the following class definition:

```
// 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);
}
```

When the above POJO is defined as a bean in a Spring IoC container, the bean instance can be made transactional by adding merely one line of XML configuration:

```

<!-- 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>

```



You can omit the `transaction-manager` attribute in the `<tx:annotation-driven>` tag if the bean name of the `PlatformTransactionManager` that you want to wire in has the name `transactionManager`. If the `PlatformTransactionManager` bean that you want to dependency-inject has any other name, then you have to use the `transaction-manager` attribute explicitly, as in the preceding example.`



**The `@EnableTransactionManagement` annotation provides equivalent support if you are using Java based configuration. Simply add the annotation to a `@Configuration` class. See the javadocs for full details.**

## Method visibility and `@Transactional`

When using proxies, you should apply the `@Transactional` annotation only to methods with `_public_visibility`. If you do annotate protected, private or package-visible methods with the `@Transactional` annotation, no error is raised, but the annotated method does not exhibit the configured transactional settings. Consider the use of AspectJ (see below) if you need to annotate non-public methods.

You can place the `@Transactional` annotation before an interface definition, a method on an interface, a class definition, or a `_public_method` on a class. However, the mere presence of the `@Transactional` annotation is not enough to activate the transactional behavior.

The `@Transactional` annotation is simply metadata that can be consumed by some runtime infrastructure that is `@Transactional`-aware and that can use the metadata to configure the appropriate beans with transactional behavior. In the preceding example, the `element_switches_on_the` transactional behavior.



Spring recommends that you only annotate concrete classes (and methods of concrete classes) with the `@Transactional` annotation, as opposed to annotating interfaces. You certainly can place the `@Transactional` annotation on an interface (or an interface method), but this works only as you would expect it to if you are using interface-based proxies. The fact that Java annotations are not *inherited from interfaces* means that if you are using class-based proxies (`proxy-target-class="true"`) or the weaving-based aspect (`mode="aspectj"`), then the transaction settings are not recognized by the proxying and weaving infrastructure, and the object will not be wrapped in a transactional proxy, which would be decidedly bad.



In proxy mode (which is the default), only external method calls coming in through the proxy are intercepted. This means that self-invocation, in effect, a method within the target object calling another method of the target object, will not lead to an actual transaction at runtime even if the invoked method is marked with `@Transactional`. Also, the proxy must be fully initialized to provide the expected behaviour so you should not rely on this feature in your initialization code, i.e. `@PostConstruct`.

Consider the use of AspectJ mode (see mode attribute in table below) if you expect self-invocations to be wrapped with transactions as well. In this case, there will not be a proxy in the first place; instead, the target class will be weaved (that is, its byte code will be modified) in order to turn `@Transactional` into runtime behavior on any kind of method.

**Table13.2.Annotation driven transaction settings**

XML Attribute	Annotation Attribute	Default
<code>transaction-manager</code>	N/A	<code>transactionManager</code>

mode	mode	proxy
proxy-target-class	proxyTargetClass	false
order	order	Ordered.LOWEST_PI



The `proxy-target-class` attribute controls what type of transactional proxies are created for classes annotated with the `@Transactional` annotation. If `proxy-target-class` is set to `true`, class-based proxies are created. If `proxy-target-class` is `false` or if the attribute is omitted, standard JDK interface-based proxies are created. (See [Section 7.6, "Proxying mechanisms"](#) for a discussion of the different proxy types.)



`@EnableTransactionManagement` and `only looks for `@Transactional` on beans in the same application context they are defined in. This means that, if you put annotation driven configuration in a `WebApplicationContext` for a `DispatcherServlet`, it only checks for `@Transactional`beans in your controllers, and not your services. See Section 18.2, "The DispatcherServlet" for more information.`

The most derived location takes precedence when evaluating the transactional settings for a method. In the case of the following example, the `DefaultFooService` class is annotated at the class level with the settings for a read-only transaction, but the `@Transactional` annotation on the `updateFoo(Foo)` method in the same class takes precedence over the transactional settings defined at the class level.

```
@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 settings

The `@Transactional` annotation is metadata that specifies that an interface, class, or method must have transactional semantics; for example, "*start a brand new read-only transaction when this method is invoked, suspending any existing transaction*". The default `@Transactional` settings are as follows:

- Propagation setting is `PROPAGATION_REQUIRED`.
- Isolation level is `ISOLATION_DEFAULT`.
- Transaction is read/write.
- Transaction timeout defaults to the default timeout of the underlying transaction system, or to none if timeouts are not supported.
- Any `RuntimeException` triggers rollback, and any checked `Exception` does not.

These default settings can be changed; the various properties of the `@Transactional` annotation are summarized in the following table:

**Table13.3.@Transactional Settings**

Property	Type	Description
<code>value</code>	String	Optional qualifier specifying the transaction manager to be used.
<code>propagation</code>	enum: <code>Propagation</code>	Optional propagation setting.
<code>isolation</code>	enum: <code>Isolation</code>	Optional isolation level.
<code>readOnly</code>	<code>boolean</code>	Read/write vs. read-only transaction
<code>timeout</code>	<code>int</code> (in seconds granularity)	Transaction timeout.
<code>rollbackFor</code>	Array of <code>Class</code> objects, which must be derived from <code>Throwable</code> .	Optional array of exception classes that <code>_must_cause</code> rollback.
<code>rollbackForClassName</code>	Array of class names. Classes must be derived from <code>Throwable</code> .	Optional array of names of exception classes that <code>_must_cause</code> rollback.
<code>noRollbackFor</code>	Array of <code>Class</code> objects, which must be derived from <code>Throwable</code> .	Optional array of exception classes that <code>_must_not_cause</code> rollback.
<code>noRollbackForClassName</code>	Array of <code>String</code> class names, which must be derived from <code>Throwable</code> .	Optional array of names of exception classes that <code>_must_not_cause</code> rollback.

Currently you cannot have explicit control over the name of a transaction, where 'name' means the transaction name that will be shown in a transaction monitor, if applicable (for example, WebLogic's transaction monitor), and in logging output. For declarative transactions, the transaction name is always the fully-qualified class name + "." + method

name of the transactionally-advised class. For example, if the `handlePayment(..)` method of the `BusinessService` class started a transaction, the name of the transaction would be: `com.foo.BusinessService.handlePayment`.

## Multiple Transaction Managers with `@Transactional`

Most Spring applications only need a single transaction manager, but there may be situations where you want multiple independent transaction managers in a single application. The `value` attribute of the `@Transactional` annotation can be used to optionally specify the identity of the `PlatformTransactionManager` to be used. This can either be the bean name or the qualifier value of the transaction manager bean. For example, using the qualifier notation, the following Java code

```
public class TransactionalService {

    @Transactional("order")
    public void setSomething(String name) { ... }

    @Transactional("account")
    public void doSomething() { ... }
}
```

could be combined with the following transaction manager bean declarations in the application context.

```
<tx:annotation-driven/>

<bean id="transactionManager1" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    ...
    <qualifier value="order"/>
</bean>

<bean id="transactionManager2" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    ...
    <qualifier value="account"/>
</bean>
```

In this case, the two methods on `TransactionalService` will run under separate transaction managers, differentiated by the "order" and "account" qualifiers. The default `target bean name `transactionManager`` will still be used if no specifically qualified `PlatformTransactionManager` bean is found.

## Custom shortcut annotations

If you find you are repeatedly using the same attributes with `@Transactional` on many different methods, then [Spring's meta-annotation support](#) allows you to define custom shortcut annotations for your specific use cases. For example, defining the following annotations

```
@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 {

}
```

allows us to write the example from the previous section as

```
public class TransactionalService {

    @OrderTx
    public void setSomething(String name) { ... }

    @AccountTx
    public void doSomething() { ... }
}
```

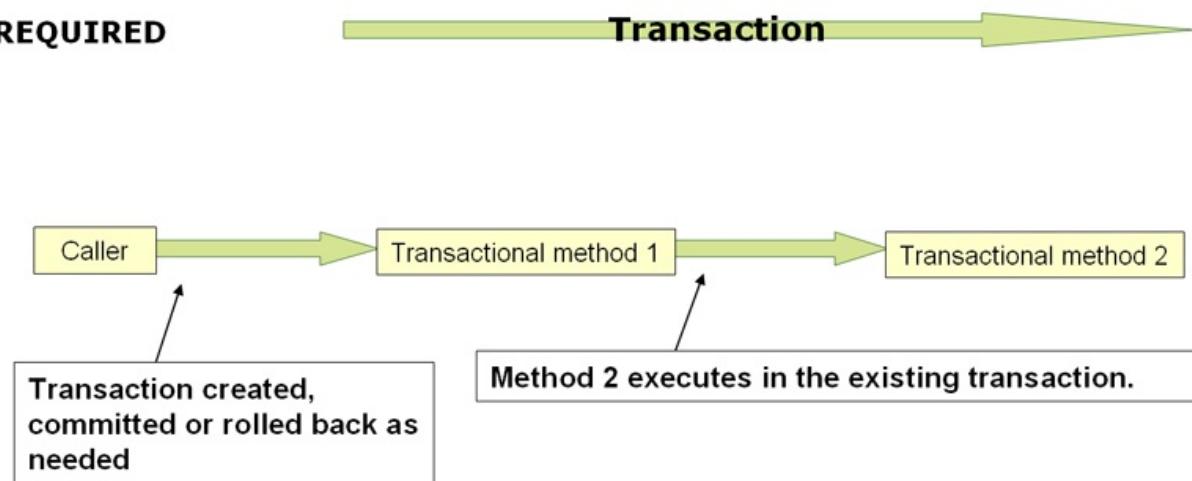
Here we have used the syntax to define the transaction manager qualifier, but could also have included propagation behavior, rollback rules, timeouts etc.

### 13.5.7 Transaction propagation

This section describes some semantics of transaction propagation in Spring. Please note that this section is not an introduction to transaction propagation proper; rather it details some of the semantics regarding transaction propagation in Spring.

In Spring-managed transactions, be aware of the difference between\_physical\_and\_logical\_transactions, and how the propagation setting applies to this difference.

#### Required



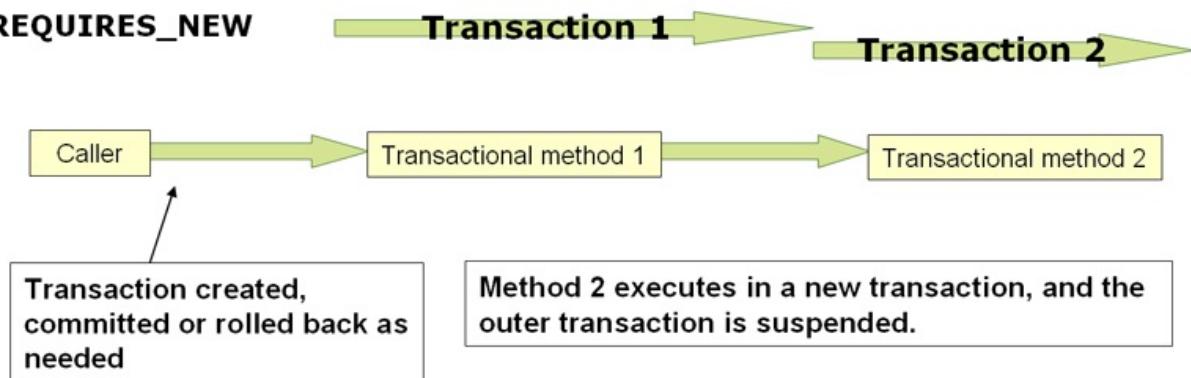
#### PROPAGATION\_REQUIRED

When the propagation setting is `PROPAGATION_REQUIRED`, a logical transaction scope is created for each method upon which the setting is applied. Each such logical transaction scope can determine rollback-only status individually, with an outer transaction scope being logically independent from the inner transaction scope. Of course, in case of standard `PROPAGATION_REQUIRED` behavior, all these scopes will be mapped to the same physical transaction. So a rollback-only marker set in the inner transaction scope does affect the outer transaction's chance to actually commit (as you would expect it to).

However, in the case where an inner transaction scope sets the rollback-only marker, the outer transaction has not decided on the rollback itself, and so the rollback (silently triggered by the inner transaction scope) is unexpected. A corresponding `UnexpectedRollbackException` is thrown at that point. This is\_expected behavior\_so that the caller of a transaction can never be misled to assume that a commit was performed when it really was not. So if an inner transaction (of which the outer caller is

not aware) silently marks a transaction as rollback-only, the outer caller still calls commit. The outer caller needs to receive an `UnexpectedRollbackException` to indicate clearly that a rollback was performed instead.

## RequiresNew



## PROPAGATION\_REQUIREMENTS\_NEW

`PROPAGATION_REQUIREMENTS_NEW`, in contrast to `PROPAGATION_REQUIRED`, uses a completely independent transaction for each affected transaction scope. In that case, the underlying physical transactions are different and hence can commit or roll back independently, with an outer transaction not affected by an inner transaction's rollback status.

## Nested

`PROPAGATION_NESTED` uses a single physical transaction with multiple savepoints that it can roll back to. Such partial rollbacks allow an inner transaction scope to trigger a rollback\_for its scope, with the outer transaction being able to continue the physical transaction despite some operations having been rolled back. This setting is typically mapped onto JDBC savepoints, so will only work with JDBC resource transactions. See Spring's `DataSourceTransactionManager`.

### 13.5.8 Advising transactional operations

Suppose you want to execute\_both\_transactional\_and\_some basic profiling advice. How do you effect this in the context of ``?

When you invoke the `updateFoo(Foo)` method, you want to see the following actions:

- Configured profiling aspect starts up.
- Transactional advice executes.
- Method on the advised object executes.
- Transaction commits.
- Profiling aspect reports exact duration of the whole transactional method invocation.



This chapter is not concerned with explaining AOP in any great detail (except as it applies to transactions). See [Chapter 7, Aspect Oriented Programming with Spring](#) for detailed coverage of the following AOP configuration and AOP in general.

Here is the code for a simple profiling aspect discussed above. The ordering of advice is controlled through the `Ordered` interface. For full details on advice ordering, see [the section called “Advice ordering”](#) . .

```
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>

```

The result of the above configuration is a `fooService` bean that has profiling and transactional aspects applied to it *in the desired order*. You configure any number of additional aspects in similar fashion.

The following example effects the same setup as above, but uses the purely XML declarative approach.

```

<?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:advisor>
        </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>

```

The result of the above configuration will be a `fooservice` bean that has profiling and transactional aspects applied to *it in that order*. If you want the profiling advice to execute \_after\_ the transactional advice on the way in, and \_before\_ the transactional advice on the way out, then you simply swap the value of the profiling aspect bean's `order` property so that it is higher than the transactional advice's order value.

You configure additional aspects in similar fashion.

### 13.5.9 Using @Transactional with AspectJ

It is also possible to use the Spring Framework's `@Transactional` support outside of a Spring container by means of an AspectJ aspect. To do so, you first annotate your classes (and optionally your classes' methods) with the `@Transactional` annotation, and then you link (weave) your application with

the `org.springframework.transaction.aspectj.AnnotationTransactionAspect` defined in the `spring-aspects.jar` file. The aspect must also be configured with a transaction manager. You can of course use the Spring Framework's IoC container to take care of dependency-injecting the aspect. The simplest way to configure the transaction management aspect is to use the `<element` and specify the `mode` attribute to `aspectj` as described in [Section 13.5.6, “Using @Transactional”](#). Because we're focusing here on applications running outside of a Spring container, we'll show you how to do it programmatically.



Prior to continuing, you may want to read [Section 13.5.6, “Using @Transactional”](#) and [Chapter 7, Aspect Oriented Programming with Spring](#) respectively.

```
// 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);
```



When using this aspect, you must annotate the *implementation\_class (and/or methods within that class), not the interface (if any) that the class implements*. AspectJ follows Java's rule that annotations on interfaces are *not* inherited.

The `@Transactional` annotation on a class specifies the default transaction semantics for the execution of any method in the class.

The `@Transactional` annotation on a method within the class overrides the default transaction semantics given by the class annotation (if present). Any method may be annotated, regardless of visibility.

To weave your applications with the `AnnotationTransactionAspect` you must either build your application with AspectJ (see the [AspectJ Development Guide](#)) or use load-time weaving. See [Section 7.8.4, “Load-time weaving with AspectJ in the Spring Framework”](#) for a discussion of load-time weaving with AspectJ.



## 13.6Programmatic transaction management

The Spring Framework provides two means of programmatic transaction management:

- Using the `TransactionTemplate`.
- Using a `PlatformTransactionManager` implementation directly.

The Spring team generally recommends the `TransactionTemplate` for programmatic transaction management. The second approach is similar to using the JTA `UserTransaction` API, although exception handling is less cumbersome.

## 13.6.1 Using the TransactionTemplate

The `TransactionTemplate` adopts the same approach as other Spring\_templates\_such as the `JdbcTemplate`. It uses a callback approach, to free application code from having to do the boilerplate acquisition and release of transactional resources, and results in code that is intention driven, in that the code that is written focuses solely on what the developer wants to do.



As you will see in the examples that follow, using the `TransactionTemplate` absolutely couples you to Spring's transaction infrastructure and APIs. Whether or not programmatic transaction management is suitable for your development needs is a decision that you will have to make yourself.

Application code that must execute in a transactional context, and that will use the `TransactionTemplate` explicitly, looks like the following. You, as an application developer, write a `TransactionCallback` implementation (typically expressed as an anonymous inner class) that contains the code that you need to execute in the context of a transaction. You then pass an instance of your custom `TransactionCallback` to the `execute(..)` method exposed on the `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();
            }
        });
    }
}
```

If there is no return value, use the convenient `TransactionCallbackWithoutResult` class with an anonymous class as follows:

```
transactionTemplate.execute(new TransactionCallbackWithoutResult() {
    protected void doInTransactionWithoutResult(TransactionStatus status) {
        updateOperation1();
        updateOperation2();
    }
});
```

Code within the callback can roll the transaction back by calling the `setRollbackOnly()` method on the supplied `TransactionStatus` object:

```
transactionTemplate.execute(new TransactionCallbackWithoutResult() {

    protected void doInTransactionWithoutResult(TransactionStatus status) {
        try {
            updateOperation1();
            updateOperation2();
        } catch (SomeBusinessException ex) {
            status.setRollbackOnly();
        }
    }
});
```

## Specifying transaction settings

You can specify transaction settings such as the propagation mode, the isolation level, the timeout, and so forth on the `TransactionTemplate` either programmatically or in configuration. `TransactionTemplate` instances by default have the [default transactional settings](#). The following example shows the programmatic customization of the transactional settings for a specific `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_REA  
D_UNCOMMITTED);  
        this.transactionTemplate.setTimeout(30); // 30 seconds  
        // and so forth...  
    }  
}
```

The following example defines a `TransactionTemplate` with some custom transactional settings, using Spring XML configuration. The `sharedTransactionTemplate` can then be injected into as many services as are required.

```
<bean id="sharedTransactionTemplate"  
      class="org.springframework.transaction.support.TransactionTemplate">  
    <property name="isolationLevelName" value="ISOLATION_READ_UNCOMMITTED"/>  
    <property name="timeout" value="30"/>  
</bean>"
```

Finally, instances of the `TransactionTemplate` class are threadsafe, in that instances do not maintain any conversational state. `TransactionTemplate` instances \_do\_ however maintain configuration state, so while a number of classes may share a single instance of a `TransactionTemplate`, if a class needs to use a `TransactionTemplate` with different settings (for example, a different isolation level), then you need to create two distinct `TransactionTemplate` instances.

## 13.6.2 Using the PlatformTransactionManager

You can also use the `org.springframework.transaction.PlatformTransactionManager` directly to manage your transaction. Simply pass the implementation of the `PlatformTransactionManager` you are using to your bean through a bean reference. Then, using the `TransactionDefinition` and `TransactionStatus` objects you can initiate transactions, roll back, and commit.

```
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);
```

## 13.7 Choosing between programmatic and declarative transaction management

Programmatic transaction management is usually a good idea only if you have a small number of transactional operations. For example, if you have a web application that require transactions only for certain update operations, you may not want to set up transactional proxies using Spring or any other technology. In this case, using the `TransactionTemplate` \_may\_be a good approach. Being able to set the transaction name explicitly is also something that can only be done using the programmatic approach to transaction management.

On the other hand, if your application has numerous transactional operations, declarative transaction management is usually worthwhile. It keeps transaction management out of business logic, and is not difficult to configure. When using the Spring Framework, rather than EJB CMT, the configuration cost of declarative transaction management is greatly reduced.

## 13.8 Transaction bound event

As of Spring 4.2, the listener of an event can be bound to a phase of the transaction. The typical example is to handle the event when the transaction has completed successfully: this allows events to be used with more flexibility when the outcome of the current transaction actually matters to the listener.

Registering a regular event listener is done via the `@EventListener` annotation. If you need to bind it to the transaction use `@TransactionalEventListener`. When you do so, the listener will be bound to the commit phase of the transaction by default.

Let's take an example to illustrate this concept. Assume that a component publish an order created event and we want to define a listener that should only handle that event once the transaction in which it has been published as committed successfully:

```
@Component
public class MyComponent {

    @TransactionalEventListener
    public void handleOrderCreatedEvent(CreationEvent<Order> creationEvent) {
        ...
    }
}
```

The `TransactionalEventListener` annotation exposes a `phase` attribute that allows to customize to which phase of the transaction the listener should be bound to. The valid phases are `BEFORE_COMMIT`, `AFTER_COMMIT` (default), `AFTER_ROLLBACK` and `AFTER_COMPLETION` that aggregates the transaction completion (be it a commit or a rollback).

If no transaction is running, the listener is not invoked at all since we can't honor the required semantics. It is however possible to override that behaviour by setting the `fallbackExecution` attribute of the annotation to `true`.

## 13.9 Application server-specific integration

Spring's transaction abstraction generally is application server agnostic. Additionally, Spring's `JtaTransactionManager` class, which can optionally perform a JNDI lookup for the `JTA UserTransaction` and `TransactionManager` objects, autodetects the location for the latter object, which varies by application server. Having access to the `JTA TransactionManager` allows for enhanced transaction semantics, in particular supporting transaction suspension. See the `JtaTransactionManager` javadocs for details.

Spring's `JtaTransactionManager` is the standard choice to run on Java EE application servers, and is known to work on all common servers. Advanced functionality such as transaction suspension works on many servers as well — including GlassFish, JBoss and Geronimo — without any special configuration required. However, for fully supported transaction suspension and further advanced integration, Spring ships special adapters for WebLogic Server and WebSphere. These adapters are discussed in the following sections.

\_For standard scenarios, including WebLogic Server and WebSphere, consider using the `convenientConfiguration` element.\_ When configured, this element automatically detects the underlying server and chooses the best transaction manager available for the platform. This means that you won't have to configure server-specific adapter classes (as discussed in the following sections) explicitly; rather, they are chosen automatically, with the standard `JtaTransactionManager` as default fallback.

## 13.9.1 IBM WebSphere

On WebSphere 6.1.0.9 and above, the recommended Spring JTA transaction manager to use is `WebsphereUowTransactionManager`. This special adapter leverages IBM's `UowManager` API, which is available in WebSphere Application Server 6.1.0.9 and later. With this adapter, Spring-driven transaction suspension (suspend/resume as initiated by `PROPAGATION_REQUIRE_NEW`) is officially supported by IBM.

## 13.9.2 Oracle WebLogic Server

On WebLogic Server 9.0 or above, you typically would use the `WebLogicJtaTransactionManager` instead of the stock `JtaTransactionManager` class. This special WebLogic-specific subclass of the normal `JtaTransactionManager` supports the full power of Spring's transaction definitions in a WebLogic-managed transaction environment, beyond standard JTA semantics: Features include transaction names, per-transaction isolation levels, and proper resuming of transactions in all cases.



### 13.10.1 Use of the wrong transaction manager for a specific DataSource

Use the *correct* `PlatformTransactionManager` implementation based on your choice of transactional technologies and requirements. Used properly, the Spring Framework merely provides a straightforward and portable abstraction. If you are using global transactions, you *must* use the `org.springframework.transaction.jta.JtaTransactionManager` class (or an *application server-specific subclass* of it) for all your transactional operations. Otherwise the transaction infrastructure attempts to perform local transactions on resources such as container `DataSource` instances. Such local transactions do not make sense, and a good application server treats them as errors.

## 13.11 Further Resources

For more information about the Spring Framework's transaction support:

- [Distributed transactions in Spring, with and without XA](#)is a JavaWorld presentation in which Spring's David Syer guides you through seven patterns for distributed transactions in Spring applications, three of them with XA and four without.
- [Java Transaction Design Strategies](#)is a book available fromInfoQthat provides a well-paced introduction to transactions in Java. It also includes side-by-side examples of how to configure and use transactions with both the Spring Framework and EJB3.



## 14.1 Introduction

The Data Access Object (DAO) support in Spring is aimed at making it easy to work with data access technologies like JDBC, Hibernate or JPA in a consistent way. This allows one to switch between the aforementioned persistence technologies fairly easily and it also allows one to code without worrying about catching exceptions that are specific to each technology.

## 14.2 Consistent exception hierarchy

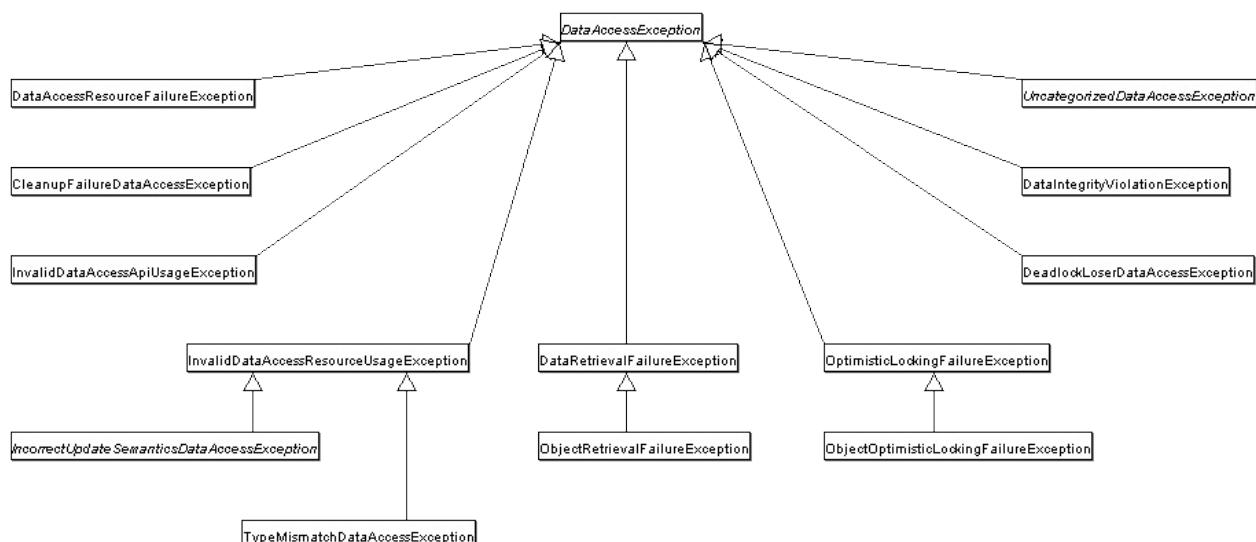
Spring provides a convenient translation from technology-specific exceptions like `SQLException` to its own exception class hierarchy with the `DataAccessException` as the root exception. These exceptions wrap the original exception so there is never any risk that one might lose any information as to what might have gone wrong.

In addition to JDBC exceptions, Spring can also wrap Hibernate-specific exceptions, converting them to a set of focused runtime exceptions (the same is true for JPA exceptions). This allows one to handle most persistence exceptions, which are non-recoverable, only in the appropriate layers, without having annoying boilerplate catch-and-throw blocks and exception declarations in one's DAOs. (One can still trap and handle exceptions anywhere one needs to though.) As mentioned above, JDBC exceptions (including database-specific dialects) are also converted to the same hierarchy, meaning that one can perform some operations with JDBC within a consistent programming model.

The above holds true for the various template classes in Spring's support for various ORM frameworks. If one uses the interceptor-based classes then the application must care about handling `HibernateExceptions` and `PersistenceExceptions` itself, preferably via delegating to `SessionFactoryUtils' convertHibernateAccessException(..)` or `convertJpaAccessException()` methods respectively. These methods convert the exceptions to ones that are compatible with the exceptions in the `org.springframework.dao` exception hierarchy.

As `PersistenceExceptions` are unchecked, they can simply get thrown too, sacrificing generic DAO abstraction in terms of exceptions though.

The exception hierarchy that Spring provides can be seen below. (Please note that the class hierarchy detailed in the image shows only a subset of the entire `DataAccessException` hierarchy.)





## 14.3 Annotations used for configuring DAO or Repository classes

The best way to guarantee that your Data Access Objects (DAOs) or repositories provide exception translation is to use the `@Repository` annotation. This annotation also allows the component scanning support to find and configure your DAOs and repositories without having to provide XML configuration entries for them.

```
@Repository  
public class SomeMovieFinder implements MovieFinder {  
    // ...  
}
```

Any DAO or repository implementation will need to access to a persistence resource, depending on the persistence technology used; for example, a JDBC-based repository will need access to a JDBC `DataSource`; a JPA-based repository will need access to an `EntityManager`. The easiest way to accomplish this is to have this resource dependency injected using one of the `@Autowired`, `@Inject`, `@Resource` or `@PersistenceContext` annotations. Here is an example for a JPA repository:

```
@Repository  
public class JpaMovieFinder implements MovieFinder {  
  
    @PersistenceContext  
    private EntityManager entityManager;  
  
    // ...  
}
```

If you are using the classic Hibernate APIs than you can inject the `SessionFactory`:

```

@Repository
public class HibernateMovieFinder implements MovieFinder {

    private SessionFactory sessionFactory;

    @Autowired
    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    // ...

}

```

Last example we will show here is for typical JDBC support. You would have the `DataSource` injected into an initialization method where you would create a `JdbcTemplate` and other data access support classes like `SimpleJdbcCall` etc using this `DataSource`.

```

@Repository
public class JdbcMovieFinder implements MovieFinder {

    private JdbcTemplate jdbcTemplate;

    @Autowired
    public void init(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    // ...

}

```



Please see the specific coverage of each persistence technology for details on how to configure the application context to take advantage of these annotations.



# 15.介绍 Spring JDBC框架

表格15.1很清楚的以操作顺序显示列举了Spring框架针对JDBC操作做的一些抽象和封装。里面区分了哪些操作Spring已经帮你做好了、哪些操作是应用开发者需要自己负责的。

表15.1. Spring JDBC – 框架和应用开发者各自分工

Action	Spring	You
定义连接参数		X
打开连接	X	
指定SQL语句		X
声明参数和提供参数值		X
准备和执行语句	X	
返回结果的迭代（如果有）	X	
具体操作每个迭代		X
异常处理	X	
事务处理.	X	
关闭连接、语句和结果集.	X	

Spring帮你屏蔽了很多JDBC底层繁琐的API操作、让你更方便的开发

## 15.1.1 选择一种JDBC数据库访问方法

JDBC数据库访问有几种基本的途径可供选择。除了JdbcTemplate的三种使用方式外，新的SimpleJdbcInsert和SimpleJdbcCall调用类通过优化数据库元数据（来简化JDBC操作），还有一种更偏向于面向对象的RDBMS对象风格的方法、有点类似于JDO的查询设计。即使你已经选择了其中一种方法、你仍然可以混合使用另外一种方法的某一个特性。所有的方法都需要JDBC2.0兼容驱动的支持，一些更高级的特性则需要使用JDBC3.0驱动支持。

- `_JdbcTemplate` 是经典的Spring JDBC访问方式，也是最常用的。这是“最基础”的方式、其他所有方式都是在 `JdbcTemplate` 的基础之上封装的。
- `_NamedParameterJdbcTemplate` 在原有 `JdbcTemplate` 的基础上做了一层包装支持命名参数特性、用于替代传统的JDBC“?”占位符。当SQL语句中包含多个参数时使用这种方式能有更好的可读性和易用性
- `_SimpleJdbcInsert` 和 `SimpleJdbcCall` 操作类主要利用JDBC驱动所提供的数据库元数据的一些特性来简化数据库操作配置。这种方式简化了编码、你只需要提供表或者存储过程的名字、以及和列名相匹配的参数Map。但前提是数据库需要提供足够的元数据。如果数据库没有提供这些元数据，需要开发者显式配置参数的映射关系。
- `_RDBMS` 对象的方式包含 `MappingSqlQuery`, `SqlUpdate` 和 `StoredProcedure`，需要你在初始化应用数据访问层时创建可重用和线程安全的对象。这种方式设计上类似于JDO查询、你可以定义查询字符串，声明参数及编译查询语句。一旦完成这些工作之后，执行方法可以根据不同的传入参数被多次调用。

## 15.1.2 包层级

Spring的JDBC框架一共包含4种不同类型的包、包括 core , datasource , object 和 support .

`org.springframework.jdbc.core` 包含 `JdbcTemplate` 类和它各种回调接口、外加一些相关的类。它的一个子包

`org.springframework.jdbc.core.simple` 包含 `SimpleJdbcInsert` 和 `SimpleJdbcCall` 等类。另一个叫 `org.springframework.jdbc.core.namedparam` 的子包包含 `NamedParameterJdbcTemplate` 及它的一些工具类。详见：

[15.2：“使用JDBC核心类控制基础的JDBC处理过程和异常处理机制”](#)，[15.4：“JDBC批量操作”](#)，[15.5：“利用SimpleJdbc类简化JDBC操作”](#)。

`org.springframework.jdbc.datasource` 包包含 `DataSource` 数据源访问的工具类，以及一些简单的 `DataSource` 实现用于测试和脱离JavaEE容器运行的JDBC代码。子包 `org.springframework.jdbc.datasource.embedded` 提供Java内置数据库例如HSQL, H2, 和 Derby的支持。详见：[15.3：“控制数据库连接”](#)，[15.8：“内置数据库支持”](#).

`org.springframework.jdbc.object` 包含用于在RDBMS查询、更新和存储过程中创建线程安全及可重用的对象类。详见[15.6：“像Java对象那样操作JDBC”](#)；这种方式类似于JDO的查询方式，不过查询返回的对象是与数据库脱离的。此包针对JDBC做了很多上层封装、而底层依赖于 `org.springframework.jdbc.core` 包。

`org.springframework.jdbc.support` 包含 `SQLException` 的转换类和一些工具类。JDBC处理过程中抛出的异常会被转换成 `org.springframework.dao` 里面定义的异常类。这意味着 SpringJDBC抽象层的代码不需要实现JDBC或者RDBMS特定的错误处理方式。所有转换的异常都没有被捕获，而是让开发者自己处理异常、具体的话既可以捕获异常也可以直接抛给上层调用者详见：[15.2.3：“SQL异常转换器”](#).



## 15.2.1 JdbcTemplate

`JdbcTemplate` 是 JDBC core 包里面的核心类。它封装了对资源的创建和释放，可以帮你避免忘记关闭连接等常见错误。它也包含了核心 JDBC 工作流的一些基础工作、例如执行和声明语句，而把 SQL 语句的生成以及查询结果的提取工作留给应用代码。`JdbcTemplate` 执行查询、更新 SQL 语句和调用存储过程，运行 `ResultSet` 迭代和抽取返回参数值。它也可以捕获 JDBC 异常并把它们转换成更加通用、解释性更强的异常层次结构、这些异常都定义在 `org.springframework.dao` 包里面。

当你在代码中使用了 `JdbcTemplate` 类，你只需要实现回调接口。`PreparedStatementCreator` 回调接口通过传入的 `connection` 类（该类包含 SQL 和任何必要的参数）创建已声明的语句。`CallableStatementCreator` 也提供类似的方式、该接口用于创建回调语句。`RowCallbackHandler` 用于获取结果集每一行的值。

可以在 DAO 实现类中通过传入 `DataSource` 引用来完成 `JdbcTemplate` 的初始化；也可以在 Spring IOC 容器里面配置、作为 DAO bean 的依赖 Bean 配置。



备注： `DataSource` 最好在 Spring IOC 容器里作为 Bean 配置起来。在上面第一种情况下，`DataSource` bean 直接传给相关的服务；第二种情况下 `DataSource` bean 传递给 `JdbcTemplate` bean。

`JdbcTemplate` 中使用的所有 SQL 以 `DEBUG` 级别记入日志（一般情况下日志的归类是 `JdbcTemplate` 对应的全限定类名，不过如果需要对 `JdbcTemplate` 进行定制的话，可能是它的子类名）

## Examples of JdbcTemplate class usage

这一节提供了 `JdbcTemplate` 类的一些使用例子。这些例子没有囊括 `JdbcTemplate` 可提供的所有功能；全部功能和用法请详见相关的 javadocs.

### 查询(SELECT)

下面是一个简单的例子、用于获取关系表里面的行数

```
int rowCount = this.jdbcTemplate.queryForObject("select count(*) from t_actor", Integer.class);
```

使用绑定变量的简单查询：

```
int countOfActorsNamedJoe = this.jdbcTemplate.queryForObject(
    "select count(*) from t_actor where first_name = ?", Integer.class, "Joe");
```

String 查询：

```
String lastName = this.jdbcTemplate.queryForObject(
    "select last_name from t_actor where id = ?",
    new Object[]{1212L}, String.class);
```

查询和填充领域模型：

```
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;
        }
    });
});
```

查询和填充多个领域对象：

```
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;
        }
    });
});
```

如果上面的两段代码实际存在于相同的应用中，建议把 RowMapper 匿名类中重复的代码抽取到单独的类中（通常是一个静态类），方便被DAO方法引用。例如，上面的代码例子更好的写法如下：

```

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;
    }
}

```

### 使用**jdbcTemplate**实现增删改

你可以使用 `update(..)` 方法实现插入，更新和删除操作。参数值可以通过可变参数或者封装在对象内传入。

```

this.jdbcTemplate.update(
    "insert into t_actor (first_name, last_name) values (?, ?)",
    "Leonor", "Watling");

```

```

this.jdbcTemplate.update(
    "update t_actor set last_name = ? where id = ?",
    "Banjo", 5276L);

```

```

this.jdbcTemplate.update(
    "delete from actor where id = ?",
    Long.valueOf(actorId));

```

### 其他**jdbcTemplate**操作

你可以使用 `execute(..)` 方法执行任何SQL，甚至是DDL语句。这个方法可以传入回调接口、绑定可变参数数组等。

```

this.jdbcTemplate.execute("create table mytable (id integer, name varchar(100))");

```

下面的例子调用一段简单的存储过程。更复杂的存储过程支持文档[covered later](#)。

```

this.jdbcTemplate.update(
    "call SUPPORT.REFRESH_ACTORS_SUMMARY(?",
    Long.valueOf(unionId));

```

## JdbcTemplate 最佳实践

`JdbcTemplate` 实例一旦配置之后是线程安全的。这点很重要因为这样你就能够配置 `JdbcTemplate` 的单例，然后安全的将其注入到多个DAO中（或者 `repositories`）。`JdbcTemplate` 是有状态的，内部存在对 `DataSource` 的引用，但是这种状态不是会话状态。

使用`JdbcTemplate`类的常用做法是在你的Spring配置文件里配置好一个`DataSource`，然后将其依赖注入进你的DAO类中（`NamedParameterJdbcTemplate` 也是如此）。`JdbcTemplate` 在 `DataSource` 的`Setter`方法中被创建。就像如下DAO类的写法一样：

```
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...
}
```

相关的配置是这样的：

```
<?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 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"/>

```

另一种替代显式配置的方式是使用component-scanning和注解注入。在这个场景下需要添加 @Repository 注解（添加这个注解可以被component-scanning扫描到），同时在 `DataSource` 的 `Setter`方法上添加 `@Autowired` 注解：

```
@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...
}
```

相关的XML配置如下：

```
<?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: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 @Component classes to co
nfigure as beans -->
    <context:component-scan base-package="org.springframework.docs.test" />

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-meth
od="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>
```

如果你使用Spring的 `JdbcDaoSupport` 类，许多JDBC相关的DAO类都从该类继承过来，这个时候相关子类需要继承 `JdbcDaoSupport` 类的 `setDataSource(..)` 方法。当然你也可以选择不从这个类继承， `JdbcDaoSupport` 本身只是提供一些便利性。

无论你选择上面提到的哪种初始方式，当你在执行SQL语句时一般都不需要重新创建 `JdbcTemplate` 实例。`JdbcTemplate`一旦被配置后其实例都是线程安全的。当你的应用需要访问多个数据库时你可能也需要多个 `JdbcTemplate` 实例，相应的也需要多个 `DataSources`，同时对应多个 `JdbcTemplates` 配置。

## 15.2.2 NamedParameterJdbcTemplate

`NamedParameterJdbcTemplate` 提供对 JDBC 语句命名参数的支持，而普通的 JDBC 语句只能使用经典的 ‘?’ 参数。`NamedParameterJdbcTemplate` 内部包装了 `JdbcTemplate`，很多功能是直接通过 `JdbcTemplate` 来实现的。本节主要描述 `NamedParameterJdbcTemplate` 不同于 `JdbcTemplate` 的点；即通过使用命名参数来操作 JDBC。

```
// 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 this.namedParameterJdbcTemplate.queryForObject(sql, namedParameters, Integer.class);
}
```

上面代码块可以看到 SQL 变量中命名参数的标记用法，以及 `namedParameters` 变量的相关赋值（类型为 `MapSqlParameterSource`）。

除此以外，你还可以在 `NamedParameterJdbcTemplate` 中传入 Map 风格的命名参数及相关的值。`NamedParameterJdbcTemplate` 类从 `NamedParameterJdbcOperations` 接口实现的其他方法用法是类似的，这里就不一一叙述了。

下面是一个 Map 风格的例子：

```

// 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<String, String> namedParameters = Collections.singletonMap("first_name", firstName);

    return this.namedParameterJdbcTemplate.queryForObject(sql, namedParameters, Integer.class);
}

```

与 `NamedParameterJdbcTemplate` 相关联的 `SqlParameterSource` 接口提供了很有用的功能（两者在同一个包里面）。在上面的代码片段中你已经看到了这个接口的一个实现例子（就是 `MapSqlParameterSource` 类）。`SqlParameterSource` 类是 `NamedParameterJdbcTemplate` 类的数据值来源。`MapSqlParameterSource` 实现非常简单、只是适配了 `java.util.Map`，其中 `keys` 就是参数名字，`value` 就是参数值。

另外一个 `SqlParameterSource` 的实现是 `BeanPropertySqlParameterSource` 类。这个类封装了任意一个 `JavaBean`（也就是任意符合 [JavaBen 规范](#) 的实例），在这个实现中，使用了 `JavaBean` 的属性作为命名参数的来源。

```

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(exampleAct
or);

    return this.namedParameterJdbcTemplate.queryForObject(sql, namedParameters, Intege
r.class);
}

```

之前提到过 `NamedParameterJdbcTemplate` 本身包装了经典的 `JdbcTemplate` 模板。如果你想调用只存在于 `JdbcTemplate` 类中的方法，你可以使用 `getJdbcOperations()` 方法，该方法返回 `JdbcOperations` 接口，通过这个接口你可以调用内部 `JdbcTemplate` 的方法。

`NamedParameterJdbcTemplate` 类在应用上下文的使用方式也可见：“[JdbcTemplate最佳实践](#)”



## 15.2.3 SQLExceptionTranslator

`SQLExceptionTranslator` 接口用于在 `SQLExceptions` 和 `spring` 自己的 `org.springframework.dao.DataAccessException` 之间做转换，要处理批量更新或者从文件中这是为了屏蔽底层的数据访问策略。其实现可以是比较通用的（例如，使用 JDBC 的 `SQLState` 编码），或者是更精确专有的（例如，使用 Oracle 的错误类型编码）

`SQLExceptionTranslator` 接口的默认实现是 `SQLErrorCodeSQLExceptionTranslator`，该实现使用的是指定数据库厂商的错误编码，因为要比 `SQLState` 的实现更加精确。错误码转换过程基于 JavaBean 类型的 `SQLErrorCode`。这个类通过 `SQLErrorCodeFactory` 创建和返回，`SQLErrorCodeFactory` 是一个基于 `sql-error-codes.xml` 配置内容来创建 `SQLErrorCode` 的工厂类。该配置中的数据库厂商代码基于 `DatabaseMetaData` 信息中返回的数据库产品名 `DatabaseProductName`，最终使用的就是你正在使用的实际数据库中错误码。

`SQLErrorCodeSQLExceptionTranslator` 按以下的顺序来匹配规则：



备注：`SQLErrorCodeFactory` 是用于定义错误码和自定义异常转换的缺省工厂类。错误码参照 `classpath` 下配置的 `sql-error-codes.xml` 文件内容，相匹配的 `SQLErrorCode` 实例基于正在使用的底层数据库的元数据名称。

- 是否存在自定义转换的子类。通常直接使用 `SQLErrorCodeSQLExceptionTranslator` 就可以了，因此此规则一般不会生效。只有你真正自己实现了一个子类才会生效。
- 是否存在 `SQLExceptionTranslator` 接口的自定义实现，通过 `SQLErrorCode` 类的 `customSQLExceptionTranslator` 属性指定
- `SQLErrorCode` 的 `customTranslations` 属性数组、类型为 `CustomSQLErrorCodeTranslation` 类实例列表、能否被匹配到
- 错误码被匹配到
- 使用兜底的转换器。`SQLExceptionSubclassTranslator` 是缺省的兜底转换器。如果此转换器也不存在的话只能使用 `SQLStateSQLExceptionTranslator`

你可以继承 `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;
    }
}

```

这个例子中，特定的错误码 `-12345` 被识别后单独转换，而其他的错误码则通过默认的转换器实现来处理。在使用自定义转换器时，有必要通过 `setExceptionTranslator` 方法传入 `JdbcTemplate`，并且使用 `JdbcTemplate` 来做所有的数据访问处理。下面是一个如何使用自定义转换器的例子

```

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);
}

```

自定义转换器需要传入 `dataSource` 对象为了能够获取 `sql-error-codes.xml` 定义的错误码

## 15.2.4 执行SQL语句

执行一条SQL语句非常方便。你只需要依赖 `DataSource` 和 `JdbcTemplate`，包括 `JdbcTemplate` 提供的工具方法。

下面的例子展示了如何创建一个新的数据表，虽然只有几行代码、但已经完全可用：

```
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)
    ");
    }
}
```

## 15.2.5 运行查询

一些查询方法会返回一个单一的结果。使用 `queryForObject(..)` 返回结果计数或特定值。当返回特定值类型时，将Java类型作为方法参数传入、最终返回的JDBC类型会被转换成相应的Java类型。如果这个过程中间出现类型转换错误，则会抛出 `InvalidDataAccessApiUsageException` 的异常。下面的例子包含两个查询方法，一个返回 `int` 类型、另一个返回了 `String` 类型。

```
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.queryForObject("select count(*) from mytable", Integer.class);
    }

    public String getName() {
        return this.jdbcTemplate.queryForObject("select name from mytable", String.class);
    }
}
```

除了返回单一查询结果的方法外，其他方法返回一个列表、列表中每一项代表查询返回的行记录。其中最通用的方式是 `queryForList(..)`，返回一个 `List`，列表每一项是一个 `Map` 类型，包含数据库对应行每一列的具体值。下面的代码块给上面的例子添加一个返回所有行的方法：

```
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");
}
```

返回的列表结果数据格式是这样的：

```
[{name=Bob, id=1}, {name=Mary, id=2}]
```

## 15.2.6 更新数据库

下面的例子根据主键更新其中一列值。在这个例子中，一条SQL语句包含行参数的占位符。参数值可以通过可变参数或者对象数组传入。元数据类型需要显式或者自动装箱成对应的包装类型

```
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)
    }
}
```

## 15.2.7 获取自增Key

`update()` 方法支持获取数据库自增Key。这个支持已成为 JDBC3.0 标准之一、更多细节详见 13.6 章。这个方法使用 `PreparedStatementCreator` 作为其第一个入参，该类可以指定所需的 `insert` 语句。另外一个参数是 `KeyHolder`，包含了更新操作成功之后产生的自增Key。这不是标准的创建 `PreparedStatement` 的方式。下面的例子可以在 Oracle 上面运行，但在其他平台上可能就不行了。

```
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
```



## 15.3.1DataSource

Spring用 `DataSource` 来保持与数据库的连接。`DataSource` 是JDBC规范的一部分同时是一种通用的连接工厂。它使得框架或者容器对应用代码屏蔽连接池或者事务管理等底层逻辑。作为开发者，你无需知道连接数据库的底层逻辑；这只是创建`datasource`的管理员该负责的模块。在开发测试过程中你可能需要同时扮演双重角色，但最终上线时你不需要知道生产数据源是如何配置的。

