Spring5 中文参考指南

简介 1.1
前言 1.2
1. “Spring”代表什么 1.2.1
2. Spring和Spring框架的历史 1.2.2
3. 设计哲学 1.2.3
4. 反馈和贡献 1.2.4
5. 开始 1.2.5
核心技术 1.3
1. IoC容器 1.3.1
1.1 IoC容器和Beans介绍 1.3.1.1
1.2 IoC容器概述 1.3.1.2
1.3 Bean概述 1.3.1.3
1.4 依赖 1.3.1.4
1.5 Bean作用域 1.3.1.5
1.6 自定义Bean 1.3.1.6
1.7 Bean的继承 1.3.1.7
1.8 容器扩展点 1.3.1.8
1.9 基于注解的容器配置 1.3.1.9
1.10 类路径扫描和托管组件 1.3.1.10
1.11 使用JSR 330标准注解 1.3.1.11
1.12 基于Java的容器配置 1.3.1.12
1.13 环境抽象 1.3.1.13
1.14 注册LoadTimeWeaver 1.3.1.14
1.15 ApplicationContext的其他功能 1.3.1.15
1.16 BeanFactory 1.3.1.16
2.资源 1.3.2
2.1介绍 1.3.2.1
2.2资源接口 1.3.2.2
2.3内置资源实现 1.3.2.3
2.4ResourceLoader 1.3.2.4
2.5ResourceLoaderAware接口 1.3.2.5
2.6资源作为依赖 1.3.2.6
2.7应用程序上下文和资源路径 1.3.2.7
3.验证，数据绑定，和类型转换 1.3.3
3.1使用Spring Validator接口 1.3.3.1
3.2将代码解析为错误消息 1.3.3.2
3.3bean操作和BeanWrapper 1.3.3.3

3.4Spring类型转换 1.3.3.4
3.5Spring字段格式化 1.3.3.5
3.6配置全局Date和Time格式 1.3.3.6
3.7Spring验证 1.3.3.7
4.SpEL Spring表达式语言 1.3.4
4.1求值 1.3.4.1
4.2bean定义中的表达式 1.3.4.2
4.3语言引用 1.3.4.3
5.Spring AOP 1.3.5
5.1什么是AOP 1.3.5.1
5.1Spring AOP的能力和目标 1.3.5.2
5.3AOP代理 1.3.5.3
5.4@AspectJ 支持 1.3.5.4
5.5基于Schema的AOP支持 1.3.5.5
5.6选择要使用的AOP声明样式 1.3.5.6
5.7混合Aspect类型 1.3.5.7
5.8代理机制 1.3.5.8
5.9程序创建@AspectJ代理 1.3.5.9
5.10在Spring应用程序中使用AspectJ 1.3.5.10
5.11更多资源 1.3.5.11
6.Spring AOP APIs 1.3.6
6.1Pointcut API 1.3.6.1
6.2Advice API 1.3.6.2
6.3Advisor API 1.3.6.3
6.4使用ProxyFactoryBean来创建AOP代理 1.3.6.4
6.5简介的代理定义 1.3.6.5
6.6使用ProxyFactory创建AOP代理 1.3.6.6
6.7操作被通知的对象 1.3.6.7
6.8使用auto-proxy功能 1.3.6.8
6.9使用TargetSource的实现 1.3.6.9
6.10定义新的Advice Types 1.3.6.10
7.Null-safety 1.3.7
8.数据缓存和解码器 1.3.8
9.附录 1.3.9
9.1XML Schemas 1.3.9.1
9.2创建XML Schemas 1.3.9.2
测试 1.4
1.Spring测试介绍 1.4.1
2.单元测试 1.4.2
2.1Mock Objects 1.4.2.1

2.2单元测试支持类 1.4.2.2
3.集成测试 1.4.3
3.1概览 1.4.3.1
3.2集成测试的目的 1.4.3.2
3.3JDBC测试支持 1.4.3.3
3.4注解 1.4.3.4
3.5Spring TestContext框架 1.4.3.5
3.6Spring MVC测试框架 1.4.3.6
3.7WebTestClient 1.4.3.7
4.更多资源 1.4.4
数据访问 1.5
1.事务管理 1.5.1
2.DAO支持 1.5.2
3.JDBC 1.5.3
3.1选择JDBC数据库访问方法 1.5.3.1
3.2包层次结构 1.5.3.2
3.3使用JDBC核心类控制基本JDBC处理和错误处理 1.5.3.3
3.4控制数据库连接 1.5.3.4
3.5JDBC批处理操作 1.5.3.5
3.6使用SimpleJdbc 1.5.3.6
3.7将JDBC操作建模为Java对象 1.5.3.7
3.8参数和数据值处理的常见问题 1.5.3.8
3.9嵌入式数据库支持 1.5.3.9
3.10初始化数据源 1.5.3.10
4.ORM 1.5.4
4.1Spring ORM介绍 1.5.4.1
4.2ORM集成的一般注意事项 1.5.4.2
4.3Hibernate 1.5.4.3
4.4JPA 1.5.4.4
5.使用Object-XML映射封装XML 1.5.5
Web Servlet 1.6
1. Spring Web MVC 1.6.1
1.1 DispatcherServlet 1.6.1.1
1.2 Filters 1.6.1.2
1.3 Controllers注解 1.6.1.3
1.4 URI链接 1.6.1.4
1.5 异步请求 1.6.1.5
1.6 CORS 1.6.1.6
1.7 Web Security 1.6.1.7
1.8 HTTP Caching 1.6.1.8

1.9 View技术 1.6.1.9
1.10 MVC配置 1.6.1.10
1.11 HTTP/2 1.6.1.11
2. REST客户端 1.6.2
3. 测试 1.6.3
4. WebSockets 1.6.4
4.1 WebSocket介绍 1.6.4.1
4.2 WebSocket API 1.6.4.2
4.3 SockJS Fallback 1.6.4.3
4.4 STOMP 1.6.4.4
Web Reactive 1.7
1.Spring WebFlux 1.7.1
1.1 Overview 1.7.1.1
1.2 Reactive Core 1.7.1.2
1.3 DispatcherHandler 1.7.1.3
1.4 Annotated Controllers 1.7.1.4
1.5 Functional Endpoints 1.7.1.5
1.6 URI Links 1.7.1.6
1.7 CORS 1.7.1.7
1.8 Web Security 1.7.1.8
1.9 View Technologies 1.7.1.9
1.10 HTTP Caching 1.7.1.10
1.11 WebFlux Config 1.7.1.11
1.12 HTTP/2 1.7.1.12
2.WebClient 1.7.2
3.WebSockets 1.7.3
4.测试 1.7.4
5.Reactive库 1.7.5

更多教程请访问:www.flydean.com

简介

Spring5参考指南

Spring Framework Documentation 5.X.RELEASE 中文版本

如何开始阅读

可访问如下网站地址：

- https://docs.flydean.com/spring-framework-documentation5/
- 更多教程请参考 flydean的博客

意见、建议

如果有意见或者建议请指正 https://github.com/ddean2009/Spring-Framework-Documentation/issues

翻译计划

- [x] I. Spring Framework Documentation
- [x] II. Spring Framework Overview
- [x] III. Core Technologies
  - [x] 1.The IOC Container
  - [x] 2.Resources
  - [x] 3.Validation,Data Binding,and Type Conversion
  - [x] 4.SPEL
  - [x] 5.Aspect Oriented Programming with Spring
  - [x] 6.Spring AOP APIs
  - [x] 7.Null-safety
  - [x] 8.Data Buffers and Codecs
  - [x] 9.Appendix
- [x] IV. Testing
- [x] V. Data Access
- [x] VI. Web on Servlet Stack
- [x] VII. Web on Reactive Stack

更多教程请访问:www.flydean.com

许可

MIT

更多教程

更多教程请参考 flydean的博客

更多教程请访问:www.flydean.com

前言

Spring让创建企业级Java程序变得容易。他提供了企业级Java程序所需要的一切内容，包括对于Groovy和Kotlin这两种JVM语言的支持，以灵活的创建多种多样的企业架构。

在Spring Framework5.1中， Spring需要 JDK 8+的支持，最新的能支持的JDK版本是 JDK 11 LTS。

Spring支持非常广泛的应用场景。在大型的企业中，应用程序通常运行了很长时间，有可能这些应用所使用的JDK版本太老了，升级不可控。或者应用已一个单独的嵌套server的jar包运行，或者应用只是一个独立的应用，不需要使用任何server。

Spring是开源的，他有一个庞大和在线的社区为真实世界里面的不同方向的使用情况提供持续的技术支反馈和支持。这使得Spring在很长的一段时间内能够成功进化。

更多教程请访问:www.flydean.com

1. “Spring”代表什么

”Spring“这个词在不同的环境中代表不同的含义。最开始的时候，它指的是Spring框架项目本身，但是随着时间的推移，又发展了其他的Spring项目。所以当现在我们提起”Spring“的时候，它指的是整个Spring家族。 我们的这个Spring参考指南指的是一切Spring的基础：Spring Framework。

Spring Framework被分成了很多个模块，我们在构建应用程序的时候可以自由选择使用哪个模块。最核心的模块是核心容器包括配置模块和依赖注入机制。除此之外，Spring Framework提供了对不同应用架构的基础支持，包括消息，交易数据，存储和网站。他还包括了Servlet-based Spring MVC web framework和最新的响应式Spring WebFlux reactive web framework。

在使用模块上有一点需要注意：Spring Framework的jar包被允许部署到JDK9的模块路径（“Jigsaw”）。当Spring Framework 5被用在开启了Jigsaw的应用程序中时，Spring会开启一个叫做“自动模块名”的功能，并在其中定义了稳定的语言级模块名字，比如（"spring.core", "spring.context"等 ），他们和其所在的jar包名字是不一样的（jar包名字已“-”分割而不是“，”，如"spring-core" 和 "spring-context"）。当然Spring Framework的jar包可以在JDK8和JDK9+环境中正常运行。

更多教程请访问:www.flydean.com

2. Spring和Spring框架的历史

Spring是在2003年，作为对之前的复杂的J2EE规范而产生的。有些人认为Spring和J2EE是竞争关系，实际上他们是互补的。Spring的编程模块虽然并没有遵照J2EE的平台规范，但是它和如下的J2EE规范紧密的结合： Servlet API (JSR 340)

WebSocket API (JSR 356)

Concurrency Utilities (JSR 236)

JSON Binding API (JSR 367)

Bean Validation (JSR 303)

JPA (JSR 338)

JMS (JSR 914)

JTA/JCA

同时，Spring框架也支持依赖注入(JSR 330)和通用注解(JSR 250)规范，开发人员可以自由选择使用。

在Spring框架5.0中，Spring至少要求Java EE 7以上（比如：Servlet 3.1+, JPA 2.1+），同时提供了对于Java EE 8中新API在运行时的支持（Servlet 4.0, JSON Binding API），这使得Spring可以全面支持多种服务器：Tomcat 8， 9, WebSphere 9,JBoss EAP 7。

Java EE在应用程序开发中的角色在随时间变化，最开始，应用程序必须被部署成为一个服务。随着SPring boot的出现，现在的应用程序可以在devops或者云环境中随意创建。在Spring 框架5中，WebFlux应用程序甚至都不需要使用Servlet API，并且可以运行在类似于Netty这种非Servlet容器中。

Spring还将继续发展下去，除了Spring框架，Spring家族还有其他的一些非常棒的项目：Spring Boot, Spring Security, Spring Data, Spring Cloud, Spring Batch等。每个项目都有自己的代码库，问题跟踪机制和发版计划。如果想知道其他项目的具体情况可以参照：spring.io/projects。

更多教程请访问:www.flydean.com

3. 设计哲学

当你接触到一个框架的时候，不仅仅要知道这个框架是怎么使用的，更需要了解框架的设计原则，下面是Spring框架的设计原则：

在每一层都提供选项。Spring可以让你尽可能的推迟选择。比如你可以通过配置文件来切换数据存储的提供方而不需要改动代码。这个规则也很好的应用在和第三方API集成中。

适应不同的视角。Spring具有灵活性，它不会强制为你决定该怎么选择。它以不同的视角支持广泛的应用需求。

保持强大的向后兼容性。Spring的发展经过了精心的管理，使得版本之间几乎没有割裂的变化。Spring支持精心选择的一系列JDK版本和第三方库，以便于维护依赖于Spring的应用程序和库。

关心API设计。Spring团队投入了大量的思想和时间来制作直观的API，这些API可以在多个版本中使用，并可以使用很多年。

为代码质量设定高标准。Spring框架将重点放在有意义、最新和准确的JavaDoc上。它是少数几个可以声明在包之间没有循环依赖关系的干净代码结构的项目之一。

更多教程请访问:www.flydean.com

4. 反馈和贡献

我们建议使用stackoverflow来提问、诊断或调试问题，我们还有一个问题页面，列出了建议使用的标记。如果你相当确定Spring框架中存在问题，或者希望提出一个特性，请使用github issues。

如果你有一个解决方案或建议的修复方案，你可以在Github上提交pull请求。为了留痕和供将来参考，除了一些细小的问题外，我们需要在问题跟踪中提交一个ticket，用以进行讨论。

有关更多详细信息，请参考根目录下面的CONTRIBUTING文件。

更多教程请访问:www.flydean.com

5. 开始

如果你刚开始使用Spring，那么你可能希望通过创建一个基于Spring引导的应用程序来开始使用Spring框架。SpringBoot提供了一种快速（和固定）的方法来创建一个可用于生产环境的基于Spring的应用程序。它基于Spring框架，使用约定的方式而不是配置方式，旨在让你尽快启动和运行。

你可以使用start.spring.io生成一个基本项目，或者遵循“入门”指南之一，例如开始构建一个RESTful Web服务。这些指南不仅易于理解，以任务为导向，而且大多数都基于Spring boot。它们还涵盖了Spring产品组合中的其他项目，可以为你在解决特定问题时做参考。

更多教程请访问:www.flydean.com

核心技术

版本号5.1.8.RELEASE

参考文档的这一部分涵盖了*Spring*框架中绝对不可或缺的所有技术。

其中最重要的是Spring框架的反转控制（IOC）容器。Spring的面向方面编程（AOP）技术在彻底的改进之后的Spring框架的IOC容器中得到了全面的应用。Spring框架有它自己的AOP框架，它在概念更易于理解，它成功地解决了Java企业编程中AOP需求的80%的热点问题。

Spring还提供了与AspectJ的集成（目前是在特性上面最丰富，并且也是在Java企业空间中最成熟的AOP实现）。

更多教程请访问:www.flydean.com

1. IoC容器

本章介绍Spring的反转控制（IOC）容器。

更多教程请访问:www.flydean.com

1.1 IoC容器和Beans介绍

本章介绍了Spring框架中控制反转（IOC）原理的实现。IOC也称为依赖注入（DI）。它是指对象仅通过构造函数参数、工厂方法的参数或从工厂方法构造或返回对象实例后，通过在其上设置的属性来定义其依赖项（即与之一起工作的其他对象）的过程。当容器在创建bean时将会注入这些依赖项。

这个过程跟bean自己控制实例化，或者通过直接构造类或服务定位器模式来定位其依赖项的方式相反（因此称为控制反转）。

org.spring framework.beans和org.springframework.context包是Springframework的IOC容器的基础。

BeanFactory接口提供了一种高级配置机制，能够管理任何类型的对象。

ApplicationContext是BeanFactory的子接口。它添加了如下功能：

- 更容易与Spring的AOP功能集成
- 消息资源处理（用于国际化）
- 事件发布
- 应用层特定的上下文，如用于Web应用程序的WebApplicationContext

简而言之，BeanFactory提供了配置框架和基本功能，ApplicationContext添加了更多特定于企业应用的功能。ApplicationContext是BeanFactory的一个完整的超集，在本章中专门用于描述Spring的IOC容器。有关使用BeanFactory的更多信息，请参见BeanFactory。

在Spring中，构成应用程序主干并由SpringIOC容器管理的对象称为bean。bean是一个由SpringIOC容器实例化、组装和管理的对象。否则，bean只是应用程序中许多对象之一。bean以及它们之间的依赖关系反映在容器使用的配置元数据中。

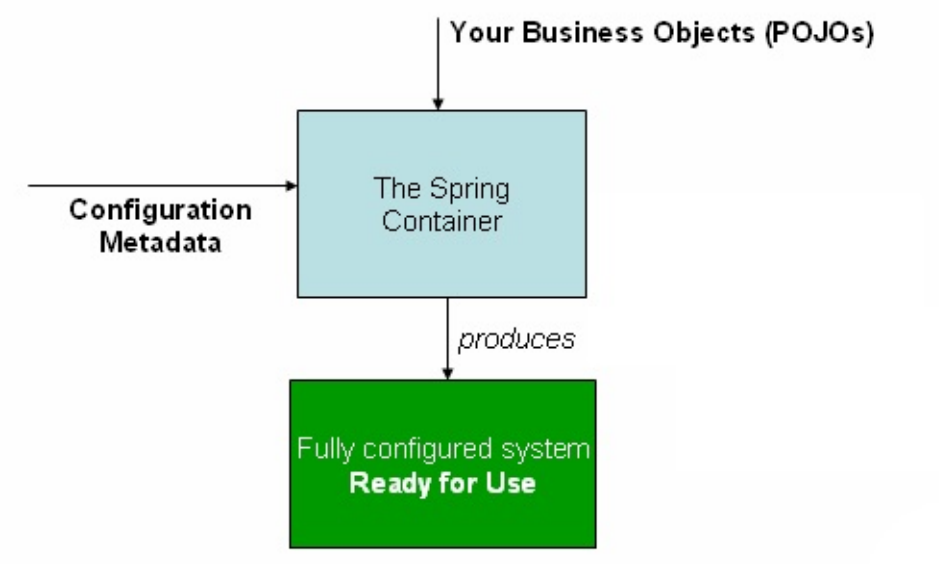
更多教程请访问:www.flydean.com

1.2 IoC容器概述

org.springframework.context.ApplicationContext接口代表着Spring IOC容器，它负责实例化、配置和组装bean。容器通过读取配置元数据获取关于要实例化、配置和组装的对象的指令。配置元数据以XML、Java注解或Java代码来表示。它定义了组成应用程序的对象以及这些对象之间的丰富依赖关系。

Spring提供了ApplicationContext接口的几个实现。在独立应用程序中，通常创建ClassPathXMLApplicationContext或FileSystemXMLApplicationContext的实例。虽然XML是定义配置元数据的传统格式，但你可以通过使用Java注解或代码作为元数据格式，并外加少量的XML配置，来声明对这些附加元数据格式的支持。

在大多数应用程序场景中，不需要显式代码来实例化SpringIOC容器的一个或多个实例。例如，在Web应用程序场景中，简单的在web.xml中用八行（或更多）标准Web描述符即可。如果你使用Spring工具套件（一个支持Eclipse的开发环境），只需点击几下鼠标或按键，就可以轻松地创建这个标准配置。 下图显示了Spring如何工作的高级视图。应用程序类与配置元数据结合在一起，这样，在创建和初始化ApplicationContext之后，你就拥有了一个完全配置且可执行的系统或应用程序。



1.2.1 配置元数据

如上图所示，SpringIOC容器使用一种形式的配置元数据。此配置元数据告诉应用程序开发人员，如何通知Spring容器实例化、配置和组装应用程序中的对象。

配置元数据一般是以简单直观的XML格式提供的，XML也会在本章的大部分内容用来表示Spring IOC容器的关键概念和特性。

> 基于XML的元数据不是唯一的配置元数据的方式。SpringIOC容器与编写配置元数据的格式完全分离。现在，更多的开发人员选择基于Java的配置来编写Spring应用程序。

有关在Spring容器中使用其他形式的元数据的信息，请参见：

- 基于注解的配置：Spring2.5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开始，Spring JavaConfig项目提供的许多特性成为核心Spring框架的一部分。因此，可以使用Java而不是XML文件来定义应用程序的bean。要使用这些新功能，请参见@Configuration、@Bean、@Import和@DependsOn注解。

更多教程请访问:www.flydean.com

Spring配置由容器必须管理的至少一个或多个bean定义组成。基于XML的配置元数据将这些bean配置为顶级< beans/>元素中的< bean/>元素。Java配置通常在@Configuration中使用@Bean注解的方法。

这些bean定义对应于组成应用程序的实际对象。一般用来定义服务层对象、数据访问对象（DAO）、表示对象（如Struts操作实例）、基础结构对象（如Hibernate会话行为）、JMS队列等。通常，我们不会在容器中配置细粒度的域对象，因为DAO和业务逻辑通常负责创建和加载域对象。但是，你可以使用Spring与AspectJ的集成来配置在IOC容器控制之外创建的对象。

下面的示例显示了基于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
        https://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>
```

- id属性是标识单个bean定义的字符串。
- class属性定义bean的类型并使用完全限定的类名。

id属性的值是指协作对象。此例中没有显示用于引用协作对象的XML。有关详细信息，请参考依赖。

1.2.2. 实例化一个容器

容器的构造函数可以接收各种外部资源的配置元数据来实例化一个容器，例如本地文件系统、Java类路径等。

```
ApplicationContext context = new ClassPathXmlApplicationContext("services.xml", "daos.xml");
```

以下例子展示了如何配置服务层对象（services.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"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- services -->

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

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

</beans>
```

以下示例显示了数据访问对象daos.xml文件：

更多教程请访问:www.flydean.com

```
<?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
        https://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>
```

在前面的例子中，服务层由PetStoreServiceImpl类和两个类型为JpaAccountDao和JpaItemDao（基于jpa对象关系映射标准）的数据访问对象组成。property name元素引用了JavaBean的名字，ref元素引用另一个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。

构建基于XML的配置元数据

让一个文件中定义的bean可以在多个XML文件使用是非常有用的。通常，架构中的逻辑层或模块都可以用单独的XML配置文件来表示。 你可以使用应用程序上下文构造函数从这些XML配置文件中加载bean。如上一节提到的，这个构造函数可以接收多个资源地址。 如果不想使用构造函数，可以使用一个或多个< import/>元素从另一个或多个文件加载bean定义。下面例子展示了如何执行此操作：

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

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

在前面的示例中，外部bean定义是从三个文件加载的：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同一目录或类路径位置，而messagesource.xml和themesource.xml必须位于导入文件位置下方的资源位置。如你所见，messageSource.xml的前导斜杠被忽略了。但是，因为我们使用的是相对路径，所以最好不要使用斜线。

根据Spring Schema的定义，要导入的文件的内容（包括顶层的< beans/>元素）必须是有效的XML bean。

> 可以（但不推荐）使用相对的“..”路径引用父目录中的文件。这样做会创建对当前应用程序外部文件的依赖关系。特别的，不建议使用这样的引用classpath:urls（例如classpath:../services.xml），因为程序在运行中解析时，将优先选择“最近”的classpath根目录，然后才会查看其父目录。当类路径配置发送变化时可能导致选择到不正确的目录。
>
> 你可以使用完全限定的资源位置，而不是相对路径：例如，文件：c:/config/services.xml或classpath:/config/services.xml。但是这样会将你的应用程序的配置耦合到特定的绝对位置。通常情况下，最好通过在运行时根据JVM系统属性解析的“$…”占位符，来为此类绝对位置保留间接寻址。

命名空间本身提供了导入指令功能。在Spring提供的一系列XML名称空间中，除了纯bean定义之外，还提供了其他配置功能，例如Context和Util名称空间。

groovy bean定义DSL

更多教程请访问:www.flydean.com

作为外部配置元数据的另一个例子，bean也可以用Spring的Groovy Bean定义DSL来表示，这在grails框架中是很常见的。通常，这种配置存在一个“.groovy”文件中，其结构如以下示例所示：

```
beans {
    dataSource(BasicDataSource) {
        driverClassName = "org.hsqldb.jdbcDriver"
        url = "jdbc:hsqldb:mem:grailsDB"
        username = "sa"
        password = ""
        settings = [mynew:"setting"]
    }
    sessionFactory(SessionFactory) {
        dataSource = dataSource
    }
    myService(MyService) {
        nestedBean = { AnotherBean bean ->
            dataSource = dataSource
        }
    }
}
```

这种配置风格在很大程度上相当于XML bean定义，甚至支持Spring的XML配置名称空间。它还允许通过import beans指令导入XML bean定义文件。

1.2.3. 使用容器

ApplicationContext是高级工厂的接口，它能够维护不同bean的注册及其依赖。通过使用方法 `T getBean(String name, Class<T> requiredType)` ，获取到bean的实例。 ApplicationContext允许你读取bean定义并访问它们，如下例所示：

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

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

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

使用Groovy配置，跟XML看起来非常相似。它有一个不同的上下文实现类，它支持Groovy（但也理解XML bean定义）。下面的示例显示了groovy配置：

```
ApplicationContext context = new GenericGroovyApplicationContext("services.groovy", "daos.groovy");
```

最灵活的是GenericApplicationContext与reader delegates结合使用，例如使用XmlBeanDefinitionReader 来加载XML文件，如下示例所示：

```
GenericApplicationContext context = new GenericApplicationContext();
new XmlBeanDefinitionReader(context).loadBeanDefinitions("services.xml", "daos.xml");
context.refresh();
```

你也可以使用GroovyBeanDefinitionReader来加载Groovy文件，如下所示：

```
GenericApplicationContext context = new GenericApplicationContext();
new GroovyBeanDefinitionReader(context).loadBeanDefinitions("services.groovy", "daos.groovy");
context.refresh();
```

你可以在同一ApplicationContext中混合使用这样的Reader，从不同的配置源读取bean定义。

更多教程请访问:www.flydean.com

然后可以使用getBean来获取到bean的实例。ApplicationContext接口还有一些其他的方法来获取bean，但理想情况下，你的应用程序代码不应该使用它们。实际上，你的应用程序代码根本不应该调用getBean（）方法，因此完全不依赖于SpringAPI。例如，Spring与Web框架的集成为各种Web框架组件（如控制器和JSF管理的bean）提供了依赖注入，允许你通过元数据（如自动注解方式）声明对特定bean的依赖。

更多教程请访问:www.flydean.com

1.3 Bean概述

Spring IOC容器管理一个或多个bean。这些bean是使用你提供给容器的配置元数据创建的（例如，以xml定义的形式）。

在容器本身中，这些bean表示为BeanDefinition对象，其中包含（除其他信息外）以下元数据：

- 包限定类名：通常是定义bean的实际实现类。
- bean行为配置元素，用于说明bean在容器中的行为（范围、生命周期回调等）。
- 对bean执行其工作所需的其他bean的引用。这些引用也称为合作者或依赖。
- 要在新创建的对象中设置的其他配置设置，例如一个管理连接池的Bean中连接池的大小限制或正在使用的连接数目。

这个些元数据包含了构造Bean定义的一组属性。下表描述了这些属性：

属性名	解释于...
Class	Instantiating Beans
Name	Naming Beans
Scope	Bean Scopes
Constructor arguments	Dependency Injection
Properties	Dependency Injection
Autowiring mode	Autowiring Collaborators
Lazy initialization mode	Lazy-initialized Beans
Initialization method	Initialization Callbacks
Destruction method	Destruction Callbacks

除了包含有关如何创建特定bean的信息的bean定义之外，ApplicationContext实现还允许注册容器外部（由用户）创建的现有对象。 通过getBeanFactory（）方法访问ApplicationContext的BeanFactory，该方法返回BeanFactory接口的DefaultListableBeanFactory实现方法。 DefaultListableBeanFactory通过registerSingleton（..）和registerBeanDefinition（..）这两个方法来注册外部Bean。 但是，应用程序通常只使用常规bean定义元数据来定义bean。

> 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省步骤中正确地解析它们。
>
> 虽然在Spring某种程度上支持覆盖现有元数据和现有的单例实例，但是官方并不支持在运行时注册新bean（与工厂的实时访问同时进行），因为这可能会导致并发访问异常或者bean容器中的状态不一致。

1.3.1. 为Bean命名

每个bean都有一个或多个标识符。这些标识符在承载bean的容器中必须是唯一的。bean通常只有一个标识符。但是，如果它需要多个标志符，则可以将额外的标志符视为别名。

在基于XML的配置元数据中，可以使用id属性或name属性或者同时使用来指定bean标识符。ID属性允许你仅指定一个ID。通常来说，这些名称是字母数字（“mybean”、“someservice”等），但它们也可以包含特殊字符。如果要为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。

在Spring3.1之前的版本中，id属性被定义为xsd:id类型，这限制了可能的字符。从3.1开始，它被定义为xsd:string类型。注意，bean id的唯一性仍然由容器强制实现，尽管不再由XML解析器实现。

更多教程请访问:www.flydean.com

你不需要强制的为bean提供名称或ID。如果不显式地提供名称或ID，容器将为该bean生成一个唯一的名称。但是，如果你希望通过使用ref元素或服务定位器风格来查找该bean，则必须提供一个名称。不提供名称仅用在使用内部bean和自动装配协作者的情况。

> Bean命名规范
>
> 在命名bean时使用标准Java规范，例如字段名。也就是说，bean名称以一个小写字母开头的驼峰写法。比如：accountManager, accountService, userDao, loginController等。
>
> bean命名的一致性使你的配置更易于阅读和理解。另外，如果你使用SpringAOP，那么一致性的Bean命名会在通知一组名字相关的Bean时很有用。
>
> 通过在类路径中扫描组件，Spring为未命名的组件生成bean名称，遵循前面描述的规则：本质上，取简单的类名并将其初始字符改为小写。但是，在特殊情况下，当有多个字符并且第一个和第二个字符都是大写时，原始的大小写将被保留。这些规则与java.beans.introspector.decapitalize定义的规则相同。

在Bean的定义外使用别名

在bean定义中，你可以为bean提供多个名称，可以通过id属性指定最多一个名称或者通过name属性指定任何数量的其他名称。 这些名称是同一Bean的等效别名，在某些情况下非常有用，例如，可以为一个通用的组件指定名字，并在应用程序的其他每个组件引用它。

只在bean实际定义的地方指定所有别名并不够用，有时需要为在别处定义的bean引入别名。在大型系统中，通常把系统拆分成多个子系统，每个子系统都有自己的对象定义集。在基于XML的配置元数据中，可以使用元素来实现这一点。以下示例显示了如何执行此操作：

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

在这种情况下，名为fromName的bean（在同一容器中）在使用这个别名定义之后，也可以称为toName。

例如，子系统A的配置元数据引用了名为subsystemA-dataSource的DataSource。子系统B的配置元数据引用了名为subsystemB-dataSource的DataSource。 当组成使用这两个子系统的主应用程序时，主应用程序以myApp-dataSource的名称引用DataSource。要使所有三个名称都指向同一对象，可以将以下别名定义添加到配置元数据中：

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

现在，每个组件和主应用程序都可以通过唯一的名称引用数据源，并保证不会与任何其他定义冲突（有效地创建了一个名称空间），但它们引用的是同一个bean。

```
JAVA配置
如果你使用JAVA配置，@Bean可以用来提供别名。
```

1.3.2. 实例化Bean

bean定义本质上是创建一个或多个对象的方法。当需要创建Bean的时候，容器会查看命名bean的创建方法，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，则需要指定要在< bean/>元素的class属性中实例化的对象的类型（或类名）。这个class属性（在内部是BeanDefinition实例上的Class属性）通常是强制的。（或者请参见使用实例工厂方法和bean定义继承进行实例化。）可以通过以下两种方式之一使用Class属性：

- 多数情况下，在容器本身直接创建bean，通过调用bean的构造函数来创建bean的实例，相当于Java代码的new。
- 少数情况下，容器调用类上的静态工厂方法来创建bean。需要要指定包含调用创建对象的静态工厂方法的实际类，静态工厂方法调用返回的对象类型可以是同一个类或完全是另一个类。

更多教程请访问:www.flydean.com

> 内部类名
>
> 如果要为静态内部类配置bean定义，则必须使用嵌套类的二进制名称。
>
> 例如，如果在com.example包中有一个名为SomeThing的类，而这个SomeThing类有一个名为OtherThing的静态内部类，bean定义中class属性的值将是com.example.SomeThing$OtherThing。
>
> 注意，在名称中使用$字符将内部类名与外部类名分开。

通过构造函数实例化

当使用构造函数创建bean时，所有普通类都可以被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的接口或以特定的方式进行编码。仅仅指定bean类就足够了。但是，根据你对特定bean使用的IOC类型，你可能需要一个默认（空）构造函数。

SpringIOC容器几乎可以管理你想要管理的任何类。它不仅限于管理真正的JavaBeans。大多数Spring用户更喜欢实际的JavaBean，它只有一个默认（无参数）构造函数，以及根据容器中的属性创建的setter和getter方法。你还可以在容器中拥有更多其他非bean风格的类。例如，如果你需要使用完全不符合JavaBean规范的历史的连接池，那么Spring也可以对其进行管理。

使用基于XML的配置元数据，可以按如下方式指定bean类：

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

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

有关向构造函数提供参数（如果需要）和在构造对象后设置对象实例属性的机制的详细信息，请参阅注入依赖。

使用静态工程方法实例化

定义使用静态工厂方法创建的bean时，使用class属性指定包含静态工厂方法的类，使用名为factory-method的属性指定工厂方法本身的名称。你应该能够调用这个方法（使用可选的参数，如后文所述），并返回一个活动对象，然后将其视为是通过构造函数创建的。这种bean定义的一个用途是调用遗留代码中的静态工厂。

下面的bean定义指定通过调用工厂方法来创建bean。定义没有指定返回对象的类型（类），只指定包含工厂方法的类。在本例中，createInstance（）方法必须是静态方法。以下示例显示如何指定工厂方法：

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

下面是这个工厂类：

```
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）参数和在从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。

使用实例里面的工厂方法创建实例

与通过静态工厂方法进行实例化类似，使用实例工厂方法进行实例化会从容器中调用现有bean的非静态方法来创建新bean。要使用此机制，请将class属性保留为空，并在factory-bean属性中，在包含要调用以创建对象的实例方法的当前（或父或父）容器中指定bean的名称。使用factory-method属性设置工厂方法本身的名称。下面的示例演示如何配置这

更多教程请访问:www.flydean.com

样的bean：

```
<!-- 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"/>
```

下面列出了相应的Java类：

```
public class DefaultServiceLocator {

    private static ClientService clientService = new ClientServiceImpl();

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

如下所示，一个工厂类可以包括多个工厂方法：

```
<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"/>
```

下面列出了相应的Java类：

```
public class DefaultServiceLocator {

    private static ClientService clientService = new ClientServiceImpl();

    private static AccountService accountService = new AccountServiceImpl();

    public ClientService createClientServiceInstance() {
        return clientService;
    }

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

这种方法表明，工厂bean本身可以通过依赖注入（DI）进行管理和配置。请参阅依赖和配置。

> 在Spring文档中，“factory bean”是指在Spring容器中配置的bean，它通过实例或静态工厂方法创建对象。 相比之下，FactoryBean（注意大写）是指Spring中的FactoryBean类。

更多教程请访问:www.flydean.com

更多教程请访问:www.flydean.com

1.4 依赖

典型的企业应用程序不会只有一个对象（或Spring术语中的bean）。即使是最简单的应用程序也有一些对象一起工作，最终呈现给用户。

下一节将解释如何从定义一些独立的bean到一个真实的应用程序，在该应用程序中通过对象协作来实现一个目标。

1.4.1. 依赖注入

依赖项注入（DI）是指：对象仅仅通过构造函数参数、工厂方法的参数或从工厂方法构造,或返回对象实例后在其上设置属性来定义其依赖项（即它们使用的其他对象）。然后，容器在创建bean时注入这些依赖项。这个过程基本上和bean通过直接类的构造器或服务定位器来定位其依赖的过程是相反的（因此称为控制反转）。

使用DI，代码更干净，当对象具有依赖关系时，去耦更有效。对象不需要查找其依赖项，甚至不知道依赖项的位置或类。 因此，你的类变得更容易测试，特别是当依赖于接口或抽象基类时。（允许在单元测试中使用stub或mock实现）

DI有两个主要的方式：基于构造函数的依赖注入和基于setter的依赖注入。

基于构造函数的依赖注入

基于构造函数的DI是通过容器调用带有多个参数的构造函数来实现的，每个参数表示一个依赖项。这个和调用带有特定参数的静态工厂方法来构造bean几乎是等效的，这里将会对构造函数和静态工厂方法的参数做相似的处理。下面的示例显示了一个只能通过构造函数注入依赖项的类：

```
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...
}
```

注意这个类没有什么特别的。它是一个POJO，不依赖于特定于容器的接口、基类或注解。

构造函数参数的解析

一般通过参数的类型来进行构造函数参数的解析匹配。如果bean定义的构造函数参数中没有潜在的歧义，那么在bean定义中定义构造函数参数的顺序就是在bean被实例化时，这些参数被提供给适当的构造函数的顺序。可以看下面的类：

```
package x.y;

public class ThingOne {

    public ThingOne(ThingTwo thingTwo, ThingThree thingThree) {
        // ...
    }
}
```

假设ThingTwo和ThingThree类与继承无关，则不存在潜在的歧义。因此，以下配置工作正常，你不需要在<constructor-arg/>元素中显式指定构造函数参数索引或类型。

更多教程请访问:www.flydean.com

```
<beans>
    <bean id="beanOne" class="x.y.ThingOne">
        <constructor-arg ref="beanTwo"/>
        <constructor-arg ref="beanThree"/>
    </bean>

    <bean id="beanTwo" class="x.y.ThingTwo"/>

    <bean id="beanThree" class="x.y.ThingThree"/>
</beans>
```

当引用另一个bean时，如果类型是已知的，可以通过类型进行匹配（前面的例子就是这样）。当使用简单类型时，例如<value>true</value>，Spring无法确定值的类型，因此在没有帮助的情况下无法按类型匹配。考虑以下情况：

```
package examples;

public class ExampleBean {

    // Number of years to calculate the Ultimate Answer
    private int years;

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

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

构造函数类型匹配

在前面的场景中，如果使用type属性显式指定构造函数参数的类型，则容器可以使用与简单类型匹配的类型。如下示例所示：

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

构造函数索引

你也可以使用index属性来显示的指定构造函数参数的位置，如下所示：

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

除了解决多个简单值的不确定性之外，指定索引还可以解决构造函数有两个相同类型参数的不确定性。

```
注意：index是从0开始的。
```

构造函数名字匹配

你也可以使用构造函数的参数名字来指定，如下所示：

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

更多教程请访问:www.flydean.com

```
</bean>
```

请记住，要使用这项功能，必须在启用调试标志的情况下编译代码，以便Spring可以从构造函数中查找参数名。

如果无法或不希望使用调试标志编译代码，可以使用@ConstructorProperties JDK注解显式命名构造函数参数。然后示例类必须如下所示：

```
package examples;

public class ExampleBean {

    // Fields omitted

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

基于Setter的注入依赖

基于Setter的DI是由容器在调用无参数构造函数或无参数静态工厂方法实例化bean之后调用bean上的setter方法来完成的。

下面的示例显示了一个类，该类只能通过使用纯setter注入来注入依赖项。这个类是传统的Java。它是一个POJO，不依赖于特定于容器的接口、基类或注解。

```
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...
}
```

ApplicationContext支持它所管理的bean的基于构造函数和基于setter的DI。在通过构造函数方法注入了一些依赖项之后，仍然可以使用基于setter的DI。 所有的依赖都会被转化成为BeanDefinition，它会与PropertyEditor实例一起使用，将属性从一种格式转换为另一种格式。 然而，大多数Spring用户并不直接使用这些类（即代码编程方式），而是使用XML bean定义、注解的组件（即用@Component、@Controller等来注解的类），或基于Java的@Configuration注解中的@Bean方法。然后，这些定义会在内部转换为BeanDefinition实例，并被用于Spring IOC容器中。

> 构造函数还是setter形式的注入？
>
> 由于你可以混合使用基于构造函数和基于setter的DI，所以对于强制依赖项使用构造函数，对于可选依赖项使用setter方法或配置方法是一个很好的实践经验。 注意，可以使用setter方法上的@Required注解使属性成为必需的依赖项；但是，最好使用构造函数注入并在程序中做相应的验证。
>
> Spring团队通常提倡构造函数注入，因为它允许你将应用程序组件实现为不可变的对象，并确保所需的依赖项不为空。 此外，构造函数注入的组件总是以完全初始化的状态返回到客户端（调用）代码。另一点需要注意的是，大量的构造函数参数是一种糟糕的代码体验，这意味着类可能做了太多的事情，应该重构以做适当的分离。
>
> Setter注入应该主要用于可在类内分配合理默认值的可选依赖项。否则，在代码使用依赖项的任何地方都必须执行非空检查。 Setter注入的一个好处是Setter方法使该类的对象能够在以后重新配置或重新注入。因此，通过JMX MBeans进行管理是Setter注入的一个成功的用例。

更多教程请访问:www.flydean.com

> 对特定类使用最有意义的DI样式。有时，在处理没有源代码的第三方类时，只有一种选择。例如，如果第三方类不公开任何setter方法，那么构造函数注入可能是DI的唯一可用形式。

依赖解析流程

容器按如下流程处理Bean的依赖解析：

- ApplicationContext被创建和实例化出来，它包含了所有bean的配置元数据。这些配置元数据可以通过XML、Java代码或注解来指定。
- 对于每个bean，其依赖项都以属性、构造函数参数或静态工厂方法的参数的形式表示。当bean实际创建时，这些依赖项会提供给bean。
- 每个属性或构造函数参数都是要设置的值的实际定义，或者是对容器中另一个bean的引用。
- 作为值的每个属性或构造函数参数都将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，如int、long、String、boolean等。

Spring容器在创建容器时会验证每个bean的配置。但是，在bean实际创建之前，不会设置bean的属性。当创建容器时，单实例并设置为预实例化（默认方式）的bean会被创建。作用域被设置为bean作用域。否则，只有在请求bean时才创建它。创建bean可能会导致创建bean图，因为bean的依赖项及其依赖项的依赖项（等等）会被创建和分配。

请注意，这些依赖项之间的解析不匹配可能会很晚出现，也就是说，在第一次创建受影响的bean时。

> 循环依赖
>
> 如果主要使用构造函数注入，则可能创建循环依赖的场景。
>
> 例如：类A通过构造函数注入需要类B的实例，类B通过构造函数注入需要类A的实例。如果为要相互注入的类A和类B配置bean，那么SpringIOC容器在运行时检测到这个循环引用，会抛出BeanCurrentlyInCreationException。
>
> 一种可能的解决方案是编辑一些类的源代码，这些类由setter而不是构造函数配置。或者，避免构造函数注入，只使用setter注入。换句话说，尽管不建议这样做，但是可以使用setter注入配置循环依赖项。
>
> 与典型情况（没有循环依赖性）不同，bean a和bean b之间的循环依赖强制将其中一个bean注入另一个bean，然后完全初始化自己（经典的鸡和蛋场景）。

一般来说，你可以相信Spring会做正确的事。它在容器加载时检测配置问题，例如不存在的bean和循环依赖项的引用。

Spring在实际创建bean时设置属性并尽可能晚地解析依赖项。这意味着，如果在创建对象或其依赖项时出现问题，那么之前正确加载的Spring容器稍后可能在请求对象时生成异常，例如：bean由于缺少或无效的属性而引发异常。

对于这样的配置问题，这种潜在的延迟可见性是ApplicationContext实现默认情况下预实例化singleton bean的原因。

在实际需要使用这些bean之前，你需要花费一些前期时间和内存来创建它们，并在创建ApplicationContext时发现配置问题，而不是稍后。

你仍然可以重写这个默认行为，以便单例bean可以延迟初始化，而不是预先实例化。

如果不存在循环依赖关系，当一个或多个协作bean被注入依赖bean时，每个协作bean在被注入依赖bean之前都被完全配置。这意味着，如果bean a依赖于bean b，那么在bean a上调用setter方法之前，spring ioc容器会完全配置bean b。

换句话说，bean被实例化（即使它不是预实例化的singleton），它的依赖被设置，并且调用相关的生命周期方法（例如configured init方法或InitializingBean回调方法）。

依赖注入的例子

下面的示例是基于XML的配置元数据使用基于setter的DI。如下所示：

```
<bean id="exampleBean" class="examples.ExampleBean">
    <!-- setter injection using the nested ref element -->
    <property name="beanOne">
        <ref bean="anotherExampleBean"/>
    </property>
```

更多教程请访问:www.flydean.com

```
    <!-- 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"/>
```

下面是相应的ExampleBean类：

```
public class ExampleBean {

    private AnotherBean beanOne;

    private YetAnotherBean beanTwo;

    private int i;

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

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

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

在上面的例子中，setter声明和XML文件中指定的属性匹配。 下面是构造函数DI的例子：

```
<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"/>
```

下面是相应的ExampleBean类：

```
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;
    }
```

更多教程请访问:www.flydean.com

```
}
```

bean定义中指定的构造函数参数用作ExampleBean的构造函数的参数。

现在考虑这个例子的一个变体，在这里，Spring不是使用构造函数，而是调用一个静态工厂方法来返回对象的一个实例：

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

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

下面是ExampleBean的例子：

```
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;
    }
}
```

静态工厂方法的参数由<constructor-arg/>元素提供，与实际使用的构造函数完全相同。工厂方法返回的类的类型不必与包含静态工厂方法的类的类型相同（尽管在本例中是这样）。实例（非静态）工厂方法可以以基本相同的方式使用（除了使用factory-bean属性而不是class属性），因此我们不在这里讨论这些细节。

1.4.2 依赖和配置的细节

如上节所示，自定义的Bean属性或者构造函数参数可以引用其他被管理的Bean，或者定义内联的值。Spring基于XML的配置元数据可以在<property/>和<constructor-arg/>元素中的配置子元素类型来实现这一功能。

直接值（基本类型，字符串或者其他）

<property/>元素的value属性将属性或构造函数参数指定为人类可读的字符串表示形式。Spring的转换服务用于将这些值从字符串转换为属性或参数的实际类型。以下示例显示了正在设置的各种值：

```
<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>
```

更多教程请访问:www.flydean.com

下面的例子使用了更简洁的p-namespace例子：

```
<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
    https://www.springframework.org/schema/beans/spring-beans.xsd">

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

</beans>
```

上面的XML更加简洁。但是，除非你使用一个IDE（如Intellij IDEA或Spring工具套件），在创建bean定义时支持自动完成属性，否则只能在运行时而不是设计时发现拼写错误。强烈建议使用这样的IDE协助开发。

你也可以配置一个java.util.Properties实例如下：

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

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

Spring容器使用JavaBeans属性编辑器机制将<value/>元素中的文本转换为java.util.properties实例。这是一个很好的快捷方式，也是Spring团队支持使用嵌套的<value/>元素而不是value属性样式的少数地方之一。

idref元素

idref元素是将容器中另一个bean的id（字符串值，而不是引用）传递给<constructor-arg/>或<property/>元素的一种简单的防止错误方法。下面的示例演示如何使用它：

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

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

前面的bean定义片段（在运行时）与以下片段完全等效：

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

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

更多教程请访问:www.flydean.com

第一个表单比第二个表单更好，因为使用idref标记可以让容器在部署时验证引用的、bean是否实际存在。在第二个例子中，不会对传递给客户机bean的targetName属性的值执行任何验证。只有当客户端bean实际被实例化时，才会发现拼写错误（最有可能是致命的结果）。如果客户端bean是prototype类型的bean，那么只有在部署容器很长时间之后，才会发现这种类型和产生的异常。

> 4.0 beans xsd不再支持idref元素的local属性，因为它不再提供常规bean引用的值。在升级到4.0模式时，请将idref local修改对idref bean。

<idref/>元素的一个常用的作用（至少在Spring2.0之前的版本中）是在用在ProxyFactoryBean中定义的AOP拦截器里。指定拦截器名称时使用<idref/>元素可防止你拼写错误拦截器ID。

引用其他的Beans

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（合作者）的引用。引用的bean是要设置其属性的bean的依赖项，在设置该属性之前，需要对其进行初始化。（如果合作者是单例bean，那么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。作用域和验证会根据你通过bean、local或parent属性来指定另一个对象的ID或name而做相应的变动。

通过<ref/>标记的bean属性指定目标bean是最通用的形式，允许在同一容器或父容器中创建对任何bean的引用，而不管它是否在同一XML文件中。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属性中的一个值相同。以下示例显示如何使用ref元素：

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

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。父属性的值可以与目标bean的id属性或目标bean的name属性中的一个值相同。目标bean必须位于当前bean的父容器中。 当你使用层级的容器结构时，并且希望用与父bean同名的代理将现有bean包装在父容器中时，就可以使用parent属性。下面展示了如何使用父属性：

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

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

内部Beans

在<property/> 或者 <constructor-arg/>元素内部的<bean/>元素定义了一个内部bean,下面是个例子：

```
<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>
```

更多教程请访问:www.flydean.com

内部bean定义不需要ID或名称。如果指定，容器也不会使用这个值作为标识符。容器在创建时也忽略范围标志，因为内部bean总是匿名的，并且总是用外部bean创建的。不可能单独访问内部bean，也不可能将它们注入到除封闭bean之外的协作bean中。

可以接收来自自定义范围的destruction回调，例如，对于包含在单例bean的请求范围内的内部bean。内部bean实例的创建与它的包含bean绑定在一起，但是destruction回调让它可以接入请求范围的生命周期。这不是一个常见的场景。内部bean通常只共享其包含bean的范围。

集合

<list/>, <set/>, <map/>,和 <props/> 分别被用来设置Java Collection类型List, Set, Map， 和 Properties 类型的属性和参数。 下面是个例子：

```
<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>
```

下面的属性可以作为map的key或者value， 也可以作为set的value:

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

集合合并

Spring容器还支持合并集合。应用程序开发人员可以定义父级< list/>、< map/>、< set/>或< props/>元素，并让子级< list/>、< map/>、< set/>或< props/>元素继承和重写父集合中的值。也就是说，子集合的值是合并父集合和子集合的元素的结果，而子集合元素覆盖父集合中指定的值。

本节讨论合并的父子bean机制。不熟悉父bean和子bean定义的读者可能希望在继续之前阅读相关部分。

下面的示例演示集合合并：

```
<beans>
    <bean id="parent" abstract="true" class="example.ComplexObject">
        <property name="adminEmails">
```

更多教程请访问:www.flydean.com

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

注意在子bean定义的adminEmails属性的<props/>元素上使用了merge=true属性。当子bean由容器解析和实例化时，生成的实例具有一个adminEmails属性集合，该集合包含将子级的adminEmails集合与父级的adminEmails集合合并的结果。以下列表显示了结果：

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

子属性集合的值集继承父集合中的所有属性元素，子集合的支持值覆盖父集合中的值。

这种合并行为同样适用于< list/>、< map/>和< set/>集合类型。< list/>元素比较特殊（即值是有序集合），父列表的值会位于子列表的所有值之前。对于映射、集合和属性集合类型，不存在排序。因此，对于容器内部使用的关联映射、集合和属性实现类型的基础集合类型，没有有效的排序语义。

集合合并的限制

不能合并不同的集合类型（如映射和列表）。如果你尝试这样做，就会抛出一个异常。合并属性必须在继承子对象上定义。在父集合定义上指定合并属性是多余的，不会导致所需的合并。

强类型集合

通过在Java 5中引入泛型类型，可以使用强类型集合。也就是说，可以声明集合类型，使其只能包含（例如）字符串元素。如果使用Spring将强类型集合注入bean，则可以利用Spring的类型转换支持，以便在将强类型集合实例的元素添加到集合之前将其转换为适当的类型。下面的Java类和bean定义显示了如何这样做：

```
public class SomeClass {

    private Map<String, Float> accounts;

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

```
<beans>
    <bean id="something" class="x.y.SomeClass">
        <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>
```

更多教程请访问:www.flydean.com

当something bean的accounts属性准备好注入时，关于强类型map的元素类型的泛型信息可以通过反射获得。因此，Spring的类型转换基础结构将各种值元素识别为float类型，并将字符串值（9.99、2.75和3.99）转换为实际的float类型。

Null和Empty字符串值

Spring将属性和参数的空值视为空Strings。以下基于XML的配置元数据片段将email属性设置为空字符串值（“”）。

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

上面的例子相当于下面的例子：

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

< null/> 元素表示null值看如下所示：

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

上面例子相当于：

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

p-namespace的XML快捷方式

p-namespace允许你使用bean元素的属性（而不是嵌套的< property/>元素）来描述协作bean或属性值。 Spring支持基于XML模式定义的带有命名空间的可扩展配置格式。本章讨论的bean配置格式在XML模式文档中定义。但是，p-namespace没有在xsd文件中定义，并且只存在于spring的核心中。

下面的示例显示了两个解析为相同结果的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
        https://www.springframework.org/schema/beans/spring-beans.xsd">

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

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

该示例在bean定义中显示了p-namespace中名为email的属性。这告诉Spring这是一个属性申明。如前所述，p-namespace没有模式定义，因此可以将property name设置为属性名。

下一个示例包括另外两个bean定义，它们都引用了另一个bean：

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

更多教程请访问:www.flydean.com

```
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://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>
```

此示例不仅包括使用p-namespace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"/>创建从bean john到bean jane的引用，但是第二个bean定义使用p:spouse ref="jane"作为属性来执行完全相同的操作。在这种情况下，spouse是属性名，而-ref部分表明这不是一个直接值，而是对另一个bean的引用。

> p-namespace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，而标准XML格式则不冲突。我们建议你仔细选择方法，并将其传达给你的团队成员，以避免生成同时使用这三种方法的XML文档。

c-namespace的XML快捷方式

和前面的p-namespace一样，Spring 3.1引入的c-namespace允许内联属性配置构造函数参数，而不是嵌套的constructor-arg元素。下面是例子：

```
<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
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="beanTwo" class="x.y.ThingTwo"/>
    <bean id="beanThree" class="x.y.ThingThree"/>

    <!-- traditional declaration with optional argument names -->
    <bean id="beanOne" class="x.y.ThingOne">
        <constructor-arg name="thingTwo" ref="beanTwo"/>
        <constructor-arg name="thingThree" ref="beanThree"/>
        <constructor-arg name="email" value="something@somewhere.com"/>
    </bean>

    <!-- c-namespace declaration with argument names -->
    <bean id="beanOne" class="x.y.ThingOne" c:thingTwo-ref="beanTwo"
        c:thingThree-ref="beanThree" c:email="something@somewhere.com"/>

</beans>
```

c命名空间和p命名空间有一样的约定，比如通过名字设置构造函数时（-ref 代表着引用），同样的，虽然他没有在XSD规范中定义（存在Spring Core中），但是必须在XML文件中声明。

对于构造函数参数名不可用的少数情况（通常如果编译字节码时没有调试信息），可以回退到参数索引，如下所示：

```
<!-- c-namespace index declaration -->
<bean id="beanOne" class="x.y.ThingOne" c:_0-ref="beanTwo" c:_1-ref="beanThree"
    c:_2="something@somewhere.com"/>
```

更多教程请访问:www.flydean.com

由于XML语法的原因，index需要有前导字符_，因为XML属性名不能以数字开头（即使某些IDE允许这样做）。相应的索引表示也可用于< constructor-arg>元素，但不常用，因为简单顺序通常就足够了。

在实践中，构造函数解析机制在匹配参数方面非常有效，因此，除非确实需要，否则我们建议通过配置使用name。

复合属性名称

设置bean properties时，可以使用复合或嵌套的属性名，只要路径的所有组件（最终属性名除外）不为null。考虑以下bean定义：

```
<bean id="something" class="things.ThingOne">
    <property name="fred.bob.sammy" value="123" />
</bean>
```

something bean有一个fred属性，fred有个bob属性，bob有个sammy属性，最终sammy被设置为123，在构造bean之后，something的fred属性和fred的bob属性不能为空。否则，将引发NullPointerException。

1.4.3 使用depends-on

如果一个bean是另一个bean的依赖项，这通常意味着一个bean被设置为另一个bean的属性。通常，你可以使用基于XML的配置元数据中的< ref/>元素来实现这一点。然而，有时bean之间的依赖关系并不直接。例如，当类中的静态初始值设定项需要触发时，例如数据库驱动程序注册。depends-on属性可以显式强制一个或多个bean在使用此元素的bean初始化之前进行初始化。以下示例使用depends-on属性表示对单个bean的依赖：

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

要表示对多个bean的依赖关系，请提供一个bean名称列表作为depends-on属性的值（逗号、空白和分号都是有效的分隔符）：

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

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

> depends-on属性既可以指定初始化时间依赖项，也可以指定相应的销毁时间依赖项（仅在单例bean的情况下）。定义与给定bean的依赖关系的依赖bean在给定bean本身被销毁之前首先被销毁。因此，depends-on还可以控制关机顺序。

1.4.4 惰性初始化bean

默认情况下，ApplicationContext作为初始化过程的一部分，以预实例化的方式建和配置所有的单例bean。一般来说，我们提倡这种预实例化方式，因为如果配置或周围环境发送错误会立即被发现，而不是几小时甚至几天之后。如果不想使用这种方式，可以通过将bean定义标记为lazy-initialized来防止单例bean的预实例化。一个lazy-initialized bean告诉IOC容器在第一次请求bean实例时创建它，而不是在启动时。

在XML中，此行为由< bean/>元素上的lazy-init 属性控制，如下示例所示：

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

更多教程请访问:www.flydean.com

当ApplicationContext使用前面的配置时，当ApplicationContext启动时，lazy bean不会被预先实例化，而not.lazy bean则会被预先实例化。

但是，当一个惰性初始化bean是一个非惰性初始化的singleton bean的依赖项时，ApplicationContext会在启动时创建惰性初始化bean，因为它必须满足singleton的依赖项。惰性初始化bean被注入到其他非惰性初始化的单例bean中。

你还可以通过在< beans/>元素上使用默认的lazy init属性在容器级别控制lazy初始化，下面的示例显示：

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

1.4.5.自动装载合作者

Spring容器可以自动连接协作bean之间的关系。通过检查ApplicationContext的内容，可以让Spring自动解析bean的合作者（其他bean）。自动装载具有以下优点：

- 自动装载可以显著减少指定属性或构造函数参数。（本章其他地方讨论的bean模板等其他机制在这方面也很有价值。）
- 自动装载可以随着对象的发展更新配置。例如，如果需要向类添加依赖项，则可以自动满足该依赖项，而无需修改配置。因此，自动装载在开发期间尤其有用，也不需要在代码库变得更稳定时切换到显式装配。

使用基于XML的配置元数据（请参见依赖注入）时，可以使用< bean/>元素的autowire属性为bean定义指定autowire模式。autowiring功能有四种模式。你可以为每个bean指定autowiring，从而可以选择要autowiring的对象。下表介绍了四种autowiring模式：

Table 2. Autowiring modes

模式	说明
no	（默认）无自动装载。bean必须引用由ref定义的元素。对于较大的部署，不建议更改默认设置，因为显式指定合作者可以提供更大的控制度和清晰性。在某种程度上，它记录了系统的结构。
byName	按属性名称自动装载。Spring寻找与需要自动装载的属性同名的bean。例如，如果bean定义被设置为按名称自动装载，并且它包含一个master属性（即，它有一个setMaster（..）方法），那么spring将查找名为master的bean定义并使用它来设置该属性。
byType	如果容器中只有一个属性类型的bean，则允许自动装载属性。如果存在多个，则会引发一个致命的异常，这表示你不能为该bean使用byType自动装载。如果没有匹配的bean，则不会发生任何事情（未设置属性）。
constructor	类似于byType，但适用于构造函数参数。如果容器中不只有一个构造函数参数类型的bean，则会引发致命错误。

使用byType或构造函数自动装载模式，可以装载数组和类型化集合。在这种情况下，将提供容器中与预期类型匹配的所有满足自动装载的候选者，以满足依赖关系。如果所需的键类型为字符串，则可以自动装载强类型Map实例。自动装载的Map实例的值由匹配预期类型的所有bean实例组成，并且Map实例的键是相应的bean名称。

自动装载的限制和缺陷

自动装载在一个项目中持续使用时效果最好。如果一般不使用自动装载，或者只使用它来装载一个或两个bean定义，开发人员会很困惑。

自动装载的局限性和缺点：

- property和constructor-arg设置中的显式依赖项始终会覆盖自动装载。不能自动装载简单属性，如基元、Strings和Classes（以及此类简单属性的数组）。这是设计上的限制。
- 自动装载不如显式装载精确。尽管如前表所述，Spring谨慎地避免猜测，以防出现可能产生意外的歧义结果。但是你的Spring管理对象之间的关系将不会在文档中明确标出。

更多教程请访问:www.flydean.com

- 有些工具可能无法为自动装载生成相应的文档
- 可以与setter方法或要自动注入的构造函数参数类型可能与容器中的多个bean定义相匹配。如果是数组、集合或映射实例，这没有问题。但是，对于期望单个值的依赖项，这种模糊性会引发异常。

在最后一种情况中，你有几个选项：

- 放弃自动注入，取而代之的是显式注入。
- 通过将bean定义的autowire候选属性设置为false，避免自动注入bean定义，如下一节所述。
- 通过将< bean/>元素的primary属性设置为true，将单个bean指定为主要候选对象.
- 使用基于注解的配置中可用的更细粒度的控件

从自动装载中排除Bean

你可以从自动装载中排除bean。在Spring的XML格式中，将< bean/>元素的autowire-candidate属性设置为false。则这个bean在自动装载时将会不可用（包括注解样式的配置，如@Autowired）。

> autowire-candidate属性设计为只影响type-based的自动注解。它不会影响显式的按名字的引用，即使指定的bean没有标记为autowire候选对象，这些引用也会得到解析。因此，如果名称匹配，则按name自动装载仍会注入bean。

你还可以根据模式匹配来限制自动装载的候选对象。顶级的< beans/>元素在其默认的default-autowire-candidates属性中接受一个或多个模式。例如，要将autowire候选状态限制为名称以Repository结尾的任何bean，请提供值*Repository。要提供多个模式，请在逗号分隔的列表中定义它们。

如果显示的设定bean的autowire-candidate属性为true或false那么模式匹配规则将失效。

这些技术对于你不希望通过自动注解注入其他bean的bean很有用。这并不意味着被排除的bean本身不能通过使用自动注解进行配置。只是说明这个bean并不是自动注解其他bean的候选者。

1.4.6. 方法注入

在大多数应用程序场景中，容器中的大多数bean都是单例的。当一个单例bean需要与另一个单例bean协作，或者一个非单例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bean的属性来处理依赖关系。当bean的生命周期不同时，就会出现一个问题。假设singleton bean a需要使用非singleton（原型）bean b，可能在a上的每个方法调用上都需要使用b。容器只创建一次singleton bean a，因此只获得一次设置属性的机会。容器不能在每次需要bean B时向bean A提供一个新的bean B实例。

解决办法是放弃一些控制反转。你可以通过实现ApplicationContextAware接口使bean aware容器，每次bean a需要b时，通过对容器进行getbean（“b”）调用来请求（通常是新的）bean b实例。下面的示例显示了这种方法：

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

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

public class CommandManager implements ApplicationContextAware {

    private ApplicationContext applicationContext;

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

更多教程请访问:www.flydean.com

```
    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;
    }
}
```

这种方法并不可取的，因为业务代码和Spring框架产生了耦合。方法注入是Spring IoC 容器的一个高级特性，它可以很好的处理这种情况。

查找方法注入

查找方法注入是指容器重写container-managed bean上的方法，并返回容器中另一个命名bean。查找通常涉及一个原型bean，如前一节中描述的场景中所示。Spring框架通过使用cglib库中的字节码动态生成覆盖该方法的子类来实现该方法注入。

> - 为了使这个动态子类工作，SpringBean容器子类不能是final类，要重写的方法也不能是final类。
> - 单元测试具有抽象方法的类需要你自己对类进行子类化，并提供抽象方法的stub实现。
> - 组件扫描还需要具体的方法，这会提供给具体的类来使用。
> - 另一个关键限制是查找方法不适用于工厂方法，尤其不适用于配置类中的@Bean方法，因为在这种情况下，容器不负责创建实例，因此无法动态创建运行时生成的子类。

对于前面代码段中的CommandManager类，Spring容器会动态重写createCommand（）方法的实现。CommandManager类没有任何Spring依赖项，正如重新编写的示例所示：

```
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的客户类（本例中为CommandManager）中，要注入的方法需要以下形式的签名：

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

如果方法是抽象的，动态生成的子类会自动实现该方法。否则会将其覆盖。下面是例子：

```
<!-- 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"/>
```

更多教程请访问:www.flydean.com

```
</bean>
```

标识为commandManager的bean在需要myCommand bean的新实例时调用自己的createCommand（）方法。如果实际上需要这样做，那么必须将myCommand bean部署为原型。如果它是单例的，则每次返回相同的myCommand bean实例。

或者，在基于注解的组件模型中，可以通过@Lookup annotation声明查找方法，如下例所示：

```
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，你可以依赖目标bean根据lookup方法声明的返回类型进行解析：

```
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，通常应该用具体的存根实现声明这种带注解的查找方法，以便它们与Spring的组件扫描规则兼容，默认情况下抽象类会被忽略。此限制不适用于显式注册或显式导入的bean类。

> 访问不同作用域目标bean的另一种方法是ObjectFactory/Provider注入点。
>
> 你还可能发现ServiceLocatorFactoryBean（在org.springframework.beans.factory.config包中）非常有用。

任意方法替换

与查找方法注入相比，方法注入的一种不太有用的形式是能够用另一个方法替换托管bean中的任意方法。你可以跳过本节的其余部分，直到实际需要此功能为止。

对于基于XML的配置元数据，可以使用replaced-method的方法元素为部署的bean替换现有的方法实现。考虑下面的类，它有一个我们想要重写的名为computeValue的方法：

```
public class MyValueCalculator {

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

    // some other methods...
}
```

一个实现了org.springframework.beans.factory.support.MethodReplacer接口的类提供了新的方法，如下所示：

```
/**
```

更多教程请访问:www.flydean.com

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

部署原始类并指定方法重写的bean定义如下所示：

```
<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"/>
```

你可以在< replaced-method/>元素中使用一个或多个< arg-type/>元素来指示要重写的方法的方法签名。只有当方法被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的子字符串。例如，以下所有内容都匹配java.lang.String：

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

因为参数的数量经常足以区分每个可能的选项，所以通过允许输入与参数类型匹配的最短字符串，可以节省大量的输入内容。

更多教程请访问:www.flydean.com

1.5 Bean作用域

当你创建一个bean定义时，你将创建一个recipe用来创建由该bean定义的类的实际实例。bean定义是一个recipe的想法很重要，因为这意味着，和类一样，你可以从一个recipe创建多个对象实例。

你不仅可以控制要插入到从特定bean定义创建的对象中的各种依赖项和配置值，还可以控制从特定bean定义创建的对象的范围。这种方法是强大和灵活的，因为你可以通过配置来选择你创建的对象的作用域，而不必在Java类级别对对象作用域进行设置。bean可以选择被部署在任何一个作用域中。

Spring框架支持六个作用域，其中四个作用域仅在使用web-aware的ApplicationContext时可用。当然，你可以创建自定义作用域。

下表介绍了支持的作用域：

Table 3. Bean Scope

作用域	描述
singleton	（默认）Spring IOC容器将单个bean定义绑定到单个对象实例。
prototype	单个bean定义可以被绑定到多个对象实例
request	单个bean定义被绑定到单个的Http请求生命周期。就是每一次HTTP请求都有属于他自己的单独实例创建，这个只在web-aware的Spring ApplicationContext中有效。
session	单个bean的定义会被绑定到Http Session的生命周期中，这个只在web-aware的Spring ApplicationContext中有效。
application	单个bean的定义会被绑定到ServletContext的生命周期中，这个只在web-aware的Spring ApplicationContext中有效。
websocket	单个bean的定义会被绑定到WebSocket的生命周期中，这个只在web-aware的Spring ApplicationContext中有效。

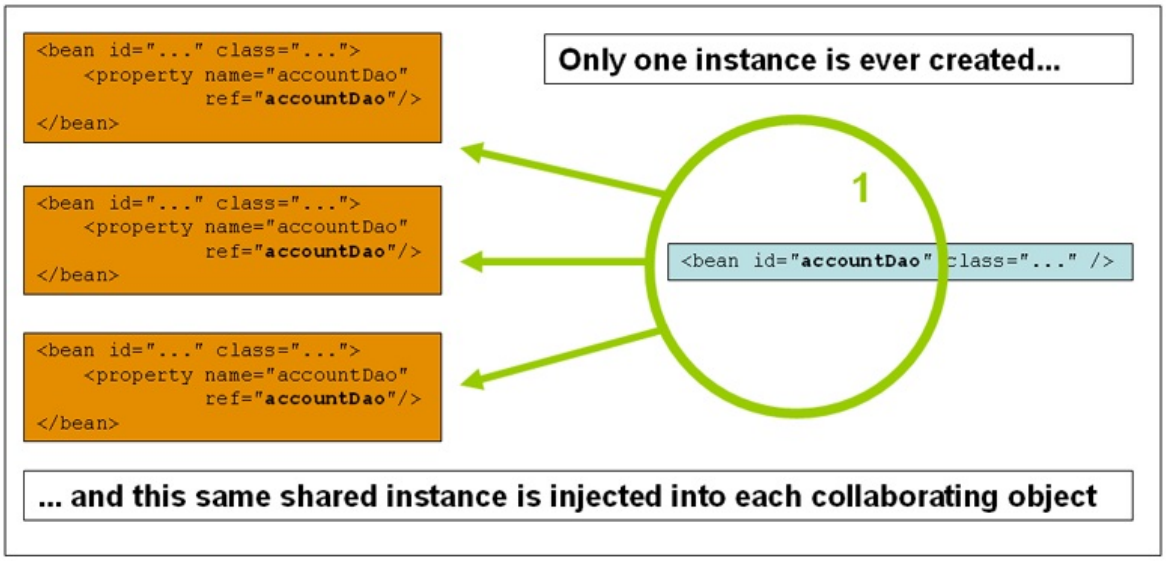
> 从Spring3.0开始，添加了thread作用域，但默认情况下不被注册：请参阅SimpleThreadScope。 从Spring4.2开始，添加了transaction作用域：SimpleTransactionScope。有关如何注册这些或任何其他自定义范围的说明，请参阅使用自定义范围。

1.5.1 Singleton作用域

只会创建一个singleton bean的共享实例，所有与该bean ID相匹配的bean的请求都会导致Spring容器返回这个特定的bean实例。

换句话说，当你定义一个bean定义并且它的作用域是Singleton时，Spring IOC容器只创建由该bean定义的对象的一个实例。这个单个实例存储在这样的单实例bean的缓存中，所有随后的请求和对这个名为bean的请求都返回缓存的对象。下图显示了Singleton作用域的工作方式：

更多教程请访问:www.flydean.com



Spring对单例bean的概念不同于《Gang of Four (GoF) 》一书中定义的单例模式。gof singleton硬编码对象的作用域，这样每个类加载器只能创建一个特定类的实例。Spring Singleton的范围是针对于每个容器和每个bean的。这意味着，如果在单个Spring容器中为特定类定义一个bean，那么Spring容器将创建由该bean定义的类的一个且仅一个实例。单例作用域是Spring中的默认作用域。要在XML中将bean定义为singleton，如下例所示：

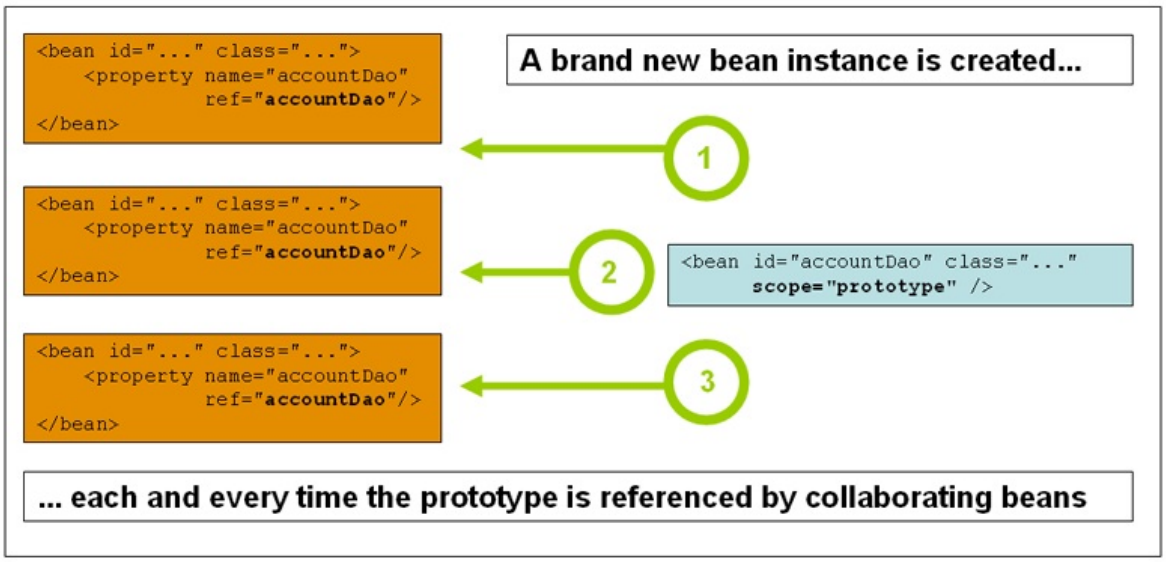
```
<bean id="accountService" class="com.something.DefaultAccountService"/>

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

1.5.2 Prototype作用域

每次请求特定bean时，bean如果是Prototype作用域，当bean被注入到另一个bean中，或者通过对容器的getBean（）方法调用请求时，都会创建一个新的bean实例。通常来说，你应该为所有有状态bean使用Prototype作用域，为无状态bean使用Singleton作用域。

下图说明了prototype作用域：



更多教程请访问:www.flydean.com

（数据访问对象（DAO）通常不配置为原型，因为典型的DAO不具有任何会话状态。使用单例模式将会更方便。）

下面的示例将bean定义为XML中的原型：

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

与其他生命周期不同，Spring不负责prototype bean的完整生命周期。容器实例化、配置或以其他方式组装原型对象，然后将其交给客户端，之后就不再记录该原型实例。

因此，尽管初始化生命周期回调方法会在所有对象上被调用，不管生命周期如何，但在原型的情况下，不会调用配置的销毁生命周期回调。客户端代码必须自行清理原型作用域内的对象，并释放原型bean所拥有的昂贵资源。

要让Spring容器释放原型作用域bean所拥有的资源，请尝试使用自定义bean post-processor，该处理器包含对需要清理的bean的引用。

从某些方面来讲，Spring容器在prototype-scoped bean中的作用类似于Java new操作符。这之后的所有生命周期管理都必须由客户端处理。（有关Spring容器中bean生命周期的详细信息，请参阅生命周期回调。）

1.5.3 Singleton Beans依赖于 Prototype-bean

当你使用依赖于原型bean的单例bean时，请注意依赖项是在实例化时解析的。因此，如果你将一个原型作用域的bean注入到一个单例bean中，那么将实例化一个新的原型bean，然后将依赖项注入到单例bean中。这个原型实例是提供给单例bean的唯一实例。

但是，假设你希望单例bean在运行时重复获取原型bean的新实例。则不能将原型bean注入到singleton bean中，因为当Spring容器实例化singleton bean并解析和注入其依赖项时，该注入只发生一次。如果在运行时需要多个原型bean的新实例，请参照方法注入。

1.5.4 Request, Session, Application, 和WebSocket作用域

Request, Session, Application, 和WebSocket作用域仅在使用web-aware的Spring ApplicationContext实现（如XMLWebApplicationContext）时可用。如果将这些作用域与常规的Spring IOC容器（如ClassPathXmlApplicationContext）一起使用，则会引发一个IllegalstateException，它会抱怨未知的bean作用域。

初始Web配置

为了支持Request, Session, Application, 和WebSocket作用域，在定义bean之前需要一些小的初始配置。（对于标准范围：singleton和prototype，不需要此初始设置。）

如何完成这个初始设置取决于你的特定servlet环境。

如果你在SpringWebMVC中访问作用域bean，实际上，在Spring DispatcherServlet处理的请求中，不需要特殊设置。DispatcherServlet已包含了所有相关状态。

如果使用servlet 2.5 Web容器，并且请求在Spring的DispatcherServlet之外处理（例如，使用JSF或Struts时），则需要注册org.springframework.web.context.request.RequestContextListener ServletRequestListener。对于servlet 3.0+，这可以通过使用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>
```

更多教程请访问:www.flydean.com

```
    ...
</web-app>
```

或者，如果监听器设置有问题，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，因此你必须根据需要更改它。下面的列表显示了Web应用程序的筛选部分：

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

DispatcherServlet, RequestContextListener, 和 RequestContextFilter 都做了同样的事情，那就是将HTTP请求对象绑定到为该请求提供服务的线程。这使得请求和会话范围的bean都可以在调用链中进一步使用。

Request scope

考虑下面的定义：

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

Spring容器通过为每个HTTP请求使用LoginAction定义来创建一个新的LoginAction bean实例。也就是说，LoginAction bean的作用域是在HTTP请求级别。你可以根据需要更改所创建实例的内部状态，因为从相同的LoginAction bean定义创建的其他实例在状态中看不到这些更改。它们是针对单个请求的。当请求完成处理时，将丢弃该请求的作用域bean。

当使用注解驱动的组件或Java配置时，可以使用@RequestScope来将组件分配给request作用域。以下示例显示了如何执行此操作：

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

Session Scope 考虑下面的定义：

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

Spring容器通过在单个HTTP Session的生命周期中使用UserPreferences bean定义创建一个新的UserPreferences bean实例。换句话说，UserPreferences bean在HTTP Session级别有效。与request scope的bean一样，你可以根据需要更改创建的实例的内部状态，因为其他也使用从相同的用户首选项bean定义创建的实例的HTTP session实例在状态中看不到这些更改，因为它们是特定于单个HTTP session的。当最终丢弃HTTP session时，也会丢弃作用于该特定HTTP session的bean。

当使用注解驱动的组件或Java配置时，可以使用@SessionScope将组件分配给session 作用域。

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

更多教程请访问:www.flydean.com

Application Scope

考虑下面的定义：

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

Spring容器对整个Web应用程序创建一次AppPreferences bean定义的AppPreferences bean新实例。也就是说，AppPreferences bean的作用域是ServletContext级别，并存储为常规ServletContext属性。这有点类似于Spring Singleton Bean，但在两个重要方面有所不同：它是每个ServletContext的单例，而不是每个Spring的“ApplicationContext”（在任何给定的Web应用程序中都可能有多个），并且它实际上是公开的，因此作为ServletContext属性可见。

当使用注解驱动的组件或Java配置时，可以使用 @ApplicationScope注解来将组件分配给application scope。以下示例显示了如何执行此操作：

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

作用域Bean作为依赖

Spring IOC容器不仅管理对象（bean）的实例化，还管理协作者（或依赖项）的注入。如果要将HTTP request-scoped bean注入（例如）较长作用域的另一个bean中，可以选择插入AOP代理来代替作用域bean。也就是说，你需要插入一个代理对象，该对象与作用域对象公开相同的公共接口，但该对象可以从相关作用域（如HTTP请求）中检索实际目标对象，并将方法调用委托给实际对象。

> 你还可以在作用域为singleton的bean之间使用 <aop:scoped-proxy/>来引用beans，然后通过一个可序列化的中间代理，从而能够在反序列化时重新获取目标singleton bean。
>
> 对prototype bean声明 <aop:scoped-proxy/>时，共享代理上的每个方法调用都会导致创建一个新的目标实例，然后将调用转发到该实例。
>
> 此外，作用域代理并不是从较短的作用域获取bean的唯一方法。你还可以将注入点（即构造函数、setter参数或autowired字段）声明为ObjectFactory<MyTargetBean>，允许每次需要时getObject（）调用根据需要检索当前实例-而不保留该实例或将其单独存储。
>
> 作为扩展变量，你可以声明ObjectFactory，它提供了几个额外的访问变量，包括getIfAvailable和getIfUnique。
>
> 它的JSR-330变种称为Provider，和Provider<MyTargetBean>一起使用，从而在每次查询时都会调用get()方法。

下面的配置文件只有一行，但是我们需要弄清楚其背后的原因。

```
<?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
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd">

    <!-- an HTTP Session-scoped bean exposed as a proxy -->
    <bean id="userPreferences" class="com.something.UserPreferences" scope="session">
        <!-- instructs the container to proxy the surrounding bean -->
        <aop:scoped-proxy/>
    </bean>
```

更多教程请访问:www.flydean.com

```
    <!-- a singleton-scoped bean injected with a proxy to the above bean -->
    <bean id="userService" class="com.something.SimpleUserService">
        <!-- a reference to the proxied userPreferences bean -->
        <property name="userPreferences" ref="userPreferences"/>
    </bean>
</beans>
```

要创建这样的代理，可以将一个子<aop:scoped-proxy/>元素插入到作用域bean定义中（请参见选择要创建的代理类型和基于XML模式的配置）。为什么在请求、会话和自定义范围的bean需要<aop:scoped-proxy/>元素？考虑下面的singleton bean定义，并将其与你需要为上述范围定义的内容进行对比（请注意，下面的userPreferences bean定义并不完整）：

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

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

在前面的示例中，singleton bean（UserManager）注入了一个HTTP会话范围的bean（UserPreferences）的引用。这里的问题是，UserManager bean是一个单例：每个容器只实例化一次，它的依赖项（在本例中只有一个，即UserPreferences bean）也只注入一次。这意味着UserManager bean只在完全相同的UserPreferences对象（即最初注入它的对象）上操作。

这不是将寿命较短的作用域bean注入寿命较长的作用域bean时需要的行为（例如，将HTTP会话作用域协作bean作为依赖项注入到singleton bean中）。相反，你需要一个UserManager对象，并且在HTTP会话的生命周期中，你需要一个特定于HTTP会话的UserPreferences对象。因此，容器创建一个对象，该对象与UserPreferences类公开完全相同的公共接口（理想情况下是一个UserPreferences实例的对象），该对象可以从作用域机制（HTTP请求、会话等）获取实际的UserPreferences对象。

容器将这个代理对象注入到UserManager bean中，后者不知道这个用户首选项引用是代理。在本例中，当一个UserManager实例在依赖项注入的UserPreferences对象上调用一个方法时，它实际上是在代理上调用一个方法。然后，代理从HTTP会话（在本例中）中获取real userPreferences对象，并将方法调用委托给检索到的real userPreferences对象。

因此，在将请求和会话范围的bean注入到协作对象时，你需要以下（正确和完整）配置，如下示例所示：

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

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

选择创建的代理类型

一般来说当Spring使用<aop:scoped-proxy/>来创建代理时，会使用CGLIB。

> CGLIB代理只拦截公共方法调用！不要在此类代理上调用非公共方法。它们不会委托给实际作用域目标对象。

或者，可以通过为<aop:scoped-proxy/>元素的proxy-target-class属性的值指定false，将spring容器配置为为此类作用域bean创建标准的基于JDK接口的代理。使用基于JDK接口的代理意味着你不需要在应用程序类路径中加入其他库。但是，它意味着作用域bean的类必须实现至少一个接口，并且所有注入作用域bean的合作者必须通过其接口之一引用bean。以下示例显示了基于接口的代理：

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

更多教程请访问:www.flydean.com

```
</bean>

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

1.5.5 自定义作用域

Bean的作用域范围是可以扩展的，你可以定义自己的作用域，甚至重新定义现有作用域，尽管后者被认为是不好的做法，并且不能覆盖内置的单例和原型作用域。

创建自定义作用域

要将自定义作用域集成到Spring容器中，需要实现本节中描述的org.springframework.beans.factory.config.Scope接口。有关如何实现自己的作用域的概念，请参阅Spring框架本身提供的作用域实现和作用域JavaDoc，后者更详细地解释了需要实现的方法。

作用域接口有四种方法来从作用域中获取对象，将它们从作用域中移除，并让它们被销毁。 例如，会话作用域实现返回会话作用域bean（如果它不存在，则在将其绑定到会话以供将来使用，该方法返回bean的新实例）。以下方法从所在作用域返回对象：

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

例如，会话作用域实现从基础会话中删除会话作用域bean。应返回对象，但如果找不到具有指定名称的对象，则可以返回空值。以下方法将对象从基础作用域中移除：

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

以下方法注册当作用域被销毁或作用域中的指定对象被销毁时应执行的回调：

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

以下方法获取基础作用域的会话标识符：

```
String getConversationId()
```

对于每个作用域，这个标识符是不同的。对于会话范围的实现，此标识符可以是会话标识符。

使用自定义作用域

当你写好一个Scope的实现之后，需要通知Spring容器，下面是注册到Spring容器的核心方法：

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

此方法在ConfigurableBeanFactory接口中，该接口可通过Spring附带的大多数具体应用程序上下文实现的BeanFactory属性获得。

registerScope（..）方法的第一个参数是与作用域关联的唯一名称。在Spring容器本身中，此类名称的示例是singleton和prototype。registerScope（..）方法的第二个参数是要注册和使用的自定义作用域实现的实际实例。

假设你编写自定义作用域实现，然后按照下一个示例所示注册它。

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

更多教程请访问:www.flydean.com

接下来你可以创建使用自定义作用域的Bean：

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

使用自定义作用域实现，你不仅限于作用域的编程式注册。你还可以通过使用CustomScopeConfigurer类以声明方式进行作用域注册，如下示例所示：

```
<?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:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        https://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="thing2" class="x.y.Thing2" scope="thread">
        <property name="name" value="Rick"/>
        <aop:scoped-proxy/>
    </bean>

    <bean id="thing1" class="x.y.Thing1">
        <property name="thing2" ref="thing2"/>
    </bean>

</beans>
```

> 当你在FactoryBean的实现中使用<aop:scoped-proxy/>时，是这个factory bean本身被代理，而不是他返回的对象被代理getObject()。

更多教程请访问:www.flydean.com

1.6 自定义Bean

Spring提供了一系列的接口让你可以自定义Bean。

1.6.1 生命周期回调

要与容器中bean的生命周期进行交互，可以实现Spring的InitializingBean和DisposableBean接口。容器为前者调用afterPropertiesSet（），为后者调用destroy（），以便bean在初始化和销毁bean时执行某些操作。

> JSR-250 的@PostConstruct和@PreDestroy注解是生命周期回调的最推荐的方式，因为他们和Spring框架解耦。
>
> 如果你不想使用JSR-250,又想解耦，那么可以考虑使用init-method 和destroy-method 这样两个定义bean的元数据。

在内部，Spring框架使用BeanPostProcessor实现来处理它可以找到的任何回调接口，并调用适当的方法。如果你需要自定义特性或其他Spring默认不提供的生命周期行为，你可以自己实现BeanPostProcessor。有关详细信息，请参见容器扩展点。

除了初始化和销毁回调之外，Spring管理的对象还可以实现生命周期接口，以便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。

生命周期回调接口在本节中进行了描述。

初始化回调

org.springframework.beans.factory.InitializingBean 接口可以让bean在Spring初始化之后，执行自定义的初始化流程，接口只有一个方法InitializingBean：

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

但是我们不建议你使用InitializingBean，这个和Spring框架耦合在一起。我们推荐使用@PostConstruct或者定义一个POJO的初始化方法。如果使用XML配置，可以使用init-method来指定一个void返回值的方法。如果使用Java配置，可以配置@Bean的initMethod属性，参照下面例子：

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

```
public class ExampleBean {

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

上面例子和下面是等价的：

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

```
public class AnotherExampleBean implements InitializingBean {

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

更多教程请访问:www.flydean.com

但是，最上面的例子和Spring是解耦的。

Destruction回调

要实现销毁回调，可以实现接口org.springframework.beans.factory.DisposableBean， 他提供了一个DisposableBean方法：

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

我们不建议你使用DisposableBean，因为他和Spring框架耦合，我们推荐你使用@PreDestroy注解，或者一个能被bean定义支持的通用方法。在XML配置中，可以使用<bean/>的destroy-method属性。在Java配置中，可以设置@Bean的destroyMethod属性。如下所示：

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

public class ExampleBean {

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

上面例子和下面作用一致：

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

public class AnotherExampleBean implements DisposableBean {

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

当然，最上面的例子和Spring解耦。

> 可以为<bean>元素的destroy-method属性指定一个特殊（推断）值，该值指示spring自动检测特定bean类上的公共close或shutdown。（可以实现java.lang.AutoCloseable或java.io.Closeable）
>
> 你还可以在<beans>元素的默认default-destroy-method属性上设置此特殊（推断）值，以将此行为应用于整个bean集（请参见默认初始化和销毁方法）。注意，这是Java配置的默认行为。

默认的Initialization和Destroy方法

在编写不使用Spring特定的InitializingBean和DisposableBean回调接口的初始化和销毁方法回调时，通常会命名为init（）、initialize（）、dispose（）。理想情况下，这样的生命周期回调方法的名称在项目中是标准化的，这样所有开发人员都使用相同的方法名称并确保一致性。

你可以配置Spring容器去“查找”特定名字的初始化和销毁方法。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名为init（）的初始化回调，而无需为每个bean定义配置init method=“init”属性。Spring IOC容器在创建bean时将会调用该方法。此功能强调命名的一致性。

假设初始化回调方法名为init（），销毁回调方法名为destroy（）。然后，你的类类似于以下示例：

```
public class DefaultBlogService implements BlogService {

    private BlogDao blogDao;

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

更多教程请访问:www.flydean.com

```
    }

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

接下来，你可以在xml中定义它：

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

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

</beans>
```

<beans>中的default-init-method表明所有的bean在Spring IOC容器进行初始化的时候，都会在合适的时间调用 init方法。

同样的，你可以在<beans>中配置default-destroy-method 。

如果想覆盖默认的回调函数，可以在<bean>中使用init-method或者destroy-method。

Spring容器保证在向bean提供所有依赖项之后立即调用配置的初始化回调。因此，在对原始bean引用调用初始化回调时，AOP拦截器等尚未作用于bean。

首先创建一个目标bean，然后才能应用一个带有拦截器链的AOP代理。如果目标bean和代理是单独定义的，那么你的代码甚至可以影响原始目标bean。因此，将拦截器应用于init方法会导致不一致的结果，因为这样做会将目标bean的生命周期耦合到其代理或拦截器，并在代码直接与原始目标bean交互时留下奇怪的语义。

总结生命周期机制

在Spring2.5中，你有3种方式来控制bean的生命周期：

- InitializingBean和DisposableBean回调接口
- 自定义init() 和destroy() 方法。
- @PostConstruct 和 @PreDestroy注解.

你可以将3种方式结合使用。

> 如果多种方式一起使用，每种方式都配置了一个不同的方法名，那么他们的执行顺序将会如下面的顺序所示。 但是如果他们都配置了同一个方法名，那么该方法只会执行一次。

如果为initialization配置了多种生命周期的多个名字，那么执行顺序如下：

- @PostConstruct 的方法注解
- InitializingBean接口里面的afterPropertiesSet() 方法
- 自定义的init() 方法

Destroy也是一样的顺序：

- @PreDestroy 的方法注解
- DisposableBean接口里面的destroy() 方法
- 自定义的destroy() 方法

启动和关闭回调

Lifecycle接口为需要管理自己生命周期的对象（例如需要启动和关闭一些后台进程）提供了最基本的方法。

更多教程请访问:www.flydean.com

```
public interface Lifecycle {

    void start();

    void stop();

    boolean isRunning();
}
```

任何Spring管理的对象都可以实现Lifecycle接口。当ApplicationContext收到一个start或者stop信号时，他会将该信号传递给所有的Lifecycle接口的实现。这是通过LifecycleProcessor委托来实现的：

```
public interface LifecycleProcessor extends Lifecycle {

    void onRefresh();

    void onClose();
}
```

LifecycleProcessor是Lifecycle的扩展。它还添加了另外两种方法来响应正在刷新和关闭的上下文。

> 请注意，org.springframework.context.Lifecycle接口只是提供了一个简单的start和stop通知协议，但是并没有实现上下文刷新时自动启动的功能。如果想更好的控制自动启动的功能，可以实现org.springframework.context.SmartLifecycle接口。
>
> 另外，stop通知不能保证一定在destruction之前执行。通常情况下所有的Lifecycle都会收到一个stop通知在总的destruction回调执行之前。但是在热刷新或者终止刷新尝试时，只会执行destroy方法。

启动和关闭调用的顺序可能很重要。如果任何两个对象之间存在“依赖”关系，则依赖方在依赖之后开始，在依赖之前停止。然而，有时，直接依赖性是未知的。你可能只知道某个类型的对象应该先于另一个类型的对象开始。在这些情况下，SmartLifecycle接口定义了另一个选项，即在其父接口上定义的getPhase（）方法。下表显示了Phased接口的定义：

```
public interface Phased {

    int getPhase();
}
```

下面是SmartLifecycle的定义：

```
public interface SmartLifecycle extends Lifecycle, Phased {

    boolean isAutoStartup();

    void stop(Runnable callback);
}
```

启动时，具有最低phase的对象首先启动。停止时，按相反顺序执行。因此，实现SmartLifecycle且其getPhase（）方法返回Integer.MIN_VALUE值的对象将是第一个开始和最后一个停止的对象。

同样的，Integer.MAX_VALUE值的phase值将表面对象应在最后启动并首先停止（可能是因为它取决于要运行的其他进程）。在考虑phase值时，还必须知道不实现SmartLifecycle的任何“正常”生命周期对象的默认阶段是0。因此，任何负phase值都表示一个对象应该在这些标准组件之前开始（并在它们之后停止）。对于任何正phase值，顺序相反。

SmartLifecycle定义的Stop方法接受回调。任何实现都必须在该实现的关闭过程完成后调用该回调的run（）方法。在需要的时候，可以实现异步的关闭，因为LifecycleProcessor接口的默认实现DefaultLifecycleProcessor在每个phase中的对象中调用该回调时，会等待一个超时值。每个phase默认超时为30秒。你可以通过在上下文中定义一个名为lifecycleProcessor的bean来覆盖默认的生命周期处理器实例。如果只想修改超时，那么定义以下内容就足够了：

更多教程请访问:www.flydean.com

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

如前所述，LifecycleProcessor接口还定义了用于刷新和关闭上下文的回调方法。当stop（）被显示调用的时候会启动shutdown的进程，但是这个方法只会在上下文关闭时被调用。另一方面，“刷新”回调启用了SmartLifecycle bean的另一个功能。刷新上下文时（在所有对象都已实例化和初始化之后），将调用该回调。此时，默认的生命周期处理器检查每个SmartLifecycle对象的isAutoStartup（）方法返回的布尔值。如果为true，则该对象将在该点启动，而不是等待显式调用上下文或其自身的start（）方法（与上下文refresh不同，对于标准上下文实现，上下文start不会自动触发）。如前所述，phase值和任何“depends-on”关系决定启动顺序。

在非Web应用程序中优雅的关闭Spring IoC容器

> 本小节只适用于非web的应用程序。当web应用程序关闭时，Spring的web基础的ApplicationContext实现，已经有代码优雅的关闭 Spring IoC 容器。

如果你使用Spring IOC容器在非web应用程序环境中（富文本桌面程序），需要注册一个shutdown hook到JVM中。这样将保证优雅的关闭，并且在单例bean中调用相关的销毁方法，让所有的资源得到释放。当然，你必须正确的配置和实现这些销毁方法。

调用ConfigurableApplicationContext接口中的registerShutdownHook()来注册一个shutdown hook， 如下所示：

```
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("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...
    }
}
```

1.6.2 ApplicationContextAware 和 BeanNameAware

当ApplicationContext创建了org.springframework.context.ApplicationContextAware接口的一个实现对象时。实例提供了一个对ApplicationContext的引用。下面是ApplicationContextAware的定义：

```
public interface ApplicationContextAware {

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

因此，bean可以手动控制创建他的ApplicationContext，可以通过ApplicationContext接口，也可以将这个引用转换成ApplicationContext的子类（比如提供了额外功能的ConfigurableApplicationContext）。

他的一个应用就是在程序中获取其他的beans。这种使用方式有时候会很有用，但是因为其和Spring耦合在一起，违背了IOC原则，所以应该避免这样使用。应该将协作者设置成bean的属性值。

ApplicationContext的其他方法提供了文件资源，发布应用程序事件和访问MessageSource的功能。

更多教程请访问:www.flydean.com

在Spring2.5中，自动装载是另外一个获取到ApplicationContext引用的方法。传统的constructor和byType自动装载模式，能将ApplicationContext当做constructor的参数，或者setter方法的参数。为了更灵活的使用，比如自动装载字段或者方法的多个参数，可以使用@Autowired注解在字段或者方法上。这样ApplicationContext将会自动装配到需要的地方。

当ApplicationContext创建了一个org.springframework.beans.factory.BeanNameAware的实现类时，这个类提供了对其在关联对象中定义的名字的引用。 如下所示：

```
public interface BeanNameAware {

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

在填充普通bean属性之后，在初始化回调（如InitializingBean、afterPropertiesSet或自定义init方法）之前该回调会被调用。

1.6.3 其他的Aware接口

除了ApplicationContextAware和BeanNameAware，Spring提供了很多Aware回调接口，让bean向容器表明它们需要依赖某种基础结构。通常来说，名字表明了依赖的类型，下表是非常重要的Aware接口列表：

名字	注入依赖
ApplicationContextAware	声明ApplicationContext
ApplicationEventPublisherAware	封闭ApplicationContext的事件发布者。
BeanClassLoaderAware	装载bean classes的类加载器
BeanFactoryAware	声明BeanFactory
BeanNameAware	声明bean的名字
BootstrapContextAware	容器运行的BootstrapContext资源适配器，通常只用在 JCA-aware 的ApplicationContext实例中
LoadTimeWeaverAware	在加载时为处理类定义而定义的weaver。
MessageSourceAware	处理消息所配置的策略（支持参数化和国际化）
NotificationPublisherAware	Spring JMX 通知publisher
ResourceLoaderAware	为资源的low-level访问配置加载器
ServletConfigAware	当前容器运行的ServletConfig，只在web-aware 的Spring ApplicationContext有效
ServletContextAware	当前容器运行的ServletContext，只在web-aware 的Spring ApplicationContext有效

再次强调，使用这些接口将会把你的代码和Spring API相耦合，违背了控制反转原则。因此我们只推荐需要访问容器的基础架构bean中使用它们。

更多教程请访问:www.flydean.com

1.7 Bean的继承

bean定义可以包含很多配置信息，包括构造函数参数、属性值和容器特定的信息，例如初始化方法、静态工厂方法名等。子bean定义从父定义继承配置数据。子定义可以覆盖一些值，或者根据需要添加其他值。使用父bean和子bean定义可以节省大量的输入。实际上，这是一种templating。

如果以编程方式使用ApplicationContext接口，则子bean定义由ChildBeanDefinition类表示。大多数用户并不需要直接使用这个类。相反，他们在类（如ClassPathXmlApplicationContext）中声明性地配置bean定义。使用基于XML的配置元数据时，可以通过使用parent属性来表名这是一个子bean定义，并将父bean指定为此属性的值。以下示例显示了如何执行此操作：

```
<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>
```

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

子bean定义从父bean继承作用域、构造函数参数值、属性值和方法重写，并可以选择添加新值。指定的任何范围、初始化方法、销毁方法或静态工厂方法设置都将覆盖相应的父设置。

其余的设置总是取自子定义：依赖、自动装载模式、依赖项检查、singleton和lazy init。

前面的示例使用abstract属性显式地将父bean定义标记为abstract。如果父定义没有指定类，则需要显式地将父bean定义标记为抽象，如下示例所示：

```
<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不能单独实例化，因为它是不完整的，并且还显式地标记为抽象的。当定义是抽象的时，它只能用作纯模板bean定义，该定义用作子定义的父定义。试图单独使用这样一个抽象父bean，将其作为另一个bean的引用属性来引用，或者使用父bean id执行显式getbean（）调用，都会返回一个错误。类似地，容器的内部preInstantiateSingletons（）方法忽略定义为抽象的bean定义。

> 默认情况下，ApplicationContext预实例化所有单例。因此，重要的是（至少对于singleton bean而言），如果你有一个（父）bean定义，而该定义指定了一个类，那么你必须确保将abstract属性设置为true，否则应用程序上下文将实际（尝试）预实例化abstract bean。

更多教程请访问:www.flydean.com

更多教程请访问:www.flydean.com

1.8 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，可以通过插入特殊集成接口的实现来扩展Spring IOC容器。接下来的几节将描述这些集成接口。

1.8.1 使用BeanPostProcessor自定义Beans

BeanPostProcessor接口定义了回调方法，你可以实现这些方法来提供自己的（或覆盖容器的默认）实例化逻辑、依赖解析逻辑等。如果希望在Spring容器完成对bean的实例化、配置和初始化之后实现一些自定义逻辑，可以插入一个或多个自定义BeanPostProcessor实现。

你可以配置多个BeanPostProcessor实例，并且可以通过设置Order属性来控制这些BeanPostProcessor实例的执行顺序。只有在BeanPostProcessor实现Ordered接口时，才能设置此属性。如果你编写自己的BeanPostProcessor，那么也应该考虑实现Ordered的接口。有关进一步的详细信息，请参阅BeanPostProcessor和有序接口的JavaDoc。另请参见BeanPostProcessor实例的编程注册说明。

> BeanPostProcessor实例对bean（或对象）实例进行操作。也就是说，Spring IOC容器实例化一个bean实例，然后beanPostProcessor实例再继续完成它们的工作。
>
> BeanPostProcessor实例的作用域是每个容器。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了beanPostProcessor，那么它只post-processes该容器中的bean。换句话说，在一个容器中定义的bean不会由在另一个容器中定义的beanPostProcessor进行后处理，即使这两个容器是同一层次结构的一部分。
>
> 要更改实际的bean定义（即定义bean的蓝图），你需要使用beanFactoryPostProcessor。

org.springframework.beans.factory.config.BeanPostProcessor接口正好由两个回调方法组成。当此类注册为容器的post-processor时，对于容器创建的每个bean实例，post-processor在调用容器初始化方法（如InitializingBean.afterPropertiesSet（）或任何声明的init方法）之前，以及在任何bean初始化之后，都会从容器中调用回调。post-processor可以对bean实例执行任何操作，包括完全忽略回调。bean post-processor通常检查回调接口，或者它可以用代理包装bean。为了提供代理包装逻辑，一些Spring AOP基础结构类被实现为bean post-processors。

ApplicationContext自动检测在实现BeanPostProcessor接口的配置元数据中定义的任何bean。ApplicationContext将这些bean注册为后处理器，以便在bean创建之后可以调用它们。bean post-processors 可以与任何其他bean相同的方式部署在容器中。

注意，当通过在configuration类上使用@Bean factory方法声明BeanPostProcessor时，factory方法的返回类型应该是实现类本身，或者至少是org.springframework.beans.factory.config.BeanPostProcessor接口，并清楚地表面该bean是post-processor。否则，ApplicationContext无法在完全创建它之前按类型自动检测到它。由于需要尽早实例化BeanPostProcessor以应用于上下文中其他bean的初始化，因此这种早期类型检测非常关键。

> 以编程方式注册BeanPostProcessor实例
>
> 虽然推荐的BeanPostProcessor注册方法是通过ApplicationContext自动检测（如前所述），但你可以使用addBeanPostProcessor方法在ConfigurableBeanFactory中以编程方式注册它们。 当你需要在注册之前评估条件逻辑，或者甚至需要跨层次结构中的上下文复制bean post processors时，这可能非常有用。但是，请注意，以编程方式添加的BeanPostProcessor实例并不尊重有序的接口。在这里，登记的顺序决定执行的顺序。还要注意，以编程方式注册的BeanPostProcessor实例总是在注册为自动检测的实例之前进行处理，而不接收任何显式排序。
>
> BeanPostProcessor实例和AOP自动代理
>
> 实现BeanPostProcessor接口的类是特殊的，由容器进行不同的处理。作为ApplicationContext特殊启动阶段的一部分，所有BeanPostProcessor实例和这些实例直接引用的bean都在启动时实例化。

更多教程请访问:www.flydean.com

> 接下来，所有beanPostProcessor实例都以排序的方式注册，并应用于容器中所有其他bean。因为AOP自动代理是作为BeanPostProcessor本身实现的，所以BeanPostProcessor实例和它们直接引用的bean都不符合自动代理的条件，因此它们中没有aspects woven。
>
> 对于任何这样的bean，你都应该看到一条日志消息：bean somebean不适合被所有的BeanPostProcessor接口处理（例如：不适合自动代理）。
>
> 如果你使用autowiring或@Resource将bean连接到BeanPostProcessor，那么在搜索类型匹配的依赖项候选项时，Spring可能会访问意外的bean，因此，它们不适合自动代理或选用其他类型的post-processing方式。例如，如果你有一个用@Resource注解的依赖项，其中字段或setter名称与bean的声明名称不直接匹配，并且没有使用name属性，那么spring将通过类型来访问其他bean。

下面的例子展示了怎么在ApplicationContext中写，注册，和使用BeanPostProcessor。

例子Hello World, BeanPostProcessor-style

第一个例子是基本应用，这个例子展示了自定义的BeanPostProcessor实现，调用toString()方法并打印到系统输出。

下面是自定义BeanPostProcessor：

```
package scripting;

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

public class InstantiationTracingBeanPostProcessor implements BeanPostProcessor {

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

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

下面的bean使用了InstantiationTracingBeanPostProcessor：

```
<?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
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/lang
        https://www.springframework.org/schema/lang/spring-lang.xsd">

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

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

</beans>
```

更多教程请访问:www.flydean.com

注意InstantiationTracingBeanPostProcessor是如何定义的。它甚至没有名称，而且，因为它是一个bean，所以可以像注入任何其他bean一样注入依赖关系。（前面的配置还定义了一个由groovy脚本支持的bean。Spring动态语言支持在标题为“动态语言支持”的章节中详细介绍。）

下面是调用的java应用程序：

```
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);
    }

}
```

程序的输出如下：

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

例子：RequiredAnnotationBeanPostProcessor

将回调接口或注解与自定义BeanPostProcessor实现结合使用是扩展Spring IOC容器的常用方法。一个例子是Spring的RequiredAnnotationBeanPostProcessor-一个随Spring发行版一起提供的BeanPostProcessor实现，它要求用（任意）注解标记的bean上的JavaBean属性都需要一个真实值。

1.8.2 使用BeanFactoryPostProcessor自定义配置元数据

我们看到的下一个扩展点是org.springframework.beans.factory.config.BeanFactoryPostProcessor。此接口的语义与BeanPostProcessor的语义相似，但有一个主要区别：BeanFactoryPostProcessor对Bean配置元数据进行操作。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BeanFactoryPostProcessor实例以外的任何bean之前对其进行更改。

你可以配置多个BeanFactoryPostProcessor实例，并且可以通过设置Order属性来控制这些BeanFactoryPostProcessor实例的运行顺序。但是，只能在BeanFactoryPostProcessor实现Ordered接口时设置此属性。如果你编写自己的BeanFactoryPostProcessor，那么也应该考虑实现Ordered的接口。有关更多详细信息，请参阅BeanFactoryPostProcessor和Ordered接口的JavaDoc。

如果要更改实际的bean实例（即从配置元数据创建的对象），则需要使用BeanPostProcessor（如前面使用BeanPostProcessor自定义bean中所述）。虽然在技术上可以在BeanFactoryPostProcessor中使用bean实例（例如，通过使用BeanFactory.getBean（）），但这样做会导致过早的bean实例化，违反标准容器生命周期。这可能会导致负面影响，例如绕过bean post处理。

此外，BeanFactoryPostProcessor实例的作用域是每个容器。这仅在使用容器层次结构时才相关。如果在一个容器中定义BeanFactoryPostProcessor，那么它只应用于该容器中的bean定义。一个容器中的bean定义不会由另一个容器中的BeanFactoryPostProcessor实例进行后处理，即使这两个容器属于同一层次结构。

bean工厂后处理器在ApplicationContext中声明时自动执行，以便将更改应用于定义容器的配置元数据。Spring包含许多预定义的bean工厂后处理器，例如PropertyOverrideConfigurer和PropertyPlaceholderConfigurer。你还可以使用自定义BeanFactoryPostProcessor-来注册自定义属性编辑器。

更多教程请访问:www.flydean.com

ApplicationContext会自动检测部署到其中实现BeanFactoryPostProcessor接口的任何bean。它在适当的时候将这些bean用作bean工厂post-processors。你可以像部署任何其他bean一样部署这些后处理器bean。

> 与BeanPostProcessors一样，你通常不希望为延迟初始化配置BeanFactoryPostProcessors。如果没有其他bean引用bean（工厂）后处理器，则该后处理器将不会被实例化。因此，将其标记为lazy初始化将被忽略，并且即使在你的元素的声明中将default-lazy-init属性设置为true，Bean(Factory)PostProcessor也将被优先实例化。

PropertyPlaceholderConfigurer类名替换

通过使用标准Java Properties格式，可以使用PropertyPlaceholderConfigurer从一个单独的文件中的bean定义外部化属性值。这样做可以使部署应用程序的人员自定义特定于环境的属性，如数据库URL和密码，而不必为容器修改一个或多个XML定义主文件从而增加复杂性或风险。

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数据源：

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

这个例子展示了属性被配置在外部的Properties文件中。在运行时，使用PropertyPlaceholderConfigurer将元数据替换成DataSource中的某些属性。要替换的值被指定为${property-name}格式的占位符，该格式遵循ant和log4j以及JSP EL样式。

真实的值取自外部的Java Properties格式的文件：

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

因此，${jdbc.username} 在运行时被替换成'sa'，其他的占位符也会被替换成相应的值。PropertyPlaceholderConfigurer检查bean定义的大多数属性和属性中的占位符。此外，还可以自定义占位符前缀和后缀。

使用Spring2.5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。可以在“位置”属性中以逗号分隔的列表形式提供一个或多个位置，如下示例所示：

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

PropertyPlaceholderConfigurer不仅在指定的属性文件中查找属性。默认情况下，如果它在指定的属性文件中找不到属性，它也会检查Java系统属性。可以通过使用以下三个支持的整数值之一设置配置器的systemPropertiesMode属性来自定义此行为：

- never (0): 从不检查系统属性。
- fallback (1): 如果属性在声明的文件里面找不到，则去检查系统属性。这个是默认的模式。
- override (2): 先检查系统属性，这会让系统属性覆盖其他的来源。

> 你可以使用PropertyPlaceholderConfigurer来替换类的名称，这个在运行时需要类的特殊实现时很有用，如下例子所示：

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

更多教程请访问:www.flydean.com

```
    <property name="locations">
        <value>classpath:com/something/strategy.properties</value>
    </property>
    <property name="properties">
        <value>custom.strategy.class=com.something.DefaultStrategy</value>
    </property>
</bean>

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

> 如果在运行时无法将类解析为有效的类，则在即将创建bean时，bean的解析将失败，这是在非lazy init bean的ApplicationContext的preInstantiateSingletons（）阶段进行的。

例子PropertyOverrideConfigurer

PropertyOverrideConfigurer是另外一个post-processor bean工程，和PropertyPlaceholderConfigurer相似，但是和它不同的是，对bean的定义来说原始定义可以有默认值或者没有任何值。 如果overriding Properties没有bean某个特定属性，那么会使用默认的上下文定义。

注意，bean定义不知道被覆盖，所以从XML定义文件中看，并不知道使用的是 override configurer。如果多个PropertyOverrideConfigurer实例为同一个bean属性定义不同的值，则最后一个实例将因覆盖机制而获胜。

属性文件配置行如下格式：

```
beanName.property=value
```

下面是例子：

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

此示例文件可以与包含名为dataSource的bean的容器定义一起使用，该bean具有driver和url属性。

也支持复合属性名，只要路径的每个组件（被重写的最终属性除外）已经非空（可能由构造函数初始化）。在下面的示例中，tom bean的fred属性的bob属性的sammy属性设置为标量值123：

```
tom.fred.bob.sammy=123
```

指定的重写值始终是文本值。它们不会转换为bean引用。当XML bean定义中的原始值指定bean引用时，此约定也适用。

使用Spring2.5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。可以在“位置”属性中以逗号分隔的列表形式提供一个或多个位置，如下示例所示：

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

1.8.3 使用FactoryBean自定义实例化逻辑

你可以为本身是工厂的对象实现org.springframework.beans.factory.factorybean接口。

FactoryBean接口是Spring IOC容器实例化逻辑的一个可插入点。如果需要在java中实现初始化代码，而不是一个（潜在的）冗长的XML中，那么你可以创建自己的FactoryBean，在该类中编写复杂的初始化，然后将自定义的FactoryBean插入容器中。

FactoryBean接口提供3个方法：

更多教程请访问:www.flydean.com

- Object getObject(): 返回工厂创建的实例，该实例可能是被共享的， 取决于该实例是单例还是多例模式。
- boolean isSingleton():判断FactoryBean返回的是单例还是多例。
- Class getObjectType():返回getObject() 方法返回的类型，如果提前不知道类型，那么返回null。

factrybean概念和接口在Spring框架中的许多地方使用。超过50个FactoryBean接口实现与Spring本身一起提供。

当需要向容器请求实际的FactoryBean实例本身而不是它生成的bean时，在调用ApplicationContext的getbean（）方法时，在bean的ID前面加上符号（&）。因此，对于ID为myBean的给定FactoryBean，在容器上调用getBean（“myBean”）返回FactoryBean生成的bean，而调用getBean（“&myBean”）则返回FactoryBean实例本身。

更多教程请访问:www.flydean.com

1.9 基于注解的容器配置

> 注解是不是比XML配置更好?
>
> 引入基于注解的配置提出了这样一个问题：这种方法是否比XML“更好”。简短的答案是“它取决于”。长的答案是：每种方法都有其优缺点，通常情况下，由开发人员决定哪种策略更适合它们。由于它们的定义方式，注解在其声明中提供了大量上下文，从而导致配置更简短。然而，XML擅长在不接触源代码或重新编译组件的情况下注入组件。一些开发人员更喜欢将注入靠近源代码，而其他人则认为带注解的类不再是POJO，而且配置变得分散且难以控制。
>
> 无论选择哪一种，Spring都能容纳这两种风格，甚至可以将它们混合在一起。值得指出的是，通过它的JavaConfig选项，Spring允许以非侵入性的方式使用注解，而不接触目标组件源代码，并且在工具方面，所有配置样式都由Spring工具套件支持。

基于注解的配置提供了XML设置的替代方案，它依赖字节码元数据来注入组件，而不是使用尖括号声明。开发人员不使用XML来描述bean连接，而是使用相关类、方法或字段声明上的注解将配置移动到组件类本身。如示例中所述：RequiredAnnotationBeanPostProcessor与注解结合使用BeanPostProcessor是扩展SpringIOC容器的常用方法。

例如，Spring2.0引入了@Required注解，标记属性必须存在。Spring2.5使得遵循相同的方法来驱动Spring的依赖注入成为可能。本质上，@Autowired注解提供了与Autowiring Collaborators中描述的相同的功能，但是具有更细粒度的控制和更广泛的适用性。Spring2.5还增加了对JSR-250注解的支持，如@PostConstruct和@PreDstroy。Spring 3为javax.inject包中包含的JSR-330（Java依赖注入）注解添加了支持，例如@Inject和@Named。有关这些注解的详细信息，请参阅相关部分。

> 注解注入在XML注入之前执行。因此，当两个同时使用时，XML配置会覆盖注解注入的属性。

同样的，你可以将它们注册为单个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隐式注册它们（注意上下文命名空间的包含）：

```
<?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
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <context:annotation-config/>

</beans>
```

（隐式注册的post-processors包括AutoWiredAnnotationBeanPostProcessor、CommonAnnotationBeanPostProcessor、PersistenceAnnotationBeanPostProcessor和上述RequiredAnnotationBeanPostProcessor。）

<context:annotation-config/> 仅在定义bean的同一应用程序上下文中查找bean上的注解。这意味着，如果在DispatcherServlet的WebApplicationContext中放置<context:annotation-config/>，它只检查控制器中的@Autowired bean，而不检查服务。有关详细信息，请参阅DispatcherServlet。

1.9.1 @Required

@Required注解适用Bean的setter方法。如下：

更多教程请访问:www.flydean.com

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

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

    // ...
}
```

此注解指示受影响的bean属性必须在配置时通过bean定义中的显式属性值或通过自动装载填充。如果未填充受影响的bean属性，则容器将引发异常。这就允许了早期和显式的失败，避免了后面的NullPointerException实例等。我们仍然建议你将断言放入bean类本身（例如，放到init方法中）。这样做会强制那些必需的引用和值，即使在容器外部使用类也是如此。

> 从Spring Framework 5.1开始，@Required注解正式被弃用，取而代之的是将构造函数注入用于所需的设置（或使用InitializingBean.afterPropertiesSet（）的自定义实现以及bean属性setter方法）。

1.9.2 使用 @Autowired

> 在本节包含的示例中，JSR 330的@Inject注解可以代替spring的@Autowired注解。有关详细信息，请参阅此处。

你可以将@Autowired注解到构造器中，如下所示：

```
public class MovieRecommender {

    private final CustomerPreferenceDao customerPreferenceDao;

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

    // ...
}
```

> 从SpringFramework4.3开始，如果目标bean只定义了一个构造函数，那么就不再需要在此类构造函数上使用@Autowired注解。但是，如果有多个构造函数可用，则必须至少对其中一个进行注解，以告诉容器使用哪一个。

@Autowired也可以注解到传统的setter方法，如下例子所示：

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

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

    // ...
}
```

也可以把注解应用到任何名字和多个参数，如下所示：

```
public class MovieRecommender {
```

更多教程请访问:www.flydean.com

```
    private MovieCatalog movieCatalog;

    private CustomerPreferenceDao customerPreferenceDao;

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

    // ...
}
```

@Autowired也可以用在字段上，和构造函数混合使用，如下所示：

```
public class MovieRecommender {

    private final CustomerPreferenceDao customerPreferenceDao;

    @Autowired
    private MovieCatalog movieCatalog;

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

    // ...
}
```

> 确保目标组件（例如movieCatalog或customerPreferenceDAO）始终通过类型来声明，这样才能用在@Autowired注入点。否则，由于在运行时找不到类型匹配，注入可能会失败。
>
> 对于通过类路径扫描找到的XML定义的bean或组件类，容器通常预先知道具体的类型。但是，对于@Bean工厂方法，你需要确保声明的返回类型具有足够的表现力。对于实现多个接口的组件或可能由其实现类型引用的组件，考虑在工厂方法上声明最特定的返回类型（至少与引用bean的注入点所要求的特定类型相同）。

还可以通过将注解添加到需要该类型数组的字段或方法，那么可以从ApplicationContext中获取到该特定类型的所有bean，如下例所示：

```
public class MovieRecommender {

    @Autowired
    private MovieCatalog[] movieCatalogs;

    // ...
}
```

这同样适用于类型化集合，如下示例所示：

```
public class MovieRecommender {

    private Set<MovieCatalog> movieCatalogs;

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

    // ...
}
```

更多教程请访问:www.flydean.com

> 你的目标bean可以实现org.springframework.core.Ordered接口，或者如果你希望数组或列表中的项按特定顺序排序，可以使用@Order或标准的@Priority注解。
>
> 否则，它们的顺序遵循容器中相应目标bean定义的注册顺序。
>
> 你可以在目标类级别和@Bean方法上声明@Order注解，可能是通过单个bean定义（在使用同一bean类的多个定义的情况下）声明的。@Order可能会影响注入点的优先级，但请注意，它们不会影响单例启动顺序，这是一个由依赖关系和@DependsOn声明确定的正交问题。
>
> 注意，标准javax.annotation.Priority注解在@Bean级别不可用，因为它不能在方法上声明。它的语义可以通过@Order values和@Primary在每种类型的单个bean上进行建模。

Map实例也可以被注入，只要key是String类型。Map value包括了所有的类型匹配的Bean，keys是该bean的名字。入下图所示：

```
public class MovieRecommender {

    private Map<String, MovieCatalog> movieCatalogs;

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

    // ...
}
```

如果注入点没有可以匹配的目标，那么自动注入会失败。如果是array, collection 或者 map，至少要有一个元素能匹配。

默认行为是将带注解的方法和字段视为指示所需依赖项。你可以如以下示例中所示更改此行为，使框架能够跳过不可满足的注入点，方法是将其标记为非必需：

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

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

    // ...
}
```

如果非必需方法的依赖项（或多个参数的依赖项之一）不可用，则根本不会调用该方法。在这种情况下，不需要填写的字段将不会被填充，从而保留其默认值。

注入的构造函数和工厂方法参数是一种特殊情况，因为@Autowired上的“required”标志的含义有所不同，因为Spring的构造函数解析算法可能处理多个构造函数。默认情况下，构造函数和工厂方法参数实际上是必需的，但在单个构造函数场景中有一些特殊规则，例如，如果没有匹配的bean可用，多元素注入点（数组、集合、映射）解析为空实例。这允许使用一个通用的实现模式，其中所有依赖项都可以在唯一的多参数构造函数中声明，例如声明为没有@Autowired注解的单个公共构造函数。

> 每个类只能标记一个带required注解的构造函数，但可以标记多个非必需的构造函数。在这种情况下，每一个都是候选对象，Spring将使用最贪婪的可以满足最多依赖关系的构造函数，也就是说，拥有最多参数的构造函数。构造函数解析算法与使用重载构造函数的未注解类相同，只是将候选对象缩小到带注解的构造函数。

更多教程请访问:www.flydean.com

> 建议使用@Autowired的'required'属性而不是使用setter方法上的@Required注解。“required”属性表示自动装载不需要该属性。如果无法自动装载，则忽略该属性。另一方面，@Required更强大，因为它强制通过容器支持的任何方式来设置属性。如果未定义任何值，则会引发相应的异常。

或者，你可以通过Java 8的java.util.Optional表示特定依赖项的非必需性质，如下示例显示：

```
public class SimpleMovieLister {

    @Autowired
    public void setMovieFinder(Optional<MovieFinder> movieFinder) {
        ...
    }
}
```

在Spring Framework 5.0中，你也可以使用@Nullable注解（任何包-中的任何类型，例如，JSR-305中的javax.annotation.Nullable）：

```
public class SimpleMovieLister {

    @Autowired
    public void setMovieFinder(@Nullable MovieFinder movieFinder) {
        ...
    }
}
```

你还可以使用@Autowired处理已知可解析依赖项的接口：BeanFactory、ApplicationContext、Environment、ResourceLoader、ApplicationEventPublisher和MessageSource。这些接口及其扩展接口（如ConfigurableApplicationContext或ResourcePatternResolver）将自动解析，无需特殊设置。以下示例自动装载ApplicationContext对象：

```
public class MovieRecommender {

    @Autowired
    private ApplicationContext context;

    public MovieRecommender() {
    }

    // ...
}
```

@Autowired, @Inject, @Value, 和 @Resource 注解是在Spring的BeanPostProcessor中处理的，这意味着你不能将这些注解用在你自己的BeanPostProcessor，BeanFactoryPostProcessor类型。这些类型必须使用XML或者@Bean方法来显示指定。

1.9.3 使用@primary进行基于微调注解的自动装载

由于按类型自动装载可能会导致多个候选者，因此通常需要对选择过程进行更多的控制。实现这一点的一种方法是使用Spring的@Primary注解。@Primary表示当多个bean是要自动连接到单值依赖项的候选对象时，应该优先考虑特定bean。如果候选对象中只存在一个主bean，则它将成为自动装载的值。 考虑将firstMovieCatalog定义为Primary MovieCatalog的以下配置：

```
@Configuration
public class MovieConfiguration {

    @Bean
    @Primary
```

更多教程请访问:www.flydean.com

```
    public MovieCatalog firstMovieCatalog() { ... }

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

    // ...
}
```

下面的MovieRecommender自动注入了firstMovieCatalog：

```
public class MovieRecommender {

    @Autowired
    private MovieCatalog movieCatalog;

    // ...
}
```

相应的Bean定义如下：

```
<?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
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://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>
```

1.9.4 带有Qualifier的基于微调注解的自动装载

当可以确定一个主要候选对象时，@Primary是一种在多个实例中按类型使用自动装载的有效方法。当你需要对选择过程进行更多控制时，可以使用Spring的@Qualifier注解。你可以将限定符值与特定参数相关联，缩小类型匹配集的范围，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，如下面的示例所示：

```
public class MovieRecommender {

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

    // ...
}
```

你还可以在单个构造函数参数或方法参数上指定@Qualifier注解，如下例所示：

更多教程请访问:www.flydean.com

```
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;
    }

    // ...
}
```

下面是相应的bean定义：

```
<?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
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <context:annotation-config/>

    <bean class="example.SimpleMovieCatalog">
        <qualifier value="main"/>

        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean class="example.SimpleMovieCatalog">
        <qualifier value="action"/>

        <!-- inject any dependencies required by this bean -->
    </bean>

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

</beans>
```

对于回退匹配，bean名称被视为默认限定符值。因此，可以使用id为main而不是嵌套限定符元素来定义bean，从而得到相同的匹配结果。

然而，尽管你可以使用此约定按名称引用特定的bean，@Autowired是关于带有可选语义限定符的类型驱动注入的根本。这意味着，即使使用bean名称回退，限定符值在类型匹配集内也始终具有收缩语义。它们不会在语义上表示对唯一bean id的引用。好的限定符值是main、EMEA或persistent，表示独立于bean id的特定组件的特征，在匿名bean定义（如前面示例中的定义）的情况下，可以自动生成这些特征。

限定符也适用于类型化集合，如前面讨论的-例如，设置。在这种情况下，根据声明的限定符，所有匹配的bean都作为集合注入。这意味着限定符不必是唯一的。相反，它们构成了过滤标准。例如，可以定义多个具有相同限定符值“action”的MovieCatalog bean，所有这些限定符值都被注入到一个用@qualifier（“action”）注解的集合中。

> 让限定符值根据目标bean名称在类型匹配候选中进行选择，不需要在注入点处使用@Qualifier注解。如果没有其他分辨标记（例如限定符或主标记），对于非唯一依赖关系情况，Spring将针对目标bean名称匹配注入点名称（即字段名称或参数名称），来选择相同的命名候选对象（如果有）。

更多教程请访问:www.flydean.com

> 也就是说，如果你打算用名称表示注解驱动的注入，那么不要主要使用@Autowired，即使它能够在类型匹配的候选对象中通过bean名称进行选择。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的目标组件，声明的类型与匹配过程无关。@Autowired有相当不同的语义：在按类型选择候选bean之后，指定的字符串限定符值只在那些类型选择的候选对象中考虑（例如，将account限定符与标记有相同限定符标签的bean匹配）。
>
> 对于本身定义为集合、映射或数组类型的bean，@Resource是一个很好的解决方案，它通过唯一的名称引用特定的集合或数组bean。也就是说，从4.3版的collection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，就可以通过spring的@Autowired类型匹配算法来匹配map和数组类型。在这种情况下，可以使用限定符值在相同类型的集合中进行选择，如前一段所述。
>
> 从4.3开始，@Autowired还考虑了注入的自引用（即，返回当前注入的bean的引用）。请注意，自注入是一种回退。对其他组件的常规依赖始终具有优先权。从这个意义上说，自我推荐人不参与定期的候选人选择，相反，它们总是以最低优先级结束。在实践中，你应该只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，你可以使用@Resource，它可以通过其唯一的名称将代理返回到当前bean。
>
> @Autowired应用于字段、构造函数和多参数方法，允许在参数级别缩小限定符注解的范围。相反，只有具有单个参数的字段和bean属性setter方法才支持@Resource。因此，如果注入目标是一个构造函数或多参数方法，那么应该坚持使用限定符。

你可以创建自己的自定义限定符注解。为此，请定义一个注解并在定义中提供@Qualifier注解，如下示例所示：

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

    String value();
}
```

然后，可以在自动装载字段和参数上提供自定义限定符，如下示例所示：

```
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，你可以为候选bean定义提供信息。可以添加<qualifier/>标记作为<bean/>标记的子元素，然后指定type和value以匹配自定义限定符注解。类型与注解的完全限定类名匹配。或者，为了方便起见，如果不存在名称冲突的风险，可以使用短类名。下面的示例演示了这两种方法：

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

更多教程请访问:www.flydean.com

```
        https://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>
```

在类路径扫描和托管组件中，可以看到基于注解的替代方案，以XML形式提供限定符元数据。具体来说，请参见为限定符元数据提供注解。

在某些情况下，使用不带值的注解可能就足够了。当注解具有更一般的用途并且可以应用于多个不同类型的依赖项时，这一点非常有用。例如，你可以提供一个脱机目录，当没有可用的Internet连接时可以搜索该目录。首先，定义简单注解，如下示例所示：

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

}
```

然后将注解添加到要自动装载的字段或属性中，如下面的示例所示：

```
public class MovieRecommender {

    @Autowired
    @Offline
    private MovieCatalog offlineCatalog;

    // ...
}
```

现在bean定义只需要一个qualifier type就够了：

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

你还可以定义自定义限定符注解，这些注解接受除简单值属性之外的命名属性，或者不接受简单值属性。如果在要自动注入的字段或参数上指定多个属性值，则bean定义必须匹配所有此类属性值，才能被视为自动连线注入。例如，考虑以下注解定义：

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

    String genre();
```

更多教程请访问:www.flydean.com

```
    Format format();
}
```

Format是一个枚举，如下：

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

要自动注入的字段将使用自定义限定符进行注解，并包含属性（genre和format）的值，如下例所示：

```
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符值。这个示例还演示了可以使用bean元属性而不是<qualifier/>元素。如果可用，则以<qualifier/>元素及其属性为准，但如果不存在这样的限定符，则自动装载机制将返回到<meta/>标记内提供的值，如下面示例中最后两个bean定义中所示：

```
<?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
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://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">
```

更多教程请访问:www.flydean.com

```
        <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>
```

1.9.5 使用泛型作为自动装载限定符

除了@Qualifier注解外，还可以使用Java泛型类型作为隐式的限定形式。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();
    }
}
```

假设前面的bean实现了一个通用接口（即Store<string>和Store<integer>），你可以@Autowire Store接口，并将泛型用作限定符，如下例所示：

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

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

通用限定符也适用于自动装载列表、映射实例和数组。以下示例自动装载通用列表：

```
// 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;
```

1.9.6 使用CustomAutowireConfigurer

CustomAutoWireConfigurer是一个BeanFactoryPostProcessor，它允许你注册自己的自定义限定符注解类型，即使它们没有用Spring的@Qualifier进行注解。下面的示例演示如何使用CustomAutoWireConfigurer：

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

更多教程请访问:www.flydean.com

```
    </property>
</bean>
```

AutowireCandidateResolver通过下面几种方式来确定自动注入的候选人：

- bean中定义的autowire-candidate
- <beans/>的default-autowire-candidates模式
- @Qualifier注解和任何自定义的注册到CustomAutowireConfigurer的注解。

当多个bean符合autowire候选条件时，“primary”的确定如下：如果候选对象中只有一个bean定义的primary属性设置为true，则选择它。

1.9.7 使用@Resource注入

Spring还支持通过在字段或bean属性setter方法上使用jsr-250 @Resource注解（javax.annotation.Resource）进行注入。这是JavaEE中常见的模式：例如，在JSF托管bean和JAX-WS端点中。Spring也支持Spring管理对象的这种模式。

@Resource具有名称属性。默认情况下，Spring将该值解释为要注入的bean名称。换句话说，它遵循名称语义，如下面的示例所示：

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

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

如果未显式指定名称，则从字段名或setter方法派生默认名称。对于字段，它采用字段名。对于setter方法，它采用bean属性名。下面的示例将把名为moviefinder的bean注入其setter方法中：

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

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

> 注解提供的名称由CommonAnnotationBeanPostProcessor知道的ApplicationContext解析为bean名称。如果显式配置Spring的SimpleJndiBeanFactory，则可以通过JNDI解析名称。但是，我们建议你依赖默认行为，并使用Spring的JNDI查找功能来保持间接寻址的级别。

在@Resource用法中，如果没有指定显式名称，并且类似于@Autowired，@Resource会找到一个主类型匹配，而不是指定的bean，并解析已知的可解析依赖项：BeanFactory、ApplicationContext、ResourceLoader、ApplicationEventPublisher，和MessageSource接口。

因此，在下面的示例中，customerPreferenceDAO字段首先查找名为“customerPreferenceDAO”的bean，然后返回到与customerPreferenceDAO类型匹配的主类型：

```
public class MovieRecommender {

    @Resource
    private CustomerPreferenceDao customerPreferenceDao;
```

更多教程请访问:www.flydean.com

```
    @Resource
    private ApplicationContext context;

    public MovieRecommender() {
    }

    // ...
}
```

1.9.8 使用@PostConstruct和@PreDestroy

CommonAnnotationBeanPostProcessor不仅识别@Resource注解，还识别JSR-250生命周期注解：javax.annotation.PostConstruct和javax.annotation.PreDestroy。在Spring2.5中引入了对这些注解的支持，它提供了生命周期回调机制的替代方案，如初始化回调和销毁回调中所述。如果CommonAnnotationBeanPostProcessor注册在Spring ApplicationContext中，则在生命周期中与相应的Spring Lifecycle Interface方法或显式声明的回调方法相同的点调用包含这些注解之一的方法。在以下示例中，缓存在初始化时预填充，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效果的详细信息，请参见组合生命周期机制。

> 与@Resource一样，@PostConstruct和@PreDestroy注解类型是JDK 6到8标准Java库的一部分。然而，整个javax.annotation包与JDK 9中的核心Java模块分离，并最终在JDK 11中被删除。如果需要，javax.annotation-api工件现在需要通过maven central获得，只需像其他库一样添加到应用程序的类路径中即可。

更多教程请访问:www.flydean.com

1.10 类路径扫描和托管组件

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（基于注解的容器配置）演示了如何通过源代码级注解提供大量的配置元数据。然而，即使在这些示例中，XML文件中也显式定义了“基本”bean定义，而注解只驱动依赖注入。

本节介绍通过扫描类路径隐式检测候选组件的选项。候选组件是与筛选条件相匹配的类，并在容器中注册了相应的bean定义。这样就不需要使用XML来执行bean注册。相反，你可以使用注解（例如@Component）、AspectJ类型表达式或你自己的自定义筛选条件来选择哪些类具有向容器注册的bean定义。

> 从Spring3.0开始，Spring JavaConfig项目提供的许多特性都是核心Spring框架的一部分。这允许你使用Java定义bean，而不是使用传统的XML文件。查看@Configuration、@Bean、@Import和@DependsOn注解，了解如何使用这些新功能的示例。

1.10.1 @Component和进一步的构造型注解

@Repository注解是一种类的一个标记，这些类实现了存储库的角色或原型（也称为数据访问对象或DAO）。

Spring提供了更多的原型注解：@Component、@Service和@Controller。@Component是任何Spring管理的组件的通用原型。@Repository、@Service和@Controller是@Component针对更具体的用例（分别在持久性、服务和表示层中）的特殊注解。因此，你可以用@Component注解组件类，但是，通过用@Repository、@Service或@Controller注解它们，你的类将更适合通过工具处理或与aspects关联。

例如，这些构造型注解是切入点的理想目标。@Repository、@Service和@Controller也可以在Spring框架的未来版本中附加语义。因此，如果你选择在服务层中使用@Component或@Service，@Service显然是更好的选择。同样，正如前面所述，@Repository已经被支持作为持久性层中自动异常转换的标记。

1.10.2 使用元注解和组合注解

大多数Spring的注解都可以作为元注解用在你自定的注解中。所谓元注解就是可以用在其他注解中的注解。 像之前提到的@Component就是@Service的元注解。如下：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @Interface Service {

    // ....
}
```

> @Component会导致@Service和@Component一样被对待。

你也可以合并使用元注解。比如：在Spring MVC中的@RestController就是由@Controller和@ResponseBody组成的。此外，组合注解还可以选择从元注解中重新声明属性以允许自定义。当你只想公开元注解属性的一个子集时，这尤其有用。例如，Spring的@SessionScope注解将作用域名称硬编码为session，但仍允许自定义proxyMode。以下列表显示了sessionScope注解的定义：

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Scope(WebApplicationContext.SCOPE_SESSION)
public @Interface SessionScope {
```

更多教程请访问:www.flydean.com

```
    /**
     * Alias for {@link Scope#proxyMode}.
     * <p>Defaults to {@link ScopedProxyMode#TARGET_CLASS}.
     */
    @AliasFor(annotation = Scope.class)
    ScopedProxyMode proxyMode() default ScopedProxyMode.TARGET_CLASS;

}
```

你可以不定义@SessionScope里面的proxyMode， 如下：

```
@Service
@SessionScope
public class SessionScopedService {
    // ...
}
```

你也可以重写proxyMode，如下：

```
@Service
@SessionScope(proxyMode = ScopedProxyMode.INTERFACES)
public class SessionScopedUserService implements UserService {
    // ...
}
```

1.10.3 自动检测类并注册bean定义

Spring可以自动检测构造型类，并用ApplicationContext注册相应的BeanDefinition实例。例如，以下两个类可用于此类自动检测：

```
@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，需要将@ComponentScan添加到@Configuration类中，其中basePackages属性是这两个类的通用父包。（或者，可以指定包含每个类的父包的逗号、分号或空格分隔列表。）

```
@Configuration
@ComponentScan(basePackages = "org.example")
public class AppConfig  {
    ...
}
```

为了简洁起见，前面的示例可能使用了注解的value属性（即@ComponentScan（“org.example”））。

更多教程请访问:www.flydean.com

下面是使用XML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
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="org.example"/>

</beans>
```

> <context:component-scan> 隐式的开启了<context:annotation-config>的功能。所以当使用<context:component-scan>的时候就不需要用<context:annotation-config>。
>
> 扫描classpath包需要在classpath中存在相应的目录条目。当你使用Ant构建JAR时，请确保你没有激活JAR任务的仅限文件的开关。此外，在某些环境中，根据安全策略，类路径目录可能不会公开-例如，JDK 1.7.0_45及更高版本上的独立应用程序（这需要在清单中设置“可信库”）请参见 https://stackoverflow.com/questions/19394570/java-jre-7u45-breaks-classloader-getresources）。
>
> 在JDK9的模块路径（Jigsaw）上，Spring的类路径扫描通常按预期工作。但是，请确保组件类暴露在module-info描述符中。如果希望Spring调用类的非公共成员，请确保它们是“打开的”（即，它们使用opens声明而不是module-info描述符中的exports声明）。

此外，在使用component-scan元素时，AutowiredAnnotationBeanPostProcessor和CommonAnnotationBeanPostProcessor都是隐式被使用的。这意味着这两个组件都是自动检测并连接在一起的-都没有XML中提供任何bean配置元数据。

> 通过将annotation-config属性的值设置为false，可以禁用AutoWiredAnnotationBeanPostProcessor和CommonAnnotationBeanPostProcessor的注册。

1.10.4. 使用Filters来定制化扫描

默认情况下，用@Component、@Repository、@Service、@Controller注解的类或用@Component注解的自定义注解是唯一检测到的候选组件。但是，可以通过应用自定义筛选器来修改和扩展此行为。将它们作为@Component scan注解的includefilters或excludefilters参数添加（或作为component scan元素的include filter或exclude filter子元素添加）。每个筛选器元素都需要类型和表达式属性。下表介绍了筛选选项：

Filter type	表达式例子	描述
annotation（默认）	org.example.SomeAnnotation	type基本的目标组件
assignable	org.example.SomeClass	目标组件可分配给（扩展或实现）的类（或接口）。
aspectj	org.example..*Service+	匹配目标组件的AspectJ类型
regex	org.example.Default.*	匹配目标主键内名的正则表达式
custom	org.example.MyTypeFilter	org.springframework.core.type .TypeFilter的自定义实现

下面例子展示了忽略 @Repository 并且使用“stub” repositories 的注解：

```
@Configuration
@ComponentScan(basePackages = "org.example",
```

更多教程请访问:www.flydean.com

```
        includeFilters = @Filter(type = FilterType.REGEX, pattern = ".*Stub.*Repository"),
        excludeFilters = @Filter(Repository.class))
public class AppConfig {
    ...
}
```

下面是同等作用的xml例子：

```
<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>
```

> 你也可以禁用默认的filters，通过设置useDefaultFilters=false在注解中或者设置<component-scan/> 中use-default-filters="false"的属性。 这样就会禁用@Component, @Repository, @Service, @Controller, 和 @Configuration 注解.

