

# Spring Framework 中文文档

---



文档名称：Spring Framework 中文文档

版本：5.1.3.RELEASE

语言：中文

在线浏览地址：[在线浏览](#)

官方地址：[官方地址](#)

最后更新时间：2019-03-03 22:56:43

关注微信公众号在线浏览文档

---

# Spring 框架概述

---

Spring 使创建 Java 企业应用程序变得容易。它提供了在企业环境中使用 Java 语言所需的一切，并支持 Groovy 和 Kotlin 作为 JVM 上的替代语言，并且可以根据应用程序的需求灵活地创建多种体系结构。从 Spring Framework 5.0 开始，Spring 需要 JDK 8(Java SE 8)，并且已经为 JDK 9 提供了现成的支持。

Spring 支持广泛的应用场景。在大型企业中，应用程序通常存在很长时间，并且必须在升级周期不受开发人员控制的 JDK 和应用程序服务器上运行。其他服务器则可以作为单个 jar 运行，并且服务器可以嵌入云环境中。还有一些可能是不需要服务器的独立应用程序(例如批处理或集成工作负载)。

Spring 是开源的。它拥有一个庞大而活跃的社区，可以根据各种实际用例提供持续的反馈。这帮助 Spring 在很长一段时间内成功地 Developing 了。

## 1.“Spring”的含义

---

术语 “Spring”在不同的上下文中表示不同的事物。它可以用来引用 Spring Framework 项目本身，而这一切都是从那里开始的。随着时间的流逝，其他 Spring 项目已经构建在 Spring Framework 之上。通常，当人们说 “Spring”时，它们表示整个项目系列。本参考文档重点关注基础：Spring 框架本身。

Spring 框架分为多个模块。应用程序可以选择所需的模块。核心容器的模块是核心，包括配置模型和依赖项注入机制。除此之外，Spring 框架为不同的应用程序体系结构提供了基础支持，包括消息传递，事务性数据和持久性以及 Web。它还包括基于 Servlet 的 Spring MVC Web 框架，以及并行的 Spring WebFlux 反应式 Web 框架。

关于模块的 Comments：Spring 的框架 jar 允许部署到 JDK 9 的模块路径(“拼图”)。为了在启用了 Jigsaw 的应用程序中使用，Spring Framework 5 jar 附带了“自动模块名称” Lists 条目，这些 Lists 条目定义了与 jar 工件无关的稳定语言级别的模块名称(“spring.core”，“spring.context”等)。名称(罐子遵循相同的命名模式，用“-”代替“.”，例如“spring-core”和“spring-

context")。当然，Spring 的框架 jar 可以在 JDK 8 和 9 的 Classpath 上正常工作。

## 2. Spring 和 Spring 框架的历史

---

响应于早期[J2EE](#)规范的复杂性，2003 年 Spring 应运而生。尽管有些人认为 Java EE 和 Spring 在竞争中，但 Spring 实际上是 Java EE 的补充。Spring 编程模型不包含 Java EE 平台规范。相反，它与 EE 伞中精心选择的各个规范集成在一起：

- Servlet API([JSR 340](#))
- WebSocket API([JSR 356](#))
- 并发 Util([JSR 236](#))
- JSON 绑定 API([JSR 367](#))
- Bean 验证([JSR 303](#))
- JPA ([JSR 338](#))
- JMS ([JSR 914](#))
- 以及必要时用于事务协调的 JTA/JCA 设置。

Spring 框架还支持依赖注入([JSR 330](#))和通用 Comments([JSR 250](#))规范，应用程序开发人员可以选择使用这些规范，而不是使用 Spring 框架提供的特定于 Spring 的机制。

从 Spring Framework 5.0 开始，Spring 至少需要 Java EE 7 级别(例如 Servlet 3.1, JPA 2.1)-同时提供与 Java EE 8 级别的较新 API 的现成集成。Servlet 4.0, JSON 绑定 API 在运行时遇到。这样可以使 Spring 与例如 Tomcat 8 和 9, WebSphere 9 和 JBoss EAP 7.

随着时间的流逝，Java EE 在应用程序开发中的作用已经演变。在 Java EE 和 Spring 的早期，创建了应用程序以将其部署到应用程序服务器。今天，借助 Spring Boot，可以以对开发人员和云友好的方式创建应用程序，并嵌入 Servlet 容器，并且可以轻松更改。从 Spring Framework 5 开始，WebFlux 应用程序甚至不直接使用 Servlet API，并且可以在非 Servlet 容器的服务器(例如 Netty)上运行。

Spring continue 创新和 Developing。除了 Spring Framework, 还有其他项目, 例如 Spring Boot, Spring Security, Spring Data, Spring Cloud, Spring Batch 等。重要的是要记住, 每个项目都有其自己的源代码存储库, 问题跟踪程序和发布节奏。有关 Spring 项目的完整列表, 请参见 [spring.io/projects](https://spring.io/projects)。

## 3.设计哲学

---

当您了解框架时, 不仅要了解框架的工作而且要遵循的原则很重要。以下是 Spring 框架的指导原则:

- 提供每个级别的选择。 Spring 使您可以尽可能推迟设计决策。例如, 您可以在不更改代码的情况下通过配置切换持久性提供程序。对于许多其他基础架构问题以及与第三方 API 的集成也是如此。
- 适应不同的观点。 Spring 拥有灵 Active, 并且对如何完成工作一无所知。它从不同的角度支持广泛的应用程序需求。
- 保持强大的向后兼容性。对 Spring 的 Developing 进行了精心 Management, 以使各个版本之间几乎没有重大更改。 Spring 支持精心选择的 JDK 版本和第三方库, 以方便维护依赖于 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, 并在许多版本和很多年中都适用的 API。
- 为代码质量设置高标准。 Spring 框架非常强调有意义, 最新和准确的 javadoc。它是极少数可以声明干净代码结构且程序包之间没有循环依赖关系的项目之一。

## 4.反馈和贡献

---

对于操作方法问题或诊断或调试问题, 我们建议使用 StackOverflow, 并且有一个[questions page](#)列出了要使用的建议标签。如果您可以肯定地确定 Spring 框架中有问题或想提出一个功能, 请使用 [JIRA 问题追踪器](#)。