当使用Spring JDBC时，你可以通过JNDI获取数据库数据源、也可以利用第三方依赖包的连接池实现来配置。比较受欢迎的三方库有Apache Jakarta Commons DBCP 和 C3P0。在Spring产品内，有自己的数据源连接实现，但仅仅用于测试目的，同时并没有使用到连接池。

这一节使用了Spring的 `DriverManagerDataSource` 实现、其他更多的实现会在后面提到。



**注意：**仅仅使用 `DriverManagerDataSource` 类只是为了测试目的、因为此类没有连接池功能，因此在并发连接请求时性能会比较差。

通过 `DriverManagerDataSource` 获取数据库连接的方式和传统JDBC是类似的。首先指定JDBC驱动的类全名，`DriverManager` 会据此来加载驱动类。接下来、提供JDBC驱动对应的URL名称。（可以从相应驱动的文档里找到具体的名称）。然后传入用户名和密码来连接数据库。下面是一个具体配置 `DriverManagerDataSource` 连接的Java代码块：

```
DriverManagerDataSource dataSource = new DriverManagerDataSource();
dataSource.setDriverClassName("org.hsqldb.jdbcDriver");
dataSource.setUrl("jdbc:hsqldb:hsq://localhost:");
dataSource.setUsername("sa");
dataSource.setPassword("");
```

接下来是相关的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" />
```

下面的例子展示的是DBCP和C3P0的基础连接配置。如果需要连接更多的连接池选项、请查看各自连接池实现的具体产品文档

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"/>
```

### 15.3.2 DataSourceUtils

`DataSourceUtils` 类是一个方便有用的工具类，提供了从JNDI获取和关闭连接等有用的静态方法。它支持线程绑定的连接、例如：使用 `DataSourceTransactionManager` 的时候，将把数据库连接绑定到当前的线程上。

### 15.3.3 SmartDataSource

实现 `SmartDataSource` 接口的实现类需要能够提供到关系数据库的连接。它继承了 `DataSource` 接口，允许使用它的类查询是否在某个特定的操作后需要关闭连接。这在当你需要重用连接时比较有用。

### 15.3.4 AbstractDataSource

`AbstractDataSource` 是 Spring `DataSource` 实现的基础抽象类，封装了 `DataSource` 的基础通用功能。你可以继承 `AbstractDataSource` 自定义 `DataSource` 实现。

### 15.3.5 SingleConnectionDataSource

`SingleConnectionDataSource` 实现了 `SmartDataSource` 接口、内部封装了一个在每次使用后都不会关闭的单一连接。显然，这种场景下无法支持多线程。

为了防止客户端代码误以为数据库连接来自连接池（就像使用持久化工具时一样）错误的调用 `close` 方法，你应将 `suppressClose` 设置为 `true`。这样，通过该类获取的将是代理连接（禁止关闭）而不是原有的物理连接。需要注意你不能将这个类强制转换成`Oracle`等数据库的原生连接。

这个类主要用于测试目的。例如，他使得测试代码能够脱离应用服务器，很方便的在单一的 JNDI 环境下调试。和 `DriverManagerDataSource` 相反，它总是重用相同的连接，这是为了避免在测试过程中创建过多的物理连接。

### 15.3.6 DriverManagerDataSource

`DriverManagerDataSource` 类实现了标准的 `DataSource` 接口，可以通过 Java Bean 属性来配置原生的 JDBC 驱动，并且每次都返回一个新的连接。

这个实现对于测试和 JavaEE 容器以外的独立环境比较有用，无论是作为一个在 Spring IOC 容器内的 `DataSource Bean`，或是在单一的 JNDI 环境中。由于 `Connection.close()` 仅仅只是简单的关闭数据库连接，因此任何能够操作 `DataSource` 的持久层代码都能很好的工作。但是，使用 JavaBean 类型的连接池，比如 `commons-dbcp` 往往更简单、即使是在测试环境下也是如此，因此更推荐 `commons-dbcp`。

### 15.3.7 TransactionAwareDataSourceProxy

`TransactionAwareDataSourceProxy` 会创建一个目标 `DataSource` 的代理，内部包装了 `DataSource`，在此基础上添加了 Spring 事务管理功能。有点类似于 JavaEE 服务器中提供的 JNDI 事务数据源。



注意：一般情况下很少用到这个类，除非现有代码在被调用的时候需要一个标准的 JDBC `DataSource` 接口实现作为参数。在这种场景下，使用 proxy 可以仍旧重用老代码，同时能够有 Spring 管理事务的能力。更多的场景下更推荐使用 `JdbcTemplate` 和 `DataSourceUtils` 等更高抽象的资源管理类。

(更多细节请查看 `TransactionAwareDataSourceProxy` 的 JavaDoc)

### 15.3.8 DataSourceTransactionManager

`DataSourceTransactionManager` 类实现了 `PlatformTransactionManager` 接口。它将 JDBC 连接从指定的数据源绑定到当前执行的线程中，允许一个线程连接对应一个数据源。

应用代码需要通过 `DataSourceUtils.getConnection(dataSource)` 来获取 JDBC 连接，而不是通过 JavaEE 标准的 `DataSource.getConnection` 来获取。它会抛出 `org.springframework.dao` 的运行时异常而不是编译时 SQL 异常。所有框架类像 `JdbcTemplate` 都默认使用这个策略。如果不需要和这个 `DataSourceTransactionManager` 类一起使用，`DataSourceUtils` 提供的功能跟一般的数据库连接策略没有什么两样，因此它可以在任何场景下使用。

`DataSourceTransactionManager` 支持自定义隔离级别，以及 JDBC 查询超时机制。为了支持后者，应用代码必须在每个创建的语句中使用 `JdbcTemplate` 或是调用 `DataSourceUtils.applyTransactionTimeout(..)` 方法。

在单一的资源使用场景下它可以替代 `JtaTransactionManager`，不要求容器去支持 JTA。如果你严格遵循连接查找的模式的话、可以通过配置来做彼此切换。JTA 本身不支持自定义隔离级别！

## 15.4 JDBC批量操作

大多数JDBC驱动在针对同一SQL语句做批处理时能够获得更好的性能。批量更新操作可以节省数据库的来回传输次数。

## 15.4.1 使用 JdbcTemplate 来进行基础的批量操作

通过 `JdbcTemplate` 实现批处理需要实现特定接口的两个方法，`BatchPreparedStatementSetter`，并且将其作为第二个参数传入到 `batchUpdate` 方法调用中。使用 `getBatchSize` 提供当前批量操作的大小。使用 `setValues` 方法设置语句的 `Value` 参数。这个方法会按 `getBatchSize` 设置中指定的调用次数。下面的例子中通过传入列表来批量更新 `actor` 表。在这个例子中整个列表使用了批量操作：

```
public class JdbcActorDao implements ActorDao {
    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    public int[] batchUpdate(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());
                }
            }
        );
        return updateCounts;
    }

    // ... additional methods
}
```

如果你需要处理批量更新或者从文件中批量读取，你可能需要确定一个合适的批处理大小，但是最后一次批处理可能达不到这个大小。在这种场景下你可以使用 `InterruptibleBatchPreparedStatementSetter` 接口，允许在输入流耗尽之后终止批处理，`isBatchExhausted` 方法使得你可以指定批处理结束时间。

## 15.4.2 对象列表的批量处理

`JdbcTemplate` 和 `NamedParameterJdbcTemplate` 都提供了批量更新的替代方案。这个时候不是实现一个特定的批量接口，而是在调用时传入所有的值列表。框架会循环访问这些值并且使用内部的SQL语句`setter`方法。你是否已声明参数对应API是不一样的。针对已声明参数你需要传入`SqlParameterSource`数组，每项对应单次的批量操作。你可以使用 `SqlParameterSource.createBatch` 方法来创建这个数组，传入JavaBean数组或是包含参数值的Map数组。

下面是一个使用已声明参数的批量更新例子：

```
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
}
```

对于使用“?”占位符的SQL语句，你需要传入带有更新值的对象数组。对象数组每一项对应SQL语句中的一个占位符，并且传入顺序需要和SQL语句中定义的顺序保持一致。

下面是使用经典JDBC“?”占位符的例子：

```
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  
}
```

上面所有的批量更新方法都返回一个数组，包含具体成功的行数。这个计数是由JDBC驱动返回的。如果拿不到计数。JDBC驱动会返回-2。

### 15.4.3 多个批处理操作

上面最后一个例子更新的批处理数量太大，最好能再分割成更小的块。最简单的方式就是你多次调用 `batchUpdate` 来实现，但是可以有更优的方法。要使用这个方法除了SQL语句，还需要传入参数集合对象，每次Batch的更新数和一个 `ParameterizedPreparedStatementSetter` 去设置预编译SQL语句的参数值。框架会循环调用提供的值并且将更新操作切割成指定数量的小批次。

下面的例子设置了更新批次数量为100的批量更新操作：

```
public class JdbcActorDao implements ActorDao {

    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    public int[][] batchUpdate(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
}
```

这个调用的批量更新方法返回一个包含int数组的二维数组，包含每次更新生效的行数。第一层数组长度代表批处理执行的数量，第二层数组长度代表每个批处理生效的更新数。每个批处理的更新数必须和所有批处理的大小匹配，除非是最后一次批处理可能小于这个数，具体依赖于更新对象的总数。每次更新语句生效的更新数由JDBC驱动提供。如果更新数量不存在，JDBC驱动会返回-2



## 15.5 利用 **SimpleJdbc** 类简化 JDBC 操作

`SimpleJdbcInsert` 类和 `SimpleJdbcCall` 类主要利用了 JDBC 驱动所提供的数据库元数据的一些特性来简化数据库操作配置。这意味着可以在前端减少配置，当然你也可以覆盖或是关闭底层的元数据处理，在代码里面指定所有的细节。

## 15.5.1 利用SimpleJdbcInsert插入数据

Let's start by looking at the `SimpleJdbcInsert` class with the minimal amount of configuration options. You should instantiate the `SimpleJdbcInsert` in the data access layer's initialization method. For this example, the initializing method is the `setDataSource` method. You do not need to subclass the `SimpleJdbcInsert` class; simply create a new instance and set the table name using the `withTableName` method. Configuration methods for this class follow the "fluid" style that returns the instance of the `SimpleJdbcInsert`, which allows you to chain all configuration methods. This example uses only one configuration method; you will see examples of multiple ones later.

让我们首先看 `SimpleJdbcInsert` 类可提供的最小配置选项。你需要在数据访问层初始化方法里面初始化 `SimpleJdbcInsert` 类。在这个例子中，初始化方法是 `setDataSource`。你不需要继承 `SimpleJdbcInsert`，只需要简单的创建其实例同时调用 `withTableName` 设置数据库名。该类的配置方法遵循“流体”样式，返回 `SimpleJdbcInsert` 的实例，它允许您链接所有配置方法。此示例仅使用一种配置方法；稍后会看到多个例子。

```
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");
    }

    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
}
```

代码中的`execute`只传入 `java.util.Map` 作为唯一参数。需要注意的是 `Map` 里面用到的 `Key` 必须和数据库中表对应的列名一一匹配。这是因为我们需要按顺序读取元数据来构造实际的插入语句。

## 15.5.2 使用SimpleJdbcInsert获取自增Key

接下来，我们对于同样的插入语句，我们并不传入id，而是通过数据库自动获取主键的方式来创建新的Actor对象并插入数据库。当我们创建 SimpleJdbcInsert 实例时，我们不仅需要指定表名，同时我们通过 usingGeneratedKeyColumns 方法指定需要数据库自动生成主键的列名。

```
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
}
```

执行插入操作时第二种方式最大的区别是你不是在Map中指定ID，而是调用 executeAndReturnKey 方法。这个方法返回 java.lang.Number 对象，可以创建一个数值类型的实例用于我们的领域模型中。你不能仅仅依赖所有的数据库都返回一个指定的Java类； java.lang.Number 是你可以依赖的基础类。如果你有多个自增列，或者自增的值是非数值型的，你可以使用 executeAndReturnKeyHolder 方法返回的 KeyHolder

### 15.5.3 使用SimpleJdbcInsert指定列

你可以在插入操作中使用 `usingColumns` 方法来指定特定的列名

```
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  
}
```

这里插入操作的执行和你依赖元数据决定更新哪个列的方式是一样的。

### 15.5.4 使用 SqlParameterSource 提供参数值

使用 Map 来指定参数值没有问题，但不是最便捷的方法。Spring提供了一些 SqlParameterSource 接口的实现类来更方便的做这些操作。第一个是 BeanPropertySqlParameterSource ，如果你有一个JavaBean兼容的类包含具体的值，使用这个类是很方便的。他会使用相关的Getter方法来获取参数值。下面是一个例子：

```
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
}
```

另外一个选择是使用 MapSqlParameterSource ，类似于Map、但是提供了一个更便捷的 addValue 方法可以用来做链式操作。

```
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
}
```

上面这些例子可以看出、配置是一样的，区别只是切换了不同的提供参数的实现方式来执行调用。

### 15.5.5 利用SimpleJdbcCall调用存储过程

SimpleJdbcCall 利用数据库元数据的特性来查找传入的参数和返回值，这样你就不需要显式去定义他们。如果你喜欢的话也以自己定义参数，尤其对于某些参数，你无法直接将他们映射到Java类上，例如 ARRAY 类型和 STRUCT 类型的参数。下面第一个例子展示了一个存储过程，从一个MySQL数据库返回 `VARCHAR` 和 `DATE` 类型。这个存储过程例子从指定的 `actor` 记录中查询返回 `first_name`, `last_name`, 和 `birth_date` 列。

```
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;
```

`in_id` 参数包含你正在查找的 `actor` 记录的 `id` 参数返回从数据库表读取的数据

SimpleJdbcCall 和 SimpleJdbcInsert 定义的方式比较类似。你需要在数据访问层的初始化代码中初始化和配置该类。相比 `StoredProcedure` 类，你不需要创建一个子类并且不需要能够从数据库元数据中查找到的参数。下面是一个使用上面存储过程的 SimpleJdbcCall 配置例子。除了 `DataSource` 以外唯一的配置选项是存储过程的名字

```

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
}

```

调用代码包括创建包含传入参数的 `SqlParameterSource`。这里需要重视的是传入参数值名字需要和存储过程中定义的参数名称相匹配。有一种场景不需要匹配、那就是你使用元数据去确定数据库对象如何与存储过程相关联。在存储过程源代码中指定的并不一定是数据库中存储的格式。有些数据库会把名字转成大写、而另外一些会使用小写或者特定的格式。

`execute` 方法接受传入参数，同时返回一个 `Map` 包含任意的返回参数，`Map` 的 `Key` 是存储过程中指定的名字。在这个例子中它们是 `out_first_name`，`out_last_name` 和 `out_birth_date`

`execute` 方法的最后一部分使用返回的数据创建 `Actor` 对象实例。再次需要强调的是 `out` 参数的名字必须是存储过程中定义的。结果 `Map` 中存储的返回参数名必须和数据库中的返回参数名（不同的数据库可能会不一样）相匹配，为了提高你代码的可重用性，你需要在查找中区分大小写，或者使用 Spring 里面的 `LinkedCaseInsensitiveMap`。如果使用 `LinkedCaseInsensitiveMap`，你需要创建自己的 `JdbcTemplate` 并且将 `setResultsMapCaseInsensitive` 属性设置为 `True`。然后你将自定义的 `JdbcTemplate` 传入到 `SimpleJdbcCall` 的构造器中。下面是这种配置的一个例子：

```
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");  
    }  
  
    // ... additional methods  
}
```

通过这样的配置，你就可以无需担心 `out` 参数值的大小写问题。

## 15.5.6 为 SimpleJdbcCall 显式定义参数

你已经了解如何通过元数据来简化参数配置，但如果你需要的话也可以显式指定参数。这样做的方法是在创建 `SimpleJdbcCall` 类同时通过 `declareParameters` 方法进行配置，这个方式可以传入一系列的 `SqlParameter`。下面的章节会详细描述如何定义一个 `SqlParameter`



**备注：**如果你使用的数据库不是Spring支持的数据库类型的话显式定义就很有必要了。当前Spring支持以下数据库的存储过程元数据查找能力：Apache Derby, DB2, MySQL, Microsoft SQL Server, Oracle, 和 Sybase. 我们同时对某些数据库内置函数支持元数据特性：比如：MySQL、Microsoft SQL Server和Oracle。

你可以选择显式定义一个、多个，或者所有参数。当你没有显式定义参数时元数据参数仍然会被使用。当你不想用元数据查找参数功能、只想指定参数时，需要调用 `withoutProcedureColumnMetaDataAccess` 方法。假设你针对同一个数据函数定义了两个或多个不同的调用方法签名，在每一个给定的签名中你需要使用 `useInParameterNames` 来指定传入参数的名称列表。

下面是一个完全自定义的存储过程调用例子

```
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
}
```

两个例子的执行结果是一样的，区别是这个例子显式指定了所有细节，而不是仅仅依赖于数据库元数据。



## 15.5.7 如何定义SqlParameters

如何定义 SimpleJdbc 类和 RDBMS 操作类的参数，详见 15.6：“像 Java 对象那样操作 JDBC”，你需要使用 `SqlParameter` 或者是它的子类。通常需要在构造器中定义参数名和 SQL 类型。SQL 类型使用 `java.sql.Types` 常量来定义。我们已经看到过类似于如下的定义：

```
new SqlParameter("in_id", Types.NUMERIC),
new SqlOutParameter("out_first_name", Types.VARCHAR),
```

上面第一行 `SqlParameter` 定义了一个传入参数。`IN` 参数可以同时在存储过程调用和 `SqlQuery` 查询中使用，它的子类在下面的章节也有覆盖。

上面第二行 `SqlOutParameter` 定义了在一次存储过程调用中使用的返回参数。还有一个 `SqlInOutParameter` 类，可以用于输入输出参数。也就是说，它既是一个传入参数，也是一个返回值。



**备注：**参数只有被定义成 `SqlParameter` 和 `SqlInOutParameter` 才可以提供输入值。不像 `StoredProcedure` 类为了考虑向后兼容允许定义为 `SqlOutParameter` 的参数可以提供输入值

For IN parameters, in addition to the name and the SQL type, you can specify a scale for numeric data or a type name for custom database types. For `out` parameters, you can provide a `RowMapper` to handle mapping of rows returned from a `REF` cursor. Another option is to specify an `SqlReturnType` that provides an opportunity to define customized handling of the return values.

对于输入参数，除了名字和 SQL 类型，你可以定义数值区间或是自定义数据类型名。针对输出参数，你可以使用 `RowMapper` 处理从 `REF` 游标返回的行映射。另外一种选择是定义 `SqlReturnType`，可以针对返回值作自定义处理。

## 15.5.8 使用SimpleJdbcCall调用内置存储函数

调用存储函数几乎和调用存储过程的方式是一样的，唯一的区别你提供的是函数名而不是存储过程名。你可以使用 `withFunctionName` 方法作为配置的一部分表示我们想要调用一个函数，以及生成函数调用相关的字符串。一个特殊的`execute`调用，`executeFunction`，用来指定这个函数并且返回一个指定类型的函数值，这意味着你不需要从结果Map获取返回值。存储过程也有一个名字为 `executeObject` 的便捷方法，但是只要一个输出参数。下面的例子基于一个名字为 `get_actor_name` 的存储函数，返回actor的全名。下面是这个函数的Mysql源代码：

```
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;
```

我们需要在初始方法中创建 `SimpleJdbcCall` 来调用这个函数

```
public class JdbcActorDao implements ActorDao {

    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcCall funcGetActorName;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        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`方法返回一个包含函数调用返回值的字符串

### 15.5.9 从SimpleJdbcCall返回ResultSet/REF游标

调用存储过程或者函数返回结果集会相对棘手一点。一些数据库会在JDBC结果处理中返回结果集，而另外一些数据库则需要明确指定返回值的类型。两种方式都需要循环迭代结果集做额外处理。通过 SimpleJdbcCall，你可以使用 returningResultSet 方法，并定义一个 RowMapper 的实现类来处理特定的返回值。当结果集在返回结果处理过程中没有被定义名称时，返回的结果集必须与定义的 RowMapper 的实现类指定的顺序保持一致。而指定的名字也会被用作返回结果集中的名称。

下面的例子使用了一个不包含输入参数的存储过程并且返回t\_actor表的所有行。下面是这个存储过程的Mysql源代码：

```
CREATE PROCEDURE read_all_actors()
BEGIN
    SELECT a.id, a.first_name, a.last_name, a.birth_date FROM t_actor a;
END;
```

调用这个存储过程你需要定义 RowMapper。因为我们定义的Map类遵循JavaBean规范，所以我们就可以使用 BeanPropertyRowMapper 作为实现类。通过将相应的class类作为参数传入到 newInstance 方法中，我们可以创建这个实现类。

```
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",
                BeanPropertyRowMapper.newInstance(Actor.class));
    }

    public List getActorsList() {
        Map m = procReadAllActors.execute(new HashMap<String, Object>());
        return (List) m.get("actors");
    }

    // ... additional methods
}
```

execute调用传入一个空Map，因为这里不需要传入任何参数。从结果Map中提取Actors列表，并且返回给调用者。



## 15.6 像Java对象那样操作JDBC

The `org.springframework.jdbc.object` package contains classes that allow you to access the database in a more object-oriented manner. As an example, you can execute queries and get the results back as a list containing business objects with the relational column data mapped to the properties of the business object. You can also execute stored procedures and run update, delete, and insert statements.

`org.springframework.jdbc.object` 包能让你更加面向对象化的访问数据库。举个例子，用户可以执行查询并返回一个list，该list作为一个结果集将把从数据库中取出的列数据映射到业务对象的属性上。你也可以执行存储过程，包括更新、删除、插入语句。



备注：许多Spring的开发者认为下面将描述的各种RDBMS操作类（`StoredProcedure` 类除外）可以直接被 `JdbcTemplate` 代替；相对于把一个查询操作封装成一个类而言，直接调用 `JdbcTemplate` 方法将更简单而且更容易理解。但这仅仅是一种观点而已，如果你认为可以从直接使用RDBMS操作类中获取一些额外的好处，你不妨根据自己的需要和喜好进行不同的选择。.

## 15.6.1 SqlQuery

`SqlQuery` 类主要封装了SQL查询，本身可重用并且是线程安全的。子类必须实现 `newRowMapper(...)` 方法，这个方法提供了一个 `RowMapper` 实例，用于在查询执行返回时创建的结果集迭代过程中每一行映射并创建一个对象。`SqlQuery` 类一般不会直接使用；因为 `MappingSqlQuery` 子类已经提供了一个更方便从列映射到Java类的实现。其他继承 `SqlQuery` 的子类有 `MappingSqlQueryWithParameters` 和 `UpdatableSqlQuery`。

## 15.6.2 MappingSqlQuery

`MappingSqlQuery` 是一个可重用的查询类，它的子类必须实现 `mapRow(..)` 方法，将结果集返回的每一行转换成指定的对象类型。下面的例子展示了一个自定义的查询例子，将 `t_actor` 关系表的数据映射成 `Actor` 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;
    }
}
```

这个类继承了 `MappingSqlQuery`，并且传入 `Actor` 类型的泛型参数。这个自定义查询类的构造函数将 `DataSource` 作为唯一的传入参数。这个构造器中你调用父类的构造器，传入 `DataSource` 以及相应的SQL参数。该SQL用于创建 `PreparedStatement`，因此它可能包含任何在执行过程中传入参数的占位符。你必须在 `SqlParameter` 中使用 `declareParameter` 方法定义每个参数。`SqlParameter` 使用 `java.sql.Types` 定义名字和JDBC类型。在你定义了所有的参数后，你需要调用 `compile()` 方法，语句被预编译后方便后续的执行。这个类在编译后是线程安全的，一旦在DAO初始化时这些实例被创建后，它们可以作为实例变量一直被重用。

```
private ActorMappingQuery actorMappingQuery;

@Autowired
public void setDataSource(DataSource dataSource) {
    this.actorMappingQuery = new ActorMappingQuery(dataSource);
}

public Customer getCustomer(Long id) {
    return actorMappingQuery.findObject(id);
}
```

这个例子中的方法通过唯一的传入参数id获取customer实例。因为我们只需要返回一个对象，所以就简单的调用 `findObject` 类就可以了，这个方法只需要传入id参数。如果我们需要一次查询返回一个列表的话，就需要使用传入可变参数数组的执行方法。

```
public List<Actor> searchForActors(int age, String namePattern) {  
    List<Actor> actors = actorSearchMappingQuery.execute(age, namePattern);  
    return actors;  
}
```

### 15.6.3 SqlUpdate

`SqlUpdate` 封装了SQL的更新操作。和查询一样，更新对象是可以被重用的，就像所有的 `RdbmsOperation` 类，更新操作能够传入参数并且在SQL定义。类似于 `sqlQuery` 诸多 `execute(..)` 方法，这个类提供了一系列 `update(..)` 方法。`SQLUpdate` 类不是抽象类，它可以被继承，比如，实现自定义的更新方法。但是你并不需要继承 `SqlUpdate` 类来达到这个目的，你可以更简单的在SQL中设置自定义参数来实现。

```
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);
    }
}
```

## 15.6.4 StoredProcedure

`StoredProcedure` 类是所有RDBMS存储过程的抽象类。该类提供了多种 `execute(..)` 方法，其访问类型都是 `protected` 的。而不是通过提供更紧密打字的子类。

继承的 `sql` 属性将是RDBMS中存储过程的名称。

为了定义一个 `StoredProcedure` 类，你需要使用 `SqlParameter` 或者它的一个子类。你必须像下面的代码例子那样在构造函数中指定参数名和SQL类型。SQL类型使用 `java.sql.Types` 常量定义。

```
new SqlParameter("in_id", Types.NUMERIC),
    new SqlOutParameter("out_first_name", Types.VARCHAR),
```

`SqlParameter` 的第一行定义了一个输入参数。输入参数可以同时被存储过程调用和使用 `sqlQuery` 的查询语句使用，他的子类会在下面的章节提到。

第二行 `SqlOutParameter` 参数定义了一个在存储过程调用中使用的输出参数。`SqlInOutParameter` 还有一个 `InOut` 参数，该参数提供了一个输入值，同时也有返回值。

对应输入参数，除了名字和SQL类型，你还能指定返回区间数值类型和自定义数据库类型。对于输出参数你可以使用 `RowMapper` 来处理REF游标返回的行映射关系。另一个选择是指定 `SqlReturnType`，能够让你定义自定义的返回值类型。

下面的程序演示了如何调用Oracle中的 `sysdate()` 函数。为了使用存储过程函数你需要创建一个 `StoredProcedure` 的子类。在这个例子中，`StoredProcedure` 是一个内部类，但是如果你需要重用 `StoredProcedure` 你需要定义成一个顶级类。这个例子没有输入参数，但是使用 `SqlOutParameter` 类定义了一个时间类型的输出参数。`execute()` 方法执行了存储过程，并且从结果集 `Map` 中获取返回的时间数据。结果集 `Map` 中包含每个输出参数对应的项，在这个例子中就只有一项，使用了参数名作为 `key`。

```

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;
        }
    }
}

```

下面是一个包含两个输出参数的存储过程例子（在这种情况下为Oracle 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>());
    }
}

```

值得注意的是 `TitlesAndGenresStoredProcedure` 构造函数中使用过的 `declareParameter(..)` 方法的重载变量是如何传递 `RowMapper` 实现实例的。这是一种非常方便有效的重用方式。两种 `RowMapper` 实现的代码如下：

`TitleMapper` 类将 `ResultSet` 映射到提供的 `ResultSet` 中每行的一个 `Title` 域对象：

```

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;
    }
}

```

GenreMapper 类将返回结果集的每一行映射成 Genre 类

```
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"));
    }
}
```

为了将参数传递给RDBMS中定义的一个或多个输入参数给存储过程，你可以定义一个强类型的 `execute(..)` 方法，该方法将调用基类的 `protected execute(Map parameters)` 方法（具有受保护的访问权限）。例如：

```
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);
    }
}
```

## 15.7 参数和数据处理的常见问题

Spring JDBC框架提供了多个方法来处理常见的参数和数据问题

## 15.7.1 为参数设置SQL的类型信息

通常Spring通过传入的参数类型决定SQL的参数类型。可以在设定参数值的时候显式提供SQL类型。有些场景下设置NULL值是有必要的。

你可以通过以下方式来设置SQL类型信息：

- 许多 `JdbcTemplate` 的更新和查询方法需要传入额外的 `int` 数组类型的参数。这个数组使用 `java.sql.Types` 类的常量值来确定相关参数的SQL类型。每个参数会有对应的类型项。
- 你可以使用 `SqlParameter` 类来包装需要额外信息的参数值。针对每个值创建一个新的实例，并且在构造函数中传入SQL类型和参数值。你还可以传入数值类型的可选区间参数
- 对于那些使用命名参数的情况，使用 `SqlParameterSource` 类型的类比如 `BeanPropertySqlParameterSource`，或 `MapSqlParameterSource`。他们都具备了为命名参数注册SQL类型的功能。

## 15.7.2 处理BLOB和CLOB对象

你可以存储图片，其他类型的二进制数据，和数据库里面的大块文本。这些大的对象叫做 BLOBS（全称：Binary Large OBject；用于二进制数据）和CLOBS（全称：Character Large OBject；用于字符数据）。在Spring中你可以使用 `JdbcTemplate` 直接处理这些大对象，并且也可以使用RDBMS对象提供的上层抽象类，或者使用 `SimpleJdbc` 类。所有这些方法使用 `LobHandler` 接口的实现类来处理LOB数据的管理（全称：Large Object）。`LobHandler` 通过 `getLobCreator` 方法提供了对 `LobCreator` 类的访问，用于创建新的LOB插入对象。

`LobCreator/LobHandler` 提供了LOB输入和输出的支持：

- BLOB

- `byte[]` — `getBlobAsBytes` 和 `setBlobAsBytes`
- `InputStream` — `getBlobAsBinaryStream` 和 `setBlobAsBinaryStream`

- CLOB

- `String` — `getClobAsString` 和 `setClobAsString`
- `InputStream` — `getClobAsAsciiStream` 和 `setClobAsAsciiStream`
- `Reader` — `getClobAsCharacterStream` 和 `setClobAsCharacterStream`

下面的例子展示了如何创建和插入一个BLOB。后面的例子我们将举例如何从数据库中将 BLOB数据读取出来

这个例子使用了 `JdbcTemplate` 和 `AbstractLobCreatingPreparedStatementCallback` 的实现类。它主要实现了一个方法，`setValues`。这个方法提供了用于在你的SQL插入语句中设置LOB列的 `LobCreator`。

针对这个例子我们假定有一个变量 `lobHandler`，已经设置了 `DefaultLobHandler` 的一个实例。通常你可以使用依赖注入来设置这个值。

```

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) { //1
        protected void setValues(PreparedStatement ps, LobCreator lobCreator) throws S
QLException {
        ps.setLong(1, 1L);
        lobCreator.setClobAsCharacterStream(ps, 2, clobReader, (int)clobIn.length());
    } //2
    }
);
blobIs.close();
clobReader.close();

```

1. 这里例子中传入的 `lobHandler` 使用了默认实现类 `DefaultLobHandler`
2. 使用 `setClobAsCharacterStream` 传入CLOB的内容
3. 使用 `setBlobAsBinaryStream` 传入BLOB的内容



备注：如果你调用从 `DefaultLobHandler.getLobCreator()` 返回的 `LobCreator` 的 `setBlobAsBinaryStream`，`setClobAsAsciiStream`，或者 `setClobAsCharacterStream` 方法，其中 `contentLength` 参数允许传入一个负值。如果指定的内容长度是负值，`DefaultLobHandler` 会使用JDBC4.0不带长度参数的`set-stream`方法，或者直接传入驱动指定的长度；JDBC驱动对未指定长度的LOB流的支持请参见相关文档。

下面是从数据库读取LOB数据的例子。我们这里再次使用 `JdbcTempalte` 并使用相同的 `DefaultLobHandler` 实例。

```

List<Map<String, Object>> l = jdbcTemplate.query("select id, a_clob, a_blob from lob_t
able",
    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"); //1
            results.put("CLOB", clobText); byte[] blobBytes = lobHandler.getBlobAsBytes(rs, "a_blo
b"); //2
            results.put("BLOB", blobBytes); return results; } });

```

1. 使用 `getClobAsString` 获取CLOB的内容

## 2. 使用 `getBlobAsBytes` 获取BLOC的内容

### 15.7.3 传入IN语句的列表值

SQL标准允许基于一个带参数列表的表达式进行查询，一个典型的例子是 `select * from T_ACTOR where id in (1, 2, 3)`。这样的可变参数列表没有被JDBC标准直接支持；你不能定义可变数量的占位符（placeholder），只能定义固定变量的占位符，或者你在动态生成SQL字符串的时候需要提前知晓所需占位符的数量。`NamedParameterJdbcTemplate` 和 `JdbcTemplate` 都使用了后面那种方式。当你传入参数时，你需要传入一个 `java.util.List` 类型，支持基本类型。而这个list将会在SQL执行时替换占位符并传入参数。



备注：在传入多个值的时候需要注意。JDBC标准不保证你能在一个 `in` 表达式列表中传入超过100个值，不少数据库会超过这个值，但是一般都会有个上限。比如Oracle的上限是1000。

除了值列表的元数据值，你可以创建 `java.util.List` 的对象数组。这个列表支持多个在 `in` 语句内定义的表达式例如 `select * from T_ACTOR where (id, last_name) in ((1, 'Johnson'), (2, 'Harrop'))`。当然有个前提你的数据库需要支持这个语法。

## 15.7.4 处理存储过程调用的复杂类型

当你调用存储过程时有时需要使用数据库特定的复杂类型。为了兼容这些类型，当存储过程调用返回时 Spring 提供了一个 `SqlReturnType` 来处理这些类型，`SqlTypeValue` 用于存储过程的传入参数。

下面是一个用户自定义类型 `ITEM_TYPE` 的 Oracle `STRUCT` 对象的返回值例子。`SqlReturnType` 有一个方法 `getTypeValue` 必须被实现。而这个接口的实现将被用作 `SqlOutParameter` 声明的一部分。

```
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;
        }
    }));

```

你可以使用 `SqlTypeValue` 类往存储过程传入像 `TestItem` 那样的 Java 对象。你必须实现 `SqlTypeValue` 接口的 `createTypeValue` 方法。你可以使用传入的连接来创建像 `StructDescriptors` 这样的数据库指定对象，或 `ArrayDescriptors`。下面是相关的例子。

```

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) th
rows 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;
}
};


```

`SqlTypeValue` 会加入到包含输入参数的`Map`中，用于执行存储过程调用。

`SqlTypeValue` 的另外一个用法是给Oracle的存储过程传入一个数组。Oracle内部有它自己的 `ARRAY` 类，在这些例子中一定会被使用，你可以使用 `SqlTypeValue` 来创建Oracle `ARRAY` 的实例，并且设置到Java `ARRAY` 类的值中。

```

final Long[] ids = new Long[] {1L, 2L};

SqlTypeValue value = new AbstractSqlTypeValue() {
    protected Object createTypeValue(Connection conn, int sqlType, String typeName) th
rows SQLException {
    ArrayDescriptor arrayDescriptor = new ArrayDescriptor(typeName, conn);
    ARRAY idArray = new ARRAY(arrayDescriptor, conn, ids);
    return idArray;
}
};


```

## 15.8 内嵌数据库支持

`org.springframework.jdbc.datasource.embedded` 包包含对内嵌Java数据库引擎的支持。如对 [HSQL](#), [H2](#), 和 [Derby](#) 原生支持，你还可以使用扩展API来嵌入新的数据库内嵌类型和 `Datasource` 实现。

### 15.8.1 为什么使用一个内嵌数据库？

内嵌数据库因为比较轻量级所以在开发阶段比较方便有用。包括配置比较容易，启动快，方便测试，并且在开发阶段方便快速设计SQL操作

## 15.8.2 使用 Spring 配置来创建内嵌数据库

如果你想要将内嵌的数据库实例作为 Bean 配置到 Spring 的 ApplicationContext 中，使用 spring-jdbc 命名空间下的 embedded-database tag

```
<jdbc:embedded-database id="dataSource" generate-name="true">
    <jdbc:script location="classpath:schema.sql"/>
    <jdbc:script location="classpath:test-data.sql"/>
</jdbc:embedded-database>
```

上面的配置创建了一个内嵌的 HSQL 数据库，并且在 classpath 下面配置 schema.sql 和 test-data.sql 资源。同时，作为一种最佳实践，内嵌数据库会被制定一个唯一生成的名字。内嵌数据库在 Spring 容器中作为 javax.sql.DataSource Bean 类型存在，并且能够被注入到所需的数据库访问对象中。

### 15.8.3 使用编程方式创建内嵌数据库

`EmbeddedDatabaseBuilder` 提供了创建内嵌数据库的流式API。当你在独立的环境中或者是在独立的集成测试中可以使用这种方法创建一个内嵌数据库，下面是一个例子：

```
EmbeddedDatabase db = new EmbeddedDatabaseBuilder()
    .generateUniqueName(true)
    .setType(H2)
    .setScriptEncoding("UTF-8")
    .ignoreFailedDrops(true)
    .addScript("schema.sql")
    .addScripts("user_data.sql", "country_data.sql")
    .build();

// perform actions against the db (EmbeddedDatabase extends javax.sql.DataSource)

db.shutdown()
```

更多支持的细节请参见：`EmbeddedDatabaseBuilder` 的JavaDoc。

`EmbeddedDatabaseBuilder` 也可以使用Java Config类来创建内嵌数据库，下面是一个例子：

```
@Configuration
public class DataSourceConfig {

    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
            .generateUniqueName(true)
            .setType(H2)
            .setScriptEncoding("UTF-8")
            .ignoreFailedDrops(true)
            .addScript("schema.sql")
            .addScripts("user_data.sql", "country_data.sql")
            .build();
    }
}
```

## 使用HSQL

Spring 支持 HSQL 1.8.0 及以上版本。HSQL 是缺省默认内嵌数据库类型。如果显式指定 HSQL，设置 `embedded-database Tag` 的 `type` 属性值为 `HSQL`。如果使用 builder API，调用 `EmbeddedDatabaseType.HSQL` 的 `setType(EmbeddedDatabaseType)` 方法。

## 使用H2

Spring 也支持 H2 数据库。设置 `embedded-database tag` 的 `type` 类型值为 `H2` 来启用 H2。如果你使用了 builder API，调用 `setType(EmbeddedDatabaseType)` 方法设置值为 `EmbeddedDatabaseType.H2`。

## 使用 Derby

Spring 也支持 Apache Derby 10.5 及以上版本，设置 `embedded-database tag` 的 `type` 属性值为 `DERBY` 来开启 DERBY。如果你使用 builder API，调用 `setType(EmbeddedDatabaseType)` 方法设置值为 `EmbeddedDatabaseType.DERBY`。

## 15.8.5 使用内嵌数据库测试数据访问层逻辑

内嵌数据库提供了数据访问层代码的轻量级测试方案，下面是使用了内嵌数据库的数据访问层集成测试模板。使用这样的模板当内嵌数据库不需要在测试类中被重用时是有用的。不过，当你希望创建可以在test集中共享的内嵌数据库。考虑使用

[Spring TestContext测试框架](#)，同时在[Spring ApplicationContext](#) 中将内嵌数据库配置成一个 Bean，具体参见15.8.2节，“

[使用Spring配置来创建内嵌数据库](#)”和15.8.3节，“[使用编程方式创建内嵌数据库](#)”。

```
public class DataAccessIntegrationTestTemplate {  
  
    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()  
            .generateUniqueName(true)  
            .addDefaultScripts()  
            .build();  
    }  
  
    @Test  
    public void testDataAccess() {  
        JdbcTemplate template = new JdbcTemplate(db);  
        template.query( /* ... */ );  
    }  
  
    @After  
    public void tearDown() {  
        db.shutdown();  
    }  
}
```

## 15.8.6 生成内嵌数据库的唯一名字

开发团队在使用内嵌数据库时经常碰到的一个错误是：当他们的测试集想对同一个数据库创建额外的实例。这种错误在以下场景经常发生，XML配置文件或者 @Configuration 类用于创建内嵌数据库，并且相关的配置在同样的测试集的多个测试场景下都被用到（例如，在同一个JVM进程中）。例如，针对内嵌数据库的不同集成测试的 ApplicationContext 配置的区别只在当前哪个Bean定义是有效的。

这些错误的根源是Spring的 EmbeddedDatabaseFactory 工厂（ XML命名空间和Java Config 对象的 EmbeddedDatabaseBuilder 内部都用到了这个）会将内嵌数据库的名字默认设置成“testdb”。针对的场景，内嵌数据库通常设置成和 Bean Id 相同的名字。（例如，常用像“ dataSource ”的名字）。结果，接下来创建内嵌数据库的尝试都没创建一个新的数据库。相反，同样的 JDBC 链接URL 被重用。创建内嵌数据的库的尝试往往从同一个配置返回了已存在的内嵌数据库实例。

为了解决这个问题 Spring 框架 4.2 提供了生成内嵌数据库唯一名的支持。例如：

- `EmbeddedDatabaseFactory.setGenerateUniqueDatabaseName()`
- `EmbeddedDatabaseBuilder.generateUniqueName()`
- `<jdbc:embedded-database generate-name="true" ... >`

## 15.8.7 内嵌数据库扩展支持

Spring JDBC 内嵌数据库支持以下两种扩展支持：

- 实现 `EmbeddedDatabaseConfigurer` 支持新的内嵌数据库类型。
- 实现 `DataSourceFactory` 支持新的 `DataSource` 实现，例如管理内嵌数据库连接的连接池

欢迎贡献内部扩展给 Spring 社区，相关网址见：[jira.spring.io](https://jira.spring.io).

## 15.9 初始Data source

`org.springframework.jdbc.datasource.init` 用于支持初始一个现有的 `DataSource`。内嵌数据库提供了创建和初始化 `Datasource` 的一个选项，但是有时你需要在另外的服务器上初始化实例。

## 15.9.1 使用 Spring XML 来初始化数据库

如果你想要初始化一个数据库你可以设置 `DataSource` Bean 的引用，使用 `spring-jdbc` 命名空间下的 `initialize-database` 标记

```
<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>
```

上面的例子执行了数据库的两个脚本：第一个脚本创建了一个 Schema，第二个往表里插入了一个测试数据集。脚本路径可以使用 Spring 中 ant 类型的查找资源的模式（例如 `classpath*:com/foo/**/sql/*-data.sql`）。如果使用了正则表达式，脚本会按照 URL 或者文件名的词法顺序执行。

默认数据库初始器会无条件执行该脚本。有时你并不想要这么做，例如。你正在执行一个已存在测试数据集的数据库脚本。下面的通用方法会避免不小心删除数据，比如像上面的例子先创建表然后再插入-第一步在表已经存在时会失败掉。

为了能够在创建和删除已有数据方面提供更多的控制，XML 命名空间提供了更多的方式。第一个是将初始化设置开启还是关闭。这个可以根据当前环境情况设置（例如用系统变量或者环境属性 Bean 中获取一个布尔值）例如：

```
<jdbc:initialize-database data-source="dataSource"
    enabled="#{systemProperties.INITIALIZE_DATABASE}">
    <jdbc:script location="..."/>
</jdbc:initialize-database>
```

第二个选项是控制当前数据的行为，这是为了提高容错性。你能够控制初始器来忽略 SQL 里面执行脚本的特定错误、例如：

```
<jdbc:initialize-database data-source="dataSource" ignore-failures="DROPS">
    <jdbc:script location="..."/>
</jdbc:initialize-database>
```

在这个例子中我们声明我们预期有时脚本会执行到一个空的数据库，那么脚本中的 `DROP` 语句会失败掉。这个失败的 SQL `DROP` 语句会被忽略，但是其他错误会抛出一个异常。这在你的 SQL 语句不支持 `DROP ... IF EXISTS`（或类似语法）时比较有用，但是你想要在重新创建时无条件的移除所有的测试数据。在这个案例中第一个脚本通常是一系列 `DROP` 语句的集合，然后是一系列 `create` 语句。

`ignore-failures` 选项可以被设置成 `NONE` (默认值), `DROPS` (忽略失败的删除), 或 `ALL` (忽略所有失败)。

每个语句都应该以 ; 隔开，或者 ; 字符不存在于所用的脚本的话使用新的一行。你可以全局控制或者单针对一个脚本控制，例如：