1.10.5 在组件内部定义Bean元数据

Spring组件还可以为容器提供bean定义元数据。你可以使用用于在@Configuration annotated类中定义bean元数据的相同@Bean注解来实现这一点。以下示例显示了如何执行此操作：

```
@Component
public class FactoryMethodComponent {

    @Bean
    @Qualifier("public")
    public TestBean publicInstance() {
        return new TestBean("publicInstance");
    }

    public void doWork() {
        // Component method implementation omitted
    }
}
```

前面的类是一个Spring组件，它的doWork（）方法中包含特定于应用程序的代码。但是，它还提供一个bean定义，该bean定义具有引用方法publicInstance（）的工厂方法。@Bean注解标识工厂方法和其他bean定义属性，例如通过@qualifier注解的限定符值。可以指定的其他方法级批注包括@Scope、@Lazy和自定义限定符批注。

除了其用于组件初始化的角色之外，你还可以将@Lazy注解放在标有@Autowired或@Inject的注入点上。在这种情况下，它会导致注入延迟解析代理。

如前所述，支持自动装载的字段和方法，并额外支持@Bean方法的自动装载。以下示例显示了如何执行此操作：

```
@Component
public class FactoryMethodComponent {

    private static int i;

    @Bean
    @Qualifier("public")
    public TestBean publicInstance() {
        return new TestBean("publicInstance");
    }
```

更多教程请访问:www.flydean.com

```
    // 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);
    }
}
```

该示例自动将字符串方法参数country注入到另一个名为privateInstance的bean上的age属性的值。Spring表达式语言元素通过符号#{ <expression> }定义属性的值。对于@Value注解，表达式解析器被预先配置为在解析表达式文本时查找bean名称。

从SpringFramework4.3开始，还可以声明InjectionPoint类型的工厂方法参数（或其更具体的子类：DependencyDescriptor）以访问触发当前bean创建的请求注入点。

注意，这只适用于bean实例的实际创建，而不适用于现有实例的注入。因此，这个特性对于原型范围的bean最有意义。对于其他作用域，factory方法只会看到触发在给定作用域中创建新bean实例的注入点（例如，触发惰性单例bean创建的依赖项）。在这种情况下，你可以使用提供的注入点元数据和语义关注。下面的示例演示如何使用InjectionPoint：

```
@Component
public class FactoryMethodComponent {

    @Bean @Scope("prototype")
    public TestBean prototypeInstance(InjectionPoint injectionPoint) {
        return new TestBean("prototypeInstance for " + injectionPoint.getMember());
    }
}
```

常规Spring组件中的@Bean方法的处理方式与Spring@Configuration类中的对应方法不同。不同的是，@Component类没有用cglib来增强以截获方法和字段的调用。cglib代理是调用@Configuration classes中通过@Bean methods内的方法或字段创建对协作对象的bean元数据引用的方法。

即使通过引用对@Bean方法的编程调用来引用其他bean，这也不是普通的Java语义调用，而是通过容器来提供通常的生命周期管理和Spring bean的代理，相反，在普通的@Component类中调用“bean”方法中具有标准Java语义的方法或字段，没有特殊的CGLIB处理或其他约束应用。

> 你可以将@Bean方法声明为静态方法，允许在不将其包含的配置类创建为实例的情况下调用它们。在定义post-processor bean（例如，BeanFactoryPostProcessor或BeanPostProcessor类型）时，这是特别有意义的，因为这样的bean在容器生命周期的早期就被初始化，应该避免在此时触发配置的其他部分。
>
> 由于技术限制，对static@Bean方法的调用永远不会被容器截获，即使是在@Configuration类（如本节前面所述）中也是如此：cglib子类只能重写非静态方法。因此，直接调用另一个@Bean方法具有标准Java语义，导致从工厂方法本身直接返回一个独立的实例。

更多教程请访问:www.flydean.com

> “@Bean”方法的Java语言可见性不会对Spring容器中的bean定义产生直接影响。你可以自由地声明你认为适合非@Configuration类以及任何静态方法的工厂方法。但是，@Configuration类中的常规@Bean方法必须是可重写的，也就是说，它们不能声明为私有或最终的。
>
> “@Bean”方法也在给定组件或配置类的基类上发现，以及在由组件或配置类实现的接口中声明的Java 8默认方法上。这允许在组成复杂的配置安排方面有很大的灵活性，即使是多个继承也可以通过Java 8默认方法（如Spring 4.2）来实现。
>
> 最后，一个类可以为同一个bean保存多个@Bean方法，作为多个工厂方法的安排，这取决于运行时可用的依赖项。这与在其他配置方案中选择“最贪婪”的构造函数或工厂方法的算法相同：在构建时选择具有最多可满足依赖项的变量，类似于容器如何在多个@Autowired构造函数之间选择。

1.10.6. 为自动检测组件命名

当一个组件作为扫描过程的一部分被自动检测时，它的bean名称由该扫描仪已知的BeanNameGenerator策略生成。默认情况下，任何包含value的Spring原型注解（@Component、@Repository、@Service和@Controller）都会将该名称提供给相应的bean定义。

如果这样的注解不包含value或任何其他检测到的组件（例如那些由自定义过滤器发现的组件），则默认bean名称生成器将返回小写的非限定类名。例如，如果检测到以下组件类，则名称为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，可以提供一个自定义的bean命名策略。首先，实现BeanNameGenerator接口，并确保包含一个默认的无参数构造函数。然后，在配置扫描器时提供完全限定的类名，如下面的示例注解和bean定义所示：

```
@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>
```

作为一般规则，当其他组件可能对其进行显式引用时，请考虑使用注解指定名称。另一方面，只要容器负责注入，自动生成的名称就足够了。

1.10.7 为自动检测的组件提供作用域

与一般的Spring管理组件一样，自动检测组件的默认和最常见的作用域是singleton。但是，有时你需要一个可以由@Scope注解指定的不同范围。可以在注解中提供作用域的名称，如下示例所示：

更多教程请访问:www.flydean.com

```
@Scope("prototype")
@Repository
public class MovieFinderImpl implements MovieFinder {
    // ...
}
```

> @Scope注解只在具体的bean类（对于带注解的组件）或工厂方法（对于@Bean方法）上进行内省。与XML bean定义不同，没有bean定义继承的概念，并且类级别的继承层次结构与元数据无关。

有关特定于Web的范围（如Spring上下文中的“请求”或“会话”）的详细信息，请参阅请求、会话、应用程序和WebSocket范围。与这些作用域的预构建注解一样，你也可以使用Spring的元注解方法编写自己的作用域注解：例如，使用@Scope（“prototype”）注解的自定义注解元，也可能声明自定义作用域代理模式。

> 要为范围解析提供自定义策略，而不是依赖基于注解的方法，可以实现ScopeMetadataResolver接口。请确保包含默认的无参数构造函数。然后，你可以在配置扫描器时提供完全限定的类名，正如下面的注解和bean定义示例所示：

```
@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>
```

当使用某些非单例作用域时，可能需要为作用域对象生成代理。为此，组件扫描元素上可以有一个scoped-proxy 属性。三个可能的值是：no、interfaces和targetClass。例如，以下配置将生成标准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>
```

1.10.8 为限定符元数据提供注解

@Qualifier注解将在基于微调注解的带有限定符的自动注入中讨论。该部分中的示例演示了在解析autowire候选时使用@Qualifier注解和自定义限定符注解来提供细粒度控制。因为这些示例是基于XML bean定义的，所以通过使用XML中bean元素的限定符或元子元素，在候选bean定义上提供了限定符元数据。当依赖类路径扫描自动检测组件时，可以在候选类上为限定符元数据提供类型级注解。以下三个示例演示了此技术：

```
@Component
@Qualifier("Action")
public class ActionMovieCatalog implements MovieCatalog {
    // ...
}
```

```
@Component
```

更多教程请访问:www.flydean.com

```
@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，请记住注解元数据绑定到类定义本身，而XML的使用允许同一类型的多个bean可以有多个不同的限定符，因为元数据是按实例而不是按class提供的。

1.10.9 生成候选组件的索引

虽然类路径扫描速度非常快，但是可以通过在编译时创建一个静态候选列表来提高大型应用程序的启动性能。在此模式下，作为组件扫描目标的所有模块都必须使用此机制。

> 你现有的@ComponentScan或<context:component-scan>指令必须保持原样，以请求上下文扫描某些包中的候选项。当ApplicationContext检测到这样的索引时，它会自动使用它，而不是扫描类路径。

要生成索引，请向每个模块添加一个附加依赖项，该模块包含作为组件扫描指令目标的组件。下面的示例说明如何使用Maven：

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context-indexer</artifactId>
        <version>5.1.8.RELEASE</version>
        <optional>true</optional>
    </dependency>
</dependencies>
```

在Gradle 4.5或者更早版本，依赖必须配置在compileOnly中，如下：

```
dependencies {
    compileOnly "org.springframework:spring-context-indexer:5.1.8.RELEASE"
}
```

在Gradle 4.6或者以后版本，依赖必须配置在annotationProcessor中，如下：

```
dependencies {
    annotationProcessor "org.springframework:spring-context-indexer:5.1.8.RELEASE"
}
```

这个过程生成一个包含在JAR文件中的META-INF/spring.components文件。

> 在你的IDE中使用此模式时，spring-context-indexer必须注册为注解处理器，以确保在更新候选组件时索引是最新的。
>
> 当在类路径上找到META-INF/Spring.components时，索引将自动启用。如果某个索引部分可用于某些库（或用例），但无法为整个应用程序生成，则可以通过将spring.index.ignore设置为true（作为系统属性或类路径根目录下的spring.properties文件）来回滚到常规类路径安排（就像根本没有索引一样）。

更多教程请访问:www.flydean.com

更多教程请访问:www.flydean.com

1.11 使用JSR 330标准注解

从Spring3.0之后，Spring提供了JSR-330标志注解的支持。这些注解和Spring注解一样被进行扫描，如果想使用它们，需要包含下面的jar包：

> 如果你使用maven，那么javax.inject工件可以在标准maven存储库中使用（https://repo1.maven.org/maven2/javax/inject/javax.inject/1/）。可以将以下依赖项添加到文件pom.xml中：

```
<dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId>
    <version>1</version>
</dependency>
```

1.11.1 使用@Inject 和 @Named 注入

你可以使用 @javax.inject.Inject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，你可以在字段级、方法级和构造函数参数级使用@Inject。此外，你可以将注入点声明为Provider，允许通过Provider.get() 调用按需访问较短作用域的bean或延迟访问其他bean。以下示例提供了前面示例的变体：

```
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(...);
        ...
    }
}
```

更多教程请访问:www.flydean.com

如果要对应注入的依赖项使用限定名，则应使用@Named注解，如下例所示：

```
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;
    }

    // ...
}
```

与@Autowired一样，@Inject也可以与java.util.Optional或@Nullable一起使用。这在这里更适用，因为@Jnject没有required的属性。以下两个示例演示如何使用@Inject和@Nullable：

```
public class SimpleMovieLister {

    @Inject
    public void setMovieFinder(Optional<MovieFinder> movieFinder) {
        ...
    }
}
```

```
public class SimpleMovieLister {

    @Inject
    public void setMovieFinder(@Nullable MovieFinder movieFinder) {
        ...
    }
}
```

1.11.2 @Component的标准等价物@Named 和 @ManagedBean

除了使用@Component，你也可以使用@javax.inject.Named 或者 javax.annotation.ManagedBean，如下：

```
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;
    }

    // ...
}
```

在不指定组件名称的情况下使用@Component是非常常见的。@Named可以以类似的方式使用，如下示例所示：

更多教程请访问:www.flydean.com

```
import javax.inject.Inject;
import javax.inject.Named;

@Named
public class SimpleMovieLister {

    private MovieFinder movieFinder;

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

    // ...
}
```

当你使用@Named或者@ManagedBean，你可以同样使用组件扫描：

```
@Configuration
@ComponentScan(basePackages = "org.example")
public class AppConfig  {
    ...
}
```

> 与@Component不同,JSR-330的@Named和JSR-250的@ManagedBean是不可组合的。你应该使用Spring的构造型模型来构建自定义组件注解。

1.11.3 JSR-330 标准注解的限制

当使用标准注解时，你应该知道一些重要功能不可用，如下表所示：

Spring	javax.inject.*	javax.inject限制/描述
@Autowired	@Inject	@Inject没有required属性，可以使用Java8的Optional代替
@Component	@Named / @ManagedBean	JSR-330没有提供组合模式，只有一种方式来标记命名组件
@Scope("singleton")	@Singleton	JSR-330默认范围像Spring的prototype，但是为了和Spring的默认值保持一致，在Spring中定义的JSR-330 bean默认是singleton。如果要使用其他的作用范围，那么需要使用Spring的@Scope注解。javax.inject也提供了一个@Scope注解。但是这个注解仅用来创建你自己的注解。
@Qualifier	@Qualifier / @Named	javax.inject.Qualifier只是一个用来构建自定义Qualifier的元注解。具体的字符串限定符（如带value的Spring的@Qualifier）可以通过javax.inject.Named关联。
@Value	-	没有相同功能
@Required	-	没有相同功能
@Lazy	-	没有相同功能
ObjectFactory	Provider	javax.inject.Provider是Spring的ObjectFactory的直接替代品，它只使用了较短的get（）方法名。它还可以与Spring的@Autowired结合使用，或者与无注解的构造函数和setter方法结合使用。

更多教程请访问:www.flydean.com

更多教程请访问:www.flydean.com

1.12 基于Java的容器配置

这节包括了如何在Java代码里面使用注解来配置Spring容器。

1.12.1 基本概念：@Bean 和 @Configuration

Spring新的Java配置支持中的中心构件是@Configuration和@Bean。

@Bean注解用于指示方法实例化、配置和初始化要由Spring IOC容器管理的新对象。对于熟悉Spring的<beans/>XML配置的用户，@Bean注解与<bean/>元素具有相同的作用。你可以将@Bean注解的方法与任何spring @Component一起使用。但是，它们最常用于@Configuration bean。

用@Configuration注解类表明它的主要用途是作为bean定义的源。此外，@Configuration classes允许通过调用同一类中的其他@Bean方法来定义bean间的依赖关系。最简单的@Configuration类如下：

```
@Configuration
public class AppConfig {

    @Bean
    public MyService myService() {
        return new MyServiceImpl();
    }
}
```

上面的AppConfig类和下面的XML配置相同：

```
<beans>
    <bean id="myService" class="com.acme.services.MyServiceImpl"/>
</beans>
```

> 完整的@Configuration与“lite”@Bean模式?
>
> 当@Bean方法在没有用@Configuration注解的类中声明时，它们被称为在“lite”模式下处理。在@Component甚至是普通老类中声明的bean方法被认为是“lite”，因为包含类的主要目的不同，@Bean方法在这里可以看做一种奖励。例如，服务组件可以通过在每个适用的组件类上附加的@Bean方法向容器公开管理视图。在这种情况下，@Bean方法是一种通用的工厂方法机制。
>
> 与完整的@Configuration不同，lite@Bean方法不能声明bean之间的依赖关系。相反，它们对其包含组件的内部状态进行操作，也可以选择对可能声明的参数进行操作。因此，这样的@Bean方法不应调用其他@Bean方法。每个这样的方法实际上只是特定bean引用的工厂方法，没有任何特殊的运行时语义。这里的积极副作用是不需要在运行时应用cglib子类化，所以在类设计方面没有限制（也就是说，包含类可能是最终的等等）。
>
> 在常见的场景中，@Bean方法将在@Configuration类中声明，确保始终使用“full”模式，因此跨方法引用将被重定向到容器的生命周期管理。这防止了通过常规Java调用意外调用相同的bean方法，这有助于减少在“Lite”模式下操作时难以跟踪的细微错误。

下面的章节将深入讨论@Bean和@Configuration注解。但是，首先，我们介绍了使用基于Java的配置创建Spring容器的各种方法。

1.12.2 通过AnnotationConfigApplicationContext来实例化Spring容器

更多教程请访问:www.flydean.com

下面是SpringAnnotationConfigApplicationContext的介绍，他是在Spring 3.0被引入的。

他是ApplicationContext的一个实现，他可以接收@Configuration， @Component ，和使用JSR-330注解的类作为输入。

当@Configuration作为输入时，@Configuration类本身被注册成为一个bean，并且他里面包含的所有@Bean 方法，都会被注册成Bean。

当@Component和JSR-330类作为输入时，它们被注册为bean定义，并且假设在必要时在这些类中使用诸如@Autowired或@Inject之类的DI元数据。

简单构造器

和ClassPathXmlApplicationContext需要XML文件作为输入一样，AnnotationConfigApplicationContext需要@Configuration 作为实例化参数，这样可以完全不使用XML文件：

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

前面提到了AnnotationConfigApplicationContext不仅仅可使用@Configuration，也可以使用@Component 或者JSR-330 annotated class 。 如下所示：

```
public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(MyServiceImpl.class, Dependency1.class, Dep
endency2.class);
    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
}
```

上面例子假设MyServiceImpl, Dependency1, 和 Dependency2 使用了Spring的依赖注入注解如： @Autowired.

使用 register(Class<?>...) 来程序的方式构建容器

你可以使用无参数构造函数来实例化AnnotationConfigApplicationContext，然后通过register()拉配置他。这个在程序的方式中比较有用：

```
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...)允许组件扫描

想使用组件扫描，可以通过如下方式：

```
@Configuration
@ComponentScan(basePackages = "com.acme")
public class AppConfig  {
    ...
}
```

> 有经验的Spring用户可能很熟悉下面的XML配置用法：

更多教程请访问:www.flydean.com

```
<beans>
    <context:component-scan base-package="com.acme"/>
</beans>
```

上面的例子中，com.acme 包被扫描任何@Component注解的类，这些类被注入到Spring容器中。AnnotationConfigApplicationContext 提供了scan(String…)方法和component-scan一样的功能。如下：

```
public static void main(String[] args) {
    AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
    ctx.scan("com.acme");
    ctx.refresh();
    MyService myService = ctx.getBean(MyService.class);
}
```

> 记住，@Configuration类是用@Component元注解的，因此它们是组件扫描的候选者。在前面的示例中，假设AppConfig是在com.acme包（或下面的任何包）中声明的，则会在调用scan（）期间取到他。在refresh（）之后，它的所有@Bean方法都被处理并注册为容器中的bean定义。

使用AnnotationConfigWebApplicationContext来支持Web应用程序

AnnotationConfigApplicationContext的一个WebApplicationContext变种是AnnotationConfigWebApplicationContext。当你配置Spring ContextLoaderListener，或者Spring MVC DispatcherServlet时，可以使用他。下面的web.xml配置了一个标准的Spring MVC web应用：

```
web-app>
    <!-- Configure ContextLoaderListener to use AnnotationConfigWebApplicationContext
        instead of the default XmlWebApplicationContext -->
    <context-param>
        <param-name>contextClass</param-name>
        <param-value>
            org.springframework.web.context.support.AnnotationConfigWebApplicationContext
        </param-value>
    </context-param>

    <!-- Configuration locations must consist of one or more comma- or space-delimited
        fully-qualified @Configuration classes. Fully-qualified packages may also be
        specified for component-scanning -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>com.acme.AppConfig</param-value>
    </context-param>

    <!-- Bootstrap the root application context as usual using ContextLoaderListener -->
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

    <!-- Declare a Spring MVC DispatcherServlet as usual -->
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <!-- Configure DispatcherServlet to use AnnotationConfigWebApplicationContext
            instead of the default XmlWebApplicationContext -->
        <init-param>
            <param-name>contextClass</param-name>
            <param-value>
                org.springframework.web.context.support.AnnotationConfigWebApplicationContext
            </param-value>
        </init-param>
        <!-- Again, config locations must consist of one or more comma- or space-delimited
            and fully-qualified @Configuration classes -->
        <init-param>
            <param-name>contextConfigLocation</param-name>
```

更多教程请访问:www.flydean.com

```
            <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>
```

1.12.3 使用@Bean注解

@Bean是方法级的注解，和XML中的<bean/>元素有同样的作用。注解支持一些属性如： *init-method* destroy-method *autowiring*

你可以在 @Configuration或者@Component 注解中使用@Bean。

声明一个Bean

要声明bean，可以使用@Bean注解对方法进行注解。你可以使用此方法在指定为方法返回值的类型的ApplicationContext中注册bean定义。默认情况下，bean名称与方法名称相同。下面的示例显示了@Bean方法声明：

```
@Configuration
public class AppConfig {

    @Bean
    public TransferServiceImpl transferService() {
        return new TransferServiceImpl();
    }
}
```

上面例子和下面相同：

```
<beans>
    <bean id="transferService" class="com.acme.TransferServiceImpl"/>
</beans>
```

两种声明都可以在ApplicationContext定义transferService。并将transferService和一个TransferServiceImpl的实例对象绑定起来：

```
transferService -> com.acme.TransferServiceImpl
```

你也可以将@Bean定义在一个返回接口的方法：

```
@Configuration
public class AppConfig {

    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl();
    }
}
```

但是，这会将高级类型预测的可见性限制为指定的接口类型（TransferService）。然后，在容器只知道一次完整类型（transferserviceimpl）的情况下，受影响的单例bean已经被实例化。非懒惰的单例bean根据其声明顺序进行实例化，因此你可能会看到不同的类型匹配结果，具体取决于另一个组件尝试通过非声明类型进行匹配的时间（例如

更多教程请访问:www.flydean.com

@Autowired TransferServiceImpl，一旦transferService bean被实例化时，它就会被解析）。

> 如果通过已声明的服务接口一致地引用你的类型，则@Bean返回类型可以安全地加入该设计决策。但是，对于实现多个接口的组件或可能由它们的实现类型引用的组件，声明更加具体的返回类型可能会更安全（至少在引用bean的注入点所要求的特定类型）。

Bean定义

@Bean注解的方法可以有多个参数，如下所示，在实例化是TransferService需要一个AccountRepository，我们使用方法参数来实现这个依赖，如下：

```
@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。有关详细信息，请参阅相关部分。

接收生命周期回调

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中的@PostConstruct和@PreDestroy注解。更多详细信息，请参见JSR-250注解。

还完全支持常规的Spring生命周期回调。如果bean实现了InitialingBean、DisposableBean或Lifecycle，那么容器将调用它们各自的方法。

还完全支持*Aware接口的标准集（如BeanFactoryAware、BeannameAware、MessageSourceAware、ApplicationContextAware等）。

@Bean注解支持指定任意的初始化和销毁回调方法，很像spring xml的init方法和bean元素上的destroy方法属性，如下例所示：

```
public class BeanOne {

    public void init() {
        // initialization logic
    }
}

public class BeanTwo {

    public void cleanup() {
        // destruction logic
    }
}

@Configuration
public class AppConfig {

    @Bean(initMethod = "init")
    public BeanOne beanOne() {
        return new BeanOne();
    }

    @Bean(destroyMethod = "cleanup")
    public BeanTwo beanTwo() {
        return new BeanTwo();
    }
}
```

更多教程请访问:www.flydean.com

> 默认情况下，使用具有公共close或shutdown方法的Java配置定义的bean将自动加入销毁回调。如果你有一个公共close或shutdown方法，并且不希望在容器关闭时调用它，则可以将@Bean（destroyMethod=“”）添加到bean定义中，以禁用默认（inferred）模式。
>
> 默认情况下，你可能希望为使用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。特别是，确保对DataSource总是这样做，因为它在JavaEE应用服务器上是有问题的。
>
> 下面的示例演示如何防止DataSource的自动销毁回调：

```
@Bean(destroyMethod="")
public DataSource dataSource() throws NamingException {
    return (DataSource) jndiTemplate.lookup("MyDS");
}
```

另外，对于@Bean方法，你通常使用编程的JNDI查找，或者使用Spring的JNDITemplate或JNDIlocatedElegate帮助器，或者直接使用JNDI InitialContext，但不使用JNDIObjectFactoryBean变量（这将强制你将返回类型声明为factoryBean类型，而不是实际的目标类型，从而使它更难用于那些打算引用所提供的资源的@Bean方法中的交叉引用调用）。

对于上述示例中的beanone，在构造期间直接调用init（）方法同样有效，如下示例所示：

```
@Configuration
public class AppConfig {

    @Bean
    public BeanOne beanOne() {
        BeanOne beanOne = new BeanOne();
        beanOne.init();
        return beanOne;
    }

    // ...
}
```

> 当你直接在Java中工作时，你可以用你的对象做任何事情，并不总是需要依赖于容器的生命周期。

声明Bean作用域

使用@Scope来声明作用域

在使用@Bean注解时候，你可以使用在Bean Scopes章节中讲到的任何定义scopes的方法。

默认的作用域是singleton， 但是你可以使用@Scope来重新，如下：

```
@Configuration
public class MyConfiguration {

    @Bean
    @Scope("prototype")
    public Encryptor encryptor() {
        // ...
    }
}
```

**@Scope和scoped-proxy

<aop:scoped-proxy/>元素。Java配置bean的@Scope注解的proxyMode属性，可以提供他等价的支持。默认值为no proxy（ScopedProxyMode.NO），但可以指定ScopedProxyMode.TARGET_CLASS或ScopedProxyMode.INTERFACES。

如果将具有范围的代理示例从XML引用文档（参见范围代理）导入到使用Java的“bean”，则类似于以下内容：

更多教程请访问:www.flydean.com

```
// 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，配置类使用@Bean方法的名称作为结果bean的名称。但是，可以使用name属性覆盖此功能，如下示例所示：

```
@Configuration
public class AppConfig {

    @Bean(name = "myThing")
    public Thing thing() {
        return new Thing();
    }
}
```

Bean的别名

正如在命名bean中所讨论的，有时需要为单个bean指定多个名称，也称为bean别名。@Bean注解的name属性为此接受一个字符串数组。下面的示例演示如何为bean设置多个别名：

```
@Configuration
public class AppConfig {

    @Bean({"dataSource", "subsystemA-dataSource", "subsystemB-dataSource"})
    public DataSource dataSource() {
        // instantiate, configure and return DataSource bean...
    }
}
```

Bean描述

有时，提供一个更详细的bean文本描述会有所帮助。当bean暴露（可能通过jmx）用于监视时，这尤其有用。 要向@Bean添加描述，可以使用@description注解，如下示例所示：

```
@Configuration
public class AppConfig {

    @Bean
    @Description("Provides a basic example of a bean")
    public Thing thing() {
        return new Thing();
    }
}
```

1.12.4 使用 @Configuration

更多教程请访问:www.flydean.com

@Configuration是一个类级注解，指示对象是bean定义的源。@Configuration类通过public @Bean注解方法声明bean。对@Configuration classes上的@Bean方法的调用也可用于定义Bean之间的依赖关系。请参见基本概念：@Bean和@Configuration以获取一般介绍。

注入bean间依赖

当bean相互依赖时，表示这种依赖就如同让一个bean方法调用另一个bean方法一样简单，如下示例所示：

```
@Configuration
public class AppConfig {

    @Bean
    public BeanOne beanOne() {
        return new BeanOne(beanTwo());
    }

    @Bean
    public BeanTwo beanTwo() {
        return new BeanTwo();
    }
}
```

在上面例子中，beanOnes通过构造函数注入，引入beanTwo。

> 只有当@Bean方法在@Configuration类中声明时，此声明bean间依赖关系的方法才有效。不能使用普通的@Component类声明bean之间的依赖关系。

查找方法注入

如前所述，查找方法注入是一个高级特性，你应该很少使用。它在单例作用域bean依赖于原型作用域bean的情况下很有用。对于这种类型的配置，使用Java提供了实现这种模式的自然手段。下面的示例演示如何使用查找方法注入：

```
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();
}
```

通过使用Java配置，可以创建CommandManager的子类，并重写抽象的createCommand（）方法，从而查找新的（prototype）命令对象。以下示例显示了如何执行此操作：

```
@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 createCommand()
    // overridden to return a new prototype Command object
    return new CommandManager() {
        protected Command createCommand() {
            return asyncCommand();
```

更多教程请访问:www.flydean.com

```
        }
    }
}
```

有关基于Java的配置如何在内部工作的进一步信息

下面展示了@Bean注解方法被调用两次：

```
@Configuration
public class AppConfig {

    @Bean
    public ClientService clientService1() {
        ClientServiceImpl clientService = new ClientServiceImpl();
        clientService.setClientDao(clientDao());
        return clientService;
    }

    @Bean
    public ClientService clientService2() {
        ClientServiceImpl clientService = new ClientServiceImpl();
        clientService.setClientDao(clientDao());
        return clientService;
    }

    @Bean
    public ClientDao clientDao() {
        return new ClientDaoImpl();
    }
}
```

clientDao()在clientService1() 和clientService2()中都被调用了一次。 因为 clientDao()创建了一个ClientDaoImpl实例，你可能认为创建了两个ClientDaoImpl实例， 但是在Spring中，bean实例通常拥有singleton作用域范围。

这是问题的关键：所有的@Configuration类都会在启动的时候被CGLIB继承，在子类中，它会去检测所有的缓存（作用域范围）的bean，然后才回去调用父类的方法去创建实例。

> 当然，根据Scope的不同情况可能不同，这里我们只讨论singleton。
>
> 在Spring3.2中，不在需要将CGLIB添加到你的classpath中了，因为CGLIB包已经被包括在了org.springframework.cglib，也就是spring-core JAR中。
>
> 使用CGLIB在启动时动态添加功能是有限制的。通常来说配置类必须不是final的。但是在4.3中，configuration 中允许添加任何构造函数，包括@Autowired或者单独的非默认的构造函数。
>
> 如果你希望避免任何cglib强加的限制，请考虑在非@Configuration类（例如，在普通的@Component类上）上声明@Bean方法。然后，@Bean方法之间的跨方法调用不会被拦截，因此你必须完全依赖于构造函数或方法级别的依赖注入。

1.12.5 组合基于Java的配置

Spring基于Java的配置特性允许你组合注解，这可以减少配置的复杂性。

使用@Import注解

与在Spring XML文件中使用<import/>元素来帮助模块化配置一样，@Import注解允许从另一个配置类加载@Bean定义，如下例所示：

```
@Configuration
public class ConfigA {
```

更多教程请访问:www.flydean.com

```
    @Bean
    public A a() {
        return new A();
    }
}

@Configuration
@Import(ConfigA.class)
public class ConfigB {

    @Bean
    public B b() {
        return new B();
    }
}
```

现在在实例化容器时，不需要显示的引入ConfigA.class 和 ConfigB.class，只要ConfigB就够了：

```
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的实例化，因为只需要处理一个类，而不需要在构造期间记住大量@Configuration类。

> 从SpringFramework4.2开始，@Import还支持引用常规组件类，类似于AnnotationConfigApplicationContext.register方法。如果你希望避免组件扫描，那么使用一些配置类作为入口点来显式定义所有组件，这尤其有用。

Imported @Bean上的注入依赖

前面的例子是可行的，但很简单。在大多数实际场景中，bean在配置类之间相互依赖。使用XML时，这不是一个问题，因为不涉及编译器，你可以声明ref="somebean"，并信任spring在容器初始化期间解决它。当使用@Configuration类时，Java编译器将约束放置在配置模型上，因为对其他bean的引用必须是有效的Java语法。

幸运的是，解决这个问题很简单。正如我们已经讨论过的那样，@Bean方法可以有任意数量的参数来描述bean依赖性。考虑以下更真实的场景，其中有几个@Configuration类，每个类取决于其他类中声明的bean：

```
@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
```

更多教程请访问:www.flydean.com

```
    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");
}
```

还有另一种方法可以达到同样的结果。记住，@Configuration类最终只是容器中的另一个bean：这意味着它们可以利用@Autowired和@Value注入以及与任何其他bean相同的其他特性。

> 确保以这种方式注入的依赖项是最简单的类型。@Configuration类在上下文的初始化过程中被提前处理，强制以这种方式注入依赖项可能会导致意外的早期初始化。尽可能采用基于参数的注入，如前面的示例所示。
>
> 另外，通过@Bean特别是BeanPostProcessor和BeanFactoryPostProcessor定义。这些方法通常应声明为static @Bean方法，而不是触发其包含的配置类的实例化。否则，@Autowired和@Value不能在configuration类本身上工作，因为它被创建为bean实例太早了。

下面的示例显示如何将一个bean自动连接到另一个bean：

```
@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);
```

更多教程请访问:www.flydean.com

```
    transferService.transfer(100.00, "A123", "C456");
}
```

> @Configuration类中的构造函数注入仅在Spring Framework 4.3之后受支持。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。在前面的示例中，RepositoryConfig构造函数上不需要@Autowired。

完全符合条件的imported beans，便于导航

在前面的场景中，使用@Autowired可以很好地工作并提供所需的模块性，但是确定在哪里声明autowired bean定义仍然有点含糊不清。例如，作为查看serviceConfig的开发人员，你如何确切知道@Autowired AccountRepositor bean的声明位置？它在代码中不是明确的，这可能很好。请记住，Spring工具套件提供了一种工具，它可以图形化的呈现显示所有东西是如何连接的，这可能是你所需要的全部。此外，你的Java IDE可以很容易地找到AccountRepository类型的所有声明和用法，并快速显示返回该类型的@Bean方法的位置。 如果这种模糊性是不可接受的，并且你希望在你的IDE中从一个@Configuration类直接导航到另一个@Configuration类，请考虑自动连接配置类本身。以下示例显示了如何执行此操作：

```
@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());
    }
}
```

在前面的情况下，定义AccountRepository是完全明确的。但是，serviceConfig现在与RepositoryConfig紧密耦合。这就是权衡。通过使用基于接口或基于抽象类的@Configuration类，可以稍微减轻这种紧密耦合。请考虑以下示例：

```
@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 {
```

更多教程请访问:www.flydean.com

```
    @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与具体的DefaultRepositoryConfig松散地耦合在一起，而内置的IDE工具仍然很有用：你可以很容易地获得RepositoryConfig实现的类型层次结构。通过这种方式，导航@Configuration类及其依赖项与导航基于接口的代码的常规过程没有什么不同。

如果要影响某些bean的启动创建顺序，请考虑将它们中的一些声明为@lazy（用于在第一次访问时创建而不是在启动时创建）或@dependson某些其他bean（确保在当前bean之前创建特定的其他bean，这超出了后者的直接依赖性的含义）。

条件的引入@Configuration和@Bean

根据某些任意的系统状态，有条件地启用或禁用完整的@Configuration类，甚至单个的@Bean方法，通常很有用。其中一个常见的例子是，只有在Spring环境中启用了特定的概要文件时，才使用@Profile注解来激活bean（有关详细信息，请参见bean定义概要文件）。

@Profile注解实际上是通过使用一个更灵活的名为@Conditional的注解来实现的。@Conditional annotation指示在注册@Bean之前应该咨询的特定org.springframework.context.annotation.Condition实现。

Condition接口的实现提供了一个返回true或false的匹配（...）方法。例如，下面的列表显示了用于@Profile的实际条件实现：

```
@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;
}
```

组合Java和XML配置

Spring的@Configuration类支持并不打算完全替代Spring XML。一些工具（如SpringXML名称空间）仍然是配置容器的理想方法。在XML方便或必要的情况下，你可以选择：以“以XML为中心”的方式实例化容器，例如使用ClassPathXmlApplicationContext，或者使用“AnnotationConfigApplicationContext”和“@ImportResource”按需导入XML，实例化容器。

以XML为中心使用@Configuration类

更多教程请访问:www.flydean.com

最好从XML引导Spring容器，并以特别的方式包含@Configuration类。例如，在使用SpringXML的大型现有代码库中，根据需要创建@Configuration类并从现有XML文件中包含它们更容易。在本节后面，我们将介绍在这种“以XML为中心”的情况下使用@Configuration类的选项。

将@Configuration classes声明为plain spring <bean/>元素

记住，@Configuration类最终是容器中的bean定义。在本系列示例中，我们创建了一个名为appconfig的@Configuration类，并将其作为<bean/>定义包含在system-test-config.xml中。由于打开了<context:annotation-config/>，容器识别@Configuration注解并正确处理appconfig中声明的@Bean方法。 下面的示例展示了Java中的一个普通配置类：

```
@Configuration
public class AppConfig {

    @Autowired
    private DataSource dataSource;

    @Bean
    public AccountRepository accountRepository() {
        return new JdbcAccountRepository(dataSource);
    }

    @Bean
    public TransferService transferService() {
        return new TransferService(accountRepository());
    }
}
```

下面是system-test-config.xml file文件：

```
<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 文件：

```
jdbc.url=jdbc:hsqldb:hsql://localhost/xdb
jdbc.username=sa
jdbc.password=
```

```
public static void main(String[] args) {
    ApplicationContext ctx = new ClassPathXmlApplicationContext("classpath:/com/acme/system-test-config.xml");
    TransferService transferService = ctx.getBean(TransferService.class);
    // ...
}
```

> 在system-test-config.xml文件中，appconfig <bean/>不声明id元素。虽然这样做是可以接受的，但这是不必要的，因为没有其他bean引用过它，并且不太可能按名称从容器中显式地提取它。类似地，数据源bean也只是按类型自动连接的，因此不严格要求显式bean id。

更多教程请访问:www.flydean.com

使用<context:component-scan/>来查找@Configuration类

因为@Configuration是用@Component元注解的，@Configuration注解类自动成为组件扫描的候选者。使用前面示例中描述的相同场景，我们可以重新定义system-test-config.xml以利用组件扫描。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/>启用了相同的功能。 以下示例显示修改后的system-test-config.xml文件：

```
<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>
```

以@Configuration类为中心使用XML与@ImportResource

在@Configuration classes是配置容器的主要机制的应用程序中，至少需要使用一些XML。在这些场景中，你可以使用@ImportResource，并且只定义所需的XML。这样做实现了“以Java为中心”的方法来配置容器，并将XML保持在最低限度。下面的示例（包括配置类、定义Bean的XML文件、属性文件和主类）显示了如何使用@ImportResource注解来实现“以Java为中心”的配置，根据需要使用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);
    }
}
```

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

properties-config.xml

```
jdbc.properties
jdbc.url=jdbc:hsqldb:hsql://localhost/xdb
jdbc.username=sa
jdbc.password=
```

```
public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);
    TransferService transferService = ctx.getBean(TransferService.class);
    // ...
```

更多教程请访问:www.flydean.com

```
}
```

更多教程请访问:www.flydean.com

1.13 环境抽象

Environment接口是集成在容器中的抽象，它为应用程序环境的两个关键方面建模：profiles 和properties。

profile是一个命名的逻辑bean定义组，只有在给定的配置文件处于活动状态时，才会在容器中注册。bean可以分配给一个配置文件，无论是用XML定义的还是用注解定义的。与配置文件相关的环境对象的角色是确定哪些配置文件（如果有）当前处于活动状态，以及默认情况下哪些配置文件（如果有）应处于活动状态。

Properties在几乎所有的应用程序中都扮演着重要的角色，可能源于各种源：属性文件、JVM系统属性、系统环境变量、JNDI、servlet上下文参数、特殊属性对象、映射对象等。环境对象与属性相关的作用是为用户提供一个方便的服务接口，用于配置属性源并从中解析属性。

1.13.1 Bean定义Profiles

bean定义Profiles在核心容器中提供了一种机制，允许在不同环境中注册不同的bean。“环境”这个词对不同的用户来说可能意味着不同的东西，并且这个特性可以帮助处理许多用例，包括：

- 在开发中处理内存中的数据源，而不是在QA或生产中从JNDI中查找相同的数据源。
- 仅在将应用程序部署到性能环境中时注册监控基础结构。
- 为客户A和客户B部署注册定制的bean实现。

在需要DataSource的实际应用程序中考虑第一个用例。在测试环境中，配置可能类似于以下内容：

```
@Bean
public DataSource dataSource() {
    return new EmbeddedDatabaseBuilder()
        .setType(EmbeddedDatabaseType.HSQL)
        .addScript("my-schema.sql")
        .addScript("my-test-data.sql")
        .build();
}
```

现在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已注册到生产应用程序服务器的JNDI目录中。我们的数据源bean现在看起来如下所示：

```
@Bean(destroyMethod="")
public DataSource dataSource() throws Exception {
    Context ctx = new InitialContext();
    return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");
}
```

问题是如何根据当前环境在使用这两种变体之间进行切换。随着时间的推移，Spring用户设计了许多实现这一点的方法，通常依赖于系统环境变量和包含$placeholder标记的XML<import/>语句的组合，这些标记根据环境变量的值解析为正确的配置文件路径。bean定义概要文件是一个核心容器特性，它为这个问题提供了解决方案。

如果我们概括前面环境特定bean定义示例中所示的用例，那么最终需要在特定上下文中注册特定bean定义，而不是在其他上下文中注册特定bean定义。可以说，你希望在情况A中注册bean定义的某个概要文件，在情况B中注册另一个概要文件。我们首先更新配置以反映这种需求。

使用@Profile

@Profile注解允许你指示一个组件在一个或多个指定的配置文件处于活动状态时可以注册。使用前面的示例，我们可以如下重写数据源配置：

更多教程请访问:www.flydean.com

```
@Configuration
@Profile("development")
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");
    }
}
```

> 如上所示，在@Bean中，你显示的使用编程式JNDI查找，通过使用Spring的JndiTemplate/JndiLocatorDelegate，或者JNDI的InitialContext， 而不是JndiObjectFactoryBean， **这会让你的返回值变成FactoryBean。

配置文件字符串可以包含简单的配置文件名（例如，production）或配置文件表达式。配置文件表达式允许表达更复杂的配置文件逻辑（例如，production & us-east）。配置文件表达式中支持以下运算符：

- ! 逻辑非
- & 逻辑与
- | 逻辑或

> 你不能不加括号来混合使用 & 和 | ，如production & us-east | eu-central 是错误的，应该production & (us-east | eu-central)。

可以将@Profile用作元注解，以创建自定义组合注解。以下示例定义了一个自定义的@Production注解，你可以将其用作@Profile（“production”）的插入替换：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Profile("production")
public @Interface Production {
}
```

> 如果@Configuration类被标记为@Profile，那么所有与该类关联的@Bean方法和@Import注解都将被忽略，除非一个或多个指定的概要文件处于活动状态。如果@Component或@Configuration类标记为@Profile（“p1”、“p2”），则除非激活了配置文件“p1”或“p2”，否则不会注册或处理该类。如果给定配置文件的前缀是not运算符（！），仅当配置文件未处于活动状态时，才会注册带注解的元素。例如，给定@Profile（“p1”，“！p2“”），如果配置文件“p1”处于活动状态或配置文件“p2”未处于活动状态，则会发生注册。

@Profile也可以用在方法级别用来包含一个特殊的bean或者配置类。如下所示：

```
@Configuration
public class AppConfig {
```

更多教程请访问:www.flydean.com

```
    @Bean("dataSource")
    @Profile("development")
    public DataSource standaloneDataSource() {
        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("dataSource")
    @Profile("production")
    public DataSource jndiDataSource() throws Exception {
        Context ctx = new InitialContext();
        return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");
    }
}
```

> 使用@Profile在@Bean方法中，可以使用特殊场景：在相同Java方法名称（类似于构造函数重载）的重载@Bean方法的情况下，需要在所有重载方法上一致地声明@Profile条件。如果条件不一致，那么只有重载方法中第一个声明的条件才重要。因此，@Profile不能用于选择具有特定参数签名的重载方法。同一bean的所有工厂方法之间的解析在创建时遵循Spring的构造函数解析算法。
>
> 如果要定义具有不同配置条件的替代bean，请使用不同的Java方法名称，使用前面的示例所示，使用“bean名称”属性指向相同的bean名称。如果参数签名都是相同的（例如，所有的变体都没有带参的工厂方法），这是在一个有效的Java类中表示这种安排的唯一方式（因为只有一个特定名称和参数签名的方法）。

Profiles在XML中使用

可以在XML中使用profile属性，如下所示：

```
<beans profile="development"
    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>
```

同样可以将两个XML合并成一个：

```
<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 -->
```

更多教程请访问:www.flydean.com

```
    <beans profile="development">
        <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>
```

spring-bean.xsd已经被约束为只允许作为文件中的最后一个元素。这有助于提供灵活性，而不会在XML文件中引起混乱。

> XML对应项不支持前面描述的配置文件表达式。但是，可以使用否定配置文件！操作元。也可以通过嵌套配置文件来应用逻辑“and”，如下示例所示：

```
<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="production">
        <beans profile="us-east">
            <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>
        </beans>
    </beans>
</beans>
```

上面的例子中，dataSource被用在production和us-east profiles。

激活Profile

既然我们已经更新了配置，我们仍然需要指示Spring哪个配置文件是活动的。如果我们现在启动示例应用程序，我们会看到一个NoSuchBeanDefinitionException被抛出，因为容器找不到名为datasource的SpringBean。

激活一个概要文件可以用几种方法完成，但最简单的方法是通过应用程序上下文提供的环境API以编程方式完成。以下示例显示了如何执行此操作：

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
ctx.getEnvironment().setActiveProfiles("development");
ctx.register(SomeConfig.class, StandaloneDataConfig.class, JndiDataConfig.class);
ctx.refresh();
```

此外，还可以通过spring.profiles.active属性声明性地激活概要文件，该属性可以通过系统环境变量、jvm系统属性、web.xml中的servlet上下文参数指定，甚至可以作为JNDI中的条目指定（请参见PropertySource抽象）。在集成测试中，可以使用Spring测试模块中的@Active profiles注解声明活动配置文件（请参见环境配置文件的上下文配置）。

请注意，配置文件不是“要么或”命题。你可以一次激活多个配置文件。通过编程，你可以为setActiveProfiles（）方法提供多个配置文件名，该方法接受String… varargs。以下示例激活多个配置文件：

```
ctx.getEnvironment().setActiveProfiles("profile1", "profile2");
```

在声明中，spring.profiles.active可以接受由逗号分隔的配置文件名列表，如下例所示：

```
-Dspring.profiles.active="profile1,profile2"
```

更多教程请访问:www.flydean.com

默认Profile

默认的Profile表示该Profile默认被激活，如下所示：

```
@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Profile被激活，那么dataSource就会被创建，你可以看成创建bean的默认方式。如果其他的Profile被激活了，那么默认的Profile就不会被使用。

你可以在环境中使用SetDefaultProfiles（）更改默认profile的名称，或者声明性地使用spring.profiles.default属性更改默认概要文件的名称。

1.13.2 PropertySource简介

Spring的Environment抽象在property sources的可配置层次结构上提供搜索操作。考虑以下列表：

```
ApplicationContext ctx = new GenericApplicationContext();
Environment env = ctx.getEnvironment();
boolean containsMyProperty = env.containsProperty("my-property");
System.out.println("Does my environment contain the 'my-property' property? " + containsMyProperty);
```

在前面的片段中，我们看到了一种高级方法，询问Spring是否为当前环境定义了my-property属性。为了回答这个问题，Environment对象对一组PropertySource对象执行搜索。PropertySource是对任何键值对源的简单抽象，Spring的StandardEnvironment配置了两个PropertySource对象：一个表示一组JVM系统属性（System.GetProperties（）），另一个表示一组系统环境变量。es（system.getenv（））。

> 这些默认属性源使用StandardEnvironment，可在独立应用程序中使用。StandardServletEnvironment使用其他默认属性源（包括servlet config和servlet上下文参数）填充。它可以选择启用JndiPropertySource。有关详细信息，请参阅JavaDoc。

具体来说，在使用StandardEnvironment时，如果运行时存在my-property系统属性或my-property环境变量，则对env.containsProperty（“my-property”）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（“my-property”）的过程中，在这两个位置都设置了my-property属性，则系统属性值“wins”并返回。请注意，属性值不会合并，而是完全被前面的条目覆盖。
>
> 对于常见的StandardServletenvironment，完整的层次结构如下，最高优先级条目位于顶部：
>
> - ServletConfig参数（如果适用-例如，对于DispatcherServlet上下文）
> - ServletContext参数（web.xml context-param 项）
> - 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（操作系统环境变量）

更多教程请访问:www.flydean.com

最重要的是，整个机制是可配置的。也许你有一个自定义的属性源，希望将其集成到这个搜索中。为此，请实现并实例化自己的PropertySource，并将其添加到当前环境的PropertySource集合中。以下示例显示了如何执行此操作：

```
ConfigurableApplicationContext ctx = new GenericApplicationContext();
MutablePropertySources sources = ctx.getEnvironment().getPropertySources();
sources.addFirst(new MyPropertySource());
```

在前面的代码中，MyPropertySource已在搜索中以最高优先级添加。如果它包含“我的属性”属性，则检测并返回该属性，而不是任何其他属性源中的“我的属性”属性。MutablePropertySources API公开了许多允许对属性源集进行精确操作的方法。

1.13.3 使用@PropertySource

@PropertySource注解提供了方便和声明式的机制为Spring的添加PropertySource。

假如一个app.properties包括key-value值，例如：testbean.name=myTestBean， 下面的@Configuration类使用@PropertySource 来调用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;
    }
}
```

@Propertysource资源位置中存在的任何$...占位符将根据已针对环境注册的属性源集进行解析，如下示例所示：

```
@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存在于已注册的某个属性源中（例如，系统属性或环境变量），则将占位符解析为相应的值。如果不是，则default/path用作默认值。如果未指定默认值且无法解析属性，则将引发IllegalArgumentException。

> 根据Java 8约定，@PropertySource注解是可重复的。但是，所有这样的@PropertySource注解都需要在同一级别声明，要么直接在配置类上声明，要么作为同一自定义注解中的元注解声明。不建议混合使用直接注解和元注解，因为直接注解有效地覆盖了元注解。

更多教程请访问:www.flydean.com

1.13.4 语句中的占位符解析

历史上，元素中占位符的值只能根据JVM系统属性或环境变量来解析。情况不再是这样了。因为环境抽象集成在整个容器中，所以很容易通过它来路由占位符的解析。这意味着你可以以任何你喜欢的方式配置解析过程。你可以更改搜索系统属性和环境变量的优先级，或者完全删除它们。你还可以根据需要将自己的属性源添加到组合中。

具体来说，只要customer property在Environment中可用，无论customer property在何处定义，以下陈述都有效：

```
<beans>
    <import resource="com/bank/service/${customer}-config.xml"/>
</beans>
```

更多教程请访问:www.flydean.com

1.14 注册LoadTimeWeaver

当类被加载进入JVM的时候，LoadTimeWeaver被Spring用来进行动态类转换。

开启 load-time weaving， 可以添加@EnableLoadTimeWeaving到@Configuration类，如下所示：

```
@Configuration
@EnableLoadTimeWeaving
public class AppConfig {
}
```

或者可以使用XML的context:load-time-weaver，如下：

```
<beans>
    <context:load-time-weaver/>
</beans>
```

一旦为ApplicationContext配置好，该ApplicationContext中的任何bean都可以实现LoadTimeWeaverAware，从而接收对加载时weaver实例的引用。这在结合Spring的JPA支持时特别有用，因为在这种支持下，JPA类转换可能需要负载时间编织。有关详细信息，请参阅LocalContainerEntityManagerFactoryBean JavaDoc。有关Aspectj加载时编织的更多信息，请参见Spring框架中的Load Time Weaving with Aspectj。

更多教程请访问:www.flydean.com

1.15 ApplicationContext的其他功能

正如在引言一章中所讨论的，org.springframework.beans.factory包提供了管理和操作bean的基本功能，包括以编程方式。context包添加了ApplicationContext接口，它扩展了BeanFactory接口，此外还扩展了其他接口，以提供更多面向应用程序框架的功能。许多人以完全声明的方式使用ApplicationContext，甚至不是以编程方式创建应用程序上下文，而是依赖于诸如ContextLoader这样的支持类来自动实例化应用程序上下文，作为JavaEE Web应用程序正常启动过程的一部分。

为了以更面向框架的方式增强BeanFactory功能，上下文包还提供以下功能：

- 通过MessageSource接口访问i18n样式的消息。
- 通过ResourceLoader接口访问资源，如URL和文件。
- 事件发布，即通过使用applicationEventPublisher接口向实现ApplicationListener接口的bean发布。
- 通过HierarchicalBeanFactory接口加载多个（层次）上下文，使每个上下文集中在一个特定的层上，例如应用程序的Web层。

1.15.1 MessageSource国际化

ApplicationContext接口扩展了一个名为MessageSource的接口，因此提供国际化（“i18n”）功能。Spring还提供了层次化的消息源接口，可以对消息进行层次化的解析。这些接口共同提供了Spring效应消息解决的基础。在这些接口上定义的方法包括：

- String getMessage(String code, Object[] args, String default, Locale loc):从MessageSource获取消息的基本方法，如果在提供的locale中没有找到消息，那么将使用默认的消息。通过使用标准库提供的MessageFormat功能，传入的任何参数都将成为替换值。
- String getMessage(String code, Object[] args, Locale loc):基本上和上面方法一样，但是有一个区别：没有默认小消息。如果消息没找到，那么NoSuchMessageException将抛出。
- String getMessage(MessageSourceResolvable resolvable, Locale locale): 所有的之前方法的属性，都被封装在名为MessageSourceResolvable的类中，你可以使用它的方法。

加载ApplicationContext时，它会自动搜索上下文中定义的MessageSource bean。MessageSource bean必须具有名称。如果找到这样的bean，那么对前面方法的所有调用都将委托给消息源。如果找不到消息源，ApplicationContext将尝试查找包含同名bean的父级。如果是这样，它将使用该bean作为消息源。如果ApplicationContext找不到消息的任何源，则会实例化一个空的DelegatingMessageSource，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。两者都实现HierarchicalMessageSource，以便执行嵌套消息传递。StaticMessageSource很少使用，但提供了向源添加消息的编程方法。以下示例显示了ResourceBundleMessageSource：

```
<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>
```

更多教程请访问:www.flydean.com

该示例假定在类路径中定义了三个资源包，分别称为format、exceptions和windows。解析消息的任何请求都以通过ResourceBundle对象解析消息的JDK标准方式处理。在本例中，假设上述两个资源包文件的内容如下：

```
# in format.properties
message=Alligators rock!

# in exceptions.properties
argument.required=The {0} argument is required.
```

下一个例子显示了一个执行MessageSource功能的程序。记住，所有ApplicationContext实现也是MessageSource实现，因此可以强制转换到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);
}
```

运行结果如下：

```
Alligators rock!
```

总而言之，MessageSource是在一个名为beans.xml的文件中定义的，该文件位于类路径的根目录下。MessageSource bean定义通过其basename属性引用许多资源束。在列表中传递给basename属性的三个文件作为文件存在于类路径的根目录中，分别称为format.properties、exceptions.properties和windows.properties。

下一个示例显示传递给消息查找的参数。这些参数被转换成字符串对象并插入到查找消息中的占位符中。

```
<beans>

    <!-- this MessageSource is being used in a web application -->
    <bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basename" value="exceptions"/>
    </bean>

    <!-- lets inject the above MessageSource into this POJO -->
    <bean id="example" class="com.something.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);
    }
}
```

execute()的执行结果如下：

更多教程请访问:www.flydean.com

```
The userDao argument is required.
```

关于国际化（“i18n”），Spring的各种消息源实现遵循与标准JDK ResourceBundle相同的区域设置解析和回退规则。简而言之，继续前面定义的示例MessageSource，如果要针对英国（en-GB）区域设置解析消息，则将分别创建名为format*en*-GB.properties、exceptions*en*-GB.properties和windows*en*-GB.properties的文件。

通常，区域设置解析由应用程序的周围环境管理。在下面的示例中，将手动指定解析（英国）消息所依据的区域设置。

```
# 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);
}
```

运行结果如下：

```
Ebagum lad, the 'userDao' argument is required, I say, required.
```

你还可以使用MessageSourceAware接口获取对已定义的任何MessageSource的引用。在ApplicationContext中定义的、实现MessageSourceAware接口的任何bean在创建和配置bean时都会与应用程序上下文的MessageSource一起注入。

> 作为ResourceBundleMessageSource的替代方案，Spring提供了一个ReloadableResourceBundleMessageSource类。此变体支持相同的包文件格式，但比基于JDK的标准ResourceBundleMessageSource实现更灵活。特别是，它允许从任何Spring资源位置（不仅是从类路径）读取文件，并支持捆绑属性文件的热重新加载（同时有效地在两者之间缓存它们）。有关详细信息，请参阅ReloadableResourceBundleMessageSource javadoc。

1.15.2 标准和自定义Events

ApplicationContext中的事件处理通过ApplicationEvent类和ApplicationListener接口提供。如果实现ApplicationListener接口的bean部署到上下文中，则每次ApplicationEvent发布到ApplicationContext时，都会通知该bean。本质上，这是标准的观测者设计模式。

> 从Spring4.2开始，事件基础结构得到了显著的改进，提供了一个基于注解的模型以及发布任意事件的能力（也就是说，不一定从ApplicationEvent扩展的对象）。当这样一个对象被发布时，我们将它包装在一个事件中。

下表描述了Spring提供的标准事件描述：

Event	解释
ContextRefreshedEvent	在初始化或刷新ApplicationContext时发布（例如，通过在ConfigurableApplicationContext接口上使用refresh（）方法）。这里，“初始化”意味着加载所有bean，检测并激活后处理器bean，预先实例化单例，并且ApplicationContext对象准备好使用。只要上下文未关闭，只要所选的ApplicationContext实际上支持此类“热”刷新，就可以多次触发刷新。例如，XMLWebApplicationContext支持热刷新，但GenericApplicationContext不支持。
ContextStartedEvent	在可配置的ApplicationContext接口上使用start（）方法启动ApplicationContext时发布。这里，“启动”意味着所有生命周期bean都会收到一个显式的启动信号。通常，此信号用于在显式停止后重新启动bean，但也可以用于启动尚未配置为自动启动的组件（例如，初始化时尚未启动的组件）。

更多教程请访问:www.flydean.com

ContextStoppedEvent	在可配置的ApplicationContext接口上使用stop（）方法停止ApplicationContext时发布。这里，“停止”意味着所有生命周期bean都会收到一个明确的停止信号。停止的上下文可以通过start（）调用重新启动。
ContextClosedEvent	在可配置的ApplicationContext接口上使用close（）方法关闭ApplicationContext时发布。这里，“关闭”意味着所有的单例beans都被销毁了。封闭的环境达到了生命的尽头。无法刷新或重新启动。
RequestHandledEvent	一个特定于Web的事件，告诉所有bean HTTP请求已被服务。此事件在请求完成后发布。此事件仅适用于使用Spring的DispatcherServlet的Web应用程序。

你也可以创建自己的事件。下面显示了继承ApplicationEvent的例子：

```
public class BlackListEvent extends ApplicationEvent {

    private final String address;
    private final String content;

    public BlackListEvent(Object source, String address, String content) {
        super(source);
        this.address = address;
        this.content = content;
    }

    // accessor and other methods...
}
```

若要发布自定义ApplicationEvent，请在ApplicationEventPublisher上调用PublishEvent（）方法。通常，这是通过创建一个实现ApplicationEventPublisherAware的类并将其注册为SpringBean来完成的。下面的示例显示了这样的类：

```
public class EmailService implements ApplicationEventPublisherAware {

    private List<String> blackList;
    private ApplicationEventPublisher publisher;

    public void setBlackList(List<String> blackList) {
        this.blackList = blackList;
    }

    public void setApplicationEventPublisher(ApplicationEventPublisher publisher) {
        this.publisher = publisher;
    }

    public void sendEmail(String address, String content) {
        if (blackList.contains(address)) {
            publisher.publishEvent(new BlackListEvent(this, address, content));
            return;
        }
        // send email...
    }
}
```

在配置时，Spring容器检测到EmailService实现了ApplicationEventPublisherAware，并自动调用setApplicationEventPublisher（）。实际上，传入的参数是Spring容器本身。你正在通过其applicationEventPublisher接口与应用程序上下文进行交互。

要接收定制的applicationEvent，可以创建一个实现applicationListener的类，并将其注册为SpringBean。下面的示例显示了这样的类：

```
public class BlackListNotifier implements ApplicationListener<BlackListEvent> {

    private String notificationAddress;
```

更多教程请访问:www.flydean.com

```
    public void setNotificationAddress(String notificationAddress) {
        this.notificationAddress = notificationAddress;
    }

    public void onApplicationEvent(BlackListEvent event) {
        // notify appropriate parties via notificationAddress...
    }
}
```

注意，applicationListener通常是用自定义事件的类型（前面的示例中是blacklistevent）参数化的。这意味着onApplicationEvent（）方法可以保持类型安全，避免任何向下强制转换的需要。你可以注册任意多个事件侦听器，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（）方法将一直阻塞，直到所有侦听器完成对事件的处理。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，它将在发布服务器的事务上下文中操作。如果需要事件发布的另一个策略，请参阅JavaDoc for Spring的ApplicationEventMulticaster接口。

下面的示例显示了用于注册和配置上述每个类的bean定义：

```
<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>
```

综上所述，当调用emailService bean的sendEmail（）方法时，如果有任何电子邮件应被列入黑名单，则会发布一个blacklistevent类型的自定义事件。黑名单通知bean注册为ApplicationListener并接收黑名单事件，此时它可以通知相应的方。

> Spring的事件机制是为同一应用程序上下文中SpringBean之间的简单通信而设计的。然而，对于更复杂的企业集成需求，单独维护的Spring Integration为构建基于众所周知的Spring编程模型的轻量级、面向模式、事件驱动的体系结构提供了完整的支持。

基于注解的Event Listeners

从Spring4.2开始，你可以使用EventListener注解在托管bean的任何公共方法上注册事件侦听器。BlackListNotifier程序可以改写如下：

```
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...
    }
}
```

方法签名再次声明它要侦听的事件类型，但这次使用灵活的名称，而不实现特定的侦听器接口。只要实际事件类型在其实现层次结构中解析泛型参数，就可以通过泛型缩小事件类型。

更多教程请访问:www.flydean.com

如果你的方法应该监听多个事件，或者你想要定义它而不使用任何参数，那么也可以在注解本身上指定事件类型。以下示例显示了如何执行此操作：

```
@EventListener({ContextStartedEvent.class, ContextRefreshedEvent.class})
public void handleContextStart() {
    ...
}
```

还可以使用定义spEL表达式的注解的条件属性添加其他运行时筛选，该表达式应与实际调用特定事件的方法相匹配。

以下示例显示了如何重写通知程序，以便仅在事件的内容属性等于我的事件时调用：

```
@EventListener(condition = "#blEvent.content == 'my-event'")
public void processBlackListEvent(BlackListEvent blEvent) {
    // notify appropriate parties via notificationAddress...
}
```

每个spEL表达式都根据专用上下文进行计算。下表列出了上下文可用的项，以便你可以将它们用于条件事件处理：

name	Location	描述	例子
Event	root object	真实的ApplicationEvent	#root.event
Arguments array	root object	调用目标的参数	#root.args[0]
Argument name	evaluation context	任何方法参数的名称。如果由于某种原因，名称不可用（例如，因为没有调试信息），参数名称也可以在 #a<#arg>下使用，其中#arg表示参数索引（从0开始）。	#blEvent or #a0 (也可以使用 #p0 or #p<#arg>)

请注意，root.event允许你访问基础事件，即使你的方法签名实际上引用了已发布的任意对象。

如果在处理另一个事件时需要发布事件，可以更改方法签名以返回应发布的事件，如下例所示：

```
@EventListener
public ListUpdateEvent handleBlackListEvent(BlackListEvent event) {
    // notify appropriate parties via notificationAddress and
    // then publish a ListUpdateEvent...
}
```

> 异步Listener不支持整个特性

此新方法为上述方法处理的每个BlackListEvent发布一个新的ListUpdateEvent。如果需要发布多个事件，可以返回事件集合。

异步侦听器

如果希望特定的侦听器异步处理事件，可以重用常规的@Async支持。以下示例显示了如何执行此操作：

```
@EventListener
@Async
public void processBlackListEvent(BlackListEvent event) {
    // BlackListEvent is processed in a separate thread
}
```

注意下面的异步事件限制：

- 如果事件listener抛出Exception，异常并不会传递给调用者。参考AsyncUncaughtExceptionHandler。
- 这样的事件listener不能发送回复。作为listener的结果，如果你想发送另外的事件，调用ApplicationEventPublisher

更多教程请访问:www.flydean.com

来手动发送。

Listeners排序

如果需要先调用一个监听器，然后再调用另一个监听器，则可以将@order注解添加到方法声明中，如下例所示：

```
@EventListener
@Order(42)
public void processBlackListEvent(BlackListEvent event) {
    // notify appropriate parties via notificationAddress...
}
```

一般事件

你还可以使用泛型进一步定义事件的结构。考虑使用EntityCreatedEvent，其中t是创建的实际实体的类型。例如，你可以创建以下侦听器定义以仅接收某个人的EntityCreatedEvent：

```
@EventListener
public void onPersonCreated(EntityCreatedEvent<Person> event) {
    ...
}
```

由于类型擦除，只有在激发的事件解析事件侦听器筛选的通用参数（比如类PersonCreatedEvent扩展EntityCreatedEvent…）时，此操作才有效。

在某些情况下，如果所有事件都遵循相同的结构（如前一个示例中的事件一样），则这可能会变得相当乏味。在这种情况下，你可以实现ResolvableTypeProvider程序，以指导框架超越运行时环境提供的内容。以下事件显示如何执行此操作：

```
public class EntityCreatedEvent<T> extends ApplicationEvent implements ResolvableTypeProvider {

    public EntityCreatedEvent(T entity) {
        super(entity);
    }

    @Override
    public ResolvableType getResolvableType() {
        return ResolvableType.forClassWithGenerics(getClass(), ResolvableType.forInstance(getSource()));
    }
}
```

> 这不仅适用于ApplicationEvent，还适用于作为事件发送的任意对象。

1.15.3 方便获取低级资源

为了优化应用程序上下文的使用和理解，你应该熟悉Spring的资源抽象，如参考资料中所述。

应用程序上下文是可用于加载资源对象的ResourceLoader。资源本质上是JDK java.net.URL类的功能更丰富的版本。实际上，资源的实现在适当的情况下包装了java.net.url的实例。Resource可以以透明的方式从几乎任何位置获取低级资源，包括从类路径、文件系统位置、使用标准URL可描述的任何位置以及其他一些变体。如果资源位置字符串是没有任何特殊前缀的简单路径，那么这些资源的来源是特定的，并且适合于实际的应用程序上下文类型。

可以将部署到应用程序上下文中的bean配置为实现特殊的回调接口ResourceLoaderAware，以便在初始化时自动回调，并将应用程序上下文本身作为ResourceLoader传入。你还可以公开资源类型的属性，以用于访问静态资源。它们像其他任何性质一样被注入其中。你可以将这些资源属性指定为简单的字符串路径，并在部署bean时依赖于从这些文本字符串到实际资源对象的自动转换。

更多教程请访问:www.flydean.com

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，并且以简单的形式根据特定的上下文实现进行适当的处理。例如，ClassPathXMLApplicationContext将简单位置路径视为类路径位置。也可以使用带有特殊前缀的位置路径（资源字符串）强制从类路径或URL加载定义，而不管实际的上下文类型如何。

1.15.4 方便的Web应用程序应用上下文实例化

可以通过使用ContextLoader等声明性地创建ApplicationContext实例。当然，你也可以使用ApplicationContext实现之一以编程方式创建applicationContext实例。

可以使用ContextLoaderListener注册ApplicationContext，如下示例所示：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/daoContext.xml /WEB-INF/applicationContext.xml</param-value>
</context-param>

<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

侦听器检查ContextConfigLocation参数。如果参数不存在，侦听器将使用/WEB-INF/ApplicationContext.xml作为默认值。当参数确实存在时，侦听器使用预定义的分隔符（逗号、分号和空格）分隔字符串，并将这些值用作搜索应用程序上下文的位置。还支持Ant-style 的路径模式。例如/WEB-INF/_context.xml（对于名称以context.xml结尾并位于WEB-INF目录中的所有文件）和/WEB-INF/**/_context.xml（对于WEB-INF的任何子目录中的所有此类文件）。

1.15.5将ApplicationContext部署为Java EE RAR文件

可以将Spring ApplicationContext作为RAR文件部署，在JavaEE RAR部署单元中封装上下文及其所有所需的bean类和库JAR。这相当于引导一个独立的ApplicationContext（仅托管在JavaEE环境中）能够访问JavaEE服务器设施。RAR部署是一种更为自然的替代方案，用于部署一个headless WAR文件——实际上，一个没有任何HTTP入口点的WAR文件，它仅用于在JavaEE环境中引导Spring ApplicationContext。

RAR部署非常适合不需要HTTP入口点，而是只包含消息端点和计划作业的应用程序上下文。在这种上下文中，bean可以使用应用服务器资源，如jta事务管理器和jndi绑定的jdbc DataSource实例和jms ConnectionFactory实例，还可以通过Spring的标准事务管理和jndi和jmx支持设施在平台的jmx服务器-注册。应用程序组件还可以通过Spring的TaskExecutor抽象与应用服务器的JCA WorkManager交互。

请参阅SpringContextResourceAdapter类的JavaDoc，了解RAR部署中涉及的配置详细信息。

对于Spring ApplicationContext作为JavaEE RAR文件的简单部署：

1. 将所有应用程序类打包成一个rar文件（这是一个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存档的根目录中。.添加一个META-INF/ra.xml部署描述符（如SpringContextResourceAdapter的JavaDoc中所示）和相应的SpringXML bean定义文件（通常为'META-INF/applicationContext.xml）。
2. 将生成的rar文件放到应用程序服务器的部署目录中。

> 这种RAR部署单元通常是独立的。它们不向外部世界公开组件，甚至不向同一应用程序的其他模块公开组件。与基于rar的ApplicationContext的交互通常通过与其他模块共享的JMS目标进行。例如，基于rar的ApplicationContext还可以调度一些作业或对文件系统（或类似系统）中的新文件做出反应。如果需要允许从外部进行同步访问，它可以（例如）导出RMI端点，该端点可以由同一台计算机上的其他应用程序模块使用。

更多教程请访问:www.flydean.com

更多教程请访问:www.flydean.com

1.16 BeanFactory

BeanFactory API为Spring的IOC功能提供了基础。其特定的契约主要用于与Spring和相关第三方框架的其他部分集成，其默认的ListableBeanFactory实现是更高级别GenericApplicationContext容器中的关键委托。

BeanFactory和相关接口（如BeanFactoryAware、InitializingBean、DisposableBean）是其他框架组件的重要集成点。通过不需要任何注解甚至反射，它们允许容器及其组件之间非常有效的交互。应用程序级bean可以使用相同的回调接口，但通常更倾向于通过注解或编程配置进行声明性依赖项注入。

请注意，核心BeanFactory API级别及其DefaultListableBeanFactory实现不会使用使用配置格式或任何组件注解。所有这些风格都通过扩展（如xmlbeanDefinitionReader和autowiredAnnotationBeanPostProcessor）传入，并作为核心元数据表示对共享BeanDefinition对象进行操作。这就是使Spring容器如此灵活和可扩展的本质所在。

1.16.1 BeanFactory 还是 ApplicationContext？

本节解释BeanFactory和ApplicationContext容器级别之间的差异以及引导的含义。

除非有充分的理由不这样做，否则应该使用ApplicationContext，将GenericApplicationContext及其子类AnnotationConfigApplicationContext作为自定义引导的常见实现。这些是Spring核心容器的主要入口点，用于所有常见用途：加载配置文件、触发类路径扫描、以编程方式注册bean定义和带注解的类，以及（从5.0开始）注册功能性bean定义。

由于ApplicationContext包含BeanFactory的所有功能，因此通常建议将其置于普通BeanFactory之上，但需要完全控制Bean处理的情况除外。在ApplicationContext（例如GenericapplicationContext实现）中，通过约定（即，通过bean名称或bean类型-特别是后处理器）检测到几种bean，而普通的DefaultListableBeanFactory对于任何特殊bean都是不可知的。

对于许多扩展容器特性，如注解处理和AOP代理，BeanPostProcessor扩展点是必不可少的。如果只使用普通的DefaultListableBeanFactory，则默认情况下不会检测到此类后处理器并激活它们。这种情况可能会令人困惑，因为bean配置实际上没有任何问题。相反，在这种情况下，需要通过额外的设置完全引导容器。

下表列出了BeanFactory和ApplicationContext接口和实现提供的功能。

功能	BeanFactory	ApplicationContext
Bean实例化和注入	Yes	Yes
和生命周期集成	No	Yes
自动BeanPostProcessor注册	No	Yes
自动BeanFactoryPostProcessor注册	No	Yes
MessageSource便捷访问（内部）	No	Yes
内置的ApplicationEvent发布机制	No	yes

显示使用DefaultListableBeanFactory注册bean的post-processor， 你需要在程序中调用addBeanPostProcessor，如下：

```
DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
// populate the factory with bean definitions

// now register any needed BeanPostProcessor instances
factory.addBeanPostProcessor(new AutowiredAnnotationBeanPostProcessor());
factory.addBeanPostProcessor(new MyBeanPostProcessor());
```

更多教程请访问:www.flydean.com

```
// now start using the factory
```

要应用BeanFactoryPostProcessor到普通的DefaultListableBeanFactory，你需要调用postProcessBeanFactory， 如下：

```
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，显式注册步骤都不方便，这就是为什么在支持Spring的应用程序中，各种ApplicationContext变量比普通的defaultListableBeanFactory更受欢迎，特别是在典型企业应用中，依赖BeanFactoryPostProcessor和BeanPostProcessor实例实现扩展容器功能时。

> AnnotationConfigApplicationContext注册了所有常见的注解后处理程序，并可以通过配置注解（如@EnableTransactionManagement）将其他处理程序置于覆盖范围内。在Spring基于注解的配置模型的抽象级别上，bean后处理器的概念变成了一个更内部的容器细节。

更多教程请访问:www.flydean.com

2.资源

本章介绍Spring如何处理资源以及如何在Spring中使用资源。

更多教程请访问:www.flydean.com

2.1介绍

Java类的标准java.net.URL类和标准的URL前缀处理器是不足够的，他们没有低级别的资源访问的所有权限。例如，没有标准化的URL，来从classpath或者相关的ServletContext获取资源。

虽然可以为特定的URL注册 handlers ，但是这太复杂了，URL接口也缺少一些非常实用的功能，比如查看资源的指向等。

更多教程请访问:www.flydean.com

2.2资源接口

Spring的资源接口是一个更强大的接口，用于抽象对低级资源的访问。以下列表显示了资源接口定义：

```
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();

}
```

Resource继承了InputStreamSource接口，下面是其定义：

```
public interface InputStreamSource {

    InputStream getInputStream() throws IOException;

}
```

Resource的比较重要的方法如下：

- getInputStream(): 定位和打开resource， 返回InputStream 来读取资源。每一次调用都会返回一个新的InputStream，调用者负责将其关闭。
- exists(): 返回boolean，用来判断资源是否存在
- isOpen(): 返回boolean，用来判断资源是不是已经存在一个open stream处理器。 true表明InputStream不能被多次读取，那么这次的读取会被关闭，以避免资源泄露。false是所有正常资源实现的返回，有可能会抛异常：InputStreamResource。
- getDescription(): 返回资源的描述，用于错误输出。通常这会返回resource URL的全名。

其他的方法可以让你获取到代表resource的URL或者File对象。

Spring本身广泛使用Resource抽象，当需要资源时，它在许多方法签名中作为参数类型。某些Spring API中的其他方法（如各种ApplicationContext实现的构造函数）采用字符串，该字符串以未加修饰或简单的形式用于创建适合该上下文实现的资源，或者通过String路径上的特殊前缀让调用方指定必须创建特定的Resource实现。

尽管在Spring中经常使用Resource接口，但在你自己的代码中单独使用作为通用实用程序类来访问资源实际上非常有用，即使你的代码不知道或不关心Spring的任何其他部分。虽然这将你的代码与Spring结合在一起，但它实际上只将它与这一小组实用程序类结合在一起，这些实用程序类可以作为URL的更强大的替代，并且可以被视为等同于为此目的而使用的任何其他库。

> Resource抽象并没有替代方法，它尽可能的使用包装。比如：UrlResource 包装了URL，然后使用被包装的URL去工作。

更多教程请访问:www.flydean.com

更多教程请访问:www.flydean.com

2.3内置资源实现

Spring有如下几种资源实现：

- UrlResource
- ClassPathResource
- FileSystemResource
- ServletContextResource
- InputStreamResource
- ByteArrayResource

2.3.1. UrlResource

UrlResource封装了java.net.URL，可用于访问通常可通过url访问的任何对象，如文件、HTTP目标、FTP目标和其他对象。所有URL都有一个标准化的字符串表示法，这样就可以使用适当的标准化前缀来表示一个URL类型与另一个URL类型。这包括file：用于访问文件系统路径，HTTP：用于通过HTTP协议访问资源，FTP：用于通过FTP访问资源，以及其他。

URL资源是由Java代码通过显式使用UrlResource构造函数创建的，但通常在调用API方法时隐式创建，该API方法需要表示一个路径的字符串参数。对于后一种情况，JavaBeans属性编辑器最终决定要创建哪种类型的资源。如果路径字符串包含已知（即）前缀（如classpath:），它将为该前缀创建适当的专用资源。但是，如果它不识别前缀，则假定该字符串是标准的URL字符串，并创建一个UrlResource。

2.3.2. ClassPathResource

此类表示应从类路径获取的资源。它使用线程上下文类加载器、给定类加载器或给定类来加载资源。

如果类路径资源驻留在文件系统中，而不是驻留在JAR中且未暴露给filesystem（由servlet引擎或其他环境），则此资源实现支持解析为java.io.File。为了解决这个问题，各种资源实现总是支持将解析作为java.net.URL。

ClassPathResource由Java代码通过显式使用ClassPathResource构造函数创建，但通常在调用API方法时隐式创建，该API方法需要表示一个路径的字符串参数。对于后一种情况，JavaBeans属性编辑器识别字符串路径上的特殊前缀ClassPath:，并在这种情况下创建ClassPathResource。

2.3.3. FileSystemResource

他是java.io.File和java.nio.file.Path的Resource实现，支持解析为File或者URL。

2.3.4. ServletContextResource

这是ServletContext的Resource实现，用于解释相关Web应用程序根目录中的相对路径。

通常来说它支持stream和URL访问，但如果web应用程序被web容器解压，资源存在于真实的文件系统中时，也可以用java.io.File来访问。

是否被解压到文件系统，还是作为JAR包来运行，取决于web容器。

2.3.5. InputStreamResource

更多教程请访问:www.flydean.com

InputStreamResource 是InputStream 的Resource实现。只有在其他Resource实现不可用的时候才考虑使用它。特别的，可以考虑使用ByteArrayResource或者其他的 file-based Resource实现。

和其他的Resource实现相反，它是一个already-opened resource的描述器，所以isOpen()会返回true。 如果你想保存资源描述器或者多次读取一个stream， 那么不要使用它。

2.3.6. ByteArrayResource

是byte array的Resource实现， 它创建了ByteArrayInputStream。

它对于从任何给定的字节数组加载内容都很有用，而不必求助于单次使用的InputStreamResource。

更多教程请访问:www.flydean.com

2.4ResourceLoader

ResourceLoader用来返回Resource实例，下面是其定义：

```
public interface ResourceLoader {

    Resource getResource(String location);

}
```

所有的 application contexts 都实现了ResourceLoader类。因此所有的application contexts 都可以用来获取Resource。

当你在特定的应用程序上下文上调用getResource（），并且指定的位置路径没有特定的前缀时，你将返回适合该特定应用程序上下文的资源类型。例如，假设对ClassPathXmlApplicationContext实例执行了以下代码片段：

```
Resource template = ctx.getResource("some/resource/path/myTemplate.txt");
```

在ClassPathXmlApplicationContext中，这个方法返回ClassPathResource，如果在FileSystemXmlApplicationContext中，方法返回FileSystemResource。 在WebApplicationContext， 方法返回ServletContextResource。 他会返回和ApplicationContext相对应的Resource实现。

因此，你可以以适合特定应用程序上下文的方式加载资源。

当然，你可以强制ClassPathResource使用，而不管ApplicationContext到底是什么。这样做你需要添加classpath:前缀。如下：

```
Resource template = ctx.getResource("classpath:some/resource/path/myTemplate.txt");
```

同样的，你可以强制使用UrlResource通过添加标准的java.net.URL前缀。

```
Resource template = ctx.getResource("file:///some/resource/path/myTemplate.txt");

Resource template = ctx.getResource("https://myhost.com/resource/path/myTemplate.txt");
```

下面是将String转换为Resource的策略：

Prefix	Example	Explanation
classpath:	classpath:com/myapp/config.xml	从classpath加载
file:	file:///data/config.xml	从文件系统以URL方式加载
http:	https://myserver/logo.png	URL方式加载
(none)	/data/config.xml	依赖于ApplicationContext实现。

更多教程请访问:www.flydean.com

2.5ResourceLoaderAware接口

ResourceLoaderAware接口是一个特殊的回调，表明该组件需要提供一个ResourceLoader的引用。 下面是ResourceLoaderAware的定义：

```
public interface ResourceLoaderAware {

    void setResourceLoader(ResourceLoader resourceLoader);
}
```

当一个类实现了ResourceLoaderAware并被部署到application context，那么整个类就被识别为ResourceLoaderAware。 application context会去调用setResourceLoader(ResourceLoader)方法，并将其自身作为参数传入（所有的Spring application contexts 都实现了ResourceLoader 接口）。

因为ApplicationContext也是ResourceLoader，bean也可以实现ApplicationContextAware接口，然后直接使用提供的application context来加载资源。但是通常来说还是使用ResourceLoader来管理资源（代码会被转换为ResourceLoader而不是整个的ApplicationContext）。

在应用程序组件中，你也可以使用自动装载ResourceLoader，来替代使用ResourceLoaderAware接口。可以使用传统的constructor或者byType的自动装载模式。但是使用基于注解的自动装载可操作性更好。 在这种情况下，ResourceLoader 被自动注入到ResourceLoader类型的字段，构造函数参数，或者方法参数， 只要他们使用了@Autowired注解。

更多教程请访问:www.flydean.com

2.6资源作为依赖

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接口来加载资源可能是有意义的。例如，考虑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，那么完全消除对ResourceLoader接口的使用是有意义的，让bean公开它所需的Resource属性，并期望将它们注入其中。

然后注入这些属性变得很简单，因为所有应用程序上下文都注册并使用一个特殊的JavaBeans PropertyEditor，它可以将String路径转换为Resource对象。因此，如果mybean具有类型Resource的template属性，那么可以为该资源配置一个简单的字符串，如下例所示：

```
<bean id="myBean" class="...">
    <property name="template" value="some/resource/path/myTemplate.txt"/>
</bean>
```

注意resource path没有前缀。因为application context 本身就是ResourceLoader， resource 将会根据context的类型不同去加载ClassPathResource ，FileSystemResource 或者ServletContextResource。

如果你想强制使用某一类型的Resource，那么可以加上前缀。下面是强制使用ClassPathResource 和UrlResource 的例子：

```
<property name="template" value="classpath:some/resource/path/myTemplate.txt">

<property name="template" value="file:///some/resource/path/myTemplate.txt"/>
```

更多教程请访问:www.flydean.com

2.7应用程序上下文和资源路径

本节介绍如何使用资源创建应用程序上下文，包括使用XML的快捷方式、如何使用通配符以及其他详细信息。

2.7.1 构造Application Contexts

application context constructor（特定context类型）通常使用string或者string数组作为resources路径的地址，比如定义context 的XML地址。

如果该地址没有前缀，那么该定义bean定义的Resource将会依赖于和他最相近的application context.比如下面的例子会使用ClassPathXmlApplicationContext：

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("conf/appContext.xml");
```

这个bean定义将会从classpath加载，因为ClassPathResource被使用。但是下面的例子将会创建FileSystemXmlApplicationContext：

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("conf/appContext.xml");
```

现在bean定义从文件系统加载。

如果加上前缀，那么将会覆盖默认的Resource类型。如下：

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("classpath:conf/appContext.xml");
```

这里虽然使用了FileSystemXmlApplicationContext，但是还是会从classpath中加载。但是如果在后面的代码中，还有需要使用到ResourceLoader的， 那么没有前缀的paths仍然会被认为是filesystem路径。

构造ClassPathXmlApplicationContext-快捷方式

ClassPathXmlApplicationContext公开了许多构造函数，以方便实例化。基本思想是，你只需提供一个字符串数组，该数组只包含XML文件本身的文件名（不包含前导路径信息），还提供一个类。然后，ClassPathXmlApplicationContext从提供的类中派生路径信息。

考虑下面的文件结构：

```
com/
  foo/
    services.xml
    daos.xml
    MessengerService.class
```

下面的示例演示如何实例化由名为services.xml和daos.xml（位于类路径上）文件中定义的bean组成的ClassPathXmlApplicationContext实例：

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

2.7.2 应用程序上下文构造函数资源路径中的通配符

更多教程请访问:www.flydean.com

应用程序上下文构造函数值中的资源路径可以是简单路径（如前所示），每个路径都有到目标资源的一对一映射，或者可以包含特殊的“classpath*：”前缀或内部ant样式正则表达式（通过使用Spring的PathMatcher实用程序）。后者都是有效的通配符。

此机制的一个用途是在需要进行组件样式的应用程序组装时使用。所有组件都可以将上下文定义片段“发布”到一个众所周知的位置路径，并且，当使用前缀为classpath*：的同一路径创建最终应用程序上下文时，所有组件片段都会自动拾取。

注意，这个通配符是特定于应用程序上下文构造函数中资源路径的使用（或者直接使用PathMatcher实用程序类层次结构时），并且在构造时解析。它与资源类型本身无关。不能使用classpath*：前缀构造实际资源，因为资源一次只指向一个资源。

Ant-style Patterns

下面是 Ant-style patterns 的路径：

```
/WEB-INF/*-context.xml
com/mycompany/**/applicationContext.xml
file:C:/some/path/*-context.xml
classpath:com/mycompany/**/applicationContext.xml
```

当路径位置包含一个Ant样式的模式时，解析器将遵循一个更复杂的过程来尝试解析通配符。它为最后一个非通配符段的路径生成一个资源，并从中获取一个URL。如果该URL不是jar:URL或容器特定的变量（例如，weblogic中的zip、websphere中的wsjar等），则从该URL获取java.io.File，并通过遍历文件系统来解析通配符。对于JAR URL，解析器要么从中获取java.net.JarURLConnection，要么手动解析JAR URL，然后遍历JAR文件的内容以解析通配符。

对可移植性的影响

如果指定的路径已经是一个文件URL（或者隐式地因为ResourceLoader是一个文件系统，或者显式地指定），则通配符是完全可移植的。

如果指定的路径是类路径位置，解析程序必须通过调用ClassLoader.getResource（）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（而不是文件的末尾），所以实际上（在类加载器JavaDoc中）没有定义在这种情况下返回的具体URL类型。实际上，它始终是一个java.io.File，表示目录（类路径资源解析为文件系统位置）或某种类型的JAR URL（类路径资源解析为JAR位置）。尽管如此，这个操作还是存在可移植性问题。

如果为最后一个非通配符段获取JAR URL，解析程序必须能够从中获取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。这在大多数环境中都有效，但在其他环境中失败，我们强烈建议在依赖JAR之前，在特定环境中彻底测试来自JAR的资源的通配符解析。

classpath*:前缀

构造基于XML的application context，路径地址可以使用classpath*: 前缀，如下：

```
ApplicationContext ctx =
    new ClassPathXmlApplicationContext("classpath*:conf/appContext.xml");
```

此特殊前缀指定必须获取与给定名称匹配的所有类路径资源（在内部，这基本上是通过调用ClassLoader.getResources（...）实现的），然后合并以形成最终的应用程序上下文定义。

> 通配符类路径依赖于基础类加载器的getResources（）方法。由于现在大多数应用程序服务器都提供自己的类加载器实现，因此其行为可能会有所不同，尤其是在处理JAR文件时。检查ClassPath*是否有效的一个简单测试是使用ClassLoader从类路径上的jar中加载文件：getClass().getClassLoader().getResources("\")。尝试使用具有相同名称但位于两个不同位置的文件进行此测试。如果返回不适当的结果，请检查应用程序服务器文档中可能影响类加载器行为的设置。

更多教程请访问:www.flydean.com

你还可以在位置路径的其余部分（例如，classpath:*META-INF/*-beans.xml）中将classpath*：前缀与PathMatcher模式组合。在这种情况下，解析策略相当简单：在最后一个非通配符路径段上使用ClassLoader.getResources()调用以获取类加载程序层次结构中的所有匹配资源，然后在每个资源上，使用前面描述的相同路径匹配器解析策略通配符子路径。

通配符的其他关注点

注意，当与Ant样式模式结合使用时，除非实际目标文件位于文件系统中，否则classpath: 在模式开始之前只能在至少有一个根目录的情况下可靠的执行。这意味着*classpath*:*.xml等模式可能不会从jar文件的根目录中检索文件，而是只从扩展目录的根目录中检索文件。

Spring检索类路径条目的能力源自JDK的 ClassLoader.getResources()）方法，该方法只返回空字符串的文件系统位置（指示要搜索的潜在根）。Spring也评估了JAR文件中的URLClassLoader运行时配置和java.class.path清单，但这并不保证会导致可移植行为。

> 扫描classpath包需要在classpath中存在相应的目录。当你使用Ant构建JAR时，不要激活JAR任务的“仅文件”开关。此外，根据某些环境中的安全策略，类路径目录可能不会被公开-例如，JDK 1.7.0 U45及更高版本上的独立应用程序（需要在清单中设置“可信库”）。请参阅https://stackoverflow.com/questions/19394570/java-jre-7u45-breaks-classloader-getresources）。在JDK9的模块路径（Jigsaw）上，Spring的类路径扫描通常按预期工作。在这里，将资源放入一个专用的目录也是非常推荐的，这样可以避免前面提到的在搜索JAR文件根级别时的可移植性问题。

Ant-style 模式classpath: resources 并不一定能保证匹配的资源，如果要搜索的根package存在于多个类路径地址中。考虑下面的资源情况：

```
com/mycompany/package1/service-context.xml
```

下面是可能用到的Ant-style path 匹配：

```
classpath:com/mycompany/**/service-context.xml
```

这样的资源只能位于一个位置，但是当使用前面的示例这样的路径来尝试解析它时，解析程序将使用getResource（“com/mycompany”）；返回的（第一个）URL。如果此基本包节点存在于多个类加载器位置中，则实际的最终资源可能找不到。因此，在这种情况下，你应该更喜欢使用具有相同ant样式模式的classpath*:来搜索包含根包的所有类路径位置。

2.7.3 FileSystemResource注意事项

未连接到FileSystemApplicationContext的FileSystemResource（即，当FileSystemApplicationContext不是实际的ResourceLoader时）会按预期处理绝对和相对路径。相对路径相对于当前工作目录，而绝对路径相对于文件系统的根目录。

但是，由于向后兼容性（历史）的原因，当FileSystemApplicationContext是ResourceLoader时，这一点会发生变化。FileSystemApplicationContext强制所有附加的FileSystemResource实例将所有位置路径视为相对路径，不管它们是否以前导斜杠开头。实际上，这意味着以下示例是等效的：

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("conf/context.xml");

ApplicationContext ctx =
    new FileSystemXmlApplicationContext("/conf/context.xml");
```

以下例子也是等效的（即使它们是不同的，因为一种情况是相对的，另一种是绝对的）：

```
FileSystemXmlApplicationContext ctx = ...;
```

更多教程请访问:www.flydean.com

```
ctx.getResource("some/resource/path/myTemplate.txt");


FileSystemXmlApplicationContext ctx = ...;
ctx.getResource("/some/resource/path/myTemplate.txt");
```

在实践中，如果需要真正的绝对文件系统路径，则应避免将绝对路径与FileSystemResource或FileSystemXmlApplicationContext一起使用，并通过使用file: URL 前缀强制使用UrlResource。以下示例说明了如何执行此操作：

```
// actual context type doesn't matter, the Resource will always be UrlResource
ctx.getResource("file:///some/resource/path/myTemplate.txt");

// force this FileSystemXmlApplicationContext to load its definition via a UrlResource
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("file:///conf/context.xml");
```

更多教程请访问:www.flydean.com

3.验证，数据绑定，和类型转换

将验证视为业务逻辑是有利弊的，Spring提供了一种验证（和数据绑定）设计，不排除其中任何一种验证。具体来说，验证不应该绑定到Web层，应该易于本地化，并且应该可以插入任何可用的验证程序。考虑到这些问题，Spring提出了一个Validator接口，它既基础又在应用程序的每一层都非常可用。

数据绑定对于让用户输入动态绑定到应用程序的域模型（或用于处理用户输入的任何对象）非常有用。Spring提供了恰当命名的DataBinder来实现这一点。Validator和DataBinder组成validation包，该包主要用于但不限于MVC框架。

BeanWrapper是Spring框架中的一个基本概念，在很多地方都有使用。但是，你可能不需要直接使用BeanWrapper。然而，由于这是参考指南，我们还是要按规则做一些说明。我们在本章中解释BeanWrapper，因为如果你要使用它，那么在尝试将数据绑定到对象时，你很可能会这样做。

Spring的DataBinder和较低级别的BeanWrapper都使用PropertyEditorsSupport实现来解析和格式化属性值。PropertyEditor和PropertyEditorSupport类型是JavaBeans规范的一部分，本章也将对此进行解释。Spring3引入了一个core.convert包，它提供了一个通用的类型转换工具，以及一个更高级的“格式”包，用于格式化UI字段值。你可以使用这些包作为PropertyEditorSupport实现的简单替代方案。本章还将讨论这些问题。

> JSR-303/JSR-349 Bean Validation
>
> 从4.0版开始，Spring框架支持Bean Validation 1.0 (JSR-303) 和Bean Validation 1.1（JSR-349），以支持安装并使其适应Spring的Validator接口。
>
> 应用程序可以选择全局启用一次bean验证，如Spring验证中所述，并专门用于所有验证需求。
>
> 应用程序还可以为每个DataBinder实例注册额外的Spring Validator实例，如配置DataBinder中所述。这对于在不使用注解的情况下插入验证逻辑可能很有用。

更多教程请访问:www.flydean.com

3.1使用Spring Validator接口

Spring提供了一个Validator接口，你可以使用它来验证对象。Validator接口通过使用Errors对象工作，以便在验证时，验证程序可以向Errors对象报告验证失败。

考虑下面的小数据对象示例：

```
public class Person {

    private String name;
    private int age;

    // the usual getters and setters...
}
```

下面例子提供了Person的验证方式，通过实现org.springframework.validation.Validator接口的下面两个方法：

- supports(Class): 该Class是否支持Validator
- validate(Object, org.springframework.validation.Errors): 验证给定的对象，如果验证失败，将错误注册到Errors对象。

实现Validator很简单，尤其当你知道Spring框架提供的ValidationUtils帮助类，下面是Person Validator的例子：

```
public class PersonValidator implements Validator {

    /**
     * This Validator validates *only* Person instances
     */
    public boolean supports(Class clazz) {
        return Person.class.equals(clazz);
    }

    public void validate(Object obj, Errors e) {
        ValidationUtils.rejectIfEmpty(e, "name", "name.empty");
        Person p = (Person) obj;
        if (p.getAge() < 0) {
            e.rejectValue("age", "negativevalue");
        } else if (p.getAge() > 110) {
            e.rejectValue("age", "too.darn.old");
        }
    }
}
```

ValidationUtils类上的static rejectIfEmpty(..) 方法用于拒绝name属性（如果该属性为空或空字符串）。查看ValidationUtils javadoc，了解它除了前面的示例外还提供了哪些功能。

虽然可以实现单个验证器类来验证富对象中的每个嵌套对象，但最好将对象的每个嵌套类的验证逻辑封装在自己的验证器实现中。“rich”对象的一个简单示例是由两个字符串属性（名字和第二个名字）和一个复杂地址对象组成的Customer。地址对象可以独立于客户对象使用，因此实现了一个不同的地址验证器。如果希望CustomerValidator重用AddressValidator类中包含的逻辑而不使用复制和粘贴，则可以在CustomerValidator中插入或实例化一个AddressValidator，如下示例所示：

```
public class CustomerValidator implements Validator {

    private final Validator addressValidator;

    public CustomerValidator(Validator addressValidator) {
        if (addressValidator == null) {
            throw new IllegalArgumentException("The supplied [Validator] is " +
```

更多教程请访问:www.flydean.com

```
                "required and must not be null.");
        }
        if (!addressValidator.supports(Address.class)) {
            throw new IllegalArgumentException("The supplied [Validator] must " +
                "support the validation of [Address] instances.");
        }
        this.addressValidator = addressValidator;
    }

    /**
     * This Validator validates Customer instances, and any subclasses of Customer too
     */
    public boolean supports(Class clazz) {
        return Customer.class.isAssignableFrom(clazz);
    }

    public void validate(Object target, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName", "field.required");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "surname", "field.required");
        Customer customer = (Customer) target;
        try {
            errors.pushNestedPath("address");
            ValidationUtils.invokeValidator(this.addressValidator, customer.getAddress(), errors);
        } finally {
            errors.popNestedPath();
        }
    }
}
```

验证错误将报告给传递给验证程序的错误对象。对于SpringWeb MVC，可以使用\标记检查错误消息，但也可以自己检查Errors对象。关于它提供的方法的更多信息可以在javadoc中找到。

更多教程请访问:www.flydean.com

3.2将代码解析为错误消息

我们讨论了数据绑定和验证。本节介绍与验证错误相对应的输出消息。在上一节所示的示例中，我们拒绝了名称和年龄字段。如果我们想使用MessageSource输出错误消息，我们可以使用拒绝字段时提供的错误代码（在本例中为“name”和“age”）。当你使用（例如，ValidationUtils类）rejectValue或Errors接口中的其他reject方法之一调用（直接或间接）时，基础实现不仅注册你传入的代码，而且还注册许多其他错误代码。MessageCodesResolver确定Errors接口寄存器的错误代码。默认情况下，将使用DefaultMessageCodesResolver，它（例如）不仅用你提供的代码注册消息，还注册包含你传递给reject方法的字段名的消息。因此，如果使用rejectValue（“age”，“too.darn.old”）拒绝字段，除了too.darn.old代码，spring还注册too.darn.old.age和too.darn.old.age.int（第一个包括字段名，第二个包括字段类型）。这样做是为了帮助开发人员确定错误消息的目标。

有关MessageCodesResolver和默认策略的详细信息，请分别在MessageCodesResolver和DefaultMessageCodesResolver的javadoc中找到。

更多教程请访问:www.flydean.com

3.3bean操作和BeanWrapper

org.springframework.beans 包遵循JavaBeans标准。一个JavaBeans类拥有默认的无参构造函数，和下面的命名约定，例如：一个命名为bingoMadness的属性，应该有一个setter方法：setBingoMadness(..) 和一个getter方法：getBingoMadness()。

beans 包里面有个很重要的类叫做BeanWrapper接口和他的实现BeanWrapperImpl。正如JavaDoc所引用的，BeanWrapper提供了设置和获取属性值（单个或批量）、获取属性描述符和查询属性的功能，以确定它们是可读的还是可写的。此外，BeanWrapper还支持嵌套属性，允许将子属性的属性设置为无限深度。BeanWrapper还支持添加标准JavaBeans属性PropertyChangeListeners和VetoableChangeListeners，而不需要在目标类中支持代码。最后，BeanWrapper提供了对设置索引属性的支持。BeanWrapper通常不直接由应用程序代码使用，而是由DataBinder和BeanFactory使用。

3.3.1 设置和获取基本的和内嵌的属性

设置和获取属性可以通过setPropertyValue，setPropertyValues，getPropertyValue和getPropertyValues方法，和一系列的可覆盖的变种。Springs 的javadoc 有详细描述。 JavaBeans规范对标记对象的属性有专门的约定：

Expression	Explanation
name	指示属性name相对于的getName() ， isName() 和 setName(..)方法
account.name	指示account内部的name属性，对应getAccount().setName() 和 getAccount().getName()方法。
account[2]	指示account的第三个元素，索引属性可以是array, list或者其他的自然排序的集合。
account[COMPANYNAME]	一个map，key是COMPANYNAMEs

（如果你不打算直接使用BeanWrapper，那么下一节对你来说并不重要。如果只使用DataBinder和BeanFactory及其默认实现，则应跳到PropertyEditors部分。）

下面的例子使用BeanWrapper来get和set属性：

```
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;
    }
}
```

更多教程请访问:www.flydean.com

```
public class Employee {

    private String name;

    private float salary;

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public float getSalary() {
        return salary;
    }

    public void setSalary(float salary) {
        this.salary = salary;
    }
}
```

下面的例子展示了怎么实例化和使用Companies， Employees：

```
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");
```

3.3.2 内置的PropertyEditor实现

Spring使用PropertyEditor的概念来实现Object和String之间的转换。用不同于对象本身的方式表示属性很方便。例如，日期可以以人类可读的方式表示（字符串：“2007-14-09”），而我们仍然可以将人类可读的形式转换回原始日期（或者，更好的方法是，将以人类可读的形式输入的任何日期转换回日期对象）。此行为可以通过注册java.beans.PropertyEditor类型的自定义编辑器来实现。在BeanWrapper上注册自定义编辑器，或者在特定的ioc容器（如前一章所述）中注册自定义编辑器，可以使它了解如何将属性转换为所需的类型。有关PropertyEditor的更多信息，请参阅Oracle中java.beans包的javadoc。

在Spring中使用属性编辑的几个示例：

- 通过使用PropertyEditor实现来设置bean的属性。当使用String作为在XML文件中声明的某个bean的属性值时，Spring（如果相应属性的setter有类参数）则使用ClassEditor尝试将参数解析为Class对象。
- 在Spring的MVC框架中解析HTTP请求参数是通过使用各种属性编辑器实现来完成的，这些实现可以手动绑定到CommandController的所有子类中。

Spring有内置的方便使用的PropertyEditor实现，他们都在org.springframework.beans.propertyeditors包里面，大多数都是通过BeanWrapperImpl来注册的。虽然这些property editor 在某些方面是可配置的，你也可以注册自定义的property editor来覆盖默认配置。下表列出了Spring提供的多种PropertyEditor：

更多教程请访问:www.flydean.com

Class	Explanation
ByteArrayPropertyEditor	字节数组编辑器。将字符串转换为相应的字节表示形式。默认情况下由BeanWrapperImpl注册。
ClassEditor	将表示类的字符串解析为实际类，反之亦然。当找不到类时，将引发IllegalArgumentException。默认情况下，由BeanWrapperImpl注册。
CustomBooleanEditor	布尔属性的可自定义属性编辑器。默认情况下，由BeanWrapperImpl注册，但可以通过将其自定义实例注册为自定义编辑器来重写。
CustomCollectionEditor	集合的属性编辑器，将任何源集合转换为给定的目标集合类型。
CustomDateEditor	java.util.date的可自定义属性编辑器，支持自定义日期格式。默认情况下未注册。必须根据需要使用适当的格式进行用户注册。
CustomNumberEditor	任何数字子类（如integer、long、float或double）的可自定义属性编辑器。默认情况下，由BeanWrapperImpl注册，但可以通过将其自定义实例注册为自定义编辑器来重写。
FileEditor	将String解析成为java.io.File对象。默认由BeanWrapperImpl来注册。
InputStreamEditor	单向属性编辑器，它可以获取一个字符串并（通过中间的ResourceEditor和Resource）生成一个InputStream，以便InputStream属性可以直接设置为字符串。请注意，默认用法不会为你关闭inputstream。默认情况下，由BeanWrapperImpl注册。
LocaleEditor	可以将字符串解析为区域设置对象，反之亦然（字符串格式为[country][variant]，与区域设置的toString（）方法相同）。默认情况下，由BeanWrapperImpl注册。
PatternEditor	可以将字符串解析为java.util.regex.Pattern对象，反之亦然。
PropertiesEditor	可以将字符串（使用java.util.Properties类的javadoc中定义的格式格式化）转换为属性对象。默认情况下，由BeanWrapperImpl注册。
StringTrimmerEditor	修剪字符串的属性编辑器。（可选）允许将空字符串转换为空值。默认情况下未注册-必须是用户注册的。
URLEditor	可以将URL的字符串表示形式解析为实际的URL对象。默认情况下，由BeanWrapperImpl注册。

Spring使用java.beans.PropertyEditorManager为可能需要的属性编辑器设置搜索路径。搜索路径还包括sun.bean.editors，其中包括字体、颜色和大多数基本类型等类型的PropertyEditor实现。还要注意，如果标准JavaBeans基础结构与它们处理的类位于同一个包中，并且与该类具有相同的名称，并且附加了Editor，那么标准JavaBeans基础结构会自动发现PropertyEditor类（无需显式注册）。例如，可以具有以下类和包结构，这足以使SomethingEditor类被识别并用作某个类型化属性的属性编辑器。

```
com
  chank
    pop
      Something
      SomethingEditor // the PropertyEditor for the Something class
```

注意，你也可以使用标准的BeanInfo JavaBeans机制。下面的示例使用BeanInfo机制显式地用关联类的属性注册一个或多个PropertyEditor实例：

```
com
  chank
    pop
      Something
      SomethingBeanInfo // the BeanInfo for the Something class
```

以下引用的SomethingBeanInfo类的Java源代码将CustomNumberEditor与Something类的age属性关联起来：

更多教程请访问:www.flydean.com

```
public class SomethingBeanInfo extends SimpleBeanInfo {

    public PropertyDescriptor[] getPropertyDescriptors() {
        try {
            final PropertyEditor numberPE = new CustomNumberEditor(Integer.class, true);
            PropertyDescriptor ageDescriptor = new PropertyDescriptor("age", Something.class) {
                public PropertyEditor createPropertyEditor(Object bean) {
                    return numberPE;
                };
            };
            return new PropertyDescriptor[] { ageDescriptor };
        }
        catch (IntrospectionException ex) {
            throw new Error(ex.toString());
        }
    }
}
```

注册自定义PropertyEditor实现

当将bean属性设置为字符串值时，SpringIOC容器最终使用标准JavaBeans PropertyEditor实现将这些字符串转换为复杂的属性类型。Spring预先注册了许多自定义的PropertyEditor实现（例如，将用字符串表示的类名转换为类对象）。此外，Java的标准JavaBeans PropertyEditor查找机制允许一个类的PropertyEditor被适当地命名，并放置在与它提供支持的类相同的包中，以便可以自动找到它。

如果需要注册其他自定义PropertyEditors，可以使用几种机制。通常不方便或不推荐使用的最手动的方法是使用ConfigurableBeanFactory接口的registerCustomEditor（）方法，前提是你有BeanFactory引用。

另一种（稍微方便一点）机制是使用一个特殊的bean工厂后处理器，称为CustomEditorConfigurer。尽管你可以将bean工厂后处理器与BeanFactory实现结合使用，但CustomEditorConfigurer有一个嵌套的属性设置，因此我们强烈建议你将其与ApplicationContext结合使用，在这里你可以以类似于任何其他bean的方式部署它，并且在那里它可以自动化。检测并应用。

注意，所有bean工厂和应用程序上下文都自动使用许多内置的属性编辑器，通过使用BeanWrapper来处理属性转换。BeanWrapper注册器的标准属性编辑器在前一节中列出。此外，ApplicationContexts还重写或添加其他编辑器，以便以适合特定应用程序上下文类型的方式处理资源查找。

标准JavaBeans PropertyEditor实例用于将表示为字符串的属性值转换为属性的实际复杂类型。你可以使用bean工厂后处理器CustomEditorConfigurer，方便地向应用程序上下文添加对其他PropertyEditor实例的支持。

考虑以下示例，它定义了一个名为ExoticType的用户类和另一个名为DependsOneXoticType的类，该类需要将ExoticType设置为属性：

```
package example;

public class ExoticType {

    private String name;

    public ExoticType(String name) {
        this.name = name;
    }
}

public class DependsOnExoticType {

    private ExoticType type;

    public void setType(ExoticType type) {
        this.type = type;
    }
}
```

更多教程请访问:www.flydean.com

正确设置之后，我们希望能够将类型属性指定为字符串，然后由属性编辑器将其转换为实际的ExoticType实例。下面的bean定义显示了如何设置此关系：

```
<bean id="sample" class="example.DependsOnExoticType">
    <property name="type" value="aNameForExoticType"/>
</bean>
```

PropertyEditor实现如下所示：

```
// converts string representation to ExoticType object
package example;

public class ExoticTypeEditor extends PropertyEditorSupport {

    public void setAsText(String text) {
        setValue(new ExoticType(text.toUpperCase()));
    }
}
```

下面例子展示了怎么使用CustomEditorConfigurer将新创建的PropertyEditor注册到ApplicationContext：

```
<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="customEditors">
        <map>
            <entry key="example.ExoticType" value="example.ExoticTypeEditor"/>
        </map>
    </property>
</bean>
```

使用PropertyEditorRegistrar

另一种机制来注册属性编辑器到Spring容器的方法就是使用PropertyEditorRegistrar。

当你需要在几个不同的情况下使用同一组属性编辑器时，此接口特别有用。你可以编写相应的注册器，并在每种情况下重用它。PropertyEditorRegistrar实例与名为PropertyEditorRegistry的接口一起工作，后者是由SpringBeanWrapper（和DataBinder）实现的接口。当与CustomEditorConfigurer（此处描述）结合使用时，PropertyEditorRegistrar实例特别方便，后者公开了一个名为setPropertyEditorRegistrars（..）的属性。以这种方式添加到CustomEditorConfigurer的PropertyEditorRegistrar实例可以很容易地与DataBinder和SpringMVC控制器共享。此外，它还避免了在自定义编辑器上同步的需要：一个PropertyEditorRegistrar应该为每个bean创建尝试创建新的PropertyEditor实例。

下面例子展示了如何创建你自己的PropertyEditorRegistrar：

```
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...
    }
}
```

org.springframework.beans.support.ResourceEditorRegistrar也是PropertyEditorRegistrar实现的一个例子，可以参照。注意他是怎么实现registerCustomEditors（）方法的。

更多教程请访问:www.flydean.com

下面例子展示了怎么配置CustomEditorConfigurer，并将CustomPropertyEditorRegistrar的实例注入其中：

```
<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="propertyEditorRegistrars">
        <list>
            <ref bean="customPropertyEditorRegistrar"/>
        </list>
    </property>
</bean>

<bean id="customPropertyEditorRegistrar"
    class="com.foo.editors.spring.CustomPropertyEditorRegistrar"/>
```

最后（对于使用Spring的MVC Web框架的用户来说，稍微偏离本章的重点），将PropertyEditorRegistrars与数据绑定控制器（如SimpleFormController）结合使用非常方便。以下示例在initBinder（..）方法的实现中使用了PropertyEditorRegistrar：

```
public final class RegisterUserController extends SimpleFormController {

    private final PropertyEditorRegistrar customPropertyEditorRegistrar;

    public RegisterUserController(PropertyEditorRegistrar propertyEditorRegistrar) {
        this.customPropertyEditorRegistrar = propertyEditorRegistrar;
    }

    protected void initBinder(HttpServletRequest request,
            ServletRequestDataBinder binder) throws Exception {
        this.customPropertyEditorRegistrar.registerCustomEditors(binder);
    }

    // other methods to do with registering a User
}
```

这种类型的PropertyEditor注册可以产生简洁的代码（initBinder（..）的实现只有一行长），并允许将公共的PropertyEditor注册代码封装在一个类中，然后根据需要在多个Controllers之间共享。

更多教程请访问:www.flydean.com

3.4Spring类型转换

Spring3 引入了一个core.convert包，提供通用的type转换方法。系统定义了SPI来实现类型转换逻辑和一个API来在运行时实现类型转换。

在Spring容器中，可以使用此系统作为PropertyEditor实现的替代方法，将外部化的bean属性值字符串转换为所需的属性类型。你还可以在应用程序中需要类型转换的任何地方使用公共API。

3.4.1 转换SPI

实现类型转换逻辑的SPI是简单且强类型的，如下接口定义所示：

```
package org.springframework.core.convert.converter;

public interface Converter<S, T> {

    T convert(S source);
}
```

要创建你自己的converter，只需要实现Converter接口，其中S表示要转换的类型，T表示要转换到的类型。

如果需要将S的集合或数组转换为T的数组或集合，也可以透明地应用此类转换器，前提是已注册了委托数组或集合转换器（默认情况下，DefaultConversionService会这样做）。

对于每次调用convert(S)， S默认是非空的。如果S为空，Converter则会抛出unchecked 异常，具体来说，它会抛出IllegalArgumentException异常，确保你的Converter实现是线程安全的。

为了方便起见，core.convert.support包中提供了几个转换器实现。其中包括从字符串到数字的转换器和其他常见类型。下面的列表显示了StringToInteger类，这是一个典型的Converter实现：

```
package org.springframework.core.convert.support;

final class StringToInteger implements Converter<String, Integer> {

    public Integer convert(String source) {
        return Integer.valueOf(source);
    }
}
```

3.4.2 ConverterFactory

当需要集中管理整个类层次结构的转换逻辑时（例如，从字符串转换为枚举对象时），可以实现ConverterFactory，如下例所示：

```
package org.springframework.core.convert.converter;

public interface ConverterFactory<S, R> {

    <T extends R> Converter<S, T> getConverter(Class<T> targetType);
}
```

参数S是需要转换的类型，R是要转换到的类的基类。然后实现getConverter(Class)方法，其中T是R的子类。

考虑StringToEnumConverterFactory的例子：

更多教程请访问:www.flydean.com

```
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());
        }
    }
}
```

3.4.3 使用GenericConverter

当需要复杂的Converter实现时，请考虑使用GenericConverter接口。与Converter相比，GenericConverter具有更灵活但不太强类型的签名，它支持在多个源类型和目标类型之间进行转换。此外，GenericConverter提供了可用的源和目标字段上下文，你可以在实现转换逻辑时使用它们。这样的上下文允许类型转换由字段注解或字段签名上声明的一般信息驱动。下面的列表显示了GenericConverter的接口定义：

```
package org.springframework.core.convert.converter;

public interface GenericConverter {

    public Set<ConvertiblePair> getConvertibleTypes();

    Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType);
}
```

要实现GenericConverter，只需要getConvertibleTypes()返回支持的source→targe， 然后在convert(Object, TypeDescriptor, TypeDescriptor)实现具体的业务。sourceType提供了要被转换的源字段的访问，targetType提供了要转换到的目标字段的访问。

GenericConverter的一个很好的例子是在Java数组和集合之间转换的转换器。例如ArrayToCollectionConverter自省声明目标集合类型的字段以解析集合的元素类型。这样，在目标字段上设置集合之前，源数组中的每个元素都可以转换为集合元素类型。

> 因为GenericConverter是一个更复杂的SPI接口，只在需要的使用它。你可以使用Converter或者ConverterFactory来进行基本的类型转换。

使用ConditionalGenericConverter

有时，你只希望在特定条件成立时运行Converter。例如，你可能只希望在目标字段上存在特定批注时运行Converter，也可能只希望在目标类上定义特定方法（如静态valueof方法）时运行Converter。ConditionalGenericConverter是GenericConverter和ConditionalConverter接口的联合，可用于定义此类自定义匹配条件：

```
public interface ConditionalConverter {

    boolean matches(TypeDescriptor sourceType, TypeDescriptor targetType);
}
```

更多教程请访问:www.flydean.com

```
public interface ConditionalGenericConverter extends GenericConverter, ConditionalConverter {
}
```

一个ConditionalGenericConverter的很好的例子是EntityConverter，它在持久实体标识符和实体引用之间进行转换。只有当目标实体类型声明静态finder方法（例如findAccount（long））时，这样的EntityConverter才可能匹配。你可以在匹配项（typescriptor、typescriptor）的实现中执行这样的finder方法检查。

3.4.4 ConversionService API

ConversionService定义了一个统一的API，用于在运行时执行类型转换逻辑。转换器通常在下面的facade接口后面执行：

```
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);

}
```

大多数ConversionService的实现同时也实现了ConverterRegistry，它提供了一个SPI用来注册转换器。在内部，ConversionService实现委托其注册的转换器来执行类型转换逻辑。

core.convert.support包中提供了一个健壮的conversionService实现。GenericConversionService是适用于大多数环境的通用实现。ConversionServiceFactory为创建通用ConversionService配置提供了方便的工厂。

3.4.5 配置ConversionService

ConversionService是一个无状态对象，设计为在应用程序启动时实例化，然后在多个线程之间共享。在Spring应用程序中，通常为每个Spring容器（或ApplicationContext）配置ConversionService实例。Spring接收转换服务，并在框架需要执行类型转换时使用它。你还可以将此ConversionService注入到任何bean中，并直接调用它。

> 如果没有ConversionService注册到Spring，那么原始的PropertyEditor将会被使用。

要注册一个默认的ConversionService到Spring，添加下面的bean定义，并将id命名为conversionService。

```
<bean id="conversionService"
    class="org.springframework.context.support.ConversionServiceFactoryBean"/>
```

一个默认的ConversionService可以转换trings, numbers, enums, collections, maps, 和其他通用类型. 要用自己的自定义转换器补充或重写默认转换器，请设置Converters属性。属性值可以实现任何Converter、ConverterFactory或GenericConverter接口。

```
<bean id="conversionService"
        class="org.springframework.context.support.ConversionServiceFactoryBean">
    <property name="converters">
        <set>
            <bean class="example.MyCustomConverter"/>
        </set>
```

更多教程请访问:www.flydean.com

```
    </property>
</bean>
```

通常ConversionService也被使用在Spring MVC中。

在特定的环境中，你可能希望在转换中使用格式化，可以参照FormattingConversionServiceFactoryBean的FormatterRegistry SPI。

3.4.6 编程方式使用ConversionService

要编程的使用ConversionService，你可以像注入其他Bean一样将其注入，如下所示：

```
@Service
public class MyService {

    @Autowired
    public MyService(ConversionService conversionService) {
        this.conversionService = conversionService;
    }

    public void doIt() {
        this.conversionService.convert(...)
    }
}
```

对于大多数用例，可以使用指定TargetType的Convert方法，但它不适用于更复杂的类型，例如参数化元素的集合。例如，如果要以编程方式将整数列表转换为字符串列表，则需要提供源类型和目标类型的正式定义。

幸运的是，TypeDescriptor提供了各种选项，使操作变得简单易懂，如下面的示例所示：

```
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)));
```

注意DefaultConversionService将会自动注册适用于大多数环境的转换器。 这包括collection converters、scalar converters和基本Object-to-String converters。使用DefaultConversionService的静态addDefaultConverters方法，可以将同一个converters注册到任何ConverterRegistry。

value types Converters被用于arrays和collections，因此无需创建一个特定的converter来从一个 Collection S到一个T Collection，假设标准的Collection处理是适当的。

更多教程请访问:www.flydean.com

3.5Spring字段格式化

如前一节所述，core.convert是一个通用类型转换系统。它提供了一个统一的ConversionServiceAPI以及一个强类型转换器SPI，用于实现从一种类型到另一种类型的转换逻辑。Spring容器使用此系统绑定bean属性值。此外，Spring表达式语言（SPEL）和DataBinder都使用此系统绑定字段值。例如，当SPEL需要将short强制为long以完成expression.setValue（object bean，object value）尝试时，core.convert系统将执行强制。

现在考虑典型客户机环境（如Web或桌面应用程序）的类型转换要求。在这种环境中，你通常从字符串转换为支持客户端回发过程，以及从字符串转换为支持视图呈现过程。此外，你通常需要本地化字符串值。更通用的core.convert转换器SPI不能直接满足这种格式要求。为了直接解决这些问题，Spring3引入了一个方便的格式化程序SPI，它为客户机环境提供了一个简单而健壮的PropertyEditor实现替代方案。

通常，当需要实现通用类型转换逻辑时，可以使用Converter SPI，例如，在java.util.Date和Long之间转换。当你在客户端环境（如Web应用程序）中工作并且需要分析和打印本地化字段值时，可以使用Formatter SPI。ConversionService为两个SPI提供统一的类型转换API。

3.5.1 Formatter SPI

用于实现字段格式化逻辑的Formatter SPI是简单且强类型的。以下列表显示Formatter接口定义：

```
package org.springframework.format;

public interface Formatter<T> extends Printer<T>, Parser<T> {
}
```

Formatter继承了Printer和Parser接口，下面是这两个接口的定义：

```
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;
}
```

如果要创建你自己的Formatter， 需要实现上面提到的Formatter接口。参数T是需要被格式化的对象，例如java.util.Date。print()方法用来将T打印出来。

parse（）用来将从客户端返回的格式化表示形式解析为T的实例。如果解析尝试失败，格式化程序应引发ParseException或IllegalArgumentException。注意确保格式化程序实现是线程安全的。

format子包提供了方便使用的Formatter几种实现，number包提供了NumberStyleFormatter, CurrencyStyleFormatter, 和 PercentStyleFormatter 使用java.text.NumberFormat来格式化Number对象。datetime包提供了DateFormatter使用java.text.DateFormat来格式化java.util.Dat对象。datetime.joda包基于joda时间库提供全面的datetime格式支持。

下面的DateFormatter是Formatter的一个实现：

```
package org.springframework.format.datetime;

public final class DateFormatter implements Formatter<Date> {
```

更多教程请访问:www.flydean.com

```
    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;
    }
}
```

Spring团队欢迎社区驱动的Formatter程序贡献。

3.5.2 注解驱动的Formatting

字段的格式化可以通过字段类型的配置或者通过注解来实现。要绑定一个注解到Formatter，可以实现AnnotationFormatterFactory接口。下面显示了AnnotationFormatterFactory接口的定义：

```
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);
}
```

要创建实现，请执行以下操作：参数化A为要与格式逻辑关联的字段的annotationType，例如org.springframework.format.annotation.DateTimeFormat。getFieldTypes（）返回可在其上使用注解的字段类型。getprinter（）返回Printer以打印注解字段的值。getParser（）返回一个Parser来解析注解字段的clientValue。

以下示例AnnotationFormatterFactory实现将@NumberFormat注解绑定到formatter程序，以指定数字样式或模式：

```
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);
```

更多教程请访问:www.flydean.com

```
    }

    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 NumberStyleFormatter(annotation.pattern());
        } else {
            Style style = annotation.style();
            if (style == Style.PERCENT) {
                return new PercentStyleFormatter();
            } else if (style == Style.CURRENCY) {
                return new CurrencyStyleFormatter();
            } else {
                return new NumberStyleFormatter();
            }
        }
    }
}
```

可以使用@NumberFormat来触发格式化，如下所示：

```
public class MyModel {

    @NumberFormat(style=Style.CURRENCY)
    private BigDecimal decimal;
}
```

Format注解API

org.springframework.format.annotation包中存在可移植format annotation API。你可以使用@NumberFormat来格式化Number字段，如Double和Long，使用@DateTimeFormat来格式化java.util.Date、java.util.Calendar, Long（毫秒时间戳）以及jsr-310 java.time和Joda-Time值类型。

以下示例使用@DateTimeFormat 将java.util.date格式化为ISO日期（yyyy-mm-dd）：

```
public class MyModel {

    @DateTimeFormat(iso=ISO.DATE)
    private Date date;
}
```

3.5.3 FormatterRegistry SPI

FormatterRegistry是用于注册格式化程序和转换器的SPI。FormattingConversionService是一种适用于大多数环境的FormatTerregistry的实现。你可以通过编程或声明方式将此变量配置为SpringBean，例如，通过使用FormattingConversionServiceFactoryBean。因为这个实现也实现了ConversionService，所以可以直接配置它与Spring的DataBinder和Spring表达式语言（SPEL）一起使用。

下面的列表显示了FormatterRegistry SPI：

```
package org.springframework.format;

public interface FormatterRegistry extends ConverterRegistry {

    void addFormatterForFieldType(Class<?> fieldType, Printer<?> printer, Parser<?> parser);

    void addFormatterForFieldType(Class<?> fieldType, Formatter<?> formatter);
```

更多教程请访问:www.flydean.com

```
    void addFormatterForFieldType(Formatter<?> formatter);

    void addFormatterForAnnotation(AnnotationFormatterFactory<?, ?> factory);
}
```

如上面所示，你可以按字段类型或注解注册formatters。

FormatterRegistry SPI允许你集中配置格式化规则，而不是在控制器之间复制这样的配置。例如，你可能希望强制所有日期字段以某种方式格式化，或者具有特定注解的字段以某种方式格式化。使用共享FormatterRegistry，你只需定义一次这些规则，这些规则在需要格式化时会自动使用。

3.5.4 FormatterRegistrar SPI

FormatterRegistrar是一个SPI，用于通过FormatterRegistry注册formatters和converters。下面的列表显示了它的接口定义：

```
package org.springframework.format;

public interface FormatterRegistrar {

    void registerFormatters(FormatterRegistry registry);
}
```

在为给定格式类别（如日期格式）注册多个相关转换器和格式化程序时，FormatterRegistrar非常有用。如果声明性registration不足-例如，当formatter 需要在与其自身不同的特定字段类型下索引，或者在注册Printer/Parser对时，它也很有用。下一节提供有关converter和formatter注册的更多信息。

3.5.5.在 Spring MVC 中配置Formatting

参考 Spring MVC 的Conversion and Formatting 章节。

更多教程请访问:www.flydean.com

3.6配置全局Date和Time格式

默认情况下，未用@DateTimeFormat注解的日期和时间字段使用DateFormat.SHORT style从字符串转换。如果愿意，可以通过定义自己的全局格式来更改此设置。

为此，你需要确保Spring不注册默认格式化程序。相反，你应该手动注册所有格式化程序。使用org.springframework.format.datetime.joda.JodaTimeFormatterRegistrar或org.springframework.format.datetime.DateFormatterRegistrar 类，具体取决于是否使用Joda-Time 库。

例如，下面的Java配置注册一个全局yyyYMdD格式（这个示例不依赖于JoDA时间库）：

```
@Configuration
public class AppConfig {

    @Bean
    public FormattingConversionService conversionService() {

        // Use the DefaultFormattingConversionService but do not register defaults
        DefaultFormattingConversionService conversionService = new DefaultFormattingConversionService(false);

        // Ensure @NumberFormat is still supported
        conversionService.addFormatterForFieldAnnotation(new NumberFormatAnnotationFormatterFactory());

        // Register date conversion with a specific global format
        DateFormatterRegistrar registrar = new DateFormatterRegistrar();
        registrar.setFormatter(new DateFormatter("yyyyMMdd"));
        registrar.registerFormatters(conversionService);

        return conversionService;
    }
}
```

如果你更喜欢XML配置， 那么可以使用FormattingConversionServiceFactoryBean，如下所示（使用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
        https://www.springframework.org/schema/beans/spring-beans.xsd>

    <bean id="conversionService" class="org.springframework.format.support.FormattingConversionServiceFactoryBe
an">
        <property name="registerDefaultFormatters" value="false" />
        <property name="formatters">
            <set>
                <bean class="org.springframework.format.number.NumberFormatAnnotationFormatterFactory" />
            </set>
        </property>
        <property name="formatterRegistrars">
            <set>
                <bean class="org.springframework.format.datetime.joda.JodaTimeFormatterRegistrar">
                    <property name="dateFormatter">
                        <bean class="org.springframework.format.datetime.joda.DateTimeFormatterFactoryBean">
                            <property name="pattern" value="yyyyMMdd"/>
                        </bean>
                    </property>
                </bean>
            </set>
        </property>
    </bean>
</beans>
```

更多教程请访问:www.flydean.com

> Joda-Time提供单独的不同类型来表示date、time和date-time值。应使用JodTimeFormatterRegStrar的DateFormatter、TimeFormatter和DateTimeFormatter属性为每种类型配置不同的格式。DateTimeFormatterFactoryBean提供了一种创建格式化程序的方便方法。
>
> 如果使用SpringMVC，请记住显式配置所使用的转换服务。对于基于Java的@Configuration，这意味着扩展WebMvcConfigurationSupport类并重写mvcConversionService（）方法。对于XML，应该使用mvc:annotation-driven元素的conversion-service属性。有关详细信息，请参见转换和格式化。

更多教程请访问:www.flydean.com

3.7Spring验证

Spring3为其验证支持引入了一些增强功能。首先，完全支持JSR-303 bean验证API。第二，当以编程方式使用时，Spring的DataBinder既可以验证对象，也可以绑定到对象。第三，SpringMVC支持声明性地验证@Controller输入。

3.7.1 JSR-303 Bean Validation API 概览

JSR303标准化了Java平台的验证约束声明和元数据。通过使用此API，你可以使用声明性验证约束来注解域模型属性，并且运行时会强制验证它们。你可以使用许多内置约束。你还可以定义自己的自定义约束。

考虑下面的示例，该示例显示了一个具有两个属性的简单PersonForm模型：

```
public class PersonForm {
    private String name;
    private int age;
}
```

JSR-303允许你针对这些属性定义声明性验证约束，如下示例所示：

```
public class PersonForm {

    @NotNull
    @Size(max=64)
    private String name;

    @Min(0)
    private int age;
}
```

当一个JSR-303验证器验证这个类的实例时，这些限制将被执行。

3.7.2 配置一个Bean Validation Provider

Spring完全支持bean验证API。这包括对将JSR-303或JSR-349 Bean验证提供者作为SpringBean引导的方便支持。这允许你在应用程序中需要验证的任何位置插入javax.validation.validatorFactory或javax.validation.validator。

你可以使用LocalValidatorFactoryBean将默认验证器配置为Springbean，如下示例所示：

```
<bean id="validator"
    class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"/>
```

前面示例中的基本配置通过使用其默认引导机制触发bean验证进行初始化。JSR-303或JSR-349 provider（如Hibernate验证器）应该出现在类路径中，并被自动检测到。

注入一个Validator

LocalValidatorFactoryBean继承了javax.validation.ValidatorFactory, javax.validation.Validator和Spring的org.springframework.validation.Validator。 你可以将对这些接口的引用注入需要调用验证逻辑的bean中。

你可以注入javax.validation.Validator 的引用，如果你想直接使用Bean Validation API ， 如下所示：

更多教程请访问:www.flydean.com

```
import javax.validation.Validator;

@Service
public class MyService {

    @Autowired
    private Validator validator;
```

你可以注入org.springframework.validation.Validator的引用，如果你想直接使用Spring Validation API， 如下所示：

```
import org.springframework.validation.Validator;

@Service
public class MyService {

    @Autowired
    private Validator validator;
}
```

配置自定义约束

每个bean验证约束由两部分组成：

- 声明约束及其可配置属性的@Constraint注解。
- 实现约束行为的javax.validation.ConstraintValidator接口的实现。

要将声明与实现关联，每个@Constraint注解都引用相应的ConstraintValidator实现类。在运行时，当在域模型中遇到约束注解时，ConstraintValidatorFactory将实例化引用的实现。

默认情况下，LocalValidatorFactoryBean配置使用Spring创建ConstraintValidator实例的SpringConstraintValidatorFactory。这让你的自定义ConstraintValidators像其他任何SpringBean一样从依赖注入中受益。

以下示例显示了一个自定义@Constraint声明，后跟一个使用spring进行依赖项注入的关联constraintvalidator实现：

```
@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;

    ...
}
```

如前面的示例所示，ConstraintValidator实现可以像任何其他SpringBean那样具有其依赖项@Autowired。

Spring驱动的方法验证

你可以通过MethodValidationPostProcessor将bean validation 1.1支持的方法验证功能（以及作为自定义扩展的hibernate validator 4.3支持的方法验证功能）集成到Spring上下文中，如下所示：

```
<bean class="org.springframework.validation.beanvalidation.MethodValidationPostProcessor"/>
```

更多教程请访问:www.flydean.com

为了符合Spring驱动的方法验证的条件，所有目标类都需要用Spring的@Validated注解进行注解。（可选，你也可以声明要使用的验证组。）有关Hibernate验证程序和Bean Validation 1.1提供程序的设置详细信息，请参阅MethodValidationPostProcessor Javadoc。

额外的配置选项

默认的LocalValidatorFactoryBean配置足以满足大多数情况。对于各种bean验证构造，有许多配置选项，从消息插值到遍历解析。有关这些选项的详细信息，请参阅LocalValidatorFactoryBean javadoc。

3.7.3 配置DataBinder

从Spring3开始，你可以使用Validator配置一个DataBinder实例。配置后，可以通过调用binder.validate（）来调用验证器。任何验证错误都会自动添加到binder的bindingresult中。

下面的示例演示如何在绑定到目标对象后以编程方式使用DataBinder调用验证逻辑：

```
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();
```

你还可以通过databinder.addValidators和databinder.replaceValidators配置具有多个Validator实例的databinder。这在将全局配置的bean验证与本地配置在databinder实例上的Spring验证结合使用时非常有用。请参见[验证MVC配置]。

3.7.4 Spring MVC 3 验证

请参考 Spring MVC 的 验证章节。

更多教程请访问:www.flydean.com

4.SpEL Spring表达式语言

Spring表达式语言（简称“SpEL”）是一种功能强大的表达式语言，支持在运行时查询和操作对象图。语言语法类似于统一EL，但提供了其他功能，最显著的是方法调用和基本的字符串模板化功能。

虽然还有其他几种Java表达式语言可用，即--OGNL、MVEL和JBOSS EL，但是，为了命名一致性，创建了Spring表达式语言，为Spring社区提供了一个支持良好的表达式语言，可以在Spring的所有产品中使用。它的语言特性是由Spring产品组合中项目的需求驱动的，包括基于Eclipse的Spring工具套件中代码完成支持的工具需求。这就是说，SPEL基于technology-agnostic的API，它允许在需要时集成其他表达式语言实现。

虽然SPEL是Spring组合中expression求值的基础，但它不直接与Spring挂钩，可以独立使用。为了自我包含，本章中的许多示例将spel用作独立的表达式语言。这需要创建一些引导基础结构类，如解析器。大多数Spring用户不需要处理这个基础结构，只需要编写表达式字符串进行求值。这种典型用法的一个例子是将spel集成到创建基于XML或注解的bean定义中，如定义bean定义的表达式支持中所示。

本章介绍表达式语言的特性、API及其语言语法。在一些地方，Inventor和Society类被用作表达评估的目标对象。这些类声明和用于填充它们的数据列在本章末尾。

表达式语言支持如下的功能:

- Literal expressions
- Boolean and relational operators
- Regular expressions
- Class expressions
- Accessing properties, arrays, lists, and maps
- Method invocation
- Relational operators
- Assignment
- Calling constructors
- Bean references
- Array construction
- Inline lists
- Inline maps
- Ternary operator
- Variables
- User-defined functions
- Collection projection
- Collection selection
- Templated expressions

更多教程请访问:www.flydean.com

4.1求值

本节介绍了SPEL接口及其表达式语言的简单用法。完整的语言引用可以在语言引用中找到。

下面的代码介绍了用于计算文本字符串表达式hello world的spel api。

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("'Hello World'");
String message = (String) exp.getValue();
```

> message的值是“Hello World”。

你最可能使用的spel类和接口位于org.springframework.expression包及其子包（如spel.support）中。

ExpressionParser接口负责分析表达式字符串。在前面的示例中，表达式字符串是由周围的单引号表示的字符串文字。表达式接口负责计算之前定义的表达式字符串。分别调用parser.parseExpression和exp.getValue时，可以引发两个异常：parseException和evaluationException。

SPEL支持广泛的特性，例如调用方法、访问属性和调用构造函数。

在下面的方法调用示例中，我们在字符串文本上调用concat方法：

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("'Hello World'.concat('!')");
String message = (String) exp.getValue();
```

> 现在message的值是'Hello World!'.

以下调用JavaBean属性的示例调用String属性Bytes：

```
ExpressionParser parser = new SpelExpressionParser();

// invokes 'getBytes()'
Expression exp = parser.parseExpression("'Hello World'.bytes");
byte[] bytes = (byte[]) exp.getValue();
```

> 字符串被转换为byte array。

spel还通过使用标准点表示法（如prop1.prop2.prop3）以及相应的属性值设置来支持嵌套属性。也可以访问公共字段。

下面的示例演示如何使用点表示法获取文本的长度：

```
ExpressionParser parser = new SpelExpressionParser();

// invokes 'getBytes().length'
Expression exp = parser.parseExpression("'Hello World'.bytes.length");
int length = (Integer) exp.getValue();
```

可以调用字符串的构造函数，而不是使用字符串文字，如下示例所示：

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("new String('hello world').toUpperCase()");
String message = exp.getValue(String.class);
```

更多教程请访问:www.flydean.com

注意泛型方法的用法：public<T>T getValue（class<T> desiredResultType）。使用此方法可以消除将表达式值强制转换为所需结果类型的需要。如果无法将值强制转换为T类型或无法使用注册的类型转换器转换，则将引发EvaluationException。

spel的更常见用法是提供一个表达式字符串，该字符串根据特定的对象实例（称为根对象）进行计算。以下示例显示如何从Inventor类的实例中检索name属性或创建布尔条件：

```
// Create and set a calendar
GregorianCalendar c = new GregorianCalendar();
c.set(1856, 7, 9);

// The constructor arguments are name, birthday, and nationality.
Inventor tesla = new Inventor("Nikola Tesla", c.getTime(), "Serbian");

ExpressionParser parser = new SpelExpressionParser();

Expression exp = parser.parseExpression("name");
String name = (String) exp.getValue(tesla);
// name == "Nikola Tesla"

exp = parser.parseExpression("name == 'Nikola Tesla'");
boolean result = exp.getValue(tesla, Boolean.class);
// result == true
```

4.1.1 理解EvaluationContext

当计算表达式以解析属性、方法或字段并帮助执行类型转换时，将使用EvaluationContext接口。Spring提供了两种实现。

- SimpleEvaluationContext:为不需要完整的SPEL语言语法并且应受到有意义限制的表达式类别公开基本的SPEL语言功能和配置选项的子集。示例包括但不限于数据绑定表达式和基于属性的筛选器。
- StandardEvaluationContext：公开全套SPEL语言功能和配置选项。你可以使用它来指定一个默认的根对象，并配置每个可用的与评估相关的策略。

SimpleEvaluationContext设计为仅支持SPEL语言语法的一个子集。它不包括Java类型引用、构造函数和bean引用。它还要求你显式选择表达式中属性和方法的支持级别。默认情况下，create（）静态工厂方法仅启用对属性的读取访问。你还可以获得一个构建器来配置所需的确切支持级别，针对以下一种或几种组合：

- 仅自定义PropertyAccessor（无反射）
- 只读访问的数据绑定属性
- 读写数据绑定属性

类型转换

默认情况下，spel使用SpringCore中提供的转换服务（org.springframework.core.convert.conversionService）。此转换服务附带许多用于常见转换的内置转换器，但也完全可扩展，因此你可以在类型之间添加自定义转换。此外，它还是generics-aware的。这意味着，当你在表达式中处理泛型类型时，spel会尝试转换来维护它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（）进行的赋值用于设置List属性。属性的类型实际上是list<boolean>。spel认识到列表中的元素在放入之前需要转换为布尔值。以下示例显示了如何执行此操作：

```
class Simple {
    public List<Boolean> booleanList = new ArrayList<Boolean>();
}

Simple simple = new Simple();
simple.booleanList.add(true);
```

更多教程请访问:www.flydean.com

```
EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();

// "false" is passed in here as a String. SpEL and the conversion service
// will recognize that it needs to be a Boolean and convert it accordingly.
parser.parseExpression("booleanList[0]").setValue(context, simple, "false");

// b is false
Boolean b = simple.booleanList.get(0);
```

4.1.2 解析配置

可以使用解析器配置对象（org.springframework.expression.spel.SpelParserConfiguration）配置SPEL表达式解析器。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，并且指定索引处的元素为空，则可以自动创建元素。这在使用由属性引用链组成的表达式时很有用。如果索引到数组或列表，并指定超出当前数组或列表大小末尾的索引，则可以自动增大数组或列表以容纳该索引。下面的示例演示如何自动增大列表：

```
class Demo {
    public List<String> list;
}

// Turn on:
// - auto null reference initialization
// - auto collection growing
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 will now be a real collection of 4 entries
// Each entry is a new empty String
```

4.1.3. SpEL编译

SpringFramework4.1包括一个基本表达式编译器。通常对表达式进行解释，这在评估期间提供了很大的动态灵活性，但不能提供最佳性能。对于偶尔使用的表达式用法，这是可以的，但是，当被其他组件（如Spring集成）使用时，性能可能非常重要，并且不需要动态性。

spel编译器旨在满足这一需求。在求值过程中，编译器生成一个Java类，该类在运行时体现表达式行为，并使用该类实现更快的表达式求值。由于没有在表达式周围键入内容，编译器在执行编译时使用在表达式的解释性计算期间收集的信息。例如，它不完全从表达式中知道属性引用的类型，但在第一个解释的计算过程中，它会发现它是什么。当然，如果各种表达式元素的类型随着时间的推移而改变，那么基于此类派生信息进行编译可能会在以后引起麻烦。因此，编译最适合于类型信息在重复计算时不会更改的表达式。

考虑以下基本表达式：

```
someArray[0].someProperty.someOtherProperty < 0.1
```

由于前面的表达式涉及数组访问、一些属性取消引用和数值操作，因此性能提高可能非常明显。在一个50000次迭代的微基准测试运行的例子中，使用解释器评估花费了75毫秒，而使用已编译的表达式版本只花费了3毫秒。

编译器配置

更多教程请访问:www.flydean.com

默认情况下不会打开编译器，但你可以通过两种不同的方式之一打开它。你可以通过使用解析器配置过程（前面讨论过）或在另一个组件中嵌入spel用法时使用系统属性来启用它。本节讨论这两个选项。

编译器可以在三种模式中的一种模式下运行，这些模式在org.springframework.expression.spel.SpelCompilerMode枚举中列举的。模式如下：

- 关闭（默认）：编译器关闭。
- IMMEDIATE：在immediate模式下，尽快编译表达式。这通常是在第一次解释评估之后。如果编译表达式失败（通常是由于前面描述的类型更改），则表达式计算的调用方将收到异常。
- MIXED：在混合模式下，表达式会随着时间的推移在解释和编译模式之间自动切换。在经过一定数量的解释运行后，它们将切换到已编译的表单，如果编译的表单出现问题（如前面所述的类型更改），表达式将自动再次切换回已解释的表单。稍后，它可能会生成另一个已编译的表单并切换到该表单。基本上，用户进入IMMEDIATE模式的异常是在内部处理的。

IMMEDIATE模式存在，因为MIXED模式可能会导致具有副作用的表达式出现问题。如果编译表达式在部分成功后报异常，它可能已经做了一些影响系统状态的事情。如果发生了这种情况，调用方可能不希望它以解释模式静默地重新运行，因为表达式的一部分可能运行了两次。

选择模式后，使用SpelParserConfiguration配置解析器。以下示例显示了如何执行此操作：

```
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);
```

指定编译器模式时，还可以指定类加载器（允许传递空值）。编译表达式在任何提供的子类加载器下创建的子类加载器中定义。必须确保，如果指定了类加载器，它可以看到表达式计算过程中涉及的所有类型。如果不指定类加载器，则使用默认的类加载器（通常是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当spel嵌入到其他组件中，并且可能无法通过配置对象对其进行配置时使用。在这些情况下，可以使用系统属性。可以将spring.expression.compiler.mode属性设置为SpelCompilerMode枚举值之一（off、immediate或mixed）。

Compiler限制

从SpringFramework4.1开始，基本的编译框架就已经就位了。然而，框架还不支持编译各种类型的表达式。最初的重点是可能在性能关键的上下文中使用的公共表达式。目前无法编译以下类型的表达式：

- 涉及赋值的表达式
- 依赖转换服务的表达式
- 使用自定义冲突解决程序或访问器的表达式
- 使用选择或投影的表达式

将来会编译更多类型的表达式。

更多教程请访问:www.flydean.com

4.2bean定义中的表达式

可以将SPEL表达式与基于XML或基于注解的配置元数据一起用于定义BeanDefinition实例。在这两种情况下，定义表达式的语法形式都是 #{ <expression string> }。

4.2.1 XML配置

可以使用表达式设置属性或构造函数参数值，如下示例所示：

```
<bean id="numberGuess" class="org.spring.samples.NumberGuess">
    <property name="randomNumber" value="#{ T(java.lang.Math).random() * 100.0 }"/>

    <!-- other properties -->
</bean>
```

SystemProperties变量是预定义的，因此可以在表达式中使用它，如下示例所示：

```
<bean id="taxCalculator" class="org.spring.samples.TaxCalculator">
    <property name="defaultLocale" value="#{ systemProperties['user.region'] }"/>

    <!-- other properties -->
</bean>
```

注意，在此上下文中，你不必在预定义变量前面加上#符号。

你还可以按名称引用其他bean属性，如下示例所示：

```
<bean id="numberGuess" class="org.spring.samples.NumberGuess">
    <property name="randomNumber" value="#{ T(java.lang.Math).random() * 100.0 }"/>

    <!-- other properties -->
</bean>

<bean id="shapeGuess" class="org.spring.samples.ShapeGuess">
    <property name="initialShapeSeed" value="#{ numberGuess.randomNumber }"/>

    <!-- other properties -->
</bean>
```

4.2.2 注解配置

要指定默认值，可以将@value注解放在字段、方法、方法或构造函数参数上。

以下示例设置字段变量的默认值：

```
public static class FieldValueTestBean

    @Value("#{ systemProperties['user.region'] }")
    private String defaultLocale;

    public void setDefaultLocale(String defaultLocale) {
        this.defaultLocale = defaultLocale;
    }

    public String getDefaultLocale() {
```

更多教程请访问:www.flydean.com

```
        return this.defaultLocale;
    }

}
```

下面的示例显示了等效的但在属性设置器方法上的示例：

```
public static class PropertyValueTestBean

    private String defaultLocale;

    @Value("#{ systemProperties['user.region'] }")
    public void setDefaultLocale(String defaultLocale) {
        this.defaultLocale = defaultLocale;
    }

    public String getDefaultLocale() {
        return this.defaultLocale;
    }

}
```

autowired方法和构造函数也可以使用@value注解，如下示例所示：

```
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;
    }

    // ...
}
```

更多教程请访问:www.flydean.com

4.3语言引用

本节介绍Spring表达式语言的工作原理。它包括以下主题：

- 文字表达式
- 属性、数组、列表、映射和索引器
- 内联 List
- 内联 Map
- Array
- 方法
- Operators
- 类型
- Constructors
- 变量
- 功能
- bean引用
- 三元运算符（if-then-else）
- elvis
- Safe Navigation Operator

4.3.1 文字表达式

支持的文本表达式类型包括字符串、数值（int、real、hex）、布尔值和null。字符串由单引号分隔。要将单引号本身放入字符串中，请使用两个单引号字符。

下面的列表显示了文本的简单用法。通常，它们不会像这样单独使用，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文本。

```
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、指数记数法和小数点。默认情况下，使用double.parseDouble（）解析实数。

4.3.2 Properties, Arrays, Lists, Maps, and Indexers

使用属性引用导航很容易。为此，请使用句点指示嵌套的属性值。

inventor类的实例pupin和tesla使用示例部分中使用的类中列出的数据填充。为了“向下”导航并获得特斯拉的出生年份和小狗出生城市，我们使用以下表达式：

```
// evals to 1856
```

更多教程请访问:www.flydean.com

```
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允许不区分大小写。数组和列表的内容是使用方括号表示法获得的，如下例所示：

```
ExpressionParser parser = new SpelExpressionParser();
EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();

// Inventions Array

// evaluates to "Induction motor"
String invention = parser.parseExpression("inventions[3]").getValue(
        context, tesla, String.class);

// Members List

// evaluates to "Nikola Tesla"
String name = parser.parseExpression("Members[0].Name").getValue(
        context, ieee, String.class);

// List and Array navigation
// evaluates to "Wireless communication"
String invention = parser.parseExpression("Members[0].Inventions[6]").getValue(
        context, ieee, String.class);
```

映射的内容是通过在括号内指定文本键值获得的。在下面的示例中，由于Officers map的键是字符串，因此我们可以指定字符串文本：

```
// 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");
```

4.3.3 内联List

你可以使用符号直接在表达式中表示列表。

```
// 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的文本组成，则会创建一个常量列表来表示表达式（而不是在每个计算上构建一个新列表）。

4.3.4 内联Maps

你还可以使用key:value表示法在表达式中直接表示映射。以下示例显示了如何执行此操作：

更多教程请访问:www.flydean.com

```
// 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);
```

{:}代表着空map。出于性能原因，如果映射本身由固定的文本或其他嵌套常量结构（列表或映射）组成，则会创建一个常量映射来表示表达式（而不是在每个计算上构建一个新的映射）。Map的key的引号是可选的。上面的示例不使用带引号的键。

4.3.5 Array构造函数

可以使用熟悉的Java语法构建数组，可以选择的提供初始化器，以便在构建时填充数组。以下示例显示了如何执行此操作：

```
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);
```

构造多维数组时，当前无法提供初始值设定项。

4.3.6 方法

可以通过使用典型的Java编程语法来调用方法。还可以对文本调用方法。还支持变量参数。以下示例演示如何调用方法：

```
// string literal, evaluates to "bc"
String bc = parser.parseExpression("'abc'.substring(1, 3)").getValue(String.class);

// evaluates to true
boolean isMember = parser.parseExpression("isMember('Mihajlo Pupin')").getValue(
        societyContext, Boolean.class);
```

4.3.7 Operators

spEL支持下面的运算符：

- 关系运算符
- 逻辑运算符
- 算数运算符
- 赋值运算符

关系运算符

使用标准运算符表示法支持关系运算符（等于、不等于、小于、小于或等于、大于和大于或等于）。以下列表显示了一些运算符示例：

```
// evaluates to true
boolean trueValue = parser.parseExpression("2 == 2").getValue(Boolean.class);
```

更多教程请访问:www.flydean.com

```
// 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);
```

> 与null的大于或小于比较遵循一个简单规则：null被视为无（不是零）。因此，任何其他值始终大于null值（x>null始终为真），任何其他值始终小于零（x < null 始终为假）。
>
> 如果你更喜欢数值比较，请避免基于数字的null比较，而是与零比较（例如，x>0或x<0）。

除了标准的关系运算符之外，spel还支持instanceof和基于正则表达式的matches运算符。下面的列表显示了这两种方法的示例：

```
// 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);
```

请小心处理基础类型，因为它们会立即装箱到包装类型，因此1 instanceof t（int）的计算结果为false，而1 instanceof t（Integer）的计算结果为true，如预期的那样。

每个符号运算符也可以指定为纯字母等价物。这样可以避免使用的符号对嵌入表达式的文档类型（如XML文档）具有特殊意义的问题。文本等价物为：

```
lt (<)

gt (>)

le (<=)

ge (>=)

eq (==)

ne (!=)

div (/)

mod (%)

not (!)
```

所有的文本操作符都不区分大小写。

逻辑操作符

SpEL支持下面的逻辑操作符：

- and
- or
- not

下面的例子展示了如何使用logical操作符：

更多教程请访问:www.flydean.com

```
// -- AND --

// evaluates to false
boolean falseValue = parser.parseExpression("true and false").getValue(Boolean.class);

// evaluates to true
String expression = "isMember('Nikola Tesla') and isMember('Mihajlo Pupin')";
boolean trueValue = parser.parseExpression(expression).getValue(societyContext, Boolean.class);

// -- OR --

// evaluates to true
boolean trueValue = parser.parseExpression("true or false").getValue(Boolean.class);

// evaluates to true
String expression = "isMember('Nikola Tesla') or isMember('Albert Einstein')";
boolean trueValue = parser.parseExpression(expression).getValue(societyContext, Boolean.class);

// -- NOT --

// evaluates to false
boolean falseValue = parser.parseExpression("!true").getValue(Boolean.class);

// -- AND and NOT --
String expression = "isMember('Nikola Tesla') and !isMember('Mihajlo Pupin')";
boolean falseValue = parser.parseExpression(expression).getValue(societyContext, Boolean.class);
```

数学运算符

可以对数字和字符串使用加法运算符。只能对数字使用减法、乘法和除法运算符。你还可以使用模量（%）和指数幂（^）运算符。执行标准运算符优先级。以下示例显示了使用中的数学运算符：

```
// Addition
int two = parser.parseExpression("1 + 1").getValue(Integer.class);  // 2

String testString = parser.parseExpression(
        "'test' + ' ' + 'string'").getValue(String.class);  // 'test string'

// Subtraction
int four = parser.parseExpression("1 - -3").getValue(Integer.class);  // 4

double d = parser.parseExpression("1000.00 - 1e4").getValue(Double.class);  // -9000

// Multiplication
int six = parser.parseExpression("-2 * -3").getValue(Integer.class);  // 6

double twentyFour = parser.parseExpression("2.0 * 3e0 * 4").getValue(Double.class);  // 24.0

// Division
int minusTwo = parser.parseExpression("6 / -3").getValue(Integer.class);  // -2

double one = parser.parseExpression("8.0 / 4e0 / 2").getValue(Double.class);  // 1.0

// Modulus
int three = parser.parseExpression("7 % 4").getValue(Integer.class);  // 3

int one = parser.parseExpression("8 / 5 % 2").getValue(Integer.class);  // 1

// Operator precedence
int minusTwentyOne = parser.parseExpression("1+2-3*8").getValue(Integer.class);  // -21
```

赋值操作

要设置属性，请使用赋值运算符（=）。这通常在对setValue的调用中完成，但也可以在对getValue的调用中完成。下面的列表显示了使用赋值运算符的两种方法：

更多教程请访问:www.flydean.com

```
Inventor inventor = new Inventor();
EvaluationContext context = SimpleEvaluationContext.forReadWriteDataBinding().build();

parser.parseExpression("Name").setValue(context, inventor, "Aleksandar Seovic");

// alternatively
String aleks = parser.parseExpression(
        "Name = 'Aleksandar Seovic'").getValue(context, inventor, String.class);
```

4.3.8 类型

你可以使用特殊的T运算符来指定java.lang.class（类型）的实例。静态方法也可以使用此运算符调用。StandardEvaluationContext使用TypeLocator来查找类型，StandardTypeLocator（可以替换）是在理解java.lang包的基础上构建的。这意味着T（）对java.lang中类型的引用不需要完全限定，但所有其他类型引用都必须是限定的。下面的示例演示如何使用T运算符：

```
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);
```

4.3.9 构造器

可以使用new运算符调用构造函数。除了基元类型（int、float等）和字符串之外，其他类型都应该使用完全限定的类名。下面的示例演示如何使用新的运算符来调用构造函数：

```
Inventor einstein = p.parseExpression(
        "new org.spring.samples.spel.inventor.Inventor('Albert Einstein', 'German')")
        .getValue(Inventor.class);

//create new inventor instance within add method of List
p.parseExpression(
        "Members.add(new org.spring.samples.spel.inventor.Inventor(
            'Albert Einstein', 'German'))").getValue(societyContext);
```

4.3.10 变量

可以使用#variableName语法引用表达式中的变量。变量是通过在EvaluationContext实现上使用setVariable方法设置的。以下示例显示如何使用变量：

```
Inventor tesla = new Inventor("Nikola Tesla", "Serbian");

EvaluationContext context = SimpleEvaluationContext.forReadWriteDataBinding().build();
context.setVariable("newName", "Mike Tesla");

parser.parseExpression("Name = #newName").getValue(context, tesla);
System.out.println(tesla.getName())  // "Mike Tesla"
```

#this和#root

更多教程请访问:www.flydean.com

#this始终是定义的，并引用当前的评估对象（不合格的引用将根据该对象进行解析）。#root变量总是被定义并引用根上下文对象。尽管#this可能会随着表达式的组件的计算而变化，但是#root始终引用根。以下示例说明如何使用#this和#root变量：

```
// 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();
EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataAccess();
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);
```

4.3.11 函数

你可以通过注册可以在表达式字符串中调用的用户定义函数来扩展spel。该函数通过EvaluationContext注册。以下示例显示如何注册用户定义函数：

```
Method method = ...;

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();
context.setVariable("myFunction", method);
```

例如，考虑以下反转字符串的实用程序方法：

```
public abstract class StringUtils {

    public static String reverseString(String input) {
        StringBuilder backwards = new StringBuilder(input.length());
        for (int i = 0; i < input.length(); i++)
            backwards.append(input.charAt(input.length() - 1 - i));
        }
        return backwards.toString();
    }
}
```

然后你可以注册并使用前面的方法，如下示例所示：

```
ExpressionParser parser = new SpelExpressionParser();

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();
context.setVariable("reverseString",
        StringUtils.class.getDeclaredMethod("reverseString", String.class));

String helloWorldReversed = parser.parseExpression(
        "#reverseString('hello')").getValue(context, String.class);
```

4.3.12 Bean引用

如果已使用bean resolver配置了评估上下文，则可以使用@符号从表达式中查找bean。以下示例显示了如何执行此操作：

更多教程请访问:www.flydean.com

```
ExpressionParser parser = new SpelExpressionParser();
StandardEvaluationContext context = new StandardEvaluationContext();
context.setBeanResolver(new MyBeanResolver());

// This will end up calling resolve(context,"something") on MyBeanResolver during evaluation
Object bean = parser.parseExpression("@something").getValue(context);
```

要访问工厂bean本身，你应该在bean名称前面加上&符号。以下示例显示了如何执行此操作：

```
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);
```

4.3.13 If-Then-Else

可以使用三元运算符在表达式中执行if-then-else条件逻辑。下面的列表显示了一个最小的示例：

```
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：

```
parser.parseExpression("Name").setValue(societyContext, "IEEE");
societyContext.setVariable("queryName", "Nikola Tesla");

expression = "isMember(#queryName)? #queryName + ' is a member of the ' " +
        "+ Name + ' Society' : #queryName + ' is not a member of the ' + Name + ' Society'";

String queryResultString = parser.parseExpression(expression)
        .getValue(societyContext, String.class);
// queryResultString = "Nikola Tesla is a member of the IEEE Society"
```

有关三元运算符的更短语法，请参阅ELVIS运算符。

4.3.14 Elvis

ELVIS运算符是三元运算符语法的缩写，在groovy语言中使用。对于三元运算符语法，通常必须重复变量两次，如下示例所示：

```
String name = "Elvis Presley";
String displayName = (name != null ? name : "Unknown");
```

相反，你可以使用Elvis操作符（以Elvis的发型命名）。下面的示例演示如何使用Elvis运算符：

```
ExpressionParser parser = new SpelExpressionParser();

String name = parser.parseExpression("name?:'Unknown'").getValue(String.class);
System.out.println(name);  // 'Unknown'
```

下面的列表显示了一个更复杂的示例：

更多教程请访问:www.flydean.com

```
ExpressionParser parser = new SpelExpressionParser();
EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();

Inventor tesla = new Inventor("Nikola Tesla", "Serbian");
String name = parser.parseExpression("Name?:'Elvis Presley'").getValue(context, tesla, String.class);
System.out.println(name);  // Nikola Tesla

tesla.setName(null);
name = parser.parseExpression("Name?:'Elvis Presley'").getValue(context, tesla, String.class);
System.out.println(name);  // Elvis Presley
```

> 可以使用ELVIS运算符在表达式中应用默认值。下面的示例演示如何在@value表达式中使用elvis运算符：
>
> @Value("#{systemProperties['pop3.port'] ?: 25}")
>
> 如果定义了系统属性pop3.port，则会注入该属性，否则会注入25。

4.3.15 Safe Navigation 运算符

Safe Navigation操作符用于避免nullpointerException，它来自groovy语言。通常，当你引用一个对象时，你可能需要在访问该对象的方法或属性之前验证它不是空的。为了避免这种情况，Safe Navigation操作符返回空值而不是抛出异常。以下示例说明如何使用Safe Navigation：

```
ExpressionParser parser = new SpelExpressionParser();
EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();

Inventor tesla = new Inventor("Nikola Tesla", "Serbian");
tesla.setPlaceOfBirth(new PlaceOfBirth("Smiljan"));

String city = parser.parseExpression("PlaceOfBirth?.City").getValue(context, tesla, String.class);
System.out.println(city);  // Smiljan

tesla.setPlaceOfBirth(null);
city = parser.parseExpression("PlaceOfBirth?.City").getValue(context, tesla, String.class);
System.out.println(city);  // null - does not throw NullPointerException!!!
```

4.3.16 集合选择

Selection是一种功能强大的表达式语言功能，通过从源集合的条目中进行选择，可以将源集合转换为另一个集合。

Selection使用的语法为.? [selectionExpression]。它过滤集合并返回包含原始元素子集的新集合。例如，selection可以让我们很容易地获得塞尔维亚发明家的列表，如下示例所示：

```
List<Inventor> list = (List<Inventor>) parser.parseExpression(
        "Members.?[Nationality == 'Serbian']").getValue(societyContext);
```

在list和map上都可以Selection。对于list，将根据每个单独的列表元素评估选择条件。针对map，选择标准针对每个映射条目（Java类型Map.Entry）进行评估。每个map项都有其键和值，可以作为属性访问，以便在选择中使用。

以下表达式返回一个新map，该映射由原始map的那些元素组成，其中输入值小于27：

```
Map newMap = parser.parseExpression("map.?[value<27]").getValue();
```

除了返回所有选定的元素之外，你还能检索第一个或最后一个值。要获取与所选内容匹配的第一个条目，语法为。.^[selectionExpression]。要获取最后一个匹配的选择，语法为.$[SelectionExpression]。

更多教程请访问:www.flydean.com

4.3.17 集合投影

Projection允许集合驱动子表达式的计算，结果是一个新集合。投影的语法是.![projectionExpression]。例如，假设我们有一个发明家列表，但是想要他们出生的城市列表。实际上，我们想为发明家列表中的每个条目评估“placeofbirth.city”。下面的示例使用投影进行此操作：

```
// returns ['Smiljan', 'Idvor' ]
List placesOfBirth = (List)parser.parseExpression("Members.![placeOfBirth.city]");
```

你还可以使用map来驱动投影，在这种情况下，投影表达式针对map中的每个条目（表示为Java Map.Entry）进行评估。跨map投影的结果是一个列表，其中包含对每个map条目的投影表达式的计算。

4.3.18 表达式模板化

表达式模板允许将文本与一个或多个计算块混合。每个评估块都由你可以定义的前缀和后缀字符分隔。常见的选择是使用#{ }作为分隔符，如下示例所示：

```
String randomPhrase = parser.parseExpression(
        "random number is #{T(java.lang.Math).random()}",
        new TemplateParserContext()).getValue(String.class);

// evaluates to "random number is 0.7038186818312008"
```

字符串的计算方法是将文本“random number is”与计算#{ }分隔符内表达式的结果（在本例中，是调用该random（）方法的结果）连接起来。parseExpression（）方法的第二个参数的类型为parserContext。ParserContext接口用于影响表达式的解析方式，以支持表达式模板化功能。TemplateParserContext的定义如下：

```
public class TemplateParserContext implements ParserContext {

    public String getExpressionPrefix() {
        return "#{";
    }

    public String getExpressionSuffix() {
        return "}";
    }

    public boolean isTemplate() {
        return true;
    }
}
```

更多教程请访问:www.flydean.com

5.Spring AOP

面向方面编程（AOP）通过提供对程序结构的另一种思考方式来补充面向对象编程（OOP）。OOP中模块化的关键单元是类，而AOP中模块化的单元是方面。方面支持跨多个类型和对象的关注点（如事务管理）的模块化。（在AOP文献中，这种关注通常被称为“横切”关注。）

Spring的关键组件之一是AOP框架。虽然SpringIOC容器不依赖于AOP（也就是说，如果不想使用AOP，就不需要使用AOP），但是AOP补充了SpringIOC，提供了一个非常强大的中间件解决方案。

> Spring AOP和AspectJ切点
>
> Spring提供了使用基于模式的方法或@Aspectj注解样式来编写自定义方面。这两种风格都提供了完整类型的通知，并可以在使用SpringAOP进行编织的同时使用AspectJ切入点语言。
>
> 本章讨论基于 schema-和@Aspectj的AOP支持。下一章将讨论较低级别的AOP支持。

AOP在Spring框架中被使用，主要用在以下几个方面：

- 提供声明性企业服务。最重要的此类服务是声明性事务管理。
- 让用户实现自定义方面，用AOP补充OOP的使用。

> 如果你只对通用声明性服务或其他预打包的声明性中间件服务（如池）感兴趣，则不需要直接使用SpringAOP，可以跳过本章的大部分内容。

更多教程请访问:www.flydean.com

5.1什么是AOP

让我们首先定义一些核心的AOP概念和术语。这些术语并非特定于Spring。不幸的是，AOP术语并不是特别直观。然而，如果Spring使用自己的术语，则会更加令人困惑。

- 方面（Aspect）：跨越多个类的模块化关注点。事务管理是企业Java应用程序中横切关注点的一个很好的例子。在SpringAOP中，方面是通过使用常规类（基于模式的方法）或使用@Aspect注解（@Aspectj样式）注解的常规类来实现的。
- 连接点（Join point）：程序执行过程中的一点，如方法的执行或异常的处理。在SpringAOP中，连接点总是表示一个方法执行。
- 通知（Advice）：一个方面在特定连接点采取的行动。不同类型的通知包括“环绕”、“前“和”后”通知。（稍后将讨论通知类型。）许多AOP框架（包括Spring）将通知建模为拦截器，并在连接点周围维护拦截器链。
- 切点（Pointcut）：与连接点匹配的谓词。通知与切入点表达式关联，并在与切入点匹配的任何连接点上运行（例如，使用特定名称执行方法）。pointcut表达式匹配的连接点概念是AOP的核心，Spring默认使用AspectJ pointcut表达式语言。
- 说明（Introduction）：代表类型声明其他方法或字段。SpringAOP允许你向任何advised对象引入新的接口（和相应的实现）。例如，你可以使用一个Introduction使bean实现一个IsModified接口，以简化缓存。（introduction在AspectJ社区中称为类型间声明。）
- 目标对象（Target object）：由一个或多个方面advised的对象。也称为“advised 对象”。因为SpringAOP是通过使用运行时代理实现的，所以这个对象始终是一个代理对象。
- AOP代理：由AOP框架创建的用于实现aspect contracts（通知方法执行等）的对象。在Spring框架中，AOP代理是JDK动态代理或CGLIB代理。
- 编织（Weaving）：将aspects与其他应用程序类型或对象链接，以创建advised的对象。这可以在编译时（例如，使用AspectJ编译器）、加载时或运行时完成。Spring AOP和其他纯Java AOP框架一样，在运行时进行编织。

Spring AOP包含以下几种通知类型：

- Before advice:在连接点之前运行但不能阻止执行到连接点的通知（除非它抛出异常）。
- After returning advice:在连接点正常完成后要运行的通知（例如，如果方法返回并且不引发异常）。
- After throwing advice: 如果方法通过引发异常而退出，则要执行的通知。
- After (finally) advice:无论连接点退出的方式如何（正常或异常返回），都要执行的通知。
- Around advice:环绕连接点（如方法调用）的通知。这是最有力的通知。around通知可以在方法调用前后执行自定义行为。它还负责通过返回自己的返回值或引发异常来选择是继续到连接点还是快捷地执行通知的方法。

Around advice是最普遍的advice。由于Spring AOP和AspectJ一样提供了一系列完整的通知类型，我们建议你使用功能最差的通知类型来实现所需的行为。例如，如果只需要使用方法的返回值更新缓存，那么最好实现after returning advice，而不是around advice，尽管around advice可以完成相同的事情。使用最具体的通知类型提供了一个更简单的编程模型，并且错误的可能性更小。例如，你不需要在用于around通知的joinpoint上调用proceed（）方法，因此，你不会调用失败。

所有通知参数都是静态类型的，因此你可以使用适当类型的通知参数（例如，方法执行返回值的类型）而不是对象数组。

切入点匹配的连接点概念是AOP的关键，它将AOP与只提供拦截的旧技术区分开来。切入点使通知能够独立于面向对象的层次结构作为目标。例如，你可以将一个around通知应用到一组跨越多个对象的方法（例如服务层中的所有业务操作）上，从而提供声明性事务管理。

更多教程请访问:www.flydean.com

5.1Spring AOP的能力和目标

Spring AOP是纯Java实现的。不需要特殊的编译过程。SpringAOP不需要控制类加载器层次结构，因此适合在servlet容器或应用程序服务器中使用。

SpringAOP目前只支持方法执行连接点（建议在SpringBean上执行方法）。虽然可以在不破坏核心SpringAOP APIs的情况下添加对字段拦截的支持，但没有实现字段拦截。如果需要通知字段访问和更新连接点，请考虑使用AspectJ之类的语言。

SpringAOP的AOP方法不同于大多数其他AOP框架。目的并不是提供最完整的AOP实现（尽管SpringAOP非常有能力）。相反，其目的是在AOP实现和SpringIOC之间提供紧密的集成，以帮助解决企业应用程序中的常见问题。

因此，例如，Spring框架的AOP功能通常与SpringIOC容器一起使用。Aspects是通过使用普通bean定义语法配置的（尽管这允许强大的“自动代理”功能）。这是与其他AOP实现的关键区别。对于SpringAOP，你不能轻松或高效地执行某些操作，例如通知非常细粒度的对象（通常是域对象）。在这种情况下，AspectJ是最佳选择。然而，我们的经验是，Spring AOP为企业Java应用程序中大多数面向AOP的问题提供了极好的解决方案。

SpringAOP从未试图与AspectJ竞争，以提供全面的AOP解决方案。我们相信，基于代理的框架（如SpringAOP）和全面的框架（如AspectJ）都是有价值的，它们是互补的，而不是竞争中的。Spring无缝地将SpringAOP和IOC与AspectJ集成在一起，以在一致的基于Spring的应用程序体系结构中实现AOP的所有使用。此集成不会影响Spring AOP API或AOP Alliance API。 SpringAOP保持向后兼容。有关SpringAOPAPIs的讨论，请参阅下一章。

> Spring框架的核心原则之一是无侵略性。这是一种观点，即不应强制你将特定于框架的类和接口引入到你的业务或域模型中。但是，在某些地方，Spring框架确实为你提供了将特定于Spring框架的依赖性引入到代码库中的选项。为你提供这些选项的基本原理是，在某些场景中，以这种方式读取或编码某些特定功能可能非常简单。然而，Spring框架（几乎）总是为你提供了选择：你可以自由地做出一个明智的决定，决定哪个选项最适合你的特定用例或场景。
>
> 与本章相关的一个选择是选择哪个AOP框架（以及哪个AOP样式）。你可以选择AspectJ、Spring AOP或两者。你还可以选择@Aspectj注解样式方法或SpringXML配置样式方法。本章选择首先介绍@Aspectj样式方法的事实不应被视为表明Spring团队倾向于使用@Aspectj注解样式方法而不是SpringXML配置样式。
>
> 请参阅选择要使用的AOP声明样式，以更完整地讨论每种样式的“原因和理由”。

更多教程请访问:www.flydean.com

5.3AOP代理

SpringAOP默认为对AOP代理使用标准的JDK动态代理。这样就可以代理任何接口（或一组接口）。

SpringAOP也可以使用CGLIB代理。如果要代理类而不是接口，则必须使用CGLIB。默认情况下，如果业务对象不实现接口，则使用cglib。由于编程到接口而不是类是很好的实践，业务类通常实现一个或多个业务接口。在某些情况下（可能很少），需要通知一个接口上没有声明的方法，或者需要将代理对象作为具体类型传递给方法，则可以强制使用cglib。

重要的是要了解SpringAOP是基于代理的。请参阅了解AOP代理以彻底检查此实现细节的实际含义。

更多教程请访问:www.flydean.com

5.4@AspectJ 支持

@AspectJ是一种将aspects声明为用注解来注解Java类的样式。@Aspectj样式是作为Aspectj 5版本的一部分由Aspectj项目引入的。Spring使用和AspectJ相同的注解，它使用了AspectJ提供的用于切入点解析和匹配的库。但是，AOP运行时仍然是纯SpringAOP，并且不依赖于AspectJ编译器或weaver。

5.4.1 开启@AspectJ 支持

要在Spring配置中使用@Aspectj特性，需要启用Spring支持，以便基于@Aspectj特性配置Spring AOP，并根据这些方面是否通知它们来自动代理bean。通过自动代理，我们的意思是，如果Spring确定bean由一个或多个方面通知，它会自动为该bean生成一个代理来拦截方法调用，并确保根据需要执行通知。

可以使用XML或Java风格的配置启用@AspectJ支持。在这两种情况下，你还需要确保aspectj的aaspectjweaver.jar位于应用程序（1.8版或更高版本）的类路径上。这个库可以在AspectJ发行版的lib目录中使用，也可以从Maven中央存储库中使用。

用Java配置启用@AspectJ支持

若要使用Java @Configuration 启用@AspectJ支持，请添加@EnableAspectJAutoProxy注解，如下示例显示：

```
@Configuration
@EnableAspectJAutoProxy
public class AppConfig {

}
```

使用XML开启@AspectJ

要使用基于XML的配置启用@Aspectj支持，请使用aop:aspectj autoproxy元素，如下示例所示：

```
<aop:aspectj-autoproxy/>
```

这假定你使用基于XML模式的配置中描述的模式支持。请参阅AOP模式以了解如何导入AOP命名空间中的标记。

5.4.2 定义一个Aspect

在启用@Aspectj支持的情况下，应用程序上下文中定义的具有@Aspectj方面（具有@Aspect注解）类的任何bean都将由spring自动检测并用于配置spring aop。接下来的两个例子展示了一个不太有用的方面所需要的最小定义。

两个示例中的第一个示例显示了应用程序上下文中的常规bean定义，该定义指向具有@Aspect注解的bean类：

```
<bean id="myAspect" class="org.xyz.NotVeryUsefulAspect">
    <!-- configure properties of the aspect here -->
</bean>
```

这两个例子中的第二个展示了notveryusefulAspect类定义，它是用org.aspectj.lang.annotation.Aspect注解的：

```
package org.xyz;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class NotVeryUsefulAspect {
```

更多教程请访问:www.flydean.com

```
}
```

方面（用@Aspect注解的类）可以有方法和字段，与任何其他类相同。它们还可以包含切入点、通知和介绍（inter-type）声明。

> 通过 component scanning 自动检查AspectJ
>
> 你可以在SpringXML配置中将方面类注册为常规bean，或者通过类路径扫描-自动检测它们，与任何其他Spring Managed bean相同。但是，请注意，@Aspect注解不足以在类路径中自动检测。为此，你需要添加一个单独的@Component注解（或者，根据Spring组件扫描程序的规则，添加一个符合条件的自定义原型注解）。
>
> 使用其他的aspects来通知aspects?
>
> 在SpringAOP中，方面本身不能成为其他方面通知的目标。类上的@Aspect注解将其标记为方面，因此将其从自动代理中排除。

5.4.3 定义一个Pointcut

切入点决定连接点，从而使我们能够控制何时执行通知。SpringAOP只支持SpringBean的方法执行连接点，因此你可以将切入点视为匹配SpringBean上方法的执行。一个切入点声明有两部分：一个包含一个名称和任何参数的签名，一个能精确地确定我们感兴趣的执行方法的切入点表达式。在aop的@Aspectj注解样式中，通过常规方法定义提供切入点签名，并使用@Pointcut注解指示切入点表达式（作为切入点签名的方法必须具有void返回类型）。

一个例子可能有助于清晰地区分切入点签名和切入点表达式。下面的示例定义一个名为anyOldTransfer的切入点，该切入点与名为Transfer的任何方法的执行相匹配：