如果您有解决方案或建议的解决方案，可以在[Github](#)上提交拉取请求。但是，请记住，对于除最琐碎的问题以外的所有问题，我们希望在问题跟踪器中记录故障单，在该跟踪器中进行讨论并保留记录以备将来参考。

有关更多详细信息，请参见[CONTRIBUTING](#)顶级项目页面上的准则。

## 5.入门

---

如果您刚刚开始使用 Spring，则可能需要通过创建基于[Spring Boot](#)的应用程序来开始使用 Spring Framework。Spring Boot 提供了一种快速(且自以为是)的方法来创建可用于生产的基于 Spring 的应用程序。它基于 Spring 框架，更倾向于约定而不是配置，并且旨在使您尽快启动并运行。

您可以使用[start.spring.io](#)生成基本项目，也可以遵循“入门”指南之一，例如[开始构建 RESTful Web 服务](#)。这些指南不仅易于理解，而且非常注重任务，并且大多数基于 Spring Boot。它们还涵盖了 Spring 产品组合中的其他项目，您在解决特定问题时可能要考虑这些项目。

# Core Technologies

---

参考文档的这一部分涵盖了 Spring 框架必不可少的所有技术。

其中最重要的是 Spring 框架的控制反转(IoC)容器。对 Spring 框架的 IoC 容器进行彻底处理之后，将全面介绍 Spring 的面向方面编程(AOP)技术。Spring 框架具有自己的 AOP 框架，该框架在概念上易于理解，并且成功解决了 Java 企业编程中 AOP 要求的 80% 的难题。

还提供了 Spring 与 AspectJ 的集成(就功能而言，目前是功能最丰富的 Java，当然还有 Java 企业领域中最成熟的 AOP 实现)。

## 1. IoC 容器

---

本章介绍了 Spring 的控制反转(IoC)容器。

### 1.1. Spring IoC 容器和 Bean 简介

本章介绍了反转控制(IoC)原则的 Spring 框架实现。 (请参阅[控制反转](#)。)IoC 也称为依赖项注入(DI)。在此过程中，对象仅通过构造函数参数，工厂方法的参数或在构造或从工厂方法返回后在对象实例上设置的属性来定义其依赖项(即，与它们一起使用的其他对象)。然后，容器在创建 bean 时注入那些依赖项。此过程从根本上讲是通过使用类的直接构造或诸如服务定位器模式之类的控件来控制其依赖项的实例化或位置的 bean 本身的逆过程(因此称为 Control Inversion)。

`org.springframework.beans` 和 `org.springframework.context` 软件包是 Spring Framework 的 IoC 容器的基础。 [BeanFactory](#) 接口提供了一种高级配置机制，能够 Management 任何类型的对象。 [ApplicationContext](#) 是 `BeanFactory` 的子接口。它增加了：

- 与 Spring 的 AOP 功能轻松集成
- 消息资源处理(用于国际化)
- Event publication
- 特定于应用程序层的上下文，例如用于 Web 应用程序的 `WebApplicationContext`。

简而言之，`BeanFactory` 提供了配置框架和基本功能，而 `ApplicationContext` 添加了更多企业特定的功能。`ApplicationContext` 是 `BeanFactory` 的完整超集，在本章中仅使用 Spring 的 IoC 容器描述。有关使用 `BeanFactory` 而不是 `ApplicationContext` 的更多信息，请参见[The BeanFactory](#)。

在 Spring 中，构成应用程序主干并由 Spring IoC 容器 Management 的对象称为 bean。 Bean 是由 Spring IoC 容器实例化，组装和以其他方式 Management 的对象。否则，bean 仅仅是应用程序中许多对象之一。 Bean 及其之间的依赖关系反映在容器使用的配置元数据中。

## 1.2. 容器概述

`org.springframework.context.ApplicationContext` 接口代表 Spring IoC 容器，并负责实例化，配置和组装 Bean。容器通过读取配置元数据来获取有关要实例化，配置和组装哪些对象的指令。配置元数据以 XML，Java 注解或 Java 代码表示。它使您能够表达组成应用程序的对象以及这

些对象之间的丰富相互依赖关系。

Spring 提供了 `ApplicationContext` 接口的几种实现。在独立应用程序中，通常创建 `ClassPathXmlApplicationContext` 或 `FileSystemXmlApplicationContext` 的实例。尽管 XML 是定义配置元数据的传统格式，但是您可以通过提供少量 XML 配置来声明性地启用对这些其他元数据格式的支持，从而指示容器将 JavaComments 或代码用作元数据格式。

在大多数应用场景中，不需要实例化用户代码即可实例化一个 Spring IoC 容器的一个或多个实例。

例如，在 Web 应用程序场景中，应用程序 `web.xml` 文件中的简单八行(约)样板

`WebDescriptorsXML` 通常就足够了(请参阅[Web 应用程序的便捷 ApplicationContext 实例化](#))。如果使用[Spring 工具套件](#)(基于 Eclipse 的开发环境)，则只需单击几次鼠标或击键即可轻松创建此样板配置。

下图显示了 Spring 的工作原理的高级视图。您的应用程序类与配置元数据结合在一起，以便在创建和初始化 `ApplicationContext` 之后，您将具有完全配置且可执行的系统或应用程序。