```
<jdbc:initialize-database data-source="dataSource" separator="@@">
    <jdbc:script location="classpath:com/foo/sql/db-schema.sql" separator=";" />
    <jdbc:script location="classpath:com/foo/sql/db-test-data-1.sql"/>
    <jdbc:script location="classpath:com/foo/sql/db-test-data-2.sql"/>
</jdbc:initialize-database>
```

在这个例子中，两个 `test-data` 脚本使用 `@@` 作为语句分隔符，并且只有 `db-schema.sql` 使用 ; . 配置指定默认分隔符是 `@@`，并且将 `db-schema` 脚本内容覆盖默认值。

如果你需要从XML命名空间获取更多控制，你可以直接简单的使用 `DataSourceInitializer`，并且在你的应用中将其定义为一个模块

#### 初始化依赖数据库的其他模块

大多数应用只需要使用数据库初始器基本不会遇到其他问题了：这些基本在Spring上下文初始化之前不会使用到数据库。如果你的应用不属于这种情况则需要使用阅读余下的内容。

数据库初始器依赖于 `Datasource` 实例，并且在初始化`callback`方法中执行脚本（类似于XML `bean` 定义中的 `init-method`，组件中的 `@Postconstruct` 方法，或是在实现了 `InitializingBean` 类的 `afterPropertiesSet()` 方法）。如果有其他 `bean` 也依赖了同样的数据源同时也在初始化回调模块中使用数据源，那么可能会存在问题因为数据还没有初始化。

一个例子是在应用启动时缓存需要从数据库 Load 并且初始化数据。

为了解决这个问题你有两个选择：改变你的缓存初始化策略将其移到更后面的阶段，或者确保数据库会首先初始化。

如果你可以控制应用的初始化行为那么第一个选项明显更容易。下面是使用这个方式的一些建议，包括：

- 缓存做懒加载，并且只在第一次访问的时候初始化，这样会提高你应用启动时间。
- 让你的缓存或者其他独立模块通过实现 `Lifecycle` 或者 `SmartLifecycle` 接口来初始化缓存。当应用上下文启动时如果 `autoStartup` 有设置，`SmartLifecycle` 会被自动启动，而 `Lifecycle` 可以在调用 `ConfigurableApplicationContext.start()` 时手动启动。
- 使用 Spring 的 `ApplicationEvent` 或者其他类似的自定义监听事件来触发缓存初始化。`ContextRefreshedEvent` 总是在 Spring 容器加载完毕的时候使用（在所有 Bean 被初始化之后）。这类事件钩子(hook)机制很多时候非常有用（这也是 `SmartLifecycle` 的默认工作机制）

第二个选项也可以很容易实现。建议如下：

- 依赖 Spring Beanfactory 的默认行为，Bean 会按照注册顺序被初始化。你可以通过在 XML 配置中设置顺序来指定你应用模块的初始化顺序，确保数据库和数据库初始化会首先被执行。
- 将 Datasource 和业务模块隔离，通过将它们放在各自独立的 ApplicationContext 上下文实例来控制其启动顺序。（例如：父上下文包含 dataSource，子上下文包含业务模块）。这种结构在 Spring Web 应用中很常用，同时也是一种通用做法



## 16.1 介绍一下Spring中的ORM

Spring框架在实现资源管理、数据访问对象（DAO）层，和事务策略等方面，支持对Java持久化API（JPA）以及原生Hibernate的集成。以Hibernate举例来说，Spring有非常赞的IoC功能，可以解决许多典型的Hibernate配置和集成问题。开发者可以通过依赖注入来配置ORM（对象关系）映射组件支持的特性。Hibernate的这些特性可以参与Spring的资源和事务管理，并且符合Spring的通用事务和DAO层的异常体系。因此，Spring团队推荐开发者使用Spring集成的方式来开发DAO层，而不是使用原生的Hibernate或者JPA的API。老版本的Spring DAO模板现在不推荐使用了，想了解这部分内容可以参考[经典ORM使用](#)一节。

当开发者创建数据访问应用程序时，Spring会为开发者选择的ORM层对应功能进行优化。而且，开发者可以根据需要来利用Spring对集成ORM的支持，开发者应该将此集成工作与维护内部类似的基础架构服务的成本和风险进行权衡。同时，开发者在使用Spring集成的时候可以很大程度上不用考虑技术，将ORM的支持当做一个库来使用，因为所有的组件都被设计为可重用的JavaBean组件了。Spring IoC容器中的ORM十分易于配置和部署。本节中的大多数示例都是讲解在Spring容器中的来如何配置。

开发者使用Spring框架来中创建自己的ORM DAO的好处如下：

- 易于测试. Spring IoC的模式使得开发者可以轻易的替换Hibernate的 SessionFactory 实例，JDBC的 DataSource 实例，事务管理器，以及映射对象（如果有必要）的配置和实现。这一特点十分利于开发者对每个模块进行独立的测试。
- 泛化数据访问异常. Spring可以将ORM工具的异常封装起来，将所有异常（可以是受检异常）封装成运行时的 DataAccessException 体系。这一特性可以令开发者在合适的逻辑层上处理绝大多数不可修复的持久化异常，避免了大量的 catch , throw 和异常的声明。开发者还可以按需来处理这些异常。其中，JDBC异常（包括一些特定DB语言）都会被封装为相同的体系，意味着开发者即使使用不同的JDBC操作，基于不同的DB，也可以保证一致的编程模型。
- 通用的资源管理. Spring的应用上下文可以通过处理配置源的位置来灵活配置Hibernate的 SessionFactory 实例，JPA的 EntityManagerFactory 实例，JDBC的 DataSource 实例以及其他类似的资源。Spring的这一特性使得这些实例的配置十分易于管理和修改。同时，Spring还为处理持久化资源的配置提供了高效，易用和安全的处理方式。举个例子，有些代码使用了Hibernate需要使用相同的 Session 来确保高效性和正确的事务处理。Spring 通过Hibernate的 SessionFactory 来获取当前的 Session ，来透明的将 Session 绑定到当前的线程。Spring为任何本地或者JTA事务环境解决了在使用Hibernate时碰到的一些常见问题。

- 集成事务管理。开发者可以通过 `@Transactional` 注解或在XML配置文件中显式配置事务AOP Advise拦截，将ORM代码封装在声明式的AOP方法拦截器中。事务的语义和异常处理（回滚等）都可以根据开发者自己的需求来定制。在后面的章节中，[Resource and transaction management](#) 中，开发者可以在不影响ORM相关代码的情况下替换使用不同的事务管理器。例如，开发者可以在本地事务和JTA之间进行交换，并在两种情况下具有相同的完整服务（如声明式事务）。而且，JDBC相关的代码在事务上完全和处理ORM部分的代码集成。这对于不适用于ORM的数据访问非常有用，例如批处理和BLOB流式传输，仍然需要与ORM操作共享常见事务。



为了更全面的ORM支持，包括支持其他类型的数据库技术（如MongoDB），开发者可能需要查看[Spring Data](#)系列项目。如果开发者是JPA用户，则可以从<https://spring.io>的查阅[开始使用JPA访问数据指南](#)一文进行简单了解。

## 16.2 集成ORM的注意事项

本节重点介绍适用于所有集成ORM技术的注意事项。在[Section 16.3, “Hibernate”](#)一节中提供了很多关于如何配置和使用这些特性提的信息。

Spring对ORM集成的主要目的是使应用层次化，可以任意选择数据访问和事务管理技术，并且为应用对象提供松耦合结构。不再将业务逻辑依赖于数据访问或者事务策略上，不再使用基于硬编码的资源查找，不再使用难以替代的单例，不再自定义服务的注册。同时，为应用提供一个简单和一致的方法来装载对象，保证他们的重用并且尽可能不依赖于容器。所有单独的数据访问功能都可以自己使用，也可以很好地与Spring的 ApplicationContext 集成，提供基于XML的配置和不需要Spring感知的普通 JavaBean 实例。在典型的Spring应用程序中，许多重要的对象都是 JavaBean：数据访问模板，数据访问对象，事务管理器，使用数据访问对象和事务管理器的业务服务，Web视图解析器，使用业务服务的Web控制器等等。

## 16.2.1 资源和事务管理

通常企业应用都会包含很多重复的的资源管理代码。很多项目总是尝试去创造自己的解决方案，有时会为了开发的方便而牺牲对错误的处理。**Spring**为资源的配置管理提供了简单易用的解决方案，在JDBC上使用模板技术，在ORM上使用AOP拦截技术。

**Spring**的基础设施提供了合适的资源处理，同时**Spring**引入了DAO层的异常体系，可以适用于任何数据访问策略。对于JDBC直连来说，前面提及到的 `JdbcTemplate` 类提供了包括连接处理，对 `SQLException` 到 `DataAccessException` 的异常封装，同时还包含对于一些特定数据库SQL错误代码的转换。对于ORM技术来说，可以参考下一节来了解异常封装的优点。

当谈到事务管理时，`JdbcTemplate` 类通过**Spring**事务管理器挂接到**Spring**事务支持，并支持JTA和JDBC事务。**Spring**通过**Hibernate**，JPA事务管理器和JTA的支持来提供**Hibernate**和JPA这类ORM技术的支持。想了解更多关于事务的描述，可以参考第13章，[事务管理](#)。

## 16.2.2 异常转义

当在DAO层中使用Hibernate或者JPA的时候，开发者必须决定该如何处理持久化技术的一些原生异常。DAO层会根据选择技术的不同而抛出 `HibernateException` 或者 `PersistenceException`。这些异常都属于运行时异常，所以无需显式声明和捕捉。同时，开发者同时还需要处理 `IllegalArgumentException` 和 `IllegalStateException` 这类异常。一般情况下，调用方通常只能将这一类异常视为致命的异常，除非他们想要自己的应用依赖于持久性技术原生的异常体系。如果需要捕获一些特定的错误，比如乐观锁获取失败一类的错误，只能选择调用方和实现策略耦合到一起。对于那些只基于某种特定ORM技术或者不需要特殊异常处理的应用来说，使用ORM本身的异常体系的代价是可以接受的。但是，Spring可以通过 `@Repository` 注解透明地应用异常转换，以解耦调用方和ORM技术的耦合：

```
@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>
```

上面的后置处理器 `PersistenceExceptionTranslationPostProcessor`，会自动查找所有的异常转义器（实现 `PersistenceExceptionTranslator` 接口的Bean），并且拦截所有标记为 `@Repository` 注解的Bean，通过代理来拦截异常，然后通过 `PersistenceExceptionTranslator` 将DAO层异常转义后的异常抛出。

总而言之：开发者可以既基于简单的持久化技术的API和注解来实现DAO，同时还受益于Spring管理的事务，依赖注入和透明异常转换（如果需要）到Spring的自定义异常层次结构。

## 16.3 Hibernate

我们将首先介绍Spring环境中的[Hibernate 5](#)，然后通过使用Hibernate 5来演示Spring集成O/R映射器的方法。本节将详细介绍许多问题，并显示DAO实现和事务划分的不同变体。这些模式中大多数可以直接转换为所有其他支持的ORM工具。本章中的以下部分将通过简单的例子来介绍其他ORM技术。



从Spring 5.0开始，Spring需要Hibernate ORM对JPA的支持要基于4.3或更高的版本，甚至Hibernate ORM 5.0+可以针对本机Hibernate Session API进行编程。请注意，Hibernate团队可能不会在5.0之前维护任何版本，仅仅专注于5.2以后的版本。

### 16.3.1 在 Spring 容器中配置 SessionFactory

开发者可以将资源如 JDBC `DataSource` 或 Hibernate `SessionFactory` 定义为 Spring 容器中的 bean 来避免将应用程序对象绑定到硬编码的资源查找上。应用对象需要访问资源的时候，都通过对的 Bean 实例进行间接查找，详情可以通过下一节的 DAO 的定义来参考。

下面引用的应用的 XML 元数据定义就展示了如何配置 JDBC 的 `DataSource` 和 Hibernate 的 `SessionFactory` 的：

```
<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:hsq://localhost:9001"/>
        <property name="username" value="sa"/>
        <property name="password" value="" />
    </bean>

    <bean id="mySessionFactory" class="org.springframework.orm.hibernate5.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>
```

这样，从本地的 Jakarta Commons DBCP 的 `BasicDataSource` 转换到 JNDI 定位的 `DataSource` 仅仅只需要修改配置文件。

```
<beans>
    <jee:jndi-lookup id="myDataSource" jndi-name="java:comp/env/jdbc/myds"/>
</beans>
```

You can also access a JNDI-located `SessionFactory` , using Spring's `JndiObjectFactoryBean` /`to retrieve and expose it. However, that is typically not common outside of an EJB context.

开发者也可以通过Spring的 JndiObjectFactoryBean 或者 `<jee:jndi-lookup>` 来获取对应Bean以访问JNDI定位的 `SessionFactory`。但是，JNDI定位的 `SessionFactory` 在EJB上下文不常见。

### 16.3.2 基于Hibernate API来实现DAO

Hibernate有一个特性称之为上下文会话，在每个Hibernate本身每个事务都管理一个当前的 `session`。这大致相当于Spring每个事务的一个Hibernate `session` 的同步。如下的DAO的实现类就是基于简单的Hibernate API实现的：

```
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();
    }
}
```

除了需要在实例中持有 `SessionFactory` 引用以外，上面的代码风格跟Hibernate文档中的例子十分相近。Spring团队强烈建议使用这种基于实例变量的实现风格，而非守旧的 `static` `HibernateUtil` 风格(总的来说，除非绝对必要，否则尽量不要使用 `static` 变量来持有资源)。

上面DAO的实现完全符合Spring依赖注入的样式：这种方式可以很好的集成Spring IoC容器，就好像Spring的 `HibernateTemplate` 代码一样。当然，DAO层的实现也可以通过纯Java的方式来配置（比如在UT中）。简单实例化 `ProductDaoImpl` 并且调用 `setSessionFactory(...)` 即可。当然，也可以使用Spring bean来进行注入，参考如下XML配置：

```
<beans>

    <bean id="myProductDao" class="product.ProductDaoImpl">
        <property name="sessionFactory" ref="mySessionFactory"/>
    </bean>

</beans>
```

上面的DAO实现方式的好处在于只依赖于Hibernate API，而无需引入Spring的class。这从非侵入性的角度来看当然是有吸引力的，毫无疑问，这种开发方式会令Hibernate开发人员将会更加自然。

然而，DAO层会抛出Hibernate自有异常 `HibernateException`（属于非检查异常，无需显式声明和使用try-catch），但是也意味着调用方会将异常看做致命异常——除非调用方将Hibernate异常体系作为应用的异常体系来处理。而在这种情况下，除非调用方自己来实现一定的策略，否则捕获一些诸如乐观锁失败之类的特定错误是不可能的。对于强烈基于Hibernate的应用程序或不需要对特殊异常处理的应用程序，这种代价可能是可以接受的。

幸运的是，Spring的 `LocalSessionFactoryBean` 可以通过Hibernate的 `SessionFactory.getCurrentSession()` 方法为所有的Spring事务策略提供支持，使用 `HibernateTransactionManager` 返回当前的Spring管理的事务的 `Session`。当然，该方法的标准行为仍然是返回与正在进行的JTA事务相关联的当前 `Session`（如果有的话）。无论开发者是使用Spring的 `JtaTransactionManager`，EJB容器管理事务（CMT）还是JTA，都会适用此行为。

总而言之：开发者可以基于纯Hibernate API来实现DAO，同时也可集成Spring来管理事务。

### 16.3.3 声明式事务划分

Spring 团队建议开发者使用 Spring 声明式的事务支持，这样可以通过 AOP 事务拦截器来替代事务 API 的显式调用。AOP 事务拦截器可以在 Spring 容器中使用 XML 或者 Java 的注解来进行配置。这种事务拦截器可以令开发者的代码和重复的事务代码相解耦，而开发者可以将精力更多集中在业务逻辑上，而业务逻辑才是应用的核心。



在继续之前，强烈建议开发者先查阅章节 [Section 13.5, 声明式事务管理](#) 的内容。

开发者可以在服务层的代码使用注解 `@Transactional`，这样可以让 Spring 容器找到这些注解，以对其中注解了的方法提供事务语义。

```
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();
    }

}
```

开发者所需要做的就是在容器中配置 `PlatformTransactionManager` 的实现，或者是在 XML 中配置 `<tx:annotation-driver/>` 标签，这样就可以在运行时支持 `@Transactional` 的处理了。参考如下 XML 代码：

```
<?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.hibernate5.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory"/>
    </bean>

    <tx:annotation-driven/>

    <bean id="myProductService" class="product.SimpleProductService">
        <property name="productDao" ref="myProductDao"/>
    </bean>

</beans>
```

### 16.3.4 编程式事务划分

开发者可以在应用程序的更高级别上对事务进行标定，而不用考虑低级别的数据访问执行了多少操作。这样不会对业务服务的实现进行限制；只需要定义一个Spring的`PlatformTransactionManager`即可。当然，`PlatformTransactionManager`可以从多处获取，但最好是通过`setTransactionManager(..)`方法以Bean来注入，正如`ProductDAO`应该由`setProductDao(..)`方法配置一样。下面的代码显示Spring应用程序上下文中的事务管理器和业务服务的定义，以及业务方法实现的示例：

```
<beans>

    <bean id="myTxManager" class="org.springframework.orm.hibernate5.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>
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...
            }
        });
    }
}
```

Spring 的 `TransactionInterceptor` 允许任何检查的应用异常到 `callback` 代码中去，而 `TransactionTemplate` 还会非受检异常触发进行回调。`TransactionTemplate` 则会因为非受检异常或者是由应用标记事务回滚(通过 `TransactionStatus` )。`TransactionInterceptor` 也是一样的处理逻辑，但是同时还允许基于方法配置回滚策略。

### 16.3.5 事务管理策略

无论是 `TransactionTemplate` 或者是 `TransactionInterceptor` 都将实际的事务处理代理到 `PlatformTransactionManager` 实例上来进行处理的，这个实例的实现可以是一个 `HibernateTransactionManager` (包含一个 `Hibernate` 的 `SessionFactory` 通过使用 `ThreadLocal` 的 `Session`)，也可以是 `JtaTransactionManager` (代理到容器的 JTA 子系统)。开发者甚至可以使用一个自定义的 `PlatformTransactionManager` 的实现。现在，如果应用有需求需要部署分布式事务的话，只是一个配置变化，就可以从本地 `Hibernate` 事务管理切换到 JTA。简单地用 Spring 的 JTA 事务实现来替换 `Hibernate` 事务管理器即可。因为引用的 `PlatformTransactionManager` 的是通用事务管理 API，事务管理器之间的切换是无需修改代码的。

对于那些跨越了多个 `Hibernate` 会话工厂的分布式事务，只需要将 `JtaTransactionManager` 和多个 `LocalSessionFactoryBean` 定义相结合即可。每个 DAO 之后会获取一个特定的 `SessionFactory` 引用。如果所有底层 JDBC 数据源都是事务性容器，那么只要使用 `JtaTransactionManager` 作为策略实现，业务服务就可以划分任意数量的 DAO 和任意数量的会话工厂的事务。

无论是 `HibernateTransactionManager` 还是 `JtaTransactionManager` 都允许使用 JVM 级别的缓存来处理 `Hibernate`，无需基于容器的事务管理器查找，或者 JCA 连接器（如果开发者没有使用 EJB 来实例化事务的话）。

`HibernateTransactionManager` 可以为指定的数据源的 `Hibernate` JDBC 的 `connection` 转成为纯 JDBC 的访问代码。如果开发者仅访问一个数据库，则开发者完全可以不使用 JTA，通过 `Hibernate` 和 JDBC 数据访问进行高级别事务划分。如果开发者已经通过 `LocalSessionFactoryBean` 的 `dataSource` 属性与 `DataSource` 设置了传入的 `SessionFactory`，`HibernateTransactionManager` 会自动将 `Hibernate` 事务公开为 JDBC 事务。或者，开发者可以通过 `HibernateTransactionManager` 的 `dataSource` 属性的配置以确定公开事务的类型。

### 16.3.6 对比由容器管理的和本地定义的资源

开发者可以在不修改一行代码的情况下，在容器管理的JNDI `sessionFactory` 和本地定义的 `SessionFactory` 之间进行切换。是否将资源定义保留在容器中，还是仅仅留在应用中，都取决于开发者使用的事务策略。相对于Spring 定义的本地 `SessionFactory` 来说，手动注册的 JNDI `SessionFactory` 没有什么优势。通过Hibernate 的JCA连接器来发布一个 `SessionFactory` 只会令代码更符合J2EE服务标准，但是并不会带来任何实际的价值。

Spring对事务支持不限于容器。使用除JTA之外的任何策略配置，事务都可以在独立或测试环境中工作。特别是在单数据库事务的典型情况下，Spring的单一资源本地事务支持是一种轻量级和强大的替代JTA的解决方案。当开发者使用本地EJB无状态会话Bean来驱动事务时，即使只访问单个数据库，并且只使用无状态会话Bean来通过容器管理的事务来提供声明式事务，开发者的代码依然是依赖于EJB容器和JTA的。同时，以编程方式直接使用JTA也需要一个J2EE环境的。JTA不涉及JTA本身和JNDI DataSource实例方面的容器依赖关系。对于非Spring，JTA驱动的Hibernate事务，开发者必须使用Hibernate JCA连接器或开发额外的Hibernate事务代码，并为JVM级缓存正确配置 `TransactionManagerLookup`。

Spring驱动的事务可以与本地定义的Hibernate `SessionFactory` 一样工作，就像本地JDBC `DataSource`访问单个数据库一样。但是，当开发者有分布式事务的要求的情况下，只能选择使用Spring JTA事务策略。JCA连接器是需要特定容器遵循一致的部署步骤的，而且显然JCA 支持是需要放在第一位的。JCA的配置需要比部署本地资源定义和Spring驱动事务的简单web 应用程序需要更多额外的工作。同时，开发者还需要使用容器的企业版，比如，如果开发者使用的是WebLogic Express的非企业版，就是不支持JCA的。具有跨越单个数据库的本地 资源和事务的Spring应用程序适用于任何基于J2EE的Web容器（不包括JTA，JCA或EJB），如Tomcat，Resin甚至是Jetty。此外，开发者可以轻松地在桌面应用程序或测试套件中重用 中间层代码。

综合前面的叙述，如果不使用EJB，请尽量使用本地的 `SessionFactory` 设置和Spring 的 `HibernateTransactionManager` 或 `JtaTransactionManager`。开发者能够得到了前面提到的所有好处，包括适当的事务性JVM级缓存和分布式事务支持，而且没有容器部署的不便。只有 必须配合EJB使用的时候，JNDI通过JCA连接器来注册Hibernate `SessionFactory` 才有价值。

### 16.3.7 Hibernate的虚假应用服务器警告

在某些具有非常严格的 XADataSource 实现的 JTA 环境（目前只有一些 WebLogic Server 和 WebSphere 版本）中，当配置 Hibernate 时，没有考虑到 JTA 的 PlatformTransactionManager 对象，可能会在应用程序服务器日志中显示虚假警告或异常。这些警告或异常经常描述正在访问的连接不再有效，或者 JDBC 访问不再有效。这通常可能是因为事务不再有效。例如，这是 WebLogic 的一个实际异常：

```
java.sql.SQLException: The transaction is no longer active - status: 'Committed'. No further JDBC access is allowed within this transaction.
```

开发者可以通过配置令 Hibernate 意识到 Spring 中同步的 JTA PlatformTransactionManager 实例的存在，这样即可消除掉前面所说的虚假警告信息。开发者有以下两种选择：

- 如果在应用程序上下文中，开发者已经直接获取了 JTA PlatformTransactionManager 对象（可能是从 JNDI 到 JndiObjectFactoryBean 或者 <jee:jndi-lookup> 标签），并将其提供给 Spring 的 Jta TransactionManager（其中最简单的方法就是指定一个引用 bean 将此 JTA PlatformTransactionManager 实例定义为 LocalSessionFactoryBean 的 jta TransactionManager 属性的值）。Spring 之后会令 PlatformTransactionManager 对象对 Hibernate 可见。
- 更有可能开发者无法获取 JTA PlatformTransactionManager 实例，因为 Spring 的 Jta TransactionManager 是可以自己找到该实例的。因此，开发者需要配置 Hibernate 令其直接查找 JTA PlatformTransactionManager。开发者可以如 Hibernate 手册中所述那样通过在 Hibernate 配置中配置应用程序服务器特定的 TransactionManagerLookup 类来执行此操作。

本节的其余部分描述了在 PlatformTransactionManager 对 Hibernate 可见和 platformTransactionManager 对 Hibernate 不可见的情况下发生的事件序列：

当 Hibernate 未配置任何对 JTA PlatformTransactionManager 的进行查找时，JTA 事务提交时会发生以下事件：

- JTA 事务提交
- Spring 的 Jta TransactionManager 与 JTA 事务同步，所以它被 JTA 事务管理器通过 afterCompletion 回调调用。
- 在其他活动中，此同步令 Spring 通过 Hibernate 的 afterTransactionCompletion 触发回调（用于清除 Hibernate 缓存），然后在 Hibernate Session 上调用 close()，从而令 Hibernate 尝试 close() JDBC 连接。

- 在某些环境中，因为事务已经提交，应用程序服务器会认为 `Connection` 不可用，导致 `Connection.close()` 调用会触发警告或错误。

当Hibernate配置了对JTA `PlatformTransactionManager` 进行查找时，JTA事务提交时会发生以下事件：

- JTA事务准备提交
- Spring的 `JtaTransactionManager` 与JTA事务同步，所以JTA事务管理器通过 `beforeCompletion` 方法来回调事务。
- Spring确定Hibernate与JTA事务同步，并且行为与前一种情况不同。假设Hibernate `Session` 需要关闭，Spring将会关闭它。
- JTA事务提交。
- Hibernate与JTA事务同步，所以JTA事务管理器通过 `afterCompletion` 方法回调事务，可以正确清除其缓存。

## 16.4 JPA

Spring JPA在 `org.springframework.orm.jpa` 包中已经可用，Spring JPA用了Hibernate集成相似的方法来提供更易于理解的JPA支持，与此同时，了解了JPA底层实现，可以理解更多的[Java Persistence API](#)。

## 16.4.1 Spring 中 JPA 配置的三个选项

Spring JPA 支持提供了三种配置 JPA `EntityManagerFactory` 的方法，之后通过 `EntityManagerFactory` 来获取对应的实体管理器。

### LocalEntityManagerFactoryBean



通常只有在简单的部署环境中使用此选项，例如在独立应用程序或者进行集成测试时，才会使用这种方式。

`LocalEntityManagerFactoryBean` 创建一个适用于应用程序且仅使用 JPA 进行数据访问的简单部署环境的 `EntityManagerFactory`。工厂 bean 会使用 JPA `PersistenceProvider` 自动检测机制，并且在大多数情况下，仅要求开发者指定持久化单元的名称：

```
<beans>
    <bean id="myEmf" class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
        <property name="persistenceUnitName" value="myPersistenceUnit"/>
    </bean>
</beans>
```

这种形式的 JPA 部署是最简单的，同时限制也很多。开发者不能引用现有的 JDBC `DataSource` bean 定义，并且不支持全局事务。而且，持久化类的织入 (weaving) (字节码转换) 是特定于提供者的，通常需要在启动时指定特定的 JVM 代理。该选项仅适用于符合 JPA Spec 的独立应用程序或测试环境。

### 从 JNDI 中获取 EntityManagerFactory



在部署到 J2EE 服务器时可以使用此选项。检查服务器的文档来了解如何将自定义 JPA 提供程序部署到服务器中，从而对服务器进行比默认更多的个性化定制。

从 JNDI 获取 `EntityManagerFactory` (例如在 Java EE 环境中)，只需要在 XML 配置中加入配置信息即可：

```
<beans>
    <jee:jndi-lookup id="myEmf" jndi-name="persistence/myPersistenceUnit"/>
</beans>
```

此操作将采用标准J2EE引导：J2EE服务器自动检测J2EE部署描述符（例如web.xml）中 persistence-unit-ref 条目和持久性单元（实际上是应用程序jar中的 META-INF/persistence.xml 文件），并为这些持久性单元定义环境上下文位置。

在这种情况下，整个持久化单元部署（包括持久化类的织入（weaving）（字节码转换））都取决于J2EE服务器。JDBC DataSource 通过 META-INF/persistence.xml 文件中的JNDI位置进行定义；而 EntityManager 事务与服务器JTA子系统集成。Spring仅使用获取的 EntityManagerFactory，通过依赖注入将其传递给应用程序对象，通常通过 JtaTransactionManager 来管理持久性单元的事务。

如果在同一应用程序中使用多个持久性单元，则这种JNDI检索的持久性单元的bean名称应与应用程序用于引用它们的持久性单元名称相匹配，例如 @PersistenceUnit 和 @PersistenceContext 注释。

## LocalContainerEntityManagerFactoryBean



在基于Spring的应用程序环境中使用此选项来实现完整的JPA功能。这包括诸如Tomcat的Web容器，以及具有复杂持久性要求的独立应用程序和集成测试。

LocalContainerEntityManagerFactoryBean 可以完全控制 EntityManagerFactory 的配置，同时适用于需要细粒度定制的环境。

LocalContainerEntityManagerFactoryBean 会基于 persistence.xml 文件， dataSourceLookup 策略和指定的 loadTimeWeaver 来创建一个 PersistenceUnitInfo 实例。因此，可以在JNDI之外使用自定义数据源并控制织入（weaving）过程。以下示例显示 LocalContainerEntityManagerFactoryBean 的典型Bean定义：

```
<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>
```

下面的例子是一个典型的 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>
```

`<exclude-unlisted-classes />` 标签表示不会进行注解实体类的扫描。指定的显式 `true` 值 — `<exclude-unlisted-classes>true</exclude-unlisted-classes>` — 也意味着不进行扫描。 `<exclude-unlisted-classes> false</exclude-unlisted-classes>` 则会触发扫描；但是，如果开发者需要进行实体类扫描，建议开发者简单地省略 `<exclude-unlisted-classes>` 元素。

`LocalContainerEntityManagerFactoryBean` 是最强大的JPA设置选项，允许在应用程序中进行灵活的本地配置。它支持连接到现有的 `JDBC DataSource`，支持本地和全局事务等。但是，它对运行时环境施加了需求，其中之一就是如果持久性提供程序需要字节码转换，就需要有织入(weaving)能力的类加载器。

此选项可能与J2EE服务器的内置JPA功能冲突。在完整的J2EE环境中，请考虑从JNDI获取 `EntityManagerFactory`。或者，在开发者的 `LocalContainerEntityManagerFactoryBean` 定义中指定一个自定义 `persistenceXmlLocation`，例如 `META-INF/my-persistence.xml`，并且只在应用程序jar文件中包含有该名称的描述符。因为J2EE服务器仅查找默认的 `META-INF/persistence.xml` 文件，所以它会忽略这种自定义持久性单元，从而避免了与Spring驱动的JPA设置之间发生冲突。（例如，这适用于Resin 3.1）

何时需要加载时间织入？

并非所有JPA提供商都需要JVM代理。Hibernate就是一个不需要JVM代理的例子。如果开发者的提供商不需要代理或开发者有其他替代方案，例如通过定制编译器或 `Ant` 任务在构建时应用增强功能，则不用使用加载时间编织器。

`LoadTimeWeaver` 是一个Spring提供的接口，它允许以特定方式插入JPA `ClassTransformer` 实例，这取决于环境是Web容器还是应用程序服务器。通过代理挂载 `ClassTransformers` 通常性能较差。代理会对整个虚拟机进行操作，并检查加载的每个类，这是生产服务器环境中最不需要的额外负载。

Spring为各种环境提供了一些 `LoadTimeWeaver` 实现，允许 `ClassTransformer` 实例仅适用于每个类加载器，而不是每个VM。

有关 `LoadTimeWeaver` 的实现及其设置的通用或定制的各种平台（如Tomcat，WebLogic，GlassFish，Resin和JBoss）的更多了解，请参阅AOP章节中的[Spring配置](#)一节。

如前面部分所述，开发者可以使用 `@EnableLoadTimeWeaving` 注解或者 `context:load-time-weaver` XML元素来配置上下文范围的 `LoadTimeWeaver`。所有 JPA `LocalContainerEntityManagerFactoryBeans` 都会自动拾取这样的全局织入器。这是设置加载时间织入器的首选方式，为平台（WebLogic，GlassFish，Tomcat，Resin，JBoss或VM代理）提供自动检测功能，并将织入组件自动传播到所有可以感知织入者的Bean：

```
<context:load-time-weaver/>
<bean id="emf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    ...
</bean>
```

开发者也可以通过 `LocalContainerEntityManagerFactoryBean` 的 `loadTimeWeaver` 属性来手动指定专用的织入器：

```
<bean id="emf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="loadTimeWeaver">
        <bean class="org.springframework.instrument.classloading.ReflectiveLoadTimeWeaver"/>
    </property>
</bean>
```

无论LTW如何配置，使用这种技术，依赖于仪器的JPA应用程序都可以在目标平台（例如：Tomcat）中运行，而不需要代理。这尤其重要的是当主机应用程序依赖于不同的JPA实现时，因为JPA转换器仅应用于类加载器级，彼此隔离。

## 处理多个持久化单元

例如，对于依赖存储在类路径中的各种JARS中的多个持久性单元位置的应用程序，Spring 将 `PersistenceUnitManager` 作为中央仓库来避免可能昂贵的持久性单元发现过程。默认实现允许指定多个位置，这些位置将通过持久性单元名称进行解析并稍后检索。（默认情况下，搜索classpath下的 `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>

```

在默认实现传递给JPA provider之前，是允许通过属性（影响全部持久化单元）或者通过 PersistenceUnitPostProcessor 以编程(对选择的持久化单元进行)进行对 PersistenceUnitInfo 进行自定义的。如果没有指定 PersistenceUnitManager，则由 LocalContainerEntityManagerFactoryBean 在内部创建和使用。

## 16.4.2 基于JPA的EntityManagerFactory和EntityManager来实现DAO



虽然 EntityManagerFactory 实例是线程安全的，但 EntityManager 实例不是。注入的 JPA EntityManager 的行为类似于从 JPA Spec 中定义的应用程序服务器的 JNDI 环境中提取的 EntityManager。它将所有调用委托给当前事务的 EntityManager (如果有);否则，它每个操作返回的都是新创建的 EntityManager，通过使用不同的 EntityManager 来保证使用时的线程安全。

通过注入的方式使用 EntityManagerFactory 或 EntityManager 来编写 JPA 代码，是不需要依赖任何 Spring 定义的类的。如果启用了 PersistenceAnnotationBeanPostProcessor，Spring 可以在实例级别和方法级别识别 @PersistenceUnit 和 @PersistenceContext 注解。使用 @PersistenceUnit 注解的纯 JPA DAO 实现可能如下所示：

```
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 DAO 对 Spring 的实现是没有任何依赖的，而且很适合与 Spring 的应用程序上下文进行集成。而且，DAO 还可以通过注解来注入默认的 EntityManagerFactory：

```
<beans>

    <!-- bean post-processor for JPA annotations -->
    <bean class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor"/>

    <bean id="myProductDao" class="product.ProductDaoImpl"/>

</beans>
```

如果不想明确定义 `PersistenceAnnotationBeanPostProcessor`，可以考虑在应用程序上下文配置中使用 `Spring` 上下文 `annotation-config` XML 元素。这样做会自动注册所有 `Spring` 标准后置处理器，用于初始化基于注解的配置，包括 `CommonAnnotationBeanPostProcessor` 等。

```
<beans>

    <!-- post-processors for all standard config annotations -->
    <context:annotation-config/>

    <bean id="myProductDao" class="product.ProductDaoImpl"/>

</beans>
```

这样的 DAO 的主要问题是它总是通过工厂创建一个新的 `EntityManager`。开发者可以通过请求事务性 `EntityManager`（也称为共享 `EntityManager`，因为它是实际的事务性 `EntityManager` 的一个共享的，线程安全的代理）来避免这种情况。

```
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();
    }
}
```

`@PersistenceContext` 注解具有可选的属性类型，默认值为 `PersistenceContextType.TRANSACTION`。此默认值是开发者所需要接收共享的 `EntityManager` 代理。替代方案 `PersistenceContextType.EXTENDED` 则完全不同：该方案会返

回一个所谓扩展的 `EntityManager`，该 `EntityManager` 不是线程安全的，因此不能在并发访问的组件（如 Spring 管理的单例 Bean）中使用。扩展实体管理器仅应用于状态组件中，比如持有会话的组件，其中 `EntityManager` 的生命周期与当前事务无关，而是完全取决于应用程序。

#### 方法和实例变量级别注入

指示依赖注入（例如 `@PersistenceUnit` 和 `@PersistenceContext`）的注解可以应用于类中的实例变量或方法，也就是表达式方法级注入和实例变量级注入。实例变量级注释简洁易用，而方法级别允许进一步处理注入的依赖关系。在这两种情况下，成员的可见性（`public`，`protected`，`private`）并不重要。

#### 类级注解怎么办？

在 J2EE 平台上，它们用于依赖关系声明，而不是资源注入。

注入的 `EntityManager` 是由 Spring 管理的（Spring 可以意识到正在进行的事务）。重要的是要注意，因为通过注解进行注入，即使新的 DAO 实现使用通过方法注入的 `EntityManager` 而不是 `EntityManagerFactory` 的注入的，在应用程序上下文 XML 中不需要进行任何修改。

这种 DAO 风格的主要优点是它只依赖于 Java Persistence API；不需要导入任何 Spring 的实现类。而且，Spring 容器可以识别 JPA 注解来实现自动的注入和管理。从非侵入的角度来看，这种风格对 JPA 开发者来说可能更为自然。

### 16.4.3 Spring驱动的JPA事务



如果开发者还没有阅读[Section 13.5, “声明式事务管理”](#)，强烈建议开发者先行阅读，这样可以更详细地了解Spring的对声明式事务支持。

JPA的推荐策略是通过JPA的本地事务支持的本地事务。Spring的 `JpaTransactionManager` 提供了许多来自本地JDBC事务的功能，例如针对任何常规JDBC连接池（不需要XA要求）指定事务的隔离级别和资源级只读优化等。

Spring JPA还允许配置 `JpaTransactionManager` 将JPA事务暴露给访问同一 `DataSource` 的 JDBC访问代码，前提是注册的 `JpaDialect` 支持检索底层 JDBC连接。Spring为EclipseLink和Hibernate JPA实现提供了实现。有关 `JpaDialect` 机制的详细信息，请参阅下一节。

## 16.4.4 JpaDialect 和 JpaVendorAdapter

作为高级功能，`JpaTransactionManager` 和 `AbstractEntityManagerFactoryBean` 的子类支持自定义 `JpaDialect`，将其作为 Bean 传递给 `jpaDialect` 属性。`JpaDialect` 实现可以以供应商特定的方式使能 Spring 支持的一些高级功能：

- 应用特定的事务语义，如自定义隔离级别或事务超时
- 为基于 JDBC 的 DAO 导出事务性 JDBC 连接
- 从 `PersistenceExceptions` 到 `Spring DataAccessExceptions` 的异常转义

这对于特殊的事务语义和异常的高级翻译特别有价值。但是 Spring 使用的默认实现 (`DefaultJpaDialect`) 是不提供任何特殊功能的。如果需要上述功能，则必须指定适当的方言才可以。

作为一个更广泛的供应商适应设施，主要用于 Spring 的全功能 `LocalContainerEntityManagerFactoryBean` 设置，`JpaVendorAdapter` 将 `JpaDialect` 的功能与其他提供者特定的默认设置相结合。指定 `HibernateJpaVendorAdapter` 或 `EclipseLinkJpaVendorAdapter` 是分别为 `Hibernate` 或 `EclipseLink` 自动配置 `EntityManagerFactory` 设置的最简单方便的方法。但是请注意，这些提供程序适配器主要是为了与 Spring 驱动的事务管理一起使用而设计的，即为了与 `JpaTransactionManager` 配合使用的。

有关其操作的更多详细信息以及在 Spring 的 JPA 支持中如何使用，请参阅 `JpaDialect` 和 `JpaVendorAdapter` 的 Javadoc。

## 16.4.5 为JPA配置JTA事务管理

作为 `JpaTransactionManager` 的替代方案，Spring还允许通过JTA在J2EE环境中或与独立的事务协调器（如Atomikos）进行多资源事务协调。除了用Spring的 `JtaTransactionManager` 替换 `JpaTransactionManager`，还有需要以下一些操作：

- 底层 JDBC连接池是需要具备XA功能，并与开发者的事务协调器集成的。这在J2EE环境中很简单，只需通过JNDI导出不同类型的 `DataSource` 即可。有关导出 `DataSource` 等详细信息，可以参考应用服务器文档。类似地，独立的事务协调器通常带有特殊的XA集成的 `DataSource` 实现。
- 需要为JTA配置JPA `EntityManagerFactory`。这是特定于提供程序的，通常通过在 `LocalContainerEntityManagerFactoryBean` 的特殊属性指定为“`jpaProperties`”。在使用 `Hibernate`的情况下，这些属性甚至是需要基于特定的版本的；请查阅 `Hibernate` 文档以获取详细信息。
- Spring的 `HibernateJpaVendorAdapter` 会强制执行某些面向Spring的默认设置，例如在 `Hibernate 5.0` 中匹配 `Hibernate` 自己的默认值的连接释放模式“`on-close`”，但在 `5.1 / 5.2` 中不再存在。对于JTA设置，不要声明 `HibernateJpaVendorAdapter` 开始，或关闭其 `prepareConnection` 标志。或者，将 `Hibernate 5.2` 的 `hibernate.connection.handling_mode` 属性设置为 `DELAYED_ACQUISITION_AND_RELEASE_AFTER_STATEMENT` 以恢复 `Hibernate` 自己的默认值。有关 `WebLogic` 的相关说明，请参考 [Section 16.3.7, “Hibernate的虚假应用服务器警告”](#) 一节。
- 或者，可以考虑从应用程序服务器本身获取 `EntityManagerFactory`，即通过JNDI查找而不是本地声明的 `LocalContainerEntityManagerFactoryBean`。服务器提供的 `EntityManagerFactory` 可能需要在服务器配置中进行特殊定义，减少了部署的移植性，但是 `EntityManagerFactory` 将为开箱即用的服务器JTA环境设置。



## 17.1 Introduction

In this chapter, we will describe Spring's Object/XML Mapping support. Object/XML Mapping, or O/X mapping for short, is the act of converting an XML document to and from an object. This conversion process is also known as XML Marshalling, or XML Serialization. This chapter uses these terms interchangeably.

Within the field of O/X mapping, a\_marshallers is responsible for serializing an object (graph) to XML. In similar fashion, an\_unmarshallers\_deserializes the XML to an object graph. This XML can take the form of a DOM document, an input or output stream, or a SAX handler.

Some of the benefits of using Spring for your O/X mapping needs are:

### 17.1.1 Ease of configuration

Spring's bean factory makes it easy to configure marshallers, without needing to construct JAXB context, JiBX binding factories, etc. The marshallers can be configured as any other bean in your application context. Additionally, XML Schema-based configuration is available for a number of marshallers, making the configuration even simpler.

## 17.1.2 Consistent Interfaces

Spring's O/X mapping operates through two global interfaces: the `Marshaller` and `Unmarshaller` interface. These abstractions allow you to switch O/X mapping frameworks with relative ease, with little or no changes required on the classes that do the marshalling. This approach has the additional benefit of making it possible to do XML marshalling with a mix-and-match approach (e.g. some marshalling performed using JAXB, other using Castor) in a non-intrusive fashion, leveraging the strength of each technology.

### 17.1.3 Consistent Exception Hierarchy

Spring provides a conversion from exceptions from the underlying O/X mapping tool to its own exception hierarchy with the `XmlMappingException` as the root exception. As can be expected, these runtime exceptions wrap the original exception so no information is lost.

## 17.2 Marshaller and Unmarshaller

As stated in the introduction, `a_marshaller_serializes` an object to XML, and `an_unmarshaller_deserializes` XML stream to an object. In this section, we will describe the two Spring interfaces used for this purpose.

## 17.2.1 Marshaller

Spring abstracts all marshalling operations behind the `org.springframework.oxm.Marshaller` interface, the main method of which is shown below.

```
public interface Marshaller {

    /**
     * Marshal the object graph with the given root into the provided Result.
     */
    void marshal(Object graph, Result result) throws XmlMappingException, IOException;
}
```

The `Marshaller` interface has one main method, which marshals the given object to a given `javax.xml.transform.Result`. `Result` is a tagging interface that basically represents an XML output abstraction: concrete implementations wrap various XML representations, as indicated in the table below.

Result implementation	Wraps XML representation
<code>DOMResult</code>	<code>org.w3c.dom.Node</code>
<code>SAXResult</code>	<code>org.xml.sax.ContentHandler</code>
<code>StreamResult</code>	<code>java.io.File</code> , <code>java.io.OutputStream</code> , OR <code>java.io.Writer</code>



Although the `marshal()` method accepts a plain object as its first parameter, most `Marshaller` implementations cannot handle arbitrary objects. Instead, an object class must be mapped in a mapping file, marked with an annotation, registered with the marshaller, or have a common base class. Refer to the further sections in this chapter to determine how your O/X technology of choice manages this.

Similar to the `Marshaller`, there is the `org.springframework.oxm.Unmarshaller` interface.

```
public interface Unmarshaller {

    /**
     * Unmarshal the given provided Source into an object graph.
     */
    Object unmarshal(Source source) throws XmlMappingException, IOException;
}
```

This interface also has one method, which reads from the given `javax.xml.transform.Source` (an XML input abstraction), and returns the object read. As with `Result`, `Source` is a tagging interface that has three concrete implementations. Each wraps a different XML representation, as indicated in the table below.

<b>Source implementation</b>	<b>Wraps XML representation</b>
DOMSource	<code>org.w3c.dom.Node</code>
SAXSource	<code>org.xml.sax.InputSource</code> , and <code>org.xml.sax.XMLReader</code>
StreamSource	<code>java.io.File</code> , <code>java.io.InputStream</code> , Or <code>java.io.Reader</code>

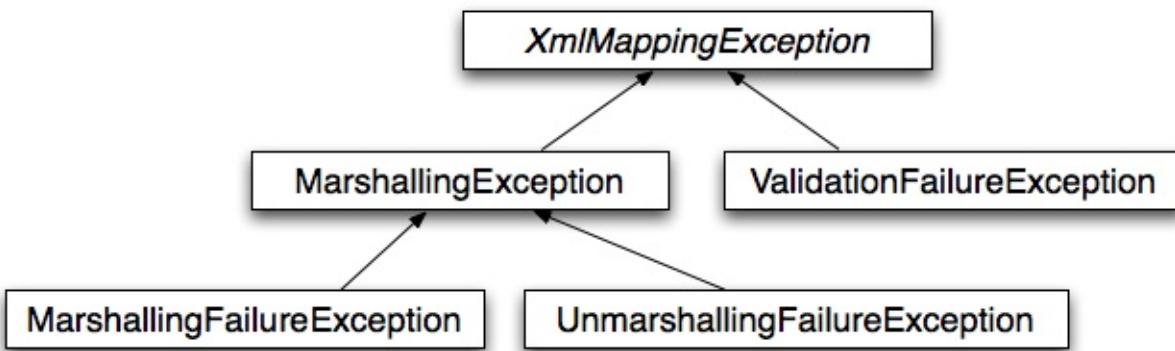
Even though there are two separate marshalling interfaces (`Marshaller` and `Unmarshaller`), all implementations found in Spring-WS implement both in one class. This means that you can wire up one marshaller class and refer to it both as a marshaller and an unmarshaller in your `applicationContext.xml`.

### 17.2.3 XmlMappingException

Spring converts exceptions from the underlying O/X mapping tool to its own exception hierarchy with the `XmlMappingException` as the root exception. As can be expected, these runtime exceptions wrap the original exception so no information will be lost.

Additionally, the `MarshallingFailureException` and `UnmarshallingFailureException` provide a distinction between marshalling and unmarshalling operations, even though the underlying O/X mapping tool does not do so.

The O/X Mapping exception hierarchy is shown in the following figure:



O/X Mapping exception hierarchy

## 17.3Using Marshaller and Unmarshaller

Spring's OXM can be used for a wide variety of situations. In the following example, we will use it to marshal the settings of a Spring-managed application as an XML file. We will use a simple JavaBean to represent the settings:

```
public class Settings {

    private boolean fooEnabled;

    public boolean isFooEnabled() {
        return fooEnabled;
    }

    public void setFooEnabled(boolean fooEnabled) {
        this.fooEnabled = fooEnabled;
    }
}
```

The application class uses this bean to store its settings. Besides a main method, the class has two methods: `saveSettings()` saves the settings bean to a file named `settings.xml` , and `loadSettings()` loads these settings again. A `main()` method constructs a Spring application context, and calls these two methods.

```
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();
}
}

```

The `Application` requires both a `marshaller` and `unmarshaller` property to be set. We can do so using the following `applicationContext.xml` :

```

<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>

```

This application context uses Castor, but we could have used any of the other marshaller instances described later in this chapter. Note that Castor does not require any further configuration by default, so the bean definition is rather simple. Also note that the `CastorMarshaller` implements both `Marshaller` and `Unmarshaller`, so we can refer to the `castorMarshaller` bean in both the `marshaller` and `unmarshaller` property of the application.

This sample application produces the following `settings.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<settings foo-enabled="false"/>
```

## 17.4 XML Schema-based Configuration

Marshaller could be configured more concisely using tags from the OXM namespace. To make these tags available, the appropriate schema has to be referenced first in the preamble of the XML configuration file. Note the 'oxm' related text below:

```
<?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">
```

Currently, the following tags are available:

- [jaxb2-marshaller](#)
- [jibx-marshaller](#)
- [castor-marshaller](#)

Each tag will be explained in its respective marshaller's section. As an example though, here is how the configuration of a JAXB2 marshaller might look like:

```
<oxm:jaxb2-marshaller id="marshaller" contextPath="org.springframework.ws.samples.airline.schema"/>
```

## 17.5 JAXB

The JAXB binding compiler translates a W3C XML Schema into one or more Java classes, a `jaxb.properties` file, and possibly some resource files. JAXB also offers a way to generate a schema from annotated Java classes.

Spring supports the JAXB 2.0 API as XML marshalling strategies, following the `Marshaller` and `Unmarshaller` interfaces described in [Section 17.2, “Marshaller and Unmarshaller”](#). The corresponding integration classes reside in the `org.springframework.oxm.jaxb` package.

## 17.5.1 Jaxb2Marshaller

The `Jaxb2Marshaller` class implements both the `Spring Marshaller` and `Unmarshaller` interface. It requires a context path to operate, which you can set using the `contextPath` property. The context path is a list of colon (:) separated Java package names that contain schema derived classes. It also offers a `classesToBeBound` property, which allows you to set an array of classes to be supported by the marshaller. Schema validation is performed by specifying one or more schema resource to the bean, like so:

```
<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 Schema-based Configuration

The `jaxb2-marshaller` tag configures a `org.springframework.oxm.jaxb.Jaxb2Marshaller`. Here is an example:

```
<oxm:jaxb2-marshaller id="marshaller" contextPath="org.springframework.ws.samples.airline.schema"/>
```

Alternatively, the list of classes to bind can be provided to the marshaller via the `class-to-be-bound` child tag:

```
<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>
```

Available attributes are:

Attribute	Description	Required
id	the id of the marshaller	no
contextPath	the JAXB Context path	no

## 17.6 Castor

Castor XML mapping is an open source XML binding framework. It allows you to transform the data contained in a java object model into/from an XML document. By default, it does not require any further configuration, though a mapping file can be used to have more control over the behavior of Castor.

For more information on Castor, refer to the [Castor web site](#). The Spring integration classes reside in the `org.springframework.oxm.castor` package.

## 17.6.1 CastorMarshaller

As with JAXB, the `CastorMarshaller` implements both the `Marshaller` and `Unmarshaller` interface. It can be wired up as follows:

```
<beans>
    <bean id="castorMarshaller" class="org.springframework.oxm.castor.CastorMarshaller"
" />
    ...
</beans>
```

## 17.6.2 Mapping

Although it is possible to rely on Castor's default marshalling behavior, it might be necessary to have more control over it. This can be accomplished using a Castor mapping file. For more information, refer to [Castor XML Mapping](#).