```
@Pointcut("execution(* transfer(..))")// the pointcut expression
private void anyOldTransfer() {}// the pointcut signature
```

构成@Pointcut注解值的pointcut表达式是一个正则的aspectj 5 pointcut表达式。有关aspectj的切入点语言的完整讨论，请参阅《aspectj编程指南》（和，有关扩展，请参阅aspectj 5开发人员的笔记本）或其中一本关于aspectj的书（例如，由Colyer等人撰写的Eclipse aspectj，或由Ramnivas Laddad撰写的Aspectj in Action）。

支持的切入点指示符

SpringAOP支持在切入点表达式中使用的以下AspectJ切入点指示符（PCD）：

- execution：用于匹配方法执行连接点。这是使用SpringAOP时要使用的主要切入点指示符。
- within：限制匹配到特定类型中的连接点（使用SpringAOP时，在匹配类型中声明的方法的执行）。
- this：将匹配限制为连接点（使用SpringAOP时方法的执行），其中bean引用（SpringAOP代理）是给定类型的实例。
- target：限制匹配到连接点（使用SpringAOP时方法的执行），其中目标对象（要代理的应用程序对象）是给定类型的实例。
- args：限制匹配到联接点（使用SpringAOP时方法的执行），其中参数是给定类型的实例。
- @target：限制匹配到连接点（使用SpringAOP时方法的执行），其中执行对象的类具有给定类型的注解。
- @args：限制匹配到联接点（使用SpringAOP时方法的执行），其中传递的实际参数的运行时类型具有给定类型的注解。
- @within：限制与具有给定注解的类型中的联接点匹配（使用SpringAOP时，在具有给定注解的类型中声明的方法的执行）。
- @annotation：限制匹配到连接点的主题（在SpringAOP中执行的方法）具有给定注解的连接点。

> 其他的pointcut类型

更多教程请访问:www.flydean.com

> 完整的AspectJ pointcut语言支持Spring中不支持的其他pointcut指示符：call、get、set、preinitialization、staticinitialization、initialization、handler、adviceexecution、withincode、cflow、cflowbelow、if、@this和@withincode。在SpringAOP解释的切入点表达式中使用这些切入点指示符会导致引发IllegalArgumentException。
>
> SpringAOP支持的切入点指示符集可以在未来的版本中扩展，以支持更多的AspectJ切入点指示符。

因为SpringAOP只限制与方法执行连接点匹配，所以前面关于切入点指示符的讨论给出了比AspectJ编程指南中更窄的定义。另外，AspectJ本身具有基于类型的语义，并且在执行连接点，this和target都引用同一个对象：执行方法的对象。SpringAOP是一个基于代理的系统，它区分代理对象本身（与this绑定）和代理背后的目标对象（与target绑定）。

> 由于Spring的AOP框架基于代理的特性，根据定义，目标对象内的调用不会被拦截。对于JDK代理，只能截获对代理的公共接口方法调用。使用cglib，可以截获代理上的公共和受保护的方法调用（如果需要，甚至可以截获包可见的方法）。但是，通过代理进行的公共交互应该始终通过公共签名进行设计。
>
> 注意，切入点定义通常与任何被截取的方法相匹配。如果一个切入点严格来说是只公开的，即使在有可能通过代理进行非公开交互的CGLIB代理场景中，也需要相应地定义它。
>
> 如果拦截需要在目标类中包含方法调用甚至构造函数，那么考虑使用Spring驱动的native AspectJ weaving，而不是Spring的基于代理的AOP框架。这构成了一种不同的AOP使用模式，具有不同的特性，因此在做出决定之前一定要熟悉weaving。

SpringAOP还支持一个额外的PCD bean。这个PCD允许你将连接点的匹配限制为特定名字的Spring Bean或一组Spring Bean（使用通配符时）。bean PCD具有以下形式：

```
bean(idOrNameOfBean)
```

idorNameOfBean令牌可以是任何Spring Bean的名称。提供了使用*字符的有限通配符支持，因此，如果为Spring Bean建立了一些命名约定，则可以编写bean PCD表达式来选择它们。与其他切入点指示符一样，bean pcd可以与&&（and）、||（or）和！（否定）运算符一起使用。

> bean PCD仅在SpringAOP中受支持，在native AspectJ编织中不受支持。它是AspectJ定义的标准PCD的特定于Spring的扩展，因此不能用于@Aspect模型中声明的方面。
>
> bean PCD在实例级别（基于Spring Bean名称概念构建）而不是仅在类型级别（基于编织的AOP受到限制）运行。基于实例的切入点指示符是Spring基于代理的AOP框架的一种特殊功能，它与SpringBean工厂紧密集成，在这里可以自然和简单的按名称标识特定的bean。

组合切入点表达式

你可以使用&&、|| 和！组合切入点表达式。也可以按名称引用切入点表达式。下面的示例显示三个切点表达式：

```
@Pointcut("execution(public * *(..))")
private void anyPublicOperation() {}

@Pointcut("within(com.xyz.someapp.trading..*)")
private void inTrading() {}

@Pointcut("anyPublicOperation() && inTrading()")
private void tradingOperation() {}
```

如前所示，最好使用较小的命名组件构建更复杂的切点表达式。当按名称引用切入点时，使用正常的Java可见性规则（可以在同一类型中看到私有切入点、在层次结构中看到protected切入点、在任何地方看到公共切入点等）。可见性不影响切入点匹配。

共享公共切入点定义

更多教程请访问:www.flydean.com

在处理企业应用程序时，开发人员通常希望从多个方面引用应用程序的模块和特定的操作集。我们建议定义一个“SystemArchitecture”方面，该方面为此目的捕获公共切入点表达式。此类方面通常类似于以下示例：

```
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() {}

}
```

你可以在需要切入点表达式的任何地方引用在这样一个方面中定义的切入点。例如，要使服务层具有事务性，可以编写以下内容：

```
<aop:config>
```

更多教程请访问:www.flydean.com

```
    <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>
```

我们会在基于模式的AOP支持中讨论<aop:config>和&l;taop:advisor>元素。在事务管理中讨论事务元素。

例子

SpringAOP用户最经常使用执行切入点指示符。执行表达式的格式如下：

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?name-pattern(param-pattern)
            throws-pattern?)
```

除返回类型模式（前面代码段中的ret-type-pattern）、名称模式和参数模式之外的所有部分都是可选的。返回类型模式确定方法的返回类型，以便匹配连接点。*最常用作返回类型模式。它匹配任何返回类型。只有当方法返回给定类型时，完全限定的类型名才匹配。名称模式与方法名匹配。你可以使用*通配符作为名称模式的全部或部分。如果指定声明类型模式，请包含后缀.将其连接到名称模式组件。参数模式稍微复杂一点：（）匹配不带参数的方法，而（..）匹配任何数量（零个或多个）的参数。（*）模式与采用任何类型参数的方法匹配。（*，string）匹配接受两个参数的方法。第一个可以是任何类型，而第二个必须是字符串。有关更多信息，请参阅AspectJ编程指南的语言语义部分。

以下示例显示了一些常见的切入点表达式：

- 执行任何公共方法：

  ```
  execution(public * *(..))
  ```

- 执行任何以set开头的方法：

  ```
  execution(* set*(..))
  ```

- 执行任何定义在AccountService类的方法：

  ```
  execution(* com.xyz.service.AccountService.*(..))
  ```

- 执行任何定义在service包中的方法：

  ```
  execution(* com.xyz.service.*.*(..))
  ```

- 执行任何定义在service包或者他的一个子包中的方法：

  ```
  execution(* com.xyz.service..*.*(..))
  ```

- 任何在service包中的连接点（仅仅是Spring AOP中执行的方法）

  ```
  within(com.xyz.service.*)
  ```

- service包或其子包中的任何连接点（仅在SpringAOP中执行的方法）：

更多教程请访问:www.flydean.com

```
within(com.xyz.service..*)
```

- 任何实现了AccountService接口的代理连接点（仅在SpringAOP中执行的方法）：

```
this(com.xyz.service.AccountService)
```

> this更加通常被用在绑定form中。

- 任何实现了AccountService接口的目标对象连接点（仅在SpringAOP中执行的方法）：

```
target(com.xyz.service.AccountService)
```

> target更加通常被用在绑定form中。

- 任何接收一个运行时是Serializable类型参数的连接点（仅在SpringAOP中执行的方法）：

```
args(java.io.Serializable)
```

> args更加通常被用在绑定form中。

注意，在本例中给出的切入点与execution(__(java.io.Serializable))不同。如果在运行时传递的参数是Serializable的，则args版本匹配；如果方法签名声明了Serializable的单个参数，则执行版本匹配。

- 任何目标对象有@Transactional 注解的连接点（仅在SpringAOP中执行的方法）：

```
@target(org.springframework.transaction.annotation.Transactional)
```

> @target也可以被用在绑定form中。

- 目标对象的声明类型具有@transactional注解的任何连接点（仅在Spring AOP中执行的方法）：

```
@within(org.springframework.transaction.annotation.Transactional)
```

> @within也可以被用在绑定form中。

- 执行方法具有@transactional注解的任何连接点（仅在Spring AOP中执行方法）：

```
@annotation(org.springframework.transaction.annotation.Transactional)
```

> @annotation也可以被用在绑定form中。

- 接受单个参数的任何连接点（仅在Spring AOP中执行方法），并且传递参数的运行时类型具有@Classified annotation:

```
@args(com.xyz.security.Classified)
```

> @args也可以被用在绑定form中。

- 名为tradeService的SpringBean上的任何连接点（仅在SpringAOP中执行方法）：

```
bean(tradeService)
```

- SpringBean上的任何连接点（仅在SpringAOP中执行方法），其名称与通配符表达式*Service匹配：

更多教程请访问:www.flydean.com

```
bean(*Service)
```

写好切入点

在编译期间，AspectJ处理切入点以优化匹配性能。检查代码并确定每个连接点是否匹配（静态或动态）给定的切入点是一个代价高昂的过程。（动态匹配意味着无法从静态分析中完全确定匹配，并且在代码中放置一个测试以确定代码运行时是否存在实际匹配）。在第一次遇到切入点声明时，AspectJ将其重写为匹配过程的最佳形式。这是什么意思？基本上，切入点是用DNF（析取标准形式）重写的，切入点的组件被排序，以便首先检查那些评估成本更低的组件。这意味着你不必担心理解各种切入点指示符的性能，并且可以在切入点声明中以任何顺序提供它们。

然而，Aspectj只能处理它被告知的内容。为了获得最佳的匹配性能，你应该考虑他们试图实现什么，并在定义中尽可能地缩小匹配的搜索空间。现有的指示符自然分为三类：kinded, scoping, 和 contextual：

- kinded指示符选择特定类型的连接点：execution, get, set, call, 和 handler.。
- scoping指示符选择一组感兴趣的连接点（可能有多种）：within和withincode
- contextual指示符根据上下文匹配（可选绑定）：this、target和@annotation

写得好的切入点应该至少包括前两种类型（kinded和scoping）。你可以根据连接点上下文包含要匹配的上下文指示符，或者绑定该上下文以在通知中使用。由于额外的处理和分析，可以仅提供一个类似的指示符或上下文指示符，但可能会影响编织性能（使用的时间和内存）。范围指示符匹配非常快，使用它们意味着AspectJ可以很快地消除不应该进一步处理的连接点组。如果可能的话，一个好的切入点应该总是包括一个。

5.4.4 声明通知

通知与切入点表达式关联，并在切入点匹配的方法执行before, after, 或 around 运行。切入点表达式可以是对命名切入点的简单引用，也可以是就地声明的切入点表达式。

Before Advice

可以使用@before在方面中声明before advice:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class BeforeExample {

    @Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
    public void doAccessCheck() {
        // ...
    }

}
```

如果使用inline切入点表达式，可以将前面的示例重写为以下示例：

```
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() {
        // ...
    }

}
```

更多教程请访问:www.flydean.com

After Returning Advice

当一个匹配的方法正常返回时执行After returning advice。你可以通过@AfterReturning注解来声明：

```
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() {
        // ...
    }

}
```

你可以在同一方面内拥有多个通知声明（以及其他成员）。在这些示例中，我们只显示一个通知声明，以集中显示每个通知声明的效果。

有时，你需要访问通知主体中返回的实际值。你可以使用@AfterReturn的形式绑定返回值以获得该访问，如下示例所示：

```
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) {
        // ...
    }

}
```

returning属性中使用的名称必须与advice方法中参数的名称相对应。当方法执行返回时，返回值作为相应的参数值传递给通知方法。返回子句还限制只匹配那些返回指定类型值的方法执行（在本例中是Object，它匹配任何返回值）。

请注意，在使用after returning advice时，不可能返回完全不同的引用。

After Throwing Advice

当匹配的方法抛出异常退出时，After throwing advice执行，你可以使用@AfterThrowing来声明它：

```
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() {
        // ...
    }

}
```

更多教程请访问:www.flydean.com

通常，你希望只有在抛出给定类型的异常时才运行通知，并且通常还需要访问通知正文中抛出的异常。可以使用throwing属性来限制匹配（如果需要-使用throwable作为异常类型），并将引发的异常绑定到advice参数。以下示例显示了如何执行此操作：

```
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) {
        // ...
    }

}
```

throwing属性中使用的名称必须与advice方法中参数的名称相对应。当通过引发异常退出方法执行时，异常作为相应的参数值传递给通知方法。throwing子句还限制只匹配那些引发指定类型的异常（在本例中是DataAccessException）的方法执行。

After (Finally) Advice

After (Finally) Advice当匹配的方法执行退出时执行，通过@After注解来定义。After advice 必须能够处理正常返回和异常返回的情况。它通常用于释放资源和类似用途。下面的示例演示如何在finally advice之后使用：

```
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

最后一种advice是around advice的。around advice运行“around”匹配方法的执行。它有机会在方法执行之前和之后都进行工作，并确定何时、如何以及即使该方法真正开始执行。如果需要以线程安全的方式（例如，启动和停止计时器）共享方法执行前后的状态，则通常使用around建议。始终使用满足你要求的最不强大的advice形式（也就是说，如果在around之前不使用around）。

使用@Around注解声明around通知。advice方法的第一个参数必须是ProceedingJoinPoint类型。在通知正文中，对ProceedingJoinPoint调用proceed（）会导致执行基础方法。proceed方法也可以传入Object[]。数组中的值在方法执行过程中用作参数。

> 当用Object[] 调用时，proceed的行为与由aspectj编译器编译的around advice的proceed的行为稍有不同。对于使用传统aspectj语言编写的around advice，传递给process的参数数量必须与传递给around advice的参数数量（而不是基础连接点接受的参数数量）匹配，并且传递给给定参数位置的值在连接点为值所绑定到的实体添加原始值（如果现在没有意义，请不要担心）。Spring采用的方法更简单，更符合其基于代理的、只执行的语义。只有在编译为Spring编写的@Aspectj方面并使用aspectj编译器和weaver的参数时，才需要注意这一区别。有一种方法可以在SpringAOP和AspectJ中编写100%兼容的方面，这将在下面关于建议参数的小节中讨论。

下面的例子展示了如何使用around advice：

更多教程请访问:www.flydean.com

```
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;
    }

}
```

around通知返回的值是方法调用方看到的返回值。例如，一个简单的缓存方面可以从缓存中返回一个值（如果有），如果没有，则调用proceed（）。请注意，可以在around建议的主体中调用proceed一次、多次或根本不调用proceed。所有这些都是合法的。

dvice Parameters

Spring提供了完全类型化的advice，这意味着你声明了advice签名中所需的参数（正如前面我们在返回和抛出示例中看到的那样），而不是一直使用Object[] 数组。我们将在本节后面的部分中了解如何使参数和其他上下文值对advice主体可用。首先，我们来看一下如何编写通用advice，以了解advice当前建议的方法。

访问当前JoinPoint

任何advice方法都可以将org.aspectj.lang.joinpoint类型的参数声明为其第一个参数（请注意，需要使用around advice从而声明ProceedingJoinPoint类型作为第一个参数，该参数是JoinPoint的子类）。JoinPoint接口提供了许多有用的方法：

- getArgs(): 返回方法参数
- getThis(): 返回代理对象
- getTarget(): 返回目标对象
- getSignature(): 返回被advice的方法描述
- toString():打印被advice方法的有用描述

传递参数给advice

我们已经看到了如何绑定返回值或异常值（在返回后和抛出advice后使用）。要使参数值对通知主体可用，可以使用参数的绑定形式。如果在args表达式中使用参数名代替类型名，则在调用通知时，相应参数的值将作为参数值传递。一个例子应该更清楚地说明这一点。假设你想要建议以account对象为第一个参数的DAO操作的执行，并且你需要访问通知主体中的帐户。你可以编写以下内容：

```
@Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation() && args(account,..)")
public void validateAccount(Account account) {
    // ...
}
```

pointcut中的args(account,..)部分有两个主要目的：首先，它将匹配限制为只执行那些方法，其中该方法至少接受一个参数，并且传递给该参数的参数是帐户的实例。其次，它通过account参数使实际的account对象可用于advice。

另一种编写方法是声明一个切入点，该切入点在与连接点匹配时“提供”account对象值，然后从通知中引用命名的切入点。如下所示：

```
@Pointcut("com.xyz.myapp.SystemArchitecture.dataAccessOperation() && args(account,..)")
private void accountDataAccessOperation(Account account) {}
```

更多教程请访问:www.flydean.com

```
@Before("accountDataAccessOperation(account)")
public void validateAccount(Account account) {
    // ...
}
```

代理对象this，目标对象 target，和注解（@within, @target, @annotation, 和 @args） 都可以用类似的方式装订。接下来的两个示例演示如何匹配用@Auditable注解的方法的执行，并提取审计代码：

两个例子中的第一个展示了@Auditable注解：

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @Interface Auditable {
    AuditCode value();
}
```

第二个例子是匹配执行@Auditable方法的advice：

```
@Before("com.xyz.lib.Pointcuts.anyPublicMethod() && @annotation(auditable)")
public void audit(Auditable auditable) {
    AuditCode code = auditable.value();
    // ...
}
```

Advice参数和泛型

SpringAOP可以处理类声明和方法参数中使用的泛型。假设你有一个如下的泛型类型：

```
public interface Sample<T> {
    void sampleGenericMethod(T param);
    void sampleGenericCollectionMethod(Collection<T> param);
}
```

通过将advice参数键入要拦截方法的参数类型，可以将方法类型的拦截限制为某些参数类型：

```
@Before("execution(* ..Sample+.sampleGenericMethod(*)) && args(param)")
public void beforeSampleMethod(MyType param) {
    // Advice implementation
}
```

这种方法不适用于泛型集合。因此，不能按以下方式定义切入点：

```
@Before("execution(* ..Sample+.sampleGenericCollectionMethod(*)) && args(param)")
public void beforeSampleMethod(Collection<MyType> param) {
    // Advice implementation
}
```

为了实现这一点，我们必须检查集合中的每一个元素，这是不合理的，因为我们也不能决定一般如何处理空值。要实现与此类似的功能，必须键入collection<? >手动检查元素的类型。

确定参数名称

advice调用中的参数绑定依赖于切入点表达式中使用的匹配名称与通知和切入点方法签名中声明的参数名称。在Java反射中，参数名是不可用的，因此Spring AOP使用以下策略来确定参数名称：

- 如果用户已显式指定参数名，则使用指定的参数名。通知和切入点注解都有一个可选的argNames属性，你可以使用它来指定带注解的方法的参数名。这些参数名在运行时可用。下面的示例演示如何使用argNames属性：

更多教程请访问:www.flydean.com

```
@Before(value="com.xyz.lib.Pointcuts.anyPublicMethod() && target(bean) && @annotation(auditable)",
        argNames="bean,auditable")
public void audit(Object bean, Auditable auditable) {
    AuditCode code = auditable.value();
    // ... use code and bean
}
```

如果第一个参数是joinpoint、ProceedingJoinPoint或JoinPoint.StaticPart类型，则可以从argnames属性的值中省略参数的名称。例如，如果修改前面的建议以接收连接点对象，argnames属性不需要包括它：

```
@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
}
```

joinPoint、proceedingjoinPoint和joinPoint.staticPart类型作为第一个参数的特殊处理对于不收集任何其他连接点上下文的advice实例特别方便。在这种情况下，可以省略argNames属性。例如，以下建议不需要声明argNames属性：

```
@Before("com.xyz.lib.Pointcuts.anyPublicMethod()")
public void audit(JoinPoint jp) {
    // ... use jp
}
```

使用“argNames”属性有点笨拙，因此如果没有指定“argNames”属性，spring aop将查看类的调试信息，并尝试从局部变量表中确定参数名。只要使用调试信息编译了类（至少为-g:vars），就会出现此信息。使用此标志编译的结果是：（1）代码更容易理解（反向工程）；（2）类文件的大小稍大（通常不连续）；（3）编译器不应用删除未使用的局部变量的优化。换言之，你不应该因为这个flag而遇到困难。

> 如果aspectj编译器（ajc）编译了@Aspectj aspect，即使没有调试信息，也不需要添加argnames属性，因为编译器保留了所需的信息。

如果在没有必要的调试信息的情况下编译了代码，那么spring aop会尝试推断绑定变量与参数之间的配对（例如，如果切入点表达式中只绑定了一个变量，而advice方法只接受一个参数，那么这种配对是明显的）。如果给定可用信息，变量的绑定不明确，则抛出一个AmbiguousBindingException。

如果上述所有策略都失败，将引发IllegalArgumentException。

Proceeding with Arguments

我们之前提到过，我们将描述如何使用在SpringAOP和AspectJ中一致工作的参数编写一个proceed调用。解决方案是确保通知签名按顺序绑定每个方法参数。以下示例显示了如何执行此操作：

```
@Around("execution(List<Account> find*(..)) && " +
        "com.xyz.myapp.SystemArchitecture.inDataAccessLayer() && " +
        "args(accountHolderNamePattern)")
public Object preProcessQueryPattern(ProceedingJoinPoint pjp,
        String accountHolderNamePattern) throws Throwable {
    String newPattern = preProcess(accountHolderNamePattern);
    return pjp.proceed(new Object[] {newPattern});
}
```

在许多情况下，你无论如何都要进行这种绑定（如前面的示例中所示）。

Advice Ordering

更多教程请访问:www.flydean.com

当多条advice都想在同一个连接点上运行时会发生什么？SpringAOP遵循与AspectJ相同的优先级规则来确定通知执行的顺序。最高优先级的通知在“on the way in”首先运行（因此，给定两个before advice，最高优先级的advice首先运行）。从连接点“On the way out”时，最高优先级的通知运行在最后（因此，给定两条after advice，最高优先级的通知将运行在第二）。

当在不同aspects定义的两条advice都需要在同一连接点上运行时，除非你另有指定，否则执行顺序是未定义的。你可以通过指定优先级来控制执行顺序。这可以通过在Aspect类中实现org.springframework.core.Ordered接口或使用order注解来以正常的Spring方式完成。给定两个aspects，从ordered.getValue（）返回较低值（或注解值）的方面具有较高的优先级。

当在同一aspect定义的两条advice都需要在同一个连接点上运行时，顺序是未定义的（因为没有办法通过对javac编译类的反射来检索声明顺序）。考虑在每个方面类的每个连接点将这些建议方法折叠成一个建议方法，或者将这些建议重构成单独的方面类，你可以在方面级别对它们进行排序。

5.4.5 Introductions

Introductions（在AspectJ中称为类型间声明）使方面能够声明实现给定的接口的建议对象，并代表这些对象提供该接口的实现。

你可以使用@DeclareParents注解进行介绍。此注解用于声明具有新的父级的匹配类型（因此是名称）。例如，给定一个名为UsageTracked的接口和一个名为DefaultUsageTracked的接口的实现，下面的方面声明服务接口的所有实现者也实现UsageTracked接口（例如，通过jmx公开统计信息）：

```
@Aspect
public class UsageTracking {

    @DeclareParents(value="com.xzy.myapp.service.*+", defaultImpl=DefaultUsageTracked.class)
    public static UsageTracked mixin;

    @Before("com.xyz.myapp.SystemArchitecture.businessService() && this(usageTracked)")
    public void recordUsage(UsageTracked usageTracked) {
        usageTracked.incrementUseCount();
    }

}
```

要实现的接口由注解字段的类型决定。@DeclareParents注解的value属性是AspectJ类型的模式。匹配任何实现UsageTracked接口的bean类型。注意，在前面的示例中，服务bean可以直接用作UsageTracked接口的实现。如果以编程方式访问bean，你将编写以下内容：

```
UsageTracked usageTracked = (UsageTracked) context.getBean("myService");
```

5.4.6 方面实例化模型

> 这是一个高级主题。如果你刚开始使用AOP，可以安全地跳过它，直到稍后。

默认情况下，应用程序上下文中每个方面都有一个实例。Aspectj将其称为单例实例化模型。可以用交替的生命周期定义方面。Spring支持AspectJ的perthis和pertarget实例化模型（percflow、percflowbelow, 和 pertypewithin当前不受支持）。

你可以通过在@Aspect注解中指定perthis子句来声明perthis方面。请考虑以下示例：

```
@Aspect("perthis(com.xyz.myapp.SystemArchitecture.businessService())")
public class MyAspect {
```

更多教程请访问:www.flydean.com

```
    private int someState;

    @Before(com.xyz.myapp.SystemArchitecture.businessService())
    public void recordServiceUsage() {
        // ...
    }

}
```

在前面的示例中，“perthis”子句的效果是为执行业务服务的每个唯一服务对象（在与切入点表达式匹配的连接点处绑定到“this”的每个唯一对象）创建一个方面实例。第一次对服务对象调用方法时，将创建方面实例。当服务对象超出范围时，方面超出范围。在创建方面实例之前，它内的任何advice都不会执行。只要创建了方面实例，它内部声明的advice就在匹配的连接点上执行，但仅当服务对象是与此方面关联的对象时。有关每个子句的详细信息，请参阅《AspectJ编程指南》。

pertarget实例化模型的工作方式与perthis完全相同，但它在匹配的连接点为每个唯一的目标对象创建一个方面实例。

5.4.7 一个AOP的例子

既然你已经了解了所有组成部分的工作原理，那么我们可以将它们放在一起做一些有用的事情。

业务服务的执行有时会由于并发问题（例如，死锁失败者）而失败。如果重试该操作，则下次尝试可能会成功。对于适合在这种情况下重试的业务服务（不需要返回用户进行冲突解决的等量操作），我们希望透明地重试该操作，以避免客户端看到PessimisticLockingFailureException。这是一个需求，它清楚地跨越服务层中的多个服务，因此是通过一个方面实现的理想选择。

因为我们想重试这个操作，所以我们需要使用around建议，这样我们可以多次调用proceed。下面的列表显示了基本方面的实现：

```
@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);
```

更多教程请访问:www.flydean.com

```
        throw lockFailureException;
    }

}
```

注意，aspect实现了Ordered的接口，因此我们可以将方面的优先级设置为高于事务通知（我们希望每次重试时都有一个新的事务）。maxRetries和order属性都由spring配置。主要操作发生在around advice的doConcurrentOperation中。请注意，目前，我们将重试逻辑应用于每个BusinessService（）。我们尝试继续，如果我们因PessimisticLockingFailureException而失败，我们会再试一次，除非我们已经用尽了所有的重试尝试。

相应的Spring配置如下：

```
<aop:aspectj-autoproxy/>

<bean id="concurrentOperationExecutor" class="com.xyz.myapp.service.impl.ConcurrentOperationExecutor">
    <property name="maxRetries" value="3"/>
    <property name="order" value="100"/>
</bean>
```

为了优化方面以便它只重试Idempotent运算，我们可以定义以下Idempotent注解：

```
@Retention(RetentionPolicy.RUNTIME)
public @Interface Idempotent {
    // marker annotation
}
```

然后，我们可以使用注解来注解服务操作的实现。将方面更改为仅重试idempotent操作涉及到的切入点表达式，以便只有@Idempotent操作匹配，如下所示：

```
@Around("com.xyz.myapp.SystemArchitecture.businessService() && " +
        "@annotation(com.xyz.myapp.service.Idempotent)")
public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {
    ...
}
```

更多教程请访问:www.flydean.com

5.5基于Schema的AOP支持

如果你喜欢基于XML的格式，Spring还提供了使用新的AOP名称空间标记定义方面的支持。支持与使用@Aspectj样式时完全相同的切入点表达式和通知类型。因此，在本节中，我们将重点介绍新的语法，并将读者引向上一节（@Aspectj support）中的讨论，以了解如何编写切入点表达式以及如何绑定通知参数。

要使用本节中描述的AOP命名空间标记，需要导入spring-aop schema，如基于XML模式的配置中所述。请参阅AOP模式以了解如何导入AOP命名空间中的标记。

在Spring配置中，所有Aspect和Advisor元素都必须放在<aop:config>元素中（在应用程序上下文配置中可以有多个<aop:config>元素）。<aop:config>元素可以包含pointcut、advisor和aspect元素（请注意，这些元素必须按该顺序声明）。

> 配置的样式大量使用了Spring的自动代理机制。如果你已经通过使用BeanNameAutoProxyCreator或类似的方法使用显式的自动代理，这可能会导致问题（例如建议没有被编织）。建议的使用模式是只使用<aop:config>样式或只使用AutoProxyCreator样式，并且永远不要混合它们。

5.5.1 定义一个Aspect

当使用schema支持时，一个方面是定义在你的Spring应用程序上下文中的规则Java bean对象。状态和行为捕获在对象的字段和方法中，切入点和通知信息捕获在XML中。

你可以使用<aop:aspect>元素声明一个方面，并使用ref属性引用支持bean，如下示例所示：

```
<aop:config>
    <aop:aspect id="myAspect" ref="aBean">
        ...
    </aop:aspect>
</aop:config>

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

支持方面的bean（在本例中是aBean）当然可以像其他任何Springbean一样配置和注入依赖项。

5.5.2 定义Pointcut

你可以在<aop:config>元素中声明一个命名的切入点，让切入点定义在多个aspects和advisors之间共享。

表示服务层中任何业务服务的执行的切入点可以定义如下：

```
<aop:config>

    <aop:pointcut id="businessService"
        expression="execution(* com.xyz.myapp.service.*.*(..))"/>

</aop:config>
```

注意，切入点表达式本身使用的是与@Aspectj支持中描述的相同的aspectj切入点表达式语言。如果使用基于schema的声明样式，则可以引用在切入点表达式中的类型（@Aspects）中定义的命名切入点。定义上述切入点的另一种方法如下：

更多教程请访问:www.flydean.com

```
<aop:config>

    <aop:pointcut id="businessService"
        expression="com.xyz.myapp.SystemArchitecture.businessService()"/>

</aop:config>
```

假设你有一个SystemArchitecture方面，如共享公共切入点定义中所述。

然后在方面中声明一个切入点与声明一个顶级切入点非常相似，如下示例所示：

```
<aop:config>

    <aop:aspect id="myAspect" ref="aBean">

        <aop:pointcut id="businessService"
            expression="execution(* com.xyz.myapp.service.*.*(..))"/>

        ...

    </aop:aspect>

</aop:config>
```

与@Aspectj方面非常相似，使用基于schema的定义样式声明的切入点可以收集连接点上下文。例如，以下切入点将此对象收集为连接点上下文，并将其传递给通知：

```
<aop:config>

    <aop:aspect id="myAspect" ref="aBean">

        <aop:pointcut id="businessService"
            expression="execution(* com.xyz.myapp.service.*.*(..)) &amp;&amp; this(service)"/>

        <aop:before pointcut-ref="businessService" method="monitor"/>

        ...

    </aop:aspect>

</aop:config>
```

通知必须声明为通过包含匹配名称的参数来接收收集的连接点上下文，如下所示：

```
public void monitor(Object service) {
    ...
}
```

当组合切入点子表达式时，&&在XML文档中很难使用，因此可以使用and、or和not关键字来代替&& || 和！。例如，前面的切入点可以更好地写如下：

```
<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"/>

        ...
```

更多教程请访问:www.flydean.com

```
    </aop:aspect>
</aop:config>
```

请注意，以这种方式定义的切入点由其XML ID引用，不能用作命名切入点以形成复合切入点。因此，schema-based定义样式中的命名切入点支持比@Aspectj样式提供的支持更为有限。

5.5.3 定义Advice

schema-based AOP 支持使用与@Aspectj样式相同的五种建议，它们具有完全相同的语义。

Before Advice

在匹配的方法执行之前运行通知。它通过在<aop:aspect>中声明使用的<aop:before>元素，如下示例所示：

```
<aop:aspect id="beforeExample" ref="aBean">

    <aop:before
        pointcut-ref="dataAccessOperation"
        method="doAccessCheck"/>

    ...

</aop:aspect>
```

这里dataAccessOperation是顶级（<aop:config>）中定义的切入点id，要以内联方式定义切入点，请使用pointcut属性替换pointcut-ref属性，如下所示：

```
<aop:aspect id="beforeExample" ref="aBean">

    <aop:before
        pointcut="execution(* com.xyz.myapp.dao.*.*(..))"
        method="doAccessCheck"/>

    ...

</aop:aspect>
```

正如我们在讨论@Aspectj样式时所指出的，使用命名的切入点可以显著提高代码的可读性。

method属性标识提供建议主体的方法（doAccessCheck）。必须为包含通知的Aspect元素引用的bean定义此方法。在执行数据访问操作（与切入点表达式匹配的方法执行连接点）之前，将调用方面bean上的doAccessCheck方法。

After Returning Advice

After Returning Advice，在匹配的方法执行正常完成时运行。它在一个<aop:aspect>中声明，方式与之前的通知相同。下面的示例演示如何声明它：

```
<aop:aspect id="afterReturningExample" ref="aBean">

    <aop:after-returning
        pointcut-ref="dataAccessOperation"
        method="doAccessCheck"/>

    ...

</aop:aspect>
```

正如在@Aspectj样式中一样，你可以在通知正文中获得返回值。为此，请使用返回属性来指定应将返回值传递到的参数的名称，如下示例所示：

更多教程请访问:www.flydean.com

```
<aop:aspect id="afterReturningExample" ref="aBean">

    <aop:after-returning
        pointcut-ref="dataAccessOperation"
        returning="retVal"
        method="doAccessCheck"/>

    ...

</aop:aspect>
```

doAccessCheck方法必须要有声明名为retval的参数。此参数的类型以与@AfterReturning相同的方式约束匹配。例如，可以按如下方式声明方法签名：

```
public void doAccessCheck(Object retVal) {...
```

After Throwing Advice

当匹配的方法执行通过引发异常退出时，在引发通知后执行。它通过使用after-throwing 元素在<aop:aspect>中声明，如下示例所示：

```
<aop:aspect id="afterThrowingExample" ref="aBean">

    <aop:after-throwing
        pointcut-ref="dataAccessOperation"
        method="doRecoveryActions"/>

    ...

</aop:aspect>
```

正如在@Aspectj样式中一样，你可以在通知正文中得到抛出的异常。要执行此操作，请使用引发属性指定异常应传递到的参数的名称，如下示例所示：

```
<aop:aspect id="afterThrowingExample" ref="aBean">

    <aop:after-throwing
        pointcut-ref="dataAccessOperation"
        throwing="dataAccessEx"
        method="doRecoveryActions"/>

    ...

</aop:aspect>
```

doRecoveryActions方法必须有声明名为DataAccessEx的参数。此参数的类型约束匹配的方式与@AfterThrowing中描述的方式相同。例如，方法签名可以声明如下：

```
public void doRecoveryActions(DataAccessException dataAccessEx) {...
```

After (Finally) Advice

无论匹配的方法执行如何退出，after（finally）通知都会运行。可以使用after元素声明它，如下示例所示：

```
<aop:aspect id="afterFinallyExample" ref="aBean">

    <aop:after
        pointcut-ref="dataAccessOperation"
        method="doReleaseLock"/>
```

更多教程请访问:www.flydean.com

```
    ...

</aop:aspect>
```

Around Advice

最后一种advice是around advice的。around通知运行“around”匹配的方法执行。它有机会在方法执行之前和之后都进行工作，并确定何时、如何以及该方法真正开始执行。around advice通常用于以线程安全的方式（例如，启动和停止计时器）共享方法执行前后的状态。始终使用满足你要求的最不强大的建议形式。如果在before advice可以完成工作，不要使用around advice。

你可以使用aop:around元素来声明around advice。advice方法的第一个参数必须是ProceedingJoinPoint类型。在通知正文中，对ProceedingJoinPoint调用proceed（）会导致执行基础方法。也可以使用Object[] 调用proceed方法。数组中的值在方法执行过程中用作参数。有关调用proceed使用Object[]的说明，请参阅Around advice。下面的示例演示如何在XML中声明around advice：

```
<aop:aspect id="aroundExample" ref="aBean">

    <aop:around
        pointcut-ref="businessService"
        method="doBasicProfiling"/>

    ...

</aop:aspect>
```

doBasicProfiling advice的实现可以与@Aspectj示例中的完全相同（当然，可以减去注解），如下示例所示：

```
public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {
    // start stopwatch
    Object retVal = pjp.proceed();
    // stop stopwatch
    return retVal;
}
```

Advice参数

schema-based的声明样式支持完全类型化的通知，方法与@Aspectj support中描述的方法相同，方法是按名称将切入点参数与通知方法参数匹配。有关详细信息，请参阅Advice 参数。如果你希望显式地为advice方法指定参数名（不依赖于前面描述的检测策略），可以使用advice元素的arg-names属性来指定参数名，该属性与advice注解中的argNames属性的处理方式相同。下面的示例演示如何以XML格式指定参数名：

```
<aop:before
    pointcut="com.xyz.lib.Pointcuts.anyPublicMethod() and @annotation(auditable)"
    method="audit"
    arg-names="auditable"/>
```

arg-names属性接受以逗号分隔的参数名称列表。

下面这个基于XSD的方法的稍微复杂一些的示例显示了一些与强类型参数一起使用的around advice：

```
package x.y.service;

public interface PersonService {

    Person getPerson(String personName, int age);
}
```

更多教程请访问:www.flydean.com

```
public class DefaultFooService implements FooService {

    public Person getPerson(String name, int age) {
        return new Person(name, age);
    }
}
```

接下来是方面。请注意，profile（..）方法接受许多强类型参数，其中第一个参数恰好是用于继续方法调用的连接点。此参数的存在表示profile（..）将用作around advice，如下示例所示：

```
package x.y;

import org.aspectj.lang.ProceedingJoinPoint;
import org.springframework.util.StopWatch;

public class SimpleProfiler {

    public Object profile(ProceedingJoinPoint call, String name, int age) throws Throwable {
        StopWatch clock = new StopWatch("Profiling for '" + name + "' and '" + age + "'");
        try {
            clock.start(call.toShortString());
            return call.proceed();
        } finally {
            clock.stop();
            System.out.println(clock.prettyPrint());
        }
    }
}
```

最后，下面的XML配置示例将影响对特定连接点执行前面的建议：

```
<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 https://www.springframework.org/schema/beans/spring-beans.x
sd
        http://www.springframework.org/schema/aop https://www.springframework.org/schema/aop/spring-aop.xsd">

    <!-- this is the object that will be proxied by Spring's AOP infrastructure -->
    <bean id="personService" class="x.y.service.DefaultPersonService"/>

    <!-- this is the actual advice itself -->
    <bean id="profiler" class="x.y.SimpleProfiler"/>

    <aop:config>
        <aop:aspect ref="profiler">

            <aop:pointcut id="theExecutionOfSomePersonServiceMethod"
                expression="execution(* x.y.service.PersonService.getPerson(String,int))
                and args(name, age)"/>

            <aop:around pointcut-ref="theExecutionOfSomePersonServiceMethod"
                method="profile"/>

        </aop:aspect>
    </aop:config>

</beans>
```

考虑下面的驱动脚本：

```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.ClassPathXmlApplicationContext;
```

更多教程请访问:www.flydean.com

```
import x.y.service.PersonService;

public final class Boot {

    public static void main(final String[] args) throws Exception {
        BeanFactory ctx = new ClassPathXmlApplicationContext("x/y/plain.xml");
        PersonService person = (PersonService) ctx.getBean("personService");
        person.getPerson("Pengo", 12);
    }
}
```

有了这样的引导类，我们将得到与标准输出类似的输出：

```
StopWatch 'Profiling for 'Pengo' and '12'': running time (millis) = 0
---------------------------------------
ms     %     Task name
---------------------------------------
00000  ?  execution(getFoo)
```

Advice Ordering

当需要在同一连接点（执行方法）执行多个通知时，排序规则如通知排序中所述。方面之间的优先级通过向支持方面的bean添加order注解或让bean实现Ordered接口来确定。

5.5.4 Introductions

介绍（在AspectJ中称为类型间声明）让Aspect声明建议的对象实现给定的接口，并代表这些对象提供该接口的实现。

你可以通过在aop:aspect中使用aop:declare-parents元素进行介绍。你可以使用aop:declare-parents元素来声明匹配类型具有新的父级（因此是名称）。例如，给定一个名为UsageTracked的接口和一个名为DefaultUsageTracked的接口的实现，下面的方面声明服务接口的所有实现者也实现UsageTracked接口。（例如，为了通过JMX公开统计信息。）

```
<aop:aspect id="usageTrackerAspect" ref="usageTracking">

    <aop:declare-parents
        types-matching="com.xzy.myapp.service.*+"
        implement-interface="com.xyz.myapp.service.tracking.UsageTracked"
        default-impl="com.xyz.myapp.service.tracking.DefaultUsageTracked"/>

    <aop:before
        pointcut="com.xyz.myapp.SystemArchitecture.businessService()
            and this(usageTracked)"
            method="recordUsage"/>

</aop:aspect>
```

支持usageTracking bean的类随后将包含以下方法：

```
public void recordUsage(UsageTracked usageTracked) {
    usageTracked.incrementUseCount();
}
```

要实现的接口由implement-interface属性确定。types-matching属性的值是AspectJ类型模式。匹配任何实现UsageTracked接口的bean类型。注意，在前面的示例中，服务bean可以直接用作UsageTracked接口的实现。要以编程方式访问bean，可以编写以下内容：

```
UsageTracked usageTracked = (UsageTracked) context.getBean("myService");
```

更多教程请访问:www.flydean.com

5.5.5 方面实例化模型

对于schema-defined 的方面，唯一支持的实例化模型是单例模型。在未来的版本中可能支持其他实例化模型。

5.5.6 Advisors

“Advisors”的概念来自于Spring中定义的AOP支持，在AspectJ中没有直接的等价物。Advisors就像一个独立的小方面，只有一条建议。通知本身由bean表示，并且必须实现在Spring的通知类型中描述的通知接口之一。Advisors可以利用AspectJ切入点表达式。

Spring使用<aop:advisor>元素支持Advisor概念。你通常会看到它与事务性advice结合使用，后者在Spring中也有自己的名称空间支持。以下示例展示了advisor：

```
<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>
```

除了前面示例中使用的pointcut-ref属性外，还可以使用pointcut属性来定义内联的pointcut表达式。

要定义advisor的优先级以便通知可以参与排序，请使用order属性定义advisor的排序值。

5.5.7 一个AOP Schema例子

本节展示了使用Schema支持重写时，来自AOP示例的并发锁定失败重试示例。

业务服务的执行有时会由于并发问题（例如，死锁）而失败。如果重试该操作，则下次尝试可能会成功。对于适合在这种情况下重试的业务服务（不需要返回用户进行冲突解决的idempotent操作），我们希望透明地重试该操作，以避免客户端看到PessimisticLockingFailureException。这是一个需求，它清楚地跨越服务层中的多个服务，因此是通过一个方面实现的理想选择。

因为我们想重试这个操作，所以我们需要使用around advice，这样我们可以多次调用proceed。下面的列表显示了基本方面实现（它是使用模式支持的常规Java类）：

```
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() {
```

更多教程请访问:www.flydean.com

```
        return this.order;
    }

    public void setOrder(int order) {
        this.order = order;
    }

    public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {
        int numAttempts = 0;
        PessimisticLockingFailureException lockFailureException;
        do {
            numAttempts++;
            try {
                return pjp.proceed();
            }
            catch(PessimisticLockingFailureException ex) {
                lockFailureException = ex;
            }
        } while(numAttempts <= this.maxRetries);
        throw lockFailureException;
    }

}
```

注意，方面实现了Ordered接口，因此我们可以将方面的优先级设置为高于事务通知（我们希望每次重试时都有一个新的事务）。maxRetries和order属性都由spring配置。主要动作发生在围绕advice方法的doConcurrentOperation中。我们试着继续。如果我们因PessimisticLockingFailureException而失败，我们会再试一次，除非我们已经用尽了所有的重试尝试。

此类与@Aspectj示例中使用的类相同，但已删除注解。

相应的Spring配置如下：

```
<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>
```

请注意，目前我们假定所有业务服务都是idempotent的。如果不是这样，我们可以通过引入一个Idempotent注解并使用它来注解服务操作的实现来改进方面，使它只重试真正的Idempotent操作，如下示例所示：

```
@Retention(RetentionPolicy.RUNTIME)
public @Interface Idempotent {
    // marker annotation
}
```

将方面更改为仅重试idempotent操作涉及到优化切入点表达式，以便只匹配@Idempotent，如下所示：

更多教程请访问:www.flydean.com

```
<aop:pointcut id="idempotentOperation"
        expression="execution(* com.xyz.myapp.service.*.*(..)) and
        @annotation(com.xyz.myapp.service.Idempotent)"/>
```

更多教程请访问:www.flydean.com

5.6选择要使用的AOP声明样式

一旦确定某个方面是实现给定需求的最佳方法，那么如何决定使用SpringAOP或AspectJ，以及Aspect语言（代码）样式、@AspectJ注解样式或SpringXML样式之间的关系呢？这些决策受许多因素的影响，包括应用程序需求、开发工具和团队对AOP的熟悉程度。

5.6.1 Spring AOP 还是全部使用 AspectJ?

用最简单的方法。SpringAOP比使用完整的AspectJ简单，因为不需要在开发和构建过程中引入AspectJ编译器/weaver。如果你只需要advise Spring Beans的执行操作，那么SpringAOP是正确的选择。如果需要advise不由Spring容器管理的对象（通常是域对象），则需要使用AspectJ。如果希望advise连接点而不是简单的方法执行（例如，field get或 set join points等），还需要使用aspectj。

使用aspectj时，可以选择aspectj语言语法（也称为“代码样式”）或@Aspectj注解样式。显然，如果你不使用Java 5 +，那么你就已经选择了：使用代码风格。如果aspects在你的设计中起着很大的作用，并且你能够为Eclipse使用AspectJ开发工具（AJDT）插件，那么AspectJ语言语法是首选选项。它更干净、更简单，因为语言是专门为编写方面而设计的。如果你不使用Eclipse或仅在你的应用程序中不起主要作用的几个方面，你可能需要考虑使用@AspectJ样式，在IDE中坚持常规Java编译，并将一个方面编织阶段添加到生成脚本中。

5.6.2 @AspectJ 或者 Spring AOP XML?

如果你选择使用SpringAOP，那么你可以选择@Aspectj或XML样式。有各种各样的权衡需要考虑。

XML样式可能对现有的Spring用户最熟悉，并且由真正的Pojos支持。当使用AOP作为配置企业服务的工具时，XML是一个不错的选择（一个好的测试是你是否将切入点表达式视为你可能希望独立更改的配置的一部分）。使用XML样式，可以从配置中更清楚地看到系统中存在哪些方面。

XML样式有两个缺点。首先，它并没有将它所处理的需求的实现完全封装在一个地方。DRY原则认为系统中任何知识都应该有一个单一的、明确的、权威的表示。当使用XML样式时，如何实现需求的知识将在支持bean类的声明和配置文件中的XML中分割开来。当你使用@Aspectj样式时，这个信息被封装在一个aspect模块中。其次，XML样式在它可以表示的内容上比@Aspectj样式稍有限制：只支持“singleton”方面实例化模型，并且不可能将XML中声明的命名切入点组合在一起。例如，在@Aspectj样式中，可以编写如下内容：

```
@Pointcut("execution(* get*())")
public void propertyAccess() {}

@Pointcut("execution(org.xyz.Account+ *(..))")
public void operationReturningAnAccount() {}

@Pointcut("propertyAccess() && operationReturningAnAccount()")
public void accountPropertyAccess() {}
```

在XML样式中，可以声明前两个切入点：

```
<aop:pointcut id="propertyAccess"
        expression="execution(* get*())"/>

<aop:pointcut id="operationReturningAnAccount"
        expression="execution(org.xyz.Account+ *(..))"/>
```

XML方法的缺点是不能通过组合这些定义来定义accountPropertyAccess切入点。

更多教程请访问:www.flydean.com

@Aspectj样式支持附加的实例化模型和更丰富的切入点组合。它具有将方面作为模块化单元保留的优点。它还具有这样的优势，@Aspectj方面可以被SpringAOP和aspectj理解（从而被消费）。因此，如果你稍后决定需要AspectJ的功能来实现额外的需求，你可以轻松地迁移到经典的AspectJ设置。总的来说，Spring团队除了简单的企业服务配置外，更喜欢@Aspectj风格的定制方面。

更多教程请访问:www.flydean.com

5.7混合Aspect类型

通过使用自动代理支持、schema-defined的<aop:aspect> aspects、<aop:advisor> declared advisors，甚至在同一配置中使用其他样式的代理和拦截器，完全可以混合@Aspectj样式的方面。所有这些都是通过使用相同的底层支持机制实现的，并且可以毫不费力地共存。

更多教程请访问:www.flydean.com

5.8代理机制

SpringAOP使用JDK动态代理或CGLIB为给定的目标对象创建代理。JDK动态代理内置于JDK中，而CGLIB是一个公共的开放源码类定义库（重新打包到Spring核心中）。

如果要代理的目标对象实现至少一个接口，则使用JDK动态代理。由目标类型实现的所有接口都是代理的。如果目标对象不实现任何接口，则创建一个CGLIB代理。

如果要强制使用CGLIB代理（例如，代理为目标对象定义的每个方法，而不仅仅是由其接口实现的方法），可以这样做。但是，你应该考虑以下问题：

- 对于cglib，不能通知final方法，因为它们不能在运行时生成的子类中被重写。
- 从Spring4.0开始，代理对象的构造函数不再被调用两次，因为cglib代理实例是通过Objenesis创建的。只有当你的JVM不允许绕过构造函数时，你可能会看到来自Spring的AOP支持的双重调用和相应的调试日志条目。

要强制使用cglib代理，请将<aop:config>元素的proxy target class属性的值设置为true，如下所示：

```
<aop:config proxy-target-class="true">
    <!-- other beans defined here... -->
</aop:config>
```

要在使用@Aspectj auto proxy支持时强制cglib代理，请将<aop:aspectj-autoproxy>元素的proxy-target-class属性设置为true，如下所示：

```
<aop:aspectj-autoproxy proxy-target-class="true"/>
```

> 多个<aop:config/>节在运行时折叠为一个统一的自动代理创建者，这将应用指定的任何<aop:config/>节（通常来自不同的XML bean定义文件）所指定的最强代理设置。这也适用于<tx:annotation-driven/>和<aop:aspectj-autoproxy/>元素。
>
> 要清楚，在<tx:annotation-driven/>、<aop:aspectj autoproxy/>或<aop:config/>元素上使用proxy target class="true"将强制对这三个元素使用cglib代理。

5.8.1 理解AOP Proxies

SpringAOP是基于代理的。在编写自己的方面或使用Spring框架提供的任何基于SpringAOP的方面之前，掌握最后一条语句的实际语义是非常重要的。

首先考虑一个场景，其中你有一个普通的、未代理的、没有什么特别之处的、直接的对象引用，如下面的代码片段所示：

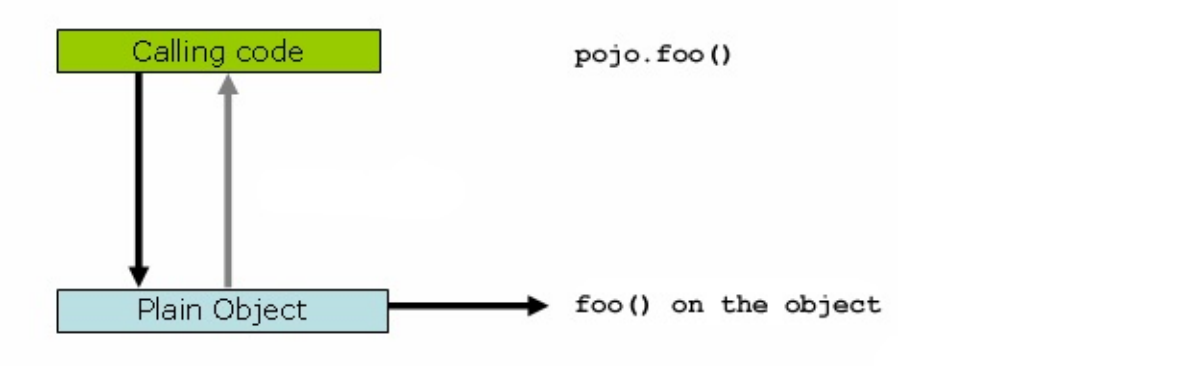
```
public class SimplePojo implements Pojo {

    public void foo() {
        // this next method invocation is a direct call on the 'this' reference
        this.bar();
    }

    public void bar() {
        // some logic...
    }
}
```

如果对对象引用调用方法，则直接对该对象引用调用该方法，如下图和列表所示：

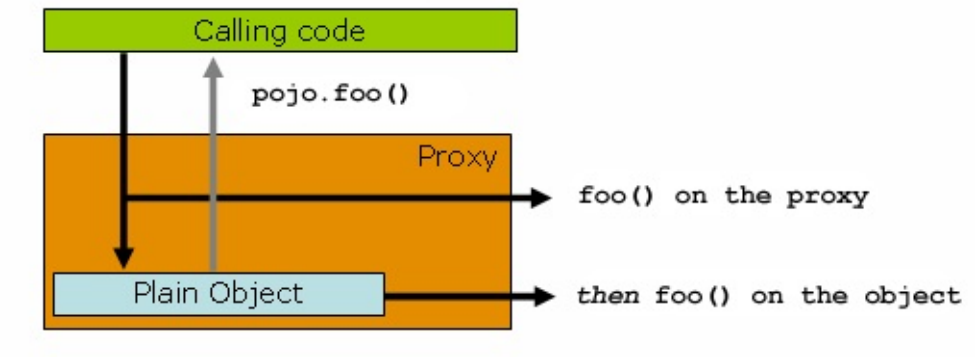
更多教程请访问:www.flydean.com




```
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();
    }
}
```

当客户代码的引用是代理时，情况略有变化。请考虑以下图表和代码段：




```
public class Main {

    public static void main(String[] args) {
        ProxyFactory factory = new ProxyFactory(new SimplePojo());
        factory.addInterface(Pojo.class);
        factory.addAdvice(new RetryAdvice());

        Pojo pojo = (Pojo) factory.getProxy();
        // this is a method call on the proxy!
        pojo.foo();
    }
}
```

这里要理解的关键是，主类的main（..）方法中的代码具有对代理的引用。这意味着对该对象引用的方法调用是对代理的调用。因此，代理可以委托给与该特定方法调用相关的所有拦截器（通知）。但是，一旦调用最终到达目标对象（在本例中为SimplePojo的引用），它可以对自身进行的任何方法调用（如this.bar（）或this.foo（））都将针对该引用而不是代理进行调用。这有重要的意义。这意味着自调用不会导致与方法调用相关联的advice获得执行的机会。

好吧，那我们该怎么办？最好的方法（这里松散地使用术语“best”）是重构代码，这样就不会发生自调用。这确实需要你做一些工作，但这是最好的，微创的方法。下一个方法绝对是可怕的，我们毫不犹豫地指出它，正是因为它是如此可怕。你可以（尽管对我们来说很痛苦）将类中的逻辑完全绑定到SpringAOP，如下示例所示：

```
public class SimplePojo implements Pojo {
```

更多教程请访问:www.flydean.com

```
    public void foo() {
        // this works, but... gah!
        ((Pojo) AopContext.currentProxy()).bar();
    }

    public void bar() {
        // some logic...
    }
}
```

这完全将你的代码与SpringAOP结合在一起，并且它使类本身意识到它正在一个AOP上下文中使用，而这个AOP上下文在AOP的面前飞驰。在创建代理时，它还需要一些额外的配置，如下示例所示：

```
public class Main {

    public static void main(String[] args) {
        ProxyFactory factory = new ProxyFactory(new SimplePojo());
        factory.adddInterface(Pojo.class);
        factory.addAdvice(new RetryAdvice());
        factory.setExposeProxy(true);

        Pojo pojo = (Pojo) factory.getProxy();
        // this is a method call on the proxy!
        pojo.foo();
    }
}
```

最后，必须注意的是，AspectJ没有这种自调用问题，因为它不是基于代理的AOP框架。

更多教程请访问:www.flydean.com

5.9程序创建@AspectJ代理

除了使用<aop:config>或<aop:aspectj-autoproxy>在配置中声明方面之外，还可以通过编程方式创建通知目标对象的代理。有关Spring的AOP API的完整详细信息，请参阅下一章。在这里，我们将重点介绍使用@Aspectj特性自动创建代理的能力。

可以使用org.springframework.aop.aspectj.annotation.AspectJProxyFactory类为一个或多个@Aspectj方面建议的目标对象创建代理。这个类的基本用法非常简单，如下示例所示：

```
// 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();
```

更多教程请访问:www.flydean.com

5.10在Spring应用程序中使用AspectJ

到目前为止，我们在这一章中介绍的所有内容都是纯Spring AOP。在本节中，如果你要使用Spring AOP之外的功能，我们将介绍如何在Spring中使用AspectJ编译器或weaver。

Spring附带了一个小的AspectJ方面库，它在你的发行版中作为spring-aspects.jar单独提供。为了使用类路径中的方面，需要将其添加到类路径中。“使用aspectj来依赖注入带有Spring的域对象”，以及“aspectj的其他Spring方面”，讨论了这个库的内容以及如何使用它。“通过使用SpringIOC来配置AspectJ方面”讨论了如何使用AspectJ编译器对AspectJ方面进行依赖注入。最后，“在Spring框架中使用AspectJ进行加载时编织”，为使用AspectJ的Spring应用程序提供了加载时编织的介绍。

5.10.1 使用AspectJ来依赖注入Spring的域对象

Spring容器实例化并配置在应用程序上下文中定义的bean。如果给定包含要应用的配置的bean定义的名称，也可以要求bean工厂配置预先存在的对象。spring-aspects.jar包含一个注解驱动的方面，它利用这个能力允许任何对象的依赖注入。该支持用于在任何容器的控制范围之外创建的对象。域对象通常属于此类，因为它们通常是通过new运算符或ORM工具作为数据库查询的结果以编程方式创建的。

@Configurable注解将类标记为符合Spring驱动配置。在最简单的情况下，可以将它纯粹用作标记注解，如下示例所示：

```
package com.xyz.myapp.domain;

import org.springframework.beans.factory.annotation.Configurable;

@Configurable
public class Account {
    // ...
}
```

以这种方式用作标记接口时，Spring通过使用与完全限定类型名（com.xyz.myapp.domain.account）同名的bean定义（通常是原型范围）来配置注解类型（本例中为account）的新实例。由于bean的默认名称是其类型的完全限定名，声明原型定义的一个方便方法是省略id属性，如下示例所示：

```
<bean class="com.xyz.myapp.domain.Account" scope="prototype">
    <property name="fundsTransferService" ref="fundsTransferService"/>
</bean>
```

如果要显式指定要使用的原型bean定义的名称，可以直接在注解中这样做，如下例所示：

```
package com.xyz.myapp.domain;

import org.springframework.beans.factory.annotation.Configurable;

@Configurable("account")
public class Account {
    // ...
}
```

Spring现在查找一个名为account的bean定义，并将其用作定义来配置新的Account实例。

更多教程请访问:www.flydean.com

你还可以使用自动装载来避免必须指定一个专用的bean定义。要让Spring应用自动装载，请使用@Configurable注解的autowire属性。你可以分别指定@Configurable(autowire=Autowire.BY_TYPE)或@Configurable(autowire=Autowire.BY_NAME)，用于按类型或按名称自动注解。作为替代方案，最好在字段或方法级别通过@Autowired或@Inject为@Configurable bean指定显式的、由注解驱动的依赖项注入（有关详细信息，请参阅基于注解的容器配置）。

最后，可以使用dependencyCheck属性（例如，@Configurable（autowire=Autowire.BY_NAME，dependencyCheck=true））对新创建和配置的对象中的对象引用启用Spring依赖性检查。如果将此属性设置为true，那么配置之后，Spring将验证是否已设置了所有属性（不是基元或集合）。

请注意，单独使用注解没有任何作用。Spring-Aspects.jar中的AnnotationBeanConfigurerAspect，才是让注解起作用的根本。本质上，aspect是，“从用@Configurable注解的类型的新对象的初始化返回后，根据注解的属性使用spring配置新创建的对象”。在此上下文中，“初始化”是指新实例化的对象（例如，用new运算符实例化的对象）以及正在进行反序列化的可序列化对象（例如，通过 readResolve()）。

> 以上段落中的一个关键短语是“本质上”。在大多数情况下，“从新对象的初始化返回后”的确切语义是正确的。在此上下文中，“初始化后”意味着在构造对象之后注入依赖项。这意味着依赖项在类的构造函数体中不可用。如果希望在构造函数主体执行之前注入依赖项，从而在构造函数主体中可用，则需要在@Configurable声明中定义此依赖项，如下所示：

```
@Configurable(preConstruction = true)
```

> 你可以在《AspectJ编程指南》的本附录中找到关于AspectJ中各种切点类型的语言语义的更多信息。

要使这项工作，注解类型必须与AspectJ Weaver一起编织。你可以使用构建时Ant或Maven任务来执行此操作（例如，请参阅《AspectJ开发环境指南》）或加载时编织（请参阅Spring框架中的加载时编织与AspectJ）。AnnotationBeanConfigureAspect本身需要由Spring进行配置（以便获得用于配置新对象的bean工厂的引用）。如果使用基于Java的配置，可以将@EnableSpringConfigureds\添加到任何“@Configuration”类，如下：

```
@Configuration
@EnableSpringConfigured
public class AppConfig {
}
```

如果你更喜欢基于XML的配置，那么Spring上下文名称空间定义了一个方便的context:spring-configured的元素，你可以使用它，如下所示：

```
<context:spring-configured/>
```

配置方面之前创建的@Configurable objects实例会导致向调试日志发出消息，并且不会对对象进行任何配置。一个例子可能是Spring配置中的bean，它在由Spring初始化时创建域对象。在这种情况下，可以使用depends-on bean属性手动指定bean依赖于配置方面。以下示例显示如何使用“depends-on”属性：

```
<bean id="myService"
        class="com.xzy.myapp.service.MyService"
        depends-on="org.springframework.beans.factory.aspectj.AnnotationBeanConfigurerAspect">

    <!-- ... -->

</bean>
```

> 不要通过bean配置器方面激活@Configurable，除非你真正打算在运行时依赖它的语义。特别是，确保在注册为容器中常规SpringBean的bean类上不使用@Configurable。这样做会导致双重初始化，一次通过容器，一次通过方面。

更多教程请访问:www.flydean.com

单元测试@Configurable对象

@Configurable支持的目标之一是能够对域对象进行独立的单元测试，而无需硬编码查找。如果@Configurable types不是由aspectj编织的，那么在单元测试期间注解没有影响。你可以在被测对象中设置mock或stub属性引用，并正常进行。如果@Configurable types已经被aspectj编织，你仍然可以像正常一样在容器外部进行单元测试，但是每次构造@Configurable对象时都会看到一条警告消息，表明它没有被spring配置。

和多个Application Contexts一起工作

用于实现@Configurable Support的AnnotationBeanConfigurerAspect是一个AspectJ单例方面。单个方面的范围与静态成员的范围相同：每个类加载器都有一个方面实例来定义类型。这意味着，如果在同一个类加载器层次结构中定义多个应用程序上下文，则需要考虑在何处定义@EnableSpringConfigured bean以及在何处将spring-aspects.jar放在类路径上。

考虑一个典型的SpringWeb应用程序配置，它具有一个定义公共业务服务的共享父应用程序上下文、支持这些服务所需的所有内容，以及每个servlet的一个子应用程序上下文（其中包含特定于该servlet的定义）。所有这些上下文都存在于同一个类加载器层次结构中，因此AnnotationBeanConfigurerAspect只能包含对其中一个上下文的引用。在这种情况下，我们建议在共享（父）应用程序上下文中定义@EnableSpringConfigured bean。这定义了你可能希望注入到域对象中的服务。结果是，你无法使用@Configurable机制（这可能不是你想要做的事情）配置域对象，因为它引用了在子（servlet特定）上下文中定义的bean。

在同一容器中部署多个Web应用程序时，请确保每个Web应用程序都使用自己的类加载器（例如，通过将spring-aspects.jar放在“WEB-INF/lib”中）加载spring-aspects.jar中的类型。如果Spring-Aspects.jar只添加到容器范围的类路径（因此由共享父类加载器加载），那么所有Web应用程序都共享相同的Aspect实例（这可能不是你想要的）。

5.10.2 AspectJ的其他Spring aspects

除了@Configurable方面，spring-aspects.jar还包含一个aspectj方面，你可以使用它来驱动spring对用@transactional annotation注解的类型和方法的事务管理。这主要面向希望在Spring容器之外使用Spring框架的事务支持的用户。

解释@transactional注解的方面是AnnotationTransactionAspect。使用此方面时，必须注解实现类（或该类中的方法或两者都使用），而不是类实现的接口（如果有的话）。AspectJ遵循Java的规则，接口上的注解不是继承的。

类上的@transactional注解指定执行类中任何公共操作的默认事务语义。

类内方法的@transactional注解将覆盖类注解（如果存在）给出的默认事务语义。任何可见性的方法都可以进行注解，包括私有方法。直接注解非公共方法是实现此类方法的事务划分的唯一方法。

> 自SpringFramework4.2以来，spring-aspects提供了类似的方面，为标准javax.transaction.transactional注解提供了完全相同的特性。有关详细信息，请查看JtaAnnotationTransactionAspect。

对于那些希望使用Spring配置和事务管理支持但不希望（或不能）使用注解的AspectJ程序员，spring-aspects.jar还包含可以扩展以提供自己的切入点定义的抽象方面。有关更多信息，请参阅AbstractBeanConfigureAspect和AbstractTransactionspect方面的源代码。作为一个例子，下面的摘录展示了如何通过使用与完全限定类名匹配的原型bean定义来编写一个方面来配置域模型中定义的对象的所有实例：

```
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);
}
```

更多教程请访问:www.flydean.com

5.10.3使用SpringIoC来配置AspectJ Aspects

当你将AspectJ方面与Spring应用程序一起使用时，很自然地希望并期望能够使用Spring配置这些方面。AspectJ运行时本身负责方面的创建，通过Spring配置AspectJ创建的方面的方法取决于方面使用的AspectJ实例化模型（per-xxx子句）。

AspectJ方面的大多数是单体方面。这些方面的配置很容易。你可以创建一个bean定义，该定义将方面类型作为普通类型引用，并包含factory-method="aspectOf" bean属性。这样可以确保Spring通过向AspectJ请求来获得AspectJ实例，而不是试图创建实例本身。以下示例显示如何使用factory-method="aspectOf"属性：

```
<bean id="profiler" class="com.xyz.profiler.Profiler"
        factory-method="aspectOf">

    <property name="profilingStrategy" ref="jamonProfilingStrategy"/>
</bean>
```

非单例方面更难配置。但是，可以通过创建原型bean定义和使用spring-aspects.jar中的@Configurable支持来配置Aspectj运行时创建的bean之后的方面实例来实现这一点。

如果你有一些@Aspectj方面要用aspectj编织（例如，对域模型类型使用加载时编织）和其他@Aspectj方面要用spring aop，并且这些方面都在spring中配置，那么你需要告诉spring aop @Aspectj auto-proxing支持在配置中定义的@Aspectj方面的哪个确切子集应用于自动代理。可以通过在<aop:aspectj-autoproxy/>声明中使用一个或多个<include/>元素来完成此操作。每个<include/>元素都指定一个名称模式，只有名称与至少一个模式匹配的bean才用于SpringAOP自动代理配置。以下示例显示如何使用<include/>元素：

```
<aop:aspectj-autoproxy>
    <aop:include name="thisBean"/>
    <aop:include name="thatBean"/>
</aop:aspectj-autoproxy>
```

> 不要被<aop:aspectj-autoproxy/>元素的名称误导。使用它可以创建SpringAOP代理。这里使用的是Aspect声明的@AspectJ样式，但不涉及AspectJ运行时。

5.10.4 Spring框架中AspectJ的Load-time Weaving

加载时间编织（LTW）指的是在将AspectJ方面加载到Java虚拟机（JVM）中的过程中，将AspectJ方面编织成应用程序的类文件。本节的重点是在Spring框架的特定上下文中配置和使用LTW。本节不是LTW的一般介绍。有关LTW的详细信息和仅使用AspectJ配置LTW（根本不涉及Spring），请参阅AspectJ开发环境指南的LTW部分。

Spring框架为AspectJ LTW带来的价值在于能够对编织过程进行更细粒度的控制。“Vanilla 的AspectJ LTW是通过使用Java（5 +）代理来实现的，它在启动JVM时通过指定VM参数来打开。因此，它是一个JVM范围的设置，在某些情况下可能很好，但通常有点过于粗糙。启用了Spring的LTW允许你在每个类加载器的基础上打开LTW，它更细粒度，并且在“单个JVM多应用程序”环境中（例如在典型的应用程序服务器环境中）更有意义。

此外，在某些环境中，加载时编织，不需要对应用程序服务器的启动脚本进行任何修改，这些脚本需要添加-javaagent:path/to/aspectjweaver.jar或（如本节后面所述）javaagent:path/to/spring-instrument.jar。开发人员配置应用程序上下文以启用加载时编织，而不是依赖通常负责部署配置（如启动脚本）的管理员。

既然销售宣传已经结束，那么让我们先浏览一下使用Spring的AspectJ LTW的一个快速示例，然后详细介绍示例中介绍的元素。有关完整的示例，请参阅PetClinic示例应用程序。

一个例子

更多教程请访问:www.flydean.com

假设你是一个应用程序开发人员，负责诊断系统中某些性能问题的原因。我们将打开一个简单的分析方面，让我们快速获得一些性能指标，而不是开发一个分析工具。然后我们可以在之后立即对特定区域应用更细粒度的分析工具。

> 这里的示例使用XML配置。你还可以配置和使用Java配置的AspectJ。具体来说，你可以使用@EnableLoadTimeWeaving注解作为<context:load-time-weaver/>的替代方法（有关详细信息，请参阅下面）。

下面的示例显示了分析方面，这是不理想的。它是一个基于时间的探查器，使用@Aspectj风格的方面声明：

```
package foo;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.util.StopWatch;
import org.springframework.core.annotation.Order;

@Aspect
public class ProfilingAspect {

    @Around("methodsToBeProfiled()")
    public Object profile(ProceedingJoinPoint pjp) throws Throwable {
        StopWatch sw = new StopWatch(getClass().getSimpleName());
        try {
            sw.start(pjp.getSignature().getName());
            return pjp.proceed();
        } finally {
            sw.stop();
            System.out.println(sw.prettyPrint());
        }
    }

    @Pointcut("execution(public * foo..*.*(..))")
    public void methodsToBeProfiled(){}
}
```

我们还需要创建一个META-INF/aop.xml文件，通知AspectJ weaver我们想要将ProfilingAspect编织到我们的类中。这个文件约定，即在Java类路径上的文件（或文件）的存在，称为META-IMF/aop.xml是标准AspectJ。下面的示例显示aop.xml文件：

```
<!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN" "https://www.eclipse.org/aspectj/dtd/aspectj.dtd">
<aspectj>

    <weaver>
        <!-- only weave classes in our application-specific packages -->
        <include within="foo.*"/>
    </weaver>

    <aspects>
        <!-- weave in just this aspect -->
        <aspect name="foo.ProfilingAspect"/>
    </aspects>

</aspectj>
```

现在我们可以继续讨论配置中特定于Spring的部分。我们需要配置一个LoadTimeWeaver（稍后解释）。这个加载时weaver是负责将一个或多个META-INF/aop.xml文件中的方面配置编织到应用程序中的类中的基本组件。好的是，它不需要很多配置（你可以指定更多选项，但稍后将详细介绍这些选项），如下面的示例所示：

```
<?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"
```

更多教程请访问:www.flydean.com

```
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://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>
```

既然所有必需的工件（方面、META-INF/aop.xml文件和Spring配置）都已就位，那么我们可以用一个main（..）方法创建以下驱动程序类来演示LTW的实际操作：

```
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();
    }
}
```

我们还有最后一件事要做。本节的介绍确实指出，可以使用Spring在每个类加载器的基础上选择性地打开LTW，这是正确的。但是，对于这个例子，我们使用Java代理（用Spring提供）来切换LTW。我们使用以下命令来运行前面显示的主类：

```
java -javaagent:C:/projects/foo/lib/global/spring-instrument.jar foo.Main
```

-javaagent是一个标志，用于指定和使代理能够检测在JVM上运行的程序。Spring框架附带这样一个代理，即InstrumentationSavingAgent，它打包在Spring-instrument.jar中，在前面的示例中作为-javaagent参数的值提供。

主程序执行时的输出类似于下一个示例。（我在calculateEntitlement（）实现中引入了一个Thread.sleep（..）语句，以便探查器实际捕获0毫秒以外的内容（01234毫秒不是AOP引入的开销）。下面的列表显示了运行分析器时得到的输出：

```
Calculating entitlement

StopWatch 'ProfilingAspect': running time (millis) = 1234
------ ----- ----------------------------
ms     %     Task name
------ ----- ----------------------------
01234  100%  calculateEntitlement
```

由于此LTW是通过使用全AspectJ来实现的，因此我们不仅限于advising SpringBeans。Main程序的以下细微变化产生相同的结果：

```
package foo;
```

更多教程请访问:www.flydean.com

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

public final class Main {

    public static void main(String[] args) {
        new ClassPathXmlApplicationContext("beans.xml", Main.class);

        EntitlementCalculationService entitlementCalculationService =
                new StubEntitlementCalculationService();

        // the profiling aspect will be 'woven' around this method execution
        entitlementCalculationService.calculateEntitlement();
    }
}
```

注意，在前面的程序中，我们如何引导Spring容器，然后完全在Spring上下文之外创建StubEntitlementCalculationService的新实例。分析advice仍然被编织在一起。

诚然，这个例子是简单的。但是，在前面的示例中已经介绍了在Spring中LTW支持的基础知识，本节的其余部分详细解释了每一位配置和使用背后的“为什么”。

> 本例中使用的ProfilingAspect可能是基本的，但非常有用。这是一个很好的开发时间方面的例子，开发人员可以在开发期间使用它，然后很容易地将其从部署到UAT或生产中的应用程序的构建中排除。

Aspects

你在LTW中使用的方面必须是AspectJ方面。你可以用aspectj语言本身编写它们，也可以用@Aspectj样式编写方面。你的方面是有效的AspectJ和SpringAOP方面。此外，编译的方面类需要在类路径上可用。

'META-INF/aop.xml'

AspectJ LTW基础结构通过使用Java类路径上的一个或多个META-INF/aop.xml 文件来配置（直接或更典型地，在JAR文件中）。

此文件的结构和内容在AspectJ参考文档的LTW部分中有详细说明。因为aop.xml文件是100%aspectj，所以我们在这里不再详细描述它。

Required libraries (JARS)

至少，你需要以下库来使用Spring框架对AspectJ LTW的支持：

- spring-aop.jar
- aspectjweaver.jar

如果使用Spring提供的代理来启用检测，还需要：

- spring-instrument.jar

Spring配置

Spring LTW支持中的关键组件是loadTimeweaver接口（在org.springframework.instrument.classloading包中），以及与Spring发行版一起提供的许多实现。loadtimeweaver负责在运行时向类加载器添加一个或多个java.lang.instrument.ClassFileTransformers，这为各种有趣的应用程序打开了大门，其中一个应用程序恰好是方面的LTW。

如果你不熟悉运行时类文件转换的概念，请参阅java.lang.instrument包的javadoc API文档，然后继续。虽然该文档并不全面，但至少你可以看到关键的接口和类（供你阅读本节时参考）。

为特定的应用程序上下文配置LoadTimeWeaver与添加一行一样简单。（请注意，你几乎肯定需要使用ApplicationContext作为Spring容器-通常，BeanFactory是不够的，因为LTW支持使用BeanFactoryPostProcessors。）

更多教程请访问:www.flydean.com

要启用Spring框架的LTW支持，你需要配置一个LoadTimeWeaver，通常通过使用@EnableLoadTimeWeaving注解完成，如下所示：

```
@Configuration
@EnableLoadTimeWeaving
public class AppConfig {
}
```

或者，如果你更喜欢基于XML的配置，请使用<context:load-time-weaver/>元素。请注意，元素是在上下文命名空间中定义的。下面的示例演示如何使用<context:load-time-weaver/>：

```
<?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
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <context:load-time-weaver/>

</beans>
```

前面的配置自动为你定义和注册许多LTW特定的基础结构bean，例如LoadTimeWeaver和AspectJWeavingEnabler。默认的LoadTimeWeaver是DefaultContextLoadTimeWeaver类，它尝试修饰自动检测到的LoadTimeWeaver。“自动检测”的LoadTimeWeaver的确切类型取决于运行时环境。下表总结了各种LoadTimeWeaver实现：

运行时环境	LoadTimeWeaver实现
Running in Apache Tomcat	TomcatLoadTimeWeaver
Running in GlassFish (limited to EAR deployments)	GlassFishLoadTimeWeaver
Running in Red Hat’s JBoss AS or WildFly	JBossLoadTimeWeaver
Running in IBM’s WebSphere	WebSphereLoadTimeWeaver
Running in Oracle’s WebLogic	WebLogicLoadTimeWeaver
JVM started with Spring InstrumentationSavingAgent (java -javaagent:path/to/spring-instrument.jar)	InstrumentationLoadTimeWeaver
回退，期望底层类加载器遵循公共约定（即addTransformer和可选的getThrowawayClassLoader方法）	ReflectiveLoadTimeWeaver

请注意，该表仅列出使用DefaultContextLoadTimeWeaver时自动检测的LoadTimeWeaver。你可以准确地指定要使用的LoadTimeWeaver实现。

若要指定具有Java配置的特定LoadTimeWeaver，请执行LoadTimeWeavingConfigurer接口并重写getLoadTimeWeaver（）方法。以下示例指定了ReflectiveLoadTimeWeaver：

```
@Configuration
@EnableLoadTimeWeaving
public class AppConfig implements LoadTimeWeavingConfigurer {

    @Override
    public LoadTimeWeaver getLoadTimeWeaver() {
        return new ReflectiveLoadTimeWeaver();
    }
}
```

更多教程请访问:www.flydean.com

如果使用基于XML的配置，则可以将完全限定的类名指定为<context:load-time-weaver/>元素上weaver-class属性的值。同样，下面的示例指定了ReflectiveLoadTimeWeaver：

```
<?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
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <context:load-time-weaver
            weaver-class="org.springframework.instrument.classloading.ReflectiveLoadTimeWeaver"/>

</beans>
```

配置定义和注册的LoadTimeWeaver稍后可以使用众所周知的名称LoadTimeWeaver从Spring容器中检索。记住，LoadTimeWeaver仅作为Spring的LTW基础结构添加一个或多个ClassFileTransformer的机制存在。执行LTW的实际ClassFileTransformer是ClassPreprocessorAgentAdapter（来自org.aspectj.weaver.loadtime包）类。有关进一步的详细信息，请参见ClassPreprocessorAgentAdapter类的类级JavaDoc，因为实际影响编织的细节超出了本文档的范围。

还有一个最后要讨论的配置属性：aspectjWeaving属性（如果使用XML，则为 aspectj-weaving属性）。此属性控制是否启用LTW。它接受三个可能值中的一个，如果属性不存在，则默认值为autodetect。下表总结了三个可能的值：

Annotation Value	XML Value	Explanation
ENABLED	on	Aspectj编织已启用，并且各个方面在加载时视情况进行编织。
DISABLED	off	LTW关闭。加载时不编织任何方面。
AUTODETECT	autodetect	如果SpringLTW基础结构可以找到至少一个META-INF/aop.xml文件，那么aspectj weaving就打开了。否则，它将关闭。这是默认值。

环境特定配置

最后一部分包含在应用程序服务器和Web容器等环境中使用Spring的LTW支持时所需的任何其他设置和配置。

Tomcat, JBoss, WebSphere, WebLogic

Tomcat、JBoss/WildFly、IBM WebSphere Application Server和Oracle WebLogic Server都提供了一个通用的应用程序类加载器，能够进行本地检测。Spring的本地LTW可以利用这些类加载器实现来提供AspectJ编织。如前所述，你可以简单地启用加载时编织。具体来说，你不需要修改jvm启动脚本来添加-javaagent:path/to/spring-instrument.jar。

注意，在jboss上，你可能需要禁用应用服务器扫描，以防止它在应用程序实际启动之前加载类。一个快速的解决方法是在工件中添加一个名为WEB-INF/jboss-scanning.xml的文件，其中包含以下内容：

```
<scanning xmlns="urn:jboss:scanning:1.0"/>
```

通用Java应用程序

当特定的loadtimeweaver实现不支持的环境中需要类检测时，JVM代理是通用的解决方案。对于这种情况，Spring提供了InstrumentationLoadTimeweaver，它需要特定于Spring（但非常通用）的JVM代理spring-instrument.jar，由common @EnableLoadTimeWeaving和<context:load-time-weaver/>设置自动检测。

要使用它，必须通过提供以下JVM选项来使用Spring代理启动虚拟机：

```
-javaagent:/path/to/spring-instrument.jar
```

更多教程请访问:www.flydean.com

请注意，这需要修改JVM启动脚本，这可能会阻止你在应用程序服务器环境中使用该脚本（取决于你的服务器和操作策略）。也就是说，对于每个JVM部署一个应用程序（如独立的Spring引导应用程序），你通常在任何情况下都控制整个JVM设置。

更多教程请访问:www.flydean.com

5.11更多资源

有关AspectJ的更多信息，请访问AspectJ网站。

Adrian Colyer等人的Eclipse Aspectj。（Addison-Wesley，2005）为AspectJ语言提供了全面的介绍和参考。

Aspectj in Action，第二版由Ramnivas Laddad（Manning，2009）强烈推荐。这本书的重点是Aspectj，但许多一般的AOP主题被探索（在一定深度）。

更多教程请访问:www.flydean.com

6.Spring AOP APIs

上一章描述了Spring对带有@Aspectj和schema-based 的方面定义的AOP的支持。在本章中，我们将讨论较低级别的SpringAOP API。对于常见的应用程序，我们建议使用带有AspectJ切点的SpringAOP，如前一章所述。

更多教程请访问:www.flydean.com

6.1Pointcut API

本节介绍Spring如何处理关键的切入点概念。

6.1.1 概念

Spring的切入点模型支持独立于通知类型的切入点重用。你可以使用相同的切入点来通知不同的建议目标。

org.springframework.aop.Pointcut接口是一个中心接口，用于将通知定向到特定的类和方法。完整interface如下：

```
public interface Pointcut {

    ClassFilter getClassFilter();

    MethodMatcher getMethodMatcher();

}
```

将Pointcut接口拆分为两部分允许重用类和方法匹配部分以及细粒度组合操作（例如，使用另一个方法匹配器执行“联合”）。

ClassFilter接口用于将切入点限制为一组给定的目标类。如果matches（）方法始终返回true，则所有目标类都匹配。下面的列表显示ClassFilter接口定义：

```
public interface ClassFilter {

    boolean matches(Class clazz);
}
```

MethodMatcher接口通常来说更加重要。完整的接口如下：

```
public interface MethodMatcher {

    boolean matches(Method m, Class targetClass);

    boolean isRuntime();

    boolean matches(Method m, Class targetClass, Object[] args);
}
```

matches（method，class）方法用于测试此切入点是否与目标类上的给定方法匹配。当创建AOP代理时可以执行此评估，以避免对每个方法调用进行测试。如果两个参数匹配的方法对于给定的方法返回true，而methodMatcher的isRuntime（）方法返回true，则在每次方法调用时都会调用三个参数匹配的方法。这允许切入点查看在执行目标通知之前立即传递给方法调用的参数。

大多数MethodMatcher实现是静态的，这意味着它们的isRuntime（）方法返回false。在本例中，从不调用三个参数matches方法。

> 如果可能，尝试使切入点静态化，允许AOP框架在创建AOP代理时缓存切入点评估的结果。

6.1.2 Pointcuts的操作

Spring支持对切入点的操作（尤其是Union和intersection）。

更多教程请访问:www.flydean.com

Union是指切入点匹配任意一个方法。intersection表示切入点同时匹配的方法。Union通常更有用。你可以使用org.springframework.aop.support.Pointcuts类中的静态方法或在同一个包中使用ComposablePointcut类来撰写切入点。然而，使用AspectJ切入点表达式通常是一种更简单的方法。

6.1.3 AspectJ表达式Pointcuts

从2.0开始，Spring使用的最重要的切入点类型是org.springframework.aop.aspectj.AspectJExpressionPointcut。这是一个切入点，它使用AspectJ提供的库来分析AspectJ切入点表达式字符串。

有关支持的aspectj切入点原语的讨论，请参阅上一章。

6.1.4 方便的Pointcut实现

Spring提供了几个方便的切入点实现。你可以直接使用其中的一些。其他的则被用在特定于应用程序的切入点中进行子类化。

静态切入点

静态切入点基于方法和目标类，不能考虑方法的参数。静态切入点对于大多数使用来说是最好的。当第一次调用方法时，Spring只能计算一次静态切入点。之后，不需要对每个方法调用再次计算切入点。 本节的其余部分描述了Spring中包含的一些静态切入点实现。

正则表达式切入点

指定静态切入点的一个明显方法是正则表达式。除了Spring，还有几个AOP框架使这成为可能。org.springframework.aop.support.JdkRegexpMethodPointcut是一个通用的正则表达式切入点，它使用JDK中的正则表达式支持。

通过JdkRegexpMethodPointcut类，可以提供模式字符串的列表。如果其中任何一个匹配，则切入点的计算结果为true。（因此，结果实际上是这些切点的并集。）

下面的示例演示如何使用JdkRegexpMethodPointcut：

```
<bean id="settersAndAbsquatulatePointcut"
        class="org.springframework.aop.support.JdkRegexpMethodPointcut">
    <property name="patterns">
        <list>
            <value>.*set.*</value>
            <value>.*absquatulate</value>
        </list>
    </property>
</bean>
```

Spring提供了一个名为RegexpMethodPointcutAdvisor的方便类，它还允许我们引用一个Advice（请记住，Advice可以是拦截器，在建议、抛出建议和其他建议之前）。在这背后，Spring使用了JdkRegexpMethodPointcut。使用RegexpMethodPointcutAdvisor简化了连接，因为一个bean同时封装了pointcut和advice，如下示例所示：

```
<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>
```

更多教程请访问:www.flydean.com

```
</bean>
```

你可以在任何Advice类型中使用RegexpMethodPointcutAdvisor。

属性驱动*Pointcuts*

一种重要的静态切入点类型是元数据驱动的切入点。这将使用元数据属性的值（通常是源级元数据）。

动态pointcuts

动态切入点的计算成本比静态切入点高。它们考虑了方法参数和静态信息。这意味着必须用每个方法调用对它们进行评估，并且不能缓存结果，因为参数会有所不同。

主要的例子是控制流切入点。

控制流切入点

Spring控制流切入点在概念上与AspectJ CFlow切入点相似，但功能较弱。（目前无法指定在与另一个切入点匹配的连接点下方执行切入点。）控制流切入点与当前调用堆栈匹配。例如，如果连接点是由com.mycompany.web包中的方法或SomeCaller类调用的，则它可能会激发。通常使用org.springframework.aop.support.ControlFlowPointcut类指定控制流切入点。

> 控制流切入点在运行时的计算成本比其他动态切入点都要高。在Java 1.4中，成本是其他动态切入点的五倍。

6.1.5 Pointcut 父类

Spring提供了有用的切入点超类来帮助你实现自己的切入点。

因为静态切入点最有用，所以你可能应该将StaticMethodMatcherPointCut子类化。这只需要实现一个抽象方法（尽管你可以重写其他方法来定制行为）。下面的示例显示如何将StaticMethodMatcherPointCut子类化：

```
class TestStaticPointcut extends StaticMethodMatcherPointcut {

    public boolean matches(Method m, Class targetClass) {
        // return true if custom criteria match
    }
}
```

还有动态切入点的超类。你可以对任何通知类型使用自定义切入点。

6.1.6 自定义Pointcuts

因为Spring AOP中的切入点是Java类，而不是语言特征（如AspectJ中），所以可以声明自定义切入点，无论是静态的还是动态的。Spring中的自定义切入点可以任意复杂。但是，如果可以的话，我们建议使用AspectJ切入点表达式语言。

> Spring的较新版本可能支持jac-提供的“语义切入点”，例如，“所有更改目标对象中实例变量的方法”。

更多教程请访问:www.flydean.com

6.2Advice API

下面我们看一下Spring AOP怎么处理advice。

6.2.1 Advice生命周期

每个Advice都是一个Spring bean。Advice实例可以在所有Advice对象之间共享，也可以对每个Advice对象都是唯一的。这对应于每个类或每个实例的通知。

Per-class advice最常用。它适用于一般的advice，如事务advisors。这些不依赖于代理对象的状态或添加新状态。他们只根据方法和参数行事。

Per-instance的advice都适用于introductions，以支持混合。在这种情况下，建议将状态添加到代理对象。

你可以在同一个AOP代理中混合使用共享和基于实例的建议。

6.2.2 Advice类型

Spring提供了几种Advice类型，并且可以扩展以支持任意Advice类型。本节介绍基本概念和标准Advice类型。

环绕拦截Advice

Spring最基本的建议类型是围绕建议进行拦截。

Spring与AOP联盟接口兼容，以获取使用方法拦截的 around advice。实现MethodInterceptor和实现around advice的类还应实现以下接口：

```
public interface MethodInterceptor extends Interceptor {

    Object invoke(MethodInvocation invocation) throws Throwable;
}
```

invoke（）方法的methodInvocation参数公开了被调用的方法、目标连接点、AOP代理以及该方法的参数。invoke（）方法应返回调用的结果：连接点的返回值。

下面的示例显示了一个简单的SystemInterceptor实现：

```
public class DebugInterceptor implements MethodInterceptor {

    public Object invoke(MethodInvocation invocation) throws Throwable {
        System.out.println("Before: invocation=[" + invocation + "]");
        Object rval = invocation.proceed();
        System.out.println("Invocation returned");
        return rval;
    }
}
```

注意对MethodInvocation的proceed（）方法的调用。这将沿着拦截器链向连接点前进。大多数拦截器调用此方法并返回其返回值。但是，与任何around通知一样，MethodIntercetor可以返回不同的值或引发异常，而不是调用继续方法。但是，如果没有充分的理由，你不想这样做。

> MethodIntercetor实现提供与其他AOP联盟兼容的AOP实现的互操作性。本节其余部分讨论的其他建议类型以特定于Spring的方式实现了常见的AOP概念。虽然在使用最具体的通知类型方面有一个优势，但是如果你可能希望在另一个AOP框架中运行该方面，那么请坚持使用围绕通知的MethodInterceptor。注意，切入点目前在框架之间

更多教程请访问:www.flydean.com

> 不可互操作，AOP联盟目前没有定义切入点接口。

Before Advice

更简单的建议类型是before建议。这不需要MethodInvocation对象，因为它只在进入方法之前被调用。

before通知的主要优点是不需要调用proceed（）方法，因此，不可能无意中中断拦截器链。

下面的列表显示了MethodBeforeAdvice接口：

```
public interface MethodBeforeAdvice extends BeforeAdvice {

    void before(Method m, Object[] args, Object target) throws Throwable;
}
```

（Spring的API设计将允许在建议之前提供字段，尽管通常的对象适用于字段拦截，但是Spring不太可能实现它。）

请注意，返回类型为void。before advice可以在连接点执行之前插入自定义行为，但不能更改返回值。如果before通知抛出异常，它将中止拦截器链的进一步执行。异常向上传播到拦截器链。如果它未选中或在被调用方法的签名上，则直接传递给客户机。否则，AOP代理会将其包装在unchecked exception中。

下面的示例显示了Spring中的一个before通知，它对所有方法调用进行计数：

```
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可以用在任意pointcut中。

Throws Advice

如果连接点引发异常，则在返回连接点后调用throws通知。Spring提供 typed throws advice。注意，这意味着org.springframework.aop.ThrowsAdvice接口不包含任何方法。它是一个标记接口，标识给定对象实现一个或多个类型化的throw advice方法。应采用以下形式：

```
afterThrowing([Method, args, target], subclassOfThrowable)
```

只需要最后一个参数。方法签名可以有一个或四个参数，这取决于advice方法对方法和参数是否感兴趣。下面的两个列表显示了属于throw建议示例的类。

如果抛出了RemoteException（包括子类），将调用以下建议：

```
public class RemoteThrowsAdvice implements ThrowsAdvice {

    public void afterThrowing(RemoteException ex) throws Throwable {
        // Do something with remote exception
    }
}
```

与前面的建议不同，下一个示例声明了四个参数，这样它就可以访问被调用的方法、方法参数和目标对象。如果抛出servletexception，将调用以下通知：

更多教程请访问:www.flydean.com

```
public class ServletThrowsAdviceWithArguments implements ThrowsAdvice {

    public void afterThrowing(Method m, Object[] args, Object target, ServletException ex) {
        // Do something with all arguments
    }
}
```

最后一个示例说明了这两个方法如何在处理RemoteException和ServletException的单个类中使用。可以在单个类中组合任意数量的throw advice方法。下面的列表显示了最后一个示例：

```
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
    }
}
```

> 如果throws advice方法本身引发异常，它将重写原始异常（即，它将更改引发给用户的异常）。重写异常通常是RuntimeException，它与任何方法签名都兼容。但是，如果throw advice方法抛出一个选中的异常，它必须与目标方法的声明异常匹配，因此在某种程度上与特定的目标方法签名耦合。不要抛出与目标方法的签名不兼容的未声明的已检查异常！
>
> Throws advice可以在任何pointcut使用。

After Returning Advice

在Spring中after returning advice，必须实现org.springframework.aop.AfterReturningAdvice接口，如下所示：

```
public interface AfterReturningAdvice extends Advice {

    void afterReturning(Object returnValue, Method m, Object[] args, Object target)
            throws Throwable;
}
```

after returning advice可以访问返回值（它不能修改）、调用的方法、方法的参数和目标。

返回通知后，以下内容将统计所有未引发异常的成功方法调用：

```
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;
    }
}
```

此建议不会更改执行路径。如果它抛出一个异常，它将被抛出拦截器链而不是返回值。

> After returning advice可以用在任何pointcut。

Introduction Advice

更多教程请访问:www.flydean.com

Spring将Introduction Advice视为一种特殊的拦截Advice。

Introduction需要一个IntroductionAdvisor和一个实现以下接口的IntroductionInterceptor：

```
public interface IntroductionInterceptor extends MethodInterceptor {

    boolean implementsInterface(Class intf);
}
```

从AOP Alliance MethodIntercetor接口继承的invoke（）方法必须实现introduction。也就是说，如果被调用的方法在引入的接口上，则引入拦截器负责处理方法调用-它不能调用proceed（）。

Introduction advice不能与任何切入点一起使用，因为它只适用于类，而不是方法级别。你只能将 introduction advice与IntroductionAdvisor一起使用，其方法如下：

```
public interface IntroductionAdvisor extends Advisor, IntroductionInfo {

    ClassFilter getClassFilter();

    void validateInterfaces() throws IllegalArgumentException;
}

public interface IntroductionInfo {

    Class[] getInterfaces();
}
```

没有methodMatcher，因此没有与 introduction advice相关联的切入点。只有类筛选是合乎逻辑的。

getInterfaces（）方法返回此advisor引入的接口。

validateInterfaces（）方法在内部用于查看所引入的接口是否可以由配置的IntroductionInterceptor实现。

考虑一下Spring测试套件中的一个示例，假设我们要将以下接口引入一个或多个对象：

```
public interface Lockable {
    void lock();
    void unlock();
    boolean locked();
}
```

这里展示一个mixin。我们希望能够将建议的对象强制转换为Lockable的对象，无论其类型如何，并调用锁定和解锁方法。如果调用lock（）方法，我们希望所有setter方法都抛出lockedException。因此，我们可以添加一个方面，使对象在不了解它的情况下保持不变：AOP的一个很好的例子。

首先，我们需要一个IntroductionInterceptor来完成这项工作。在本例中，我们扩展了org.springframework.aop.support.DelegatingIntroductionInterceptor便利类。我们可以直接实现IntroductionInterceptor，但对于大多数情况，使用DelegatingIntroductionInterceptor是最好的。

DelegingIntroductionInterceptor设计用于将introduction委托给所引入接口的实际实现，从而隐藏使用拦截来完成此操作。可以使用构造函数参数将委托设置为任何对象。默认委托（使用无参数构造函数时）是this。

因此，在下一个示例中，委托是DelegatingIntroductionInterceptorr的lockMixin子类。对于委托（默认情况下，它本身），DelegatingIntroductionInterceptor实例查找委托实现的所有接口（而不是IntroductionInterceptor），并支持针对其中任何一个进行introductions。LockMixin等子类可以调用SuppressInterface（Class Intf）方法来抑制不应公开的接口。然而，不管一个IntroductionInterceptor准备支持多少个接口，IntroductionAdvisor被用来控制哪些接口实际上被公开。引入的接口隐藏了目标对同一接口的任何实现。

更多教程请访问:www.flydean.com

因此，lockmixin扩展了DelegatingIntroductionInterceptor并实现了Lockable接口。超类会自动选取可以支持introduction的Lockable，所以我们不需要指定它。我们可以用这种方式引入任意数量的接口。

注意使用locked的实例变量。这有效地为目标对象中保存的状态添加了额外的状态。

下面的示例显示了lockmixin类的示例：

```
public class LockMixin extends DelegatingIntroductionInterceptor implements Lockable {

    private boolean locked;

    public void lock() {
        this.locked = true;
    }

    public void unlock() {
        this.locked = false;
    }

    public boolean locked() {
        return this.locked;
    }

    public Object invoke(MethodInvocation invocation) throws Throwable {
        if (locked() && invocation.getMethod().getName().indexOf("set") == 0) {
            throw new LockedException();
        }
        return super.invoke(invocation);
    }

}
```

通常，你不需要重写invoke（）方法。DelegatingIntroductionInterceptor实现（如果引入了该方法，则调用delegate方法，否则将进入连接点）通常就足够了。

在本例中，我们需要添加一个检查：如果处于锁定模式，则不能调用setter方法。

所需的introduction只需要保存一个不同的lockmixin实例并指定引入的接口（在本例中，仅Lockable）。一个更复杂的例子可能会引用介绍拦截器（将被定义为原型）。在这种情况下，没有与LockMixin相关的配置，因此我们使用new创建它。下面的示例显示了我们的LockMixinAdvisor类：

```
public class LockMixinAdvisor extends DefaultIntroductionAdvisor {

    public LockMixinAdvisor() {
        super(new LockMixin(), Lockable.class);
    }
}
```

我们可以非常简单地应用这个advisor，因为它不需要配置。（然而，在没有IntroductionAdvisor的情况下，不可能使用IntroductionInterceptor。）与通常的introductions一样，顾问必须是有状态的。对于每个建议的对象，我们需要一个不同的LockMixinAdvisor实例，因此需要LockMixin。advisor是被advised对象状态的一部分。

我们可以通过在XML配置中使用advised.addAdvisor（）方法或（推荐的方法）以编程方式应用此Advisor，就像任何其他Advisor一样。下面讨论的所有代理创建选项，包括“auto-proxy creators”，正确处理introductions 和 stateful 的混合。

更多教程请访问:www.flydean.com

6.3Advisor API

在Spring中，Advisor是一个方面，它只包含与切入点表达式关联的单个advice对象。

除了介绍的特殊情况外，任何advisor都可以与任何advice一起使用。org.springframework.aop.support.DefaultPointcutAdvisor是最常用的Advisor类。它可以与MethodInterceptor、BeforeAdvice或ThrowsAdvice一起使用。

在同一个AOP代理中，可以在Spring中混合Advisor和Advice类型。例如，在一个代理配置中，你可以使用 interception around advice，抛出通知，并在通知之前使用拦截。Spring自动创建必要的拦截器链。

更多教程请访问:www.flydean.com

6.4使用ProxyFactoryBean来创建AOP代理

如果将SpringIOC容器（applicationContext或beanFactory）用于业务对象，你希望使用Spring的AOP FactoryBean实现之一。（请记住，工厂bean引入了一个间接层，允许它创建不同类型的对象。）

> SpringAOP支持还使用了工厂级的bean。

在Spring中创建AOP代理的基本方法是使用org.springframework.aop.framework.ProxyFactoryBean。这可以完全控制切入点、任何适用的advice以及它们的顺序。但是，如果你不需要这样的控制，可以选择更简单的选项。

6.4.1 Basics

ProxyFactoryBean与其他Spring FactoryBean实现一样，引入了一个间接级别。如果定义一个名为foo的ProxyFactoryBean，则引用foo的对象不会看到ProxyFactoryBean实例本身，而是由ProxyFactoryBean中getobject（）方法的实现创建的对象。此方法创建包装目标对象的AOP代理。

使用ProxyFactoryBean或其他具有IOC意识的类来创建AOP代理最重要的好处之一是，通知和切入点也可以由IOC管理。这是一个强大的特性，使得某些方法很难用其他AOP框架实现。例如，一个建议本身可以引用应用程序对象（除了目标之外，目标应该在任何AOP框架中都可用），这得益于依赖注入提供的所有可插入性。

6.4.2 JavaBean属性

与Spring提供的大多数FactoryBean实现相同，proxyFactoryBean类本身就是一个JavaBean。其属性用于：

- 指定要代理的目标。
- 指定是否使用cglib（稍后介绍，另请参阅基于jdk和cglib的代理）。

一些关键属性继承自org.spring framework.aop.framework.ProxyConfig（Spring中所有AOP代理工厂的超类）。这些关键属性包括：

- proxyTargetClass：如果要代理目标类，而不是目标类的接口，则为true。如果将此属性值设置为true，则会创建CGLIB代理（也可参见基于JDK和CGLIB的代理）。
- 优化：控制是否对通过cglib创建的代理应用积极的优化。除非你完全理解相关AOP代理如何处理优化，否则不应随意使用此设置。这目前仅用于cglib代理。它对JDK动态代理没有影响。
- 冻结：如果代理配置被冻结，则不再允许更改配置。这对于轻微的优化和不希望调用方能够在创建代理后（通过建议的接口）操纵代理的情况都很有用。此属性的默认值为false，因此允许进行更改（例如添加其他通知）。
- ExposeProxy：确定当前代理是否应在ThreadLocal中公开，以便目标可以访问它。如果目标需要获取代理，并且ExposeProxy属性设置为true，则该目标可以使用AopContext.currentProxy（）方法。

ProxyFactoryBean特有的其他属性包括：

- proxyInterfaces：字符串接口名称的数组。如果没有提供，将使用目标类的cglib代理（也可参见基于jdk和cglib的代理）。
- 拦截器：要应用的顾问、拦截器或其他建议名称的字符串数组。Ordering非常重要，以先到先得的方式提供。也就是说，列表中的第一个拦截器是第一个能够拦截调用的拦截器。

名称是当前工厂中的bean名称，包括来自祖先工厂的bean名称。这里不能提到bean引用，因为这样做会导致ProxyFactoryBean忽略通知的singleton设置。

你可以附加一个带有星号（*）的拦截器名称。这样做会导致所有AdvisorBean的应用，其名称以要应用的星号前面的部分开头。你可以在使用“全局”Advisor中找到使用此功能的示例。

更多教程请访问:www.flydean.com

singleton：工厂是否应该返回单个对象，无论调用getObject（）方法的频率如何。一些FactoryBean实现提供了这样的方法。默认值为true。如果你想使用有状态的建议（例如，对于有状态的混合），可以使用原型建议和单例值false。

6.4.3 JDK- and CGLIB-based代理

本节是关于ProxyFactoryBean如何选择为特定目标对象（要代理）创建基于JDK的代理或基于CGLIB的代理的最终文档。

> ProxyFactoryBean在创建基于jdk或cglib的代理方面的行为在Spring的1.2.x和2.0版本之间发生了变化。在自动检测接口方面，ProxyFactoryBean现在显示出与TransactionProxyFactoryBean类相似的语义。

如果要代理的目标对象的类（以下简称为目标类）不实现任何接口，则创建基于CGLIB的代理。这是最简单的场景，因为JDK代理是基于接口的，没有接口意味着JDK代理甚至不可能实现。你可以插入目标bean并通过设置interceptorNames属性指定拦截器列表。注意，即使ProxyFactoryBean的proxyTargetClass属性设置为false，也会创建基于cglib的代理。（这样做毫无意义，最好从bean定义中删除，因为它充其量是多余的，最坏的情况是令人困惑。）

如果目标类实现一个（或多个）接口，则创建的代理的类型取决于proxyFactoryBean的配置。

如果proxyFactoryBean的proxyTargetClass属性设置为true，则会创建基于cglib的代理。这是有道理的，符合最不令人惊讶的原则。即使proxyFactoryBean的proxyInterfaces属性已设置为一个或多个完全限定的接口名称，proxyTargetClass属性设置为true的事实也会导致基于cglib的代理生效。

如果proxyFactoryBean的proxyInterfaces属性已设置为一个或多个完全限定的接口名称，则将创建基于JDK的代理。创建的代理实现在proxyInterfaces属性中指定的所有接口。如果目标类恰好实现了比proxyInterfaces属性中指定的接口多得多的接口，那么这一切都很好，但是这些附加接口不是由返回的代理实现的。

如果尚未设置proxyfactorybean的proxyInterfaces属性，但目标类确实实现了一个（或多个）接口，则proxyfactorybean会自动检测到目标类确实实现了至少一个接口，并创建了基于JDK的代理。实际代理的接口是目标类实现的所有接口。实际上，这与向proxyInterfaces属性提供目标类实现的每个接口的列表相同。然而，它的工作量明显减少，而且不容易出现排版错误。

6.4.4 代理接口

考虑一个简单的ProxyFactoryBean实例。这个例子涉及：

- 代理的目标bean。这是示例中的PersonTarget bean定义。
- 用来提供建议的Advisor和Interceptor。
- AOP代理bean定义，用于指定目标对象（PersonTarget bean）

下面的列表显示了示例：

```
<bean id="personTarget" class="com.mycompany.PersonImpl">
    <property name="name" value="Tony"/>
    <property name="age" value="51"/>
</bean>

<bean id="myAdvisor" class="com.mycompany.MyAdvisor">
    <property name="someProperty" value="Custom string property value"/>
</bean>

<bean id="debugInterceptor" class="org.springframework.aop.interceptor.DebugInterceptor">
</bean>

<bean id="person"
    class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces" value="com.mycompany.Person"/>

    <property name="target" ref="personTarget"/>
```

更多教程请访问:www.flydean.com

```
    <property name="interceptorNames">
        <list>
            <value>myAdvisor</value>
            <value>debugInterceptor</value>
        </list>
    </property>
</bean>
```

注意，interceptorNames属性采用一个字符串列表，其中包含当前工厂中拦截器或顾问的bean名称。你可以在返回之前、之后使用顾问、拦截器，并抛出advice对象。advisors的ordering意义重大。

> 你可能想知道为什么列表不包含bean引用。原因是，如果proxyfactorybean的singleton属性设置为false，它必须能够返回独立的代理实例。如果任何顾问本身是原型，则需要返回一个独立的实例，因此需要能够从工厂获取原型的实例。持有reference是不够的。

前面显示的PersonBean定义可以用来代替Person实现，如下所示：

```
Person person = (Person) factory.getBean("person");
```

同一个IOC上下文中的其他bean可以用普通Java对象表示它的强类型依赖性。以下示例显示了如何执行此操作：

```
<bean id="personUser" class="com.mycompany.PersonUser">
    <property name="person"><ref bean="person"/></property>
</bean>
```

此示例中的PersonUser类暴露了一个Person类型的属性。就其而言，可以透明地使用AOP代理来代替“真实的”人员实现。但是，它的类将是动态代理类。可以将其转换为Advised接口（稍后讨论）。

你可以使用匿名内部bean隐藏目标和代理之间的区别。只有ProxyFactoryBean定义不同。使用该advice仅为了完整性。下面的示例演示如何使用匿名内部bean：

```
<bean id="myAdvisor" class="com.mycompany.MyAdvisor">
    <property name="someProperty" value="Custom string property value"/>
</bean>

<bean id="debugInterceptor" class="org.springframework.aop.interceptor.DebugInterceptor"/>

<bean id="person" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces" value="com.mycompany.Person"/>
    <!-- Use inner bean, not local reference to target -->
    <property name="target">
        <bean class="com.mycompany.PersonImpl">
            <property name="name" value="Tony"/>
            <property name="age" value="51"/>
        </bean>
    </property>
    <property name="interceptorNames">
        <list>
            <value>myAdvisor</value>
            <value>debugInterceptor</value>
        </list>
    </property>
</bean>
```

使用匿名内部bean的优点是只有一个person类型的对象。如果我们希望防止应用程序上下文的用户获得对un-advised对象的引用，或者需要避免SpringIOC autowiring的任何模糊性，那么这是非常有用的。也可以说，proxyfactorybean定义是独立的，这是一个优势。但是，有时能够从工厂获得un-advised的目标实际上是一种优势（例如，在某些测试场景中）。

更多教程请访问:www.flydean.com

6.4.5 代理Classes

如果你需要代理一个类，而不是一个或多个接口呢?

假设在前面的示例中，没有Person界面。我们需要通知一个名为person的类，该类没有实现任何业务接口。在这种情况下，可以将spring配置为使用cglib代理，而不是动态代理。为此，请将前面显示的proxyFactoryBean上的proxyTargetClass属性设置为true。虽然最好是编程到接口而不是类，但是在处理遗留代码时，advise不实现接口的类的能力是非常有用的。（一般来说，Spring不是规定性的。虽然它使应用好的实践变得容易，但它避免了强制使用特定的方法。）

如果你愿意，你可以在任何情况下强制使用cglib，即使你确实有接口。

CGLIB代理通过在运行时生成目标类的子类来工作。Spring将生成的子类配置为将方法调用委托给原始目标。子类用于实现装饰器模式，编织在advice中。

cglib代理通常对用户是透明的。但是，有一些问题需要考虑：

- 无法通知Final方法，因为它们不能被重写。
- 不需要将cglib添加到类路径中。从Spring3.2开始，cglib被重新包装并包含在SpringCoreJar中。换句话说，基于cglib的AOP和JDK动态代理一样“开箱即用”。

CGLIB代理和动态代理的性能差别不大。在这种情况下，性能不应是决定性的考虑因素。

6.4.6 使用全局的Advisors

通过向拦截器名称添加星号，所有具有与星号之前部分匹配的bean名称的advisor都将添加到advisor链中。如果你需要添加一组标准的“全局”Advisor，这将非常有用。以下示例定义了两个全局Advisors：

```
<bean id="proxy" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target" ref="service"/>
    <property name="interceptorNames">
        <list>
            <value>global*</value>
        </list>
    </property>
</bean>

<bean id="global_debug" class="org.springframework.aop.interceptor.DebugInterceptor"/>
<bean id="global_performance" class="org.springframework.aop.interceptor.PerformanceMonitorInterceptor"/>
```

更多教程请访问:www.flydean.com

6.5简介的代理定义

特别是当你对transactional代理下定义时，你可以用大量类似的代理定义来结束。父类和子类bean，内部bean都可以实现更加干净和简洁的代理定义。

首先，我们为代理创建一个父类模板定义，如下所示：

```
<bean id="txProxyTemplate" abstract="true"
        class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager" ref="transactionManager"/>
    <property name="transactionAttributes">
        <props>
            <prop key="*">PROPAGATION_REQUIRED</prop>
        </props>
    </property>
</bean>
```

上面的类并没有实例化，所以他并不是完整的。然后每个需要被创建的代理都是它的子类，它包装了定义为内部类的代理target，因为该target不会以他自己的方式使用。下面是一个子类的例子：

```
<bean id="myService" parent="txProxyTemplate">
    <property name="target">
        <bean class="org.springframework.samples.MyServiceImpl">
        </bean>
    </property>
</bean>
```

你可以覆盖父类的属性，如下所示：

```
<bean id="mySpecialService" parent="txProxyTemplate">
    <property name="target">
        <bean class="org.springframework.samples.MySpecialServiceImpl">
        </bean>
    </property>
    <property name="transactionAttributes">
        <props>
            <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>
            <prop key="find*">PROPAGATION_REQUIRED,readOnly</prop>
            <prop key="load*">PROPAGATION_REQUIRED,readOnly</prop>
            <prop key="store*">PROPAGATION_REQUIRED</prop>
        </props>
    </property>
</bean>
```

注意在父类的例子中，我们显示的设置abstract为true，如上面所示，所以他并没有被初始化，Application contexts（而不是简单的bean factories），会默认初始化所有的singletons。因此，如果你有一个父bean的定义，你打算只把他作为模板，并且他是一个类，那么你必须将他的abstract属性设置为true。否则application context 会尝试预先实例化他。

更多教程请访问:www.flydean.com

6.6使用ProxyFactory创建AOP代理

用Spring编程创建AOP代理很容易。这允许你使用SpringAOP而不依赖SpringIOC。

由目标对象实现的接口被自动代理。下面的列表显示使用一个拦截器和一个顾问为目标对象创建代理：

```
ProxyFactory factory = new ProxyFactory(myBusinessInterfaceImpl);
factory.addAdvice(myMethodInterceptor);
factory.addAdvisor(myAdvisor);
MyBusinessInterface tb = (MyBusinessInterface) factory.getProxy();
```

第一步是构造org.springframework.aop.framework.proxyFactory类型的对象。你可以使用目标对象创建此对象，如前面的示例所示，或者指定要在备用构造函数中代理的接口。

你可以添加建议（拦截器作为一种专门的建议）、顾问，或者两者兼而有之，并在代理工厂的生命周期中操纵它们。如果添加一个IntroductionInterceptionAroundAdvisor，则可以使代理实现其他接口。

ProxyFactory（继承自AdvisedSupport）上还有一些方便方法，允许你添加其他通知类型，如before和throw advice。AdvisedSupport是ProxyFactory和ProxyFactoryBean的超类。

> 在大多数应用程序中，将AOP代理创建与IOC框架集成是最佳实践。我们建议你一般用AOP将Java代码的配置外部化。

更多教程请访问:www.flydean.com

6.7操作被通知的对象

但是，你可以通过使用org.springframework.aop.framework.advised接口来创建AOP代理，从而对其进行操作。任何AOP代理都可以强制转换到此接口，不管它实现了哪个其他接口。此接口包括以下方法：

```
Advisor[] getAdvisors();

void addAdvice(Advice advice) throws AopConfigException;

void addAdvice(int pos, Advice advice) throws AopConfigException;

void addAdvisor(Advisor advisor) throws AopConfigException;

void addAdvisor(int pos, Advisor advisor) throws AopConfigException;

int indexOf(Advisor advisor);

boolean removeAdvisor(Advisor advisor) throws AopConfigException;

void removeAdvisor(int index) throws AopConfigException;

boolean replaceAdvisor(Advisor a, Advisor b) throws AopConfigException;

boolean isFrozen();
```

getAdvisors（）方法返回已添加到工厂的每个Advisor、拦截器或其他通知类型的顾问。如果添加了Advisor，则此索引处返回的advisor是你添加的对象。如果你添加了一个拦截器或其他通知类型，那么Spring将它包装在一个顾问中，并使用一个始终返回true的切入点。因此，如果添加了一个SystemInterceptor，则为该索引返回的Advisor是一个返回SystemInterceptor的DefaultPointcutAdvisor，以及一个匹配所有类和方法的切入点。

addAdvisor（）方法可用于添加任何Advisor。通常，持有切入点和建议的Advisor是DefaultPointcutAdvisor，你可以将其用于任何建议或切入点（但不用于introductions）。

默认情况下，即使创建了代理，也可以添加或删除顾问或拦截器。唯一的限制是不可能添加或删除简介顾问，因为工厂的现有代理不显示接口更改。（你可以从工厂获取新代理以避免此问题。）

下面的示例显示将AOP代理强制转换到Advised的接口，并检查和操作其advice：

```
Advised advised = (Advised) myObject;
Advisor[] advisors = advised.getAdvisors();
int oldAdvisorCount = advisors.length;
System.out.println(oldAdvisorCount + " advisors");

// Add an advice like an interceptor without a pointcut
// Will match all proxied methods
// Can use for interceptors, before, after returning or throws advice
advised.addAdvice(new DebugInterceptor());

// Add selective advice using a pointcut
advised.addAdvisor(new DefaultPointcutAdvisor(mySpecialPointcut, myAdvice));

assertEquals("Added two advisors", oldAdvisorCount + 2, advised.getAdvisors().length);
```

> 尽管无疑存在合法的使用案例，但是否建议（不打算使用双关语）修改生产中业务对象的建议仍然存在疑问。但是，它在开发中非常有用（例如，在测试中）。我们有时发现，能够以拦截器或其他建议的形式添加测试代码，进入我们想要测试的方法调用中是非常有用的。（例如，在将事务标记为回滚之前，建议可以进入为该方法创建的事务中，或者运行SQL检查数据库是否正确更新。）

更多教程请访问:www.flydean.com

根据创建代理的方式，通常可以设置frozen标志。在这种情况下，Advised isfrozen（）方法返回true，任何通过添加或删除来修改通知的尝试都会导致AopConfigException。在某些情况下（例如，为了防止调用代码删除安全拦截器），冻结建议对象的状态的功能非常有用。

更多教程请访问:www.flydean.com

6.8使用auto-proxy功能

到目前为止，我们已经考虑通过使用ProxyFactoryBean或类似的工厂bean显式地创建AOP代理。

Spring还允许我们使用“auto-proxy”bean定义，它可以自动代理选定的bean定义。这是在Spring的“bean post processor”基础设施上构建的，它支持在容器加载时修改任何bean定义。

在这个模型中，你在XML bean定义文件中设置了一些特殊的bean定义，以配置自动代理基础结构。这允许你声明符合自动代理条件的目标。你不需要使用ProxyFactoryBean。

有两种方法可以做到这一点：

- 通过使用在当前上下文中引用特定bean的自动代理创建者。
- 一个需要单独考虑的自动代理创建的特殊情况：由源代码级元数据属性驱动的自动代理创建。

6.8.1 Auto-proxy Bean定义

本节介绍由org.springframework.aop.framework.autoproxy包提供的自动代理创建者。

BeanNameAutoProxyCreator

BeannameAutoProxyCreator类是一个BeanPostProcessor，它自动为名称与文字值或通配符匹配的Bean创建AOP代理。下面的示例演示如何创建BeanNameAutoProxyCreator bean：

```
<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
    <property name="beanNames" value="jdk*,onlyJdk"/>
    <property name="interceptorNames">
        <list>
            <value>myInterceptor</value>
        </list>
    </property>
</bean>
```

与ProxyFactoryBean一样，有一个interceptorNames属性而不是一个拦截器列表，以允许原型顾问的正确行为。命名的“拦截器”可以是顾问或任何建议类型。

与一般的自动代理一样，使用BeanNameAutoProxyCreator的主要目的是将相同的配置一致地应用于多个对象，配置量最小。它是将声明性事务应用于多个对象的常用选择。

名称匹配的bean定义（如前面示例中的jdkMyBean和onlyjdk）是带有目标类的普通旧bean定义。AOP代理由BeanNameAutoProxyCreator自动创建。同样的建议也适用于所有匹配的bean。注意，如果使用advisors（而不是前面示例中的拦截器），则切入点可能会不同地应用于不同的bean。

DefaultAdvisorAutoProxyCreator

一个更通用和强大的自动代理创建者是DefaultAdvisorAutoProxyCreator。这将在当前上下文中自动应用合格的advisors，而不需要在自动代理顾问的bean定义中包含特定的bean名称。它提供了与BeanNameAutoProxyCreator相同的配置和避免重复的优点。

使用此机制涉及：

- 指定DefaultAdvisorAutoProxyCreator bean定义。
- 在相同或相关的上下文中指定任意数量的顾问。请注意，这些必须是顾问，而不是拦截器或其他建议。这是必要的，因为必须有一个切入点来评估，以检查每个建议对候选bean定义的合格性。

更多教程请访问:www.flydean.com

DefaultAdvisorAutoProxyCreator自动评估每个顾问中包含的切入点，以查看它应该应用于每个业务对象（如示例中的BusinessObject1和BusinessObject2）的建议。

这意味着任何数量的顾问都可以自动应用于每个业务对象。如果任何顾问中没有与业务对象中的任何方法匹配的切入点，则不会代理该对象。当为新的业务对象添加bean定义时，如果需要，它们将被自动代理。

一般来说，自动代理的优点是使调用者或依赖项无法获得未建议的对象。在此applicationContext上调用getbean（“businessobject1”）将返回AOP代理，而不是目标业务对象。（前面显示的“内部bean”也提供了这个好处。）

下面的示例创建一个DefaultAdvisorAutoProxyCreator bean和本节讨论的其他元素：

```
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>

<bean class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
    <property name="transactionInterceptor" ref="transactionInterceptor"/>
</bean>

<bean id="customAdvisor" class="com.mycompany.MyAdvisor"/>

<bean id="businessObject1" class="com.mycompany.BusinessObject1">
    <!-- Properties omitted -->
</bean>

<bean id="businessObject2" class="com.mycompany.BusinessObject2"/>
```

如果你希望将相同的建议一致地应用于许多业务对象，那么DefaultAdvisorAutoProxyCreator非常有用。一旦基础结构定义就位，就可以添加新的业务对象，而不包括特定的代理配置。你还可以通过对配置的最小更改轻松地进入其他方面（例如，跟踪或性能监视方面）。

DefaultAdvisorAutoProxyCreator支持过滤（通过使用命名约定，只评估某些advisors ，从而允许在同一工厂中使用多个不同配置的AdvisorAutoProxyCreators）和排序。Advisors可以实现org.springframework.core.Ordered接口，以确保在出现问题时正确排序。在前面的示例中使用的TransactionAttributeSourceAdvisor具有可配置的order值。默认设置是无序的。

更多教程请访问:www.flydean.com

6.9使用TargetSource的实现

6.9 使用TargetSource的实现

Spring提供了TargetSource的概念，用org.springframework.aop.TargetSource接口表示。此接口负责返回实现连接点的“目标对象”。每次AOP代理处理方法调用时，都会要求TargetSource实现一个目标实例。

使用SpringAOP的开发人员通常不需要直接使用TargetSource实现，但这提供了支持池、热交换和其他复杂目标的强大方法。例如，池目标源可以通过使用池来管理实例，为每次调用返回不同的目标实例。

如果不指定TargetSource，则使用默认实现包装本地对象。每次调用都会返回相同的目标（如你所期望的那样）。

本节的其余部分描述了Spring提供的标准目标源以及如何使用它们。

> 当使用自定义目标源时，你的目标通常需要是原型而不是单例bean定义。这允许Spring在需要时创建新的目标实例。

6.9.1 热交换目标源

存在org.springframework.aop.target.HotSwappableTargetSource以允许AOP代理的目标切换，同时允许调用方保留对它的引用。

更改目标源的目标将立即生效。HotSwappableTargetSource是线程安全的。

你可以使用HotSwappableTargetSource上的swap（）方法更改目标，如下示例所示：

```
HotSwappableTargetSource swapper = (HotSwappableTargetSource) beanFactory.getBean("swapper");
Object oldTarget = swapper.swap(newTarget);
```

下面是需要的XML定义：

```
<bean id="initialTarget" class="mycompany.OldTarget"/>

<bean id="swapper" class="org.springframework.aop.target.HotSwappableTargetSource">
    <constructor-arg ref="initialTarget"/>
</bean>

<bean id="swappable" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="targetSource" ref="swapper"/>
</bean>
```

swap()的执行改变了swappable bean的目标。持有对该bean的引用的客户端不知道该更改，并且立即开始使用新的目标。

尽管此示例不添加任何advice（不需要添加advice以使用TargetSource），但任何TargetSource都可以与任意advice结合使用。

6.9.2 Pooling Target Sources

使用pooling target source为无状态会话EJB提供了类似的编程模型，其中维护了相同实例的池，方法调用将释放池中的对象。

更多教程请访问:www.flydean.com

Spring池和SLSB池的一个重要区别是Spring池可以应用于任何POJO。与一般的Spring一样，此服务可以以非侵入性的方式应用。

Spring支持Commons Pool 2.2，它提供了一个相当有效的池实现。你需要应用程序类路径上的commons-pool来使用此功能。你还可以将org.springframework.aop.target.AbstractPoolingTargetSource子类化，以支持任何其他池API。

> 在Spring Framework 4.2中，Commons Pool 1.5+已不被推荐使用。

下面是一个配置的例子：

```
<bean id="businessObjectTarget" class="com.mycompany.MyBusinessObject"
        scope="prototype">
    ... properties omitted
</bean>

<bean id="poolTargetSource" class="org.springframework.aop.target.CommonsPool2TargetSource">
    <property name="targetBeanName" value="businessObjectTarget"/>
    <property name="maxSize" value="25"/>
</bean>

<bean id="businessObject" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="targetSource" ref="poolTargetSource"/>
    <property name="interceptorNames" value="myInterceptor"/>
</bean>
```

请注意，目标对象（前面示例中的BusinessObjectTarget）必须是原型。这允许poolingTargetSource实现创建目标的新实例，以便在必要时扩大池。有关其属性的信息，请参阅AbstractPoolingTargetSource的JavaDoc和你希望使用的具体子类。maxSize是最基本的属性，并必须要被设置。

在这种情况下，myInterceptor是需要在相同的IOC上下文中定义的拦截器的名称。但是，不需要指定拦截器来使用池。如果只需要池而不需要其他建议，那么根本不要设置interceptorNames属性。

你可以将Spring配置为能够将任何池对象强制转换为org.springframework.aop.target.PoolingConfig接口，该接口通过简介公开池的配置和当前大小的信息。你需要定义类似以下内容的顾问：

```
<bean id="poolConfigAdvisor" class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
    <property name="targetObject" ref="poolTargetSource"/>
    <property name="targetMethod" value="getPoolingConfigMixin"/>
</bean>
```

通过对AbstractPoolingTargetSource类调用一个方便的方法来获得此Advisor，因此使用了MethodInvokingFactoryBean。此顾问的名称（此处为PoolConfigdVisor）必须位于公开池对象的ProxyFactoryBean中拦截器名称的列表中。

其定义如下：

```
PoolingConfig conf = (PoolingConfig) beanFactory.getBean("businessObject");
System.out.println("Max pool size is " + conf.getMaxSize());
```

通常不需要Pooling无状态服务对象。我们不认为这应该是默认的选择，因为大多数无状态对象都是自然线程安全的，如果缓存资源，实例池也会有问题。

使用自动代理可以使池的使用更加简单。TargetSource的实现可以被用在任何auto-proxy的创建者。

6.9.3 Prototype Target Sources

更多教程请访问:www.flydean.com

设置“prototype”目标源类似于设置池TargetSource。在这种情况下，每次方法调用都会创建目标的新实例。虽然在现代的JVM中创建新对象的成本并不高，但是连接新对象（满足其IOC依赖性）的成本可能更高。因此，如果没有很好的理由，你不应该使用这种方法。

为此，你可以修改前面显示的poolTargetSource定义，如下所示（为了清晰起见，我们还更改了名称）：

```
<bean id="prototypeTargetSource" class="org.springframework.aop.target.PrototypeTargetSource">
    <property name="targetBeanName" ref="businessObjectTarget"/>
</bean>
```

唯一的属性是目标bean的名称。在TargetSource实现中使用继承以确保一致的命名。与池目标源一样，目标bean必须是原型bean定义。

6.9.4 ThreadLocal Target Sources

如果需要为每个传入请求（即每个线程）创建一个对象，则ThreadLocal目标源非常有用。ThreadLocal的概念提供了一个JDK范围的工具，可以透明地将资源存储在线程旁边。设置ThreadLocalTargetSource与其他类型的目标源的说明几乎相同，如下示例所示：

```
<bean id="threadlocalTargetSource" class="org.springframework.aop.target.ThreadLocalTargetSource">
    <property name="targetBeanName" value="businessObjectTarget"/>
</bean>
```

ThreadLocal实例在多线程和多类加载器环境中错误地使用它们时会出现严重问题（可能导致内存泄漏）。你应该始终考虑在其他类中包装一个threadlocal，而不要直接使用threadlocal本身（包装类中除外）。此外，你应该始终记住正确地设置和取消设置（后者只涉及对ThreadLocal.set(null)）线程的本地资源。在任何情况下都应进行取消设置，因为不取消设置可能会导致问题行为。Spring的threadLocal支持为你提供了这一点，应该始终考虑在没有其他适当处理代码的情况下使用threadLocal实例。

更多教程请访问:www.flydean.com

6.10定义新的Advice Types

SpringAOP设计为可扩展的。虽然拦截实现策略目前在内部使用，但是除了around advice的拦截之外，还可以支持任意的通知类型，包括before, throws advice,和after returning advice。

org.springframework.aop.framework.adapter包是一个SPI包，它允许在不更改核心框架的情况下添加对新自定义通知类型的支持。对自定义Advice类型的唯一约束是它必须实现org.aopalliance.aop.Advice接口。

有关更多信息，请参阅org.springframework.aop.framework.adapter javadoc。

更多教程请访问:www.flydean.com

7.Null-safety

虽然Java不允许用它的类型系统来表示null-safety，但是Spring框架现在在org.springframework.lang包中提供以下注解，以允许你声明API和字段的可空性：

@Nullable：注解，指示特定参数、返回值或字段可以为空。 @NonNull：表示特定参数、返回值或字段不能为空的注解（对于分别应用@NonNullApi 和@NonNullFields的参数/返回值和字段不需要）。 @NonNullApi：包级别的注解，声明非空作为参数和返回值的默认语义。 @NonNullFields:包级别的注解，声明非空字段作为字段的默认语义。

Spring框架本身利用这些注解，但它们也可以用于任何基于Spring的Java项目中，以声明 null-safe API和可选的 null-safe 字段。尚不支持泛型类型参数、varargs和数组元素为空性，但应在即将发布的版本中提供，有关最新信息，请参阅spr-15942。可以为空性声明预计将在Spring框架版本之间进行微调，包括较小的版本。方法体内部使用的类型的Nullability超出了此功能的范围。

> 其他公共库（如Reactor和spring data）提供了null-safe API，这些API使用类似的空性安排，为spring应用程序开发人员提供了一致的总体体验。

7.1 使用案例

除了为SpringFrameworkAPI的可空性提供显式声明外，这些注解还可以由一个IDE（如IDEA或Eclipse）使用，以提供与null-safety相关的有用警告，以避免在运行时发生NullPointerException。

它们还用于在Kotlin项目中使SpringAPI为空，因为Kotlin本身支持空安全。Kotlin支持文档中提供了更多详细信息。

7.2 JSR-305 元注解

Spring注解是用JSR 305注解（一个隐藏的，但广泛分布的JSR）进行元注解的。JSR-305元注解允许工具供应商（如IDEA或Kotlin）以通用的方式提供空安全支持，而无需对Spring注解进行硬编码支持。

没有必要也不建议向项目类路径添加JSR-305依赖项，以利用Spring空安全API。只有基于Spring的库等项目在其代码库中使用空安全注解时，才应添加com.google.code.findbugs:jsr305:3.0.2和compileOnly Gradle配置或maven provided范围以避免编译警告。

更多教程请访问:www.flydean.com

8.数据缓存和解码器

Java NIO提供ByteBuffer，但许多库在顶部构建自己的字节缓冲区API，尤其是对于重用缓冲区和/或使用直接缓冲区有利于性能的网络操作。 例如，Netty具有ByteBuf层次结构，Undertow使用XNIO，Jetty使用pooled字节缓冲区并释放回调，依此类推。 spring-core模块提供了一组抽象来处理各种字节缓冲区API，如下所示：

- DataBufferFactory抽象创建数据缓冲区。
- DataBuffer表示可以合并的字节缓冲区。
- DataBufferUtils为数据缓冲区提供实用程序方法。
- Codecs将流数据缓冲流解码或编码为更高级别的对象。

8.1 DataBufferFactory

DataBufferFactory用以下两种方式之一创建数据缓冲区：

1. 分配一个新的数据缓冲区，可选择预先指定capacity（如果已知），即使DataBuffer的实现可以按需增长和缩小，这也更有效。
2. 包装现有的byte []或java.nio.ByteBuffer，它使用DataBuffer实现来修饰给定的数据，并且不涉及分配。

请注意，WebFlux应用程序不直接创建DataBufferFactory，而是通过ServerHttpResponse或客户端的ClientHttpRequest访问它。 工厂类型取决于底层客户端或服务器，例如 Reactor Netty的NettyDataBufferFactory，其他的DefaultDataBufferFactory。

8.2 DataBuffer

DataBuffer接口提供与java.nio.ByteBuffer类似的操作，但也带来了一些额外的好处，其中一些受Netty ByteBuf的启发。 以下是部分列表：

- 使用独立位置进行读写，即不需要调用flip（）来在读写之间交替。
- 与java.lang.StringBuilder一样，按需扩展容量。
- 通过PooledDataBuffer汇集缓冲区和引用计数。
- 以java.nio.ByteBuffer，InputStream或OutputStream的形式查看缓冲区。
- 确定给定字节的索引或最后一个索引。

8.3 PooledDataBuffer

正如Javadoc for ByteBuffer中所解释的，字节缓冲区可以是直接缓冲区，也可以是非直接缓冲区。直接缓冲区可以驻留在Java堆之外，这样就无需复制本机I / O操作。这使得直接缓冲区对于通过套接字接收和发送数据特别有用，但是创建和释放它们也更加昂贵，这导致了pooling缓冲区的想法。

PooledDataBuffer是DataBuffer的扩展，它有助于引用计数，这对于字节缓冲池是必不可少的。它是如何工作的？当分配PooledDataBuffer时，引用计数为1.调用retain（）递增计数，而对release（）的调用则递减计数。只要计数大于0，就保证缓冲区不被释放。当计数减少到0时，可以释放pooled缓冲区，这实际上可能意味着缓冲区的保留内存返回到内存池。

请注意，不是直接对PooledDataBuffer进行操作，在大多数情况下，最好使用DataBufferUtils中的方法，只有当它是PooledDataBuffer的实例时才应用release或retain到DataBuffer。

8.4 DataBufferUtils

更多教程请访问:www.flydean.com

DataBufferUtils提供了许多用于操作数据缓冲区的实用方法：

- 将数据缓冲区流加入单个缓冲区中，可能只有零拷贝，例如：如果底层字节缓冲区API支持可以复合缓冲区。
- 将InputStream或NIO Channel转换为Flux，反之亦然，将Publisher转换为OutputStream或NIO Channel。
- 如果缓冲区是PooledDataBuffer的实例，则释放或保留DataBuffer的方法。
- 从字节流中跳过或取出，直到特定的字节数。

8.5 Codecs

org.springframework.core.codec 包提供了下面的策略接口：

- Encoder 用来将Publisher 编码为数据缓存的stream。
- Decoder 将Publisher解码为更高级别的对象流。

spring-core模块提供byte[]，ByteBuffer，DataBuffer，Resource和String编码器和解码器实现。 spring-web模块增加了Jackson JSON，Jackson Smile，JAXB2，Protocol Buffers和其他编码器和解码器。 请参阅WebFlux部分中的编解码器。

8.6 使用DataBuffer

使用数据缓冲区时，必须特别注意确保缓冲区被释放，因为它们可能被pooled。我们将使用codecs来说明它是如何工作的，但概念更普遍适用。让我们看看codecs在内部管理数据缓冲区时做了什么。

Decoder是在创建更高级别对象之前，最后一个读取输入数据缓冲区的，因此必须按如下方式释放它们：

1. 如果Decoder只是读取每个输入缓冲区并准备立即释放它，它可以通过DataBufferUtils.release（dataBuffer）来实现。
2. 如果Decoder正在使用Flux或Mono运算符（如flatMap，reduce等）在内部预取和缓存数据项，或者正在使用filter，skip和其他省略项的运算符，那么doOnDiscard(PooledDataBuffer.class, DataBufferUtils::release)必须被添加到组合链中以确保在丢弃之前（错误或取消信号）释放这些缓冲区，。
3. 如果Decoder以任何其他方式保持一个或多个数据缓冲区，则必须确保在完全读取时（或者在读取和释放高速缓存数据缓冲区之前发生错误或取消信号）释放它们。

请注意，DataBufferUtils#join提供了一种安全有效的方法，可将数据缓冲区流聚合到单个数据缓冲区中。同样，skipUntilByteCount和takeUntilByteCount是decoders使用的其他安全方法。

Encoder分配其他人必须读取（和释放）的数据缓冲区。所以Encoder没什么可做的。但是，如果在使用数据填充缓冲区时发生序列化错误，则Encoder必须注意释放数据缓冲区。例如：

```
DataBuffer buffer = factory.allocateBuffer();
boolean release = true;
try {
    // serialize and populate buffer..
    release = false;
}
finally {
    if (release) {
        DataBufferUtils.release(buffer);
    }
}
return buffer;
```

Encoder的使用者负责释放它接收的数据缓冲区。 在WebFlux应用程序中，Encoder的输出用于写入HTTP服务器响应或客户端HTTP请求，在这种情况下，释放数据缓冲区是代码写入服务器响应或客户端的责任。

请注意，在Netty上运行时，可以使用调试选项来排除缓冲区泄漏。

更多教程请访问:www.flydean.com

更多教程请访问:www.flydean.com

9.附录

附录

更多教程请访问:www.flydean.com

9.1XML Schemas

附录的本部分列出了和核心容器相关的XML schemas。

9.1.1 util Schema

顾名思义，util标签处理常见的实用程序配置问题，例如配置集合，引用常量等。 要在util schema中使用标记，你需要在Spring XML配置文件的顶部有以下前导码（片段中的文本引用正确的模式，以便util命名空间中的标记可供你使用）：

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

        <!-- bean definitions here -->
</beans>
```

使用<util:constant/>

考虑如下的bean定义：

```
<bean id="..." class="...">
    <property name="isolation">
        <bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE"
                class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean" />
    </property>
</bean>
```

上述配置使用Spring FactoryBean实现（FieldRetrievingFactoryBean）将bean上隔离属性的值设置为java.sql.Connection.TRANSACTION_SERIALIZABLE常量的值。 这一切都很好，但它很冗长，并且（不必要地）将Spring的内部管道暴露给最终用户。

以下基于XML Schema的版本更简洁，清楚地表达了开发人员的意图（“注入此常量值”），并且它可读性更好：

```
<bean id="..." class="...">
    <property name="isolation">
        <util:constant static-field="java.sql.Connection.TRANSACTION_SERIALIZABLE"/>
    </property>
</bean>
```

使用字段值设置Bean属性或者构造函数参数

FieldRetrievingFactoryBean是一个FactoryBean，用于检索静态或非静态字段值。 它通常用于检索公共静态最终常量，然后可以使用它来为另一个bean设置属性值或构造函数参数。

以下示例显示了如何使用staticField属性公开静态字段：

```
<bean id="myField"
        class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean">
    <property name="staticField" value="java.sql.Connection.TRANSACTION_SERIALIZABLE"/>
</bean>
```

更多教程请访问:www.flydean.com

还有一个便捷使用形式，其中静态字段被指定为bean名称，如以下示例所示：

```
<bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE"
        class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean"/>
```

这确实意味着bean id不再有任何选择（因此引用它的任何其他bean也必须使用这个更长的名称），但这个表单定义非常简洁，非常方便用作内部 bean，因为不必为bean引用指定id，如下例所示：

```
<bean id="..." class="...">
    <property name="isolation">
        <bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE"
                class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean" />
    </property>
</bean>
```

你还可以访问另一个bean的非静态（实例）字段，如FieldRetrievingFactoryBean类的API文档中所述。

在Spring中，将枚举值作为属性或构造函数参数注入bean中很容易。 实际上你不需要做任何事情或了解Spring内部的任何内容（甚至是关于诸如FieldRetrievingFactoryBean之类的类）。 以下示例枚举显示了注入枚举值的容易程度：

```
package javax.persistence;

public enum PersistenceContextType {

    TRANSACTION,
    EXTENDED
}
```

考虑如下PersistenceContextType的set方法和相应的bean定义：

```
package example;

public class Client {

    private PersistenceContextType persistenceContextType;

    public void setPersistenceContextType(PersistenceContextType type) {
        this.persistenceContextType = type;
    }
}
```

```
<bean class="example.Client">
    <property name="persistenceContextType" value="TRANSACTION"/>
</bean>
```

使用Using <util:property-path/>

考虑如下例子：

```
<!-- target bean to be referenced by name -->
<bean id="testBean" class="org.springframework.beans.TestBean" scope="prototype">
    <property name="age" value="10"/>
    <property name="spouse">
        <bean class="org.springframework.beans.TestBean">
            <property name="age" value="11"/>
        </bean>
    </property>
</bean>
```

更多教程请访问:www.flydean.com

```
<!-- results in 10, which is the value of property 'age' of bean 'testBean' -->
<bean id="testBean.age" class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>
```

上述配置使用Spring FactoryBean实现（PropertyPathFactoryBean）创建名为testBean.age的bean（类型为int），其值等于testBean bean的age属性。

现在考虑以下示例，它添加了一个<util：property-path />元素：

```
<!-- target bean to be referenced by name -->
<bean id="testBean" class="org.springframework.beans.TestBean" scope="prototype">
    <property name="age" value="10"/>
    <property name="spouse">
        <bean class="org.springframework.beans.TestBean">
            <property name="age" value="11"/>
        </bean>
    </property>
</bean>

<!-- results in 10, which is the value of property 'age' of bean 'testBean' -->
<util:property-path id="name" path="testBean.age"/>
```

<property-path />元素的path属性的值遵循beanName.beanProperty的形式。 在这种情况下，它会获取名为testBean的bean的age属性。 该age属性的值是10。

使用<util:property-path/>设置Bean属性或构造函数值

PropertyPathFactoryBean是一个FactoryBean，用于计算给定目标对象的属性路径。 目标对象可以直接指定，也可以通过bean名称指定。 然后，你可以在另一个bean定义中将此值用作属性值或构造函数参数。

以下示例按名称显示了针对另一个bean使用的路径：

```
// target bean to be referenced by name
<bean id="person" class="org.springframework.beans.TestBean" scope="prototype">
    <property name="age" value="10"/>
    <property name="spouse">
        <bean class="org.springframework.beans.TestBean">
            <property name="age" value="11"/>
        </bean>
    </property>
</bean>

// results in 11, which is the value of property 'spouse.age' of bean 'person'
<bean id="theAge"
        class="org.springframework.beans.factory.config.PropertyPathFactoryBean">
    <property name="targetBeanName" value="person"/>
    <property name="propertyPath" value="spouse.age"/>
</bean>
```

下面的例子中，path被用作内部bean：

```
<!-- results in 12, which is the value of property 'age' of the inner bean -->
<bean id="theAge"
        class="org.springframework.beans.factory.config.PropertyPathFactoryBean">
    <property name="targetObject">
        <bean class="org.springframework.beans.TestBean">
            <property name="age" value="12"/>
        </bean>
    </property>
    <property name="propertyPath" value="age"/>
</bean>
```

更多教程请访问:www.flydean.com

还有一个快捷方式表单，其中bean名称是属性路径。 以下示例显示了快捷方式：

```
<!-- results in 10, which is the value of property 'age' of bean 'person' -->
<bean id="person.age"
        class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>
```

这个表单确实意味着bean的名称没有选择。 对它的任何引用也必须使用相同的id，即路径。 如果用作内部bean，则根本不需要引用它，如下例所示：

```
<bean id="..." class="...">
    <property name="age">
        <bean id="person.age"
                class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>
    </property>
</bean>
```

你可以在实际定义中专门设置结果类型。 对于大多数用例来说，这不是必需的，但它有时可能很有用。 有关此功能的更多信息，请参阅javadoc。

使用<util:properties/>

考虑下面的例子：

```
<!-- creates a java.util.Properties instance with values loaded from the supplied location -->
<bean id="jdbcConfiguration" class="org.springframework.beans.factory.config.PropertiesFactoryBean">
    <property name="location" value="classpath:com/foo/jdbc-production.properties"/>
</bean>
```

上面的配置使用了FactoryBean实现（PropertiesFactoryBean）实例化了java.util.Properties，并从一个特定的Resource地址加载值。

以下示例使用util：properties元素来进行更简洁的表示：

```
<!-- creates a java.util.Properties instance with values loaded from the supplied location -->
<util:properties id="jdbcConfiguration" location="classpath:com/foo/jdbc-production.properties"/>
```

使用<util:list/>

考虑下面的例子：

```
<!-- creates a java.util.List instance with values loaded from the supplied 'sourceList' -->
<bean id="emails" class="org.springframework.beans.factory.config.ListFactoryBean">
    <property name="sourceList">
        <list>
            <value>pechorin@hero.org</value>
            <value>raskolnikov@slums.org</value>
            <value>stavrogin@gov.org</value>
            <value>porfiry@gov.org</value>
        </list>
    </property>
</bean>
```

前面的配置使用Spring FactoryBean实现（ListFactoryBean）来创建java.util.List实例，并使用从提供的sourceList获取的值对其进行初始化。

以下示例使用<util：list/>元素进行更简洁的表示：

```
<!-- creates a java.util.List instance with the supplied values -->
<util:list id="emails">
```

更多教程请访问:www.flydean.com

```
    <value>pechorin@hero.org</value>
    <value>raskolnikov@slums.org</value>
    <value>stavrogin@gov.org</value>
    <value>porfiry@gov.org</value>
</util:list>
```

你还可以使用<util：list />元素上的list-class属性显式控制实例化和填充的List的确切类型。 例如，如果我们确实需要实例化java.util.LinkedList，我们可以使用以下配置：

```
<util:list id="emails" list-class="java.util.LinkedList">
    <value>jackshaftoe@vagabond.org</value>
    <value>eliza@thinkingmanscrumpet.org</value>
    <value>vanhoek@pirate.org</value>
    <value>d'Arcachon@nemesis.org</value>
</util:list>
```

如果没有 list-class ，容器会自己选择list实现。

使用<util:map/>

考虑下面的例子：

```
<!-- creates a java.util.Map instance with values loaded from the supplied 'sourceMap' -->
<bean id="emails" class="org.springframework.beans.factory.config.MapFactoryBean">
    <property name="sourceMap">
        <map>
            <entry key="pechorin" value="pechorin@hero.org"/>
            <entry key="raskolnikov" value="raskolnikov@slums.org"/>
            <entry key="stavrogin" value="stavrogin@gov.org"/>
            <entry key="porfiry" value="porfiry@gov.org"/>
        </map>
    </property>
</bean>
```

上述配置使用Spring FactoryBean实现（MapFactoryBean）创建一个java.util.Map实例，该实例使用从提供的“sourceMap”获取的键值对进行初始化。

以下示例使用<util：map />元素进行更简洁的表示：

```
<!-- creates a java.util.Map instance with the supplied key-value pairs -->
<util:map id="emails">
    <entry key="pechorin" value="pechorin@hero.org"/>
    <entry key="raskolnikov" value="raskolnikov@slums.org"/>
    <entry key="stavrogin" value="stavrogin@gov.org"/>
    <entry key="porfiry" value="porfiry@gov.org"/>
</util:map>
```

你还可以使用<util：map />元素上的'map-class'属性显式控制实例化和填充的Map的确切类型。 例如，如果我们确实需要实例化java.util.TreeMap，我们可以使用以下配置：

```
<util:map id="emails" map-class="java.util.TreeMap">
    <entry key="pechorin" value="pechorin@hero.org"/>
    <entry key="raskolnikov" value="raskolnikov@slums.org"/>
    <entry key="stavrogin" value="stavrogin@gov.org"/>
    <entry key="porfiry" value="porfiry@gov.org"/>
</util:map>
```

如果未提供“map-class”属性，则容器将选择Map实现。

使用<util:set/>

更多教程请访问:www.flydean.com

考虑下面的例子：

```
<!-- creates a java.util.Set instance with values loaded from the supplied 'sourceSet' -->
<bean id="emails" class="org.springframework.beans.factory.config.SetFactoryBean">
    <property name="sourceSet">
        <set>
            <value>pechorin@hero.org</value>
            <value>raskolnikov@slums.org</value>
            <value>stavrogin@gov.org</value>
            <value>porfiry@gov.org</value>
        </set>
    </property>
</bean>
```

上述配置使用Spring FactoryBean实现（SetFactoryBean）创建一个java.util.Set实例，该实例使用从提供的sourceSet获取的值进行初始化。

以下示例使用<util:set/>元素进行更简洁的表示：

```
<!-- creates a java.util.Set instance with the supplied values -->
<util:set id="emails">
    <value>pechorin@hero.org</value>
    <value>raskolnikov@slums.org</value>
    <value>stavrogin@gov.org</value>
    <value>porfiry@gov.org</value>
</util:set>
```

你还可以使用<util：set/>元素上的set-class属性显式控制实例化和填充的Set的确切类型。 例如，如果我们确实需要实例化java.util.TreeSet，我们可以使用以下配置：

```
<util:set id="emails" set-class="java.util.TreeSet">
    <value>pechorin@hero.org</value>
    <value>raskolnikov@slums.org</value>
    <value>stavrogin@gov.org</value>
    <value>porfiry@gov.org</value>
</util:set>
```

如果未提供set-class属性，则容器将选择Set实现。

9.1.2 aop Schema

aop标签用于配置Spring中的所有AOP，包括Spring自己的基于代理的AOP框架和Spring与AspectJ AOP框架的集成。这些标签在题为“面向方面编程与Spring”的章节中进行了全面介绍。

为了完整性，要在aop模式中使用标记，你需要在Spring XML配置文件的顶部包含以下前导码（片段中的文本引用正确的模式，以便aop命名空间中的标记可以被你使用）：

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

    <!-- bean definitions here -->
</beans>
```

更多教程请访问:www.flydean.com

9.1.3 context Schema

context标记处理与管道相关的ApplicationContext配置 - 也就是说，通常不是对最终用户很重要的bean，而是在Spring中执行大量“grunt”工作的bean，例如BeanfactoryPostProcessors。 以下代码段引用了正确的架构，以便你可以使用上下文命名空间中的元素：

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

    <!-- bean definitions here -->
</beans>
```

使用<property-placeholder/>

此元素激活$ {...}占位符的替换，这些占位符是针对指定的属性文件（作为Spring资源位置）解析的。 此元素是一种便捷机制，可为你设置PropertyPlaceholderConfigurer。 如果你需要更多地控制PropertyPlaceholderConfigurer，你可以自己明确定义一个。

使用<annotation-config/>

此元素激活Spring基础结构以检测bean类中的注解：

- Spring的@Configuration模型
- @Autowired / @Inject和@Value
- JSR-250的@Resource，@PostConstruct和@PreDestroy（如果有的话）
- JPA的@PersistenceContext和@PersistenceUnit（如果可用）
- Spring的@EventListener

或者，你可以选择显式激活这些注解的各个BeanPostProcessors。

> 此元素不会激活Spring的@Transactional注解的处理; 你可以使用<tx：annotation-driven />元素来实现此目的。同样，Spring的缓存注解也需要明确启用。

使用<component-scan/>

详细信息请参考：annotation-based container configuration.

使用<load-time-weaver/>

详细信息请参考：load-time weaving with AspectJ in the Spring Framework.

使用<spring-configured/>

详细信息请参考：using AspectJ to dependency inject domain objects with Spring.

使用<mbean-export/>

详细信息请参考： configuring annotation-based MBean export.

9.1.4 Beans Schema

更多教程请访问:www.flydean.com

最后但并非最不重要的是，我们在beans schema中有元素。自框架诞生以来，这些元素一直存在于Spring中。 这里没有显示bean schema中各种元素的示例，因为它们在依赖性和配置中得到了相当全面的介绍（实际上，在整个章节中）。

请注意，你可以向<bean/> XML定义添加零个或多个键值对。 如果有的话，使用这些额外的元数据完成的工作完全取决于你自己的自定义逻辑（因此，如果你按照标题为XML Schema Authoring的附录中所述编写自己的自定义元素，通常只会使用它。）

以下示例显示了周围<bean />上下文中的<meta />元素（请注意，没有任何逻辑可以解释它，元数据实际上是无用的）。

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

    <bean id="foo" class="x.y.Foo">
        <meta key="cacheName" value="foo"/>
        <property name="name" value="Rick"/>
    </bean>

</beans>
```

在前面的示例中，你可以假设有一些逻辑使用bean定义并设置一些使用提供的元数据的缓存基础结构。

更多教程请访问:www.flydean.com

9.2创建XML Schemas

从版本2.0开始，Spring提供了一种机制，用于为基本的Spring XML格式添加基于模式的扩展，以定义和配置bean。本节介绍如何编写自己的自定义XML bean定义解析器并将这些解析器集成到Spring IoC容器中。

为了便于创作使用模式感知XML编辑器的配置文件，Spring的可扩展XML配置机制基于XML Schema。如果你不熟悉Spring标准Spring发行版附带的当前XML配置扩展，则应首先阅读标题为[xsd-config]的附录。

要创建新的XML配置扩展：

1. 编写XML模式来描述你的自定义元素。
2. 编写自定义NamespaceHandler实现。
3. 编写一个或多个BeanDefinitionParser实现（这是完成实际工作的地方）。
4. 使用Spring注册新工件。

为了统一示例，我们创建了一个XML扩展（一个自定义XML元素），它允许我们配置SimpleDateFormat类型的对象（来自java.text包）。完成后，我们将能够定义SimpleDateFormat类型的bean定义，如下所示：

```
<myns:dateformat id="dateFormat"
    pattern="yyyy-MM-dd HH:mm"
    lenient="true"/>
```

（我们在本附录后面会包含更详细的示例。第一个简单示例的目的是引导你完成制作自定义扩展的基本步骤。）

9.2.1 编写 Schema

创建用于Spring的IoC容器的XML配置扩展，首先要创建一个XML Schema来描述扩展。 对于我们的示例，我们使用以下模式来配置SimpleDateFormat对象：

```
<!-- myns.xsd (inside package org/springframework/samples/xml) -->

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.mycompany.com/schema/myns"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:beans="http://www.springframework.org/schema/beans"
        targetNamespace="http://www.mycompany.com/schema/myns"
        elementFormDefault="qualified"
        attributeFormDefault="unqualified">

    <xsd:import namespace="http://www.springframework.org/schema/beans"/>

    <xsd:element name="dateformat">
        <xsd:complexType>
            <xsd:complexContent>
                <xsd:extension base="beans:identifiedType">
                    <xsd:attribute name="lenient" type="xsd:boolean"/>
                    <xsd:attribute name="pattern" type="xsd:string" use="required"/>
                </xsd:extension>
            </xsd:complexContent>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

前面的模式允许我们使用<myns：dateformat />元素直接在XML应用程序上下文文件中配置SimpleDateFormat对象，如以下示例所示：

更多教程请访问:www.flydean.com

```
<myns:dateformat id="dateFormat"
    pattern="yyyy-MM-dd HH:mm"
    lenient="true"/>
```

请注意，在创建基础结构类之后，前面的XML代码段与以下XML代码段基本相同：

```
<bean id="dateFormat" class="java.text.SimpleDateFormat">
    <constructor-arg value="yyyy-HH-dd HH:mm"/>
    <property name="lenient" value="true"/>
</bean>
```

前两个片段中的第二个在容器中创建一个bean（由名称为SimpleDateFormat类型的dateFormat标识），并设置了几个属性。

> 基于schema的创建配置格式的方法允许与具有schema感知XML编辑器的IDE紧密集成。 通过使用正确创作的架构，你可以使用自动完成功能让用户在枚举中定义的几个配置选项之间进行选择。

9.2.2 编写NamespaceHandler

除了schema之外，我们还需要一个NamespaceHandler来解析Spring在解析配置文件时遇到的这个特定命名空间的所有元素。对于此示例，NamespaceHandler应该负责解析myns：dateformat元素。

NamespaceHandler接口有三个方法：

- init（）：允许初始化NamespaceHandler，并在使用处理程序之前由Spring调用。
- BeanDefinition parse（Element，ParserContext）：当Spring遇到顶级元素（未嵌套在bean定义或不同的命名空间内）时调用。此方法本身可以注册bean定义，返回bean定义或两者同时。
- BeanDefinitionHolder decorate（Node，BeanDefinitionHolder，ParserContext）：当Spring遇到不同命名空间的属性或嵌套元素时调用。一个或多个bean定义的装饰（例如）与Spring支持的范围一起使用。我们首先展示一个简单的例子，不使用decoration，之后我们在一个更高级的例子中展示decoration。

虽然你可以为整个命名空间编写自己的NamespaceHandler（因此提供解析命名空间中每个元素的代码），但通常情况是，Spring XML配置文件中的每个顶级XML元素都会生成一个bean定义（在我们的例子中，单个<myns：dateformat />元素导致单个SimpleDateFormat bean定义）。 Spring提供了许多支持此场景的便捷类。在以下示例中，我们使用NamespaceHandlerSupport类：