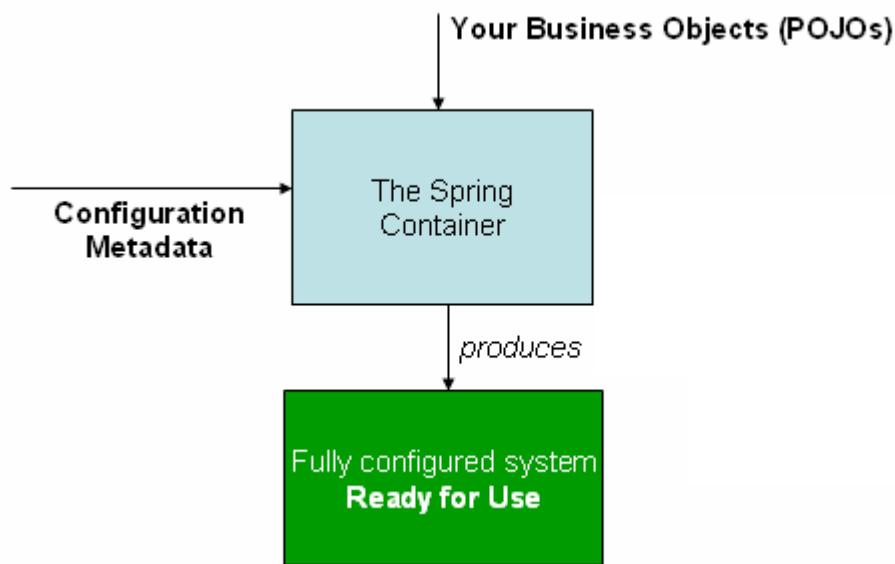


图 1. Spring IoC 容器

### 1.2.1. 配置元数据

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

传统上，配置元数据以简单直观的 XML 格式提供，这是本章大部分内容用来传达 Spring IoC 容器的关键概念和功能的内容。

### iNote

基于 XML 的元数据不是配置元数据的唯一允许形式。Spring IoC 容器本身与实际写入此配置元数据的格式完全脱钩。如今，许多开发人员为自己的 Spring 应用程序选择[Java-based configuration](#)。

有关在 Spring 容器中使用其他形式的元数据的信息，请参见：

- [Annotation-based configuration](#): Spring 2.5 引入了对基于 Comments 的配置元数据的支持。
  -
- [Java-based configuration](#): 从 Spring 3.0 开始，Spring JavaConfig 项目提供的许多功能成为了核心 Spring Framework 的一部分。因此，您可以使用 Java 而不是 XML 文件来定义应用程序类外部的 bean。要使用这些新功能，请参见[@Configuration](#), [@Bean](#), [@Import](#) 和 [@DependsOn](#) 注解。

Spring 配置由容器必须 Management 的至少一个(通常是一个以上)bean 定义组成。基于 XML 的配置元数据将这些 bean 配置为顶级 `<beans/>` 元素内的 `<bean/>` 元素。Java 配置通常在 `@Configuration` 类中使用 `@Bean` Comments 的方法。

这些 bean 定义对应于组成应用程序的实际对象。通常，您定义服务层对象，数据访问对象(DAO)，表示对象(例如 Struts `Action` 实例)，基础结构对象(例如 Hibernate `SessionFactories`)，JMS `Queues` 等)。通常，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，您可以使用 Spring 与 AspectJ 的集成来配置在 IoC 容器的控制范围之外创建的对象。参见[使用 AspectJ 与 Spring 依赖注入域对象](#)。

以下示例显示了基于 XML 的配置元数据的基本结构：

```
<?xml version="1.0" encoding="UTF-8"?>
```

```

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

    <bean id="..." class="..."> (1) (2)
        <!-- 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>

```

- (1) `id` 属性是标识单个 bean 定义的字符串。
- (2) `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。在此示例中未显示用于引用协作对象的 XML。有关更多信息，请参见 [Dependencies](#)。

## 1.2.2. 实例化容器

提供给 `ApplicationContext` 构造函数的位置路径是资源字符串，这些资源字符串使容器可以从各种外部资源(例如本地文件系统, Java `CLASSPATH` 等)加载配置元数据。

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

### iNote

了解了 Spring 的 IoC 容器之后，您可能想了解更多有关 Spring 的 `Resource` 抽象(如 [Resources](#) 中所述)，该抽象为从 URI 语法中定义的位置读取 `InputStream` 提供了一种方便的机制。特别是，`Resource` 路径用于构建应用程序上下文，如[应用程序上下文和资源路径](#)中所述。

以下示例显示了服务层对象 (`services.xml`) 配置文件：

```

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

    <!-- services -->

    <bean id="petStore" class="org.springframework.samples.jpetstore.services.PetStoreService">
        <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：

```

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

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

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

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

</beans>

```

在前面的示例中，服务层由 `PetStoreServiceImpl` 类和两个类型为 `JpaAccountDao` 和 `JpaItemDao` 的数据访问对象组成(基于 JPA 对象关系 Map 标准)。 `property name` 元素引用 JavaBean 属性的名称，而 `ref` 元素引用另一个 bean 定义的名称。 `id` 和 `ref` 元素之间的这种联系表达了协作对象之间的依赖性。有关配置对象的依赖项的详细信息，请参见[Dependencies](#)。

## 构成基于 XML 的配置元数据

使 bean 定义跨越多个 XML 文件可能很有用。通常，每个单独的 XML 配置文件都代表体系结构中的逻辑层或模块。

您可以使用应用程序上下文构造函数从所有这些 XML 片段中加载 bean 定义。该构造函数具有多个 `Resource` 位置，如[previous section](#) 所示。或者，使用一个或多个 `<import/>` 元素从另一个文件中加载 bean 定义。以下示例显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，从三个文件 `services.xml`，`messageSource.xml` 和 `themeSource.xml` 加载了外部 bean 定义。所有位置路径都相对于进行导入的定义文件，因此 `services.xml` 必须与进行导入的文件位于同一目录或 Classpath 位置，而 `messageSource.xml` 和 `themeSource.xml` 必须位于导入文件位置下方的 `resources` 位置。如您所见，斜杠被忽略。但是，鉴于这些路径是相对的，最好不要使用任何斜线。根据 Spring Schema，导入的文件的内容(包括顶级 `<beans/>` 元素)必须是有效的 XML bean 定义。

### iNote

可以但不建议使用相对的“`../`”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。特别是，不建议对 `classpath: URL`(例如 `classpath:../services.xml`) 使用此引用，在 URL 中，运行时解析过程选择“最近”Classpath 根，然后查看其父目录。Classpath 配置的更改可能导致选择其他错误的目录。您始终可以使用标准资源位置而不是相对路径：例如 `file:C:/config/services.xml` 或 `classpath:/config/services.xml`。但是，请注意，您正在将应用程序的配置耦合到特定的绝对位置。通常，最好为这样的绝对位置保留一个间接寻址，例如通过在运行时针对 JVM 系统属性解析的“ `${…}` ”占位符。

名称空间本身提供了导入指令功能。 Spring 提供的一系列 XML 名称空间(例如 `context` 和 `util` 名称空间)中提供了超出普通 bean 定义的其他配置功能。

### Groovy Bean 定义 DSL

作为外部化配置元数据的另一个示例， Bean 定义也可以在 Spring 的 Groovy Bean 定义 DSL 中表达，如 Grails 框架所知。通常，这种配置位于 “.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
        }
    }
}
```