The mapping can be set using the `mappingLocation` resource property, indicated below with a classpath resource.

```
<beans>
    <bean id="castorMarshaller" class="org.springframework.oxm.castor.CastorMarshaller"
    >
        <property name="mappingLocation" value="classpath:mapping.xml" />
    </bean>
</beans>
```

## XML Schema-based Configuration

The `castor-marshaller` tag configures a `org.springframework.oxm.castor.CastorMarshaller`. Here is an example:

```
<oxm:castor-marshaller id="marshaller"
    mapping-location="classpath:org/springframework/oxm/castor/mapping.xml"/>
```

The marshaller instance can be configured in two ways, by specifying either the location of a mapping file (through the `mapping-location` property), or by identifying Java POJOs (through the `target-class` or `target-package` properties) for which there exist corresponding XML descriptor classes. The latter way is usually used in conjunction with XML code generation from XML schemas.

Available attributes are:

<b>Attribute</b>	<b>Description</b>	<b>Required</b>
<code>id</code>	the id of the marshaller	no
<code>encoding</code>	the encoding to use for unmarshalling from XML	no
<code>target-class</code>	a Java class name for a POJO for which an XML class descriptor is available (as generated through code generation)	no
<code>target-package</code>	a Java package name that identifies a package that contains POJOs and their corresponding Castor XML descriptor classes (as generated through code generation from XML schemas)	no
<code>mapping-location</code>	location of a Castor XML mapping file	

## 17.7 JiBX

The JiBX framework offers a solution similar to that which Hibernate provides for ORM: a binding definition defines the rules for how your Java objects are converted to or from XML. After preparing the binding and compiling the classes, a JiBX binding compiler enhances the class files, and adds code to handle converting instances of the classes from or to XML.

For more information on JiBX, refer to the [JiBX web site](#). The Spring integration classes reside in the `org.springframework.oxm.jibx` package.

## 17.7.1 JibxMarshaller

The `JibxMarshaller` class implements both the `Marshaller` and `Unmarshaller` interface. To operate, it requires the name of the class to marshal in, which you can set using the `targetClass` property. Optionally, you can set the binding name using the `bindingName` property. In the next sample, we bind the `Flights` class:

```
<beans>
    <bean id="jibxFlightsMarshaller" class="org.springframework.oxm.jibx.JibxMarshaller">
        <property name="targetClass">org.springframework.oxm.jibx.Flights</property>
    </bean>
    ...
</beans>
```

A `JibxMarshaller` is configured for a single class. If you want to marshal multiple classes, you have to configure multiple `JibxMarshaller`s with different `targetClass` property values.

## XML Schema-based Configuration

The `jibx-marshaller` tag configures a `org.springframework.oxm.jibx.JibxMarshaller`. Here is an example:

```
<oxm:jibx-marshaller id="marshaller" target-class="org.springframework.ws.samples.airline.schema.Flight"/>
```

Available attributes are:

Attribute	Description	Required
<code>id</code>	the id of the marshaller	no
<code>target-class</code>	the target class for this marshaller	yes
<code>bindingName</code>	the binding name used by this marshaller	no

## 17.8 XStream

XStream is a simple library to serialize objects to XML and back again. It does not require any mapping, and generates clean XML.

For more information on XStream, refer to the [XStream web site](#). The Spring integration classes reside in the `org.springframework.oxm.xstream` package.

## 17.8.1 XStreamMarshaller

The `xstreamMarshaller` does not require any configuration, and can be configured in an application context directly. To further customize the XML, you can set `alias map`, which consists of string aliases mapped to classes:

```
<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>
```



By default, XStream allows for arbitrary classes to be unmarshalled, which can result in security vulnerabilities. As such, it is *not recommended to use the XStreamMarshaller to unmarshal XML from external sources*(i.e. the Web), as this can result in *security vulnerabilities*. If you do use the `xstreamMarshaller` to unmarshal XML from an external source, set the `supportedClasses` property on the `xstreamMarshaller`, like so: ... This will make sure that only the registered classes are eligible for unmarshalling. Additionally, you can register [custom converters](#) to make sure that only your supported classes can be unmarshalled. You might want to add a `catchAllConverter` as the last converter in the list, in addition to converters that explicitly support the domain classes that should be supported. As a result, default XStream converters with lower priorities and possible security vulnerabilities do not get invoked.



Note that XStream is an XML serialization library, not a data binding library. Therefore, it has limited namespace support. As such, it is rather unsuitable for usage within Web services.

# PartV.The Web

This part of the reference documentation covers Spring Framework's support for the presentation tier (and specifically web-based presentation tiers) including support for WebSocket-style messaging in web applications.

Spring Framework's own web framework, [Spring Web MVC](#), is covered in the first couple of chapters. Subsequent chapters are concerned with Spring Framework's integration with other web technologies, such as [JSF](#).

The section then concludes with comprehensive coverage of the Spring Framework [Chapter22, WebSocket Support](#) (including [Section22.4, “STOMP Over WebSocket Messaging Architecture”](#)).

- [Chapter18, Web MVC framework](#)
- [Chapter19, View technologies](#)
- [Chapter21, Integrating with other web frameworks](#)
- [Chapter22, WebSocket Support](#)



## 18.1 Spring Web MVC框架的介绍

Spring Web模型视图控制器（MVC）框架是围绕一个 `DispatcherServlet` 设计的，它将请求分派给处理器，具有可配置的处理器映射，视图解析，区域设置，本地化和主题解析，并且支持上传文件。默认的处理是基于注解 `@Controller` 和 `@RequestMapping`，提供一系列灵活的处理方法。随着 Spring 3.0 的推出，通过 `@PathVariable` 或者其他注解，`@Controller` 机制开始允许你去创建 Rest风格的 web 站点和应用。

在 Spring Web MVC 和 Spring 中一条关键的准则是“对扩展开放，对修改关闭”

在 Spring Web MVC 中一些核心类的方法被标注为 `final`，由于开发者不能用自己的方法去覆盖这些方法，这并不是任意的，而是特别考虑到这个原则。

对于这个准则的解释，请参考 Seth Ladd 的 *Expert Spring Web MVC and Web Flow*；具体参见第一版第 117 页的“A Look At Design”一节。或者参见

- [Bob Martin, The Open-Closed Principle \(PDF\)](#)

当你使用 Spring MVC 时，你不能在 `final` 方法增加切面。例如，你不能

在 `AbstractController.setSynchronizeOnSession()` 增加切面，有关 AOP 代理的更多信息以及为什么不能再 `Final` 方法增加切面，查看第 7.6.1 节“[了解 AOP 代理](#)”。

在 Spring Web MVC 中，您可以使用任何对象作为命令或表单支持对象；您不需要实现一个特别架构接口或者基类。Spring 数据绑定非常灵活：例如，你可以使用程序将类型不匹配当作验证错误而不是系统错误。因此，您不需要将您的业务对象的属性复制为简单的无格式的字符串，仅用于处理无效提交，或者正确转换字符串。相反，通常最好直接绑定到您的业务对象。

Spring 的视图处理也是相当灵活，控制器通常负责准备具有数据和选择视图名称的模型映射，但它也可以直接写入响应流并完成请求。视图名称解析可通过文件扩展或 `Accept` 标头内容类型协商进行高度配置，通过 `bean` 名称，属性文件或甚至自定义的 `ViewResolver` 实现。模型（MVC 中的 M）是一个 `Map` 接口，可以完全提取视图技术，你可以直接与基于模板的渲染技术（如 JSP 和 FreeMarker）集成，或直接生成 XML，JSON，Atom 和许多其他类型的内容。模型 `Map` 可以简单地转换成适当的格式，如 JSP 请求属性或 FreeMarker 模板模型。

## 18.1.1 Spring Web MVC的特点

### Spring Web 流程

Spring Web 流程 (SWF) 的目的是成为最好的Web页面应用流程管理方案，SWF与Servlet 和 Portlet 环境中的Spring MVC和JSF等现有框架集成。如果你有一个这样的业务流程，使用会话模型比纯粹的请求要优，那么SWF可能是一个选择。

SWF允许您将逻辑页面流作为在不同情况下可重用的自包含模块捕获，因此非常适合构建引导用户通过驱动业务流程的受控导航的Web应用程序模块。

更多关于SWF的信息，请点击[Spring Web Flow website](#).

Spring 的Web模块包含许多独特的web支持特性：

- 明确并分离的角色.每个角色-控制器，验证器，命令对象，构建对象，模型对象，分发器，映射处理器，视图解析等等都是完全的一个特定对象
- 框架和应用程序类作为*JavaBeans*的强大而直接的配置。此配置功能包括跨上下文的简单引用，例如从Web控制器到业务对象和验证器。
- 可适配，无入侵，灵活，定义您需要的任何控制器方法签名，可能使用给定方案的参数注释之一（例如 `@RequestParam` , `@RequestHeader` , `@PathVariable` 等）。
- 可重用的业务代码，不需要重复，使用现有的业务对象作为命令或表单对象，而不是仿照它们来扩展特定的框架基类。
- 自定义绑定和验证，类型不匹配作为应用程序级验证错误，保持违规值，本地化日期和数字绑定等，而不是只使用仅包含字符串的表单对象进行手动解析和转换为业务对象。
- 自定义的处理程序映射和视图解析，从简单的URL配置策略到复杂的，特制的策略，Spring比Web MVC框架更灵活，这些框架需要特定的技术。
- 灵活的模型转换，具有名称/值的模型传输Map支持与任何视图技术的轻松集成。
- 本地，时区，主题自定义，支持具有或不具有Spring标签库的JSP，支持JSTL，支持FreeMarker而不需要额外的网桥等等。
- 一个简单而强大的JSP标签库，被称为*Spring*标签库，为数据绑定和主题等功能提供支持。自定义标签允许在标记代码方面具有最大的灵活性。有关标签库描述符的信息，请参见附录[Chapter 40,spring JSP Tag Library](#)
- 在Spring 2.0中引入的JSP表单标签库，使得在JSP页面中的写入表单更容易。有关标签库描述符的信息，请参见附录[Chapter 41,spring-form JSP Tag Library](#)

- Bean的生命周期范围限定在当前的HTTP请求或HTTP Session中。这不是Spring MVC本身的一个特定功能，而是Spring MVC使用的WebApplicationContext容器。这些bean范围在[Section 3.5.4, “Request, session, application, and WebSocket scopes”](#)

## 18.1.2 其他MVC实现的可插拔性

对于某些项目，非Spring MVC实现更为可取。许多团队希望利用他们现有的技能和工具投资，例如使用JSF。

如果您不想使用Spring的Web MVC，但打算利用Spring提供的其他解决方案，您可以轻松地将您选择的Web MVC框架与Spring集成。通过其 `ContextLoaderListener` 简单地启动一个 Spring根应用程序上下文，并通过任何动作对象中的 `servletContext` 属性（或Spring各自的帮助方法）访问它。没有涉及“插件”，因此不需要专门的集成。从Web层的角度来看，您只需使用Spring作为库，将根应用程序上下文实例作为入口点。

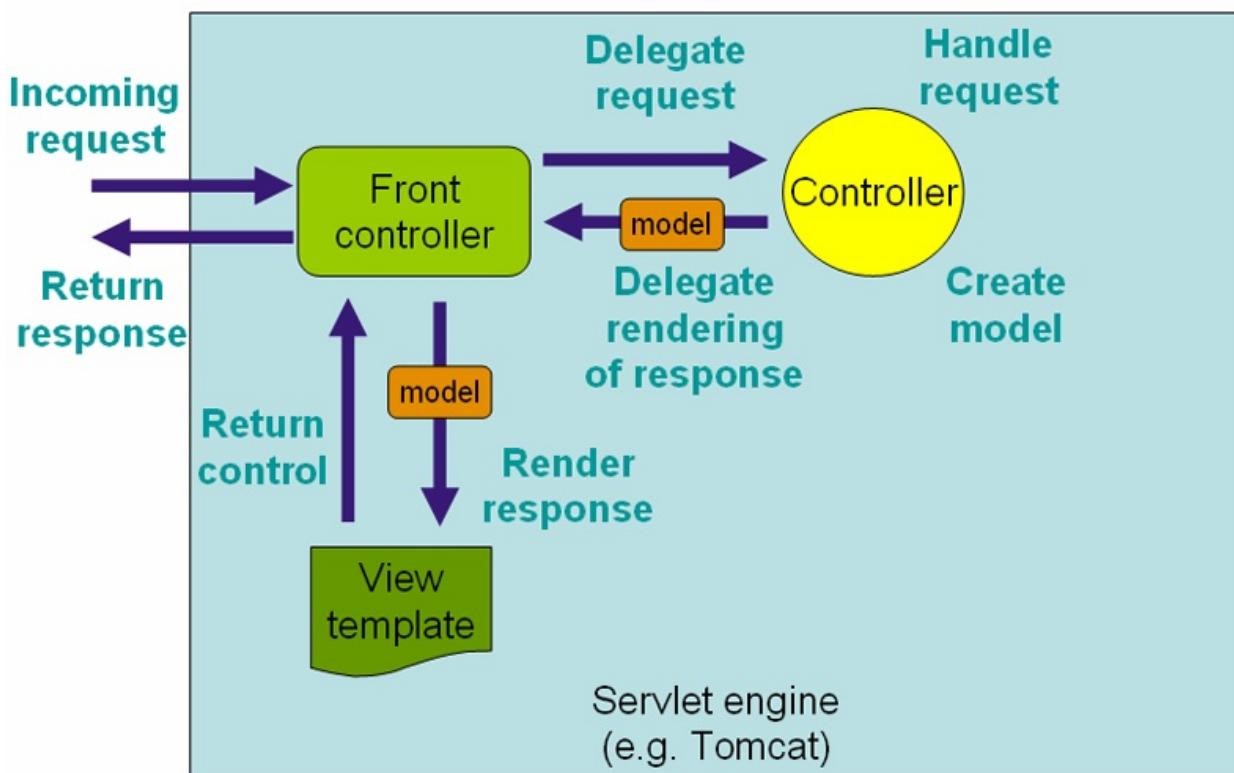
即使没有Spring的Web MVC，您的注册bean和Spring的服务也可以在您的指尖。在这种情况下，Spring不会与其他Web框架竞争。它简单地解决了纯Web MVC框架从bean配置到数据访问和事务处理的许多方面。所以您可以使用Spring中间层和/或数据访问层来丰富您的应用程序，即使您只想使用JDBC或Hibernate的事务抽象。

## 18.2 DispatcherServlet

Spring的Web MVC框架与许多其他Web MVC框架一样，以请求为驱动，围绕一个中央Servlet设计，将请求发送给控制器，并提供了其他促进Web应用程序开发的功能。然而，Spring的DispatcherServlet做得更多。它和Spring IoC容器整合一起，它允许你使用Spring每个特性。

Spring Web MVC DispatcherServlet的请求处理工作流程如下图所示。对设计模式熟悉的读者将会认识到，DispatcherServlet是“前端控制器”设计模式的表达（这是Spring Web MVC与许多其他领先的Web框架共享的模式）。

Figure 18.1. 在Spring Web MVC中请求处理流程(hight level)



DispatcherServlet是一个实际的Servlet（它继承自HttpServlet基类），因此在Web应用程序中被声明。您需要使用URL映射来映射要DispatcherServlet处理的请求。以下是Servlet 3.0+环境中的标准Java EE Servlet配置：

```

public class MyWebApplicationInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext container) {
        ServletRegistration.Dynamic registration = container.addServlet("example", new
DispatcherServlet());
        registration.setLoadOnStartup(1);
        registration.addMapping("/example/*");
    }

}

```

在前面的示例中，以 `/example` 开头的所有请求都将由名为 `Example` 的 `DispatcherServlet` 实例处理。

`WebApplicationInitializer` 是由 Spring MVC 提供的接口，可确保您的基于代码的配置被检测并自动用于初始化任何 `Servlet 3` 容器。这个名为 `AbstractAnnotationConfigDispatcherServletInitializer` 的接口的抽象基类实现通过简单地指定其 `servlet` 映射和列出配置类来更容易地注册 `DispatcherServlet`，甚至建议您设置 Spring MVC 应用程序。有关更多详细信息，请参阅 [基于代码的 Servlet 容器初始化](#)。

`DispatcherServlet` 是一个实际的 `Servlet`（它继承自 `HttpServlet` 基类），因此在 Web 应用程序的 `web.xml` 中声明。您需要通过使用同一 `web.xml` 文件中的 URL 映射来映射要 `DispatcherServlet` 处理的请求。这是标准的 Java EE Servlet 配置。

以下示例 `web.xml` 显示了这样的 `DispatcherServlet` 声明和映射：

```

<web-app>
    <servlet>
        <servlet-name>example</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-clas
s>
        <load-on-startup>1</load-on-startup>
    </servlet>

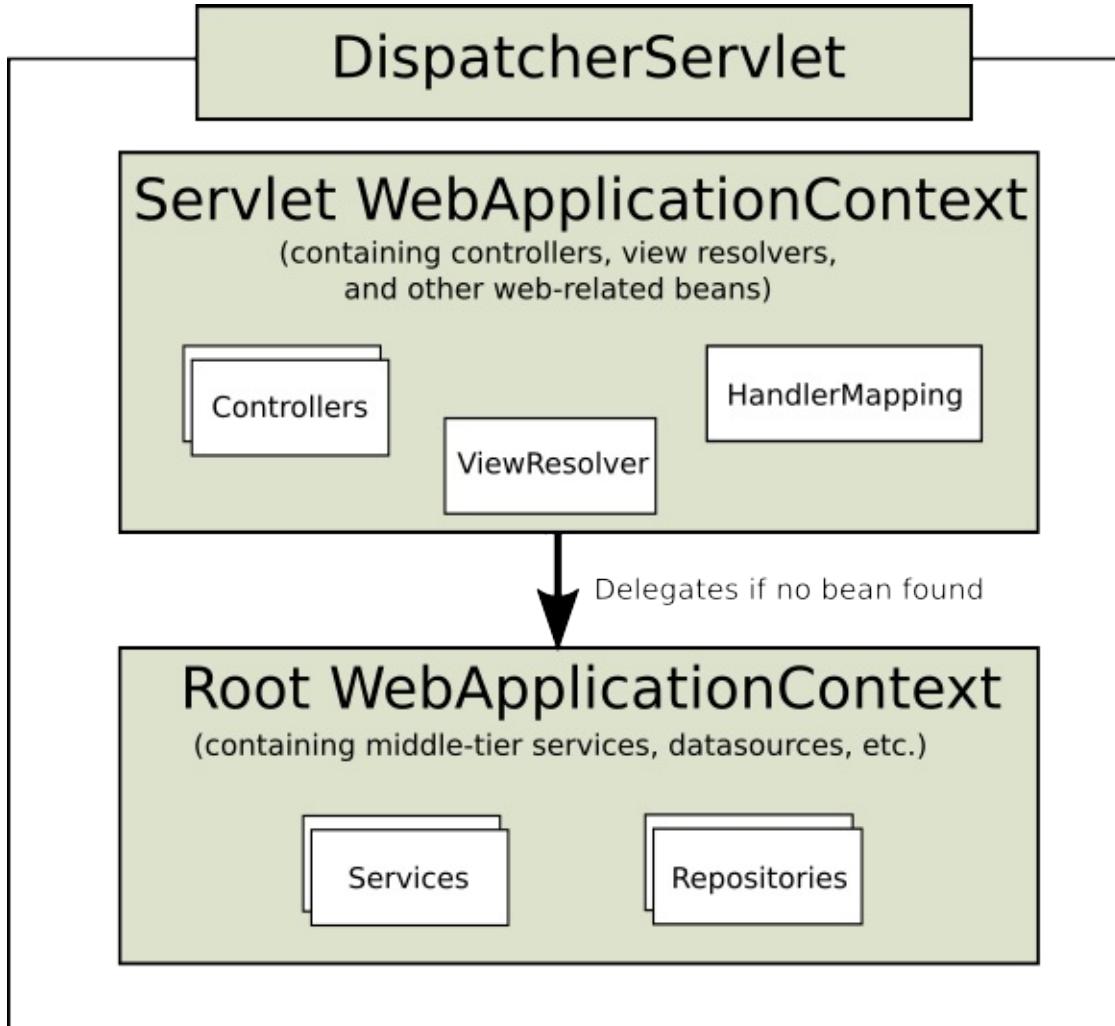
    <servlet-mapping>
        <servlet-name>example</servlet-name>
        <url-pattern>/example/*</url-pattern>
    </servlet-mapping>
</web-app>

```

如 [第 3.15 节“ApplicationContext 的附加功能”](#) 中所述，Spring 中的 `ApplicationContext` 实例可以被限定。在 Web MVC 框架中，每个 `DispatcherServlet` 都有自己的 `WebApplicationContext`，它继承了已经在根 `WebApplicationContext` 中定义的所有 `bean`。根 `WebApplicationContext` 应

该包含应该在其他上下文和Servlet实例之间共享的所有基础架构bean。这些继承的bean可以在特定于servlet的范围内被覆盖，您可以在给定的Servlet实例本地定义新的范围特定的bean。

**Figure18.2.Spring Web MVC 中的典型上下文层次结构**



在初始化 DispatcherServlet 时，Spring MVC将在Web应用程序的 WEB-INF 目录中查找名为 [servlet-name] -servlet.xml 的文件，并创建在那里定义的bean，覆盖使用相同名称定义的任何bean的定义在全球范围内。

请考虑以下 DispatcherServlet Servlet配置（在 web.xml 文件中）：

```

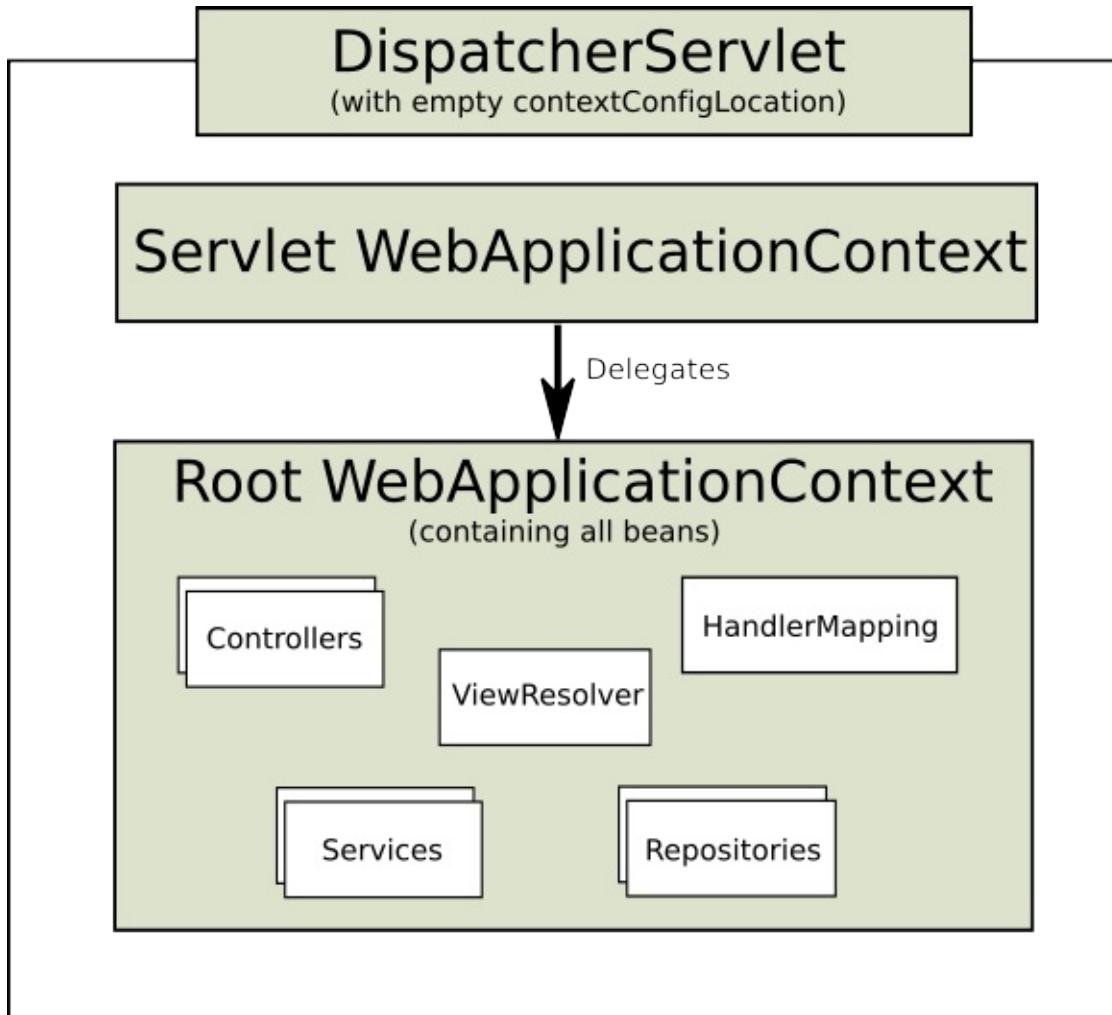
<web-app>
  <servlet>
    <servlet-name>golfing</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <!--
    <load-on-startup>1</load-on-startup>
  -->
  </servlet>
  <servlet-mapping>
    <servlet-name>golfing</servlet-name>
    <url-pattern>/golfing/*</url-pattern>
  </servlet-mapping>
</web-app>

```

使用上述Servlet配置，您将需要在应用程序中有一个名为 /WEB-INF/golfing-servlet.xml 的文件；该文件将包含您所有的Spring Web MVC特定组件（bean）。您可以通过Servlet初始化参数更改此配置文件的确切位置（有关详细信息，请参阅下文）。

单个DispatcherServlet方案也可能只有一个根上下文。

**Figure18.3.Single root context in Spring Web MVC**



这可以通过设置一个空的ContextConfigLocation servlet init参数进行配置，如下所示：

```
<web-app>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/root-context.xml</param-value>
    </context-param>
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-clas
s>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value></param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listene
r-class>
    </listener>
</web-app>
```

`WebApplicationContext` 是普通 `ApplicationContext` 的扩展，它具有 Web 应用程序所需的一些额外功能。它与正常的 `ApplicationContext` 不同之处在于它能够解析主题（参见第 18.9 节“[使用主题](#)”），并且知道它与哪个 `Servlet` 相关联（通过连接到 `ServletContext`）。

`WebApplicationContext` 绑定在 `ServletContext` 中，并且通过在 `RequestContextUtils` 类上使用静态方法，您可以随时查找 `WebApplicationContext`，如果您需要访问它。

请注意，我们可以通过基于 Java 的配置实现相同的方式：

```
public class GolfingWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        // GolfingAppConfig defines beans that would be in root-context.xml
        return new Class[] { GolfingAppConfig.class };
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        // GolfingWebConfig defines beans that would be in golfing-servlet.xml
        return new Class[] { GolfingWebConfig.class };
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/golfing/*" };
    }

}
```

## 18.2.1 WebApplicationContext 中的特殊 Bean 类型

Spring `DispatcherServlet` 使用特殊的bean来处理请求并呈现适当的视图。这些bean是 Spring MVC的一部分。您可以通过在 `WebApplicationContext` 中简单配置一个或多个选择要使用的特殊bean。但是，您最初不需要这样做，因为Spring MVC维护一个默认bean列表，如果您没有配置任何内容。更多的在下一节。首先看下表列出 `DispatcherServlet` 依赖的特殊bean类型。

**Table 18.1.** 在 `WebApplicationContext` 中的特殊bean类型

Bean type	Explanation
<code>HandlerMapping</code>	根据一些标准将传入的请求映射到处理程序和前处理程序和后处理程序列表（处理程序拦截器），其细节由 <code>HandlerMapping</code> 实现而异。最流行的实现支持注释控制器，但其他实现也存在。
<code>HandlerAdapter</code>	帮助 <code>DispatcherServlet</code> 调用映射到请求的处理程序，而不管实际调用哪个处理程序。例如，调用带注释的控制器需要解析各种注释。因此， <code>HandlerAdapter</code> 的主要目的是屏蔽 <code>DispatcherServlet</code> 和这些细节
<code>HandlerExceptionResolver</code>	映射视图的异常，也允许更复杂的异常处理代码。
<code>ViewResolver</code>	将基于逻辑字符串的视图名称解析为实际的 <code>View</code> 类型。
<code>LocaleResolver&amp;LocaleContextResolver</code>	解决客户端正在使用的区域设置以及可能的时区，以便能够提供国际化的视图
<code>ThemeResolver</code>	解决您的Web应用程序可以使用的主题，例如，提供个性化的布局
<code>MultipartResolver</code>	解析multi-part请求，以支持从HTML表单处理文件上传。
<code>FlashMapManager</code>	存储并检索可以用于将属性从一个请求传递到另一个请求的“输入”和“输出” <code>FlashMap</code> ，通常是通过重定向。

## 18.2.2 默认DispatcherServlet 配置

如上一节中针对每个特殊bean所述，`DispatcherServlet` 会维护默认使用的实现列表。此信息保存在包 `org.springframework.web.servlet` 中的文件 `DispatcherServlet.properties` 中。

所有特殊bean都有一些合理的默认值。不久之后，您将需要自定义这些bean提供的一个或多个属性。例如，将 `InternalResourceViewResolver` 设置的 `prefix` 属性配置为视图文件的父位置是很常见的。

无论细节如何，在这里了解的重要概念是，一旦您在 `WebApplicationContext` 中配置了一个特殊的bean（如 `InternalResourceViewResolver`），您可以有效地覆盖该特殊bean类型所使用的默认实现列表。例如，如果配置了 `InternalResourceViewResolver`，则会忽略 `ViewResolver` 实现的默认列表。

在[第18.16节“配置Spring MVC”](#)中，您将了解配置Spring MVC的其他选项，包括MVC Java配置和MVC XML命名空间，这两者都提供了一个简单的起点，并且对Spring MVC的工作原理几乎不了解。无论您如何选择配置应用程序，本节中介绍的概念都是基础的，应该对您有所帮助。

### 18.2.3 DispatcherServlet 处理序列

在您设置了 `DispatcherServlet` 并且针对该特定 `DispatcherServlet` 启动了一个请求后，`DispatcherServlet` 将按如下所示开始处理请求：

- 在请求中搜索并绑定 `WebApplicationContext` 作为控件和进程中的其他元素可以使用的属性。默认情况下，它将在 `DispatcherServlet.WEB_APPLICATION_CONTEXT_ATTRIBUTE` 键下绑定。
- 语言环境解析器被绑定到请求，以使进程中的元素能够解决在处理请求时使用的区域设置（渲染视图，准备数据等）。如果您不需要语言环境解析，则不需要它。
- 主题解析器被绑定到使得诸如视图之类的元素确定要使用哪个主题的请求。如果不使用主题，可以忽略它。
- 如果指定了多部分文件解析器，则会检查该请求的多部分；如果找到多部分，则请求被包装在一个 `MultipartHttpServletRequest` 中，以便进程中的其他元素进一步处理。有关多部分处理的更多信息，请参见[第18.10节“Spring的多部分（文件上传）支持”](#)。
- 搜索适当的处理器。如果找到处理器，则执行与处理器（预处理器，后处理器和控制器）关联的执行链，以便准备模型或呈现。
- 如果返回模型，则呈现视图。如果没有返回模型（可能是由于预处理器或后处理器拦截请求，可能是出于安全原因），因为请求可能已经被满足，所以不会呈现任何视图。

在 `WebApplicationContext` 中声明的处理器异常解析程序在处理请求期间提取异常。使用这些异常解析器允许您定义自定义行为来解决异常。

Spring `DispatcherServlet` 还支持返回由 Servlet API 指定的最后修改日期。确定特定请求的最后修改日期的过程很简单：`DispatcherServlet` 查找适当的处理器映射，并测试发现的处理器是否实现了 `LastModified` 接口。如果是，则 `LastModified` 接口的 `long getLastModified(request)` 方法的值将返回给客户端。

您可以通过将 Servlet 初始化参数（`init-param` 元素）添加到 `web.xml` 文件中的 Servlet 声明来自定义单独的 `DispatcherServlet` 实例。有关支持的参数列表，请参见下表。

**Table18.2.DispatcherServlet** 初始化参数

Parameter	Explanation
contextClass	实现 <code>WebApplicationContext</code> 的类，它实例化了这个 <code>Servlet</code> 使用的上下文。默认情况下，使用 <code>XmlWebApplicationContext</code> 。
contextConfigLocation	传递给上下文实例（由 <code>contextClass</code> 指定）以指示可以找到上下文的字符串。该字符串可能包含多个字符串（使用逗号作为分隔符）来支持多个上下文。在具有两次定义的 <code>bean</code> 的多个上下文位置的情况下，优先级最高。
namespace	<code>WebApplicationContext</code> 的命名空间。默认为 <code>[servlet-name]-servlet</code> 。

## 18.3 实现 Controllers

**Controllers** 提供对通常通过服务接口定义的应用程序行为的访问。Controllers 解释用户输入并将其转换为由视图表示给用户的模型。Spring 以非常抽象的方式实现控制器，使您能够创建各种各样的控制器。

Spring 2.5 引入了一种基于注释的编程模型，用于使用诸如 `@RequestMapping`，`@RequestParam`，`@ModelAttribute` 等注释的 MVC 控制器。以这种风格实现的控制器不必扩展特定的基类或实现特定的接口。此外，它们通常不直接依赖于 Servlet API，但是如果需要，您可以轻松地配置对 Servlet 设施的访问。



可用于 [Github 中的弹性项目 Org](#)，许多 Web 应用程序利用本节中描述的注释支持，包括 `MvcShowcase`，`MvcAjax`，`MvcBasic`，`PetClinic`，`PetCare` 等。

```
@Controller
public class HelloWorldController {

    @RequestMapping("/helloWorld")
    public String helloWorld(Model model) {
        model.addAttribute("message", "Hello World!");
        return "helloWorld";
    }
}
```

您可以看到，`@Controller` 和 `@RequestMapping` 注释允许灵活的方法名称和签名。在这个特殊的例子中，该方法接受一个 `Model` 并返回一个视图名称作为一个 `String`，但是可以使用各种其他的方法参数和返回值，如本节稍后所述。`@Controller` 和 `@RequestMapping` 和许多其他注释构成了 Spring MVC 实现的基础。本节介绍这些注释以及它们在 Servlet 环境中最常用的注释。

### 18.3.1 使用@Controller 定义控制器

`@Controller` 注释表示特定的类用于控制器的角色。Spring 不需要扩展任何控制器基类或引用 Servlet API。但是，如果需要，您仍然可以参考 Servlet 特定的功能。

`@Controller` 注释作为注释类的构造型，表示其作用。调度程序扫描这些注释类的映射方法，并检测 `@RequestMapping` 注释（请参阅下一节）。

您可以使用调度程序上下文中的标准 Spring bean 定义来明确定义带注释的控制器 bean。但是，`@Controller` 构造型还允许自动检测，与 Spring 通用支持对齐，用于检测类路径中的组件类并自动注册它们的 bean 定义。要启用自动检测这些带注释的控制器，您可以向组态添加组件扫描。使用 spring-context 模式，如以下 XML 代码片段所示：

```
<?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>
```

### 18.3.2 使用 @RequestMapping 映射请求

您可以使用 `@RequestMapping` 注释将诸如 `/appointments` 的 URL 映射到整个类或特定的处理程序方法。通常，类级注释将特定的请求路径（或路径模式）映射到表单控制器上，其他方法级注释缩小了特定 HTTP 请求方法（“GET”，“POST”等）的主映射，或 HTTP 请求参数条件。

Petcare示例中的以下示例显示了使用此注释的Spring MVC应用程序中的控制器：

```

@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(path = "/{day}", method = RequestMethod.GET)
    public Map<String, Appointment> getForDay(@PathVariable @DateTimeFormat(iso=ISO.DATE) Date day, Model model) {
        return appointmentBook.getAppointmentsForDay(day);
    }

    @RequestMapping(path = "/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";
    }
}

```

在上面的例子中，`@RequestMapping` 用在很多地方。第一个用法是类型（类）级别，这表示此控制器中的所有处理程序方法都相对于 `/appointments` 路径。`get()` 方法还有一个`@RequestMapping` 细化：它只接受 GET 请求，这意味着 `/appointments` 的 HTTP GET 调用此

方法。`add()` 有一个类似的细化，`getNewForm()` 将HTTP方法和路径的定义组合成一个，以便通过该方法处理 `appointments/new` 的GET请求。

`getForDay()` 方法显示了 `@RequestMapping` : URI模板的另一种用法。（参见“[URI模板模式](#)”一节）。

类级别上的 `@RequestMapping` 不是必需的。没有它，所有的路径都是绝对的，而不是相对的。`PetClinic`示例应用程序的以下示例显示了使用 `@RequestMapping` 的多操作控制器：

```

@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());
    }

}

```

上述示例不指定 GET 与 PUT , POST 等，因为 `@RequestMapping` 默认映射所有HTTP方法。使用 `@RequestMapping(method=GET)` 或 `@GetMapping` 来缩小映射。

## 组合`@RequestMapping`变体

Spring Framework 4.3引入了 `@RequestMapping` 注释的以下方法级组合变体，有助于简化常见HTTP方法的映射，并更好地表达注释处理程序方法的语义。例如，`@GetMapping` 可以被读取为GET `@RequestMapping`。

- `@GetMapping`
- `@PostMapping`
- `@PutMapping`
- `@DeleteMapping`
- `@PatchMapping`

以下示例显示了使用已组合的 `@RequestMapping` 注释简化的上一节中的 `AppointmentsController` 的修改版本。

```

@Controller
@RequestMapping("/appointments")
public class AppointmentsController {

    private final AppointmentBook appointmentBook;

    @Autowired
    public AppointmentsController(AppointmentBook appointmentBook) {
        this.appointmentBook = appointmentBook;
    }

    @GetMapping
    public Map<String, Appointment> get() {
        return appointmentBook.getAppointmentsForToday();
    }

    @GetMapping("/{day}")
    public Map<String, Appointment> getForDay(@PathVariable @DateTimeFormat(iso=ISO.DATE) Date day, Model model) {
        return appointmentBook.getAppointmentsForDay(day);
    }

    @GetMapping("/new")
    public AppointmentForm getNewForm() {
        return new AppointmentForm();
    }

    @PostMapping
    public String add(@Valid AppointmentForm appointment, BindingResult result) {
        if (result.hasErrors()) {
            return "appointments/new";
        }
        appointmentBook.addAppointment(appointment);
        return "redirect:/appointments";
    }
}

```

## @Controller 和 AOP 代理

在某些情况下，控制器可能需要在运行时用AOP代理进行装饰。一个例子是如果您选择在控制器上直接使用 `@Transactional` 注释。在这种情况下，对于控制器，我们建议使用基于类的代理。这通常是控制器的默认选项。但是，如果控制器必须实现不是Spring Context回调的接口（例如 `InitializingBean`，`* Aware` 等），则可能需要显式配置基于类的代理。例如，使用

```
<tx:annotation-driven />，更改为 <tx:annotation-driven proxy-target-class ="true"/>。
```

## Spring MVC 3.1 中的 @RequestMapping 方法的新支持类

Spring 3.1 分别为 `@RequestMapping` 方法引入了一组新的支持类，分别叫做 `RequestMappingHandlerMapping` 和 `RequestMappingHandlerAdapter`。它们被推荐使用，甚至需要利用 Spring MVC 3.1 中的新功能和未来。默认情况下，MVC 命名空间和 MVC Java 配置启用新的支持类，但是如果不用，则必须显式配置。本节介绍旧支持类和新支持类之间的一些重要区别。

在 Spring 3.1 之前，类型和方法级请求映射在两个单独的阶段进行了检查—首先由 `DefaultAnnotationHandlerMapping` 选择一个控制器，并且实际的调用方法被 `AnnotationMethodHandlerAdapter` 缩小。

使用 Spring 3.1 中的新支持类，`RequestMappingHandlerMapping` 是唯一可以决定哪个方法应该处理请求的地方。将控制器方法作为从类型和方法级 `@RequestMapping` 信息派生的每个方法的映射的唯一端点的集合。

这使得一些新的可能性。一旦 `HandlerInterceptor` 或 `HandlerExceptionResolver` 现在可以期望基于对象的处理器是 `HandlerMethod`，它允许它们检查确切的方法，其参数和关联的注释。URL 的处理不再需要跨不同的控制器进行拆分。

还有下面几件事情已经不复存在了：

- 首先使用 `SimpleUrlHandlerMapping` 或 `BeanNameUrlHandlerMapping` 选择控制器，然后基于 `@RequestMapping` 注释来缩小方法。
- 依赖于方法名称作为一种落后机制，以消除两个 `@RequestMapping` 方法之间的差异，这两个方法没有明确的路径映射 URL 路径，通过 HTTP 方法。在新的支持类中，`@RequestMapping` 方法必须被唯一地映射。
- 如果没有其他控制器方法更具体地匹配，请使用单个默认方法（无显式路径映射）处理请求。在新的支持类中，如果找不到匹配方法，则会引发 404 错误。

上述功能仍然支持现有的支持类。不过要利用新的 Spring MVC 3.1 功能，您需要使用新的支持类。

## URI 模版模式

可以使用 URI 模板方便地访问 `@RequestMapping` 方法中 URL 的所选部分。

URI 模板是一个类似 URI 的字符串，包含一个或多个变量名称。当您替换这些变量的值时，模板将成为一个 URI。所提出的 RFC 模板 RFC 定义了 URI 如何参数化。例如，URI 模板 `http://www.example.com/users/{userId}` 包含变量 `userId`。将 `fred` 的值分配给变量会得到 `http://www.example.com/users/fred`。

在Spring MVC中，您可以使用方法参数上的 `@PathVariable` 注释将其绑定到URI模板变量的值：

```
@GetMapping("/owners/{ownerId}")
public String findOwner(@PathVariable String ownerId, Model model) {
    Owner owner = ownerService.findOwner(ownerId);
    model.addAttribute("owner", owner);
    return "displayOwner";
}
```

URI模板“ / owners / {ownerId} ”指定变量名 `ownerId` 。当控制器处理此请求时，`ownerId` 的值将设置为在URI的适当部分中找到的值。例如，当 `/owner/fred` 出现请求时，`ownerId` 的值为 `fred` 。



要处理 `@PathVariable` 注释，Spring MVC需要按名称找到匹配的URI模板变量。您可以在注释中指定它：`*@GetMapping("/owners/{ownerId}")*public String findOwner(**@PathVariable("ownerId")** String theOwner, Model model) { // implementation omitted}` 或者如果URI模板变量名称与方法参数名称匹配，则可以省略该详细信息。只要您的代码使用Java 8编译调试信息或参数编译器标志，Spring MVC将将方法参数名称与URI模板变量名称相匹配：`*@GetMapping("/owners/{ownerId}")*public String findOwner(**@PathVariable** String ownerId, Model model) { // implementation omitted}`

一个方法可以有任何数量的 `@PathVariable` 注释：

```
@GetMapping("/owners/{ownerId}/pets/{petId}")
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";
}
```

当在 `Map <String, String>` 参数上使用 `@PathVariable` 注释时，映射将填充所有URI模板变量。

URI模板可以从类型和方法级别 `@RequestMapping` 注释中进行组合。因此，可以使  
用 `/owners/42/pets/21` 等URL调用 `findPet()` 方法。

```

@Controller
@RequestMapping("/owners/{ownerId}")
public class RelativePathUriTemplateController {

    @RequestMapping("/pets/{petId}")
    public void findPet(@PathVariable String ownerId, @PathVariable String petId, Model model) {
        // implementation omitted
    }

}

```

`@PathVariable` 参数可以是 any 简单的 `type such`，如 `int`，`long`，`Date` 等。如果没有这样做，Spring 将自动转换为适当的类型或抛出 `TypeMismatchException`。您还可以注册解析附加数据类型的支持。See the 部分称为“[方法参数和类型转换](#)”和“[定制 WebDataBinder 初始化](#)”一节。

## 具有正则表达式的URI模板模式

有时您需要更精确地定义 URI 模板变量。考虑 URL “`/spring-web/spring-web-3.0.5.jar`”。你怎么把它分解成多个部分？

`@RequestMapping` 注释支持在 URI 模板变量中使用正则表达式。语法是 `{varName : regex}`，其中第一部分定义了变量名，第二部分定义了正则表达式。例如：

```

@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 模板之外，`@RequestMapping` 注释和所有组合的 `@RequestMapping` 变体也支持 Ant 样式的路径模式（例如 `/myPath/*.do`）。还支持 URI 模板变量和 Ant-style glob 的组合（例如 `/owners/*/pets/{petId}`）。

## 路径模式比较

当 URL 匹配多个模式时，使用排序来查找最具体的匹配。

具有较少量 URI 变量和通配符的模式被认为更具体。例如 `/hotels/{hotel}/*` 具有 1 个 URI 变量和 1 个通配符，被认为比 `/hotels/{hotel}/**` 更具体，其中 1 个 URI 变量和 2 个通配符。

如果两个模式具有相同的计数，那么较长的模式被认为更具体。例如 `/foo/bar*` 比较长，被认为比 `/foo/*` 更具体。

当两个模式具有相同的计数和长度时，具有较少通配符的模式被认为更具体。例如 `/hotels/{hotel}` 比 `/hotels/*` 更具体。

下面有些额外增加的特殊的规则：

- 默认映射模式 `/**` 比任何其他模式都要小。例如 `/api/{a}/{b}/{c}` 更具体。
- 诸如 `/public/**` 之类的前缀模式比不包含双通配符的任何其他模式都不那么具体。例如 `/public/path3/{a}/{b}/{c}` 更具体。

有关详细信息，请参阅 `AntPathMatcher` 中的 `AntPatternComparator`。请注意，可以自定义 `PathMatcher`（参见 [Section 18.16.11, “Path Matching”](#)）。

## 具有占位符的路径模式

`@RequestMapping` 注释中的模式支持对本地属性 and/or 系统属性和环境变量的  `${...}`  占位符。在将控制器映射到的路径可能需要通过配置进行定制的情况下，这可能是有用的。有关占位符的更多信息，请参阅 `PropertyPlaceholderConfigurer` 类的 `javadocs`。

## 后缀模式匹配

默认情况下，Spring MVC 执行 `".**"` 后缀模式匹配，以便映射到 `/person` 的控制器也隐式映射到 `/person.*`。这使得通过 URL 路径（例如 `/person.pdf`, `/person.xml`）可以轻松地请求资源的不同表示。

后缀模式匹配可以关闭或限制为一组明确注册用于内容协商的路径扩展。通常建议通过诸如 `/person/{id}` 之类的常见请求映射来减少歧义，其中点可能不表示文件扩展名，例如 `/person/joe@email.com` VS `/person/joe@email.com.json`。此外，如下面的说明中所解释的，后缀模式匹配以及内容协商可能在某些情况下用于尝试恶意攻击，并且有充分的理由有意义地限制它们。

有关后缀模式匹配配置，请参见 [Section 18.16.11, “Path Matching”](#)，内容协商配置 [Section 18.16.6, “Content Negotiation”](#)。

## 后缀模式匹配和 RFD

反射文件下载（RFD）攻击是由 Trustwave 在 2014 年的一篇论文中首次描述的。攻击类似于 XSS，因为它依赖于响应中反映的输入（例如查询参数，URI 变量）。然而，不是将 JavaScript 插入到 HTML 中，如果基于文件扩展名（例如 `.bat`, `.cmd`）双击，则 RFD 攻击依赖于浏览器切换来执行下载并将响应视为可执行脚本。

在 Spring MVC `@ResponseBody` 和 `ResponseEntity` 方法存在风险，因为它们可以呈现客户端可以通过 URL 路径扩展请求的不同内容类型。但是请注意，单独禁用后缀模式匹配或禁用仅用于内容协商的路径扩展都可以有效地防止 RFD 攻击。

为了全面保护 RFD，在呈现响应体之前，Spring MVC 添加了 `Content-Disposition:inline;filename=f.txt` 头来建议一个固定和安全的下载文件。只有当 URL 路径包含既不是白名单的文件扩展名，也没有明确注册用于内容协商的目的，这是完成的。但是，当 URL 直接输入浏览器时，可能会产生副作用。

许多常见的路径扩展名默认为白名单。此外，REST API 调用通常不是直接用于浏览器中的 URL。然而，使用自定义 `HttpMessageConverter` 实现的应用程序可以明确地注册用于内容协商的文件扩展名，并且不会为此类扩展添加 `Content-Disposition` 头。见第 18.16.6 节“[Content Negotiation](#)”。



这是 [CVE-2015-5211](#) 工作的一部分。以下是报告中的其他建议：1、编码而不是转义 JSON 响应。这也是 OWASP XSS 的建议。有关 Spring 的例子，请参阅 [spring-jackson-owasp](#) 2、将后缀模式匹配配置为关闭或仅限于明确注册的后缀。3、配置使用属性“useJaf”和“ignoreUnknownPathExtensions”设置为 `false` 的内容协商，这将导致具有未知扩展名的 URL 的 406 响应。但是请注意，如果 URL 自然希望有一个结束点，这可能不是一个选择4、添加 `x-Content-Type-Options: nosniff` 头到响应。Spring Security 4 默认情况下执行此操作。

## 矩阵变量

URI 规范 [RFC 3986](#) 定义了在路径段中包含名称 - 值对的可能性。规格中没有使用具体术语。可以应用一般的“URI 路径参数”，尽管来自 Tim Berners-Lee 的旧帖子的更独特的“[Matrix URIs](#)”也经常被使用并且是相当熟知的。在 Spring MVC 中，这些被称为矩阵变量。

矩阵变量可以出现在任何路径段中，每个矩阵变量用“;”分隔（分号）。例如：“/cars;color=red;year=2012”。多个值可以是“,”（逗号）分隔 “color=red,green,blue”，或者变量名称可以重复 “color=red;color=green;color=blue”。

如果 URL 预期包含矩阵变量，则请求映射模式必须使用 URI 模板来表示它们。这确保了请求可以正确匹配，无论矩阵变量是否存在，以及它们提供什么顺序。

以下是提取矩阵变量“q”的示例：