```
package org.springframework.samples.xml;

import org.springframework.beans.factory.xml.NamespaceHandlerSupport;

public class MyNamespaceHandler extends NamespaceHandlerSupport {

    public void init() {
        registerBeanDefinitionParser("dateformat", new SimpleDateFormatBeanDefinitionParser());
    }

}
```

你可能会注意到这个类中实际上并没有很多解析逻辑。 实际上，NamespaceHandlerSupport类具有内置的委托概念。它支持注册任何数量的BeanDefinitionParser实例，当它需要解析其命名空间中的元素时，它会委托给它们。 这种干净的关注分离让NamespaceHandler处理其命名空间中所有自定义元素的解析编排，同时委托BeanDefinitionParsers执行XML解析的繁琐工作。 这意味着每个BeanDefinitionParser仅包含用于解析单个自定义元素的逻辑，我们可以在下一步中看到。

更多教程请访问:www.flydean.com

9.2.3 使用BeanDefinitionParser

如果NamespaceHandler遇到已映射到特定bean定义解析器的类型的XML元素（在本例中为dateformat），则使用BeanDefinitionParser。 换句话说，BeanDefinitionParser负责解析模式中定义的一个不同的顶级XML元素。 在解析器中，我们可以访问XML元素（以及它的子元素），以便我们可以解析自定义XML内容，如下例所示：

```
package org.springframework.samples.xml;

import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.xml.AbstractSingleBeanDefinitionParser;
import org.springframework.util.StringUtils;
import org.w3c.dom.Element;

import java.text.SimpleDateFormat;

public class SimpleDateFormatBeanDefinitionParser extends AbstractSingleBeanDefinitionParser {

    protected Class getBeanClass(Element element) {
        return SimpleDateFormat.class;
    }

    protected void doParse(Element element, BeanDefinitionBuilder bean) {
        // this will never be null since the schema explicitly requires that a value be supplied
        String pattern = element.getAttribute("pattern");
        bean.addConstructorArg(pattern);

        // this however is an optional property
        String lenient = element.getAttribute("lenient");
        if (StringUtils.hasText(lenient)) {
            bean.addPropertyValue("lenient", Boolean.valueOf(lenient));
        }
    }

}
```

在这个简单的例子中，这就是我们需要做的一切。 我们的单个BeanDefinition的创建由AbstractSingleBeanDefinitionParser超类处理，bean定义的唯一标识符的提取和设置也是如此。

9.2.4 注册Handler和Schema

编码完成。剩下要做的就是让Spring XML解析基础架构了解我们的自定义元素。我们通过在两个专用属性文件中注册我们的自定义namespaceHandler和自定义XSD文件来实现。这些属性文件都放在应用程序的META-INF目录中，例如，可以与JAR文件中的二进制类一起分发。 Spring XML解析基础结构通过使用这些特殊属性文件自动获取新扩展，其格式将在接下来的两节中详细介绍。

编写META-INF/spring.handlers

名为spring.handlers的属性文件包含XML Schema URI到命名空间处理程序类的映射。对于我们的示例，我们需要编写以下内容：

```
http\://www.mycompany.com/schema/myns=org.springframework.samples.xml.MyNamespaceHandler
```

（：字符是Java属性格式的有效分隔符，因此：URI中的字符需要使用反斜杠进行转义。）

键值对的第一部分（键）是与自定义命名空间扩展关联的URI，需要与自定义XSD架构中指定的targetNamespace属性的值完全匹配。

写'META-INF/spring.schemas'

更多教程请访问:www.flydean.com

名为spring.schemas的属性文件包含XML Schema位置的映射（在模式声明中，在使用该模式作为xsi：schemaLocation属性的一部分的XML文件中引用）到类路径资源。需要此文件来防止Spring绝对必须使用需要Internet访问权限的默认EntityResolver来检索模式文件。如果在此属性文件中指定映射，Spring将在类路径中搜索模式（在本例中为org.springframework.samples.xml包中的myns.xsd）。以下代码段显示了我们需要为自定义架构添加的行：

```
HTTP\://www.mycompany.com/schema/myns/myns.xsd=org/springframework/samples/xml/myns.xsd
```

（请记住：必须转义：字符。）

建议你在类路径上与NamespaceHandler和BeanDefinitionParser类一起部署XSD文件（或多个文件）。

9.2.5 使用Spring XML Configuration的自定义扩展

使用自己实现的自定义扩展与使用Spring提供的“自定义”扩展之一没有什么不同。 以下示例使用Spring XML配置文件中前面步骤中开发的自定义<dateformat />元素：

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

    <!-- as a top-level bean -->
    <myns:dateformat id="defaultDateFormat" pattern="yyyy-MM-dd HH:mm" lenient="true"/>

    <bean id="jobDetailTemplate" abstract="true">
        <property name="dateFormat">
            <!-- as an inner bean -->
            <myns:dateformat pattern="HH:mm MM-dd-yyyy"/>
        </property>
    </bean>

</beans>
```

9.2.6 更多例子

本节介绍自定义XML扩展的一些更详细的示例。

嵌套的自定义元素

本节中提供的示例显示了如何编写满足以下配置目标所需的各种工件：

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

    <foo:component id="bionic-family" name="Bionic-1">
        <foo:component name="Mother-1">
            <foo:component name="Karate-1"/>
            <foo:component name="Sport-1"/>
        </foo:component>
```

更多教程请访问:www.flydean.com

```
        <foo:component name="Rock-1"/>
    </foo:component>

</beans>
```

上述配置将自定义扩展嵌套在彼此之内。 实际由<foo：component />元素配置的类是Component类（在下一个示例中显示）。 请注意Component类如何不公开components属性的setter方法。 这使得通过使用setter注入为Component类配置bean定义很困难（或者更不可能）。 以下清单显示了Component类：

```
package com.foo;

import java.util.ArrayList;
import java.util.List;

public class Component {

    private String name;
    private List<Component> components = new ArrayList<Component> ();

    // mmm, there is no setter method for the 'components'
    public void addComponent(Component component) {
        this.components.add(component);
    }

    public List<Component> getComponents() {
        return components;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

此问题的典型解决方案是创建一个自定义FactoryBean，公开components属性的setter属性。 以下清单显示了这样的自定义FactoryBean：

```
package com.foo;

import org.springframework.beans.factory.FactoryBean;

import java.util.List;

public class ComponentFactoryBean implements FactoryBean<Component> {

    private Component parent;
    private List<Component> children;

    public void setParent(Component parent) {
        this.parent = parent;
    }

    public void setChildren(List<Component> children) {
        this.children = children;
    }

    public Component getObject() throws Exception {
        if (this.children != null && this.children.size() > 0) {
            for (Component child : children) {
                this.parent.addComponent(child);
            }
        }
```

更多教程请访问:www.flydean.com

```
        return this.parent;
    }

    public Class<Component> getObjectType() {
        return Component.class;
    }

    public boolean isSingleton() {
        return true;
    }
}
```

这很好用，但它向最终用户公开了很多Spring管道。 我们要做的是编写一个隐藏所有Spring管道的自定义扩展。 如果我们坚持前面描述的步骤，我们首先创建XSD架构来定义自定义标签的结构，如下面的清单所示：

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<xsd:schema xmlns="http://www.foo.com/schema/component"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.foo.com/schema/component"
        elementFormDefault="qualified"
        attributeFormDefault="unqualified">

    <xsd:element name="component">
        <xsd:complexType>
            <xsd:choice minOccurs="0" maxOccurs="unbounded">
                <xsd:element ref="component"/>
            </xsd:choice>
            <xsd:attribute name="id" type="xsd:ID"/>
            <xsd:attribute name="name" use="required" type="xsd:string"/>
        </xsd:complexType>
    </xsd:element>

</xsd:schema>
```

再次按照前面描述的过程，我们再创建一个自定义NamespaceHandler：

```
package com.foo;

import org.springframework.beans.factory.xml.NamespaceHandlerSupport;

public class ComponentNamespaceHandler extends NamespaceHandlerSupport {

    public void init() {
        registerBeanDefinitionParser("component", new ComponentBeanDefinitionParser());
    }
}
```

接下来是自定义BeanDefinitionParser。 请记住，我们正在创建一个描述ComponentFactoryBean的BeanDefinition。 以下清单显示了我们的自定义BeanDefinitionParser实现：

```
package com.foo;

import org.springframework.beans.factory.config.BeanDefinition;
import org.springframework.beans.factory.support.AbstractBeanDefinition;
import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.support.ManagedList;
import org.springframework.beans.factory.xml.AbstractBeanDefinitionParser;
import org.springframework.beans.factory.xml.ParserContext;
import org.springframework.util.xml.DomUtils;
import org.w3c.dom.Element;

import java.util.List;
```

更多教程请访问:www.flydean.com

```
public class ComponentBeanDefinitionParser extends AbstractBeanDefinitionParser {

    protected AbstractBeanDefinition parseInternal(Element element, ParserContext parserContext) {
        return parseComponentElement(element);
    }

    private static AbstractBeanDefinition parseComponentElement(Element element) {
        BeanDefinitionBuilder factory = BeanDefinitionBuilder.rootBeanDefinition(ComponentFactoryBean.class);
        factory.addPropertyValue("parent", parseComponent(element));

        List<Element> childElements = DomUtils.getChildElementsByTagName(element, "component");
        if (childElements != null && childElements.size() > 0) {
            parseChildComponents(childElements, factory);
        }

        return factory.getBeanDefinition();
    }

    private static BeanDefinition parseComponent(Element element) {
        BeanDefinitionBuilder component = BeanDefinitionBuilder.rootBeanDefinition(Component.class);
        component.addPropertyValue("name", element.getAttribute("name"));
        return component.getBeanDefinition();
    }

    private static void parseChildComponents(List<Element> childElements, BeanDefinitionBuilder factory) {
        ManagedList<BeanDefinition> children = new ManagedList<BeanDefinition>(childElements.size());
        for (Element element : childElements) {
            children.add(parseComponentElement(element));
        }
        factory.addPropertyValue("children", children);
    }
}
```

最后，需要通过修改META-INF/spring.handlers和META-INF/spring.schemas文件，在Spring XML基础结构中注册各种工件，如下所示：

```
#in 'META-INF/spring.handlers'
http\://www.foo.com/schema/component=com.foo.ComponentNamespaceHandler

# in 'META-INF/spring.schemas'
http\://www.foo.com/schema/component/component.xsd=com/foo/component.xsd
```

自定义Normal Elements的属性

编写自己的自定义解析器和相关工件并不难。 但是，有时这不是正确的做法。 考虑需要向现有bean定义添加元数据的场景。 在这种情况下，你当然不希望编写自己的整个自定义扩展。 相反，你只想在现有bean定义元素中添加其他属性。

通过另一个示例，假设你为服务对象（未知）访问集群JCache定义了bean定义，并且你希望确保在周围集群中急切地启动指定的JCache实例。 以下清单显示了这样一个定义：

```
<bean id="checkingAccountService" class="com.foo.DefaultCheckingAccountService"
        jcache:cache-name="checking.account">
    <!-- other dependencies here... -->
</bean>
```

然后，我们可以在解析'jcache：cache-name'属性时创建另一个BeanDefinition。 然后，此BeanDefinition为我们初始化命名的JCache。 我们还可以修改'checkingAccountService'的现有BeanDefinition，以便它依赖于这个新的JCache初始化BeanDefinition。 以下清单显示了我们的JCacheInitializer：

```
package com.foo;
```

更多教程请访问:www.flydean.com

```
public class JCacheInitializer {

    private String name;

    public JCacheInitializer(String name) {
        this.name = name;
    }

    public void initialize() {
        // lots of JCache API calls to initialize the named cache...
    }
}
```

现在我们可以转到自定义扩展。 首先，我们需要编写描述自定义属性的XSD架构，如下所示：

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<xsd:schema xmlns="http://www.foo.com/schema/jcache"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.foo.com/schema/jcache"
        elementFormDefault="qualified">

    <xsd:attribute name="cache-name" type="xsd:string"/>

</xsd:schema>
```

接下来，我们需要创建关联的NamespaceHandler，如下所示：

```
package com.foo;

import org.springframework.beans.factory.xml.NamespaceHandlerSupport;

public class JCacheNamespaceHandler extends NamespaceHandlerSupport {

    public void init() {
        super.registerBeanDefinitionDecoratorForAttribute("cache-name",
            new JCacheInitializingBeanDefinitionDecorator());
    }

}
```

接下来，我们需要创建解析器。 请注意，在这种情况下，因为我们要解析XML属性，所以我们编写BeanDefinitionDecorator而不是BeanDefinitionParser。 以下清单显示了我们的BeanDefinitionDecorator实现：

```
package com.foo;

import org.springframework.beans.factory.config.BeanDefinitionHolder;
import org.springframework.beans.factory.support.AbstractBeanDefinition;
import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.xml.BeanDefinitionDecorator;
import org.springframework.beans.factory.xml.ParserContext;
import org.w3c.dom.Attr;
import org.w3c.dom.Node;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class JCacheInitializingBeanDefinitionDecorator implements BeanDefinitionDecorator {

    private static final String[] EMPTY_STRING_ARRAY = new String[0];

    public BeanDefinitionHolder decorate(Node source, BeanDefinitionHolder holder,
            ParserContext ctx) {
```

更多教程请访问:www.flydean.com

```
        String initializerBeanName = registerJCacheInitializer(source, ctx);
        createDependencyOnJCacheInitializer(holder, initializerBeanName);
        return holder;
    }

    private void createDependencyOnJCacheInitializer(BeanDefinitionHolder holder,
            String initializerBeanName) {
        AbstractBeanDefinition definition = ((AbstractBeanDefinition) holder.getBeanDefinition());
        String[] dependsOn = definition.getDependsOn();
        if (dependsOn == null) {
            dependsOn = new String[]{initializerBeanName};
        } else {
            List dependencies = new ArrayList(Arrays.asList(dependsOn));
            dependencies.add(initializerBeanName);
            dependsOn = (String[]) dependencies.toArray(EMPTY_STRING_ARRAY);
        }
        definition.setDependsOn(dependsOn);
    }

    private String registerJCacheInitializer(Node source, ParserContext ctx) {
        String cacheName = ((Attr) source).getValue();
        String beanName = cacheName + "-initializer";
        if (!ctx.getRegistry().containsBeanDefinition(beanName)) {
            BeanDefinitionBuilder initializer = BeanDefinitionBuilder.rootBeanDefinition(JCacheInitializer.clas
s);
            initializer.addConstructorArg(cacheName);
            ctx.getRegistry().registerBeanDefinition(beanName, initializer.getBeanDefinition());
        }
        return beanName;
    }
}
```

最后，我们需要通过修改META-INF/spring.handlers和META-INF /spring.schemas文件来注册Spring XML基础结构中的各种工件，如下所示：

```
# in 'META-INF/spring.handlers'
http\://www.foo.com/schema/jcache=com.foo.JCacheNamespaceHandler

# in 'META-INF/spring.schemas'
http\://www.foo.com/schema/jcache/jcache.xsd=com/foo/jcache.xsd
```

更多教程请访问:www.flydean.com

测试

本章介绍Spring对集成测试的支持和单元测试的最佳实践。 Spring团队提倡测试驱动开发（TDD）。 Spring团队发现正确使用控制反转（IoC）肯定会使单元测试和集成测试更容易（因为类的setter方法和适当的构造函数的存在使得它们更容易在测试中连接在一起而不必设置服务定位器注册表和类似结构）。

更多教程请访问:www.flydean.com

1.Spring测试介绍

测试是企业软件开发不可或缺的一部分。 本章重点介绍IoC原理对单元测试的附加值以及Spring Framework对集成测试支持的好处。 （对企业中的测试进行全面处理超出了本参考手册的范围。）

更多教程请访问:www.flydean.com

2.单元测试

与传统的Java EE开发相比，依赖注入应该使你的代码对容器的依赖性降低。构成应用程序的POJO应该在JUnit或TestNG测试中可测试，对象使用new运算符实例化，不使用Spring或任何其他容器。你可以使用模拟对象（与其他有价值的测试技术结合使用）来单独测试代码。如果你遵循Spring的体系结构建议，那么代码库的干净分层和组件化便于单元测试。例如，你可以通过存根或模拟DAO或存储库接口来测试服务层对象，而无需在运行单元测试时访问持久数据。

真正的单元测试通常运行得非常快，因为没有要设置的运行时基础结构。强调真正的单元测试作为开发方法的一部分可以提高你的工作效率。你可能不需要测试章节的这一部分来帮助你为基于IoC的应用程序编写有效的单元测试。但是，对于某些单元测试场景，Spring Framework提供了模拟对象和测试支持类，本章将对其进行介绍。

更多教程请访问:www.flydean.com

2.1Mock Objects

Spring包含许多专门用于模拟的包：

- Environment
- JNDI
- Servlet API
- Spring Web Reactive

2.1.1 Environment

org.springframework.mock.env包中包含Environment和PropertySource抽象的模拟实现（请参阅Bean定义配置文件和PropertySource抽象）。 MockEnvironment和MockPropertySource对于开发依赖于特定于环境的属性的代码的容器外测试非常有用。

2.1.2 JNDI

org.springframework.mock.jndi包中包含JNDI SPI的实现，你可以使用它为测试套件或独立应用程序设置简单的JNDI环境。 例如，如果JDBC DataSource实例在测试代码中绑定到与Java EE容器中相同的JNDI名称，则可以在测试方案中重用应用程序代码和配置而无需修改。

2.1.3 Servlet API

org.springframework.mock.web包中包含一组全面的Servlet API模拟对象，可用于测试Web上下文，控制器和过滤器。 这些模拟对象的目标是使用Spring的Web MVC框架，并且通常比动态模拟对象（例如EasyMock）或替代Servlet API模拟对象（例如MockObjects）更方便使用。

> 从Spring Framework 5.0开始，org.springframework.mock.web中的模拟对象基于Servlet 4.0 API。

Spring MVC Test框架以模拟Servlet API对象为基础，为Spring MVC提供集成测试框架。 请参阅Spring MVC测试框架。

2.1.4 Spring Web Reactive

org.springframework.mock.http.server.reactive包中包含用于WebFlux应用程序的ServerHttpRequest和ServerHttpResponse的模拟实现。 org.springframework.mock.web.server包中包含一个依赖于那些模拟请求和响应对象的模拟ServerWebExchange。

MockServerHttpRequest和MockServerHttpResponse都从与服务器特定的实现相同的抽象基类扩展，并与它们共享行为。例如，模拟请求一旦创建就是不可变的，但你可以使用ServerHttpRequest中的mutate（）方法创建修改后的实例。

为了使模拟响应正确地实现写入合同并返回写入完成句柄（即Mono ），它默认使用带有cache().then()的Flux，它缓冲数据并使其成为可能。可用于测试中的断言。应用程序可以设置自定义写入功能（例如，测试无限流）。

WebTestClient以模拟请求和响应为基础，为不使用HTTP服务器测试WebFlux应用程序提供支持。客户端还可以用于正在运行的服务器的端到端测试。

更多教程请访问:www.flydean.com

更多教程请访问:www.flydean.com

2.2单元测试支持类

Spring包含许多可以帮助进行单元测试的类。 它们分为两类：

- 一般测试实用程序
- Spring MVC测试实用程序

2.2.1 一般测试实用程序

org.springframework.test.util包中包含几个用于单元和集成测试的通用实用程序。

ReflectionTestUtils是基于反射的实用程序方法的集合。你可以在测试需要更改常量值，设置非公共字段，调用非公共setter方法或在测试应用程序代码时调用非公共配置或生命周期回调方法的场景中使用这些方法。用例如下：

- 容忍私有或受保护字段访问的ORM框架（例如JPA和Hibernate），而不是域实体中属性的公共setter方法。
- Spring支持注解（例如@Autowired，@Inject和@Resource），它们为私有或受保护字段，setter方法和配置方法提供依赖注入。
- 使用@PostConstruct和@PreDestroy等注解进行生命周期回调方法。

AopTestUtils是AOP相关实用程序方法的集合。你可以使用这些方法获取对隐藏在一个或多个Spring代理后面的基础目标对象的引用。例如，如果你已使用诸如EasyMock或Mockito之类的库将bean配置为动态模拟，并且模拟包装在Spring代理中，则可能需要直接访问底层模拟以配置对它的期望并执行验证。对于Spring的核心AOP实用程序，请参阅AopUtils和AopProxyUtils。

2.2.2 Spring MVC测试实用程序

org.springframework.test.web包中包含ModelAndViewAssert，你可以将其与JUnit，TestNG或任何其他用于处理Spring MVC ModelAndView对象的单元测试的测试框架结合使用。

> 单元测试Spring MVC控制器
>
> 要将Spring MVC Controller类单元测试为POJO，请使用ModelAndViewAssert与Spring的Servlet API模拟中的MockHttpServletRequest，MockHttpSession等结合使用。 有关Spring MVC和REST Controller类的完整集成测试以及Spring MVC的WebApplicationContext配置，请使用Spring MVC Test Framework。

更多教程请访问:www.flydean.com

3.集成测试

本节（本章的大部分内容）介绍了Spring应用程序的集成测试。 它包括以下主题：

- 概括
- 集成测试的目标
- JDBC测试支持
- 注解
- Spring TestContext框架
- Spring MVC测试框架

更多教程请访问:www.flydean.com

3.1概览

能够执行某些集成测试非常重要，无需部署到应用程序服务器或连接到其他企业基础结构。这样做可以让你测试以下内容：

- Spring IoC容器上下文的正确连接。
- 使用JDBC或ORM工具进行数据访问。这可以包括诸如SQL语句的正确性，Hibernate查询，JPA实体映射等等。

Spring Framework为spring-test模块中的集成测试提供了一流的支持。实际JAR文件的名称可能包含发行版本，也可能是长的org.springframework.test形式，具体取决于你从何处获取（有关说明，请参阅依赖关系管理部分）。该库包含org.springframework.test包，其中包含用于使用Spring容器进行集成测试的有价值的类。此测试不依赖于应用程序服务器或其他部署环境。这些测试比单元测试运行得慢，但比依赖部署到应用服务器的等效Selenium测试或远程测试快得多。

单元和集成测试支持以注解驱动的Spring TestContext Framework的形式提供。 TestContext框架与使用中的实际测试框架无关，它允许在各种环境中检测测试，包括JUnit，TestNG等。

更多教程请访问:www.flydean.com

3.2集成测试的目的

Spring的集成测试支持有以下主要目标：

- 在测试之间管理Spring IoC容器缓存。
- 提供测试装置实例的依赖注入。
- 提供适合集成测试的事务管理。
- 提供特定于Spring的基类，帮助开发人员编写集成测试。

接下来的几节将介绍每个目标，并提供实现和配置详细信息的链接。

3.2.1 上下文管理和缓存

Spring TestContext Framework提供Spring ApplicationContext实例和WebApplicationContext实例的一致加载以及这些上下文的缓存。支持缓存加载的上下文非常重要，因为启动时间可能会成为一个问题 - 不是因为Spring本身的开销，而是因为Spring容器实例化的对象需要时间来实例化。例如，具有50到100个Hibernate映射文件的项目可能需要10到20秒来加载映射文件，并且在每个测试装置中运行每个测试之前产生该成本会导致整体测试运行速度变慢，从而降低开发人员的工作效率。

测试类通常声明XML或Groovy配置元数据的资源位置数组（通常在类路径中）或用于配置应用程序的带注解类的数组。这些位置或类与web.xml或生产部署的其他配置文件中指定的位置或类相同或类似。

默认情况下，一旦加载，配置的ApplicationContext将重复用于每个测试。因此，每个测试套件仅产生一次设置成本，并且后续测试执行要快得多。在此上下文中，术语“测试套件”表示所有测试都在同一JVM中运行 - 例如，所有测试都是针对给定项目或模块从Ant，Maven或Gradle构建的。在不太可能的情况下，测试会破坏应用程序上下文并需要重新加载（例如，通过修改bean定义或应用程序对象的状态），可以将TestContext框架配置为在执行下一个之前重新加载配置并重建应用程序上下文测试。

请参阅使用TestContext框架的上下文管理和上下文缓存。

3.2.2 测试装置的依赖注入

当TestContext框架加载应用程序上下文时，它可以选择使用依赖注入来配置测试类的实例。这提供了一种方便的机制，可以通过使用应用程序上下文中的预配置bean来设置测试装置。这里的一个重要好处是你可以在各种测试场景中重用应用程序上下文（例如，用于配置Spring管理的对象图，事务代理，DataSource实例等），从而避免复制单个测试的复杂测试装置设置的需要案例。

作为示例，考虑我们有一个类（HibernateTitleRepository）实现Title域实体的数据访问逻辑的场景。我们想编写测试以下方面的集成测试：

- Spring配置：基本上，与HibernateTitleRepository bean的配置相关的所有内容是否正确和存在?
- Hibernate映射文件配置：是否所有映射都正确并且是否正确的延迟加载设置?
- HibernateTitleRepository的逻辑：此类的已配置实例是否按预期执行?

请参阅使用TestContext框架对测试装置进行依赖注入。

3.2.3 事务管理

访问真实数据库的测试中的一个常见问题是它们对持久性存储的状态的影响。即使你使用开发数据库，对状态的更改也可能会影响将来的测试。此外，许多操作（例如插入或修改持久数据）无法在事务之外执行（或验证）。

更多教程请访问:www.flydean.com

TestContext框架解决了这个问题。默认情况下，框架为每个测试创建并回滚事务。你可以编写可以假定存在事务的代码。如果在测试中调用事务代理对象，则根据其配置的事务语义，它们的行为正确。此外，如果测试方法在为测试管理的事务中运行时删除所选表的内容，则事务默认回滚，并且数据库返回到执行测试之前的状态。通过使用在测试的应用程序上下文中定义的PlatformTransactionManager bean，为测试提供事务支持。

如果你希望事务提交（异常，但在你希望特定测试填充或修改数据库时偶尔有用），你可以通过使用@Commit注解告诉TestContext框架使事务提交而不是回滚。

使用TestContext框架查看事务管理。

3.2.4 集成测试的支持类

Spring TestContext Framework提供了几个抽象支持类，可以简化集成测试的编写。 这些基本测试类提供了定义良好的钩子到测试框架中，以及方便的实例变量和方法，使你可以访问：

- ApplicationContext，用于执行显式bean查找或测试整个上下文的状态。
- JdbcTemplate，用于执行SQL语句以查询数据库。 你可以在执行与数据库相关的应用程序代码之前和之后使用此类查询来确认数据库状态，Spring确保此类查询在与应用程序代码相同的事务范围内运行。 与ORM工具结合使用时，请务必避免误报。

此外，你可能希望使用特定于项目的实例变量和方法创建自己的自定义应用程序范围的超类。

请参阅TestContext框架的支持类。

更多教程请访问:www.flydean.com

3.3JDBC测试支持

org.springframework.test.jdbc包中包含JdbcTestUtils，它是与JDBC相关的实用程序函数的集合，旨在简化标准数据库测试方案。具体来说，JdbcTestUtils提供以下静态实用程序方法。

- countRowsInTable（..）：计算给定表中的行数。
- countRowsInTableWhere（..）：使用提供的WHERE子句计算给定表中的行数。
- deleteFromTables（..）：删除指定表中的所有行。
- deleteFromTableWhere（..）：使用提供的WHERE子句从给定的表中删除行。
- dropTables（..）：删除指定的表。

> AbstractTransactionalJUnit4SpringContextTests和AbstractTransactionalTestNGSpringContextTests提供了方便的方法，可以委托JdbcTestUtils中的上述方法。
>
> spring-jdbc模块支持配置和启动嵌入式数据库，你可以在与数据库交互的集成测试中使用它。有关详细信息，请参阅嵌入式数据库支持和使用嵌入式数据库测试数据访.

更多教程请访问:www.flydean.com

3.4注解

3.4 注解

本节介绍了在测试Spring应用程序时可以使用的注解。 它包括以下主题：

- Spring测试注解
- 标准注解支持
- Spring JUnit 4测试注解
- Spring JUnit Jupiter测试注解
- 测试的元注解支持

3.4.1 Spring测试注解

Spring Framework提供了以下一组特定于Spring的注解，你可以在单元和集成测试中结合TestContext框架使用它们。 有关详细信息，请参阅相应的javadoc，包括默认属性值，属性别名和其他详细信息。

Spring的测试注解包括以下内容：

- @BootstrapWith
- @ContextConfiguration
- @WebAppConfiguration
- @ContextHierarchy
- @ActiveProfiles
- @TestPropertySource
- @DirtiesContext
- @TestExecutionListeners
- @Commit
- @Rollback
- @BeforeTransaction
- @AfterTransaction
- @Sql
- @SqlConfig
- @SqlGroup

@BootstrapWith

@BootstrapWith是一个类级注解，可用于配置Spring TestContext Framework的引导方式。 具体来说，你可以使用@BootstrapWith指定自定义TestContextBootstrapper。 有关更多详细信息，请参阅有关引导TestContext框架的部分。

@ContextConfiguration

@ContextConfiguration定义类级元数据，用于确定如何为集成测试加载和配置ApplicationContext。 具体来说，@ContextConfiguration声明应用程序上下文资源位置或用于加载上下文的带注解的类。

资源位置通常是位于类路径中的XML配置文件或Groovy脚本，而带注解的类通常是@Configuration类。 但是，资源位置也可以引用文件系统中的文件和脚本，带注解的类可以是组件类，依此类推。

以下示例显示了引用XML文件的@ContextConfiguration批注：

```
@ContextConfiguration("/test-config.xml")
public class XmlApplicationContextTests {
    // class body...
}
```

更多教程请访问:www.flydean.com

以下示例显示了引用类的@ContextConfiguration批注：

```
@ContextConfiguration(classes = TestConfig.class)
public class ConfigClassApplicationContextTests {
    // class body...
}
```

作为声明资源位置或带注解的类的替代或补充，你可以使用@ContextConfiguration来声明ApplicationContextInitializer类。 以下示例显示了这种情况：

```
@ContextConfiguration(initializers = CustomContextIntializer.class)
public class ContextInitializerTests {
    // class body...
}
```

你也可以选择使用@ContextConfiguration来声明ContextLoader策略。 但请注意，你通常不需要显式配置加载程序，因为默认加载程序支持初始化程序以及资源位置或带注解的类。

以下示例使用位置和加载器：

```
@ContextConfiguration(locations = "/test-context.xml", loader = CustomContextLoader.class)
public class CustomLoaderXmlApplicationContextTests {
    // class body...
}
```

> @ContextConfiguration支持继承资源位置或配置类以及超类声明的上下文初始化程序。

@WebAppConfiguration

@WebAppConfiguration是一个类级别的注解，可用于声明为集成测试加载的ApplicationContext应该是WebApplicationContext。 仅在测试类上存在@WebAppConfiguration可确保为测试加载WebApplicationContext，使用默认值“file:src/main/webapp”作为Web应用程序根目录的路径（即资源） 基本路径）。 在后台使用资源基本路径来创建MockServletContext，它充当测试的WebApplicationContext的ServletContext。

以下示例显示如何使用@WebAppConfiguration批注：

```
@ContextConfiguration
@WebAppConfiguration
public class WebAppTests {
    // class body...
}
```

要覆盖默认值，可以使用隐式值属性指定不同的基本资源路径。 支持classpath：和file：资源前缀。 如果未提供资源前缀，则假定该路径是文件系统资源。 以下示例显示如何指定类路径资源：

```
@ContextConfiguration
@WebAppConfiguration("classpath:test-web-resources")
public class WebAppTests {
    // class body...
}
```

请注意，@WebAppConfiguration必须与@ContextConfiguration结合使用，可以在单个测试类中，也可以在测试类层次结构中使用。 有关更多详细信息，请参阅@WebAppConfiguration javadoc。

@ContextHierarchy

更多教程请访问:www.flydean.com

@ContextHierarchy是一个类级别注解，用于为集成测试定义ApplicationContext实例的层次结构。 应使用一个或多个@ContextConfiguration实例的列表声明@ContextHierarchy，每个实例定义上下文层次结构中的级别。 以下示例演示了在单个测试类中使用@ContextHierarchy（@ContextHierarchy也可以在测试类层次结构中使用）：

```
@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...
}
```

如果需要合并或覆盖测试类层次结构中上下文层次结构的给定级别的配置，则必须通过在类层次结构中的每个相应级别为@ContextConfiguration中的name属性提供相同的值来显式命名该级别。 有关更多示例，请参阅上下文层次结构和@ContextHierarchy javadoc。

@ActiveProfiles

@ActiveProfiles是一个类级注解，用于在为集成测试加载ApplicationContext时声明哪些bean定义配置文件应该处于活动状态。

以下示例表明dev配置文件应处于活动状态：

```
@ContextConfiguration
@ActiveProfiles("dev")
public class DeveloperTests {
    // class body...
}
```

以下示例表明dev和集成配置文件都应处于活动状态：

```
@ContextConfiguration
@ActiveProfiles({"dev", "integration"})
public class DeveloperIntegrationTests {
    // class body...
}
```

@ActiveProfiles默认支持继承超类声明的活动bean定义配置文件。 你还可以通过实现自定义ActiveProfilesResolver并使用@ActiveProfiles的解析程序属性对其进行注册来以编程方式解析活动Bean定义概要文件。

@TestPropertySource

@TestPropertySource是一个类级别注解，可用于配置属性文件的位置和内联属性，以便为集成测试加载的ApplicationContext添加到环境中的PropertySource集合中。

测试属性源的优先级高于从操作系统环境或Java系统属性加载的属性源，以及应用程序通过@PropertySource或以编程方式声明性地添加的属性源。 因此，测试属性源可用于有选择地覆盖系统和应用程序属性源中定义的属性。 此外，内联属性的优先级高于从资源位置加载的属性。

以下示例演示如何从类路径声明属性文件：

更多教程请访问:www.flydean.com

```
@ContextConfiguration
@TestPropertySource("/test.properties")
public class MyIntegrationTests {
    // class body...
}
```

以下示例演示如何声明内联属性：

```
@ContextConfiguration
@TestPropertySource(properties = { "timezone = GMT", "port: 4242" })
public class MyIntegrationTests {
    // class body...
}
```

@DirtiesContext

@DirtiesContext指示在执行测试期间底层Spring ApplicationContext已被弄脏（即，测试以某种方式修改或损坏它 - 例如，通过更改单例bean的状态）并且应该关闭。当应用程序上下文被标记为脏时，它将从测试框架的缓存中删除并关闭。因此，对于需要具有相同配置元数据的上下文的任何后续测试，都会重建基础Spring容器。

你可以将@DirtiesContext用作同一类或类层次结构中的类级别和方法级别注解。在这种情况下，ApplicationContext在任何此类带注解的方法之前或之后以及当前测试类之前或之后被标记为脏，具体取决于配置的methodMode和classMode。

以下示例说明了各种配置方案的上下文何时会变脏：

- 在当前测试类之前，在类模式设置为BEFORE_CLASS的类上声明。

```
@DirtiesContext(classMode = BEFORE_CLASS)
public class FreshContextTests {
    // some tests that require a new Spring container
}
```

- 在当前测试类之后，在类模式设置为AFTER_CLASS（即默认类模式）的类上声明。

```
@DirtiesContext
public class ContextDirtyingTests {
    // some tests that result in the Spring container being dirtied
}
```

- 在当前测试类中的每个测试方法之前，在类模式设置为BEFORE_EACH_TEST_METHOD的类上声明时。

```
@DirtiesContext(classMode = BEFORE_EACH_TEST_METHOD)
public class FreshContextTests {
    // some tests that require a new Spring container
}
```

- 在当前测试类中的每个测试方法之后，在类模式设置为AFTER_EACH_TEST_METHOD的类上声明。

```
@DirtiesContext(classMode = AFTER_EACH_TEST_METHOD)
public class ContextDirtyingTests {
    // some tests that result in the Spring container being dirtied
}
```

- 在当前测试之前，在方法模式设置为BEFORE_METHOD的方法上声明。

```
@DirtiesContext(methodMode = BEFORE_METHOD)
```

更多教程请访问:www.flydean.com

```
@Test
public void testProcessWhichRequiresFreshAppCtx() {
    // some logic that requires a new Spring container
}
```

- 在当前测试之后，在方法模式设置为AFTER_METHOD（即默认方法模式）的方法上声明。

```
@DirtiesContext
@Test
public void testProcessWhichDirtiesAppCtx() {
    // some logic that results in the Spring container being dirtied
}
```

如果在测试中使用@DirtiesContext，其上下文配置为具有@ContextHierarchy的上下文层次结构的一部分，则可以使用hierarchyMode标志来控制如何清除上下文高速缓存。 默认情况下，使用详尽的算法来清除上下文缓存，不仅包括当前级别，还包括共享当前测试常见的祖先上下文的所有其他上下文层次结构。 驻留在公共祖先上下文的子层次结构中的所有ApplicationContext实例将从上下文缓存中删除并关闭。 如果穷举算法对于特定用例而言过度，则可以指定更简单的当前级别算法，如以下示例所示。

```
@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
    }
}
```

有关EXHAUSTIVE和CURRENT_LEVEL算法的更多详细信息，请参阅DirtiesContext.HierarchyMode javadoc。

@TestExecutionListeners

@TestExecutionListeners定义了类级元数据，用于配置应该使用TestContextManager注册的TestExecutionListener实现。 通常，@TestExecutionListeners与@ContextConfiguration一起使用。

以下示例显示如何注册两个TestExecutionListener实现：

```
@ContextConfiguration
@TestExecutionListeners({CustomTestExecutionListener.class, AnotherTestExecutionListener.class})
public class CustomTestExecutionListenerTests {
    // class body...
}
```

默认情况下，@TestExecutionListeners支持继承的侦听器。 有关示例和更多详细信息，请参阅javadoc。

@Commit

@Commit表示应在测试方法完成后提交事务测试方法的事务。 你可以使用@Commit作为@Rollback（false）的直接替换，以更明确地传达代码的意图。 类似于@Rollback，@Commit也可以声明为类级别或方法级别的注解。

以下示例显示了如何使用@Commit注解：

更多教程请访问:www.flydean.com

```
@Commit
@Test
public void testProcessWithoutRollback() {
    // ...
}
```

@Rollback

@Rollback指示在测试方法完成后是否应回滚事务测试方法的事务。 如果为true，则回滚事务。 否则，提交事务（另请参阅@Commit）。 即使没有显式声明@Rollback，Spring TestContext Framework中的集成测试回滚也默认为true。

当声明为类级别注解时，@Rollback定义测试类层次结构中所有测试方法的默认回滚语义。 当声明为方法级注解时，@Rollback定义特定测试方法的回滚语义，可能会覆盖类级别的@Rollback或@Commit语义。

以下示例导致不回滚测试方法的结果（即，结果提交到数据库）：

```
@Rollback(false)
@Test
public void testProcessWithoutRollback() {
    // ...
}
```

@BeforeTransaction

@BeforeTransaction指示在启动事务之前应该运行带注解的void方法，对于已经配置为使用Spring的@Transactional注解在事务中运行的测试方法。 从Spring Framework 4.3开始，@BeforeTransaction方法不需要是公共的，可以在基于Java 8的接口默认方法中声明。

以下示例显示如何使用@BeforeTransaction批注：

```
@BeforeTransaction
void beforeTransaction() {
    // logic to be executed before a transaction is started
}
```

@AfterTransaction

对于已经配置为在事务中使用Spring的@Transactional注解运行的测试方法，@AfterTransaction指示应在事务结束后运行带注解的void方法。 从Spring Framework 4.3开始，@AfterTransaction方法不需要是公共的，可以在基于Java 8的接口默认方法中声明。

```
@AfterTransaction
void afterTransaction() {
    // logic to be executed after a transaction has ended
}
```

@Sql

@Sql用于注解测试类或测试方法，以配置在集成测试期间针对给定数据库运行的SQL脚本。 以下示例显示了如何使用它：

```
@Test
@Sql({"/test-schema.sql", "/test-user-data.sql"})
public void userTest {
    // execute code that relies on the test schema and test data
}
```

@SqlConfig

更多教程请访问:www.flydean.com

@SqlConfig定义元数据，用于确定如何解析和运行使用@Sql批注配置的SQL脚本。 以下示例显示了如何使用它：

```
@Test
@Sql(
    scripts = "/test-user-data.sql",
    config = @SqlConfig(commentPrefix = "`", separator = "@@")
)
public void userTest {
    // execute code that relies on the test data
}
```

@SqlGroup

@SqlGroup是一个容器注解，它聚合了几个@Sql注解。 你可以原生使用@SqlGroup来声明多个嵌套的@Sql注解，或者你可以将它与Java 8对可重复注解的支持结合使用，其中@Sql可以在同一个类或方法上多次声明，隐式生成此容器注解。 以下示例显示如何声明SQL组：

```
@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
}
```

3.4.2 标准注解支持

对于Spring TestContext Framework的所有配置，标准语义支持以下注解。 请注意，这些注解并非特定于测试，可以在Spring Framework中的任何位置使用。

@Autowired

@Qualifier

@Resource（javax.annotation）如果存在JSR-250

@ManagedBean（javax.annotation）如果存在JSR-250

@Inject（javax.inject）如果存在JSR-330

@Named（javax.inject）如果存在JSR-330

@PersistenceContext（javax.persistence）如果存在JPA

@PersistenceUnit（javax.persistence）如果存在JPA

@Required

@Transactional

> JSR-250生命周期注解
>
> 在Spring TestContext Framework中，你可以在ApplicationContext中配置的任何应用程序组件上使用@PostConstruct和@PreDestroy以及标准语义。 但是，这些生命周期注解在实际测试类中的使用有限。
>
> 如果测试类中的方法使用@PostConstruct注解，则该方法在基础测试框架的任何before方法之前运行（例如，使用JUnit Jupiter的@BeforeEach注解的方法），并且该方法适用于测试类中的每个测试方法。 另一方面，如果测试类中的方法使用@PreDestroy注解，则该方法永远不会运行。 因此，在测试类中，我们建议你使用来自底层测试框架的测试生命周期回调，而不是@PostConstruct和@PreDestroy。

更多教程请访问:www.flydean.com

3.4.3 Spring JUnit 4 测试注解

仅当与SpringRunner，Spring的JUnit 4规则或Spring的JUnit 4支持类一起使用时，才支持以下注解：

- @IfProfileValue
- @ProfileValueSourceConfiguration
- @Timed
- @Repeat

@IfProfileValue

@IfProfileValue表示为特定测试环境启用了带注解的测试。如果配置的ProfileValueSource返回所提供名称的匹配值，则启用测试。否则，测试被禁用，并且有效地被忽略。

你可以在类级别，方法级别或两者中应用@IfProfileValue。 @IfProfileValue的类级别使用优先于该类或其子类中的任何方法的方法级别使用。具体而言，如果在类级别和方法级别启用了测试，则启用测试。缺少@IfProfileValue意味着隐式启用了测试。这类似于JUnit 4的@Ignore注解的语义，除了@Ignore的存在总是禁用测试。

以下示例显示了具有@IfProfileValue批注的测试：

```
@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，你可以使用值列表（使用OR语义）配置@IfProfileValue，以在JUnit 4环境中为测试组实现类似TestNG的支持。请考虑以下示例：

```
@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是一个类级注解，它指定在检索通过@IfProfileValue注解配置的配置文件值时要使用的ProfileValueSource的类型。 如果未为测试声明@ProfileValueSourceConfiguration，则默认使用SystemProfileValueSource。 以下示例显示如何使用@ProfileValueSourceConfiguration：

```
@ProfileValueSourceConfiguration(CustomProfileValueSource.class)
public class CustomProfileValueSourceTests {
    // class body...
}
```

@Timed

@Timed表示带注解的测试方法必须在指定的时间段内（以毫秒为单位）完成执行。 如果文本执行时间超过指定的时间段，则测试失败。

时间段包括运行测试方法本身，测试的任何重复（参见@Repeat），以及测试夹具的任何设置或拆除。 以下示例显示了如何使用它：

```
@Timed(millis = 1000)
public void testProcessWithOneSecondTimeout() {
    // some logic that should not take longer than 1 second to execute
}
```

更多教程请访问:www.flydean.com

Spring的@Timed注解与JUnit 4的@Test（timeout = ...）支持有不同的语义。 具体来说，由于JUnit 4处理测试执行超时的方式（即，通过在单独的线程中执行测试方法），如果测试时间过长，@Test（timeout = ...）会提前让测试失败。 另一方面，Spring的@Timed没有提前让测试失败，而是在失败之前等待测试完成。

@Repeat

@Repeat表示必须重复运行带注解的测试方法。 在注解中指定测试方法的执行次数。

要重复的执行范围包括执行测试方法本身以及测试夹具的任何设置或拆除。 以下示例显示了如何使用@Repeat批注：

```
@Repeat(10)
@Test
public void testProcessRepeatedly() {
    // ...
}
```

3.4.4 Spring JUnit Jupiter 测试注解

仅当与SpringExtension和JUnit Jupiter（即JUnit 5中的编程模型）结合使用时，才支持以下注解：

- @SpringJUnitConfig
- @SpringJUnitWebConfig
- @EnabledIf
- @DisabledIf

@SpringJUnitConfig

@SpringJUnitConfig是一个组合注解，它将来自JUnit Jupiter的@ExtendWith（SpringExtension.class）与来自Spring TestContext Framework的@ContextConfiguration相结合。 它可以在类级别用作@ContextConfiguration的替代品。 关于配置选项，@ContextConfiguration和@SpringJUnitConfig之间的唯一区别是可以使用@SpringJUnitConfig中的value属性声明带注解的类。

以下示例显示如何使用@SpringJUnitConfig批注指定配置类：

```
@SpringJUnitConfig(TestConfig.class)
class ConfigurationClassJUnitJupiterSpringTests {
    // class body...
}
```

以下示例显示如何使用@SpringJUnitConfig批注指定配置文件的位置：

```
@SpringJUnitConfig(locations = "/test-config.xml")
class XmlJUnitJupiterSpringTests {
    // class body...
}
```

@SpringJUnitWebConfig

@SpringJUnitWebConfig是一个组合注解，它将来自JUnit Jupiter的@ExtendWith（SpringExtension.class）与来自Spring TestContext Framework的@ContextConfiguration和@WebAppConfiguration相结合。 你可以在类级别使用它作为@ContextConfiguration和@WebAppConfiguration的替代品。 关于配置选项，@ContextConfiguration和@SpringJUnitWebConfig之间的唯一区别是你可以使用@SpringJUnitWebConfig中的value属性声明带注解的类。 此外，只能使用@SpringJUnitWebConfig中的resourcePath属性覆盖@WebAppConfiguration中的value属性。

以下示例显示如何使用@SpringJUnitWebConfig批注指定配置类：

```
@SpringJUnitWebConfig(TestConfig.class)
```

更多教程请访问:www.flydean.com

```
class ConfigurationClassJUnitJupiterSpringWebTests {
    // class body...
}
```

以下示例显示如何使用@SpringJUnitWebConfig批注指定配置文件的位置：

```
@SpringJUnitWebConfig(locations = "/test-config.xml")
class XmlJUnitJupiterSpringWebTests {
    // class body...
}
```

@EnabledIf

@EnabledIf用于表示已启用带注解的JUnit Jupiter测试类或测试方法，并且如果提供的表达式的计算结果为true，则应运行该函数。 具体来说，如果表达式求值为Boolean.TRUE或字符串等于true（忽略大小写），则启用测试。 在类级别应用时，默认情况下也会自动启用该类中的所有测试方法。

表达式可以是以下任何一种：

- Spring Expression Language（SpEL）表达式。 例如：@EnabledIf("#{systemProperties['os.name'].toLowerCase().contains('mac')}")
- Spring环境中可用的属性的占位符。 例如：@EnabledIf("${smoke.tests.enabled}")
- 文字文字。 例如：@EnabledIf("true")

但请注意，不是属性占位符动态解析结果的文本文字的实用价值为零，因为@EnabledIf（“false”）等同于@Disabled而@EnabledIf（“true”）在逻辑上毫无意义。

你可以使用@EnabledIf作为元注解来创建自定义组合注解。 例如，你可以创建自定义@EnabledOnMac注解，如下所示：

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@EnabledIf(
    expression = "#{systemProperties['os.name'].toLowerCase().contains('mac')}",
    reason = "Enabled on Mac OS"
)
public @interface EnabledOnMac {}
```

@DisabledIf

@DisabledIf用于表示已禁用带注解的JUnit Jupiter测试类或测试方法，并且如果提供的表达式的计算结果为true，则不应执行。 具体来说，如果表达式求值为Boolean.TRUE或字符串等于true（忽略大小写），则禁用测试。 在类级别应用时，该类中的所有测试方法也会自动禁用。

表达式可以是如下格式：

- Spring Expression Language（SpEL）表达式。 例如：@DisabledIf("#{systemProperties['os.name'].toLowerCase().contains('mac')}")
- Spring环境中可用的属性的占位符。 例如： @DisabledIf("${smoke.tests.disabled}")
- 文字文字。 例如：@DisabledIf（“true”）

但请注意，不是属性占位符动态解析结果的文本文字实际值为零，因为@DisabledIf（“true”）等同于@Disabled，而@DisabledIf（“false”）在逻辑上没有意义。

你可以使用@DisabledIf作为元注解来创建自定义组合注解。 例如，你可以创建自定义@DisabledOnMac注解，如下所示：

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
```

更多教程请访问:www.flydean.com

```
@DisabledIf(
    expression = "#{systemProperties['os.name'].toLowerCase().contains('mac')}",
    reason = "Disabled on Mac OS"
)
public @interface DisabledOnMac {}
```

3.4.5 测试的元注解支持

你可以将大多数与测试相关的注解用作元注解，以创建自定义组合注解并减少测试套件中的配置重复。

你可以将以下各项作为元注解与TestContext框架结合使用。

- @BootstrapWith
- @ContextConfiguration
- @ContextHierarchy
- @ActiveProfiles
- @TestPropertySource
- @DirtiesContext
- @WebAppConfiguration
- @TestExecutionListeners
- @Transactional
- @BeforeTransaction
- @AfterTransaction
- @Commit
- @Rollback
- @sql
- @SqlConfig
- @SqlGroup
- @Repeat（仅在JUnit 4上受支持）
- @Timed（仅在JUnit 4上受支持）
- @IfProfileValue（仅在JUnit 4上受支持）
- @ProfileValueSourceConfiguration（仅在JUnit 4上受支持）
- @SpringJUnitConfig（仅在JUnit Jupiter上受支持）
- @SpringJUnitWebConfig（仅在JUnit Jupiter上受支持）
- @EnabledIf（仅在JUnit Jupiter上受支持）
- @DisabledIf（仅在JUnit Jupiter上受支持）

请考虑以下示例：

```
@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 { }
```

如果我们发现我们在基于JUnit 4的测试套件中重复了前面的配置，我们可以通过引入一个自定义组合注解来集中Spring的常用测试配置来减少重复，如下所示：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
```

更多教程请访问:www.flydean.com

```
@ContextConfiguration({"/app-config.xml", "/test-data-access-config.xml"})
@ActiveProfiles("dev")
@Transactional
public @interface TransactionalDevTestConfig { }
```

然后我们可以使用我们的自定义@TransactionalDevTestConfig注解来简化各个基于JUnit 4的测试类的配置，如下所示：

```
@RunWith(SpringRunner.class)
@TransactionalDevTestConfig
public class OrderRepositoryTests { }

@RunWith(SpringRunner.class)
@TransactionalDevTestConfig
public class UserRepositoryTests { }
```

如果我们编写使用JUnit Jupiter的测试，我们可以进一步减少代码重复，因为JUnit 5中的注解也可以用作元注解。 请考虑以下示例：

```
@ExtendWith(SpringExtension.class)
@ContextConfiguration({"/app-config.xml", "/test-data-access-config.xml"})
@ActiveProfiles("dev")
@Transactional
class OrderRepositoryTests { }

@ExtendWith(SpringExtension.class)
@ContextConfiguration({"/app-config.xml", "/test-data-access-config.xml"})
@ActiveProfiles("dev")
@Transactional
class UserRepositoryTests { }
```

如果我们发现我们在基于JUnit Jupiter的测试套件中重复上述配置，我们可以通过引入一个自定义组合注解来集中Spring和JUnit Jupiter的常用测试配置来减少重复，如下所示：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@ExtendWith(SpringExtension.class)
@ContextConfiguration({"/app-config.xml", "/test-data-access-config.xml"})
@ActiveProfiles("dev")
@Transactional
public @interface TransactionalDevTestConfig { }
```

然后我们可以使用我们的自定义@TransactionalDevTestConfig注解来简化各个基于JUnit Jupiter的测试类的配置，如下所示：

```
@TransactionalDevTestConfig
class OrderRepositoryTests { }

@TransactionalDevTestConfig
class UserRepositoryTests { }
```

由于JUnit Jupiter支持使用@Test，@RepeatedTest，ParameterizedTest和其他作为元注解，因此你还可以在测试方法级别创建自定义组合注解。 例如，如果我们希望创建一个组合注解，它将来自JUnit Jupiter的@Test和@Tag注解与Spring中的@Transactional注解相结合，我们可以创建一个@TransactionalIntegrationTest注解，如下所示：

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Transactional
@Tag("integration-test") // org.junit.jupiter.api.Tag
```

更多教程请访问:www.flydean.com

```
@Test // org.junit.jupiter.api.Test
public @interface TransactionalIntegrationTest { }
```

然后我们可以使用我们的自定义@TransactionalIntegrationTest批注来简化各个基于JUnit Jupiter的测试方法的配置，如下所示：

```
@TransactionalIntegrationTest
void saveOrder() { }

@TransactionalIntegrationTest
void deleteOrder() { }
```

更多教程请访问:www.flydean.com

3.5Spring TestContext框架

Spring TestContext Framework（位于org.springframework.test.context包中）提供了通用的，注解驱动的单元和集成测试支持，它与使用中的测试框架无关。 TestContext框架也非常重视约定优于配置，合理的默认值可以通过基于注解的配置覆盖。

除了通用测试基础结构之外，TestContext框架还为JUnit 4，JUnit Jupiter（AKA JUnit 5）和TestNG提供了显式支持。对于JUnit 4和TestNG，Spring提供了抽象支持类。此外，Spring为JUnit 4提供了一个自定义JUnit Runner和自定义JUnit规则，并为JUnit Jupiter提供了一个自定义扩展，可以编写所谓的POJO测试类。 POJO测试类不需要扩展特定的类层次结构，例如抽象支持类。

以下部分概述了TestContext框架的内部结构。如果你只对使用框架感兴趣并且不想使用自己的自定义侦听器或自定义加载器扩展它，请随意直接进入配置（上下文管理，依赖注入，事务管理），支持类和注解支持部分。

3.5.1 关键抽象

框架的核心包括TestContextManager类和TestContext，TestExecutionListener和SmartContextLoader接口。 为每个测试类创建一个TestContextManager（例如，用于在JUnit Jupiter中的单个测试类中执行所有测试方法）。 反过来，TestContextManager管理一个包含当前测试上下文的TestContext。 TestContextManager还会在测试进行时更新TestContext的状态，并委托给TestExecutionListener实现，这些实现通过提供依赖注入，管理事务等来检测实际的测试执行。 SmartContextLoader负责为给定的测试类加载ApplicationContext。 有关各种实现的更多信息和示例，请参阅javadoc和Spring测试套件。

TestContext

TestContext封装了执行测试的上下文（与使用中的实际测试框架无关），并为其负责的测试实例提供上下文管理和缓存支持。 TestContext还委托SmartContextLoader在需要时加载ApplicationContext。

TestContextManager

TestContextManager是Spring TestContext Framework的主要入口点，负责管理单个TestContext并在明确定义的测试执行点向每个注册的TestExecutionListener发送信号事件：

- 在特定测试框架的任何“before class”或“before all”方法之前。
- 测试实例后处理。
- 在特定测试框架的任何“before”或“before each”方法之前。
- 在执行测试方法之前，但在测试设置之后。
- 在执行测试方法之后但在测试之前立即拆除。
- 在特定测试框架的任何“after”或“after each”方法之后。
- 在特定测试框架的任何“after class”或“after all”方法之后。

TestExecutionListener

TestExecutionListener定义API，以响应由注册侦听器的TestContextManager发布的测试执行事件。 请参阅TestExecutionListener配置。

Context Loaders

ContextLoader是一个策略接口，用于为Spring TestContext Framework管理的集成测试加载ApplicationContext。你应该实现SmartContextLoader而不是此接口，以提供对带注解的类，活动Bean定义配置文件，测试属性源，上下文层次结构和WebApplicationContext支持的支持。

更多教程请访问:www.flydean.com

SmartContextLoader是Spring 3.1中引入的ContextLoader接口的扩展，取代了原始的最小ContextLoader SPI。具体来说，SmartContextLoader可以选择处理资源位置，带注解的类或上下文初始值设定项。此外，SmartContextLoader可以在其加载的上下文中设置活动Bean定义概要文件和测试属性源。

Spring提供以下实现：

- DelegatingSmartContextLoader：两个默认加载器之一，它内部委托给AnnotationConfigContextLoader，GenericXmlContextLoader或GenericGroovyXmlContextLoader，具体取决于为测试类声明的配置或默认位置或默认配置类的存在。仅当Groovy位于类路径上时才启用Groovy支持。
- WebDelegatingSmartContextLoader：两个默认加载器之一，它内部委托给AnnotationConfigWebContextLoader，GenericXmlWebContextLoader或GenericGroovyXmlWebContextLoader，具体取决于为测试类声明的配置或默认位置或默认配置类的存在。仅当测试类中存在@WebAppConfiguration时，才使用Web ContextLoader。仅当Groovy位于类路径上时才启用Groovy支持。
- AnnotationConfigContextLoader：从带注解的类中加载标准ApplicationContext。
- AnnotationConfigWebContextLoader：从带注解的类中加载WebApplicationContext。
- GenericGroovyXmlContextLoader：从Groovy脚本或XML配置文件的资源位置加载标准ApplicationContext。
- GenericGroovyXmlWebContextLoader：从Groovy脚本或XML配置文件的资源位置加载WebApplicationContext。
- GenericXmlContextLoader：从XML资源位置加载标准ApplicationContext。
- GenericXmlWebContextLoader：从XML资源位置加载WebApplicationContext。
- GenericPropertiesContextLoader：从Java属性文件加载标准ApplicationContext。

3.5.2 引导TestContext框架

Spring TestContext Framework内部的默认配置足以满足所有常见用例。但是，有时开发团队或第三方框架要更改默认的ContextLoader，实现自定义TestContext或ContextCache，扩充ContextCustomizerFactory和TestExecutionListener实现的默认集，依此类推。对于TestContext框架如何操作的这种低级控制，Spring提供了一个引导策略。

TestContextBootstrapper定义了用于引导TestContext框架的SPI。 TestContextManager使用TestContextBootstrapper为当前测试加载TestExecutionListener实现并构建它管理的TestContext。你可以使用@BootstrapWith直接或作为元注解为测试类（或测试类层次结构）配置自定义引导策略。如果未使用@BootstrapWith显式配置引导程序，则使用DefaultTestContextBootstrapper或WebTestContextBootstrapper，具体取决于@WebAppConfiguration的存在。

由于TestContextBootstrapper SPI将来可能会发生变化（以适应新的需求），我们强烈建议实施者不要直接实现此接口，而是扩展AbstractTestContextBootstrapper或其中一个具体的子类。

3.5.3 TestExecutionListener 配置

Spring提供了以下默认注册的TestExecutionListener实现，完全按以下顺序：

- ServletTestExecutionListener：为WebApplicationContext配置Servlet API模拟。
- DirtiesContextBeforeModesTestExecutionListener：处理“之前”模式的@DirtiesContext注解。
- DependencyInjectionTestExecutionListener：为测试实例提供依赖项注入。
- DirtiesContextTestExecutionListener：处理“after”模式的@DirtiesContext注解。
- TransactionalTestExecutionListener：使用默认回滚语义提供事务测试执行。
- SqlScriptsTestExecutionListener：运行使用@Sql批注配置的SQL脚本。

注册自定义TestExecutionListener实现

你可以使用@TestExecutionListeners批注为测试类及其子类注册自定义TestExecutionListener实现。有关详细信息和示例，请参阅注解支持和@TestExecutionListeners的javadoc。

自动发现默认的TestExecutionListener实现

更多教程请访问:www.flydean.com

使用@TestExecutionListeners注册自定义TestExecutionListener实现适用于在有限测试场景中使用的自定义侦听器。但是，如果需要在测试套件中使用自定义侦听器，则会变得很麻烦。从Spring Framework 4.1开始，通过支持通过SpringFactoriesLoader机制自动发现默认的TestExecutionListener实现来解决此问题。

具体来说，spring-test模块在其META-INF/spring.factories属性文件中的org.springframework.test.context.TestExecutionListener项下声明所有核心默认的TestExecutionListener`实现。第三方框架和开发人员可以通过自己的META-INF/spring.factories属性文件以相同的方式将自己的TestExecutionListener实现提供给默认侦听器列表。

订购TestExecutionListener实现

当TestContext框架通过前面提到的SpringFactoriesLoader机制发现默认的TestExecutionListener实现时，实例化的侦听器通过使用Spring的AnnotationAwareOrderComparator进行排序，它遵循Spring的Ordered接口和@Order注解进行排序。 AbstractTestExecutionListener和Spring提供的所有默认TestExecutionListener实现都使用适当的值进行Ordered。因此，第三方框架和开发人员应确保通过实现Ordered或声明@Order以正确的顺序注册其默认的TestExecutionListener实现。有关为每个核心侦听器分配的值的详细信息，请参阅javadoc以获取核心默认TestExecutionListener实现的getOrder（）方法。

合并TestExecutionListener实现

如果通过@TestExecutionListeners注册了自定义TestExecutionListener，则不会注册默认侦听器。在大多数常见的测试场景中，除了任何自定义侦听器之外，这还有效地迫使开发人员手动声明所有默认侦听器。以下清单演示了这种配置风格：

```
@ContextConfiguration
@TestExecutionListeners({
    MyCustomTestExecutionListener.class,
    ServletTestExecutionListener.class,
    DirtiesContextBeforeModesTestExecutionListener.class,
    DependencyInjectionTestExecutionListener.class,
    DirtiesContextTestExecutionListener.class,
    TransactionalTestExecutionListener.class,
    SqlScriptsTestExecutionListener.class
})
public class MyTest {
    // class body...
}
```

这种方法的挑战在于它要求开发人员确切地知道默认情况下注册了哪些监听器。此外，默认侦听器集可以在发行版之间更改 - 例如，Spring Framework 4.1中引入了SqlScriptsTestExecutionListener，而Spring Framework 4.2中引入了DirtiesContextBeforeModesTestExecutionListener。此外，Spring Security等第三方框架通过使用上述自动发现机制注册了自己的默认TestExecutionListener实现。

为避免必须知道并重新声明所有默认侦听器，可以将@TestExecutionListeners的mergeMode属性设置为MergeMode.MERGE_WITH_DEFAULTS。 MERGE_WITH_DEFAULTS指示本地声明的侦听器应与默认侦听器合并。合并算法确保从列表中删除重复项，并根据AnnotationAwareOrderComparator的语义对生成的合并侦听器集进行排序，如Ordering TestExecutionListener实现中所述。如果侦听器实现Ordered或使用@Order注解，它可以影响它与默认值合并的位置。否则，在合并时，本地声明的侦听器将附加到默认侦听器列表中。

例如，如果前一个示例中的MyCustomTestExecutionListener类将其顺序值（例如，500）配置为小于ServletTestExecutionListener（恰好为1000）的顺序，则MyCustomTestExecutionListener可以自动与默认值列表合并在ServletTestExecutionListener前面，前面的示例可以替换为以下内容：

```
@ContextConfiguration
@TestExecutionListeners(
    listeners = MyCustomTestExecutionListener.class,
    mergeMode = MERGE_WITH_DEFAULTS
)
public class MyTest {
```

更多教程请访问:www.flydean.com

```
    // class body...
}
```

3.5.4 Context 管理

每个TestContext都为其负责的测试实例提供上下文管理和缓存支持。 测试实例不会自动接收对已配置的ApplicationContext的访问权限。 但是，如果测试类实现ApplicationContextAware接口，则会向测试实例提供对ApplicationContext的引用。 请注意，AbstractJUnit4SpringContextTests和AbstractTestNGSpringContextTests实现ApplicationContextAware，因此可以自动提供对ApplicationContext的访问。

> @Autowired ApplicationContext
>
> 作为实现ApplicationContextAware接口的替代方法，你可以通过字段或setter方法上的@Autowired注解为测试类注入应用程序上下文，如以下示例所示：

```
@RunWith(SpringRunner.class)
@ContextConfiguration
public class MyTest {

    @Autowired
    private ApplicationContext applicationContext;

    // class body...
}
```

同样，如果你的测试配置为加载WebApplicationContext，则可以将Web应用程序上下文注入到测试中，如下所示：

```
@RunWith(SpringRunner.class)
@WebAppConfiguration
@ContextConfiguration
public class MyWebAppTest {

    @Autowired
    private WebApplicationContext wac;

    // class body...
}
```

> 使用@Autowired的依赖注入是由DependencyInjectionTestExecutionListener提供的，默认情况下配置它（参见测试夹具的依赖注入）。

使用TestContext框架的测试类不需要扩展任何特定类或实现特定接口来配置其应用程序上下文。相反，通过在类级别声明@ContextConfiguration批注来实现配置。如果测试类未显式声明应用程序上下文资源位置或带注解的类，则配置的ContextLoader将确定如何从默认位置或默认配置类加载上下文。除了上下文资源位置和带注解的类之外，还可以通过应用程序上下文初始化程序配置应用程序上下文。

以下部分说明如何使用Spring的@ContextConfiguration批注通过使用XML配置文件，Groovy脚本，带注解的类（通常为@Configuration类）或上下文初始化程序来配置测试ApplicationContext。或者，你可以为高级用例实现和配置自己的自定义SmartContextLoader。

Context Configuration with XML resources

要使用XML配置文件为测试加载ApplicationContext，请使用@ContextConfiguration注解测试类，并使用包含XML配置元数据的资源位置的数组配置locations属性。 普通路径或相对路径（例如，context.xml）被视为相对于定义测试类的包的类路径资源。 以斜杠开头的路径被视为绝对类路径位置（例如，/ org / example / config.xml）。 表示资源URL的路径（即，以classpath：，file：，http：等为前缀的路径）按原样使用。

更多教程请访问:www.flydean.com

```
@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通过标准Java值属性支持locations属性的别名。 因此，如果你不需要在@ContextConfiguration中声明其他属性，则可以省略位置属性名称的声明，并使用以下示例中演示的缩写格式声明资源位置：

```
@RunWith(SpringRunner.class)
@ContextConfiguration({"/app-config.xml", "/test-config.xml"})
public class MyTest {
    // class body...
}
```

如果省略@ContextConfiguration批注中的位置和值属性，则TestContext框架会尝试检测默认的XML资源位置。 具体来说，GenericXmlContextLoader和GenericXmlWebContextLoader根据测试类的名称检测默认位置。 如果你的类名为com.example.MyTest，则GenericXmlContextLoader将从“classpath：com / example / MyTest-context.xml”加载你的应用程序上下文。 以下示例显示了如何执行此操作：

```
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

要使用使用Groovy Bean Definition DSL的Groovy脚本为测试加载ApplicationContext，可以使用@ContextConfiguration注解测试类，并使用包含Groovy脚本资源位置的数组配置locations或value属性。 Groovy脚本的资源查找语义与XML配置文件中描述的相同。

> 启用Groovy脚本支持
>
> 如果Groovy在类路径上，则支持使用Groovy脚本在Spring TestContext Framework中加载ApplicationContext。

以下示例显示如何指定Groovy配置文件：

```
@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...
}
```

如果省略@ContextConfiguration批注中的位置和值属性，TestContext框架将尝试检测默认的Groovy脚本。 具体来说，GenericGroovyXmlContextLoader和GenericGroovyXmlWebContextLoader根据测试类的名称检测默认位置。 如果你的类名为com.example.MyTest，则Groovy上下文加载器将从“classpath：com / example / MyTestContext.groovy”加载你的应用程序上下文。 以下示例显示了如何使用默认值：

```
package com.example;
```

更多教程请访问:www.flydean.com

```
@RunWith(SpringRunner.class)
// ApplicationContext will be loaded from
// "classpath:com/example/MyTestContext.groovy"
@ContextConfiguration
public class MyTest {
    // class body...
}
```

> 同时声明XML配置和Groovy脚本
>
> 你可以使用@ContextConfiguration的locations或value属性同时声明XML配置文件和Groovy脚本。 如果配置的资源位置的路径以.xml结尾，则使用XmlBeanDefinitionReader加载它。 否则，使用GroovyBeanDefinitionReader加载它。
>
> 以下清单显示了如何在集成测试中将两者结合使用：

```
@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

要使用带注解的类为测试加载ApplicationContext（请参阅基于Java的容器配置），可以使用@ContextConfiguration注解测试类，并使用包含对带注解类的引用的数组配置classes属性。 以下示例显示了如何执行此操作：

```
@RunWith(SpringRunner.class)
// ApplicationContext will be loaded from AppConfig and TestConfig
@ContextConfiguration(classes = {AppConfig.class, TestConfig.class})
public class MyTest {
    // class body...
}
```

如果省略@ContextConfiguration批注中的classes属性，TestContext框架将尝试检测是否存在默认配置类。 具体来说，AnnotationConfigContextLoader和AnnotationConfigWebContextLoader将检测满足配置类实现要求的测试类的所有静态嵌套类，如@Configuration javadoc中所指定。 请注意，配置类的名称是任意的。 此外，如果需要，测试类可以包含多个静态嵌套配置类。 在以下示例中，OrderServiceTest类声明一个名为Config的静态嵌套配置类，该类自动用于加载测试类的ApplicationContext：

```
@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;
```

更多教程请访问:www.flydean.com

```
    @Test
    public void testOrderService() {
        // test the orderService
    }

}
```

Mixing XML, Groovy Scripts, and Annotated Classes

有时可能需要混合XML配置文件，Groovy脚本和带注解的类（通常为@Configuration类）来为测试配置ApplicationContext。例如，如果在生产中使用XML配置，则可能决定要使用@Configuration类为测试配置特定的Spring管理组件，反之亦然。

此外，一些第三方框架（例如Spring Boot）为同时从不同类型的资源加载ApplicationContext提供了一流的支持（例如，XML配置文件，Groovy脚本和@Configuration类）。历史上，Spring Framework并未支持标准部署。因此，Spring Framework在spring-test模块中提供的大多数SmartContextLoader实现仅支持每个测试上下文的一种资源类型。但是，这并不意味着你不能同时使用两者。一般规则的一个例外是GenericGroovyXmlContextLoader和GenericGroovyXmlWebContextLoader同时支持XML配置文件和Groovy脚本。此外，第三方框架可以选择通过@ContextConfiguration支持位置和类的声明，并且，通过TestContext框架中的标准测试支持，你可以使用以下选项。

如果要使用资源位置（例如，XML或Groovy）和@Configuration类来配置测试，则必须选择一个作为入口点，并且必须包含或导入另一个。例如，在XML或Groovy脚本中，你可以通过使用组件扫描包含@Configuration类或将它们定义为普通的Spring bean，而在@Configuration类中，你可以使用@ImportResource来导入XML配置文件或Groovy脚本。请注意，此行为在语义上等同于你在生产中配置应用程序的方式：在生产配置中，你可以定义一组XML或Groovy资源位置或一组@Configuration类，从中加载生产ApplicationContext，但你仍然拥有自由包含或导入其他类型的配置。

Context Configuration with Context Initializers

要使用上下文初始化程序为测试配置ApplicationContext，请使用@ContextConfiguration注解测试类，并使用包含对实现ApplicationContextInitializer的类的引用的数组配置initializers属性。 然后使用声明的上下文初始值设定项初始化为测试加载的ConfigurableApplicationContext。 请注意，每个声明的初始化程序支持的具体ConfigurableApplicationContext类型必须与正在使用的SmartContextLoader（通常是GenericApplicationContext）创建的ApplicationContext类型兼容。 此外，调用初始值设定项的顺序取决于它们是实现Spring的Ordered接口还是使用Spring的@Order注解或标准@Priority注解进行注解。 以下示例显示了如何使用初始值设定项：

```
@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...
}
```

你还可以完全省略@ContextConfiguration中的XML配置文件，Groovy脚本或带注解的类的声明，而是仅声明ApplicationContextInitializer类，然后它们负责在上下文中注册bean - 例如，通过以编程方式从XML文件加载bean定义或配置类。 以下示例显示了如何执行此操作：

```
@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...
}
```

更多教程请访问:www.flydean.com

Context Configuration Inheritance

@ContextConfiguration支持boolean inheritLocations和inheritInitializers属性，这些属性表示是否应继承超类声明的资源位置或带注解的类和上下文初始值设定项。两个标志的默认值均为true。这意味着测试类继承资源位置或带注解的类以及任何超类声明的上下文初始值设定项。具体而言，测试类的资源位置或带注解的类将附加到超类声明的资源位置列表或带注解的类。类似地，给定测试类的初始值设定项被添加到由测试超类定义的初始化程序集中。因此，子类可以选择扩展资源位置，带注解的类或上下文初始化器。

如果@ContextConfiguration中的inheritLocations或inheritInitializers属性设置为false，则资源位置或带注解的类和上下文初始化器分别为测试类阴影并有效地替换超类定义的配置。

在下一个使用XML资源位置的示例中，ExtendedTest的ApplicationContext按顺序从base-config.xml和extended-config.xml加载。因此，在extended-config.xml中定义的Bean可以覆盖（即替换）base-config.xml中定义的bean。以下示例显示了一个类如何扩展另一个类并使用其自己的配置文件和超类的配置文件：

```
@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...
}
```

类似地，在下一个使用带注解的类的示例中，ExtendedTest的ApplicationContext按顺序从BaseConfig和ExtendedConfig类加载。 因此，在ExtendedConfig中定义的Bean可以覆盖（即替换）BaseConfig中定义的Bean。 以下示例显示了一个类如何扩展另一个类并使用它自己的配置类和超类的配置类：

```
@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...
}
```

在下一个使用上下文初始化器的示例中，使用BaseInitializer和ExtendedInitializer初始化ExtendedTest的ApplicationContext。 但请注意，调用初始值设定项的顺序取决于它们是实现Spring的Ordered接口还是使用Spring的@Order注解或标准@Priority注解进行注解。 以下示例显示了一个类如何扩展另一个类并使用它自己的初始化程序和超类的初始化程序：

```
@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
```

更多教程请访问:www.flydean.com

```
@ContextConfiguration(initializers = ExtendedInitializer.class)
public class ExtendedTest extends BaseTest {
    // class body...
}
```

Context Configuration with Environment Profiles

Spring 3.1在框架中引入了环境和配置文件概念（AKA“bean定义配置文件”）的一流支持，并且可以配置集成测试以激活各种测试场景的特定bean定义配置文件。 这是通过使用@ActiveProfiles批注对测试类进行批注并提供在为测试加载ApplicationContext时应激活的配置文件列表来实现的。

> 你可以将@ActiveProfiles与新SmartContextLoader SPI的任何实现一起使用，但旧版ContextLoader SPI的实现不支持@ActiveProfiles。

考虑XML配置和@Configuration类的两个示例：

```
<!-- 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")
```

更多教程请访问:www.flydean.com

```
public class TransferServiceTest {

    @Autowired
    private TransferService transferService;

    @Test
    public void testTransferService() {
        // test the transferService
    }
}
```

运行TransferServiceTest时，将从类路径根目录中的app-config.xml配置文件加载其ApplicationContext。如果检查app-config.xml，则可以看到accountRepository bean依赖于dataSource bean。但是，dataSource未定义为顶级bean。相反，dataSource定义了三次：在生产配置文件中，在dev配置文件中，以及在默认配置文件中。

通过使用@ActiveProfiles（“dev”）注解TransferServiceTest，我们指示Spring TestContext Framework加载ApplicationContext，并将活动配置文件设置为{“dev”}。因此，创建嵌入式数据库并使用测试数据填充，并且accountRepository bean连接到对开发DataSource的引用。这可能是我们在集成测试中想要的。

将bean分配给默认配置文件有时很有用。仅当没有专门激活其他配置文件时，才会包含默认配置文件中的Bean。你可以使用它来定义要在应用程序的默认状态中使用的“回退”bean。例如，你可以显式为开发和生产配置文件提供数据源，但在这些数据源都不活动时将其定义为默认值。

以下代码清单演示了如何使用@Configuration类而不是XML实现相同的配置和集成测试：

```
@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();
    }
}
```

更多教程请访问:www.flydean.com

```
@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
    }
}
```

在此变体中，我们将XML配置拆分为四个独立的@Configuration类：

- TransferServiceConfig：使用@Autowired通过依赖注入获取dataSource。
- StandaloneDataConfig：为适合开发人员测试的嵌入式数据库定义dataSource。
- JndiDataConfig：定义在生产环境中从JNDI检索的dataSource。
- DefaultDataConfig：如果没有配置文件处于活动状态，则为默认嵌入式数据库定义dataSource。

与基于XML的配置示例一样，我们仍然使用@ActiveProfiles（“dev”）注解TransferServiceTest，但这次我们使用@ContextConfiguration批注指定所有四个配置类。测试类的主体本身保持完全不变。

通常情况下，在给定项目中的多个测试类中使用单组配置文件。因此，为了避免@ActiveProfiles注解的重复声明，你可以在基类上声明一次@ActiveProfiles，并且子类自动从基类继承@ActiveProfiles配置。在下面的示例中，@ActiveProfiles（以及其他注解）的声明已移至抽象超类AbstractIntegrationTest：

```
package com.bank.service;

@RunWith(SpringRunner.class)
@ContextConfiguration(classes = {
        TransferServiceConfig.class,
        StandaloneDataConfig.class,
```

更多教程请访问:www.flydean.com

```
        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还支持inheritProfiles属性，该属性可用于禁用活动配置文件的继承，如以下示例所示：

```
package com.bank.service;

// "dev" profile overridden with "production"
@ActiveProfiles(profiles = "production", inheritProfiles = false)
public class ProductionTransferServiceTest extends AbstractIntegrationTest {
    // test body
}
```

此外，有时需要以编程方式而不是声明性地解析测试的活动配置文件 - 例如，基于：

- 目前的操作系统。
- 是否在持续集成构建服务器上执行测试。
- 存在某些环境变量。
- 存在自定义类级注解。
- 其他问题。

要以编程方式解析活动Bean定义概要文件，可以实现自定义ActiveProfilesResolver并使用@ActiveProfiles的resolver属性进行注册。 有关详细信息，请参阅相应的javadoc。 以下示例演示如何实现和注册自定义OperatingSystemActiveProfilesResolver：

```
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};
```

更多教程请访问:www.flydean.com

```
    }
}
```

Context Configuration with Test Property Sources

Spring 3.1在框架中引入了具有属性源层次结构的环境概念的一流支持。 从Spring 4.1开始，你可以使用特定于测试的属性源配置集成测试。 与@Configuration类上使用的@PropertySource注解相比，你可以在测试类上声明@TestPropertySource注解，以声明测试属性文件或内联属性的资源位置。 这些测试属性源被添加到环境中的PropertySource集合中，用于为带注解的集成测试加载的ApplicationContext。

> 你可以将@TestPropertySource与SmartContextLoader SPI的任何实现一起使用，但旧版ContextLoader SPI的实现不支持@TestPropertySource。
>
> SmartContextLoader的实现通过MergedContextConfiguration中的getPropertySourceLocations（）和getPropertySourceProperties（）方法获得对合并的测试属性源值的访问。

声明测试属性源

你可以使用@TestPropertySource的locations或value属性配置测试属性文件。

支持传统和基于XML的属性文件格式 - 例如，“classpath：/com/example/test.properties”或“file：///path/to/file.xml”。

每个路径都被解释为Spring资源。 普通路径（例如，“test.properties”）被视为相对于定义测试类的包的类路径资源。 以斜杠开头的路径被视为绝对类路径资源（例如：“/ org / example / test.xml”）。 使用指定的资源协议加载引用URL的路径（例如，以classpath：，file：或http :)为前缀的路径。 不允许使用资源位置通配符（例如* /。属性）：每个位置必须仅评估一个.properties或.xml资源。

以下示例使用测试属性文件：

```
@ContextConfiguration
@TestPropertySource("/test.properties")
public class MyIntegrationTests {
    // class body...
}
```

你可以使用@TestPropertySource的properties属性以键值对的形式配置内联属性，如下一个示例所示。 所有键值对都作为具有最高优先级的单个测试PropertySource添加到封闭环境中。

键值对支持的语法与为Java属性文件中的条目定义的语法相同：

- key=value
- key:value
- key value

以下示例设置两个内联属性：

```
@ContextConfiguration
@TestPropertySource(properties = {"timezone = GMT", "port: 4242"})
public class MyIntegrationTests {
    // class body...
}
```

默认属性文件检测

如果将@TestPropertySource声明为空注解（即，没有位置或属性属性的显式值），则会尝试检测相对于声明注解的类的默认属性文件。例如，如果带注解的测试类是com.example.MyTest，则相应的默认属性文件是classpath：com / example / MyTest.properties。如果无法检测到默认值，则抛出IllegalStateException。

优先权

更多教程请访问:www.flydean.com

测试属性源的优先级高于从操作系统环境，Java系统属性或应用程序通过使用@PropertySource或以编程方式声明性地添加的属性源的优先级。因此，测试属性源可用于有选择地覆盖系统和应用程序属性源中定义的属性。此外，内联属性的优先级高于从资源位置加载的属性。

在下一个示例中，时区和端口属性以及“/test.properties”中定义的任何属性都会覆盖在系统和应用程序属性源中定义的同名属性。此外，如果“/test.properties”文件定义了时区和端口属性的条目，那么这些条目将被使用properties属性声明的内联属性覆盖。以下示例显示如何在文件和内联中指定属性：

```
@ContextConfiguration
@TestPropertySource(
    locations = "/test.properties",
    properties = {"timezone = GMT", "port: 4242"}
)
public class MyIntegrationTests {
    // class body...
}
```

继承和覆盖测试属性源

@TestPropertySource支持boolean inheritLocations和inheritProperties属性，这些属性表示是否应继承属性文件的资源位置和超类声明的内联属性。两个标志的默认值均为true。这意味着测试类继承了任何超类声明的位置和内联属性。具体而言，测试类的位置和内联属性将附加到超类声明的位置和内联属性。因此，子类可以选择扩展位置和内联属性。请注意，稍后出现的属性会影响（即覆盖）先前出现的同名属性。此外，上述优先规则也适用于继承的测试属性源。

如果@TestPropertySource中的inheritLocations或inheritProperties属性设置为false，则测试类的位置或内联属性分别影响并有效地替换超类定义的配置。

在下一个示例中，仅使用base.properties文件作为测试属性源来加载BaseTest的ApplicationContext。相反，使用base.properties和extended.properties文件作为测试属性源位置来加载ExtendedTest的ApplicationContext。以下示例显示如何使用属性文件在子类及其超类中定义属性：

```
@TestPropertySource("base.properties")
@ContextConfiguration
public class BaseTest {
    // ...
}

@TestPropertySource("extended.properties")
@ContextConfiguration
public class ExtendedTest extends BaseTest {
    // ...
}
```

在下一个示例中，仅使用内联key1属性加载BaseTest的ApplicationContext。 相反，使用内联的key1和key2属性加载ExtendedTest的ApplicationContext。 以下示例显示如何使用内联属性在子类及其超类中定义属性：

```
@TestPropertySource(properties = "key1 = value1")
@ContextConfiguration
public class BaseTest {
    // ...
}

@TestPropertySource(properties = "key2 = value2")
@ContextConfiguration
public class ExtendedTest extends BaseTest {
    // ...
}
```

Loading a WebApplicationContext

更多教程请访问:www.flydean.com

Spring 3.2引入了对在集成测试中加载WebApplicationContext的支持。要指示TestContext框架加载WebApplicationContext而不是标准ApplicationContext，可以使用@WebAppConfiguration注解相应的测试类。

测试类上存在@WebAppConfiguration指示TestContext框架（TCF）应为你的集成测试加载WebApplicationContext（WAC）。在后台，TCF确保创建MockServletContext并将其提供给测试的WAC。默认情况下，MockServletContext的基本资源路径设置为src / main / webapp。这被解释为相对于JVM根目录的路径（通常是项目的路径）。如果你熟悉Maven项目中Web应用程序的目录结构，则你知道src / main / webapp是WAR根目录的默认位置。如果需要覆盖此缺省值，则可以提供@WebAppConfiguration批注的备用路径（例如，@WebAppConfiguration（“src / test / webapp”））。如果你希望从类路径而不是文件系统引用基本资源路径，则可以使用Spring的classpath：前缀。

请注意，Spring对WebApplicationContext实现的测试支持与其对标准ApplicationContext实现的支持相同。使用WebApplicationContext进行测试时，可以使用@ContextConfiguration自由声明XML配置文件，Groovy脚本或@Configuration类。你也可以自由使用任何其他测试注解，例如@ActiveProfiles，@TestExecutionListeners，@Sql，@Rollback等。

本节中的其余示例显示了用于加载WebApplicationContext的一些配置选项。以下示例显示了TestContext框架对约定优于配置的支持：

```
@RunWith(SpringRunner.class)

// defaults to "file:src/main/webapp"
@WebAppConfiguration

// detects "WacTests-context.xml" in the same package
// or static nested @Configuration classes
@ContextConfiguration

public class WacTests {
    //...
}
```

如果使用@WebAppConfiguration注解测试类而未指定资源基路径，则资源路径有效默认为file：src / main / webapp。类似地，如果声明@ContextConfiguration而未指定资源位置，带注解的类或上下文初始值设定项，Spring会尝试使用约定来检测配置是否存在（即WacTests-context.xml与WacTests类相同的包或静态） 嵌套的@Configuration类）。

以下示例说明如何使用@WebAppConfiguration显式声明资源基路径，并使用@ContextConfiguration显示XML资源位置：

```
@RunWith(SpringRunner.class)

// file system resource
@WebAppConfiguration("webapp")

// classpath resource
@ContextConfiguration("/spring/test-servlet-config.xml")

public class WacTests {
    //...
}
```

这里要注意的重要一点是具有这两个注解的路径的不同语义。 默认情况下，@WebAppConfiguration资源路径是基于文件系统的，而@ContextConfiguration资源位置是基于类路径的。

以下示例显示我们可以通过指定Spring资源前缀来覆盖两个注解的默认资源语义：

```
@RunWith(SpringRunner.class)

// classpath resource
```

更多教程请访问:www.flydean.com

```
@WebAppConfiguration("classpath:test-web-resources")

// file system resource
@ContextConfiguration("file:src/main/webapp/WEB-INF/servlet-config.xml")

public class WacTests {
    //...
}
```

使用Web Mocks

为了提供全面的Web测试支持，Spring 3.2引入了一个默认启用的ServletTestExecutionListener。在针对WebApplicationContext进行测试时，此TestExecutionListener在每个测试方法之前使用Spring Web的RequestContextHolder设置默认线程局部状态，并基于使用@WebAppConfiguration配置的基本资源路径创建MockHttpServletRequest，MockHttpServletResponse和ServletWebRequest。 ServletTestExecutionListener还确保可以将MockHttpServletResponse和ServletWebRequest注入到测试实例中，并且一旦测试完成，它就会清除线程本地状态。

一旦为测试加载了WebApplicationContext，你可能会发现需要与Web模拟进行交互 - 例如，设置测试夹具或在调用Web组件后执行断言。以下示例显示可以将哪些模拟自动装配到测试实例中。请注意，WebApplicationContext和MockServletContext都在测试套件中缓存，而其他模拟由ServletTestExecutionListener按照测试方法进行管理。

```
@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

一旦TestContext框架为测试加载ApplicationContext（或WebApplicationContext），该上下文就会被缓存并重用于在同一测试套件中声明相同唯一上下文配置的所有后续测试。要了解缓存的工作原理，了解“独特”和“测试套件”的含义非常重要。

ApplicationContext可以通过用于加载它的配置参数的组合来唯一标识。因此，配置参数的唯一组合用于生成用于缓存上下文的密钥。 TestContext框架使用以下配置参数来构建上下文缓存键：

- locations（来自@ContextConfiguration）
- classes（来自@ContextConfiguration）
- contextInitializerClasses（来自@ContextConfiguration）
- contextCustomizers（来自ContextCustomizerFactory）
- contextLoader（来自@ContextConfiguration）

更多教程请访问:www.flydean.com

- parent（来自@ContextHierarchy）
- activeProfiles（来自@ActiveProfiles）
- propertySourceLocations（来自@TestPropertySource）
- propertySourceProperties（来自@TestPropertySource）
- resourceBasePath（来自@WebAppConfiguration）

例如，如果TestClassA为@ContextConfiguration的locations（或value）属性指定{“app-config.xml”，“test-config.xml”}，则TestContext框架会加载相应的ApplicationContext并将其存储在静态上下文缓存中在仅基于这些位置的密钥下。因此，如果TestClassB还为其位置定义{“app-config.xml”，“test-config.xml”}（通过继承显式或隐式）但未定义@WebAppConfiguration，则不同的ContextLoader，不同的活动配置文件，不同上下文初始化器，不同的测试属性源或不同的父上下文，然后两个测试类共享相同的ApplicationContext。这意味着加载应用程序上下文的设置成本仅产生一次（每个测试套件），并且后续测试执行要快得多。

> 测试套件和分叉流程
>
> Spring TestContext框架将应用程序上下文存储在静态缓存中。 这意味着上下文实际上存储在静态变量中。 换句话说，如果测试在单独的进程中执行，则在每次测试执行之间清除静态高速缓存，这有效地禁用了高速缓存机制。
>
> 要从缓存机制中受益，所有测试必须在同一进程或测试套件中运行。 这可以通过在IDE中作为一个组执行所有测试来实现。 类似地，当使用诸如Ant，Maven或Gradle之类的构建框架执行测试时，确保构建框架不在测试之间进行分配是很重要的。 例如，如果Maven Surefire插件的forkMode设置为always或pertest，则TestContext框架无法在测试类之间缓存应用程序上下文，因此构建过程运行速度明显更慢。

从Spring Framework 4.3开始，上下文缓存的大小受限于默认的最大大小32.每当达到最大大小时，最近最少使用（LRU）驱逐策略用于驱逐和关闭过时的上下文。你可以通过设置名为spring.test.context.cache.maxSize的JVM系统属性，从命令行或构建脚本配置最大大小。或者，你可以使用SpringProperties API以编程方式设置相同的属性。

由于在给定的测试套件中加载大量应用程序上下文会导致套件执行不必要的长时间，因此确切地知道已加载和缓存了多少上下文通常是有益的。要查看基础上下文缓存的统计信息，可以将org.springframework.test.context.cache日志记录类别的日志级别设置为DEBUG。

在不太可能的情况下，测试会破坏应用程序上下文并需要重新加载（例如，通过修改bean定义或应用程序对象的状态），你可以使用@DirtiesContext注解测试类或测试方法（请参阅@的讨论） @DirtiesContext中的DirtiesContext）。这指示Spring在运行需要相同应用程序上下文的下一个测试之前从缓存中删除上下文并重建应用程序上下文。请注意，对于@DirtiesContext注解的支持由DirtiesContextBeforeModesTestExecutionListener和DirtiesContextTestExecutionListener提供，默认情况下已启用。

Context Hierarchies

在编写依赖于加载的Spring ApplicationContext的集成测试时，通常可以针对单个上下文进行测试。但是，有时候对ApplicationContext实例的层次结构进行测试是有益的，甚至是必要的。例如，如果你正在开发Spring MVC Web应用程序，则通常会有Spring的ContextLoaderListener加载的根WebApplicationContext和Spring的DispatcherServlet加载的子WebApplicationContext。这将生成父子上下文层次结构，其中共享组件和基础结构配置在根上下文中声明，并由特定于Web的组件在子上下文中使用。可以在Spring Batch应用程序中找到另一个用例，其中你经常有一个父上下文，它提供共享批处理基础结构的配置，以及一个用于配置特定批处理作业的子上下文。

从Spring Framework 3.2.2开始，你可以编写使用上下文层次结构的集成测试，方法是在单个测试类或测试类层次结构中使用@ContextHierarchy注解声明上下文配置。如果在测试类层次结构中的多个类上声明了上下文层次结构，则还可以合并或覆盖上下文层次结构中特定的命名级别的上下文配置。合并层次结构中给定级别的配置时，配置资源类型（即XML配置文件或带注解的类）必须一致。否则，在使用不同资源类型配置的上下文层次结构中具有不同级别是完全可以接受的。

本节中剩余的基于JUnit 4的示例显示了需要使用上下文层次结构的集成测试的常见配置方案。

带上下文层次结构的单个测试类

更多教程请访问:www.flydean.com

ControllerIntegrationTests表示Spring MVC Web应用程序的典型集成测试场景，它声明了一个由两个级别组成的上下文层次结构，一个用于根WebApplicationContext（使用TestAppConfig @Configuration类加载），另一个用于调度程序servlet WebApplicationContext（通过使用加载） WebConfig @Configuration类）。自动装配到测试实例中的WebApplicationContext是子上下文的WebApplicationContext（即层次结构中的最低上下文）。以下清单显示了此配置方案：

```
@RunWith(SpringRunner.class)
@WebAppConfiguration
@ContextHierarchy({
    @ContextConfiguration(classes = TestAppConfig.class),
    @ContextConfiguration(classes = WebConfig.class)
})
public class ControllerIntegrationTests {

    @Autowired
    private WebApplicationContext wac;

    // ...
}
```

具有隐式父上下文的类层次结构

此示例中的测试类定义测试类层次结构中的上下文层次结构。 AbstractWebTests在Spring驱动的Web应用程序中声明根WebApplicationContext的配置。 但请注意，AbstractWebTests不声明@ContextHierarchy。 因此，AbstractWebTests的子类可以选择参与上下文层次结构或遵循@ContextConfiguration的标准语义。 SoapWebServiceTests和RestWebServiceTests都扩展了AbstractWebTests并使用@ContextHierarchy定义了一个上下文层次结构。 结果是加载了三个应用程序上下文（每个@ContextConfiguration声明一个），并且基于AbstractWebTests中的配置加载的应用程序上下文被设置为为具体子类加载的每个上下文的父上下文。 以下清单显示了此配置方案：

```
@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 SoapWebServiceTests extends AbstractWebTests {}

@ContextHierarchy(@ContextConfiguration("/spring/rest-ws-config.xml")
public class RestWebServiceTests extends AbstractWebTests {}
```

具有合并的上下文层次结构配置的类层次

此示例中的类显示了命名层次结构级别的使用，以便合并上下文层次结构中特定级别的配置。 BaseTests在层次结构中定义了两个级别，父级和子级。 ExtendedTests扩展了BaseTests，并通过确保@ContextConfiguration中name属性中声明的名称都是子级，指示Spring TestContext Framework合并子层次结构级别的上下文配置。 结果是加载了三个应用程序上下文：一个用于/app-config.xml，一个用于/user-config.xml，另一个用于{“/ user-config.xml”，“/ order-config.xml”}。 与前面的示例一样，从/app-config.xml加载的应用程序上下文被设置为从/user-config.xml和{“/ user-config.xml”，“/ order-config”加载的上下文的父上下文.XML“}。 以下清单显示了此配置方案：

```
@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")
)
```

更多教程请访问:www.flydean.com

```
public class ExtendedTests extends BaseTests {}
```

具有重写的上下文层次结构配置的类层次结构

与前面的示例相比，此示例演示了如何通过将@ContextConfiguration中的inheritLocations标志设置为false来覆盖上下文层次结构中给定命名级别的配置。 因此，ExtendedTests的应用程序上下文仅从/test-user-config.xml加载，并将其父级设置为从/app-config.xml加载的上下文。 以下清单显示了此配置方案：

```
@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 {}
```

> 在上下文层次结构中消除上下文
>
> 如果在其上下文配置为上下文层次结构的一部分的测试中使用@DirtiesContext，则可以使用hierarchyMode标志来控制如何清除上下文高速缓存。 有关更多详细信息，请参阅Spring Testing Annotations中的@DirtiesContext和@DirtiesContext javadoc的讨论。

3.5.5 测试夹具的依赖注入

使用DependencyInjectionTestExecutionListener（默认配置）时，测试实例的依赖项将从使用@ContextConfiguration或相关注解配置的应用程序上下文中的bean中注入。 你可以使用setter injection，field injection或两者，具体取决于你选择的注解以及是否将它们放在setter方法或字段上。 如果你正在使用JUnit Jupiter，你也可以选择使用构造函数注入（请参阅使用SpringExtension的依赖注入）。 为了与Spring基于注解的注入支持保持一致，你还可以使用Spring的@Autowired注解或JSR-330的@Inject注解进行字段和setter注入。

> 为了测试JUnit Jupiter以外的框架，TestContext框架不参与测试类的实例化。 因此，对构造函数使用@Autowired或@Inject对测试类没有影响。
>
> 虽然在生产代码中不鼓励字段注入，但实际注入在测试代码中实际上非常自然。 差异的基本原理是你永远不会直接实例化你的测试类。 因此，无需在测试类上调用公共构造函数或setter方法。

因为@Autowired用于按类型执行自动装配，如果你有多个相同类型的bean定义，则不能依赖此方法来处理这些特定的bean。在这种情况下，你可以将@Autowired与@Qualifier结合使用。从Spring 3.0开始，你还可以选择将@Inject与@Named结合使用。或者，如果你的测试类可以访问其ApplicationContext，则可以使用（例如）对applicationContext.getBean（“titleRepository”，TitleRepository.class）的调用来执行显式查找。

如果你不希望将依赖项注入应用于测试实例，请不要使用@Autowired或@Inject注解字段或setter方法。或者，你可以通过使用@TestExecutionListeners显式配置类并从侦听器列表中省略DependencyInjectionTestExecutionListener.class来完全禁用依赖项注入。

考虑一下测试HibernateTitleRepository类的场景，如目标部分所述。接下来的两个代码清单演示了在字段和setter方法上使用@Autowired。在所有示例代码列表之后呈现应用程序上下文配置。

> 以下代码清单中的依赖项注入行为并非特定于JUnit 4.相同的DI技术可与任何支持的测试框架结合使用。

更多教程请访问:www.flydean.com

> 以下示例调用静态断言方法，例如assertNotNull（），但不使用Assert预先调用。 在这种情况下，假设通过示例中未显示的import static 声明正确导入了该方法。

第一个代码清单显示了一个基于JUnit 4的测试类实现，它使用@Autowired进行字段注入：

```
@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);
    }
}
```

或者，你可以将类配置为使用@Autowired进行setter注入，如下所示：

```
@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);
    }
}
```

前面的代码清单使用@ContextConfiguration批注引用的相同XML上下文文件（即repository-config.xml）。 以下显示了此配置：

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

更多教程请访问:www.flydean.com

> 如果你从Spring提供的测试基类扩展到其自身的一个setter方法上使用@Autowired，则可能在应用程序上下文中定义了多个受影响类型的bean（例如，多个DataSource bean）。 在这种情况下，你可以覆盖setter方法并使用@Qualifier批注指示特定的目标bean，如下所示（但请确保也委托给超类中的重写方法）：

```
// ...

    @Autowired
    @Override
    public void setDataSource(@Qualifier("myDataSource") DataSource dataSource) {
        super.setDataSource(dataSource);
    }

// ...
```

> 指定的限定符值指示要注入的特定DataSource bean，将类型匹配集缩小到特定bean。 它的值与相应<bean>定义中的<qualifier>声明匹配。 bean名称用作回退限定符值，因此你还可以有效地指向特定的bean（如前所示，假设myDataSource是bean id）。

3.5.6 测试Request- 和Session-scoped Beans

Spring从早年开始就支持Request和session-scoped bean。从Spring 3.2开始，你可以按照以下步骤测试请求范围和会话范围的bean：

- 通过使用@WebAppConfiguration注解测试类，确保为测试加载WebApplicationContext。
- 将模拟请求或会话注入测试实例并根据需要准备测试夹具。
- 调用从配置的WebApplicationContext检索的Web组件（具有依赖项注入）。
- 对模拟执行断言。

下一个代码段显示了登录用例的XML配置。请注意，userService bean依赖于请求范围的loginAction bean。此外，LoginAction是通过使用SpEL表达式实例化的，该表达式从当前HTTP请求中检索用户名和密码。在我们的测试中，我们希望通过TestContext框架管理的模拟配置这些请求参数。以下清单显示了此用例的配置：

```
<beans>

    <bean id="userService" class="com.example.SimpleUserService"
            c:loginAction-ref="loginAction"/>

    <bean id="loginAction" class="com.example.LoginAction"
            c:username="#{request.getParameter('user')}"
            c:password="#{request.getParameter('pswd')}"
            scope="request">
        <aop:scoped-proxy/>
    </bean>

</beans>
```

在RequestScopedBeanTests中，我们将UserService（即测试中的主题）和MockHttpServletRequest注入我们的测试实例。 在我们的requestScope（）测试方法中，我们通过在提供的MockHttpServletRequest中设置请求参数来设置我们的测试夹具。 当在我们的userService上调用loginUser（）方法时，我们可以确保用户服务可以访问当前MockHttpServletRequest的请求范围的loginAction（也就是我们刚刚设置参数的那个）。 然后，我们可以根据用户名和密码的已知输入对结果执行断言。 以下清单显示了如何执行此操作：

```
@RunWith(SpringRunner.class)
@ContextConfiguration
@WebAppConfiguration
public class RequestScopedBeanTests {

    @Autowired UserService userService;
```

更多教程请访问:www.flydean.com

```
    @Autowired MockHttpServletRequest request;

    @Test
    public void requestScope() {
        request.setParameter("user", "enigma");
        request.setParameter("pswd", "$pr!ng");

        LoginResults results = userService.loginUser();
        // assert results
    }
}
```

以下代码片段类似于我们之前针对请求范围的bean看到的代码片段。 但是，这次，userService bean依赖于会话范围的userPreferences bean。 请注意，UserPreferences bean是使用SpEL表达式实例化的，该表达式从当前HTTP会话中检索主题。 在我们的测试中，我们需要在TestContext框架管理的模拟会话中配置主题。 以下示例显示了如何执行此操作：

```
<beans>

    <bean id="userService" class="com.example.SimpleUserService"
            c:userPreferences-ref="userPreferences" />

    <bean id="userPreferences" class="com.example.UserPreferences"
            c:theme="#{session.getAttribute('theme')}"
            scope="session">
        <aop:scoped-proxy/>
    </bean>

</beans>
```

在SessionScopedBeanTests中，我们将UserService和MockHttpSession注入到我们的测试实例中。 在我们的sessionScope（）测试方法中，我们通过在提供的MockHttpSession中设置预期的主题属性来设置我们的测试夹具。 当我们的userService上调用processUserPreferences（）方法时，我们可以确保用户服务可以访问当前MockHttpSession的会话范围的userPreferences，并且我们可以根据配置的主题对结果执行断言。 以下示例显示了如何执行此操作：

```
@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
    }
}
```

3.5.7 事物管理

在TestContext框架中，事务由TransactionalTestExecutionListener管理，即使你未在测试类上显式声明@TestExecutionListeners，也会默认配置该事务。但是，要启用对事务的支持，必须在ApplicationContext中配置一个加载了@ContextConfiguration语义的PlatformTransactionManager bean（稍后会提供更多详细信息）。此外，你必须在类或方法级别为测试声明Spring的@Transactional注解。

更多教程请访问:www.flydean.com

测试管理的事务

测试管理的事务是通过使用TransactionalTestExecutionListener以编程方式管理的事务，或使用TestTransaction以编程方式管理的事务（稍后介绍）。你不应将此类事务与Spring管理的事务（在为测试加载的ApplicationContext中由Spring直接管理的事务）或应用程序管理的事务（在测试调用的应用程序代码中以编程方式管理的事务）混淆。 Spring管理和应用程序管理的事务通常参与测试管理的事务。但是，如果Spring管理的事务或应用程序管理的事务配置了除REQUIRED或SUPPORTS之外的任何传播类型，则应谨慎使用（有关详细信息，请参阅有关事务传播的讨论）。

> **抢先超时和测试管理的交易**
>
> 当使用来自测试框架的任何形式的抢先超时以及Spring的测试管理事务时，必须小心。
>
> 具体来说，在调用当前测试方法之前，Spring的测试支持将事务状态绑定到当前线程（通过java.lang.ThreadLocal变量）。如果测试框架在新线程中调用当前测试方法以支持抢占超时，则不会在测试管理事务中调用当前测试方法中执行的任何操作。因此，任何此类操作的结果都不会与测试管理的事务一起回滚。相反，即使测试管理的事务由Spring正确回滚，这些操作也将被提交到持久存储 - 例如，关系数据库。
>
> 可能发生这种情况包括但不限于以下情况。
>
> JUnit 4的@Test（timeout = ...）支持和TimeOut规则
>
> JUnit Jupiter在org.junit.jupiter.api.Assertions类中的assertTimeoutPreemptively（...）方法
>
> TestNG的@Test（timeOut = ...）支持

启用和禁用事务

使用@Transactional注解测试方法会导致测试在事务中运行，默认情况下，该事务在测试完成后自动回滚。 如果使用@Transactional注解测试类，则该类层次结构中的每个测试方法都在事务中运行。 未使用@Transactional注解的测试方法（在类或方法级别）不在事务中运行。 此外，使用@Transactional注解但传播类型设置为NOT_SUPPORTED的测试不会在事务中运行。

请注意，AbstractTransactionalJUnit4SpringContextTests和AbstractTransactionalTestNGSpringContextTests已预先配置为类级别的事务支持。

以下示例演示了为基于Hibernate的UserRepository编写集成测试的常见方案：

```
@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
```

更多教程请访问:www.flydean.com

```
        sessionFactory.getCurrentSession().flush();
        assertNumUsers(count + 1);
    }

    protected int countRowsInTable(String tableName) {
        return JdbcTestUtils.countRowsInTable(this.jdbcTemplate, tableName);
    }

    protected void assertNumUsers(int expected) {
        assertEquals("Number of rows in the [user] table.", expected, countRowsInTable("user"));
    }
}
```

如事务回滚和提交行为中所述，在运行createUser（）方法后无需清理数据库，因为对TransactionalTestExecutionListener会自动回滚对数据库所做的任何更改。

事务回滚和提交行为

默认情况下，测试完成后将自动回滚测试事务; 但是，事务提交和回滚行为可以通过@Commit和@Rollback注解以声明方式配置。 有关更多详细信息，请参阅注解支持部分中的相应条目。

程序化交易管理

从Spring Framework 4.1开始，你可以使用TestTransaction中的静态方法以编程方式与测试管理的事务进行交互。 例如，你可以在测试方法中，在方法之前，在方法之后使用TestTransaction来启动或结束当前的测试管理事务，或者为回滚或提交配置当前测试管理的事务。 每当启用TransactionalTestExecutionListener时，都会自动提供对TestTransaction的支持。

以下示例演示了TestTransaction的一些功能。 有关详细信息，请参阅TestTransaction的javadoc。

```
@ContextConfiguration(classes = TestConfig.class)
public class ProgrammaticTransactionManagementTests extends
        AbstractTransactionalJUnit4SpringContextTests {

    @Test
    public void transactionalTest() {
        // assert initial state in test database:
        assertNumUsers(2);

        deleteFromTables("user");

        // changes to the database will be committed!
        TestTransaction.flagForCommit();
        TestTransaction.end();
        assertFalse(TestTransaction.isActive());
        assertNumUsers(0);

        TestTransaction.start();
        // perform other actions against the database that will
        // be automatically rolled back after the test completes...
    }

    protected void assertNumUsers(int expected) {
        assertEquals("Number of rows in the [user] table.", expected, countRowsInTable("user"));
    }
}
```

在交易之外运行代码

有时，你可能需要在事务测试方法之前或之后但在事务上下文之外执行某些代码 - 例如，在运行测试之前验证初始数据库状态或在测试运行后验证预期的事务提交行为（如果 test配置为提交事务）。 TransactionalTestExecutionListener完全支持@BeforeTransaction和@AfterTransaction注解。 你可以使用其中一个注解在测试类或测试接口中的任何void

更多教程请访问:www.flydean.com

default方法中注解任何void方法，并且TransactionalTestExecutionListener可确保你的before transaction方法或事务方法在适当的时间运行。

> 之前的任何方法（例如使用JUnit Jupiter的@BeforeEach注解的方法）和任何后续方法（例如使用JUnit Jupiter的@AfterEach注解的方法）都在事务中运行。 此外，对于未配置为在事务中运行的测试方法，不会运行使用@BeforeTransaction或@AfterTransaction注解的方法。

配置事务管理器

TransactionalTestExecutionListener期望在Spring ApplicationContext中为测试定义PlatformTransactionManager bean。如果测试的ApplicationContext中有多个PlatformTransactionManager实例，则可以使用@Transactional（“myTxMgr”）或@Transactional（transactionManager =“myTxMgr”）声明限定符，或者可以通过@Configuration类实现TransactionManagementConfigurer。有关用于在测试的ApplicationContext中查找事务管理器的算法的详细信息，请参阅testContextTransactionUtils.retrieveTransactionManager（）的javadoc。

演示所有与交易相关的注解

以下基于JUnit 4的示例显示了一个虚构的集成测试场景，该场景突出显示了所有与事务相关的注解。该示例不是为了演示最佳实践，而是为了演示如何使用这些注解。有关更多信息和配置示例，请参阅注解支持部分。 @Sql的事务管理包含一个使用@Sql进行声明性SQL脚本执行并使用默认事务回滚语义的附加示例。以下示例显示了相关注解：

```
@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
    }

}
```

> **在测试ORM代码时避免误报**
>
> 当你测试操作Hibernate会话或JPA持久性上下文的状态的应用程序代码时，请确保在运行该代码的测试方法中刷新基础工作单元。 未能刷新基础工作单元可能会产生误报：你的测试通过，但相同的代码会在实时生产环境中引发异常。 请注意，这适用于维护内存工作单元的任何ORM框架。 在下面基于Hibernate的示例测试用例中，一个方法演示了误报，另一个方法正确公开了刷新会话的结果：

更多教程请访问:www.flydean.com

```
// ...

@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();
}
```

以下示例显示JPA的匹配方法：

```
// ...

@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();
}
```

3.5.8 执行SQL脚本

在针对关系数据库编写集成测试时，执行SQL脚本来修改数据库模式或将测试数据插入表中通常是有益的。 spring-jdbc模块通过在加载Spring ApplicationContext时执行SQL脚本来支持初始化嵌入或现有数据库。有关详细信息，请参阅嵌入式数据库支持和使用嵌入式数据库测试数据

尽管在加载ApplicationContext时初始化数据库以进行一次测试非常有用，但有时在集成测试期间能够修改数据库是至关重要的。以下部分说明如何在集成测试期间以编程方式和声明方式执行SQL脚本。

以编程方式执行SQL脚本

Spring提供了以下选项，用于在集成测试方法中以编程方式执行SQL脚本。

org.springframework.jdbc.datasource.init.ScriptUtils

org.springframework.jdbc.datasource.init.ResourceDatabasePopulator

org.springframework.test.context.junit4.AbstractTransactionalJUnit4SpringContextTests

更多教程请访问:www.flydean.com

org.springframework.test.context.testng.AbstractTransactionalTestNGSpringContextTests

ScriptUtils提供了一组用于处理SQL脚本的静态实用程序方法，主要用于框架内部使用。但是，如果你需要完全控制SQL脚本的解析和执行方式，那么ScriptUtils可能比后面描述的其他一些替代方案更适合你的需求。有关更多详细信息，请参阅ScriptUtils中各个方法的javadoc。

ResourceDatabasePopulator提供基于对象的API，以便使用外部资源中定义的SQL脚本以编程方式填充，初始化或清理数据库。 ResourceDatabasePopulator提供用于配置解析和运行脚本时使用的字符编码，语句分隔符，注解分隔符和错误处理标志的选项。每个配置选项都有一个合理的默认值。有关默认值的详细信息，请参阅javadoc。要运行在ResourceDatabasePopulator中配置的脚本，可以调用populate（Connection）方法对java.sql.Connection执行populator，或者调用execute（DataSource）方法对javax.sql.DataSource执行populator。以下示例为测试模式和测试数据指定SQL脚本，将语句分隔符设置为@@，并针对DataSource执行脚本：

```
@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
}
```

请注意，ResourceDatabasePopulator内部委托给ScriptUtils来解析和运行SQL脚本。类似地，AbstractTransactionalJUnit4SpringContextTests和AbstractTransactionalTestNGSpringContextTests中的executeSqlScript（..）方法在内部使用ResourceDatabasePopulator来运行SQL脚本。有关更多详细信息，请参阅javadoc以获取各种executeSqlScript（..）方法。

使用@Sql以声明方式执行SQL脚本

除了上述以编程方式运行SQL脚本的机制之外，你还可以在Spring TestContext Framework中声明性地配置SQL脚本。具体来说，你可以在测试类或测试方法上声明@Sql批注，以配置应在集成测试方法之前或之后针对给定数据库运行的SQL脚本的资源路径。请注意，方法级声明会覆盖类级声明，并且SqlScriptsTestExecutionListener会提供对@Sql的支持，默认情况下会启用它。

路径资源语义

每个路径都被解释为Spring资源。普通路径（例如，“schema.sql”）被视为相对于定义测试类的包的类路径资源。以斜杠开头的路径被视为绝对类路径资源（例如，“/ org / example / sysma.sql”）。引用URL的路径（例如，以classpath：，file：，http :)为前缀的路径是使用指定的资源协议加载的。

以下示例显示如何在类级别和基于JUnit Jupiter的集成测试类中的方法级别使用@Sql：

```
@SpringJUnitConfig
@Sql("/test-schema.sql")
class DatabaseTests {

    @Test
    void emptySchemaTest {
        // execute code that uses the test schema without any test data
    }

    @Test
    @Sql({"/test-schema.sql", "/test-user-data.sql"})
    void userTest {
        // execute code that uses the test schema and test data
    }
}
```

更多教程请访问:www.flydean.com

默认脚本检测

如果未指定SQL脚本，则会尝试检测默认脚本，具体取决于声明@Sql的位置。如果无法检测到默认值，则抛出IllegalStateException。

- 类级声明：如果带注解的测试类是com.example.MyTest，则相应的默认脚本是classpath：com / example / MyTest.sql。
- 方法级声明：如果带注解的测试方法名为testMethod（）并且在类com.example.MyTest中定义，则相应的默认脚本为classpath：com / example / MyTest.testMethod.sql。

声明多个@Sql集

如果需要为给定的测试类或测试方法配置多组SQL脚本，但具有不同的语法配置，不同的错误处理规则或每组不同的执行阶段，则可以声明多个@Sql实例。使用Java 8，你可以使用@Sql作为可重复的注解。否则，你可以使用@SqlGroup批注作为显式容器来声明@Sql的多个实例。

以下示例显示如何将@Sql用作Java 8的可重复注解：

```
@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
}
```

在前面示例中显示的方案中，test-schema.sql脚本对单行注解使用不同的语法。

以下示例与前面的示例相同，只是@Sql声明在@SqlGroup中组合在一起，以便与Java 6和Java 7兼容。

```
@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
}
```

脚本执行阶段

默认情况下，SQL脚本在相应的测试方法之前执行。 但是，如果需要在测试方法之后运行特定的脚本集（例如，清理数据库状态），则可以使用@Sql中的executionPhase属性，如以下示例所示：

```
@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
}
```

请注意，ISOLATED和AFTER_TEST_METHOD分别从Sql.TransactionMode和Sql.ExecutionPhase静态导入。

使用@SqlConfig进行脚本配置

更多教程请访问:www.flydean.com

你可以使用@SqlConfig批注配置脚本解析和错误处理。当在集成测试类上声明为类级别注解时，@SqlConfig将作为测试类层次结构中所有SQL脚本的全局配置。当使用@Sql批注的config属性直接声明时，@SqlConfig用作在封闭的@Sql批注中声明的SQL脚本的本地配置。 @SqlConfig中的每个属性都有一个隐式默认值，该值在相应属性的javadoc中记录。由于在Java语言规范中为注解属性定义了规则，遗憾的是，不能将值null赋给注解属性。因此，为了支持继承的全局配置的覆盖，@SqlConfig属性具有显式默认值“”（对于字符串）或DEFAULT（对于枚举）。这种方法允许@SqlConfig的本地声明通过提供除“”或DEFAULT之外的值来有选择地覆盖@SqlConfig的全局声明中的各个属性。只要本地@SqlConfig属性不提供“”或“DEFAULT”以外的显式值，就会继承全局@SqlConfig属性。因此，显式本地配置会覆盖全局配置。

@Sql和@SqlConfig提供的配置选项等同于ScriptUtils和ResourceDatabasePopulator支持的配置选项，但它们是<jdbc：initialize-database /> XML命名空间元素提供的超集。有关详细信息，请参阅@Sql和@SqlConfig中各个属性的javadoc。

@Sql的事务管理

默认情况下，SqlScriptsTestExecutionListener为使用@Sql配置的脚本推断所需的事务语义。具体而言，SQL脚本在没有事务的情况下运行，在现有的Spring管理的事务中（例如，由TransactionalTestExecutionListener管理的事务，用于使用@Transactional注解的测试），或者在隔离的事务中运行，具体取决于transactionMode的配置值@SqlConfig中的属性以及测试的ApplicationContext中是否存在PlatformTransactionManager。但是，作为最低限度，测试的ApplicationContext中必须存在javax.sql.DataSource。

如果SqlScriptsTestExecutionListener用于检测DataSource和PlatformTransactionManager并推断事务语义的算法不符合你的需要，则可以通过设置@SqlConfig的dataSource和transactionManager属性来指定显式名称。此外，你可以通过设置@SqlConfig的transactionMode属性来控制事务传播行为（例如，脚本是否应该在隔离的事务中运行）。虽然对使用@Sql进行事务管理的所有受支持选项的详尽讨论超出了本参考手册的范围，但@SqlConfig和SqlScriptsTestExecutionListener的javadoc提供了详细信息，以下示例显示了使用JUnit Jupiter和事务测试的典型测试方案与@Sql：

```
@SpringJUnitConfig(TestDatabaseConfig.class)
@Transactional
class TransactionalSqlScriptsTests {

    final JdbcTemplate jdbcTemplate;

    @Autowired
    TransactionalSqlScriptsTests(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    @Test
    @Sql("/test-data.sql")
    void usersTest() {
        // verify state in test database:
        assertNumUsers(2);
        // execute code that uses the test data...
    }

    int countRowsInTable(String tableName) {
        return JdbcTestUtils.countRowsInTable(this.jdbcTemplate, tableName);
    }

    void assertNumUsers(int expected) {
        assertEquals(expected, countRowsInTable("user"),
            "Number of rows in the [user] table.");
    }
}
```

请注意，运行usersTest（）方法后无需清理数据库，因为对数据库所做的任何更改（在测试方法内或在/test-data.sql脚本中）都会自动回滚。 TransactionalTestExecutionListener（有关详细信息，请参阅事务管理）。

更多教程请访问:www.flydean.com

3.5.9 并行测试执行

Spring Framework 5.0引入了在使用Spring TestContext Framework时在单个JVM中并行执行测试的基本支持。 通常，这意味着大多数测试类或测试方法可以并行执行，而无需对测试代码或配置进行任何更改。

> 有关如何设置并行测试执行的详细信息，请参阅测试框架，构建工具或IDE的文档。

请记住，在测试套件中引入并发可能会导致意外的副作用，奇怪的运行时行为以及间歇性或看似随机失败的测试。 因此，Spring Team提供了以下关于何时不并行执行测试的一般指导原则。

如果测试，请不要并行执行测试：

使用Spring的@DirtiesContext支持。

使用JUnit 4的@FixMethodOrder支持或任何旨在确保测试方法按特定顺序运行的测试框架功能。 但请注意，如果并行执行整个测试类，则不适用。

更改共享服务或系统的状态，例如数据库，消息代理，文件系统等。 这适用于内存和外部系统。

如果并行测试执行失败，并且异常表明当前测试的ApplicationContext不再处于活动状态，这通常意味着ApplicationContext已从另一个线程中的ContextCache中删除。

这可能是由于使用了@DirtiesContext或者是由于ContextCache的自动驱逐。 如果@DirtiesContext是罪魁祸首，你需要找到一种方法来避免使用@DirtiesContext或从并行执行中排除此类测试。 如果已超出ContextCache的最大大小，则可以增加高速缓存的最大大小。 有关详细信息，请参阅有关上下文缓存的讨论

> Spring TestContext Framework中的并行测试执行只有在底层TestContext实现提供复制构造函数时才有可能，如testContext的javadoc中所述。 Spring中使用的DefaultTestContext提供了这样的构造函数。 但是，如果你使用提供自定义TestContext实现的第三方库，则需要验证它是否适合并行测试执行。

3.5.10 TestContext框架支持类

本节描述了支持Spring TestContext Framework的各种类。

Spring JUnit 4 Runner

Spring TestContext Framework通过自定义运行器（在JUnit 4.12或更高版本上支持）提供与JUnit 4的完全集成。通过使用@RunWith（SpringJUnit4ClassRunner.class）或更短的@RunWith（SpringRunner.class）变体来注解测试类，开发人员可以实现标准的基于JUnit 4的单元和集成测试，同时获得TestContext框架的好处，例如支持加载应用程序上下文，测试实例的依赖项注入，事务测试方法执行等。如果你想将Spring TestContext Framework与替代运行器（例如JUnit 4的Parameterized runner）或第三方运行程序（例如MockitoJUnitRunner）一起使用，你可以选择使用Spring对JUnit规则的支持。

以下代码清单显示了配置测试类以使用自定义Spring Runner运行的最低要求：

```
@RunWith(SpringRunner.class)
@TestExecutionListeners({})
public class SimpleTest {

    @Test
    public void testMethod() {
        // execute test logic...
    }
}
```

在前面的示例中，@TestExecutionListeners配置了一个空列表，以禁用默认侦听器，否则需要通过@ContextConfiguration配置ApplicationContext。

更多教程请访问:www.flydean.com

Spring JUnit 4规则

org.springframework.test.context.junit4.rules包提供以下JUnit 4规则（JUnit 4.12或更高版本支持）：

- SpringClassRule
- SpringMethodRule

SpringClassRule是一个JUnit TestRule，它支持Spring TestContext Framework的类级功能，而SpringMethodRule是一个JUnit MethodRule，它支持Spring TestContext Framework的实例级和方法级功能。

与SpringRunner相比，Spring基于规则的JUnit支持具有独立于任何org.junit.runner.Runner实现的优势，因此可以与现有的替代运行程序（例如JUnit 4的Parameterized）或第三方结合使用跑步者（如MockitoJUnitRunner）。

要支持TestContext框架的完整功能，必须将SpringClassRule与SpringMethodRule结合使用。以下示例显示了在集成测试中声明这些规则的正确方法：

```
// Optionally specify a non-Spring Runner via @RunWith(...)
@ContextConfiguration
public class IntegrationTest {

    @ClassRule
    public static final SpringClassRule springClassRule = new SpringClassRule();

    @Rule
    public final SpringMethodRule springMethodRule = new SpringMethodRule();

    @Test
    public void testMethod() {
        // execute test logic...
    }
}
```

JUnit 4支持类

org.springframework.test.context.junit4包为基于JUnit 4的测试用例提供以下支持类（在JUnit 4.12或更高版本上受支持）：

- AbstractJUnit4SpringContextTests
- AbstractTransactionalJUnit4SpringContextTests

AbstractJUnit4SpringContextTests是一个抽象基础测试类，它将Spring TestContext Framework与JUnit 4环境中的显式ApplicationContext测试支持集成在一起。扩展AbstractJUnit4SpringContextTests时，可以访问受保护的applicationContext实例变量，该变量可用于执行显式bean查找或测试整个上下文的状态。

AbstractTransactionalJUnit4SpringContextTests是AbstractJUnit4SpringContextTests的抽象事务扩展，它为JDBC访问添加了一些便利功能。此类要求在ApplicationContext中定义javax.sql.DataSource bean和PlatformTransactionManager bean。扩展AbstractTransactionalJUnit4SpringContextTests时，可以访问受保护的jdbcTemplate实例变量，该变量可用于运行SQL语句以查询数据库。你可以在运行与数据库相关的应用程序代码之前和之后使用此类查询来确认数据库状态，Spring确保此类查询在与应用程序代码相同的事务范围内运行。与ORM工具结合使用时，请务必避免误报。正如JDBC测试支持中所提到的，AbstractTransactionalJUnit4SpringContextTests还提供了方便的方法，通过使用前面提到的jdbcTemplate委托给JdbcTestUtils中的方法。此外，AbstractTransactionalJUnit4SpringContextTests提供了一个executeSqlScript（..）方法，用于针对配置的DataSource运行SQL脚本。

> 这些类方便扩展。 如果你不希望将测试类绑定到特定于Spring的类层次结构，则可以使用@RunWith（SpringRunner.class）或Spring的JUnit规则来配置你自己的自定义测试类。

JUnit Jupiter的SpringExtension

更多教程请访问:www.flydean.com

Spring TestContext Framework提供了与JUnit 5中引入的JUnit Jupiter测试框架的完全集成。通过使用@ExtendWith（SpringExtension.class）注解测试类，你可以实现基于JUnit Jupiter的标准单元和集成测试，同时获得以下好处： TestContext框架，例如支持加载应用程序上下文，依赖注入测试实例，事务测试方法执行等。

此外，由于JUnit Jupiter中的丰富扩展API，Spring提供了以下功能，超出了Spring支持JUnit 4和TestNG的功能集：

- 测试构造函数，测试方法和测试生命周期回调方法的依赖注入。有关更多详细信息，请参阅使用SpringExtension的依赖注入。
- 强大支持基于SpEL表达式，环境变量，系统属性等的条件测试执行。有关更多详细信息和示例，请参阅Spring JUnit Jupiter测试注解中的@EnabledIf和@DisabledIf文档。
- 自定义组合注解，结合了Spring和JUnit Jupiter的注解。有关详细信息，请参阅Meta-Annotation Support for Testing中的@TransactionalDevTestConfig和@TransactionalIntegrationTest示例。

以下代码清单显示了如何配置测试类以将SpringExtension与@ContextConfiguration结合使用：

```
// Instructs JUnit Jupiter to extend the test with Spring support.
@ExtendWith(SpringExtension.class)
// Instructs Spring to load an ApplicationContext from TestConfig.class
@ContextConfiguration(classes = TestConfig.class)
class SimpleTests {

    @Test
    void testMethod() {
        // execute test logic...
    }
}
```

由于你还可以在JUnit 5中使用注解作为元注解，因此Spring提供了@SpringJUnitConfig和@SpringJUnitWebConfig组合注解，以简化测试ApplicationContext和JUnit Jupiter的配置。

以下示例使用@SpringJUnitConfig来减少上一个示例中使用的配置量：

```
// Instructs Spring to register the SpringExtension with JUnit
// Jupiter and load an ApplicationContext from TestConfig.class
@SpringJUnitConfig(TestConfig.class)
class SimpleTests {

    @Test
    void testMethod() {
        // execute test logic...
    }
}
```

类似地，以下示例使用@SpringJUnitWebConfig创建WebApplicationContext以与JUnit Jupiter一起使用：

```
// Instructs Spring to register the SpringExtension with JUnit
// Jupiter and load a WebApplicationContext from TestWebConfig.class
@SpringJUnitWebConfig(TestWebConfig.class)
class SimpleWebTests {

    @Test
    void testMethod() {
        // execute test logic...
    }
}
```

有关更多详细信息，请参阅Spring JUnit Jupiter测试注解中的@SpringJUnitConfig和@SpringJUnitWebConfig的文档。

SpringExtension的依赖注入

更多教程请访问:www.flydean.com

SpringExtension从JUnit Jupiter实现了ParameterResolver扩展API，它允许Spring为测试构造函数，测试方法和测试生命周期回调方法提供依赖注入。

具体来说，SpringExtension可以将测试的ApplicationContext中的依赖项注入到使用@BeforeAll，@AfterAll，@BeforeEach，@AfterEach，@Test，@RequestTest，@ParameterizedTest等注解的测试构造函数和方法中。

构造函数注入

如果JUnit Jupiter测试类的构造函数中的参数是ApplicationContext类型（或其子类型），或者使用@Autowired，@Qualifier或@Value进行注解或元注解，则Spring会为该特定参数注入值使用来自test的ApplicationContext的相应bean。如果所有参数都应由Spring提供，你还可以使用@Autowired直接注解测试构造函数。

> 如果测试类的构造函数本身用@Autowired注解，则Spring承担解析构造函数中所有参数的责任。因此，没有向JUnit Jupiter注册的其他ParameterResolver可以解析此类构造函数的参数。
>
> 如果在测试方法之前或之后使用@DirtiesContext关闭测试的ApplicationContext，则不能将测试类的构造函数注入与JUnit Jupiter的@TestInstance（PER_CLASS）支持一起使用。
>
> 原因是@TestInstance（PER_CLASS）指示JUnit Jupiter在测试方法调用之间缓存测试实例。因此，测试实例将保留对最初从随后关闭的ApplicationContext注入的bean的引用。由于测试类的构造函数只会在这种情况下调用一次，因此依赖注入不会再次发生，后续测试将与关闭的ApplicationContext中的bean交互，这可能会导致错误。
>
> 要将@DirtiesContext与“测试前方法”或“测试后方法”模式结合使用@TestInstance（PER_CLASS），必须通过字段或setter注入来配置Spring的依赖关系，以便可以在测试之间重新注入它们方法调用。

在下面的示例中，Spring将从TestConfig.class加载的ApplicationContext中的OrderService bean注入OrderServiceIntegrationTests构造函数。

```
@SpringJUnitConfig(TestConfig.class)
class OrderServiceIntegrationTests {

    private final OrderService orderService;

    @Autowired
    OrderServiceIntegrationTests(OrderService orderService) {
        this.orderService = orderService.
    }

    // tests that use the injected OrderService
}
```

请注意，此功能允许测试依赖项是final的，因此是不可变的。

方法注入

如果JUnit Jupiter测试方法或测试生命周期回调方法中的参数是ApplicationContext类型（或其子类型），或者使用@Autowired，@Qualifier或@Value进行注解或元注解，则Spring会为其注入值 来自test的ApplicationContext的相应bean的特定参数。

在下面的示例中，Spring将从TestConfig.class加载的ApplicationContext中的OrderService注入到deleteOrder（）测试方法中：

```
@SpringJUnitConfig(TestConfig.class)
class OrderServiceIntegrationTests {

    @Test
    void deleteOrder(@Autowired OrderService orderService) {
        // use orderService from the test's ApplicationContext
    }
}
```

更多教程请访问:www.flydean.com

由于JUnit Jupiter中的ParameterResolver支持的健壮性，你还可以将多个依赖项注入单个方法，不仅来自Spring，还来自JUnit Jupiter本身或其他第三方扩展。

以下示例说明如何同时将Spring和JUnit Jupiter注入依赖项放入placeOrderRepeatedly（）测试方法。

```
@SpringJUnitConfig(TestConfig.class)
class OrderServiceIntegrationTests {

    @RepeatedTest(10)
    void placeOrderRepeatedly(RepetitionInfo repetitionInfo,
            @Autowired OrderService orderService) {

        // use orderService from the test's ApplicationContext
        // and repetitionInfo from JUnit Jupiter
    }
}
```

请注意，使用JUnit Jupiter中的@RepeatedTest可以让测试方法获得对RepetitionInfo的访问权限。

TestNG支持类

org.springframework.test.context.testng包为基于TestNG的测试用例提供以下支持类：

- AbstractTestNGSpringContextTests
- AbstractTransactionalTestNGSpringContextTests

AbstractTestNGSpringContextTests是一个抽象基础测试类，它将Spring TestContext Framework与TestNG环境中的显式ApplicationContext测试支持集成在一起。扩展AbstractTestNGSpringContextTests时，可以访问受保护的applicationContext实例变量，该变量可用于执行显式bean查找或测试整个上下文的状态。

AbstractTransactionalTestNGSpringContextTests是AbstractTestNGSpringContextTests的抽象事务扩展，它为JDBC访问添加了一些便利功能。此类要求在ApplicationContext中定义javax.sql.DataSource bean和PlatformTransactionManager bean。扩展AbstractTransactionalTestNGSpringContextTests时，可以访问受保护的jdbcTemplate实例变量，该变量可用于执行SQL语句以查询数据库。你可以在运行与数据库相关的应用程序代码之前和之后使用此类查询来确认数据库状态，Spring确保此类查询在与应用程序代码相同的事务范围内运行。与ORM工具结合使用时，请务必避免误报。正如JDBC测试支持中所提到的，AbstractTransactionalTestNGSpringContextTests还提供了方便的方法，通过使用上述的jdbcTemplate委托给JdbcTestUtils中的方法。此外，AbstractTransactionalTestNGSpringContextTests提供了一个executeSqlScript（..）方法，用于针对配置的DataSource运行SQL脚本。

> 这些类是便于扩展的。 如果你不希望将测试类绑定到特定于Spring的类层次结构，则可以使用@ContextConfiguration，@TestExecutionListeners等配置自己的自定义测试类，并使用TestContextManager手动检测测试类。 有关如何检测测试类的示例，请参阅AbstractTestNGSpringContextTests的源代码。

更多教程请访问:www.flydean.com

3.6Spring MVC测试框架

Spring MVC Test框架提供了一流的支持，可以使用流畅的API测试Spring MVC代码，你可以将其与JUnit，TestNG或任何其他测试框架一起使用。 它构建在spring-test模块的Servlet API模拟对象之上，因此不使用正在运行的Servlet容器。 它使用DispatcherServlet提供完整的Spring MVC运行时行为，并提供对使用TestContext框架加载实际Spring配置的支持以及独立模式，你可以在其中手动实例化控制器并一次测试一个。

Spring MVC Test还为测试使用RestTemplate的代码提供客户端支持。 客户端测试模拟服务器响应，也不使用正在运行的服务器。

> Spring Boot提供了编写完整的端到端集成测试的选项，包括正在运行的服务器。 如果这是你的目标，请参阅Spring Boot参考页面。 有关容器外和端到端集成测试之间差异的更多信息，请参阅容器外和端到端集成测试之间的差异。

3.6.1 服务端测试

你可以使用JUnit或TestNG为Spring MVC控制器编写一个普通的单元测试。为此，实例化控制器，使用模拟或存根依赖项注入它，并调用其方法（根据需要传递MockHttpServletRequest，MockHttpServletResponse和其他）。但是，在编写这样的单元测试时，仍有许多未经测试：例如，请求映射，数据绑定，类型转换，验证等等。此外，还可以调用其他控制器方法（如@InitBinder，@ModelAttribute和@ExceptionHandler）作为请求处理生命周期的一部分。

Spring MVC Test的目标是通过执行请求并通过实际的DispatcherServlet生成响应来提供测试控制器的有效方法。

Spring MVC Test建立在Spring测试模块中可用的Servlet API的熟悉的“模拟”实现之上。这允许执行请求并生成响应，而无需在Servlet容器中运行。在大多数情况下，一切都应该像在运行时一样工作，但有一些值得注意的例外，如容器外和端到端集成测试之间的差异所述。以下基于JUnit Jupiter的示例使用Spring MVC Test：

```
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@SpringJUnitWebConfig(locations = "test-servlet-context.xml")
class ExampleTests {

    private MockMvc mockMvc;

    @BeforeEach
    void setup(WebApplicationContext wac) {
        this.mockMvc = MockMvcBuilders.webAppContextSetup(wac).build();
    }

    @Test
    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"));
    }
}
```

上述测试依赖于TestContext框架的WebApplicationContext支持，从位于与测试类相同的包中的XML配置文件加载Spring配置，但也支持基于Java和Groovy的配置。请参阅这些示例测试。

MockMvc实例用于对/accounts/1执行GET请求，并验证生成的响应的状态为200，内容类型为application/json，响应正文具有名为name的值为Lee的JSON属性。 Jayway JsonPath项目支持jsonPath语法。验证执行请求结果的许多其他选项将在本文档的后面部分讨论。

更多教程请访问:www.flydean.com

静态Imports

上一节示例中的fluent API需要一些静态导入，例如MockMvcRequestBuilders.*，MockMvcResultMatchers.*和MockMvcBuilders.*。查找这些类的简单方法是搜索与MockMvc *匹配的类型。如果你使用Eclipse或基于Eclipse的Spring Tool Suite，请确保在Java→编辑器→内容辅助→收藏夹下的Eclipse首选项中将它们添加为“最喜欢的静态成员”。这样做可以在键入静态方法名称的第一个字符后使用内容辅助。其他IDE（例如IntelliJ）可能不需要任何其他配置。检查静态成员的代码完成支持。

设置选择

你有两个主要选项来创建MockMvc的实例。第一种是通过TestContext框架加载Spring MVC配置，该框架加载Spring配置并将WebApplicationContext注入测试以用于构建MockMvc实例。以下示例显示了如何执行此操作：

```
@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();
    }

    // ...

}
```

你的第二个选择是手动创建控制器实例而不加载Spring配置。 相反，将自动创建与MVC JavaConfig或MVC名称空间大致相当的基本默认配置。 你可以在一定程度上自定义它。 以下示例显示了如何执行此操作：

```
public class MyWebTests {

    private MockMvc mockMvc;

    @Before
    public void setup() {
        this.mockMvc = MockMvcBuilders.standaloneSetup(new AccountController()).build();
    }

    // ...

}
```

你应该使用哪种设置选项?

webAppContextSetup加载你实际的Spring MVC配置，从而实现更完整的集成测试。 由于TestContext框架缓存了加载的Spring配置，因此即使你在测试套件中引入了更多测试，它也有助于保持测试快速运行。 此外，你可以通过Spring配置将模拟服务注入控制器，以便专注于测试Web层。 以下示例使用Mockito声明模拟服务：

```
<bean id="accountService" class="org.mockito.Mockito" factory-method="mock">
    <constructor-arg value="org.example.AccountService"/>
</bean>
```

然后，你可以将模拟服务注入测试以设置和验证你的期望，如以下示例所示：

更多教程请访问:www.flydean.com

```
@RunWith(SpringRunner.class)
@WebAppConfiguration
@ContextConfiguration("test-servlet-context.xml")
public class AccountTests {

    @Autowired
    private WebApplicationContext wac;

    private MockMvc mockMvc;

    @Autowired
    private AccountService accountService;

    // ...

}
```

另一方面，standaloneSetup更接近单元测试。它一次测试一个控制器。你可以使用模拟依赖项手动注入控制器，但不涉及加载Spring配置。这些测试更侧重于样式，并且更容易看到正在测试哪个控制器，是否需要任何特定的Spring MVC配置，等等。 standaloneSetup也是编写临时测试以验证特定行为或调试问题的一种非常方便的方法。

与大多数“整合与单元测试”辩论一样，没有正确或错误的答案。但是，使用standaloneSetup确实需要额外的webAppContextSetup测试才能验证Spring MVC配置。或者，你可以使用webAppContextSetup编写所有测试，以便始终针对你的实际Spring MVC配置进行测试。

设置功能

无论你使用哪个MockMvc构建器，所有MockMvcBuilder实现都提供了一些常用且非常有用的功能。例如，你可以为所有请求声明Accept标头，并期望所有响应中的状态为200以及Content-Type标头，如下所示：

```
// 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();
```

此外，第三方框架（和应用程序）可以预先打包设置指令，例如MockMvcConfigurer中的设置指令。 Spring Framework有一个这样的内置实现，可以帮助跨请求保存和重用HTTP会话。 你可以按如下方式使用它：

```
// static import of SharedHttpSessionConfigurer.sharedHttpSession

MockMvc mockMvc = MockMvcBuilders.standaloneSetup(new TestController())
        .apply(sharedHttpSession())
        .build();

// Use mockMvc to perform requests...
```

执行请求

你可以执行使用任何HTTP方法的请求，如以下示例所示：

```
mockMvc.perform(post("/hotels/{id}", 42).accept(MediaType.APPLICATION_JSON));
```

你还可以执行内部使用MockMultipartHttpServletRequest的文件上载请求，以便不会实际解析多部分请求。 相反，你必须将其设置为类似于以下示例：

```
mockMvc.perform(multipart("/doc").file("a1", "ABC".getBytes("UTF-8")));
```

更多教程请访问:www.flydean.com

你可以在URI模板样式中指定查询参数，如以下示例所示：

```
mockMvc.perform(get("/hotels?thing={thing}", "somewhere"));
```

你还可以添加表示查询或表单参数的Servlet请求参数，如以下示例所示：

```
mockMvc.perform(get("/hotels").param("thing", "somewhere"));
```

如果应用程序代码依赖于Servlet请求参数，并且未明确检查查询字符串（通常情况下），则使用哪个选项无关紧要。 但请记住，URI模板提供的查询参数已被解码，而通过param（...）方法提供的请求参数预计已被解码。

在大多数情况下，最好将上下文路径和Servlet路径保留在请求URI之外。 如果必须使用完整请求URI进行测试，请确保相应地设置contextPath和servletPath以使请求映射有效，如以下示例所示：

```
mockMvc.perform(get("/app/main/hotels/{id}").contextPath("/app").servletPath("/main"))
```

在前面的示例中，为每个执行的请求设置contextPath和servletPath是很麻烦的。 相反，你可以设置默认请求属性，如以下示例所示：

```
public class MyWebTests {

    private MockMvc mockMvc;

    @Before
    public void setup() {
        mockMvc = standaloneSetup(new AccountController())
            .defaultRequest(get("/")
            .contextPath("/app").servletPath("/main")
            .accept(MediaType.APPLICATION_JSON)).build();
    }
```

上述属性会影响通过MockMvc实例执行的每个请求。 如果在给定请求上也指定了相同的属性，则它将覆盖默认值。 这就是默认请求中的HTTP方法和URI无关紧要的原因，因为必须在每个请求中指定它们。

定义期望

你可以通过在执行请求后附加一个或多个.andExpect（..）调用来定义期望，如以下示例所示：

```
mockMvc.perform(get("/accounts/1")).andExpect(status().isOk());
```

MockMvcResultMatchers.*提供了许多期望，其中一些期望与更详细的期望进一步嵌套。

期望分为两大类。 第一类断言验证响应的属性（例如，响应状态，标头和内容）。 这些是断言最重要的结果。

第二类断言超出了回应范围。 这些断言允许你检查Spring MVC的特定方面，例如处理请求的控制器方法，是否引发和处理异常，模型的内容是什么，选择了哪个视图，添加了哪些Flash属性等等。 它们还允许你检查Servlet特定方面，例如请求和会话属性。

以下测试断言绑定或验证失败：

```
mockMvc.perform(post("/persons"))
    .andExpect(status().isOk())
    .andExpect(model().attributeHasErrors("person"));
```

很多时候，在编写测试时，转储执行的请求的结果很有用。 你可以按如下方式执行此操作，其中print（）是来自MockMvcResultHandlers的静态导入：

更多教程请访问:www.flydean.com

```
mockMvc.perform(post("/persons"))
    .andDo(print())
    .andExpect(status().isOk())
    .andExpect(model().attributeHasErrors("person"));
```

只要请求处理不会导致未处理的异常，print（）方法就会将所有可用的结果数据打印到System.out。 Spring Framework 4.2引入了一个log（）方法和print（）方法的另外两个变体，一个接受OutputStream，另一个接受Writer。例如，调用print（System.err）将结果数据打印到System.err，而调用print（myWriter）将结果数据打印到自定义编写器。 如果要记录结果数据而不是打印结果数据，可以调用log（）方法，该方法将结果数据记录为org.springframework.test.web.servlet.result日志记录类别下的单个DEBUG消息。

在某些情况下，你可能希望直接访问结果并验证无法验证的内容。 这可以通过在所有其他期望之后附加.andReturn（）来实现，如以下示例所示：

```
MvcResult mvcResult = mockMvc.perform(post("/persons")).andExpect(status().isOk()).andReturn();
```

如果所有测试重复相同的期望，你可以在构建MockMvc实例时设置一次共同期望，如以下示例所示：

```
standaloneSetup(new SimpleController())
    .alwaysExpect(status().isOk())
    .alwaysExpect(content().contentType("application/json;charset=UTF-8"))
    .build()
```

请注意，如果不创建单独的MockMvc实例，则始终应用常见期望并且无法覆盖。

当JSON响应内容包含使用Spring HATEOAS创建的超媒体链接时，你可以使用JsonPath表达式验证生成的链接，如以下示例所示：

```
mockMvc.perform(get("/people").accept(MediaType.APPLICATION_JSON))
    .andExpect(jsonPath("$.links[?(@.rel == 'self')].href").value("http://localhost:8080/people"));
```

当XML响应内容包含使用Spring HATEOAS创建的超媒体链接时，你可以使用XPath表达式验证生成的链接：

```
Map<String, String> ns = Collections.singletonMap("ns", "http://www.w3.org/2005/Atom");
mockMvc.perform(get("/handle").accept(MediaType.APPLICATION_XML))
    .andExpect(xpath("/person/ns:link[@rel='self']/@href", ns).string("http://localhost:8080/people"));
```

Filter注册

设置MockMvc实例时，可以注册一个或多个Servlet Filter实例，如以下示例所示：

```
mockMvc = standaloneSetup(new PersonController()).addFilters(new CharacterEncodingFilter()).build();
```

注册过滤器通过Spring测试中的MockFilterChain调用，最后一个过滤器委托给DispatcherServlet。

流式响应

Spring MVC Test内置了无流量响应的无容器测试选项。使用Spring MVC流选项的应用程序可以使用WebTestClient对正在运行的服务器执行端到端的集成测试。 Spring Boot也支持这一点，你可以使用WebTestClient测试正在运行的服务器。另一个优点是能够使用项目Reactor中的StepVerifier，它允许声明对数据流的期望。

容器外和端到端集成测试之间的差异

如前所述，Spring MVC Test构建于spring-test模块的Servlet API模拟对象之上，并不使用正在运行的Servlet容器。因此，与实际客户端和服务器运行的完整端到端集成测试相比，存在一些重要差异。

更多教程请访问:www.flydean.com

最简单的思考方法是从空白的MockHttpServletRequest开始。无论你添加什么是请求变成什么。可能会让你感到意外的事情是默认情况下没有上下文路径;没有jsessionid cookie;没有转发，错误或异步调度;因此，没有实际的JSP渲染。相反，“转发”和“重定向”的URL保存在MockHttpServletResponse中，并且可以满足期望。

这意味着，如果使用JSP，则可以验证请求转发到的JSP页面，但不呈现HTML。换句话说，不调用JSP。但请注意，所有其他不依赖转发的呈现技术（如Thymeleaf和Freemarker）都会按预期将HTML呈现给响应主体。通过@ResponseBody方法渲染JSON，XML和其他格式也是如此。

或者，你可以考虑使用@WebIntegrationTest从Spring Boot获得完整的端到端集成测试支持。请参阅Spring Boot参考指南。

每种方法都有利弊。 Spring MVC Test中提供的选项在从经典单元测试到完全集成测试的规模上是不同的停止。可以肯定的是，Spring MVC Test中没有一个选项属于经典单元测试类别，但它们更接近它。例如，你可以通过将模拟服务注入控制器来隔离Web层，在这种情况下，你只通过DispatcherServlet测试Web层，但使用实际的Spring配置，因为你可以单独测试数据访问层与其上面的层。此外，你可以使用独立设置，一次关注一个控制器，并手动提供使其工作所需的配置。

使用Spring MVC Test时的另一个重要区别是，从概念上讲，这样的测试是服务器端的，因此你可以检查使用了什么处理程序，是否使用HandlerExceptionResolver处理异常，模型的内容是什么，绑定错误是什么有，和其他细节。这意味着编写期望更容易，因为服务器不是黑盒子，就像通过实际的HTTP客户端测试它一样。这通常是经典单元测试的一个优点：它更容易编写，推理和调试，但不能取代完全集成测试的需要。同时，重要的是不要忽视响应是最重要的检查事实。简而言之，即使在同一个项目中，也存在多种样式和测试策略的空间。

进一步的服务器端测试示例

框架自己的测试包括许多样本测试，旨在展示如何使用Spring MVC测试。你可以浏览这些示例以获取更多想法。此外，spring-mvc-showcase项目还具有基于Spring MVC Test的全面测试覆盖率。

3.6.2 HtmlUnit集成

Spring提供了MockMvc和HtmlUnit之间的集成。 这简化了使用基于HTML的视图时执行端到端测试的过程。 这种集成让你：

- 使用HtmlUnit，WebDriver和Geb等工具轻松测试HTML页面，而无需部署到Servlet容器。
- 在页面中测试JavaScript。
- （可选）使用模拟服务进行测试以加快测试速度。
- 在容器内端到端测试和容器外集成测试之间共享逻辑。

MockMvc使用不依赖于Servlet容器的模板技术（例如，Thymeleaf，FreeMarker等），但它不适用于JSP，因为它们依赖于Servlet容器。

为何选择HtmlUnit?

想到的最明显的问题是“为什么我需要这个？”答案最好通过探索一个非常基本的示例应用程序找到。 假设你有一个Spring MVC Web应用程序，它支持Message对象上的CRUD操作。 该应用程序还支持分页所有消息。 你会怎么做测试呢?

使用Spring MVC Test，我们可以轻松测试是否能够创建Message，如下所示：

```
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"));
```

更多教程请访问:www.flydean.com

如果我们想测试允许我们创建消息的表单视图，该怎么办？ 例如，假设我们的表单看起来像以下代码段：

```
<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>
```

我们如何确保我们的表单产生正确的请求以创建新消息？ 天真的尝试可能类似于以下内容：

```
mockMvc.perform(get("/messages/form"))
        .andExpect(xpath("//input[@name='summary']").exists())
        .andExpect(xpath("//textarea[@name='text']").exists());
```

该测试有一些明显的缺点。 如果我们更新控制器以使用参数消息而不是文本，我们的表单测试将继续通过，即使HTML表单与控制器不同步。 要解决这个问题，我们可以结合两个测试，如下所示：

```
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"));
```

这样可以降低我们的测试错误传递的风险，但仍然存在一些问题：

- 如果我们的页面上有多个表单怎么办？不可否认，我们可以更新我们的XPath表达式，但是由于我们考虑了更多因素，它们变得更加复杂：字段是否是正确的类型？字段是否已启用？等等。
- 另一个问题是我们正在做的工作量是我们预期的两倍。我们必须首先验证视图，然后我们使用我们刚刚验证的相同参数提交视图。理想情况下，这可以一次完成。
- 最后，我们仍然无法解释一些事情。例如，如果表单具有我们希望测试的JavaScript验证，该怎么办？

总体问题是测试网页不涉及单个交互。相反，它是用户如何与网页交互以及该网页如何与其他资源交互的组合。例如，表单视图的结果用作用户输入以创建消息。此外，我们的表单视图可能会使用影响页面行为的其他资源，例如JavaScript验证。

集成测试到救援？

为解决前面提到的问题，我们可以执行端到端集成测试，但这有一些缺点。考虑测试让我们浏览消息的视图。我们可能需要以下测试：

- 我们的页面是否向用户显示通知，指示消息为空时没有结果可用？
- 我们的页面是否正确显示单个消息？
- 我们的页面是否正确支持分页？

更多教程请访问:www.flydean.com

要设置这些测试，我们需要确保我们的数据库包含正确的消息。这导致了许多额外的挑战：

- 确保数据库中存在正确的消息可能很繁琐。 （考虑外键约束。）
- 测试可能会变慢，因为每个测试都需要确保数据库处于正确的状态。
- 由于我们的数据库需要处于特定状态，因此我们无法并行运行测试。
- 对自动生成的ID，时间戳等项目执行断言可能很困难。

这些挑战并不意味着我们应该放弃端到端的集成测试。相反，我们可以通过重构我们的详细测试来减少端到端集成测试的数量，以使用运行速度更快，更可靠且没有副作用的模拟服务。然后，我们可以实现少量真正的端到端集成测试，以验证简单的工作流程，以确保一切正常工作。

输入HtmlUnit Integration

那么我们如何才能在测试页面交互之间取得平衡，并在测试套件中保持良好的性能呢？答案是：“通过将MockMvc与HtmlUnit集成。”

HtmlUnit集成选项

当你想要将MockMvc与HtmlUnit集成时，你有许多选项：

- MockMvc和HtmlUnit：如果要使用原始HtmlUnit库，请使用此选项。
- MockMvc和WebDriver：使用此选项可以简化集成和端到端测试之间的开发和重用代码。
- MockMvc和Geb：如果要在集成和端到端测试之间使用Groovy进行测试，简化开发和重用代码，请使用此选项。

MockMvc和HtmlUnit

本节介绍如何集成MockMvc和HtmlUnit。如果要使用原始HtmlUnit库，请使用此选项。

MockMvc和HtmlUnit设置

首先，确保在net.sourceforge.htmlunit：htmlunit中包含了测试依赖项。为了将HtmlUnit与Apache HttpComponents 4.5+一起使用，你需要使用HtmlUnit 2.18或更高版本。

我们可以使用MockMvcWebClientBuilder轻松创建一个与MockMvc集成的HtmlUnit WebClient，如下所示：

```
@Autowired
WebApplicationContext context;

WebClient webClient;

@Before
public void setup() {
    webClient = MockMvcWebClientBuilder
            .webAppContextSetup(context)
            .build();
}
```

这可确保将任何引用localhost作为服务器的URL定向到我们的MockMvc实例，而无需真正的HTTP连接。 正常情况下，使用网络连接请求任何其他URL。 这让我们可以轻松测试CDN的使用。

MockMvc和HtmlUnit用法

现在我们可以像往常一样使用HtmlUnit，但不需要将我们的应用程序部署到Servlet容器。 例如，我们可以请求视图使用以下内容创建消息：

```
HtmlPage createMsgFormPage = webClient.getPage("http://localhost/messages/form");
```

一旦我们引用了HtmlPage，我们就可以填写表单并提交它以创建一条消息，如下例所示：

```
HtmlForm form = createMsgFormPage.getHtmlElementById("messageForm");
HtmlTextInput summaryInput = createMsgFormPage.getHtmlElementById("summary");
```

更多教程请访问:www.flydean.com

```
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();
```

最后，我们可以验证是否已成功创建新消息。 以下断言使用AssertJ库：

```
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!");
```

上述代码以多种方式改进了我们的MockMvc测试。 首先，我们不再需要显式验证我们的表单，然后创建一个看起来像表单的请求。 相反，我们请求表单，填写表单并提交表单，从而显着减少开销。

另一个重要因素是HtmlUnit使用Mozilla Rhino引擎来评估JavaScript。 这意味着我们还可以在页面中测试JavaScript的行为。

有关使用HtmlUnit的其他信息，请参阅HtmlUnit文档。

高级MockMvcWebClientBuilder

在目前为止的示例中，我们通过构建基于Spring TestContext Framework为我们加载的WebApplicationContext的WebClient，以最简单的方式使用了MockMvcWebClientBuilder。 在以下示例中重复此方法：

```
@Autowired
WebApplicationContext context;

WebClient webClient;

@Before
public void setup() {
    webClient = MockMvcWebClientBuilder
            .webAppContextSetup(context)
            .build();
}
```

我们还可以指定其他配置选项，如以下示例所示：

```
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();
}
```

作为替代方案，我们可以通过单独配置MockMvc实例并将其提供给MockMvcWebClientBuilder来执行完全相同的设置，如下所示：

更多教程请访问:www.flydean.com

```
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();
```

这更加冗长，但是，通过使用MockMvc实例构建WebClient，我们可以轻松获得MockMvc的全部功能。

MockMvc和WebDriver

在前面的部分中，我们已经了解了如何将MockMvc与原始HtmlUnit API结合使用。 在本节中，我们使用Selenium WebDriver中的其他抽象来使事情变得更加容易。

为什么选择WebDriver和MockMvc?