这种配置样式在很大程度上等同于 XML bean 定义，甚至支持 Spring 的 XML 配置名称空间。它还允许通过 `importBeans` 指令导入 XML bean 定义文件。

### 1.2.3. 使用容器

`ApplicationContext` 是高级工厂的界面，该工厂能够维护不同 bean 及其依赖关系的注册表。通过使用方法 `T getBean(String name, Class<T> requiredType)`，您可以检索 bean 的实例。

`ApplicationContext` 允许您读取 bean 定义并访问它们，如以下示例所示：

```
// 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，引导看起来非常相似。它有一个不同的上下文实现类，该类可识别 Groovy(但也了解 XML Bean 定义)。以下示例显示了 Groovy 配置：

```
ApplicationContext context = new GenericGroovyApplicationContext("services.groovy", "da...
```

最灵活的变体是 `GenericApplicationContext` 与阅 Reader 委托组合，例如，对于 XML 文件，是 `XmlBeanDefinitionReader`，如以下示例所示：

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

您还可以将 `GroovyBeanDefinitionReader` 用于 Groovy 文件，如以下示例所示：

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

您可以在相同的 `ApplicationContext` 上混合并匹配此类阅读器委托，从不同的配置源读取 Bean 定义。

然后，您可以使用 `getBean` 来检索 bean 的实例。`ApplicationContext` 接口还有其他几种检索 bean 的方法，但是理想情况下，您的应用程序代码永远不要使用它们。实际上，您的应用程序代码应该根本不调用 `getBean()` 方法，因此完全不依赖于 Spring API。例如，Spring 与 Web 框架的集成为各种 Web 框架组件(例如控制器和 JSFManagement 的 Bean)提供了依赖注入，使您可以通过元数据(例如自动装配 Comments)声明对特定 Bean 的依赖。

## 1.3. Bean 总览

Spring IoC 容器 Management 一个或多个 bean。这些 bean 是使用您提供给容器的配置元数据创建的(例如，以 XML `<bean/>` 定义的形式)。

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

- 包限定的类名：通常，定义了 Bean 的实际实现类。
- Bean 行为配置元素，用于声明 Bean 在容器中的行为(作用域，生命周期回调等)。
- 引用其他 bean 进行其工作所需的 bean。这些引用也称为协作者或依赖项。
- 要在新创建的对象中设置的其他配置设置，例如，池的大小限制或在 Management 连接池的 bean 中使用的连接数。

该元数据转换为构成每个 bean 定义的一组属性。下表描述了这些属性：

表 1. *bean* 定义

Property	Explained in...
Class	<a href="#">Instantiating Beans</a>
Name	<a href="#">Naming Beans</a>
Scope	<a href="#">Bean Scopes</a>
Constructor arguments	<a href="#">Dependency Injection</a>
Properties	<a href="#">Dependency Injection</a>
Autowiring mode	<a href="#">Autowiring Collaborators</a>
延迟初始化模式	<a href="#">Lazy-initialized Beans</a>

Property	Explained in...
Initialization method	<a href="#">Initialization Callbacks</a>
Destruction method	<a href="#">Destruction Callbacks</a>

除了包含有关如何创建特定 bean 的信息的 bean 定义之外，`ApplicationContext` 实现还允许注册在容器外部(由用户)创建的现有对象。这是通过 `getBeanFactory()` 方法访问 `ApplicationContext` 的 BeanFactory 来完成的，该方法返回 BeanFactory `DefaultListableBeanFactory` 的实现。`DefaultListableBeanFactory` 通过 `registerSingleton(..)` 和 `registerBeanDefinition(..)` 方法支持此注册。但是，典型的应用程序只能与通过常规 bean 定义元数据定义的 bean 一起使用。

### Note

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推理它们。虽然在某种程度上支持覆盖现有元数据和现有单例实例，但是在运行时(与对工厂的实时访问同时)对新 bean 的注册不被正式支持，并且可能导致并发访问异常，bean 容器中的状态不一致或都。

## 1.3.1. 命名 bean

每个 bean 具有一个或多个标识符。这些标识符在承载 Bean 的容器内必须唯一。一个 bean 通常只有一个标识符。但是，如果需要多个，则可以将多余的别名视为别名。

在基于 XML 的配置元数据中，您可以使用 `id` 属性和 `name` 属性，或同时使用这两者来指定 bean 标识符。`id` 属性可让您精确指定一个 ID。按照惯例，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，并用逗号(,)，分号(;)或空格分隔。作为历史记录，在 Spring 3.1 之前的版本

中，`id` 属性定义为 `xsd:ID` 类型，该类型限制了可能的字符。从 3.1 开始，它被定义为 `xsd:string` 类型。请注意，bean `id` 的唯一性仍由容器强制执行，尽管不再由 XML 解析器执行。

您不需要为 Bean 提供 `name` 或 `id`。如果未明确提供 `name` 或 `id`，则容器将为该 bean 生成一个唯一的名称。但是，如果要按名称引用该 bean，则通过使用 `ref` 元素或 [Service Locator](#) 样式查找，必须提供一个名称。不提供名称的动机与使用 [inner beans](#) 和 [autowiring collaborators](#) 有关。