```
// GET /pets/42;q=11;r=22

@GetMapping("/pets/{petId}")
public void findPet(@PathVariable String petId, @MatrixVariable int q) {

    // petId == 42
    // q == 11

}
```

由于所有路径段都可能包含矩阵变量，因此在某些情况下，您需要更具体地确定变量预期位于何处：

```
// GET /owners/42;q=11/pets/21;q=22

@GetMapping("/owners/{ownerId}/pets/{petId}")
public void findPet(
    @MatrixVariable(name="q", pathVar="ownerId") int q1,
    @MatrixVariable(name="q", pathVar="petId") int q2) {

    // q1 == 11
    // q2 == 22

}
```

矩阵变量可以定义为可选参数，并指定一个默认值：

```
// GET /pets/42

@GetMapping("/pets/{petId}")
public void findPet(@MatrixVariable(required=false, defaultValue="1") int q) {

    // q == 1

}
```

所有矩阵变量可以在Map中获得：

```
// GET /owners/42;q=11;r=12/pets/21;q=22;s=23

@GetMapping("/owners/{ownerId}/pets/{petId}")
public void findPet(
    @MatrixVariable MultiValueMap<String, String> matrixVars,
    @MatrixVariable(pathVar="petId") MultiValueMap<String, String> petMatrixVars)
{

    // matrixVars: [{"q" : [11,22], "r" : 12, "s" : 23}]
    // petMatrixVars: [{"q" : 11, "s" : 23}]

}
```

请注意，为了使用矩阵变量，您必须

将 `RequestMappingHandlerMapping` 的 `removeSemicolonContent` 属性设置为 `false`。默认设置为 `true`。

MVC Java 配置和 MVC 命名空间都提供了使用矩阵变量的选项。

如果您使用 Java 配置，使用 MVC Java Config 的高级自定义部分将介绍如何自定义 `RequestMappingHandlerMapping`。

在 MVC 命名空间中，`<mvc:annotation-driven>` 元素具有一个应该设置为 `true` 的 `enable-matrix-variables` 属性。默认情况下设置为 `false`。

```
<?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 enable-matrix-variables="true"/>

</beans>
```

## Consumable Media 类型

您可以通过指定 consumable media 类型的列表来缩小主要映射。只有当 `Content-Type` 请求头与指定的媒体类型匹配时，才会匹配该请求。例如：

```

@PostMapping(path = "/pets", consumes = "application/json")
public void addPet(@RequestBody Pet pet, Model model) {
    // implementation omitted
}

```

consumable media类型表达式也可以在 `!text/plain` 中否定，以匹配除 `Content-Type` 或 `text/plain` 之外的所有请求。还要考虑使用 `MediaType` 中提供的常量，例如 `APPLICATION_JSON_VALUE` 和 `APPLICATION_JSON_UTF8_VALUE`。



在类型和方法级别上支持 `_consumes_condition`。与大多数其他条件不同，当在类型级别使用时，方法级 `consumable types` 将覆盖而不是扩展类型级别的 `consumable types`。

## Producible Media 类型

您可以通过指定 `producible media` 类型列表来缩小主要映射。只有当 `Accept` 请求头匹配这些值之一时，才会匹配该请求。此外，使用产生条件确保用于产生响应的实际内容类型与产生条件中指定的媒体类型相关。例如：

```

@GetMapping(path = "/pets/{petId}", produces = MediaType.APPLICATION_JSON_UTF8_VALUE)
@ResponseBody
public Pet getPet(@PathVariable String petId, Model model) {
    // implementation omitted
}

```



在 `_products_condition` 中指定的媒体类型也可以选择指定一个字符集。例如，在上面的代码片段中，我们指定与默认配置在 `MappingJackson2HttpMessageConverter` 中的介质类型相同的介质类型，包括 `UTF-8 charset`。

就像消费一样，`producible media` 类型表达式可以被否定为 `!text/plain`，以匹配除了 `Accept` 头文件值为 `text/plain` 的所有请求。还要考虑使用 `MediaType` 中提供的常量，例如 `APPLICATION_JSON_VALUE` 和 `APPLICATION_JSON_UTF8_VALUE`。



在类型和方法级别上支持 `_produces_condition`。与大多数其他条件不同，当在类型级别使用时，方法级 `producible` 类型将覆盖而不是扩展类型级 `producible` 类型。

## 请求参数和头部值

您可以通过请求参数条件（如 "myParam"，"!myParam" 或 "myParam=myValue"）缩小请求匹配。前两个测试请求参数存在/不存在，第三个为特定参数值。下面是一个请求参数值条件的例子：

```
@Controller
@RequestMapping("/owners/{ownerId}")
public class RelativePathUriTemplateController {

    @GetMapping(path = "/pets/{petId}", params = "myParam=myValue")
    public void findPet(@PathVariable String ownerId, @PathVariable String petId, Model model) {
        // implementation omitted
    }

}
```

也可以根据特定的请求头值来测试请求头存在/不存在或匹配：

```
@Controller
@RequestMapping("/owners/{ownerId}")
public class RelativePathUriTemplateController {

    @GetMapping(path = "/pets", headers = "myHeader=myValue")
    public void findPet(@PathVariable String ownerId, @PathVariable String petId, Model model) {
        // implementation omitted
    }

}
```



虽然可以使用媒体类型通配符匹配`toContent-Type_and_Accept_header`值（例如“`content-type = text/*`”将匹配“`text/plain`”和“`text/html`”），但建议分别使用`_consumes_and_produces_cignitions`。它们专门用于此目的。

## HTTP 头部和 HTTP 可选项

映射到“GET”的`@RequestMapping`方法也隐式映射到“HEAD”，即不需要明确声明“HEAD”。处理HTTP HEAD请求就像是HTTP GET一样，而不是仅写入正文，仅计数字节数，并设置“Content-Length”头。

`@RequestMapping`方法内置支持HTTP选项。默认情况下，通过将所有`@RequestMapping`方法上显式声明的具有匹配URL模式的HTTP方法设置为“允许”响应头来处理HTTP OPTIONS请求。当没有明确声明HTTP方法时，“允许”标题设置为“GET，HEAD，POST，PUT，PATCH，

DELETE，OPTIONS”。理想地总是声明 `@RequestMapping` 方法要处理的HTTP方法，或者使用专用的组合 `@RequestMapping` 变体之一（参见“[Composed @RequestMapping Variants](#)”一节）。

虽然不需要 `@RequestMapping` 方法可以映射到HTTP HEAD或HTTP选项，也可以两者兼容。

### 18.3.3 定义 @RequestMapping 处理方法

`@RequestMapping` 处理方法可以有非常灵活的签名。支持的方法参数和返回值将在以下部分中介绍。大多数参数可以按任意顺序使用，唯一的例外是 `BindingResult` 参数。这将在下一节中介绍。



Spring 3.1 为 `@RequestMapping` 方法引入了一组新的支持类，分别称为 `RequestMappingHandlerMapping` 和 `RequestMappingHandlerAdapter`。它们被推荐使用，甚至需要利用 Spring MVC 3.1 中的新特性并继续前进。新的支持类默认从 MVC 命名空间启用，并且使用 MVC Java 配置，但是如果两者都不使用，则必须明确配置。

#### 支持的方法参数类型

下面是支持的方法参数类型：

- `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，而不涉及本机 Servlet API。
- `Request` 或 `response` 对象(Servlet API)。选择任意特定的请求或响应类型，例如 `ServletRequest` 或 `HttpServletRequest` 或 Spring's `MultipartRequest` / `MultipartHttpServletRequest`
- `Session` 对象 (Servlet API) 类型为 `HttpSession`。此类型的参数强制存在相应的会话。因此，这样的论证从不为 `null`。



会话访问可能不是线程安全的，特别是在 Servlet 环境中。如果允许多个请求同时访问会话，请考虑将 `RequestMappingHandlerAdapter` 的“`synchronizeOnSession`”标志设置为“`true`”。

- `java.servlet.http.PushBuilder` 用于关联的 Servlet 4.0 推送构建器 API，允许编程的 HTTP/2 资源推送。
- `java.security.Principal` (或一个特定的 `Principal` 实现类 (如果已知))，包含当前验证的用户。
- `org.springframework.http.HttpMethod` 为 HTTP 请求方法，表示为 Spring 的  `HttpMethod` 枚举。

- 由当前请求区域设置的 `java.util.Locale`，由最具体的语言环境解析器确定，实际上是在MVC环境中配置的 `LocaleResolver / LocaleContextResolver`。
- 与当前请求相关联的时区的 `java.util.TimeZone`（Java 6+）/ `java.time.ZoneId`（Java 8+），由 `LocaleContextResolver` 确定。
- `java.io.InputStream / java.io.Reader`，用于访问请求的内容。该值是由Servlet API公开的原始`InputStream / Reader`。
- `java.io.OutputStream / java.io.Writer` 用于生成响应的内容。该值是由Servlet API公开的原始`OutputStream / Writer`。
- `@PathVariable` 注释参数，用于访问URI模板变量。请参阅[the section called “URI Template Patterns”](#)
- `@MatrixVariable` 注释参数，用于访问位于URI路径段中的名称/值对。请参阅[the section called “Matrix Variables”](#)。
- `@RequestParam` 用于访问特定Servlet请求参数的注释参数。参数值将转换为声明的方法参数类型。请参阅[the section called “Binding request parameters to method parameters with @RequestParam”](#)。
- `@RequestHeader` 用于访问特定Servlet请求HTTP标头的注释参数。参数值将转换为声明的方法参数类型。请参阅[the section called “Mapping request header attributes with the @RequestHeader annotation”](#)。
- `@RequestBody` 用于访问HTTP请求体的注释参数。使用 `HttpMessageConverters` 将参数值转换为声明的方法参数类型。请参阅[the section called “Mapping the request body with the @RequestBody annotation”](#)。
- `@RequestPart`注释参数，用于访问“multipart / form-data”请求部分的内容。请参见[Section 18.10.5, “Handling a file upload request from programmatic clients”](#)和[Section 18.10, “Spring’s multipart \(file upload\) support”](#)
- `@SessionAttribute` 用于访问现有的永久会话属性（例如，用户认证对象）的注释参数，而不是通过 `@SessionAttributes` 作为控制器工作流的一部分临时存储在会话中的模型属性。
- `@RequestAttribute` 用于访问请求属性的注释参数。
- `HttpEntity` 参数访问Servlet请求HTTP头和内容。请求流将使用 `HttpMessageConverters` 转换为实体。请参阅[the section called “Using HttpEntity”](#)。
- `java.util.Map / org.springframework.ui.Model / org.springframework.ui.ModelMap` 用于丰富暴露于Web视图的隐式模型。

- 使用 `org.springframework.web.servlet.mvc.support.RedirectAttributes` 来指定在重定向情况下使用的精确的属性集，并且还添加 `Flash` 属性（临时存储在服务器端的属性，使其可以在请求之后使用重定向）。请参见 [the section called “Passing Data To the Redirect Target”](#) 和 [Section 18.6, “Using flash attributes”](#)。
- 根据 `@InitBinder` 方法和/或 `HandlerAdapter` 配置，命令或表单对象将请求参数绑定到 `bean` 属性（通过 `setter`）或直接转换为字段，并进行可定制的类型转换。请参阅 `RequestMappingHandlerAdapter` 上的 `webBindingInitializer` 属性。默认情况下，这些命令对象及其验证结果将作为模型属性公开，使用命令类名称 – 例如。对于“`some.package.OrderAddress`”类型的命令对象的 `model` 属性“`orderAddress`”。`ModelAttribute` 注释可以用于方法参数来自定义所使用的模型属性名称。
- `org.springframework.validation.Errors` / `org.springframework.validation.BindingResult` 验证前一个命令或表单对象的结果（即在前面的方法参数）。
- 用于将表单处理标记为完整的 `org.springframework.web.bind.support.SessionStatus` 状态句柄，它触发在处理程序类型级别上由 `@SessionAttributes` 注释指示的会话属性的清除。
- `org.springframework.web.util.UriComponentsBuilder` 用于准备与当前请求的主机，端口，方案，上下文路径以及 `servlet` 映射的文字部分相关的 URL 的构建器。

The `Errors` or `BindingResult` parameters have to follow the model object that is being bound immediately as the method signature might have more than one model object and Spring will create a separate `BindingResult` instance for each of them so the following sample won't work:

`Errors` 或 `BindingResult` 参数必须遵循正在绑定的模型对象，因为方法签名可能有多个模型对象，Spring 将为每个模型对象创建一个单独的 `BindingResult` 实例，因此以下示例将不起作用：

**BindingResult** 和 **@ModelAttribute** 的排序无效。

```
@PostMapping
public String processSubmit(@ModelAttribute("pet") Pet pet, Model model, BindingResult
result) { ... }
```

注意，`Pet` 和 `BindingResult` 之间有一个 `Model` 参数。要使其工作，您必须重新排序参数如下：

```
@PostMapping
public String processSubmit(@ModelAttribute("pet") Pet pet, BindingResult result, Model model) { ... }
```



支持JDK 1.8的 `java.util.Optional` 作为方法参数类型，其注解具有所需的属性（例如 `@RequestParam`，`@RequestHeader` 等）。在这些情况下使用 `java.util.Optional` 相当于 `required=false`。

## 支持的方法返回类型

以下是支持的返回类型：

- 一个 `ModelAndView` 对象，其中模型隐含地丰富了命令对象和 `@ModelAttribute` 注解引用数据访问器方法的结果。
- 一个 `Model` 对象，其视图名称通过 `RequestToViewNameTranslator` 隐式确定，隐式丰富了命令对象的模型以及 `@ModelAttribute` 注解引用数据访问器方法的结果。
- 用于暴露模型的 `Map` 对象，其视图名称通过 `RequestToViewNameTranslator` 隐式确定，隐式丰富了命令对象的模型以及 `@ModelAttribute` 注解引用数据访问器方法的结果。
- 一个 `View` 对象，其模型通过命令对象和 `@ModelAttribute` 注解引用数据访问器方法隐式确定。处理程序方法也可以通过声明一个 `Model` 参数（见上文）以编程方式丰富模型。
- 解释为逻辑视图名称的 `String` 值，模型通过命令对象和 `@ModelAttribute` 注解引用数据访问器方法隐式确定。处理程序方法也可以通过声明一个 `Model` 参数（见上文）以编程方式丰富模型。
- 如果方法处理响应本身（通过直接写入响应内容，为此目的声明一个类型为 `ServletResponse` / `HttpServletResponse` 的参数），或者如果视图名称通过 `RequestToViewNameTranslator` 隐式确定（不在处理程序方法签名）。
- 如果该方法用 `@ResponseBody` 注解，则返回类型将写入响应HTTP主体。返回值将使用 `HttpMessageConverters` 转换为声明的方法参数类型。请参阅 [the section called “Mapping the response body with the @ResponseBody annotation”](#)。
- 一个 `HttpEntity` 或 `ResponseEntity` 对象来提供对Servlet响应HTTP头和内容的访问。实体将使用 `HttpMessageConverters` 转换为响应流。请参阅[the section called “Using HttpEntity”](#)。
- 一个 `HttpHeaders` 对象返回没有正文的响应。
- 当应用程序想要在由Spring MVC管理的线程中异步生成返回值时，可以返回 `Callable`。

- 当应用程序想从自己选择的线程生成返回值时，可以返回 `DeferredResult`。
- 当应用程序想要从线程池提交中产生值时，可以返回 `ListenableFuture` 或 `CompletableFuture / CompletionStage`。
- 可以返回 `ResponseBodyEmitter` 以异步地将多个对象写入响应；也支持作为 `ResponseEntity` 内的主体。
- 可以返回 `SseEmitter` 以将异步的 Server-Sent 事件写入响应；也支持作为 `ResponseEntity` 内的主体。
- 可以返回 `StreamingResponseBody` 以异步写入响应 `OutputStream`；也支持作为 `ResponseEntity` 内的主体。
- 任何其他返回类型都被认为是要暴露给视图的单一模型属性，使用在方法级别（或基于返回类型类名称的默认属性名称）中通过 `@ModelAttribute` 指定的属性名称。该模型隐含地丰富了命令对象和 `@ModelAttribute` 注解引用数据访问器方法的结果。

## 通过 `@RequestParam` 绑定请求参数到方法

使用 `@RequestParam` 注解将请求参数绑定到控制器中的方法参数。

以下代码片段显示用法：

```

@Controller
@RequestMapping("/pets")
@SessionAttributes("pet")
public class EditPetForm {

    // ...

    @GetMapping
    public String setupForm(@RequestParam("petId") int petId, ModelMap model) {
        Pet pet = this.clinic.loadPet(petId);
        model.addAttribute("pet", pet);
        return "petForm";
    }

    // ...
}

```

默认情况下，使用此注解的参数是必需的，但您可以通过将 `@RequestParam` 的必需属性设置为 `false`（例如 `@RequestParam(name="id", required=false)`）来指定参数是可选的。

如果目标方法参数类型不是 `String`，则会自动应用类型转换。请参阅[the section called “Method Parameters And Type Conversion”](#)。

当在 `Map<String, String>` 或 `MultiValueMap<String, String>` 参数上使用 `@RequestParam` 注解时，映射将填充所有请求参数。

## 使用`@RequestBody`注解映射请求体

`@RequestBody` 方法参数注解表示方法参数应绑定到HTTP请求体的值。例如：

```
@PutMapping("/something")
public void handle(@RequestBody String body, Writer writer) throws IOException {
    writer.write(body);
}
```

通过使用 `HttpMessageConverter` 将请求体转换为 `method` 参数。`HttpMessageConverter` 负责将 HTTP 请求消息转换为对象，并从对象转换为 HTTP 响应体。

`RequestMappingHandlerAdapter` 支持带有以下默

认 `HttpMessageConverters` 的 `@RequestBody` 注解：

- `ByteArrayHttpMessageConverter` 转换字节数组。
- `StringHttpMessageConverter` 转换字符串。
- `FormHttpMessageConverter` 将表单数据转换为 `MultiValueMap <String, String>`。
- `SourceHttpMessageConverter` converts to/from a `javax.xml.transform.Source`.

有关这些转换器的更多信息，请参见 [Message Converters](#)。另请注意，如果使用 MVC 命名空间或 MVC Java 配置，默认情况下会注册更广泛的消息转换器。有关详细信息，请参见

[Section 18.16.1, “Enabling the MVC Java Config or the MVC XML Namespace”](#)。

如果您打算读写 XML，则需要使用 `org.springframework.oxm` 包中的特定 `Marshaller` 和 `Unmarshaller` 实现配置 `MarshallingHttpMessageConverter`。下面的示例显示了如何直接在配置中执行此操作，但是如果您的应用程序通过 MVC 命名空间或 MVC Java 配置进行配置，请参见 [Section 18.16.1, “Enabling the MVC Java Config or the MVC XML Namespace”](#)。

```

<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>

```

`@RequestBody` 方法参数可以用 `@Valid` 注解，在这种情况下，它将使用配置的 `validator` 实例进行验证。当使用MVC命名空间或MVC Java配置时，会自动配置一个JSR-303验证器，假设在类路径上可用JSR-303实现。

就像 `@ModelAttribute` 参数一样，可以使用一个 `Errors` 参数来检查错误。如果未声明此类参数，则将引发 `MethodArgumentNotValidException` 异常。该异常在 `DefaultHandlerExceptionResolver` 中处理，它将向客户端发送一个 `400` 错误。



有关通过MVC命名空间或MVC Java配置配置消息转换器和验证器的信息，请参见 [Section 18.16.1, “Enabling the MVC Java Config or the MVC XML Namespace”](#)。

## 使用`@ResponseBody`注解映射响应体

`@ResponseBody` 注解类似于 `@RequestBody`。这个注解可以放在一个方法上，并且指示返回类型应该直接写入HTTP响应体（而不是放置在模型中，或者解释为视图名称）。例如：

```

@GetMapping("/something")
@ResponseBody
public String helloWorld() {
    return "Hello World";
}

```

上面的例子将会导致文本 `Hello World` 被写入到HTTP响应流中。

与 `@RequestBody` 一样，Spring 通过使用 `HttpMessageConverter` 将返回的对象转换为响应正文。有关这些转换器的更多信息，请参阅上一节和 [Message Converters](#)。

## 使用 `@RestController` 注解创建 REST 控制器

控制器实现 REST API 是一种非常普通的用例，因此只能提供 JSON、XML 或自定义的 `MediaType` 内容。为了方便起见，您可以使用 `@RestController` 注解您的控制器类，而不是使用 `@ResponseBody` 注解所有的 `@RequestMapping` 方法。

`@RestController` 是一个结合了 `@ResponseBody` 和 `@Controller` 的构造型注解。更重要的是，它给你的控制器带来了更多的意义，也可能在框架的未来版本中带来额外的语义。

与常规 `@Controller`s 一样，`@RestController` 可以由 `@ControllerAdvice` 或 `@RestControllerAdvice` bean 来协助。有关详细信息，请参阅 [the section called “Advising controllers with `@ControllerAdvice` and `@RestControllerAdvice`”](#) 部分。

## 使用 `HttpEntity`

这 `HttpEntity` 是相似的 `@RequestBody` 和 `@ResponseBody`。除了访问请求和响应主体之外 `HttpEntity`（和响应特定子类  `ResponseEntity`）还允许访问请求和响应头，如下所示：

```
@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);
}
```

上述示例获取 `MyRequestHeader` 请求标头的值，并将其作为字节数组读取。它将 `MyResponseHeader` 响应添加到 `Hello World` 响应流中，并将响应状态代码设置为 201（已创建）。

至于 `@RequestBody` 和 `@ResponseBody`，Spring 使用 `HttpMessageConverter` 从和请求和响应流转换。有关这些转换器的更多信息，请参阅上一部分和 [消息转换器](#)。

## 在方法上使用 `@ModelAttribute`

该 `@ModelAttribute` 注解可以对方法或方法的参数来使用。本节将介绍其在方法上的用法，下一节将介绍其在方法参数上的用法。

方法上的 `@ModelAttribute` 指示该方法的目的是添加一个或多个模型属性。这些方法支持与 `@RequestMapping` 方法相同的参数类型，但不能直接映射到请求。相反，控制器中的 `@ModelAttribute` 方法在相同控制器内的 `@RequestMapping` 方法之前被调用。几个例子：

```
// 添加一个属性
// 该方法的返回值被添加到名为“account”的模型中
// 您可以通过@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` 方法用于填充具有常用属性的模型，例如使用状态或宠物类型填充下拉列表，或者检索诸如 `Account` 的命令对象，以便使用它来表示 `HTML` 表单上的数据。后一种情况在下一节进一步讨论。

注意两种风格的 `@ModelAttribute` 方法。在第一个方法中，该方法通过返回它隐式地添加一个属性。在第二个方法中，该方法接受 `Model` 并添加任意数量的模型属性。您可以根据需要选择两种风格。

控制器可以有多种 `@ModelAttribute` 方法。所有这些方法都 `@RequestMapping` 在相同控制器的方法之前被调用。

`@ModelAttribute` 方法也可以在一个 `@ControllerAdvice` 注释类中定义，并且这种方法适用于许多控制器。有关更多详细信息，请参阅[“使用@ControllerAdvice和@RestControllerAdvice建议控制器”一节](#)。



当没有明确指定模型属性名称时会发生什么？在这种情况下，根据其类型将默认名称分配给模型属性。例如，如果该方法返回类型的对象 `Account`，则使用的默认名称为“`account`”。您可以通过 `@ModelAttribute` 注释的值更改它。如果直接添加属性 `Model`，请使用适当的重载 `addAttribute(..)` 方法 — 即，带有或不带有属性名称。

该 `@ModelAttribute` 注解可在使用 `@RequestMapping` 方法为好。在这种情况下，`@RequestMapping` 方法的返回值将被解释为模型属性而不是视图名称。视图名称是基于视图名称约定导出的，非常类似于返回的方法 `void` - 请参见第18.13.3节“View – RequestToViewNameTranslator”。

## 在方法参数上使用`@ModelAttribute`

如上一节所述 `@ModelAttribute`，可以在方法或方法参数上使用。本节介绍了其在方法参数中的用法。

一个 `@ModelAttribute` 上的方法参数指示参数应该从模型中检索。如果模型中不存在，参数首先被实例化，然后添加到模型中。一旦出现在模型中，参数的字段应该从具有匹配名称的所有请求参数中填充。这被称为Spring MVC中的数据绑定，这是一种非常有用的机制，可以节省您逐个解析每个表单字段。

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public String processSubmit(@ModelAttribute Pet pet) { }
```

鉴于上述例子，`Pet`实例可以从哪里来？有几个选择：

- 由于使用 `@SessionAttributes` - 可能已经在模型中 - 请参阅“使用`@SessionAttributes`将模型属性存储在请求之间的HTTP会话中”一节。
- 由于 `@ModelAttribute` 在同一控制器中的方法，它可能已经在模型中 - 如上一节所述。
- 它可以基于URI模板变量和类型转换器（下面更详细地解释）来检索。
- 它可以使用其默认构造函数实例化。

一种 `@ModelAttribute` 方法是从数据库中检索属性的常用方法，可以通过使用可选地在请求之间存储属性 `@SessionAttributes`。在某些情况下，通过使用URI模板变量和类型转换器来检索属性可能很方便。这是一个例子：

```
@PutMapping("/accounts/{account}")
public String save(@ModelAttribute("account") Account account) {
    // ...
}
```

在此示例中，模型属性（即“`account`”）的名称与URI模板变量的名称相匹配。如果您注册 `converter`，可以将 `String` 帐户值转换为一个 `Account` 实例，则上述示例将无需使用 `@ModelAttribute` 方法。

下一步是数据绑定。该 `WebDataBinder` 级比赛要求参数名称-包括查询字符串参数和表单域-以模拟通过名称属性字段。在必要时已经应用了类型转换（从字符串到目标字段类型）之后填充匹配字段。数据绑定和验证在[第5章验证，数据绑定和类型转换中介绍](#)。自定义控制器级别的数据绑定过程将在“[自定义WebDataBinder初始化](#)”一节中介绍。

由于数据绑定，可能会出现错误，例如缺少必填字段或类型转换错误。要检查这些错误，请在 `BindingResult` 参数后立即添加一个 `@ModelAttribute` 参数：

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public String processSubmit(@ModelAttribute("pet") Pet pet, BindingResult result) {

    if (result.hasErrors()) {
        return "petForm";
    }

    // ...

}
```

使用一个 `BindingResult` 你可以检查是否发现错误，在这种情况下，渲染相同的形式通常是在 Spring 的 `<errors>` 表单标签的帮助下显示错误的。

请注意，在某些情况下，在没有数据绑定的情况下获取模型中的属性可能是有用的。对于这种情况，您可以将其注入 `Model` 控制器，或者使用注释上的 `binding` 标志：

```
@ModelAttribute
public AccountForm setUpForm() {
    return new AccountForm();
}

@ModelAttribute
public Account findAccount(@PathVariable String accountId) {
    return accountRepository.findOne(accountId);
}

@PostMapping("update")
public String update(@Valid AccountUpdateForm form, BindingResult result,
    @ModelAttribute(binding=false) Account account) {

    // ...
}
```

除了数据绑定之外，您还可以使用自己的自定义验证器调用验证，传递与 `BindingResult` 用于记录数据绑定错误相同的验证器。这允许在一个地方累积数据绑定和验证错误，并随后向用户报告：

```

@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public String processSubmit(@ModelAttribute("pet") Pet pet, BindingResult result) {

    new PetValidator().validate(pet, result);
    if (result.hasErrors()) {
        return "petForm";
    }

    // ...

}

```

或者您可以通过添加JSR-303 `@Valid` 注解自动调用验证：

```

@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public String processSubmit(@Valid @ModelAttribute("pet") Pet pet, BindingResult result) {

    if (result.hasErrors()) {
        return "petForm";
    }

    // ...

}

```

有关如何配置和使用验证的详细信息，请参见第5.8节“[Spring验证](#)”和第5章验证，数据绑定和类型转换。

## 使用 `@SessionAttributes` 将模型属性存储在请求之间的HTTP会话中

类型级 `@SessionAttributes` 注释声明特定处理程序使用的会话属性。这通常将列出模型属性或模型属性的类型，这些模型属性或类型应该透明地存储在会话或某些会话存储中，作为后续请求之间的格式支持bean。

以下代码片段显示了此注解的用法，指定了模型属性名称：

```

@Controller
@RequestMapping("/editPet.do")
@SessionAttributes("pet")
public class EditPetForm {
    // ...
}

```

## 使用 **@SessionAttribute** 访问预先存在的全局会话属性

如果您需要访问全局管理的预先存在的会话属性，即控制器外部（例如，通过过滤器），并且可能存在或可能不存在，`@SessionAttribute` 则会使用方法参数上的注解：

```
@RequestMapping("/")
public String handle(@SessionAttribute User user) {
    // ...
}
```

对于需要添加或删除会话属性的用例，请考虑注

入 `org.springframework.web.context.request.WebRequest` 或 `javax.servlet.http.HttpSession` 控制方法。

为了在会话中临时存储模型属性作为控制器工作流的一部分，请考虑使用“[使用 \*\*@SessionAttributes\*\* 将模型属性存储在请求之间的HTTP会话中](#)” `SessionAttributes` 中所述的一节。

## 使用 **@RequestAttribute** 来访问请求属性

到类似 `@SessionAttribute` 的 `@RequestAttribute` 注解可以被用于访问由滤波器或拦截器创建的预先存在的请求属性：

```
@RequestMapping("/")
public String handle(@RequestAttribute Client client) {
    // ...
}
```

## 使用“**application / x-www-form-urlencoded**”数据

以前的章节介绍了 `@ModelAttribute` 如何支持浏览器客户端的表单提交请求。建议与非浏览器客户端的请求一起使用相同的注解。然而，在使用HTTP PUT请求时，有一个显着的区别。浏览器可以通过HTTP GET或HTTP POST提交表单数据。非浏览器客户端也可以通过HTTP PUT提交表单。这提出了一个挑战，因为Servlet规范要求 `ServletRequest.getParameter*()` 一系列方法仅支持HTTP POST的表单域访问，而不支持HTTP PUT。

为了支持HTTP PUT和PATCH请求，该 `spring-web` 模块提供了 `HttpPutFormContentFilter` 可以在以下配置中的过滤器 `web.xml`：

```

<filter>
    <filter-name>httpPutFormFilter</filter-name>
    <filter-class>org.springframework.web.filter.HttpPutFormContentFilter</filter-clas
s>
</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>

```

上述过滤器拦截具有内容类型的HTTP PUT和PATCH请求 `application/x-www-form-urlencoded`，从请求的正文 中读取表单数据，并包装 `ServletRequest` 以便通过 `ServletRequest.getParameter*()` 一系列方法使表单数据可用。



由于 `HttpPutFormContentFilter` 消耗了请求的正文，因此不应配置为依赖其他转换器的 `PUT`或`PATCH` URL `application/x-www-form-urlencoded`。这包括 `@RequestBody MultiValueMap and HttpEntity`。

## 使用`@CookieValue`注解映射Cookie值

该 `@CookieValue` 注解允许将方法参数绑定到HTTP cookie的值。

让我们考虑以下cookie已被接收到http请求：

```
JSESSIONID=415A4AC178C59DACE0B2C9CA727CDD84
```

以下代码示例演示如何获取 `JSESSIONID` cookie 的值：

```

@RequestMapping("/displayHeaderInfo.do")
public void displayHeaderInfo(@CookieValue("JSESSIONID") String cookie) {
    //...
}

```

如果目标方法参数类型不是，则会自动应用类型转换 `String`。请参阅“[方法参数和类型转换](#)”一节。

## 使用`@RequestHeader`注解映射请求标头属性

该 `@RequestHeader` 注解允许将一个方法参数绑定到请求头。

以下是一个示例请求标头：

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

以下代码示例演示了如何获取 `Accept-Encoding` 和 `Keep-Alive` 标题的值：

```
@RequestMapping("/displayHeaderInfo.do")
public void displayHeaderInfo(@RequestHeader("Accept-Encoding") String encoding,
    @RequestHeader("Keep-Alive") long keepAlive) {
    //...
}
```

如果方法参数不是 `String`，则会自动应用类型转换。查看[the section called “Method Parameters And Type Conversion”](#)一节。

在 `Map`，`MultiValueMap` 或 `HttpHeaders` 参数上使用 `@RequestHeader` 注解时，映射将填充所有标题值。



内置支持可用于将逗号分隔的字符串转换为字符串或类型转换系统已知的其他类型的数组/集合。例如，注解的方法参数 `@RequestHeader("Accept")` 可以是类型 `String`，也可以是 `String[]` 或 `List`。

## 方法参数和类型转换

从请求中提取的基于字符串的值（包括请求参数，路径变量，请求头和cookie值）可能需要转换为方法参数或字段的目标类型（例如，将请求参数绑定到参数中的字段 `@ModelAttribute`）。他们一定会。如果目标类型不是 `String`，Spring将自动转换为相应的类型。支持所有简单的类型，如`int`, `long`, `Date`等。您可以进一步自定义通过转换过程 `WebDataBinder`（见[称为“定制WebDataBinder初始化”一节](#)），或者通过注册 `Formatters` 与 `FormattingConversionService`（参见[5.6节，“春字段格式”](#)）。

## 自定义 `WebDataBinder` 初始化

要通过 Spring 定制与 PropertyEditor 的请求参数绑定 WebDataBinder，可以使用 @InitBinder 控制器中的-annotated @InitBinder 方法， @ControllerAdvice 类中的方法或提供自定义 WebBindingInitializer。有关更多详细信息，请参阅“[使用@ControllerAdvice 和 @RestControllerAdvice 建议控制器器](#)”一节。

### 使用@InitBinder 自定义数据绑定

Annotating controller methods with @InitBinder allows you to configure web data binding directly within your controller class. @InitBinder identifies methods that initialize the WebDataBinder that will be used to populate command and form object arguments of annotated handler methods.

Such init-binder methods support all arguments that @RequestMapping methods support, except for command/form objects and corresponding validation result objects. Init-binder methods must not have a return value. Thus, they are usually declared as void. Typical arguments include WebDataBinder in combination with WebRequest or java.util.Locale, allowing code to register context-specific editors.

The following example demonstrates the use of @InitBinder to configure a CustomDateEditor for all java.util.Date form properties.

注解控制器方法， @InitBinder 允许您直接在控制器类中配置 Web 数据绑定。 @InitBinder 识别用于初始化 WebDataBinder 将用于填充命名和表示注解处理程序方法的对象参数的方法。

这种 init-binder 方法支持方法支持的所有参数 @RequestMapping，除了命令/表单对象和相应的验证结果对象。Init-binder 方法不能有返回值。因此，它们通常被声明为 void。典型的参数包括 WebDataBinder 与 WebRequest 或者 java.util.Locale 允许代码注册上下文相关的编辑器。

以下示例演示如何使用 @InitBinder 为所有 java.util.Date 表单属性配置 CustomDateEditor。

```
@Controller
public class MyFormController {

    @InitBinder
    protected void initBinder(WebDataBinder binder) {
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        dateFormat.setLenient(false);
        binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, false));
    }

    // ...
}
```

或者，从 Spring 4.2 起，考虑使用 `addCustomFormatter` 来指定 `Formatter` 实现而不是 `PropertyEditor` 实例。如果您碰巧 `Formatter` 在共享 `FormattingConversionService` 中安装一个基于安装程序的方法，那么特别有用的方法可以重用于控制器特定的绑定规则调整。

```
@Controller
public class MyFormController {

    @InitBinder
    protected void initBinder(WebDataBinder binder) {
        binder.addCustomFormatter(new DateFormatter("yyyy-MM-dd"));
    }

    // ...
}
```

### 配置自定义 `WebBindingInitializer`

要外部化数据绑定初始化，您可以提供 `WebBindingInitializer` 接口的自定义实现，然后通过为其提供自定义 `bean` 配置来启用 `RequestMappingHandlerAdapter`，从而覆盖默认配置。

PetClinic 应用程序中的以下示例显示了使用该接口的自定义实现的 `WebBindingInitializer` 配置 `org.springframework.samples.petclinic.web.ClinicBindingInitializer`，它配置了几个 PetClinic 控制器所需的 `PropertyEditor`。

```
<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>
```

`@InitBinder` 方法也可以在 `@ControllerAdvice` 注解类中定义，在这种情况下，它们适用于匹配的控制器。这提供了使用 `WebBindingInitializer` 的替代方法。有关更多详细信息，请参阅“[使用 `@ControllerAdvice` 和 `@RestControllerAdvice` 建议控制器](#)”一节。

## 通过 `@ControllerAdvice` 和 `@RestControllerAdvice` 为控制器提供建议

`@ControllerAdvice` 注解是一个组件注解允许实现类自动检测通过类路径扫描。当使用 MVC 命名空间或 MVC Java 配置时，它将自动启用。

使用 `@ControllerAdvice` 注解的类可以包

含 `@ExceptionHandler`，`@InitBinder` 和 `@ModelAttribute` 注解的方法，这些方法将适用于 `@RequestMapping` 所有控制器的层次结构的方法，而不是内声明它们控制器层次。

`@RestControllerAdvice` 是 `@ExceptionHandler` 方法默认使用 `@ResponseBody` 语义的替代方案。

`@ControllerAdvice` 和 `@RestControllerAdvice` 都可以定位控制器的一个子集：

```
// Target all Controllers annotated with @RestController
@ControllerAdvice(annotations = RestController.class)
public class AnnotationAdvice {}

// Target all Controllers within specific packages
@ControllerAdvice("org.example.controllers")
public class BasePackageAdvice {}

// Target all Controllers assignable to specific classes
@ControllerAdvice(assignableTypes = {ControllerInterface.class, AbstractController.class})
public class AssignableTypesAdvice {}
```

查看 [@ControllerAdvice 文档](#) 了解更多详细信息。

## Jackson序列化视图支持

有时将内容过滤将被序列化到HTTP响应主体的对象有时是有用的。为了提供这样的功能，Spring MVC内置了对[Jackson的Serialization Views](#)进行渲染的支持。

要将它用于返回  `ResponseEntity` 的 `@ResponseBody` 控制器方法或控制器方法，只需添加 `@JsonView` 注解，并指定要使用的视图类或接口的类参数即可：

```

@RestController
public class UserController {

    @GetMapping("/user")
    @JsonView(User.WithoutPasswordView.class)
    public User getUser() {
        return new User("eric", "7!jd#h23");
    }

    public class User {

        public interface WithoutPasswordView {};
        public interface WithPasswordView extends WithoutPasswordView {};

        private String username;
        private String password;

        public User() {}

        public User(String username, String password) {
            this.username = username;
            this.password = password;
        }

        @JsonView(WithoutPasswordView.class)
        public String getUsername() {
            return this.username;
        }

        @JsonView(WithPasswordView.class)
        public String getPassword() {
            return this.password;
        }
    }
}

```



请注意，尽管 `@JsonView` 允许指定多个类，但在控制器方法上的使用只支持一个类参数。如果需要启用多个视图，请考虑使用复合接口。

对于依赖于视图分辨率的控制器，只需将序列化视图类添加到模型中：

```

@Controller
public class UserController extends AbstractController {

    @GetMapping("/user")
    public String getUser(Model model) {
        model.addAttribute("user", new User("eric", "7!jd#h23"));
        model.addAttribute(JsonView.class.getName(), User.WithoutPasswordView.class);
        return "userView";
    }
}

```

## Jackson JSONP 支持

为了启用JSONP支持 `@ResponseBody` 和 `ResponseEntity` 方法，声明一个 `@ControllerAdvice` 扩展的bean，`AbstractJsonpResponseBodyAdvice` 如下所示，构造函数参数指示JSONP查询参数名称：

```

@ControllerAdvice
public class JsonpAdvice extends AbstractJsonpResponseBodyAdvice {

    public JsonpAdvice() {
        super("callback");
    }
}

```

对于依赖于视图分辨率的控制器，当请求具有名为 `jsonp` 或者的查询参数时，将自动启用 JSONP `callback`。这些名字可以通过 `jsonpParameterNames` 财产定制。

### 18.3.4 异步请求处理

Spring MVC 3.2介绍了基于Servlet 3的异步请求处理。与往常一样，一个控制器方法现在可以返回一个 `java.util.concurrent.Callable` 并从Spring MVC管理的线程生成返回值，而不是返回一个值。同时，主要的Servlet容器线程被退出并被释放并允许处理其他请求。Spring MVC `Callable` 在一个单独的线程中调用一个单独的线程 `TaskExecutor`，当 `Callable` 返回时，请求被分派回到Servlet容器，以使用返回的值来恢复处理 `Callable`。这是一个这样一个控制器方法的例子：

```
@PostMapping
public Callable<String> processUpload(final MultipartFile file) {

    return new Callable<String>() {
        public String call() throws Exception {
            // ...
            return "someView";
        }
    };
}
```

另一个选项是控制器方法返回一个实例 `DeferredResult`。在这种情况下，返回值也将从任何线程生成，即不由Spring MVC管理的线程。例如，可以响应于诸如JMS消息，计划任务等的一些外部事件而产生结果。这是这样一个控制器方法的例子：

```
@RequestMapping("/quotes")
@ResponseBody
public DeferredResult<String> quotes() {
    DeferredResult<String> deferredResult = new DeferredResult<String>();
    // Save the deferredResult somewhere..
    return deferredResult;
}

// In some other thread...
deferredResult.setResult(data);
```

没有任何Servlet 3.0异步请求处理功能的知识可能难以理解。这肯定有助于阅读。以下是有关基本机制的几个基本事实：

- 通过调用 `request.startAsync()`，可以将 `ServletRequest` 置于异步模式。这样做的主要效果是，Servlet以及任何过滤器都可以退出，但响应将保持打开状态，以便稍后完成处理。

- 可以用于进一步控制异步处理的 `request.startAsync()` 返回调用 `AsyncContext`。例如，它提供的方法 `dispatch` 类似于Servlet API中的转发，但它允许应用程序在Servlet容器线程上恢复请求处理。
- 在 `ServletRequest` 提供对当前 `DispatcherType` 可用于处理所述初始请求，一个异步调度，正向，以及其他调度类型之间进行区分。

考虑到上述情况，以下是使用 `Callable` 进行异步请求处理的事件序列：

- 控制器返回一个 `Callable`。
- Spring MVC开始异步处理，并将 `Callable` 提交给 `TaskExecutor`，以便在单独的线程中进行处理。
- `DispatcherServlet` 和所有Filter都退出Servlet容器线程，但响应保持打开状态。
- `Callable` 生成一个结果，Spring MVC将请求发送回Servlet容器以恢复处理。
- `DispatcherServlet` 被再次调用，处理从 `Callable` 异步产生的结果中恢复。

`DeferredResult` 的序列非常相似，除非由应用程序生成任何线程的异步结果：

- 控制器返回一个 `DeferredResult` 并将其保存在某些内存中的队列或列表中，可以访问它。
- Spring MVC启动异步处理。
- `DispatcherServlet` 和所有配置的Filter都退出请求处理线程，但响应保持打开状态。
- 应用程序从某个线程设置 `DeferredResult`，Spring MVC将请求分派回Servlet容器。
- `DispatcherServlet`再次被调用，并且处理以异步产生的结果继续。

有关异步请求处理的动机的进一步背景，何时或为什么使用它，请阅读此[博客文章系列](#)。

## 异步请求异常处理

如果从控制器方法返回的 `callable` 在执行时引发异常，会发生什么情况？简短的答案与控制器方法引发异常时发生的情况相同。它经历了常规的异常处理机制。更长的解释是，当 `callable` 引发异常Spring MVC调度到Servlet容器与异常的结果，并导致恢复请求处理与异常，而不是一个控制器方法的返回值。当使用 `DeferredResources` 时，您可以选择是否使用 `Exception` 实例调用 `setResult` 或 `setErrorResult`。

## 拦截异步请求

`HandlerInterceptor` 还可以实现 `AsyncHandlerInterceptor` 以实现 `afterConcurrentHandlingStarted` 回调，在异步处理启动时调用，而不是 `postHandle` 和 `afterCompletion`。

`HandlerInterceptor` 也可以注册一个 `CallableProcessingInterceptor` 或者一个 `DeferredResultProcessingInterceptor`，以便更深入地集成异步请求的生命周期，例如处理超时事件。有关更多详细信息，请参阅 `AsyncHandlerInterceptor` 的 Javadoc。

`DeferredResult` 类型还提供诸如 `onTimeout(Runnable)` 和 `onCompletion(Runnable)` 等方法。有关更多详细信息，请参阅 `DeferredResult` 的 Javadoc。

使用 `Callable` 时，可以用 `WebAsyncTask` 的实例包装它，该实例还提供了超时和完成的注册方法。

## HTTP 流式传输

控制器方法可以异步地使用 `DeferredResult` 和 `Callable` 产生其返回值，并且可以用于实现诸如[长轮询的技术](#)，其中服务器可以尽快将事件推送到客户端。

如果您想在单个 HTTP 响应中推送多个事件怎么办？这是一个与“长查询”相关的技术，被称为“HTTP 流”。Spring MVC 可以通过 `ResponseBodyEmitter` 可以用于发送多个对象的返回值类型来实现，而不是像通常情况那样 `@ResponseBody` 发送的对象，其中每个发送的对象都被写入到响应中 `HttpMessageConverter`。

这是一个例子：

```
RequestMapping("/events")
public ResponseBodyEmitter handle() {
    ResponseBodyEmitter emitter = new ResponseBodyEmitter();
    // Save the emitter somewhere..
    return emitter;
}

// In some other thread
emitter.send("Hello once");

// and again later on
emitter.send("Hello again");

// and done at some point
emitter.complete();
```

请注意，`ResponseBodyEmitter` 也可以用作  `ResponseEntity` 中的主体，以便自定义响应的状态和标题。

## HTTP 流与服务器发送的事件

`SseEmitter` 是 `ResponseBodyEmitter` 的一个子类，为 [服务器发送事件](#) 提供支持。服务器发送的事件是相同的“HTTP流式传输”技术的另一种变体，除了从服务器推送的事件根据W3C服务器发送的事件规范格式化。

服务器发送的事件可用于其预期的目的，即将事件从服务器推送到客户端。在 Spring MVC 中很容易，只需要返回一个 `SseEmitter` 类型的值。

请注意，[Internet Explorer](#) 不支持服务器发送事件，而对于更高级的 Web 应用程序消息传递场景（如在线游戏，协作，财务应用程序等），最好考虑 Spring 的 [WebSocket](#) 支持，其中包括 [SockJS](#) 风格的 [WebSocket](#) 仿真回落到非常广泛的浏览器（包括 [Internet Explorer](#)）以及更高级别的消息传递模式，用于通过更多以消息为中心的体系结构中的发布订阅模型与客户端进行交互。有关进一步的背景，请参阅 [以下博文](#)。

## HTTP 直接流向 OutputStream

`ResponseBodyEmitter` 允许通过 `HttpMessageConverter` 将对象写入响应来发送事件。这可能是最常见的情况，例如编写 `JSON` 数据时。但是有时候绕过消息转换并直接写入响应 `OutputStream`（例如文件下载）是有用的。这可以通过 `StreamingResponseBody` 返回值类型来完成。