我们已经可以使用HtmlUnit和MockMvc，那么我们为什么要使用WebDriver呢？ Selenium WebDriver提供了一个非常优雅的API，可以让我们轻松地组织我们的代码。 为了更好地展示它的工作原理，我们将在本节中探讨一个示例。

假设我们需要确保正确创建消息。 测试涉及查找HTML表单输入元素，填写它们以及进行各种断言。

这种方法导致许多单独的测试，因为我们也想测试错误条件。 例如，如果我们只填写表单的一部分，我们希望确保收到错误。 如果我们填写整个表格，则应在之后显示新创建的消息。

如果其中一个字段被命名为“summary”，我们可能会在我们的测试中有多个类似于以下重复的内容：

```
HtmlTextInput summaryInput = currentPage.getHtmlElementById("summary");
summaryInput.setValueAttribute(summary);
```

那么如果我们将id更改为smmry会发生什么？ 这样做会迫使我们更新所有测试以包含此更改。 这违反了DRY原则，因此我们理想情况下应将此代码提取到自己的方法中，如下所示：

```
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);
}
```

这样做可确保我们在更改UI时不必更新所有测试。

我们甚至可以更进一步，将此逻辑放在一个Object中，该Object表示我们当前所在的HtmlPage，如下例所示：

```
public class CreateMessagePage {

    final HtmlPage currentPage;

    final HtmlTextInput summaryInput;

    final HtmlSubmitInput submit;

    public CreateMessagePage(HtmlPage currentPage) {
```

更多教程请访问:www.flydean.com

```
        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());
    }
}
```

以前，这种模式称为页面对象模式。 虽然我们当然可以使用HtmlUnit执行此操作，但WebDriver提供了一些我们将在以下各节中探讨的工具，以使此模式更容易实现。

MockMvc和WebDriver设置

要将Selenium WebDriver与Spring MVC Test框架一起使用，请确保你的项目包含对org.seleniumhq.selenium的测试依赖项：selenium-htmlunit-driver。

我们可以使用MockMvcHtmlUnitDriverBuilder轻松创建一个与MockMvc集成的Selenium WebDriver，如下例所示：

```
   @Autowired
    WebApplicationContext context;

    WebDriver driver;

    @Before
    public void setup() {
        driver = MockMvcHtmlUnitDriverBuilder
                .webAppContextSetup(context)
                .build();
    }
```

上面的示例确保将任何引用localhost作为服务器的URL定向到我们的MockMvc实例，而无需真正的HTTP连接。 正常情况下，使用网络连接请求任何其他URL。 这让我们可以轻松测试CDN的使用。

MockMvc和WebDriver用法

现在我们可以像往常一样使用WebDriver，但不需要将我们的应用程序部署到Servlet容器。 例如，我们可以请求视图使用以下内容创建消息：

```
CreateMessagePage page = CreateMessagePage.to (driver) ;
```

然后我们可以填写表单并提交它以创建消息，如下所示：

```
ViewMessagePage viewMessagePage =
    page.createMessage(ViewMessagePage.class, expectedSummary, expectedText);
```

更多教程请访问:www.flydean.com

这通过利用页面对象模式改进了HtmlUnit测试的设计。 正如我们在Why WebDriver和MockMvc？中提到的那样，我们可以将页面对象模式与HtmlUnit一起使用，但使用WebDriver会更容易。 请考虑以下CreateMessagePage实现：

```
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);
    }
}
```

最后，我们可以验证是否已成功创建新消息。 以下断言使用AssertJ断言库：

```
assertThat(viewMessagePage.getMessage()).isEqualTo(expectedMessage);
assertThat(viewMessagePage.getSuccess()).isEqualTo("Successfully created a new message");
```

我们可以看到我们的ViewMessagePage允许我们与自定义域模型进行交互。 例如，它公开了一个返回Message对象的方法：

```
public Message getMessage() throws ParseException {
    Message message = new Message();
    message.setId(getId());
    message.setCreated(getCreated());
    message.setSummary(getSummary());
    message.setText(getText());
    return message;
}
```

然后我们可以在断言中使用富域对象。

最后，我们不要忘记在测试完成时关闭WebDriver实例，如下所示：

```
@After
public void destroy() {
    if (driver != null) {
        driver.close();
    }
}
```

高级MockMvcHtmlUnitDriverBuilder

更多教程请访问:www.flydean.com

在目前为止的示例中，我们通过构建基于Spring TestContext Framework为我们加载的WebApplicationContext的WebDriver，以最简单的方式使用了MockMvcHtmlUnitDriverBuilder。 这里重复这种方法，如下：

```
@Autowired
WebApplicationContext context;

WebDriver driver;

@Before
public void setup() {
    driver = MockMvcHtmlUnitDriverBuilder
            .webAppContextSetup(context)
            .build();
}
```

我们还可以指定其他配置选项，如下所示：

```
    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();
    }
```

作为替代方案，我们可以通过单独配置MockMvc实例并将其提供给MockMvcHtmlUnitDriverBuilder来执行完全相同的设置，如下所示：

```
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();
```

这更加冗长，但是，通过使用MockMvc实例构建WebDriver，我们可以轻松获得MockMvc的全部功能。

MockMvc和Geb

在上一节中，我们了解了如何将MockMvc与WebDriver一起使用。 在本节中，我们使用Geb使我们的测试甚至是Groovy-er。

为什么选择Geb和MockMvc?

Geb由WebDriver支持，因此它提供了许多与WebDriver相同的好处。 但是，通过为我们处理一些样板代码，Geb使事情变得更加容易。

MockMvc和Geb设置

更多教程请访问:www.flydean.com

我们可以使用使用MockMvc的Selenium WebDriver轻松初始化Geb浏览器，如下所示：

```
def setup() {
    browser.driver = MockMvcHtmlUnitDriverBuilder
        .webAppContextSetup(context)
        .build()
}
```

这可确保将引用localhost作为服务器的任何URL定向到我们的MockMvc实例，而无需真正的HTTP连接。 正常使用网络连接请求任何其他URL。 这让我们可以轻松测试CDN的使用。

MockMvc和Geb用法

现在我们可以像往常一样使用Geb，但不需要将我们的应用程序部署到Servlet容器。 例如，我们可以请求视图使用以下内容创建消息：

```
to CreateMessagePage
```

然后我们可以填写表单并提交它以创建消息，如下所示：

```
when:
form.summary = expectedSummary
form.text = expectedMessage
submit.click(ViewMessagePage)
```

任何无法识别的方法调用或属性访问或未找到的引用都将转发到当前页面对象。 这样就删除了直接使用WebDriver时需要的大量样板代码。

与直接使用WebDriver一样，这通过使用页面对象模式改进了HtmlUnit测试的设计。 如前所述，我们可以将页面对象模式与HtmlUnit和WebDriver一起使用，但使用Geb更容易。 考虑一下我们新的基于Groovy的CreateMessagePage实现：

```
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() }
    }
}
```

我们的CreateMessagePage扩展了Page。 我们不会详细介绍Page，但总的来说，它包含了我们所有页面的常用功能。 我们定义了一个可以在其中找到此页面的URL。 这让我们可以导航到页面，如下所示：

```
to CreateMessagePage
```

我们还有一个at闭包，用于确定我们是否在指定页面。 如果我们在正确的页面上，它应该返回true。 这就是为什么我们可以断言我们在正确的页面上，如下所示：

```
then:
at CreateMessagePage
errors.contains('This field is required.')
```

接下来，我们创建一个内容闭包，指定页面中所有感兴趣的区域。 我们可以使用jQuery-ish Navigator API来选择我们感兴趣的内容。

更多教程请访问:www.flydean.com

最后，我们可以验证是否已成功创建新消息，如下所示：

```
then:
at ViewMessagePage
success == 'Successfully created a new message'
id
date
summary == expectedSummary
message == expectedMessage
```

3.6.3. 客户端REST测试

你可以使用客户端测试来测试内部使用RestTemplate的代码。 我们的想法是声明预期的请求并提供“存根”响应，以便你可以专注于单独测试代码（即，无需运行服务器）。 以下示例显示了如何执行此操作：

```
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();
```

在前面的示例中，MockRestServiceServer（客户端REST测试的中心类）使用自定义ClientHttpRequestFactory配置RestTemplate，该客户端根据预期断言实际请求并返回“存根”响应。 在这种情况下，我们期望对/ greeting发出请求，并希望返回带有text / plain内容的200响应。 我们可以根据需要定义其他预期请求和存根响应。 当我们定义期望的请求和存根响应时，RestTemplate可以像往常一样在客户端代码中使用。 在测试结束时，mockServer.verify（）可用于验证是否已满足所有期望。

默认情况下，请求按预期声明的顺序进行。 你可以在构建服务器时设置ignoreExpectOrder选项，在这种情况下，将检查所有期望（按顺序）以查找给定请求的匹配项。 这意味着允许请求以任何顺序出现。 以下示例使用ignoreExpectOrder：

```
server = MockRestServiceServer.bindTo(restTemplate).ignoreExpectOrder(true).build();
```

即使默认情况下无序请求，每个请求也只允许执行一次。 expect方法提供了一个重载变量，它接受指定计数范围的ExpectedCount参数（例如，once，manyTimes，max，min，between等）。 以下示例使用时间：

```
RestTemplate restTemplate = new RestTemplate();

MockRestServiceServer mockServer = MockRestServiceServer.bindTo(restTemplate).build();
mockServer.expect(times(2), requestTo("/something")).andRespond(withSuccess());
mockServer.expect(times(3), requestTo("/somewhere")).andRespond(withSuccess());

// ...

mockServer.verify();
```

请注意，如果未设置ignoreExpectOrder（默认值），因此请求按声明顺序排列，则该顺序仅适用于任何预期请求中的第一个。 例如，如果“/ something”预期两次，然后是“/ somewhere”三次，那么在向“/ somewhere”发出请求之前应该有“/ something”的请求，但是，除了后续的“/”之外 某些“和”/某处“，请求可以随时出现。

作为上述所有选项的替代方案，客户端测试支持还提供了ClientHttpRequestFactory实现，你可以将其配置为RestTemplate以将其绑定到MockMvc实例。 这允许使用实际的服务器端逻辑处理请求但不运行服务器。 以下示例显示了如何执行此操作：

更多教程请访问:www.flydean.com

```
MockMvc mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).build();
this.restTemplate = new RestTemplate(new MockMvcClientHttpRequestFactory(mockMvc));

// Test code that uses the above RestTemplate ...
```

静态进口

与服务器端测试一样，用于客户端测试的流畅API需要一些静态导入。 通过搜索MockRest.*很容易找到它们。 Eclipse用户应该在Java→编辑器→内容辅助→收藏夹下的Eclipse首选项中添加MockRestRequestMatchers.*和MockRestResponseCreators。*作为“最喜欢的静态成员”。 这允许在键入静态方法名称的第一个字符后使用内容辅助。 其他IDE（例如IntelliJ）可能不需要任何其他配置。 检查静态成员对代码完成的支持。

客户端REST测试的更多示例

Spring MVC Test自己的测试包括客户端REST测试的示例测试。

更多教程请访问:www.flydean.com

3.7WebTestClient

WebTestClient是围绕WebClient的一个瘦shell，使用它来执行请求并公开一个专用的，流畅的API来验证响应。 WebTestClient通过使用模拟请求和响应绑定到WebFlux应用程序，或者它可以通过HTTP连接测试任何Web服务器。

3.7.1 Setup

要创建WebTestClient，你必须选择多个服务器设置选项之一。 实际上，你要么将WebFlux应用程序配置为绑定到，要么使用URL连接到正在运行的服务器。

绑定到控制器

以下示例显示如何创建服务器设置以一次测试一个@Controller：

```
    client = WebTestClient.bindToController(new TestController()).build();
```

上面的示例加载WebFlux Java配置并注册给定的控制器。 通过使用模拟请求和响应对象，在没有HTTP服务器的情况下测试生成的WebFlux应用程序。 构建器上有更多方法可以自定义默认的WebFlux Java配置。

绑定到路由器功能

以下示例显示如何从RouterFunction设置服务器：

```
RouterFunction<?> route = ...
    client = WebTestClient.bindToRouterFunction(route).build();
```

在内部，配置传递给RouterFunctions.toWebHandler。 通过使用模拟请求和响应对象，在没有HTTP服务器的情况下测试生成的WebFlux应用程序。

绑定到ApplicationContext

以下示例说明如何从应用程序的Spring配置或其某个子集设置服务器：

```
    @RunWith(SpringRunner.class)
    @ContextConfiguration(classes = WebConfig.class)
    public class MyTests {

        @Autowired
        private ApplicationContext context;

        private WebTestClient client;

        @Before
        public void setUp() {
            client = WebTestClient.bindToApplicationContext(context).build();
        }
```

在内部，配置被传递给WebHttpHandlerBuilder以设置请求处理链。 有关更多详细信息，请参阅WebHandler API。 通过使用模拟请求和响应对象，在没有HTTP服务器的情况下测试生成的WebFlux应用程序。

绑定到服务器

以下服务器设置选项允许你连接到正在运行的服务器：

```
    client = WebTestClient.bindToServer().baseUrl("http://localhost:8080").build();
```

更多教程请访问:www.flydean.com

Client Builder

除了前面描述的服务器设置选项之外，你还可以配置客户端选项，包括基本URL，默认标头，客户端过滤器等。 bindToServer之后可以使用这些选项。 对于所有其他人，你需要使用configureClient（）从服务器配置转换为客户端配置，如下所示：

```
client = WebTestClient.bindToController(new TestController())
            .configureClient()
            .baseUrl("/test")
            .build();
```

3.7.2 写Tests

WebTestClient提供与WebClient相同的API，直到使用exchange（）执行请求。 在exchange（）之后是用于验证响应的链式API工作流。

通常，你首先声明响应状态和标头，如下所示：

```
   client.get().uri("/persons/1")
            .accept(MediaType.APPLICATION_JSON_UTF8)
            .exchange()
            .expectStatus().isOk()
            .expectHeader().contentType(MediaType.APPLICATION_JSON_UTF8)
            // ...
```

然后指定如何解码和使用响应正文：

- expectBody（Class <T>）：解码为单个对象。
- expectBodyList（Class <T>）：解码并收集List <T>的对象。
- expectBody（）：为JSON内容或空体解码为byte []。

然后你可以为身体使用内置断言。 以下示例显示了一种方法：

```
client.get().uri("/persons")
            .exchange()
            .expectStatus().isOk()
            .expectBodyList(Person.class).hasSize(3).contains(person)
```

你还可以超越内置断言并创建自己的断言，如以下示例所示：

```
 client.get().uri("/persons/1")
            .exchange()
            .expectStatus().isOk()
            .expectBody(Person.class)
            .consumeWith(result -> {
                // custom assertions (e.g. AssertJ)...
            });
```

你也可以退出工作流程并获得结果，如下所示：

```
   EntityExchangeResult<Person> result = client.get().uri("/persons/1")
            .exchange()
            .expectStatus().isOk()
            .expectBody(Person.class)
            .returnResult();
```

> 当你需要使用泛型解码为目标类型时，请查找接受ParameterizedTypeReference而不是Class <T>的重载方法。

更多教程请访问:www.flydean.com

无内容

如果响应没有内容（或者你不关心它），请使用Void.class，以确保释放资源。 以下示例显示了如何执行此操作：

```
client.get().uri("/persons/123")
            .exchange()
            .expectStatus().isNotFound()
            .expectBody(Void.class);
```

或者，如果要断言没有响应内容，可以使用类似于以下内容的代码：

```
client.post().uri("/persons")
            .body(personMono, Person.class)
            .exchange()
            .expectStatus().isCreated()
            .expectBody().isEmpty();
```

JSON Content

使用expectBody（）时，响应将作为byte []使用。 这对原始内容断言很有用。 例如，你可以使用JSONAssert来验证JSON内容，如下所示：

```
client.get().uri("/persons/1")
            .exchange()
            .expectStatus().isOk()
            .expectBody()
            .json("{\"name\":\"Jane\"}")
```

你还可以使用JSONPath表达式，如下所示：

```
client.get().uri("/persons")
            .exchange()
            .expectStatus().isOk()
            .expectBody()
            .jsonPath("$[0].name").isEqualTo("Jane")
            .jsonPath("$[1].name").isEqualTo("Jason");
```

Streaming Responses

要测试无限流（例如，“text / event-stream”或“application / stream + json”），你需要在响应状态和标头断言之后立即退出链式API（通过使用returnResult），如下所示 示例显示：

```
    FluxExchangeResult<MyEvent> result = client.get().uri("/events")
            .accept(TEXT_EVENT_STREAM)
            .exchange()
            .expectStatus().isOk()
            .returnResult(MyEvent.class);
```

现在你可以使用Flux <T>，在它们到来时断言已解码的对象，然后在满足测试目标时取消。 我们建议使用reactor-test模块中的StepVerifier来执行此操作，如以下示例所示：

```
    Flux<Event> eventFux = result.getResponseBody();

    StepVerifier.create(eventFlux)
            .expectNext(person)
            .expectNextCount(4)
            .consumeNextWith(p -> ...)
            .thenCancel()
            .verify();
```

更多教程请访问:www.flydean.com

Request Body

在构建请求时，WebTestClient提供了与WebClient相同的API，实现主要是简单的传递。 有关如何使用正文准备请求的示例，请参阅WebClient文档，包括提交表单数据，多部分请求等。

更多教程请访问:www.flydean.com

4.更多资源

有关测试的更多信息，请参阅以下资源：

- JUnit：“面向程序员的Java测试框架”。由Spring Framework在其测试套件中使用。
- TestNG：受JUnit启发的测试框架，增加了对注解，测试组，数据驱动测试，分布式测试和其他功能的支持。
- AssertJ：“流畅的Java断言”，包括对Java 8 lambdas，流和其他功能的支持。
- 模拟对象：维基百科中的文章。
- MockObjects.com：专门用于模拟对象的网站，一种用于改进测试驱动开发中的代码设计的技术。
- Mockito：基于Test Spy模式的Java模拟库。
- EasyMock：Java库“通过使用Java的代理机制动态生成接口（以及通过类扩展的对象）提供Mock对象。”Spring Framework在其测试套件中使用。
- JMock：支持使用模拟对象进行Java代码的测试驱动开发的库。
- DbUnit：针对数据库驱动项目的JUnit扩展（也可用于Ant和Maven），以及在测试运行之间将数据库置于已知状态。
- Grinder：Java负载测试框架。

更多教程请访问:www.flydean.com

数据访问

这部分参考文档涉及数据访问以及数据访问层与业务或服务层之间的交互。

Spring详细介绍了Spring的全面事务管理支持，然后详细介绍了Spring Framework集成的各种数据访问框架和技术。

更多教程请访问:www.flydean.com

1.事务管理

全面的事务支持是使用Spring Framework的最有说服力的理由之一。 Spring Framework为事务管理提供了一致的抽象，具有以下优势：

- 跨不同事务API的一致编程模型，例如Java Transaction API（JTA），JDBC，Hibernate和Java Persistence API（JPA）。
- 支持声明式事务管理。
- 比复杂事务API（如JTA）更简单的编程事务管理API。
- 与Spring的数据访问抽象集成。

以下部分描述了Spring Framework的事务功能和技术：

- Spring Framework的事务支持模型的优点描述了为什么要使用Spring Framework的事务抽象而不是EJB容器管理事务（CMT）或选择通过专有API（如Hibernate）来驱动本地事务。
- 理解Spring Framework事务抽象概述了核心类，并描述了如何从各种源配置和获取DataSource实例。
- 将资源与事务同步描述了应用程序代码如何确保正确创建，重用和清理资源。
- 声明式事务管理描述了对声明式事务管理的支持。
- 程序化事务管理包括对程序化（即明确编码）事务管理的支持。
- 事务绑定事件描述了如何在事务中使用应用程序事件。

本章还包括对最佳实践，应用程序服务器集成以及常见问题解决方案的讨论。

1.1 Spring Framework的事务支持模型的优点

传统上，Java EE开发人员有两种事务管理选择：全局或本地事务，这两种事务都有很大的局限性。在接下来的两节中将对全局和本地事务管理进行审查，然后讨论Spring Framework的事务管理支持如何解决全局和本地事务模型的局限性。

1.1.1 全局事务

全局事务允许你使用多个事务资源，通常是关系数据库和消息队列。应用程序服务器通过JTA管理全局事务，这是一个繁琐的API（部分原因是它的异常模型）。此外，JTA UserTransaction通常需要从JNDI获取，这意味着你还需要使用JNDI才能使用JTA。全局事务的使用限制了应用程序代码的任何潜在重用，因为JTA通常仅在应用程序服务器环境中可用。

以前，使用全局事务的首选方法是通过EJB CMT（容器管理事务）。 CMT是一种声明式事务管理（与程序化事务管理不同）。 EJB CMT消除了与事务相关的JNDI查找的需要，尽管EJB本身的使用需要使用JNDI。它消除了编写Java代码以控制事务的大部分但不是全部的需要。重要的缺点是CMT与JTA和应用服务器环境相关联。此外，仅当选择在EJB中实现业务逻辑（或至少在事务EJB外观之后）时，它才可用。一般来说，EJB的负面影响是如此之大，以至于这不是一个有吸引力的主张，特别是面对声明式事务管理的令人信服的替代方案。

1.1.2 本地交易

本地事务是特定于资源的，例如与JDBC连接关联的事务。本地事务可能更容易使用，但具有明显的缺点：它们无法跨多个事务资源工作。例如，使用JDBC连接管理事务的代码无法在全局JTA事务中运行。由于应用程序服务器不参与事务管理，因此无法确保跨多个资源的正确性。 （值得注意的是，大多数应用程序使用单个事务资源。）另一个缺点是本地事务对编程模型是侵入性的。

1.1.3 Spring Framework的一致性编程模型

更多教程请访问:www.flydean.com

Spring解决了全局和本地事务的缺点。它允许应用程序开发人员在任何环境中使用一致的编程模型你只需编写一次代码，就可以从不同环境中的不同事务管理策略中受益。 Spring Framework提供了声明式和编程式事务管理。大多数用户更喜欢声明式事务管理，我们建议在大多数情况下使用。

通过编程式事务管理，开发人员可以使用Spring Framework事务抽象，它可以在任何底层事务基础结构上运行。使用首选的声明性模型，开发人员通常很少编写或不编写与事务管理相关的代码，因此不依赖于Spring Framework事务API或任何其他事务API。

> **你是否需要应用程序服务器进行事务管理?**
>
> Spring Framework的事务管理支持改变了关于企业Java应用程序何时需要应用程序服务器的传统规则。
>
> 特别是，你不需要纯粹用于通过EJB进行声明式事务的应用程序服务器。实际上，即使你的应用程序服务器具有强大的JTA功能，你也可以决定Spring Framework的声明式事务提供比EJB CMT更强大的功能和更高效的编程模型。
>
> 通常，只有当你的应用程序需要处理跨多个资源的事务时，才需要应用程序服务器的JTA功能，这对许多应用程序来说并不是必需的。许多高端应用程序使用单个高度可伸缩的数据库（例如Oracle RAC）。独立的事务管理器（例如Atomikos Transactions和JOTM）是其他选项。当然，你可能需要其他应用程序服务器功能，例如Java消息服务（JMS）和Java EE连接器体系结构（JCA）。
>
> Spring Framework使你可以选择何时将应用程序扩展到完全加载的应用程序服务器。使用EJB CMT或JTA的唯一替代方法是使用本地事务（例如JDBC连接上的代码）编写代码，并且如果你需要在全局容器管理的事务中运行代码，则会面临大量的返工。使用Spring Framework，只需要更改配置文件中的一些bean定义（而不是代码）。

1.2 理解Spring Framework事务抽象

Spring事务抽象的关键是事务策略的概念。 事务策略由org.springframework.transaction.PlatformTransactionManager接口定义，如下所示：

```
public interface PlatformTransactionManager {

    TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException;

    void commit(TransactionStatus status) throws TransactionException;

    void rollback(TransactionStatus status) throws TransactionException;
}
```

这主要是服务提供者接口（SPI），尽管你可以从应用程序代码中以编程方式使用它。因为PlatformTransactionManager是一个接口，所以可以根据需要轻松地模拟或存根。它与查找策略无关，例如JNDI。PlatformTransactionManager实现的定义与Spring Framework IoC容器中的任何其他对象（或bean）相同。仅使用此优势使Spring Framework事务成为一种有价值的抽象，即使你使用JTA也是如此。与直接使用JTA相比，你可以更轻松地测试事务代码。

同样，为了与Spring的理念保持一致，可以取消选中任何PlatformTransactionManager接口的方法抛出的TransactionException（即，它扩展了java.lang.RuntimeException类）。交易基础设施故障几乎总是致命的。在极少数情况下，应用程序代码实际上可以从事务失败中恢复，应用程序开发人员仍然可以选择捕获并处理TransactionException。重点是开发人员不会被迫这样做。

getTransaction（..）方法返回TransactionStatus对象，具体取决于TransactionDefinition参数。如果当前调用堆栈中存在匹配的事务，则返回的TransactionStatus可能表示新事务或可以表示现有事务。后一种情况的含义是，与Java EE事务上下文一样，TransactionStatus与执行线程相关联。

TransactionDefinition接口指定：

- Propagation：通常，在事务范围内执行的所有代码都在该事务中运行。但是，如果在事务上下文已存在时执行事

更多教程请访问:www.flydean.com

务方法，则可以指定行为。例如，代码可以继续在现有事务中运行（常见情况），或者可以暂停现有事务并创建新事务。 Spring提供了EJB CMT中熟悉的所有事务传播选项。要阅读有关Spring中事务传播的语义，请参阅事务传播。

- Isolation：此事务与其他事务的工作隔离的程度。例如，此事务是否可以看到来自其他事务的未提交的写入？
- Timeout：此事务在超时之前运行多长时间并由底层事务基础结构自动回滚。
- Read-only status：你可以在代码读取时使用只读事务，但不能修改数据。在某些情况下，只读事务可能是一种有用的优化，例如当你使用Hibernate时。

这些设置反映了标准的事务概念。如有必要，请参阅讨论事务隔离级别和其他核心事务概念的资源。理解这些概念对于使用Spring Framework或任何事务管理解决方案至关重要。

TransactionStatus接口为事务代码提供了一种控制事务执行和查询事务状态的简单方法。这些概念应该是熟悉的，因为它们对于所有事务API都是通用的。以下清单显示了TransactionStatus接口：

```
public interface TransactionStatus extends SavepointManager {

    boolean isNewTransaction();

    boolean hasSavepoint();

    void setRollbackOnly();

    boolean isRollbackOnly();

    void flush();

    boolean isCompleted();

}
```

无论你是在Spring中选择声明式还是程序化事务管理，定义正确的PlatformTransactionManager实现都是绝对必要的。你通常通过依赖注入来定义此实现。

PlatformTransactionManager实现通常需要了解它们工作的环境：JDBC，JTA，Hibernate等。 以下示例显示如何定义本地PlatformTransactionManager实现（在本例中，使用纯JDBC）。

你可以通过创建类似于以下内容的bean来定义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>
```

然后，相关的PlatformTransactionManager bean定义具有对DataSource定义的引用。 它应该类似于以下示例：

```
<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

如果在Java EE容器中使用JTA，则使用通过JNDI获得的容器DataSource以及Spring的JtaTransactionManager。 以下示例显示了JTA和JNDI查找版本的外观：

```
<?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="
```

更多教程请访问:www.flydean.com

```
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/jee
        https://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>
```

JtaTransactionManager不需要知道DataSource（或任何其他特定资源），因为它使用容器的全局事务管理基础结构。

> dataSource bean的前面定义使用jee名称空间中的<jndi-lookup />标记。有关更多信息，请参阅JEE架构。

你还可以轻松使用Hibernate本地事务，如以下示例所示。在这种情况下，你需要定义一个Hibernate LocalSessionFactoryBean，你的应用程序代码可以使用它来获取Hibernate Session实例。

DataSource bean定义类似于前面显示的本地JDBC示例，因此未在以下示例中显示。

> 如果DataSource（由任何非JTA事务管理器使用）通过JNDI查找并由Java EE容器管理，则它应该是非事务性的，因为Spring Framework（而不是Java EE容器）管理事务。

在这种情况下，txManager bean是HibernateTransactionManager类型。与DataSourceTransactionManager需要对DataSource的引用一样，HibernateTransactionManager需要对SessionFactory的引用。以下示例声明了sessionFactory和txManager bean：

```
<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>
```

如果使用Hibernate和Java EE容器管理的JTA事务，则应使用与之前的JDBC JTA示例相同的JtaTransactionManager，如以下示例所示：

```
<bean id="txManager" class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

> 如果使用JTA，则无论使用何种数据访问技术，无论是JDBC，Hibernate JPA还是任何其他支持的技术，你的事务管理器定义应该看起来都一样。 这是因为JTA事务是全局事务，可以登记任何事务资源。

在所有这些情况下，应用程序代码不需要更改。 你可以仅通过更改配置来更改事务的管理方式，即使该更改意味着从本地事务转移到全局事务，反之亦然。

1.3 资源与事务同步

更多教程请访问:www.flydean.com

现在应该清楚如何创建不同的事务管理器以及它们如何链接到需要与事务同步的相关资源（例如，将DataSourceTransactionManager连接到JDBC数据源，HibernateTransactionManager连接到Hibernate SessionFactory等等）。本节描述应用程序代码（直接或间接使用诸如JDBC，Hibernate或JPA之类的持久性API）如何确保正确创建，重用和清理这些资源。本节还讨论了如何（可选）通过相关的PlatformTransactionManager触发事务同步。

1.3.1 高级同步方法

首选方法是使用Spring最高级别的基于模板的持久性集成API，或者将本机ORM API与事务感知工厂bean或代理一起使用，以管理本机资源工厂。这些事务感知解决方案在内部处理资源创建和重用，清理，资源的可选事务同步以及异常映射。因此，用户数据访问代码不必解决这些任务，而是可以完全专注于非样板持久性逻辑。通常，你使用本机ORM API或使用模板方法通过使用JdbcTemplate进行JDBC访问。这些解决方案将在本参考文档的后续章节中详细介绍。

1.3.2。低级同步方法 诸如DataSourceUtils（用于JDBC），EntityManagerFactoryUtils（用于JPA），SessionFactoryUtils（用于Hibernate）等的类存在于较低级别。当你希望应用程序代码直接处理本机持久性API的资源类型时，你可以使用这些类来确保获得正确的Spring Framework托管实例，（可选）同步事务，并且在此过程中发生的异常是正确映射到一致的API。

例如，在JDBC的情况下，你可以改为使用Spring的org.springframework.jdbc.datasource.DataSourceUtils类，而不是传统的JDBC方法来调用DataSource上的getConnection（）方法，如下所示：

```
Connection conn = DataSourceUtils.getConnection (dataSource) ;
```

如果现有事务已经与其同步（链接）了连接，则返回该实例。否则，方法调用将触发新连接的创建，该连接（可选）与任何现有事务同步，并可在随后的同一事务中重用。如前所述，任何SQLException都包含在Spring Framework CannotGetJdbcConnectionException中，Spring Framework是未经检查的DataAccessException类型的层次结构之一。这种方法为你提供了比从SQLException轻松获得的更多信息，并确保跨数据库甚至跨不同持久性技术的可移植性。

这种方法在没有Spring事务管理（事务同步是可选的）的情况下也可以工作，因此无论你是否使用Spring进行事务管理，都可以使用它。

当然，一旦你使用了Spring的JDBC支持，JPA支持或Hibernate支持，你通常不希望使用DataSourceUtils或其他帮助程序类，因为你通过Spring抽象工作比直接使用相关API更快乐。例如，如果你使用Spring JdbcTemplate或jdbc.object包来简化JDBC的使用，则在幕后进行正确的连接检索，你无需编写任何特殊代码。

1.3.3 TransactionAwareDataSourceProxy

在最低级别存在TransactionAwareDataSourceProxy类。这是目标DataSource的代理，它包装目标DataSource以添加对Spring管理的事务的感知。在这方面，它类似于Java EE服务器提供的事务性JNDI数据源。

你几乎从不需要或不想使用此类，除非必须调用现有代码并传递标准JDBC DataSource接口实现。在这种情况下，此代码可能可用但参与Spring管理的事务。你可以使用前面提到的更高级别的抽象来编写新代码。

1.4 声明式事务管理

大多数Spring Framework用户选择声明式事务管理。此选项对应用程序代码的影响最小，因此与非侵入式轻量级容器的理想最为一致。 使用Spring面向方面编程（AOP），Spring Framework的声明式事务管理成为可能。但是，由于事务方面代码随Spring Framework发行版一起提供并且可能以样板方式使用，因此通常不必理解AOP概念以有效使用此代码。

Spring Framework的声明式事务管理类似于EJB CMT，因为你可以将事务行为（或缺少它）指定到单个方法级别。如有必要，你可以在事务上下文中进行setRollbackOnly（）调用。两种类型的事务管理之间的区别是：

更多教程请访问:www.flydean.com

与绑定到JTA的EJB CMT不同，Spring Framework的声明式事务管理适用于任何环境。它可以通过调整配置文件使用JDBC，JPA或Hibernate来处理JTA事务或本地事务。

你可以将Spring Framework声明式事务管理应用于任何类，而不仅仅是EJB等特殊类。

Spring Framework提供了声明性回滚规则，这是一个没有EJB等价的功能。提供了对回滚规则的编程和声明性支持。

Spring Framework允许你使用AOP自定义事务行为。例如，你可以在事务回滚的情况下插入自定义行为。你还可以添加任意建议以及事务建议。使用EJB CMT，除了使用setRollbackOnly（）之外，你无法影响容器的事务管理。

Spring框架不支持跨远程调用传播事务上下文，就像高端应用程序服务器那样。如果你需要此功能，我们建议你使用EJB。但是，在使用此类功能之前请仔细考虑，因为通常情况下，人们不希望事务跨越远程调用。

回滚规则的概念很重要。它们允许你指定哪些异常（和throwable）应该导致自动回滚。你可以在配置中以声明方式指定，而不是在Java代码中。因此，尽管你仍然可以在TransactionStatus对象上调用setRollbackOnly（）来回滚当前事务，但大多数情况下你可以指定MyApplicationException必须始终导致回滚的规则。此选项的显着优势是业务对象不依赖于事务基础结构。例如，它们通常不需要导入Spring事务API或其他Spring API。

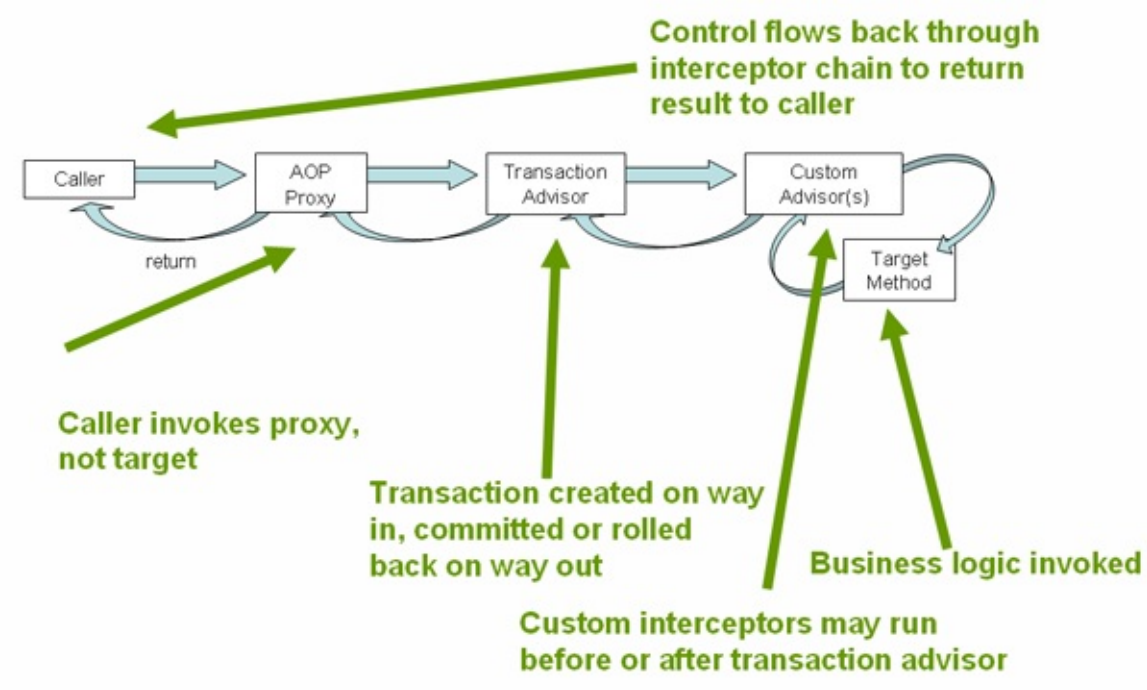
虽然EJB容器默认行为会自动回滚系统异常（通常是运行时异常）上的事务，但EJB CMT不会在应用程序异常（即java.rmi.RemoteException以外的已检查异常）上自动回滚事务。虽然声明式事务管理的Spring默认行为遵循EJB约定（回滚仅在未经检查的异常上自动执行），但定制此行为通常很有用。

1.4.1 理解Spring Framework的声明式事务实现

仅仅通过@Transactional注解告诉你注解你的类是不够的，将@EnableTransactionManagement添加到你的配置中，并期望你了解它是如何工作的。为了提供更深入的理解，本节解释了在发生与事务相关的问题时Spring Framework的声明式事务基础结构的内部工作原理。

关于Spring Framework的声明式事务支持，最重要的概念是通过AOP代理启用此支持，并且事务性建议由元数据（当前基于XML或基于注解）驱动。 AOP与事务元数据的组合产生一个AOP代理，该代理使用TransactionInterceptor和适当的PlatformTransactionManager实现来驱动围绕方法调用的事务。

以下图像显示了在事务代理上调用方法的概念视图：



1.4.2 声明性事务实现的示例

更多教程请访问:www.flydean.com

请考虑以下接口及其附带实现。 此示例使用Foo和Bar类作为占位符，以便你可以专注于事务使用而无需关注特定的域模型。 出于此示例的目的，DefaultFooService类在每个实现的方法的主体中抛出UnsupportedOperationException实例的事实是好的。 通过该行为，你可以查看事务，然后回滚以响应UnsupportedOperationException实例。 以下清单显示了FooService接口：

```
// 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);

}
```

以下示例显示了上述接口的实现：

```
package x.y.service;

public class DefaultFooService implements FooService {

    public Foo getFoo(String fooName) {
        throw new UnsupportedOperationException();
    }

    public Foo getFoo(String fooName, String barName) {
        throw new UnsupportedOperationException();
    }

    public void insertFoo(Foo foo) {
        throw new UnsupportedOperationException();
    }

    public void updateFoo(Foo foo) {
        throw new UnsupportedOperationException();
    }

}
```

假设FooService接口的前两个方法getFoo（String）和getFoo（String，String）必须在具有只读语义的事务的上下文中执行，并且其他方法，insertFoo（Foo）和updateFoo（ Foo），必须在具有读写语义的事务的上下文中执行。 以下配置将在接下来的几段中详细说明：

```
<!-- 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
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/tx
        https://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd">
```

更多教程请访问:www.flydean.com

```
    <!-- this is the service object that we want to make transactional -->
    <bean id="fooService" class="x.y.service.DefaultFooService"/>

    <!-- the transactional advice (what 'happens'; see the <aop:advisor/> bean below) -->
    <tx:advice id="txAdvice" transaction-manager="txManager">
        <!-- the transactional semantics... -->
        <tx:attributes>
            <!-- all methods starting with 'get' are read-only -->
            <tx:method name="get*" read-only="true"/>
            <!-- other methods use the default transaction settings (see below) -->
            <tx:method name="*"/>
        </tx:attributes>
    </tx:advice>

    <!-- ensure that the above transactional advice runs for any execution
        of an operation defined by the FooService interface -->
    <aop:config>
        <aop:pointcut id="fooServiceOperation" expression="execution(* x.y.service.FooService.*(..))"/>
        <aop:advisor advice-ref="txAdvice" pointcut-ref="fooServiceOperation"/>
    </aop:config>

    <!-- don't forget the DataSource -->
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
        <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
        <property name="url" value="jdbc:oracle:thin:@rj-t42:1521:elvis"/>
        <property name="username" value="scott"/>
        <property name="password" value="tiger"/>
    </bean>

    <!-- similarly, don't forget the PlatformTransactionManager -->
    <bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!-- other <bean/> definitions here -->

</beans>
```

检查前面的配置。 它假定你要创建一个服务对象，fooService bean，transactional。 要应用的事务语义封装在<tx：advice />定义中。 <tx：advice />定义读作“所有方法，从get开始，将在只读事务的上下文中执行，所有其他方法将使用默认事务语义执行”。 <tx：advice />标记的transaction-manager属性设置为将驱动事务的PlatformTransactionManager bean的名称（在本例中为txManager bean）。

> 如果要连接的PlatformTransactionManager的bean名称具有名称transactionManager，则可以省略事务建议（<tx：advice />）中的transaction-manager属性。 如果要连接的PlatformTransactionManager bean具有任何其他名称，则必须显式使用transaction-manager属性，如上例所示。

<aop：config />定义确保txAdvice bean定义的事务性建议在程序中的适当位置执行。 首先，定义一个切入点，该切入点与FooService接口（fooServiceOperation）中定义的任何操作的执行相匹配。 然后使用顾问程序将切入点与txAdvice相关联。 结果表明，在执行fooServiceOperation时，将运行txAdvice定义的建议。

<aop：pointcut />元素中定义的表达式是AspectJ切入点表达式。 有关Spring中切入点表达式的更多详细信息，请参阅AOP部分。

常见的要求是使整个服务层具有事务性。 执行此操作的最佳方法是更改切入点表达式以匹配服务层中的任何操作。 以下示例显示了如何执行此操作：

```
<aop:config>
    <aop:pointcut id="fooServiceMethods" expression="execution(* x.y.service.*.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="fooServiceMethods"/>
</aop:config>
```

现在我们已经分析了配置，你可能会问自己，“所有这些配置实际上做了什么？”

更多教程请访问:www.flydean.com

前面显示的配置用于围绕从fooService bean定义创建的对象创建事务代理。 使用事务建议配置代理，以便在代理上调用适当的方法时，根据与该方法关联的事务配置，启动，挂起，标记为只读等事务。 考虑以下测试驱动前面显示的配置的程序：

```
public final class Boot {

    public static void main(final String[] args) throws Exception {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("context.xml", Boot.class);
        FooService fooService = (FooService) ctx.getBean("fooService");
        fooService.insertFoo (new Foo());
    }
}
```

运行前面的程序的输出应类似于以下内容（为清楚起见，LogFJ输出和DefaultFooService类的insertFoo（..）方法抛出的UnsupportedOperationException中的堆栈跟踪已被截断）：

```
<!-- the Spring container is starting up... -->
[AspectJInvocationContextExposingAdvisorAutoProxyCreator] - Creating implicit proxy for bean 'fooService' with 0
 common interceptors and 1 specific interceptors

<!-- the DefaultFooService is actually proxied -->
[JdkDynamicAopProxy] - Creating JDK dynamic proxy for [x.y.service.DefaultFooService]

<!-- ... the insertFoo(..) method is now being invoked on the proxy -->
[TransactionInterceptor] - Getting transaction for x.y.service.FooService.insertFoo

<!-- the transactional advice kicks in here... -->
[DataSourceTransactionManager] - Creating new transaction with name [x.y.service.FooService.insertFoo]
[DataSourceTransactionManager] - Acquired Connection [org.apache.commons.dbcp.PoolableConnection@a53de4] for JD
BC transaction

<!-- the insertFoo(..) method from DefaultFooService throws an exception... -->
[RuleBasedTransactionAttribute] - Applying rules to determine whether transaction should rollback on java.lang.
UnsupportedOperationException
[TransactionInterceptor] - Invoking rollback for transaction on x.y.service.FooService.insertFoo due to throwab
le [java.lang.UnsupportedOperationException]

<!-- and the transaction is rolled back (by default, RuntimeException instances cause rollback) -->
[DataSourceTransactionManager] - Rolling back JDBC transaction on Connection [org.apache.commons.dbcp.PoolableC
onnection@a53de4]
[DataSourceTransactionManager] - Releasing JDBC Connection after transaction
[DataSourceUtils] - Returning JDBC Connection to DataSource

Exception in thread "main" java.lang.UnsupportedOperationException at x.y.service.DefaultFooService.insertFoo(D
efaultFooService.java:14)
<!-- AOP infrastructure stack trace elements removed for clarity -->
at $Proxy0.insertFoo(Unknown Source)
at Boot.main(Boot.java:11)
```

1.4.3 回滚声明性事务

上一节概述了如何在应用程序中以声明方式为类（通常是服务层类）指定事务设置的基础知识。本节介绍如何以简单的声明方式控制事务回滚。

向Spring Framework的事务基础结构指示事务的工作将被回滚的推荐方法是从当前在事务上下文中执行的代码中抛出异常。 Spring Framework的事务基础结构代码捕获任何未处理的异常，因为它冒泡调用堆栈并确定是否将事务标记为回滚。

在其默认配置中，Spring Framework的事务基础结构代码仅在运行时未经检查的异常情况下标记用于回滚的事务。也就是说，抛出的异常是RuntimeException的实例或子类。 （默认情况下，错误实例也会导致回滚）。从事务方法抛出的已检查异常不会导致在默认配置中回滚。

更多教程请访问:www.flydean.com

你可以准确配置哪些Exception类型标记用于回滚的事务，包括已检查的异常。以下XML代码段演示了如何为已检查的特定于应用程序的Exception类型配置回滚：

```
<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>
```

如果你不希望在抛出异常时回滚事务，则还可以指定“无回滚规则”。 以下示例告诉Spring Framework的事务基础结构即使面对未处理的InstrumentNotFoundException也要提交话务员事务：

```
<tx:advice id="txAdvice">
    <tx:attributes>
    <tx:method name="updateStock" no-rollback-for="InstrumentNotFoundException"/>
    <tx:method name="*"/>
    </tx:attributes>
</tx:advice>
```

当Spring Framework的事务基础结构捕获异常并且它查询配置的回滚规则以确定是否将事务标记为回滚时，最强匹配规则获胜。 因此，在以下配置的情况下，除了InstrumentNotFoundException之外的任何异常都会导致后续事务的回滚：

```
<tx:advice id="txAdvice">
    <tx:attributes>
    <tx:method name="*" rollback-for="Throwable" no-rollback-for="InstrumentNotFoundException"/>
    </tx:attributes>
</tx:advice>
```

你还可以以编程方式指示所需的回滚。 虽然很简单，但这个过程非常具有侵入性，并且会将你的代码紧密地耦合到Spring Framework的事务基础结构中。 以下示例显示如何以编程方式指示所需的回滚：

```
public void resolvePosition() {
    try {
        // some business logic...
    } catch (NoProductInStockException ex) {
        // trigger rollback programmatically
        TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();
    }
}
```

如果可能的话，强烈建议你使用声明性方法进行回滚。 如果你绝对需要程序化回滚，则可以使用程序化回滚，但它的使用方式可以实现基于POJO的简洁体系结构。

1.4.4 为不同的Bean配置不同的事务语义

考虑具有多个服务层对象的情况，并且你希望对每个对象应用完全不同的事务配置。 你可以通过使用不同的切入点和advice-ref属性值定义不同的<aop：advisor />元素来实现。

作为比较，首先假设你的所有服务层类都在根x.y.service包中定义。 要使所有作为在该包（或子包中）中定义的类的实例的bean以及以Service结尾的名称具有默认的事务配置，你可以编写以下内容：

```
<?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="
```

更多教程请访问:www.flydean.com

```
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/tx
        https://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/aop
        https://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>
```

以下示例显示如何使用完全不同的事务设置配置两个不同的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:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/tx
        https://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/aop
        https://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"/>
```

更多教程请访问:www.flydean.com

```
    <!-- 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>
```

1.4.5 <tx:advice/>设置

本节总结了你可以使用<tx：advice />标记指定的各种事务设置。 默认的<tx：advice />设置为：

- 传播设置是必需的。
- 隔离级别为DEFAULT。
- 该事务是读写的。
- 事务超时默认为基础事务系统的默认超时，如果不支持超时，则为none。
- 任何RuntimeException都会触发回滚，而任何已检查的Exception都不会。

你可以更改这些默认设置。 下表总结了嵌套在<tx：advice/>和<tx：attributes />标记内的<tx：method />标记的各种属性：

属性	是否必须?	默认值	描述
name	Yes		与事务属性关联的方法名称。 通配符（）字符可用于将相同的事务属性设置与多个方法相关联（例如，*get* ， handle ， *on* Event等）。
propagation	No	REQUIRED	事务传播行为。
isolation	No	DEFAULT	事务隔离级别。 仅适用于REQUIRED或REQUIRES_NEW的传播设置。
timeout	No	-1	事务超时（秒）。 仅适用于传播REQUIRED或REQUIRES_NEW。
read-only	No	false	读写与只读事务。 仅适用于REQUIRED或REQUIRES_NEW。
rollback-for	No		以逗号分隔的触发回滚的异常实例列表。 例如，com.foo.MyBusinessException，ServletException。
no-rollback-for	No		以逗号分隔的不触发回滚的异常实例列表。 例如，com.foo.MyBusinessException，ServletException。

1.4.6 使用@Transactional

除了基于XML的事务配置声明方法之外，你还可以使用基于注解的方法。 直接在Java源代码中声明事务语义会使声明更接近受影响的代码。 不存在过度耦合的危险，因为无论如何，用于事务处理的代码几乎总是以这种方式部署。

> 标准的javax.transaction.Transactional注解也被支持作为Spring自己的注解的替代品。 有关更多详细信息，请参阅JTA 1.2文档。

更多教程请访问:www.flydean.com

使用@Transactional注解所提供的易用性最好通过一个示例来说明，该示例将在后面的文本中进行说明。 考虑以下类定义：

```
// the service class that we want to make transactional
@Transactional
public class DefaultFooService implements FooService {

    Foo getFoo(String fooName);

    Foo getFoo(String fooName, String barName);

    void insertFoo(Foo foo);

    void updateFoo(Foo foo);
}
```

如上所述在类级别使用，注解指示声明类（及其子类）的所有方法的默认值。 或者，每个方法都可以单独注解。 请注意，类级别注解不适用于类层次结构中的祖先类; 在这种情况下，需要在本地重新声明方法才能参与子类级别的注解。

当如上所述的POJO类在Spring上下文中定义为bean时，可以通过@Configuration类中的@EnableTransactionManagement批注使bean实例进行事务处理。 有关详细信息，请参阅javadoc。

在XML配置中，<tx：annotation-driven />标签提供了类似的便利：

```
<!-- 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
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/tx
        https://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/aop
        https://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>
```

> 如果要连接的PlatformTransactionManager的bean名称具有名称transactionManager，则可以省略<tx：annotation-driven />标记中的transaction-manager属性。 如果要依赖注入的PlatformTransactionManager bean具有任何其他名称，则必须使用transaction-manager属性，如上例所示。
>
> **方法可见性和@Transactional**
>
> 使用代理时，应仅将@Transactional注解应用于具有公共可见性的方法。如果使用@Transactional注解对带保护的，私有的或包可见的方法进行注解，则不会引发错误，但带注解的方法不会显示已配置的事务设置。如果需要注解非公共方法，请考虑使用AspectJ（稍后介绍）。

更多教程请访问:www.flydean.com

你可以将@Transactional注解应用于接口定义，接口上的方法，类定义或类上的公共方法。但是，仅仅存在@Transactional注解不足以激活事务行为。 @Transactional注解仅仅是某些运行时基础结构可以使用的元数据，这些运行时基础结构是@Transactional-aware，可以使用元数据来配置具有事务行为的适当bean。在前面的示例中，<tx：annotation-driven />元素打开事务行为。

> Spring团队建议你使用@Transactional注解仅注解具体类（以及具体类的方法），而不是注解接口。你当然可以将@Transactional注解放在接口（或接口方法）上，但这只能在你使用基于接口的代理时按预期工作。 Java注解不是从接口继承的事实意味着，如果使用基于类的代理（proxy-target-class ="true"）或基于编织的方面（mode ="aspectj"），则事务设置不是由代理和编织基础结构识别，并且对象不包含在事务代理中。
>
> 在代理模式（默认设置）下，只拦截通过代理进入的外部方法调用。这意味着自调用（实际上，目标对象中的方法调用目标对象的另一个方法）在运行时不会导致实际的事务，即使调用的方法用@Transactional标记。此外，必须完全初始化代理以提供预期的行为，因此你不应该在初始化代码（即@PostConstruct）中依赖此功能。

如果你希望自我调用也包含在事务中，请考虑使用AspectJ模式（请参阅下表中的mode属性）。在这种情况下，首先没有代理。相反，编织目标类（即，修改其字节代码）以将@Transactional转换为任何类型方法的运行时行为。

XML属性	注解属性	默认值	描述
transaction-manager	N/A（参见TransactionManagementConfigurer javadoc）	transactionManager	要使用的事务管理器的名称。仅当事务管理器的名称不是transactionManag时才需要，如前面的示例所示。
mode	mode	proxy	默认模式（代理）使用Spring的AOP框架处理要注解的注解bean（遵循代理语义，如前所述，仅适用于通过代理进入的方法调用）。替代模式（aspectj）使用Spring的AspectJ事务方面编译受影响的类，修改目标类字节代码以应用于任何类型的方法调用。 AspectJ编织要类路径中的spring-aspects.jar以及启用加载时编织（或编译时编织）。（有关如何设置加载时编织的详细信息，请参阅Spring配置。）
proxy-target-class	proxyTargetClass	false	仅适用于代理模式。控制为使用@Transactional注解注解的类创建的事务代理类型。如果proxy-target-class属性设置为true，则创建基于的代理。如果proxy target-class为false或者省略了该属性，则会创建基于标准JDK接口的代理。（有关不同代

更多教程请访问:www.flydean.com

			理类型的详细检查，请参阅代理机制。）
order	order	Ordered.LOWEST_PRECEDENCE	定义应用于使用@Transactional注解的bean的事务通知的顺序。（有关与AOP建议排序相关的规则的更多信息，请参阅建议排序。）没有指定的排序意味着AOP子系统确定建议的顺序。

> 处理@Transactional注解的默认建议模式是proxy，它允许仅通过代理拦截调用。同一类中的本地调用不能以这种方式截获。对于更高级的拦截模式，请考虑结合编译时或加载时编织切换到aspectj模式。
>
> proxy-target-class属性控制为使用@Transactional注解注解的类创建的事务代理类型。如果proxy-target-class设置为true，则创建基于类的代理。如果proxy-target-class为false或者省略了该属性，则会创建基于标准JDK接口的代理。（有关不同代理类型的讨论，请参阅[aop-proxying]。）
>
> @EnableTransactionManagement和<tx：annotation-driven />仅在定义它们的同一应用程序上下文中的bean上查找@Transactional。这意味着，如果你将注解驱动的配置放在DispatcherServlet的WebApplicationContext中，它只会在你的控制器而不是你的服务中检查@Transactional bean。有关更多信息，请参阅MVC。

在评估方法的事务设置时，派生最多的位置优先。在以下示例的情况下，DefaultFooService类在类级别使用只读事务的设置进行注解，但同一类中updateFoo（Foo）方法的@Transactional注解优先于定义的事务设置在class级别。

```
@Transactional(readOnly = true)
public class DefaultFooService implements FooService {

    public Foo getFoo(String fooName) {
        // do something
    }

    // these settings have precedence for this method
    @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
    public void updateFoo(Foo foo) {
        // do something
    }
}
```

@Transactional设置

@Transactional注解是指定接口，类或方法必须具有事务语义的元数据（例如，“在调用此方法时启动全新的只读事务，暂停任何现有事务”）。 默认的@Transactional设置如下：

- 传播设置为PROPAGATION_REQUIRED。
- 隔离级别为ISOLATION_DEFAULT。
- 该事务是读写的。
- 事务超时默认为基础事务系统的默认超时，如果不支持超时，则默认为none。
- 任何RuntimeException都会触发回滚，而任何已检查的Exception都不会。

你可以更改这些默认设置。 下表总结了@Transactional注解的各种属性：

Property	Type	Description
value	String	可选限定符，指定要使用的事务管理器。
propagation	enum: Propagation	可选的传播设置。

更多教程请访问:www.flydean.com

isolation	enum: Isolation	可选的隔离级别。 仅适用于REQUIRED或REQUIRES_NEW的传播值。
timeout	int（以秒为单位）	可选的事务超时。 仅适用于REQUIRED或REQUIRES_NEW的传播值。
readOnly	boolean	读写与只读事务。 仅适用于REQUIRED或REQUIRES_NEW的值。
rollbackFor	Class对象的数组，必须从Throwable派生。	必须引发回滚的可选异常类数组。
rollbackForClassName	类名数组。 这些类必须从Throwable派生。	必须导致回滚的可选异常类名称数组。
noRollbackFor	Class对象的数组，必须从Throwable派生。	可选的异常类数组，不得导致回滚。
noRollbackForClassName	String类名称的数组，必须从Throwable派生。	可选不会导致回滚的异常类名称的可选数组。

目前，你无法明确控制事务的名称，其中“name”表示事务监视器中显示的事务名称（如果适用）（例如，WebLogic的事务监视器）和日志记录输出。对于声明性事务，事务名称始终是完全限定的类名+。 +事务建议类的方法名称。例如，如果BusinessService类的handlePayment（..）方法启动了事务，则事务的名称将为：com.example.BusinessService.handlePayment。

使用@Transactional的多个Transaction Managers

大多数Spring应用程序只需要一个事务管理器，但是在某些应用程序中可能需要多个独立的事务管理器。你可以使用@Transactional批注的value属性来选择性地指定要使用的PlatformTransactionManager的标识。这可以是bean名称或事务管理器bean的限定符值。例如，使用限定符表示法，你可以将以下Java代码与应用程序上下文中的以下事务管理器bean声明组合使用：

```
public class TransactionalService {

    @Transactional("order")
    public void setSomething(String name) { ... }

    @Transactional("account")
    public void doSomething() { ... }
}
```

下面是bean定义：

```
<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>
```

在这种情况下，TransactionalService上的两个方法在单独的事务管理器下运行，由顺序和帐户限定符区分。 如果未找到特定限定的PlatformTransactionManager bean，则仍会使用默认的<tx：annotation-driven>目标bean名称transactionManager。

自定义快捷方式注解

更多教程请访问:www.flydean.com

如果你发现在许多不同方法上重复使用与@Transactional相同的属性，Spring的元注解支持允许你为特定用例定义自定义快捷方式注解。 例如，请考虑以下注解定义：

```
@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 {
}
```

前面的注解允许我们编写上一节中的示例，如下所示：

```
public class TransactionalService {

    @OrderTx
    public void setSomething(String name) { ... }

    @AccountTx
    public void doSomething() { ... }
}
```

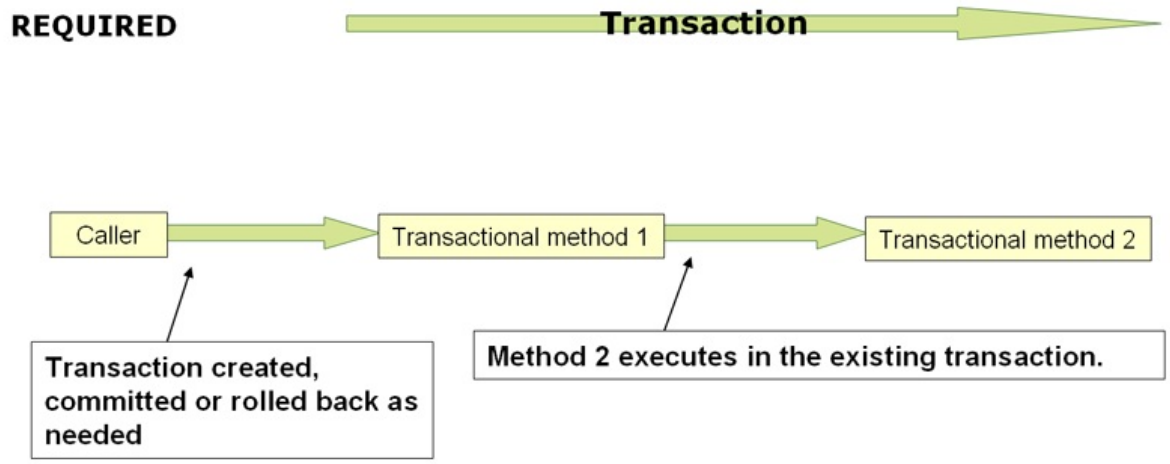
在前面的示例中，我们使用语法来定义事务管理器限定符，但我们也可以包含传播行为，回滚规则，超时和其他功能。

1.4.7 交易传播

本节描述了Spring中事务传播的一些语义。 请注意，本节不是适当的事务传播介绍。 相反，它详细介绍了Spring中有关事务传播的一些语义。

在Spring管理的事务中，请注意物理和逻辑事务之间的区别，以及传播设置如何应用于此差异。

理解 PROPAGATION_REQUIRED



PROPAGATION_REQUIRED在当前范围内（如果尚未存在事务）或参与为更大范围定义的现有“外部”事务，则在本地执行物理事务。这是同一线程中常见调用堆栈安排的一个很好的默认值（例如，委托给几个存储库方法的服务外观，其中所有底层资源都必须参与服务级别事务）。

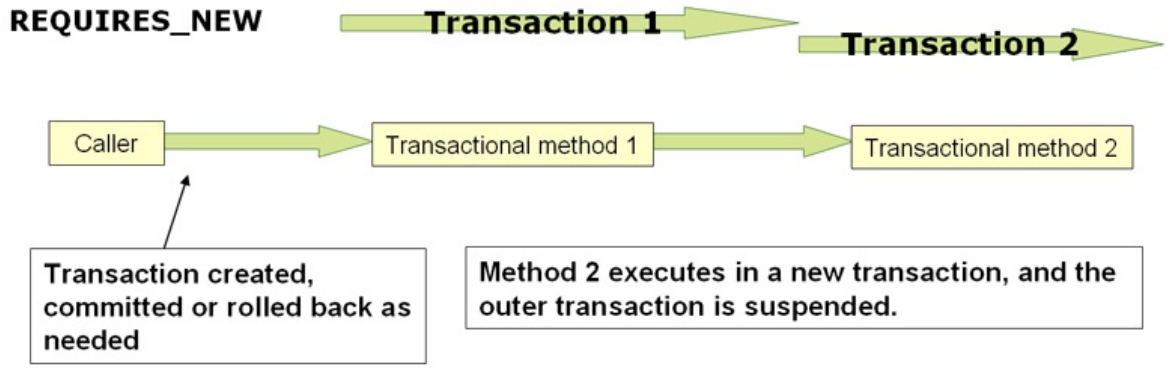
更多教程请访问:www.flydean.com

> 默认情况下，参与事务加入外部作用域的特征，静默忽略本地隔离级别，超时值或只读标志（如果有）。如果希望在参与具有不同隔离级别的现有事务时拒绝隔离级别声明，请考虑在事务管理器上将validateExistingTransactions标志切换为true。此非宽松模式还拒绝只读不匹配（即，尝试参与只读外部作用域的内部读写事务）。

当传播设置为PROPAGATION_REQUIRED时，将为应用该设置的每个方法创建逻辑事务范围。每个这样的逻辑事务范围可以单独确定仅回滚状态，外部事务范围在逻辑上独立于内部事务范围。在标准PROPAGATION_REQUIRED行为的情况下，所有这些范围都映射到同一物理事务。因此，内部事务范围中的仅回滚标记集确实会影响外部事务实际提交的机会。

但是，在内部事务作用域设置仅回滚标记的情况下，外部事务尚未决定回滚本身，因此回滚（由内部事务作用域静默触发）是意外的。此时抛出相应的UnexpectedRollbackException。这是预期的行为，因此事务的调用者永远不会被误导以假定在实际上没有执行提交。因此，如果内部事务（外部调用者不知道）默认将事务标记为仅回滚，则外部调用者仍调用commit。外部调用者需要接收UnexpectedRollbackException以清楚地指示已执行回滚。

了解PROPAGATION_REQUIRES_NEW



与PROPAGATION_REQUIRED相反，PROPAGATION_REQUIRES_NEW始终对每个受影响的事务范围使用独立的物理事务，从不参与外部范围的现有事务。在这样的安排中，底层资源事务是不同的，因此可以独立地提交或回滚，外部事务不受内部事务的回滚状态的影响，并且内部事务的锁在完成后立即释放。这样一个独立的内部事务也可以声明它自己的隔离级别，超时和只读设置，而不是继承外部事务的特性。

了解PROPAGATION_NESTED

PROPAGATION_NESTED使用具有多个保存点的单个物理事务，它可以回滚到该事务。这种部分回滚允许内部事务作用域触发其作用域的回滚，外部事务能够继续物理事务，尽管已经回滚了一些操作。此设置通常映射到JDBC保存点，因此它仅适用于JDBC资源事务。请参阅Spring的DataSourceTransactionManager。

1.4.8 为交易操作提供Advising

假设你要执行事务操作和一些基本概要分析建议。 你如何在<tx：annotation-driven />的背景下实现这一点?

当你调用updateFoo（Foo）方法时，你希望看到以下操作：

- 配置的性能分析方面开始。
- 交易建议执行。
- 建议对象上的方法执行。
- 交易提交。
- 概要分析方面报告整个事务方法调用的确切持续时间。

> 本章不涉及详细解释AOP（除非适用于交易）。 有关AOP配置和AOP的详细信息，请参阅AOP。

以下代码显示了前面讨论的简单分析方面：

更多教程请访问:www.flydean.com

```
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;
    }
}
```

通过Ordered接口控制建议的顺序。 有关建议订购的完整详细信息，请参阅建议订购。

以下配置创建一个fooService bean，该bean以所需顺序应用了分析和事务方面：

```
<?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
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/tx
        https://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/aop
        https://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"/>
```

更多教程请访问:www.flydean.com

```
            </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>
```

你可以以类似的方式配置任意数量的其他方面。

以下示例创建与前两个示例相同的设置，但使用纯XML声明方法：

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

    <bean id="fooService" class="x.y.service.DefaultFooService"/>

    <!-- the profiling advice -->
    <bean id="profiler" class="x.y.SimpleProfiler">
        <!-- execute before the transactional advice (hence the lower order number) -->
        <property name="order" value="1"/>
    </bean>

    <aop:config>
        <aop:pointcut id="entryPointMethod" expression="execution(* x.y..*Service.*(..))"/>
        <!-- will execute after the profiling advice (c.f. the order attribute) -->

        <aop:advisor advice-ref="txAdvice" pointcut-ref="entryPointMethod" order="2"/>
        <!-- order value is higher than the profiling aspect -->

        <aop:aspect id="profilingAspect" ref="profiler">
            <aop:pointcut id="serviceMethodWithReturnValue"
                    expression="execution(!void x.y..*Service.*(..))"/>
            <aop:around method="profile" pointcut-ref="serviceMethodWithReturnValue"/>
        </aop:aspect>

    </aop:config>

    <tx:advice id="txAdvice" transaction-manager="txManager">
        <tx:attributes>
            <tx:method name="get*" read-only="true"/>
            <tx:method name="*"/>
        </tx:attributes>
    </tx:advice>

    <!-- other <bean/> definitions such as a DataSource and a PlatformTransactionManager here -->

</beans>
```

更多教程请访问:www.flydean.com

前面配置的结果是一个fooService bean，它具有按顺序应用于它的分析和事务方面。如果你希望在关于出路的事务性建议之前和之前的事务性建议之后执行性能分析建议，则可以交换性能分析方面bean的order属性的值，使其高于事务性建议的顺序值。

你可以以类似的方式配置其他方面。

1.4.9 将@Transactional与AspectJ一起使用

你还可以通过AspectJ方面在Spring容器外部使用Spring Framework的@Transactional支持。为此，首先使用@Transactional注解注解你的类（以及可选的类的方法），然后使用spring-aspects.jar文件中定义的org.springframework.transaction.aspectj.AnnotationTransactionAspect链接（编织）你的应用程序。 。你还必须使用事务管理器配置方面。你可以使用Spring Framework的IoC容器来处理依赖注入方面。配置事务管理方面的最简单方法是使用<tx：annotation-driven />元素并将aspect属性指定为aspectj，如使用@Transactional中所述。因为我们专注于在Spring容器之外运行的应用程序，所以我们将向你展示如何以编程方式执行此操作。

> 在继续之前，你可能希望分别阅读使用@Transactional和AOP。

以下示例显示如何创建事务管理器并配置AnnotationTransactionAspect以使用它：

```
// construct an appropriate transaction manager
DataSourceTransactionManager txManager = new DataSourceTransactionManager(getDataSource());

// configure the AnnotationTransactionAspect to use it; this must be done before executing any transactional me
thods
AnnotationTransactionAspect.aspectOf().setTransactionManager(txManager);
```

使用此方面时，必须注解实现类（或该类中的方法或两者），而不是类实现的接口（如果有）。 AspectJ遵循Java的规则，即接口上的注解不会被继承。 类上的@Transactional注解指定了在类中执行任何公共方法的默认事务语义。

类中方法的@Transactional注解会覆盖类注解（如果存在）给出的默认事务语义。 无论可见性如何，你都可以注解任何方法。

要使用AnnotationTransactionAspect编写应用程序，必须使用AspectJ构建应用程序（请参阅AspectJ开发指南）或使用加载时编织。 有关使用AspectJ进行加载时编织的讨论，请参阅Spring Framework中使用AspectJ进行加载时编织。

1.5 程序化交易管理

Spring Framework提供了两种程序化事务管理方法，使用：

- TransactionTemplate。
- 直接使用PlatformTransactionManager。

Spring团队通常建议使用TransactionTemplate进行程序化事务管理。第二种方法类似于使用JTA UserTransaction API，尽管异常处理不那么麻烦。

1.5.1 使用TransactionTemplate

TransactionTemplate采用与其他Spring模板相同的方法，例如JdbcTemplate。它使用回调方法（使应用程序代码免于必须进行样板采集和释放事务资源）并产生意图驱动的代码，因为你的代码仅关注你想要做的事情。

> 如下面的示例所示，使用TransactionTemplate绝对将你与Spring的事务基础结构和API结合在一起。程序化事务管理是否适合你的开发需求是你必须自己做出的决定。

更多教程请访问:www.flydean.com

必须在事务上下文中执行并且显式使用TransactionTemplate的应用程序代码类似于下一个示例。作为应用程序开发人员，你可以编写一个TransactionCallback实现（通常表示为匿名内部类），该实现包含你需要在事务上下文中执行的代码。然后，你可以将自定义TransactionCallback的实例传递给TransactionTemplate上公开的execute（..）方法。以下示例显示了如何执行此操作：

```
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) {
        this.transactionTemplate = new TransactionTemplate(transactionManager);
    }

    public Object someServiceMethod() {
        return transactionTemplate.execute(new TransactionCallback() {
            // the code in this method executes in a transactional context
            public Object doInTransaction(TransactionStatus status) {
                updateOperation1();
                return resultOfUpdateOperation2();
            }
        });
    }
}
```

如果没有返回值，则可以使用方便的TransactionCallbackWithoutResult类和匿名类，如下所示：

```
transactionTemplate.execute(new TransactionCallbackWithoutResult() {
    protected void doInTransactionWithoutResult(TransactionStatus status) {
        updateOperation1();
        updateOperation2();
    }
});
```

回调中的代码可以通过调用提供的TransactionStatus对象上的setRollbackOnly（）方法来回滚事务，如下所示：

```
transactionTemplate.execute(new TransactionCallbackWithoutResult() {

    protected void doInTransactionWithoutResult(TransactionStatus status) {
        try {
            updateOperation1();
            updateOperation2();
        } catch (SomeBusinessException ex) {
            status.setRollbackOnly();
        }
    }
});
```

指定交易设置

你可以通过编程方式或在配置中在TransactionTemplate上指定事务设置（例如传播模式，隔离级别，超时等）。 默认情况下，TransactionTemplate实例具有默认的事务设置。 以下示例显示了特定TransactionTemplate的事务设置的编程自定义：

```
public class SimpleService implements Service {

    private final TransactionTemplate transactionTemplate;

    public SimpleService(PlatformTransactionManager transactionManager) {
        this.transactionTemplate = new TransactionTemplate(transactionManager);
```

更多教程请访问:www.flydean.com

```
        // the transaction settings can be set here explicitly if so desired
        this.transactionTemplate.setIsolationLevel(TransactionDefinition.ISOLATION_READ_UNCOMMITTED);
        this.transactionTemplate.setTimeout(30); // 30 seconds
        // and so forth...
    }
}
```

以下示例使用Spring XML配置定义具有一些自定义事务设置的TransactionTemplate：

```
<bean id="sharedTransactionTemplate"
        class="org.springframework.transaction.support.TransactionTemplate">
    <property name="isolationLevelName" value="ISOLATION_READ_UNCOMMITTED"/>
    <property name="timeout" value="30"/>
</bean>"
```

然后，你可以将sharedTransactionTemplate注入所需数量的服务。

最后，TransactionTemplate类的实例是线程安全的，因为实例不保持任何会话状态。 但是，TransactionTemplate实例会保持配置状态。 因此，虽然许多类可以共享TransactionTemplate的单个实例，但如果类需要使用具有不同设置的TransactionTemplate（例如，不同的隔离级别），则需要创建两个不同的TransactionTemplate实例。

1.5.2 使用PlatformTransactionManager

你也可以直接使用org.springframework.transaction.PlatformTransactionManager来管理你的交易。 为此，请通过bean引用将你使用的PlatformTransactionManager的实现传递给bean。 然后，通过使用TransactionDefinition和TransactionStatus对象，你可以启动事务，回滚和提交。 以下示例显示了如何执行此操作：

```
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
// explicitly setting the transaction name is something that can be done only 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);
```

1.6 在程序化和声明式事务管理之间进行选择

只有在你进行少量事务操作时，程序化事务管理通常才是一个好主意。例如，如果你的Web应用程序仅需要针对某些更新操作进行事务处理，则可能不希望使用Spring或任何其他技术来设置事务代理。在这种情况下，使用TransactionTemplate可能是一个很好的方法。能够明确设置事务名称也是可以通过使用编程方法进行事务管理来完成的。

另一方面，如果你的应用程序有许多事务操作，则声明式事务管理通常是值得的。它使事务管理不受业务逻辑的影响，并且不难配置。使用Spring Framework而不是EJB CMT时，声明式事务管理的配置成本大大降低。

1.7 交易约束事件

从Spring 4.2开始，事件的监听器可以绑定到事务的一个阶段。典型示例是在事务成功完成时处理事件。这样做可以在当前事务的结果对于监听器实际上很重要时更灵活地使用事件。

更多教程请访问:www.flydean.com

你可以使用@EventListener注解注册常规事件侦听器。如果需要将其绑定到事务，请使用@TransactionalEventListener。执行此操作时，默认情况下，侦听器将绑定到事务的提交阶段。

下一个例子显示了这个概念。假设一个组件发布一个订单创建的事件，并且我们想要定义一个只应该在发布它的事务成功提交后才应该处理该事件的监听器。以下示例设置此类事件侦听器：

```
@Component
public class MyComponent {

    @TransactionalEventListener
    public void handleOrderCreatedEvent(CreationEvent<Order> creationEvent) {
        ...
    }
}
```

@TransactionalEventListener注解公开了一个阶段属性，该属性允许你自定义侦听器应绑定到的事务的阶段。有效阶段是BEFORE_COMMIT，AFTER_COMMIT（默认值），AFTER_ROLLBACK和AFTER_COMPLETION，它们聚合事务完成（无论是提交还是回滚）。

如果没有运行事务，则根本不调用侦听器，因为我们无法遵守所需的语义。但是，你可以通过将注解的fallbackExecution属性设置为true来覆盖该行为。

1.8 特定于应用程序服务器的集

Spring的事务抽象通常与应用程序服务器无关。此外，Spring的JtaTransactionManager类（可以选择对JTA UserTransaction和TransactionManager对象执行JNDI查找）自动检测后一个对象的位置，该位置因应用程序服务器而异。有权访问JTA TransactionManager允许增强的事务语义 - 特别是支持事务暂停。有关详细信息，请参阅JtaTransactionManager javadoc。

Spring的JtaTransactionManager是在Java EE应用程序服务器上运行的标准选择，并且已知可在所有常见服务器上运行。高级功能（例如事务暂停）也适用于许多服务器（包括GlassFish，JBoss和Geronimo），无需任何特殊配置。但是，对于完全支持的事务挂起和进一步的高级集成，Spring包含用于WebLogic Server和WebSphere的特殊适配器。以下各节将讨论这些适配器。

对于标准方案（包括WebLogic Server和WebSphere），请考虑使用方便的<tx：jta-transaction-manager />配置元素。配置后，此元素会自动检测基础服务器并选择可用于平台的最佳事务管理器。这意味着你无需显式配置特定于服务器的适配器类（如以下部分所述）。相反，它们是自动选择的，标准的JtaTransactionManager是默认的后备。

1.8.1 IBM WebSphere

在WebSphere 6.1.0.9及更高版本中，要使用的推荐Spring JTA事务管理器是WebSphereUowTransactionManager。此特殊适配器使用IBM的UOWManager API，该API在WebSphere Application Server 6.1.0.9及更高版本中可用。使用此适配器，IBM正式支持Spring驱动的事务暂停（由PROPAGATION_REQUIRES_NEW启动的暂停和恢复）。

1.8.2 Oracle WebLogic Server

在WebLogic Server 9.0或更高版本上，通常使用WebLogicJtaTransactionManager而不是库存JtaTransactionManager类。普通JtaTransactionManager的这个特殊的WebLogic特定子类在WebLogic管理的事务环境中支持Spring的事务定义的全部功能，超出了标准的JTA语义。功能包括事务名称，每事务隔离级别以及在所有情况下正确恢复事务。

1.9 解决常见问题

本节介绍一些常见问题的解决方案。

更多教程请访问:www.flydean.com

1.9.1 使用错误的事务管理器获取特定的数据源

根据你选择的事务技术和要求，使用正确的PlatformTransactionManager实现。如果使用得当，Spring Framework只提供了简单易用的抽象。如果使用全局事务，则必须对所有事务操作使用org.springframework.transaction.jta.JtaTransactionManager类（或其特定于应用程序服务器的子类）。否则，事务基础结构尝试在诸如容器DataSource实例之类的资源上执行本地事务。这样的本地事务没有意义，一个好的应用程序服务器将它们视为错误。

1.10 更多资源

有关Spring Framework的事务支持的更多信息，请参阅：

- Spring中的分布式事务，有和没有XA是一个JavaWorld演示文稿，其中Spring的David Syer引导你完成Spring应用程序中分布式事务的七种模式，其中三种模式使用XA，另外四种没有。
- Java Transaction Design Strategies是InfoQ提供的一本书，它提供了对Java中事务的快速介绍。它还包括如何使用Spring Framework和EJB3配置和使用事务的并排示例。

更多教程请访问:www.flydean.com

2.DAO支持

Spring对数据访问对象（DAO）的支持旨在使以一致的方式轻松使用数据访问技术（例如JDBC，Hibernate或JPA）。这使你可以轻松地在上述持久性技术之间进行切换，还使你无需担心捕获每种技术特有的异常即可进行编码。

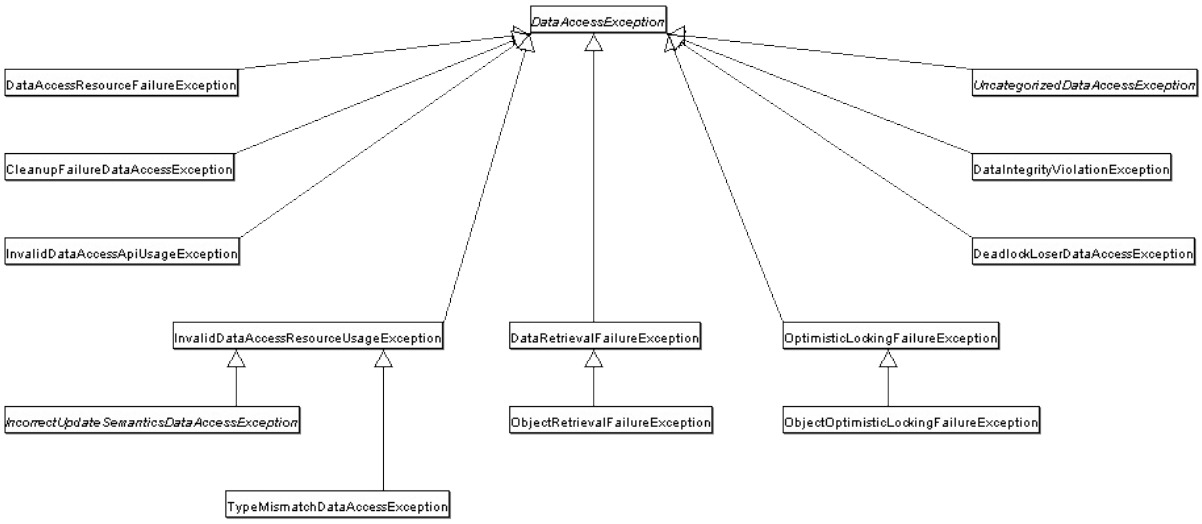
2.1 一致的异常层次结构

Spring提供了从特定于技术的异常（例如SQLException）到其自己的异常类层次结构的便捷转换，该异常类层次结构以DataAccessException作为根异常。这些异常包装了原始异常，因此你永远不会丢失任何可能出错的信息。

除了JDBC异常，Spring还可以包装JPA和Hibernate特定的异常，将它们转换为一组集中的运行时异常。这样，你就可以仅在适当的层中处理大多数不可恢复的持久性异常，而无需在DAO中使用烦人的样板捕获和抛出块以及异常声明。（尽管你仍然可以在任何需要的地方捕获和处理异常。）如上所述，JDBC异常（包括特定于数据库的方言）也被转换为相同的层次结构，这意味着你可以在一致的编程模型中对JDBC执行某些操作。 。

在Spring对各种ORM框架的支持中，上述讨论对于各种模板类均适用。如果使用基于拦截器的类，则应用程序必须关心如何处理HibernateExceptions和PersistenceExceptions本身，最好分别委派给SessionFactoryUtils的convertHibernateAccessException（..）或convertJpaAccessException（）方法。这些方法将异常转换为与org.springframework.dao异常层次结构中的异常兼容的异常。由于未选中PersistenceException，因此也可能引发它们（不过，在异常方面牺牲了通用DAO抽象）。

下图显示了Spring提供的异常层次结构。 （请注意，图像中详细描述的类层次结构仅显示整个DataAccessException层次结构的子集。）



2.2 用于配置DAO或存储库类的注解

保证数据访问对象（DAO）或存储库提供异常转换的最佳方法是使用@Repository批注。 此批注还允许组件扫描支持查找和配置DAO和存储库，而无需为它们提供XML配置条目。 以下示例显示了如何使用@Repository批注：

```
@Repository
public class SomeMovieFinder implements MovieFinder {
    // ...
}
```

更多教程请访问:www.flydean.com

任何DAO或存储库实现都需要访问持久性资源，具体取决于所使用的持久性技术。 例如，基于JDBC的存储库需要访问JDBC DataSource，而基于JPA的存储库需要访问EntityManager。 完成此操作的最简单方法是使用@Autowired，@ Inject，@ Resource或@PersistenceContext注解之一注入此资源依赖项。 以下示例适用于JPA存储库：

```
@Repository
public class JpaMovieFinder implements MovieFinder {

    @PersistenceContext
    private EntityManager entityManager;

    // ...

}
```

如果你使用经典的Hibernate API，则可以注入SessionFactory，如以下示例所示：

```
@Repository
public class HibernateMovieFinder implements MovieFinder {

    private SessionFactory sessionFactory;

    @Autowired
    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    // ...

}
```

我们在这里展示的最后一个例子是典型的JDBC支持。 你可以将DataSource注入到初始化方法中，你可以使用此DataSource在其中创建JdbcTemplate和其他数据访问支持类（如SimpleJdbcCall等）。 以下示例自动装配DataSource：

```
@Repository
public class JdbcMovieFinder implements MovieFinder {

    private JdbcTemplate jdbcTemplate;

    @Autowired
    public void init(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    // ...

}
```

更多教程请访问:www.flydean.com

3.JDBC

Spring Framework JDBC抽象提供的值可能最好通过下表中列出的操作序列来显示。 该表显示了Spring负责的操作以及你负责的操作。

Action	Spring	You
Define connection parameters.		x
Open the connection.	x	
Specify the SQL statement.		x
Declare parameters and provide parameter values		x
Prepare and execute the statement.	x	
Set up the loop to iterate through the results (if any).	x	
Do the work for each iteration.		x
Process any exception.	x	
Handle transactions.	x	
Close the connection, the statement, and the resultset.	x	

Spring框架负责所有可能使JDBC成为乏味的API的低级细节。

更多教程请访问:www.flydean.com

3.1选择JDBC数据库访问方法

你可以选择几种方法来构成JDBC数据库访问的基础。除了JdbcTemplate的三种形式之外，新的SimpleJdbcInsert和SimpleJdbcCall方法还优化了数据库元数据，并且RDBMS Object样式采用了一种类似于JDO Query设计的面向对象的方法。一旦开始使用这些方法之一，你仍然可以混合搭配以包含来自其他方法的功能。所有方法都需要JDBC 2.0兼容的驱动程序，而某些高级功能需要JDBC 3.0驱动程序。

JdbcTemplate是经典且最流行的Spring JDBC方法。这种“最低级别”的方法以及其他所有方法都在后台使用了JdbcTemplate。

NamedParameterJdbcTemplate包装JdbcTemplate以提供命名参数而不是传统的JDBC？占位符。当你有多个SQL语句参数时，此方法可提供更好的文档编制和易用性。

SimpleJdbcInsert和SimpleJdbcCall优化数据库元数据以限制必要的配置量。这种方法简化了编码，因此你只需要提供表或过程的名称，并提供与列名称匹配的参数映射即可。仅当数据库提供足够的元数据时，此方法才有效。如果数据库不提供此元数据，则必须提供参数的显式配置。

RDBMS对象（包括MappingSqlQuery，SqlUpdate和StoredProcedure）要求你在数据访问层初始化期间创建可重用且线程安全的对象。此方法以JDO Query为模型，其中你定义查询字符串，声明参数并编译查询。完成后，可以使用各种参数值多次调用execute方法。

更多教程请访问:www.flydean.com

3.2包层次结构

Spring框架的JDBC抽象框架由四个不同的软件包组成：

- core：org.springframework.jdbc.core软件包包含JdbcTemplate类及其各种回调接口，以及各种相关类。名为org.springframework.jdbc.core.simple的子程序包包含SimpleJdbcInsert和SimpleJdbcCall类。另一个名为org.springframework.jdbc.core.namedparam的子程序包包含NamedParameterJdbcTemplate类和相关的支持类。请参阅使用JDBC核心类控制基本JDBC处理和错误处理，JDBC批处理操作和使用SimpleJdbc类简化JDBC操作。
- datasource：org.springframework.jdbc.datasource软件包包含一个实用程序类，用于轻松访问DataSource和各种简单DataSource实现，可用于在Java EE容器之外测试和运行未修改的JDBC代码。名为org.springfamework.jdbc.datasource.embedded的子包为使用Java数据库引擎（例如HSQL，H2和Derby）创建嵌入式数据库提供了支持。请参阅控制数据库连接和嵌入式数据库支持。
- object：org.springframework.jdbc.object包包含一些类，这些类将RDBMS查询，更新和存储过程表示为线程安全的可重用对象。请参阅将JDBC操作建模为Java对象。尽管查询返回的对象自然会与数据库断开连接，但是JDO对此方法进行了建模。较高级别的JDBC抽象取决于org.springframework.jdbc.core软件包中的较低级别的抽象。
- support：org.springframework.jdbc.support软件包提供SQLException转换功能和一些实用程序类。 JDBC处理期间引发的异常将转换为org.springframework.dao包中定义的异常。这意味着使用Spring JDBC抽象层的代码不需要实现JDBC或RDBMS特定的错误处理。所有翻译的异常均未选中，这使你可以选择捕获可以从中恢复的异常，同时将其他异常传播到调用方。请参见使用SQLExceptionTranslator。

更多教程请访问:www.flydean.com

3.3使用JDBC核心类控制基本JDBC处理和错误处理

本节介绍如何使用JDBC核心类来控制基本的JDBC处理，包括错误处理。它包括以下主题：

- 使用JdbcTemplate
- 使用NamedParameterJdbcTemplate
- 使用SQLExceptionTranslator
- 运行声明
- 运行查询
- 更新数据库
- 检索自动生成的密钥

3.3.1 使用JdbcTemplate

JdbcTemplate是JDBC核心软件包中的中心类。它处理资源的创建和释放，这有助于你避免常见的错误，例如忘记关闭连接。它执行核心JDBC工作流程的基本任务（例如，语句创建和执行），而使应用程序代码提供SQL并提取结果。JdbcTemplate类：

- 运行SQL查询
- 更新语句和存储过程调用
- 对ResultSet实例执行迭代并提取返回的参数值。
- 捕获JDBC异常并将其转换为org.springframework.dao包中定义的通用，信息量更大的异常层次结构。 （请参见一致的异常层次结构。）

当将JdbcTemplate用于代码时，只需实现回调接口，即可为它们提供明确定义的协定。给定JdbcTemplate类提供的Connection，PreparedStatementCreator回调接口创建一个准备好的语句，提供SQL和任何必要的参数。创建可调用语句的CallableStatementCreator接口也是如此。 RowCallbackHandler接口从ResultSet的每一行提取值。

你可以通过直接实例化DataSource引用在DAO实现中使用JdbcTemplate，也可以在Spring IoC容器中对其进行配置，并将其作为Bean引用提供给DAO。

> 应该始终在Spring IoC容器中将DataSource配置为Bean。在第一种情况下，bean被直接提供给服务。在第二种情况下，将其提供给准备好的模板。

此类发出的所有SQL都在DEBUG级别下记录，该类别对应于模板实例的完全合格的类名称（通常为JdbcTemplate，但是如果使用JdbcTemplate类的自定义子类，则可能有所不同）。

以下各节提供了一些JdbcTemplate用法的示例。这些示例不是JdbcTemplate公开的所有功能的详尽列表。请参阅附带的javadoc。

查询（SELECT）

以下查询获取关系中的行数：

```
int rowCount = this.jdbcTemplate.queryForObject("select count(*) from t_actor", Integer.class);
```

以下查询使用绑定变量：

```
int countOfActorsNamedJoe = this.jdbcTemplate.queryForObject(
        "select count(*) from t_actor where first_name = ?", Integer.class, "Joe");
```

以下查询查找字符串：

更多教程请访问:www.flydean.com

```
String lastName = this.jdbcTemplate.queryForObject(
        "select last_name from t_actor where id = ?",
        new Object[]{1212L}, String.class);
```

以下查询查找并填充单个域对象：

```
Actor actor = this.jdbcTemplate.queryForObject(
        "select first_name, last_name from t_actor where id = ?",
        new Object[]{1212L},
        new RowMapper<Actor>() {
            public Actor mapRow(ResultSet rs, int rowNum) throws SQLException {
                Actor actor = new Actor();
                actor.setFirstName(rs.getString("first_name"));
                actor.setLastName(rs.getString("last_name"));
                return actor;
            }
        });
```

以下查询查找并填充许多域对象：

```
List<Actor> actors = this.jdbcTemplate.query(
        "select first_name, last_name from t_actor",
        new RowMapper<Actor>() {
            public Actor mapRow(ResultSet rs, int rowNum) throws SQLException {
                Actor actor = new Actor();
                actor.setFirstName(rs.getString("first_name"));
                actor.setLastName(rs.getString("last_name"));
                return actor;
            }
        });
```

如果最后两个代码片段实际存在于同一个应用程序中，那么删除两个RowMapper匿名内部类中存在的重复并将它们提取到一个可以引用的单个类（通常是静态嵌套类）中是有意义的。 根据需要通过DAO方法。 例如，最好编写前面的代码片段，如下所示：

```
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;
    }
}
```

Updating (INSERT, UPDATE, and DELETE) with JdbcTemplate

你可以使用update（..）方法执行插入，更新和删除操作。 参数值通常作为变量参数提供，或者作为对象数组提供。

以下示例插入一个新条目：

```
this.jdbcTemplate.update(
        "insert into t_actor (first_name, last_name) values (?, ?)",
        "Leonor", "Watling");
```

以下示例更新现有条目：

更多教程请访问:www.flydean.com

```
this.jdbcTemplate.update(
        "update t_actor set last_name = ? where id = ?",
        "Banjo", 5276L);
```

以下示例删除现有条目：

```
this.jdbcTemplate.update(
        "delete from actor where id = ?",
        Long.valueOf(actorId));
```

其他JdbcTemplate操作

你可以使用execute（..）方法来运行任何任意SQL。 因此，该方法通常用于DDL语句。 带有回调接口，绑定变量数组等的变体极大地超载了它。 以下示例创建一个表：

```
this.jdbcTemplate.execute("create table mytable (id integer, name varchar(100))");
```

下面调用一个存储过程：

```
this.jdbcTemplate.update(
        "call SUPPORT.REFRESH_ACTORS_SUMMARY(?)",
        Long.valueOf(unionId));
```

JdbcTemplate最佳实践

一旦配置，JdbcTemplate类的实例是线程安全的。 这很重要，因为这意味着你可以配置JdbcTemplate的单个实例，然后安全地将此共享引用注入到多个DAO（或存储库）中。 JdbcTemplate是有状态的，因为它维护对DataSource的引用，但是此状态不是会话状态。

使用JdbcTemplate类（和关联的NamedParameterJdbcTemplate类）的常见做法是在Spring配置文件中配置DataSource，然后将共享的DataSource bean依赖注入到DAO类中。 在数据源的设置器中创建了JdbcTemplate。 这导致类似于以下内容的DAO：

```
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...
}
```

下面是相应的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"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <bean id="corporateEventDao" class="com.example.JdbcCorporateEventDao">
        <property name="dataSource" ref="dataSource"/>
    </bean>
```

更多教程请访问:www.flydean.com

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

    <context:property-placeholder location="jdbc.properties"/>

</beans>
```

显式配置的替代方法是使用组件扫描和注解支持进行依赖项注入。 在这种情况下，可以使用@Repository注解该类（这使其成为组件扫描的候选对象），并使用@Autowired注解DataSource setter方法。 以下示例显示了如何执行此操作：

```
@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...
}
```

下面是相应的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"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <!-- Scans within the base package of the application for @Component classes to configure as beans -->
    <context:component-scan base-package="org.springframework.docs.test" />

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

    <context:property-placeholder location="jdbc.properties"/>

</beans>
```

如果你使用Spring的JdbcDaoSupport类，并且从中扩展了各种JDBC支持的DAO类，则你的子类将从JdbcDaoSupport类继承一个setDataSource（..）方法。 你可以选择是否从此类继承。 提供JdbcDaoSupport类只是为了方便。

不管你选择使用（或不使用）以上哪种模板初始化样式，都不需要在每次要运行SQL时都创建JdbcTemplate类的新实例。 配置完成后，JdbcTemplate实例是线程安全的。 如果你的应用程序访问多个数据库，则可能需要多个JdbcTemplate实例，这需要多个DataSources，以及随后的多个不同配置的JdbcTemplate实例。

更多教程请访问:www.flydean.com

3.3.2 使用NamedParameterJdbcTemplate

与仅使用经典占位符（'? '）编程的JDBC语句相反，NamedParameterJdbcTemplate类增加了使用命名参数对JDBC语句进行编程的支持。 NamedParameterJdbcTemplate类包装JdbcTemplate并委托给包装的JdbcTemplate完成其许多工作。 本节仅描述NamedParameterJdbcTemplate类的与JdbcTemplate本身不同的那些区域，即使用命名参数对JDBC语句进行编程。 下面的示例演示如何使用NamedParameterJdbcTemplate：

```
// 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);
}
```

请注意，在分配给sql变量的值和插入namednames变量（MapSqlParameterSource类型）的相应值中使用了命名参数符号。

另外，你也可以使用基于Map的样式将命名参数及其对应的值传递给NamedParameterJdbcTemplate实例.NamedParameterJdbcOperations公开和NamedParameterJdbcTemplate类实现的其余方法遵循类似的模式，此处不再赘述。

以下示例说明了基于Map的样式的使用：

```
// 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);
}
```

SqlParameterSource接口是与NamedParameterJdbcTemplate相关的一个不错的功能（并且存在于同一Java包中）。 你已经在前面的代码片段之一（MapSqlParameterSource类）中看到了此接口的实现示例。 SqlParameterSource是NamedParameterJdbcTemplate的命名参数值的源。 MapSqlParameterSource类是一个简单的实现，它是围绕java.util.Map的适配器，其中键是参数名称，值是参数值。

另一个SqlParameterSource实现是BeanPropertySqlParameterSource类。 此类包装任意JavaBean（即，遵循JavaBean约定的类的实例），并将包装的JavaBean的属性用作命名参数值的源。

以下示例显示了典型的JavaBean：

```
public class Actor {
```

更多教程请访问:www.flydean.com

```
    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...

}
```

以下示例使用NamedParameterJdbcTemplate返回上一示例中显示的类的成员数：

```
// some JDBC-backed DAO class...
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

public void setDataSource(DataSource dataSource) {
    this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);
}

public int countOfActors(Actor exampleActor) {

    // notice how the named parameters match the properties of the above 'Actor' class
    String sql = "select count(*) from T_ACTOR where first_name = :firstName and last_name = :lastName";

    SqlParameterSource namedParameters = new BeanPropertySqlParameterSource(exampleActor);

    return this.namedParameterJdbcTemplate.queryForObject(sql, namedParameters, Integer.class);
}
```

请记住，NamedParameterJdbcTemplate类包装了经典的JdbcTemplate模板。 如果需要访问包装的JdbcTemplate实例以访问仅在JdbcTemplate类中提供的功能，则可以使用getJdbcOperations（）方法通过JdbcOperations接口访问包装的JdbcTemplate。

另请参见JdbcTemplate最佳实践，以获取有关在应用程序上下文中使用NamedParameterJdbcTemplate类的指导。

3.3.3 使用SQLExceptionTranslator

SQLExceptionTranslator是要由可在SQLExceptions和Spring自己的org.springframework.dao.DataAccessException之间进行转换的类实现的接口，该类与数据访问策略无关。为了提高精度，实现可以是通用的（例如，使用SQLState代码用于JDBC）或专有的（例如，使用Oracle错误代码）。

SQLErrorCodeSQLExceptionTranslator是默认使用的SQLExceptionTranslator的实现。此实现使用特定的供应商代码。它比SQLState实现更精确。错误代码转换基于JavaBean类型类（称为SQLErrorCodes）中保存的代码。此类是由SQLErrorCodesFactory创建和填充的，顾名思义，该类是一个工厂，用于根据名为sql-error-codes.xml的配置文件的内容创建SQLErrorCodes。此文件使用供应商代码填充，并基于从DatabaseMetaData中获取的DatabaseProductName填充。使用你正在使用的实际数据库的代码。

SQLErrorCodeSQLExceptionTranslator按以下顺序应用匹配规则：

1. 子类实现的任何自定义转换。通常，将使用提供的具体SQLErrorCodeSQLExceptionTranslator，因此此规则不适

更多教程请访问:www.flydean.com

用。仅当你确实提供了子类实现时，它才适用。
2. 作为SQLErrorCodes类的customSqlExceptionTranslator属性提供的SQLExceptionTranslator接口的任何自定义实现。
3. 搜索CustomSQLErrorCodesTranslation类的实例列表（为SQLErrorCodes类的customTranslations属性提供），以查找匹配项。
4. 错误代码匹配被应用。
5. 使用后备翻译器。 SQLExceptionSubclassTranslator是默认的后备转换器。如果此翻译不可用，则下一个后备翻译器是SQLStateSQLExceptionTranslator。

> 默认情况下，使用SQLErrorCodesFactory定义错误代码和自定义异常转换。 从类路径的名为sql-error-codes.xml的文件中查找它们，并根据使用中数据库的数据库元数据中的数据库名称找到匹配的SQLErrorCodes实例。

你可以扩展SQLErrorCodeSQLExceptionTranslator，如以下示例所示：

```
public class CustomSQLErrorCodesTranslator extends SQLErrorCodeSQLExceptionTranslator {

    protected DataAccessException customTranslate(String task, String sql, SQLException sqlex) {
        if (sqlex.getErrorCode() == -12345) {
            return new DeadlockLoserDataAccessException(task, sqlex);
        }
        return null;
    }
}
```

在前面的示例中，特定的错误代码（-12345）被转换，而其他错误则由默认转换器实现转换。 要使用此自定义转换器，必须通过setExceptionTranslator方法将其传递给JdbcTemplate，并且必须在需要此转换器的所有数据访问处理中使用此JdbcTemplate。 以下示例显示了如何使用此自定义转换器：

```
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);
}
```

定制转换器会传递一个数据源，以便在sql-error-codes.xml中查找错误代码。

3.3.4 运行声明

运行SQL语句需要很少的代码。 你需要一个数据源和一个JdbcTemplate，包括JdbcTemplate随附的便捷方法。 下面的示例显示了创建一个新表的最小但功能齐全的类需要包含的内容：

```
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;
```

更多教程请访问:www.flydean.com

```
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))");
    }
}
```

3.3.5 运行查询

一些查询方法返回单个值。 要从一行中检索计数或特定值，请使用queryForObject（..）。 后者将返回的JDBC Type转换为作为参数传递的Java类。 如果类型转换无效，则抛出InvalidDataAccessApiUsageException。 以下示例包含两种查询方法，一种用于int，另一种用于查询String：

```
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);
    }
}
```

除了单个结果查询方法外，还有几种方法返回一个列表，其中包含查询返回的每一行的条目。 最通用的方法是queryForList（..），该方法返回一个List，其中每个元素都是一个Map，使用列名作为键，每个列包含一个条目。 如果在前面的示例中添加一种方法来检索所有行的列表，则可能如下所示：

```
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");
}
```

返回的列表类似于以下内容：

```
[{name=Bob, id=1}, {name=Mary, id=2}]
```

更多教程请访问:www.flydean.com

3.3.6 更新数据库

下面的示例更新某个主键的列：

```
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);
    }
}
```

在前面的示例中，SQL语句具有用于行参数的占位符。 你可以将参数值作为varargs或作为对象数组传递。 因此，你应该在原语包装器类中显式包装原语，或者应该使用自动装箱。

3.3.7 检索自动生成的密钥

update（）便捷方法支持检索由数据库生成的主键。 此支持是JDBC 3.0标准的一部分。 有关详细信息，请参见规范的第13.6章。 该方法将PreparedStatementCreator作为其第一个参数，这是指定所需插入语句的方式。 另一个参数是KeyHolder，它包含从更新成功返回时生成的密钥。 没有标准的单一方法来创建适当的PreparedStatement（这说明了为什么方法签名就是这样）。 以下示例在Oracle上有效，但在其他平台上可能不适用：

```
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
```

更多教程请访问:www.flydean.com

3.4控制数据库连接

本节内容包括：

- 使用DataSource
- 使用DataSourceUtils
- 实现SmartDataSource
- 扩展AbstractDataSource
- 使用SingleConnectionDataSource
- 使用DriverManagerDataSource
- 使用TransactionAwareDataSourceProxy
- 使用DataSourceTransactionManager

3.4.1 使用数据源

Spring通过数据源获得与数据库的连接。数据源是JDBC规范的一部分，是通用的连接工厂。它允许容器或框架从应用程序代码中隐藏连接池和事务管理问题。作为开发人员，你无需了解有关如何连接到数据库的详细信息。这是设置数据源的管理员的责任。你很可能在开发和测试代码时同时担当这两个角色，但是不必一定要知道如何配置生产数据源。

使用Spring的JDBC层时，你可以从JNDI获取数据源，也可以使用第三方提供的连接池实现来配置自己的数据源。流行的实现是Apache Jakarta Commons DBCP和C3P0。 Spring发行版中的实现仅用于测试目的，不提供池化。

本部分使用Spring的DriverManagerDataSource实现，稍后将介绍其他一些实现。

> 你应仅将DriverManagerDataSource类用于测试目的，因为它不提供缓冲池，并且在发出多个连接请求时性能不佳。

要配置DriverManagerDataSource：

1. 通常会获得JDBC连接，因此获得与DriverManagerDataSource的连接。
2. 指定JDBC驱动程序的标准类名，以便DriverManager可以加载驱动程序类。
3. 提供在JDBC驱动程序之间变化的URL。 （有关正确的值，请参阅驱动程序的文档。）
4. 提供用户名和密码以连接到数据库。

以下示例显示了如何在Java中配置DriverManagerDataSource：

```
DriverManagerDataSource dataSource = new DriverManagerDataSource();
dataSource.setDriverClassName("org.hsqldb.jdbcDriver");
dataSource.setUrl("jdbc:hsqldb:hsql://localhost:");
dataSource.setUsername("sa");
dataSource.setPassword("");
```

下面是相应的XML配置：

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>

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

更多教程请访问:www.flydean.com

接下来的两个示例显示了DBCP和C3P0的基本连接和配置。 要了解更多有助于控制池功能的选项，请参阅相应连接池实现的产品文档。

以下示例显示了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"/>
```

下面是C3P0的例子：

```
<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"/>
```

3.4.2 使用DataSourceUtils

DataSourceUtils类是一种方便且功能强大的帮助器类，它提供静态方法从JNDI获取连接并在必要时关闭连接。它支持使用例如DataSourceTransactionManager进行线程绑定的连接。

3.4.3 实施SmartDataSource

SmartDataSource接口应该由可以提供与关系数据库的连接的类来实现。它扩展了DataSource接口，以允许使用它的类查询在给定操作后是否应关闭连接。当你知道需要重用连接时，这种用法很有效。

3.4.4 扩展AbstractDataSource

AbstractDataSource是Spring的DataSource实现的抽象基类。它实现了所有DataSource实现通用的代码。如果编写自己的DataSource实现，则应该扩展AbstractDataSource类。

3.4.5 使用SingleConnectionDataSource

SingleConnectionDataSource类是SmartDataSource接口的实现，该接口包装了每次使用后都未关闭的单个Connection。这不是多线程功能。

如果假设共享池连接（例如使用持久性工具）时有任何客户端代码调用关闭，则应将preventClose属性设置为true。此设置将返回用于封装物理连接的封闭代理。请注意，你不能再将此对象转换为本地Oracle Connection或类似对象。

SingleConnectionDataSource主要是一个测试类。例如，它结合简单的JNDI环境，可以在应用服务器外部轻松测试代码。与DriverManagerDataSource相比，它始终重用同一连接，避免了过多的物理连接创建。

3.4.6 使用DriverManagerDataSource

更多教程请访问:www.flydean.com

DriverManagerDataSource类是标准DataSource接口的实现，该接口通过bean属性配置纯JDBC驱动程序，并每次返回一个新的Connection。

此实现对于Java EE容器外部的测试和独立环境非常有用，可以作为Spring IoC容器中的DataSource bean或与简单的JNDI环境结合使用。假定使用池的Connection.close（）调用将关闭连接，因此任何可识别DataSource的持久性代码都应起作用。但是，即使在测试环境中，使用JavaBean风格的连接池（例如commons-dbcp）也是如此容易，以至总是总是最好在DriverManagerDataSource上使用这样的连接池。

3.4.7 使用TransactionAwareDataSourceProxy

TransactionAwareDataSourceProxy是目标数据源的代理。代理包装该目标数据源以增加对Spring管理的事务的认识。在这方面，它类似于Java EE服务器提供的事务性JNDI数据源。

> 除非需要调用已经存在的代码并通过标准的JDBC DataSource接口实现，否则很少需要使用此类。 在这种情况下，你仍然可以使该代码可用，同时使该代码参与Spring托管的事务。 通常，最好使用更高级别的资源管理抽象来编写自己的新代码，例如JdbcTemplate或DataSourceUtils。

3.4.8 使用DataSourceTransactionManager

DataSourceTransactionManager类是单个JDBC数据源的PlatformTransactionManager实现。它将JDBC连接从指定的数据源绑定到当前正在执行的线程，可能允许每个数据源一个线程连接。

通过DataSourceUtils.getConnection（DataSource）而不是Java EE的标准DataSource.getConnection检索JDBC连接需要应用程序代码。它抛出未经检查的org.springframework.dao异常，而不是经过检查的SQLException。所有框架类（例如JdbcTemplate）都隐式使用此策略。如果不与该事务管理器一起使用，则查找策略的行为与普通策略完全相同。因此，可以在任何情况下使用它。

DataSourceTransactionManager类支持自定义隔离级别和超时，这些隔离级别和超时将作为适当的JDBC语句查询超时应用。为了支持后者，应用程序代码必须使用JdbcTemplate或为每个创建的语句调用DataSourceUtils.applyTransactionTimeout（..）方法。

在单资源情况下，可以使用此实现而不是JtaTransactionManager，因为它不需要容器支持JTA。只要你坚持要求的连接查找模式，就可以在两者之间进行切换只是配置问题。 JTA不支持自定义隔离级别。

更多教程请访问:www.flydean.com

3.5JDBC批处理操作

如果将多个调用批处理到同一条准备好的语句，则大多数JDBC驱动程序都会提高性能。 通过将更新分组，可以限制到数据库的往返次数。

3.5.1 使用JdbcTemplate的基本批处理操作

你可以通过实现特殊接口的两个方法BatchPreparedStatementSetter并将该实现作为batchUpdate方法调用中的第二个参数传入来完成JdbcTemplate批处理。 你可以使用getBatchSize方法提供当前批处理的大小。 你可以使用setValues方法设置准备好的语句的参数值。 此方法称为你在getBatchSize调用中指定的次数。 以下示例根据列表中的条目更新actor表，并将整个列表用作批处理：

```
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) {
        return this.jdbcTemplate.batchUpdate(
                "update t_actor set first_name = ?, last_name = ? where id = ?",
                new BatchPreparedStatementSetter() {
                    public void setValues(PreparedStatement ps, int i) throws SQLException {
                        ps.setString(1, actors.get(i).getFirstName());
                        ps.setString(2, actors.get(i).getLastName());
                        ps.setLong(3, actors.get(i).getId().longValue());
                    }
                    public int getBatchSize() {
                        return actors.size();
                    }
                });
    }

    // ... additional methods
}
```

如果处理更新流或从文件读取，则可能具有首选的批处理大小，但最后一批可能没有该数量的条目。在这种情况下，可以使用InterruptibleBatchPreparedStatementSetter接口，该接口可在输入源耗尽后中断批处理。 isBatchExhausted方法使你可以发出批处理结束的信号。

3.5.2 对象列表的批处理操作

JdbcTemplate和NamedParameterJdbcTemplate都提供了另一种提供批处理更新的方式。无需实现特殊的批处理接口，而是将调用中的所有参数值作为列表提供。框架遍历这些值并使用内部的准备好的语句设置器。 API会有所不同，具体取决于你是否使用命名参数。对于命名参数，你提供一个SqlParameterSource数组，该批处理的每个成员都有一个条目。你可以使用SqlParameterSourceUtils.createBatch便捷方法创建此数组，传入一个bean样式的对象数组（带有与参数相对应的getter方法），字符串键的Map实例（包含对应的参数作为值），或者混合使用都。

以下示例显示了使用命名参数的批量更新：

```
public class JdbcActorDao implements ActorDao {

    private NamedParameterTemplate namedParameterJdbcTemplate;
```

更多教程请访问:www.flydean.com

```
    public void setDataSource(DataSource dataSource) {
        this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);
    }

    public int[] batchUpdate(List<Actor> actors) {
        return this.namedParameterJdbcTemplate.batchUpdate(
                "update t_actor set first_name = :firstName, last_name = :lastName where id = :id",
                SqlParameterSourceUtils.createBatch(actors));
    }

    // ... additional methods
}
```

对于使用经典的SQL语句？ 占位符，则传入包含包含更新值的对象数组的列表。 该对象数组在SQL语句中的每个占位符必须有一个条目，并且它们的顺序必须与SQL语句中定义的顺序相同。

以下示例与前面的示例相同，不同之处在于它使用经典的JDBC？占位符：

```
public class JdbcActorDao implements ActorDao {

    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    public int[] batchUpdate(final List<Actor> actors) {
        List<Object[]> batch = new ArrayList<Object[]>();
        for (Actor actor : actors) {
            Object[] values = new Object[] {
                    actor.getFirstName(), actor.getLastName(), actor.getId()};
            batch.add(values);
        }
        return this.jdbcTemplate.batchUpdate(
                "update t_actor set first_name = ?, last_name = ? where id = ?",
                batch);
    }

    // ... additional methods
}
```

我们前面介绍的所有批处理更新方法都返回一个int数组，其中包含每个批处理条目的受影响行数。此计数由JDBC驱动程序报告。如果该计数不可用，则JDBC驱动程序将返回值-2。

> 在这种情况下，通过在基础PreparedStatement上自动设置值，需要从给定的Java类型派生每个值的对应JDBC类型。尽管这通常效果很好，但存在潜在的问题（例如，包含Map的空值）。在这种情况下，Spring默认情况下会调用ParameterMetaData.getParameterType，这对于JDBC驱动程序可能会很昂贵。如果遇到性能问题，应使用最新的驱动程序版本，并考虑将spring.jdbc.getParameterType.ignore属性设置为true（作为JVM系统属性或在类路径根目录中的spring.properties文件中）。，如Oracle 12c（SPR-16139）所述。
>
> 或者，你可以考虑通过“ BatchPreparedStatementSetter”（如前所示），通过为基于“ List ”的调用提供的显式类型数组，通过在服务器上的“ registerSqlType”调用来显式指定相应的JDBC类型。自定义“ MapSqlParameterSource”实例，或者通过“ BeanPropertySqlParameterSource”实例从Java声明的属性类型中获取SQL类型，即使对于null值也是如此。

3.5.3 具有多个批次的批次操作

更多教程请访问:www.flydean.com

前面的批处理更新示例处理的批处理过大，你想将它们分解成几个较小的批处理。 你可以通过多次调用batchUpdate方法来使用前面提到的方法来执行此操作，但是现在有了一个更方便的方法。 除了SQL语句外，此方法还包含一个对象集合，该对象包含参数，每个批处理要进行的更新次数以及一个ParameterizedPreparedStatementSetter来设置准备好的语句的参数值。 框架遍历提供的值，并将更新调用分成指定大小的批处理。

以下示例显示了使用100的批量大小的批量更新：

```
public class JdbcActorDao implements ActorDao {

    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    public int[][] batchUpdate(final Collection<Actor> actors) {
        int[][] updateCounts = jdbcTemplate.batchUpdate(
                "update t_actor set first_name = ?, last_name = ? where id = ?",
                actors,
                100,
                new ParameterizedPreparedStatementSetter<Actor>() {
                    public void setValues(PreparedStatement ps, Actor argument) throws SQLException {
                        ps.setString(1, argument.getFirstName());
                        ps.setString(2, argument.getLastName());
                        ps.setLong(3, argument.getId().longValue());
                    }
                });
        return updateCounts;
    }

    // ... additional methods
}
```

此调用的批处理更新方法返回一个int数组数组，该数组包含每个批处理的数组条目以及每个更新受影响的行数的数组。 顶层数组的长度指示已执行的批处理数量，第二层数组的长度指示该批处理中的更新数量。 每个批次中的更新数量应该是为所有批次提供的批次大小（最后一个可能更少），这取决于所提供的更新对象的总数。 每个更新语句的更新计数是JDBC驱动程序报告的计数。 如果该计数不可用，则JDBC驱动程序将返回值-2。

更多教程请访问:www.flydean.com

3.6使用SimpleJdbc

SimpleJdbcInsert和SimpleJdbcCall类通过利用可通过JDBC驱动程序检索的数据库元数据来提供简化的配置。这意味着您可以更少地进行前期配置，但是如果您希望在代码中提供所有详细信息，则可以覆盖或关闭元数据处理。

3.6.1 使用SimpleJdbcInsert插入数据

我们首先查看具有最少配置选项的SimpleJdbcInsert类。您应该在数据访问层的初始化方法中实例化SimpleJdbcInsert。对于此示例，初始化方法是setDataSource方法。您不需要子类化SimpleJdbcInsert类。而是可以创建一个新实例，并使用withTableName方法设置表名。此类的配置方法遵循流畅的样式，该样式返回SimpleJdbcInsert的实例，该实例使您可以链接所有配置方法。以下示例仅使用一种配置方法（我们稍后将显示多种方法的示例）：

```
public class JdbcActorDao implements ActorDao {

    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcInsert insertActor;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        this.insertActor = new SimpleJdbcInsert(dataSource).withTableName("t_actor");
    }

    public void add(Actor actor) {
        Map<String, Object> parameters = new HashMap<String, Object>(3);
        parameters.put("id", actor.getId());
        parameters.put("first_name", actor.getFirstName());
        parameters.put("last_name", actor.getLastName());
        insertActor.execute(parameters);
    }

    // ... additional methods
}
```

这里使用的execute方法将纯java.util.Map作为其唯一参数。 这里要注意的重要一点是，用于Map的键必须与数据库中定义的表的列名匹配。 这是因为我们读取元数据来构造实际的insert语句。

3.6.2 通过使用SimpleJdbcInsert检索自动生成的密钥

下一个示例使用与前面的示例相同的插入，但是它没有传递id，而是检索自动生成的键并将其设置在新的Actor对象上。当创建SimpleJdbcInsert时，除了指定表名之外，它还使用usingGeneratedKeyColumns方法指定生成的键列的名称。以下清单显示了它的工作方式：

```
    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());
```

更多教程请访问:www.flydean.com

```
        Number newId = insertActor.executeAndReturnKey(parameters);
        actor.setId(newId.longValue());
    }

    // ... additional methods
}
```

使用第二种方法运行插入时的主要区别在于，您没有将ID添加到Map中，而是调用了executeAndReturnKey方法。 这将返回一个java.lang.Number对象，您可以使用该对象创建域类中使用的数字类型的实例。 您不能依赖所有数据库在这里返回特定的Java类。 java.lang.Number是您可以依赖的基类。 如果您有多个自动生成的列，或者生成的值是非数字的，则可以使用从executeAndReturnKeyHolder方法返回的KeyHolder。

3.6.3 为SimpleJdbcInsert指定列

您可以使用usingColumns方法指定列名列表来限制插入的列，如以下示例所示：

```
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
}
```

插入的执行与依靠元数据确定要使用的列的执行相同。

3.6.4 使用SqlParameterSource提供参数值

使用Map提供参数值可以很好地工作，但这不是最方便使用的类。 Spring提供了SqlParameterSource接口的几个实现，您可以使用它们来代替。 第一个是BeanPropertySqlParameterSource，如果您有一个包含值的JavaBean兼容类，则这是一个非常方便的类。 它使用相应的getter方法提取参数值。 下面的示例演示如何使用BeanPropertySqlParameterSource：

```
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");
```

更多教程请访问:www.flydean.com

```
    }

    public void add(Actor actor) {
        SqlParameterSource parameters = new BeanPropertySqlParameterSource(actor);
        Number newId = insertActor.executeAndReturnKey(parameters);
        actor.setId(newId.longValue());
    }

    // ... additional methods
}
```

另一个选项是MapSqlParameterSource，它类似于Map，但是提供了可以链接的更方便的addValue方法。 以下示例显示了如何使用它：

```
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
}
```

如您所见，配置是相同的。 只有执行代码才能更改为使用这些替代输入类。

3.6.5 用SimpleJdbcCall调用存储过程

SimpleJdbcCall类使用数据库中的元数据来查找in和out参数的名称，因此您不必显式声明它们。 如果愿意，可以声明参数，也可以声明参数（例如ARRAY或STRUCT）没有自动映射到Java类的参数。 第一个示例显示了一个简单的过程，该过程仅从MySQL数据库返回VARCHAR和DATE格式的标量值。 示例过程读取指定的参与者条目，并以out参数的形式返回first_name，last_name和birth_date列。 以下清单显示了第一个示例：

```
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参数包含您要查找的参与者的ID。 out参数返回从表读取的数据。

更多教程请访问:www.flydean.com

您可以以类似于声明SimpleJdbcInsert的方式声明SimpleJdbcCall。 您应该在数据访问层的初始化方法中实例化并配置该类。 与StoredProcedure类相比，您无需创建子类，也无需声明可以在数据库元数据中查找的参数。 下面的SimpleJdbcCall配置示例使用前面的存储过程（除DataSource之外，唯一的配置选项是存储过程的名称）：

```
public class JdbcActorDao implements ActorDao {

    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcCall procReadActor;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        this.procReadActor = new SimpleJdbcCall(dataSource)
                .withProcedureName("read_actor");
    }

    public Actor readActor(Long id) {
        SqlParameterSource in = new MapSqlParameterSource()
                .addValue("in_id", id);
        Map out = procReadActor.execute(in);
        Actor actor = new Actor();
        actor.setId(id);
        actor.setFirstName((String) out.get("out_first_name"));
        actor.setLastName((String) out.get("out_last_name"));
        actor.setBirthDate((Date) out.get("out_birth_date"));
        return actor;
    }

    // ... additional methods
}
```

您为执行调用编写的代码涉及创建包含IN参数的SqlParameterSource。您必须为输入值提供的名称与存储过程中声明的参数名称的名称匹配。大小写不必匹配，因为您使用元数据来确定在存储过程中应如何引用数据库对象。源中为存储过程指定的内容不一定是存储过程在数据库中存储的方式。一些数据库将名称转换为全部大写，而其他数据库则使用小写或指定的大小写。

execute方法采用IN参数并返回一个Map，该Map包含由存储过程中指定的名称键入的所有out参数。在这种情况下，它们是out_first_name，out_last_name和out_birth_date。

execute方法的最后一部分创建一个Actor实例，以用于返回检索到的数据。同样，重要的是使用out参数的名称，因为它们在存储过程中已声明。同样，结果映射表中存储的out参数名称中的大小写与数据库中out参数名称中的大小写匹配，这在数据库之间可能会有所不同。为了使代码更具可移植性，您应该执行不区分大小写的查找或指示Spring使用LinkedCaseInsensitiveMap。为此，您可以创建自己的JdbcTemplate并将setResultsMapCaseInsensitive属性设置为true。然后，您可以将此自定义的JdbcTemplate实例传递到SimpleJdbcCall的构造函数中。以下示例显示了此配置：

```
public class JdbcActorDao implements ActorDao {

    private SimpleJdbcCall procReadActor;

    public void setDataSource(DataSource dataSource) {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
        jdbcTemplate.setResultsMapCaseInsensitive(true);
        this.procReadActor = new SimpleJdbcCall(jdbcTemplate)
                .withProcedureName("read_actor");
    }

    // ... additional methods
}
```

通过执行此操作，可以避免在用于返回参数名称的情况下发生冲突。

更多教程请访问:www.flydean.com

3.6.6 明确声明要用于SimpleJdbcCall的参数

在本章的前面，我们描述了如何从元数据推导出参数，但是如果需要，可以显式声明它们。您可以通过使用defineParameters方法创建和配置SimpleJdbcCall来实现，该方法将可变数量的SqlParameter对象作为输入。有关如何定义SqlParameter的详细信息，请参见下一部分。

> 如果您使用的数据库不是Spring支持的数据库，则必须进行显式声明。当前，Spring支持针对以下数据库的存储过程调用的元数据查找：Apache Derby，DB2，MySQL，Microsoft SQL Server，Oracle和Sybase。我们还支持MySQL，Microsoft SQL Server和Oracle存储功能的元数据查找。

您可以选择显式声明一个，一些或所有参数。在未显式声明参数的地方，仍使用参数元数据。要绕过对潜在参数的元数据查找的所有处理，并且仅使用声明的参数，可以将不带ProcedureColumnMetaDataAccess的方法作为声明的一部分。假设您为数据库函数声明了两个或多个不同的调用签名。在这种情况下，您调用useInParameterNames来指定要包含在给定签名中的IN参数名称的列表。

下面的示例显示一个完全声明的过程调用，并使用前面示例中的信息：

```
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
}
```

两个示例的执行和最终结果相同。 第二个示例明确指定所有详细信息，而不是依赖于元数据。

3.6.7 如何定义SqlParameters

要为SimpleJdbc类和RDBMS操作类（在JDBC操作建模为Java对象中发现）定义参数，可以使用SqlParameter或其子类之一。 为此，通常在构造函数中指定参数名称和SQL类型。 通过使用java.sql.Types常量指定SQL类型。 在本章的前面，我们看到了类似于以下内容的声明：

```
new SqlParameter("in_id", Types.NUMERIC),
new SqlOutParameter("out_first_name", Types.VARCHAR),
```

带有SqlParameter的第一行声明一个IN参数。通过使用SqlQuery及其子类（可以在理解SqlQuery中找到），可以将IN参数用于存储过程调用和查询。

第二行（带有SqlOutParameter）声明在存储过程调用中使用的out参数。还有一个用于InOut参数的SqlInOutParameter（为过程提供IN值并返回值的参数）。

更多教程请访问:www.flydean.com

> 仅声明为SqlParameter和SqlInOutParameter的参数用于提供输入值。这与StoredProcedure类不同，后者（出于向后兼容的原因）允许为声明为SqlOutParameter的参数提供输入值。

对于IN参数，除了名称和SQL类型外，还可以为数字数据指定小数位，或者为自定义数据库类型指定类型名。对于out参数，可以提供RowMapper来处理从REF游标返回的行的映射。另一个选择是指定一个SqlReturnType，它提供了一个机会来定义返回值的自定义处理。

3.6.8 通过使用SimpleJdbcCall调用存储的函数

可以使用与调用存储过程几乎相同的方式来调用存储函数，除了提供函数名而不是过程名。 您将withFunctionName方法用作配置的一部分，以指示您要调用函数，并生成函数调用的相应字符串。 专门的执行调用（executeFunction）用于执行函数，它以指定类型的对象的形式返回函数的返回值，这意味着您不必从结果图中检索返回值。 对于只有一个out参数的存储过程，也可以使用类似的便捷方法（名为executeObject）。 以下示例（对于MySQL）基于一个名为get_actor_name的存储函数，该函数返回演员的全名：

```
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;
```

要调用此函数，我们再次在初始化方法中创建一个SimpleJdbcCall，如以下示例所示：

```
public class JdbcActorDao implements ActorDao {

    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcCall funcGetActorName;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
        jdbcTemplate.setResultsMapCaseInsensitive(true);
        this.funcGetActorName = new SimpleJdbcCall(jdbcTemplate)
                .withFunctionName("get_actor_name");
    }

    public String getActorName(Long id) {
        SqlParameterSource in = new MapSqlParameterSource()
                .addValue("in_id", id);
        String name = funcGetActorName.executeFunction(String.class, in);
        return name;
    }

    // ... additional methods
}
```

使用的executeFunction方法返回一个String，该String包含函数调用的返回值。

3.6.9 从SimpleJdbcCall返回ResultSet或REF游标

调用返回结果集的存储过程或函数有点棘手。 一些数据库在JDBC结果处理期间返回结果集，而另一些数据库则需要显式注册的特定类型的参数。 两种方法都需要进行额外的处理才能遍历结果集并处理返回的行。 通过SimpleJdbcCall，可以使用returningResultSet方法并声明要用于特定参数的RowMapper实现。 如果在结果处理期间返回了结果集，则没

更多教程请访问:www.flydean.com

有定义名称，因此返回的结果必须与您声明RowMapper实现的顺序匹配。 指定的名称仍用于将处理后的结果列表存储在由execute语句返回的结果映射中。

下一个示例（对于MySQL）使用存储过程，该存储过程不使用IN参数，并且返回t_actor表中的所有行：

```
CREATE PROCEDURE read_all_actors()
BEGIN
 SELECT a.id, a.first_name, a.last_name, a.birth_date FROM t_actor a;
END;
```

要调用此过程，可以声明RowMapper。 因为要映射到的类遵循JavaBean规则，所以可以使用BeanPropertyRowMapper，该类是通过在newInstance方法中传入要映射的必需类而创建的。 以下示例显示了如何执行此操作：

```
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>(0));
        return (List) m.get("actors");
    }

    // ... additional methods
}
```

execute调用传递一个空的Map，因为此调用不带任何参数。 然后从结果图中检索参与者列表，并将其返回给调用者。

更多教程请访问:www.flydean.com

3.7将JDBC操作建模为Java对象

org.springframework.jdbc.object包包含一些类，这些类使您可以更面向对象地访问数据库。 例如，您可以执行查询并以包含业务对象的列表的形式返回结果，该业务对象的关系列数据映射到业务对象的属性。 您还可以运行存储过程并运行update，delete和insert语句。

> 许多Spring开发人员认为，下面描述的各种RDBMS操作类（StoredProcedure类除外）通常可以用直接的JdbcTemplate调用代替。 通常，编写直接在JdbcTemplate上调用方法的DAO方法（与将查询封装为完整的类相反）更容易。
>
> 但是，如果您通过使用RDBMS操作类获得可测量的价值，则应继续使用这些类。

3.7.1 了解SqlQuery

SqlQuery是可重用的，线程安全的类，它封装了SQL查询。 子类必须实现newRowMapper（..）方法以提供RowMapper实例，该实例可以为通过在查询执行期间创建的ResultSet进行迭代而获得的每一行创建一个对象。 很少直接使用SqlQuery类，因为MappingSqlQuery子类为将行映射到Java类提供了更为方便的实现。 扩展SqlQuery的其他实现是MappingSqlQueryWithParameters和UpdatableSqlQuery。

3.7.2 使用MappingSqlQuery

MappingSqlQuery是可重用的查询，其中具体的子类必须实现抽象的mapRow（..）方法，以将提供的ResultSet的每一行转换为指定类型的对象。 以下示例显示了一个自定义查询，该查询将t_actor关系中的数据映射到Actor类的实例：

```
public class ActorMappingQuery extends MappingSqlQuery<Actor> {

    public ActorMappingQuery(DataSource ds) {
        super(ds, "select id, first_name, last_name from t_actor where id = ?");
        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;
    }

}
```

该类扩展了使用Actor类型参数化的MappingSqlQuery。 此客户查询的构造函数将DataSource作为唯一参数。 在此构造函数中，可以使用数据源和应执行以检索此查询的行的SQL调用超类上的构造函数。 此SQL用于创建PreparedStatement，因此它可能包含在执行期间要传递的任何参数的占位符。 您必须使用传入SqlParameter的defineParameter方法声明每个参数。 SqlParameter具有名称，并且具有java.sql.Types中定义的JDBC类型。 定义所有参数后，可以调用compile（）方法，以便可以准备语句并稍后运行。 此类在编译后是线程安全的，因此，只要在初始化DAO时创建这些实例，就可以将它们保留为实例变量并可以重用。 下面的示例演示如何定义此类：

```
private ActorMappingQuery actorMappingQuery;

@Autowired
public void setDataSource(DataSource dataSource) {
```

更多教程请访问:www.flydean.com

```
    this.actorMappingQuery = new ActorMappingQuery(dataSource);
}

public Customer getCustomer(Long id) {
    return actorMappingQuery.findObject(id);
}
```

前面示例中的方法检索具有传入的id作为唯一参数的customer。 由于只希望返回一个对象，因此我们将id作为参数调用findObject便捷方法。 相反，如果有一个查询返回一个对象列表并采用其他参数，则将使用其中一种执行方法，该方法采用以varargs形式传入的参数值数组。 以下示例显示了这种方法：

```
public List<Actor> searchForActors(int age, String namePattern) {
    List<Actor> actors = actorSearchMappingQuery.execute(age, namePattern);
    return actors;
}
```

3.7.3 使用SqlUpdate

SqlUpdate类封装了SQL更新。 与查询一样，更新对象是可重用的，并且与所有RdbmsOperation类一样，更新可以具有参数并在SQL中定义。 此类提供了许多类似于查询对象的execute（..）方法的update（..）方法。 SQLUpdate类是具体的。 可以将其子类化-例如，添加自定义更新方法。 但是，您不必子类化SqlUpdate类，因为可以通过设置SQL和声明参数来轻松地对其进行参数化。 以下示例创建一个名为execute的自定义更新方法：

```
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);
    }
}
```

3.7.4 使用StoredProcedure

StoredProcedure类是RDBMS存储过程的对象抽象的超类。此类是抽象的，并且其各种execute（..）方法具有受保护的访问权限，还可以防止使用除了通过提供更严格类型的子类之外的其他形式的访问。

继承的sql属性是RDBMS中存储过程的名称。

要为StoredProcedure类定义参数，可以使用SqlParameter或其子类之一。您必须在构造函数中指定参数名称和SQL类型，如以下代码片段所示：

更多教程请访问:www.flydean.com

```
new SqlParameter("in_id", Types.NUMERIC),
new SqlOutParameter("out_first_name", Types.VARCHAR),
```

使用java.sql.Types常量指定SQL类型。

第一行（带有SqlParameter）声明一个IN参数。您可以将IN参数用于存储过程调用和使用SqlQuery及其子类的查询（了解SqlQuery中介绍）。

第二行（带有SqlOutParameter）声明将在存储过程调用中使用的out参数。还有一个用于InOut参数的SqlInOutParameter（为过程提供in值并返回值的参数）。

对于in参数，除了名称和SQL类型之外，还可以为数字数据指定比例，或者为自定义数据库类型指定类型名称。对于out参数，可以提供RowMapper来处理从REF游标返回的行的映射。另一个选择是指定一个SqlReturnType，它允许您定义返回值的自定义处理。

下一个简单DAO的示例使用StoredProcedure调用任何Oracle数据库附带的函数（sysdate（））。要使用存储过程功能，您必须创建一个扩展StoredProcedure的类。在此示例中，StoredProcedure类是一个内部类。但是，如果需要重用StoredProcedure，则可以将其声明为顶级类。此示例没有输入参数，但是使用SqlOutParameter类将输出参数声明为日期类型。 execute（）方法将运行该过程，并从结果Map中提取返回的日期。通过使用参数名称作为键，结果Map为每个声明的输出参数（在这种情况下只有一个）都有一个条目。以下清单显示了我们的自定义StoredProcedure类：

```
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;
        }
    }

}
```

更多教程请访问:www.flydean.com

下面的StoredProcedure示例包含两个输出参数（在本例中为Oracle REF游标）：

```
import java.util.HashMap;
import java.util.Map;
import javax.sql.DataSource;
import oracle.jdbc.OracleTypes;
import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.object.StoredProcedure;

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>());
    }
}
```

请注意如何在TitlesAndGenresStoredProcedure构造函数中使用的clarifyParameter（..）方法的重载变体如何传递给RowMapper实现实例。 这是重用现有功能的非常方便而强大的方法。 接下来的两个示例提供了两个RowMapper实现的代码。

TitleMapper类将提供的ResultSet中每一行的ResultSet映射到Title域对象，如下所示：

```
import java.sql.ResultSet;
import java.sql.SQLException;
import com.foo.domain.Title;
import org.springframework.jdbc.core.RowMapper;

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类针对提供的ResultSet中的每一行将ResultSet映射到Genre域对象，如下所示：

```
import java.sql.ResultSet;
import java.sql.SQLException;
import com.foo.domain.Genre;
import org.springframework.jdbc.core.RowMapper;

public final class GenreMapper implements RowMapper<Genre> {

    public Genre mapRow(ResultSet rs, int rowNum) throws SQLException {
        return new Genre(rs.getString("name"));
    }
}
```

更多教程请访问:www.flydean.com

要将参数传递给RDBMS中定义中具有一个或多个输入参数的存储过程，可以编写一个强类型化execute（..）方法的代码，该方法将委托给超类中的非类型execute（Map）方法，例如 以下示例显示：

```
import java.sql.Types;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;
import javax.sql.DataSource;
import oracle.jdbc.OracleTypes;
import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.StoredProcedure;

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);
    }
}
```

更多教程请访问:www.flydean.com

3.8参数和数据值处理的常见问题

Spring Framework的JDBC支持提供的不同方法中存在参数和数据值的常见问题。本节介绍如何解决它们。

3.8.1 提供参数的SQL类型信息

通常，Spring根据传入的参数类型确定参数的SQL类型。可以在设置参数值时显式提供要使用的SQL类型。有时需要正确设置NULL值。

您可以通过几种方式提供SQL类型信息：

- JdbcTemplate的许多更新和查询方法都采用int数组形式的附加参数。该数组用于通过使用java.sql.Types类中的常量值来指示相应参数的SQL类型。为每个参数提供一个条目。
- 您可以使用SqlParameterValue类包装需要此附加信息的参数值。为此，请为每个值创建一个新实例，然后在构造函数中传入SQL类型和参数值。您还可以为数字值提供可选的比例参数。
- 对于使用命名参数的方法，可以使用SqlParameterSource类，BeanPropertySqlParameterSource或MapSqlParameterSource。它们都具有用于为任何命名参数值注册SQL类型的方法。

3.8.2 处理BLOB和CLOB对象

您可以在数据库中存储图像，其他二进制数据和大块文本。这些大对象称为二进制数据的BLOB（二进制大型对象），而字符数据称为CLOB（字符大型对象）。在Spring中，可以直接使用JdbcTemplate来处理这些大对象，也可以使用RDBMS Objects和SimpleJdbc类提供的更高抽象来处理这些大对象。所有这些方法都使用LobHandler接口的实现来对LOB（大对象）数据进行实际管理。 LobHandler通过getLobCreator方法提供对LobCreator类的访问，该方法用于创建要插入的新LOB对象。

LobCreator和LobHandler为LOB输入和输出提供以下支持：

- BLOB
  - byte []：getBlobAsBytes和setBlobAsBytes
  - InputStream：getBlobAsBinaryStream和setBlobAsBinaryStream
- CLOB
  - String：getClobAsString和setClobAsString
  - InputStream：getClobAsAsciiStream和setClobAsAsciiStream
  - Reader：getClobAsCharacterStream和setClobAsCharacterStream

下一个示例显示了如何创建和插入BLOB。稍后，我们展示如何从数据库中读取回它。

本示例使用JdbcTemplate和AbstractLobCreatingPreparedStatementCallback的实现。它实现了一种方法setValues。此方法提供了一个LobCreator，我们可以使用它来设置SQL插入语句中的LOB列的值。

对于此示例，我们假设存在一个变量lobHandler，该变量已经设置为DefaultLobHandler的实例。通常，您可以通过依赖注入来设置此值。

以下示例显示如何创建和插入BLOB：

```
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(
```

更多教程请访问:www.flydean.com

```
    "INSERT INTO lob_table (id, a_clob, a_blob) VALUES (?, ?, ?)",
    new AbstractLobCreatingPreparedStatementCallback(lobHandler) {
        protected void setValues(PreparedStatement ps, LobCreator lobCreator) throws SQLException {
            ps.setLong(1, 1L);
            lobCreator.setClobAsCharacterStream(ps, 2, clobReader, (int)clobIn.length());
            lobCreator.setBlobAsBinaryStream(ps, 3, blobIs, (int)blobIn.length());
        }
    }
);

blobIs.close();
clobReader.close();
```

> 如果在从DefaultLobHandler.getLobCreator（）返回的LobCreator上调用setBlobAsBinaryStream，setClobAsAsciiStream或setClobAsCharacterStream方法，则可以选择为contentLength参数指定负值。 如果指定的内容长度为负，则DefaultLobHandler将使用set-stream方法的JDBC 4.0变体，而不使用length参数。 否则，它将指定的长度传递给驱动程序。
>
> 请参阅有关JDBC驱动程序的文档，以用于验证它是否支持流式LOB而不提供内容长度。

现在是时候从数据库中读取LOB数据了。 同样，您将JdbcTemplate与相同的实例变量lobHandler和对DefaultLobHandler的引用一起使用。 以下示例显示了如何执行此操作：

```
List<Map<String, Object>> l = jdbcTemplate.query("select id, a_clob, a_blob from lob_table",
    new RowMapper<Map<String, Object>>() {
        public Map<String, Object> mapRow(ResultSet rs, int i) throws SQLException {
            Map<String, Object> results = new HashMap<String, Object>();
            String clobText = lobHandler.getClobAsString(rs, "a_clob");
            results.put("CLOB", clobText);
            byte[] blobBytes = lobHandler.getBlobAsBytes(rs, "a_blob");
            results.put("BLOB", blobBytes);
            return results;
        }
    });
```

3.8.3 传入IN子句的值列表

SQL标准允许基于包含变量值列表的表达式选择行。 一个典型的示例是select * from T_ACTOR where id in (1, 2, 3)。 JDBC标准不直接为准备好的语句支持此变量列表。 您不能声明可变数量的占位符。 您需要准备好所需数量的占位符的多种变体，或者一旦知道需要多少个占位符，就需要动态生成SQL字符串。 NamedParameterJdbcTemplate和JdbcTemplate中提供的命名参数支持采用后一种方法。 您可以将值作为原始对象的java.util.List传入。 此列表用于插入所需的占位符，并在语句执行期间传递值。

> 传递许多值时要小心。 JDBC标准不保证您可以为in表达式列表使用100个以上的值。 各种数据库都超过了这个数目，但是它们通常对允许多少个值有硬性限制。 例如，Oracle的限制为1000。

除了值列表中的原始值之外，您还可以创建对象数组的java.util.List。 该列表可以支持为in子句定义的多个表达式，例如，T_ACTOR的select * from（（1，'Johnson'），（2，'Harrop'\））中的（id，last_name）。 当然，这要求您的数据库支持此语法。

3.8.4 处理存储过程调用的复杂类型

调用存储过程时，有时可以使用特定于数据库的复杂类型。 为了容纳这些类型，Spring提供了一个SqlReturnType来处理从存储过程调用返回的值，并提供SqlTypeValue作为参数作为参数传递给存储过程。

SqlReturnType接口具有必须实现的单个方法（名为getTypeValue）。 此接口用作SqlOutParameter声明的一部分。 以下示例显示了返回用户声明类型为ITEM_TYPE的Oracle STRUCT对象的值：

更多教程请访问:www.flydean.com

```
public class TestItemStoredProcedure extends StoredProcedure {

    public TestItemStoredProcedure(DataSource dataSource) {
        ...
        declareParameter(new SqlOutParameter("item", OracleTypes.STRUCT, "ITEM_TYPE",
            new SqlReturnType() {
                public Object getTypeValue(CallableStatement cs, int colIndx, int sqlType, String typeName) thr
ows 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;
                }
            }));
        ...
    }
```

您可以使用SqlTypeValue将Java对象（例如TestItem）的值传递给存储过程。 SqlTypeValue接口具有必须实现的单个方法（名为createTypeValue）。 传入活动连接，您可以使用它来创建特定于数据库的对象，例如StructDescriptor实例或ArrayDescriptor实例。 下面的示例创建一个StructDescriptor实例：

```
final TestItem testItem = new TestItem(123L, "A test item",
        new SimpleDateFormat("yyyy-M-d").parse("2010-12-31"));

SqlTypeValue value = new AbstractSqlTypeValue() {
    protected Object createTypeValue(Connection conn, int sqlType, String typeName) throws SQLException {
        StructDescriptor itemDescriptor = new StructDescriptor(typeName, conn);
        Struct item = new STRUCT(itemDescriptor, conn,
        new Object[] {
            testItem.getId(),
            testItem.getDescription(),
            new java.sql.Date(testItem.getExpirationDate().getTime())
        });
        return item;
    }
};
```

现在，您可以将此SqlTypeValue添加到包含用于存储过程的execute调用的输入参数的Map中。

SqlTypeValue的另一个用途是将值数组传递给Oracle存储过程。 在这种情况下，Oracle有其自己的内部ARRAY类，您可以使用SqlTypeValue创建Oracle ARRAY的实例，并使用Java ARRAY中的值填充它，如以下示例所示：

```
final Long[] ids = new Long[] {1L, 2L};

SqlTypeValue value = new AbstractSqlTypeValue() {
    protected Object createTypeValue(Connection conn, int sqlType, String typeName) throws SQLException {
        ArrayDescriptor arrayDescriptor = new ArrayDescriptor(typeName, conn);
        ARRAY idArray = new ARRAY(arrayDescriptor, conn, ids);
        return idArray;
    }
};
```

更多教程请访问:www.flydean.com

3.9嵌入式数据库支持

org.springframework.jdbc.datasource.embedded包为嵌入式Java数据库引擎提供支持。 本地提供对HSQL，H2和Derby的支持。 您还可以使用可扩展的API来插入新的嵌入式数据库类型和DataSource实现。

3.9.1 为什么要使用嵌入式数据库?

嵌入式数据库由于其轻量级的特性，因此在项目的开发阶段可能会很有用。 好处包括易于配置，快速启动时间，可测试性以及在开发过程中快速演化SQL的能力。

3.9.2 使用Spring XML创建嵌入式数据库

如果要在Spring ApplicationContext中将嵌入式数据库实例作为Bean公开，则可以在spring-jdbc命名空间中使用Embedded-database标记：

```
<jdbc:embedded-database id="dataSource" generate-name="true">
    <jdbc:script location="classpath:schema.sql"/>
    <jdbc:script location="classpath:test-data.sql"/>
</jdbc:embedded-database>
```

前面的配置创建了一个嵌入式HSQL数据库，该数据库中使用SQL从类路径根目录中的schema.sql和test-data.sql资源中填充。 另外，作为最佳实践，将为嵌入式数据库分配一个唯一生成的名称。 嵌入式数据库作为javax.sql.DataSource类型的Bean在Spring容器中使用，然后可以根据需要将其注入到数据访问对象中。

3.9.3 以编程方式创建嵌入式数据库

EmbeddedDatabaseBuilder类提供了一种流畅的API，用于以编程方式构造嵌入式数据库。 当您需要在独立环境或独立集成测试中创建嵌入式数据库时，可以使用此方法，如以下示例所示：

```
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()
```

您还可以使用EmbeddedDatabaseBuilder通过Java配置创建嵌入式数据库，如以下示例所示：

```
@Configuration
public class DataSourceConfig {

    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
                .generateUniqueName(true)
                .setType(H2)
```

更多教程请访问:www.flydean.com

```
                .setScriptEncoding("UTF-8")
                .ignoreFailedDrops(true)
                .addScript("schema.sql")
                .addScripts("user_data.sql", "country_data.sql")
                .build();
    }
}
```

3.9.4 选择嵌入式数据库类型

本节介绍如何选择Spring支持的三个嵌入式数据库之一。它包括以下主题：

- 使用HSQL
- 使用H2
- 使用Derby

使用HSQL

Spring支持HSQL 1.8.0及更高版本。如果未明确指定类型，则HSQL是默认的嵌入式数据库。要明确指定HSQL，请将嵌入式数据库标记的type属性设置为HSQL。如果使用构建器API，请使用EmbeddedDatabaseType.HSQL调用setType（EmbeddedDatabaseType）方法。

使用H2

Spring支持H2数据库。要启用H2，请将嵌入式数据库标记的type属性设置为H2。如果使用构建器API，请使用EmbeddedDatabaseType.H2调用setType（EmbeddedDatabaseType）方法。

使用Derby Spring支持Apache Derby 10.5及更高版本。要启用Derby，请将嵌入式数据库标记的type属性设置为DERBY。如果使用构建器API，请使用EmbeddedDatabaseType.DERBY调用setType（EmbeddedDatabaseType）方法。

3.9.5 使用嵌入式数据库测试数据访问逻辑

嵌入式数据库提供了一种轻量级的方法来测试数据访问代码。下一个示例是使用嵌入式数据库的数据访问集成测试模板。当嵌入式数据库不需要在测试类之间重用时，使用这种模板可以一次性使用。但是，如果您希望创建一个在测试套件中共享的嵌入式数据库，请考虑使用Spring TestContext Framework并将嵌入式数据库配置为Spring ApplicationContext中的Bean，如使用Spring XML创建嵌入式数据库和创建一个嵌入式数据库中所述。以编程方式嵌入数据库。以下清单显示了测试模板：

```
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( /* ... */ );
    }
```

更多教程请访问:www.flydean.com

```
    @After
    public void tearDown() {
        db.shutdown();
    }

}
```

3.9.6 为嵌入式数据库生成唯一名称

如果开发团队的测试套件无意中尝试重新创建同一数据库的其他实例，则开发团队经常会遇到错误。如果XML配置文件或@Configuration类负责创建嵌入式数据库，然后在同一测试套件（即，同一JVM进程）内的多个测试场景中重用相应的配置，则这很容易发生。集成测试针对其ApplicationContext配置仅在哪些bean定义配置文件处于活动状态方面有所不同的嵌入式数据库进行。

造成此类错误的根本原因是，如果未另行指定，Spring的EmbeddedDatabaseFactory（由<jdbc：embedded-database> XML命名空间元素和EmbeddedDatabaseBuilder for Java配置在内部使用）会将嵌入式数据库的名称设置为testdb。对于<jdbc：embedded-database>，通常为嵌入式数据库分配一个名称，该名称等于bean的ID（通常是类似于dataSource的名称）。因此，随后创建嵌入式数据库的尝试不会产生新的数据库。取而代之的是，相同的JDBC连接URL被重用，并且尝试创建新的嵌入式数据库实际上指向的是从相同配置创建的现有嵌入式数据库。

为了解决这个常见问题，Spring Framework 4.2提供了对生成嵌入式数据库的唯一名称的支持。要启用使用生成的名称，请使用以下选项之一。

- EmbeddedDatabaseFactory.setGenerateUniqueDatabaseName（）
- EmbeddedDatabaseBuilder.generateUniqueName（）
- <jdbc：embedded-database generate-name =“ true”……>

3.9.7 扩展嵌入式数据库支持

您可以通过两种方式扩展Spring JDBC嵌入式数据库的支持：

- 实现EmbeddedDatabaseConfigurer以支持新的嵌入式数据库类型。
- 实现DataSourceFactory以支持新的DataSource实现，例如用于管理嵌入式数据库连接的连接池。

我们鼓励您在GitHub Issues上为Spring社区贡献扩展。

更多教程请访问:www.flydean.com

3.10初始化数据源

org.springframework.jdbc.datasource.init包提供了对初始化现有DataSource的支持。 嵌入式数据库支持提供了一种为应用程序创建和初始化数据源的选项。 但是，有时您可能需要初始化在某处的服务器上运行的实例。

3.10.1 使用Spring XML初始化数据库

如果要初始化数据库，并且可以提供对DataSource bean的引用，则可以在spring-jdbc命名空间中使用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>
```

前面的示例对数据库运行两个指定的脚本。第一个脚本创建模式，第二个脚本用测试数据集填充表。脚本位置还可以是带有通配符的模式，这些通配符具有用于Spring中资源的常用Ant样式（例如，classpath ： */com / foo / ** / sql / -* data.sql）。如果使用模式，则脚本以其URL或文件名的词法顺序运行。

数据库初始化程序的默认行为是无条件运行所提供的脚本。这可能并不总是您想要的。例如，如果您对已经有测试数据的数据库运行脚本。通过遵循先创建表然后插入数据的通用模式（如前所示），可以减少意外删除数据的可能性。如果表已经存在，则第一步失败。

但是，为了更好地控制现有数据的创建和删除，XML名称空间提供了一些其他选项。第一个是用于打开和关闭初始化的标志。您可以根据环境进行设置（例如，从系统属性或环境Bean中提取布尔值）。以下示例从系统属性获取值：

```
<jdbc:initialize-database data-source="dataSource"
    enabled="#{systemProperties.INITIALIZE_DATABASE}">
    <jdbc:script location="..."/>
</jdbc:initialize-database>
```

第二种选择是容忍失败。 为此，您可以控制初始化程序忽略脚本执行的SQL中某些错误的能力，如以下示例所示：

```
<jdbc:initialize-database data-source="dataSource" ignore-failures="DROPS">
    <jdbc:script location="..."/>
</jdbc:initialize-database>
```

在前面的示例中，我们说我们期望有时脚本是针对空数据库运行的，并且脚本中有一些DROP语句可能因此失败。 因此失败的SQL DROP语句将被忽略，但是其他失败将导致异常。 如果您的SQL方言不支持DROP .....如果存在（或类似），但您想在重新创建之前无条件删除所有测试数据，则此功能很有用。 在那种情况下，第一个脚本通常是一组DROP语句，然后是一组CREATE语句。

可以将ignore-failures选项设置为NONE（默认值），DROPS（忽略失败的丢弃）或ALL（忽略所有失败）。

每个语句都应用; 或换行结束。 您可以全局控制该脚本，也可以按脚本控制，如以下示例所示：

```
<jdbc:initialize-database data-source="dataSource" separator="@@">
    <jdbc:script location="classpath:com/myapp/sql/db-schema.sql" separator=";"/>
    <jdbc:script location="classpath:com/myapp/sql/db-test-data-1.sql"/>
    <jdbc:script location="classpath:com/myapp/sql/db-test-data-2.sql"/>
</jdbc:initialize-database>
```

更多教程请访问:www.flydean.com

在此示例中，两个测试数据脚本使用@@作为语句分隔符，只有db-schema.sql使用;。此配置指定默认分隔符为@@，并覆盖db-schema脚本的默认分隔符。

如果您需要比从XML名称空间获得更多控制权，则可以直接使用DataSourceInitializer并将其定义为应用程序中的组件。

初始化依赖于数据库的其他组件

大量的应用程序（那些在Spring上下文启动之后才使用数据库的应用程序）可以使用数据库初始化程序，而不会带来更多麻烦。如果您的应用程序不是其中之一，则可能需要阅读本节的其余部分。

数据库初始化程序依赖于DataSource实例，并运行其初始化回调中提供的脚本（类似于XML bean定义中的init方法，组件中的@PostConstruct方法或实现InitializingBean的组件中的afterPropertiesSet（）方法。）。如果其他Bean依赖于同一数据源并在初始化回调中使用该数据源，则可能存在问题，因为尚未初始化数据。一个常见的例子是一个高速缓存，它会在应用程序启动时急于初始化并从数据库加载数据。

要解决此问题，您有两个选择：将高速缓存初始化策略更改为以后的阶段，或者确保首先初始化数据库初始化程序。

如果应用程序在您的控制之下，则更改缓存初始化策略可能很容易，否则就不那么容易。有关如何实现这一点的一些建议包括：

- 使缓存在首次使用时延迟初始化，从而缩短了应用程序的启动时间。
- 让您的缓存或单独的组件初始化缓存实现Lifecycle或SmartLifecycle。当应用程序上下文启动时，您可以通过设置其SmartStartup标志来自动启动SmartLifecycle，并且可以通过在封闭的上下文中调用ConfigurableApplicationContext.start（）来手动启动Lifecycle。
- 使用Spring ApplicationEvent或类似的自定义观察器机制来触发缓存初始化。 ContextRefreshedEvent总是在准备好使用时（在所有bean都初始化之后）由上下文发布，因此通常是一个有用的钩子（默认情况下，SmartLifecycle的工作方式）。

确保首先初始化数据库初始化程序也很容易。关于如何实现这一点的一些建议包括：

- 依靠Spring BeanFactory的默认行为，即按注册顺序初始化bean。通过采用XML配置中的一组<import />元素（对应用程序模块进行排序）的通用做法，并确保首先列出数据库和数据库初始化。
- 通过将数据源和使用它的业务组件分开，并通过将它们放在单独的ApplicationContext实例中来控制它们的启动顺序（例如，父上下文包含DataSource，子上下文包含业务组件）。这种结构在Spring Web应用程序中很常见，但可以更普遍地应用。

更多教程请访问:www.flydean.com

4.ORM

本节介绍使用对象关系映射（ORM）时的数据访问。

更多教程请访问:www.flydean.com

4.1Spring ORM介绍

Spring Framework支持与Java Persistence API（JPA）集成，并支持用于资源管理，数据访问对象（DAO）实现和事务策略的本地Hibernate。例如，对于Hibernate，它具有一流的支持以及一些便捷的IoC功能，可解决许多典型的Hibernate集成问题。您可以通过“依赖关系注入”为OR（对象关系）映射工具配置所有受支持的功能。他们可以参与Spring的资源和交易管理，并且符合Spring的通用交易和DAO异常层次结构。推荐的集成样式是针对普通的Hibernate或JPA API编写DAO。

当您创建数据访问应用程序时，Spring会为您选择的ORM层显着增强。您可以根据需要利用尽可能多的集成支持，并且应该将这种集成工作与内部构建类似基础架构的成本和风险进行比较。不管使用哪种技术，您都可以像使用库一样使用许多ORM支持，因为所有内容都被设计为一组可重用的JavaBean。 Spring IoC容器中的ORM有助于配置和部署。因此，本节中的大多数示例都显示了Spring容器内部的配置。

使用Spring框架创建ORM DAO的好处包括：

- 更容易的测试。 Spring的IoC方法使交换Hibernate SessionFactory实例，JDBC DataSource实例，事务管理器和映射对象实现（如果需要）的实现和配置位置变得容易。反过来，这使得隔离每个与持久性相关的代码片段的测试变得容易得多。
- 常见的数据访问异常。 Spring可以包装您的ORM工具中的异常，将其从专有（可能已检查）的异常转换为通用的运行时DataAccessException层次结构。通过此功能，您可以仅在适当的层中处理大多数不可恢复的持久性异常，而不会烦人样板捕获，抛出和异常声明。您仍然可以根据需要捕获和处理异常。请记住，JDBC异常（包括特定于DB的方言）也将转换为相同的层次结构，这意味着您可以在一致的编程模型中使用JDBC执行某些操作。
- 常规资源管理。 Spring应用程序上下文可以处理Hibernate SessionFactory实例，JPA EntityManagerFactory实例，JDBC DataSource实例和其他相关资源的位置和配置。这使得这些值易于管理和更改。 Spring提供了对持久性资源的高效，便捷和安全的处理。例如，使用Hibernate的相关代码通常需要使用相同的Hibernate Session，以确保效率和适当的事务处理。通过通过Hibernate SessionFactory公开当前Session，Spring可以轻松地透明地创建Session并将其绑定到当前线程。因此，对于任何本地或JTA事务环境，Spring都解决了典型的Hibernate使用中的许多长期问题。
- 集成交易管理。您可以通过@Transactional批注或通过在XML配置文件中显式配置事务AOP建议，使用声明性的，面向方面的编程（AOP）风格的方法拦截器包装ORM代码。在这两种情况下，都会为您处理事务语义和异常处理（回滚等）。如资源和事务管理中所述，您还可以交换各种事务管理器，而不会影响与ORM相关的代码。例如，您可以在两种情况下使用相同的完整服务（例如声明性事务）在本地事务和JTA之间进行交换。此外，与JDBC相关的代码可以与您用于执行ORM的代码完全事务集成。这对于不适合ORM（例如批处理和BLOB流）但仍需要与ORM操作共享常见事务的数据访问很有用。

更多教程请访问:www.flydean.com

4.2ORM集成的一般注意事项

本节重点介绍适用于所有ORM技术的注意事项。Hibernate部分提供了更多详细信息，并在具体上下文中显示了这些功能和配置。

Spring的ORM集成的主要目标是清晰的应用程序分层（具有任何数据访问和事务技术）以及松散耦合应用程序对象-不再需要业务服务依赖数据访问或事务处理策略，不再需要硬编码的资源查找，更多难以替换的单例，不再需要自定义服务注册表。目标是采用一种简单且一致的方法来连接应用程序对象，使它们尽可能可重用，并且尽可能不受容器依赖。所有单独的数据访问功能都可以单独使用，但可以与Spring的应用程序上下文概念很好地集成，从而提供基于XML的配置和对不需要Spring感知的纯JavaBean实例的交叉引用。在典型的Spring应用程序中，许多重要的对象是JavaBean,包括：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。