## Bean 命名约定

约定是在命名 bean 时将标准 Java 约定用于实例字段名称。也就是说，bean 名称以小写字母开头，并从那里用驼峰式大小写。这样的名称的示例包括 `accountManager`，`accountService`，`userDao`，`loginController` 等。

一致地命名 Bean 使您的配置更易于阅读和理解。另外，如果您使用 Spring AOP，则在将建议应用于按名称相关的一组 bean 时，它会很有帮助。

### iNote

通过在 Classpath 中进行组件扫描，Spring 会按照前面描述的规则为未命名的组件生成 Bean 名称：从本质上讲，采用简单的类名称并将其初始字符转换为小写。但是，在(不寻常的)特殊情况下，如果有多个字符并且第一个和第二个字符均为大写字母，则会保留原始大小写。这些规则与 `java.beans.Introspector.decapitalized` (Spring 在此使用) 定义的规则相同。

## 在 Bean 定义之外别名 Bean

在 bean 定义本身中，可以使用 `id` 属性指定的最多一个名称和 `name` 属性中任意数量的其他名称的组合来为 bean 提供多个名称。这些名称可以是同一个 bean 的等效别名，并且在某些情况下很有用，例如，通过使用特定于该组件本身的 bean 名称，让应用程序中的每个组件都引用一个公共依赖项。

但是，在实际定义 bean 的地方指定所有别名并不总是足够的。有时需要为在别处定义的 bean 引入别名。在大型系统中通常是这种情况，在大型系统中，配置在每个子系统之间分配，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，您可以使用 `<alias/>` 元素来完成此操作。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，使用该别名定义后，名为 `fromName` 的 bean(在同一容器中)也可以称为 `toName`。  
。

例如，子系统 A 的配置元数据可以通过名称 `subsystemA-dataSource` 引用数据源。子系统 B 的配置元数据可以通过名称 `subsystemB-dataSource` 引用数据源。组成使用这两个子系统的主应用程序时，主应用程序使用 `myApp-dataSource` 的名称引用数据源。要使所有三个名称都引用相同的对象，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，并且可以保证不与任何其他定义冲突(有效地创建名称空间)，但是它们引用的是同一 bean。

## Java-configuration

如果使用 Java 配置，则 `@Bean` 注解可用于提供别名。有关详情，请参见[使用@BeanComments](#)。

### 1.3.2. 实例化 bean

Bean 定义实质上是创建一个或多个对象的方法。当被询问时，容器将查看命名 bean 的配方，并使用该 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，则可以在 `<bean/>` 元素的 `class` 属性中指定要实例化的对象的类型(或类)。通常，此 `class` 属性(在内部是 `BeanDefinition` 实例上的 `Class` 属性)。(有关 exceptions，请参阅[使用实例工厂方法实例化](#)和[Bean 定义继承](#)。)可以通过以下两种方式之一使用

## Class 属性：

- 通常，在容器本身通过反射性地调用其构造函数直接创建 Bean 的情况下，指定要构造的 Bean 类，这在某种程度上等效于 `new` 运算符的 Java 代码。
- 要指定包含被调用以创建对象的 `static` 工厂方法的实际类，在不太常见的情况下，容器将在类上调用 `static` 工厂方法以创建 Bean。从 `static` 工厂方法的调用返回的对象类型可以是同一类，也可以是完全不同的另一类。

## 内部类名称

如果要为 `static` 嵌套类配置 Bean 定义，则必须使用嵌套类的二进制名称。

例如，如果您在 `com.example` 包中有一个名为 `SomeThing` 的类，并且此 `SomeThing` 类具有一个名为 `OtherThing` 的 `static` 嵌套类，则 Bean 定义上 `class` 属性的值为 `com.example.SomeThing$OtherThing`。

请注意，名称中使用了 `$` 字符以将嵌套的类名与外部类名分开。

## 用构造函数实例化

当通过构造方法创建一个 bean 时，所有普通类都可以被 Spring 使用并与之兼容。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。只需指定 bean 类就足够了。但是，根据您用于该特定 bean 的 IoC 的类型，您可能需要一个默认(空)构造函数。

Spring IoC 容器几乎可以 Management 您要 Management 的任何类。它不仅限于 Managementtrue 的 JavaBean。大多数 Spring 用户更喜欢实际的 JavaBean，它仅具有默认(无参数)构造函数，并具有根据容器中的属性建模的适当的 setter 和 getter。您还可以在容器中具有更多奇特的非 bean 样式类。例如，如果您需要使用绝对不符合 JavaBean 规范的旧式连接池，则 Spring 也可以对其进行 Management。

使用基于 XML 的配置元数据，您可以如下指定 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)并在构造对象之后设置对象实例属性的机制的详细信息, 请参见[Injecting Dependencies](#)。

## 使用静态工厂方法实例化

定义使用静态工厂方法创建的 bean 时, 请使用 `class` 属性来指定包含 `static` 工厂方法的类, 并使用名为 `factory-method` 的属性来指定工厂方法本身的名称。您应该能够调用此方法(带有可选参数, 如稍后所述)并返回一个活动对象, 该对象随后将被视为已通过构造函数创建。这种 bean 定义的一种用法是在旧代码中调用 `static` 工厂。

以下 bean 定义指定通过调用工厂方法来创建 bean。该定义不指定返回对象的类型(类), 而仅指定包含工厂方法的类。在此的示例 `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"/>
```