这是一个例子：

```
@RequestMapping("/download")
public StreamingResponseBody handle() {
    return new StreamingResponseBody() {
        @Override
        public void writeTo(OutputStream outputStream) throws IOException {
            // write...
        }
    };
}
```

请注意，`StreamingResponseBody` 也可以用作  `ResponseEntity` 中的主体，以便自定义响应的状态和标题。

## 配置异步请求处理

### Servlet 容器配置

对于配置 `web.xml` 为确保更新到版本 3.0 的应用程序：

```

<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
version="3.0">

  ...

</web-app>

```

必须在 `DispatcherServlet` 上通过 `web.xml` 中的 `<async-supported>true</async-supported>` 子元素启用异步支持。另外，任何参与异步请求处理的 `Filter` 都必须配置为支持`ASYNC`分派器类型。为 `Spring Framework` 提供的所有过滤器启用`ASYNC`分派器类型应该是安全的，因为它们通常会扩展 `OncePerRequestFilter`，并且运行时检查过滤器是否需要参与异步调度。

以下是一些`web.xml`配置示例：

```

<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">

  <filter>
    <filter-name>Spring_OpenEntityManagerInViewFilter</filter-name>
    <filter-class>org.springframework.~.OpenEntityManagerInViewFilter</filter-clas
s>
    <async-supported>true</async-supported>
  </filter>

  <filter-mapping>
    <filter-name>Spring_OpenEntityManagerInViewFilter</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>ASYNC</dispatcher>
  </filter-mapping>

</web-app>

```

如果通过 `WebApplicationInitializer` 使用 `Servlet 3`，基于 `Java` 的配置，则还需要像 `web.xml` 一样设置“`asyncSupported`”标志以及`ASYNC`分派器类型。为了简化所有这些配置，可以考虑扩展 `AbstractDispatcherServletInitializer`，或者更好的 `AbstractAnnotationConfigDispatcherServletInitializer`，它可以自动设置这些选项，并且可以很容易地注册 `Filter` 实例。

## Spring MVC 配置

MVC Java 配置和 MVC 命名空间提供了用于配置异步请求处理的选项。 WebMvcConfigurer 具有 `configureAsyncSupport` 方法，而 `<mvc:annotation-driven>` 具有 `<async-support>` 子元素。

这些允许你配置默认的超时值用于异步请求，如果没有设置则取决于底层的 Servlet 容器（例如 Tomcat 上的 10 秒）。你也可以配置一个 `AsyncTaskExecutor` 来执行从控制器方法返回的 `callable` 实例。强烈建议配置此属性，因为默认情况下，Spring MVC 使用 `simpleAsyncTaskExecutor`。MVC Java 配置和 MVC 命名空间还允许您注册 `CallableProcessingInterceptor` 和 `DeferredResultProcessingInterceptor` 实例。

如果您需要覆盖特定 `DeferredResult` 的默认超时值，可以使用适当的类构造函数来完成。同样，对于 `Callable`，可以将其包装在 `WebAsyncTask` 中，并使用适当的类构造函数来自定义超时值。`WebAsyncTask` 的类构造函数也允许提供一个 `AsyncTaskExecutor`。

### 18.3.5 测试控制器

该 `spring-test` 模块提供一流的支持，用于测试带注解的控制器。参见第 11.6 节“[Spring MVC 测试框架](#)”。

## 18.4 处理程序映射

在以前的Spring版本中，用户需要在Web应用程序上下文中定义一个或多个 `HandlerMapping` bean，以将传入的Web请求映射到适当的处理程序。随着注解控制器的引入，您通常不需要这样做，因为 `RequestMappingHandlerMapping` 自动在所有 `@Controller` bean 上查找 `@RequestMapping` 注解。但请记住，从 `AbstractHandlerMapping` 扩展的所有 `HandlerMapping` 类都具有以下可用于自定义其行为的属性：

- `interceptors` 使用的拦截器列表。`HandlerInterceptor` 在 [Section 18.4.1, “Intercepting requests with a HandlerInterceptor”](#) 中讨论。
- `defaultHandler` 当此处理程序映射不会导致匹配处理程序时使用的缺省处理程序。
- `order` 基于 `order` 属性的值（请参阅 `org.springframework.core.Ordered` 接口），Spring 对上下文中可用的所有处理程序映射进行排序，并应用第一个匹配的处理程序。
- `alwaysUseFullPath` 如果为 `true`，则 Spring 将使用当前 Servlet 上下文中的完整路径来查找合适的处理程序。如果为 `false`（默认），则使用当前 Servlet 映射中的路径。例如，如果使用 `/testing/*` 映射 Servlet，并且 `alwaysUseFullPath` 属性设置为 `true`，则使用 `/testing/viewPage.html`，而如果该属性设置为 `false`，则使用 `/viewPage.html`。
- 从 Spring 2.5 开始，`urlDecode` 默认认为 `true`。如果您想比较编码路径，请将此标志设置为 `false`。但是，`HttpServletRequest` 总是以解码形式公开 Servlet 路径。请注意，与编码路径相比，Servlet 路径不匹配。

以下示例显示如何配置拦截器：

```
<beans>
    <bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping">
        <property name="interceptors">
            <bean class="example.MyInterceptor"/>
        </property>
    </bean>
</beans>
```

## 18.4.1 用 HandlerInterceptor 拦截请求

Spring 的处理程序映射机制包括处理程序拦截器，当您想要将特定的功能应用于某些请求时（例如，检查主体），这是非常有用的。

位于处理程序映射中的拦截器必须从 `org.springframework.web.servlet` 包中实现 `HandlerInterceptor`。这个接口定义了三个方法：`preHandle(..)` 在实际的处理器被执行之前被调用；`postHandle(..)` 被调用后，处理程序被执行；并且在完成请求完成之后调用 `afterCompletion(..)`。这三种方法应该提供足够的灵活性来进行各种预处理和后处理。

`preHandle(..)` 方法返回一个布尔值。您可以使用此方法来中断或继续处理执行链。当此方法返回 `true` 时，处理程序执行链将继续；当它返回 `false` 时，`DispatcherServlet` 假定拦截器本身已经处理了请求（并且例如呈现了适当的视图）并且不继续执行执行链中的其他拦截器和实际处理器。

拦截器可以使用 `interceptors` 属性进行配置，拦截器属性存在于从 `AbstractHandlerMapping` 扩展的所有 `HandlerMapping` 类中。这在下面的例子中显示：

```
<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;
        }
        response.sendRedirect("http://host.com/outsideOfficeHours.html");
        return false;
    }
}

```

由此映射处理的任何请求都被 `TimeBasedAccessInterceptor` 拦截。如果当前时间在办公时间之外，用户被重定向到一个静态的HTML文件，比如说你只能在办公时间访问网站。



当使用 `RequestMappingHandlerMapping` 时，实际的处理程序是 `HandlerMethod` 的一个实例，它标识了将被调用的特定控制器方法。

正如你所看到的，Spring适配器类 `HandlerInterceptorAdapter` 使扩展 `HandlerInterceptor` 面更容易。



在上面的示例中，配置的拦截器将应用于使用带注解的控制器方法处理的所有请求。如果要缩小拦截器所应用的URL路径，可以使用MVC名称空间或MVC Java配置，或者声明 `MappedInterceptor` 类型的bean实例来执行此操作。请参见[Section 18.16.1, “Enabling the MVC Java Config or the MVC XML Namespace”](#)。

请注意，`HandlerInterceptor` 的 `postHandle` 方法并不总是非常适合与 `@ResponseBody` 和 `ResponseEntity` 方法一起使用。在这种情况下 `HttpMessageConverter` 会在调用 `postHandle` 之前写入并提交响应，从而无法更改响应，例如添加标题。相反，应用程

序可以实现 `ResponseBodyAdvice`，并将其声明为 `@ControllerAdvice bean`，或直接在 `RequestMappingHandlerAdapter` 上进行配置。

## 18.5 解决观点

所有用于Web应用程序的MVC框架都提供了一种处理视图的方法。Spring提供了视图解析器，它使您能够在浏览器中渲染模型，而不需要将您视为特定的视图技术。开箱即用，例如，Spring允许您使用JSP，FreeMarker模板和XSLT视图。请参见第19章，查看技术对于如何整合并使用不同的视图技术的讨论。

对于Spring处理视图的方式来说，两个重要的接口是 `ViewResolver` 和 `View`。

`ViewResolver` 提供了视图名称和实际视图之间的映射。`View` 接口处理请求的准备，并将请求交给其中一种视图技术。

## 18.5.1 使用 ViewResolver 界面解析视图

如第18.3节，“**实施控制器**”中所讨论的，Spring Web MVC控制器中的所有处理器方法必须明确地（例如，通过返回 `String`，`View` 或 `ModelAndView`）或隐式地（即基于惯例）。Spring 中的视图由逻辑视图名称来解决，并由视图解析器解决。Spring有不少解析器。这个表格列出了其中的大部分；下面有几个例子。

**Table18.3.** 查看解析器

视图解析器	描述
<code>AbstractCachingViewResolver</code>	抽象视图解析器缓存视图。通常情况下，需要准备才能使用；扩展此视图解析器提供缓存。
<code>XmlViewResolver</code>	<code>ViewResolver</code> 的实现，它接受一个用 XML 编写的配置文件，其中使用与 Spring 的 XML bean 工厂相同的 DTD。默认配置文件是 <code>/WEB-INF/views.xml</code> 。
<code>ResourceBundleViewResolver</code>	<code>ViewResolver</code> 的实现，它使用由 <code>bundle</code> 基名指定的 <code>ResourceBundle</code> 中的 <code>bean</code> 定义。通常，您可以在位于类路径中的属性文件中定义该包。默认文件名是 <code>views.properties</code> 。
<code>UrlBasedViewResolver</code>	<code>ViewResolver</code> 接口的简单实现，它实现了逻辑视图名称直接解析到 URL，而没有明确的映射定义。如果您的逻辑名称以直观的方式与您的视图资源的名称相匹配，而不需要任意映射，则这是适当的。
<code>InternalResourceViewResolver</code>	<code>UrlBasedViewResolver</code> 的便捷子类支持 <code>InternalResourceView</code> （实际上是 Servlet 和 JSP）和子类（如 <code>JstlView</code> 和 <code>TilesView</code> ）。您可以使用 <code>setViewClass(...)</code> 为由此解析器生成的所有视图指定视图类。有关详细信息，请参阅 <code>UrlBasedViewResolver</code> javadocs。
<code>FreeMarkerViewResolver</code>	<code>UrlBasedViewResolver</code> 方便的子类，支持 <code>FreeMarkerView</code> 和它们的自定义子类。
<code>ContentNegotiatingViewResolver</code>	实现基于请求文件名或 <code>Accept</code> 头来解析视图的 <code>ViewResolver</code> 接口。请参见第18.5.4节“ <code>ContentNegotiatingViewResolver</code> ”。

例如，使用 JSP 作为视图技术，可以使用 `UrlBasedViewResolver`。此视图解析器将视图名称转换为 URL，并将请求转交给 `RequestDispatcher` 以呈现视图。

```
<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>
```

当将 `test` 作为逻辑视图名称返回时，此视图解析器将请求转发给 `RequestDispatcher`，`RequestDispatcher` 将请求发送到 `/WEB-INF/jsp/test.jsp`。

当您在 Web 应用程序中结合不同的视图技术时，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>
```

`ResourceBundleViewResolver` 检查由基本名称标识的 `ResourceBundle`，对于它应该解析的每个视图，它使用属性 `[viewname].(class)` 作为视图类的值和属性 `[viewname].url` 的值作为视图 URL。范例可以在下一章讨论视图技术。正如你所看到的，你可以识别一个父视图，属性文件中的所有视图都从其中“扩展”。例如，这样你可以指定一个默认的视图类。



`AbstractCachingViewResolver` 的子类缓存查看它们解析的实例。缓存提高了某些视图技术的性能。可以通过将 `cache` 属性设置为 `false` 来关闭缓存。此外，如果您必须在运行时刷新某个视图（例如，在修改 FreeMarker 模板时），则可以使用 `removeFromCache(String viewName, Locale loc)` 方法。

## 18.5.2 链接视图解析器

Spring 支持多个视图解析器。因此，您可以链接解析器，并在某些情况下覆盖特定的视图。您可以通过向应用程序上下文中添加多个解析器来链接视图解析器，并在必要时通过设置 `order` 属性来指定排序。请记住，订单属性越高，视图解析器在链中的位置越晚。

在以下示例中，视图解析器链由两个解析器组成：一个 `InternalResourceViewResolver`（始终自动定位为链中的最后一个解析器）以及一个用于指定 Excel 视图的 `XmlViewResolver`。Excel 视图不受 `InternalResourceViewResolver` 支持。

```
<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没有产生视图，Spring 会检查其他视图解析器的上下文。如果存在额外的视图解析器，Spring 将继续检查它们，直到视图解决。如果没有视图解析器返回一个视图，Spring 会抛出一个 `ServletException` 异常。

视图解析器的合约指定一个视图 `resolver_can_return` 为 `null` 以指示视图找不到。然而并不是所有的视图解析器都这样做，因为在某些情况下，解析器根本无法检测视图是否存在。例如，`InternalResourceViewResolver` 在内部使用 `RequestDispatcher`，调度是确定 JSP 是否存在的唯一方法，但此操作只能执行一次。`FreeMarkerViewResolver` 和其他的一样。检查特定视图解析器的 `javadoc` 以查看是否报告不存在的视图。因此，将一个 `InternalResourceViewResolver` 放在链中的最后一个结果链中没有被完全检查，因为 `InternalResourceViewResolver` `will_always_return` 一个视图！

### 18.5.3 重定向到视图

如前所述，控制器通常返回一个逻辑视图名称，视图解析器解析为一个特定的视图技术。对于通过 Servlet 或 JSP 引擎处理的 JSP 等视图技术，此解决方案通常通过 `InternalResourceViewResolver` 和 `InternalResourceView` 的组合来处理，`InternalResourceView` 和 `InternalResourceView` 通过 Servlet API 的 `RequestDispatcher.forward(..)` 方法或 `RequestDispatcher.include()` 方法。对于其他视图技术，如 FreeMarker、XSLT 等，视图本身直接将内容写入响应流。

在呈现视图之前，有时需要发送 HTTP 重定向回客户端。例如，当使用 POST 数据调用一个控制器时，这是可取的，而响应实际上是对另一个控制器的委托（例如成功的表单提交）。在这种情况下，正常的内部转发意味着另一个控制器也会看到相同的 POST 数据，如果可能会将其与其他预期数据混淆，则这可能会造成问题。在显示结果之前执行重定向的另一个原因是消除了用户多次提交表单数据的可能性。在这种情况下，浏览器将首先发送一个初始 POST；它会接收到重定向到不同的 URL 的响应；最后浏览器会对重定向响应中的 URL 进行后续的 GET 操作。因此，从浏览器的角度来看，当前页面并不反映 POST 的结果，而是 GET 的结果。最终的效果是用户无法通过 POST 执行刷新而意外地重新发布相同的数据。刷新强制结果页面的 GET，而不是重新发送初始 POST 数据。

## RedirectView

作为控制器响应的结果，强制重定向的一种方法是让控制器创建并返回 Spring 的 `RedirectView` 实例。在这种情况下，`DispatcherServlet` 不使用普通视图解析机制。而是因为它已经被赋予了（重定向）视图，`DispatcherServlet` 只是指示视图来完成它的任务。`RedirectView` 依次调用 `HttpServletResponse.sendRedirect()` 向客户端浏览器发送 HTTP 重定向。

如果使用 `RedirectView` 并且视图是由控制器本身创建的，则建议您将重定向 URL 配置为注入控制器，以使其不会被烘焙到控制器中，而是在上下文中与视图名称一起配置。在一节“重定向：前缀”有利于这种脱钩。

### 将数据传递到重定向目标

默认情况下，所有的模型属性都被认为是作为重定向 URL 中的 URI 模板变量公开的。其余属性是原始类型或集合/基本类型数组，都自动附加为查询参数。

如果为重定向专门准备了模型实例，那么将基元类型属性附加为查询参数可能是期望的结果。但是，在带注解的控制器中，模型可能包含为渲染目的而添加的附加属性（例如，下拉字段值）。为了避免在 URL 中出现这样的属性，`@RequestMapping` 方法可以声明 `RedirectAttributes` 类型的参数，并使用它来指定可用于 `RedirectView` 的确切属性。如果方法确实重定向，则使用 `RedirectAttributes` 的内容。否则使用模型的内容。

如果为重定向准备了模型实例，则将原始类型属性作为查询参数附加可能是所需的结果。然而，在注解控制器中，模型可能包含为渲染目的添加的附加属性（例如下拉字段值）。为了避免这种属性出现在URL中的可能性，一种 `@RequestMapping` 方法可以声明一个类型的参数，`RedirectAttributes` 并使用它来指定可供使用的确切属性 `RedirectView`。如果方法重定向，则使用内容 `RedirectAttributes`。否则使用模型的内容。

`RequestMappingHandlerAdapter` 提供了一个名为 “`ignoreDefaultModelOnRedirect`” 的标志，可用于指示默认 `Model` 的内容，如果控制器方法重定向，则永远不要使用该模型。相反，控制器方法应声明 `RedirectAttributes` 类型的属性，或者如果不这样做，则不应将任何属性传递给 `RedirectView`。MVC命名空间和MVC Java配置都将此标志设置为 `false`，以保持向后兼容性。但是，对于新应用程序，我们建议将其设置为 `true`

请注意，当前请求中的URI模板变量在扩展重定向URL时自动可用，不需要通过 `Model` 或 `RedirectAttributes` 显式添加。例如：

```
@PostMapping("/files/{path}")
public String upload(...) {
    // ...
    return "redirect:files/{path}";
}
```

将数据传递到重定向目标的另一种方法是通过 `Flash` 属性。与其他重定向属性不同，`Flash` 属性保存在HTTP会话中（因此不会出现在URL中）。有关详细信息，请参见第18.6节“使用Flash属性”。

## 重定向：前缀

虽然使用 `RedirectView` 可以正常工作，但如果控制器本身创建 `RedirectView`，则不会避免控制器意识到重定向正在发生。这实在是不理想，让事情太紧密。控制器不应该在乎如何处理响应。一般来说，它只能在注入视图名称的情况下运行。

特殊的 `redirect:` 前缀可以让你做到这一点。如果返回具有前缀 `redirect:` 的视图名称，则 `UrlBasedViewResolver`（和所有子类）将认识到这是需要重定向的特殊指示。视图名称的其余部分将被视为重定向URL。

请注意，控制器处理程序使用注释 `@ResponseStatus`，注释值优先于设置的响应状态 `RedirectView`。

实际效果与控制器返回 `RedirectView` 的效果相同，但现在控制器本身可以简单地按照逻辑视图名称操作。逻辑视图名称，例如 `redirect:/myapp/some/resource` 将相对于当前Servlet上下文重定向，而诸如 `redirect:http://myhost.com/some/arbitrary/path` 之类的名称将重定向到绝对URL。

请注意，控制器处理程序使用 `@ResponseStatus` 注解，注解值优先于 `RedirectView` 设置的响应状态。

### 转发：前缀

对于最终由 `UrlBasedViewResolver` 和子类解析的视图名称，也可以使用特殊的 `forward:` 前缀。这将创建一个内部 `ResourceView`（最终执行一个 `RequestDispatcher.forward()`），其余的视图名称被视为一个URL。因此，这个前缀对

于 `InternalResourceViewResolver` 和 `InternalResourceView`（例如JSP）是没有用的。但是，当您主要使用其他视图技术时，前缀可能会很有帮助，但仍希望强制转发资源，以便由 `Servlet/JSP` 引擎处理。（请注意，您也可以链接多个视图解析器）。

与 `redirect:` 前缀一样，如果带有 `forward:` 前缀的视图名称被注入到控制器中，则控制器不会检测到处理响应方面的任何特殊情况。

## 18.5.4 ContentNegotiatingViewResolver

`ContentNegotiatingViewResolver` 不会自己解析视图，而是委托给其他视图解析器，选择类似于客户端请求的视图。有两种策略可以让客户从服务器请求表示：

- 对每个资源使用不同的URI，通常在URI中使用不同的文件扩展名。例如，URI  
`http://www.example.com/users/fred.pdf` 请求用户fred的PDF表示，而  
`http://www.example.com/users/fred.xml` 请求XML表示。
- 为客户端使用相同的URI来定位资源，但是设置 `Accept` HTTP请求头来列出它理解的媒体类型。例如，`http://www.example.com/users/fred` 的一个HTTP请求的 `Accept` 标头设置为 `application/pdf`，请求用户fred的PDF表示，同时 `http://www.example.com/users/fred` 使用 `Accept` 头设置来 `text/xml` 请求XML表示。这个策略被称为内容谈判。



`Accept` 头的一个问题是，不可能在HTML中的Web浏览器中设置它。例如，在Firefox中，它被固定为：`Accept:`

`text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8` 因此，通常会看到使用开发基于浏览器的Web应用程序时，每个表示的URI都是不同的。

为了支持资源的多重表示，Spring提供了 `ContentNegotiatingViewResolver` 来根据HTTP请求的文件扩展名或者 `Accept` 头来解析一个视图。`ContentNegotiatingViewResolver` 本身并不执行视图解析，而是委托给你指定的视图解析器列表 `bean`属性 `viewResolvers`。

`ContentNegotiatingViewResolver` 通过将请求媒体类型与 `ViewResolver` 关联的 `view` 所支持的媒体类型（也称为 `Content-Type`）进行比较来选择适当的 `view` 来处理请求。具有兼容的 `Content-Type` 的列表中的第一个 `view` 将表示返回给客户端。如果 `ViewResolver` 链无法提供兼容的视图，则会查看通过 `DefaultViews` 属性指定的视图列表。后面的选项适用于可以呈现当前资源的适当表示的单例视图，而不管逻辑视图的名称如何。`Accept` 头可能包含通配符，例如`text/*`，在这种情况下，`Content-Type`为 `text/xml` 的 `view` 是兼容的匹配。

要支持基于文件扩展名的视图的自定义分辨率，请使用 `ContentNegotiationManager`：请参见第18.16.6节“内容协商”。

以下是 `ContentNegotiatingViewResolver` 的配置示例：

```

<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
    <property name="viewResolvers">
        <list>
            <bean class="org.springframework.web.servlet.view.BeanNameViewResolver"/>
            <bean class="org.springframework.web.servlet.view.InternalResourceViewReso
lver">
                <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.MappingJackson2Json
View"/>
        </list>
    </property>
</bean>

<bean id="content" class="com.foo.samples.rest.SampleContentAtomView"/>

```

`InternalResourceViewResolver` 处理视图名称和JSP页面的翻译，而 `BeanNameViewResolver` 则根据bean的名称返回一个视图。（有关Spring如何查找和实例化视图的更多详细信息，请参阅“[使用ViewResolver界面解析视图](#)”。）在本例中，内容bean是从 `AbstractAtomFeedView` 继承的类，它返回Atom RSS提要。有关创建Atom Feed表示的更多信息，请参阅Atom Views部分。

In the above configuration, if a request is made with an `.html` extension, the view resolver looks for a view that matches the `text/html` media type.

The `InternalResourceViewResolver` provides the matching view for `text/html`. If the request is made with the file extension `.atom`, the view resolver looks for a view that matches the `application/atom+xml` media type. This view is provided by the `BeanNameViewResolver` that maps to the `SampleContentAtomView` if the view name returned is `content`. If the request is made with the file extension `.json`, the `MappingJackson2JsonView` instance from the `DefaultViews` list will be selected regardless of the view name. Alternatively, client requests can be made without a file extension but with the `Accept` header set to the preferred media-type, and the same resolution of request to views would occur.

在上面的配置中，如果使用 `.html` 扩展名进行请求，则视图解析器将查找与 `text/html` 媒体类型相匹配的视图。`InternalResourceViewResolver` 提供了匹配的视图 `fortext/html`。如果请求是使用文件扩展名 `.atom` 创建的，则视图解析器会查找与 `application/atom+xml` 媒体类型相匹配的视图。如果返回的视图名称是内容，则此视图由映射到 `SampleContentAtomView` 的 `BeanNameViewResolver` 提供。如果请求是使用文件扩展

名 `.json` 创建的，则无论视图名称如何，都会选择 `DefaultViews` 列表中的 `MappingJackson2JsonView` 实例。或者，客户端请求可以在没有文件扩展名的情况下进行，但将 `Accept` 头设置为首选的媒体类型，并且对视图的请求的分辨率也会发生。



如果“ContentNegotiatingViewResolver”的ViewResolver列表未被明确配置，它会自动使用应用程序上下文中定义的任何ViewResolvers。

下面显示了相应的控制器代码，该代码返回Atom RSS提要，其格式为 `http://localhost/content.atom` 或 `http://localhost/content`，并带有 `application/atom+xml` 的 `Accept` 头。

```
@Controller
public class ContentController {

    private List<SampleContent> contentList = new ArrayList<SampleContent>();

    @GetMapping("/content")
    public ModelAndView getContent() {
        ModelAndView mav = new ModelAndView();
        mav.setViewName("content");
        mav.addObject("sampleContentList", contentList);
        return mav;
    }

}
```

## 18.6 使用flash属性

Flash属性提供了一种请求来存储供另一个使用的属性的方法。重定向时这是最常用的，例如，Post/Redirect/Get模式。在重定向（通常在会话中）之前，Flash属性会临时保存，以便在重定向之后立即将其删除。

Spring MVC有两个支持flash属性的主要抽象。`FlashMap` 用于保存Flash属性，而 `FlashMapManager` 用于存储，检索和管理 `FlashMap` 实例。

Flash属性支持始终处于“打开”状态，不需要显式启用即使未使用也不会导致创建HTTP会话。在每个请求中，都有一个“输入” `FlashMap`，其中包含从前一个请求（如果有）传递的属性和具有属性的“输出” `FlashMap`，用于保存后续请求。两个 `FlashMap` 实例都可以通过 `RequestContextUtils` 中的静态方法从Spring MVC中的任何地方访问。

带注解的控制器通常不需要直接使用 `FlashMap`。相反，`@RequestMapping` 方法可以接受类型为 `RedirectAttributes` 的参数，并使用它为重定向场景添加Flash属性。通过 `RedirectAttributes` 添加的 Flash 属性会自动传播到“输出” `FlashMap`。同样，在重定向之后，来自“输入” `FlashMap` 的属性将自动添加到提供目标URL的控制器的 `Model` 中。

### 匹配请求到Flash属性

Flash属性的概念存在于许多其他Web框架中，并且已经被证明有时会暴露给并发问题。这是因为根据定义，闪存属性将被存储直到下一个请求。然而，“下一个”请求可能不是预期的接收者，而是另一个异步请求（例如轮询或资源请求），在这种情况下，flash属性过早地被删除。

为了减少这种问题的可能性，`RedirectView` 使用目标重定向URL的路径和查询参数自动“标记” `FlashMap` 实例。在查找“输入” `FlashMap` 时，默认的 `FlashMapManager` 将该信息与传入的请求进行匹配。

这不能完全消除并发问题的可能性，但是仍然可以通过重定向URL中已经提供的信息来大大减少它。因此，使用Flash属性主要用于重定向方案。

## 18.7 构建URI

Spring MVC提供了一种使用 `UriComponentsBuilder` 和 `UriComponents` 构建和编码URI的机制。

例如，您可以扩展和编码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();
```

请注意，`UriComponents`是不可变的，如果需要，`expand()` 和 `encode()` 操作将返回新的实例。

您还可以使用各个URI组件进行扩展和编码：

```
UriComponents uriComponents = UriComponentsBuilder.newInstance()
    .scheme("http").host("example.com").path("/hotels/{hotel}/bookings/{booking}")
.build()
    .expand("42", "21")
    .encode();
```

在Servlet环境中，`ServletUriComponentsBuilder` 子类提供静态工厂方法从Servlet请求中复制可用的URL信息：

```
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，您可以选择将可用信息的一个子集复制到上下文路径（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()
```

或者在 `DispatcherServlet` 按名称映射（例如 `/main/*`）的情况下，还可以包含 `servlet` 映射的文字部分：

```
// 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()
```

## 18.7.1 构建控制器和方法的URI

Spring MVC还提供了一种构建控制器方法链接的机制。例如，给出：

```
@Controller
@RequestMapping("/hotels/{hotel}")
public class BookingController {

    @GetMapping("/bookings/{booking}")
    public String getBooking(@PathVariable Long booking) {

        // ...
    }
}
```

您可以通过名称参考方法来准备链接：

```
UriComponents uriComponents = MvcUriComponentsBuilder
    .fromMethodName(BookingController.class, "getBooking", 21).buildAndExpand(42);

URI uri = uriComponents.encode().toUri();
```

在上面的例子中，我们提供了实际的方法参数值，在这种情况下是long值21，被用作路径变量并被插入到URL中。此外，我们提供了值42以填充任何剩余的URI变量，例如从类型级别请求映射继承的“hotel”变量。如果方法有更多的参数，你可以为URL提供不需要的参数。通常，只有 `@PathVariable` 和 `@RequestParam` 参数与构造URL相关。

还有其他方法可以使用 `MvcUriComponentsBuilder`。例如，您可以使用类似于通过代理进行模拟测试的技术，以避免通过名称引用控制器方法（该示例假定为 `MvcUriComponentsBuilder.on` 的静态导入）：

```
UriComponents uriComponents = MvcUriComponentsBuilder
    .fromMethodCall(on(BookingController.class).getBooking(21)).buildAndExpand(42);

URI uri = uriComponents.encode().toUri();
```

上面的例子在 `MvcUriComponentsBuilder` 中使用静态方法。在内部，他们依靠 `ServletUriComponentsBuilder` 从当前请求的方案，主机，端口，上下文路径和servlet路径中准备基本URL。这在大多数情况下运作良好，但有时可能不足。例如，您可能在请求的上下文之外（例如，准备链接的批处理过程），或者您可能需要插入路径前缀（例如，从请求路径中删除并重新插入链接的区域设置前缀）。

对于这种情况，您可以使用接受 `UriComponentsBuilder` 的静态“`fromXxx`”重载方法来使用基本 URL。或者，您可以使用基本URL创建 `MvcUriComponentsBuilder` 实例，然后使用基于实例的“`withXxx`”方法。例如：

```
UriComponentsBuilder base = ServletUriComponentsBuilder.fromCurrentContextPath().path("/en");
MvcUriComponentsBuilder builder = MvcUriComponentsBuilder.relativeTo(base);
builder.withMethodCall(on(BookingController.class).getBooking(21)).buildAndExpand(42);

URI uri = uriComponents.encode().toUri();
```

## 18.7.2 构建来自视图的控制器和方法的URI

您还可以从视图（如JSP，Thymeleaf，FreeMarker）建立到注解控制器的链接。这可以使用 `MvcUriComponentsBuilder` 中的 `fromMappingName` 方法来完成，该方法引用按名称映射。

每个 `@RequestMapping` 被分配一个基于类的大写字母和完整的方法名称的默认名称。例如，类 `FooController` 中的方法 `getFoo` 被分配名称“FC#`getFoo`”。可以通过创建 `HandlerMethodMappingNamingStrategy` 实例并将其插入到 `RequestMappingHandlerMapping` 中来替换或定制此策略。默认策略实现也查看 `@RequestMapping` 上的名称属性，并使用它（如果存在）。这意味着，如果分配的默认映射名称与另一个映射名称（例如重载的方法）冲突，则可以在 `@RequestMapping` 上明确指定一个名称。



分配的请求映射名称在启动时将记录在TRACE级别。

Spring JSP标记库提供了一个名为 `mvcUrl` 的函数，可用于根据此机制来准备控制器方法的链接。

例如给出：

```
@RequestMapping("/people/{id}/addresses")
public class PersonAddressController {

    @RequestMapping("/{country}")
    public ResponseEntity getAddress(@PathVariable String country) { ... }
}
```

您可以从JSP准备一个链接，如下所示：

```
<%@ taglib uri="http://www.springframework.org/tags" prefix="s" %>
...
<a href="${s:mvcUrl('PAC#getAddress').arg(0, 'US').buildAndExpand('123')}">Get Address</a>
```

上面的例子依赖于Spring标记库中声明的 `mvcUrl` JSP函数（即META-INF / `spring.tld`）。对于更高级的情况（例如上一节中介绍的自定义基本URL），可以很容易地定义自己的函数或使用自定义标记文件，以便使用具有自定义基本URL的特定 `MvcUriComponentsBuilder` 实例。

## 18.8 使用区域设置

Spring体系结构的大部分支持国际化，就像Spring web MVC框架一。`DispatcherServlet` 使您能够使用客户端的语言环境自动解析消息。这是用 `LocaleResolver` 对象完成的。

当请求进入时，`DispatcherServlet` 会查找一个语言环境解析器，如果找到，它会尝试使用它来设置语言环境。使用 `RequestContext.getLocale()` 方法，您始终可以检索由区域设置解析程序解析的区域设置。

除了自动区域设置解析之外，您还可以将拦截器附加到处理程序映射（有关处理程序映射拦截器的更多信息，请参见[第18.4.1节“拦截与HandlerInterceptor的请求”](#)），以在特定情况下更改区域设置，例如，基于在请求中的一个参数。

区域设置解析器和拦截器在 `org.springframework.web.servlet.i18n` 包中定义，并以常规方式在应用程序上下文中配置。这里是Spring中包含的一个语言环境解析器的选择。

## 18.8.1 获取时区信息

除了获取客户端的语言环境之外，了解他们的时区通常也很有用。`LocaleContextResolver` 接口为 `LocaleResolver` 提供了一个扩展，它允许解析器提供更丰富的 `LocaleContext`，其中可能包含时区信息。

如果可用，用户的 `TimeZone` 可以使用 `RequestContext.getTimeZone()` 方法获得。时区信息将自动被 Spring 的 `ConversionService` 注册的日期/时间 `converter` 和 `Formatter` 对象使用。

## 18.8.2 AcceptHeaderLocaleResolver

此语言环境解析程序检查客户端（例如，Web浏览器）发送的请求中的 `accept-language` 标头。通常这个头域包含客户端操作系统的语言环境。请注意，此解析器不支持时区信息。

### 18.8.3 CookieLocaleResolver

此语言环境解析程序将检查客户端上可能存在的 `Cookie`，以查看是否指定了 `Locale` 或 `TimeZone`。如果是这样，它使用指定的细节。使用此语言环境解析器的属性，您可以指定 `cookie` 的名称以及最大年龄。下面找到一个定义 `CookieLocaleResolver` 的例子。

```
<bean id="localeResolver" class="org.springframework.web.servlet.i18n.CookieLocaleResolver">

    <property name="cookieName" value="clientlanguage"/>

    <!-- 以秒为单位。如果设置为 -1，则 cookie 不会持久化（在浏览器关闭时删除）-->
    <property name="cookieMaxAge" value="100000"/>

</bean>
```

**Table 18.4. CookieLocaleResolver 属性**

属性	默认	描述
cookieName	classname + LOCALE	cookie 的名称
cookieMaxAge	Servlet 容器默认	Cookie 在客户端上保持持续的最长时间。如果指定了 -1，则 cookie 不会被持久化；只有客户端关闭浏览器才可用。
cookiePath	/	将 Cookie 的可见性限制在您网站的某个部分。指定 cookiePath 时，cookie 将只对该路径及其下方的路径可见。.

## 18.8.4 SessionLocaleResolver

`SessionLocaleResolver` 允许您从可能与用户请求关联的会话中检索 `Locale` 和 `TimeZone`。与 `CookieLocaleResolver` 相比，这种策略将本地选择的区域设置存储在 `Servlet` 容器的 `HttpSession` 中。因此，这些设置对于每个会话都是临时的，因此在每个会话终止时都会丢失。

请注意，与 `Spring Session` 项目等外部会话管理机制没有直接关系。这个 `SessionLocaleResolver` 将简单地根据当前的 `HttpServletRequest` 评估和修改相应的 `HttpSession` 属性。

## 18.8.5 LocaleChangeInterceptor

You can enable changing of locales by adding the `LocaleChangeInterceptor` to one of the handler mappings (see [Section 18.4, “Handler mappings”](#)). It will detect a parameter in the request and change the locale. It calls `setLocale()` on the `LocaleResolver` that also exists in the context. The following example shows that calls to all `*.view` resources containing a parameter named `siteLanguage` will now change the locale. So, for example, a request for the following URL, `http://www.sf.net/home.view?siteLanguage=nl` will change the site language to Dutch.

您可以通过将 `LocaleChangeInterceptor` 添加到其中一个处理器映射（请参见 [第 18.4 节“处理器映射”](#)）来启用区域设置的更改。它将检测请求中的参数并更改语言环境。它调用上下文中也存在的 `LocaleResolver` 上的 `setLocale()`。以下示例显示调用包含名为 `siteLanguage` 的参数的所有 `*.view` 资源现在将更改语言环境。因此，例如，对以下 URL（`http://www.sf.net/home.view?siteLanguage=nl`）的请求会将网站语言更改为荷兰语。

```
<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>
```



### 18.9.1 主题概述

您可以应用 Spring Web MVC 框架主题来设置应用程序的整体外观，从而提高用户体验。主题是静态资源的集合，通常是样式表和图像，它们会影响应用程序的视觉风格。

## 18.9.2 定义主题

要在您的Web应用程序中使用主题，您必须设

置 `org.springframework.ui.context.ThemeSource` 接口的实现。`WebApplicationContext` 接口扩展了 `ThemeSource`，但将其职责委托给专用的实现。默认情况下，委托将是一个 `org.springframework.ui.context.support.ResourceBundleThemeSource` 实现，它从类路径的根目录加载属性文件。要使用自定义的 `ThemeSource` 实现或配置 `ResourceBundleThemeSource` 的基本名称前缀，可以在应用程序上下文中使用保留名称 `themeSource` 注册一个bean。Web应用程序上下文自动检测具有该名称的bean并使用它。

使用 `ResourceBundleThemeSource` 时，主题是在一个简单的属性文件中定义的。属性文件列出组成主题的资源。这里是一个例子：

```
styleSheet=/themes/cool/style.css
background=/themes/cool/img/coolBg.jpg
```

属性的键是从视图代码引用主题元素的名称。对于JSP，通常使用 `spring:theme` 自定义标签来实现，这与 `spring:message` 标签非常相似。以下JSP片段使用前面示例中定义的主题来自定义外观：

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<html>
    <head>
        <link rel="stylesheet" href="
    </head>
    <body style="background=<spring:theme code='background' />">
        ...
    </body>
</html>
```

默认情况下，`ResourceBundleThemeSource` 使用空的基本名称前缀。因此，属性文件是从类路径的根目录加载的。因此，您可以将 `cool.properties` 主题定义放在类路径根目录中，例如 `/WEB-INF/classes` 中。`ResourceBundleThemeSource` 使用标准的Java资源包加载机制，允许主题完全国际化。例如，我们可以有一个 `/WEB-INF/classes/cool_nl.properties` 引用一个带有荷兰文字的特殊背景图片。

### 18.9.3 主题解析器

在定义主题之后，如上一节所述，决定使用哪个主题。`DispatcherServlet` 将查找一个名为 `themeResolver` 的 bean 来找出使用哪个 `ThemeResolver` 实现。一个主题解析器的工作方式与 `LocaleResolver` 非常相似。它检测用于特定请求的主题，也可以更改请求的主题。Spring 提供以下主题解析器：

**Table 18.5. ThemeResolver 实现**

类	描述
<code>FixedThemeResolver</code>	选择一个固定的主题，使用 <code>defaultThemeName</code> 属性设置。
<code>SessionThemeResolver</code>	主题维护在用户的 HTTP 会话中。它只需要为每个会话设置一次，但不会在会话之间持久化。
<code>CookieThemeResolver</code>	所选主题存储在客户端的 cookie 中。

Spring 还提供了一个 `ThemeChangeInterceptor`，它允许使用一个简单的请求参数来更改每个请求的主题。



## 18.10.1 介绍

Spring的内置多部分支持处理Web应用程序中的文件上传。您可以使用 `org.springframework.web.multipart` 包中定义的可插入 `MultipartResolver` 对象来启用此多部分支持。Spring提供了一个 `MultipartResolver` 实现，用于与[Commons FileUpload](#)的实现一起使用，另一个用于Servlet 3.0多部分请求分析。

默认情况下，Spring没有多部分处理，因为有些开发人员想要自己处理多部分。通过将多部分解析器添加到Web应用程序的上下文中来启用Spring多部分处理。检查每个请求以查看它是否包含多部分。如果找不到多部分，请求将按预期继续。如果在请求中找到`multipart`，则使用在上下文中声明的 `MultipartResolver`。之后，请求中的多部分属性就像任何其他属性一样处理。

## 18.10.2与 Commons FileUpload一起使用 MultipartResolver

以下示例显示如何使用 CommonsMultipartResolver :

```
<bean id="multipartResolver"
      class="org.springframework.web.multipart.commons.CommonsMultipartResolver">

    <!-- 可用属性之一；最大文件大小（以字节为单位）-->
    <property name="maxUploadSize" value="100000"/>

</bean>
```

当然，你也需要在你的类路径中放入适当的jar，以便多部分解析器工作。

在 CommonsMultipartResolver 的情况下，你需要使用 commons-fileupload.jar 。

当 Spring DispatcherServlet 检测到多部分请求时，它会激活已经在上下文中声明的解析器并移交请求。然后，解析器将当前的 HttpServletRequest 包装成支持多部分文件上传的 MultipartHttpServletRequest 。使用 MultipartHttpServletRequest ，您可以获得有关此请求包含的多部分的信息，实际上可以在您的控制器中访问多部分文件。

### 18.10.3 在Servlet 3.0中使用 MultipartResolver

为了使用基于Servlet 3.0的多部分解析，您需要在 `web.xml` 中使用 “`multipart-config`” 部分标记 `DispatcherServlet`，或者使用编程Servlet注册中的 `javax.servlet.MultipartConfigElement` 标记，或者使用自定义的Servlet类可能在你的Servlet类中有一个 `javax.servlet.annotation.MultipartConfig` 注解。配置设置（如最大大小或存储位置）需要在该Servlet注册级别应用，因为Servlet 3.0不允许从 `MultipartResolver` 完成这些设置。

一旦使用上述方式之一启用了Servlet 3.0 multipart解析，您可以将 `StandardServletMultipartResolver` 添加到您的Spring配置中：

```
<bean id="multipartResolver"
      class="org.springframework.web.multipart.support.StandardServletMultipartResol
ver">
</bean>
```

## 18.10.4 处理表单中的文件上传

在 `MultipartResolver` 完成其作业之后，请求将像其他处理一样进行处理。首先，创建一个带有文件输入的表单，允许用户上传表单。编码属性 (`enctype = "multipart/form-data"`) 让浏览器知道如何将表单编码为多部分请求：

```
<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>
```

下一步是创建一个处理文件上传的控制器。除了我们在方法参数中使用 `MultipartHttpServletRequest` 或 `MultipartFile` 之外，这个控制器与标准的 `@Controller` 非常相似：

```
@Controller
public class FileUploadController {

    @PostMapping("/form")
    public String handleFormUpload(@RequestParam("name") String name,
                                   @RequestParam("file") MultipartFile file) {

        if (!file.isEmpty()) {
            byte[] bytes = file.getBytes();
            // 将字节存储在某个地方
            return "redirect:uploadSuccess";
        }

        return "redirect:uploadFailure";
    }
}
```

请注意，`@RequestParam` 方法参数如何映射到表单中声明的输入元素。在这个例子中，`byte[]` 没有做任何事情，但实际上你可以将它保存在数据库中，将它存储在文件系统中，等等。

使用Servlet 3.0多部分解析时，还可以使用 `javax.servlet.http.Part` 作为方法参数：

```
@Controller
public class FileUploadController {

    @PostMapping("/form")
    public String handleFormUpload(@RequestParam("name") String name,
        @RequestParam("file") Part file) {

        InputStream inputStream = file.getInputStream();
        // 从上传的文件中存储字节
        return "redirect:uploadSuccess";
    }

}
```

## 18.10.5 处理来自编程客户端的文件上传请求

也可以从RESTful服务场景中的非浏览器客户端提交多部分请求。以上所有示例和配置都适用于此。但是，与通常提交文件和简单表单域的浏览器不同，编程客户机还可以发送更复杂的特定内容类型的数据 - 例如带文件的多部分请求，带JSON格式数据的第二部分：

```
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 ...
```

You could access the part named "meta-data" with a `@RequestParam("meta-data") String metadata` controller method argument. However, you would probably prefer to accept a strongly typed object initialized from the JSON formatted data in the body of the request part, very similar to the way `@RequestBody` converts the body of a non-multipart request to a target object with the help of an `HttpMessageConverter`.