4.2.1 资源和事务管理

典型的业务应用程序中充斥着重复的资源管理代码。许多项目试图发明自己的解决方案，有时为了编程方便而牺牲了对故障的正确处理。 Spring提倡简单的解决方案来进行适当的资源处理，即通过在JDBC情况下进行模板化以及为ORM技术应用AOP拦截器来实现IoC。

基础结构提供适当的资源处理，并将特定的API异常适当地转换为未经检查的基础结构异常层次结构。 Spring引入了DAO异常层次结构，适用于任何数据访问策略。对于直接JDBC，上一节中提到的JdbcTemplate类提供连接处理以及将SQLException正确转换为DataAccessException层次结构，包括将特定于数据库的SQL错误代码转换为有意义的异常类。对于ORM技术，请参阅下一节以了解如何获得相同的异常转换好处。

在事务管理方面，JdbcTemplate类通过相应的Spring事务管理器挂接到Spring事务支持并支持JTA和JDBC事务。对于受支持的ORM技术，Spring通过Hibernate和JPA事务管理器提供了Hibernate和JPA支持以及JTA支持。有关事务支持的详细信息，请参见事务管理一章。

4.2.2 异常事务

在DAO中使用Hibernate或JPA时，必须决定如何处理持久性技术的本机异常类。根据技术的不同，DAO会抛出HibernateException或PersistenceException的子类。这些异常都是运行时异常，不必声明或捕获。您可能还必须处理IllegalArgumentException和IllegalStateException。这意味着调用者只能将异常视为一般致命的，除非他们想依赖于持久性技术自己的异常结构。如果不将调用者与实现策略联系在一起，则无法捕获特定原因（例如乐观锁定失败）。这种权衡可能对于基于ORM的应用程序或不需要任何特殊异常处理（或两者）都可以接受。但是，Spring允许通过@Repository批注透明地应用异常转换。以下示例（一个用于Java配置，一个用于XML配置）显示了如何执行此操作：

```
@Repository
public class ProductDaoImpl implements ProductDao {

    // class body here...

}
~~

~~~xml
<beans>

    <!-- Exception translation bean post processor -->
    <bean class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"/>

    <bean id="myProductDao" class="product.ProductDaoImpl"/>
```

更多教程请访问:www.flydean.com

```
</beans>
```

后处理器自动查找所有异常转换器（PersistenceExceptionTranslator接口的实现），并建议所有标有@Repository批注的bean，以便发现的转换器可以拦截适当的转换并将其应用于引发的异常。

总之，您可以基于纯持久性技术的API和注解来实现DAO，同时仍可以从Spring管理的事务，依赖项注入和透明的异常转换（如果需要）到Spring的自定义异常层次结构中受益。

更多教程请访问:www.flydean.com

4.3Hibernate

我们从对Spring环境中的Hibernate 5的介绍开始，用它来演示Spring用于集成OR映射器的方法。 本节详细讨论了许多问题，并展示了DAO实现和事务划分的不同变体。 这些模式中的大多数都可以直接转换为所有其他受支持的ORM工具。 然后，本章后面的部分将介绍其他ORM技术，并显示一些简短的示例。

> 从Spring Framework 5.0开始，Spring需要Hibernate ORM 4.3或更高版本才能提供JPA支持，甚至需要Hibernate ORM 5.0+才能针对本机Hibernate Session API进行编程。 请注意，Hibernate团队不再维护5.1之前的任何版本，并且可能很快会专注于5.3+。

4.3.1 Spring容器中的SessionFactory设置

为了避免将应用程序对象与硬编码的资源查找绑定在一起，可以在Spring容器中将资源（例如JDBC DataSource或Hibernate SessionFactory）定义为bean。 需要访问资源的应用程序对象通过bean引用接收对此类预定义实例的引用，如下一节中的DAO定义所示。

以下XML应用程序上下文定义摘录显示了如何在其上设置JDBC数据源和Hibernate SessionFactory：

```
<beans>

    <bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
        <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
        <property name="url" value="jdbc:hsqldb:hsql://localhost:9001"/>
        <property name="username" value="sa"/>
        <property name="password" value=""/>
    </bean>

    <bean id="mySessionFactory" class="org.springframework.orm.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>
```

从本地Jakarta Commons DBCP BasicDataSource切换到位于JNDI的DataSource（通常由应用程序服务器管理）仅是配置问题，如以下示例所示：

```
<beans>
    <jee:jndi-lookup id="myDataSource" jndi-name="java:comp/env/jdbc/myds"/>
</beans>
```

您还可以使用Spring的JndiObjectFactoryBean / <jee：jndi-lookup>检索并公开它，从而访问位于JNDI的SessionFactory。但是，这通常在EJB上下文之外并不常见。

> Spring还提供了LocalSessionFactoryBuilder变体，可与@Bean样式配置和编程设置无缝集成（不涉及FactoryBean）。

更多教程请访问:www.flydean.com

> LocalSessionFactoryBean和LocalSessionFactoryBuilder都支持后台引导，并且Hibernate初始化与给定引导执行程序（例如SimpleAsyncTaskExecutor）上的应用程序引导线程并行运行。在LocalSessionFactoryBean上，可以通过bootstrapExecutor属性使用。在程序化LocalSessionFactoryBuilder上，有一个重载的buildSessionFactory方法，该方法带有引导执行程序参数。
>
> 从Spring Framework 5.1开始，此类native Hibernate设置还可以在native Hibernate访问时候公开用于标准JPA交互的JPA EntityManagerFactory。有关详细信息，请参见JPA的nativeHibernate设置。

4.3.2 基于Plain Hibernate API实现DAO

Hibernate具有称为上下文会话的功能，其中，Hibernate本身在每个事务中管理一个当前会话。 这大致相当于Spring在每个事务中同步一个Hibernate Session。 基于简单的Hibernate API，相应的DAO实现类似于以下示例：

```
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();
    }
}
```

除了将SessionFactory保留在实例变量中之外，此样式与Hibernate参考文档和示例类似。 我们强烈建议您在Hibernate的CaveatEmptor示例应用程序中的老式静态HibernateUtil类上使用基于实例的设置。 （通常，除非绝对必要，否则不要在静态变量中保留任何资源。）

前面的DAO示例遵循依赖项注入模式。 就像针对Spring的HibernateTemplate进行编码一样，它非常适合Spring IoC容器。 您还可以在纯Java中设置这种DAO（例如，在单元测试中）。 为此，将其实例化并使用所需的工厂引用调用setSessionFactory（..）。 作为Spring bean的定义，DAO类似于以下内容：

```
<beans>

    <bean id="myProductDao" class="product.ProductDaoImpl">
        <property name="sessionFactory" ref="mySessionFactory"/>
    </bean>

</beans>
```

这种DAO样式的主要优点是它仅依赖于Hibernate API。不需要导入任何Spring类。从非侵入性的角度来看，这很有吸引力，并且对于Hibernate开发人员而言可能更自然。

但是，DAO抛出普通的HibernateException（未经检查，因此不必声明或捕获），这意味着调用方只能将异常视为一般致命的消息-除非他们希望依赖于Hibernate自己的异常层次结构。如果不将调用者与实现策略联系在一起，则无法捕获特定原因（例如乐观锁定失败）。这种权衡对于基于Hibernate的应用程序，不需要任何特殊异常处理或两者都可以接受。

幸运的是，Spring的LocalSessionFactoryBean支持任何Spring交易策略的Hibernate的SessionFactory.getCurrentSession（）方法，即使使用HibernateTransactionManager，也返回当前的Spring管理的交易会话。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）。无论您使用Spring的JtaTransactionManager，EJB容器管理的交易（CMT）还是JTA，此行为均适用。

更多教程请访问:www.flydean.com

总之，您可以基于普通的Hibernate API实现DAO，同时仍然能够参与Spring管理的事务。

4.3.3 声明式事务划分

我们建议您使用Spring的声明式事务支持，该支持使您可以将Java代码中的显式事务划分API调用替换为AOP事务拦截器。 您可以使用Java批注或XML在Spring容器中配置此事务拦截器。 这种声明式事务处理功能使您可以使业务服务免于重复的事务划分代码，并专注于添加业务逻辑，这是应用程序的真正价值。

> 在继续之前，我们强烈建议您阅读声明式事务管理（如果您尚未阅读的话）。

您可以使用@Transactional批注来批注服务层，并指示Spring容器查找这些批注并为这些批注的方法提供事务性语义。 以下示例显示了如何执行此操作：

```
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();
    }

}
```

在容器中，您需要设置PlatformTransactionManager实现（作为bean）和<tx：annotation-driven />条目，并在运行时选择@Transactional处理。 以下示例显示了如何执行此操作：

```
<?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
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/tx
        https://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/aop
        https://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>
```

更多教程请访问:www.flydean.com

```
</beans>
```

4.3.4 程序化交易划分

您可以在应用程序的更高级别中划分事务，在跨越任意数量的操作的较低级别数据访问服务之上。 对周围业务服务的实施也没有限制。 它只需要一个Spring PlatformTransactionManager。 同样，后者可以来自任何地方，但最好通过setTransactionManager（..）方法作为bean的引用。 另外，应通过setProductDao（..）方法设置productDAO。 以下几对代码片段显示了Spring应用程序上下文中的事务管理器和业务服务定义，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允许将任何已检查的应用程序异常与回调代码一起引发，而TransactionTemplate仅限于回调中未检查的异常。 如果未检查的应用程序异常或应用程序将事务标记为仅回滚（通过设置TransactionStatus），则TransactionTemplate会触发回滚。 默认情况下，TransactionInterceptor的行为方式相同，但允许每种方法配置可回滚策略。

4.3.5 交易管理策略

TransactionTemplate和TransactionInterceptor都将实际的事务处理委托给PlatformTransactionManager实例（对于Hibernate，可以通过使用ThreadLocal会话将其委托给HibernateTransactionManager（对于单个Hibernate SessionFactory））或JtaTransactionManager（委托给容器的JTA子系统）。应用程序。您甚至可以使用自定义

更多教程请访问:www.flydean.com

PlatformTransactionManager实现。从本机Hibernate事务管理切换到JTA（例如，面对应用程序的某些部署的分布式事务要求时）仅是配置问题。您可以将Hibernate事务管理器替换为Spring的JTA事务实现。事务划分和数据访问代码都无需更改即可工作，因为它们使用通用的事务管理API。

对于跨多个Hibernate会话工厂的分布式事务，可以将JtaTransactionManager作为具有多个LocalSessionFactoryBean定义的事务策略进行组合。然后，每个DAO都会将一个特定的SessionFactory引用传递到其相应的bean属性中。如果所有基础JDBC数据源都是事务性容器数据源，那么只要使用JtaTransactionManager作为策略，业务服务就可以在任何数量的DAO和任何会话工厂之间划分事务，而无需特别注意。

HibernateTransactionManager和JtaTransactionManager都允许使用Hibernate进行正确的JVM级别的缓存处理，而无需特定于容器的事务管理器查找或JCA连接器（如果您不使用EJB来初始化事务）。

HibernateTransactionManager可以将Hibernate JDBC连接导出到特定DataSource的纯JDBC访问代码。此功能允许使用混合的Hibernate和JDBC数据访问进行高级事务划分，而无需JTA，前提是您仅访问一个数据库。如果已通过LocalSessionFactoryBean类的dataSource属性设置了传入的SessionFactory和DataSource，则HibernateTransactionManager会将Hibernate事务自动公开为JDBC事务。另外，您可以通过HibernateTransactionManager类的dataSource属性显式指定应该为其公开事务的DataSource。

4.3.6 比较容器管理的资源和本地定义的资源

您可以在容器管理的JNDI SessionFactory和本地定义的JNDI SessionFactory之间切换，而无需更改单行应用程序代码。将资源定义保留在容器中还是在应用程序中本地保留，主要取决于您使用的事务策略。与Spring定义的本地SessionFactory相比，手动注册的JNDI SessionFactory没有任何好处。通过Hibernate的JCA连接器部署SessionFactory可以增加参与Java EE服务器的管理基础结构的附加价值，但是并不能增加任何实际价值。

Spring的交易支持不限于容器。当使用除JTA之外的任何其他策略配置时，事务支持还可以在独立或测试环境中工作。尤其是在单数据库事务的典型情况下，Spring的单资源本地事务支持是JTA的轻量级功能强大的替代方案。当您使用本地EJB无状态会话Bean驱动事务时，即使您仅访问单个数据库并且仅使用无状态会话Bean通过容器管理的事务来提供声明性事务，也要依赖EJB容器和JTA。以编程方式直接使用JTA还需要Java EE环境。就JTA本身和JNDI DataSource实例而言，JTA不仅仅涉及容器依赖项。对于非Spring，JTA驱动的Hibernate事务，您必须使用Hibernate JCA连接器或额外的Hibernate事务代码，并将TransactionManagerLookup配置为正确的JVM级别的缓存。

如果Spring访问的事务访问单个数据库，则它们可以与本地定义的Hibernate SessionFactory一起使用，也可以与本地JDBC数据源一起使用。因此，只有在分配了交易需求时才需要使用Spring的JTA交易策略。 JCA连接器需要特定于容器的部署步骤，并且首先需要（显然）JCA支持。与部署具有本地资源定义和Spring驱动的事务的简单Web应用程序相比，此配置需要更多的工作。另外，如果使用（例如）不提供JCA的WebLogic Express，则通常需要容器的企业版。具有跨单个数据库的本地资源和事务的Spring应用程序可以在任何Java EE Web容器（没有JTA，JCA或EJB）中工作，例如Tomcat，Resin甚至普通Jetty。此外，您可以轻松地在桌面应用程序或测试套件中重用这样的中间层。

考虑到所有因素，如果您不使用EJB，请坚持使用本地SessionFactory设置和Spring的HibernateTransactionManager或JtaTransactionManager。您将获得所有好处，包括适当的事务性JVM级别的缓存和分布式事务，而不会给容器部署带来不便。通过JCA连接器对Hibernate SessionFactory进行JNDI注册仅在与EJB结合使用时才增加价值。

4.3.7 Hibernate虚假的应用程序服务器警告

在具有非常严格的XADataSource实现的某些JTA环境中（当前仅某些WebLogic Server和WebSphere版本），当配置Hibernate而不考虑该环境的JTA PlatformTransactionManager对象时，虚假警告或异常可能会出现在应用程序服务器日志中。 这些警告或异常指示正在访问的连接不再有效或JDBC访问不再有效，这可能是因为事务不再有效。 例如，这是WebLogic的实际异常：

```
java.sql.SQLException: The transaction is no longer active - status: 'Committed'. No
further JDBC access is allowed within this transaction.
```

更多教程请访问:www.flydean.com

您可以通过使Hibernate知道与之同步的JTA PlatformTransactionManager实例（以及Spring实例）来解决此警告。您可以通过以下两种方式执行此操作：

- 如果在您的应用程序上下文中，您已经直接获取了JTA PlatformTransactionManager对象（大概是通过Jndi通过JndiObjectFactoryBean或<jee：jndi-lookup>从JNDI获取）并将其提供给例如Spring的JtaTransactionManager，则最简单的方法是指定对将此JTA PlatformTransactionManager实例定义为LocalSessionFactoryBean的jtaTransactionManager属性的值的bean。然后，Spring使该对象可用于Hibernate。
- 很可能您还没有JTA PlatformTransactionManager实例，因为Spring的JtaTransactionManager可以自行找到它。因此，您需要配置Hibernate以直接查找JTA PlatformTransactionManager。您可以通过在Hibernate配置中配置特定于应用程序服务器的TransactionManagerLookup类来实现此目的，如Hibernate手册中所述。

本节的其余部分描述了在Hibernate不了解JTA PlatformTransactionManager的情况下发生的事件序列。

当未对JTA PlatformTransactionManager进行任何意识的Hibernate配置时，在JTA事务提交时会发生以下事件：

- JTA事务提交。
- Spring的JtaTransactionManager已同步到JTA事务，因此JTA事务管理器通过afterCompletion回调对其进行回调。
- 在其他活动中，此同步可以通过Spring的Hibernate的afterTransactionCompletion回调（用于清除Hibernate缓存）触发Spring到Hibernate的回调，然后在Hibernate会话上进行显式的close（）调用，这将导致Hibernate尝试进行close（） JDBC连接。
- 在某些环境中，此Connection.close（）调用随后触发警告或错误，因为应用程序服务器不再认为Connection可用，因为事务已被提交。

当Hibernate配置为具有JTA PlatformTransactionManager意识时，在JTA事务提交时会发生以下事件：

- JTA事务已准备好提交。
- Spring的JtaTransactionManager已同步到JTA事务，因此JTA事务管理器通过beforeCompletion回调来调用该事务。
- Spring意识到，Hibernate本身已同步到JTA事务，并且其行为与以前的场景不同。假设完全需要关闭Hibernate会话，Spring现在将其关闭。
- JTA事务提交。
- Hibernate已同步到JTA事务，因此JTA事务管理器通过afterCompletion回调来调用该事务，并且可以正确清除其缓存。

更多教程请访问:www.flydean.com

4.4JPA

可以在org.springframework.orm.jpa包下获得的Spring JPA以类似于与Hibernate集成的方式，为Java Persistence API提供了全面的支持，同时使用到了底层实现以提供附加功能。

4.4.1 在Spring环境中设置JPA的三个选项

Spring JPA支持提供了三种设置JPA EntityManagerFactory的方法，供应用程序用来获取实体管理器。

- 使用LocalEntityManagerFactoryBean
- 从JNDI获取EntityManagerFactory
- 使用LocalContainerEntityManagerFactoryBean

使用LocalEntityManagerFactoryBean

您只能在简单的部署环境（例如独立应用程序和集成测试）中使用此选项。

LocalEntityManagerFactoryBean创建一个EntityManagerFactory，适用于应用程序仅使用JPA进行数据访问的简单部署环境。 工厂bean使用JPA PersistenceProvider自动检测机制（根据JPA的Java SE自举），并且在大多数情况下，仅要求您指定持久性单元名称。 以下XML示例配置了这样的bean：

```
<beans>
    <bean id="myEmf" class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
        <property name="persistenceUnitName" value="myPersistenceUnit"/>
    </bean>
</beans>
```

这种形式的JPA部署是最简单和最有限的。 您不能引用现有的JDBC DataSource bean定义，并且不存在对全局事务的支持。 此外，持久类的编织（字节码转换）是特定于提供程序的，通常需要在启动时指定特定的JVM代理。 该选项仅对于设计了JPA规范的独立应用程序和测试环境就足够了。

从JNDI获取EntityManagerFactory

部署到Java EE服务器时，可以使用此选项。 查看服务器的文档，以了解如何将自定义JPA提供程序部署到服务器中，以允许使用与服务器默认设置不同的提供程序。

从JNDI获得EntityManagerFactory（例如在Java EE环境中）是更改XML配置的问题，如以下示例所示：

```
<beans>
    <jee:jndi-lookup id="myEmf" jndi-name="persistence/myPersistenceUnit"/>
</beans>
```

此操作假定标准Java EE引导。 Java EE服务器自动检测Java EE部署描述符中的持久性单元（实际上是应用程序jar中的META-INF / persistence.xml文件）和persistence-unit-ref条目，并定义环境命名这些持久性单元的上下文位置。

在这种情况下，整个持久性单元部署，包括持久性类的编织（字节码转换），都取决于Java EE服务器。 JDBC数据源是通过META-INF / persistence.xml文件中的JNDI位置定义的。 EntityManager事务与服务器的JTA子系统集成在一起。 Spring仅使用获得的EntityManagerFactory，通过依赖关系注入将其传递给应用程序对象，并管理持久性单元的事务（通常通过JtaTransactionManager）。

如果在同一应用程序中使用多个持久性单元，则此类JNDI检索的持久性单元的bean名称应与应用程序用来引用它们的持久性单元名称匹配（例如，在@PersistenceUnit和@PersistenceContext批注中）。

更多教程请访问:www.flydean.com

使用LocalContainerEntityManagerFactoryBean 您可以在基于Spring的应用程序环境中将此选项用于完整的JPA功能。这包括Web容器（例如Tomcat），独立应用程序以及具有复杂持久性要求的集成测试。

> 如果要专门配置Hibernate设置，则直接的替代方法是使用Hibernate 5.2或5.3并设置本机Hibernate LocalSessionFactoryBean而不是纯JPA LocalContainerEntityManagerFactoryBean，从而使其与JPA访问代码以及本机Hibernate访问代码交互。 有关详细信息，请参见用于JPA交互的本机Hibernate设置。

LocalContainerEntityManagerFactoryBean可以完全控制EntityManagerFactory的配置，适用于需要细粒度自定义的环境。 LocalContainerEntityManagerFactoryBean基于persistence.xml文件，提供的dataSourceLookup策略和指定的loadTimeWeaver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编织过程。 以下示例显示了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 />快捷方式表示不应对带注解的实体类进行扫描。显式的“ true”值（<exclude-unlist-classes> true </ exclude-unlist-classes />）也意味着不进行扫描。 <exclude-unlisted-classes> false </ exclude-unlisted-classes />会触发扫描。但是，如果您希望进行实体类扫描，我们建议省略exclude-unlisted-classes元素。

使用LocalContainerEntityManagerFactoryBean是最强大的JPA设置选项，可以在应用程序中进行灵活的本地配置。它支持到现有JDBC数据源的链接，同时支持本地和全局事务，等等。但是，这也对运行时环境提出了要求，例如，如果持久性提供程序要求字节码转换，则需要具有可用的可编织类加载器。

此选项可能与Java EE服务器的内置JPA功能冲突。在完整的Java EE环境中，请考虑从JNDI获取您的EntityManagerFactory。或者，在LocalContainerEntityManagerFactoryBean定义上指定自定义persistenceXmlLocation（例如，META-INF / my-persistence.xml），并在应用程序jar文件中仅包含具有该名称的描述符。由于Java EE服务器仅查找默认的META-INF / persistence.xml文件，因此它会忽略此类自定义持久性单元，因此避免了与Spring预先驱动的JPA设置发生冲突。 （例如，这适用于Resin 3.1。）

> 什么时候需要加载时编织?
>
> 并非所有的JPA提供程序都需要JVM代理。Hibernate是一个没有的例子。如果您的提供程序不需要代理，或者您还有其他选择，例如在构建时通过自定义编译器或Ant任务应用增强功能，则不应使用加载时织布器。

LoadTimeWeaver接口是Spring提供的类，该类使JPA ClassTransformer实例可以以特定方式插入，具体取决于环境是Web容器还是应用程序服务器。通过代理连接ClassTransformers通常效率不高。代理针对整个虚拟机工作，并检查每个已加载的类，这在生产服务器环境中通常是不希望的。

Spring为各种环境提供了许多LoadTimeWeaver实现，让ClassTransformer实例仅应用于每个类加载器，而不应用于每个VM。

请参阅AOP章节中的Spring配置以获取有关LoadTimeWeaver实现及其设置的更多信息，这些实现是通用的或针对各种平台（例如Tomcat，JBoss和WebSphere）定制的。

如Spring配置中所述，您可以通过使用context：load-time-weaver XML元素的@EnableLoadTimeWeaving批注来配置上下文范围的LoadTimeWeaver。所有JPA LocalContainerEntityManagerFactoryBean实例都会自动选择此类全局编织器。下面的示例显示了设置加载时间编织器，提供对平台的自动检测（例如Tomcat的具有编织功能的类加载器或Spring的JVM代理）以及将编织器自动传播到所有可感知编织器的bean的首选方法：

```
<context:load-time-weaver/>
<bean id="emf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    ...
</bean>
```

但是，您可以根据需要通过loadTimeWeaver属性手动指定专用的编织器，如以下示例所示：

```
<bean id="emf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="loadTimeWeaver">
        <bean class="org.springframework.instrument.classloading.ReflectiveLoadTimeWeaver"/>
    </property>
</bean>
```

无论LTW的配置方式如何，通过使用此技术，依赖于检测的JPA应用程序都可以在目标平台（例如Tomcat）中运行，而无需代理。 当托管应用程序依赖于不同的JPA实现时，这尤其重要，因为JPA转换器仅在类加载器级别应用，因此彼此隔离。

处理多个持久性单元

对于依赖多个持久性单元位置的应用程序（例如，存储在类路径中的各种JARS中），Spring提供了PersistenceUnitManager充当中央存储库，并避免了持久性单元发现过程，这可能会很昂贵。 默认实现允许指定多个位置。 解析这些位置，然后通过持久性单元名称进行检索。 （默认情况下，在类路径中搜索META-INF / persistence.xml文件。）以下示例配置多个位置：

```
<bean id="pum" class="org.springframework.orm.jpa.persistenceunit.DefaultPersistenceUnitManager">
    <property name="persistenceXmlLocations">
        <list>
            <value>org/springframework/orm/jpa/domain/persistence-multi.xml</value>
            <value>classpath:/my/package/**/custom-persistence.xml</value>
            <value>classpath*:META-INF/persistence.xml</value>
        </list>
    </property>
    <property name="dataSources">
        <map>
            <entry key="localDataSource" value-ref="local-db"/>
            <entry key="remoteDataSource" value-ref="remote-db"/>
        </map>
    </property>
    <!-- if no datasource is specified, use this one -->
    <property name="defaultDataSource" ref="remoteDataSource"/>
</bean>

<bean id="emf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="persistenceUnitManager" ref="pum"/>
    <property name="persistenceUnitName" value="myCustomUnit"/>
</bean>
```

默认实现允许以声明方式（通过影响所有托管单元的属性）或以编程方式（通过允许选择持久单元的PersistenceUnitPostProcessor）自定义PersistenceUnitInfo实例（在将其馈送到JPA提供程序之前）。 如果未指定PersistenceUnitManager，则LocalContainerEntityManagerFactoryBean将创建一个内部使用的对象。

更多教程请访问:www.flydean.com

后台引导

LocalContainerEntityManagerFactoryBean通过bootstrapExecutor属性支持后台引导，如以下示例所示：

```
<bean id="emf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="bootstrapExecutor">
        <bean class="org.springframework.core.task.SimpleAsyncTaskExecutor"/>
    </property>
</bean>
```

实际的JPA提供程序引导将移交给指定的执行程序，然后并行运行到应用程序引导线程。 公开的EntityManagerFactory代理可以注入其他应用程序组件，甚至可以响应EntityManagerFactoryInfo配置检查。 但是，一旦其他组件访问了实际的JPA提供程序（例如，调用createEntityManager），这些调用就会阻塞，直到后台引导完成为止。 特别是，当您使用Spring Data JPA时，请确保还为其存储库设置了延迟引导。

4.4.2 基于JPA的DAO实现：EntityManagerFactory和EntityManager

> 尽管EntityManagerFactory实例是线程安全的，但EntityManager实例不是。 注入的JPA EntityManager的行为就像是从应用程序服务器的JNDI环境中提取的EntityManager一样，如JPA规范所定义。 它将所有调用委派给当前的事务性EntityManager（如果有）。 否则，它会退回到每个操作新创建的EntityManager上，实际上使它的使用成为线程安全的。

通过使用注入的EntityManagerFactory或EntityManager，可以在没有任何Spring依赖项的情况下针对普通JPA编写代码。 如果启用了PersistenceAnnotationBeanPostProcessor，Spring可以在字段和方法级别上了解@PersistenceUnit和@PersistenceContext批注。 以下示例显示了使用@PersistenceUnit批注的普通JPA DAO实现：

```
public class ProductDaoImpl implements ProductDao {

    private EntityManagerFactory emf;

    @PersistenceUnit
    public void setEntityManagerFactory(EntityManagerFactory emf) {
        this.emf = emf;
    }

    public Collection loadProductsByCategory(String category) {
        try (EntityManager em = this.emf.createEntityManager()) {
            Query query = em.createQuery("from Product as p where p.category = ?1");
            query.setParameter(1, category);
            return query.getResultList();
        }
    }
}
```

前面的DAO不依赖于Spring，并且仍然非常适合Spring应用程序上下文。 此外，DAO利用注解的优势要求注入默认的EntityManagerFactory，如以下示例bean定义所示：

```
<beans>

    <!-- bean post-processor for JPA annotations -->
    <bean class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor"/>

    <bean id="myProductDao" class="product.ProductDaoImpl"/>

</beans>
```

更多教程请访问:www.flydean.com

作为显式定义PersistenceAnnotationBeanPostProcessor的替代方法，请考虑在应用程序上下文配置中使用Spring context：annotation-config XML元素。 这样做会自动注册所有Spring标准后处理器以进行基于注解的配置，包括CommonAnnotationBeanPostProcessor等。

考虑以下示例：

```
<beans>

    <!-- post-processors for all standard config annotations -->
    <context:annotation-config/>

    <bean id="myProductDao" class="product.ProductDaoImpl"/>

</beans>
```

这种DAO的主要问题在于，它总是在工厂中始终创建一个新的EntityManager。 您可以通过请求注入事务性EntityManager（也称为“共享的EntityManager”，因为它是实际事务性EntityManager的共享的线程安全代理）来代替工厂注入，从而避免了这种情况。 以下示例显示了如何执行此操作：

```
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批注具有一个称为type的可选属性，默认为PersistenceContextType.TRANSACTION。 您可以使用此默认值来接收共享的EntityManager代理。 替代方案PersistenceContextType.EXTENDED是完全不同的事情。 这导致了所谓的扩展EntityManager，它不是线程安全的，因此不能在并发访问的组件（例如Spring管理的单例bean）中使用。 扩展的EntityManager实例仅应用于有状态的组件（例如，驻留在会话中），并且EntityManager的生命周期不依赖于当前事务，而完全取决于应用程序。

> 方法级和字段级注入
>
> 您可以在类中的字段或方法上应用指示依赖项注入的注解（例如@PersistenceUnit和@PersistenceContext），因此表达式为“方法级注入”和“字段级注入”。字段级注解简洁明了，易于使用，而方法级注解则允许对注入的依赖项进行进一步处理。在这两种情况下，成员的可见性（公共，受保护或私有）都无关紧要。
>
> 那类级别的注解呢?
>
> 在Java EE平台上，它们用于依赖性声明，而不用于资源注入。

注入的EntityManager是Spring管理的（意识到正在进行的事务）。即使新的DAO实现使用EntityManager而不是EntityManagerFactory的方法级注入，由于注解的使用，应用程序上下文XML也不需要更改。

这种DAO样式的主要优点是它仅取决于Java Persistence API。不需要导入任何Spring类。此外，由于可以理解JPA批注，因此Spring容器会自动应用注入。从非侵入性的角度来看，这是有吸引力的，并且对于JPA开发人员而言，感觉会更自然。

4.4.3 Spring驱动的JPA交易

> 强烈建议您阅读声明式事务管理（如果尚未阅读），以更详细地介绍Spring的声明式事务支持。

对于JPA，推荐的策略是通过JPA的本地交易支持进行本地交易。 Spring的JpaTransactionManager提供了许多本地JDBC事务中已知的功能（例如，特定于事务的隔离级别和资源级别的只读优化），而不需要任何常规JDBC连接池（无需XA）。

如果注册的JpaDialect支持检索基础JDBC连接，Spring JPA还允许已配置的JpaTransactionManager向访问同一数据源的JDBC访问代码公开JPA事务。 Spring为EclipseLink和Hibernate JPA实现提供了方言。有关JpaDialect机制的详细信息，请参见下一部分。

> 作为一种直接替代方案，Spring的本机HibernateTransactionManager能够与Spring Framework 5.1和Hibernate 5.2 / 5.3以及更高版本的JPA访问代码进行交互，以适应多种Hibernate规范并提供JDBC交互。结合LocalSessionFactoryBean设置，这特别有意义。有关详细信息，请参见用于JPA交互的本机Hibernate设置。

4.4.4 了解JpaDialect和JpaVendorAdapter

作为高级功能，JpaTransactionManager和AbstractEntityManagerFactoryBean的子类允许将自定义JpaDialect传递到jpaDialect Bean属性中。 JpaDialect实现通常可以通过特定于供应商的方式来启用Spring支持的以下高级功能：

- 应用特定的事务语义（例如自定义隔离级别或事务超时）
- 检索事务性JDBC连接（用于公开基于JDBC的DAO）
- 将PersistenceExceptions高级转换为Spring DataAccessExceptions

这对于特殊的事务语义和异常的高级翻译特别有价值。默认实现（DefaultJpaDialect）不提供任何特殊功能，如果需要前面列出的功能，则必须指定适当的方言。

> 作为主要用于Spring的全功能LocalContainerEntityManagerFactoryBean设置的更广泛的提供程序适应工具，JpaVendorAdapter将JpaDialect的功能与其他特定于提供程序的默认值结合在一起。指定HibernateJpaVendorAdapter或EclipseLinkJpaVendorAdapter分别是为Hibernate或EclipseLink自动配置EntityManagerFactory设置的最便捷方法。请注意，这些提供程序适配器主要是设计用于与Spring驱动的事务管理一起使用的（即，与JpaTransactionManager一起使用）。

有关其操作以及在Spring的JPA支持中如何使用它们的更多详细信息，请参见JpaDialect和JpaVendorAdapter javadoc。

4.4.5 使用JTA事务管理设置JPA

作为JpaTransactionManager的替代，Spring还允许通过JTA在Java EE环境中或与独立事务协调器（例如Atomikos）一起进行多资源事务协调。除了选择Spring的JtaTransactionManager而不是JpaTransactionManager之外，您还需要采取其他一些步骤：

- 基础的JDBC连接池需要具有XA功能，并且必须与事务协调器集成在一起。这在Java EE环境中通常很简单，它通过JNDI公开了另一种数据源。有关详细信息，请参见您的应用程序服务器文档。类似地，一个独立的事务协调器通常带有特殊的XA集成的DataSource实现。再次，检查其文档。
- 需要为JTA配置JPA EntityManagerFactory设置。这是特定于提供程序的，通常是通过将特殊属性指定为LocalContainerEntityManagerFactoryBean上的jpaProperties。对于Hibernate，这些属性甚至是特定于版本的。有关详细信息，请参见Hibernate文档。
- Spring的HibernateJpaVendorAdapter强制执行某些面向Spring的默认值，例如连接释放模式，即合上，该模式与Hibernate 5.0中Hibernate自己的默认值匹配，但在5.1 / 5.2中则不再适用。对于JTA设置，请不要声明HibernateJpaVendorAdapter开头或关闭其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自己的默认设置。有关WebLogic的相关说明，请参见带有Hibernate的Spurious Application Server警告。
- 或者，考虑从应用程序服务器本身（即通过JNDI查找而不是本地声明的LocalContainerEntityManagerFactoryBean）获取EntityManagerFactory。服务器提供的EntityManagerFactory可能在服务器配置中需要特殊定义（使部署的可移植性降低），但已针对服务器的JTA环境进行了设置。

更多教程请访问:www.flydean.com

4.4.6 JPA交互的本机Hibernate设置和本机Hibernate事务

从Spring Framework 5.1和Hibernate 5.2 / 5.3开始，与HibernateTransactionManager结合使用的本地LocalSessionFactoryBean设置允许与@PersistenceContext和其他JPA访问代码进行交互。现在，Hibernate SessionFactory本地实现JPA的EntityManagerFactory接口，而Hibernate Session句柄本身就是JPA EntityManager。Spring的JPA支持工具会自动检测本地的Hibernate会话。

因此，在许多情况下，这种本地Hibernate设置可以替代标准JPA LocalContainerEntityManagerFactoryBean和JpaTransactionManager组合，从而允许在同一本地事务中与@PersistenceContext EntityManager旁边的SessionFactory.getCurrentSession（）（以及HibernateTemplate）进行交互。这样的设置还提供了更强的Hibernate集成和更大的配置灵活性，因为它不受JPA引导合同的约束。

在这种情况下，您不需要HibernateJpaVendorAdapter配置，因为Spring的本机Hibernate设置提供了更多功能（例如，自定义Hibernate Integrator设置，Hibernate 5.3 Bean容器集成以及对只读事务的更强优化）。最后但并非最不重要的一点是，您还可以通过LocalSessionFactoryBuilder表达本机的Hibernate设置，并与@Bean样式配置无缝集成（不涉及FactoryBean）。

就像JPA LocalContainerEntityManagerFactoryBean一样，LocalSessionFactoryBean和LocalSessionFactoryBuilder支持后台引导。有关简介，请参见后台引导。

在LocalSessionFactoryBean上，可以通过bootstrapExecutor属性使用。在编程的LocalSessionFactoryBuilder上，重载的buildSessionFactory方法采用引导执行程序参数。

更多教程请访问:www.flydean.com

5.使用Object-XML映射封装XML

5.1 简介

本章介绍了Spring的Object-XML Mapping支持。对象XML映射（简称O-X映射）是将XML文档与对象进行相互转换的动作。此转换过程也称为XML编组或XML序列化。本章可以互换使用这些术语。

在O-X映射领域中，marshaller负责将对象（图形）序列化为XML。以类似的方式，unmarshaller将XML反序列化为对象图。该XML可以采用DOM文档，输入或输出流或SAX处理程序的形式。

使用Spring满足O / X映射需求的一些好处是：

- 易于配置
- 一致的接口
- 一致的异常层次结构

5.1.1 易于配置

使用Spring的bean工厂，可以轻松配置编组器，而无需构造JAXB上下文，JiBX绑定工厂等。您可以像在应用程序上下文中配置任何其他bean一样配置编组器。此外，许多编组人员都可以使用基于XML名称空间的配置，从而使配置更加简单。

5.1.2 一致的接口

Spring的O-X映射通过两个全局接口运行：Marshaller和Unmarshaller。这些抽象使您可以相对轻松地切换O-X映射框架，而进行编组的类几乎不需要更改。这种方法还有一个好处，就是可以以非介入方式使用混合匹配方法（例如，一些使用JAXB进行的编组和某些由Castor进行的编组）进行XML编组，从而让您充分利用每种方法的优势技术。

5.1.3 一致的异常层次结构

Spring提供了从底层O-X映射工具到XmlMappingException作为根异常的异常层次转换。这些运行时异常包装原始异常，因此不会丢失任何信息。

5.2 Marshaller和Unmarshaller

如引言中所述，封送器将对象序列化为XML，解组器将XML流反序列化为对象。 本节描述了用于此目的的两个Spring接口。

5.2.1 了解Marshaller

Spring在org.springframework.oxm.Marshaller接口背后抽象了所有编组操作，其主要方法如下：

```
public interface Marshaller {

    /**
     * Marshal the object graph with the given root into the provided Result.
     */
    void marshal(Object graph, Result result) throws XmlMappingException, IOException;
}
```

更多教程请访问:www.flydean.com

Marshaller接口有一个主要方法，该方法将给定的对象封送给给定的javax.xml.transform.Result。 结果是一个标记接口，该接口基本上表示XML输出抽象。 如下表所示，具体的实现包装了各种XML表示形式：

结果实现	XML封装
DOMResult	org.w3c.dom.Node
SAXResult	org.xml.sax.ContentHandler
StreamResult	java.io.File, java.io.OutputStream, or java.io.Writer

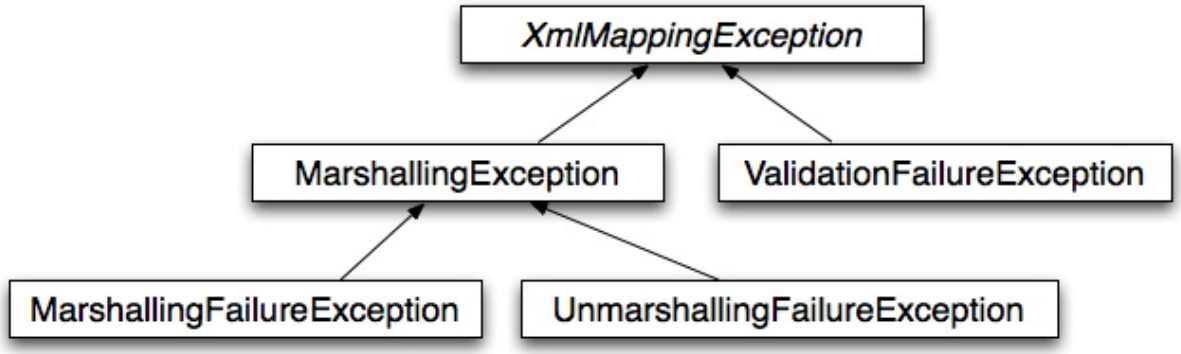
即使有两个单独的编组接口（Marshaller和Unmarshaller），Spring-WS中的所有实现都在一个类中实现。 这意味着您可以连接一个编组类，并在applicationContext.xml中将其称为编组和解组。

5.2.3 理解XmlMappingException

Spring使用XmlMappingException作为根异常，将基础O-X映射工具中的异常转换为它自己的异常层次结构。 这些运行时异常包装了原始异常，因此不会丢失任何信息。

此外，即使基础的O-X映射工具没有这样做，MarshallingFailureException和UnmarshallingFailureException也会在编组和解组操作之间进行区分。

O-X映射异常层次结构如下图所示：



使用Marshaller和Unmarshaller

您可以在多种情况下使用Spring的OXM。 在下面的示例中，我们使用它来将Spring托管应用程序的设置作为XML文件进行编组。 在下面的示例中，我们使用一个简单的JavaBean来表示设置：

```
public class Settings {

    private boolean fooEnabled;

    public boolean isFooEnabled() {
        return fooEnabled;
    }

    public void setFooEnabled(boolean fooEnabled) {
        this.fooEnabled = fooEnabled;
    }
}
```

应用程序类使用此bean存储其设置。 除了主要方法之外，该类还有两个方法：saveSettings（）将settings Bean保存到名为settings.xml的文件中，loadSettings（）再次加载这些设置。 下面的main（）方法构造一个Spring应用程序上下文并调用这两个方法：

更多教程请访问:www.flydean.com

```
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 {
        try (FileOutputStream os = new FileOutputStream(FILE_NAME)) {
            this.marshaller.marshal(settings, new StreamResult(os));
        }
    }

    public void loadSettings() throws IOException {
        try (FileInputStream is = new FileInputStream(FILE_NAME)) {
            this.settings = (Settings) this.unmarshaller.unmarshal(new StreamSource(is));
        }
    }

    public static void main(String[] args) throws IOException {
        ApplicationContext appContext =
                new ClassPathXmlApplicationContext("applicationContext.xml");
        Application application = (Application) appContext.getBean("application");
        application.saveSettings();
        application.loadSettings();
    }
}
```

该应用程序需要同时设置封送程序和取消封送程序属性。 我们可以使用以下applicationContext.xml来做到这一点：

```
<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>
```

该应用程序上下文使用Castor，但是我们可以使用本章后面介绍的任何其他编组实例。 请注意，默认情况下，Castor不需要任何进一步的配置，因此Bean的定义非常简单。 还要注意，CastorMarshaller同时实现了Marshaller和Unmarshaller，因此我们可以在应用程序的marshaller和unmarshaller属性中引用castorMarshaller bean。

该示例应用程序生成以下settings.xml文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<settings foo-enabled="false"/>
```

更多教程请访问:www.flydean.com

5.4 XML配置命名空间

您可以使用OXM名称空间中的标签来更简洁地配置编组器。 要使这些标签可用，您必须首先在XML配置文件的序言中引用适当的架构。 以下示例显示了如何执行此操作：

```
<?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
  https://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/oxm https://www.springframework.org/schema/oxm/spring-oxm.xsd">
```

当前，该模式使以下元素可用：

jaxb2-marshaller

jibx-marshaller

castor-marshaller

每个标签均在其各自的编组部分中进行了说明。 但是，作为示例，JAXB2编组器的配置可能类似于以下内容：

```
<oxm:jaxb2-marshaller id="marshaller" contextPath="org.springframework.ws.samples.airline.schema"/>
```

5.5 JAXB

JAXB绑定编译器将W3C XML Schema转换为一个或多个Java类，一个jaxb.properties文件以及可能的一些资源文件。 JAXB还提供了一种从带注解的Java类生成模式的方法。

遵循Marshaller和Unmarshaller中描述的Marshaller和Unmarshaller接口，Spring支持JAXB 2.0 API作为XML编组策略。 相应的集成类位于org.springframework.oxm.jaxb包中。

5.5.1 使用Jaxb2Marshaller

Jaxb2Marshaller类实现了Spring的Marshaller和Unmarshaller接口。它需要上下文路径才能运行。您可以通过设置contextPath属性来设置上下文路径。上下文路径是冒号分隔的Java程序包名称的列表，其中包含模式派生的类。它还提供了classesToBeBound属性，该属性使您可以设置编组支持的类的数组。通过向Bean指定一个或多个模式资源来执行模式验证，如以下示例所示：

```
<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配置命名空间

更多教程请访问:www.flydean.com

jaxb2-marshaller元素配置org.springframework.oxm.jaxb.Jaxb2Marshaller，如以下示例所示：

```
<oxm:jaxb2-marshaller id="marshaller" contextPath="org.springframework.ws.samples.airline.schema"/>
```

或者，您可以使用要绑定的子类元素提供绑定到编组器的类的列表：

```
<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>
```

下表描述了可用的属性：

Attribute	Description	Required
id	The ID of the marshaller	No
contextPath	The JAXB Context path	No

5.6 Castor

Castor XML映射是一个开源XML绑定框架。 它使您可以将Java对象模型中包含的数据与XML文档进行相互转换。 默认情况下，它不需要任何进一步的配置，尽管您可以使用映射文件来更好地控制Castor的行为。

有关Castor的更多信息，请参见Castor网站。 Spring集成类位于org.springframework.oxm.castor包中。

5.6.1 使用CastorMarshaller

与JAXB一样，CastorMarshaller也实现Marshaller和Unmarshaller接口。 可以按以下方式进行连接：

```
<beans>
    <bean id="castorMarshaller" class="org.springframework.oxm.castor.CastorMarshaller" />
    ...
</beans>
```

5.6.2 Mapping

尽管可以依赖Castor的默认编组行为，但可能需要对其进行更多控制。 您可以通过使用Castor映射文件来获得更多控制。 有关更多信息，请参见Castor XML映射。

您可以通过使用mappingLocation资源属性来设置映射，在以下示例中使用类路径资源来指示：

```
<beans>
    <bean id="castorMarshaller" class="org.springframework.oxm.castor.CastorMarshaller" >
        <property name="mappingLocation" value="classpath:mapping.xml" />
    </bean>
</beans>
```

XML配置命名空间

castor-marshaller标记配置org.springframework.oxm.castor.CastorMarshaller，如以下示例所示：

```
<oxm:castor-marshaller id="marshaller" mapping-location="classpath:org/springframework/oxm/castor/mapping.xml"/
>
```

更多教程请访问:www.flydean.com

您可以通过两种方式配置marshaller实例：通过指定映射文件的位置（通过mapping-location属性）或通过标识Java POJO（通过target-class或target-package属性）来确定存在其对应XML的Java POJO。 描述符类。 后一种方法通常与从XML模式生成XML代码结合使用。

下表描述了可用的属性：

Attribute	Description	Required
id	The ID of the marshaller	No
encoding	The encoding to use for unmarshalling from XML	No
target-class	A Java class name for a POJO for which an XML class descriptor is available (as generated through code generation)	No
target-package	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
mapping-location	Location of a Castor XML mapping file	No

5.7 JiBX

JiBX框架提供了与Hibernate为ORM提供的解决方案类似的解决方案：绑定定义定义了Java对象如何与XML相互转换的规则。 在准备好绑定并编译了类之后，JiBX绑定编译器将增强类文件并添加代码以处理将类的实例从XML转换为XML的过程。

有关JiBX的更多信息，请参见JiBX网站。 Spring集成类位于org.springframework.oxm.jibx包中。

5.7.1 使用JibxMarshaller

JibxMarshaller类同时实现Marshaller和Unmarshaller接口。 要进行操作，需要输入要编组的类的名称，您可以使用targetClass属性进行设置。 （可选）您可以通过设置bindingName属性来设置绑定名称。 在下面的示例中，我们绑定了Flights类：

```
<beans>
    <bean id="jibxFlightsMarshaller" class="org.springframework.oxm.jibx.JibxMarshaller">
        <property name="targetClass">org.springframework.oxm.jibx.Flights</property>
    </bean>
    ...
</beans>
```

JibxMarshaller为单个类配置。 如果要封送多个类，则必须使用不同的targetClass属性值配置多个JibxMarshaller实例。

XML配置命名空间

jibx-marshaller标记配置org.springframework.oxm.jibx.JibxMarshaller，如以下示例所示：

```
<oxm:jibx-marshaller id="marshaller" target-class="org.springframework.ws.samples.airline.schema.Flight"/>
```

5.8 XStream

XStream是一个简单的库，用于将对象序列化为XML并再次返回。 它不需要任何映射并生成干净的XML。

有关XStream的更多信息，请参见XStream网站。 Spring集成类位于org.springframework.oxm.xstream包中。

更多教程请访问:www.flydean.com

5.8.1 使用XStreamMarshaller

XStreamMarshaller不需要任何配置，可以直接在应用程序上下文中进行配置。 要进一步自定义XML，可以设置一个别名映射，该映射由映射到类的字符串别名组成，如以下示例所示：

```
<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>
```

> 默认情况下，XStream允许将任意类取消编组，这可能导致不安全的Java序列化效果。因此，我们不建议使用XStreamMarshaller从外部源（即Web）解组XML，因为这可能会导致安全漏洞。
>
> 如果选择使用XStreamMarshaller从外部源解组XML，请在XStreamMarshaller上设置supportedClasses属性，如以下示例所示：

```
<bean id =“ xstreamMarshaller” class =“ org.springframework.oxm.xstream.XStreamMarshaller”>
    <property name =“ supportedClasses” value =“ org.springframework.oxm.xstream.Flight” />
    ...
</ bean>
```

> 这样做可以确保只有注册的班级才有资格进行编组。
>
> 此外，您可以注册自定义转换器，以确保只能解组受支持的类。除了明确支持应支持的域类的转换器之外，您可能还想添加CatchAllConverter作为列表中的最后一个转换器。结果，不会调用具有较低优先级和可能的安全漏洞的默认XStream转换器。

更多教程请访问:www.flydean.com

Web Servlet

文档的此部分涵盖对基于Servlet API构建并部署到Servlet容器的Servlet堆栈Web应用程序的支持。 各个章节包括Spring MVC，View Technologies，CORS支持和WebSocket支持。

更多教程请访问:www.flydean.com

1. Spring Web MVC

Spring Web MVC是基于Servlet API构建的原始Web框架，从一开始就已包含在Spring框架中。 正式名称“ Spring Web MVC”来自其源模块的名称（spring-webmvc），但它通常被称为“ Spring MVC”。

与Spring Web MVC并行，Spring Framework 5.0引入了一个反应式堆栈Web框架，其名称“ Spring WebFlux”也基于其源模块（spring-webflux）。 本节介绍Spring Web MVC。 下一节将介绍Spring WebFlux。

更多教程请访问:www.flydean.com

1.1 DispatcherServlet

与其他许多Web框架一样，Spring MVC围绕前端控制器模式进行设计，在该模式下，中央Servlet DispatcherServlet提供了用于请求处理的共享算法，而实际工作是由可配置的委托组件执行的。 该模型非常灵活，并支持多种工作流程。

与任何Servlet一样，都需要根据Servlet规范使用Java配置或在web.xml中声明和映射DispatcherServlet。 反过来，DispatcherServlet使用Spring配置发现请求映射，视图解析，异常处理等所需的委托组件。

以下Java配置示例注册并初始化DispatcherServlet，该容器由Servlet容器自动检测到（请参阅Servlet Config）：

```
public class MyWebApplicationInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletCxt) {

        // Load Spring web application configuration
        AnnotationConfigWebApplicationContext ac = new AnnotationConfigWebApplicationContext();
        ac.register(AppConfig.class);
        ac.refresh();

        // Create and register the DispatcherServlet
        DispatcherServlet servlet = new DispatcherServlet(ac);
        ServletRegistration.Dynamic registration = servletCxt.addServlet("app", servlet);
        registration.setLoadOnStartup(1);
        registration.addMapping("/app/*");
    }
}
```

> 除了直接使用ServletContext API外，您还可以扩展AbstractAnnotationConfigDispatcherServletInitializer并覆盖特定方法（请参见Context Hierarchy下的示例）。

以下web.xml配置示例注册并初始化DispatcherServlet：

```
<web-app>

    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/app-context.xml</param-value>
    </context-param>

    <servlet>
        <servlet-name>app</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <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>app</servlet-name>
        <url-pattern>/app/*</url-pattern>
    </servlet-mapping>

</web-app>
```

更多教程请访问:www.flydean.com

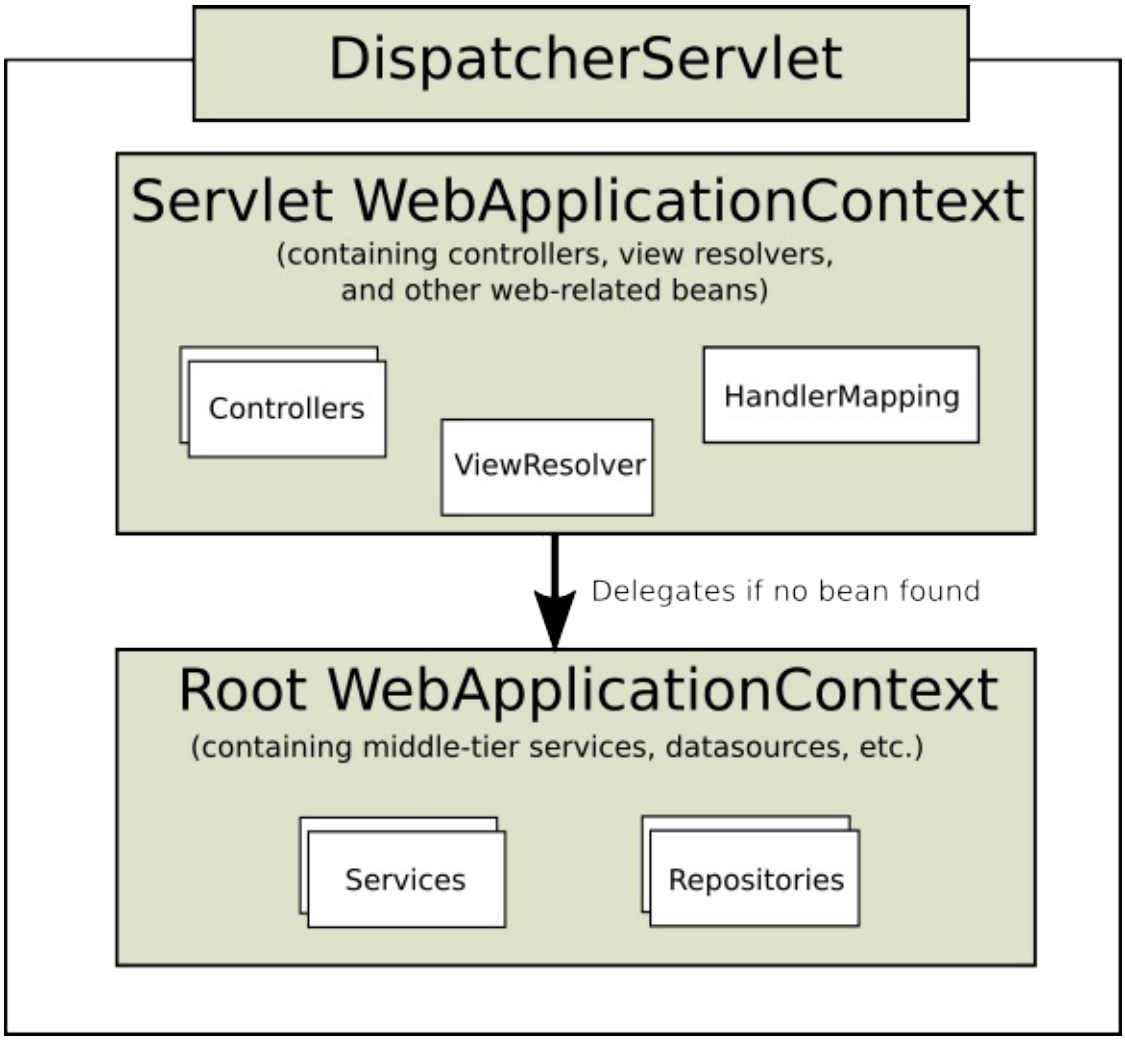
> Spring Boot遵循不同的初始化顺序。 Spring Boot并没有陷入Servlet容器的生命周期，而是使用Spring配置来引导自身和嵌入式Servlet容器。 在Spring配置中检测到过滤器和Servlet声明，并在Servlet容器中注册。 有关更多详细信息，请参见Spring Boot文档。

1.1.1 Context 层级关系

DispatcherServlet期望WebApplicationContext（纯ApplicationContext的扩展）为其自身的配置。WebApplicationContext具有指向ServletContext和与其关联的Servlet的链接。它还绑定到ServletContext，以便应用程序可以在RequestContextUtils上使用静态方法来查找WebApplicationContext（如果需要访问它们）。

对于许多应用程序而言，拥有一个WebApplicationContext很简单并且足够。也可能具有上下文层次结构，其中一个根WebApplicationContext在多个DispatcherServlet（或其他Servlet）实例之间共享，每个实例都有其自己的子WebApplicationContext配置。有关上下文层次结构功能的更多信息，请参见ApplicationContext的其他功能。

根WebApplicationContext通常包含需要在多个Servlet实例之间共享的基础结构Bean，例如数据存储库和业务服务。这些Bean是有效继承的，并且可以在Servlet特定的子WebApplicationContext中重写（即重新声明），该子WebApplicationContext通常包含给定Servlet本地的Bean。下图显示了这种关系：



下面的示例配置一个WebApplicationContext层次结构：

```
public class MyWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
```

更多教程请访问:www.flydean.com

```
        return new Class<?>[] { RootConfig.class };
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class<?>[] { App1Config.class };
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/app1/*" };
    }
}
```

> 如果不需要应用程序上下文层次结构，则应用程序可以通过getRootConfigClasses（）返回所有配置，并从getServletConfigClasses（）返回null。

以下示例显示了web.xml等效项：

```
<web-app>

    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/root-context.xml</param-value>
    </context-param>

    <servlet>
        <servlet-name>app1</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/app1-context.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>app1</servlet-name>
        <url-pattern>/app1/*</url-pattern>
    </servlet-mapping>

</web-app>
```

> 如果不需要应用程序上下文层次结构，则应用程序可以仅配置“根”上下文，并将contextConfigLocation Servlet参数保留为空。

1.1.2 特殊bean类型

DispatcherServlet委托给特殊的Bean处理请求并呈现适当的响应。 所谓“特殊bean”，是指实现框架协定的Spring管理对象实例。 这些通常带有内置合同，但是您可以自定义它们的属性并扩展或替换它们。

下表列出了DispatcherServlet检测到的特殊bean：

Bean类型	说明
HandlerMapping	将请求与拦截器列表一起映射到处理程序，以进行预处理和后期处理。 映射基于某些条件，其细节因HandlerMapping实现而异。HandlerMapping的两个主要实现是RequestMappingHandlerMapping（支持@RequestMapping带注解的方法）和SimpleUrlHandlerMapping（维护对处理程序的URI路径模式的显式注册）。

更多教程请访问:www.flydean.com

HandlerAdapter	帮助DispatcherServlet调用映射到请求的处理程序，而不管实际如何调用该处理程序。 例如，调用带注解的控制器需要解析注解。 HandlerAdapter的主要目的是使DispatcherServlet免受此类细节的影响。
HandlerExceptionResolver	解决异常的策略，可能将它们映射到处理程序，HTML错误视图或其他目标。 请参阅例外。
ViewResolver	解析从处理程序返回的实际基于字符串的基于逻辑的视图名称，以实际的视图呈现给响应。 请参阅查看分辨率和查看技术。
LocaleResolver, LocaleContextResolver	解析客户正在使用的语言环境以及可能的时区，以便能够提供国际化的视图。 请参阅区域设置。
ThemeResolver	解决Web应用程序可以使用的主题，例如，以提供个性化的布局。 请参阅主题。
MultipartResolver	借助一些多部分解析库来解析多部分请求的抽象（例如，浏览器表单文件上传）。 请参见多部分分解器。
FlashMapManager	存储和检索“输入”和“输出” FlashMap，可用于将属性从一个请求传递到另一个请求，通常跨重定向。 请参见Flash属性。

1.1.3 Web MVC Config

应用程序可以声明处理请求所需的特殊Bean类型中列出的基础结构Bean。 DispatcherServlet检查每个特殊bean的WebApplicationContext。 如果没有匹配的bean类型，它将使用DispatcherServlet.properties中列出的默认类型。

在大多数情况下，MVC Config是最佳起点。 它使用Java或XML声明所需的bean，并提供了更高级别的配置回调API来对其进行自定义。

1.1.4 Servlet Config

在Servlet 3.0+环境中，您可以选择以编程方式配置Servlet容器，作为替代方案或与web.xml文件结合使用。 下面的示例注册一个DispatcherServlet：

```
import org.springframework.web.WebApplicationInitializer;

public class MyWebApplicationInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext container) {
        XmlWebApplicationContext appContext = new XmlWebApplicationContext();
        appContext.setConfigLocation("/WEB-INF/spring/dispatcher-config.xml");

        ServletRegistration.Dynamic registration = container.addServlet("dispatcher", new DispatcherServlet(app
Context));
        registration.setLoadOnStartup(1);
        registration.addMapping("/");
    }
}
```

WebApplicationInitializer是Spring MVC提供的接口，可确保检测到您的实现并将其自动用于初始化任何Servlet 3容器。 WebApplicationInitializer的抽象基类实现名为AbstractDispatcherServletInitializer，它通过重写指定Servlet映射和DispatcherServlet配置位置的方法，使注册DispatcherServlet更容易。

对于使用基于Java的Spring配置的应用程序，建议这样做，如以下示例所示：

```
public class MyWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
```

更多教程请访问:www.flydean.com

```
    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，则应直接从AbstractDispatcherServletInitializer进行扩展，如以下示例所示：

```
public class MyWebAppInitializer extends AbstractDispatcherServletInitializer {

    @Override
    protected WebApplicationContext createRootApplicationContext() {
        return null;
    }

    @Override
    protected WebApplicationContext createServletApplicationContext() {
        XmlWebApplicationContext cxt = new XmlWebApplicationContext();
        cxt.setConfigLocation("/WEB-INF/spring/dispatcher-config.xml");
        return cxt;
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
}
```

AbstractDispatcherServletInitializer还提供了一种方便的方法来添加Filter实例，并将其自动映射到DispatcherServlet，如以下示例所示：

```
public class MyWebAppInitializer extends AbstractDispatcherServletInitializer {

    // ...

    @Override
    protected Filter[] getServletFilters() {
        return new Filter[] {
            new HiddenHttpMethodFilter(), new CharacterEncodingFilter() };
    }
}
```

每个过滤器都会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位置，以在DispatcherServlet及其映射的所有过滤器上启用异步支持。 默认情况下，此标志设置为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。

1.1.5 Processing

DispatcherServlet处理请求的方式如下：

更多教程请访问:www.flydean.com

- 搜索WebApplicationContext并将其绑定在请求中，作为控制器和流程中其他元素可以使用的属性。默认情况下，它绑定在DispatcherServlet.WEB_APPLICATION_CONTEXT_ATTRIBUTE键下。
- 语言环境解析器绑定到请求，以使流程中的元素解析在处理请求（呈现视图，准备数据等）时要使用的语言环境。如果不需要语言环境解析，则不需要语言环境解析器。
- 主题解析器绑定到请求，以使诸如视图之类的元素确定要使用的主题。如果不使用主题，则可以将其忽略。
- 如果指定多部分文件解析器，则将检查请求中是否有多部分。如果找到多部分，则将该请求包装在MultipartHttpServletRequest中，以供流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Multipart Resolver。
- 搜索适当的处理程序。如果找到处理程序，则执行与处理程序（预处理器，后处理器和控制器）关联的执行链，以准备模型或渲染。或者，对于带注解的控制器，可以呈现响应（在HandlerAdapter中），而不是返回视图。
- 如果返回模型，则呈现视图。如果没有返回任何模型（可能是由于预处理器或后处理器拦截了该请求，可能出于安全原因），则不会呈现任何视图，因为该请求可能已经被满足。

WebApplicationContext中声明的HandlerExceptionResolver Bean用于解决在请求处理期间引发的异常。这些异常解析器允许定制逻辑以解决异常。有关更多详细信息，请参见异常。

Spring DispatcherServlet还支持Servlet API所指定的last-modification-date的返回。确定特定请求的最后修改日期的过程很简单：DispatcherServlet查找适当的处理程序映射，并测试找到的处理程序是否实现了LastModified接口。如果是这样，则将LastModified接口的long getLastModified（request）方法的值返回给客户端。

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声明中，来定制各个DispatcherServlet实例。下表列出了受支持的参数：

参数	说明
contextClass	实现ConfigurableWebApplicationContext的类，将由此Servlet实例化并在本地配置。 默认情况下，使用XmlWebApplicationContext。
contextConfigLocation	传递给上下文实例的字符串（由contextClass指定），以指示可以在哪里找到上下文。 该字符串可能包含多个字符串（使用逗号作为分隔符）以支持多个上下文。 对于具有两次定义的bean的多个上下文位置，以最新位置为准。
namespace	WebApplicationContext的命名空间。 缺省为[servlet-name] -servlet。
throwExceptionIfNoHandlerFound	在找不到请求的处理程序时是否引发NoHandlerFoundException。 然后可以使用HandlerExceptionResolver捕获该异常（例如，通过使用@ExceptionHandler控制器方法），然后将其作为其他任何异常进行处理。默认情况下，它设置为false，在这种情况下，DispatcherServlet将响应状态设置为404（NOT_FOUND），而不会引发异常。请注意，如果还配置了默认servlet处理，则始终将未解决的请求转发到默认servlet，并且永远不会引发404。

1.1.6 Interception

所有HandlerMapping实现都支持处理程序拦截器，当您要将特定功能应用于某些请求时（例如，检查主体），该拦截器很有用。拦截器必须使用三种方法从org.springframework.web.servlet包中实现HandlerInterceptor，这三种方法应提供足够的灵活性来执行各种预处理和后处理：

- preHandle（..）：在执行实际处理程序之前
- postHandle（..）：处理程序执行后
- afterCompletion（..）：在完成请求之后

preHandle（..）方法返回一个布尔值。您可以使用此方法来中断或继续执行链的处理。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（例如，渲染了一个适当的视图），并且不会继续执行其他拦截器和执行链中的实际处理程序。

更多教程请访问:www.flydean.com

有关如何配置拦截器的示例，请参见MVC配置部分中的拦截器。您还可以通过在各个HandlerMapping实现上使用setter直接注册它们。

请注意，对于@ResponseBody和ResponseEntity方法，postHandle的用处不大，在HandlerAdapter内以及postHandle之前，将其写入并提交响应。这意味着对响应进行任何更改为时已晚，例如添加额外的标头。对于这种情况，您可以实现ResponseBodyAdvice并将其声明为Controller Advice Bean或直接在RequestMappingHandlerAdapter上对其进行配置。

1.1.7 Exceptions

如果异常在请求映射期间发生或从请求处理程序（例如@Controller）抛出，则DispatcherServlet委托给HandlerExceptionResolver Bean链来解决该异常并提供替代处理，通常是错误响应。

下表列出了可用的HandlerExceptionResolver实现：

HandlerExceptionResolver	Description
SimpleMappingExceptionResolver	异常类名称和错误视图名称之间的映射。 对于在浏览器应用程序中呈现错误页面很有用。
DefaultHandlerExceptionResolver	解决了Spring MVC引发的异常，并将它们映射到HTTP状态代码。 另请参见备用ResponseEntityExceptionHandler和REST API异常。
ResponseStatusExceptionResolver	使用@ResponseStatus批注解决异常，并根据批注中的值将其映射到HTTP状态代码。
ExceptionHandlerExceptionResolver	通过调用@Controller或@ControllerAdvice类中的@ExceptionHandler方法来解决异常。 请参见@ExceptionHandler方法。

解析器链

您可以通过在Spring配置中声明多个HandlerExceptionResolver bean并根据需要设置其order属性来形成异常解析器链。 order属性越高，异常解析器的定位就越晚。

HandlerExceptionResolver的约定指定它可以返回：

- 指向错误视图的ModelAndView。
- 如果在解析程序中处理了异常，则为空的ModelAndView。
- 如果该异常仍未解决，则为null，以供后续解析器尝试；如果该异常仍在末尾，则允许将其冒泡到Servlet容器。

MVC Config自动为默认的Spring MVC异常，@ResponseStatus注解的异常以及对@ExceptionHandler方法的支持声明内置解析器。您可以自定义该列表或替换它。

容器错误页面

如果任何HandlerExceptionResolver都无法解决异常，因此该异常可以传播，或者如果响应状态设置为错误状态（即4xx，5xx），则Servlet容器可以在HTML中呈现默认错误页面。要自定义容器的默认错误页面，可以在web.xml中声明错误页面映射。以下示例显示了如何执行此操作：

```
<error-page>
    <location>/error</location>
</error-page>
```

给定前面的示例，当异常冒出或响应具有错误状态时，Servlet容器在容器内向配置的URL（例如/ error）进行ERROR调度。 然后由DispatcherServlet处理它，可能将其映射到@Controller，可以实现该错误以使用模型返回错误视图名称或呈现JSON响应，如以下示例所示：

```
@RestController
public class ErrorController {
```

更多教程请访问:www.flydean.com

```
    @RequestMapping(path = "/error")
    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;
    }
}
```

> Servlet API没有提供在Java中创建错误页面映射的方法。 但是，您可以同时使用WebApplicationInitializer和最小的web.xml。

1.1.8 View解析

Spring MVC定义了ViewResolver和View接口，这些接口使您可以在浏览器中呈现模型，而无需将您与特定的视图技术联系在一起。 ViewResolver提供了视图名称和实际视图之间的映射。 在移交给特定的视图技术之前，视图解决了数据准备问题。

下表提供了有关ViewResolver层次结构的更多详细信息：

ViewResolver	Description
AbstractCachingViewResolver	AbstractCachingViewResolver的子类缓存它们解析的视图实例。 缓存可以提高某些视图技术的性能。 您可以通过将cache属性设置为false来关闭缓存。 此外，如果必须在运行时刷新某个视图（例如，当修改FreeMarker模板时），则可以使用removeFromCache（String viewName，Locale loc）方法。
XmlViewResolver	ViewResolver的实现，它接受用XML编写的配置文件，该配置文件的DTD与Spring的XML bean工厂相同。 默认配置文件是/WEB-INF/views.xml。
ResourceBundleViewResolver	在ResourceBundle中使用bean定义的ViewResolver的实现，由包基本名称指定。 对于应该解析的每个视图，它将属性[viewname]。（class）的值用作视图类，并将属性[viewname] .url的值用作视图URL。 您可以在“查看技术”一章中找到示例。
UrlBasedViewResolver	ViewResolver接口的简单实现会影响逻辑视图名称到URL的直接解析，而无需显式映射定义。 如果您的逻辑名称以直接的方式与视图资源的名称匹配，而无需任意映射，则这是适当的。
InternalResourceViewResolver	UrlBasedViewResolver的方便子类，它支持InternalResourceView（实际上是Servlet和JSP）以及诸如JstlView和TilesView之类的子类。 您可以使用setViewClass（..）为该解析器生成的所有视图指定视图类。 有关详细信息，请参见UrlBasedViewResolver Javadoc。
FreeMarkerViewResolver	UrlBasedViewResolver的便利子类，支持FreeMarkerView及其自定义子类。
ContentNegotiatingViewResolver	ViewResolver接口的实现，该接口根据请求文件名或Accept标头解析视图。 请参阅内容协商。

Handling

您可以通过声明多个解析器bean，并在必要时通过设置order属性以指定顺序来链接视图解析器。 请记住，order属性越高，视图解析器在链中的定位就越晚。

ViewResolver的协定指定它可以返回null，以指示找不到该视图。 但是，对于JSP和InternalResourceViewResolver，确定JSP是否存在的唯一方法是通过RequestDispatcher进行调度。 因此，您必须始终将InternalResourceViewResolver配置为在视图解析器的总体顺序中排在最后。

更多教程请访问:www.flydean.com

配置视图分辨率就像将ViewResolver bean添加到Spring配置中一样简单。 MVC Config为View解析器和添加无逻辑的View Controller提供了专用的配置API，这对于无需控制器逻辑的HTML模板呈现非常有用。

Redirecting

视图名称中的特殊redirect：前缀使您可以执行重定向。 UrlBasedViewResolver（及其子类）将其识别为需要重定向的指令。 视图名称的其余部分是重定向URL。

最终效果与控制器返回RedirectView的效果相同，但是现在控制器本身可以根据逻辑视图名称进行操作。 逻辑视图名称（例如redirect：/ myapp / some / resource）相对于当前Servlet上下文进行重定向，而名称诸如redirect：https：//myhost.com/some/arbitrary/path则重定向到绝对URL。

请注意，如果使用@ResponseStatus注解控制器方法，则注解值优先于RedirectView设置的响应状态。

Forwarding

您还可以使用特殊的forward：前缀表示最终由UrlBasedViewResolver和子类解析的视图名称。 这将创建一个InternalResourceView，它执行RequestDispatcher.forward（）。 因此，此前缀在InternalResourceViewResolver和InternalResourceView（对于JSP）中没有用，但是如果您使用另一种视图技术但仍然希望强制转发由Servlet / JSP引擎处理的资源，则该前缀很有用。 请注意，您也可以链接多个视图解析器。

Content Negotiation

ContentNegotiatingViewResolver不会解析视图本身，而是委派给其他视图解析器，并选择类似于客户端请求的表示形式的视图。可以从Accept标头或查询参数（例如，“ / path？format = pdf”）中确定表示形式。

ContentNegotiatingViewResolver通过将请求媒体类型与其与每个ViewResolvers关联的View支持的媒体类型（也称为Content-Type）进行比较，从而选择合适的View处理该请求。列表中具有兼容Content-Type的第一个View将表示形式返回给客户端。如果ViewResolver链无法提供兼容的视图，请查阅通过DefaultViews属性指定的视图列表。后一个选项适用于单身视图，无论逻辑视图名称如何，该视图都可以呈现当前资源的适当表示形式。 Accept标头可以包含通配符（例如text / *），在这种情况下，其Content-Type为text / xml的View是兼容的匹配。

1.1.9 Locale

正如Spring Web MVC框架所做的那样，Spring体系结构的大多数部分都支持国际化。使用DispatcherServlet，您可以使用客户端的语言环境自动解析消息。这是通过LocaleResolver对象完成的。

当请求进入时，DispatcherServlet会寻找一个语言环境解析器，如果找到了它，它会尝试使用它来设置语言环境。通过使用RequestContext.getLocale（）方法，您始终可以检索由语言环境解析器解析的语言环境。

除了自动的语言环境解析之外，您还可以在处理程序映射上附加一个拦截器（有关处理程序映射拦截器的更多信息，请参见拦截），以在特定情况下（例如，基于请求中的参数）更改语言环境。

语言环境解析器和拦截器在org.springframework.web.servlet.i18n包中定义，并以常规方式在应用程序上下文中进行配置。 Spring包含以下选择的语言环境解析器。

Time Zone

除了获取客户的语言环境外，了解其时区通常也很有用。 LocaleContextResolver接口提供了LocaleResolver的扩展，该扩展使解析程序可以提供更丰富的LocaleContext，其中可能包含时区信息。

如果可用，可以使用RequestContext.getTimeZone（）方法获取用户的TimeZone。通过Spring的ConversionService注册的任何日期/时间转换器和格式器对象都会自动使用时区信息。

Header Resolver

该语言环境解析器检查客户端（例如，Web浏览器）发送的请求中的accept-language标头。通常，此标头字段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。

更多教程请访问:www.flydean.com

Cookie Resolver

此语言环境解析器检查客户端上可能存在的Cookie，以查看是否指定了语言环境或TimeZone。如果是这样，它将使用指定的详细信息。通过使用此语言环境解析器的属性，可以指定Cookie的名称以及最长期限。以下示例定义了CookieLocaleResolver：

```
<bean id="localeResolver" class="org.springframework.web.servlet.i18n.CookieLocaleResolver">

    <property name="cookieName" value="clientlanguage"/>

    <!-- in seconds. If set to -1, the cookie is not persisted (deleted when browser shuts down) -->
    <property name="cookieMaxAge" value="100000"/>

</bean>
```

下面是CookieLocaleResolver的属性描述：

Property	Default	Description
cookieName	classname + LOCALE	cookie的名字
cookieMaxAge	Servlet container default	Cookie在客户端上保留的最长时间。 如果指定-1，则cookie将不会保留。它仅在客户端关闭浏览器之前可用。
cookiePath	/	将Cookie的可见性限制为网站的特定部分。 指定cookiePath时，该cookie仅对该路径及其下方的路径可见。

Session Resolver

通过SessionLocaleResolver，您可以从可能与用户请求关联的会话中检索Locale和TimeZone。与CookieLocaleResolver相比，此策略将本地选择的语言环境设置存储在Servlet容器的HttpSession中。结果，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会丢失。

请注意，与外部会话管理机制（例如Spring Session项目）没有直接关系。此SessionLocaleResolver针对当前HttpServletRequest评估并修改相应的HttpSession属性。

Locale Interceptor

您可以通过将LocaleChangeInterceptor添加到HandlerMapping定义之一来启用语言环境更改。它检测请求中的参数并相应地更改语言环境，在调度程序的应用程序上下文中在LocaleResolver上调用setLocale方法。下一个示例显示对包含名为siteLanguage的参数的所有* .view资源的调用现在会更改语言环境。因此，例如，对URL的请求https://www.sf.net/home.view?siteLanguage=nl会将站点语言更改为荷兰语。以下示例显示如何拦截语言环境：

```
<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>
```

更多教程请访问:www.flydean.com

```
</bean>
```

1.1.10 Themes

您可以应用Spring Web MVC框架主题来设置应用程序的整体外观，从而增强用户体验。主题是静态资源的集合，通常影响样式表和图像，这些样式表和图像会影响应用程序的视觉样式。

定义主题

要在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，委托是org.springframework.ui.context.support.ResourceBundleThemeSource实现，该实现从类路径的根加载属性文件。要使用自定义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Bean。 Web应用程序上下文会自动检测到具有该名称的bean并使用它。

当您使用ResourceBundleThemeSource时，将在一个简单的属性文件中定义一个主题。属性文件列出了组成主题的资源，如以下示例所示：

```
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定制标记来执行此操作，该标记与spring：message标记非常相似。 以下JSP片段使用上一个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="<spring:theme code='styleSheet'/>" type="text/css"/>
    </head>
    <body style="background=<spring:theme code='background'/>">
        ...
    </body>
</html>
```

默认情况下，ResourceBundleThemeSource使用一个空的基本名称前缀。 结果，从类路径的根加载属性文件。 因此，您可以将cool.properties主题定义放在类路径的根目录中（例如，在/ WEB-INF / classes中）。 ResourceBundleThemeSource使用标准的Java资源束加载机制，允许主题的完全国际化。 例如，我们可以有一个/WEB-INF/classes/cool_nl.properties，它引用带有荷兰文字的特殊背景图像。

Resolving Themes

定义主题后，如上一节所述，您可以决定要使用哪个主题。 DispatcherServlet查找一个名为themeResolver的bean，以找出要使用的ThemeResolver实现。 主题解析器的工作方式与LocaleResolver几乎相同。 它可以检测用于特定请求的主题，还可以更改请求的主题。 下表描述了Spring提供的主题解析器：

Class	Description
FixedThemeResolver	选择一个固定的主题，该主题通过使用defaultThemeName属性设置。
SessionThemeResolver	该主题在用户的HTTP会话中维护。 每个会话只需设置一次，但在会话之间不会保留。
CookieThemeResolver	所选主题存储在客户端的cookie中。

Spring还提供了ThemeChangeInterceptor，可以使用简单的请求参数对每个请求进行主题更改。

更多教程请访问:www.flydean.com

1.1.11 Multipart Resolver

org.springframework.web.multipart包中的MultipartResolver是一种用于解析包括文件上传在内的多部分请求的策略。有一种基于Commons FileUpload的实现，另一种基于Servlet 3.0多部分请求解析。

要启用多部分处理，您需要在DispatcherServlet Spring配置中声明一个名为multipartResolver的MultipartResolver bean。 DispatcherServlet会检测到它并将其应用于传入的请求。当收到内容类型为multipart / form-data的POST时，解析程序将解析内容并将当前的HttpServletRequest包装为MultipartHttpServletRequest，以提供对已解析部分的访问权，此外还可以将其公开为请求参数。

Apache Commons FileUpload

要使用Apache Commons FileUpload，可以配置名称为multipartResolver的CommonsMultipartResolver类型的Bean。您还需要commons-fileupload作为对类路径的依赖。

Servlet 3.0

需要通过Servlet容器配置启用Servlet 3.0多部分解析。为此：

- 在Java中，在Servlet注册上设置MultipartConfigElement。
- 在web.xml中，将“ <multipart-config>”部分添加到Servlet声明中。

以下示例显示了如何在Servlet注册上设置MultipartConfigElement：

```
public class AppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

    // ...

    @Override
    protected void customizeRegistration(ServletRegistration.Dynamic registration) {

        // Optionally also set maxFileSize, maxRequestSize, fileSizeThreshold
        registration.setMultipartConfig(new MultipartConfigElement("/tmp"));
    }

}
```

Servlet 3.0配置到位后，您可以添加名称为multipartResolver的StandardServletMultipartResolver类型的Bean。

1.1.12 Logging

Spring MVC中的DEBUG级别的日志被设计为紧凑，最少且人性化的。 它侧重于一遍又一遍有用的高价值信息，而其他信息仅在调试特定问题时才有用。

TRACE级别的日志记录通常遵循与DEBUG相同的原则（例如，也不应是消防水带），但可用于调试任何问题。 此外，某些日志消息在TRACE和DEBUG上可能显示不同级别的详细信息。

良好的日志记录来自使用日志的经验。 如果您发现任何不符合既定目标的东西，请告诉我们。

敏感数据

调试和跟踪日志记录可能会记录敏感信息。 这就是默认情况下屏蔽请求参数和标头，并且必须通过DispatcherServlet上的enableLoggingRequestDetails属性显式启用它们的完整登录的原因。

以下示例显示了如何通过使用Java配置来执行此操作：

```
public class MyInitializer
        extends AbstractAnnotationConfigDispatcherServletInitializer {
```

更多教程请访问:www.flydean.com

```
    @Override
    protected Class<?>[] getRootConfigClasses() {
        return ... ;
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return ... ;
    }

    @Override
    protected String[] getServletMappings() {
        return ... ;
    }

    @Override
    protected void customizeRegistration(Dynamic registration) {
        registration.setInitParameter("enableLoggingRequestDetails", "true");
    }

}
```

更多教程请访问:www.flydean.com

1.2 Filters

spring-web模块提供了一些有用的过滤器：

- Form Data
- Forwarded Headers
- Shallow ETag
- CORS

1.2.1 Form Data

浏览器只能通过HTTP GET或HTTP POST提交表单数据，但非浏览器客户端也可以使用HTTP PUT，PATCH和DELETE。 Servlet API需要ServletRequest.getParameter*（）方法来仅支持HTTP POST的表单字段访问。

spring-web模块提供FormContentFilter来拦截内容类型为application /x-www-form-urlencoded的HTTP PUT，PATCH和DELETE请求，从请求主体中读取表单数据，并包装ServletRequest以使 表单数据可通过ServletRequest.getParameter*（）方法族获得。

1.2.2 Forwarded Headers

当请求通过代理（例如负载平衡器）进行处理时，主机，端口和方案可能会更改，这使得从客户端角度创建指向正确的主机，端口和方案的链接带来了挑战。

RFC 7239定义了代理可以用来提供有关原始请求的信息的HTTP转发头。还有其他非标准标头，包括X-Forwarded-Host，X-Forwarded-Port，X-Forwarded-Proto，X-Forwarded-Ssl和X-Forwarded-Prefix。

ForwardedHeaderFilter是一个Servlet过滤器，它根据转发的标头修改请求的主机，端口和方案，然后删除这些标头。

对于转发的标头，存在安全方面的考虑，因为应用程序无法知道标头是由代理添加的，还是由恶意客户端添加的。这就是为什么应配置信任边界处的代理以删除来自外部的不受信任的转发标头的原因。您还可以使用removeOnly = true配置ForwardedHeaderFilter，在这种情况下，它将删除但不使用标头。

1.2.3 Shallow ETag

ShallowEtagHeaderFilter筛选器通过缓存写入响应的内容并从中计算MD5哈希值来创建“浅” ETag。 客户端下一次发送时，它会执行相同的操作，但是还会将计算出的值与If-None-Match请求标头进行比较，如果两者相等，则返回304（NOT_MODIFIED）。

此策略可节省网络带宽，但不会节省CPU，因为必须为每个请求计算完整响应。 如前所述，控制器级别的其他策略可以避免计算。 请参阅HTTP缓存。

该过滤器具有writeWeakETag参数，该参数将过滤器配置为写入弱ETag，类似于以下内容：W /“02a2d595e6ed9a0b24f027f2b63b134d6”（在RFC 7232第2.3节中定义）。

1.2.4 CORS

Spring MVC通过控制器上的注解为CORS配置提供了细粒度的支持。 但是，当与Spring Security一起使用时，我们建议您依赖内置的CorsFilter，该内置CorsFilter必须在Spring Security的过滤器链之前订购。

更多教程请访问:www.flydean.com

更多教程请访问:www.flydean.com

1.3 Controllers注解

Spring MVC提供了一个基于注解的编程模型，其中@Controller和@RestController组件使用注解来表达请求映射，请求输入，异常处理等。 带注解的控制器具有灵活的方法签名，无需扩展基类或实现特定的接口。 以下示例显示了由注解定义的控制器：

```
@Controller
public class HelloController {

    @GetMapping("/hello")
    public String handle(Model model) {
        model.addAttribute("message", "Hello World!");
        return "index";
    }
}
```

在前面的示例中，该方法接受Model并以String的形式返回视图名称，但是还存在许多其他选项，本章稍后将对其进行说明。

1.3.1 Declaration

您可以使用Servlet的WebApplicationContext中的标准Spring bean定义来定义控制器bean。 @Controller构造型允许自动检测，与Spring对在类路径中检测@Component类并为其自动注册Bean定义的常规支持保持一致。 它还充当带注解类的构造型，表明其作为Web组件的作用。

要启用对此类@Controller bean的自动检测，可以将组件扫描添加到Java配置中，如以下示例所示：

```
@Configuration
@ComponentScan("org.example.web")
public class WebConfig {

    // ...
}
```

以下示例显示了与先前示例等效的XML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:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="org.example.web"/>

    <!-- ... -->

</beans>
```

@RestController是一个组合的批注，其本身使用@Controller和@ResponseBody进行了元注解，以指示一个控制器，其每个方法都继承了类型级别的@ResponseBody批注，因此直接将其写入响应主体（与视图分辨率相对应）并使用HTML模板。

更多教程请访问:www.flydean.com

AOP Proxies

在某些情况下，您可能需要在运行时用AOP代理装饰控制器。 一个示例是，如果您选择直接在控制器上具有@Transactional批注。 在这种情况下，特别是对于控制器，我们建议使用基于类的代理。 这通常是控制器的默认选择。 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 />，您可以更改为<tx：annotation-driven /> proxy-target-class =“ true” />。

1.3.2 Request Mapping

您可以使用@RequestMapping批注将请求映射到控制器方法。 它具有各种属性，可以通过URL，HTTP方法，请求参数，标头和媒体类型进行匹配。 您可以在类级别使用它来表示共享的映射，也可以在方法级别使用它来缩小到特定的端点映射。

@RequestMapping还有HTTP方法特定的快捷方式：

- @GetMapping
- @PostMapping
- @PutMapping
- @DeleteMapping
- @PatchMapping

快捷方式是提供的“自定义注解”，因为可以说，大多数控制器方法应该映射到特定的HTTP方法，而不是使用@RequestMapping，后者默认情况下与所有HTTP方法匹配。 同时，在类级别仍需要@RequestMapping来表示共享映射。

以下示例具有类型和方法级别的映射：

```
@RestController
@RequestMapping("/persons")
class PersonController {

    @GetMapping("/{id}")
    public Person getPerson(@PathVariable Long id) {
        // ...
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public void add(@RequestBody Person person) {
        // ...
    }
}
```

URI patterns

您可以使用以下全局模式和通配符来映射请求：

- ？ 匹配一个字符
- *匹配路径段中的零个或多个字符
- **匹配零个或多个路径段

您还可以声明URI变量并使用@PathVariable访问其值，如以下示例所示：

```
@GetMapping("/owners/{ownerId}/pets/{petId}")
public Pet findPet(@PathVariable Long ownerId, @PathVariable Long petId) {
    // ...
}
```

更多教程请访问:www.flydean.com

您可以在类和方法级别声明URI变量，如以下示例所示：

```
@Controller
@RequestMapping("/owners/{ownerId}")
public class OwnerController {

    @GetMapping("/pets/{petId}")
    public Pet findPet(@PathVariable Long ownerId, @PathVariable Long petId) {
        // ...
    }
}
```

URI变量会自动转换为适当的类型，或者引发TypeMismatchException。 默认情况下支持简单类型（int，long，Date等），您可以注册对任何其他数据类型的支持。 请参阅类型转换和DataBinder。

您可以显式地命名URI变量（例如，@PathVariable（“ customId”）），但是如果名称相同并且您的代码是使用调试信息或Java 8上的-parameters编译器标志进行编译的，则可以省略该详细信息。 。

语法{varName：regex}声明带有正则表达式的URI变量，语法为{varName：regex}。 例如，给定URL“ /spring-web-3.0.5 .jar”，以下方法提取名称，版本和文件扩展名：

```
@GetMapping("/{name:[a-z-]+}-{version:\\d\\.\\d\\.\\d}{ext:\\.[a-z]+}")
public void handle(@PathVariable String version, @PathVariable String ext) {
    // ...
}
```

URI路径模式也可以嵌入$ {...}占位符，这些占位符在启动时通过针对本地，系统，环境和其他属性源使用PropertyPlaceHolderConfigurer进行解析。 例如，您可以使用它来基于一些外部配置参数化基本URL。

> Spring MVC使用spring-core中的PathMatcher协定和AntPathMatcher实现来进行URI路径匹配。

Pattern Comparison

当多个模式与URL匹配时，必须将它们进行比较以找到最佳匹配。这可以通过使用AntPathMatcher.getPatternComparator（String path）来完成，该工具查找更具体的模式。

如果模式的URI变量（计数为1），单通配符（计数为1）和双通配符（计数为2）的数量较少，则模式的含义不太明确。给定相等的分数，则选择更长的模式。给定相同的分数和长度，将选择具有比通配符更多URI变量的模式。

默认映射模式（/ ）**从评分中排除，并且始终排在最后。另外，前缀模式（例如/ public / ）**也被认为比其他没有双通配符的模式不太明确。

有关完整的详细信息，请参阅AntPathMatcher中的AntPatternComparator，并且请记住，您可以自定义PathMatcher实现。请参阅配置部分中的路径匹配。

后缀匹配

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.*。然后，文件扩展名用于解释请求的内容类型以用于响应（即，代替Accept标头），例如/person.pdf、/person.xml等。

当浏览器用来发送难以一致解释的Accept标头时，以这种方式使用文件扩展名是必要的。目前，这已不再是必须的，使用Accept标头应该是首选。

随着时间的流逝，文件扩展名的使用已经以各种方式证明是有问题的。当使用URI变量，路径参数和URI编码进行覆盖时，可能会引起歧义。关于基于URL的授权和安全性的推理（请参阅下一部分以了解更多详细信息）也变得更加困难。

若要完全禁用文件扩展名，必须设置以下两项：

- useSuffixPatternMatching（false），请参阅PathMatchConfigurer
- preferredPathExtension（false），请参阅ContentNegotiationConfigurer

更多教程请访问:www.flydean.com

基于URL的内容协商仍然有用（例如，在浏览器中键入URL时）。为此，我们建议使用基于查询参数的策略，以避免文件扩展名附带的大多数问题。或者，如果必须使用文件扩展名，请考虑通过ContentNegotiationConfigurer的mediaTypes属性将它们限制为已显式注册的扩展名列表。

后缀匹配和RFD

反射文件下载（RFD）攻击与XSS相似，它依赖于响应中反映的请求输入（例如，查询参数和URI变量）。但是，RFD攻击不是将JavaScript插入HTML，而是依靠浏览器切换来执行下载，并在以后双击时将响应视为可执行脚本。

在Spring MVC中，@ ResponseBody和ResponseEntity方法存在风险，因为它们可以呈现不同的内容类型，客户端可以通过URL路径扩展来请求这些内容类型。禁用后缀模式匹配并使用路径扩展进行内容协商可以降低风险，但不足以防止RFD攻击。

为了防止RFD攻击，Spring MVC在呈现响应主体之前，添加了Content-Disposition:inline;filename=f.txt 标头来建议固定且安全的下载文件。仅当URL路径包含既未列入白名单也未明确注册用于内容协商的文件扩展名时，才执行此操作。但是，当直接在浏览器中键入URL时，它可能会产生副作用。

默认情况下，许多常见的路径扩展名都被列入白名单。具有自定义HttpMessageConverter实现的应用程序可以显式注册文件扩展名以进行内容协商，以避免为这些扩展名添加Content-Disposition标头。请参阅内容类型。

Consumable Media Types

您可以根据请求的Content-Type缩小请求映射，如以下示例所示：

```
@PostMapping(path = "/pets", consumes = "application/json")
public void addPet(@RequestBody Pet pet) {
    // ...
}
```

consumes属性还支持否定表达式-例如，!text/plain表示除text / plain之外的任何内容类型。

您可以在类级别上声明一个共享的consumes属性。 但是，与大多数其他请求映射属性不同，在类级别使用方法级别时，方法级别使用属性覆盖而不是扩展类级别声明。

> MediaType为常用的媒体类型提供常量，例如APPLICATION_JSON_VALUE和APPLICATION_XML_VALUE。

Producible Media Types

您可以根据接受请求标头和控制器方法生成的内容类型列表来缩小请求映射，如以下示例所示：

```
@GetMapping(path = "/pets/{petId}", produces = "application/json;charset=UTF-8")
@ResponseBody
public Pet getPet(@PathVariable String petId) {
    // ...
}
```

媒体类型可以指定字符集。 支持否定的表达式-例如，!text/plain表示除“ text / plain”以外的任何内容类型。

> 对于JSON内容类型，即使RFC7159明确声明“未为此注册定义任何字符集参数”，也应指定UTF-8字符集，因为某些浏览器要求它正确解释UTF-8特殊字符。

您可以在类级别声明共享的Produces属性。 但是，与大多数其他请求映射属性不同，在类级别使用时，方法级别会产生属性覆盖，而不是扩展类级别的声明。

> MediaType为常用的媒体类型（例如APPLICATION_JSON_UTF8_VALUE和APPLICATION_XML_VALUE）提供常数。

Parameters, headers

更多教程请访问:www.flydean.com

您可以根据请求参数条件来缩小请求映射。 您可以测试是否存在请求参数（myParam），是否存在一个请求参数（!myParam）或特定值（myParam = myValue）。 以下示例显示如何测试特定值：

```
@GetMapping(path = "/pets/{petId}", params = "myParam=myValue")
public void findPet(@PathVariable String petId) {
    // ...
}
```

您还可以将其与请求标头条件一起使用，如以下示例所示：

```
@GetMapping(path = "/pets", headers = "myHeader=myValue")
public void findPet(@PathVariable String petId) {
    // ...
}
```

HTTP HEAD, OPTIONS

@GetMapping（和@RequestMapping（method = HttpMethod.GET））透明地支持HTTP HEAD以进行请求映射。控制器方法不需要更改。应用于javax.servlet.http.HttpServlet的响应包装器确保将Content-Length标头设置为写入的字节数（实际上未写入响应）。

@GetMapping（和@RequestMapping（method = HttpMethod.GET））被隐式映射到并支持HTTP HEAD。像处理HTTP GET一样处理HTTP HEAD请求，不同之处在于，不是写入正文，而是计算字节数并设置Content-Length头。

默认情况下，通过将“允许响应”标头设置为所有具有匹配URL模式的@RequestMapping方法中列出的HTTP方法列表来处理HTTP OPTIONS。

对于没有HTTP方法声明的@RequestMapping，将Allow标头设置为GET，HEAD，POST，PUT，PATCH，DELETE，OPTIONS。控制器方法应始终声明支持的HTTP方法（例如，通过使用HTTP方法特定的变体：@ GetMapping，@ PostMapping等）。

您可以将@RequestMapping方法显式映射到HTTP HEAD和HTTP OPTIONS，但这在通常情况下不是必需的。

Custom Annotations

Spring MVC支持将组合注解用于请求映射。 这些注解本身使用@RequestMapping进行元注解，并且旨在以更狭窄，更具体的用途重新声明@RequestMapping属性的子集（或全部）。

@ GetMapping，@ PostMapping，@ PutMapping，@ DeleteMapping和@PatchMapping是组合注解的示例。 之所以提供它们，是因为可以说，大多数控制器方法应该映射到特定的HTTP方法，而不是使用@RequestMapping，后者默认情况下与所有HTTP方法都匹配。 如果需要组合注解的示例，请查看如何声明它们。

Spring MVC还支持带有自定义请求匹配逻辑的自定义请求映射属性。 这是一个更高级的选项，它需要子类化RequestMappingHandlerMapping并覆盖getCustomMethodCondition方法，您可以在其中检查自定义属性并返回自己的RequestCondition。

Explicit Registrations

您可以以编程方式注册处理程序方法，这些方法可用于动态注册或高级案例，例如同一处理程序在不同URL下的不同实例。 下面的示例注册一个处理程序方法：

```
@Configuration
public class MyConfig {

    @Autowired
    public void setHandlerMapping(RequestMappingHandlerMapping mapping, UserHandler handler)
            throws NoSuchMethodException {

        RequestMappingInfo info = RequestMappingInfo
                .paths("/user/{id}").methods(RequestMethod.GET).build();
```

更多教程请访问:www.flydean.com

```
        Method method = UserHandler.class.getMethod("getUser", Long.class);

        mapping.registerMapping(info, handler, method);
    }

}
```

1.3.3 Handler Methods

@RequestMapping处理程序方法具有灵活的签名，可以从一系列受支持的控制器方法参数和返回值中进行选择。

方法参数

下表描述了受支持的控制器方法参数。 任何参数均不支持反应性类型。

支持JDK 8的java.util.Optional作为方法参数，并与具有必需属性（例如，@ RequestParam，@ RequestHeader等）的注解结合在一起，等效于required = false。

Controller method argument	Description
WebRequest, NativeWebRequest	通用访问请求参数以及请求和会话属性，而无需直接使用Servlet API。
javax.servlet.ServletRequest, javax.servlet.ServletResponse	选择任何特定的请求或响应类型，例如ServletRequest，HttpServletRequest或Spring的MultipartRequest，MultipartHttpServletRequest。
javax.servlet.http.HttpSession	强制会话的存在。 结果，这样的参数永远不会为空。 请注意，会话访问不是线程安全的。 如果允许多个请求并发访问会话，请考虑将RequestMappingHandlerAdapter实例的syncnizeOnSession标志设置为true。
javax.servlet.http.PushBuilder	用于程序化HTTP / 2资源推送的Servlet 4.0推送构建器API。 请注意，根据Servlet规范，如果客户端不支持HTTP / 2功能，则注入的PushBuilder实例可以为null。
java.security.Principal	当前经过身份验证的用户-可能是特定的Principal实现类（如果已知）。
HttpMethod	请求的HTTP方法
java.util.Locale	当前的请求区域设置，由可用的最特定的LocaleResolver（实际上是配置的LocaleResolver或LocaleContextResolver）确定。
java.util.TimeZone + java.time.ZoneId	与当前请求关联的时区，由LocaleContextResolver确定。
java.io.InputStream, java.io.Reader	用于访问Servlet API公开的原始请求正文。
java.io.OutputStream, java.io.Writer	用于访问Servlet API公开的原始响应正文。
@PathVariable	用于访问URI模板变量。
@MatrixVariable	用于访问URI路径段中的名称/值对
@RequestParam	用于访问Servlet请求参数，包括多部分文件。 参数值将转换为声明的方法参数类型。 参见@RequestParam以及Multipart。请注意，对于简单参数值，@ RequestParam的使用是可选的。 请参阅此表末尾的“其他任何参数”。
@RequestHeader	用于访问请求标头。 标头值将转换为声明的方法参数类型。 请参阅@RequestHeader。
@CookieValue	用于访问cookie。 Cookies值将转换为声明的方法参数类型。 请参阅@CookieValue。

更多教程请访问:www.flydean.com

@RequestBody	用于访问HTTP请求正文。 正文内容通过使用HttpMessageConverter实现转换为声明的方法参数类型。 请参阅@RequestBody。
HttpEntity<B>	用于访问请求标头和正文。 主体使用HttpMessageConverter进行转换。 请参见HttpEntity。
@RequestPart	要访问multipart / form-data请求中的零件，请使用HttpMessageConverter转换零件的主体。 请参阅多部分。
java.util.Map, org.springframework.ui.Model, org.springframework.ui.ModelMap	用于访问HTML控制器中使用的模型，并作为视图渲染的一部分公开给模板。
RedirectAttributes	指定在重定向的情况下使用的属性（即追加到查询字符串中），并指定要临时存储的属性，直到重定向后的请求为止。 请参阅重定向属性和Flash属性。
@ModelAttribute	用于访问模型中的现有属性（如果不存在，则进行实例化），并应用数据绑定和验证。 请参阅@ModelAttribute以及Model和DataBinder。请注意，@ ModelAttribute的使用是可选的（例如，设置其属性）。 请参阅此表末尾的“其他任何参数”。
Errors, BindingResult	用于访问来自命令对象的验证和数据绑定（即，@ ModelAttribute参数）的错误或来自@RequestBody或@RequestPart参数的验证的错误。 您必须在经过验证的方法参数后立即声明一个Errors或BindingResult参数。
SessionStatus + class-level @SessionAttributes	为了标记表单处理完成，将触发清除通过类级别@SessionAttributes注解声明的会话属性。 有关更多详细信息，请参见@SessionAttributes。
UriComponentsBuilder	用于准备相对于当前请求的主机，端口，方案，上下文路径以及servlet映射的文字部分的URL。 请参阅URI链接。
@SessionAttribute	对于访问任何会话属性，与由于类级@SessionAttributes声明而存储在会话中的模型属性相反。 有关更多详细信息，请参见@SessionAttribute。
@RequestAttribute	用于访问请求属性。 有关更多详细信息，请参见@RequestAttribute。
Any other argument	如果方法参数与该表中的任何较早值都不匹配，并且为简单类型（由BeanUtils # isSimpleProperty确定，则将其解析为@RequestParam。否则，将其解析为@ModelAttribute

Return Values

下表描述了受支持的控制器方法返回值。 所有返回值都支持反应性类型。

Controller method return value	Description
@ResponseBody	返回值通过HttpMessageConverter实现转换并写入响应。 请参阅@ResponseBody。
HttpEntity<B>, ResponseEntity<B>	指定完整响应（包括HTTP标头和正文）的返回值将通过HttpMessageConverter实现进行转换，并写入响应中。 请参阅ResponseEntity。
HttpHeaders	用于返回带有标头且没有正文的响应。
String	使用ViewResolver实现解析的视图名称，并与隐式模型一起使用-通过命令对象和@ModelAttribute方法确定。 处理程序方法还可以通过声明Model参数来以编程方式丰富模型（请参阅显式注册）。
View	一个View实例，用于与隐式模型一起呈现（通过命令对象和@ModelAttribute方法确定）。 处理程序方法还可以通过声明Model参数来以编程方式丰富模型（请参阅显式注册）。
java.util.Map, org.springframework.ui.Model	要添加到隐式模型的属性，其中视图名称通过RequestToViewNameTranslator隐式确定。
	要添加到模型的属性，视图名称通过

更多教程请访问:www.flydean.com

@ModelAttribute	RequestToViewNameTranslator隐式确定。请注意，@ ModelAttribute是可选的。 请参阅此表末尾的“其他任何返回值”。
ModelAndView object	要使用的视图和模型属性，以及响应状态（可选）。
void	如果返回值类型为void（或返回值为null）的方法还具有ServletResponse，OutputStream参数或@ResponseStatus批注，则认为该方法已完全处理了响应。 如果控制器进行了肯定的ETag或lastModified时间戳检查，也是如此（请参阅控制器以获取详细信息）。如果以上条件都不成立，则对于REST控制器，void返回类型也可以指示“无响应正文”，对于HTML控制器，则默认选择视图名称。
DeferredResult<V>	例如，由于某个事件或回调的结果，从任何线程异步生成任何上述返回值。 请参阅异步请求和DeferredResult。
Callable<V>	在Spring MVC管理的线程中异步产生上述任何返回值。 请参阅异步请求和可调用。
ListenableFuture<V>, java.util.concurrent.CompletionStage<V>, java.util.concurrent.CompletableFuture<V>	为方便起见（例如，当基础服务返回其中之一时），替代DeferredResult。
ResponseBodyEmitter, SseEmitter	使用HttpMessageConverter实现异步发出要写入响应的对象流。 也支持作为ResponseEntity的主体。 请参阅异步请求和HTTP流。
StreamingResponseBody	异步写入响应OutputStream。 也支持作为ResponseEntity的主体。 请参阅异步请求和HTTP流。
Reactive types — Reactor, RxJava, or others through ReactiveAdapterRegistry	DeferredResult的替代方法，其中包含收集到List的多值流（例如Flux，Observable）。对于流场景（例如，文本/事件流，应用程序/ json +流），使用SseEmitter和ResponseBodyEmitter代替，其中ServletOutputStream阻塞I / O在Spring MVC管理的线程上执行，而反压则不利于完成 每个写入。请参阅异步请求和响应类型。
Any other return value	如果返回值不是由BeanUtils确定的简单类型，则它与该表中的任何早先值都不匹配且为String或void的任何返回值均被视为视图名称（通过RequestToViewNameTranslator选择默认视图名称）。 #isSimpleProperty。 简单类型的值仍然无法解析。

Type Conversion

如果参数声明为String以外的形式，则某些表示基于String的请求输入的带注解的控制器方法参数（例如@ RequestParam，@ RequestHeader，@ PathVariable，@ MatrixVariable和@CookieValue）可能需要类型转换。

在这种情况下，将根据配置的转换器自动应用类型转换。 默认情况下，支持简单类型（int，long，Date和其他）。 您可以通过WebDataBinder（请参见DataBinder）或通过在FormattingConversionService中注册Formatter来自定义类型转换。 参见Spring字段格式。

Matrix Variables

RFC 3986讨论了路径段中的名称/值对。 在Spring MVC中，基于Tim Berners-Lee的“旧帖子”，我们将其称为“矩阵变量”，但它们也可以称为URI路径参数。

矩阵变量可以出现在任何路径段中，每个变量用分号分隔，多个值用逗号分隔（例如，/ cars; color = red，green; year = 2012）。 也可以通过重复的变量名称指定多个值（例如，color = red; color = green; color = blue）。

如果期望URL包含矩阵变量，则控制器方法的请求映射必须使用URI变量来屏蔽该变量内容，并确保可以成功地匹配请求，而与矩阵变量的顺序和状态无关。 以下示例使用矩阵变量：

```
// GET /pets/42;q=11;r=22

@GetMapping("/pets/{petId}")
public void findPet(@PathVariable String petId, @MatrixVariable int q) {
```

更多教程请访问:www.flydean.com

```
    // petId == 42
    // q == 11
}
```

鉴于所有路径段都可能包含矩阵变量，因此有时您可能需要消除矩阵变量应位于哪个路径变量的歧义。以下示例说明了如何做到这一点：

```
// 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
}
```

可以将矩阵变量定义为可选变量，并指定默认值，如以下示例所示：

```
// GET /pets/42

@GetMapping("/pets/{petId}")
public void findPet(@MatrixVariable(required=false, defaultValue="1") int q) {

    // q == 1
}
```

要获取所有矩阵变量，可以使用MultiValueMap，如以下示例所示：

```
// 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" : 22, "s" : 23]
}
```

请注意，您需要启用矩阵变量的使用。 在MVC Java配置中，您需要通过Path Matching设置带有removeSemicolonContent = false的UrlPathHelper。 在MVC XML名称空间中，可以设置<mvc：annotation-driven enable-matrix-variables =“ true” />。

@RequestParam

您可以使用@RequestParam批注将Servlet请求参数（即查询参数或表单数据）绑定到控制器中的方法参数。

以下示例显示了如何执行此操作：

```
@Controller
@RequestMapping("/pets")
public class EditPetForm {

    // ...

    @GetMapping
    public String setupForm(@RequestParam("petId") int petId, Model model) {
        Pet pet = this.clinic.loadPet(petId);
        model.addAttribute("pet", pet);
```

更多教程请访问:www.flydean.com

```
        return "petForm";
    }

    // ...

}
```

默认情况下，使用此批注的方法参数是必需的，但是您可以通过将@RequestParam批注的required标志设置为false或使用java.util.Optional包装器声明该参数，来指定方法参数是可选的。

如果目标方法参数类型不是字符串，则将自动应用类型转换。请参阅类型转换。

将参数类型声明为数组或列表，可以为同一参数名称解析多个参数值。

如果将@RequestParam批注声明为Map <String，String>或MultiValueMap <String，String>，但未在批注中指定参数名称，则将使用每个给定参数名称的请求参数值填充映射。

请注意，@ RequestParam的使用是可选的（例如，设置其属性）。默认情况下，任何简单值类型的参数（由BeanUtils＃isSimpleProperty确定）并且没有被任何其他参数解析器解析，就如同使用@RequestParam进行了注解一样。

@RequestHeader

您可以使用@RequestHeader批注将请求标头绑定到控制器中的方法参数。

考虑以下带有标头的请求：

```
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
```

以下示例获取Accept-Encoding和Keep-Alive标头的值：

```
@GetMapping("/demo")
public void handle(
        @RequestHeader("Accept-Encoding") String encoding,
        @RequestHeader("Keep-Alive") long keepAlive) {
    //...
}
```

如果目标方法的参数类型不是String，则将自动应用类型转换。 请参阅类型转换。

在Map <String，String>，MultiValueMap <String，String>或HttpHeaders参数上使用@RequestHeader批注时，将使用所有标头值填充该映射。

> 内置支持可用于将逗号分隔的字符串转换为数组或字符串集合或类型转换系统已知的其他类型。 例如，用@RequestHeader（“ Accept”）注解的方法参数可以是String类型，也可以是String []或List <String>。

@CookieValue

您可以使用@CookieValue批注将HTTP cookie的值绑定到控制器中的方法参数。

考虑带有以下cookie的请求：

```
JSESSIONID=415A4AC178C59DACE0B2C9CA727CDD84
```

以下示例显示如何获取cookie值：

更多教程请访问:www.flydean.com

```
@GetMapping("/demo")
public void handle(@CookieValue("JSESSIONID") String cookie) {
    //...
}
```

@ModelAttribute

您可以在方法参数上使用@ModelAttribute批注，以从模型访问属性，或者将其实例化（如果不存在）。 model属性还覆盖了名称与字段名称匹配的HTTP Servlet请求参数中的值。 这称为数据绑定，它使您不必处理解析和转换单个查询参数和表单字段的工作。 以下示例显示了如何执行此操作：

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public String processSubmit(@ModelAttribute Pet pet) { }
```

上面的Pet实例解析如下：

- 从模型（如果已通过使用模型添加）。
- 通过使用@SessionAttributes在HTTP会话中进行。
- 来自通过Converter传递的URI路径变量（请参见下一个示例）。
- 从默认构造函数的调用开始。
- 从调用具有与Servlet请求参数匹配的参数的“主要构造函数”开始。 参数名称是通过JavaBeans @ConstructorProperties或字节码中运行时保留的参数名称确定的。

尽管通常使用模型来用属性填充模型，但另一种替代方法是依赖于Converter <String，T>与URI路径变量约定结合使用。 在以下示例中，模型属性名称account与URI路径变量account匹配，并且通过将String帐号传递给已注册的Converter <String，Account>来加载Account：

```
@PutMapping("/accounts/{account}")
public String save(@ModelAttribute("account") Account account) {
    // ...
}
```

获取模型属性实例后，将应用数据绑定。 WebDataBinder类将Servlet请求参数名称（查询参数和表单字段）与目标Object上的字段名称进行匹配。 必要时在应用类型转换后填充匹配字段。 有关数据绑定（和验证）的更多信息，请参见验证。 有关自定义数据绑定的更多信息，请参见DataBinder。

数据绑定可能导致错误。 默认情况下，引发BindException。 但是，要检查控制器方法中的此类错误，可以在@ModelAttribute旁边立即添加BindingResult参数，如以下示例所示：

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public String processSubmit(@ModelAttribute("pet") Pet pet, BindingResult result) {
    if (result.hasErrors()) {
        return "petForm";
    }
    // ...
}
```

在某些情况下，您可能希望访问没有数据绑定的模型属性。 对于这种情况，可以将模型注入控制器中并直接访问它，或者设置@ModelAttribute（binding = false），如以下示例所示：

```
@ModelAttribute
public AccountForm setUpForm() {
    return new AccountForm();
}

@ModelAttribute
public Account findAccount(@PathVariable String accountId) {
```

更多教程请访问:www.flydean.com

```
    return accountRepository.findOne(accountId);
}

@PostMapping("update")
public String update(@Valid AccountForm form, BindingResult result,
        @ModelAttribute(binding=false) Account account) {
    // ...
}
```

您可以通过添加javax.validation.Valid注解或Spring的@Validated注解（Bean验证和Spring验证）自动在数据绑定后应用验证。 以下示例显示了如何执行此操作：

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public String processSubmit(@Valid @ModelAttribute("pet") Pet pet, BindingResult result) {
    if (result.hasErrors()) {
        return "petForm";
    }
    // ...
}
```

请注意，使用@ModelAttribute是可选的（例如，设置其属性）。 默认情况下，任何不是简单值类型（由BeanUtils # isSimpleProperty确定）且未被其他任何参数解析器解析的参数都将被视为使用@ModelAttribute注解。

@SessionAttributes

@SessionAttributes用于在请求之间的HTTP Servlet会话中存储模型属性。 它是类型级别的注解，用于声明特定控制器使用的会话属性。 这通常列出应透明地存储在会话中以供后续访问请求的模型属性的名称或模型属性的类型。

以下示例使用@SessionAttributes批注：

```
@Controller
@SessionAttributes("pet")
public class EditPetForm {
    // ...
}
```

在第一个请求上，将名称为pet的模型属性添加到模型时，该属性会自动提升到HTTP Servlet会话并保存在该会话中。 它会一直保留在那里，直到另一个控制器方法使用SessionStatus方法参数来清除存储，如以下示例所示：

```
@Controller
@SessionAttributes("pet")
public class EditPetForm {

    // ...

    @PostMapping("/pets/{id}")
    public String handle(Pet pet, BindingResult errors, SessionStatus status) {
        if (errors.hasErrors) {
            // ...
        }
            status.setComplete();
            // ...
        }
    }
}
```

@SessionAttribute

如果您需要访问全局存在（例如，在控制器外部（例如，通过过滤器）管理）并且可能存在或可能不存在的预先存在的会话属性，则可以在方法参数上使用@SessionAttribute注解，例如 以下示例显示：

更多教程请访问:www.flydean.com

```
@RequestMapping("/")
public String handle(@SessionAttribute User user) {
    // ...
}
```

对于需要添加或删除会话属性的用例，请考虑将org.springframework.web.context.request.WebRequest或javax.servlet.http.HttpSession注入到控制器方法中。

要在控制器工作流中将模型属性临时存储在会话中，请考虑使用@SessionAttributes，如@SessionAttributes中所述。

@RequestAttribute

与@SessionAttribute相似，您可以使用@RequestAttribute批注来访问先前创建的预先存在的请求属性（例如，通过Servlet Filter或HandlerInterceptor）：

```
@GetMapping("/")
public String handle(@RequestAttribute Client client) {
    // ...
}
```

Redirect Attributes

默认情况下，所有模型属性均被视为在重定向URL中作为URI模板变量公开。在其余属性中，那些属于原始类型或原始类型的集合或数组的属性会自动附加为查询参数。

如果专门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理想的结果。但是，在带注解的控制器中，模型可以包含为渲染目的添加的其他属性（例如，下拉字段值）。为避免此类属性出现在URL中的可能性，@RequestMapping方法可以声明RedirectAttributes类型的参数，并使用它指定可用于RedirectView的确切属性。如果该方法确实重定向，则使用RedirectAttributes的内容。否则，将使用模型的内容。

RequestMappingHandlerAdapter提供了一个名为ignoreDefaultModelOnRedirect的标志，您可以使用该标志指示如果控制器方法重定向，则绝不要使用默认Model的内容。相反，控制器方法应声明一个RedirectAttributes类型的属性，或者，如果没有这样做，则不应将任何属性传递给RedirectView。 MVC名称空间和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，展开重定向URL时，本请求中的URI模板变量会自动变为可用，您无需通过Model或RedirectAttributes显式添加它们。以下示例显示了如何定义重定向：

```
@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使用flash attributes。 与其他重定向属性不同，flash attributes保存在HTTP会话中（因此不会出现在URL中）。 有关更多信息，请参见Flash属性。

Flash Attributes

Flash属性为一个请求提供了一种存储要在另一个请求中使用的属性的方式。重定向时最常需要此操作，例如Post-Redirect-Get模式。 Flash属性在重定向之前（通常在会话中）被临时保存，以便在重定向之后可供请求使用，并立即被删除。

Spring MVC有两个主要的抽象来支持Flash属性。 FlashMap用于保存Flash属性，而FlashMapManager用于存储，检索和管理FlashMap实例。

更多教程请访问:www.flydean.com

Flash属性支持始终处于“打开”状态，无需显式启用。但是，如果不使用它，则永远不会导致HTTP会话创建。在每个请求上，都有一个“输入” FlashMap，其属性是从上一个请求（如果有）传递过来的，而“输出” FlashMap的属性是为后续请求保存的。可以通过RequestContextUtils中的静态方法从Spring MVC中的任何位置访问这两个FlashMap实例。

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RedirectAttributes类型的参数，并使用它为重定向方案添加Flash属性。通过RedirectAttributes添加的Flash属性会自动传播到“输出” FlashMap。同样，重定向后，来自“输入” FlashMap的属性会自动添加到服务于目标URL的控制器的模型中。

> 将请求与Flash属性匹配 Flash属性的概念存在于许多其他Web框架中，并已证明有时会遇到并发问题。 这是因为根据定义，闪存属性将存储到下一个请求。 但是，“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，过早删除了闪存属性。
>
> 为了减少此类问题的可能性，RedirectView会自动使用目标重定向URL的路径和查询参数“标记” FlashMap实例。反过来，默认FlashMapManager在查找“输入” FlashMap时会将信息与传入请求匹配。
>
> 这不能完全消除并发问题的可能性，但可以通过重定向URL中已经可用的信息大大减少并发问题。 因此，我们建议您主要将Flash属性用于重定向方案。

Multipart

启用MultipartResolver后，带有multipart / form-data的POST请求的内容将被解析并作为常规请求参数进行访问。 以下示例访问一个常规表单字段和一个上载文件：

```
@Controller
public class FileUploadController {

    @PostMapping("/form")
    public String handleFormUpload(@RequestParam("name") String name,
            @RequestParam("file") MultipartFile file) {

        if (!file.isEmpty()) {
            byte[] bytes = file.getBytes();
            // store the bytes somewhere
            return "redirect:uploadSuccess";
        }
        return "redirect:uploadFailure";
    }
}
```

将参数类型声明为List <MultipartFile>允许解析相同参数名称的多个文件。

如果将@RequestParam批注声明为Map <String，MultipartFile>或MultiValueMap <String，MultipartFile>，但未在批注中指定参数名称，则将使用每个给定参数名称的多部分文件填充映射。

> 通过Servlet 3.0多部分解析，您还可以声明javax.servlet.http.Part而不是Spring的MultipartFile作为方法参数或集合值类型。

您还可以将多部分内容用作数据绑定到命令对象的一部分。 例如，前面示例中的表单字段和文件可以是表单对象上的字段，如以下示例所示：

```
class MyForm {

    private String name;

    private MultipartFile file;

    // ...
}

@Controller
public class FileUploadController {
```

更多教程请访问:www.flydean.com

```
    @PostMapping("/form")
    public String handleFormUpload(MyForm form, BindingResult errors) {
        if (!form.getFile().isEmpty()) {
            byte[] bytes = form.getFile().getBytes();
            // store the bytes somewhere
            return "redirect:uploadSuccess";
        }
        return "redirect:uploadFailure";
    }
}
```

在RESTful服务方案中，也可以从非浏览器客户端提交多部分请求。 以下示例显示了带有JSON的文件：

```
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 ...
```

您可以使用@RequestParam作为字符串访问“元数据”部分，但您可能希望将其从JSON反序列化（类似于@RequestBody）。 在使用HttpMessageConverter进行转换后，使用@RequestPart批注来访问多部分：

```
@PostMapping("/")
public String handle(@RequestPart("meta-data") MetaData metadata,
        @RequestPart("file-data") MultipartFile file) {
    // ...
}
```

您可以将@RequestPart与javax.validation.Valid结合使用，也可以使用Spring的@Validated注解，这两种注解都会导致应用标准Bean验证。 默认情况下，验证错误会导致MethodArgumentNotValidException，并将其转换为400（BAD_REQUEST）响应。 另外，您可以通过Errors或BindingResult参数在控制器内本地处理验证错误，如以下示例所示：

```
@PostMapping("/")
public String handle(@Valid @RequestPart("meta-data") MetaData metadata,
        BindingResult result) {
    // ...
}
```

@RequestBody

您可以使用@RequestBody批注使请求正文通过HttpMessageConverter读取并反序列化为Object。 以下示例使用@RequestBody参数：

```
@PostMapping("/accounts")
public void handle(@RequestBody Account account) {
    // ...
}
```

更多教程请访问:www.flydean.com

您可以使用MVC Config的“消息转换器”选项来配置或自定义消息转换。

您可以将@RequestBody与javax.validation.Valid或Spring的@Validated注解结合使用，这两种注解都会导致应用标准Bean验证。 默认情况下，验证错误会导致MethodArgumentNotValidException，并将其转换为400（BAD_REQUEST）响应。 另外，您可以通过Errors或BindingResult参数在控制器内本地处理验证错误，如以下示例所示：

```
@PostMapping("/accounts")
public void handle(@Valid @RequestBody Account account, BindingResult result) {
    // ...
}
```

HttpEntity

HttpEntity或多或少与使用@RequestBody相同，但它基于公开请求标头和正文的容器对象。 以下清单显示了一个示例：

```
@PostMapping("/accounts")
public void handle(HttpEntity<Account> entity) {
    // ...
}
```

@ResponseBody

您可以在方法上使用@ResponseBody批注，以通过HttpMessageConverter将返回序列化为响应主体。 以下清单显示了一个示例：

```
@GetMapping("/accounts/{id}")
@ResponseBody
public Account handle() {
    // ...
}
```

在类级别也支持@ResponseBody，在这种情况下，所有控制器方法都将继承它。 这就是@RestController的效果，它只不过是带有@Controller和@ResponseBody标记的元注解。

您可以将@ResponseBody与反应类型一起使用。 有关更多详细信息，请参见异步请求和响应类型。

您可以使用MVC Config的“消息转换器”选项来配置或自定义消息转换。

您可以将@ResponseBody方法与JSON序列化视图结合使用。 有关详细信息，请参见Jackson JSON。

ResponseEntity

ResponseEntity类似于@ResponseBody，但具有状态和标头。 例如：

```
@GetMapping("/something")
public ResponseEntity<String> handle() {
    String body = ... ;
    String etag = ... ;
    return ResponseEntity.ok().eTag(etag).build(body);
}
```

Spring MVC支持使用单值反应类型异步生成ResponseEntity，和/或为主体使用单值和多值反应类型。

Jackson JSON

Spring提供了对Jackson JSON库的支持。

更多教程请访问:www.flydean.com

Spring MVC为Jackson的序列化视图提供了内置支持，该视图仅可呈现Object中所有字段的一部分。 要将其与@ResponseBody或ResponseEntity控制器方法一起使用，可以使用Jackson的@JsonView批注来激活序列化视图类，如以下示例所示：

```
@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;
    }
}
```

> @JsonView允许使用一组视图类，但是每个控制器方法只能指定一个。 如果需要激活多个视图，则可以使用复合界面。

对于依赖视图解析的控制器，可以将序列化视图类添加到模型中，如以下示例所示：

```
@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";
    }
}
```

1.3.4 Model

您可以使用@ModelAttribute批注：

- 在@RequestMapping方法中的方法参数上，可从模型创建或访问对象，并将其通过WebDataBinder绑定到请求。

更多教程请访问:www.flydean.com

- 作为@Controller或@ControllerAdvice类中的方法级注解，可在任何@RequestMapping方法调用之前帮助初始化模型。
- 在@RequestMapping方法上标记其返回值的是模型属性。

本节讨论@ModelAttribute方法-前面列表中的第二项。控制器可以具有任意数量的@ModelAttribute方法。所有此类方法均在同一控制器中的@RequestMapping方法之前调用。也可以通过@ControllerAdvice在控制器之间共享@ModelAttribute方法。有关更多详细信息，请参见“控制器建议”部分。

@ModelAttribute方法具有灵活的方法签名。它们支持与@RequestMapping方法相同的许多参数，除了@ModelAttribute本身或与请求正文相关的任何东西。

下面的示例显示一个@ModelAttribute方法：

```
@ModelAttribute
public void populateModel(@RequestParam String number, Model model) {
    model.addAttribute(accountRepository.findAccount(number));
    // add more ...
}
```

以下示例仅添加一个属性：

```
@ModelAttribute
public Account addAccount(@RequestParam String number) {
    return accountRepository.findAccount(number);
}
```

> 如果未明确指定名称，则根据Object 类型选择默认名称，如javadoc中针对的解释Conventions。您始终可以使用重载addAttribute方法或通过nameon 的属性@ModelAttribute（用于返回值）来分配显式名称。

您也可以将@ModelAttribute用作@RequestMapping方法上的方法级注解，在这种情况下，@ RequestMapping方法的返回值将解释为模型属性。 通常不需要这样做，因为它是HTML控制器的默认行为，除非返回值是一个String，否则它将被解释为视图名称。 @ModelAttribute还可以自定义模型属性名称，如以下示例所示：

```
@GetMapping("/accounts/{id}")
@ModelAttribute("myAccount")
public Account handle() {
    // ...
    return account;
}
```

1.3.5 DataBinder

@Controller或@ControllerAdvice类可以具有用于初始化WebDataBinder实例的@InitBinder方法，而这些方法又可以：

- 将请求参数（即表单或查询数据）绑定到模型对象。
- 将基于字符串的请求值（例如请求参数，路径变量，标头，Cookie等）转换为控制器方法参数的目标类型。
- 呈现HTML表单时，将模型对象的值格式化为String值。

@InitBinder方法可以注册特定于控制器的java.bean.PropertyEditor或Spring Converter和Formatter组件。 另外，您可以使用MVC配置在全局共享的FormattingConversionService中注册Converter和Formatter类型。

@InitBinder方法支持与@RequestMapping方法相同的许多参数，除了@ModelAttribute（命令对象）参数。 通常，它们使用WebDataBinder参数（用于注册）和无效的返回值声明。 以下清单显示了一个示例：

```
@Controller
public class FormController {
```

更多教程请访问:www.flydean.com

```
    @InitBinder
    public void initBinder(WebDataBinder binder) {
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        dateFormat.setLenient(false);
        binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, false));
    }

    // ...
}
```

另外，当通过共享的FormattingConversionService使用基于Formatter的设置时，可以重新使用相同的方法并注册特定于控制器的Formatter实现，如以下示例所示：

```
@Controller
public class FormController {

    @InitBinder
    protected void initBinder(WebDataBinder binder) {
        binder.addCustomFormatter(new DateFormatter("yyyy-MM-dd"));
    }

    // ...
}
```

1.3.6 Exceptions

@Controller和@ControllerAdvice类可以具有@ExceptionHandler方法来处理控制器方法中的异常，如以下示例所示：

```
@Controller
public class SimpleController {

    // ...

    @ExceptionHandler
    public ResponseEntity<String> handle(IOException ex) {
        // ...
    }
}
```

该异常可能与正在传播的顶级异常（即，直接IOException被抛出）匹配，也可能与顶级包装器异常（例如，包装在IllegalStateException内部的IOException）内的直接原因匹配。

对于匹配的异常类型，如前面的示例所示，最好将目标异常声明为方法参数。 当多个异常方法匹配时，根源异常匹配通常比原因异常匹配更可取。 更具体地说，ExceptionDepthComparator用于根据从引发的异常类型开始的深度对异常进行排序。

另外，注解声明可以缩小异常类型以使其匹配，如以下示例所示：

```
@ExceptionHandler({FileSystemException.class, RemoteException.class})
public ResponseEntity<String> handle(IOException ex) {
    // ...
}
```

您甚至可以使用带有非常通用的参数签名的特定异常类型的列表，如以下示例所示：

```
@ExceptionHandler({FileSystemException.class, RemoteException.class})
public ResponseEntity<String> handle(Exception ex) {
    // ...
}
```

更多教程请访问:www.flydean.com

> 根和原因异常匹配之间的区别可能令人惊讶。
>
> 在前面显示的IOException变体中，通常使用实际的FileSystemException或RemoteException实例作为参数来调用该方法，因为这两个实例均从IOException扩展。 但是，如果任何此类匹配异常都在本身是IOException的包装器异常中传播，则传入的异常实例就是该包装器异常。
>
> 在handle（Exception）变体中，行为甚至更简单。 在包装方案中，总是使用包装程序异常来调用此方法，在这种情况下，实际匹配的异常可以通过ex.getCause（）找到。 仅当将它们作为顶级异常抛出时，传入的异常才是实际的FileSystemException或RemoteException实例。

我们通常建议您在参数签名中尽可能具体，以减少根类型和原因异常类型之间不匹配的可能性。考虑将多重匹配方法分解为单独的@ExceptionHandler方法，每个方法均通过其签名匹配单个特定的异常类型。

在多@ControllerAdvice安排中，我们建议在以相应顺序优先的@ControllerAdvice上声明您的主根异常映射。虽然根源异常匹配是原因的首选，但这是在给定控制器或@ControllerAdvice类的方法之间定义的。这意味着优先级较高的@ControllerAdvice Bean上的原因匹配优于优先级较低的@ControllerAdvice Bean上的任何匹配（例如，根）。

最后但并非最不重要的一点是，@ExceptionHandler方法实现可以选择通过以原始形式重新抛出异常来退出处理给定异常实例。在仅对根级别匹配或无法静态确定的特定上下文中的匹配感兴趣的情况下，这很有用。重新抛出的异常会在其余的解析链中传播，就像给定的@ExceptionHandler方法最初不会匹配一样。

Spring MVC中对@ExceptionHandler方法的支持建立在DispatcherServlet级别HandlerExceptionResolver机制上。

Method Arguments

@ExceptionHandler 方法支持以下参数：

Method argument	Description
Exception type	用于访问引发的异常。
HandlerMethod	用于访问引发异常的控制器方法。
WebRequest, NativeWebRequest	对请求参数以及请求和会话属性的常规访问，而无需直接使用Servlet API。
javax.servlet.ServletRequest, javax.servlet.ServletResponse	选择任何特定的请求或响应类型（例如ServletRequest或HttpServletRequest或或Spring的MultipartRequest或MultipartHttpServletRequest）。
javax.servlet.http.HttpSession	强制会话的存在。结果，这种论据永远不会null。请注意，会话访问不是线程安全的。考虑将RequestMappingHandlerAdapter实例的synchronizeOnSession标志设置 为true是否允许多个请求同时访问会话。
java.security.Principal	当前经过身份验证的用户-可能是特定的Principal实现类（如果已知）。
HttpMethod	请求的HTTP方法。
java.util.Locale	当前请求的语言环境，由最具体的LocaleResolver可用语言决定，实际上是配置的LocaleResolver或LocaleContextResolver。
java.util.TimeZone, java.time.ZoneId	与当前请求关联的时区，由决定LocaleContextResolver。
java.io.OutputStream, java.io.Writer	用于访问原始响应主体（由Servlet API公开）。
java.util.Map, org.springframework.ui.Model, org.springframework.ui.ModelMap	用于访问模型以进行错误响应。永远是空的。
RedirectAttributes	指定在重定向的情况下使用的属性（要附加到查询字符串中），并指定要临时存储的属性，直到重定向后的请求为止。请参阅重定向属性和Flash属性。
	与访问由于类级@SessionAttributes声明而存储在会话中的模型属性相

更多教程请访问:www.flydean.com

@SessionAttribute	反，用于访问任何会话属性。请参阅@SessionAttribute以获取更多详细信息。
@RequestAttribute	用于访问请求属性。请参阅@RequestAttribute以获取更多详细信息。

返回值

@ExceptionHandler 方法支持以下返回值：

返回值	描述
@ResponseBody	返回值通过HttpMessageConverter实例转换并写入响应。请参阅@ResponseBody。
HttpEntity， ResponseEntity	返回值指定完整的响应（包括HTTP标头和正文）将通过HttpMessageConverter实例转换并写入响应。参见ResponseEntity。
String	一个视图名称，将通过ViewResolver实现来解析，并与隐式模型一起使用-通过命令对象和@ModelAttribute方法确定。该处理程序方法还可以通过声明一个Model 参数（如前所述）以编程方式丰富模型。
View	一个View实例，用于与隐式模型一起呈现（通过命令对象和@ModelAttribute方法确定）。 该处理程序方法还可以通过声明Model参数（如前所述）以编程方式丰富模型。
java.util.Map，org.springframework.ui.Model	要添加到隐式模型的属性，其视图名称是通过隐式确定的RequestToViewNameTranslator。
@ModelAttribute	要添加到模型的属性，其视图名称通过隐式确定RequestToViewNameTranslator。请注意，这@ModelAttribute是可选的。请参见表末尾的“其他任何返回值”。
ModelAndView 宾语	要使用的视图和模型属性，以及响应状态（可选）。
void	用的方法void返回类型（或null返回值）被认为已经完全处理的响应，如果它也具有ServletResponse一个OutputStream参数，或@ResponseStatus注解。如果控制器进行了肯定检查ETag或lastModified时间戳检查，则情况也是如此（有关详细信息，请参阅控制器）。如果以上条件都不成立，则void返回类型还可以为REST控制器指示“无响应正文”，或者为HTML控制器指示默认视图名称选择。
任何其他返回值	如果返回值与上述任何一个都不匹配并且不是简单类型（由BeanUtils # isSimpleProperty确定 ），则默认情况下会将其视为要添加到模型的模型属性。如果是简单类型，则仍然无法解析。

REST API exceptions

REST服务的常见要求是在响应正文中包含错误详细信息。 Spring框架不会自动执行此操作，因为响应主体中错误详细信息的表示是特定于应用程序的。 但是，@ RestController可以将@ExceptionHandler方法与ResponseEntity返回值一起使用，以设置响应的状态和主体。 也可以在@ControllerAdvice类中声明此类方法，以将其全局应用。

在响应主体中使用错误详细信息实现全局异常处理的应用程序应考虑扩展ResponseEntityExceptionHandler，它提供了Spring MVC引发的异常的处理并提供了自定义响应主体的钩子。 要使用此功能，请创建ResponseEntityExceptionHandler的子类，并使用@ControllerAdvice对其进行注解，重写必要的方法，并将其声明为Spring bean。

1.3.7 Controller Advice

通常，@ ExceptionHandler，@ InitBinder和@ModelAttribute方法在声明它们的@Controller类（或类层次结构）中应用。如果要使此类方法更全局地应用（跨控制器），则可以在标有@ControllerAdvice或@RestControllerAdvice的类中声明它们。

更多教程请访问:www.flydean.com

@ControllerAdvice标有@Component，这意味着可以通过组件扫描将此类注册为Spring Bean。@RestControllerAdvice还是一个标有@ControllerAdvice和@ResponseBody的元注解，这实际上意味着@ExceptionHandler方法通过消息转换（与视图分辨率或模板渲染）呈现到响应主体。

在启动时，@RequestMapping和@ExceptionHandler方法的基础结构类检测@ControllerAdvice类型的Spring bean，然后在运行时应用其方法。全局@ExceptionHandler方法（来自@ControllerAdvice）在本地方法（来自@Controller）之后应用。相比之下，全局@ModelAttribute和@InitBinder方法在本地方法之前应用。

默认情况下，@ ControllerAdvice方法适用于每个请求（即所有控制器），但是您可以通过使用批注上的属性将其范围缩小到控制器的子集，如以下示例所示：

```
// Target all Controllers annotated with @RestController
@ControllerAdvice(annotations = RestController.class)
public class ExampleAdvice1 {}

// Target all Controllers within specific packages
@ControllerAdvice("org.example.controllers")
public class ExampleAdvice2 {}

// Target all Controllers assignable to specific classes
@ControllerAdvice(assignableTypes = {ControllerInterface.class, AbstractController.class})
public class ExampleAdvice3 {}
```

前面示例中的选择器在运行时进行评估，如果广泛使用，可能会对性能产生负面影响。 有关更多详细信息，请参见@ControllerAdvice javadoc。

更多教程请访问:www.flydean.com

1.4 URI链接

本部分介绍了Spring框架中可用于URI的各种选项。

1.4.1 UriComponents

UriComponentsBuilder有助于从具有变量的URI模板中构建URI，如以下示例所示：

```
UriComponents uriComponents = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}")
        .queryParam("q", "{q}")
        .encode()
        .build();

URI uri = uriComponents.expand("Westin", "123").toUri();
```

可以将前面的示例合并为一个链，并通过buildAndExpand进行缩短，如以下示例所示：

```
URI uri = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}")
        .queryParam("q", "{q}")
        .encode()
        .buildAndExpand("Westin", "123")
        .toUri();
```

您可以通过直接转到URI（这意味着编码）来进一步缩短它，如以下示例所示：

```
URI uri = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}")
        .queryParam("q", "{q}")
        .build("Westin", "123");
```

您可以使用完整的URI模板进一步缩短它，如以下示例所示：

```
URI uri = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}?q={q}")
        .build("Westin", "123");
```

1.4.2 UriBuilder

UriComponentsBuilder实现UriBuilder。 您可以依次使用UriBuilderFactory创建UriBuilder。 UriBuilderFactory和UriBuilder一起提供了一种可插入的机制，可以基于共享配置（例如基本URL，编码首选项和其他详细信息）从URI模板构建URI。

您可以使用UriBuilderFactory配置RestTemplate和WebClient以自定义URI的准备。 DefaultUriBuilderFactory是UriBuilderFactory的默认实现，该实现在内部使用UriComponentsBuilder并公开共享的配置选项。

以下示例显示如何配置RestTemplate：

```
// import org.springframework.web.util.DefaultUriBuilderFactory.EncodingMode;

String baseUrl = "https://example.org";
DefaultUriBuilderFactory factory = new DefaultUriBuilderFactory(baseUrl);
```

更多教程请访问:www.flydean.com

```
factory.setEncodingMode(EncodingMode.TEMPLATE_AND_VARIABLES);

RestTemplate restTemplate = new RestTemplate();
restTemplate.setUriTemplateHandler(factory);
```

下面的例子配置了WebClient：

```
// import org.springframework.web.util.DefaultUriBuilderFactory.EncodingMode;

String baseUrl = "https://example.org";
DefaultUriBuilderFactory factory = new DefaultUriBuilderFactory(baseUrl);
factory.setEncodingMode(EncodingMode.TEMPLATE_AND_VARIABLES);

WebClient client = WebClient.builder().uriBuilderFactory(factory).build();
```

此外，您也可以直接使用DefaultUriBuilderFactory。 它类似于使用UriComponentsBuilder，但不是静态工厂方法，它是一个包含配置和首选项的实际实例，如以下示例所示：

```
String baseUrl = "https://example.com";
DefaultUriBuilderFactory uriBuilderFactory = new DefaultUriBuilderFactory(baseUrl);

URI uri = uriBuilderFactory.uriString("/hotels/{hotel}")
        .queryParam("q", "{q}")
        .build("Westin", "123");
```

1.4.3 URI Encoding

UriComponentsBuilder在两个级别公开了编码选项：

- UriComponentsBuilder # encode（）：首先对URI模板进行预编码，然后在扩展时严格对URI变量进行编码。
- UriComponents # encode（）：扩展URI变量后，对URI组件进行编码。

这两个选项都用转义的八位字节替换非ASCII和非法字符。 但是，第一个选项还会替换出现在URI变量中的具有保留含义的字符。

> 考虑“;”，这在路径上是合法的，但具有保留的含义。 第一个选项代替“;” URI变量中带有“%3B”，但URI模板中没有。 相比之下，第二个选项永远不会替换“;”，因为它是路径中的合法字符。

在大多数情况下，第一个选项可能会产生预期的结果，因为它将URI变量视为要完全编码的不透明数据，而选项2仅在URI变量有意包含保留字符的情况下才有用。

以下示例使用第一个选项：

```
URI uri = UriComponentsBuilder.fromPath("/hotel list/{city}")
            .queryParam("q", "{q}")
            .encode()
            .buildAndExpand("New York", "foo+bar")
            .toUri();

    // Result is "/hotel%20list/New%20York?q=foo%2Bbar"
```

您可以通过直接转到URI（这意味着编码）来缩短前面的示例，如以下示例所示：

```
URI uri = UriComponentsBuilder.fromPath("/hotel list/{city}")
            .queryParam("q", "{q}")
            .build("New York", "foo+bar")
```

您可以使用完整的URI模板进一步缩短它，如以下示例所示：

更多教程请访问:www.flydean.com

```
URI uri = UriComponentsBuilder.fromPath("/hotel list/{city}?q={q}")
            .build("New York", "foo+bar")
```

WebClient和RestTemplate通过UriBuilderFactory策略在内部扩展和编码URI模板。 两者都可以使用自定义策略进行配置。 如下例所示：

```
String baseUrl = "https://example.com";
DefaultUriBuilderFactory factory = new DefaultUriBuilderFactory(baseUrl)
factory.setEncodingMode(EncodingMode.TEMPLATE_AND_VALUES);

// Customize the RestTemplate..
RestTemplate restTemplate = new RestTemplate();
restTemplate.setUriTemplateHandler(factory);

// Customize the WebClient..
WebClient client = WebClient.builder().uriBuilderFactory(factory).build();
```

DefaultUriBuilderFactory实现在内部使用UriComponentsBuilder来扩展和编码URI模板。作为工厂，它提供了一个位置，可以根据以下一种编码模式来配置编码方法：

- TEMPLATE_AND_VALUES：使用UriComponentsBuilder # encode（）（对应于较早列表中的第一个选项）对URI模板进行预编码，并在扩展时严格编码URI变量。
- VALUES_ONLY：不对URI模板进行编码，而是在将其扩展到模板之前通过UriUtils # encodeUriUriVariables对URI变量进行严格编码。
- URI_COMPONENTS：在扩展URI变量后，使用UriComponents # encode（）（对应于先前列表中的第二个选项）对URI组件值进行编码。
- NONE：未应用编码。

由于历史原因和向后兼容性，将RestTemplate设置为EncodingMode.URI_COMPONENTS。 WebClient依赖于DefaultUriBuilderFactory中的默认值，该默认值已从5.0.x中的EncodingMode.URI_COMPONENTS更改为5.1中的EncodingMode.TEMPLATE_AND_VALUES。

1.4.4 Relative Servlet Requests

您可以使用ServletUriComponentsBuilder创建相对于当前请求的URI，如以下示例所示：

```
HttpServletRequest request = ...

// Re-uses host, scheme, port, path and query string...

ServletUriComponentsBuilder ucb = ServletUriComponentsBuilder.fromRequest(request)
        .replaceQueryParam("accountId", "{id}").build()
        .expand("123")
        .encode();
```

您可以创建相对于上下文路径的URI，如以下示例所示：

```
// Re-uses host, port and context path...

ServletUriComponentsBuilder ucb = ServletUriComponentsBuilder.fromContextPath(request)
        .path("/accounts").build()
```

您可以创建相对于Servlet的URI（例如/ main / *），如以下示例所示：

```
// Re-uses host, port, context path, and Servlet prefix...
```

更多教程请访问:www.flydean.com

```
ServletUriComponentsBuilder ucb = ServletUriComponentsBuilder.fromServletMapping(request)
        .path("/accounts").build()
```

> 从5.1开始，ServletUriComponentsBuilder会忽略来自Forwarded和X-Forwarded- *标头的信息，这些标头指定了客户端起源的地址。 考虑使用ForwardedHeaderFilter提取和使用或丢弃此类标头。

1.4.5 Links to Controllers

Spring MVC提供了一种准备到控制器方法的链接的机制。 例如，以下MVC控制器允许创建链接：

```
@Controller
@RequestMapping("/hotels/{hotel}")
public class BookingController {

    @GetMapping("/bookings/{booking}")
    public ModelAndView getBooking(@PathVariable Long booking) {
        // ...
    }
}
```

您可以通过按名称引用方法来准备链接，如以下示例所示：

```
UriComponents uriComponents = MvcUriComponentsBuilder
    .fromMethodName(BookingController.class, "getBooking", 21).buildAndExpand(42);

URI uri = uriComponents.encode().toUri();
```

在前面的示例中，我们提供了实际的方法参数值（在本例中为long值：21），用作路径变量并插入到URL中。 此外，我们提供值42来填充所有剩余的URI变量，例如从类型级别请求映射继承的hotel变量。 如果该方法具有更多参数，则可以为URL不需要的参数提供null。 通常，只有@PathVariable和@RequestParam参数与构造URL有关。

还有其他使用MvcUriComponentsBuilder的方法。 例如，您可以使用类似于代理的模拟测试技术来避免按名称引用控制器方法，如以下示例所示（该示例假定静态导入MvcUriComponentsBuilder.on）：

```
UriComponents uriComponents = MvcUriComponentsBuilder
    .fromMethodCall(on(BookingController.class).getBooking(21)).buildAndExpand(42);

URI uri = uriComponents.encode().toUri();
```

> 当控制器方法签名可用于fromMethodCall的链接创建时，其设计受到限制。 除了需要适当的参数签名外，返回类型还存在技术限制（即，为链接生成器调用生成运行时代理），因此返回类型不得为最终值。 特别是，视图名称的通用String返回类型在这里不起作用。 您应该改用ModelAndView甚至普通对象（具有String返回值）。

较早的示例在MvcUriComponentsBuilder中使用静态方法。 在内部，它们依靠ServletUriComponentsBuilder从当前请求的方案，主机，端口，上下文路径和Servlet路径准备基本URL。 在大多数情况下，此方法效果很好。 但是，有时可能不足。 例如，您可能不在请求的上下文之内（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，从请求路径中删除且需要重新设置的语言环境前缀）。 插入链接）。

在这种情况下，可以使用静态的fromXxx重载方法，这些方法接受UriComponentsBuilder以使用基本URL。 或者，您可以使用基本URL创建MvcUriComponentsBuilder的实例，然后使用基于实例的withXxx方法。 例如，以下清单使用withMethodCall：

```
UriComponentsBuilder base = ServletUriComponentsBuilder.fromCurrentContextPath().path("/en");
MvcUriComponentsBuilder builder = MvcUriComponentsBuilder.relativeTo(base);
builder.withMethodCall(on(BookingController.class).getBooking(21)).buildAndExpand(42);
```

更多教程请访问:www.flydean.com

```
URI uri = uriComponents.encode().toUri();
```

从5.1开始，MvcUriComponentsBuilder将忽略来自Forwarded和X-Forwarded- *标头的信息，这些标头指定了客户端起源的地址。 考虑使用ForwardedHeaderFilter提取和使用或丢弃此类标头。

1.4.6 Links in Views

在Thymeleaf，FreeMarker或JSP之类的视图中，您可以通过引用每个请求映射的隐式或显式分配的名称来构建到带注解的控制器的链接。

考虑以下示例：

```
@RequestMapping("/people/{id}/addresses")
public class PersonAddressController {

    @RequestMapping("/{country}")
    public HttpEntity getAddress(@PathVariable String country) { ... }
}
```

给定前面的控制器，您可以按照以下步骤准备来自JSP的链接：

```
<%@ taglib uri="http://www.springframework.org/tags" prefix="s" %>
...
<a href="${s:mvcUrl('PAC#getAddress').arg(0,'US').buildAndExpand('123')}">Get Address</a>
```

上面的示例依赖于Spring标记库中声明的mvcUrl函数（即META-INF / spring.tld），但是很容易定义您自己的函数或为其他模板技术准备类似的函数。

这是这样的。 在启动时，每个@RequestMapping都会通过HandlerMethodMappingNamingStrategy分配一个默认名称，该默认名称的实现使用类的大写字母和方法名称（例如，ThingController中的getThing方法变为“ TC # getThing”）。 如果名称冲突，则可以使用@RequestMapping（name =“ ..”）来分配一个明确的名称，或实现自己的HandlerMethodMappingNamingStrategy。

更多教程请访问:www.flydean.com

1.5 异步请求

Spring MVC与Servlet 3.0异步请求处理具有广泛的集成：

- 控制器方法中的DeferredResult和Callable返回值，并为单个异步返回值提供基本支持。
- 控制器可以流式传输多个值，包括SSE和原始数据。
- 控制器可以使用反应式客户端并返回反应式类型以进行响应处理。

1.5.1 DeferredResult

一旦在Servlet容器中启用了异步请求处理功能，控制器方法就可以使用DeferredResult包装任何受支持的控制器方法返回值，如以下示例所示：

```
@GetMapping("/quotes")
@ResponseBody
public DeferredResult<String> quotes() {
    DeferredResult<String> deferredResult = new DeferredResult<String>();
    // Save the deferredResult somewhere..
    return deferredResult;
}

// From some other thread...
deferredResult.setResult(data);
```

控制器可以从另一个线程异步生成返回值，例如，响应外部事件（JMS消息），计划任务或其他事件。

1.5.2 Callable

控制器可以使用java.util.concurrent.Callable包装任何受支持的返回值，如以下示例所示：

```
@PostMapping
public Callable<String> processUpload(final MultipartFile file) {

    return new Callable<String>() {
        public String call() throws Exception {
            // ...
            return "someView";
        }
    };

}
```

然后，可以通过配置的TaskExecutor运行给定任务来获取返回值。

1.5.3 Processing

这是Servlet异步请求处理的非常简洁的概述：

- 可以通过调用request.startAsync（）将ServletRequest置于异步模式。这样做的主要作用是可以退出Servlet（以及所有过滤器），但是响应保持打开状态，以便以后完成处理。
- 对request.startAsync（）的调用返回AsyncContext，可将其用于进一步控制异步处理。例如，它提供了分派方法，该方法与Servlet API中的转发方法类似，不同之处在于，它使应用程序可以恢复对Servlet容器线程的请求处

更多教程请访问:www.flydean.com

理。
- ServletRequest提供对当前DispatcherType的访问，您可以使用它来区分处理初始请求，异步分派，转发和其他分派器类型。

DeferredResult处理工作如下：

- 控制器返回DeferredResult并将其保存在一些内存队列或列表中，可以在其中访问。
- Spring MVC调用request.startAsync（）。
- 同时，DispatcherServlet和所有已配置的过滤器退出请求处理线程，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。
- 再次调用DispatcherServlet，并使用异步产生的返回值恢复处理。

Callable的处理方式如下：

- 控制器返回一个Callable。
- Spring MVC调用request.startAsync（）并将Callable提交给TaskExecutor以在单独的线程中进行处理。
- 同时，DispatcherServlet和所有过滤器退出Servlet容器线程，但响应保持打开状态。
- 最终，Callable产生一个结果，Spring MVC将请求分派回Servlet容器以完成处理。
- 再次调用DispatcherServlet，并使用Callable中异步产生的返回值恢复处理。

有关更多背景知识，您还可以阅读在Spring MVC 3.2中引入了异步请求处理支持的博客文章。

Exception Handling

使用DeferredResult时，可以选择是调用带有异常的setResult还是setErrorResult。在这两种情况下，Spring MVC都将请求分派回Servlet容器以完成处理。然后将其视为控制器方法返回了给定值，或者好像它产生了给定的异常。然后，异常将通过常规的异常处理机制进行处理（例如，调用@ExceptionHandler方法）。

使用Callable时，会发生类似的处理逻辑，主要区别是从Callable返回结果或引发异常。

拦截

HandlerInterceptor实例的类型可以为AsyncHandlerInterceptor，以在启动异步处理的初始请求（而不是postHandle和afterCompletion）上接收afterConcurrentHandlingStarted回调。

HandlerInterceptor实现也可以注册CallableProcessingInterceptor或DeferredResultProcessingInterceptor，以与异步请求的生命周期进行更深入的集成（例如，处理超时事件）。有关更多详细信息，请参见AsyncHandlerInterceptor。

DeferredResult提供onTimeout（Runnable）和onCompletion（Runnable）回调。有关更多详细信息，请参见DeferredResult的javadoc。可以用Callable代替WebAsyncTask，它公开了超时和完成回调的其他方法。

与WebFlux相比

Servlet API最初是为通过Filter-Servlet链进行一次传递而构建的。 Servlet 3.0中添加了异步请求处理，使应用程序可以退出Filter-Servlet链，但保留响应以进行进一步处理。 Spring MVC异步支持围绕该机制构建。当控制器返回DeferredResult时，退出Filter-Servlet链，并释放Servlet容器线程。稍后，在设置DeferredResult时，将进行ASYNC调度（到相同的URL），在此期间，控制器将再次映射，但不是调用它，而是使用DeferredResult值（就像控制器返回了它）来恢复处理。 。

相比之下，Spring WebFlux既不是基于Servlet API构建的，也不需要这种异步请求处理功能，因为它在设计上是异步的。异步处理已内置在所有框架协定中，并在请求处理的所有阶段得到内在支持。

从编程模型的角度来看，Spring MVC和Spring WebFlux都支持异步和响应类型作为控制器方法中的返回值。 Spring MVC甚至支持流式传输，包括反应性背压。但是，与WebFlux不同，WebFlux依赖于非阻塞I / O，并且每次写入都不需要额外的线程，因此对响应的单个写入仍然处于阻塞状态（并在单独的线程上执行）。

另一个基本区别是，Spring MVC在控制器方法参数中不支持异步或响应类型（例如，@ RequestBody，@ RequestPart等），也没有对异步和响应类型作为模型属性的任何显式支持。 Spring WebFlux确实支持所有这些。

更多教程请访问:www.flydean.com

1.5.4 HTTP Streaming

您可以将DeferredResult和Callable用于单个异步返回值。 如果要产生多个异步值并将那些值写入响应，该怎么办？ 本节介绍如何执行此操作。

对象

您可以使用ResponseBodyEmitter返回值生成对象流，其中每个对象都使用HttpMessageConverter序列化并写入响应，如以下示例所示：

```
@GetMapping("/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();
```

您还可以将ResponseBodyEmitter用作ResponseEntity的主体，以自定义响应的状态和标头。

当发射器抛出IOException时（例如，如果远程客户端离开了），应用程序将不负责清理连接，并且不应调用Emitter.complete或Emitter.completeWithError。 取而代之的是，该Servlet容器自动启动AsyncListener错误通知，其中Spring MVC在其中进行completeWithError调用。 依次，此调用对应用程序执行一个最终的ASYNC调度，在此期间，Spring MVC调用已配置的异常解析器并完成请求。

SSE

SseEmitter（ResponseBodyEmitter的子类）提供对服务器发送事件的支持，其中从服务器发送的事件根据W3C SSE规范进行格式化。 要从控制器生成SSE流，请返回SseEmitter，如以下示例所示：

```
@GetMapping(path="/events", produces=MediaType.TEXT_EVENT_STREAM_VALUE)
public SseEmitter handle() {
    SseEmitter emitter = new SseEmitter();
    // 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();
```

虽然SSE是流式传输到浏览器的主要选项，但请注意Internet Explorer不支持服务器发送事件。 考虑将Spring的WebSocket消息与针对广泛浏览器的SockJS后备传输（包括SSE）结合使用。

另请参阅上一节以获取有关异常处理的注解。

Raw Data

更多教程请访问:www.flydean.com

有时，绕过消息转换并直接流到响应OutputStream很有用（例如，用于文件下载）。 您可以使用StreamingResponseBody返回值类型来执行此操作，如以下示例所示：

```
@GetMapping("/download")
public StreamingResponseBody handle() {
    return new StreamingResponseBody() {
        @Override
        public void writeTo(OutputStream outputStream) throws IOException {
            // write...
        }
    };
}
```

您可以将StreamingResponseBody用作ResponseEntity中的主体，以自定义响应的状态和标头。

1.5.5 Reactive Types

Spring MVC支持在控制器中使用反应式客户端库（另请参阅WebFlux部分中的反应式库）。 这包括来自spring-webflux的WebClient和其他资源，例如Spring Data反应数据存储库。 在这种情况下，能够从控制器方法返回反应类型是很方便的。

反应性返回值的处理方式如下：

- 与使用DeferredResult相似，单值承诺也适用于此。 示例包括Mono（Reactor）或Single（RxJava）。
- 与使用ResponseBodyEmitter或SseEmitter相似，适用于具有流媒体类型（例如application / stream + json或text / event-stream）的多值流。 示例包括Flux（反应堆）或Observable（RxJava）。 应用程序还可以返回Flux <ServerSentEvent>或Observable <ServerSentEvent>。
- 类似于使用DeferredResult <List <? >>，适用于任何其他媒体类型（例如application / json）的多值流。

Spring MVC通过spring-core的ReactiveAdapterRegistry支持Reactor和RxJava，这使其可以适应多个反应式库。

为了流式传输到响应，支持反应性背压，但响应的写仍处于阻塞状态，并通过配置的TaskExecutor在单独的线程上执行，以避免阻塞上游源（例如，从WebClient返回的Flux）。 默认情况下，SimpleAsyncTaskExecutor用于阻止写操作，但是在负载下不适合。 如果计划使用响应类型进行流传输，则应使用MVC配置来配置任务执行程序。

1.5.6 Disconnects

当远程客户端离开时，Servlet API不提供任何通知。 因此，在通过SseEmitter或响应类型流式传输到响应时，定期发送数据很重要，因为如果客户端断开连接，写入将失败。 发送可以采取空（仅评论）SSE事件或另一端必须将其解释为心跳和忽略的任何其他数据的形式。

或者，考虑使用具有内置心跳机制的Web消息传递解决方案（例如，基于WebSocket的STOMP或具有SockJS的WebSocket）。

1.5.7 Configuration

必须在Servlet容器级别启用异步请求处理功能。 MVC配置还为异步请求提供了多个选项。

Servlet容器

过滤器和Servlet声明具有asyncSupported标志，需要将其设置为true才能启用异步请求处理。另外，应声明过滤器映射以处理ASYNC javax.servlet.DispatchType。

在Java配置中，当您使用AbstractAnnotationConfigDispatcherServletInitializer初始化Servlet容器时，这是自动完成的。

更多教程请访问:www.flydean.com

在web.xml配置中，可以将<async-supported> true </ async-supported>添加到DispatcherServlet和Filter声明中，并添加<dispatcher> ASYNC </ dispatcher>来过滤映射。

Spring MVC

MVC配置公开了以下与异步请求处理相关的选项：

- Java配置：在WebMvcConfigurer上使用configureAsyncSupport回调。
- XML名称空间：使用<mvc：annotation-driven>下的<async-support>元素。

您可以配置以下内容：

- 异步请求的默认超时值（如果未设置）取决于底层的Servlet容器（例如，在Tomcat上为10秒）。
- AsyncTaskExecutor用于在使用响应类型进行流式传输时阻止写入，并用于执行从控制器方法返回的Callable实例。如果您使用反应类型进行流式传输或具有返回Callable的控制器方法，我们强烈建议配置此属性，因为默认情况下，它是SimpleAsyncTaskExecutor。
- DeferredResultProcessingInterceptor实现和CallableProcessingInterceptor实现。

请注意，您还可以在DeferredResult，ResponseBodyEmitter和SseEmitter上设置默认超时值。对于Callable，可以使用WebAsyncTask提供超时值。

更多教程请访问:www.flydean.com

1.6 CORS

Spring MVC使您可以处理CORS（跨域资源共享）。本节介绍如何执行此操作。

1.6.1 介绍

出于安全原因，浏览器禁止AJAX调用当前来源以外的资源。例如，您可以将您的银行帐户放在一个标签中，将evil.com放在另一个标签中。来自evil.com的脚本不能使用您的凭据向您的银行API发出AJAX请求，例如从您的帐户中提取资金！

跨域资源共享（CORS）是大多数浏览器实施的W3C规范，可让您指定授权哪种类型的跨域请求，而不是使用基于IFRAME或JSONP的安全性较低且功能较弱的变通办法。

1.6.2 Processing

CORS规范区分飞行前，简单和实际请求。要了解CORS的工作原理，您可以阅读本文以及其他内容，或者参阅规范以获取更多详细信息。

Spring MVC HandlerMapping实现为CORS提供内置支持。成功将请求映射到处理程序后，HandlerMapping实现将检查给定请求和处理程序的CORS配置，并采取进一步的措施。飞行前请求直接处理，而简单和实际的CORS请求被拦截，验证并设置了必需的CORS响应标头。

为了启用跨域请求（即存在Origin标头，并且与请求的主机不同），您需要具有一些显式声明的CORS配置。如果找不到匹配的CORS配置，则预检请求将被拒绝。没有将CORS标头添加到简单和实际CORS请求的响应中，因此，浏览器拒绝了它们。

可以使用基于URL模式的CorsConfiguration映射分别配置每个HandlerMapping。在大多数情况下，应用程序使用MVC Java配置或XML名称空间声明此类映射，这导致将单个全局映射传递给所有HandlerMappping实例。

您可以将HandlerMapping级别的全局CORS配置与更细粒度的处理程序级别的CORS配置结合使用。例如，带注解的控制器可以使用类或方法级别的@CrossOrigin注解（其他处理程序可以实现CorsConfigurationSource）。

全局和本地配置组合的规则通常是相加的，例如，所有全局和所有本地起源。对于只能接受单个值的那些属性（例如allowCredentials和maxAge），局部属性将覆盖全局值。有关更多详细信息，请参见CorsConfiguration # combine（CorsConfiguration）。

1.6.3 @CrossOrigin

@CrossOrigin批注启用带注解的控制器方法上的跨域请求，如以下示例所示：

```
@RestController
@RequestMapping("/account")
public class AccountController {

    @CrossOrigin
    @GetMapping("/{id}")
    public Account retrieve(@PathVariable Long id) {
        // ...
    }

    @DeleteMapping("/{id}")
    public void remove(@PathVariable Long id) {
        // ...
```

更多教程请访问:www.flydean.com

```
    }
}
```

默认情况下，@ CrossOrigin允许：

- All origins.
- All headers.
- 控制器方法映射到的所有HTTP方法。

默认情况下，allowedCredentials未启用，因为它建立了一个信任级别，可以公开敏感的用户特定信息（例如cookie和CSRF令牌），并且仅在适当的地方使用。

maxAge设置为30分钟。

@CrossOrigin在类级别上也受支持，并且被所有方法继承，如以下示例所示：

```
@CrossOrigin(origins = "https://domain2.com", maxAge = 3600)
@RestController
@RequestMapping("/account")
public class AccountController {

    @GetMapping("/{id}")
    public Account retrieve(@PathVariable Long id) {
        // ...
    }

    @DeleteMapping("/{id}")
    public void remove(@PathVariable Long id) {
        // ...
    }
}
```

您可以在类级别和方法级别上使用@CrossOrigin，如以下示例所示：

```
@CrossOrigin(maxAge = 3600)
@RestController
@RequestMapping("/account")
public class AccountController {

    @CrossOrigin("https://domain2.com")
    @GetMapping("/{id}")
    public Account retrieve(@PathVariable Long id) {
        // ...
    }

    @DeleteMapping("/{id}")
    public void remove(@PathVariable Long id) {
        // ...
    }
}
```

1.6.4 Global Configuration

除了细粒度的控制器方法级别配置外，您可能还想定义一些全局CORS配置。 您可以在任何HandlerMapping上分别设置基于URL的CorsConfiguration映射。 但是，大多数应用程序使用MVC Java配置或MVC XNM命名空间来执行此操作。

默认情况下，全局配置启用以下功能：

- All origins.