以下示例显示了可与前面的 bean 定义一起使用的类:

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

有关为工厂方法提供(可选)参数并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, 请参见[依赖性和详细配置](#)。

## 使用实例工厂方法实例化

与通过[静态工厂方法](#)实例化类似, 使用实例工厂方法实例化从容器中调用现有 bean 的非静态方法

以创建新 bean。要使用此机制，请将 `class` 属性留空，并在 `factory-bean` 属性中，在当前(或父容器或祖先容器)中指定包含要创建该对象的实例方法的 bean 的名称。使用 `factory-method` 属性设置工厂方法本身的名字。以下示例显示了如何配置此类 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" />
```

以下示例显示了相应的 Java 类：

```
public class DefaultServiceLocator {

    private static ClientService clientService = new ClientServiceImpl();

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

一个工厂类也可以包含一个以上的工厂方法，如以下示例所示：

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

<bean id="clientService"
    factory-bean="serviceLocator"
    factory-method="createClientServiceInstance" />

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

以下示例显示了相应的 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 本身可以通过依赖项注入(DI)进行 Management 和配置。参见[依赖性和详细配置](#)。

### iNote

在 Spring 文档中，“factory bean”是指在 Spring 容器中配置并通过[instance](#)或[static](#)工厂方法创建对象的 Bean。相反，[FactoryBean](#) (注意大小写)是指特定于 Spring 的[FactoryBean](#)。

## 1.4. Dependencies

典型的企业应用程序不包含单个对象(或 Spring 术语中的 bean)。即使是最简单的应用程序，也有一些对象可以协同工作，以呈现最终用户视为一致的应用程序。下一部分将说明如何从定义多个独立的 Bean 定义到实现对象协作以实现目标的完全实现的应用程序。

### 1.4.1. 依赖注入

依赖注入(DI)是一个过程，通过该过程，对象只能通过构造函数参数，工厂方法的参数或在构造或创建对象实例后在对象实例上设置的属性来定义其依赖关系(即，与它们一起工作的其他对象)。从工厂方法返回。然后，容器在创建 bean 时注入那些依赖项。从根本上讲，此过程是通过使用类的直接构造或服务定位器模式来自己控制其依赖关系的实例化或位置的 Bean 本身的逆过程(因此称为 Control Inversion)。

使用 DI 原理，代码更简洁，当为对象提供依赖项时，去耦会更有效。该对象不查找其依赖项，也不知道依赖项的位置或类。结果，您的类变得更易于测试，尤其是当依赖项依赖于接口或抽象 Base Class 时，它们允许在单元测试中使用存根或模拟实现。

DI 存在两个主要变体：[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关系注入

基于构造函数的 DI 是通过容器调用具有多个参数(每个参数代表一个依赖项)的构造函数来完成的。

调用带有特定参数的 `static` 工厂方法来构造 Bean 几乎是等效的，并且本次讨论也将构造函数和 `static` 工厂方法的参数视为类似。以下示例显示了只能通过构造函数注入进行依赖项注入的类：

```
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，该类没有什么特别的。它是一个 POJO，不依赖于特定于容器的接口，Base Class 或 Comments。

## 构造函数参数解析

构造函数参数解析匹配通过使用参数的类型进行。如果 Bean 定义的构造函数参数中不存在潜在的歧义，则在实例化 Bean 时，在 Bean 定义中定义构造函数参数的 Sequences 就是将这些参数提供给适当的构造函数的 Sequences。考虑以下类别：

```
package x.y;  
  
public class ThingOne {  
  
    public ThingOne(ThingTwo thingTwo, ThingThree thingThree) {  
        // ...  
    }  
}
```

假设 `ThingTwo` 和 `ThingThree` 类没有通过继承关联，则不存在潜在的歧义。因此，以下配置可以正常工作，并且您无需在 `<constructor-arg>` 元素中显式指定构造函数参数索引或类型。

```

<beans>
    <bean id="thingOne" class="x.y.ThingOne">
        <constructor-arg ref="thingTwo"/>
        <constructor-arg ref="thingThree"/>
    </bean>

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

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

```

当引用另一个 bean 时，类型是已知的，并且可以发生匹配(与前面的示例一样)。当使用简单类型(例如 `<value>true</value>`)时，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进行匹配。考虑以下类别：

```

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;
    }
}

```

## 构造函数参数类型匹配

在上述情况下，如果您使用 `type` 属性显式指定了构造函数参数的类型，则容器可以使用简单类型的类型匹配。如下例所示：

```

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

```

## 构造函数参数索引

您可以使用 `index` 属性来显式指定构造函数参数的索引，如以下示例所示：

```

<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歧义性之外，指定索引还可以解决歧义，其中构造函数具有两个相同类型的参数。

### iNote

索引从 0 开始。

### 构造函数参数名称

您还可以使用构造函数参数名称来消除歧义，如以下示例所示：

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

请记住，要立即使用该功能，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称。如果您不能或不想使用 debug 标志编译代码，则可以使用 [@ConstructorProperties](#) JDKComments 显式命名构造函数参数。然后，该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 的依赖项注入

基于设置器的 DI 是通过在调用无参数构造函数或无参数 **static** 工厂方法以实例化您的 bean 之后，在您的 bean 上调用 setter 方法来完成的。