You can use the `@RequestPart` annotation instead of the `@RequestParam` annotation for this purpose. It allows you to have the content of a specific multipart passed through an `HttpMessageConverter` taking into consideration the 'Content-Type' header of the multipart:

您可以使用 `@RequestParam("meta-data") String metadata` 控制器方法参数来访问名为"meta-data"的部分。但是，您可能更愿意接受一个从请求部分主体中的JSON格式的数据初始化的强类型对象，这与 `@RequestBody` 在非帮助的情况下将非多部分请求的主体转换为目标对象的方式非常相似一个 `HttpMessageConverter`。

您可以使用 `@RequestPart` 注解而不是 `@RequestParam` 注解来达到此目的。它允许您通过 `HttpMessageConverter` 考虑多部分的 'Content-Type' 标题来传递特定多部分的内容：

```
@PostMapping("/someUrl")
public String onSubmit(@RequestPart("meta-data") MetaData metadata,
                      @RequestPart("file-data") MultipartFile file) {
    // ...
}
```

请注意，`MultipartFile` 方法参数如何通过 `@RequestParam` 或 `@RequestPart` 可互换地访问。然而，在这种情况下，`@RequestPart("meta-data") MetaData` 方法参数被读取为基于其 `'Content-Type'` 头的JSON内容，并且借助于 `MappingJackson2HttpMessageConverter` 进行转换。



## 18.11.1 HandlerExceptionResolver

Spring `HandlerExceptionResolver` 实现处理控制器执行期间发生的意外异常。

`HandlerExceptionResolver` 有点类似于你可以在 web 应用程序描述符 `web.xml` 中定义的异常映射。但是，他们提供了一个更灵活的方式来做到这一点。例如，它们提供有关抛出异常时正在执行哪个处理器的信息。此外，处理异常的编程方式为您提供了更多的选择，可以在将请求转发到另一个 URL（与使用特定于 Servlet 的异常映射相同的最终结果）之前做出适当的响应。

除了实现 `HandlerExceptionResolver` 接口（这只是实现 `resolveException(Exception, Handler)` 方法并返回一个 `ModelAndView`）之外，还可以使用提供的 `SimpleMappingExceptionResolver` 或创建 `@ExceptionHandler` 方法。

`SimpleMappingExceptionResolver` 使您可以将可能引发的任何异常的类名称映射到视图名称。这在功能上等同于 Servlet API 的异常映射功能，但是也可以从不同的处理器实现更多细化的异常映射。另一方面，`@ExceptionHandler` 注解可用于应该被调用来处理异常的方法。这样的方法可以在 `@Controller` 中本地定义，也可以在 `@ControllerAdvice` 类中定义时应用于许多 `@Controller` 类。以下部分更详细地解释了这一点。

## 18.11.2 @ExceptionHandler

`HandlerExceptionResolver` 接口和 `SimpleMappingExceptionResolver` 实现允许您在转发到这些视图之前，将 `Exceptions` 和一些可选的 Java 逻辑声明性地映射到特定的视图。但是，在某些情况下，特别是依赖 `@ResponseBody` 方法而不是视图分辨率时，直接设置响应的状态并可选地将错误内容写入响应的主体可能更方便。

你可以用 `@ExceptionHandler` 方法来做到这一点。在控制器中声明时，这些方法适用于该控制器（或其任何子类）的 `@RequestMapping` 方法引发的异常。您也可以在 `@ControllerAdvice` 类中声明一个 `@ExceptionHandler` 方法，在这种情况下，它可以处理来自许多控制器的 `@RequestMapping` 方法的异常。下面是一个 controller-local `@ExceptionHandler` 方法的例子：

```
@Controller
public class SimpleController {

    // @RequestMapping methods omitted ...

    @ExceptionHandler(IOException.class)
    public ResponseEntity<String> handleIOException(IOException ex) {
        // prepare responseEntity
        return ResponseEntity;
    }

}
```

`@ExceptionHandler` 值可以设置为一个 `Exception` 类型的数组。如果抛出一个与列表中的某个类型匹配的异常，那么将调用用匹配的 `@ExceptionHandler` 注解的方法。如果未设置注释值，则使用列为方法参数的异常类型。

就像使用 `@RequestMapping` 注解标注的标准控制器方法一样，`@ExceptionHandler` 方法的方法参数和返回值可以是灵活的。例如，可以在 Servlet 环境中访问 `HttpServletRequest`。返回类型可以是 `String`，它被解释为一个视图名称，一个 `ModelAndView` 对象，一个 `ResponseEntity`，或者你也可以添加 `@ResponseBody` 方法返回值与消息转换器转换并写入响应流。

### 18.11.3 处理标准的Spring MVC 异常

在处理请求时，Spring MVC 可能会引发一些异常。`SimpleMappingExceptionResolver` 可以根据需要轻松地将任何异常映射到默认错误视图。但是，在使用以自动方式解释响应的客户端时，您需要在响应中设置特定的状态码。根据引发的异常，状态码可能表示客户端错误（4xx）或服务器错误（5xx）。

`DefaultHandlerExceptionResolver` 将 Spring MVC 异常转换为特定的错误状态代码。默认情况下，MVC 命名空间，MVC Java 配置以及 `DispatcherServlet`（即不使用 MVC 命名空间或 Java 配置时）都被注册。下面列出的是这个解析器处理的一些异常和相应状态码：

Exception	HTTP Status Code
<code>BindException</code>	400 (Bad Request)
<code>ConversionNotSupportedException</code>	500 (Internal Server Error)
<code>HttpMediaTypeNotAcceptableException</code>	406 (Not Acceptable)
<code>HttpMediaTypeNotSupportedException</code>	415 (Unsupported Media Type)
<code>HttpMessageNotReadableException</code>	400 (Bad Request)
<code>HttpMessageNotWritableException</code>	500 (Internal Server Error)
<code>HttpRequestMethodNotSupportedException</code>	405 (Method Not Allowed)
<code>MethodArgumentNotValidException</code>	400 (Bad Request)
<code>MissingPathVariableException</code>	500 (Internal Server Error)
<code>MissingServletRequestParameterException</code>	400 (Bad Request)
<code>MissingServletRequestPartException</code>	400 (Bad Request)
<code>NoHandlerFoundException</code>	404 (Not Found)
<code>NoSuchRequestHandlingMethodException</code>	404 (Not Found)
<code>TypeMismatchException</code>	400 (Bad Request)

`DefaultHandlerExceptionResolver` 通过设置响应的状态透明地工作。但是，当应用程序可能需要为每个错误响应添加对开发人员友好的内容（例如，提供 REST API 时）时，它不会将任何错误内容写入响应主体。您可以通过视图解析来准备模型和视图并呈现错误内容，即通过配置 `ContentNegotiatingViewResolver`，`MappingJackson2JsonView` 等。但是，您可能更喜欢使用 `@ExceptionHandler` 方法。

如果您更喜欢通过 `@ExceptionHandler` 方法编写错误内容，则可以扩展 `ResponseEntityExceptionHandler`。这是 `@ControllerAdvice` 类的便利基础，它提供了一个 `@ExceptionHandler` 方法来处理标准的 Spring MVC 异常并返回  `ResponseEntity`。这允许您

自定义响应并使用消息转换器编写错误内容。有关更多详细信息，请参阅 [ResponseEntityExceptionHandler javadocs](#)。

## 18.11.4 REST控制器异常处理

`@RestController` 可以使用 `@ExceptionHandler` 方法返回一个  `ResponseEntity`，在响应的主体中提供响应状态和错误细节。这些方法也可以添加到 `@ControllerAdvice` 类中，以处理子集或所有控制器的异常处理。

一个常见的要求是在响应的主体中包含错误细节。Spring不会自动执行此操作（虽然Spring Boot会这样做），因为响应主体中错误详细信息的表示形式是特定于应用程序的。

希望在响应主体中实现具有错误细节的全局异常处理策略的应用程序应考虑扩展抽象基类  `ResponseEntityExceptionHandler`，该类提供对Spring MVC引发的异常的处理，并提供挂钩来自定义响应主体以及处理其他异常。只需将扩展类声明为Spring bean，并使用 `@ControllerAdvice` 对其进行注释。有关更多详细信息，请参阅请参阅  `ResponseEntityExceptionHandler`。

### 18.11.5 使用 @ResponseStatus 注解业务异常

业务异常可以使用 `@ResponseStatus` 进行注释。当引发异常时，`ResponseStatusExceptionResolver` 通过相应地设置响应的状态来处理它。默认情况下，`DispatcherServlet` 注册 `ResponseStatusExceptionResolver`，并可供使用。

## 18.11.6 自定义默认Servlet容器错误页面

当响应的状态设置为错误状态码并且响应的主体为空时，Servlet容器通常会呈现HTML格式的错误页面。要定制容器的默认错误页面，可以在 `web.xml` 中声明一个 `<error-page>`。直到 Servlet 3，该元素必须映射到特定的状态码或异常类型。从Servlet 3开始，不需要映射错误页面，这实际上意味着指定的位置定制了默认的Servlet容器错误页面。

```
<error-page>
    <location>/error</location>
</error-page>
```

请注意，错误页面的实际位置可以是JSP页面或容器中的其他URL，包括通过 `@Controller` 方法处理的URL：

在编写错误信息时，可以通过控制器中的请求属性访问 `HttpServletResponse` 上设置的状态码和错误消息：

```
@Controller
public class ErrorController {

    @RequestMapping(path = "/error", produces = MediaType.APPLICATION_JSON_UTF8_VALUE)
    @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JSP中：

```
<%@ page contentType="application/json" pageEncoding="UTF-8"%>
{
    status:<%=request.getAttribute("javax.servlet.error.status_code") %>,
    reason:<%=request.getAttribute("javax.servlet.error.message") %>
}
```

## 18.12 网络安全

[Spring Security](#)项目提供了保护Web应用程序免受恶意攻击的功能。查看“[CSRF保护](#)”，“[安全响应头](#)”以及“[Spring MVC集成](#)”部分中的参考文档。请注意，使用[Spring Security](#)来保护应用程序不一定是所有功能都需要的。例如，可以通过将 `CsrfFilter` 和 `CsrfRequestValueProcessor` 添加到您的配置来添加CSRF保护。请参阅[Spring MVC展示](#)示例。

另一个选择是使用专用于Web Security的框架。[HDIV](#)就是一个这样的框架，并且与[Spring MVC](#)集成在一起。

## 18.13 关于配置支持的公约

For a lot of projects, sticking to established conventions and having reasonable defaults is just what they (the projects) need, and Spring Web MVC now has explicit support for *convention over configuration*. What this means is that if you establish a set of naming conventions and suchlike, you can\_substantially\_cut down on the amount of configuration that is required to set up handler mappings, view resolvers, `ModelAndView` instances, etc. This is a great boon with regards to rapid prototyping, and can also lend a degree of (always good-to-have) consistency across a codebase should you choose to move forward with it into production.

Convention-over-configuration support addresses the three core areas of MVC: models, views, and controllers.

对于很多项目来说，坚持已有的约定和合理的默认值就是他们（项目）需要的东西，而Spring Web MVC现在已经明确支持配置公约。这意味着如果你建立了一组命名约定等类似的东西，你可以大大减少设置处理器映射，查看解析器， `ModelAndView` 实例等所需的配置数量。这对于快速原型设计，还可以在整个代码库中提供一定程度的（始终具有良好的一致性），如果您选择将其投入生产。

Convention-over-configuration支持解决了MVC的三个核心领域：模型，视图和控制器。

对于很多项目，坚持既定的约定和合理的默认值就是它们（项目）所需要的，而Spring Web MVC现在已经明确地支持约定的配置。这意味着如果您建立了一组命名约定等等，您可以大幅度减少设置处理器映射，查看解析器， `ModelAndView` 实例等所需的配置量。这对于快速原型，并且如果您选择将其推向生产，还可以在代码库中提供一定程度的（始终如一的）一致性。

公约超配置支持解决了MVC的三个核心领域：模型，视图和控制器。

## 18.13.1 Controller ControllerClassNameHandlerMapping

`ControllerClassNameHandlerMapping` 类是 `HandlerMapping` 实现，它使用约定来确定请求URL和处理这些请求的 `Controller` 实例之间的映射。

考虑以下简单的 `Controller` 实现。请特别注意 `class` 的名称。

```
public class ViewShoppingCartController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) {
        // the implementation is not hugely important for this example...
    }

}
```

这里是相应的 Spring Web MVC 配置文件的一个片段：

```
<bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping"/>

<bean id="viewShoppingCart" class="x.y.z.ViewShoppingCartController">
    <!-- 根据需要注入依赖关系... -->
</bean>
```

`ControllerClassNameHandlerMapping` 查找在其应用程序上下文中定义的所有各种处理程序（或 `Controller`）Bean，并从名称中除去 `Controller` 以定义其处理程序映射。因此，`ViewShoppingCartController` 映射到 `/viewshoppingcart*` 请求URL。

让我们来看一些更多的例子，让中心思想变得非常熟悉。（请注意URL中的所有小写字母，而不是骆驼式的 `Controller` 类名称。）

- `WelcomeController` 映射到 `/welcome*` 请求URL
- `HomeController` 映射到 `/home*` 请求URL
- `IndexController` 映射到 `/index*` 请求URL
- `RegisterController` 映射到 `/register*` 请求URL

在 `MultiActionController` 处理程序类的情况下，生成的映射稍微复杂一些。以下示例中的 `Controller` 名称假定为 `MultiActionController` 实现：

- `AdminController` 映射到 `/admin/*` 请求URL
- `CatalogController` 映射到 `/catalog/*` 请求URL

如果按照惯例将 `Controller` 实现命名为 `xxxController`，那么 `ControllerClassNameHandlerMapping` 将为您节省定义和维护 `SimpleUrlHandlerMapping`（或类似的）的繁琐工作。

`ControllerClassNameHandlerMapping` 类扩展了 `AbstractHandlerMapping` 基类，因此您可以像处理许多其他 `HandlerMapping` 实现一样定义 `HandlerInterceptor` 实例和其他所有内容。

## 18.13.2 Model ModelMap (ModelAndView)

`ModelMap` 类本质上是一个荣耀的 `Map`，它可以使添加的对象在 `View` 中（或在其上）显示，并遵循一个通用的命名约定。考虑下面的控制器实现；注意到对象被添加到  `ModelAndView` 而没有指定任何关联的名字。

```
public class DisplayShoppingCartController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) {

        List cartItems = // 获取CartItem对象的列表
        User user = // 让用户进行购物

        ModelAndView mav = new ModelAndView("displayShoppingCart"); <-- 逻辑视图名称

        mav.addObject(cartItems); <-- 看ma，没有名字，只是对象
        mav.addObject(user); <-- and again ma!

        return mav;
    }
}
```

`ModelAndView` 类使用一个 `ModelMap` 类，它是一个自定义的 `Map` 实现，当一个对象被添加到它时，该实现自动为一个对象生成一个键。在像 `User` 这样的标量对象的情况下，为添加对象确定名称的策略是使用对象类的简短类名称。以下示例是为放置在 `ModelMap` 实例中的标量对象生成的名称。

- `x.y.User` 添加 的实例将生成名称 `user` 。
- `x.y.Registration` 添加 的实例将生成名称 `registration` 。
- `x.y.Foo` 添加 的实例将生成名称 `foo` 。
- `java.util.HashMap` 添加的实例将生成名称 `hashMap` 。你可能想在这种情况下明确名称，因为 `hashMap` 不是直观的。
- 添加 `null` 将导致引发 `IllegalArgumentException` 。如果要添加的对象（或多个对象）可能为 `null` ，那么您还需要明确名称。

什么，没有自动多元化？

Spring Web MVC的常规配置支持支持不支持自动多元化。也就是说，不能将一个 `List` 的 `Person` 对象列表添加到  `ModelAndView` 中，并将生成的名称作为 `people` 。

这个决定是经过一番辩论后做出来的，最后以“最低惊奇原则”获胜。

添加一个 `Set` 或者一个 `List` 之后生成一个名字的策略就是查看这个集合，获取这个集合中第一个对象的简短类名，并且在名字后面加上 `List`。对于数组也是如此，但是对于数组，不需要查看数组的内容。一些例子会使集合的名字生成的语义更清晰：

- 添加了零个或多个 `x.y.User` 元素的 `x.y.User[]` 数组将生成名称 `userList`。
- 添加了零个或多个 `x.y.User` 元素的 `x.y.Foo[]` 数组将会生成名称 `fooList`。
- 添加了一个或多个 `x.y.User` 元素的 `java.util.ArrayList` 将生成的名称 `userList`。
- 添加了一个或多个 `x.y.Foo` 元素的 `java.util.HashSet` 将具有生成的名称 `fooList`。
- 根本不会添加一个空 `java.util.ArrayList`（实际上，`addObject(..)` 调用本质上是无操作的）。

### 18.13.3 View - RequestToViewNameTranslator

`RequestToViewNameTranslator` 接口在没有明确提供这样的逻辑 `view` 名称时确定一个逻辑视图名称。它只有一个实现，即 `DefaultRequestToViewNameTranslator` 类。

`DefaultRequestToViewNameTranslator` 将请求URL映射到逻辑视图名称，如下例所示：

```
public class RegistrationController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse
response) {
        // 处理请求...
        ModelAndView mav = new ModelAndView();
        // 根据需要添加数据到模型...
        return mav;
        // 注意没有设置View或逻辑视图名称
    }

}
```

```
<?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">

    <!-- 这个具有众所周知的名称的bean为我们生成视图名称 -->
    <bean id="viewNameTranslator"
        class="org.springframework.web.servlet.view.DefaultRequestToViewNameTransl
ator"/>

    <bean class="x.y.RegistrationController">
        <!-- 根据需要注入依赖项 -->
    </bean>

    <!-- 将请求URL映射到控制器名称 -->
    <bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandle
rMapping"/>

    <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResour
ceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
    </bean>

```

请注意，在 `handleRequest(..)` 方法的实现中，返回的 `ModelAndView` 中没有设置 `view` 或逻辑视图名称。`DefaultRequestToViewNameTranslator` 的任务是从请求的 URL 生成逻辑视图名称。在与 `ControllerClassNameHandlerMapping` 一起使用的上述 `RegistrationController` 的情况下，`http://localhost/registration.html` 的请求 URL 导致由 `DefaultRequestToViewNameTranslator` 生成的注册的逻辑视图名称。这个逻辑视图名称然后被 `InternalResourceViewResolver bean` 解析到 `/WEB-INF/jsp/registration.jsp` 视图中。



您不需要明确定义 `DefaultRequestToViewNameTranslator bean`。如果你喜欢 `DefaultRequestToViewNameTranslator` 的默认设置，你可以依靠 Spring Web MVC `DispatcherServlet` 实例化这个类的实例（如果没有明确配置的话）。

当然，如果您需要更改默认设置，则需要明确配置您自己的 `DefaultRequestToViewNameTranslator bean`。请查阅全面的 `DefaultRequestToViewNameTranslator javadocs`，了解有关可配置的各种属性的详细信息。

## 18.14 HTTP缓存支持

一个好的HTTP缓存策略可以显着提高Web应用程序的性能和客户的体验。'Cache-Control' HTTP响应标头主要负责这一点，以及诸如'Last-Modified' 和 'ETag' 的条件标头。

'Cache-Control' HTTP响应标头建议专用缓存（例如浏览器）和公共缓存（例如代理）如何缓存HTTP响应以供进一步重用。

[ETag](#)（实体标签）是HTTP/1.1兼容的Web服务器返回的HTTP响应头，用于确定给定URL的内容更改。它可以被认为是 `Last-Modified` 头的更复杂的后继者。当服务器返回带有ETag头的表示时，客户端可以在随后的GET中，在 `If-None-Match` 头中使用该头。如果内容没有改变，则服务器返回 `304: Not Modified`。

本节描述了在Spring Web MVC应用程序中配置HTTP缓存的不同选择。

## 18.14.1 缓存控制HTTP头

Spring Web MVC 支持许多用例和为应用程序配置“Cache-Control”头的方法。虽然[RFC 7234 第5.2.2节](#)完整地描述了该头文件及其可能的指令，但有几种方法可以解决最常见的情况。

Spring Web MVC 在其几个API中使用了一个配置约定：`setCachePeriod(int seconds)`：

- `-1` 值不会生成 `'Cache-Control'` 响应头。
- 使用 `'Cache-Control: no-store'` 指令时，`0` 值将会阻止缓存。
- 一个 `n > 0` 值将使用 `'Cache-Control: max-age=n'` 指令缓存给定的响应秒数。

`CacheControl` 构建器类简单地描述了可用的“Cache-Control”指令，使得构建自己的HTTP缓存策略更加容易。一旦构建完成，一个 `CacheControl` 实例可以被接受为几个Spring Web MVC API中的一个参数。

```
// 缓存一小时 - "Cache-Control: max-age=3600"
CacheControl ccCacheOneHour = CacheControl.maxAge(1, TimeUnit.HOURS);

// 防止缓存 - "Cache-Control: no-store"
CacheControl ccNoStore = CacheControl.noStore();

// 在公共和私有缓存中缓存十天,
// 公共缓存不应该转换响应
// "Cache-Control: max-age=864000, public, no-transform"
CacheControl ccCustom = CacheControl.maxAge(10, TimeUnit.DAYS)
    .noTransform().cachePublic();
```

## 18.14.2 HTTP缓存支持静态资源

静态资源应该使用 appropriate 的 'Cache-Control' 和条件头来达到最佳性能。配置一个 `ResourceHttpRequestHandler` 来为静态资源提供服务，不仅通过读取文件的元数据本地写入 'Last-Modified' 头，而且还可以正确配置 'Cache-Control' 头文件。

您可以在 `ResourceHttpRequestHandler` 上设置 `cachePeriod` 属性，也可以使用支持更多特定指令的 `CacheControl` 实例：

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**")
            .addResourceLocations("/public-resources/")
            .setCacheControl(CacheControl.maxAge(1, TimeUnit.HOURS).cachePublic());
    }

}
```

在 XML 中：

```
<mvc:resources mapping="/resources/**" location="/public-resources/">
    <mvc:cache-control max-age="3600" cache-public="true"/>
</mvc:resources>
```

### 18.14.3 支持控制器中的Cache-Control，ETag和Last-Modified响应头

控制器可以支持 'Cache-Control'，'ETag' 和/或 'If-Modified-Since' HTTP请求。如果在响应中设置 'Cache-Control' 头部，这确实是推荐的。这包括计算给定请求的lastmodified long 和/或Etag值，并将其与 'If-Modified-Since' 请求标头值进行比较，并可能返回状态码 304（未修改）的响应。

如“[使用HttpEntity](#)”一节所述，控制器可以使用 `HttpEntity` 类型与请求/响应进行交互。返回 `ResponseEntity` 的控制器可以包含HTTP缓存信息，如下所示：

```
@GetMapping("/book/{id}")
public ResponseEntity<Book> showBook(@PathVariable Long id) {

    Book book = findBook(id);
    String version = book.getVersion();

    return ResponseEntity
        .ok()
        .cacheControl(CacheControl.maxAge(30, TimeUnit.DAYS))
        .eTag(version) // lastModified is also available
        .body(book);
}
```

如果客户端发送的条件标头与控制器设置的缓存信息匹配，这样做将不仅包括响应中的头 'ETag' 和 'Cache-Control' 头，还会将响应转换 `HTTP 304 Not Modified` 为空体。

`@RequestMapping` 方法也可能希望支持相同的行为。这可以实现如下：

```
@RequestMapping
public String myHandleMethod(WebRequest webRequest, Model model) {

    long lastModified = // 1. 应用程序特定的计算

    if (request.checkNotModified(lastModified)) {
        // 2. 快捷方式退出 - 无需进一步处理必需
        return null;
    }

    // 3. 或另外请求处理，实际准备内容
    model.addAttribute(...);
    return "myViewName";
}
```

这里有两个关键元素：调用 `request.checkNotModified(lastModified)` 并返回 `null`。前者在返回 `true` 之前设置适当的响应状态和标题。后者与前者结合使得 Spring MVC 不再对请求做进一步的处理。

请注意，有3种变体：

- `request.checkNotModified(lastModified)` 将 `lastModified` 与 '`If-Modified-Since`' 或 '`If-Unmodified-Since`' 请求头进行比较
- `request.checkNotModified(eTag)` 将 `eTag` 与 '`If-None-Match`' 请求标头进行比较
- `request.checkNotModified(eTag, lastModified)` 同时具有这两个条件，这意味着两个条件都应该是有效的

当接收到有条件的 '`GET`' / '`HEAD`' 请求时，`checkNotModified` 将检查资源是否未被修改，如果是的话，将导致 `HTTP 304 Not Modified` 响应。如果出现 '`POST`' / '`PUT`' / '`DELETE`' 请求，`checkNotModified` 将检查资源是否未被修改，如果已经被修改，将导致 `HTTP 409 Precondition Failed` 响应阻止并发修改。

## 18.14.4 浅层 ETag 支持

ETags 的支持由 Servlet 过滤器 `ShallowEtagHeaderFilter` 提供。这是一个简单的 Servlet 过滤器，因此可以与任何 Web 框架结合使用。`ShallowEtagHeaderFilter` 过滤器创建所谓的浅 ETag（相对于深 ETag，稍后会详细介绍）。过滤器缓存呈现的 JSP（或其他内容）的内容，生成一个 MD5 哈希，然后将其作为 ETag 响应头。下一次客户端发送相同资源的请求时，它将使用该散列作为 `If-None-Match` 值。过滤器检测到这一点，再次呈现视图，并比较两个散列。如果相等，则返回 `304`。

请注意，此策略可节省网络带宽而不是 CPU，因为必须为每个请求计算完整响应。控制器级别的其他策略（如上所述）可以节省网络带宽并避免计算。

此过滤器具有 `awriteETag` 参数，该参数配置过滤器写入弱 ETag，如下所示：`w/"02a2d595e6ed9a0b24f027f2b63b134d6"`，如[RFC 7232 第 2.3 节](#)中所定义。

在 `web.xml` 中配置 `ShallowEtagHeaderFilter`：

```
<filter>
    <filter-name>etagFilter</filter-name>
    <filter-class>org.springframework.web.filter.ShallowEtagHeaderFilter</filter-class>
    <!-- 配置过滤器写入弱ETag的可选参数
    <init-param>
        <param-name>writeWeakETag</param-name>
        <param-value>true</param-value>
    </init-param>
    -->
</filter>

<filter-mapping>
    <filter-name>etagFilter</filter-name>
    <servlet-name>petclinic</servlet-name>
</filter-mapping>
```

或者在 Servlet 3.0+ 环境中，

```
public class MyWebAppInitializer extends AbstractDispatcherServletInitializer {

    // ...

    @Override
    protected Filter[] getServletFilters() {
        return new Filter[] { new ShallowEtagHeaderFilter() };
    }

}
```

有关详细信息，请参见[第18.15节“基于代码的Servlet容器初始化”](#)。

## 18.15 基于代码的Servlet容器初始化

在Servlet 3.0+环境中，您可以选择以编程方式配置Servlet容器作为替代方法或与 web.xml 文件结合使用。以下是一个注册 DispatcherServlet 的例子：

```
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Spring MVC提供的一个接口，它确保您的实现被检测到并自动用来初始化任何Servlet 3容器。一个名

为 AbstractDispatcherServletInitializer 的 WebApplicationInitializer 的抽象基类实现，通过简单地重写方法来指定Servlet映射和 DispatcherServlet 配置的位置，使注册 DispatcherServlet 变得更加容易。

推荐使用基于Java的Spring配置的应用程序：

```

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[] { "/" };
    }

}

```

如果使用基于XML的Spring配置，则应该直接从 `AbstractDispatcherServletInitializer` 扩展：

```

public class MyWebAppInitializer extends AbstractDispatcherServletInitializer {

    @Override
    protected WebApplicationContext createRootApplicationContext() {
        return null;
    }

    @Override
    protected WebApplicationContext createServletApplicationContext() {
        XmlWebApplicationContext ctxt = new XmlWebApplicationContext();
        ctxt.setConfigLocation("/WEB-INF/spring dispatcher-config.xml");
        return ctxt;
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }

}

```

`AbstractDispatcherServletInitializer` 还提供了添加 `Filter` 实例并将其自动映射到 `DispatcherServlet` 的便捷方式：

```
public class MyWebAppInitializer extends AbstractDispatcherServletInitializer {  
    // ...  
  
    @Override  
    protected Filter[] getServletFilters() {  
        return new Filter[] { new HiddenHttpMethodFilter(), new CharacterEncodingFilte  
r() };  
    }  
}
```

每个过滤器都根据其具体类型添加一个默认名称，并自动映射到 `DispatcherServlet`。

`AbstractDispatcherServletInitializer` 的 `isAsyncSupported` 受保护方法提供了一个单独的位置来启用 `DispatcherServlet` 上的异步支持以及映射到它的所有过滤器。默认情况下，这个标志被设置为 `true`。

最后，如果您需要进一步自定义 `DispatcherServlet` 本身，则可以覆盖 `createDispatcherServlet` 方法。

## 18.16 配置 Spring MVC

第18.2.1节“[WebApplicationContext中的特殊Bean类型](#)”和第18.2.2节“[默认DispatcherServlet配置](#)”介绍了Spring MVC的特殊bean以及 DispatcherServlet 使用的默认实现。在本节中，您将学习到另外两种配置Spring MVC的方法。即MVC Java配置和MVC XML命名空间。

MVC Java配置和MVC命名空间提供了类似的默认配置，覆盖了 DispatcherServlet 的默认设置。我们的目标是让大多数应用程序不必创建相同的配置，也可以提供更高级别的构造，用于配置Spring MVC，作为一个简单的起点，并且不需要事先知道底层配置。

您可以根据您的偏好选择MVC Java配置或MVC命名空间。同样如您将在下面看到的，使用MVC Java配置，可以更容易地看到底层配置以及直接对创建的Spring MVC bean进行细粒度的自定义。但是我们从头开始。

## 18.16.1 启用MVC Java Config或MVC XML命名空间

要启用MVC Java配置，请将注解 `@EnableWebMvc` 添加到其中一个 `@Configuration` 类中：

```
@Configuration
@EnableWebMvc
public class WebConfig {

}
```

要在XML中实现相同的效果，请在`DispatcherServlet`上下文中使用 `mvc:annotation-driven` 元素（如果没有定义`DispatcherServlet`上下文，则在根上下文中）：

```
<?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>
```

上面注册了一个 `RequestMappingHandlerMapping`，一个 `RequestMappingHandlerAdapter` 和一个 `ExceptionHandlerExceptionResolver`（以及其他），以支持使用注解的控制器方法（如 `@RequestMapping`，`@ExceptionHandler` 等）处理请求。

它还支持以下功能：

1. 除了用于数据绑定的JavaBean `PropertyEditor`之外，还通过[ConversionService](#)实例进行Spring 3样式类型转换。
2. 支持通过 `ConversionService` 使用 `@NumberFormat` 注解[格式化](#)数字字段。
3. 支持使用 `@DateTimeFormat` 注解[格式化](#) `Date`，`Calendar`，`Long` 和[Joda时间](#)字段。
4. 如果JSR-303提供程序存在于类路径中，则支持[validating](#)使用 `@Valid` 验证 `@Controller` 输入。
5. `HttpMessageConverter` 支持 `@RequestBody` 方法参数和 `@ResponseBody` 方法从 `@RequestMapping` 或 `@ExceptionHandler` 方法返回值。

这是由mvc设置的HttpMessageConverters的完整列表：annotation-driven：

- i. `ByteArrayHttpMessageConverter` 转换字节数组。
- ii. `StringHttpMessageConverter` 转换字符串。
- iii. `ResourceHttpMessageConverter` 为所有媒体类型转换  
为 `org.springframework.core.io.Resource`。
- iv. `SourceHttpMessageConverter` 转换为/从一个`javax.xml.transform.Source`。
- v. `FormHttpMessageConverter` 将表单数据转换为 `MultiValueMap` 或从 `MultiValueMap` 转换而来。
- vi. `Jaxb2RootElementHttpMessageConverter` 将Java对象转换为/从XML转换 - 如果存在 JAXB2并且类路径中不存在Jackson 2 XML扩展，则添加它。
- vii. `MappingJackson2HttpMessageConverter` 转换为/从JSON - 如果Jackson 2存在于类路径中，则添加。
- viii. `MappingJackson2XmlHttpMessageConverter` 转换为/从XML - 如果Jackson 2 XML扩展存在于类路径中，则添加。
- ix. `MappingJackson2SmileHttpMessageConverter` converts to/from Smile (binary JSON) — added if Jackson 2 Smile extension is present on the classpath.
- x. `MappingJackson2SmileHttpMessageConverter` 转换为/从Smile (二进制JSON) - 添加如果Jackson 2 CBOR extension存在于类路径。
- xi. `AtomFeedHttpMessageConverter` 转换Atom提要 - 如果Rome出现在类路径中，则添加Atom提要。
- xii. `RssChannelHttpMessageConverter` 转换RSS源 - 如果Rome出现在类路径中，则添加。

有关如何定制这些默认转换器的更多信息，请参见[第18.16.12节“消息转换器”](#)。



Jackson JSON和XML转换器是使用 `Jackson2ObjectMapperBuilder` 创建的 `ObjectMapper` 实例创建的，以便提供更好的默认配置。此构建器使用以下几种方法自定义Jackson的默认属性：`DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES` 被禁用。`MapperFeature.DEFAULT_VIEW_INCLUSION` 被禁用。如果在类路径中检测到以下模块，它将自动注册下列已知模块：`jackson-datatype-jdk7`：支持 `java.nio.file.Path` 等Java 7类型。`jackson-datatype-joda`：支持Joda-Time类型。`jackson-datatype-jsr310`：支持Java 8 Date&Time API类型。`jackson-datatype-jdk8`：支持其他Java 8类型，如 `optional`。



## 18.16.2 定制提供的配置

要在Java中定制默认配置，只需实现 `WebMvcConfigurer` 接口，或者更可能扩展 `WebMvcConfigurerAdapter` 类并覆盖所需的方法：

```
@Configuration  
@EnableWebMvc  
public class WebConfig extends WebMvcConfigurerAdapter {  
  
    // Override configuration methods...  
  
}
```

要自定义 `<mvc:annotation-driven/>` 的缺省配置，请检查它支持的属性和子元素。您可以查看[Spring MVC XML模式](#)或使用IDE的代码完成功能来发现可用的属性和子元素。

### 18.16.3 转换和格式化

默认情况下，`Number` 和 `Date` 类型的格式化程序已安装，包括支持 `@NumberFormat` 和 `@DateTimeFormat` 注解。如果 Joda 时间存在于类路径中，则还会安装对 Joda 时间格式库的完全支持。要注册自定义格式器和转换器，请重写 `addFormatters` 方法：

```
@Configuration  
@EnableWebMvc  
public class WebConfig extends WebMvcConfigurerAdapter {  
  
    @Override  
    public void addFormatters(FormatterRegistry registry) {  
        // Add formatters and/or converters  
    }  
}
```

在MVC命名空间中，添加 `<mvc:annotation-driven>` 时会应用相同的默认值。要注册自定义格式化器和转换器，只需提供一个 `ConversionService`：

```
<?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.FormattingConversionServiceFacto
ryBean">
        <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>
```



有关何时使用 `FormatterRegistrars` 的更多信息，请参见 [Section 5.6.4, “FormatterRegistrar SPI”](#) 和 `FormattingConversionServiceFactoryBean`。

## 18.16.4 验证

Spring提供了一个验证器接口，可用于在应用程序的所有层进行验证。在Spring MVC中，您可以将其配置为用作全局 validator 实例，以便在遇到 @Valid 或 @Validated 控制器方法参数时使用该实例，并且/或者通过 @InitBinder 方法将其用作控制器内的本地 validator 程序。全局和本地验证器实例可以组合起来提供复合验证。

Spring还支持JSR-303 / JSR-349 Bean验证，通过 LocalValidatorFactoryBean 使 Spring org.springframework.validation.Validator 接口适应 Bean 验证 javax.validation.Validator 契约。这个类可以作为一个全局验证器插入到 Spring MVC 中，如下所述。

默认情况下，使用 @EnableWebMvc 或者 <mvc:annotation-driven> 当在类路径上检测到一个 Bean 验证提供程序（如 Hibernate Validator）时，通过 LocalValidatorFactoryBean 在 Spring MVC 中自动注册 Bean 验证支持。



有时将 LocalValidatorFactoryBean 注入到控制器或其他类中很方便。最简单的方法是声明自己的 @Bean，并用 @Primary 标记以避免与 MVC Java 配置提供的冲突。如果您更喜欢使用 MVC Java 配置中的那个，那么您将需要重写 WebMvcConfigurerAdapter 中的 mvcValidator 方法，并声明该方法显式返回 LocalValidatorFactory 而不是 Validator。有关如何切换以扩展提供的配置的信息，请参见 [Section 18.16.13, “Advanced Customizations with MVC Java Config”](#)。

或者，您可以配置您自己的全局 validator 实例：

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public Validator getValidator() {
        // return "global" validator
    }

}
```

和 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 validator="globalValidator"/>

</beans>
```

要将全局和本地验证相结合，只需添加一个或多个本地验证程序即可：

```
@Controller
public class MyController {

    @InitBinder
    protected void initBinder(WebDataBinder binder) {
        binder.addValidators(new FooValidator());
    }
}
```

有了这个最小的配置，任何时候遇到 `@Valid` 或 `@Validated` 方法参数，它都将被配置的验证器验证。任何验证违规将自动作为方法参数可访问的 `BindingResult` 中的错误公开，也可以在 Spring MVC HTML 视图中呈现。

## 18.16.5 拦截器

您可以将 `HandlerInterceptors` 或 `WebRequestInterceptors` 配置为应用于所有传入请求或限制为特定的URL路径模式。

在Java中注册拦截器的示例：

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new LocaleInterceptor());
        registry.addInterceptor(new ThemeInterceptor()).addPathPatterns("/**").exclude
        PathPatterns("/admin/**");
        registry.addInterceptor(new SecurityInterceptor()).addPathPatterns("/secure/*"
    );
    }

}
```

在XML中使用 `<mvc:interceptors>` 元素：

```
<mvc:interceptors>
    <bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor"/>
    <mvc:interceptor>
        <mvc:mapping path="/**"/>
        <mvc:exclude-mapping path="/admin/**"/>
        <bean class="org.springframework.web.servlet.theme.ThemeChangeInterceptor"/>
    </mvc:interceptor>
    <mvc:interceptor>
        <mvc:mapping path="/secure/*"/>
        <bean class="org.example.SecurityInterceptor"/>
    </mvc:interceptor>
</mvc:interceptors>
```

## 18.16.6 内容谈判

您可以配置Spring MVC如何根据请求确定请求的媒体类型。可用的选项是检查文件扩展名的URL路径，检查“Accept”头，特定的查询参数，或者在没有请求时返回默认的内容类型。默认情况下，首先检查请求URI中的路径扩展，然后检查“Accept”标头。

MVC Java配置和MVC命名空间默认情况下注册json，xml，rss，atom，如果相应的依赖关系在类路径上。额外的路径扩展到媒体类型的映射也可以被明确注册，并且为了RFD攻击检测的目的（参见[“后缀模式匹配和RFD”部分以获得更多细节](#)）也具有将它们白名单作为安全扩展的效果。

以下是通过MVC Java配置自定义内容协商选项的示例：

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void configureContentNegotiation(ContentNegotiationConfigurer configurer) {
        configurer.mediaType("json", MediaType.APPLICATION_JSON);
    }
}
```

在MVC名称空间中，`<mvc:annotation-driven>` 元素具有 `content-negotiation-manager` 属性，该属性需要一个 `ContentNegotiationManager`，而 `ContentNegotiationManager` 又可以使 `ContentNegotiationManagerFactoryBean` 创建：

```
<mvc:annotation-driven content-negotiation-manager="contentNegotiationManager"/>

<bean id="contentNegotiationManager" class="org.springframework.web.accept.ContentNegotiationManagerFactoryBean">
    <property name="mediaTypes">
        <value>
            json=application/json
            xml=application/xml
        </value>
    </property>
</bean>
```

如果不使用MVC Java配置或MVC命名空间，则需要创建 `ContentNegotiationManager` 的一个实例，并使用它来配置 `RequestMappingHandlerMapping` 以实现请求映射，`RequestMappingHandlerAdapter` 和 `ExceptionHandlerExceptionResolver` 实现内容协商。

请注意，`ContentNegotiatingViewResolver` 现在也可以使用 `ContentNegotiationManager` 进行配置，因此您可以在整个Spring MVC中使用一个共享实例。

在更高级的情况下，配置多个 `ContentNegotiationManager` 实例可能会很有用，而这些实例又可能包含自定义的 `contentNegotiationStrategy` 实现。例如，你可以配置 `ExceptionHandlerExceptionResolver` 和一个 `ContentNegotiationManager`，它始终将请求的媒体类型解析为 `"application/json"`。或者，如果没有请求内容类型，您可能需要插入具有逻辑的自定义策略来选择默认内容类型（例如`XML`或`JSON`）。

## 18.16.7 视图控制器

这是定义一个 `ParameterizableViewController` 的快捷方式，可以在调用时立即转发到视图。在静态情况下使用它，在视图生成响应之前没有执行Java控制器逻辑。

在Java中将 "/" 的请求转发到名为 "home" 的视图的示例：

```
@Configuration  
@EnableWebMvc  
public class WebConfig extends WebMvcConfigurerAdapter {  
  
    @Override  
    public void addViewControllers(ViewControllerRegistry registry) {  
        registry.addViewController("/").setViewName("home");  
    }  
  
}
```

XML中的相同内容使用 `<mvc:view-controller>` 元素：

```
<mvc:view-controller path="/" view-name="home"/>
```

## 18.16.8 查看解析器

MVC配置简化了视图解析器的注册。

以下是一个Java配置示例，它使用FreeMarker HTML模板和Jackson作为JSON呈现的默认 view 来配置内容协商视图分辨率：

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.enableContentNegotiation(new MappingJackson2JsonView());
        registry.jsp();
    }

}
```

和XML一样：

```
<mvc:view-resolvers>
    <mvc:content-negotiation>
        <mvc:default-views>
            <bean class="org.springframework.web.servlet.view.json.MappingJackson2Json
View"/>
        </mvc:default-views>
    </mvc:content-negotiation>
    <mvc:jsp/>
</mvc:view-resolvers>
```

但是请注意，FreeMarker，Tiles，Groovy Markup和脚本模板也需要配置底层视图技术。

MVC命名空间提供了专用的元素。例如FreeMarker：

```
<mvc:view-resolvers>
    <mvc:content-negotiation>
        <mvc:default-views>
            <bean class="org.springframework.web.servlet.view.json.MappingJackson2Json
View"/>
        </mvc:default-views>
    </mvc:content-negotiation>
    <mvc:freemarker cache="false"/>
</mvc:view-resolvers>

<mvc:freemarker-configure>
    <mvc:template-loader-path location="/freemarker"/>
</mvc:freemarker-configure>
```

在Java配置中只需添加相应的“Configurer” bean：

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.enableContentNegotiation(new MappingJackson2JsonView());
        registry.freeMarker().cache(false);
    }

    @Bean
    public FreeMarkerConfigurer freeMarkerConfigurer() {
        FreeMarkerConfigurer configurer = new FreeMarkerConfigurer();
        configurer.setTemplateLoaderPath("/WEB-INF/");
        return configurer;
    }

}
```

## 18.16.9 服务于资源

这个选项允许一个特定的URL模式之后的静态资源请求由一个 `ResourceHttpRequestHandler` 从任何一个资源位置列表中提供。这提供了一种方便的方式来提供来自Web应用程序根目录以外的位置的静态资源，包括类路径上的位置。`cache-period` 属性可以用来设置将来过期头（1年是诸如Page Speed和YSlow等优化工具的推荐），以便客户更有效地使用它们。该处理程序还可以正确评估 `Last-Modified` 标头（如果存在），以便适当地返回 `304` 状态码，避免客户端已经缓存的资源产生不必要的开销。例如，要使用Web应用程序根目录中 `public-resources` 目录中的 `/resources/**` 的URL模式提供资源请求，可以使用：

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**").addResourceLocations("/public-resources/");
    }

}
```

和XML一样：

```
<mvc:resources mapping="/resources/**" location="/public-resources/" />
```

要为这些资源提供1年的将来期限，以确保最大限度地利用浏览器缓存并减少浏览器发出的HTTP请求：

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**").addResourceLocations("/public-resources/").setCachePeriod(31556926);
    }

}
```

在XML中：

```
<mvc:resources mapping="/resources/**" location="/public-resources/" cache-period="31556926"/>
```

有关更多详细信息，请参阅[HTTP缓存对静态资源的支持](#)。

`mapping` 属性必须是可由 `SimpleUrlHandlerMapping` 使用的Ant模式，而 `location` 属性必须指定一个或多个有效的资源目录位置。可以使用逗号分隔的值列表来指定多个资源位置。指定的位置将按照指定的顺序检查任何给定请求是否存在资源。例如，要启用来自Web应用程序根目录和来自 `/META-INF/public-web-resources/` 的已知路径的资源，请在类路径的任何jar中使用：

```
@EnableWebMvc
@Configuration
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**")
            .addResourceLocations("/", "classpath:/META-INF/public-web-resources/");
    }
}
```

在XML中：

```
<mvc:resources mapping="/resources/**" location="/, classpath:/META-INF/public-web-resources/" />
```

当部署应用程序的新版本时可能会更改资源时，建议您将版本字符串并入用于请求资源的映射模式，以便您可以强制客户端请求新部署的应用程序资源版本。对版本化URL的支持已内置到框架中，并且可以通过在资源处理器上配置资源链来启用。该链由多个 `ResourceResolver` 实例组成，后面紧跟着一个或多个 `ResourceTransformer` 实例。他们一起可以提供任意的解决方案和资源转换。

内置的 `VersionResourceResolver` 可以配置不同的策略。例如，`FixedVersionStrategy` 可以使用属性，日期或其他作为版本。`ContentVersionStrategy` 使用从资源内容（称为“fingerprinting”URL）计算出的MD5哈希值。请注意，在服务资源时，`VersionResourceResolver` 将自动使用已解析的版本字符串作为HTTP ETag标头值。

`ContentVersionStrategy` 是一个很好的默认选择，除非无法使用（例如使用JavaScript模块加载器）。您可以针对不同的模式配置不同的版本策略，如下所示。请记住，计算基于内容的版本是昂贵的，因此应该在生产中启用资源链高速缓存。

Java配置示例；

```

@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**")
            .addResourceLocations("/public-resources/")
            .resourceChain(true).addResolver(
                new VersionResourceResolver().addContentVersionStrategy("/**"));
    }

}

```

XML example:

```

<mvc:resources mapping="/resources/**" location="/public-resources/">
    <mvc:resource-chain>
        <mvc:resource-cache/>
        <mvc:resolvers>
            <mvc:version-resolver>
                <mvc:content-version-strategy patterns="/**"/>
            </mvc:version-resolver>
        </mvc:resolvers>
    </mvc:resource-chain>
</mvc:resources>

```

为了上述工作，应用程序还必须呈现带有版本的URL。最简单的方法是配置包装响应的 `ResourceUrlEncodingFilter`，并覆盖 `encodeURL` 方法。这将在JSP，FreeMarker和其他任何调用 `encodeURL` 方法的视图技术中起作用。或者，应用程序也可以直接注入并使用 `ResourceUrlProvider` bean，该bean是使用MVC Java配置和MVC命名空间自动声明的。

WebJars也支持 `WebJarsResourceResolver`，当 "org.webjars:webjars-locator" 库位于类路径上时，会自动注册。此解析器允许资源链从HTTP GET请求解析版本不可知的库 "GET /jquery/jquery.min.js" 将返回资源 "/jquery/1.2.0/jquery.min.js"。它还通过在模板→重写资源URL来工作。

## 18.16.10 回到“Default”Servlet服务资源

这允许将 `DispatcherServlet` 映射到 "/" (从而覆盖容器默认 `Servlet` 的映射) , 同时允许静态资源请求由容器的默认 `Servlet` 处理。它使用 URL 映射 “`/**`” 配置 `DefaultServletHttpRequestHandler` , 并且相对于其他 URL 映射的优先级最低。

这个处理程序将把所有请求转发给默认的 `Servlet` 。因此重要的是它保持最后的所有其他 URL `HandlerMappings` 的顺序。如果您使用 `<mvc:annotation-driven>` , 或者如果您正在设置自己的自定义 `HandlerMapping` 实例, 请确保将其 `order` 属性设置为比 `DefaultServletHttpRequestHandler` (即 `Integer.MAX_VALUE` ) 更低的值。

要使用默认设置启用该功能, 请使用 :

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {
        configurer.enable();
    }

}
```

或者在 XML 中 :

```
<mvc:default-servlet-handler/>
```

覆盖 “`/`” `Servlet` 映射的警告是, 默认 `Servlet` 的 `RequestDispatcher` 必须通过名称而不是路径来检索。`DefaultServletHttpRequestHandler` 将尝试在启动时使用大多数主要 `Servlet` 容器 (包括 Tomcat , Jetty , GlassFish , JBoss , Resin , WebLogic 和 WebSphere ) 的已知名称列表来自动检测容器的默认 `Servlet` 。如果默认的 `Servlet` 已经被自定义配置了一个不同的名字, 或者当默认的 `Servlet` 名字是未知的时候使用了一个不同的 `Servlet` 容器, 那么默认的 `Servlet` 的名字必须被显式地提供, 如下例所示 :

```
@Configuration  
@EnableWebMvc  
public class WebConfig extends WebMvcConfigurerAdapter {  
  
    @Override  
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer config  
urer) {  
        configurer.enable("myCustomDefaultServlet");  
    }  
  
}
```

或者在XML中：

```
<mvc:default-servlet-handler default-servlet-name="myCustomDefaultServlet"/>
```

## 18.16.11 路径匹配

这允许自定义与URL映射和路径匹配相关的各种设置。有关各个选项的详细信息，请查看[PathMatchConfigurer API](#)。

以下是Java配置中的一个例子：

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void configurePathMatch(PathMatchConfigurer configurer) {
        configurer
            .setUseSuffixPatternMatch(true)
            .setUseTrailingSlashMatch(false)
            .setUseRegisteredSuffixPatternMatch(true)
            .setPathMatcher(antPathMatcher())
            .setUrlPathHelper(urlPathHelper());
    }

    @Bean
    public UrlPathHelper urlPathHelper() {
        //...
    }

    @Bean
    public PathMatcher antPathMatcher() {
        //...
    }

}
```

和XML一样，使用 `<mvc:path-matching>` 元素：

```
<mvc:annotation-driven>
    <mvc:path-matching
        suffix-pattern="true"
        trailing-slash="false"
        registered-suffixes-only="true"
        path-helper="pathHelper"
        path-matcher="pathMatcher"/>
</mvc:annotation-driven>

<bean id="pathHelper" class="org.example.app.MyPathHelper"/>
<bean id="pathMatcher" class="org.example.app.MyPathMatcher"/>
```



## 18.16.12 消息转换器

`HttpMessageConverter` 的定制可以在 Java 配置中通过覆盖 `configureMessageConverters()` 来实现，如果你想替换 Spring MVC 创建的默认转换器，或者如果你只是想定制它们或者添加额外的转换器到默认转换器，可以重写 `extendMessageConverters()`。

下面是一个例子，添加了 Jackson JSON 和 XML 转换器的自定义 `ObjectMapper` 而不是默认的：

```
@Configuration
@EnableWebMvc
public class WebConfiguration extends WebMvcConfigurerAdapter {

    @Override
    public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {
        Jackson2ObjectMapperBuilder builder = new Jackson2ObjectMapperBuilder()
            .indentOutput(true)
            .dateFormat(new SimpleDateFormat("yyyy-MM-dd"))
            .modulesToInstall(new ParameterNamesModule());
        converters.add(new MappingJackson2HttpMessageConverter(builder.build()));
        converters.add(new MappingJackson2XmlHttpMessageConverter(builder.xml().build()));
    }
}
```

在这个例子中，`Jackson2ObjectMapperBuilder` 用于为 `MappingJackson2HttpMessageConverter` 和 `MappingJackson2XmlHttpMessageConverter` 创建通用配置，并启用缩进功能，自定义日期格式以及添加对访问参数名称的支持的 [jackson-module-parameter-names](#) 注册（Java 8 中添加的功能）。

	使用 Jackson XML 支持启用缩进除了 [ jackson-dataformat-xml ] ( <a href="https://search.maven.org/#search">https://search.maven.org/#search</a>	ga	1	a%3A"jackson-dataformat-xml")之外，还需要 [ woodstox-core-asl ] ( <a href="https://search.maven.org/#search">https://search.maven.org/#search</a>	ga
---	---	----	---	--	----

其他有趣的 Jackson 模块可用：

1. [jackson-datatype-money](#): 支持 `javax.money` 类型（非官方模块）
2. [jackson-datatype-hibernate](#): 支持 Hibernate 的特定类型和属性（包括延迟加载方面）

在 XML 中也可以这样做：