更多教程请访问:www.flydean.com

- All headers.
- GET，HEAD和POST方法。

默认情况下，allowedCredentials未启用，因为它建立了一个信任级别，可以公开敏感的用户特定信息（例如cookie和CSRF令牌），并且仅在适当的地方使用。

maxAge设置为30分钟。

Java Configuration

要在MVC Java配置中启用CORS，可以使用CorsRegistry回调，如以下示例所示：

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {

        registry.addMapping("/api/**")
            .allowedOrigins("https://domain2.com")
            .allowedMethods("PUT", "DELETE")
            .allowedHeaders("header1", "header2", "header3")
            .exposedHeaders("header1", "header2")
            .allowCredentials(true).maxAge(3600);

        // Add more mappings...
    }
}
```

XML Configuration

要在XML名称空间中启用CORS，可以使用<mvc：cors>元素，如以下示例所示：

```
<mvc:cors>

    <mvc:mapping path="/api/**"
        allowed-origins="https://domain1.com, https://domain2.com"
        allowed-methods="GET, PUT"
        allowed-headers="header1, header2, header3"
        exposed-headers="header1, header2" allow-credentials="true"
        max-age="123" />

    <mvc:mapping path="/resources/**"
        allowed-origins="https://domain1.com" />

</mvc:cors>
```

1.6.5 CORS Filter

您可以通过内置的CorsFilter应用CORS支持。

> 如果您尝试将CorsFilter与Spring Security一起使用，请记住Spring Security内置了对CORS的支持。

要配置过滤器，请将CorsConfigurationSource传递给其构造函数，如以下示例所示：

```
CorsConfiguration config = new CorsConfiguration();

// Possibly...
// config.applyPermitDefaultValues()

config.setAllowCredentials(true);
```

更多教程请访问:www.flydean.com

```
config.addAllowedOrigin("https://domain1.com");
config.addAllowedHeader("*");
config.addAllowedMethod("*");

UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
source.registerCorsConfiguration("/**", config);

CorsFilter filter = new CorsFilter(source);
```

更多教程请访问:www.flydean.com

1.7 Web Security

Spring Security项目提供了保护Web应用程序免受恶意利用的支持。 请参阅Spring Security参考文档，包括：

- Spring MVC安全性
- Spring MVC测试支持
- CSRF保护
- 安全响应标题

HDIV是另一个与Spring MVC集成的Web安全框架。

更多教程请访问:www.flydean.com

1.8 HTTP Caching

HTTP缓存可以显着提高Web应用程序的性能。 HTTP缓存围绕Cache-Control响应标头以及随后的条件请求标头（例如Last-Modified和ETag）。 缓存控制为私有（例如浏览器）和公共（例如代理）缓存提供有关如何缓存和重用响应的建议。 ETag标头用于发出条件请求，如果内容未更改，则可能导致没有主体的304（NOT_MODIFIED）。 ETag可以看作是Last-Modified头的更复杂的后继者。

本节描述了Spring Web MVC中与HTTP缓存相关的选项。

1.8.1 CacheControl

CacheControl支持配置与Cache-Control标头相关的设置，并在许多地方作为参数被接受：

- WebContentInterceptor
- WebContentGenerator
- 控制器
- 静态资源

尽管RFC 7234描述了Cache-Control响应标头的所有可能的指令，但CacheControl类型采用了面向用例的方法，着重于常见方案：

```
// Cache for an hour - "Cache-Control: max-age=3600"
CacheControl ccCacheOneHour = CacheControl.maxAge(1, TimeUnit.HOURS);

// Prevent caching - "Cache-Control: no-store"
CacheControl ccNoStore = CacheControl.noStore();

// Cache for ten days in public and private caches,
// public caches should not transform the response
// "Cache-Control: max-age=864000, public, no-transform"
CacheControl ccCustom = CacheControl.maxAge(10, TimeUnit.DAYS).noTransform().cachePublic();
```

WebContentGenerator还接受一个更简单的cachePeriod属性（以秒为单位定义），该属性的工作方式如下：

- -1值不会生成Cache-Control响应标头。
- 值为0可以防止使用“ Cache-Control：无存储”指令进行缓存。
- n> 0值通过使用'Cache-Control：max-age = n'指令将给定响应缓存n秒。

1.8.2 控制器

控制器可以添加对HTTP缓存的显式支持。 我们建议您这样做，因为需要先计算资源的lastModified或ETag值，然后才能将其与条件请求标头进行比较。 控制器可以将ETag标头和Cache-Control设置添加到ResponseEntity，如以下示例所示：

```
@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
```

更多教程请访问:www.flydean.com

```
            .body(book);
}
```

如果与条件请求标头的比较表明内容未更改，则前面的示例发送带有空主体的304（NOT_MODIFIED）响应。 否则，ETag和Cache-Control标头将添加到响应中。

您还可以在控制器中针对条件请求标头进行检查，如以下示例所示：

```
@RequestMapping
public String myHandleMethod(WebRequest webRequest, Model model) {

    long eTag = ...

    if (request.checkNotModified(eTag)) {
        return null;
    }

    model.addAttribute(...);
    return "myViewName";
}
```

可以使用三种变体来检查针对eTag值，lastModified值或两者的条件请求。 对于有条件的GET和HEAD请求，可以将响应设置为304（NOT_MODIFIED）。 对于条件POST，PUT和DELETE，您可以将响应设置为409（PRECONDITION_FAILED），以防止并发修改。

1.8.3 Static Resources

您应该将静态资源与Cache-Control和条件响应标头一起提供，以实现最佳性能。 请参阅有关配置静态资源的部分。

1.8.4 ETag Filter

您可以使用ShallowEtagHeaderFilter添加从响应内容计算得出的“shallow” eTag值，从而节省带宽，但不节省CPU时间。 请参阅浅ETag。

更多教程请访问:www.flydean.com

1.9 View技术

Spring MVC中视图技术的使用是可插入的，无论您决定使用Thymeleaf，Groovy标记模板，JSP还是其他技术，主要取决于配置更改。 本章介绍与Spring MVC集成的视图技术。 我们假设您已经熟悉View Resolution。

1.9.1 Thymeleaf

Thymeleaf是一种现代的服务器端Java模板引擎，它强调可以通过双击在浏览器中预览的自然HTML模板，这对于独立处理UI模板（例如，由设计人员）而无需使用非常有用。 正在运行的服务器。 如果要替换JSP，Thymeleaf提供了最广泛的功能集之一，以使这种过渡更加容易。 Thymeleaf是积极开发和维护的。 有关更完整的介绍，请参见Thymeleaf项目主页。

Thymeleaf与Spring MVC的集成由Thymeleaf项目管理。 该配置涉及一些Bean声明，例如ServletContextTemplateResolver，SpringTemplateEngine和ThymeleafViewResolver。 有关更多详细信息，请参见Thymeleaf + Spring。

1.9.2 FreeMarker

Apache FreeMarker是一个模板引擎，用于生成从HTML到电子邮件等的任何类型的文本输出。 Spring框架具有内置的集成，可以将Spring MVC与FreeMarker模板一起使用。

View配置

以下示例显示如何将FreeMarker配置为一种视图技术：

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.freemarker();
    }

    // Configure FreeMarker...

    @Bean
    public FreeMarkerConfigurer freeMarkerConfigurer() {
        FreeMarkerConfigurer configurer = new FreeMarkerConfigurer();
        configurer.setTemplateLoaderPath("/WEB-INF/freemarker");
        return configurer;
    }
}
```

以下示例显示了如何在XML中进行配置：

```
<mvc:annotation-driven/>

<mvc:view-resolvers>
    <mvc:freemarker/>
</mvc:view-resolvers>

<!-- Configure FreeMarker... -->
<mvc:freemarker-configurer>
    <mvc:template-loader-path location="/WEB-INF/freemarker"/>
</mvc:freemarker-configurer>
```

更多教程请访问:www.flydean.com

另外，您也可以声明FreeMarkerConfigurer bean，以完全控制所有属性，如以下示例所示：

```
<bean id="freemarkerConfig" class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">
    <property name="templateLoaderPath" value="/WEB-INF/freemarker/"/>
</bean>
```

您的模板需要存储在上例所示的FreeMarkerConfigurer指定的目录中。 给定上述配置，如果您的控制器返回欢迎的视图名称，则解析器将查找/WEB-INF/freemarker/welcome.ftl模板。

FreeMarker Configuration

您可以通过在FreeMarkerConfigurer bean上设置适当的bean属性，将FreeMarker的“设置”和“ SharedVariables”直接传递给FreeMarker配置对象（由Spring管理）。 freemarkerSettings属性需要一个java.util.Properties对象，而freemarkerVariables属性需要一个java.util.Map。 以下示例显示了如何执行此操作：

```
<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"/>
```

有关设置和变量应用于Configuration对象的详细信息，请参见FreeMarker文档。

表格处理

Spring提供了一个供JSP使用的标签库，其中包含一个<spring：bind />元素。该元素主要允许表单显示来自表单支持对象的值，并显示来自Web或业务层中Validator的验证失败的结果。 Spring还支持FreeMarker中的相同功能，并带有用于生成表单输入元素本身的附加便利宏。

绑定宏

spring-webmvc.jar文件中为这两种语言维护了一组标准宏，因此它们始终可用于经过适当配置的应用程序。

Spring库中定义的某些宏被视为内部（私有）宏，但是在宏定义中不存在这种范围，使所有宏对调用代码和用户模板可见。以下各节仅关注您需要在模板中直接调用的宏。如果您希望直接查看宏代码，则该文件名为spring.ftl，位于org.springframework.web.servlet.view.freemarker包中。

简单绑定

在充当Spring MVC控制器的表单视图的HTML表单（vm或ftl模板）中，您可以使用类似于下一个示例的代码绑定到字段值，并以类似于JSP的方式显示每个输入字段的错误消息。当量。以下示例显示了先前配置的personForm视图：

```
<!-- 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>
```

更多教程请访问:www.flydean.com

```
        ...
        <input type="submit" value="submit"/>
    </form>
    ...
</html>
```

<@ spring.bind>需要一个'path'参数，该参数由命令对象的名称（除非您在FormController属性中进行了更改，否则为'command'）组成，后跟一个句点和字段的名称。您希望绑定到的命令对象。您还可以使用嵌套字段，例如command.address.street。 bind宏假定由web.xml中的ServletContext参数defaultHtmlEscape指定的默认HTML转义行为。

名为<@ spring.bindEscaped>的宏的可选形式带有第二个参数，并明确指定在状态错误消息或值中应使用HTML转义。您可以根据需要将其设置为true或false。附加的表单处理宏可简化HTML转义的使用，并且您应尽可能使用这些宏。下一节将对它们进行说明。

输入宏

两种语言的附加便利宏均简化了绑定和表单生成（包括验证错误显示）。从来没有必要使用这些宏来生成表单输入字段，并且您可以将它们与简单的HTML混合或匹配，或者直接调用我们之前强调的spring绑定宏。

下表可用的宏显示了每个FTL定义和参数列表：

macro	FTL definition
message (output a string from a resource bundle based on the code parameter)	<@spring.message code/>
messageText (output a string from a resource bundle based on the code parameter, falling back to the value of the default parameter)	<@spring.messageText code, text/>
url (prefix a relative URL with the application’s context root)	<@spring.url relativeUrl/>
formInput (standard input field for gathering user input)	<@spring.formInput path, attributes, fieldType/>
formHiddenInput (hidden input field for submitting non-user input)	<@spring.formHiddenInput path, attributes/>
formPasswordInput (standard input field for gathering passwords. Note that no value is ever populated in fields of this type.)	<@spring.formPasswordInput path, attributes/>
formTextarea (large text field for gathering long, freeform text input)	<@spring.formTextarea path, attributes/>
formSingleSelect (drop down box of options that let a single required value be selected)	<@spring.formSingleSelect path, options, attributes/>
formMultiSelect (a list box of options that let the user select 0 or more values)	<@spring.formMultiSelect path, options, attributes/>
formRadioButtons (a set of radio buttons that let a single selection be made from the available choices)	<@spring.formRadioButtons path, options separator, attributes/>
formCheckboxes (a set of checkboxes that let 0 or more values be selected)	<@spring.formCheckboxes path, options, separator, attributes/>
formCheckbox (a single checkbox)	<@spring.formCheckbox path, attributes/>
showErrors (simplify display of validation errors for the bound field)	<@spring.showErrors separator, classOrStyle/>

- 在FTL（FreeMarker）中，实际上并不需要formHiddenInput和formPasswordInput，因为您可以使用常规的formInput宏，将hidden或password指定为fieldType参数的值。

更多教程请访问:www.flydean.com

以上任何宏的参数都具有一致的含义：

- path：要绑定的字段名称（即“ command.name”）
- options：可以在输入字段中选择的所有可用值的映射。映射的键表示从表单回发并绑定到命令对象的值。针对键存储的地图对象是在表单上显示给用户的标签，并且可能与表单回发的相应值不同。通常，这种地图由控制器作为参考数据提供。您可以使用任何Map实现，具体取决于所需的行为。对于严格排序的地图，可以将SortedMap（例如TreeMap）与合适的Comparator一起使用；对于应按插入顺序返回值的任意地图，请使用Commons-collection中的LinkedHashMap或LinkedMap。
- 分隔符：当多个选项可用作离散元素（单选按钮或复选框）时，用于分隔列表中的每个字符的字符序列（例如
）。
- 属性：HTML标记本身包含的任意标记或文本的附加字符串。该字符串实际上是由宏回显的。例如，在textarea字段中，您可以提供属性（例如'rows =“ 5” cols =“ 60”'），也可以传递样式信息，例如'style =“ border：1px solid silver”。
- classOrStyle：对于showErrors宏，包装每个错误的span元素使用的CSS类的名称。如果未提供任何信息（或该值为空），则错误将包装在<b> </ b>标记中。

以下各节概述了宏的示例（某些在FTL中，有些在VTL中）。两种语言之间存在用法差异的地方，在注解中进行了说明。

输入栏位 formInput宏带有path参数（command.name）和一个附加的attribute参数（在后面的示例中为空）。该宏与所有其他表单生成宏一起，对path参数执行隐式Spring绑定。绑定保持有效，直到发生新的绑定为止，因此showErrors宏无需再次传递path参数。它在最后创建绑定的字段上进行操作。

showErrors宏采用分隔符参数（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-这次是类名或样式属性。注意，FreeMarker可以为attributes参数指定默认值。下面的示例演示如何使用formInput和showWErrors宏：

```
<@spring.formInput "command.name"/>
<@spring.showErrors "<br>"/>
```

下一个示例显示表单片段的输出，生成名称字段，并在提交表单后在字段中没有值的情况下显示验证错误。 验证通过Spring的Validation框架进行。

生成的HTML类似于以下示例：

```
Name:
<input type="text" name="name" value="">
<br>
    <b>required</b>
<br>
<br>
```

formTextarea宏的工作方式与formInput宏相同，并且接受相同的参数列表。 通常，第二个参数（属性）用于传递样式信息或文本区域的行和列属性。

选择字段

您可以使用四个选择字段宏在HTML表单中生成常见的UI值选择输入：

- formSingleSelect
- formMultiSelect
- formRadioButtons
- formCheckboxes

四个宏中的每个宏都接受一个选项映射，其中包含表单字段的值和与该值相对应的标签。 值和标签可以相同。

下一个示例是FTL中的单选按钮。 表单支持对象为此字段指定默认值“London”，因此无需验证。 呈现表单时，将在模型中以“ cityMap”为名称提供可供选择的整个城市列表作为参考数据。 以下清单显示了示例：

更多教程请访问:www.flydean.com

```
...
Town:
<@spring.formRadioButtons "command.address.town", cityMap, ""/><br><br>
```

上面的清单呈现了一行单选按钮，每个单选按钮用于cityMap中的每个值，并使用分隔符“”。 没有提供其他属性（缺少该宏的最后一个参数）。 cityMap对地图中的每个键值对使用相同的String。 地图的键是表单实际作为POSTed请求参数提交的键。 映射值是用户看到的标签。 在前面的示例中，给定三个知名城市的列表以及表单支持对象中的默认值，HTML类似于以下内容：

```
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>
```

如果您的应用程序希望通过内部代码处理城市（例如），则可以使用合适的键创建代码地图，如以下示例所示：

```
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，该代码将生成输出，其中无线电值是相关代码，但是用户仍然可以看到更加用户友好的城市名称，如下所示：

```
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 Escaping

前面介绍的表单宏的默认用法将产生符合HTML 4.01的HTML元素，并使用web xml支持中定义的HTML转义默认值（由Spring的绑定支持使用）。 要使元素符合XHTML或覆盖默认的HTML转义值，您可以在模板（或模型中对模板可见的两个变量）中指定两个变量。 在模板中指定它们的优点是，可以在稍后的模板处理中将它们更改为不同的值，以为表单中的不同字段提供不同的行为。

要为您的标记切换到XHTML兼容性，请为名为xhtmlCompliant的模型或上下文变量指定true值，如以下示例所示：

```
<#-- for FreeMarker -->
<#assign xhtmlCompliant = true>
```

处理完该指令后，Spring宏生成的任何元素现在都符合XHTML。

以类似的方式，您可以指定每个字段的HTML转义，如以下示例所示：

```
<#-- until this point, default HTML escaping is used -->

<#assign htmlEscape = true>
<#-- 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 -->
```

更多教程请访问:www.flydean.com

1.9.3 Groovy Markup

Groovy标记模板引擎主要旨在生成类似XML的标记（XML，XHTML，HTML5等），但是您可以使用它来生成任何基于文本的内容。 Spring框架具有内置的集成，可以将Spring MVC与Groovy标记一起使用。

> Groovy标记模板引擎需要Groovy 2.3.1+。

Configuration

以下示例显示了如何配置Groovy标记模板引擎：

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.groovy();
    }

    // Configure the Groovy Markup Template Engine...

    @Bean
    public GroovyMarkupConfigurer groovyMarkupConfigurer() {
        GroovyMarkupConfigurer configurer = new GroovyMarkupConfigurer();
        configurer.setResourceLoaderPath("/WEB-INF/");
        return configurer;
    }
}
```

以下示例显示了如何在XML中进行配置：

```
<mvc:annotation-driven/>

<mvc:view-resolvers>
    <mvc:groovy/>
</mvc:view-resolvers>

<!-- Configure the Groovy Markup Template Engine... -->
<mvc:groovy-configurer resource-loader-path="/WEB-INF/"/>
```

Example

与传统的模板引擎不同，Groovy标记依赖于使用构建器语法的DSL。 以下示例显示了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')
    }
}
```

1.9.4 Script Views

Spring框架具有内置的集成，可以将Spring MVC与可以在JSR-223 Java脚本引擎之上运行的任何模板库一起使用。 我们已经在不同的脚本引擎上测试了以下模板库：

更多教程请访问:www.flydean.com

Scripting Library	Scripting Engine
Handlebars	Nashorn
Mustache	Nashorn
React	Nashorn
EJS	Nashorn
ERB	JRuby
String templates	Jython
Kotlin Script templating	Kotlin

> 集成任何其他脚本引擎的基本规则是，它必须实现ScriptEngine和Invocable接口。

Requirements

您需要在类路径上具有脚本引擎，其细节因脚本引擎而异：

- Java 8+随附了Nashorn JavaScript引擎。 强烈建议使用可用的最新更新版本。
- 应该将JRuby添加为对Ruby支持的依赖。
- 应该将Jython添加为对Python支持的依赖项。
- 应该添加org.jetbrains.kotlin：kotlin-script-util依赖项和包含org.jetbrains.kotlin.script.jsr223.KotlinJsr223JvmLocalScriptEngineFactory行的META-INF / services / javax.script.ScriptEngineFactory文件。 有关更多详细信息，请参见此示例。

您需要具有脚本模板库。 针对Javascript的一种方法是通过WebJars。

Script Templates

您可以声明一个ScriptTemplateConfigurer bean来指定要使用的脚本引擎，要加载的脚本文件，调用呈现模板的函数等等。 以下示例使用Mustache模板和Nashorn JavaScript引擎：

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @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;
    }
}
```

以下示例显示了XML中的相同排列：

```
<mvc:annotation-driven/>

<mvc:view-resolvers>
    <mvc:script-template/>
</mvc:view-resolvers>
```

更多教程请访问:www.flydean.com

```
<mvc:script-template-configurer engine-name="nashorn" render-object="Mustache" render-function="render">
    <mvc:script location="mustache.js"/>
</mvc:script-template-configurer>
```

对于Java和XML配置，该控制器看起来没有什么不同，如以下示例所示：

```
@Controller
public class SampleController {

    @GetMapping("/sample")
    public String test(Model model) {
        model.addObject("title", "Sample title");
        model.addObject("body", "Sample body");
        return "template";
    }
}
```

以下示例显示了Mustache模板：

```
<html>
    <head>
        <title>{{title}}</title>
    </head>
    <body>
        <p>{{body}}</p>
    </body>
</html>
```

使用以下参数调用render函数：

- String template：模板内容
- Map model：视图模型
- RenderingContext renderingContext：RenderingContext，用于访问应用程序上下文，语言环境，模板加载器和URL（自5.0起）

Mustache.render（）与该签名本地兼容，因此您可以直接调用它。

如果您的模板技术需要一些自定义，则可以提供一个实现自定义渲染功能的脚本。 例如，Handlerbars需要在使用模板之前先对其进行编译，并且需要使用polyfill来模拟某些服务器端脚本引擎中不可用的浏览器功能。

以下示例显示了如何执行此操作：

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @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;
    }
}
```

更多教程请访问:www.flydean.com

当您将非线程安全脚本引擎与不是为并发设计的模板库一起使用时，例如，在Nashorn上运行的Handlebars或React，必须将sharedEngine属性设置为false。 在这种情况下，由于此错误，需要Java 8u60或更高版本。

polyfill.js仅定义Handlebars正常运行所需的window对象，如下所示：

```
var window = {};
```

这个基本的render.js实现在使用模板之前先对其进行编译。 生产就绪的实现还应该存储任何重用的缓存模板或预编译的模板。 您可以在脚本方面进行操作（并处理所需的任何自定义，例如管理模板引擎配置）。 以下示例显示了如何执行此操作：

```
function render(template, model) {
    var compiledTemplate = Handlebars.compile(template);
    return compiledTemplate(model);
}
```

1.9.5 JSP and JSTL

Spring框架具有内置的集成，可以将Spring MVC与JSP和JSTL一起使用。

View Resolvers

使用JSP开发时，可以声明一个InternalResourceViewResolver或ResourceBundleViewResolver bean。

ResourceBundleViewResolver依靠属性文件来定义映射到类和URL的视图名称。 使用ResourceBundleViewResolver，您可以仅使用一个解析器来混合不同类型的视图，如以下示例所示：

```
<!-- the ResourceBundleViewResolver -->
<bean id="viewResolver" class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
    <property name="basename" value="views"/>
</bean>

# And a sample properties file is used (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
```

InternalResourceViewResolver也可以用于JSP。 作为最佳实践，我们强烈建议您将JSP文件放在“ WEB-INF”目录下的目录中，以便客户端无法直接访问。

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <property name="suffix" value=".jsp"/>
</bean>
```

JSP与JSTL

使用JSP标准标记库（JSTL）时，必须使用特殊的视图类JstlView，因为JSTL需要一些准备工作，然后I18N功能才能正常工作。

Spring的JSP标签库

如前几章所述，Spring提供了请求参数到命令对象的数据绑定。为了促进结合这些数据绑定功能的JSP页面的开发，Spring提供了一些使事情变得更加容易的标记。所有Spring标记都具有HTML转义功能，以启用或禁用字符转义。

更多教程请访问:www.flydean.com

spring-webmvc.jar中包含spring.tld标记库描述符（TLD）。有关单个标签的全面参考，请浏览API参考或查看标签库说明。

Spring的表单标签库

从2.0版开始，Spring使用JSP和Spring Web MVC时，提供了一组全面的数据绑定感知标签，用于处理表单元素。每个标签都支持其对应的HTML标签对应项的属性集，从而使标签变得熟悉且易于使用。标记生成的HTML符合HTML 4.01 / XHTML 1.0。

与其他表单/输入标签库不同，Spring的表单标签库与Spring Web MVC集成在一起，使标签可以访问命令对象和控制器处理的参考数据。正如我们在以下示例中所示，表单标签使JSP易于开发，读取和维护。

我们浏览一下表单标签，并查看一个如何使用每个标签的示例。我们包含了生成的HTML代码段，其中某些标记需要进一步的注解。

Configuration

表单标签库捆绑在spring-webmvc.jar中。库描述符称为spring-form.tld。

要使用此库中的标记，请在JSP页面顶部添加以下指令：

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

表单标签

该标签呈现HTML'form'元素，并向内部标签公开绑定路径以进行绑定。 它将命令对象放在PageContext中，以便内部标签可以访问该命令对象。 该库中的所有其他标签都是form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firstName和lastName之类的属性的JavaBean。 我们可以将其用作表单控制器的表单支持对象，该表单控制器返回form.jsp。 以下示例显示了form.jsp的外观：

```
<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>
```

页面控制器从放置在PageContext中的命令对象中检索firstName和lastName值。 继续阅读以了解如何将内部标签与表单标签一起使用的更复杂的示例。

下面的清单显示了生成的HTML，它看起来像标准格式：

```
<form method="POST">
    <table>
        <tr>
            <td>First Name:</td>
            <td><input name="firstName" type="text" value="Harry"/></td>
        </tr>
        <tr>
            <td>Last Name:</td>
```

更多教程请访问:www.flydean.com

```
            <td><input name="lastName" type="text" value="Potter"/></td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="Save Changes"/>
            </td>
        </tr>
    </table>
</form>
```

前面的JSP假定表单支持对象的变量名是command。 如果已将表单支持对象以另一个名称（肯定是最佳实践）放入模型中，则可以将表单绑定到命名变量，如以下示例所示：

```
<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>
```

input标签

默认情况下，此标记呈现具有绑定值和type ='text'的HTML输入元素。 有关此标签的示例，请参见表单标签。 您还可以使用HTML5特定类型，例如电子邮件，电话，日期和其他。

checkbox标签

此标记呈现类型设置为复选框的HTML输入标记。

假设我们的用户有喜好，例如通讯订阅和兴趣爱好列表。 以下示例显示了Preferences类：

```
public class Preferences {

    private boolean receiveNewsletter;
    private String[] interests;
    private String favouriteWord;

    public boolean isReceiveNewsletter() {
        return receiveNewsletter;
    }

    public void setReceiveNewsletter(boolean receiveNewsletter) {
        this.receiveNewsletter = receiveNewsletter;
    }

    public String[] getInterests() {
        return interests;
    }

    public void setInterests(String[] interests) {
        this.interests = interests;
    }

    public String getFavouriteWord() {
```

更多教程请访问:www.flydean.com

```
        return favouriteWord;
    }

    public void setFavouriteWord(String favouriteWord) {
        this.favouriteWord = favouriteWord;
    }
}
```

相应的form.jsp可能类似于以下内容：

```
<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 Again
st 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>
```

复选框标记有三种方法，它们可以满足您所有的复选框需求。

- 方法一：当绑定值为java.lang.Boolean类型时，如果绑定值为true，则将input（checkbox）标记为已选中。 value属性对应于setValue（Object）value属性的解析值。
- 方法二：当绑定值的类型为array或java.util.Collection时，如果绑定的Collection中存在已配置的setValue（Object）值，则将input（checkbox）标记为已选中。
- 方法三：对于任何其他绑定值类型，如果已配置的setValue（Object）等于绑定值，则将input（复选框）标记为已选中。

请注意，无论采用哪种方法，都会生成相同的HTML结构。 以下HTML代码段定义了一些复选框：

```
<tr>
    <td>Interests:</td>
    <td>
        Quidditch: <input name="preferences.interests" type="checkbox" value="Quidditch"/>
        <input type="hidden" value="1" name="_preferences.interests"/>
        Herbology: <input name="preferences.interests" type="checkbox" value="Herbology"/>
        <input type="hidden" value="1" name="_preferences.interests"/>
        Defence Against the Dark Arts: <input name="preferences.interests" type="checkbox" value="Defence Again
st the Dark Arts"/>
        <input type="hidden" value="1" name="_preferences.interests"/>
    </td>
</tr>
```

更多教程请访问:www.flydean.com

您可能不希望在每个复选框之后看到其他隐藏字段。如果未选中HTML页面中的复选框，则提交表单后，其值就不会作为HTTP请求参数的一部分发送到服务器，因此我们需要一种解决方法来使HTML中的这个问题生效，以使Spring表单数据绑定生效。 checkbox标记遵循现有的Spring约定，其中包括每个复选框均带有下划线（_）前缀的隐藏参数。通过这样做，您可以有效地告诉Spring“复选框在表单中可见，并且我希望表单数据绑定到我的对象以反映复选框的状态，无论如何。”

checkboxes标签

该标签呈现类型设置为复选框的多个HTML输入标签。

本节以前面的复选框标签节中的示例为基础。有时，您希望不必在JSP页面中列出所有可能的爱好。您宁愿在运行时提供可用选项的列表，然后将其传递给标记。这就是复选框标签的目的。您可以传入包含items属性中可用选项的数组，列表或映射。通常，bound属性是一个集合，因此它可以保存用户选择的多个值。以下示例显示了使用此标记的JSP：

```
<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>
```

本示例假定interestList是一个可用作模型属性的列表，其中包含要从中选择的值的字符串。 如果您使用的是地图，则将地图输入键用作值，并将地图输入的值用作要显示的标签。 您还可以使用自定义对象，在其中您可以通过使用itemValue提供值的属性名称，并通过使用itemLabel提供标签。

radiobutton标签

此标记呈现类型设置为radio的HTML input元素。

典型的用法模式涉及绑定到相同属性但值不同的多个标记实例，如以下示例所示：

```
<tr>
    <td>Sex:</td>
    <td>
        Male: <form:radiobutton path="sex" value="M"/> <br/>
        Female: <form:radiobutton path="sex" value="F"/>
    </td>
</tr>
```

radiobuttons tag

此标记呈现类型设置为radio的多个HTML输入元素。

与checkboxes标记一样，您可能希望将可用选项作为运行时变量传递。 对于此用法，您可以使用单选按钮标签。 您传入包含items属性中可用选项的数组，列表或映射。 如果您使用地图，则将地图输入键用作值，并将地图输入的值用作要显示的标签。 您还可以使用一个自定义对象，在其中您可以通过使用itemValue提供值的属性名称，并通过使用itemLabel提供标签，如以下示例所示：

```
<tr>
    <td>Sex:</td>
    <td><form:radiobuttons path="sex" items="${sexOptions}"/></td>
</tr>
```

password tag

更多教程请访问:www.flydean.com

该标签呈现HTML输入标签，其类型设置为具有绑定值的password。

```
<tr>
    <td>Password:</td>
    <td>
        <form:password path="password"/>
    </td>
</tr>
```

请注意，默认情况下，不显示密码值。 如果确实希望显示密码值，则可以将showPassword属性的值设置为true，如以下示例所示：

```
<tr>
    <td>Password:</td>
    <td>
        <form:password path="password" value="^76525bvHGq" showPassword="true"/>
    </td>
</tr>
```

select标签

此标记呈现HTML“ select”元素。 它支持将数据绑定到所选选项以及使用嵌套选项和选项标签。

假定用户具有技能列表。 相应的HTML可能如下所示：

```
<tr>
    <td>Skills:</td>
    <td><form:select path="skills" items="${skills}"/></td>
</tr>
```

如果用户的技能是Herbology，则“技能”行的HTML源可能如下：

```
<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>
```

option tag

此标记呈现HTML选项元素。 根据边界值设置为选中状态。 以下HTML显示了其典型输出：

```
<tr>
    <td>House:</td>
    <td>
        <form:select path="house">
            <form:option value="Gryffindor"/>
            <form:option value="Hufflepuff"/>
            <form:option value="Ravenclaw"/>
            <form:option value="Slytherin"/>
        </form:select>
    </td>
</tr>
```

如果用户的房屋位于Gryffindor，则“House”行的HTML源代码如下：

更多教程请访问:www.flydean.com

```
<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>
```

options tag

此标记呈现HTML选项元素的列表。 它基于绑定值设置选定的属性。 以下HTML显示了其典型输出：

```
<tr>
    <td>Country:</td>
    <td>
        <form:select path="country">
            <form:option value="-" label="--Please Select"/>
            <form:options items="${countryList}" itemValue="code" itemLabel="name"/>
        </form:select>
    </td>
</tr>
```

如果用户居住在英国，则“国家/地区”行的HTML来源如下：

```
<tr>
    <td>Country:</td>
    <td>
        <select name="country">
            <option value="-">--Please Select</option>
            <option value="AT">Austria</option>
            <option value="UK" selected="selected">United Kingdom</option>
            <option value="US">United States</option>
        </select>
    </td>
</tr>
```

如前面的示例所示，选项标签与选项标签的组合使用会生成相同的标准HTML，但可以让您在JSP中显式指定一个仅用于显示（它所属的位置）的值，例如 例如：“-请选择”。

items属性通常填充有item对象的集合或数组。 如果指定了itemValue和itemLabel，则引用这些item对象的bean属性。 否则，项目对象本身将变成字符串。 或者，您可以指定一个项目映射，在这种情况下，映射键将解释为选项值，并且映射值对应于选项标签。 如果同时也指定了itemValue或itemLabel（或两者），则item value属性应用于映射键，item label属性应用于映射值。

textarea标签

此标记呈现HTML textarea元素。 以下HTML显示了其典型输出：

```
<tr>
    <td>Notes:</td>
    <td><form:textarea path="notes" rows="3" cols="20"/></td>
    <td><form:errors path="notes"/></td>
</tr>
```

hidden tag

更多教程请访问:www.flydean.com

该标签呈现一个HTML输入标签，其类型设置为使用绑定值隐藏。 要提交未绑定的隐藏值，请使用HTML输入标签，其类型设置为hidden。 以下HTML显示了其典型输出：

```
<form:hidden path="house"/>
```

如果我们选择将house value作为隐藏的value提交，则HTML如下所示：

```
<input name="house" type="hidden" value="Gryffindor"/>
```

error tag

此标记在HTML span元素中呈现字段错误。 它提供对在控制器中创建的错误或由与控制器关联的任何验证程序创建的错误的访问。

假设一旦提交表单，我们想显示firstName和lastName字段的所有错误消息。 我们有一个名为UserValidator的User类实例的验证器，如以下示例所示：

```
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.");
    }
}
```

form.jsp如下所示：

```
<form:form>
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName"/></td>
            <%-- Show errors for firstName field --%>
            <td><form:errors path="firstName"/></td>
        </tr>

        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName"/></td>
            <%-- Show errors for lastName field --%>
            <td><form:errors path="lastName"/></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes"/>
            </td>
        </tr>
    </table>
</form:form>
```

如果我们在firstName和lastName字段中提交具有空值的表单，则HTML将如下所示：

```
<form method="POST">
    <table>
        <tr>
            <td>First Name:</td>
            <td><input name="firstName" type="text" value=""/></td>
```

更多教程请访问:www.flydean.com

```
            <%-- Associated errors to firstName field displayed --%>
            <td><span name="firstName.errors">Field is required.</span></td>
        </tr>

        <tr>
            <td>Last Name:</td>
            <td><input name="lastName" type="text" value=""/></td>
            <%-- Associated errors to lastName field displayed --%>
            <td><span name="lastName.errors">Field is required.</span></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes"/>
            </td>
        </tr>
    </table>
</form>
```

如果我们要显示给定页面的整个错误列表怎么办？ 下一个示例显示errors标签还支持一些基本的通配符功能。

- path =“ *”：显示所有错误。
- path =“ lastName”：显示与lastName字段相关的所有错误。
- 如果省略路径，则仅显示对象错误。

以下示例在页面顶部显示错误列表，然后在字段旁边显示特定于字段的错误：

```
<form:form>
    <form:errors path="*" cssClass="errorBox"/>
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName"/></td>
            <td><form:errors path="firstName"/></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName"/></td>
            <td><form:errors path="lastName"/></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes"/>
            </td>
        </tr>
    </table>
</form:form>
```

HTML 如下所示：

```
<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">
```

更多教程请访问:www.flydean.com

```
                <input type="submit" value="Save Changes"/>
            </td>
        </tr>
    </table>
</form>
```

spring-webmvc.jar中包含spring-form.tld标记库描述符（TLD）。有关单个标签的全面参考，请浏览API参考或查看标签库说明。

HTTP方法转换

REST的一个关键原则是使用“统一接口”。这意味着可以使用相同的四种HTTP方法（GET，PUT，POST和DELETE）来操纵所有资源（URL）。对于每种方法，HTTP规范都定义了确切的语义。例如，GET应该始终是安全的操作，这意味着它没有副作用，而PUT或DELETE应该是幂等的，这意味着您可以一遍又一遍地重复这些操作，但是最终结果应该相同。虽然HTTP定义了这四种方法，但HTML仅支持两种：GET和POST。幸运的是，有两种可能的解决方法：您可以使用JavaScript进行PUT或DELETE，或者可以使用“ real”方法作为附加参数（在HTML表单中建模为隐藏的输入字段）进行POST。 Spring的HiddenHttpMethodFilter使用了后一种技巧。该过滤器是一个普通的Servlet过滤器，因此，它可以与任何Web框架（不仅仅是Spring MVC）结合使用。将此过滤器添加到web.xml，然后将带有隐藏方法参数的POST转换为相应的HTTP方法请求。

为了支持HTTP方法转换，Spring MVC表单标签已更新为支持设置HTTP方法。例如，以下代码片段来自“宠物诊所”样本：

```
<form:form method="delete">
    <p class="submit"><input type="submit" value="Delete Pet"/></p>
</form:form>
```

前面的示例执行HTTP POST，并将“真实” DELETE方法隐藏在请求参数后面。 它由在web.xml中定义的HiddenHttpMethodFilter拾取，如以下示例所示：

```
<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>
```

以下示例显示了相应的@Controller方法：

```
@RequestMapping(method = RequestMethod.DELETE)
public String deletePet(@PathVariable int ownerId, @PathVariable int petId) {
    this.clinic.deletePet(petId);
    return "redirect:/owners/" + ownerId;
}
```

HTML5标签

Spring表单标签库允许输入动态属性，这意味着您可以输入任何HTML5特定的属性。

表单输入标签支持输入文本以外的类型属性。 这旨在允许呈现新的HTML5特定输入类型，例如电子邮件，日期，范围等。 请注意，由于默认类型为text，因此不需要输入type ='text'。

1.9.6 Tiles

更多教程请访问:www.flydean.com

您可以像使用其他视图技术一样，将Tiles集成到使用Spring的Web应用程序中。 本节以广泛的方式描述如何执行此操作。

> 本节重点介绍org.springframework.web.servlet.view.tiles3软件包中Spring对Tiles版本3的支持。

依存关系

为了能够使用Tiles，您必须在Tiles 3.0.1或更高版本上添加一个依赖项，并将其传递依赖项添加到您的项目中。

Configuration

为了能够使用Tiles，必须使用包含定义的文件对其进行配置（有关定义和其他Tiles概念的基本信息，请参见 https://tiles.apache.org）。 在春季，这是通过使用TilesConfigurer完成的。 以下示例ApplicationContext配置说明了如何执行此操作：

```
<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>
```

前面的示例定义了五个包含定义的文件。 这些文件都位于WEB-INF / defs目录中。 在WebApplicationContext初始化时，将加载文件，并初始化定义工厂。 完成之后，定义文件中包含的Tiles可以用作Spring Web应用程序中的视图。 为了能够使用视图，您必须拥有一个ViewResolver以及与Spring一起使用的任何其他视图技术。 您可以使用UrlBasedViewResolver和ResourceBundleViewResolver这两个实现中的任何一个。

您可以通过添加下划线然后添加语言环境来指定特定于语言环境的Tiles定义，如以下示例所示：

```
<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>
```

通过上述配置，tile_fr_FR.xml用于具有fr_FR语言环境的请求，默认情况下使用tile.xml。

> 由于下划线用于指示语言环境，因此建议不要在图块定义的文件名中使用下划线。

UrlBasedViewResolver

UrlBasedViewResolver为其要解析的每个视图实例化给定的viewClass。 以下bean定义了一个UrlBasedViewResolver：

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.UrlBasedViewResolver">
    <property name="viewClass" value="org.springframework.web.servlet.view.tiles3.TilesView"/>
</bean>
```

ResourceBundleViewResolver

必须为ResourceBundleViewResolver提供一个属性文件，该文件包含解析程序可以使用的视图名称和视图类。 以下示例显示了ResourceBundleViewResolver的bean定义以及相应的视图名称和视图类（摘自Pet Clinic示例）：

更多教程请访问:www.flydean.com

```
<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
...
```

使用ResourceBundleViewResolver时，可以轻松地混合使用不同的视图技术。

请注意，TilesView类支持JSTL（JSP标准标记库）。

SimpleSpringPreparerFactory和SpringBeanPreparerFactory 作为一项高级功能，Spring还支持两个特殊的Tiles PreparerFactory实现。有关如何在Tiles定义文件中使用ViewPreparer引用的详细信息，请参见Tiles文档。

您可以指定SimpleSpringPreparerFactory以根据指定的准备器类自动连接ViewPreparer实例，应用Spring的容器回调以及应用配置的Spring BeanPostProcessor。如果已激活Spring的上下文范围的注解配置，则将自动检测和应用ViewPreparer类中的注解。请注意，这与默认的PreparerFactory一样，期望Tiles定义文件中的preparer类。

您可以指定SpringBeanPreparerFactory以对指定的准备程序名称（而不是类）进行操作，从而从DispatcherServlet的应用程序上下文中获取相应的Spring bean。在这种情况下，完整的bean创建过程由Spring应用程序上下文控制，从而允许使用显式依赖项注入配置，作用域bean等。注意，您需要为每个准备器名称定义一个Spring bean定义（在Tiles定义中使用）。下面的示例演示如何在TilesConfigurer bean上定义SpringBeanPreparerFactory属性：

```
<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>

    <!-- resolving preparer names as Spring bean definition names -->
    <property name="preparerFactoryClass"
            value="org.springframework.web.servlet.view.tiles3.SpringBeanPreparerFactory"/>

</bean>
```

1.9.7 RSS and Atom

AbstractAtomFeedView和AbstractRssFeedView都继承自AbstractFeedView基类，分别用于提供Atom和RSS Feed视图。 它们基于java.net的ROME项目，位于org.springframework.web.servlet.view.feed包中。

AbstractAtomFeedView要求您实现buildFeedEntries（）方法，并可选地覆盖buildFeedMetadata（）方法（默认实现为空）。 以下示例显示了如何执行此操作：

```
public class SampleContentAtomView extends AbstractAtomFeedView {

    @Override
```

更多教程请访问:www.flydean.com

```
    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
    }

}
```

相似的要求适用于实现AbstractRssFeedView，如以下示例所示：

```
public class SampleContentRssView 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
    }
}
```

如果需要访问Locale，则buildFeedItems（）和buildFeedEntries（）方法会传递HTTP请求。 仅针对Cookie或其他HTTP标头的设置传入HTTP响应。 方法返回后，提要将自动写入响应对象。

有关创建Atom视图的示例，请参见Alef Arendsen的Spring Team Blog条目。

1.9.8 PDF and Excel

Spring提供了返回HTML以外的输出的方法，包括PDF和Excel电子表格。本节介绍如何使用这些功能。

Document view简介

HTML页面并非始终是用户查看模型输出的最佳方法，而Spring使从模型数据动态生成PDF文档或Excel电子表格变得简单。该文档是视图，并从服务器以正确的内容类型进行流传输，以（希望）使客户端PC能够运行其电子表格或PDF查看器应用程序作为响应。

为了使用Excel视图，您需要将Apache POI库添加到您的类路径中。为了生成PDF，您需要添加（最好是）OpenPDF库。

> 如果可能，您应该使用基础文档生成库的最新版本。特别是，我们强烈建议您使用OpenPDF（例如，OpenPDF 1.2.12）而不是过时的原始iText 2.1.7，因为OpenPDF会得到积极维护并修复了不可信任PDF内容的重要漏洞。

PDF views

单词列表的简单PDF视图可以扩展org.springframework.web.servlet.view.document.AbstractPdfView并实现buildPdfDocument（）方法，如以下示例所示：

```
public class PdfWordList extends AbstractPdfView {

    protected void buildPdfDocument(Map<String, Object> model, Document doc, PdfWriter writer,
            HttpServletRequest request, HttpServletResponse response) throws Exception {
```

更多教程请访问:www.flydean.com

```
        List<String> words = (List<String>) model.get("wordList");
        for (String word : words) {
            doc.add(new Paragraph(word));
        }
    }
}
```

控制器可以从外部视图定义（按名称引用）返回此视图，也可以从处理程序方法返回为视图实例。

Excel view

从Spring Framework 4.2开始，提供org.springframework.web.servlet.view.document.AbstractXlsView作为Excel视图的基类。 它基于Apache POI，具有取代过时的AbstractExcelView类的专用子类（AbstractXlsxView和AbstractXlsxStreamingView）。

编程模型类似于AbstractPdfView，其中buildExcelDocument（）作为中央模板方法，控制器能够从外部定义（按名称）或从处理程序方法作为View实例返回这种视图。

1.9.9 Jackson

Spring提供了对Jackson JSON库的支持。

基于Jackson的JSON MVC视图

MappingJackson2JsonView使用Jackson library的ObjectMapper将响应内容呈现为JSON。默认情况下，模型映射的全部内容（特定于框架的类除外）均编码为JSON。对于需要过滤地图内容的情况，可以使用modelKeys属性指定一组特定的模型属性进行编码。您也可以使用extractValueFromSingleKeyModel属性，以将单键模型中的值直接提取并序列化，而不是作为模型属性的映射。

您可以根据需要使用Jackson提供的注解来自定义JSON映射。当需要进一步控制时，可以在需要为特定类型提供自定义JSON序列化器和反序列化器的情况下，通过ObjectMapper属性注入自定义ObjectMapper。

基于Jackson的XML视图

MappingJackson2XmlView使用Jackson XML扩展程序的XmlMapper将响应内容呈现为XML。如果模型包含多个条目，则应使用modelKey bean属性显式设置要序列化的对象。如果模型包含单个条目，则会自动序列化。

您可以根据需要使用JAXB或Jackson提供的注解自定义XML映射。当需要进一步控制时，可以通过ObjectMapper属性注入自定义XmlMapper，对于自定义XML，需要为特定类型提供序列化器和反序列化器的情况。

1.9.10 XML Marshalling

MarshallingView使用XML Marshaller（在org.springframework.oxm包中定义）将响应内容呈现为XML。 您可以使用MarshallingView实例的modelKey bean属性显式设置要编组的对象。 或者，视图遍历所有模型属性，并封送Marshaller支持的第一个类型。 有关org.springframework.oxm包中功能的更多信息，请参见使用O / X映射器编组XML。

1.9.11 XSLT Views

XSLT是XML的一种转换语言，在Web应用程序中作为一种视图技术而流行。 如果您的应用程序自然处理XML，或者您的模型可以轻松转换为XML，则XSLT可以作为视图技术的不错选择。 下一节显示了如何将XML文档生成为模型数据，以及如何在Spring Web MVC应用程序中使用XSLT对其进行转换。

这个例子是一个普通的Spring应用程序，它在Controller中创建一个单词列表并将它们添加到模型图中。 返回该地图以及XSLT视图的视图名称。 有关Spring Web MVC的Controller界面的详细信息，请参见带注解的控制器。 XSLT控制器将单词列表转换为准备转换的简单XML文档。

更多教程请访问:www.flydean.com

Beans

Configuration是简单Spring Web应用程序的标准配置：MVC配置必须定义XsltViewResolver bean和常规MVC注解配置。 以下示例显示了如何执行此操作：

```
@EnableWebMvc
@ComponentScan
@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Bean
    public XsltViewResolver xsltViewResolver() {
        XsltViewResolver viewResolver = new XsltViewResolver();
        viewResolver.setPrefix("/WEB-INF/xsl/");
        viewResolver.setSuffix(".xslt");
        return viewResolver;
    }
}
```

Controller

我们还需要一个封装了词生成逻辑的控制器。

控制器逻辑封装在@Controller类中，其处理程序方法定义如下：

```
@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";
    }
}
```

到目前为止，我们仅创建了一个DOM文档并将其添加到Model映射中。 请注意，您还可以将XML文件作为资源加载并使用它代替自定义DOM文档。

有可用的软件包自动“对象化”对象图，但是在Spring中，您可以完全灵活地以任何选择的方式从模型中创建DOM。 这样可以防止XML转换在模型数据的结构中扮演过多的角色，这在使用工具来管理DOMification流程时是一种危险。

Transformation

最后，XsltViewResolver解析"home” XSLT模板文件，并将DOM文档合并到其中以生成我们的视图。 如XsltViewResolver配置所示，XSLT模板位于WEB-INF / xsl目录的war文件中，并以xslt文件扩展名结尾。

以下示例显示了XSLT转换：

```
<?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"/>
```

更多教程请访问:www.flydean.com

```
    <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>
```

前面的转换呈现为以下HTML：

```
<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>
```

更多教程请访问:www.flydean.com

1.10 MVC配置

MVC Java配置和MVC XML名称空间提供适用于大多数应用程序的默认配置以及用于自定义它的配置API。

有关配置API中不可用的更多高级定制，请参阅Advanced Java Config和Advanced XML Config。

您不需要了解由MVC Java配置和MVC名称空间创建的基础bean。 如果要了解更多信息，请参阅特殊Bean类型和Web MVC配置。

1.10.1 开启MVC Configuration

在Java配置中，可以使用@EnableWebMvc批注启用MVC配置，如以下示例所示：

```
@Configuration
@EnableWebMvc
public class WebConfig {
}
```

在XML配置中，可以使用<mvc：annotation-driven>元素来启用MVC配置，如以下示例所示：

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

    <mvc:annotation-driven/>

</beans>
```

前面的示例注册了许多Spring MVC基础结构Bean，并适应了类路径上可用的依赖项（例如，JSON，XML等的有效负载转换器）。

1.10.2 MVC Config API

在Java配置中，可以实现WebMvcConfigurer接口，如以下示例所示：

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    // Implement configuration methods...
}
```

在XML中，您可以检查<mvc：annotation-driven />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。

1.10.3 Type Conversion

更多教程请访问:www.flydean.com

默认情况下，将安装Number和Date类型的格式化程序，包括对@NumberFormat和@DateTimeFormat批注的支持。 如果类路径中存在Joda-Time，则还将安装对Joda-Time格式库的完全支持。

在Java配置中，您可以注册自定义格式器和转换器，如以下示例所示：

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addFormatters(FormatterRegistry registry) {
        // ...
    }
}
```

以下示例显示了如何在XML中实现相同的配置：

```
<?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
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/mvc
        https://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <mvc:annotation-driven conversion-service="conversionService"/>

    <bean id="conversionService"
            class="org.springframework.format.support.FormattingConversionServiceFactoryBean">
        <property name="converters">
            <set>
                <bean class="org.example.MyConverter"/>
            </set>
        </property>
        <property name="formatters">
            <set>
                <bean class="org.example.MyFormatter"/>
                <bean class="org.example.MyAnnotationFormatterFactory"/>
            </set>
        </property>
        <property name="formatterRegistrars">
            <set>
                <bean class="org.example.MyFormatterRegistrar"/>
            </set>
        </property>
    </bean>

</beans>
```

> 有关何时使用FormatterRegistrar实现的更多信息，请参见FormatterRegistrar SPI和FormattingConversionServiceFactoryBean。

1.10.4 Validation

默认情况下，如果Bean验证存在于类路径中（例如，Hibernate Validator），则LocalValidatorFactoryBean将注册为全局验证器，以与@Valid一起使用，并在控制器方法参数上进行Validated。

在Java配置中，您可以自定义全局Validator实例，如以下示例所示：

```
@Configuration
```

更多教程请访问:www.flydean.com

```
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public Validator getValidator(); {
        // ...
    }
}
```

以下示例显示了如何在XML中实现相同的配置：

```
<?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
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/mvc
        https://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <mvc:annotation-driven validator="globalValidator"/>

</beans>
```

请注意，您还可以在本地注册Validator实现，如以下示例所示：

```
@Controller
public class MyController {

    @InitBinder
    protected void initBinder(WebDataBinder binder) {
        binder.addValidators(new FooValidator());
    }

}
```

> 如果您需要在某个地方注入LocalValidatorFactoryBean，请创建一个bean并用@Primary标记它，以避免与MVC配置中声明的那个冲突。

1.10.5 Interceptors

在Java配置中，您可以注册拦截器以应用于传入的请求，如以下示例所示：

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new LocaleChangeInterceptor());
        registry.addInterceptor(new ThemeChangeInterceptor()).addPathPatterns("/**").excludePathPatterns("/admi
n/**");
        registry.addInterceptor(new SecurityInterceptor()).addPathPatterns("/secure/*");
    }
}
```

以下示例显示了如何在XML中实现相同的配置：

```
<mvc:interceptors>
```

更多教程请访问:www.flydean.com

```
    <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>
```

1.10.6 Content Types

您可以配置Spring MVC如何根据请求确定请求的媒体类型（例如，Accept标头，URL路径扩展，查询参数等）。

默认情况下，首先检查URL路径扩展名-将json，xml，rss和atom注册为已知扩展名（取决于类路径依赖项）。 接受标头是第二次检查。

考虑将这些默认值更改为“仅接受”标头，并且，如果必须使用基于URL的内容类型解析，请考虑对路径扩展使用查询参数策略。 有关更多详细信息，请参见后缀匹配和后缀匹配以及RFD。

在Java配置中，您可以自定义请求的内容类型解析，如以下示例所示：

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void configureContentNegotiation(ContentNegotiationConfigurer configurer) {
        configurer.mediaType("json", MediaType.APPLICATION_JSON);
        configurer.mediaType("xml", MediaType.APPLICATION_XML);
    }
}
```

以下示例显示了如何在XML中实现相同的配置：

```
<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>
```

1.10.7 Message Converters

您可以通过覆盖configureMessageConverters（）（以替换Spring MVC创建的默认转换器）或覆盖extendMessageConverters（）（以自定义默认转换器或向默认转换器添加其他转换器），以Java配置自定义HttpMessageConverter。

以下示例使用自定义的ObjectMapper代替默认的添加了XML和Jackson JSON转换器：

```
@Configuration
@EnableWebMvc
```

更多教程请访问:www.flydean.com

```
public class WebConfiguration implements WebMvcConfigurer {

    @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.createXmlMapper(true).build()));
    }
}
```

在前面的示例中，使用Jackson2ObjectMapperBuilder为启用了缩进的MappingJackson2HttpMessageConverter和MappingJackson2XmlHttpMessageConverter创建通用配置，自定义日期格式以及jackson-module-parameter-names的注册，从而增加了对访问参数名称的支持（增加了一项功能）在Java 8中）。

该构建器自定义Jackson的默认属性，如下所示：

- DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES被禁用。
- MapperFeature.DEFAULT_VIEW_INCLUSION已禁用。

如果在类路径中检测到以下知名模块，它还将自动注册以下知名模块：

- jackson-datatype-jdk7：支持Java 7类型，例如java.nio.file.Path。
- jackson-datatype-joda：支持Joda-Time类型。
- jackson-datatype-jsr310：支持Java 8日期和时间API类型。
- jackson-datatype-jdk8：支持其他Java 8类型，例如Optional。

> 除了jackson-dataformat-xml之外，使用Jackson XML支持启用缩进还需要woodstox-core-asl依赖性。

其他有趣的Jackson模块也可用：

- jackson-datatype-money：支持javax.money类型（非官方模块）。
- jackson-datatype-hibernate：支持特定于Hibernate的类型和属性（包括延迟加载方面）。

以下示例显示了如何在XML中实现相同的配置：

```
<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"/>
```

1.10.8 View Controllers

这是定义ParameterizableViewController的快捷方式，该参数可在调用时立即转发到视图。 在视图生成响应之前没有Java控制器逻辑要执行的静态情况下，可以使用它。

更多教程请访问:www.flydean.com

以下Java配置示例将对/的请求转发到名为home的视图：

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/").setViewName("home");
    }
}
```

以下示例通过使用<mvc：view-controller>元素，实现了与上一示例相同的操作，但使用XML：

```
<mvc:view-controller path="/" view-name="home"/>
```

1.10.9 View Resolvers

MVC配置简化了视图解析器的注册。

以下Java配置示例通过使用JSP和Jackson作为JSON呈现的默认视图来配置内容协商视图解析：

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.enableContentNegotiation(new MappingJackson2JsonView());
        registry.jsp();
    }
}
```

以下示例显示了如何在XML中实现相同的配置：

```
<mvc:view-resolvers>
    <mvc:content-negotiation>
        <mvc:default-views>
            <bean class="org.springframework.web.servlet.view.json.MappingJackson2JsonView"/>
        </mvc:default-views>
    </mvc:content-negotiation>
    <mvc:jsp/>
</mvc:view-resolvers>
```

但是请注意，FreeMarker，Tiles，Groovy标记和脚本模板也需要配置基础视图技术。

MVC命名空间提供了专用元素。 以下示例适用于FreeMarker：

```
<mvc:view-resolvers>
    <mvc:content-negotiation>
        <mvc:default-views>
            <bean class="org.springframework.web.servlet.view.json.MappingJackson2JsonView"/>
        </mvc:default-views>
    </mvc:content-negotiation>
    <mvc:freemarker cache="false"/>
</mvc:view-resolvers>

<mvc:freemarker-configurer>
    <mvc:template-loader-path location="/freemarker"/>
</mvc:freemarker-configurer>
```

更多教程请访问:www.flydean.com

在Java配置中，您可以添加相应的Configurer bean，如以下示例所示：

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.enableContentNegotiation(new MappingJackson2JsonView());
        registry.freeMarker().cache(false);
    }

    @Bean
    public FreeMarkerConfigurer freeMarkerConfigurer() {
        FreeMarkerConfigurer configurer = new FreeMarkerConfigurer();
        configurer.setTemplateLoaderPath("/freemarker");
        return configurer;
    }
}
```

1.10.10 Static Resources

此选项提供了一种方便的方法来从基于资源的位置列表中提供静态资源。

在下一个示例中，给定一个以/ resources开头的请求，相对路径用于在Web应用程序根目录下或/ static下的类路径下相对于/ public查找和提供静态资源。 这些资源的使用期限为一年，以确保最大程度地利用浏览器缓存并减少浏览器发出的HTTP请求。 还评估Last-Modified头，如果存在，则返回304状态代码。

以下清单显示了如何使用Java配置进行操作：

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**")
            .addResourceLocations("/public", "classpath:/static/")
            .setCachePeriod(31556926);
    }
}
```

以下示例显示了如何在XML中实现相同的配置：

```
<mvc:resources mapping="/resources/**"
    location="/public, classpath:/static/"
    cache-period="31556926" />
```

资源处理程序还支持ResourceResolver实现和ResourceTransformer实现的链，您可以使用它们创建用于处理优化资源的工具链。

您可以将VersionResourceResolver用于基于资源，固定应用程序版本或其他内容计算出的MD5哈希的版本化资源URL。 ContentVersionStrategy（MD5哈希）是一个不错的选择-带有一些值得注意的例外，例如与模块加载器一起使用的JavaScript资源。

以下示例显示如何在Java配置中使用VersionResourceResolver：

```
@Configuration
```

更多教程请访问:www.flydean.com

```
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**")
                .addResourceLocations("/public/")
                .resourceChain(true)
                .addResolver(new VersionResourceResolver().addContentVersionStrategy("/**"));
    }
}
```

以下示例显示了如何在XML中实现相同的配置：

```
<mvc:resources mapping="/resources/**" location="/public/">
    <mvc:resource-chain resource-cache="true">
        <mvc:resolvers>
            <mvc:version-resolver>
                <mvc:content-version-strategy patterns="/**"/>
            </mvc:version-resolver>
        </mvc:resolvers>
    </mvc:resource-chain>
</mvc:resources>
```

然后，您可以使用ResourceUrlProvider重写URL，并应用完整的解析器和转换器链-例如插入版本。 MVC配置提供了ResourceUrlProvider bean，以便可以将其注入其他对象。您也可以使用ResourceUrlEncodingFilter对Thymeleaf，JSP，FreeMarker以及其他依赖于HttpServletResponse # encodeURL的URL标签的重写透明化。

请注意，在同时使用EncodedResourceResolver（例如，用于服务压缩或brotli编码的资源）和VersionResourceResolver时，必须按此顺序注册。这样可以确保始终基于未编码文件可靠地计算基于内容的版本。

WebJars还通过WebJarsResourceResolver支持，当org.webjars：webjars-locator-core库存在于类路径中时，WebJars将自动注册。解析程序可以重写URL以包括jar的版本，并且还可以与没有版本的传入URL进行匹配，例如，从/jquery/jquery.min.js到/jquery/1.2.0/jquery.min.js。

1.10.11 Default Servlet

Spring MVC允许将DispatcherServlet映射到/（从而覆盖了容器默认Servlet的映射），同时仍允许容器默认Servlet处理静态资源请求。 它使用/ **的URL映射和相对于其他URL映射的最低优先级配置DefaultServletHttpRequestHandler。

该处理程序将所有请求转发到默认Servlet。 因此，它必须按所有其他URL HandlerMappings的顺序保留在最后。 如果使用<mvc：annotation-driven>，就是这种情况。 另外，如果您设置自己的自定义HandlerMapping实例，请确保将其order属性设置为小于DefaultServletHttpRequestHandler的Integer.MAX_VALUE的值。

下面的示例说明如何使用默认设置启用功能：

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {
        configurer.enable();
    }
}
```

以下示例显示了如何在XML中实现相同的配置：

```
<mvc:default-servlet-handler/>
```

更多教程请访问:www.flydean.com

覆盖/ Servlet映射的警告是，必须通过名称而不是通过路径来检索默认Servlet的RequestDispatcher。 DefaultServletHttpRequestHandler尝试使用大多数主要Servlet容器（包括Tomcat，Jetty，GlassFish，JBoss，Resin，WebLogic和WebSphere）的已知名称列表，在启动时自动检测容器的默认Servlet。 如果默认Servlet是使用其他名称自定义配置的，或者在默认Servlet名称未知的情况下使用了不同的Servlet容器，那么您必须明确提供默认Servlet的名称，如以下示例所示：

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {
        configurer.enable("myCustomDefaultServlet");
    }

}
```

以下示例显示了如何在XML中实现相同的配置：

```
<mvc:default-servlet-handler default-servlet-name="myCustomDefaultServlet"/>
```

1.10.12 Path Matching

您可以自定义与URL的路径匹配和处理有关的选项。 有关各个选项的详细信息，请参见PathMatchConfigurer javadoc。

以下示例显示了如何在Java配置中自定义路径匹配：

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void configurePathMatch(PathMatchConfigurer configurer) {
        configurer
            .setUseSuffixPatternMatch(true)
            .setUseTrailingSlashMatch(false)
            .setUseRegisteredSuffixPatternMatch(true)
            .setPathMatcher(antPathMatcher())
            .setUrlPathHelper(urlPathHelper())
            .addPathPrefix("/api",
                    HandlerTypePredicate.forAnnotation(RestController.class));
    }

    @Bean
    public UrlPathHelper urlPathHelper() {
        //...
    }

    @Bean
    public PathMatcher antPathMatcher() {
        //...
    }

}
```

以下示例显示了如何在XML中实现相同的配置：

更多教程请访问:www.flydean.com

```
<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"/>
```

1.10.13 Advanced Java Config

@EnableWebMvc导入DelegatingWebMvcConfiguration，其中：

- 为Spring MVC应用程序提供默认的Spring配置
- 检测并委托给WebMvcConfigurer实现以自定义该配置。

对于高级模式，可以删除@EnableWebMvc并直接从DelegatingWebMvcConfiguration扩展而不是实现WebMvcConfigurer，如以下示例所示：

```
@Configuration
public class WebConfig extends DelegatingWebMvcConfiguration {

    // ...

}
```

您可以将现有方法保留在WebConfig中，但是现在您还可以覆盖基类中的bean声明，并且在类路径上仍然可以具有任意数量的其他WebMvcConfigurer实现。

1.10.14 Advanced XML Config

MVC命名空间没有高级模式。 如果您需要在bean上自定义一个不能更改的属性，则可以使用Spring ApplicationContext的BeanPostProcessor生命周期挂钩，如以下示例所示：

```
@Component
public class MyPostProcessor implements BeanPostProcessor {

    public Object postProcessBeforeInitialization(Object bean, String name) throws BeansException {
        // ...
    }
}
```

请注意，您需要将MyPostProcessor声明为bean，可以用XML显式声明，也可以通过<component-scan />声明将其检测出来。

更多教程请访问:www.flydean.com

1.11 HTTP/2

Servlet 4容器需要支持HTTP / 2，并且Spring Framework 5与Servlet API 4兼容。从编程模型的角度来看，应用程序不需要做任何特定的事情。 但是，有一些与服务器配置有关的注意事项。 有关更多详细信息，请参见HTTP / 2 Wiki页面。

Servlet API确实公开了一种与HTTP / 2相关的构造。 您可以使用javax.servlet.http.PushBuilder主动将资源推送到客户端，并且它作为@RequestMapping方法的方法参数而受支持。

更多教程请访问:www.flydean.com

2. REST客户端

本节描述了客户端对REST端点的访问选项。

2.1 RestTemplate

RestTemplate是执行HTTP请求的同步客户端。它是原始的Spring REST客户端，并且在基础HTTP客户端库上公开了简单的模板方法API。

> 从5.0开始，无阻塞，反应式WebClient提供了RestTemplate的现代替代方案，并有效支持同步和异步以及流方案。 RestTemplate将在将来的版本中弃用，并且以后将不会添加主要的新功能。

有关详细信息，请参见REST端点。

2.2 WebClient

WebClient是执行HTTP请求的非阻塞，反应式客户端。它是在5.0中引入的，它提供了RestTemplate的现代替代方案，并有效支持同步和异步以及流方案。

与RestTemplate相比，WebClient支持以下内容：

- 非阻塞I / O。
- 反应性产生背压。
- 高并发，硬件资源更少。
- 利用Java 8 lambda的功能风格，流畅的API。
- 同步和异步交互。
- 从服务器流向上或从服务器向下流。

有关更多详细信息，请参见WebClient。

更多教程请访问:www.flydean.com

3. 测试

本节总结了Spring MVC应用程序在spring-test中可用的选项。

- Servlet API模拟：用于单元测试控制器，过滤器和其他Web组件的Servlet API合约的模拟实现。有关更多详细信息，请参见Servlet API模拟对象。
- TestContext Framework：支持在JUnit和TestNG测试中加载Spring配置，包括跨测试方法高效地缓存已加载的配置，并支持通过MockServletContext加载WebApplicationContext。有关更多详细信息，请参见TestContext Framework。
- Spring MVC测试：一种框架，也称为MockMvc，用于通过DispatcherServlet（即支持注解）测试带注解的控制器，该框架具有Spring MVC基础结构，但没有HTTP服务器。有关更多详细信息，请参见Spring MVC Test。
- 客户端REST：spring-test提供了一个MockRestServiceServer，您可以将其用作模拟服务器，以测试内部使用RestTemplate的客户端代码。有关更多详细信息，请参见客户端REST测试。
- WebTestClient：构建用于测试WebFlux应用程序，但是它也可以用于通过HTTP连接到任何服务器的端到端集成测试。它是一个无阻塞的反应式客户端，非常适合测试异步和流传输方案。

更多教程请访问:www.flydean.com

4. WebSockets

参考文档的此部分涵盖对Servlet堆栈的支持，包括原始WebSocket交互的WebSocket消息传递，通过SockJS进行WebSocket仿真以及通过STOMP作为WebSocket的子协议进行发布-订阅消息传递。

更多教程请访问:www.flydean.com

4.1 WebSocket介绍

WebSocket协议RFC 6455提供了一种标准化方法，可通过单个TCP连接在客户端和服务器之间建立全双工双向通信通道。 它是与HTTP不同的TCP协议，但旨在通过端口80和443在HTTP上工作，并允许重复使用现有的防火墙规则。

WebSocket交互始于一个HTTP请求，该请求使用HTTP Upgrade标头进行升级，或在这种情况下切换到WebSocket协议。 以下示例显示了这种交互：

```
GET /spring-websocket-portfolio/portfolio HTTP/1.1
Host: localhost:8080
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: Uc9l9TMkWGbHFD2qnFHltg==
Sec-WebSocket-Protocol: v10.stomp, v11.stomp
Sec-WebSocket-Version: 13
Origin: http://localhost:8080
```

具有WebSocket支持的服务器代替通常的200状态代码，返回的输出类似于以下内容：

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: 1qVdfYHU9hPOl4JYYNXF623Gzn0=
Sec-WebSocket-Protocol: v10.stomp
```

成功握手后，HTTP升级请求的基础TCP套接字将保持打开状态，客户端和服务器均可继续发送和接收消息。

WebSockets的工作原理的完整介绍超出了本文档的范围。 请参阅RFC 6455，HTML5的WebSocket章节或Web上的许多简介和教程中的任何一个。

请注意，如果WebSocket服务器在Web服务器（例如nginx）后面运行，则可能需要对其进行配置，以将WebSocket升级请求传递到WebSocket服务器。 同样，如果应用程序在云环境中运行，请检查与WebSocket支持相关的云提供商的说明。

4.1.1 HTTP 和 WebSocket

尽管WebSocket被设计为与HTTP兼容并以HTTP请求开头，但重要的是要了解这两个协议导致了截然不同的体系结构和应用程序编程模型。

在HTTP和REST中，应用程序被建模为许多URL。为了与应用程序交互，客户端访问那些URL，即请求-响应样式。服务器根据HTTP URL，方法和标头将请求路由到适当的处理程序。

相反，在WebSockets中，通常只有一个URL用于初始连接。随后，所有应用程序消息在同一TCP连接上流动。这指向了一个完全不同的异步，事件驱动的消息传递体系结构。

WebSocket也是一种低级传输协议，与HTTP不同，它不对消息的内容规定任何语义。这意味着除非客户端和服务器就消息语义达成一致，否则就无法路由或处理消息。

WebSocket客户端和服务器可以通过HTTP握手请求上的Sec-WebSocket-Protocol标头，协商使用更高级别的消息传递协议（例如，STOMP）。在这种情况下，他们需要提出自己的约定。

4.1.2 什么时候使用WebSockets

更多教程请访问:www.flydean.com

WebSockets可以使网页具有动态性和交互性。但是，在许多情况下，结合使用Ajax和HTTP流或长轮询可以提供一种简单有效的解决方案。

例如，新闻，邮件和社交订阅源需要动态更新，但是每几分钟进行一次更新可能是完全可以的。另一方面，协作，游戏和金融应用程序需要更接近实时。

仅延迟并不是决定因素。如果消息量相对较少（例如，监视网络故障），则HTTP流或轮询可以提供有效的解决方案。低延迟，高频率和高音量的结合才是使用WebSocket的最佳案例。

还请记住，在Internet上，控件之外的限制性代理可能会阻止WebSocket交互，这可能是因为未将它们配置为传递Upgrade标头，或者是因为它们关闭了长期处于空闲状态的连接。这意味着与面向公众的应用程序相比，将WebSocket用于防火墙内部的应用程序是一个更直接的决定。

更多教程请访问:www.flydean.com

4.2 WebSocket API

Spring框架提供了一个WebSocket API，可用于编写处理WebSocket消息的客户端和服务器端应用程序。

4.2.1 WebSocketHandler

创建WebSocket服务器就像实现WebSocketHandler一样简单，或者更可能地，扩展TextWebSocketHandler或BinaryWebSocketHandler。 以下示例使用TextWebSocketHandler：

```
import org.springframework.web.socket.WebSocketHandler;
import org.springframework.web.socket.WebSocketSession;
import org.springframework.web.socket.TextMessage;

public class MyHandler extends TextWebSocketHandler {

    @Override
    public void handleTextMessage(WebSocketSession session, TextMessage message) {
        // ...
    }

}
```

有专用的WebSocket Java配置和XML名称空间支持，用于将前面的WebSocket处理程序映射到特定的URL，如以下示例所示：

```
import org.springframework.web.socket.config.annotation.EnableWebSocket;
import org.springframework.web.socket.config.annotation.WebSocketConfigurer;
import org.springframework.web.socket.config.annotation.WebSocketHandlerRegistry;

@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(myHandler(), "/myHandler");
    }

    @Bean
    public WebSocketHandler myHandler() {
        return new MyHandler();
    }

}
```

下面的示例显示与前面的示例等效的XML配置：

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

    <websocket:handlers>
        <websocket:mapping path="/myHandler" handler="myHandler"/>
    </websocket:handlers>
```

更多教程请访问:www.flydean.com

```
    <bean id="myHandler" class="org.springframework.samples.MyHandler"/>

</beans>
```

前面的示例用于Spring MVC应用程序，应包含在DispatcherServlet的配置中。 但是，Spring的WebSocket支持不依赖于Spring MVC。 在WebSocketHttpRequestHandler的帮助下将WebSocketHandler集成到其他HTTP服务环境中相对简单。

当直接或间接使用WebSocketHandler API时，例如 通过STOMP消息传递，由于基础标准WebSocket会话（JSR-356）不允许并发发送，因此应用程序必须同步消息的发送。 一种选择是用ConcurrentWebSocketSessionDecorator包装WebSocketSession。

4.2.2 WebSocket Handshake

定制初始HTTP WebSocket握手请求的最简单方法是通过HandshakeInterceptor，它公开了“握手之前”和“握手之后”的方法。 您可以使用此类拦截器来阻止握手或使任何属性对WebSocketSession可用。 以下示例使用内置的拦截器将HTTP会话属性传递到WebSocket会话：

```
@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(new MyHandler(), "/myHandler")
            .addInterceptors(new HttpSessionHandshakeInterceptor());
    }

}
```

下面的示例显示与前面的示例等效的XML配置：

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

    <websocket:handlers>
        <websocket:mapping path="/myHandler" handler="myHandler"/>
        <websocket:handshake-interceptors>
            <bean class="org.springframework.web.socket.server.support.HttpSessionHandshakeInterceptor"/>
        </websocket:handshake-interceptors>
    </websocket:handlers>

    <bean id="myHandler" class="org.springframework.samples.MyHandler"/>

</beans>
```

一个更高级的选项是扩展DefaultHandshakeHandler，它执行WebSocket握手的步骤，包括验证客户端来源，协商子协议以及其他详细信息。 如果应用程序需要配置自定义RequestUpgradeStrategy以便适应尚不支持的WebSocket服务器引擎和版本，则它可能还需要使用此选项（有关此主题的更多信息，请参阅部署）。 Java配置和XML名称空间都使配置自定义HandshakeHandler成为可能。

更多教程请访问:www.flydean.com

> Spring提供了一个WebSocketHandlerDecorator基类，您可以使用该基类来装饰WebSocketHandler并具有其他行为。 使用WebSocket Java配置或XML名称空间时，默认情况下会提供并添加日志记录和异常处理实现。 ExceptionWebSocketHandlerDecorator捕获由任何WebSocketHandler方法引起的所有未捕获的异常，并关闭状态为1011（指示服务器错误）的WebSocket会话。

4.2.3 Deployment

易于将Spring WebSocket API集成到Spring MVC应用程序中，在该应用程序中，DispatcherServlet同时提供HTTP WebSocket握手和其他HTTP请求。通过调用WebSocketHttpRequestHandler，也很容易将其集成到其他HTTP处理方案中。这是方便且易于理解的。但是，对于JSR-356运行时，需要特别注意。

Java WebSocket API（JSR-356）提供了两种部署机制。第一个涉及启动时的Servlet容器类路径扫描（Servlet 3功能）。另一个是在Servlet容器初始化时使用的注册API。这两种机制都无法对所有HTTP处理（包括WebSocket握手和所有其他HTTP请求）（例如Spring MVC的DispatcherServlet）使用单个“前端控制器”。

这是JSR-356的一个重要限制，即使在JSR-356运行时中运行，Spring的WebSocket支持也可以通过服务器特定的RequestUpgradeStrategy实现解决。 Tomcat，Jetty，GlassFish，WebLogic，WebSphere和Undertow（和WildFly）目前存在此类策略。

已经创建了克服Java WebSocket API中的上述限制的请求，可以在eclipse-ee4j / websocket-api # 211中进行跟踪。 Tomcat，Undertow和WebSphere提供了自己的API替代方案，使之可以做到这一点，而Jetty也可以实现。 我们希望更多的服务器可以做到这一点。

第二个考虑因素是，期望支持JSR-356的Servlet容器执行ServletContainerInitializer（SCI）扫描，这可能会减慢应用程序的启动速度，在某些情况下会大大降低速度。 如果在升级到支持JSR-356的Servlet容器版本后观察到重大影响，则应该可以通过使用web.xml中的<absolute-ordering />元素有选择地启用或禁用Web片段（和SCI扫描），如以下示例所示：

```
<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
        https://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    version="3.0">

    <absolute-ordering/>

</web-app>
```

然后，您可以按名称选择性地启用Web片段，例如Spring自己的SpringServletContainerInitializer，它提供对Servlet 3 Java初始化API的支持。 以下示例显示了如何执行此操作：

```
<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
        https://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    version="3.0">

    <absolute-ordering>
        <name>spring_web</name>
    </absolute-ordering>

</web-app>
```

4.2.4 Server Configuration

更多教程请访问:www.flydean.com

每个基础的WebSocket引擎都公开控制运行时特征的配置属性，例如消息缓冲区大小的大小，空闲超时等。

对于Tomcat，WildFly和GlassFish，可以将ServletServerContainerFactoryBean添加到WebSocket Java配置中，如以下示例所示：

```
@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @Bean
    public ServletServerContainerFactoryBean createWebSocketContainer() {
        ServletServerContainerFactoryBean container = new ServletServerContainerFactoryBean();
        container.setMaxTextMessageBufferSize(8192);
        container.setMaxBinaryMessageBufferSize(8192);
        return container;
    }

}
```

下面的示例显示与前面的示例等效的XML配置：

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

    <bean class="org.springframework...ServletServerContainerFactoryBean">
        <property name="maxTextMessageBufferSize" value="8192"/>
        <property name="maxBinaryMessageBufferSize" value="8192"/>
    </bean>

</beans>
```

> 对于客户端WebSocket配置，应使用WebSocketContainerFactoryBean（XML）或ContainerProvider.getWebSocketContainer（）（Java配置）。

对于Jetty，您需要提供一个预先配置的Jetty WebSocketServerFactory，并将其通过WebSocket Java配置插入Spring的DefaultHandshakeHandler中。 以下示例显示了如何执行此操作：

```
@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(echoWebSocketHandler(),
            "/echo").setHandshakeHandler(handshakeHandler());
    }

    @Bean
    public DefaultHandshakeHandler handshakeHandler() {

        WebSocketPolicy policy = new WebSocketPolicy(WebSocketBehavior.SERVER);
        policy.setInputBufferSize(8192);
        policy.setIdleTimeout(600000);

        return new DefaultHandshakeHandler(
                new JettyRequestUpgradeStrategy(new WebSocketServerFactory(policy)));
    }
```

更多教程请访问:www.flydean.com

```
}
```

下面的示例显示与前面的示例等效的XML配置：

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

    <websocket:handlers>
        <websocket:mapping path="/echo" handler="echoHandler"/>
        <websocket:handshake-handler ref="handshakeHandler"/>
    </websocket:handlers>

    <bean id="handshakeHandler" class="org.springframework...DefaultHandshakeHandler">
        <constructor-arg ref="upgradeStrategy"/>
    </bean>

    <bean id="upgradeStrategy" class="org.springframework...JettyRequestUpgradeStrategy">
        <constructor-arg ref="serverFactory"/>
    </bean>

    <bean id="serverFactory" class="org.eclipse.jetty...WebSocketServerFactory">
        <constructor-arg>
            <bean class="org.eclipse.jetty...WebSocketPolicy">
                <constructor-arg value="SERVER"/>
                <property name="inputBufferSize" value="8092"/>
                <property name="idleTimeout" value="600000"/>
            </bean>
        </constructor-arg>
    </bean>

</beans>
```

4.2.5 Allowed Origins

从Spring Framework 4.1.5开始，WebSocket和SockJS的默认行为是仅接受同源请求。也可以允许所有或指定的来源列表。此检查主要用于浏览器客户端。没有任何措施可以阻止其他类型的客户端修改Origin头值（有关更多详细信息，请参阅RFC 6454：Web Origin概念）。

三种可能的行为是：

- 仅允许相同来源的请求（默认）：在此模式下，启用SockJS时，Iframe HTTP响应标头X-Frame-Options设置为SAMEORIGIN，并且JSONP传输被禁用，因为它不允许检查请求。因此，启用此模式时，不支持IE6和IE7。
- 允许指定的来源列表：每个允许的来源必须以http：//或https：//开头。在此模式下，启用SockJS时，将禁用IFrame传输。因此，启用此模式时，不支持IE6到IE9。
- 允许所有origins：要启用此模式，应提供*作为允许的origins值。在这种模式下，所有传输都可用。

您可以配置WebSocket和SockJS允许的来源，如以下示例所示：

```
import org.springframework.web.socket.config.annotation.EnableWebSocket;
import org.springframework.web.socket.config.annotation.WebSocketConfigurer;
import org.springframework.web.socket.config.annotation.WebSocketHandlerRegistry;

@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {
```

更多教程请访问:www.flydean.com

```
    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(myHandler(), "/myHandler").setAllowedOrigins("https://mydomain.com");
    }

    @Bean
    public WebSocketHandler myHandler() {
        return new MyHandler();
    }

}
```

下面的示例显示与前面的示例等效的XML配置：

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

    <websocket:handlers allowed-origins="https://mydomain.com">
        <websocket:mapping path="/myHandler" handler="myHandler" />
    </websocket:handlers>

    <bean id="myHandler" class="org.springframework.samples.MyHandler"/>

</beans>
```

更多教程请访问:www.flydean.com

4.3 SockJS Fallback

在公共Internet上，控件外的限制性代理可能会阻止WebSocket交互，这可能是因为它们未配置为传递Upgrade标头，或者是因为它们关闭了长期处于空闲状态的连接。

解决此问题的方法是WebSocket仿真，即先尝试使用WebSocket，然后尝试使用基于HTTP的技术来模拟WebSocket交互并公开相同的应用程序级API。

在Servlet堆栈上，Spring框架为SockJS协议提供服务器（和客户端）支持。

4.3.1 Overview

SockJS的目标是让应用程序使用WebSocket API，但在运行时必要时使用非WebSocket替代方案，而无需更改应用程序代码。

SockJS包括：

- SockJS协议以可执行叙述测试的形式定义。
- SockJS JavaScript客户端-一种在浏览器中使用的客户端库。
- SockJS服务器实现，包括Spring框架spring-websocket模块中的一个。
- spring-websocket模块中的SockJS Java客户端（从4.1版开始）。

SockJS设计用于浏览器。它使用多种技术来支持各种浏览器版本。有关SockJS传输类型和浏览器的完整列表，请参见SockJS客户端页面。传输分为三大类：WebSocket，HTTP流和HTTP长轮询。有关这些类别的概述，请参阅此博客文章。

SockJS客户端首先发送GET / info以从服务器获取基本信息。在那之后，它必须决定使用哪种交通工具。如果可能，请使用WebSocket。如果没有，在大多数浏览器中，至少有一个HTTP流选项。如果不是，则使用HTTP（长）轮询。

所有传输请求都具有以下URL结构：

```
http://host:port/myApp/myEndpoint/{server-id}/{session-id}/{transport}
```

这里：

- {server-id}对于在集群中路由请求很有用，但否则不使用。
- {session-id}关联属于SockJS会话的HTTP请求。
- {transport}表示传输类型（例如，websocket，xhr-streaming等）。

WebSocket传输仅需要单个HTTP请求即可进行WebSocket握手。此后所有消息在该套接字上交换。

HTTP传输需要更多请求。例如，Ajax / XHR流依赖于对服务器到客户端消息的一个长时间运行的请求，以及对客户端到服务器消息的其他HTTP POST请求。长轮询类似于长轮询，只是它在每次服务器到客户端发送后结束当前请求。

SockJS添加了最少的消息框架。例如，服务器最初发送字母o（“open”帧），消息以a [“ message1”，“ message2”]（JSON编码数组）发送，如果没有消息，则以字母h（“heartbeat”帧）发送持续25秒（默认情况下），然后使用字母c（“close”框）关闭会话。

要了解更多信息，请在浏览器中运行示例并查看HTTP请求。 SockJS客户端允许修复传输列表，因此可以一次查看每个传输。 SockJS客户端还提供了调试标志，该标志可在浏览器控制台中启用有用的消息。在服务器端，您可以为org.springframework.web.socket启用TRACE日志记录。有关更多详细信息，请参见SockJS协议旁白测试。

4.3.2 Enabling SockJS

更多教程请访问:www.flydean.com

您可以通过Java配置启用SockJS，如以下示例所示：

```
@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(myHandler(), "/myHandler").withSockJS();
    }

    @Bean
    public WebSocketHandler myHandler() {
        return new MyHandler();
    }

}
```

下面的示例显示与前面的示例等效的XML配置：

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

    <websocket:handlers>
        <websocket:mapping path="/myHandler" handler="myHandler"/>
        <websocket:sockjs/>
    </websocket:handlers>

    <bean id="myHandler" class="org.springframework.samples.MyHandler"/>

</beans>
```

前面的示例用于Spring MVC应用程序，应包含在DispatcherServlet的配置中。 但是，Spring的WebSocket和SockJS支持不依赖于Spring MVC。 在SockJsHttpRequestHandler的帮助下，将其集成到其他HTTP服务环境中相对简单。

在浏览器端，应用程序可以使用sockjs-client（版本1.0.x）。 它模拟W3C WebSocket API，并与服务器通信以选择最佳的传输选项，具体取决于运行它的浏览器。 请参阅sockjs-client页面和浏览器支持的传输类型列表。 客户端还提供了几个配置选项，例如用于指定要包括的传输。

4.3.3 IE 8 and 9

Internet Explorer 8和9仍在使用。它们是拥有SockJS的关键原因。本节涵盖有关在这些浏览器中运行的重要注意事项。

SockJS客户端通过使用Microsoft的XDomainRequest在IE 8和9中支持Ajax / XHR流。跨域有效，但不支持发送Cookie。 Cookies对于Java应用程序通常是必不可少的。但是，由于SockJS客户端可用于多种服务器类型（而不仅仅是Java类型），因此它需要知道cookie是否重要。如果是这样，则SockJS客户端更喜欢Ajax / XHR进行流传输。否则，它依赖于基于iframe的技术。

SockJS客户端发出的第一个/ info请求是对可能影响客户端选择传输方式的信息的请求。这些详细信息之一是服务器应用程序是否依赖Cookie（例如，出于身份验证目的或使用粘性会话进行群集）。 Spring对SockJS的支持包括一个名为sessionCookieNeeded的属性。由于大多数Java应用程序都依赖JSESSIONID cookie，因此默认情况下启用此功能。如果您的应用程序不需要它，则可以关闭此选项，然后SockJS客户端应在IE 8和9中选择xdr-streaming。

更多教程请访问:www.flydean.com

如果您确实使用基于iframe的传输，请记住，可以通过将HTTP响应标头X-Frame-Options设置为DENY，SAMEORIGIN或ALLOW-FROM <origin>来指示浏览器阻止在给定页面上使用iframe 。这用于防止点击劫持。

> Spring Security 3.2+提供了对每个响应设置X-Frame-Options的支持。 默认情况下，Spring Security Java配置将其设置为DENY。 在3.2中，Spring Security XML名称空间默认情况下不设置该标头，但可以配置为这样做。 将来，它可能会默认设置。
>
> 有关如何配置X-Frame-Options标头的设置的详细信息，请参见Spring Security文档的Default Security Headers。 您也可以查看SEC-2501以获取更多背景信息。

如果您的应用程序添加了X-Frame-Options响应标头（应如此！）并依赖于基于iframe的传输，则需要将标头值设置为SAMEORIGIN或ALLOW-FROM <origin>。 Spring SockJS支持还需要知道SockJS客户端的位置，因为它是从iframe加载的。 默认情况下，iframe设置为从CDN位置下载SockJS客户端。 最好将此选项配置为使用与应用程序源相同的URL。

以下示例显示了如何在Java配置中执行此操作：

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/portfolio").withSockJS()
                .setClientLibraryUrl("http://localhost:8080/myapp/js/sockjs-client.js");
    }

    // ...

}
```

XML名称空间通过<websocket：sockjs>元素提供了类似的选项。

> 在初始开发过程中，请启用SockJS客户端开发模式，以防止浏览器缓存本应缓存的SockJS请求（如iframe）。 有关如何启用它的详细信息，请参见SockJS客户端页面。

4.3.4 Heartbeats

SockJS协议要求服务器发送心跳消息，以防止代理断定连接已挂起。 Spring SockJS配置具有一个名为heartbeatTime的属性，可用于自定义频率。 默认情况下，假设没有其他消息在该连接上发送，则心跳将在25秒后发送。 这个25秒的值符合以下IETF对公共Internet应用程序的建议。

> 在WebSocket和SockJS上使用STOMP时，如果STOMP客户端和服务器协商要交换的心跳，则会禁用SockJS心跳。

Spring SockJS支持还允许您配置TaskScheduler来调度心跳任务。 任务调度程序由线程池支持，其默认设置基于可用处理器的数量。 您应该考虑根据您的特定需求自定义设置。

4.3.5 Client Disconnects

HTTP流和HTTP长轮询SockJS传输要求连接保持打开的时间比平常更长。有关这些技术的概述，请参见此博客文章。

在Servlet容器中，这是通过Servlet 3异步支持完成的，该支持允许退出Servlet容器线程，处理请求并继续写入另一个线程的响应。

更多教程请访问:www.flydean.com

一个特定的问题是，Servlet API不会为已离开的客户端提供通知。请参阅eclipse-ee4j / servlet-api # 44。但是，Servlet容器在随后尝试写入响应时会引发异常。由于Spring的SockJS服务支持服务器发送的心跳信号（默认情况下，每25秒发送一次），这意味着通常会在该时间段内（或更早，如果消息发送频率更高）检测到客户端断开连接。

> 结果，由于客户端已断开连接，可能会发生网络I / O故障，从而可能在日志中填充不必要的堆栈跟踪。 Spring会尽最大努力找出代表客户端断开连接（特定于每个服务器）的网络故障，并通过使用专用日志类别DISCONNECTED_CLIENT_LOG_CATEGORY（在AbstractSockJsSession中定义）来记录最少的消息。如果需要查看堆栈跟踪，可以将该日志类别设置为TRACE。

4.3.6 SockJS and CORS

如果允许跨域请求（请参阅“允许的起源”），则SockJS协议将CORS用于XHR流和轮询传输中的跨域支持。因此，除非在响应中检测到CORS标头的存在，否则将自动添加CORS标头。因此，如果已经将应用程序配置为提供CORS支持（例如，通过Servlet过滤器），则Spring的SockJsService会跳过这一部分。

也可以通过在Spring的SockJsService中设置preventCors属性来禁用添加这些CORS标头。

SockJS需要以下标头和值：

- Access-Control-Allow-Origin：从Origin请求标头的值初始化。
- Access-Control-Allow-Credentials：始终设置为true。
- Access-Control-Request-Headers：从等效请求标头中的值初始化。
- Access-Control-Allow-Methods：传输支持的HTTP方法（请参阅TransportType枚举）。
- Access-Control-Max-Age：设置为31536000（1年）。

有关确切的实现，请参见AbstractSockJsService中的addCorsHeaders和源代码中的TransportType枚举。

另外，如果CORS配置允许，请考虑排除带有SockJS端点前缀的URL，从而让Spring的SockJsService处理它。

4.3.7 SockJsClient

Spring提供了一个SockJS Java客户端，无需使用浏览器即可连接到远程SockJS端点。当需要通过公用网络在两个服务器之间进行双向通信时（即，网络代理可能会阻止使用WebSocket协议），这特别有用。 SockJS Java客户端对于测试目的也非常有用（例如，模拟大量并发用户）。

SockJS Java客户端支持websocket，xhr-streaming和xhr-polling传输。其余的仅在浏览器中有意义。

您可以使用以下命令配置WebSocketTransport：

- JSR-356运行时中的StandardWebSocketClient。
- 通过使用Jetty 9+本机WebSocket API来创建JettyWebSocketClient。
- Spring的WebSocketClient的任何实现。

根据定义，XhrTransport支持xhr-streaming和xhr-polling，因为从客户端的角度来看，除了用于连接服务器的URL之外没有其他区别。当前有两种实现：

- RestTemplateXhrTransport使用Spring的RestTemplate进行HTTP请求。
- JettyXhrTransport使用Jetty的HttpClient进行HTTP请求。

以下示例显示了如何创建SockJS客户端并连接到SockJS端点：

```
List<Transport> transports = new ArrayList<>(2);
transports.add(new WebSocketTransport(new StandardWebSocketClient()));
transports.add(new RestTemplateXhrTransport());

SockJsClient sockJsClient = new SockJsClient(transports);
sockJsClient.doHandshake(new MyWebSocketHandler(), "ws://example.com:8080/sockjs");
```

更多教程请访问:www.flydean.com

> SockJS对消息使用JSON格式的数组。 默认情况下，使用Jackson 2，并且需要在类路径上。 或者，您可以配置SockJsMessageCodec的自定义实现，并在SockJsClient上对其进行配置。

要使用SockJsClient模拟大量并发用户，您需要配置基础HTTP客户端（用于XHR传输）以允许足够数量的连接和线程。 以下示例显示了如何使用Jetty进行操作：

```
HttpClient jettyHttpClient = new HttpClient();
jettyHttpClient.setMaxConnectionsPerDestination(1000);
jettyHttpClient.setExecutor(new QueuedThreadPool(1000));
```

以下示例显示了与服务器端SockJS相关的属性（有关详细信息，请参见javadoc），您还应考虑自定义：

```
@Configuration
public class WebSocketConfig extends WebSocketMessageBrokerConfigurationSupport {

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/sockjs").withSockJS()
            .setStreamBytesLimit(512 * 1024)
            .setHttpMessageCacheSize(1000)
            .setDisconnectDelay(30 * 1000);
    }

    // ...
}
```

更多教程请访问:www.flydean.com

4.4 STOMP

WebSocket协议定义了两种消息类型（文本消息和二进制消息），但是其内容未定义。 该协议定义了一种机制，供客户端和服务器协商用于在WebSocket之上使用的子协议（即高级消息协议），以定义每种协议可以发送的消息类型，格式，内容。 每个消息，依此类推。 子协议的使用是可选的，但无论哪种方式，客户端和服务器都需要就定义消息内容的某种协议达成共识。

4.4.1 Overview

STOMP（面向简单文本的消息传递协议）最初是为脚本语言（例如Ruby，Python和Perl）创建的，以连接到企业消息代理。 它旨在解决常用消息传递模式的最小子集。 STOMP可以在任何可靠的双向流网络协议上使用，例如TCP和WebSocket。 尽管STOMP是面向文本的协议，但是消息有效负载可以是文本或二进制。

STOMP是基于帧的协议，其帧以HTTP为模型。 以下清单显示了STOMP帧的结构：

```
COMMAND
header1:value1
header2:value2

Body^@
```

客户端可以使用SEND或SUBSCRIBE命令发送或订阅消息，以及描述消息的内容和接收者的目的地标头。这启用了一种简单的发布-订阅机制，您可以使用该机制通过代理将消息发送到其他连接的客户端，或者将消息发送到服务器以请求执行某些工作。

使用Spring的STOMP支持时，Spring WebSocket应用程序将充当客户端的STOMP代理。消息被路由到@Controller消息处理方法或简单的内存代理，该代理跟踪订阅并向订阅的用户广播消息。您还可以将Spring配置为与专用的STOMP代理（例如RabbitMQ，ActiveMQ等）一起使用，以实际广播消息。在那种情况下，Spring维护与代理的TCP连接，将消息中继到该代理，并将消息从该代理向下传递到已连接的WebSocket客户端。因此，Spring Web应用程序可以依靠基于HTTP的统一安全性，通用验证以及用于消息处理的熟悉的编程模型。

以下示例显示了一个订阅以接收股票报价的客户端，服务器可能会定期发出该股票报价（例如，通过计划任务，该任务通过SimpMessagingTemplate向代理发送消息）：

```
SUBSCRIBE
id:sub-1
destination:/topic/price.stock.*

^@
```

以下示例显示了一个发送交易请求的客户端，服务器可以通过@MessageMapping方法处理该请求：

```
SEND
destination:/queue/trade
content-type:application/json
content-length:44

{"action":"BUY","ticker":"MMM","shares",44}^@
```

执行后，服务器可以向客户广播交易确认消息和详细信息。

更多教程请访问:www.flydean.com

在STOMP规范中，目的地的含义是故意不透明的。 它可以是任何字符串，并且完全由STOMP服务器定义它们所支持的目的地的语义和语法。 但是，目的地通常是类似路径的字符串，其中/ topic / ..表示发布-订阅（一对多），而/ queue /表示点对点（一对一）消息 交流。

STOMP服务器可以使用MESSAGE命令向所有订户广播消息。 以下示例显示了服务器向订阅的客户端发送股票报价的服务器：

```
MESSAGE
message-id:nxahklf6-1
subscription:sub-1
destination:/topic/price.stock.MMM

{"ticker":"MMM","price":129.45}^@
```

服务器无法发送未经请求的消息。 来自服务器的所有消息都必须响应特定的客户端订阅，并且服务器消息的subscription-id标头必须与客户端订阅的id标头匹配。

前面的概述旨在提供对STOMP协议的最基本的了解。 我们建议您全面阅读协议规范。

4.4.2 Benefits

与使用原始WebSocket相比，使用STOMP作为子协议可以使Spring Framework和Spring Security提供更丰富的编程模型。 关于HTTP与原始TCP以及它如何使Spring MVC和其他Web框架提供丰富的功能，可以得出相同的观点。 以下是好处列表：

- 无需发明自定义消息协议和消息格式。
- 可以使用STOMP客户端，包括Spring框架中的Java客户端。
- 您可以（可选）使用消息代理（例如RabbitMQ，ActiveMQ和其他代理）来管理订阅和广播消息。
- 可以在任意数量的@Controller实例中组织应用程序逻辑，并且可以基于STOMP目标标头将消息路由到它们，而不是通过给定连接使用单个WebSocketHandler处理原始WebSocket消息。
- 您可以使用Spring Security基于STOMP目的地和消息类型来保护消息。

4.4.3 Enable STOMP

spring-messaging和spring-websocket模块中提供了STOMP over WebSocket支持。 一旦有了这些依赖关系，就可以使用SockJS Fallback通过WebSocket公开STOMP端点，如以下示例所示：

```java
import org.springframework.web.socket.config.annotation.EnableWebSocketMessageBroker;
import org.springframework.web.socket.config.annotation.StompEndpointRegistry;

@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/portfolio").withSockJS();
    }

    @Override
    public void configureMessageBroker(MessageBrokerRegistry config) {
        config.setApplicationDestinationPrefixes("/app");
        config.enableSimpleBroker("/topic", "/queue");
    }
}
```

更多教程请访问:www.flydean.com

下面的示例显示与前面的示例等效的XML配置：

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

    <websocket:message-broker application-destination-prefix="/app">
        <websocket:stomp-endpoint path="/portfolio">
            <websocket:sockjs/>
        </websocket:stomp-endpoint>
        <websocket:simple-broker prefix="/topic, /queue"/>
    </websocket:message-broker>

</beans>
```

> 对于内置的简单代理，/ topic和/ queue前缀没有任何特殊含义。 它们仅是区分发布订阅消息传递和点对点消息传递的约定（即，许多订户与一个消费者）。 使用外部代理时，请检查代理的STOMP页面以了解其支持哪种STOMP目标和前缀。

要从浏览器进行连接，对于SockJS，您可以使用sockjs-client。 对于STOMP，许多应用程序都使用了jmesnil / stomp-websocket库（也称为stomp.js），该库功能齐全，已在生产中使用多年，但不再维护。 目前，JSteunou / webstomp-client是该库中最活跃且发展最快的后继程序。 以下示例代码基于此：

```
var socket = new SockJS("/spring-websocket-portfolio/portfolio");
var stompClient = webstomp.over(socket);

stompClient.connect({}, function(frame) {
}
```

另外，如果您通过WebSocket（没有SockJS）进行连接，则可以使用以下代码：

```
var socket = new WebSocket("/spring-websocket-portfolio/portfolio");
var stompClient = Stomp.over(socket);

stompClient.connect({}, function(frame) {
}
```

请注意，前面示例中的stompClient不需要指定登录名和密码标头。 即使这样做，它们也会在服务器端被忽略（或更确切地说，被覆盖）。 有关身份验证的更多信息，请参见连接到代理和身份验证。

4.4.4 WebSocket Server

要配置基础的WebSocket服务器，请应用“服务器配置”中的信息。 对于Jetty，但是您需要通过StompEndpointRegistry设置HandshakeHandler和WebSocketPolicy：

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/portfolio").setHandshakeHandler(handshakeHandler());
    }
```

更多教程请访问:www.flydean.com

```
    @Bean
    public DefaultHandshakeHandler handshakeHandler() {

        WebSocketPolicy policy = new WebSocketPolicy(WebSocketBehavior.SERVER);
        policy.setInputBufferSize(8192);
        policy.setIdleTimeout(600000);

        return new DefaultHandshakeHandler(
                new JettyRequestUpgradeStrategy(new WebSocketServerFactory(policy)));
    }
}
```

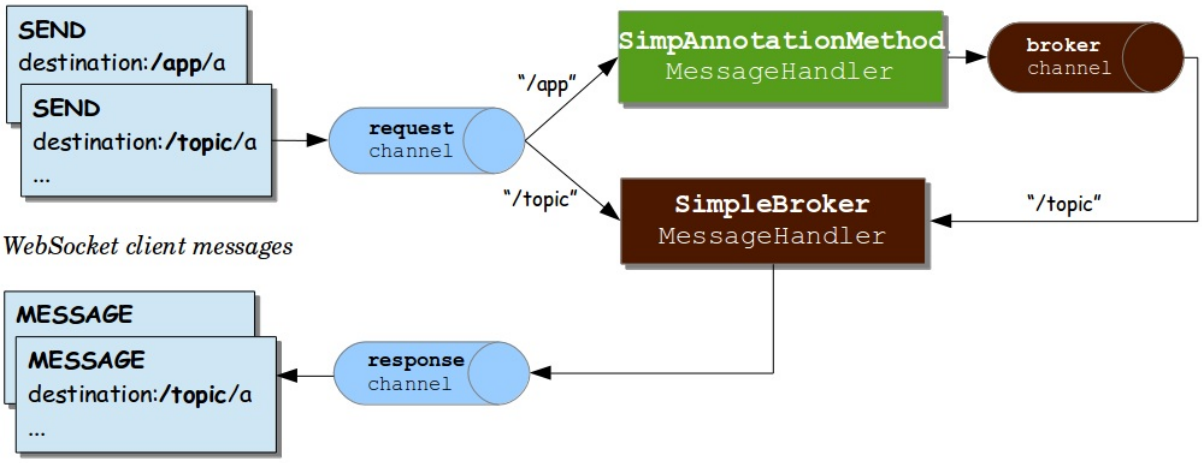
4.4.5 Flow of Messages

公开STOMP端点后，Spring应用程序将成为已连接客户端的STOMP代理。本节描述服务器端的消息流。

spring-messaging模块包含对起源于Spring Integration的消息传递应用程序的基础支持，后来被提取并合并到Spring Framework中，以便在许多Spring项目和应用程序场景中更广泛地使用。下面的列表简要描述了一些可用的消息传递抽象：

- Message：消息的简单表示形式，包括标题和有效负载。
- MessageHandler：处理消息的合同。
- MessageChannel：用于发送使生产者和使用者之间松散耦合的消息的合同。
- SubscribableChannel：具有MessageHandler订阅者的MessageChannel。
- ExecutorSubscribableChannel：使用Executor传递消息的SubscribableChannel。

Java配置（即@EnableWebSocketMessageBroker）和XML名称空间配置（即<websocket：message-broker>）都使用前面的组件来组装消息工作流。下图显示了启用简单内置消息代理时使用的组件：

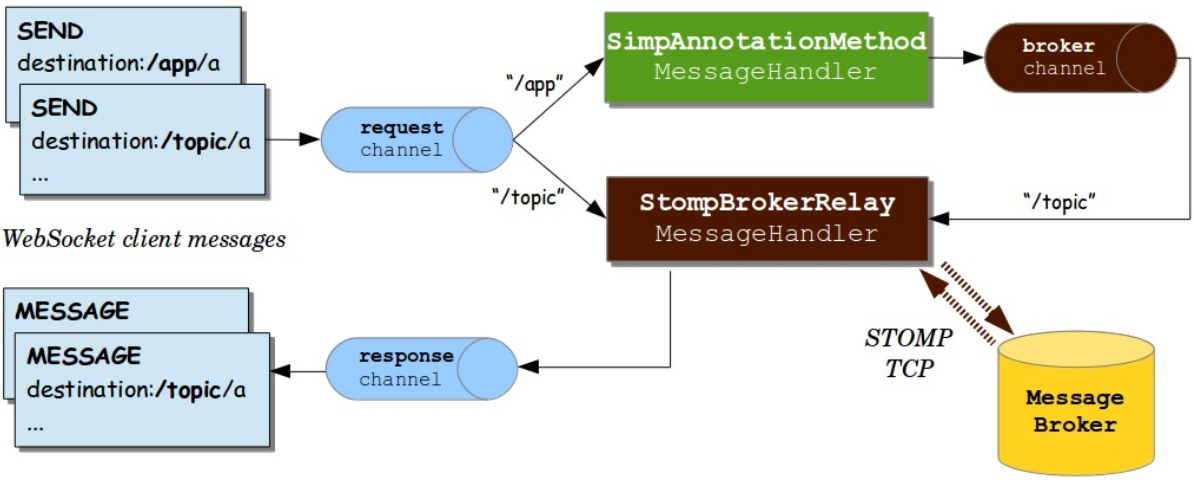


上图显示了三个消息通道：

- clientInboundChannel：用于传递从WebSocket客户端收到的消息。
- clientOutboundChannel：用于向WebSocket客户端发送服务器消息。
- brokerChannel：用于从服务器端应用程序代码内将消息发送到消息代理。

下图显示了将外部代理（例如RabbitMQ）配置为用于管理订阅和广播消息时使用的组件：

更多教程请访问:www.flydean.com



前面两个图之间的主要区别是使用“代理中继”将消息通过TCP传递到外部STOMP代理，以及将消息从代理向下传递给订阅的客户端。

当从WebSocket连接接收到消息时，消息将被解码为STOMP帧，转换为Spring消息表示形式，然后发送到clientInboundChannel进行进一步处理。例如，目标标头以/ app开头的STOMP消息可以路由到带注解的控制器中的@MessageMapping方法，而/ topic和/ queue消息可以直接路由到消息代理。

处理来自客户端的STOMP消息的带注解的@Controller可以通过brokerChannel将消息发送到消息代理，并且代理通过clientOutboundChannel将消息广播给匹配的订户。相同的控制器还可以响应HTTP请求执行相同的操作，因此客户端可以执行HTTP POST，然后@PostMapping方法可以将消息发送到消息代理，以广播到订阅的客户端。

我们可以通过一个简单的示例跟踪流程。考虑以下示例，该示例设置了服务器：

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/portfolio");
    }

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.setApplicationDestinationPrefixes("/app");
        registry.enableSimpleBroker("/topic");
    }
}

@Controller
public class GreetingController {

    @MessageMapping("/greeting") {
    public String handle(String greeting) {
        return "[" + getTimestamp() + ": " + greeting;
    }
}
```

前面的示例支持以下流程：

- 客户端连接到http：// localhost：8080 / portfolio，一旦建立了WebSocket连接，STOMP帧就开始在其上流动。
- 客户端发送SUBSCRIBE帧，其目标标头为/ topic / greeting。接收并解码后，该消息将发送到clientInboundChannel，然后路由到消息代理，该代理存储客户端订阅。
- 客户端向/ app / greeting发送一个aSEND帧。 / app前缀有助于将其路由到带注解的控制器。删除/ app前缀后，目标的其余/ greeting部分将映射到GreetingController中的@MessageMapping方法。

更多教程请访问:www.flydean.com

- 从GreetingController返回的值将根据返回值和/ topic / greeting的默认目标标头（从输入目标使用/ app替换为/ topic的情况下）转换为带有有效负载的Spring消息。结果消息将发送到brokerChannel并由消息代理处理。
- 消息代理找到所有匹配的订户，并通过clientOutboundChannel向每个发送一个MESSAGE帧，消息从此处被编码为STOMP帧并通过WebSocket连接发送。

下一节将提供有关带注解方法的更多详细信息，包括支持的参数类型和返回值。

4.4.6 Annotated Controllers

应用程序可以使用带注解的@Controller类来处理来自客户端的消息。 这样的类可以声明@ MessageMapping，@ SubscribeMapping和@ExceptionHandler方法，如以下主题所述：

- @MessageMapping
- @SubscribeMapping
- @MessageExceptionHandler
- @MessageMapping 您可以使用@MessageMapping注解根据消息的目的地路由消息的方法。 在方法级别和类型级别都支持它。 在类型级别，@MessageMapping用于表示控制器中所有方法之间的共享映射。

默认情况下，映射值是Ant样式的路径模式（例如/ thing ，*/ thing / **），包括对模板变量的支持（例如/ thing / {id}）。 可以通过@DestinationVariable方法参数引用这些值。 应用程序还可以切换到以点分隔的映射的目标约定，如“作为分隔符的点”中所述。

支持的方法参数

下表描述了方法参数：

Method argument	Description
Message	用于访问完整的消息。
MessageHeaders	用于访问中的标头Message。
MessageHeaderAccessor，SimpMessageHeaderAccessor和StompHeaderAccessor	用于通过类型化访问器方法访问标头。
@Payload	为了访问消息的有效负载，由configure转换（例如，从JSON转换）MessageConverter。不需要此注解，因为默认情况下会假定没有其他自变量匹配。您可以使用@javax.validation.Valid或Spring的注解有效负载参数@Validated，以使有效负载参数被自动验证。
@Header	用于访问特定的标头值-org.springframework.core.convert.converter.Converter如有必要，还可以使用进行类型转换 。
@Headers	用于访问消息中的所有标题。此参数必须可分配给 java.util.Map。
@DestinationVariable	用于访问从消息目标中提取的模板变量。根据需要将值转换为声明的方法参数类型。
java.security.Principal	反映在WebSocket HTTP握手时登录的用户。

Return Values

默认情况下，@ MessageMapping方法的返回值通过匹配的MessageConverter序列化为有效负载，并作为消息发送到brokerChannel，从该Channel广播到订阅者。出站邮件的目的地与入站邮件的目的地相同，但以/ topic为前缀。

您可以使用@SendTo和@SendToUser批注来自定义输出消息的目标。 @SendTo用于自定义目标位置或指定多个目标。 @SendToUser用于将输出消息定向到仅与输入消息关联的用户。请参阅用户目标。

更多教程请访问:www.flydean.com

您可以在同一方法上同时使用@SendTo和@SendToUser，并且它们在类级别都受支持，在这种情况下，它们充当类中方法的默认值。但是，请记住，任何方法级别的@SendTo或@SendToUser批注都会在类级别覆盖所有此类批注。

消息可以异步处理，@ MessageMapping方法可以返回ListenableFuture，CompletableFuture或CompletionStage。

请注意，@ SendTo和@SendToUser只是一种便利，等同于使用SimpMessagingTemplate发送消息。如有必要，对于更高级的方案，可以直接使用SimpMessagingTemplate来使用@MessageMapping方法。这可以代替返回值，也可以附加于返回值。请参阅发送消息。

@SubscribeMapping

@SubscribeMapping与@MessageMapping相似，但是将映射范围缩小到仅订阅消息。它支持相同的@ MessageMappingclientOutboundChannelbrokerChannel @ SendTo @ SendToUser

什么时候有用？假定代理映射到/ topic和/ queue，而应用程序控制器映射到/ app。在此设置中，代理将所有打算重复广播的/ topic和/ queue订阅都存储起来，并且不需要应用程序参与其中。客户端还可以订阅某个/ app目的地，并且控制器可以响应该订阅而返回一个值，而无需经纪人，而无需再次存储或使用该订阅（实际上是一次请求-答复交换）。一个用例是在启动时用初始数据填充UI。

什么时候没有用？不要尝试将代理和控制器映射到相同的目标前缀，除非出于某种原因您希望两者都独立处理消息（包括订阅）。入站消息是并行处理的。无法保证经纪人还是控制者首先处理给定的消息。如果要在存储预订并准备好广播时通知目标，则客户端应请求服务器是否支持收据（简单代理不支持）。例如，对于Java STOMP客户端，您可以执行以下操作添加收据：

```
@Autowired
private TaskScheduler messageBrokerTaskScheduler;

// During initialization..
stompClient.setTaskScheduler(this.messageBrokerTaskScheduler);

// When subscribing..
StompHeaders headers = new StompHeaders();
headers.setDestination("/topic/...");
headers.setReceipt("r1");
FrameHandler handler = ...;
stompSession.subscribe(headers, handler).addReceiptTask(() -> {
    // Subscription ready...
});
```

服务器端的选项是在brokerChannel上注册ExecutorChannelInterceptor并实现afterMessageHandled方法，该方法在处理包括订阅在内的消息之后被调用。

@MessageExceptionHandler

应用程序可以使用@MessageExceptionHandler方法来处理@MessageMapping方法中的异常。 如果要访问异常实例，则可以在批注本身中声明异常，也可以通过方法参数声明异常。 以下示例通过方法参数声明异常：

```
@Controller
public class MyController {

    // ...

    @MessageExceptionHandler
    public ApplicationError handleException(MyException exception) {
        // ...
        return appError;
    }
}
```

更多教程请访问:www.flydean.com

@MessageExceptionHandler方法支持灵活的方法签名，并支持与@MessageMapping方法相同的方法参数类型和返回值。

通常，@ MessageExceptionHandler方法适用于声明它们的@Controller类（或类层次结构）。 如果希望此类方法在全局范围内（跨控制器）应用，则可以在标有@ControllerAdvice的类中声明它们。 这与Spring MVC中可用的类似支持相当。

4.4.7 Sending Messages

如果要从应用程序的任何部分向连接的客户端发送消息怎么办？ 任何应用程序组件都可以将消息发送到brokerChannel。 最简单的方法是注入SimpMessagingTemplate并使用它发送消息。 通常，您将按类型注入它，如以下示例所示：

```
@Controller
public class GreetingController {

    private SimpMessagingTemplate template;

    @Autowired
    public GreetingController(SimpMessagingTemplate template) {
        this.template = template;
    }

    @RequestMapping(path="/greetings", method=POST)
    public void greet(String greeting) {
        String text = "[" + getTimestamp() + "]:" + greeting;
        this.template.convertAndSend("/topic/greetings", text);
    }

}
```

但是，如果存在另一个相同类型的bean，也可以通过其名称（brokerMessagingTemplate）对其进行限定。

4.4.8 Simple Broker

内置的简单消息代理处理来自客户端的订阅请求，将其存储在内存中，并将消息广播到具有匹配目标的已连接客户端。 该代理支持类似路径的目标，包括对Ant样式目标模式的订阅。

> 应用程序还可以使用点分隔（而不是斜杠分隔）目标。 请参见点作为分隔符。

如果配置了任务调度程序，则简单代理支持STOMP心跳。 为此，您可以声明自己的调度程序，也可以使用内部自动声明和使用的调度程序。 以下示例显示如何声明自己的调度程序：

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    private TaskScheduler messageBrokerTaskScheduler;

    @Autowired
    public void setMessageBrokerTaskScheduler(TaskScheduler taskScheduler) {
        this.messageBrokerTaskScheduler = taskScheduler;
    }

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {

        registry.enableSimpleBroker("/queue/", "/topic/")
                .setHeartbeatValue(new long[] {10000, 20000})
```

更多教程请访问:www.flydean.com

```
                .setTaskScheduler(this.messageBrokerTaskScheduler);

        // ...
    }
}
```

4.4.9 External Broker

简单代理非常适合入门，但是仅支持STOMP命令的一个子集（它不支持ack，回执和其他一些功能），它依赖于简单的消息发送循环，并且不适合于集群。 或者，您可以升级应用程序以使用功能齐全的消息代理。

请参阅STOMP文档以了解您选择的消息代理（例如RabbitMQ，ActiveMQ和其他），安装代理，并在启用STOMP支持的情况下运行它。 然后，您可以在Spring配置中启用STOMP代理中继（而不是简单代理）。

以下示例配置启用了功能齐全的代理：

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/portfolio").withSockJS();
    }

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.enableStompBrokerRelay("/topic", "/queue");
        registry.setApplicationDestinationPrefixes("/app");
    }

}
```

下面的示例显示与前面的示例等效的XML配置：

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

    <websocket:message-broker application-destination-prefix="/app">
        <websocket:stomp-endpoint path="/portfolio" />
            <websocket:sockjs/>
        </websocket:stomp-endpoint>
        <websocket:stomp-broker-relay prefix="/topic,/queue" />
    </websocket:message-broker>

</beans>
```

先前配置中的STOMP代理中继是Spring MessageHandler，它通过将消息转发到外部消息代理来处理消息。 为此，它建立到代理的TCP连接，将所有消息转发给它，然后通过它们的WebSocket会话将从代理收到的所有消息转发给客户端。 本质上，它充当双向转发消息的“中继”。

> 将io.projectreactor.netty：reactor-netty和io.netty：netty-all所有依赖项添加到项目中以进行TCP连接管理。

此外，应用程序组件（例如HTTP请求处理方法，业务服务等）也可以将消息发送到代理中继，如“发送消息”中所述，以将消息广播到订阅的WebSocket客户端。

更多教程请访问:www.flydean.com

实际上，代理中继可实现健壮且可伸缩的消息广播。

4.4.10 Connecting to a Broker

STOMP代理中继器维护与代理的单个“系统” TCP连接。此连接仅用于源自服务器端应用程序的消息，而不用于接收消息。您可以为此连接配置STOMP凭据（即STOMP框架登录名和密码标头）。这在XML名称空间和Java配置中都以systemLogin和systemPasscode属性（默认值为guest和guest）公开。

STOMP代理中继还为每个连接的WebSocket客户端创建一个单独的TCP连接。您可以配置用于代表客户端创建的所有TCP连接的STOMP凭据。这在XML名称空间和Java配置中都显示为具有默认值guest和guest的clientLogin和clientPasscode属性。

> STOMP代理中继始终在代表客户端转发给代理的每个CONNECT帧上设置登录和密码标头。因此，WebSocket客户端无需设置这些标头。他们被忽略。如“身份验证”部分所述，WebSocket客户端应改为依靠HTTP身份验证来保护WebSocket端点并建立客户端身份。

STOMP代理中继还通过“系统” TCP连接向消息代理发送和从消息代理接收心跳。您可以配置发送和接收心跳的间隔（默认情况下，每个间隔为10秒）。如果与代理的连接断开，则代理中继每5秒继续尝试重新连接，直到成功。

当与代理的“系统”连接丢失并重新建立时，任何Spring bean都可以实现ApplicationListener <BrokerAvailabilityEvent>来接收通知。例如，当没有活动的“系统”连接时，广播股票报价的股票报价服务可以停止尝试发送消息。

默认情况下，STOMP代理中继始终连接到同一主机和端口，如果连接断开，则根据需要重新连接。如果希望提供多个地址，则在每次尝试连接时，都可以配置地址供应商，而不是固定的主机和端口。以下示例显示了如何执行此操作：

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig extends AbstractWebSocketMessageBrokerConfigurer {

    // ...

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.enableStompBrokerRelay("/queue/", "/topic/").setTcpClient(createTcpClient());
        registry.setApplicationDestinationPrefixes("/app");
    }

    private ReactorNettyTcpClient<byte[]> createTcpClient() {
        return new ReactorNettyTcpClient<>(
                client -> client.addressSupplier(() -> ... ),
                new StompReactorNettyCodec());
    }
}
```

您还可以使用virtualHost属性配置STOMP代理中继。 此属性的值设置为每个CONNECT帧的主机标头，并且很有用（例如，在建立TCP连接的实际主机与提供基于云的STOMP服务的主机不同的云环境中））。

4.4.11 Dots as Separators

将消息路由到@MessageMapping方法时，它们将与AntPathMatcher匹配。 默认情况下，模式应使用斜杠（/）作为分隔符。 这是Web应用程序中的一个良好约定，类似于HTTP URL。 但是，如果您更习惯于消息传递约定，则可以切换为使用点（.）作为分隔符。

以下示例显示了如何在Java配置中执行此操作：

```
@Configuration
@EnableWebSocketMessageBroker
```

更多教程请访问:www.flydean.com

```
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    // ...

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.setPathMatcher(new AntPathMatcher("."));
        registry.enableStompBrokerRelay("/queue", "/topic");
        registry.setApplicationDestinationPrefixes("/app");
    }
}
```

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

    <websocket:message-broker application-destination-prefix="/app" path-matcher="pathMatcher">
        <websocket:stomp-endpoint path="/stomp"/>
        <websocket:stomp-broker-relay prefix="/topic,/queue" />
    </websocket:message-broker>

    <bean id="pathMatcher" class="org.springframework.util.AntPathMatcher">
        <constructor-arg index="0" value="."/>
    </bean>

</beans>
```

之后，控制器可以使用点（.）作为@MessageMapping方法中的分隔符，如以下示例所示：

```
@Controller
@MessageMapping("red")
public class RedController {

    @MessageMapping("blue.{green}")
    public void handleGreen(@DestinationVariable String green) {
        // ...
    }
}
```

客户端现在可以将消息发送到/app/red.blue.green123。

在前面的示例中，我们没有更改“代理中继”上的前缀，因为这些前缀完全取决于外部消息代理。 有关您使用的代理的信息，请参见STOMP文档页面，以查看其对目标标头支持的约定。

另一方面，“简单代理”确实依赖于已配置的PathMatcher，因此，如果切换分隔符，该更改也将应用于代理，以及代理将目标从消息匹配到订阅中的模式的方式。

4.4.12 Authentication

每个通过WebSocket进行的STOMP消息传递会话均以HTTP请求开头。这可以是升级到WebSockets的请求（即WebSocket握手），或者在SockJS后备的情况下，可以是一系列SockJS HTTP传输请求。

许多Web应用程序已经具有身份验证和授权来保护HTTP请求。通常，使用某种机制（例如登录页面，HTTP基本认证或其他方式）通过Spring Security对用户进行认证。经过身份验证的用户的安全上下文保存在HTTP会话中，并与同一基于cookie的会话中的后续请求关联。

更多教程请访问:www.flydean.com

因此，对于WebSocket握手或SockJS HTTP传输请求，通常已经有一个可以通过HttpServletRequest # getUserPrincipal（）访问的经过身份验证的用户。 Spring会自动将该用户与为其创建的WebSocket或SockJS会话相关联，并随后与该会话中通过用户标头传输的所有STOMP消息相关联。

简而言之，典型的Web应用程序除了已经为安全起见，就不需要采取任何其他措施。通过基于cookie的HTTP会话（然后与为该用户创建的WebSocket或SockJS会话相关联）维护的安全上下文在HTTP请求级别对用户进行身份验证，并导致在每个消息流上标记用户头通过应用程序。

请注意，STOMP协议在CONNECT帧上确实具有登录名和密码标头。这些最初是设计用于并且仍然需要的，例如，基于TCP的STOMP。但是，对于默认情况下，对于基于WebSocket的STOMP，Spring会在STOMP协议级别忽略授权标头，并假定该用户已经在HTTP传输级别进行了身份验证，并期望WebSocket或SockJS会话包含已通过身份验证的用户。

> Spring Security提供了WebSocket子协议授权，该授权使用ChannelInterceptor来基于消息中的用户头对消息进行授权。另外，Spring Session提供了WebSocket集成，以确保当WebSocket会话仍处于活动状态时，用户HTTP会话不会过期。

4.4.13 Token Authentication

Spring Security OAuth支持基于令牌的安全性，包括JSON Web令牌（JWT）。您可以将其用作Web应用程序中的身份验证机制，包括上一节中所述的WebSocket交互中的STOMP（即，通过基于cookie的会话维护身份）。

同时，基于cookie的会话并非总是最合适的（例如，在不维护服务器端会话的应用程序中或在通常使用标头进行身份验证的移动应用程序中）。

WebSocket协议RFC 6455“没有规定服务器可以在WebSocket握手期间对客户端进行身份验证的任何特定方式。”但是，实际上，浏览器客户端只能使用标准身份验证标头（即基本HTTP身份验证）或cookie，并且不能使用（例如）提供自定义标题。同样，SockJS JavaScript客户端也不提供通过SockJS传输请求发送HTTP标头的方法。请参阅sockjs-client发行号196。相反，它确实允许发送可用于发送令牌的查询参数，但是有其自身的缺点（例如，令牌可能会无意中与服务器日志中的URL一起记录）。

> 前面的限制适用于基于浏览器的客户端，不适用于基于Spring Java的STOMP客户端，该客户端支持通过WebSocket和SockJS请求发送标头。

因此，希望避免使用cookie的应用程序可能没有在HTTP协议级别进行身份验证的任何好的替代方法。他们可能更喜欢在STOMP消息传递协议级别使用标头进行身份验证，而不是使用cookie。这样做需要两个简单的步骤：

- 使用STOMP客户端在连接时传递身份验证头。
- 使用ChannelInterceptor处理身份验证头。

下一个示例使用服务器端配置来注册自定义身份验证拦截器。请注意，拦截器仅需要认证并在CONNECT消息上设置用户标头。 Spring记录并保存经过身份验证的用户，并将其与同一会话上的后续STOMP消息相关联。以下示例显示了如何注册自定义身份验证拦截器：

```
@Configuration
@EnableWebSocketMessageBroker
public class MyConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void configureClientInboundChannel(ChannelRegistration registration) {
        registration.interceptors(new ChannelInterceptor() {
            @Override
            public Message<?> preSend(Message<?> message, MessageChannel channel) {
                StompHeaderAccessor accessor =
                        MessageHeaderAccessor.getAccessor(message, StompHeaderAccessor.class);
                if (StompCommand.CONNECT.equals(accessor.getCommand())) {
                    Authentication user = ... ; // access authentication header(s)
                    accessor.setUser(user);
```

更多教程请访问:www.flydean.com

```
                }
                return message;
            }
        });
    }
}
```

此外，请注意，目前，当您使用Spring Security的消息授权时，需要确保在Spring Security之前订购了ChannelInterceptor身份验证配置。 最好通过在标有@Order（Ordered.HIGHEST_PRECEDENCE + 99）的WebSocketMessageBrokerConfigurer的实现中声明自定义拦截器来最好地完成此操作。

4.4.14 User Destinations

应用程序可以发送针对特定用户的消息，并且Spring的STOMP支持可以识别以/ user /为前缀的目标。例如，客户端可能订阅了/ user / queue / position-updates目标。该目标由UserDestinationMessageHandler处理，并转换为用户会话唯一的目标（例如/ queue / position-updates-user123）。这提供了订阅通用命名目的地的便利，同时确保与预订同一目的地的其他用户不发生冲突，以便每个用户都可以接收唯一的库存头寸更新。

在发送方，可以将消息发送到一个目的地，例如/ user / {username} / queue / position-updates，然后由UserDestinationMessageHandler将其转换为一个或多个目的地，每个与该用户关联的会话一个目的地。这使应用程序中的任何组件都可以发送针对特定用户的消息，而不必知道他们的姓名和通用目的地。注解和消息传递模板也支持此功能。

消息处理方法可以通过@SendToUser批注将消息发送给与正在处理的消息相关联的用户（在类级别上也支持共享公共目标），如以下示例所示：

```
@Controller
public class PortfolioController {

    @MessageMapping("/trade")
    @SendToUser("/queue/position-updates")
    public TradeResult executeTrade(Trade trade, Principal principal) {
        // ...
        return tradeResult;
    }
}
```

如果用户具有多个会话，则默认情况下，所有订阅给定目标的会话都是目标。 但是，有时可能仅需要将发送正在处理的消息的会话作为目标。 您可以通过将broadcast属性设置为false来做到这一点，如以下示例所示：

```
@Controller
public class MyController {

    @MessageMapping("/action")
    public void handleAction() throws Exception{
        // raise MyBusinessException here
    }

    @MessageExceptionHandler
    @SendToUser(destinations="/queue/errors", broadcast=false)
    public ApplicationError handleException(MyBusinessException exception) {
        // ...
        return appError;
    }
}
```

更多教程请访问:www.flydean.com

> 尽管用户目的地通常暗指经过身份验证的用户，但这并不是严格要求的。 不与已认证用户关联的WebSocket会话可以订阅用户目的地。 在这种情况下，@ SendToUser批注的行为与broadcast = false完全相同（也就是说，仅针对发送正在处理的消息的会话）。

您可以从任何应用程序组件向用户目标发送消息，例如，注入由Java配置或XML名称空间创建的SimpMessagingTemplate。 （如果使用@Qualifier进行资格鉴定，则bean名称为“ brokerMessagingTemplate”。）下面的示例演示如何执行此操作：

```
@Service
public class TradeServiceImpl implements TradeService {

    private final SimpMessagingTemplate messagingTemplate;

    @Autowired
    public TradeServiceImpl(SimpMessagingTemplate messagingTemplate) {
        this.messagingTemplate = messagingTemplate;
    }

    // ...

    public void afterTradeExecuted(Trade trade) {
        this.messagingTemplate.convertAndSendToUser(
                trade.getUserName(), "/queue/position-updates", trade.getResult());
    }
}
```

> 当将用户目标与外部消息代理一起使用时，应检查代理文档以了解如何管理非活动队列，以便在用户会话结束时，所有唯一的用户队列都将被删除。 例如，当您使用诸如/exchange/amq.direct/position-updates之类的目标时，RabbitMQ会创建自动删除队列。 因此，在这种情况下，客户端可以订阅/user/exchange/amq.direct/position-updates。 同样，ActiveMQ具有用于清除非活动目标的配置选项。

在多应用程序服务器方案中，由于用户连接到其他服务器，因此用户目的地可能无法解析。 在这种情况下，您可以配置目标以广播未解决的消息，以便其他服务器可以尝试。 这可以通过Java配置中MessageBrokerRegistry的userDestinationBroadcast属性以及XML中message-broker元素的user-destination-broadcast属性来完成。

4.4.15 Order of Messages

来自代理的消息被发布到clientOutboundChannel，从那里被写入WebSocket会话。 由于该通道由ThreadPoolExecutor支持，因此消息在不同的线程中处理，并且客户端接收到的结果序列可能与发布的确切顺序不匹配。

如果这是一个问题，请启用setPreservePublishOrder标志，如以下示例所示：

```
@Configuration
@EnableWebSocketMessageBroker
public class MyConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    protected void configureMessageBroker(MessageBrokerRegistry registry) {
        // ...
        registry.setPreservePublishOrder(true);
    }

}
```

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

更多教程请访问:www.flydean.com

```
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/websocket
        https://www.springframework.org/schema/websocket/spring-websocket.xsd">

    <websocket:message-broker preserve-publish-order="true">
        <!-- ... -->
    </websocket:message-broker>

</beans>
```

设置该标志后，同一客户端会话中的消息将一次发布到clientOutboundChannel，这样可以保证发布顺序。 请注意，这会产生很小的性能开销，因此，仅在需要时才应启用它。

4.4.16 Events

通过实施Spring的ApplicationListener接口，可以发布并接收多个ApplicationContext事件：

- BrokerAvailabilityEvent：指示代理何时可用或不可用。当“简单”代理在启动时立即可用并保持运行状态时，STOMP“代理中继”可能会失去与功能齐全的代理的连接（例如，如果代理重新启动）。代理中继具有重新连接逻辑，并在代理返回时重新建立与代理的“系统”连接。因此，只要状态从连接变为断开，反之亦然，就会发布此事件。使用SimpMessagingTemplate的组件应该订阅此事件，并避免在代理不可用时避免发送消息。无论如何，他们应该准备在发送消息时处理MessageDeliveryException。
- SessionConnectEvent：在收到新的STOMP CONNECT来指示新的客户端会话开始时发布。该事件包含代表连接的消息，包括会话ID，用户信息（如果有）和客户端发送的所有自定义标头。这对于跟踪客户端会话很有用。预订此事件的组件可以使用SimpMessageHeaderAccessor或StompMessageHeaderAccessor包装包含的消息。
- SessionConnectedEvent：在SessionConnectEvent之后不久，当代理已发送STOMP CONNECTED帧以响应CONNECT时，发布该消息。此时，可以认为STOMP会话已完全建立。
- SessionSubscribeEvent：在收到新的STOMP SUBSCRIBE时发布。
- SessionUnsubscribeEvent：在收到新的STOMP UNSUBSCRIBE时发布。
- SessionDisconnectEvent：在STOMP会话结束时发布。 DISCONNECT可能是从客户端发送的，也可能是在关闭WebSocket会话时自动生成的。在某些情况下，每个会话多次发布此事件。关于多个断开事件，组件应该是幂等的。

> 当您使用功能齐全的代理时，如果代理暂时不可用，则STOMP“代理中继”会自动重新连接“系统”连接。但是，客户端连接不会自动重新连接。假设启用了心跳，客户端通常会注意到代理在10秒内没有响应。客户端需要实现自己的重新连接逻辑。

4.4.17 Interception

事件为STOMP连接的生命周期提供通知，但不是为每条客户端消息提供通知。 应用程序还可以注册一个ChannelInterceptor来拦截处理链中任何消息。 以下示例显示如何拦截来自客户端的入站消息：

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void configureClientInboundChannel(ChannelRegistration registration) {
        registration.interceptors(new MyChannelInterceptor());
    }
}
```

定制ChannelInterceptor可以使用StompHeaderAccessor或SimpMessageHeaderAccessor来访问有关消息的信息，如以下示例所示：

更多教程请访问:www.flydean.com

```
public class MyChannelInterceptor implements ChannelInterceptor {

    @Override
    public Message<?> preSend(Message<?> message, MessageChannel channel) {
        StompHeaderAccessor accessor = StompHeaderAccessor.wrap(message);
        StompCommand command = accessor.getStompCommand();
        // ...
        return message;
    }
}
```

应用程序还可以实现ExecutorChannelInterceptor，它是ChannelInterceptor的子接口，在处理消息的线程中具有回调。虽然针对发送到通道的每个消息调用一次ChannelInterceptor，但ExecutorChannelInterceptor在订阅到通道消息的每个MessageHandler线程中提供了挂钩。

请注意，与前面介绍的SesionDisconnectEvent一样，DISCONNECT消息可以来自客户端，也可以在关闭WebSocket会话时自动生成。 在某些情况下，对于每个会话，拦截器可能会多次拦截此消息。 关于多个断开事件，组件应该是幂等的。

4.4.18 STOMP Client

Spring提供了一个基于WebSocket的STOMP客户端和一个基于TCP的STOMP客户端。

首先，您可以创建和配置WebSocketStompClient，如以下示例所示：

```
WebSocketClient webSocketClient = new StandardWebSocketClient();
WebSocketStompClient stompClient = new WebSocketStompClient(webSocketClient);
stompClient.setMessageConverter(new StringMessageConverter());
stompClient.setTaskScheduler(taskScheduler); // for heartbeats
```

在前面的示例中，您可以用SockJsClient替换StandardWebSocketClient，因为这也是WebSocketClient的实现。SockJsClient可以使用基于WebSocket或HTTP的传输作为后备。 有关更多详细信息，请参见SockJsClient。

接下来，您可以建立连接并为STOMP会话提供处理程序，如以下示例所示：

```
String url = "ws://127.0.0.1:8080/endpoint";
StompSessionHandler sessionHandler = new MyStompSessionHandler();
stompClient.connect(url, sessionHandler);
```

会话准备就绪后，将通知处理程序，如以下示例所示：

```
public class MyStompSessionHandler extends StompSessionHandlerAdapter {

    @Override
    public void afterConnected(StompSession session, StompHeaders connectedHeaders) {
        // ...
    }
}
```

一旦建立会话，就可以发送任何有效负载并使用配置的MessageConverter对其进行序列化，如以下示例所示：

```
session.send("/topic/something", "payload");
```

您还可以订阅目的地。 subscription方法需要一个用于订阅中消息的处理程序，并返回可用于取消订阅的Subscription句柄。 对于每个收到的消息，处理程序可以指定有效负载应反序列化的目标对象类型，如以下示例所示：

更多教程请访问:www.flydean.com

```
session.subscribe("/topic/something", new StompFrameHandler() {

    @Override
    public Type getPayloadType(StompHeaders headers) {
        return String.class;
    }

    @Override
    public void handleFrame(StompHeaders headers, Object payload) {
        // ...
    }

});
```

要启用STOMP心跳，您可以使用TaskScheduler配置WebSocketStompClient并有选择地自定义心跳间隔（写非活动状态为10秒，导致发送心跳，读非活动状态为10秒，关闭连接）。

> 当您使用WebSocketStompClient进行性能测试以模拟同一台计算机上的数千个客户端时，请考虑关闭心跳，因为每个连接都计划自己的心跳任务，并且并未针对在同一台计算机上运行的大量客户端进行优化。

STOMP协议还支持回执，在发送或订阅处理完毕后，客户端必须在其中添加一个回执标头，服务器以RECEIPT帧响应该回执标头。为此，StompSession提供了setAutoReceipt（boolean），它使收据头添加到每个后续的send或subscription事件上。或者，您也可以手动将收据标头添加到StompHeaders。发送和订阅都返回一个Receiptable实例，您可以使用该实例注册接收成功和失败的回调。要使用此功能，必须为客户端配置TaskScheduler以及收据过期之前的时间（默认为15秒）。

请注意，StompSessionHandler本身是一个StompFrameHandler，它除了处理来自消息处理的异常的handleException回调和处理包含ConnectionLostException的传输级错误的handleTransportError之外，还可以处理ERROR帧。

4.4.19 WebSocket Scope

每个WebSocket会话都有一个属性映射。 该映射作为标头附加到入站客户端消息，可以通过控制器方法进行访问，如以下示例所示：

```
@Controller
public class MyController {

    @MessageMapping("/action")
    public void handle(SimpMessageHeaderAccessor headerAccessor) {
        Map<String, Object> attrs = headerAccessor.getSessionAttributes();
        // ...
    }
}
```

您可以在websocket范围内声明一个Spring托管的bean。 您可以将WebSocket作用域的bean注入控制器以及在clientInboundChannel上注册的所有通道拦截器。 这些通常是单例，并且比任何单独的WebSocket会话寿命更长。 因此，您需要对作用域WebSocket的bean使用作用域代理模式，如以下示例所示：

```
@Component
@Scope(scopeName = "websocket", proxyMode = ScopedProxyMode.TARGET_CLASS)
public class MyBean {

    @PostConstruct
    public void init() {
        // Invoked after dependencies injected
    }

    // ...
```

更多教程请访问:www.flydean.com

```
    @PreDestroy
    public void destroy() {
        // Invoked when the WebSocket session ends
    }
}

@Controller
public class MyController {

    private final MyBean myBean;

    @Autowired
    public MyController(MyBean myBean) {
        this.myBean = myBean;
    }

    @MessageMapping("/action")
    public void handle() {
        // this.myBean from the current WebSocket session
    }
}
```

与任何自定义范围一样，Spring首次在控制器中对其进行访问时会初始化一个新的MyBean实例，并将该实例存储在WebSocket会话属性中。 随后将返回相同的实例，直到会话结束。 WebSocket范围的bean调用了所有Spring生命周期方法，如前面的示例所示。

4.4.20 Performance

关于性能，没有灵丹妙药。影响它的因素很多，包括消息的大小和数量，应用程序方法是否执行需要阻止的工作以及外部因素（例如网络速度和其他问题）。本部分的目的是提供可用配置选项的概述，以及有关如何进行扩展的一些想法。

在消息传递应用程序中，消息通过通道传递以进行异步执行，并由线程池支持。配置这样的应用程序需要对通道和消息流有充分的了解。因此，建议查看消息流。

最明显的开始是配置支持clientInboundChannel和clientOutboundChannel的线程池。默认情况下，两者都配置为可用处理器数量的两倍。

如果带注解的方法中的消息处理主要是受CPU限制的，则clientInboundChannel的线程数应保持接近处理器数。如果他们所做的工作更多地受到IO限制，并且需要阻塞或等待数据库或其他外部系统，则可能需要增加线程池大小。

> ThreadPoolExecutor具有三个重要属性：核心线程池大小，最大线程池大小，以及队列存储没有可用线程的任务的容量。
>
> 常见的混淆点是，配置核心池大小（例如10）和最大池大小（例如20）会导致线程池具有10到20个线程。实际上，如果将容量保留为其默认值Integer.MAX_VALUE，则由于所有其他任务都已排队，因此线程池永远不会超过核心池的大小。
>
> 请参阅ThreadPoolExecutor的javadoc，以了解这些属性如何工作并了解各种排队策略。

在clientOutboundChannel方面，所有关于向WebSocket客户端发送消息。如果客户端在快速网络上，则线程数应保持接近可用处理器数。如果它们很慢或带宽很低，它们将花费更长的时间来消耗消息并给线程池增加负担。因此，必须增加线程池的大小。

尽管clientInboundChannel的工作负载可以预测-毕竟，它是基于应用程序的工作-但是，如何配置“clientOutboundChannel”却比较困难，因为它基于应用程序无法控制的因素。因此，还有两个与消息发送有关的属性：sendTimeLimit和sendBufferSizeLimit。您可以使用这些方法来配置发送消息到客户端时允许发送多长时间以及可以缓冲多少数据。

更多教程请访问:www.flydean.com

通常的想法是，在任何给定时间，只能使用单个线程将其发送给客户端。同时，所有其他消息都将被缓冲，并且您可以使用这些属性来决定允许发送消息花费多长时间以及在此期间可以缓冲多少数据。有关其他重要信息，请参见XML模式的javadoc和文档。

以下示例显示了可能的配置：

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void configureWebSocketTransport(WebSocketTransportRegistration registration) {
        registration.setSendTimeLimit(15 * 1000).setSendBufferSizeLimit(512 * 1024);
    }

    // ...

}
```

下面的示例显示与前面的示例等效的XML配置：

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

    <websocket:message-broker>
        <websocket:transport send-timeout="15000" send-buffer-size="524288" />
        <!-- ... -->
    </websocket:message-broker>

</beans>
```

您还可以使用前面显示的WebSocket传输配置来配置传入STOMP消息的最大允许大小。 从理论上讲，WebSocket消息的大小几乎是无限的。 实际上，WebSocket服务器施加了限制，例如Tomcat 8K和Jetty 64K。 因此，STOMP客户端（例如JavaScript webstomp-client等）在16K边界处拆分较大的STOMP消息，并将其作为多个WebSocket消息发送，这需要服务器进行缓冲和重新组装。

Spring的STOMP-over-WebSocket支持可以做到这一点，因此应用程序可以为STOMP消息配置最大大小，而与WebSocket服务器特定的消息大小无关。 请记住，如有必要，将自动调整WebSocket消息的大小，以确保它们最多可以承载16K WebSocket消息。

以下示例显示了一种可能的配置：

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void configureWebSocketTransport(WebSocketTransportRegistration registration) {
        registration.setMessageSizeLimit(128 * 1024);
    }

    // ...

}
```

更多教程请访问:www.flydean.com

下面的示例显示与前面的示例等效的XML配置：

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

    <websocket:message-broker>
        <websocket:transport message-size="131072" />
        <!-- ... -->
    </websocket:message-broker>