下面的示例显示只能通过使用纯 `setter` 注入来依赖注入的类。此类是常规的 Java。它是一个 POJO，不依赖于容器特定的接口，Base Class 或 Comments。

```
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` 支持其 Management 的 bean 的基于构造函数和基于 `setter` 的 DI。在已经通过构造函数方法注入了某些依赖项之后，它还支持基于 `setter` 的 DI。您可以以 `BeanDefinition` 的形式配置依赖项，并与 `PropertyEditor` 实例结合使用以将属性从一种格式转换为另一种格式。但是，大多数 Spring 用户不是直接(即以编程方式)使用这些类，而是使用 XML `bean` 定义，带 Comments 的组件(即以 `@Component`，`@Controller` 等进行 Comments 的类)或基于 Java 的 `@Bean` 方法。`@Configuration` 个类。然后将这些源在内部转换为 `BeanDefinition` 的实例，并用于加载整个 Spring IoC 容器实例。

基于构造函数或基于 `setter` 的 DI?

由于可以混合使用基于构造函数的 DI 和基于 `setter` 的 DI，因此将构造函数用于强制性依赖项并将 `setter` 方法或配置方法用于可选依赖性是一个很好的经验法则。请注意，在 `setter` 方法上使用 `@Required` 注解可用于使属性成为必需的依赖项。

Spring 团队通常提倡构造函数注入，因为它可以让您将应用程序组件实现为不可变对象，并确保所需的依赖项不是 `null`。此外，注入构造函数的组件始终以完全初始化的状态返回到 Client 端(调用)代码。附带说明一下，大量的构造函数自变量是一种不好的代码味道，这意味着该类可能承担了太多的职责，应该对其进行重构以更好地解决关注点分离问题。

`Setter` 注入主要应仅用于可以在类中分配合理的默认值的可选依赖项。否则，必须在代码使用依赖

项的任何地方执行非空检查。setter 注入的一个好处是，setter 方法使该类的对象在以后可以重新配置或重新注入。因此，通过[JMX MBeans](#)进行 Management 是塞特注入的引人注目的用例。

使用最适合特定类的 DI 风格。有时，在处理您没有源代码的第三方类时，将为您做出选择。例如，如果第三方类未公开任何 setter 方法，则构造函数注入可能是 DI 的唯一可用形式。

## 依赖关系解决流程

容器执行 bean 依赖项解析，如下所示：

- `ApplicationContext` 用描述所有 bean 的配置元数据创建和初始化。可以通过 XML, Java 代码或 Comments 来指定配置元数据。
- 对于每个 bean，其依赖项都以属性，构造函数参数或 static-factory 方法的参数(如果使用它而不是常规构造函数)的形式表示。在实际创建 Bean 时，会将这些依赖项提供给 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，例如 `int`，`long`，`String`，`boolean` 等等。

在创建容器时，Spring 容器会验证每个 bean 的配置。但是，在实际创建 Bean 之前，不会设置 Bean 属性本身。创建容器时，将创建具有单例作用域并设置为预先实例化(默认)的 Bean。范围在 [Bean Scopes](#) 中定义。否则，仅在请求时才创建 Bean。创建和分配 bean 的依赖关系及其依赖关系(依此类推)时，创建 bean 可能会导致创建一个 bean 图。请注意，这些依赖项之间的分辨率不匹配可能会在后期出现，即在第一次创建受影响的 bean 时。

## Circular dependencies

如果主要使用构造函数注入，则可能会创建无法解决的循环依赖方案。

例如：A 类通过构造函数注入需要 B 类的实例，而 B 类通过构造函数注入需要 A 类的实例。如果为将类 A 和 B 相互注入而配置了 bean，则 Spring IoC 容器会在运行时检测到此循环引用，并抛出 `BeanCurrentlyInCreationException`。

一种可能的解决方案是编辑某些类的源代码，这些类的源代码由设置者而不是构造函数来配置。或者，避免构造函数注入，而仅使用 `setter` 注入。换句话说，尽管不建议这样做，但是您可以使用 `setter` 注入配置循环依赖关系。

与典型情况(没有循环依赖项)不同，`bean A` 和 `bean B` 之间的循环依赖关系迫使其中一个 `bean` 在完全初始化之前被注入另一个 `bean`(经典的“养鸡和鸡蛋”场景)。

通常，您可以信任 Spring 做正确的事。它在容器加载时检测配置问题，例如对不存在的 Bean 的引用和循环依赖项。在实际创建 Bean 时，Spring 设置属性并尽可能晚地解决依赖关系。这意味着如果创建该对象或其依赖项之一时出现问题，则正确加载了 Spring 的容器以后可以在您请求对象时生成异常-例如，由于缺少或无效，Bean 引发异常属性。某些配置问题的这种潜在的延迟可见性是为什么默认情况下 `ApplicationContext` 实现会实例化单例 bean。在实际需要它们之前，要花一些前期时间和内存来创建它们，您会在创建 `ApplicationContext` 时发现配置问题，而不是稍后。您仍然可以覆盖此默认行为，以便单例 bean 延迟初始化，而不是预先实例化。

如果不存在循环依赖关系，则在将一个或多个协作 Bean 注入从属 Bean 时，每个协作 Bean 都将被完全配置，然后再注入到从属 Bean 中。这意味着，如果 `bean A` 依赖于 `bean B`，则 Spring IoC 容器会在对 `bean A` 调用 `setter` 方法之前完全配置 `beanB`。换句话说，实例化了 `bean`(如果它不是预先实例化的单例)，设置其依赖项，并调用相关的生命周期方法(例如[配置的 init 方法](#)或[InitializingBean 回调方法](#))。

## 依赖项注入的示例

以下示例将基于 XML 的配置元数据用于基于 `setter` 的 DI。Spring XML 配置文件的一小部分指定了一些 `bean` 定义，如下所示：

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

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

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

以下示例显示了相应的 `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;
    }
}
```

在前面的示例中，声明了 `setter` 以与 XML 文件中指定的属性匹配。以下示例使用基于构造函数的 DI：

```
<bean id="exampleBean" class="examples.ExampleBean">
    <!-- constructor injection using the nested ref element -->
    <constructor-arg>
        <ref bean="anotherExampleBean"/>
    </constructor-arg>

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

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

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

以下示例显示了相应的 `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;
}
}

```

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

现在考虑这个示例的一个变体，在该变体中，不是使用构造函数，而是告诉 Spring 调用 `static` 工厂方法以返回对象的实例：

```

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

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

```

以下示例显示了相应的 `ExampleBean` 类：

```

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;
    }
}

```

`static` 工厂方法的参数由 `<constructor-arg/>` 元素提供，与实际使用构造函数的情况完全相

同。 `factory` 方法返回的类的类型不必与包含 `static` `factory` 方法的类的类型相同(尽管在此示例中为)。实例(非静态)工厂方法可以以基本上相同的方式使用(除了使用 `factory-bean` 属性而不是 `class` 属性), 因此在此不讨论这些细节。

### 1.4.2. 依赖性和详细配置

如[previous section](#)中所述, 您可以将 `bean` 属性和构造函数参数定义为对其他托管 `bean`(协作者)的引用或内联定义的值。为此, Spring 的基于 XML 的配置元数据在其 `<property/>` 和 `<constructor-arg/>` 元素中支持子元素类型。

#### 直值(Primitives , 字符串等)

`<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[conversion service](#)用于将这些值从 `String` 转换为属性或参数的实际类型。以下示例显示了设置的各种值:

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

以下示例使用[p-namespace](#)进行更简洁的 XML 配置:

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

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

</beans>
```

前面的 XML 更简洁。但是，除非在创建 bean 定义时使用支持自动属性完成的 IDE(例如[IntelliJ IDEA](#)或[Spring 工具套件](#))，否则错字是在运行时而不是设计时发现的。强烈建议您使用此类 IDE 帮助。

您还可以配置 `java.util.Properties` 实例，如下所示：

```
<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 `PropertyEditor` 机制将 `<value/>` 元素内的文本转换为 `java.util.Properties` 实例。这是一个不错的捷径，并且是 Spring 团队偏爱使用嵌套 `<value/>` 元素而不是 `value` 属性样式的几个地方之一。