```
<mvc:annotation-driven>
    <mvc:message-converters>
        <bean class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter">
            <property name="objectMapper" ref="objectMapper"/>
        </bean>
        <bean class="org.springframework.http.converter.xml.MappingJackson2XmlHttpMessageConverter">
            <property name="objectMapper" ref="xmlMapper"/>
        </bean>
    </mvc:message-converters>
</mvc:annotation-driven>

<bean id="objectMapper" class="org.springframework.http.converter.json.Jackson2ObjectMapperFactoryBean"
    p:indentOutput="true"
    p:simpleDateFormat="yyyy-MM-dd"
    p:modulesToInstall="com.fasterxml.jackson.module.paramnames.ParameterNamesModule"
/>

<bean id="xmlMapper" parent="objectMapper" p:createXmlMapper="true"/>
```

## 18.16.13 使用MVC Java配置进行高级自定义

从上面的例子可以看出，MVC Java配置和MVC命名空间提供了更高层次的构造，不需要深入了解为您创建的底层bean。相反，它可以帮助您专注于您的应用程序需求。但是，在某些时候，您可能需要更细粒度的控制，或者您可能只是想了解底层配置。

更细粒度控制的第一步是查看为您创建的底层bean。在MVC Java配置中，您可以在 `WebMvcConfigurationSupport` 中看到javadocs和 `@Bean` 方法。此类中的配置通过 `@EnableWebMvc` 注解自动导入。实际上，如果您打开 `@EnableWebMvc`，则可以看到 `@Import` 语句。

更细粒度控制的下一步是定制 `WebMvcConfigurationSupport` 中创建的一个bean的属性，或者提供自己的实例。这需要两件事 - 删除 `@EnableWebMvc` 注解，以防止导入，然后从 `WebMvcConfigurationSupport` 的子类 `DelegatingWebMvcConfiguration` 扩展。这里是一个例子：

```
@Configuration
public class WebConfig extends DelegatingWebMvcConfiguration {

    @Override
    public void addInterceptors(InterceptorRegistry registry){
        // ...
    }

    @Override
    @Bean
    public RequestMappingHandlerAdapter requestMappingHandlerAdapter() {
        // Create or let "super" create the adapter
        // Then customize one of its properties
    }

}
```



一个应用程序应该只有一个配置来扩展 `DelegatingWebMvcConfiguration` 或一个 `@EnableWebMvc` 注解类，因为它们都注册了相同的底层bean。用这种方式修改bean并不妨碍你使用本节前面所示的任何更高级的结构。`WebMvcConfigurerAdapter` 子类和 `WebMvcConfigurer` 实现仍在使用中。

## 18.16.14 使用MVC命名空间进行高级自定义

对您创建的配置进行细粒度的控制对于MVC命名空间来说有点难度。

如果您需要这样做，而不是复制它提供的配置，请考虑配置一个 BeanPostProcessor ，该 BeanPostProcessor 检测要按类型自定义的bean，然后根据需要修改其属性。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
        }
    }

}
```

请注意， MyPostProcessor 需要包含在 <component scan /> 中才能被检测到，或者如果您愿意，可以使用XML bean声明式声明它。



Spring架构与其他MVC框架所不同的重要一点是视图技术，比如，决定使用Groovy Markup Templates 或者Thymeleaf代替JSP仅仅是配置的问题。这个章节主要设计主流的视图技术，以及简单提及怎样使用新的技术。这个章节假设你已经熟悉第18.5节“[Resolving views](#)”，该章节涵盖了视图怎样与MVC框架结合的基本知识。

Thymeleaf 是一种与MVC架构结合很好的视图技术。不仅仅 Spring 团队而且 Thymeleaf 自身也提供了很好的支持。

配置 Thymeleaf 对 Spring 的支持通常需要定义几个 bean, 像 `ServletContextTemplateResolver` , `SpringTemplateEngine` 和 `ThymeleafViewResolver` 。如需要更多详细信息请点击文档 [Thymeleaf+Spring](#) 。

[Groovy Markup Template Engine](#) 是另一种被Spring支持的视图技术，此模板引擎是一种主要用于生成类似XML的标记（XML，XHTML，HTML5，...）的模板引擎，但可用于生成任何基于文本的内容。

这需要在classpath上配置Groovy 2.3.1+。

### 19.3.1 配置

配置 Groovy Markup Template Engine 相当容易：

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.groovy();
    }

    @Bean
    public GroovyMarkupConfigurer groovyMarkupConfigurer() {
        GroovyMarkupConfigurer configurer = new GroovyMarkupConfigurer();
        configurer.setResourceLoaderPath("/WEB-INF/");
        return configurer;
    }
}
```

使用 MVC 命名空间的 XML 文本：

```
<mvc:annotation-driven/>

<mvc:view-resolvers>
    <mvc:groovy/>
</mvc:view-resolvers>

<mvc:groovy-configure resource-loader-path="/WEB-INF/" />
```

## 19.3.2 例子

和传统模板引擎不同，这一个依赖于使用构建器语法的DSL。以下是HTML页面的示例模板：

```
yieldUnescaped '<!DOCTYPE html>'  
html(lang:'en') {  
    head {  
        meta('http-equiv':'Content-Type' content="text/html; charset=utf-8")  
        title('My page')  
    }  
    body {  
        p('This is an example of HTML contents')  
    }  
}
```

## 19.4 FreeMarker

FreeMarker是一种模板语言，可以用作Spring MVC应用程序中的视图技术。更多关于模板语言的信息，请点击站点 [FreeMarker](#)。

### 19.4.1 依赖

您的Web应用程序需要包含 `freemarker-2.x.jar` 才能使用FreeMarker。通常这包含在 `WEB-INF/lib` 文件夹中，其中jar保证被Java EE服务器找到并添加到您的应用程序的类路径中。当然，假设你已经在你的 '`WEB-INF/lib`' 目录下有了 `spring-webmvc.jar` !

## 19.4.2 上下文配置

通过将相关的configurer bean定义添加到您的 `'*-servlet.xml'` 来初始化一个合适的配置，如下所示：

```
<!-- 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>
```



对于非web应用程序，将 `FreeMarkerConfigurationFactoryBean` 添加到您的应用程序上下文定义文件中。

### 19.4.3 创建模板

您的模板需要存储在上面显示的 `FreeMarkerConfigurer` 指定的目录中。如果使用突出显示的视图解析器，那么逻辑视图名称与模板文件名称的关联类似于用于JSP的 `InternalResourceViewResolver`。因此，如果您的控制器返回一个包含“welcome”视图名称的 `ModelAndView` 对象，则解析器将查找 `/WEB-INF/freemarker/welcome.ftl` 模板。

## 19.4.4 高级FreeMarker配置

通过在 `FreeMarkerConfigurer` bean 上设置合适的 `bean` 属性，FreeMarker 的 `'Settings'` 和 `'SharedVariables'` 可以直接传递给 Spring 管理的 `FreeMarker Configuration` 对象。`freemarkerSettings` 属性需要一个 `java.util.Properties` 对象，`freemarkerVariables` 属性需要一个 `java.util.Map`。

```
<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"/>
```

有关适用于 `Configuration` 对象的设置和变量的详细信息，请参阅 [FreeMarker 文档](#)。

## 19.4.5 绑定支持和表单处理

Spring提供了一个用于JSP的标签库，其中包含（其中包含）`<spring:bind/>` 标签。此标记主要允许表单从表单支持对象显示值，并显示来自Web或业务层的 Validator 的失败验证结果。Spring还支持FreeMarker中的相同功能，还有其他方便的宏用于生成表单输入元素。

### 绑定macros

在这两种语言的 `spring-webmvc.jar` 文件中都保留了一组标准的macros，因此它们始终可供适当配置的应用程序使用。

在Spring库中定义的一些macros被认为是内部的（私有的），但macros定义中不存在这样的范围，使调用代码和用户模板的所有macros都可见。以下部分仅关注您需要从模板中直接调用的macros。如果您希望直接查看macros代码，则在

包 `org.springframework.web.servlet.view.freemarker` 中将该文件称为 `spring.ftl`。

### 简单绑定

在作为Spring MVC控制器表单视图的HTML表单（vm / ftl模板）中，可以使用与以下类似的代码绑定到字段值，并以与JSP等效的方式类似的方式显示每个输入字段的错误消息。下面显示了以前配置的 `personForm` 视图的示例代码：

```
<!-- 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 "myModelObject.name"/>
    <input type="text"
      name="${spring.status.expression}"
      value="${spring.status.value?html}/><br>
    <#list spring.status.errorMessages as error> <b>$[error]</b> <br> </#list>
    <br>
    ...
    <input type="submit" value="submit"/>
  </form>
  ...
</html>
```

`<@spring.bind>` 需要一个'path'参数，它由你的命令对象的名字组成（除非你在 `FormController` 属性中改变它，否则它将是'command'），后面跟着一个句点和命令中的字段名你想绑定的对象。也可以使用嵌套字段，如“`command.address.street`”。 `bind` macro采

用由 web.xml 中的ServletContext参数 defaultHtmlEscape 指定的默认HTML转义行为。

macro的可选形式称为 <@spring.bindEscaped> 接受第二个参数，并明确指定是否应该在状态错误消息或值中使用HTML转义。根据需要设置为true或false。额外的表单处理macro简化了HTML转义的使用，应尽可能使用这些macro。他们在下一节解释。

## 表单输入生成 macros

这两种语言的其他便利macros简化了绑定和表单生成（包括验证错误显示）。从来有必要使用这些macros来生成表单输入字段，并且可以将它们与简单的HTML混合并进行匹配，或者直接调用之前突出显示的弹簧绑定macros。

下面的可用macro表显示了每个VTL和FTL定义以及每个参数列表。

**Table 19.1. Table of macro definitions**

macro	FTL 定义	消息	(根据代码参数输出字符串)
	<@spring.message code/>	<b>messageText</b> (根据代码参数从资源束输出一个字符串，退回到默认参数的值)	<@spring.message code, text/>
	<b>url</b> (使用应用程序的上下文根前缀相对URL)	<@spring.url relativeUrl/>	<b>formInput</b> (用于输入的标准输入)
	<@spring.formInput path, attributes, fieldType/>	<b>formHiddenInput</b> * (用于提交非用户输入的隐藏输入字段)	<@spring.formInput path, attribute
	<b>formPasswordInput</b> * (用于收集密码的标准输入字段。请注意，不会在此类型的字段中填充任何值)	<@spring.formPasswordInput path, attributes/>	<b>formTextarea</b> 用于收集长文本输入)
	<@spring.formTextarea path, attributes/>	<b>formSingleSelect</b> (下拉框选项允许选择单个所需的值)	<@spring.formTextarea path, options,
	<b>formMultiSelect</b> (允许用户选择0个或更多值的选项列表框)	<@spring.formMultiSelect path, options, attributes/>	<b>formRadioButton</b> 单选按钮允许从一行单次选择)
	<@spring.formRadioButtons path, options separator, attributes/>	<b>formCheckboxes</b> (一组允许选择0个或更多值的复选框)	<@spring.formRadioButtons path, options, attributes/>
	<b>formCheckbox</b> (a single checkbox)	<@spring.formCheckbox path, attributes/>	<b>showErrors</b> (字段的验证错误)

- 在FTL (FreeMarker) 中，这两个macro实际上并不是必需的，因为您可以使用常规 formInput 宏指定“hidden”或“password”作为 fieldType 参数的值。

上述任何一个macro的参数都有一致的含义：

- **path:** 要绑定的字段的名称（即“`command.name`”）
- **选项：**可以在输入字段中选择的所有可用值的映射。映射关键字表示将从表单中返回并绑定到命令对象的值。映射存储在键上的映射对象是表单上显示给用户的标签，可能与表单发回的相应值不同。通常这种映射是由控制器提供的参考数据。任何Map实现可以根据需求的行为选择。对于严格排序的映射，可以使用 `SortedMap` （如带有合适的 `Comparator` 的 `TreeMap`），对于任何应该以插入顺序返回值的Maps，请使用 `LinkedHashMap` 或Commons-Collection中的 `LinkedMap`。
- **分隔符：**其中多个选项可用作离散元素（单选按钮或复选框），用于分隔列表中的每一个（即“`<br>`”）的字符序列。
- **属性：**包含在HTML标签本身内的任意标签或文本的附加字符串。这个字符串被macro指令回显。例如，在`textarea`字段中您可以将属性设置为“`rows = "5" cols = "60"`或者可以传递样式信息，例如`'style="border:1px solid silver'"`。
- **classOrStyle :**对于`showErrors`宏，包含每个错误的`span`标签将使用的CSS类的名称。如果没有提供信息（或值为空），那么错误将被包裹在`<b></b>`标签中。

这些宏的例子在FTL中列出了一些，在VTL中列出了一些。在两种语言之间存在使用差异的地方，他们在笔记中解释。

### 输入的值

`formInput`宏接受路径参数（`command.name`）和一个在上例中为空的附加属性参数。该宏以及所有其他表单生成宏，对路径参数执行隐式弹簧绑定。绑定保持有效，直到发生新的绑定为止，所以`showErrors`宏不需要再次传递路径参数 - 它只是在绑定上次创建的任何字段上运行。

`showErrors`宏需要一个分隔符参数（将用于分隔给定字段上的多个错误的字符），并且还接受第二个参数，这次是类名称或样式属性。请注意，FreeMarker能够为`attributes`参数指定默认值。

```
<@spring.formInput "command.name"/>
<@spring.showErrors "<br>"/>
```

输出显示在生成名称字段的表单片段中，并且在字段中提交表单时没有值时显示验证错误。验证通过Spring的验证框架进行。

生成的HTML如下所示：

```
Name:  
<input type="text" name="name" value="">  
<br>  
    <b>required</b>  
<br>  
<br>
```

`formTextarea`宏与`formInput`宏的工作方式相同，并接受相同的参数列表。通常，第二个参数（属性）将用于传递文本区域的样式信息或行和列属性。

### 选择字段

四个选择字段宏可用于在HTML表单中生成常见的UI值选择输入。

- `formSingleSelect`
- `formMultiSelect`
- `formRadioButtons`
- `formCheckboxes`

四个宏中的每一个都接受一个包含表单字段值的选项的Map，以及与该值相对应的标签。值和标签可以是相同的。

FTL中的单选按钮示例如下。表单支持对象为该字段指定默认值'London'，因此不需要验证。当表单呈现时，整个城市列表将作为模型中的参考数据以'cityMap'名称提供。

```
...  
Town:  
<@spring.formRadioButtons "command.address.town", cityMap, ""/><br><br>
```

这将呈现一行单选按钮，每个城市地图中的值使用分隔符""。没有提供额外的属性（缺少宏的最后一个参数）。`cityMap`对地图中的每个键值对使用相同的字符串。地图的键是表单实际提交的POST请求参数，`map`值是用户看到的标签。在上面的例子中，给定一个三个知名城市的列表，并在窗体支持对象中使用一个默认值，HTML将会是

```
Town:  
<input type="radio" name="address.town" value="London">London</input>  
<input type="radio" name="address.town" value="Paris" checked="checked">Paris</input>  
<input type="radio" name="address.town" value="New York">New York</input>
```

例如，如果您的应用程序希望通过内部代码处理城市，则可以使用适当的键（如下面的示例）创建代码`map`。

```

protected Map<String, String> referenceData(HttpServletRequest request) throws Exception {
    Map<String, String> cityMap = new LinkedHashMap<>();
    cityMap.put("LDN", "London");
    cityMap.put("PRS", "Paris");
    cityMap.put("NYC", "New York");

    Map<String, String> model = new HashMap<>();
    model.put("cityMap", cityMap);
    return model;
}

```

现在代码将产生输出，其中无线电值是相关的代码，但是用户仍然看到更多用户友好的城市名称。

```

Town:
<input type="radio" name="address.town" value="LDN">London</input>
<input type="radio" name="address.town" value="PRS" checked="checked">Paris</input>
<input type="radio" name="address.town" value="NYC">New York</input>

```

## HTML转义和符合XHTML标准

上面的表单宏的默认使用将导致符合HTML 4.01的HTML标记，并使用Spring绑定支持所使用的web.xml中定义的HTML转义的默认值。为了使标记符合XHTML或覆盖默认的HTML转义值，您可以在模板中指定两个变量（或者在您的模型中，您的模板中可以看到它们）。在模板中指定它们的好处是可以在模板处理中稍后将其更改为不同的值，以便为表单中的不同字段提供不同的行为。

要切换到标记的XHTML合规性，请为名为xhtmlCompliant的模型/上下文变量指定值“true”：

```

<!-- for FreeMarker -->
<#assign xhtmlCompliant = true in spring>

```

在处理此指令后，由Spring宏生成的任何标签现在都将符合XHTML标准。

以类似的方式，每个字段可以指定HTML转义：

```

<!-- 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 -->

```



## 19.5 JSP & JSTL

Spring为JSP和JSTL视图提供了一些开箱即用的解决方案。使用JSP或JSTL是使用 `WebApplicationContext` 中定义的普通视图解析器完成的。此外，当然你需要编写一些实际呈现视图的JSP。



设置你的应用程序使用JSTL是一个常见的错误来源，主要是由于对不同的servlet规范，JSP和JSTL版本号，它们是什么意思以及如何正确地声明taglibs造成混淆。文章[How to Reference and Use JSTL in your Web Application](#)提供了常见陷阱的有用指南以及如何避免它们。请注意，从Spring 3.0开始，受支持的最小servlet版本是2.4（JSP 2.0和JSTL 1.1），这在一定程度上减少了混淆的范围。

## 19.5.1 视图解析

就像其他任何与 Spring 集成的视图技术一样，对于 JSP，您需要一个视图解析器来解析视图。在使用 JSP 进行开发时，最常用的视图解析器是 `InternalResourceViewResolver` 和 `ResourceBundleViewResolver`。两者都在 `WebApplicationContext` 中声明：

```
<!-- the ResourceBundleViewResolver -->
<bean id="viewResolver" class="org.springframework.web.servlet.view.ResourceBundleView
Resolver">
    <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
```

正如你所看到的，`ResourceBundleViewResolver` 需要一个属性文件来定义映射到 1) 类和 2) URL 的视图名称。使用 `ResourceBundleViewResolver`，您可以只使用一个解析器混合不同类型的视图。

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceVi
ewResolver">
    <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>
    <property name="prefix" value="/WEB-INF/jsp//>
    <property name="suffix" value=".jsp"/>
</bean>
```

如上所述，可以将 `InternalResourceBundleViewResolver` 配置为使用 JSP。作为最佳做法，我们强烈建议将您的 JSP 文件放置在 'WEB-INF' 目录下的目录中，这样客户端就不能直接访问了。

### 19.5.2 'Plain-old' JSPs versus JSTL

当使用Java标准标签库时，必须使用特殊的视图类 `JstlView`，因为JSTL需要做一些准备工作，比如I18N特性才能工作。

### 19.5.3 Additional tags facilitating development

Spring provides data binding of request parameters to command objects as described in earlier chapters. To facilitate the development of JSP pages in combination with those data binding features, Spring provides a few tags that make things even easier. All Spring tags have `_HTML escaping_features` to enable or disable escaping of characters.

The tag library descriptor (TLD) is included in the `spring-webmvc.jar`. Further information about the individual tags can be found in the appendix entitled[???](#).

Spring提供了请求参数与命令对象的数据绑定，如前面章节所述。为了促进JSP页面的开发并结合这些数据绑定功能，Spring提供了一些使事情变得更简单的标签。所有Spring标签都有`_HTML escaping_features`来启用或禁用字符转义。

标签库描述符（TLD）包含在 `spring-webmvc.jar` 中。关于单个标签的更多信息可以在名为[???](#)的附录中找到。

## 19.5.4 使用Spring的表单标签库

从版本2.0开始，Spring提供了一套全面的数据绑定感知标签，用于处理使用JSP和Spring Web MVC的表单元素。每个标签都提供对其对应的HTML标签对应的一组属性的支持，使标签变得熟悉和直观地使用。标签生成的HTML符合HTML 4.01 / XHTML 1.0。

与其他form / input标签库不同，Spring的form标签库与Spring Web MVC集成，使标签可以访问控制器处理的命令对象和引用数据。正如您在下面的示例中所看到的，表单标签使得JSP更易于开发，读取和维护。

让我们通过表单标签，看看每个标签是如何使用的例子。我们已经包括生成的HTML片段，其中某些标签需要进一步的评论。

### 配置

表单标签库捆绑在 spring-webmvc.jar 中。库描述符称为 spring-form.tld。

要使用此库中的标签，请将以下指令添加到JSP页面的顶部：

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

其中 form 是要用于来自此库的标记的标记名称前缀。

### form标签

这个标签呈现一个HTML的'form'标签，并暴露一个绑定路径到内部标签进行绑定。它将命令对象放在 PageContext 中，以便命令对象可以被内部标签访问。这个库中的所有其他标签都是标签的嵌套标签。

假设我们有一个名为 user 的域对象。它是一个具有诸如 firstName 和 lastName 之类的属性的JavaBean。我们将使用它作为我们的form控制器返回 form.jsp 的形式支持对象。下面是 form.jsp 的一个例子：

```
<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>
```

The `firstName` and `lastName` values are retrieved from the command object placed in the `PageContext` by the page controller. Keep reading to see more complex examples of how inner tags are used with the `form` tag.

The generated HTML looks like a standard form:

```
<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>
```

The preceding JSP assumes that the variable name of the form backing object is '`command`' . If you have put the form backing object into the model under another name (definitely a best practice), then you can bind the form to the named variable like so:

```
<form:form modelAttribute="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>
```

## The input tag

This tag renders an HTML 'input' tag using the bound value and type='text' by default. For an example of this tag, see [the section called “The form tag”](#). Starting with Spring 3.1 you can use other types such HTML5-specific types like 'email', 'tel', 'date', and others.

## The checkbox tag

This tag renders an HTML 'input' tag with type 'checkbox'.

Let's assume our `User` has preferences such as newsletter subscription and a list of hobbies. Below is an example of the `Preferences` class:

```
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;  
    }  
}
```

The `form.jsp` would look like:

```

<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>
            <%-- Approach 2: Property is of an array or of type java.util.Collection - -%>
            <td>
                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>
            <%-- Approach 3: Property is of type java.lang.Object --%>
            <td>
                Magic: <form:checkbox path="preferences.favouriteWord" value="Magic"/>
            </td>
        </tr>
    </table>
</form:form>

```

There are 3 approaches to the `checkbox` tag which should meet all your checkbox needs.

- Approach One - When the bound value is of type `java.lang.Boolean` , the `input(checkbox)` is marked as 'checked' if the bound value is `true` . The `value` attribute corresponds to the resolved value of the `setValue(Object)` value property.
- Approach Two - When the bound value is of type `array` or `java.util.Collection` , the `input(checkbox)` is marked as 'checked' if the configured `setValue(Object)` value is present in the bound `collection` .
- Approach Three - For any other bound value type, the `input(checkbox)` is marked as 'checked' if the configured `setValue(Object)` is equal to the bound value.

Note that regardless of the approach, the same HTML structure is generated. Below is an HTML snippet of some checkboxes:

```

<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>

```

What you might not expect to see is the additional hidden field after each checkbox. When a checkbox in an HTML page is `not checked`, its value will not be sent to the server as part of the HTTP request parameters once the form is submitted, so we need a workaround for this quirk in HTML in order for Spring form data binding to work. The `checkbox` tag follows the existing Spring convention of including a hidden parameter prefixed by an underscore ("\_") for each checkbox. By doing this, you are effectively telling Spring that "the checkbox was visible in the form and I want my object to which the form data will be bound to reflect the state of the checkbox no matter what".

## The checkboxes tag

This tag renders multiple HTML 'input' tags with type 'checkbox'.

Building on the example from the previous `checkbox` tag section. Sometimes you prefer not to have to list all the possible hobbies in your JSP page. You would rather provide a list at runtime of the available options and pass that in to the tag. That is the purpose of the `checkboxes` tag. You pass in an `Array`, a `List` or a `Map` containing the available options in the "items" property. Typically the bound property is a collection so it can hold multiple values selected by the user. Below is an example of the JSP using this tag:

```

<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>

```



This example assumes that the "interestList" is a `List` available as a model attribute containing strings of the values to be selected from. In the case where you use a `Map`, the map entry key will be used as the value and the map entry's value will be used as the label to be displayed. You can also use a custom object where you can provide the property names for the value using "itemValue" and the label using "itemLabel".

## The radiobutton tag

This tag renders an HTML 'input' tag with type 'radio'.

A typical usage pattern will involve multiple tag instances bound to the same property but with different values.

```

<tr>
    <td>Sex:</td>
    <td>
        Male: <form:radio button path="sex" value="M"/> <br/>
        Female: <form:radio button path="sex" value="F"/>
    </td>
</tr>

```

## The radiobuttons tag

This tag renders multiple HTML 'input' tags with type 'radio'.

Just like the `checkboxes` tag above, you might want to pass in the available options as a runtime variable. For this usage you would use the `radiobuttons` tag. You pass in an `Array`, a `List` or a `Map` containing the available options in the "items" property. In the case where you use a `Map`, the map entry key will be used as the value and the map entry's value will be used as the label to be displayed. You can also use a custom object where you can provide the property names for the value using "itemValue" and the label using "itemLabel".

```
<tr>
    <td>Sex:</td>
    <td><form:radioButtons path="sex" items="${sexOptions}" /></td>
</tr>
```

## The password tag

This tag renders an HTML 'input' tag with type 'password' using the bound value.

```
<tr>
    <td>Password:</td>
    <td>
        <form:password path="password"/>
    </td>
</tr>
```

Please note that by default, the password value is not shown. If you do want the password value to be shown, then set the value of the 'showPassword' attribute to true, like so.

```
<tr>
    <td>Password:</td>
    <td>
        <form:password path="password" value="^76525bvHGq" showPassword="true"/>
    </td>
</tr>
```

## The select tag

This tag renders an HTML 'select' element. It supports data binding to the selected option as well as the use of nested `option` and `options` tags.

Let's assume a `User` has a list of skills.

```
<tr>
    <td>Skills:</td>
    <td><form:select path="skills" items="${skills}" /></td>
</tr>
```

If the `User`'s skill were in Herbology, the HTML source of the 'Skills' row would look like:

```
<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>
```

## The option tag

This tag renders an HTML 'option'. It sets 'selected' as appropriate based on the bound value.

```
<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>
```

If the user's house was in Gryffindor, the HTML source of the 'House' row would look like:

```
<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>
```

## The options tag

This tag renders a list of HTML 'option' tags. It sets the 'selected' attribute as appropriate based on the bound value.

```
<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>
```

If the `user` lived in the UK, the HTML source of the 'Country' row would look like:

```
<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>
```

As the example shows, the combined usage of an `option` tag with the `options` tag generates the same standard HTML, but allows you to explicitly specify a value in the JSP that is for display only (where it belongs) such as the default string in the example: "-- Please Select".

The `items` attribute is typically populated with a collection or array of item objects. `itemValue` and `itemLabel` simply refer to bean properties of those item objects, if specified; otherwise, the item objects themselves will be stringified. Alternatively, you may specify a `Map` of items, in which case the map keys are interpreted as option values and the map values correspond to option labels. If `itemValue` and/or `itemLabel` happen to be specified as well, the item value property will apply to the map key and the item label property will apply to the map value.

## The `textarea` tag

This tag renders an HTML 'textarea'.

```
<tr>
    <td>Notes:</td>
    <td><form:textarea path="notes" rows="3" cols="20"/></td>
    <td><form:errors path="notes"/></td>
</tr>
```

## The hidden tag

This tag renders an HTML 'input' tag with type 'hidden' using the bound value. To submit an unbound hidden value, use the HTML `input` tag with type 'hidden'.

```
<form:hidden path="house"/>
```

If we choose to submit the 'house' value as a hidden one, the HTML would look like:

```
<input name="house" type="hidden" value="Gryffindor"/>
```

## The errors tag

This tag renders field errors in an HTML 'span' tag. It provides access to the errors created in your controller or those that were created by any validators associated with your controller.

Let's assume we want to display all error messages for the `firstName` and `lastName` fields once we submit the form. We have a validator for instances of the `User` class called `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.");
    }
}
```

The `form.jsp` would look like:

```

<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>

```

If we submit a form with empty values in the `firstName` and `lastName` fields, this is what the HTML would look like:

```

<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>

```

What if we want to display the entire list of errors for a given page? The example below shows that the `errors` tag also supports some basic wildcarding functionality.

- `path="*"` - displays all errors
- `path="lastName"` - displays all errors associated with the `lastName` field
- if `path` is omitted - object errors only are displayed

The example below will display a list of errors at the top of the page, followed by field-specific errors next to the fields:

```
<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>
```

The HTML would look like:

```

<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>
    </table>
</form>

```

## HTTP Method Conversion

A key principle of REST is the use of the Uniform Interface. This means that all resources (URLs) can be manipulated using the same four HTTP methods: GET, PUT, POST, and DELETE. For each method, the HTTP specification defines the exact semantics. For instance, a GET should always be a safe operation, meaning that it has no side effects, and a PUT or DELETE should be idempotent, meaning that you can repeat these operations over and over again, but the end result should be the same. While HTTP defines these four methods, HTML only supports two: GET and POST. Fortunately, there are two possible workarounds: you can either use JavaScript to do your PUT or DELETE, or simply do a POST with the 'real' method as an additional parameter (modeled as a hidden input field in an HTML form). This latter trick is what Spring's `HiddenHttpMethodFilter` does. This filter is a plain Servlet Filter and therefore it can be used in combination with any web framework (not just Spring MVC). Simply add this filter to your `web.xml`, and a POST with a hidden `_method` parameter will be converted into the corresponding HTTP method request.

To support HTTP method conversion the Spring MVC form tag was updated to support setting the HTTP method. For example, the following snippet taken from the updated Petclinic sample

```
<form:form method="delete">
    <p class="submit"><input type="submit" value="Delete Pet"/></p>
</form:form>
```

This will actually perform an HTTP POST, with the 'real' DELETE method hidden behind a request parameter, to be picked up by the `HiddenHttpMethodFilter`, as defined in web.xml:

```
<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>
```

The corresponding `@Controller` method is shown below:

```
@RequestMapping(method = RequestMethod.DELETE)
public String deletePet(@PathVariable int ownerId, @PathVariable int petId) {
    this.clinic.deletePet(petId);
    return "redirect:/owners/" + ownerId;
}
```

## HTML5 Tags

Starting with Spring 3, the Spring form tag library allows entering dynamic attributes, which means you can enter any HTML5 specific attributes.

In Spring 3.1, the form input tag supports entering a type attribute other than 'text'. This is intended to allow rendering new HTML5 specific input types such as 'email', 'date', 'range', and others. Note that entering type='text' is not required since 'text' is the default type.

## 19.6 Script templates

It is possible to integrate any templating library running on top of a JSR-223 script engine in web applications using Spring. The following describes in a broad way how to do this. The script engine must implement both `ScriptEngine` and `Invocable` interfaces.

It has been tested with:

- [Handlebars](#) running on [Nashorn](#)
- [Mustache](#) running on [Nashorn](#)
- [React](#) running on [Nashorn](#)
- [EJS](#) running on [Nashorn](#)
- [ERB](#) running on [JRuby](#)
- [String templates](#) running on [Python](#)

## 19.6.1 Dependencies

To be able to use script templates integration, you need to have available in your classpath the script engine:

- [Nashorn](#)Javascript engine is provided builtin with Java 8+. Using the latest update release available is highly recommended.
- [Rhino](#)Javascript engine is provided builtin with Java 6 and Java 7. Please notice that using Rhino is not recommended since it does not support running most template engines.
- [JRuby](#)dependency should be added in order to get Ruby support.
- [Jython](#)dependency should be added in order to get Python support.

You should also need to add dependencies for your script based template engine. For example, for Javascript you can use[WebJar](#)s to add Maven/Gradle dependencies in order to make your javascript libraries available in the classpath.

## 19.6.2 How to integrate script based templating

To be able to use script templates, you have to configure it in order to specify various parameters like the script engine to use, the script files to load and what function should be called to render the templates. This is done thanks to a `ScriptTemplateConfigurer` bean and optional script files.

For example, in order to render Mustache templates thanks to the Nashorn Javascript engine provided with Java 8+, you should declare the following configuration:

```
@Configuration
@EnableWebMvc
public class MustacheConfig extends WebMvcConfigurerAdapter {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.scriptTemplate();
    }

    @Bean
    public ScriptTemplateConfigurer configurer() {
        ScriptTemplateConfigurer configurer = new ScriptTemplateConfigurer();
        configurer.setEngineName("nashorn");
        configurer.setScripts("mustache.js");
        configurer.setRenderObject("Mustache");
        configurer.setRenderFunction("render");
        return configurer;
    }
}
```

The XML counterpart using MVC namespace is:

```
<mvc:annotation-driven/>

<mvc:view-resolvers>
    <mvc:script-template/>
</mvc:view-resolvers>

<mvc:script-template-configure engine-name="nashorn" render-object="Mustache" render-
function="render">
    <mvc:script location="mustache.js"/>
</mvc:script-template-configure>
```

The controller is exactly what you should expect:

```

@Controller
public class SampleController {

    @RequestMapping
    public ModelAndView test() {
        ModelAndView mav = new ModelAndView();
        mav.addObject("title", "Sample title"). addObject("body", "Sample body");
        mav.setViewName("template.html");
        return mav;
    }
}

```

And the Mustache template is:

```

<html>
  <head>
    <title>{{title}}</title>
  </head>
  <body>
    <p>{{body}}</p>
  </body>
</html>

```

The render function is called with the following parameters:

- `String template` : the template content
- `Map model` : the view model
- `String url` : the template url (since 4.2.2)

`Mustache.render()` is natively compatible with this signature, so you can call it directly.

If your templating technology requires some customization, you may provide a script that implements a custom render function. For example, `Handlerbars` needs to compile templates before using them, and requires a `polyfill` in order to emulate some browser facilities not available in the server-side script engine.

```

@Configuration
@EnableWebMvc
public class MustacheConfig extends WebMvcConfigurerAdapter {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.scriptTemplate();
    }

    @Bean
    public ScriptTemplateConfigurer configurer() {
        ScriptTemplateConfigurer configurer = new ScriptTemplateConfigurer();
        configurer.setEngineName("nashorn");
        configurer.setScripts("polyfill.js", "handlebars.js", "render.js");
        configurer.setRenderFunction("render");
        configurer.setSharedEngine(false);
        return configurer;
    }
}

```



Setting the `sharedEngine` property to `false` is required when using non thread-safe script engines with templating libraries not designed for concurrency, like Handlebars or React running on Nashorn for example. In that case, Java 8u60 or greater is required due to [this bug](#).

`polyfill.js` only defines the `window` object needed by Handlebars to run properly:

```
var window = {};
```

This basic `render.js` implementation compiles the template before using it. A production ready implementation should also store and reused cached templates / pre-compiled templates. This can be done on the script side, as well as any customization you need (managing template engine configuration for example).

```

function render(template, model) {
    var compiledTemplate = Handlebars.compile(template);
    return compiledTemplate(model);
}

```

Check out Spring script templates unit tests ([java,resources](#)) for more configuration examples.



## 19.7 XML Marshalling View

The `MarshallingView` uses an XML `Marshaller` defined in the `org.springframework.oxm` package to render the response content as XML. The object to be marshalled can be set explicitly using `MarshallingView`'s `modelKey` bean property. Alternatively, the view will iterate over all model properties and marshal the first type that is supported by the `Marshaller`. For more information on the functionality in the `org.springframework.oxm` package refer to the chapter [Marshalling XML using O/X Mappers](#).

## 19.8 Tiles

It is possible to integrate Tiles - just as any other view technology - in web applications using Spring. The following describes in a broad way how to do this.



This section focuses on Spring's support for Tiles v3 in the `org.springframework.web.servlet.view.tiles3` package.

## 19.8.1 Dependencies

To be able to use Tiles, you have to add a dependency on Tiles version 3.0.1 or higher and its transitive dependencies to your project.

## 19.8.2 How to integrate Tiles

To be able to use Tiles, you have to configure it using files containing definitions (for basic information on definitions and other Tiles concepts, please have a look at <http://tiles.apache.org>). In Spring this is done using the `TilesConfigurer`. Have a look at the following piece of example ApplicationContext configuration:

```
<bean id="tilesConfigurer" class="org.springframework.web.servlet.view.tiles3.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>
```

As you can see, there are five files containing definitions, which are all located in the '`WEB-INF/defs`' directory. At initialization of the `WebApplicationContext`, the files will be loaded and the definitions factory will be initialized. After that has been done, the Tiles includes in the definition files can be used as views within your Spring web application. To be able to use the views you have to have a `ViewResolver` just as with any other view technology used with Spring. Below you can find two possibilities, the `UrlBasedViewResolver` and the `ResourceBundleViewResolver`.

You can specify locale specific Tiles definitions by adding an underscore and then the locale. For example:

```
<bean id="tilesConfigurer" class="org.springframework.web.servlet.view.tiles3.TilesConfigurer">
    <property name="definitions">
        <list>
            <value>/WEB-INF/defs/tiles.xml</value>
            <value>/WEB-INF/defs/tiles_fr_FR.xml</value>
        </list>
    </property>
</bean>
```

With this configuration, `tiles_fr_FR.xml` will be used for requests with the `fr_FR` locale, and `tiles.xml` will be used by default.



Since underscores are used to indicate locales, it is recommended to avoid using them otherwise in the file names for Tiles definitions.

## UrlBasedViewResolver

The `UrlBasedViewResolver` instantiates the given `viewClass` for each view it has to resolve.

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.UrlBasedViewResolver">
    <property name="viewClass" value="org.springframework.web.servlet.view.tiles3.TilesView"/>
</bean>
```

## ResourceBundleViewResolver

The `ResourceBundleViewResolver` has to be provided with a property file containing viewnames and viewclasses the resolver can use:

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
    <property name="basename" value="views"/>
</bean>
```

```
...
welcomeView.(class)=org.springframework.web.servlet.view.tiles3.TilesView
welcomeView.url=welcome (this is the name of a Tiles definition)

vetsView.(class)=org.springframework.web.servlet.view.tiles3.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
...
```

As you can see, when using the `ResourceBundleViewResolver`, you can easily mix different view technologies.

Note that the `TilesView` class supports JSTL (the JSP Standard Tag Library) out of the box.

## SimpleSpringPreparerFactory and SpringBeanPreparerFactory

As an advanced feature, Spring also supports two special `Tiles PreparerFactory` implementations. Check out the Tiles documentation for details on how to use `ViewPreparer` references in your Tiles definition files.

Specify `SimpleSpringPreparerFactory` to autowire `ViewPreparer` instances based on specified preparer classes, applying Spring's container callbacks as well as applying configured Spring BeanPostProcessors. If Spring's context-wide annotation-config has been activated, annotations in `ViewPreparer` classes will be automatically detected and applied. Note that this expects `preparer_classes_` in the Tiles definition files, just like the default `PreparerFactory` does.

Specify `SpringBeanPreparerFactory` to operate on specified `preparer_names_` instead of classes, obtaining the corresponding Spring bean from the DispatcherServlet's application context. The full bean creation process will be in the control of the Spring application context in this case, allowing for the use of explicit dependency injection configuration, scoped beans etc. Note that you need to define one Spring bean definition per preparer name (as used in your Tiles definitions).

```
<bean id="tilesConfigurer" class="org.springframework.web.servlet.view.tiles3.TilesCon
figurer">
    <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.tiles3.SpringBeanPreparerFacto
ry"/>
</bean>
```

## 19.9 XSLT

XSLT is a transformation language for XML and is popular as a view technology within web applications. XSLT can be a good choice as a view technology if your application naturally deals with XML, or if your model can easily be converted to XML. The following section shows how to produce an XML document as model data and have it transformed with XSLT in a Spring Web MVC application.

## 19.9.1 My First Words

This example is a trivial Spring application that creates a list of words in the `Controller` and adds them to the model map. The map is returned along with the view name of our XSLT view. See [Section 18.3, “Implementing Controllers”](#) for details of Spring Web MVC’s `Controller` interface. The XSLT Controller will turn the list of words into a simple XML document ready for transformation.

### Bean definitions

Configuration is standard for a simple Spring application. The MVC configuration has to define a `XsltViewResolver` bean and regular MVC annotation configuration.

```
@EnableWebMvc
@ComponentScan
@Configuration
public class WebConfig extends WebMvcConfigurerAdapter {

    @Bean
    public XsltViewResolver xsltViewResolver() {
        XsltViewResolver viewResolver = new XsltViewResolver();
        viewResolver.setPrefix("/WEB-INF/xsl/");
        viewResolver.setSuffix(".xslt");
        return viewResolver;
    }

}
```

And we need a Controller that encapsulates our word generation logic.

### Standard MVC controller code

The controller logic is encapsulated in a `@Controller` class, with the handler method being defined like so...

```

@Controller
public class XsltController {

    @RequestMapping("/")
    public String home(Model model) throws Exception {

        Document document = DocumentBuilderFactory.newInstance().newDocumentBuilder().
newDocument();
        Element root = document.createElement("wordList");

        List<String> words = Arrays.asList("Hello", "Spring", "Framework");
        for (String word : words) {
            Element wordNode = document.createElement("word");
            Text textNode = document.createTextNode(word);
            wordNode.appendChild(textNode);
            root.appendChild(wordNode);
        }

        model.addAttribute("wordList", root);
        return "home";
    }
}

```

So far we've only created a DOM document and added it to the Model map. Note that you can also load an XML file as a `Resource` and use it instead of a custom DOM document.

Of course, there are software packages available that will automatically 'domify' an object graph, but within Spring, you have complete flexibility to create the DOM from your model in any way you choose. This prevents the transformation of XML playing too great a part in the structure of your model data which is a danger when using tools to manage the domification process.

Next, `XsltViewResolver` will resolve the "home" XSLT template file and merge the DOM document into it to generate our view.

## Document transformation

Finally, the `XsltViewResolver` will resolve the "home" XSLT template file and merge the DOM document into it to generate our view. As shown in the `XsltViewResolver` configuration, XSLT templates live in the war file in the '`WEB-INF/xsl`' directory and end with a "`xslt`" file extension.

```

<?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>
                <ul>
                    <xsl:apply-templates/>
                </ul>
            </body>
        </html>
    </xsl:template>

    <xsl:template match="word">
        <li><xsl:value-of select="."/>/</li>
    </xsl:template>

</xsl:stylesheet>

```

This is rendered as:

```

<html>
    <head>
        <META http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Hello!</title>
    </head>
    <body>
        <h1>My First Words</h1>
        <ul>
            <li>Hello</li>
            <li>Spring</li>
            <li>Framework</li>
        </ul>
    </body>
</html>

```

## 19.10.1 Introduction

Returning an HTML page isn't always the best way for the user to view the model output, and Spring makes it simple to generate a PDF document or an Excel spreadsheet dynamically from the model data. The document is the view and will be streamed from the server with the correct content type to (hopefully) enable the client PC to run their spreadsheet or PDF viewer application in response.

In order to use Excel views, you need to add the 'poi' library to your classpath, and for PDF generation, the iText library.

## 19.10.2 Configuration and setup

Document based views are handled in an almost identical fashion to XSLT views, and the following sections build upon the previous one by demonstrating how the same controller used in the XSLT example is invoked to render the same model as both a PDF document and an Excel spreadsheet (which can also be viewed or manipulated in Open Office).

### Document view definitions

First, let's amend the views.properties file (or xml equivalent) and add a simple view definition for both document types. The entire file now looks like this with the XSLT view shown from earlier:

```
home.(class)=xslt.HomePage
home.stylesheetLocation=/WEB-INF/xsl/home.xslt
home.root=words

xl.(class)=excel.HomePage

pdf.(class)=pdf.HomePage
```

*If you want to start with a template spreadsheet or a fillable PDF form to add your model data to, specify the location as the 'url' property in the view definition*

### Controller code

The controller code we'll use remains exactly the same from the XSLT example earlier other than to change the name of the view to use. Of course, you could be clever and have this selected based on a URL parameter or some other logic - proof that Spring really is very good at decoupling the views from the controllers!

### Subclassing for Excel views

Exactly as we did for the XSLT example, we'll subclass suitable abstract classes in order to implement custom behavior in generating our output documents. For Excel, this involves writing a subclass of `org.springframework.web.servlet.view.document.AbstractExcelView` (for Excel files generated by POI)

or `org.springframework.web.servlet.view.document.AbstractJExcelView` (for JExcelApi-generated Excel files) and implementing the `buildExcelDocument()` method.

Here's the complete listing for our POI Excel view which displays the word list from the model map in consecutive rows of the first column of a new spreadsheet:

```
package excel;

// imports omitted for brevity

public class HomePage extends AbstractExcelView {

    protected void buildExcelDocument(Map model, HSSFWorkbook wb, HttpServletRequest r
eq,
        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));
        }
    }
}
```

And the following is a view generating the same Excel file, now using 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)));
        }
    }
}

```

Note the differences between the APIs. We've found that the JExcelApi is somewhat more intuitive, and furthermore, JExcelApi has slightly better image-handling capabilities. There have been memory problems with large Excel files when using JExcelApi however.

If you now amend the controller such that it returns `x1` as the name of the view (`return new ModelAndView("x1", map);`) and run your application again, you should find that the Excel spreadsheet is created and downloaded automatically when you request the same page as before.

## Subclassing for PDF views

The PDF version of the word list is even simpler. This time, the class extends `org.springframework.web.servlet.view.document.AbstractPdfView` and implements the `buildPdfDocument()` method as follows:

```
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)));
        }
    }
}
```

Once again, amend the controller to return the `pdf` view with `return new ModelAndView("pdf", map);`, and reload the URL in your application. This time a PDF document should appear listing each of the words in the model map.

## 19.11 Feed Views

Both `AbstractAtomFeedView` and `AbstractRssFeedView` inherit from the base class `AbstractFeedView` and are used to provide Atom and RSS Feed views respectfully. They are based on java.net's [ROME](#) project and are located in the package `org.springframework.web.servlet.view.feed`.

`AbstractAtomFeedView` requires you to implement the `buildFeedEntries()` method and optionally override the `buildFeedMetadata()` method (the default implementation is empty), as shown below.

```
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
    }

}
```

Similar requirements apply for implementing `AbstractRssFeedView`, as shown below.

```
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
    }

}
```

The `buildFeedItems()` and `buildFeedEntires()` methods pass in the HTTP request in case you need to access the Locale. The HTTP response is passed in only for the setting of cookies or other HTTP headers. The feed will automatically be written to the response object after the method returns.

For an example of creating an Atom view please refer to Alef Arendsen's Spring Team [Blogentry](#).

## 19.12 JSON Mapping View

The `MappingJackson2JsonView` uses the Jackson library's `ObjectMapper` to render the response content as JSON. By default, the entire contents of the model map (with the exception of framework-specific classes) will be encoded as JSON. For cases where the contents of the map need to be filtered, users may specify a specific set of model attributes to encode via the `RenderedAttributes` property. The `extractValueFromSingleKeyModel` property may also be used to have the value in single-key models extracted and serialized directly rather than as a map of model attributes.

JSON mapping can be customized as needed through the use of Jackson's provided annotations. When further control is needed, a custom `ObjectMapper` can be injected through the `ObjectMapper` property for cases where custom JSON serializers/deserializers need to be provided for specific types.

JSONP is supported and automatically enabled when the request has a query parameter named `jsonp` or `callback`. The JSONP query parameter name(s) could be customized through the `jsonpParameterNames` property.

## 19.13 XML Mapping View

The `MappingJackson2XmlView` uses the [Jackson XML extension](#)'s `XmlMapper` to render the response content as XML. If the model contains multiple entries, the object to be serialized should be set explicitly using the `modelKey` bean property. If the model contains a single entry, it will be serialized automatically.

XML mapping can be customized as needed through the use of JAXB or Jackson's provided annotations. When further control is needed, a custom `XmlMapper` can be injected through the `ObjectMapper` property for cases where custom XML serializers/deserializers need to be provided for specific types.