</beans>
```

关于扩展的重要一点涉及使用多个应用程序实例。 当前，您无法使用简单代理执行此操作。 但是，当您使用功能齐全的代理（例如RabbitMQ）时，每个应用程序实例都连接到代理，并且从一个应用程序实例广播的消息可以通过代理广播到通过任何其他应用程序实例连接的WebSocket客户端。

4.4.22 Testing

使用Spring的STOMP-over-WebSocket支持时，有两种主要的方法来测试应用程序。首先是编写服务器端测试以验证控制器的功能及其带注解的消息处理方法。第二个是编写涉及运行客户端和服务器的完整的端到端测试。

两种方法不是互斥的。相反，每个人在整体测试策略中都有自己的位置。服务器端测试更加集中，更易于编写和维护。另一方面，端到端集成测试更完整，测试更多，但编写和维护也更加复杂。

服务器端测试的最简单形式是编写控制器单元测试。但是，这还不够有用，因为控制器所做的很多事情都取决于其注解。纯单元测试根本无法测试。

理想情况下，应该像在运行时那样调用被测控制器，就像使用Spring MVC Test框架测试处理HTTP请求的控制器的方法一样，即不运行Servlet容器而是依靠Spring框架来调用被测控制器。带注解的控制器。与Spring MVC Test一样，您有两种可能的选择，要么使用“基于上下文的”设置，要么使用“独立的”设置：

- 在Spring TestContext框架的帮助下加载实际的Spring配置，将clientInboundChannel注入为测试字段，并使用它发送消息以由控制器方法处理。
- 手动设置调用控制器（即SimpAnnotationMethodMessageHandler）并将控制器消息直接传递到它所需的最低Spring框架基础结构。

在股票投资组合样本应用程序的测试中演示了这两种设置方案。

第二种方法是创建端到端集成测试。为此，您需要以嵌入式模式运行WebSocket服务器，并将其作为WebSocket客户端连接到该服务器，该客户端发送包含STOMP帧的WebSocket消息。针对股票样本应用程序的测试还通过将Tomcat用作嵌入式WebSocket服务器和用于测试目的的简单STOMP客户端，演示了此方法。

更多教程请访问:www.flydean.com

Web Reactive

文档的此部分涵盖对基于Reactive Streams API构建的反应堆Web应用程序的支持，该应用程序可在非阻塞服务器（例如Netty，Undertow和Servlet 3.1+容器）上运行。 各个章节涵盖了Spring WebFlux框架，响应式WebClient，对测试的支持以及响应式库。 对于Servlet堆栈Web应用程序，请参阅Servlet堆栈上的Web。

更多教程请访问:www.flydean.com

1.Spring WebFlux

Spring框架中包含的原始Web框架Spring Web MVC是专门为Servlet API和Servlet容器而构建的。 反应性堆栈Web框架Spring WebFlux在更高版本5.0中添加。 它是完全非阻塞的，支持Reactive Streams背压，并在Netty，Undertow和Servlet 3.1+容器等服务器上运行。

这两个Web框架都反映了其源模块的名称（spring-webmvc和spring-webflux），并在Spring Framework中并存。 每个模块都是可选的。 应用程序可以使用一个模块或另一个模块，或者在某些情况下同时使用这两个模块，例如，带有响应式WebClient的Spring MVC控制器。

更多教程请访问:www.flydean.com

1.1 Overview

为什么创建Spring WebFlux?

答案的一部分是需要一个非阻塞式的Web堆栈来处理少量线程的并发并使用更少的硬件资源进行扩展。 Servlet 3.1确实提供了用于非阻塞I / O的API。但是，使用它会导致Servlet API的其余部分偏离，在Servlet API中，合同是同步的（Filter，Servlet）或阻塞的（getParameter，getPart）。这是促使新的通用API成为所有非阻塞运行时的基础的动机。这很重要，因为服务器（例如Netty）已在异步，非阻塞空间中建立良好。

答案的另一部分是函数式编程。就像在Java 5中添加注解会创造机会（例如带注解的REST控制器或单元测试）一样，在Java 8中添加lambda表达式也会为Java中的功能API创造机会。这对于无阻塞的应用程序和连续样式的API（如由CompletableFuture和ReactiveX普及的API）是有利的，这些API允许以声明方式构成异步逻辑。在编程模型级别，Java 8使Spring WebFlux能够与带注解的控制器一起提供功能性的Web端点。

1.1.1 定义“反应式”

我们谈到了“无障碍”和“功能性”，但是反应式意味着什么？

术语“反应性”是指围绕对更改做出反应的编程模型-网络组件对I / O事件做出反应，UI控制器对鼠标事件做出反应等。从这个意义上说，非阻塞是反应性的，因为随着操作完成或数据可用，我们现在处于响应通知的模式，而不是被阻塞。

我们Spring团队还有另一个重要机制与“反应性”相关联，这是不阻碍背压的机制。在同步命令式代码中，阻塞调用是强制调用者等待的一种自然的背压形式。在非阻塞代码中，控制事件的速率非常重要，这样快速的生产者就不会淹没其目的地。

Reactive Streams是一个小的规范（在Java 9中也采用了），它定义了带有反压力的异步组件之间的交互。例如，数据存储库（充当发布者）可以生成HTTP服务器（充当订阅者）然后可以将其写入响应的数据。 Reactive Streams的主要目的是让订阅者控制发布者生成数据的速度或速度。

> 常见问题：如果publisher不能放慢脚步怎么办？ 反应流的目的仅仅是建立机制和边界。 如果发布者无法放慢速度，则必须决定是缓冲，删除还是失败。

1.1.2 Reactive API

反应流对于互操作性起着重要作用。库和基础结构组件对此很感兴趣，但是由于它太底层了，它作为应用程序API的用处不大。应用程序需要更高级别且功能更丰富的API来构成异步逻辑，这与Java 8 Stream API相似，但不仅适用于集合。这就是反应式库的作用。

Reactor是Spring WebFlux的首选反应库。它提供了Mono和Flux API类型，以通过与ReactiveX运算符词汇对齐的丰富运算符集来处理0..1（Mono）和0..N（Flux）的数据序列。 Reactor是Reactive Streams库，因此，它的所有运算符都支持无阻塞背压。 Reactor非常注重服务器端Java。它是与Spring紧密合作开发的。

WebFlux需要Reactor作为核心依赖项，但是它可以通过Reactive Streams与其他React库进行互操作。通常，WebFlux API接受普通的发布者作为输入，在内部将其适应于Reactor类型，使用它，然后返回Flux或Mono作为输出。因此，您可以将任何发布服务器作为输入传递，并且可以对输出应用操作，但是您需要调整输出以与其他反应式库一起使用。只要可行（例如，带注解的控制器），WebFlux就会透明地适应RxJava或其他反应式库的使用。有关更多详细信息，请参见反应式库。

1.1.3 Programming Models

更多教程请访问:www.flydean.com

spring-web模块包含Spring WebFlux基础的反应式基础，包括HTTP抽象，用于支持的服务器的Reactive Streams适配器，编解码器以及与Servlet API相似但具有非阻塞合同的核心WebHandler API。

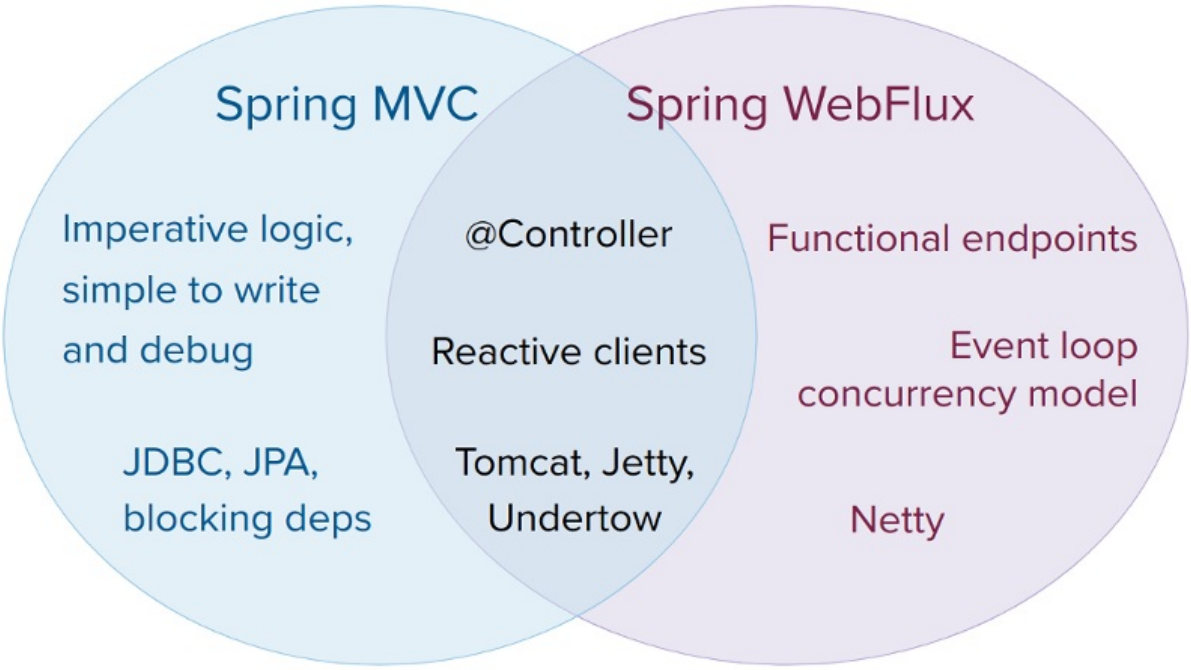
在此基础上，Spring WebFlux提供了两种编程模型的选择：

- 带注解的控制器：与Spring MVC一致，并基于来自spring-web模块的相同注解。 Spring MVC和WebFlux控制器都支持反应式（Reactor和RxJava）返回类型，因此，区分它们并不容易。一个显着的区别是WebFlux还支持反应式@RequestBody参数。
- 功能端点：基于Lambda的轻量级功能编程模型。您可以将其视为一个小型库或一组实用程序，应用程序可以使用它们来路由和处理请求。带注解的控制器的最大区别在于，应用程序负责从头到尾的请求处理，而不是通过注解声明意图并被回调。

1.1.4 Applicability

Spring MVC还是WebFlux?

一个自然的问题要问，但建立了一个不合理的二分法。 实际上，两者共同努力扩大了可用选项的范围。 两者的设计旨在实现彼此的连续性和一致性，它们可以并行使用，并且来自双方的反馈对双方都有利。 下图显示了两者之间的关系，它们的共同点以及各自的独特支持：



我们建议您考虑以下几点：

- 如果您有运行正常的Spring MVC应用程序，则无需更改。命令式编程是编写，理解和调试代码的最简单方法。您有最大的库选择空间，因为从历史上看，大多数库都处于阻塞状态。
- 如果您已经在购买无阻塞Web堆栈，那么Spring WebFlux在此空间中提供的执行模型优势与其他模型相同，并且还提供服务器选择（Netty，Tomcat，Jetty，Undertow和Servlet 3.1+容器），选择编程模型（带注解的控制器和功能性Web端点），以及选择反应式库（Reactor，RxJava或其他）。
- 如果您对与Java 8 lambda或Kotlin一起使用的轻量级功能性Web框架感兴趣，则可以使用Spring WebFlux功能性Web端点。对于要求较低复杂性的较小应用程序或微服务（可以受益于更高的透明度和控制）而言，这也是一个不错的选择。
- 在微服务架构中，您可以混合使用带有Spring MVC或Spring WebFlux控制器或带有Spring WebFlux功能端点的应用程序。在两个框架中都支持相同的基于注解的编程模型，这使得重用知识变得更加容易，同时还为正确的工作选择了正确的工具。
- 评估应用程序的一种简单方法是检查其依赖关系。如果您要使用阻塞性持久性API（JPA，JDBC）或网络API，则

更多教程请访问:www.flydean.com

Spring MVC至少是常见体系结构的最佳选择。使用Reactor和RxJava在单独的线程上执行阻塞调用在技术上是可行的，但您不会充分利用非阻塞Web堆栈。

- 如果您的Spring MVC应用程序具有对远程服务的调用，请尝试响应式WebClient。您可以直接从Spring MVC控制器方法返回反应类型（Reactor，RxJava或其他）。每个呼叫的等待时间或呼叫之间的相互依赖性越大，好处就越明显。 Spring MVC控制器也可以调用其他反应式组件。
- 如果您有庞大的团队，请牢记向无阻塞，功能性和声明性编程的过渡过程中的学习曲线陡峭。一种无需完全切换即可开始的实用方法是使用反应式WebClient。除此之外，从小处着手并衡量收益。我们希望，对于广泛的应用程序，这种转变是不必要的。如果不确定要寻找什么好处，请先了解无阻塞I / O的工作原理（例如，单线程Node.js上的并发性）及其影响。

1.1.5 Servers

Tomcat，Jetty，Servlet 3.1+容器以及非Servlet运行时（例如Netty和Undertow）都支持Spring WebFlux。所有服务器都适应于低级通用API，因此可以跨服务器支持更高级别的编程模型。

Spring WebFlux不具有内置支持来启动或停止服务器。但是，从Spring配置和WebFlux基础结构组装应用程序并用几行代码运行它很容易。

Spring Boot具有一个WebFlux启动器，可以自动执行这些步骤。默认情况下，入门者使用Netty，但通过更改Maven或Gradle依赖关系，可以轻松切换到Tomcat，Jetty或Undertow。 Spring Boot默认为Netty，因为它在异步，非阻塞空间中得到更广泛的使用，并允许客户端和服务器共享资源。

Tomcat和Jetty可以与Spring MVC和WebFlux一起使用。但是请记住，它们的使用方式非常不同。 Spring MVC依靠Servlet阻塞I / O，并允许应用程序在需要时直接使用Servlet API。 Spring WebFlux依赖于Servlet 3.1非阻塞I / O，并在低级适配器后面使用Servlet API，并且不公开供直接使用。

对于Undertow，Spring WebFlux直接使用Undertow API，而无需使用Servlet API。

1.1.6 Performance

Performance具有许多特征和意义。 反应和非阻塞通常不会使应用程序运行得更快。 在某些情况下，它们可以（例如，如果使用WebClient并行执行远程调用）。 总体而言，以非阻塞方式进行操作需要更多的工作，这可能会稍微增加所需的处理时间。

反应性和非阻塞性的主要预期好处是能够以较少的固定数量的线程和较少的内存进行扩展。 这使应用程序在负载下更具弹性，因为它们以更可预测的方式扩展。 但是，为了观察这些好处，您需要有一些延迟（包括缓慢的和不可预测的网络I / O）。 这就是反应堆开始显示其优势的地方，差异可能很大。

1.1.7 Concurrency Model

Spring MVC和Spring WebFlux都支持带注解的控制器，但是并发模型以及对阻塞和线程的默认假设存在关键差异。

在Spring MVC（通常是servlet应用程序）中，假定应用程序可以阻塞当前线程（例如，用于远程调用），因此，servlet容器使用大线程池来吸收请求期间的潜在阻塞。 处理。

在Spring WebFlux（通常是非阻塞服务器）中，假定应用程序未阻塞，因此，非阻塞服务器使用固定大小的小型线程池（事件循环工作器）来处理请求。

> “按比例缩放”和“少量线程”听起来可能是矛盾的，但是从不阻塞当前线程（而是依靠回调）意味着您不需要额外的线程，因为没有阻塞调用可以吸收。

调用阻止API

更多教程请访问:www.flydean.com

如果确实需要使用阻止库怎么办？ Reactor和RxJava都提供了publishOn运算符以继续在其他线程上进行处理。这意味着容易逃生。但是请记住，阻塞式API不适用于此并发模型。

可变状态

在Reactor和RxJava中，您可以通过运算符声明逻辑，然后在运行时形成一个反应式管道，在其中以不同的阶段顺序处理数据。这样做的主要好处是，它使应用程序不必保护可变状态，因为该管道中的应用程序代码永远不会被同时调用。

线程模型

您期望在运行Spring WebFlux的服务器上看到哪些线程?

- 在“原始” Spring WebFlux服务器上（例如，没有数据访问权限或其他可选依赖项），您可以期望该服务器有一个线程，而其他几个线程则可以进行请求处理（通常与CPU核心数量一样多）。但是，Servlet容器可能以更多的线程（例如，Tomcat上为10）开始，以同时支持servlet（阻塞）I / O和Servlet 3.1（非阻塞）I / O使用。
- 反应式WebClient以事件循环样式运行。因此，您可以看到与之相关的固定数量的处理线程（例如，带有Reactor Netty连接器的react-http-nio-）。但是，如果客户端和服务器都使用Reactor Netty，则默认情况下，两者共享事件循环资源。
- Reactor和RxJava提供称为调度程序的线程池抽象，以与publishOn运算符配合使用，该运算符用于将处理切换到其他线程池。调度程序具有建议特定并发策略的名称-例如，“并行”（对于具有有限数量的线程的CPU绑定工作）或“弹性”（对于具有大量线程的I / O绑定）。如果看到这样的线程，则意味着某些代码正在使用特定的线程池调度程序策略。
- 数据访问库和其他第三方依赖性也可以创建和使用自己的线程。

Configuring

Spring框架不提供启动和停止服务器的支持。要为服务器配置线程模型，您需要使用服务器特定的配置API，或者，如果您使用的是Spring Boot，请检查每个服务器的Spring Boot配置选项。您可以直接配置WebClient。对于所有其他库，请参阅其各自的文档。

更多教程请访问:www.flydean.com

1.2 Reactive Core

spring-web模块包含以下对反应式Web应用程序的基本支持：

对于服务器请求处理，有两个级别的支持。

- HttpHandler：使用非阻塞I / O和Reactive Streams背压进行HTTP请求处理的基本协定，以及Reactor Netty，Undertow，Tomcat，Jetty以及任何Servlet 3.1+容器的适配器。
- WebHandler API：稍高级别的通用Web API，用于处理请求，在此之上构建了具体的编程模型，例如带注解的控制器和功能端点。

对于客户端，有一个基本的ClientHttpConnector协定，以执行具有非阻塞I / O和响应流反压力的HTTP请求，以及用于Reactor Netty和响应式Jetty HttpClient的适配器。 应用程序中使用的更高级别的WebClient基于此基本协定。

对于客户端和服务器，编解码器用于序列化和反序列化HTTP请求和响应内容。

1.2.1 HttpHandler

HttpHandler是具有单个方法的简单协定，用于处理请求和响应。 它故意是最小的，它的主要也是唯一的目的是成为对不同HTTP服务器API的最小抽象。

下表描述了受支持的服务器API：

Server name	Server API used	Reactive Streams support
Netty	Netty API	Reactor Netty
Undertow	Undertow API	spring-web: Undertow to Reactive Streams bridge
Tomcat	Servlet 3.1 non-blocking I/O; Tomcat API to read and write ByteBuffers vs byte[]	spring-web: Servlet 3.1 non-blocking I/O to Reactive Streams bridge
Jetty	Servlet 3.1 non-blocking I/O; Jetty API to write ByteBuffers vs byte[]	spring-web: Servlet 3.1 non-blocking I/O to Reactive Streams bridge
Servlet 3.1 container	Servlet 3.1 non-blocking I/O	spring-web: Servlet 3.1 non-blocking I/O to Reactive Streams bridge

下表描述了服务器依赖性（另请参阅受支持的版本）：

Server name	Group id	Artifact name
Reactor Netty	io.projectreactor.netty	reactor-netty
Undertow	io.undertow	undertow-core
Tomcat	org.apache.tomcat.embed	tomcat-embed-core

下面的代码段显示了如何将HttpHandler适配器与每个服务器API一起使用：

Reactor Netty

```
HttpHandler handler = ...
ReactorHttpHandlerAdapter adapter = new ReactorHttpHandlerAdapter(handler);
HttpServer.create(host, port).newHandler(adapter).block();
```

更多教程请访问:www.flydean.com

Undertow

```
HttpHandler handler = ...
UndertowHttpHandlerAdapter adapter = new UndertowHttpHandlerAdapter(handler);
Undertow server = Undertow.builder().addHttpListener(port, host).setHandler(adapter).build();
server.start();
```

Tomcat

```
HttpHandler handler = ...
Servlet servlet = new TomcatHttpHandlerAdapter(handler);

Tomcat server = new Tomcat();
File base = new File(System.getProperty("java.io.tmpdir"));
Context rootContext = server.addContext("", base.getAbsolutePath());
Tomcat.addServlet(rootContext, "main", servlet);
rootContext.addServletMappingDecoded("/", "main");
server.setHost(host);
server.setPort(port);
server.start();
```

Jetty

```
HttpHandler handler = ...
Servlet servlet = new JettyHttpHandlerAdapter(handler);

Server server = new Server();
ServletContextHandler contextHandler = new ServletContextHandler(server, "");
contextHandler.addServlet(new ServletHolder(servlet), "/");
contextHandler.start();

ServerConnector connector = new ServerConnector(server);
connector.setHost(host);
connector.setPort(port);
server.addConnector(connector);
server.start();
```

Servlet 3.1+ Container

要将其作为WAR部署到任何Servlet 3.1+容器，您可以扩展WAR并将其包括在AbstractReactiveWebInitializer中。 该类使用ServletHttpHandlerAdapter包装HttpHandler并将其注册为Servlet。

1.2.2 WebHandler API

org.springframework.web.server包基于HttpHandler契约构建，以提供通用的Web API，以通过多个WebExceptionHandler，多个WebFilter和单个WebHandler组件的链来处理请求。 通过简单地指向自动检测组件的Spring ApplicationContext和/或通过向构建器注册组件，可以将该链与WebHttpHandlerBuilder放在一起。

尽管HttpHandler的目标很简单，即抽象化不同HTTP服务器的使用，但WebHandler API的目的是提供Web应用程序中常用的更广泛的功能集，例如：

- 具有属性的用户会话。
- 请求属性。
- 请求的解析的语言环境或主体。
- 访问已解析和缓存的表单数据。
- 多部分数据的抽象。

Special bean types

更多教程请访问:www.flydean.com

下表列出了WebHttpHandlerBuilder可以在Spring ApplicationContext中自动检测的组件，或者可以直接向其注册的组件：

Bean name	Bean type	Count	Description
<any>	WebExceptionHandler	0..N	提供对WebFilter实例链和目标 链 异常的处理WebHandler。有关更 细信息，请参见Exceptions。
<any>	WebFilter	0..N	在其余过滤链和目标之前和之后应 拦截式逻辑WebHandler。有关更 细信息，请参见过滤器。
webHandler	WebHandler	1	请求的处理程序。
webSessionManager	WebSessionManager	0..1	WebSession通过上的方法公开的 的管理器ServerWebExchange。 DefaultWebSessionManager默认
serverCodecConfigurer	ServerCodecConfigurer	0..1	用于访问HttpMessageReader实例 解析表单数据和多部分数据，然后 过上的方法公开这些数据 ServerWebExchange。 ServerCodecConfigurer.create()默
localeContextResolver	LocaleContextResolver	0..1	LocaleContext通过在上的方法公开 解析器ServerWebExchange。 AcceptHeaderLocaleContextReso 默认。
forwardedHeaderTransformer	ForwardedHeaderTransformer	0..1	对于处理转发的类型标头，可以通 提取和删除它们或仅通过删除它们 进行。默认不使用。

表格数据

ServerWebExchange 公开以下访问表单数据的方法：

```
Mono<MultiValueMap<String, String>> getFormData();
```

DefaultServerWebExchange使用配置的HttpMessageReader将表单数据（application / x-www-form-urlencoded）解析为MultiValueMap。 默认情况下，FormHttpMessageReader配置为由ServerCodecConfigurer Bean使用（请参阅Web Handler API）。

Multipart Data

ServerWebExchange公开以下访问多部分数据的方法：

```
Mono <MultiValueMap <String, Part >> getMultipartData () ;
```

DefaultServerWebExchange使用配置的HttpMessageReader <MultiValueMap <String，Part >>将multipart / form-data内容解析为MultiValueMap。 当前，Synchronoss NIO Multipart是唯一受支持的第三方库，并且是我们知道的用于非阻塞解析多部分请求的唯一库。 通过ServerCodecConfigurer bean启用它（请参阅Web Handler API）。

要以流方式解析多部分数据，可以使用从HttpMessageReader <Part>返回的Flux <Part>。 例如，在带注解的控制器中，使用@RequestPart意味着按名称对各个部分进行类似于Map的访问，因此需要完全解析多部分数据。 相反，您可以使用@RequestBody将内容解码为Flux <Part>而不收集到MultiValueMap。

Forwarded Headers

当请求通过代理（例如负载平衡器）进行处理时，主机，端口和方案可能会更改，这使得从客户端角度创建指向正确的主机，端口和方案的链接带来了挑战。

更多教程请访问:www.flydean.com

RFC 7239定义了代理可以用来提供有关原始请求的信息的HTTP转发头。还有其他非标准标头，包括X-Forwarded-Host，X-Forwarded-Port，X-Forwarded-Proto，X-Forwarded-Ssl和X-Forwarded-Prefix。

ForwardedHeaderFilter是一个Servlet过滤器，它根据转发的标头修改请求的主机，端口和方案，然后删除这些标头。

对于转发的标头，存在安全方面的考虑，因为应用程序无法知道标头是由代理添加的，还是由恶意客户端添加的。这就是为什么应配置信任边界处的代理以删除来自外部的不受信任的转发标头的原因。您还可以使用removeOnly = true配置ForwardedHeaderFilter，在这种情况下，它将删除但不使用标头。

> 在5.1版本中，ForwardedHeaderFilter被ForwardedHeaderTransformer弃用并取代，因此可以在创建交换之前更早地处理转发的标头。 如果仍然配置了过滤器，则将其从过滤器列表中删除，而改用ForwardedHeaderTransformer。

1.2.3 Filters

在WebHandler API中，可以使用WebFilter在其余过滤器和目标WebHandler的其余处理链之前和之后应用拦截样式的逻辑。 使用WebFlux Config时，注册WebFilter就像将其声明为Spring bean一样简单，并且（可选）通过在bean声明上使用@Order或实现Ordered来表达优先级。

CORS

Spring WebFlux通过控制器上的注解为CORS配置提供了细粒度的支持。 但是，当您将其与Spring Security结合使用时，我们建议您依赖内置的CorsFilter，该产品必须在Spring Security的过滤器链之前订购。

有关更多详细信息，请参见有关CORS和CORS WebFilter的部分。

1.2.4 Exceptions

在WebHandler API中，可以使用WebExceptionHandler来处理WebFilter实例链和目标WebHandler链中的异常。 使用WebFlux Config时，注册WebExceptionHandler就像将其声明为Spring bean一样简单，并且（可选）通过在bean声明上使用@Order或实现Ordered来表达优先级。

下表描述了可用的WebExceptionHandler实现：

Exception Handler	Description
ResponseStatusExceptionHandler	通过将响应设置为异常的HTTP状态代码，提供对ResponseStatusException类型的异常的处理。
WebFluxResponseStatusExceptionHandler	ResponseStatusExceptionHandler的扩展，它也可以确定任何异常时@ResponseStatus批注的HTTP状态代码。该处理程序在WebFlux Config中声明。

1.2.5 Codecs

spring-web和spring-core模块提供支持，通过具有Reactive Streams背压的非阻塞I / O，可以将字节内容与更高级别的对象之间的字节序列进行序列化和反序列化。以下介绍了此支持：

- 编码器和解码器是底层协议，用于独立于HTTP编码和解码内容。
- HttpMessageReader和HttpMessageWriter是对HTTP消息内容进行编码和解码的协定。
- 可以使用EncoderHttpMessageWriter来包装Encoder，以使其适合在Web应用程序中使用，而可以使用DecoderHttpMessageReader来包装Decoder。
- DataBuffer抽象了不同的字节缓冲区表示形式（例如Netty ByteBuf，java.nio.ByteBuffer等），并且是所有编解码器都在处理的内容。有关此主题的更多信息，请参见“ Spring核心”部分中的数据缓冲区和编解码器。

更多教程请访问:www.flydean.com

spring-core模块提供byte []，ByteBuffer，DataBuffer，Resource和String编码器和解码器实现。 spring-web模块提供了Jackson JSON，Jackson Smile，JAXB2，Protocol Buffers和其他编码器和解码器，以及仅Web的HTTP消息读取器和写入器实现，用于表单数据，多部分内容，服务器发送的事件等。

ClientCodecConfigurer和ServerCodecConfigurer通常用于配置和自定义要在应用程序中使用的编解码器。请参阅有关配置HTTP消息编解码器的部分。

Jackson JSON

存在Jackson库时，都支持JSON和二进制JSON（Smile）。

Jackson2Decoder的工作方式如下：

- Jackson的异步，非阻塞解析器用于将字节块流聚合到TokenBuffer的每个块中，每个代表JSON对象。
- 每个TokenBuffer都传递给Jackson的ObjectMapper以创建更高级别的对象。
- 解码为单值发布者（例如Mono）时，有一个TokenBuffer。
- 当解码为多值发布者（例如Flux）时，一旦为一个完整的对象接收到足够的字节，每个TokenBuffer就会传递给ObjectMapper。输入内容可以是JSON数组，如果内容类型为“ application / stream + json”，则可以是行分隔的JSON。

Jackson2Encoder的工作方式如下：

- 对于单个值发布者（例如Mono），只需通过ObjectMapper对其进行序列化即可。
- 对于具有“ application / json”的多值发布者，默认情况下使用Flux # collectToList（）收集值，然后序列化结果集合。
- 对于具有流媒体类型（例如application / stream + json或application / stream + x-jackson-smile）的多值发布者，请使用行分隔的JSON格式分别编码，写入和刷新每个值。
- 对于SSE，将为每个事件调用Jackson2Encoder，并刷新输出以确保交付没有延迟。

> 默认情况下，Jackson2Encoder和Jackson2Decoder都不支持String类型的元素。 相反，默认假设是一个字符串或一系列字符串表示要由CharSequenceEncoder呈现的序列化JSON内容。 如果您需要从Flux <String>呈现JSON数组，请使用Flux # collectToList（）并对Mono <List <String >>进行编码。

Form Data

FormHttpMessageReader和FormHttpMessageWriter支持对“ application / x-www-form-urlencoded”内容进行解码和编码。

在经常需要从多个位置访问表单内容的服务器端，ServerWebExchange提供了专用的getFormData（）方法，该方法通过FormHttpMessageReader解析内容，然后缓存结果以进行重复访问。请参阅WebHandler API部分中的表单数据。

一旦使用getFormData（），就无法再从请求正文中读取原始原始内容。因此，应用程序应始终通过ServerWebExchange来访问缓存的表单数据，而不是从原始请求正文中进行读取。

Multipart

MultipartHttpMessageReader和MultipartHttpMessageWriter支持对“ multipart / form-data”内容进行解码和编码。反过来，MultipartHttpMessageReader委托给另一个HttpMessageReader进行实际解析为Flux <Part>，然后将这些部分简单地收集到MultiValueMap中。目前，Synchronoss NIO Multipart用于实际解析。

在可能需要从多个位置访问多部分表单内容的服务器端，ServerWebExchange提供了专用的getMultipartData（）方法，该方法通过MultipartHttpMessageReader解析内容，然后缓存结果以进行重复访问。请参阅WebHandler API部分中的多部分数据。

一旦使用getMultipartData（），就无法再从请求正文中读取原始原始内容。因此，应用程序必须始终使用getMultipartData（）来重复，类似地图地访问零件，否则必须依靠SynchronossPartHttpMessageReader来一次性访问Flux <Part>。

Streaming

更多教程请访问:www.flydean.com

在流式传输到HTTP响应（例如text / event-stream，application / stream + json）时，定期发送数据很重要，这样才能尽快（而不是稍后）可靠地检测到断开连接的客户端。这样的发送可以是仅注解，空的SSE事件或任何其他可以有效充当心跳的“无操作”数据。

DataBuffer

DataBuffer是WebFlux中字节缓冲区的表示形式。该参考书的Spring Core部分在有关数据缓冲区和编解码器的部分中有更多内容。要理解的关键点是，在诸如Netty之类的某些服务器上，字节缓冲区被池化并且对引用计数进行计数，并且在消耗字节缓冲区时必须将其释放以避免内存泄漏。

WebFlux应用程序通常不需要关心此类问题，除非它们直接使用或产生数据缓冲区，而不是依赖于编解码器与更高级别的对象进行转换。或者，除非他们选择创建自定义编解码器。对于这种情况，请查看“数据缓冲区和编解码器”中的信息，尤其是有关“使用数据缓冲区”的部分。

1.2.6 Logging

Spring WebFlux中的DEBUG级别日志记录被设计为紧凑，最小化和人性化。它侧重于一遍又一遍有用的高价值信息，而其他信息则仅在调试特定问题时才有用。

TRACE级别的日志记录通常遵循与DEBUG相同的原则（例如，也不应成为firehose），但可用于调试任何问题。另外，某些日志消息在TRACE vs DEBUG上可能显示不同级别的详细信息。

良好的日志记录来自使用日志的经验。如果您发现任何不符合既定目标的东西，请告诉我们。

Log Id

在WebFlux中，单个请求可以在多个线程上执行，并且线程ID对于关联属于特定请求的日志消息没有用。这就是为什么WebFlux日志消息默认情况下带有特定于请求的ID的原因。

在服务器端，日志ID存储在ServerWebExchange属性（LOG_ID_ATTRIBUTE）中，而可从ServerWebExchange # getLogPrefix（）获得基于该ID的全格式前缀。在WebClient端，日志ID存储在ClientRequest属性（LOG_ID_ATTRIBUTE）中，而完全格式的前缀可从ClientRequest # logPrefix（）获得。

Sensitive Data

调试和跟踪日志记录可以记录敏感信息。这就是默认情况下屏蔽表单参数和标题的原因，并且必须显式启用它们的完整日志记录。

以下示例显示了如何对服务器端请求执行此操作：

```
@Configuration
@EnableWebFlux
class MyConfig implements WebFluxConfigurer {

    @Override
    public void configureHttpMessageCodecs(ServerCodecConfigurer configurer) {
        configurer.defaultCodecs().enableLoggingRequestDetails(true);
    }
}
```

以下示例显示了如何针对客户端请求执行此操作：

```
Consumer<ClientCodecConfigurer> consumer = configurer ->
        configurer.defaultCodecs().enableLoggingRequestDetails(true);

WebClient webClient = WebClient.builder()
        .exchangeStrategies(ExchangeStrategies.builder().codecs(consumer).build())
        .build();
```

更多教程请访问:www.flydean.com

更多教程请访问:www.flydean.com

1.3 DispatcherHandler

Spring WebFlux与Spring MVC类似，是围绕前端控制器模式设计的，其中中央WebHandler DispatcherHandler提供了用于请求处理的共享算法，而实际工作是由可配置的委托组件执行的。该模型非常灵活，并支持多种工作流程。

DispatcherHandler从Spring配置中发现所需的委托组件。它还被设计为Spring Bean本身，并实现ApplicationContextAware来访问其运行的上下文。如果以WebHandler的bean名称声明了DispatcherHandler，则WebHttpHandlerBuilder会发现它，而WebHttpHandlerBuilder会将请求处理链组合在一起，如WebHandler API中所述。

WebFlux应用程序中的Spring配置通常包含：

- Bean名称为webHandler的DispatcherHandler
- WebFilter和WebExceptionHandler bean
- DispatcherHandler special beans

将配置提供给WebHttpHandlerBuilder以构建处理链，如以下示例所示：

```
ApplicationContext context = ...
HttpHandler handler = WebHttpHandlerBuilder.applicationContext(context);
```

生成的HttpHandler已准备好与服务器适配器一起使用。

1.3.1 Special Bean Types

Bean type	Explanation
HandlerMapping	将请求映射到处理程序。 映射基于某些条件，这些条件的详细信息因HandlerMapping实现而有所不同-带有注解的控制器，简单的URL模式映射以及其他。主要的HandlerMapping实现是用于@RequestMapping注解方法的RequestMappingHandlerMapping，用于功能端点路由的RouterFunctionMapping和用于URI路径模式和WebHandler实例的显式注册的SimpleUrlHandlerMapping。
HandlerAdapter	帮助DispatcherHandler调用映射到请求的处理程序，而不管实际如何调用该处理程序。 例如，调用带注解的控制器需要解析注解。 HandlerAdapter的主要目的是使DispatcherHandler免受此类细节的影响。
HandlerResultHandler	处理来自处理程序调用的结果，并最终确定响应。 请参阅结果处理。

1.3.2 WebFlux Config

应用程序可以声明处理请求所需的基础结构bean（在Web Handler API和DispatcherHandler下列出）。 但是，在大多数情况下，WebFlux Config是最佳起点。 它声明所需的bean，并提供更高级别的配置回调API对其进行自定义。

> Spring Boot依靠WebFlux配置来配置Spring WebFlux，并且还提供了许多额外的方便选项。

1.3.3 Processing

DispatcherHandler处理请求的方式如下：

- 要求每个HandlerMapping查找一个匹配的处理程序，并使用第一个匹配项。
- 如果找到处理程序，则通过适当的HandlerAdapter执行该处理程序，该处理程序将执行的返回值公开为HandlerResult。

更多教程请访问:www.flydean.com

- 通过直接写入响应或使用视图渲染，将HandlerResult提供给适当的HandlerResultHandler以完成处理。

1.3.4 Result Handling

通过HandlerAdapter调用处理程序的返回值连同其他一些上下文一起包装为HandlerResult，并传递给第一个支持该处理程序的HandlerResultHandler。 下表显示了可用的HandlerResultHandler实现，所有实现都在WebFlux Config中声明：

Result Handler Type	Return Values	Default Order
ResponseEntityResultHandler	ResponseEntity, typically from @Controller instances.	0
ServerResponseResultHandler	ServerResponse, typically from functional endpoints.	0
ResponseBodyResultHandler	Handle return values from @ResponseBody methods or @RestController classes.	100
ViewResolutionResultHandler	CharSequence, View, Model, Map, Rendering, or any other Object is treated as a model attribute.	Integer.MAX_VALUE

1.3.5 Exceptions

从HandlerAdapter返回的HandlerResult可以公开基于某些特定于处理程序的机制进行错误处理的函数。 在以下情况下将调用此错误函数：

- 处理程序（例如，@ Controller）调用失败。
- 通过HandlerResultHandler处理处理程序返回值失败。

只要在从处理程序返回的反应类型产生任何数据项之前发生错误信号，错误函数就可以更改响应（例如，更改为错误状态）。

这就是支持@Controller类中的@ExceptionHandler方法的方式。 相比之下，Spring MVC中对相同功能的支持建立在HandlerExceptionResolver之上。 这通常不重要。 但是，请记住，在WebFlux中，不能使用@ControllerAdvice来处理在选择处理程序之前发生的异常。

另请参见“带注解的控制器”部分中的“管理异常”或“ WebHandler API”部分中的“异常”。

1.3.6 View Resolution

View Resolution使您可以使用HTML模板和模型渲染到浏览器，而无需将您与特定的视图技术联系在一起。 在Spring WebFlux中，通过专用的HandlerResultHandler支持视图解析，该HandlerResultHandler使用ViewResolver实例将String（表示逻辑视图名称）映射到View实例。 然后使用View呈现响应。

Handling

传递给ViewResolutionResultHandler的HandlerResult包含处理程序的返回值和包含请求处理期间添加的属性的模型。 返回值将作为以下值之一进行处理：

- 字符串，CharSequence：通过配置的ViewResolver实现列表解析为View的逻辑视图名称。
- void：根据请求路径选择默认视图名称，减去前导斜线和尾部斜杠，然后将其解析为视图。当未提供视图名称（例如，返回模型属性）或异步返回值（例如，Mono完成为空）时，也会发生同样的情况。
- Rendering：用于视图分辨率方案的API。通过代码完成探索IDE中的选项。
- Model, Map：额外的模型属性将添加到请求的模型中。

更多教程请访问:www.flydean.com

- Any other：任何其他返回值（由BeanUtils # isSimpleProperty确定的简单类型除外）都将被视为要添加到模型的模型属性。属性名称是通过使用约定从类名称派生的，除非存在处理程序方法@ModelAttribute批注。

该模型可以包含异步，反应式类型（例如，来自Reactor或RxJava）。在渲染之前，AbstractView将此类模型属性解析为具体值并更新模型。单值反应类型被解析为单个值或无值（如果为空），而多值反应类型（例如Flux <T>）被收集并解析为List <T>。

配置视图分辨率就像在Spring配置中添加ViewResolutionResultHandler bean一样简单。 WebFlux Config提供了专用于视图分辨率的配置API。

有关与Spring WebFlux集成的视图技术的更多信息，请参见View Technologies。

Redirecting

视图名称中的特殊redirect：前缀使您可以执行重定向。 UrlBasedViewResolver（和子类）将其识别为需要重定向的指令。 视图名称的其余部分是重定向URL。

最终效果与控制器已返回RedirectView或Rendering.redirectTo（“ abc”）。build（）相同，但是现在控制器本身可以根据逻辑视图名称进行操作。 视图名称（例如redirect：/ some / resource）是相对于当前应用程序的，而视图名称（例如redirect：https：//example.com/arbitrary/path）将重定向到绝对URL。

Content Negotiation

ViewResolutionResultHandler支持内容协商。 它将请求媒体类型与每个选定视图支持的媒体类型进行比较。 使用支持请求的媒体类型的第一个视图。

为了支持JSON和XML之类的媒体类型，Spring WebFlux提供了HttpMessageWriterView，这是一个通过HttpMessageWriter呈现的特殊视图。 通常，您可以通过WebFlux配置将其配置为默认视图。 如果默认视图与请求的媒体类型匹配，则始终会选择和使用它们。

更多教程请访问:www.flydean.com

1.4 Annotated Controllers

Spring WebFlux提供了一个基于注解的编程模型，其中@Controller和@RestController组件使用注解来表达请求映射，请求输入，处理异常等。 带注解的控制器具有灵活的方法签名，无需扩展基类或实现特定的接口。

以下清单显示了一个基本示例：

```
@RestController
public class HelloController {

    @GetMapping("/hello")
    public String handle() {
        return "Hello WebFlux";
    }
}
```

1.4.1 @Controller

您可以使用标准的Spring bean定义来定义控制器bean。 @Controller构造型允许自动检测，并且与Spring常规支持保持一致，以支持在类路径中检测@Component类并为其自动注册Bean定义。 它还充当带注解类的构造型，表明其作为Web组件的作用。

要启用对此类@Controller bean的自动检测，可以将组件扫描添加到Java配置中，如以下示例所示：

```
@Configuration
@ComponentScan("org.example.web")
public class WebConfig {

    // ...
}
```

@RestController是一个组合式注解，其本身使用@Controller和@ResponseBody进行了元注解，表示一个控制器，其每个方法都继承了类型级别的@ResponseBody注解，因此直接将其写入响应主体（与视图分辨率相对）并使用 HTML 模板。

1.4.2 Request Mapping

您可以使用@RequestMapping批注将请求映射到控制器方法。 它具有各种属性，可以通过URL，HTTP方法，请求参数，标头和媒体类型进行匹配。 您可以在类级别使用它来表示共享的映射，也可以在方法级别使用它来缩小到特定的端点映射。

@RequestMapping还有HTTP方法特定的快捷方式：

- @GetMapping
- @PostMapping
- @PutMapping
- @DeleteMapping
- @PatchMapping

快捷方式是提供的“自定义注解”，因为可以说，大多数控制器方法应该映射到特定的HTTP方法，而不是使用@RequestMapping，后者默认情况下与所有HTTP方法匹配。 同时，在类级别仍需要@RequestMapping来表示共享映射。

更多教程请访问:www.flydean.com

以下示例具有类型和方法级别的映射：

```
@RestController
@RequestMapping("/persons")
class PersonController {

    @GetMapping("/{id}")
    public Person getPerson(@PathVariable Long id) {
        // ...
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public void add(@RequestBody Person person) {
        // ...
    }
}
```

URI patterns

您可以使用以下全局模式和通配符来映射请求：

- ？ 匹配一个字符
- *匹配路径段中的零个或多个字符
- **匹配零个或多个路径段

您还可以声明URI变量并使用@PathVariable访问其值，如以下示例所示：

```
@GetMapping("/owners/{ownerId}/pets/{petId}")
public Pet findPet(@PathVariable Long ownerId, @PathVariable Long petId) {
    // ...
}
```

您可以在类和方法级别声明URI变量，如以下示例所示：

```
@Controller
@RequestMapping("/owners/{ownerId}")
public class OwnerController {

    @GetMapping("/pets/{petId}")
    public Pet findPet(@PathVariable Long ownerId, @PathVariable Long petId) {
        // ...
    }
}
```

URI变量会自动转换为适当的类型，或者引发TypeMismatchException。 默认情况下支持简单类型（int，long，Date等），您可以注册对任何其他数据类型的支持。 请参阅类型转换和DataBinder。

您可以显式地命名URI变量（例如，@PathVariable（“ customId”）），但是如果名称相同并且您的代码是使用调试信息或Java 8上的-parameters编译器标志进行编译的，则可以省略该详细信息。 。

语法{varName：regex}声明带有正则表达式的URI变量，语法为{varName：regex}。 例如，给定URL“ /spring-web-3.0.5 .jar”，以下方法提取名称，版本和文件扩展名：

```
@GetMapping("/{name:[a-z-]+}-{version:\\d\\.\\d\\.\\d}{ext:\\.[a-z]+}")
public void handle(@PathVariable String version, @PathVariable String ext) {
    // ...
}
```

更多教程请访问:www.flydean.com

URI路径模式也可以嵌入$ {...}占位符，这些占位符在启动时通过针对本地，系统，环境和其他属性源使用PropertyPlaceHolderConfigurer进行解析。 例如，您可以使用它来基于一些外部配置参数化基本URL。

> Spring WebFlux使用PathPattern和PathPatternParser获得URI路径匹配支持。 这两个类都位于spring-web中，并专门设计用于在运行时匹配大量URI路径模式的Web应用程序中的HTTP URL路径。

Spring WebFlux不支持后缀模式匹配，这与Spring MVC不同，后者的映射（例如/ person）也匹配到/person.*。 对于基于URL的内容协商，如果需要，我们建议使用查询参数，该参数更简单，更明确，并且不易受到基于URL路径的攻击。

Pattern Comparison

当多个模式与URL匹配时，必须将它们进行比较以找到最佳匹配。 这是通过PathPattern.SPECIFICITY_COMPARATOR完成的，该工具查找更具体的模式。

对于每个模式，都会根据URI变量和通配符的数量计算得分，其中URI变量的得分低于通配符。 总得分较低的模式将获胜。 如果两个模式的分数相同，则选择更长的时间。

包罗万象的模式（例如*，{varName}）不计入评分，而是始终排在最后。 如果两种模式都适用，则选择更长的时间。

Consumable Media Types

您可以根据请求的Content-Type缩小请求映射，如以下示例所示：

```
@PostMapping(path = "/pets", consumes = "application/json")
public void addPet(@RequestBody Pet pet) {
    // ...
}
```

消耗属性还支持否定表达式-例如，！text / plain表示除text / plain之外的任何内容类型。

您可以在类级别上声明一个共享的消耗属性。 但是，与大多数其他请求映射属性不同，在类级别使用方法级别时，方法级别使用属性覆盖而不是扩展类级别声明。

> MediaType提供常用媒体类型的常量，例如APPLICATION_JSON_VALUE和APPLICATION_XML_VALUE。

Producible Media Types

您可以根据接受请求标头和控制器方法生成的内容类型列表来缩小请求映射，如以下示例所示：

```
@GetMapping(path = "/pets/{petId}", produces = "application/json;charset=UTF-8")
@ResponseBody
public Pet getPet(@PathVariable String petId) {
    // ...
}
```

媒体类型可以指定字符集。 支持否定的表达式-例如，!text/plain表示除“ text / plain”以外的任何内容类型。

> 对于JSON内容类型，即使RFC7159明确声明“未为此注册定义任何字符集参数”，也应指定UTF-8字符集，因为某些浏览器要求它正确解释UTF-8特殊字符。

您可以在类级别声明共享的Produces属性。 但是，与大多数其他请求映射属性不同，在类级别使用时，方法级别会产生属性覆盖，而不是扩展类级别的声明。

> MediaType为常用的媒体类型（例如APPLICATION_JSON_UTF8_VALUE和APPLICATION_XML_VALUE）提供常数。

Parameters, headers

您可以根据请求参数条件来缩小请求映射。 您可以测试是否存在请求参数（myParam），是否存在一个请求参数（!myParam）或特定值（myParam = myValue）。 以下示例显示如何测试特定值：

更多教程请访问:www.flydean.com

```
@GetMapping(path = "/pets/{petId}", params = "myParam=myValue")
public void findPet(@PathVariable String petId) {
    // ...
}
```

您还可以将其与请求标头条件一起使用，如以下示例所示：

```
@GetMapping(path = "/pets", headers = "myHeader=myValue")
public void findPet(@PathVariable String petId) {
    // ...
}
```

HTTP HEAD, OPTIONS

@GetMapping（和@RequestMapping（method = HttpMethod.GET））透明地支持HTTP HEAD以进行请求映射。控制器方法不需要更改。应用于javax.servlet.http.HttpServlet的响应包装器确保将Content-Length标头设置为写入的字节数（实际上未写入响应）。

@GetMapping（和@RequestMapping（method = HttpMethod.GET））被隐式映射到并支持HTTP HEAD。像处理HTTP GET一样处理HTTP HEAD请求，不同之处在于，不是写入正文，而是计算字节数并设置Content-Length头。

默认情况下，通过将“允许响应”标头设置为所有具有匹配URL模式的@RequestMapping方法中列出的HTTP方法列表来处理HTTP OPTIONS。

对于没有HTTP方法声明的@RequestMapping，将Allow标头设置为GET，HEAD，POST，PUT，PATCH，DELETE，OPTIONS。控制器方法应始终声明支持的HTTP方法（例如，通过使用HTTP方法特定的变体：@ GetMapping，@ PostMapping等）。

您可以将@RequestMapping方法显式映射到HTTP HEAD和HTTP OPTIONS，但这在通常情况下不是必需的。

Custom Annotations

Spring MVC支持将组合注解用于请求映射。 这些注解本身使用@RequestMapping进行元注解，并且旨在以更狭窄，更具体的用途重新声明@RequestMapping属性的子集（或全部）。

@ GetMapping，@ PostMapping，@ PutMapping，@ DeleteMapping和@PatchMapping是组合注解的示例。 之所以提供它们，是因为可以说，大多数控制器方法应该映射到特定的HTTP方法，而不是使用@RequestMapping，后者默认情况下与所有HTTP方法都匹配。 如果需要组合注解的示例，请查看如何声明它们。

Spring MVC还支持带有自定义请求匹配逻辑的自定义请求映射属性。 这是一个更高级的选项，它需要子类化RequestMappingHandlerMapping并覆盖getCustomMethodCondition方法，您可以在其中检查自定义属性并返回自己的RequestCondition。

Explicit Registrations

您可以以编程方式注册处理程序方法，这些方法可用于动态注册或高级案例，例如同一处理程序在不同URL下的不同实例。 下面的示例注册一个处理程序方法：

```
@Configuration
public class MyConfig {

    @Autowired
    public void setHandlerMapping(RequestMappingHandlerMapping mapping, UserHandler handler)
            throws NoSuchMethodException {

        RequestMappingInfo info = RequestMappingInfo
                .paths("/user/{id}").methods(RequestMethod.GET).build();

        Method method = UserHandler.class.getMethod("getUser", Long.class);
```

更多教程请访问:www.flydean.com

```
        mapping.registerMapping(info, handler, method);
    }

}
```

1.4.3 Handler Methods

@RequestMapping处理程序方法具有灵活的签名，可以从一系列受支持的控制器方法参数和返回值中进行选择。

方法参数

下表描述了受支持的控制器方法参数。 任何参数均不支持反应性类型。

支持JDK 8的java.util.Optional作为方法参数，并与具有必需属性（例如，@ RequestParam，@ RequestHeader等）的注解结合在一起，等效于required = false。

Controller method argument	Description
WebRequest, NativeWebRequest	通用访问请求参数以及请求和会话属性，而无需直接使用Servlet API。
javax.servlet.ServletRequest, javax.servlet.ServletResponse	选择任何特定的请求或响应类型，例如ServletRequest，HttpServletRequest或Spring的MultipartRequest，MultipartHttpServletRequest。
javax.servlet.http.HttpSession	强制会话的存在。 结果，这样的参数永远不会为空。 请注意，会话访问不是线程安全的。 如果允许多个请求并发访问会话，请考虑将RequestMappingHandlerAdapter实例的syncnizeOnSession标志设置为true。
javax.servlet.http.PushBuilder	用于程序化HTTP / 2资源推送的Servlet 4.0推送构建器API。 请注意，根据Servlet规范，如果客户端不支持HTTP / 2功能，则注入的PushBuilder实例可以为null。
java.security.Principal	当前经过身份验证的用户-可能是特定的Principal实现类（如果已知）。
HttpMethod	请求的HTTP方法
java.util.Locale	当前的请求区域设置，由可用的最特定的LocaleResolver（实际上是配置的LocaleResolver或LocaleContextResolver）确定。
java.util.TimeZone + java.time.ZoneId	与当前请求关联的时区，由LocaleContextResolver确定。
java.io.InputStream, java.io.Reader	用于访问Servlet API公开的原始请求正文。
java.io.OutputStream, java.io.Writer	用于访问Servlet API公开的原始响应正文。
@PathVariable	用于访问URI模板变量。
@MatrixVariable	用于访问URI路径段中的名称/值对
@RequestParam	用于访问Servlet请求参数，包括多部分文件。 参数值将转换为声明的方法参数类型。 参见@RequestParam以及Multipart。请注意，对于简单参数值，@ RequestParam的使用是可选的。 请参阅此表末尾的“其他任何参数”。
@RequestHeader	用于访问请求标头。 标头值将转换为声明的方法参数类型。 请参阅@RequestHeader。
@CookieValue	用于访问cookie。 Cookies值将转换为声明的方法参数类型。 请参阅@CookieValue。
@RequestBody	用于访问HTTP请求正文。 正文内容通过使用HttpMessageConverter实现转换为声明的方法参数类型。 请参阅@RequestBody。

更多教程请访问:www.flydean.com

HttpEntity<B>	用于访问请求标头和正文。 主体使用HttpMessageConverter进行转换。 请参见HttpEntity。
@RequestPart	要访问multipart / form-data请求中的零件，请使用HttpMessageConverter转换零件的主体。 请参阅多部分。
java.util.Map, org.springframework.ui.Model, org.springframework.ui.ModelMap	用于访问HTML控制器中使用的模型，并作为视图渲染的一部分公开给模板。
RedirectAttributes	指定在重定向的情况下使用的属性（即追加到查询字符串中），并指定要临时存储的属性，直到重定向后的请求为止。 请参阅重定向属性和Flash属性。
@ModelAttribute	用于访问模型中的现有属性（如果不存在，则进行实例化），并应用数据绑定和验证。 请参阅@ModelAttribute以及Model和DataBinder。请注意，@ ModelAttribute的使用是可选的（例如，设置其属性）。 请参阅此表末尾的“其他任何参数”。
Errors, BindingResult	用于访问来自命令对象的验证和数据绑定（即，@ ModelAttribute参数）的错误或来自@RequestBody或@RequestPart参数的验证的错误。 您必须在经过验证的方法参数后立即声明一个Errors或BindingResult参数。
SessionStatus + class-level @SessionAttributes	为了标记表单处理完成，将触发清除通过类级别@SessionAttributes注解声明的会话属性。 有关更多详细信息，请参见@SessionAttributes。
UriComponentsBuilder	用于准备相对于当前请求的主机，端口，方案，上下文路径以及servlet映射的文字部分的URL。 请参阅URI链接。
@SessionAttribute	对于访问任何会话属性，与由于类级@SessionAttributes声明而存储在会话中的模型属性相反。 有关更多详细信息，请参见@SessionAttribute。
@RequestAttribute	用于访问请求属性。 有关更多详细信息，请参见@RequestAttribute。
Any other argument	如果方法参数与该表中的任何较早值都不匹配，并且为简单类型（由BeanUtils # isSimpleProperty确定，则将其解析为@RequestParam。否则，将其解析为@ModelAttribute

Return Values

下表描述了受支持的控制器方法返回值。 所有返回值都支持反应性类型。

Controller method return value	Description
@ResponseBody	返回值通过HttpMessageConverter实现转换并写入响应。 请参阅@ResponseBody。
HttpEntity<B>, ResponseEntity<B>	指定完整响应（包括HTTP标头和正文）的返回值将通过HttpMessageConverter实现进行转换，并写入响应中。 请参阅ResponseEntity。
HttpHeaders	用于返回带有标头且没有正文的响应。
String	使用ViewResolver实现解析的视图名称，并与隐式模型一起使用-通过命令对象和@ModelAttribute方法确定。 处理程序方法还可以通过声明Model参数来以编程方式丰富模型（请参阅显式注册）。
View	一个View实例，用于与隐式模型一起呈现（通过命令对象和@ModelAttribute方法确定）。 处理程序方法还可以通过声明Model参数来以编程方式丰富模型（请参阅显式注册）。
java.util.Map, org.springframework.ui.Model	要添加到隐式模型的属性，其中视图名称通过RequestToViewNameTranslator隐式确定。
@ModelAttribute	要添加到模型的属性，视图名称通过RequestToViewNameTranslator隐式确定。请注意，@ ModelAttribute是可选的。 请参阅此表末尾的“其他任何返回值”。

更多教程请访问:www.flydean.com

ModelAndView object	要使用的视图和模型属性，以及响应状态（可选）。
void	如果返回值类型为void（或返回值为null）的方法还具有ServletResponse，OutputStream参数或@ResponseStatus批注，则认为该方法已完全处理了响应。 如果控制器进行了肯定的ETag或lastModified时间戳检查，也是如此（请参阅控制器以获取详细信息）。如果以上条件都不成立，则对于REST控制器，void返回类型也可以指示“无响应正文”，对于HTML控制器，则默认选择视图名称。
DeferredResult<V>	例如，由于某个事件或回调的结果，从任何线程异步生成任何上述返回值。 请参阅异步请求和DeferredResult。
Callable<V>	在Spring MVC管理的线程中异步产生上述任何返回值。 请参阅异步请求和可调用。
ListenableFuture<V>, java.util.concurrent.CompletionStage<V>, java.util.concurrent.CompletableFuture<V>	为方便起见（例如，当基础服务返回其中之一时），替代DeferredResult。
ResponseBodyEmitter, SseEmitter	使用HttpMessageConverter实现异步发出要写入响应的对象流。 也支持作为ResponseEntity的主体。 请参阅异步请求和HTTP流。
StreamingResponseBody	异步写入响应OutputStream。 也支持作为ResponseEntity的主体。 请参阅异步请求和HTTP流。
Reactive types — Reactor, RxJava, or others through ReactiveAdapterRegistry	DeferredResult的替代方法，其中包含收集到List的多值流（例如Flux，Observable）。对于流场景（例如，文本/事件流，应用程序/ json +流），使用SseEmitter和ResponseBodyEmitter代替，其中ServletOutputStream阻塞I / O在Spring MVC管理的线程上执行，而反压则不利于完成 每个写入。请参阅异步请求和响应类型。
Any other return value	如果返回值不是由BeanUtils确定的简单类型，则它与该表中的任何早先值都不匹配且为String或void的任何返回值均被视为视图名称（通过RequestToViewNameTranslator选择默认视图名称）。 #isSimpleProperty。 简单类型的值仍然无法解析。

Type Conversion

如果参数声明为String以外的形式，则某些表示基于String的请求输入的带注解的控制器方法参数（例如@ RequestParam，@ RequestHeader，@ PathVariable，@ MatrixVariable和@CookieValue）可能需要类型转换。

在这种情况下，将根据配置的转换器自动应用类型转换。 默认情况下，支持简单类型（int，long，Date和其他）。 您可以通过WebDataBinder（请参见DataBinder）或通过在FormattingConversionService中注册Formatter来自定义类型转换。 参见Spring字段格式。

Matrix Variables

RFC 3986讨论了路径段中的名称/值对。 在Spring MVC中，基于Tim Berners-Lee的“旧帖子”，我们将其称为“矩阵变量”，但它们也可以称为URI路径参数。

矩阵变量可以出现在任何路径段中，每个变量用分号分隔，多个值用逗号分隔（例如，/ cars; color = red，green; year = 2012）。 也可以通过重复的变量名称指定多个值（例如，color = red; color = green; color = blue）。

如果期望URL包含矩阵变量，则控制器方法的请求映射必须使用URI变量来屏蔽该变量内容，并确保可以成功地匹配请求，而与矩阵变量的顺序和状态无关。 以下示例使用矩阵变量：

```
// GET /pets/42;q=11;r=22

@GetMapping("/pets/{petId}")
public void findPet(@PathVariable String petId, @MatrixVariable int q) {

    // petId == 42
    // q == 11
```

更多教程请访问:www.flydean.com

```
}
```

鉴于所有路径段都可能包含矩阵变量，因此有时您可能需要消除矩阵变量应位于哪个路径变量的歧义。以下示例说明了如何做到这一点：

```
// 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
}
```

可以将矩阵变量定义为可选变量，并指定默认值，如以下示例所示：

```
// GET /pets/42

@GetMapping("/pets/{petId}")
public void findPet(@MatrixVariable(required=false, defaultValue="1") int q) {

    // q == 1
}
```

要获取所有矩阵变量，可以使用MultiValueMap，如以下示例所示：

```
// 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" : 22, "s" : 23]
}
```

请注意，您需要启用矩阵变量的使用。 在MVC Java配置中，您需要通过Path Matching设置带有removeSemicolonContent = false的UrlPathHelper。 在MVC XML名称空间中，可以设置<mvc：annotation-driven enable-matrix-variables =“ true” />。

@RequestParam

您可以使用@RequestParam批注将Servlet请求参数（即查询参数或表单数据）绑定到控制器中的方法参数。

以下示例显示了如何执行此操作：

```
@Controller
@RequestMapping("/pets")
public class EditPetForm {

    // ...

    @GetMapping
    public String setupForm(@RequestParam("petId") int petId, Model model) {
        Pet pet = this.clinic.loadPet(petId);
        model.addAttribute("pet", pet);
        return "petForm";
    }
```

更多教程请访问:www.flydean.com

```
    // ...

}
```

默认情况下，使用此批注的方法参数是必需的，但是您可以通过将@RequestParam批注的required标志设置为false或使用java.util.Optional包装器声明该参数，来指定方法参数是可选的。

如果目标方法参数类型不是字符串，则将自动应用类型转换。请参阅类型转换。

将参数类型声明为数组或列表，可以为同一参数名称解析多个参数值。

如果将@RequestParam批注声明为Map <String，String>或MultiValueMap <String，String>，但未在批注中指定参数名称，则将使用每个给定参数名称的请求参数值填充映射。

请注意，@ RequestParam的使用是可选的（例如，设置其属性）。默认情况下，任何简单值类型的参数（由BeanUtils＃isSimpleProperty确定）并且没有被任何其他参数解析器解析，就如同使用@RequestParam进行了注解一样。

@RequestHeader

您可以使用@RequestHeader批注将请求标头绑定到控制器中的方法参数。

考虑以下带有标头的请求：

```
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
```

以下示例获取Accept-Encoding和Keep-Alive标头的值：

```
@GetMapping("/demo")
public void handle(
        @RequestHeader("Accept-Encoding") String encoding,
        @RequestHeader("Keep-Alive") long keepAlive) {
    //...
}
```

如果目标方法的参数类型不是String，则将自动应用类型转换。 请参阅类型转换。

在Map <String，String>，MultiValueMap <String，String>或HttpHeaders参数上使用@RequestHeader批注时，将使用所有标头值填充该映射。

> 内置支持可用于将逗号分隔的字符串转换为数组或字符串集合或类型转换系统已知的其他类型。 例如，用@RequestHeader（“ Accept”）注解的方法参数可以是String类型，也可以是String []或List <String>。

@CookieValue

您可以使用@CookieValue批注将HTTP cookie的值绑定到控制器中的方法参数。

考虑带有以下cookie的请求：

```
JSESSIONID=415A4AC178C59DACE0B2C9CA727CDD84
```

以下示例显示如何获取cookie值：

```
@GetMapping("/demo")
public void handle(@CookieValue("JSESSIONID") String cookie) {
    //...
```

更多教程请访问:www.flydean.com

```
}
```

@ModelAttribute

您可以在方法参数上使用@ModelAttribute批注，以从模型访问属性，或者将其实例化（如果不存在）。 model属性还覆盖了名称与字段名称匹配的HTTP Servlet请求参数中的值。 这称为数据绑定，它使您不必处理解析和转换单个查询参数和表单字段的工作。 以下示例显示了如何执行此操作：

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public String processSubmit(@ModelAttribute Pet pet) { }
```

上面的Pet实例解析如下：

- 从模型（如果已通过使用模型添加）。
- 通过使用@SessionAttributes在HTTP会话中进行。
- 来自通过Converter传递的URI路径变量（请参见下一个示例）。
- 从默认构造函数的调用开始。
- 从调用具有与Servlet请求参数匹配的参数的“主要构造函数”开始。 参数名称是通过JavaBeans @ConstructorProperties或字节码中运行时保留的参数名称确定的。

尽管通常使用模型来用属性填充模型，但另一种替代方法是依赖于Converter <String，T>与URI路径变量约定结合使用。 在以下示例中，模型属性名称account与URI路径变量account匹配，并且通过将String帐号传递给已注册的Converter <String，Account>来加载Account：

```
@PutMapping("/accounts/{account}")
public String save(@ModelAttribute("account") Account account) {
    // ...
}
```

获取模型属性实例后，将应用数据绑定。 WebDataBinder类将Servlet请求参数名称（查询参数和表单字段）与目标Object上的字段名称进行匹配。 必要时在应用类型转换后填充匹配字段。 有关数据绑定（和验证）的更多信息，请参见验证。 有关自定义数据绑定的更多信息，请参见DataBinder。

数据绑定可能导致错误。 默认情况下，引发BindException。 但是，要检查控制器方法中的此类错误，可以在@ModelAttribute旁边立即添加BindingResult参数，如以下示例所示：

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public String processSubmit(@ModelAttribute("pet") Pet pet, BindingResult result) {
    if (result.hasErrors()) {
        return "petForm";
    }
    // ...
}
```

在某些情况下，您可能希望访问没有数据绑定的模型属性。 对于这种情况，可以将模型注入控制器中并直接访问它，或者设置@ModelAttribute（binding = false），如以下示例所示：

```
@ModelAttribute
public AccountForm setUpForm() {
    return new AccountForm();
}

@ModelAttribute
public Account findAccount(@PathVariable String accountId) {
    return accountRepository.findOne(accountId);
}

@PostMapping("update")
```

更多教程请访问:www.flydean.com

```
public String update(@Valid AccountForm form, BindingResult result,
        @ModelAttribute(binding=false) Account account) {
    // ...
}
```

您可以通过添加javax.validation.Valid注解或Spring的@Validated注解（Bean验证和Spring验证）自动在数据绑定后应用验证。 以下示例显示了如何执行此操作：

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public String processSubmit(@Valid @ModelAttribute("pet") Pet pet, BindingResult result) {
    if (result.hasErrors()) {
        return "petForm";
    }
    // ...
}
```

请注意，使用@ModelAttribute是可选的（例如，设置其属性）。 默认情况下，任何不是简单值类型（由BeanUtils # isSimpleProperty确定）且未被其他任何参数解析器解析的参数都将被视为使用@ModelAttribute注解。

@SessionAttributes

@SessionAttributes用于在请求之间的HTTP Servlet会话中存储模型属性。 它是类型级别的注解，用于声明特定控制器使用的会话属性。 这通常列出应透明地存储在会话中以供后续访问请求的模型属性的名称或模型属性的类型。

以下示例使用@SessionAttributes批注：

```
@Controller
@SessionAttributes("pet")
public class EditPetForm {
    // ...
}
```

在第一个请求上，将名称为pet的模型属性添加到模型时，该属性会自动提升到HTTP Servlet会话并保存在该会话中。 它会一直保留在那里，直到另一个控制器方法使用SessionStatus方法参数来清除存储，如以下示例所示：

```
@Controller
@SessionAttributes("pet")
public class EditPetForm {

    // ...

    @PostMapping("/pets/{id}")
    public String handle(Pet pet, BindingResult errors, SessionStatus status) {
        if (errors.hasErrors) {
            // ...
        }
            status.setComplete();
            // ...
        }
    }
}
```

@SessionAttribute

如果您需要访问全局存在（例如，在控制器外部（例如，通过过滤器）管理）并且可能存在或可能不存在的预先存在的会话属性，则可以在方法参数上使用@SessionAttribute注解，例如 以下示例显示：

```
@RequestMapping("/")
public String handle(@SessionAttribute User user) {
    // ...
}
```

更多教程请访问:www.flydean.com

对于需要添加或删除会话属性的用例，请考虑将org.springframework.web.context.request.WebRequest或javax.servlet.http.HttpSession注入到控制器方法中。

要在控制器工作流中将模型属性临时存储在会话中，请考虑使用@SessionAttributes，如@SessionAttributes中所述。

@RequestAttribute

与@SessionAttribute相似，您可以使用@RequestAttribute批注来访问先前创建的预先存在的请求属性（例如，通过Servlet Filter或HandlerInterceptor）：

```
@GetMapping("/")
public String handle(@RequestAttribute Client client) {
    // ...
}
```

Redirect Attributes

默认情况下，所有模型属性均被视为在重定向URL中作为URI模板变量公开。在其余属性中，那些属于原始类型或原始类型的集合或数组的属性会自动附加为查询参数。

如果专门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理想的结果。但是，在带注解的控制器中，模型可以包含为渲染目的添加的其他属性（例如，下拉字段值）。为避免此类属性出现在URL中的可能性，@RequestMapping方法可以声明RedirectAttributes类型的参数，并使用它指定可用于RedirectView的确切属性。如果该方法确实重定向，则使用RedirectAttributes的内容。否则，将使用模型的内容。

RequestMappingHandlerAdapter提供了一个名为ignoreDefaultModelOnRedirect的标志，您可以使用该标志指示如果控制器方法重定向，则绝不要使用默认Model的内容。相反，控制器方法应声明一个RedirectAttributes类型的属性，或者，如果没有这样做，则不应将任何属性传递给RedirectView。 MVC名称空间和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，展开重定向URL时，本请求中的URI模板变量会自动变为可用，您无需通过Model或RedirectAttributes显式添加它们。以下示例显示了如何定义重定向：

```
@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使用flash attributes。 与其他重定向属性不同，flash attributes保存在HTTP会话中（因此不会出现在URL中）。 有关更多信息，请参见Flash属性。

Flash Attributes

Flash属性为一个请求提供了一种存储要在另一个请求中使用的属性的方式。重定向时最常需要此操作，例如Post-Redirect-Get模式。 Flash属性在重定向之前（通常在会话中）被临时保存，以便在重定向之后可供请求使用，并立即被删除。

Spring MVC有两个主要的抽象来支持Flash属性。 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，无需显式启用。但是，如果不使用它，则永远不会导致HTTP会话创建。在每个请求上，都有一个“输入” FlashMap，其属性是从上一个请求（如果有）传递过来的，而“输出” FlashMap的属性是为后续请求保存的。可以通过RequestContextUtils中的静态方法从Spring MVC中的任何位置访问这两个FlashMap实例。

更多教程请访问:www.flydean.com

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RedirectAttributes类型的参数，并使用它为重定向方案添加Flash属性。通过RedirectAttributes添加的Flash属性会自动传播到“输出” FlashMap。同样，重定向后，来自“输入” FlashMap的属性会自动添加到服务于目标URL的控制器的模型中。

> 将请求与Flash属性匹配 Flash属性的概念存在于许多其他Web框架中，并已证明有时会遇到并发问题。 这是因为根据定义，闪存属性将存储到下一个请求。 但是，“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，过早删除了闪存属性。
>
> 为了减少此类问题的可能性，RedirectView会自动使用目标重定向URL的路径和查询参数“标记” FlashMap实例。反过来，默认FlashMapManager在查找“输入” FlashMap时会将信息与传入请求匹配。
>
> 这不能完全消除并发问题的可能性，但可以通过重定向URL中已经可用的信息大大减少并发问题。 因此，我们建议您主要将Flash属性用于重定向方案。

Multipart

启用MultipartResolver后，带有multipart / form-data的POST请求的内容将被解析并作为常规请求参数进行访问。 以下示例访问一个常规表单字段和一个上载文件：

```
@Controller
public class FileUploadController {

    @PostMapping("/form")
    public String handleFormUpload(@RequestParam("name") String name,
            @RequestParam("file") MultipartFile file) {

        if (!file.isEmpty()) {
            byte[] bytes = file.getBytes();
            // store the bytes somewhere
            return "redirect:uploadSuccess";
        }
        return "redirect:uploadFailure";
    }
}
```

将参数类型声明为List <MultipartFile>允许解析相同参数名称的多个文件。

如果将@RequestParam批注声明为Map <String，MultipartFile>或MultiValueMap <String，MultipartFile>，但未在批注中指定参数名称，则将使用每个给定参数名称的多部分文件填充映射。

> 通过Servlet 3.0多部分解析，您还可以声明javax.servlet.http.Part而不是Spring的MultipartFile作为方法参数或集合值类型。

您还可以将多部分内容用作数据绑定到命令对象的一部分。 例如，前面示例中的表单字段和文件可以是表单对象上的字段，如以下示例所示：

```
class MyForm {

    private String name;

    private MultipartFile file;

    // ...
}

@Controller
public class FileUploadController {

    @PostMapping("/form")
    public String handleFormUpload(MyForm form, BindingResult errors) {
        if (!form.getFile().isEmpty()) {
            byte[] bytes = form.getFile().getBytes();
            // store the bytes somewhere
```

更多教程请访问:www.flydean.com

```
            return "redirect:uploadSuccess";
        }
        return "redirect:uploadFailure";
    }
}
```

在RESTful服务方案中，也可以从非浏览器客户端提交多部分请求。 以下示例显示了带有JSON的文件：

```
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 ...
```

您可以使用@RequestParam作为字符串访问“元数据”部分，但您可能希望将其从JSON反序列化（类似于@RequestBody）。 在使用HttpMessageConverter进行转换后，使用@RequestPart批注来访问多部分：

```
@PostMapping("/")
public String handle(@RequestPart("meta-data") MetaData metadata,
        @RequestPart("file-data") MultipartFile file) {
    // ...
}
```

您可以将@RequestPart与javax.validation.Valid结合使用，也可以使用Spring的@Validated注解，这两种注解都会导致应用标准Bean验证。 默认情况下，验证错误会导致MethodArgumentNotValidException，并将其转换为400（BAD_REQUEST）响应。 另外，您可以通过Errors或BindingResult参数在控制器内本地处理验证错误，如以下示例所示：

```
@PostMapping("/")
public String handle(@Valid @RequestPart("meta-data") MetaData metadata,
        BindingResult result) {
    // ...
}
```

@RequestBody

您可以使用@RequestBody批注使请求正文通过HttpMessageConverter读取并反序列化为Object。 以下示例使用@RequestBody参数：

```
@PostMapping("/accounts")
public void handle(@RequestBody Account account) {
    // ...
}
```

您可以使用MVC Config的“消息转换器”选项来配置或自定义消息转换。

更多教程请访问:www.flydean.com

您可以将@RequestBody与javax.validation.Valid或Spring的@Validated注解结合使用，这两种注解都会导致应用标准Bean验证。 默认情况下，验证错误会导致MethodArgumentNotValidException，并将其转换为400（BAD_REQUEST）响应。 另外，您可以通过Errors或BindingResult参数在控制器内本地处理验证错误，如以下示例所示：

```
@PostMapping("/accounts")
public void handle(@Valid @RequestBody Account account, BindingResult result) {
    // ...
}
```

HttpEntity

HttpEntity或多或少与使用@RequestBody相同，但它基于公开请求标头和正文的容器对象。 以下清单显示了一个示例：

```
@PostMapping("/accounts")
public void handle(HttpEntity<Account> entity) {
    // ...
}
```

@ResponseBody

您可以在方法上使用@ResponseBody批注，以通过HttpMessageConverter将返回序列化为响应主体。 以下清单显示了一个示例：

```
@GetMapping("/accounts/{id}")
@ResponseBody
public Account handle() {
    // ...
}
```

在类级别也支持@ResponseBody，在这种情况下，所有控制器方法都将继承它。 这就是@RestController的效果，它只不过是带有@Controller和@ResponseBody标记的元注解。

您可以将@ResponseBody与反应类型一起使用。 有关更多详细信息，请参见异步请求和响应类型。

您可以使用MVC Config的“消息转换器”选项来配置或自定义消息转换。

您可以将@ResponseBody方法与JSON序列化视图结合使用。 有关详细信息，请参见Jackson JSON。

ResponseEntity

ResponseEntity类似于@ResponseBody，但具有状态和标头。 例如：

```
@GetMapping("/something")
public ResponseEntity<String> handle() {
    String body = ... ;
    String etag = ... ;
    return ResponseEntity.ok().eTag(etag).build(body);
}
```

Spring MVC支持使用单值反应类型异步生成ResponseEntity，和/或为主体使用单值和多值反应类型。

Jackson JSON

Spring提供了对Jackson JSON库的支持。

更多教程请访问:www.flydean.com

Spring MVC为Jackson的序列化视图提供了内置支持，该视图仅可呈现Object中所有字段的一部分。 要将其与@ResponseBody或ResponseEntity控制器方法一起使用，可以使用Jackson的@JsonView批注来激活序列化视图类，如以下示例所示：

```
@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;
    }
}
```

> @JsonView允许使用一组视图类，但是每个控制器方法只能指定一个。 如果需要激活多个视图，则可以使用复合界面。

对于依赖视图解析的控制器，可以将序列化视图类添加到模型中，如以下示例所示：

```
@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";
    }
}
```

1.4.4 Model

您可以使用@ModelAttribute批注：

- 在@RequestMapping方法中的方法参数上，可从模型创建或访问对象，并将其通过WebDataBinder绑定到请求。

更多教程请访问:www.flydean.com

- 作为@Controller或@ControllerAdvice类中的方法级注解，可在任何@RequestMapping方法调用之前帮助初始化模型。
- 在@RequestMapping方法上标记其返回值的是模型属性。

本节讨论@ModelAttribute方法-前面列表中的第二项。控制器可以具有任意数量的@ModelAttribute方法。所有此类方法均在同一控制器中的@RequestMapping方法之前调用。也可以通过@ControllerAdvice在控制器之间共享@ModelAttribute方法。有关更多详细信息，请参见“控制器建议”部分。

@ModelAttribute方法具有灵活的方法签名。它们支持与@RequestMapping方法相同的许多参数，除了@ModelAttribute本身或与请求正文相关的任何东西。

下面的示例显示一个@ModelAttribute方法：

```
@ModelAttribute
public void populateModel(@RequestParam String number, Model model) {
    model.addAttribute(accountRepository.findAccount(number));
    // add more ...
}
```

以下示例仅添加一个属性：

```
@ModelAttribute
public Account addAccount(@RequestParam String number) {
    return accountRepository.findAccount(number);
}
```

> 如果未明确指定名称，则根据Object 类型选择默认名称，如javadoc中针对的解释Conventions。您始终可以使用重载addAttribute方法或通过nameon 的属性@ModelAttribute（用于返回值）来分配显式名称。

与Spring MVC不同，Spring WebFlux在模型中显式支持反应式类型（例如Mono <Account>或io.reactivex.Single <Account>）。 可以在@RequestMapping调用时将此类异步模型属性透明地解析（并更新模型）为其实际值，只要声明了@ModelAttribute参数而没有包装，如以下示例所示：

```
@ModelAttribute
public void addAccount(@RequestParam String number) {
    Mono<Account> accountMono = accountRepository.findAccount(number);
    model.addAttribute("account", accountMono);
}

@PostMapping("/accounts")
public String handle(@ModelAttribute Account account, BindingResult errors) {
    // ...
}
```

另外，任何具有反应性类型包装器的模型属性都将在视图渲染之前解析为其实际值（并更新了模型）。

您也可以将@ModelAttribute用作@RequestMapping方法上的方法级注解，在这种情况下，@ RequestMapping方法的返回值将解释为模型属性。 通常不需要这样做，因为它是HTML控制器的默认行为，除非返回值是一个String，否则它将被解释为视图名称。 @ModelAttribute还可以帮助自定义模型属性名称，如以下示例所示：

```
@GetMapping("/accounts/{id}")
@ModelAttribute("myAccount")
public Account handle() {
    // ...
    return account;
}
```

更多教程请访问:www.flydean.com

1.4.5 DataBinder

@Controller或@ControllerAdvice类可以具有用于初始化WebDataBinder实例的@InitBinder方法，而这些方法又可以：

- 将请求参数（即表单或查询数据）绑定到模型对象。
- 将基于字符串的请求值（例如请求参数，路径变量，标头，Cookie等）转换为控制器方法参数的目标类型。
- 呈现HTML表单时，将模型对象的值格式化为String值。

@InitBinder方法可以注册特定于控制器的java.bean.PropertyEditor或Spring Converter和Formatter组件。 另外，您可以使用MVC配置在全局共享的FormattingConversionService中注册Converter和Formatter类型。

@InitBinder方法支持与@RequestMapping方法相同的许多参数，除了@ModelAttribute（命令对象）参数。 通常，它们使用WebDataBinder参数（用于注册）和无效的返回值声明。 以下清单显示了一个示例：

```
@Controller
public class FormController {

    @InitBinder
    public void initBinder(WebDataBinder binder) {
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        dateFormat.setLenient(false);
        binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, false));
    }

    // ...
}
```

另外，当通过共享的FormattingConversionService使用基于Formatter的设置时，可以重新使用相同的方法并注册特定于控制器的Formatter实现，如以下示例所示：

```
@Controller
public class FormController {

    @InitBinder
    protected void initBinder(WebDataBinder binder) {
        binder.addCustomFormatter(new DateFormatter("yyyy-MM-dd"));
    }

    // ...
}
```

1.4.6 Managing Exceptions

@Controller和@ControllerAdvice类可以具有@ExceptionHandler方法来处理来自控制器方法的异常。 下面的示例包括这样的处理程序方法：

```
@Controller
public class SimpleController {

    // ...

    @ExceptionHandler
    public ResponseEntity<String> handle(IOException ex) {
        // ...
    }
}
```

更多教程请访问:www.flydean.com

该异常可以与正在传播的顶级异常（即，直接IOException被抛出）匹配，也可以与顶级包装器异常（例如，包装在IllegalStateException内部的IOException）内的直接原因匹配。

对于匹配的异常类型，最好将目标异常声明为方法参数，如前面的示例所示。或者，注解声明可以缩小异常类型以使其匹配。我们通常建议在参数签名中尽可能具体，并在以相应顺序优先的@ControllerAdvice上声明您的主根异常映射。有关详细信息，请参见MVC部分。

> WebFlux中的@ExceptionHandler方法支持与@RequestMapping方法相同的方法参数和返回值，但与请求正文和@ModelAttribute相关的方法参数除外。

HandlerAdapter为@RequestMapping方法提供对Spring WebFlux中@ExceptionHandler方法的支持。有关更多详细信息，请参见DispatcherHandler。

REST API exceptions

REST服务的常见要求是在响应正文中包含错误详细信息。 Spring框架不会自动这样做，因为响应主体中错误详细信息的表示是特定于应用程序的。 但是，@ RestController可以将@ExceptionHandler方法与ResponseEntity返回值一起使用，以设置响应的状态和主体。 也可以在@ControllerAdvice类中声明此类方法，以将其全局应用。

> 请注意，Spring WebFlux与Spring MVC ResponseEntityExceptionHandler没有等效项，因为WebFlux仅引发ResponseStatusException（或其子类），并且不需要将其转换为HTTP状态代码。

1.4.7 Controller Advice

通常，@ ExceptionHandler，@ InitBinder和@ModelAttribute方法在声明它们的@Controller类（或类层次结构）中应用。如果要使此类方法更全局地应用（跨控制器），则可以在标有@ControllerAdvice或@RestControllerAdvice的类中声明它们。

@ControllerAdvice标有@Component，这意味着可以通过组件扫描将此类注册为Spring Bean。@RestControllerAdvice还是一个标有@ControllerAdvice和@ResponseBody的元注解，这实际上意味着@ExceptionHandler方法通过消息转换（与视图分辨率或模板渲染）呈现到响应主体。

在启动时，@RequestMapping和@ExceptionHandler方法的基础结构类检测@ControllerAdvice类型的Spring bean，然后在运行时应用其方法。全局@ExceptionHandler方法（来自@ControllerAdvice）在本地方法（来自@Controller）之后应用。相比之下，全局@ModelAttribute和@InitBinder方法在本地方法之前应用。

默认情况下，@ ControllerAdvice方法适用于每个请求（即所有控制器），但是您可以通过使用批注上的属性将其范围缩小到控制器的子集，如以下示例所示：

```
// Target all Controllers annotated with @RestController
@ControllerAdvice(annotations = RestController.class)
public class ExampleAdvice1 {}

// Target all Controllers within specific packages
@ControllerAdvice("org.example.controllers")
public class ExampleAdvice2 {}

// Target all Controllers assignable to specific classes
@ControllerAdvice(assignableTypes = {ControllerInterface.class, AbstractController.class})
public class ExampleAdvice3 {}
```

前面示例中的选择器在运行时进行评估，如果广泛使用，可能会对性能产生负面影响。 有关更多详细信息，请参见@ControllerAdvice javadoc。

更多教程请访问:www.flydean.com

更多教程请访问:www.flydean.com

1.5 Functional Endpoints

Spring WebFlux包含WebFlux.fn，这是一个轻量级的函数编程模型，其中的函数用于路由和处理请求，而契约则是为不变性而设计的。它是基于注解的编程模型的替代方案，但可以在相同的Reactive Core基础上运行。

1.5.1 总览

在WebFlux.fn中，HTTP请求由HandlerFunction处理：该函数接受ServerRequest并返回延迟的ServerResponse（即Mono <ServerResponse>）。作为请求对象的请求都具有不可变的协定，这些协定为JDK 8提供了对HTTP请求和响应的友好访问。 HandlerFunction等效于基于注解的编程模型中@RequestMapping方法的主体。

传入的请求通过RouterFunction路由到处理程序函数：该函数接受ServerRequest并返回延迟的HandlerFunction（即Mono <HandlerFunction>）。当路由器功能匹配时，返回处理程序功能。否则为空Mono。 RouterFunction等效于@RequestMapping批注，但主要区别在于路由器功能不仅提供数据，还提供行为。

RouterFunctions.route（）提供了一个路由器构建器，可简化路由器的创建过程，如以下示例所示：

```
import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.web.reactive.function.server.RequestPredicates.*;
import static org.springframework.web.reactive.function.server.RouterFunctions.route;

PersonRepository repository = ...
PersonHandler handler = new PersonHandler(repository);

RouterFunction<ServerResponse> route = route()
    .GET("/person/{id}", accept(APPLICATION_JSON), handler::getPerson)
    .GET("/person", accept(APPLICATION_JSON), handler::listPeople)
    .POST("/person", handler::createPerson)
    .build();


public class PersonHandler {

    // ...

    public Mono<ServerResponse> listPeople(ServerRequest request) {
        // ...
    }

    public Mono<ServerResponse> createPerson(ServerRequest request) {
        // ...
    }

    public Mono<ServerResponse> getPerson(ServerRequest request) {
        // ...
    }
}
```

运行RouterFunction的一种方法是将其转换为HttpHandler并通过内置服务器适配器之一进行安装：

- RouterFunctions.toHttpHandler（RouterFunction）
- RouterFunctions.toHttpHandler（RouterFunction，HandlerStrategies）

大多数应用程序都可以通过WebFlux Java配置运行，请参阅运行服务器。

1.5.2 HandlerFunction

更多教程请访问:www.flydean.com

ServerRequest和ServerResponse是不可变的接口，它们提供JDK 8友好的HTTP请求和响应访问。 请求和响应都为反应流提供了对体流的反压力。 请求主体用Reactor Flux或Mono表示。 响应主体由任何Reactive Streams Publisher组成，包括Flux和Mono。 有关更多信息，请参见反应式库。

ServerRequest

ServerRequest提供对HTTP方法，URI，标头和查询参数的访问，而通过body方法提供对主体的访问。

下面的示例将请求正文提取到Mono <String>：

```
Mono<String> string = request.bodyToMono(String.class);
```

下面的示例将主体提取到Flux <Person>，其中Person对象从某种序列化形式（例如JSON或XML）解码：

```
Flux<Person> people = request.bodyToFlux(Person.class);
```

前面的示例是使用更通用的ServerRequest.body（BodyExtractor）的快捷方式，该请求接受BodyExtractor功能策略接口。 实用程序类BodyExtractors提供对许多实例的访问。 例如，前面的示例也可以编写如下：

```
Mono<String> string = request.body(BodyExtractors.toMono(String.class));
Flux<Person> people = request.body(BodyExtractors.toFlux(Person.class));
```

下面的示例显示如何访问表单数据：

```
Mono<MultiValueMap<String, String> map = request.body(BodyExtractors.toFormData());
```

以下示例显示了如何以Map形式访问多部分数据：

```
Mono<MultiValueMap<String, Part> map = request.body(BodyExtractors.toMultipartData());
```

下面的示例演示如何以流方式一次访问多个部分：

```
Flux<Part> parts = request.body(BodyExtractors.toParts());
```

ServerResponse

ServerResponse提供对HTTP响应的访问，并且由于它是不可变的，因此可以使用一种build方法来创建它。您可以使用构建器来设置响应状态，添加响应标题或提供正文。以下示例使用JSON内容创建200（确定）响应：

```
Mono<Person> person = ...
ServerResponse.ok().contentType(MediaType.APPLICATION_JSON).body(person, Person.class);
```

下面的示例演示如何构建Location不带标头的201（已创建）响应：

```
URI location = ...
ServerResponse.created(location).build();
```

Handler Classes

我们可以将处理程序函数编写为lambda，如以下示例所示：

```
HandlerFunction<ServerResponse> helloWorld =
  request -> ServerResponse.ok().body(fromObject("Hello World"));
```

更多教程请访问:www.flydean.com

这很方便，但是在应用程序中我们需要多个功能，并且多个内联lambda可能会变得凌乱。 因此，将相关的处理程序功能分组到一个处理程序类中很有用，该类的作用与基于注解的应用程序中的@Controller相似。 例如，以下类公开了反应型Person存储库：

```
import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.web.reactive.function.ServerResponse.ok;
import static org.springframework.web.reactive.function.BodyInserters.fromObject;

public class PersonHandler {

    private final PersonRepository repository;

    public PersonHandler(PersonRepository repository) {
        this.repository = repository;
    }

    public Mono<ServerResponse> listPeople(ServerRequest request) {
        Flux<Person> people = repository.allPeople();
        return ok().contentType(APPLICATION_JSON).body(people, Person.class);
    }

    public Mono<ServerResponse> createPerson(ServerRequest request) {
        Mono<Person> person = request.bodyToMono(Person.class);
        return ok().build(repository.savePerson(person));
    }

    public Mono<ServerResponse> getPerson(ServerRequest request) {
        int personId = Integer.valueOf(request.pathVariable("id"));
        return repository.getPerson(personId)
            .flatMap(person -> ok().contentType(APPLICATION_JSON).body(fromObject(person)))
            .switchIfEmpty(ServerResponse.notFound().build());
    }
}
```

Validation

功能端点可以使用Spring的验证工具将验证应用于请求正文。 例如，给定Person的自定义Spring Validator实现：

```
public class PersonHandler {

    private final Validator validator = new PersonValidator();

    // ...

    public Mono<ServerResponse> createPerson(ServerRequest request) {
        Mono<Person> person = request.bodyToMono(Person.class).doOnNext(this::validate);
        return ok().build(repository.savePerson(person));
    }

    private void validate(Person person) {
        Errors errors = new BeanPropertyBindingResult(body, "person");
        validator.validate(body, errors);
        if (errors.hasErrors) {
            throw new ServerWebInputException(errors.toString());
        }
    }
```

处理程序还可以通过创建和注入基于LocalValidatorFactoryBean的全局Validator实例来使用标准的bean验证API（JSR-303）。

1.5.3 RouterFunction

更多教程请访问:www.flydean.com

路由器功能用于将请求路由到相应的HandlerFunction。通常，您不是自己编写路由器功能，而是使用RouterFunctions实用工具类上的方法来创建一个。 RouterFunctions.route（）（无参数）为您提供了流畅的生成器，用于创建路由器功能，而RouterFunctions.route（RequestPredicate，HandlerFunction）提供了直接创建路由器的方法。

通常，建议使用route（）构建器，因为它为典型的映射方案提供了便捷的快捷方式，而无需发现静态导入。例如，路由器功能构建器提供了GET（String，HandlerFunction）方法来为GET请求创建映射。和POST（String，HandlerFunction）进行POST。

除了基于HTTP方法的映射外，路由构建器还提供了一种在映射到请求时引入其他谓词的方法。对于每个HTTP方法，都有一个重载的变体，它以RequestPredicate作为参数，尽管可以表达其他约束。

Predicates

您可以编写自己的RequestPredicate，但是RequestPredicates实用程序类根据请求路径，HTTP方法，内容类型等提供常用的实现。以下示例使用请求谓词基于Accept头创建约束：

```
RouterFunction<ServerResponse> route = RouterFunctions.route()
    .GET("/hello-world", accept(MediaType.TEXT_PLAIN),
        request -> Response.ok().body(fromObject("Hello World")));
```

您可以使用以下命令组合多个请求谓词：

- RequestPredicate.and（RequestPredicate）-两者都必须匹配。
- RequestPredicate.or（RequestPredicate）-两者都可以匹配。

RequestPredicates中的许多谓词都是组成的。例如，RequestPredicates.GET（String）由RequestPredicates.method（HttpMethod）和RequestPredicates.path（String）组成。上面显示的示例还使用了两个请求谓词，因为构建器在内部使用RequestPredicates.GET并将其与接受谓词组合在一起。

Routes

路由器功能按顺序评估：如果第一个路由不匹配，则评估第二个路由，依此类推。因此，在通用路由之前声明更具体的路由是有意义的。请注意，此行为不同于基于注解的编程模型，在该模型中，将自动选择“最特定”的控制器方法。

使用路由器功能生成器时，所有定义的路由都组成一个RouterFunction，从build（）返回。还有其他方法可以将多个路由器功能组合在一起：

- 在RouterFunctions.route（）构建器上添加（RouterFunction）
- RouterFunction.and（RouterFunction）
- RouterFunction.andRoute（RequestPredicate，HandlerFunction）-带有嵌套RouterFunctions.route（）的RouterFunction.and（）的快捷方式。

以下示例显示了四种路线的组成：

```
import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.web.reactive.function.server.RequestPredicates.*;

PersonRepository repository = ...
PersonHandler handler = new PersonHandler(repository);

RouterFunction<ServerResponse> otherRoute = ...

RouterFunction<ServerResponse> route = route()
    .GET("/person/{id}", accept(APPLICATION_JSON), handler::getPerson)
    .GET("/person", accept(APPLICATION_JSON), handler::listPeople)
    .POST("/person", handler::createPerson)
    .add(otherRoute)
    .build();
```

Nested Routes

更多教程请访问:www.flydean.com

一组路由器功能通常具有共享谓词，例如共享路径。 在上面的示例中，共享谓词将是与/ person匹配的路径谓词，由三个路由使用。 使用注解时，您可以通过使用映射到/ person的类型级别@RequestMapping注解来删除此重复项。 在WebFlux.fn中，可以通过路由器功能构建器上的path方法共享路径谓词。 例如，以上示例的最后几行可以通过使用嵌套路由以以下方式进行改进：

```
RouterFunction<ServerResponse> route = route()
    .path("/person", builder -> builder
        .GET("/{id}", accept(APPLICATION_JSON), handler::getPerson)
        .GET("", accept(APPLICATION_JSON), handler::listPeople)
        .POST("/person", handler::createPerson))
    .build();
```

请注意，路径的第二个参数是使用路由器构建器的使用者。

尽管基于路径的嵌套是最常见的，但是您可以通过使用构建器上的nest方法来嵌套在任何种类的谓词上。 上面的内容仍然包含一些以共享的Accept-header谓词形式出现的重复。 通过将nest方法与accept一起使用，我们可以进一步改进：

```
RouterFunction<ServerResponse> route = route()
    .path("/person", b1 -> b1
        .nest(accept(APPLICATION_JSON), b2 -> b2
            .GET("/{id}", handler::getPerson)
            .GET("", handler::listPeople))
        .POST("/person", handler::createPerson))
    .build();
```

1.5.4 Running a Server

如何在HTTP服务器中运行路由器功能？一个简单的选项是使用以下方法之一将路由器功能转换为HttpHandler：

- RouterFunctions.toHttpHandler（RouterFunction）
- RouterFunctions.toHttpHandler（RouterFunction，HandlerStrategies）

然后，可以通过遵循HttpHandler来获取特定于服务器的指令，将返回的HttpHandler与许多服务器适配器一起使用。

Spring Boot还使用了一个更典型的选项，即通过WebFlux Config使用基于DispatcherHandler的设置来运行，该配置使用Spring配置声明处理请求所需的组件。 WebFlux Java配置声明以下基础结构组件以支持功能端点：

- RouterFunctionMapping：在Spring配置中检测一个或多个RouterFunction <? > bean，通过RouterFunction.andOther组合它们，并将请求路由到生成的组成RouterFunction。
- HandlerFunctionAdapter：简单的适配器，它使DispatcherHandler调用映射到请求的HandlerFunction。
- ServerResponseResultHandler：通过调用ServerResponse的writeTo方法来处理HandlerFunction调用的结果。

前面的组件使功能端点适合于DispatcherHandler请求处理生命周期，并且（如果有）声明的控制器也可以（可能）与带注解的控制器并排运行。这也是Spring Boot WebFlux启动器启用功能端点的方式。

以下示例显示了WebFlux Java配置（有关如何运行它，请参见DispatcherHandler）：

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Bean
    public RouterFunction<?> routerFunctionA() {
        // ...
    }

    @Bean
    public RouterFunction<?> routerFunctionB() {
        // ...
```

更多教程请访问:www.flydean.com

```
    }

    // ...

    @Override
    public void configureHttpMessageCodecs(ServerCodecConfigurer configurer) {
        // configure message conversion...
    }

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        // configure CORS...
    }

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        // configure view resolution for HTML rendering...
    }
}
```

1.5.5 Filtering Handler Functions

您可以使用路由功能构建器上的before，after或filter方法来过滤处理程序函数。 使用注解，可以通过使用@ControllerAdvice，ServletFilter或同时使用两者来实现类似的功能。 该过滤器将应用于构建器构建的所有路由。 这意味着在嵌套路由中定义的过滤器不适用于“顶级”路由。 例如，考虑以下示例：

```
RouterFunction<ServerResponse> route = route()
    .path("/person", b1 -> b1
        .nest(accept(APPLICATION_JSON), b2 -> b2
            .GET("/{id}", handler::getPerson)
            .GET("", handler::listPeople)
            .before(request -> ServerRequest.from(request)
                .header("X-RequestHeader", "Value")
                .build()))
        .POST("/person", handler::createPerson))
    .after((request, response) -> logResponse(response))
    .build();
```

路由器构建器上的filter方法采用HandlerFilterFunction：该函数采用ServerRequest和HandlerFunction并返回ServerResponse。 handler函数参数代表链中的下一个元素。 这通常是路由到的处理程序，但是如果应用了多个，它也可以是另一个过滤器。

现在，我们可以在路由中添加一个简单的安全过滤器，假设我们拥有一个可以确定是否允许特定路径的SecurityManager。 以下示例显示了如何执行此操作：

```
SecurityManager securityManager = ...

RouterFunction<ServerResponse> route = route()
    .path("/person", b1 -> b1
        .nest(accept(APPLICATION_JSON), b2 -> b2
            .GET("/{id}", handler::getPerson)
            .GET("", handler::listPeople))
        .POST("/person", handler::createPerson))
    .filter((request, next) -> {
        if (securityManager.allowAccessTo(request.path())) {
            return next.handle(request);
        }
        else {
            return ServerResponse.status(UNAUTHORIZED).build();
        }
    })
    .build();
```

更多教程请访问:www.flydean.com

前面的示例演示了调用next.handle（ServerRequest）是可选的。 当允许访问时，我们仅允许执行处理函数。

除了在路由器功能构建器上使用filter方法之外，还可以通过RouterFunction.filter（HandlerFilterFunction）将过滤器应用于现有路由器功能。

> 通过专用的CorsWebFilter提供对功能端点的CORS支持。

更多教程请访问:www.flydean.com

1.6 URI Links

本部分介绍了Spring框架中可用于URI的各种选项。

1.6.1 UriComponents

UriComponentsBuilder有助于从具有变量的URI模板中构建URI，如以下示例所示：

```
UriComponents uriComponents = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}")
        .queryParam("q", "{q}")
        .encode()
        .build();

URI uri = uriComponents.expand("Westin", "123").toUri();
```

可以将前面的示例合并为一个链，并通过buildAndExpand进行缩短，如以下示例所示：

```
URI uri = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}")
        .queryParam("q", "{q}")
        .encode()
        .buildAndExpand("Westin", "123")
        .toUri();
```

您可以通过直接转到URI（这意味着编码）来进一步缩短它，如以下示例所示：

```
URI uri = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}")
        .queryParam("q", "{q}")
        .build("Westin", "123");
```

您可以使用完整的URI模板进一步缩短它，如以下示例所示：

```
URI uri = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}?q={q}")
        .build("Westin", "123");
```

1.6.2 UriBuilder

UriComponentsBuilder实现UriBuilder。 您可以依次使用UriBuilderFactory创建UriBuilder。 UriBuilderFactory和UriBuilder一起提供了一种可插入的机制，可以基于共享配置（例如基本URL，编码首选项和其他详细信息）从URI模板构建URI。

您可以使用UriBuilderFactory配置RestTemplate和WebClient以自定义URI的准备。 DefaultUriBuilderFactory是UriBuilderFactory的默认实现，该实现在内部使用UriComponentsBuilder并公开共享的配置选项。

以下示例显示如何配置RestTemplate：

```
// import org.springframework.web.util.DefaultUriBuilderFactory.EncodingMode;

String baseUrl = "https://example.org";
DefaultUriBuilderFactory factory = new DefaultUriBuilderFactory(baseUrl);
```

更多教程请访问:www.flydean.com

```
factory.setEncodingMode(EncodingMode.TEMPLATE_AND_VARIABLES);

RestTemplate restTemplate = new RestTemplate();
restTemplate.setUriTemplateHandler(factory);
```

下面的例子配置了WebClient：

```
// import org.springframework.web.util.DefaultUriBuilderFactory.EncodingMode;

String baseUrl = "https://example.org";
DefaultUriBuilderFactory factory = new DefaultUriBuilderFactory(baseUrl);
factory.setEncodingMode(EncodingMode.TEMPLATE_AND_VARIABLES);

WebClient client = WebClient.builder().uriBuilderFactory(factory).build();
```

此外，您也可以直接使用DefaultUriBuilderFactory。 它类似于使用UriComponentsBuilder，但不是静态工厂方法，它是一个包含配置和首选项的实际实例，如以下示例所示：

```
String baseUrl = "https://example.com";
DefaultUriBuilderFactory uriBuilderFactory = new DefaultUriBuilderFactory(baseUrl);

URI uri = uriBuilderFactory.uriString("/hotels/{hotel}")
        .queryParam("q", "{q}")
        .build("Westin", "123");
```

1.6.3 URI Encoding

UriComponentsBuilder在两个级别公开了编码选项：

- UriComponentsBuilder # encode（）：首先对URI模板进行预编码，然后在扩展时严格对URI变量进行编码。
- UriComponents # encode（）：扩展URI变量后，对URI组件进行编码。

这两个选项都用转义的八位字节替换非ASCII和非法字符。 但是，第一个选项还会替换出现在URI变量中的具有保留含义的字符。

> 考虑“;”，这在路径上是合法的，但具有保留的含义。 第一个选项代替“;” URI变量中带有“%3B”，但URI模板中没有。 相比之下，第二个选项永远不会替换“;”，因为它是路径中的合法字符。

在大多数情况下，第一个选项可能会产生预期的结果，因为它将URI变量视为要完全编码的不透明数据，而选项2仅在URI变量有意包含保留字符的情况下才有用。

以下示例使用第一个选项：

```
URI uri = UriComponentsBuilder.fromPath("/hotel list/{city}")
            .queryParam("q", "{q}")
            .encode()
            .buildAndExpand("New York", "foo+bar")
            .toUri();

    // Result is "/hotel%20list/New%20York?q=foo%2Bbar"
```

您可以通过直接转到URI（这意味着编码）来缩短前面的示例，如以下示例所示：

```
URI uri = UriComponentsBuilder.fromPath("/hotel list/{city}")
            .queryParam("q", "{q}")
            .build("New York", "foo+bar")
```

您可以使用完整的URI模板进一步缩短它，如以下示例所示：

更多教程请访问:www.flydean.com

```
URI uri = UriComponentsBuilder.fromPath("/hotel list/{city}?q={q}")
            .build("New York", "foo+bar")
```

WebClient和RestTemplate通过UriBuilderFactory策略在内部扩展和编码URI模板。 两者都可以使用自定义策略进行配置。 如下例所示：

```
String baseUrl = "https://example.com";
DefaultUriBuilderFactory factory = new DefaultUriBuilderFactory(baseUrl)
factory.setEncodingMode(EncodingMode.TEMPLATE_AND_VALUES);

// Customize the RestTemplate..
RestTemplate restTemplate = new RestTemplate();
restTemplate.setUriTemplateHandler(factory);

// Customize the WebClient..
WebClient client = WebClient.builder().uriBuilderFactory(factory).build();
```

DefaultUriBuilderFactory实现在内部使用UriComponentsBuilder来扩展和编码URI模板。作为工厂，它提供了一个位置，可以根据以下一种编码模式来配置编码方法：

- TEMPLATE_AND_VALUES：使用UriComponentsBuilder # encode（）（对应于较早列表中的第一个选项）对URI模板进行预编码，并在扩展时严格编码URI变量。
- VALUES_ONLY：不对URI模板进行编码，而是在将其扩展到模板之前通过UriUtils # encodeUriUriVariables对URI变量进行严格编码。
- URI_COMPONENTS：在扩展URI变量后，使用UriComponents # encode（）（对应于先前列表中的第二个选项）对URI组件值进行编码。
- NONE：未应用编码。

由于历史原因和向后兼容性，将RestTemplate设置为EncodingMode.URI_COMPONENTS。 WebClient依赖于DefaultUriBuilderFactory中的默认值，该默认值已从5.0.x中的EncodingMode.URI_COMPONENTS更改为5.1中的EncodingMode.TEMPLATE_AND_VALUES。

1.6.4 Relative Servlet Requests

您可以使用ServletUriComponentsBuilder创建相对于当前请求的URI，如以下示例所示：

```
HttpServletRequest request = ...

// Re-uses host, scheme, port, path and query string...

ServletUriComponentsBuilder ucb = ServletUriComponentsBuilder.fromRequest(request)
        .replaceQueryParam("accountId", "{id}").build()
        .expand("123")
        .encode();
```

您可以创建相对于上下文路径的URI，如以下示例所示：

```
// Re-uses host, port and context path...

ServletUriComponentsBuilder ucb = ServletUriComponentsBuilder.fromContextPath(request)
        .path("/accounts").build()
```

您可以创建相对于Servlet的URI（例如/ main / *），如以下示例所示：

```
// Re-uses host, port, context path, and Servlet prefix...
```

更多教程请访问:www.flydean.com

```
ServletUriComponentsBuilder ucb = ServletUriComponentsBuilder.fromServletMapping(request)
        .path("/accounts").build()
```

> 从5.1开始，ServletUriComponentsBuilder会忽略来自Forwarded和X-Forwarded- *标头的信息，这些标头指定了客户端起源的地址。 考虑使用ForwardedHeaderFilter提取和使用或丢弃此类标头。

1.6.5 Links to Controllers

Spring MVC提供了一种准备到控制器方法的链接的机制。 例如，以下MVC控制器允许创建链接：

```
@Controller
@RequestMapping("/hotels/{hotel}")
public class BookingController {

    @GetMapping("/bookings/{booking}")
    public ModelAndView getBooking(@PathVariable Long booking) {
        // ...
    }
}
```

您可以通过按名称引用方法来准备链接，如以下示例所示：

```
UriComponents uriComponents = MvcUriComponentsBuilder
    .fromMethodName(BookingController.class, "getBooking", 21).buildAndExpand(42);

URI uri = uriComponents.encode().toUri();
```

在前面的示例中，我们提供了实际的方法参数值（在本例中为long值：21），用作路径变量并插入到URL中。 此外，我们提供值42来填充所有剩余的URI变量，例如从类型级别请求映射继承的hotel变量。 如果该方法具有更多参数，则可以为URL不需要的参数提供null。 通常，只有@PathVariable和@RequestParam参数与构造URL有关。

还有其他使用MvcUriComponentsBuilder的方法。 例如，您可以使用类似于代理的模拟测试技术来避免按名称引用控制器方法，如以下示例所示（该示例假定静态导入MvcUriComponentsBuilder.on）：

```
UriComponents uriComponents = MvcUriComponentsBuilder
    .fromMethodCall(on(BookingController.class).getBooking(21)).buildAndExpand(42);

URI uri = uriComponents.encode().toUri();
```

> 当控制器方法签名可用于fromMethodCall的链接创建时，其设计受到限制。 除了需要适当的参数签名外，返回类型还存在技术限制（即，为链接生成器调用生成运行时代理），因此返回类型不得为最终值。 特别是，视图名称的通用String返回类型在这里不起作用。 您应该改用ModelAndView甚至普通对象（具有String返回值）。

较早的示例在MvcUriComponentsBuilder中使用静态方法。 在内部，它们依靠ServletUriComponentsBuilder从当前请求的方案，主机，端口，上下文路径和Servlet路径准备基本URL。 在大多数情况下，此方法效果很好。 但是，有时可能不足。 例如，您可能不在请求的上下文之内（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，从请求路径中删除且需要重新设置的语言环境前缀）。 插入链接）。

在这种情况下，可以使用静态的fromXxx重载方法，这些方法接受UriComponentsBuilder以使用基本URL。 或者，您可以使用基本URL创建MvcUriComponentsBuilder的实例，然后使用基于实例的withXxx方法。 例如，以下清单使用withMethodCall：

```
UriComponentsBuilder base = ServletUriComponentsBuilder.fromCurrentContextPath().path("/en");
MvcUriComponentsBuilder builder = MvcUriComponentsBuilder.relativeTo(base);
builder.withMethodCall(on(BookingController.class).getBooking(21)).buildAndExpand(42);
```

更多教程请访问:www.flydean.com

```
URI uri = uriComponents.encode().toUri();
```

从5.1开始，MvcUriComponentsBuilder将忽略来自Forwarded和X-Forwarded- *标头的信息，这些标头指定了客户端起源的地址。 考虑使用ForwardedHeaderFilter提取和使用或丢弃此类标头。

1.6.6 Links in Views

在Thymeleaf，FreeMarker或JSP之类的视图中，您可以通过引用每个请求映射的隐式或显式分配的名称来构建到带注解的控制器的链接。

考虑以下示例：

```
@RequestMapping("/people/{id}/addresses")
public class PersonAddressController {

    @RequestMapping("/{country}")
    public HttpEntity getAddress(@PathVariable String country) { ... }
}
```

给定前面的控制器，您可以按照以下步骤准备来自JSP的链接：

```
<%@ taglib uri="http://www.springframework.org/tags" prefix="s" %>
...
<a href="${s:mvcUrl('PAC#getAddress').arg(0,'US').buildAndExpand('123')}">Get Address</a>
```

上面的示例依赖于Spring标记库中声明的mvcUrl函数（即META-INF / spring.tld），但是很容易定义您自己的函数或为其他模板技术准备类似的函数。

这是这样的。 在启动时，每个@RequestMapping都会通过HandlerMethodMappingNamingStrategy分配一个默认名称，该默认名称的实现使用类的大写字母和方法名称（例如，ThingController中的getThing方法变为“ TC # getThing”）。 如果名称冲突，则可以使用@RequestMapping（name =“ ..”）来分配一个明确的名称，或实现自己的HandlerMethodMappingNamingStrategy。

更多教程请访问:www.flydean.com

1.7 CORS

Spring WebFlux使您可以处理CORS（跨域资源共享）。本节介绍如何执行此操作。

1.7.1 介绍

出于安全原因，浏览器禁止AJAX调用当前来源以外的资源。例如，您可以将您的银行帐户放在一个标签中，将evil.com放在另一个标签中。来自evil.com的脚本不能使用您的凭据向您的银行API发出AJAX请求，例如从您的帐户中提取资金！

跨域资源共享（CORS）是大多数浏览器实施的W3C规范，可让您指定授权哪种类型的跨域请求，而不是使用基于IFRAME或JSONP的安全性较低且功能较弱的变通办法。

1.7.2 Processing

CORS规范区分飞行前，简单和实际请求。要了解CORS的工作原理，您可以阅读本文以及其他内容，或者参阅规范以获取更多详细信息。

Spring MVC HandlerMapping实现为CORS提供内置支持。成功将请求映射到处理程序后，HandlerMapping实现将检查给定请求和处理程序的CORS配置，并采取进一步的措施。飞行前请求直接处理，而简单和实际的CORS请求被拦截，验证并设置了必需的CORS响应标头。

为了启用跨域请求（即存在Origin标头，并且与请求的主机不同），您需要具有一些显式声明的CORS配置。如果找不到匹配的CORS配置，则预检请求将被拒绝。没有将CORS标头添加到简单和实际CORS请求的响应中，因此，浏览器拒绝了它们。

可以使用基于URL模式的CorsConfiguration映射分别配置每个HandlerMapping。在大多数情况下，应用程序使用MVC Java配置或XML名称空间声明此类映射，这导致将单个全局映射传递给所有HandlerMappping实例。

您可以将HandlerMapping级别的全局CORS配置与更细粒度的处理程序级别的CORS配置结合使用。例如，带注解的控制器可以使用类或方法级别的@CrossOrigin注解（其他处理程序可以实现CorsConfigurationSource）。

全局和本地配置组合的规则通常是相加的，例如，所有全局和所有本地起源。对于只能接受单个值的那些属性（例如allowCredentials和maxAge），局部属性将覆盖全局值。有关更多详细信息，请参见CorsConfiguration # combine（CorsConfiguration）。

1.7.3 @CrossOrigin

@CrossOrigin批注启用带注解的控制器方法上的跨域请求，如以下示例所示：

```
@RestController
@RequestMapping("/account")
public class AccountController {

    @CrossOrigin
    @GetMapping("/{id}")
    public Account retrieve(@PathVariable Long id) {
        // ...
    }

    @DeleteMapping("/{id}")
    public void remove(@PathVariable Long id) {
        // ...
```

更多教程请访问:www.flydean.com

```
    }
}
```

默认情况下，@ CrossOrigin允许：

- All origins.
- All headers.
- 控制器方法映射到的所有HTTP方法。

默认情况下，allowedCredentials未启用，因为它建立了一个信任级别，可以公开敏感的用户特定信息（例如cookie和CSRF令牌），并且仅在适当的地方使用。

maxAge设置为30分钟。

@CrossOrigin在类级别上也受支持，并且被所有方法继承，如以下示例所示：

```
@CrossOrigin(origins = "https://domain2.com", maxAge = 3600)
@RestController
@RequestMapping("/account")
public class AccountController {

    @GetMapping("/{id}")
    public Account retrieve(@PathVariable Long id) {
        // ...
    }

    @DeleteMapping("/{id}")
    public void remove(@PathVariable Long id) {
        // ...
    }
}
```

您可以在类级别和方法级别上使用@CrossOrigin，如以下示例所示：

```
@CrossOrigin(maxAge = 3600)
@RestController
@RequestMapping("/account")
public class AccountController {

    @CrossOrigin("https://domain2.com")
    @GetMapping("/{id}")
    public Account retrieve(@PathVariable Long id) {
        // ...
    }

    @DeleteMapping("/{id}")
    public void remove(@PathVariable Long id) {
        // ...
    }
}
```

1.7.4 Global Configuration

除了细粒度的控制器方法级别配置外，您可能还想定义一些全局CORS配置。 您可以在任何HandlerMapping上分别设置基于URL的CorsConfiguration映射。 但是，大多数应用程序使用MVC Java配置或MVC XNM命名空间来执行此操作。

默认情况下，全局配置启用以下功能：

- All origins.

更多教程请访问:www.flydean.com

- All headers.
- GET，HEAD和POST方法。

默认情况下，allowedCredentials未启用，因为它建立了一个信任级别，可以公开敏感的用户特定信息（例如cookie和CSRF令牌），并且仅在适当的地方使用。

maxAge设置为30分钟。

Java Configuration

要在MVC Java配置中启用CORS，可以使用CorsRegistry回调，如以下示例所示：

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {

        registry.addMapping("/api/**")
            .allowedOrigins("https://domain2.com")
            .allowedMethods("PUT", "DELETE")
            .allowedHeaders("header1", "header2", "header3")
            .exposedHeaders("header1", "header2")
            .allowCredentials(true).maxAge(3600);

        // Add more mappings...
    }
}
```

XML Configuration

要在XML名称空间中启用CORS，可以使用<mvc：cors>元素，如以下示例所示：

```
<mvc:cors>

    <mvc:mapping path="/api/**"
        allowed-origins="https://domain1.com, https://domain2.com"
        allowed-methods="GET, PUT"
        allowed-headers="header1, header2, header3"
        exposed-headers="header1, header2" allow-credentials="true"
        max-age="123" />

    <mvc:mapping path="/resources/**"
        allowed-origins="https://domain1.com" />

</mvc:cors>
```

1.7.5 CORS Filter

您可以通过内置的CorsFilter应用CORS支持。

> 如果您尝试将CorsFilter与Spring Security一起使用，请记住Spring Security内置了对CORS的支持。

要配置过滤器，请将CorsConfigurationSource传递给其构造函数，如以下示例所示：

```
CorsConfiguration config = new CorsConfiguration();

// Possibly...
// config.applyPermitDefaultValues()

config.setAllowCredentials(true);
```

更多教程请访问:www.flydean.com

```
config.addAllowedOrigin("https://domain1.com");
config.addAllowedHeader("*");
config.addAllowedMethod("*");

UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
source.registerCorsConfiguration("/**", config);

CorsFilter filter = new CorsFilter(source);
```

更多教程请访问:www.flydean.com

1.8 Web Security

Spring Security项目提供了保护Web应用程序免受恶意利用的支持。 请参阅Spring Security参考文档，包括：

- WebFlux安全
- WebFlux测试支持
- CSRF保护
- 安全响应标题

更多教程请访问:www.flydean.com

1.9 View Technologies

Spring WebFlux中视图技术的使用是可插入的，无论您决定使用Thymeleaf，Groovy标记模板，JSP还是其他技术，主要取决于配置更改。 本章介绍与Spring WebFlux集成的视图技术。 我们假设您已经熟悉View Resolution。

1.9.1 Thymeleaf

Thymeleaf是一种现代的服务器端Java模板引擎，它强调可以通过双击在浏览器中预览的自然HTML模板，这对于独立处理UI模板（例如，由设计人员）而无需使用非常有用。 正在运行的服务器。 如果要替换JSP，Thymeleaf提供了最广泛的功能集之一，以使这种过渡更加容易。 Thymeleaf是积极开发和维护的。 有关更完整的介绍，请参见Thymeleaf项目主页。

Thymeleaf与Spring MVC的集成由Thymeleaf项目管理。 该配置涉及一些Bean声明，例如ServletContextTemplateResolver，SpringTemplateEngine和ThymeleafViewResolver。 有关更多详细信息，请参见Thymeleaf + Spring。

1.9.2 FreeMarker

Apache FreeMarker是一个模板引擎，用于生成从HTML到电子邮件等的任何类型的文本输出。 Spring框架具有内置的集成，可以将Spring MVC与FreeMarker模板一起使用。

View配置

以下示例显示如何将FreeMarker配置为一种视图技术：

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.freemarker();
    }

    // Configure FreeMarker...

    @Bean
    public FreeMarkerConfigurer freeMarkerConfigurer() {
        FreeMarkerConfigurer configurer = new FreeMarkerConfigurer();
        configurer.setTemplateLoaderPath("/WEB-INF/freemarker");
        return configurer;
    }
}
```

您的模板需要存储在上例所示的FreeMarkerConfigurer指定的目录中。 给定上述配置，如果您的控制器返回欢迎的视图名称，则解析器将查找/WEB-INF/freemarker/welcome.ftl模板。

FreeMarker Configuration

您可以通过在FreeMarkerConfigurer bean上设置适当的bean属性，将FreeMarker的“设置”和“ SharedVariables”直接传递给FreeMarker配置对象（由Spring管理）。 freemarkerSettings属性需要一个java.util.Properties对象，而freemarkerVariables属性需要一个java.util.Map。 以下示例显示了如何使用FreeMarkerConfigurer：

```
@Configuration
@EnableWebFlux
```

更多教程请访问:www.flydean.com

```
public class WebConfig implements WebFluxConfigurer {

    // ...

    @Bean
    public FreeMarkerConfigurer freeMarkerConfigurer() {
        Map<String, Object> variables = new HashMap<>();
        variables.put("xml_escape", new XmlEscape());

        FreeMarkerConfigurer configurer = new FreeMarkerConfigurer();
        configurer.setTemplateLoaderPath("classpath:/templates");
        configurer.setFreemarkerVariables(variables);
        return configurer;
    }
}
```

有关设置和变量应用于Configuration对象的详细信息，请参见FreeMarker文档。

1.9.3 Script Views

Spring框架具有内置的集成，可以将Spring MVC与可以在JSR-223 Java脚本引擎之上运行的任何模板库一起使用。 我们已经在不同的脚本引擎上测试了以下模板库：

Scripting Library	Scripting Engine
Handlebars	Nashorn
Mustache	Nashorn
React	Nashorn
EJS	Nashorn
ERB	JRuby
String templates	Jython
Kotlin Script templating	Kotlin

> 集成任何其他脚本引擎的基本规则是，它必须实现ScriptEngine和Invocable接口。

Requirements

您需要在类路径上具有脚本引擎，其细节因脚本引擎而异：

- Java 8+随附了Nashorn JavaScript引擎。 强烈建议使用可用的最新更新版本。
- 应该将JRuby添加为对Ruby支持的依赖。
- 应该将Jython添加为对Python支持的依赖项。
- 应该添加org.jetbrains.kotlin：kotlin-script-util依赖项和包含org.jetbrains.kotlin.script.jsr223.KotlinJsr223JvmLocalScriptEngineFactory行的META-INF / services / javax.script.ScriptEngineFactory文件。 有关更多详细信息，请参见此示例。

您需要具有脚本模板库。 针对Javascript的一种方法是通过WebJars。

Script Templates

您可以声明一个ScriptTemplateConfigurer bean来指定要使用的脚本引擎，要加载的脚本文件，调用呈现模板的函数等等。 以下示例使用Mustache模板和Nashorn JavaScript引擎：

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {
```

更多教程请访问:www.flydean.com

```
    @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;
    }
}
```

使用以下参数调用render函数：

- String template：模板内容
- Map model：视图模型
- RenderingContext renderingContext：RenderingContext，用于访问应用程序上下文，语言环境，模板加载器和URL（自5.0起）

Mustache.render（）与该签名本地兼容，因此您可以直接调用它。

如果您的模板技术需要一些自定义，则可以提供一个实现自定义渲染功能的脚本。 例如，Handlerbars需要在使用模板之前先对其进行编译，并且需要使用polyfill来模拟某些服务器端脚本引擎中不可用的浏览器功能。

以下示例显示了如何执行此操作：

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @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;
    }
}
```

当您将非线程安全脚本引擎与不是为并发设计的模板库一起使用时，例如，在Nashorn上运行的Handlebars或React，必须将sharedEngine属性设置为false。 在这种情况下，由于此错误，需要Java 8u60或更高版本。

polyfill.js仅定义Handlebars正常运行所需的window对象，如下所示：

```
var window = {};
```

这个基本的render.js实现在使用模板之前先对其进行编译。 生产就绪的实现还应该存储任何重用的缓存模板或预编译的模板。 您可以在脚本方面进行操作（并处理所需的任何自定义，例如管理模板引擎配置）。 以下示例显示了如何执行此操作：

更多教程请访问:www.flydean.com

```
function render(template, model) {
    var compiledTemplate = Handlebars.compile(template);
    return compiledTemplate(model);
}
```

1.9.4 JSON and XML

出于内容协商的目的，根据客户端请求的内容类型，能够在使用HTML模板呈现模型或以其他格式（例如JSON或XML）呈现模型之间进行切换非常有用。为了支持此操作，Spring WebFlux提供了HttpMessageWriterView，您可以使用它插入spring-web中的任何可用编解码器，例如Jackson2JsonEncoder，Jackson2SmileEncoder或Jaxb2XmlEncoder。

与其他视图技术不同，HttpMessageWriterView不需要ViewResolver，而是配置为默认视图。您可以配置一个或多个此类默认视图，并包装不同的HttpMessageWriter实例或Encoder实例。在运行时使用与请求的内容类型匹配的内容。

在大多数情况下，模型包含多个属性。要确定要序列化的对象，可以使用模型属性的名称配置HttpMessageWriterView进行渲染。如果模型仅包含一个属性，则使用该属性。

更多教程请访问:www.flydean.com

1.10 HTTP Caching

HTTP缓存可以显着提高Web应用程序的性能。 HTTP缓存围绕Cache-Control响应标头以及随后的条件请求标头（例如Last-Modified和ETag）。 缓存控制为私有（例如浏览器）和公共（例如代理）缓存提供有关如何缓存和重用响应的建议。 ETag标头用于发出条件请求，如果内容未更改，则可能导致没有主体的304（NOT_MODIFIED）。 ETag可以看作是Last-Modified头的更复杂的后继者。

本节描述了Spring Web MVC中与HTTP缓存相关的选项。

1.10.1 CacheControl

CacheControl支持配置与Cache-Control标头相关的设置，并在许多地方作为参数被接受：

- WebContentInterceptor
- WebContentGenerator
- 控制器
- 静态资源

尽管RFC 7234描述了Cache-Control响应标头的所有可能的指令，但CacheControl类型采用了面向用例的方法，着重于常见方案：

```
// Cache for an hour - "Cache-Control: max-age=3600"
CacheControl ccCacheOneHour = CacheControl.maxAge(1, TimeUnit.HOURS);

// Prevent caching - "Cache-Control: no-store"
CacheControl ccNoStore = CacheControl.noStore();

// Cache for ten days in public and private caches,
// public caches should not transform the response
// "Cache-Control: max-age=864000, public, no-transform"
CacheControl ccCustom = CacheControl.maxAge(10, TimeUnit.DAYS).noTransform().cachePublic();
```

WebContentGenerator还接受一个更简单的cachePeriod属性（以秒为单位定义），该属性的工作方式如下：

- -1值不会生成Cache-Control响应标头。
- 值为0可以防止使用“ Cache-Control：无存储”指令进行缓存。
- n> 0值通过使用'Cache-Control：max-age = n'指令将给定响应缓存n秒。

1.10.2 控制器

控制器可以添加对HTTP缓存的显式支持。 我们建议您这样做，因为需要先计算资源的lastModified或ETag值，然后才能将其与条件请求标头进行比较。 控制器可以将ETag标头和Cache-Control设置添加到ResponseEntity，如以下示例所示：

```
@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
```

更多教程请访问:www.flydean.com

```
            .body(book);
}
```

如果与条件请求标头的比较表明内容未更改，则前面的示例发送带有空主体的304（NOT_MODIFIED）响应。 否则，ETag和Cache-Control标头将添加到响应中。

您还可以在控制器中针对条件请求标头进行检查，如以下示例所示：

```
@RequestMapping
public String myHandleMethod(WebRequest webRequest, Model model) {

    long eTag = ...

    if (request.checkNotModified(eTag)) {
        return null;
    }

    model.addAttribute(...);
    return "myViewName";
}
```

可以使用三种变体来检查针对eTag值，lastModified值或两者的条件请求。 对于有条件的GET和HEAD请求，可以将响应设置为304（NOT_MODIFIED）。 对于条件POST，PUT和DELETE，您可以将响应设置为409（PRECONDITION_FAILED），以防止并发修改。

1.10.3 Static Resources

您应该将静态资源与Cache-Control和条件响应标头一起提供，以实现最佳性能。 请参阅有关配置静态资源的部分。

更多教程请访问:www.flydean.com

1.11 WebFlux Config

WebFlux Java配置声明使用带注解的控制器或功能端点来声明处理请求所必需的组件，并且它提供了用于自定义配置的API。 这意味着您不需要了解Java配置创建的基础bean。 但是，如果您想了解它们，则可以在WebFluxConfigurationSupport中查看它们，或阅读有关特殊Bean类型中的内容的更多信息。

对于配置API中没有的更高级的自定义设置，您可以通过“高级配置模式”完全控制配置。

1.11.1 Enabling WebFlux Config

您可以在Java配置中使用@EnableWebFlux批注，如以下示例所示：

```
@Configuration
@EnableWebFlux
public class WebConfig {
}
```

前面的示例注册了许多Spring WebFlux基础结构Bean，并适应了classpath上可用的依赖项（对于JSON，XML等）。

1.11.2 WebFlux config API

在Java配置中，可以实现WebFluxConfigurer接口，如以下示例所示：

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    // Implement configuration methods...

}
```

1.11.3 Conversion, formatting

默认情况下，将安装Number和Date类型的格式化程序，包括对@NumberFormat和@DateTimeFormat批注的支持。 如果类路径中存在Joda-Time，则还将安装对Joda-Time格式库的完全支持。

下面的示例演示如何注册自定义格式器和转换器：

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void addFormatters(FormatterRegistry registry) {
        // ...
    }

}
```

> 有关何时使用FormatterRegistrar实现的更多信息，请参见FormatterRegistrar SPI和FormattingConversionServiceFactoryBean。

更多教程请访问:www.flydean.com

1.11.4 Validation

默认情况下，如果Bean验证存在于类路径中（例如，Hibernate Validator），则LocalValidatorFactoryBean将注册为全局验证器，以与@Valid和@Controller方法参数中的Validated一起使用。

在Java配置中，您可以自定义全局Validator实例，如以下示例所示：

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public Validator getValidator(); {
        // ...
    }

}
```

请注意，您还可以在本地注册Validator实现，如以下示例所示：

```
@Controller
public class MyController {

    @InitBinder
    protected void initBinder(WebDataBinder binder) {
        binder.addValidators(new FooValidator());
    }

}
```

如果需要在某处注入LocalValidatorFactoryBean，请创建一个bean并用@Primary进行标记，以避免与MVC配置中声明的bean发生冲突。

1.11.5 Content Type Resolvers

您可以配置Spring WebFlux如何根据请求为@Controller实例确定所请求的媒体类型。 默认情况下，仅选中Accept标头，但您也可以启用基于查询参数的策略。

以下示例显示如何自定义请求的内容类型解析：

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void configureContentTypeResolver(RequestedContentTypeResolverBuilder builder) {
        // ...
    }
}
```

1.11.6 HTTP message codecs

以下示例显示如何自定义读取和写入请求和响应主体的方式：

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {
```

更多教程请访问:www.flydean.com

```
    @Override
    public void configureHttpMessageCodecs(ServerCodecConfigurer configurer) {
        // ...
    }
}
```

ServerCodecConfigurer提供了一组默认的读取器和写入器。 您可以使用它来添加更多的读取器和写入器，自定义默认的读取器或完全替换默认的读取器和写入器。

对于Jackson JSON和XML，请考虑使用Jackson2ObjectMapperBuilder，该工具使用以下属性自定义Jackson的默认属性：

- DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES被禁用。
- MapperFeature.DEFAULT_VIEW_INCLUSION已禁用。

如果在类路径中检测到以下知名模块，它还将自动注册以下知名模块：

- jackson-datatype-jdk7：支持Java 7类型，例如java.nio.file.Path。
- jackson-datatype-joda：支持Joda-Time类型。
- jackson-datatype-jsr310：支持Java 8日期和时间API类型。
- jackson-datatype-jdk8：支持其他Java 8类型，例如Optional。

1.11.7 View Resolvers

下面的示例显示如何配置视图分辨率：

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        // ...
    }
}
```

ViewResolverRegistry具有与Spring Framework集成的视图技术的快捷方式。 以下示例使用FreeMarker（这也需要配置基础FreeMarker视图技术）：

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {


    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.freeMarker();
    }

    // Configure Freemarker...

    @Bean
    public FreeMarkerConfigurer freeMarkerConfigurer() {
        FreeMarkerConfigurer configurer = new FreeMarkerConfigurer();
        configurer.setTemplateLoaderPath("classpath:/templates");
        return configurer;
    }
}
```

更多教程请访问:www.flydean.com

您还可以插入任何ViewResolver实现，如以下示例所示：

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {


    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        ViewResolver resolver = ... ;
        registry.viewResolver(resolver);
    }
}
```

为了支持“内容协商”并通过视图解析（HTML之外）呈现其他格式，您可以基于HttpMessageWriterView实现配置一个或多个默认视图，该实现接受spring-web中的任何可用编解码器。 以下示例显示了如何执行此操作：

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {


    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.freeMarker();

        Jackson2JsonEncoder encoder = new Jackson2JsonEncoder();
        registry.defaultViews(new HttpMessageWriterView(encoder));
    }

    // ...
}
```

1.11.8 Static Resources

此选项提供了一种方便的方法来从基于资源的位置列表中提供静态资源。

在下一个示例中，给定一个以/ resources开头的请求，相对路径用于在类路径上查找和提供相对于/ static的静态资源。 资源的有效期为一年，以确保最大程度地利用浏览器缓存并减少浏览器发出的HTTP请求。 还评估Last-Modified头，如果存在，则返回304状态代码。 以下列表显示了示例：

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**")
            .addResourceLocations("/public", "classpath:/static/")
            .setCacheControl(CacheControl.maxAge(365, TimeUnit.DAYS));
    }

}
```

资源处理程序还支持一系列ResourceResolver实现和ResourceTransformer实现，可用于创建用于处理优化资源的工具链。

您可以根据从内容，固定应用程序版本或其他信息计算出的MD5哈希，将VersionResourceResolver用于版本化的资源URL。 ContentVersionStrategy（MD5哈希）是一个不错的选择，但有一些值得注意的例外（例如与模块加载器一起使用的JavaScript资源）。

更多教程请访问:www.flydean.com

以下示例显示如何在Java配置中使用VersionResourceResolver：

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**")
                .addResourceLocations("/public/")
                .resourceChain(true)
                .addResolver(new VersionResourceResolver().addContentVersionStrategy("/**"));
    }

}
```

您可以使用ResourceUrlProvider重写URL并应用完整的解析器和转换器链（例如，插入版本）。 WebFlux配置提供了ResourceUrlProvider，以便可以将其注入其他资源。

与Spring MVC不同，目前，在WebFlux中，由于没有视图技术可以利用解析器和转换器的无阻塞链，因此无法透明地重写静态资源URL。当仅提供本地资源时，解决方法是直接使用ResourceUrlProvider（例如，通过自定义元素）并进行阻止。

请注意，在同时使用EncodedResourceResolver（例如，Gzip，Brotli编码）和VersionedResourceResolver时，必须按该顺序注册它们，以确保始终基于未编码文件可靠地计算基于内容的版本。

WebJars还通过WebJarsResourceResolver支持，当org.webjars：webjars-locator-core库存在于类路径中时，WebJars将自动注册。解析程序可以重写URL以包括jar的版本，并且还可以与没有版本的传入URL进行匹配，例如，从/jquery/jquery.min.js到/jquery/1.2.0/jquery.min.js。

1.11.9 Path Matching

您可以自定义与路径匹配有关的选项。 有关各个选项的详细信息，请参见PathMatchConfigurer javadoc。 以下示例显示如何使用PathMatchConfigurer：

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void configurePathMatch(PathMatchConfigurer configurer) {
        configurer
            .setUseCaseSensitiveMatch(true)
            .setUseTrailingSlashMatch(false)
            .addPathPrefix("/api",
                    HandlerTypePredicate.forAnnotation(RestController.class));
    }

}
```

> Spring WebFlux依赖于请求路径的解析表示形式来访问解码的路径段值，该请求路径称为RequestPath，并且已删除了分号内容（即路径或矩阵变量）。 这意味着，与Spring MVC不同，您无需指示是否解码请求路径，也无需指示是否出于路径匹配目的而删除分号内容。
>
> Spring WebFlux还不支持后缀模式匹配，这与Spring MVC不同，在Spring MVC中，我们也建议不要依赖它。

1.11.10 Advanced Configuration Mode

更多教程请访问:www.flydean.com

@EnableWebFlux导入DelegatingWebFluxConfiguration：

- 为WebFlux应用程序提供默认的Spring配置
- 检测并委托给WebFluxConfigurer实现以自定义该配置。

对于高级模式，您可以删除@EnableWebFlux并直接从DelegatingWebFluxConfiguration扩展而不是实现WebFluxConfigurer，如以下示例所示：

```
@Configuration
public class WebConfig extends DelegatingWebFluxConfiguration {

    // ...

}
```

您可以将现有方法保留在Web Config中，但是现在您还可以覆盖基类中的bean声明，并且在类路径上仍然具有任意数量的其他WebMvcConfigurer实现。

更多教程请访问:www.flydean.com

1.12 HTTP/2

Servlet 4容器需要支持HTTP / 2，并且Spring Framework 5与Servlet API 4兼容。从编程模型的角度来看，应用程序不需要做任何特定的事情。 但是，有一些与服务器配置有关的注意事项。 有关更多详细信息，请参见HTTP / 2 Wiki页面。

当前，Spring WebFlux不支持Netty的HTTP / 2。 也没有支持以编程方式将资源推送到客户端。

更多教程请访问:www.flydean.com

2.WebClient

Spring WebFlux包括用于HTTP请求的反应性，非阻塞WebClient。 客户端具有功能性，流利的API，具有用于声明式组合的反应式类型，请参见反应式库。 WebFlux客户端和服务器依靠相同的非阻塞编解码器对请求和响应内容进行编码和解码。

WebClient在内部委托给HTTP客户端库。 默认情况下，它使用Reactor Netty，内置了对Jetty反应式HttpClient的支持，其他的则可以通过ClientHttpConnector插入。

2.1 Configuration

创建WebClient的最简单方法是通过静态工厂方法之一：

- WebClient.create（）
- WebClient.create（String baseUrl）

上面的方法使用默认设置的Reactor Netty HttpClient，并期望io.projectreactor.netty：reactor-netty位于类路径上。

您还可以将WebClient.builder（）与其他选项一起使用：

- uriBuilderFactory：定制的UriBuilderFactory用作基本URL。
- defaultHeader：每个请求的标题。
- defaultCookie：每个请求的cookie。
- defaultRequest：消费者自定义每个请求。
- filter：针对每个请求的客户端过滤器。
- exchangeStrategies：HTTP消息读取器/写入器定制。
- clientConnector：HTTP客户端库设置。

以下示例配置HTTP编解码器：

```
ExchangeStrategies strategies = ExchangeStrategies.builder()
         .codecs(configurer -> {
             // ...
         })
         .build();

 WebClient client = WebClient.builder()
         .exchangeStrategies(strategies)
         .build();
```

构建后，WebClient实例是不可变的。 但是，您可以克隆它并构建修改后的副本，而不会影响原始实例，如以下示例所示：

```
 WebClient client1 = WebClient.builder()
        .filter(filterA).filter(filterB).build();

WebClient client2 = client1.mutate()
        .filter(filterC).filter(filterD).build();

// client1 has filterA, filterB

// client2 has filterA, filterB, filterC, filter
```

2.1.1 Reactor Netty

更多教程请访问:www.flydean.com

要自定义Reactor Netty设置，只需提供一个预先配置的HttpClient：

```
HttpClient httpClient = HttpClient.create().secure(sslSpec -> ...);

 WebClient webClient = WebClient.builder()
         .clientConnector(new ReactorClientHttpConnector(httpClient))
         .build()
```

Resources

默认情况下，HttpClient会参与Reactor.netty.http.HttpResources中包含的全局Reactor Netty资源，包括事件循环线程和连接池。 这是推荐的模式，因为固定的共享资源是事件循环并发的首选。 在这种模式下，全局资源将保持活动状态，直到进程退出。

如果服务器为该进程计时，则通常无需显式关闭。 但是，如果服务器可以启动或停止进程内（例如，部署为WAR的Spring MVC应用程序），则可以声明类型为ReactorResourceFactory的Spring托管Bean，其中globalResources = true（默认值）以确保Reactor 关闭Spring ApplicationContext时，将关闭Netty全局资源，如以下示例所示：

```
@Bean
  public ReactorResourceFactory reactorResourceFactory() {
      return new ReactorResourceFactory();
  }
```

您也可以选择不参与全局Reactor Netty资源。 但是，在这种模式下，确保所有Reactor Netty客户端和服务器实例使用共享资源是您的重担，如以下示例所示：

```
 @Bean
  public ReactorResourceFactory resourceFactory() {
      ReactorResourceFactory factory = new ReactorResourceFactory();
      factory.setGlobalResources(false);
      return factory;
  }

  @Bean
  public WebClient webClient() {

      Function<HttpClient, HttpClient> mapper = client -> {
          // Further customizations...
      };

      ClientHttpConnector connector =
              new ReactorClientHttpConnector(resourceFactory(), mapper);

      return WebClient.builder().clientConnector(connector).build();
  }
```

Timeouts

要配置连接超时：

```
import io.netty.channel.ChannelOption;

HttpClient httpClient = HttpClient.create()
        .tcpConfiguration(client ->
                client.option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 10000));
```

要配置读取和/或写入超时值：

```
import io.netty.handler.timeout.ReadTimeoutHandler;
```

更多教程请访问:www.flydean.com

```
import io.netty.handler.timeout.WriteTimeoutHandler;

HttpClient httpClient = HttpClient.create()
        .tcpConfiguration(client ->
                client.doOnConnected(conn -> conn
                        .addHandlerLast(new ReadTimeoutHandler(10))
                        .addHandlerLast(new WriteTimeoutHandler(10))));
```

2.1.2 Jetty

以下示例显示如何自定义Jetty HttpClient设置：

```
HttpClient httpClient = new HttpClient();
   httpClient.setCookieStore(...);
   ClientHttpConnector connector = new JettyClientHttpConnector(httpClient);

   WebClient webClient = WebClient.builder().clientConnector(connector).build()
```

默认情况下，HttpClient创建自己的资源（执行程序，ByteBufferPool，调度程序），这些资源将保持活动状态，直到进程退出或调用stop（）为止。

您可以在Jetty客户端（和服务器）的多个实例之间共享资源，并通过声明JettyResourceFactory类型的Spring托管bean来确保在关闭Spring ApplicationContext时关闭资源，如以下示例所示：

```
 @Bean
   public JettyResourceFactory resourceFactory() {
       return new JettyResourceFactory();
   }

   @Bean
   public WebClient webClient() {

       Consumer<HttpClient> customizer = client -> {
           // Further customizations...
       };

       ClientHttpConnector connector =
               new JettyClientHttpConnector(resourceFactory(), customizer);

       return WebClient.builder().clientConnector(connector).build();
   }
```

2.2 retrieve()

Retrieve（）方法是获取响应主体并将其解码的最简单方法。 以下示例显示了如何执行此操作：

```
WebClient client = WebClient.create("https://example.org");

   Mono<Person> result = client.get()
           .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
           .retrieve()
           .bodyToMono(Person.class);
```

您还可以从响应中解码出一个对象流，如以下示例所示：

```
Flux<Quote> result = client.get()
           .uri("/quotes").accept(MediaType.TEXT_EVENT_STREAM)
           .retrieve()
           .bodyToFlux(Quote.class);
```

更多教程请访问:www.flydean.com

默认情况下，具有4xx或5xx状态代码的响应会导致WebClientResponseException或其HTTP状态特定的子类之一，例如WebClientResponseException.BadRequest，WebClientResponseException.NotFound等。 您还可以使用onStatus方法来自定义结果异常，如以下示例所示：

```
 Mono<Person> result = client.get()
        .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
        .retrieve()
        .onStatus(HttpStatus::is4xxServerError, response -> ...)
        .onStatus(HttpStatus::is5xxServerError, response -> ...)
        .bodyToMono(Person.class);
```

使用onStatus时，如果期望响应包含内容，则onStatus回调应使用它。 否则，内容将自动耗尽以确保释放资源。

2.3 exchange()

与检索方法相比，exchange（）方法提供了更多的控制。 以下示例等效于retrieve（），但也提供了对ClientResponse的访问：

```
Mono<Person> result = client.get()
        .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
        .exchange()
        .flatMap(response -> response.bodyToMono(Person.class));
```

在此级别，您还可以创建完整的ResponseEntity：

```
 Mono<ResponseEntity<Person>> result = client.get()
        .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
        .exchange()
        .flatMap(response -> response.toEntity(Person.class));
```

注意（与retrieve（）不同），对于exchange（），没有针对4xx和5xx响应的自动错误信号。 您必须检查状态码并决定如何进行。

> 使用exchange（）时，必须始终使用ClientResponse的body或toEntity方法中的任何一种，以确保释放资源并避免HTTP连接池的潜在问题。 如果不需要响应内容，则可以使用bodyToMono（Void.class）。 但是，如果响应中确实包含内容，则连接将关闭并且不会放回池中。

2.4 Request Body

可以从对象对请求主体进行编码，如以下示例所示：

```
Mono<Person> personMono = ... ;

   Mono<Void> result = client.post()
        .uri("/persons/{id}", id)
        .contentType(MediaType.APPLICATION_JSON)
        .body(personMono, Person.class)
        .retrieve()
        .bodyToMono(Void.class);
```

您还可以对对象流进行编码，如以下示例所示：

```
   Flux<Person> personFlux = ... ;
```

更多教程请访问:www.flydean.com

```
    Mono<Void> result = client.post()
            .uri("/persons/{id}", id)
            .contentType(MediaType.APPLICATION_STREAM_JSON)
            .body(personFlux, Person.class)
            .retrieve()
            .bodyToMono(Void.class);
```

或者，如果您具有实际值，则可以使用syncBody快捷方式，如以下示例所示：

```
Person person = ... ;

    Mono<Void> result = client.post()
            .uri("/persons/{id}", id)
            .contentType(MediaType.APPLICATION_JSON)
            .syncBody(person)
            .retrieve()
            .bodyToMono(Void.class)
```

2.4.1 Form Data

要发送表单数据，可以提供MultiValueMap <String，String>作为正文。 请注意，内容由FormHttpMessageWriter自动设置为application / x-www-form-urlencoded。 下面的示例演示如何使用MultiValueMap <String，String>：

```
     MultiValueMap<String, String> formData = ... ;

    Mono<Void> result = client.post()
            .uri("/path", id)
            .syncBody(formData)
            .retrieve()
            .bodyToMono(Void.class)
```

您还可以使用BodyInserters在线提供表单数据，如以下示例所示：

```
import static org.springframework.web.reactive.function.BodyInserters.*;

    Mono<Void> result = client.post()
            .uri("/path", id)
            .body(fromFormData("k1", "v1").with("k2", "v2"))
            .retrieve()
            .bodyToMono(Void.class);
```

2.4.2 Multipart Data

要发送多部分数据，您需要提供一个MultiValueMap <String，？ >，其值可以是代表零件内容的对象实例或代表零件内容和标题的HttpEntity实例。 MultipartBodyBuilder提供了一个方便的API来准备多部分请求。 下面的示例演示如何创建MultiValueMap <String，？ >：

```
 MultipartBodyBuilder builder = new MultipartBodyBuilder();
    builder.part("fieldPart", "fieldValue");
    builder.part("filePart", new FileSystemResource("...logo.png"));
    builder.part("jsonPart", new Person("Jason"));

    MultiValueMap<String, HttpEntity<?>> parts = builder.build();
```

在大多数情况下，您不必为每个部分指定Content-Type。 内容类型是根据选择用于对其进行序列化的HttpMessageWriter自动确定的，对于资源而言，取决于文件扩展名。 如有必要，您可以通过重载的构建器part方法之一显式提供MediaType以供每个零件使用。

更多教程请访问:www.flydean.com

准备好MultiValueMap之后，将其传递到WebClient的最简单方法是通过syncBody方法，如以下示例所示：

```
MultipartBodyBuilder builder = ...;

   Mono<Void> result = client.post()
           .uri("/path", id)
           .syncBody(builder.build())
           .retrieve()
           .bodyToMono(Void.class);
```

如果MultiValueMap包含至少一个非String值，它也可以表示常规表单数据（即application / x-www-form-urlencoded），则无需将Content-Type设置为multipart / form-data。 使用MultipartBodyBuilder时，总是这样，以确保HttpEntity包装器。

作为MultipartBodyBuilder的替代方案，您还可以通过内置的BodyInserters提供内联样式的多部分内容，如以下示例所示：

```
import static org.springframework.web.reactive.function.BodyInserters.*;

   Mono<Void> result = client.post()
           .uri("/path", id)
           .body(fromMultipartData("fieldPart", "value").with("filePart", resource))
           .retrieve()
           .bodyToMono(Void.class);
```

2.5 Client Filters

您可以通过WebClient.Builder注册客户端过滤器（ExchangeFilterFunction），以拦截和修改请求，如以下示例所示：

```
WebClient client = WebClient.builder()
        .filter((request, next) -> {

            ClientRequest filtered = ClientRequest.from(request)
                    .header("foo", "bar")
                    .build();

            return next.exchange(filtered);
        })
        .build();
```

这可以用于跨领域的关注，例如身份验证。 以下示例使用过滤器通过静态工厂方法进行基本身份验证：

```
WebClient client = WebClient.builder()
        .filter(basicAuthentication("user", "password"))
        .build();
```

过滤器全局应用于每个请求。 要更改特定请求的过滤器行为，您可以将请求属性添加到ClientRequest，然后链中的所有过滤器都可以访问该请求属性，如以下示例所示：

```
WebClient client = WebClient.builder()
        .filter((request, next) -> {
            Optional<Object> usr = request.attribute("myAttribute");
            // ...
        })
        .build();

client.get().uri("https://example.org/")
        .attribute("myAttribute", "...")
        .retrieve()
```

更多教程请访问:www.flydean.com

```
        .bodyToMono(Void.class);

    }
```

您还可以复制现有的WebClient，插入新的过滤器或删除已注册的过滤器。 以下示例在索引0处插入一个基本身份验证过滤器：

```
WebClient client = webClient.mutate()
        .filters(filterList -> {
            filterList.add(0, basicAuthentication("user", "password"));
        })
        .build();
```

2.6 Synchronous Use

通过在结果末尾进行阻塞，可以以同步方式使用WebClient：

```
Person person = client.get().uri("/person/{id}", i).retrieve()
    .bodyToMono(Person.class)
    .block();

List<Person> persons = client.get().uri("/persons").retrieve()
    .bodyToFlux(Person.class)
    .collectList()
    .block();
```

但是，如果需要进行多次通话，则可以避免单独阻止每个响应，而等待合并的结果，这样会更有效：

```
Mono<Person> personMono = client.get().uri("/person/{id}", personId)
        .retrieve().bodyToMono(Person.class);

Mono<List<Hobby>> hobbiesMono = client.get().uri("/person/{id}/hobbies", personId)
        .retrieve().bodyToFlux(Hobby.class).collectList();

Map<String, Object> data = Mono.zip(personMono, hobbiesMono, (person, hobbies) -> {
            Map<String, String> map = new LinkedHashMap<>();
            map.put("person", personName);
            map.put("hobbies", hobbies);
            return map;
        })
        .block();
```

以上仅是一个示例。 还有许多其他模式和运算符可用于构建响应式管道，该响应式管道可进行许多远程调用（可能是嵌套的，相互依赖的），而不会阻塞到最后。

> 您永远不必阻塞Spring MVC控制器。 只需从控制器方法返回结果Flux或Mono。

2.7 Testing

要测试使用WebClient的代码，可以使用模拟Web服务器，例如OkHttp MockWebServer。 要查看其用法示例，请查看Spring Framework测试套件中的WebClientIntegrationTests或OkHttp存储库中的静态服务器示例。

更多教程请访问:www.flydean.com

更多教程请访问:www.flydean.com

3.WebSockets

3.1 WebSocket介绍

WebSocket协议RFC 6455提供了一种标准化方法，可通过单个TCP连接在客户端和服务器之间建立全双工双向通信通道。 它是与HTTP不同的TCP协议，但旨在通过端口80和443在HTTP上工作，并允许重复使用现有的防火墙规则。

WebSocket交互始于一个HTTP请求，该请求使用HTTP Upgrade标头进行升级，或在这种情况下切换到WebSocket协议。 以下示例显示了这种交互：

```
GET /spring-websocket-portfolio/portfolio HTTP/1.1
Host: localhost:8080
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: Uc9l9TMkWGbHFD2qnFHltg==
Sec-WebSocket-Protocol: v10.stomp, v11.stomp
Sec-WebSocket-Version: 13
Origin: http://localhost:8080
```

具有WebSocket支持的服务器代替通常的200状态代码，返回的输出类似于以下内容：

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: 1qVdfYHU9hPOl4JYYNXF623Gzn0=
Sec-WebSocket-Protocol: v10.stomp
```

成功握手后，HTTP升级请求的基础TCP套接字将保持打开状态，客户端和服务器均可继续发送和接收消息。

WebSockets的工作原理的完整介绍超出了本文档的范围。 请参阅RFC 6455，HTML5的WebSocket章节或Web上的许多简介和教程中的任何一个。

请注意，如果WebSocket服务器在Web服务器（例如nginx）后面运行，则可能需要对其进行配置，以将WebSocket升级请求传递到WebSocket服务器。 同样，如果应用程序在云环境中运行，请检查与WebSocket支持相关的云提供商的说明。

3.1.1 HTTP 和 WebSocket

尽管WebSocket被设计为与HTTP兼容并以HTTP请求开头，但重要的是要了解这两个协议导致了截然不同的体系结构和应用程序编程模型。

在HTTP和REST中，应用程序被建模为许多URL。为了与应用程序交互，客户端访问那些URL，即请求-响应样式。服务器根据HTTP URL，方法和标头将请求路由到适当的处理程序。

相反，在WebSockets中，通常只有一个URL用于初始连接。随后，所有应用程序消息在同一TCP连接上流动。这指向了一个完全不同的异步，事件驱动的消息传递体系结构。

WebSocket也是一种低级传输协议，与HTTP不同，它不对消息的内容规定任何语义。这意味着除非客户端和服务器就消息语义达成一致，否则就无法路由或处理消息。

WebSocket客户端和服务器可以通过HTTP握手请求上的Sec-WebSocket-Protocol标头，协商使用更高级别的消息传递协议（例如，STOMP）。在这种情况下，他们需要提出自己的约定。

3.1.2 什么时候使用WebSockets

更多教程请访问:www.flydean.com

WebSockets可以使网页具有动态性和交互性。但是，在许多情况下，结合使用Ajax和HTTP流或长轮询可以提供一种简单有效的解决方案。

例如，新闻，邮件和社交订阅源需要动态更新，但是每几分钟进行一次更新可能是完全可以的。另一方面，协作，游戏和金融应用程序需要更接近实时。

仅延迟并不是决定因素。如果消息量相对较少（例如，监视网络故障），则HTTP流或轮询可以提供有效的解决方案。低延迟，高频率和高音量的结合才是使用WebSocket的最佳案例。

还请记住，在Internet上，控件之外的限制性代理可能会阻止WebSocket交互，这可能是因为未将它们配置为传递Upgrade标头，或者是因为它们关闭了长期处于空闲状态的连接。这意味着与面向公众的应用程序相比，将WebSocket用于防火墙内部的应用程序是一个更直接的决定。

3.2 WebSocket API

Spring框架提供了一个WebSocket API，可用于编写处理WebSocket消息的客户端和服务器端应用程序。

3.2.1. Server

要创建WebSocket服务器，您可以首先创建WebSocketHandler。 以下示例显示了如何执行此操作：

```
import org.springframework.web.reactive.socket.WebSocketHandler;
import org.springframework.web.reactive.socket.WebSocketSession;

public class MyWebSocketHandler implements WebSocketHandler {

    @Override
    public Mono<Void> handle(WebSocketSession session) {
        // ...
    }
}
```

然后，可以将其映射到URL并添加一个WebSocketHandlerAdapter，如以下示例所示：

```
@Configuration
static class WebConfig {

    @Bean
    public HandlerMapping handlerMapping() {
        Map<String, WebSocketHandler> map = new HashMap<>();
        map.put("/path", new MyWebSocketHandler());

        SimpleUrlHandlerMapping mapping = new SimpleUrlHandlerMapping();
        mapping.setUrlMap(map);
        mapping.setOrder(-1); // before annotated controllers
        return mapping;
    }

    @Bean
    public WebSocketHandlerAdapter handlerAdapter() {
        return new WebSocketHandlerAdapter();
    }
}
```

3.2.2 WebSocketHandler

WebSocketHandler的handle方法采用WebSocketSession并返回Mono <Void>来指示会话的应用程序处理何时完成。通过两个流处理会话，一个流用于入站消息，一个流用于出站消息。 下表描述了两种处理流的方法：

更多教程请访问:www.flydean.com

WebSocketSession method	Description
Flux<WebSocketMessage> receive()	提供对入站消息流的访问，并在关闭连接时完成。
Mono<Void> send(Publisher<WebSocketMessage>)	获取传出消息的源，编写消息，并返回一个Mono <Void>，当源完成并完成写入时，Mono <Void>完成。

WebSocketHandler必须将入站和出站流组成一个统一的流，并返回反映该流完成情况的Mono <Void>。 根据应用程序要求，统一流程在以下情况下完成：

- 入站或出站消息流都已完成。
- 入站流完成（即，连接已关闭），而出站流是无限的。
- 在选定的时间点，通过WebSocketSession的close方法。

将入站和出站消息流组合在一起时，无需检查连接是否打开，因为响应式流信号会终止活动。 入站流接收完成或错误信号，而出站流接收取消信号。

处理程序最基本的实现是处理入站流的实现。 以下示例显示了这样的实现：

```
class ExampleHandler implements WebSocketHandler {

    @Override
    public Mono<Void> handle(WebSocketSession session) {
        return session.receive()
                .doOnNext(message -> {
                    // ...
                })
                .concatMap(message -> {
                    // ...
                })
                .then();
    }
}
```

> 对于嵌套的异步操作，您可能需要在使用池化数据缓冲区的底层服务器（例如Netty）上调用message.retain（）。 否则，在您有机会读取数据之前，可能会释放数据缓冲区。 有关更多背景信息，请参见数据缓冲区和编解码器。

以下实现将入站和出站流组合在一起：

```
class ExampleHandler implements WebSocketHandler {

    @Override
    public Mono<Void> handle(WebSocketSession session) {

        Flux<WebSocketMessage> output = session.receive()
                .doOnNext(message -> {
                    // ...
                })
                .concatMap(message -> {
                    // ...
                })
                .map(value -> session.textMessage("Echo " + value));

        return session.send(output);
    }
}
```

入站和出站流可以是独立的，并且只能为了完成而加入，如以下示例所示：

```
class ExampleHandler implements WebSocketHandler {

    @Override
```

更多教程请访问:www.flydean.com

```
    public Mono<Void> handle(WebSocketSession session) {

        Mono<Void> input = session.receive()
                .doOnNext(message -> {
                    // ...
                })
                .concatMap(message -> {
                    // ...
                })
                .then();

        Flux<String> source = ... ;
        Mono<Void> output = session.send(source.map(session::textMessage));

        return Mono.zip(input, output).then();
    }
}
```

3.2.3 DataBuffer

DataBuffer是WebFlux中字节缓冲区的表示形式。 该参考书的Spring Core部分在有关数据缓冲区和编解码器的部分中有更多内容。 要理解的关键点是，在诸如Netty之类的某些服务器上，字节缓冲区被池化并且对引用计数进行计数，并且在消耗字节缓冲区时必须将其释放以避免内存泄漏。

在Netty上运行时，如果应用程序希望保留输入数据缓冲区以确保它们不被释放，则必须使用DataBufferUtils.retain（dataBuffer），并在使用完缓冲区后随后使用DataBufferUtils.release（dataBuffer）。

3.2.4 Handshake

WebSocketHandlerAdapter委托给WebSocketService。 默认情况下，它是HandshakeWebSocketService的实例，该实例对WebSocket请求执行基本检查，然后对所使用的服务器使用RequestUpgradeStrategy。 当前，内置了对Reactor Netty，Tomcat，Jetty和Undertow的支持。

HandshakeWebSocketService公开了一个sessionAttributePredicate属性，该属性允许设置Predicate <String>从WebSession中提取属性并将其插入WebSocketSession的属性中。

3.2.5 Server Configation

每个服务器的RequestUpgradeStrategy公开了可用于基础WebSocket引擎的WebSocket相关配置选项。 以下示例在Tomcat上运行时设置WebSocket选项：

```
@Configuration
static class WebConfig {

    @Bean
    public WebSocketHandlerAdapter handlerAdapter() {
        return new WebSocketHandlerAdapter(webSocketService());
    }

    @Bean
    public WebSocketService webSocketService() {
        TomcatRequestUpgradeStrategy strategy = new TomcatRequestUpgradeStrategy();
        strategy.setMaxSessionIdleTimeout(0L);
        return new HandshakeWebSocketService(strategy);
    }
}
```

检查服务器的升级策略，以查看可用的选项。 当前，只有Tomcat和Jetty公开了此类选项。

更多教程请访问:www.flydean.com

3.2.6 CORS

配置CORS并限制对WebSocket端点的访问的最简单方法是让WebSocketHandler实现CorsConfigurationSource并返回带有允许的源，标头和其他详细信息的CorsConfiguraiton。 如果无法执行此操作，则还可以在SimpleUrlHandler上设置corsConfigurations属性，以通过URL模式指定CORS设置。 如果同时指定了两者，则使用CorsConfiguration上的Combine方法将它们合并。

3.2.7 Client

Spring WebFlux为WebSocketClient抽象提供了Reactor Netty，Tomcat，Jetty，Undertow和标准Java（即JSR-356）的实现。

> Tomcat客户端实际上是标准Java客户端的扩展，在WebSocketSession处理中具有一些额外功能，以利用特定于Tomcat的API暂停接收消息以产生反压。

要启动WebSocket会话，您可以创建客户端的实例并使用其execute方法：

```
WebSocketClient client = new ReactorNettyWebSocketClient();

URI url = new URI("ws://localhost:8080/path");
client.execute(url, session ->
        session.receive()
                .doOnNext(System.out::println)
                .then());
```

一些客户端（例如Jetty）实现了生命周期，需要先停止和启动，然后才能使用它们。 所有客户端都具有与基础WebSocket客户端的配置有关的构造器选项。

更多教程请访问:www.flydean.com

4.测试

Spring测试模块提供ServerHttpRequest，ServerHttpResponse和ServerWebExchange的模拟实现。 有关模拟对象的讨论，请参见Spring Web Reactive。

WebTestClient建立在这些模拟请求和响应对象的基础上，以提供对测试不使用HTTP服务器的WebFlux应用程序的支持。 您也可以将WebTestClient用于端到端集成测试。

更多教程请访问:www.flydean.com

5.Reactive库

spring-webflux依赖于反应堆核心，并在内部使用它来构成异步逻辑并提供Reactive Streams支持。通常，WebFlux API返回Flux或Mono（因为它们在内部使用），并且宽容地接受任何Reactive Streams Publisher实现作为输入。 Flux vs Mono的使用很重要，因为它有助于表达基数，例如，期望单个或多个异步值，并且对于决策（例如在编码或解码HTTP消息时）至关重要。

对于带注解的控制器，WebFlux透明地适应应用程序选择的反应式库。这是在ReactiveAdapterRegistry的帮助下完成的，该工具为反应性库和其他异步类型提供了可插入的支持。该注册表具有对RxJava和CompletableFuture的内置支持，但是您也可以注册其他注册表。

对于功能性API（例如功能性端点，WebClient等），WebFlux API的一般规则适用于-Flux和Mono作为返回值，而Reactive Streams Publisher作为输入。提供发布者（无论是自定义的还是来自其他反应性库的发布者）时，只能将其视为语义未知（0..N）的流。但是，如果知道语义，则可以用Flux或Mono.from（Publisher）包装它，而不用传递原始Publisher。

例如，给定的发布者不是Mono，Jackson JSON消息编写者需要多个值。如果媒体类型暗示无限流（例如application / json + stream），则将分别写入和刷新值。否则，值将缓冲到列表中并呈现为JSON数组。