## idref 元素

`idref` 元素只是将容器中另一个 bean 的 `id` (字符串值-不是引用)传递给 `<constructor-arg/>` 或 `<property/>` 元素的一种防错方法。以下示例显示了如何使用它：

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

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

前面的 bean 定义代码段(在运行时)与以下代码段完全等效：

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

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

第一种形式优于第二种形式，因为使用 `idref` 标签可以使容器在部署时验证所引用的命名 Bean 实际上是否存在。在第二个变体中，不对传递给 `client` bean 的 `targetName` 属性的值执行验证。仅在实际实例化 `client` bean 时才发现拼写错误(极有可能导致致命的结果)。如果 `client` bean 是 prototype bean，则只能在部署容器很长时间之后发现此错字和所产生的异常。

### Note

`idref` 元素上的 `local` 属性在 4.0 bean XSD 中不再受支持，因为它不再提供常规 `bean` 引用上的值。升级到 4.0 模式时，将现有的 `idref local` 引用更改为 `idref bean`。

`<idref/>` 元素带来价值的一个常见地方(至少在 Spring 2.0 之前的版本中)是 `ProxyFactoryBean` bean 定义中的 [AOP interceptors](#) 配置。指定拦截器名称时使用 `<idref/>` 元素可防止您拼写错误的拦截器 ID。

### 对其他 Bean 的引用(协作者)

`ref` 元素是 `<constructor-arg/>` 或 `<property/>` 定义元素内的最后一个元素。在这里，您将一个 bean 的指定属性的值设置为对容器 Management 的另一个 bean(协作者)的引用。引用的 bean 是要设置其属性的 bean 的依赖关系，并且在设置属性之前根据需要对其进行初始化。(如果协作者是单例 bean，则它可能已经由容器初始化了。)所有引用最终都是对另一个对象的引用。范围和验证取决于是否通过 `bean`，`local`，或 `parent` 属性指定另一个对象的 ID 或名称。

通过 `<ref/>` 标签的 `bean` 属性指定目标 bean 是最通用的形式，并且允许创建对同一容器或父容器中任何 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` 属性中的值之一相同。下面的示例演示如何使用 `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。 `parent` 属性的值可以与目标 Bean 的 `id` 属性或目标 Bean 的 `name` 属性中的值相同。目标 Bean 必须位于当前容器的父容器中。主要在具有容器层次结构并且要使用与父 bean 名称相同的代理将现有 bean 封装在父容器中时，才应使用此 bean 参考变量。以下一对清单显示了如何使用 `parent` 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>
```

### iNote

`ref` 元素上的 `local` 属性在 4.0 bean XSD 中不再受支持，因为它不再提供常规 `bean` 引用上的值。升级到 4.0 模式时，将现有的 `ref local` 引用更改为 `ref bean`。

## Inner Beans

`<property/>` 或 `<constructor-arg/>` 元素内的 `<bean/>` 元素定义了一个内部 bean，如以下示例所示：

```
<bean id="outer" class="...">
    <!-- instead of using a reference to a target bean, simply define the target bean in
    <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>
```

内部 bean 定义不需要定义的 ID 或名称。如果指定，则容器不使用该值作为标识符。容器在创建时

也将忽略 `scope` 标志，因为内部 Bean 始终是匿名的，并且始终与外部 Bean 一起创建。不可能独立地访问内部 bean 或将其注入到协作 bean 中而不是封装到封闭 bean 中。

作为一个特例，可以从自定义作用域中接收销毁回调，例如对于单例 bean 中包含的请求范围内的 bean。内部 bean 实例的创建与其包含的 bean 绑定在一起，但是销毁回调使它可以参与请求范围的生命周期。这不是常见的情况。内部 bean 通常只共享其包含 bean 的作用域。

## Collections

`<list/>`，`<set/>`，`<map/>` 和 `<props/>` 元素分别设置 Java `Collection` 类型 `List`，`Set`，`Map` 和 `Properties` 的属性和参数。以下示例显示了如何使用它们：

```
<bean id="moreComplexObject" class="example.ComplexObject">
    <!-- results in a setAdminEmails(java.util.Properties) call -->
    <property name="adminEmails">
        <props>
            <prop key="administrator">[emailprotected]</prop>
            <prop key="support">[emailprotected]</prop>
            <prop key="development">[emailprotected]</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>
```

Map 键或值的值或设置值也可以是以下任意元素：

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

## Collection Merging

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，并使子级 `<list/>`，`<map/>`，`<set/>` 或 `<props/>` 元素继承并覆盖父级集合中的值。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，子集合的元素将覆盖父集合中指定的值。

关于合并的本节讨论了父子 bean 机制。不熟悉父 bean 和子 bean 定义的 Reader 可能希望先阅读 [relevant section](#)，然后再 continue。

下面的示例演示了集合合并：

```
<beans>
    <bean id="parent" abstract="true" class="example.ComplexObject">
        <property name="adminEmails">
            <props>
                <prop key="administrator">[emailprotected]</prop>
                <prop key="support">[emailprotected]</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">[emailprotected]</prop>
                <prop key="support">[emailprotected]</prop>
            </props>
        </property>
    </bean>
<beans>
```

注意 `child` bean 定义的 `adminEmails` 属性的 `<props/>` 元素上使用了 `merge=true` 属性。当 `child` bean 被容器解析并实例化时，生成的实例具有 `adminEmails Properties` 集合，该集合包含将子代的 `adminEmails` 集合与父代的 `adminEmails` 集合合并的结果。以下清单显示了结果：

```
[emailprotected]
[emailprotected]
[emailprotected]
```

子 `Properties` 集合的值集继承了父 `<props/>` 的所有属性元素，而子 `support` 值的值覆盖父集

合中的值。

此合并行为类似地适用于 `<list/>`，`<map/>` 和 `<set/>` 集合类型。在 `<list/>` 元素的特定情况下，将保留与 `List` 集合类型关联的语义(即，值 `ordered` 集合的概念)。父级的值位于所有子级列表的值之前。对于 `Map`，`Set` 和 `Properties` 集合类型，不存在排序。因此，对于容器内部使用的相关 `Map`，`Set` 和 `Properties` 实现类型基础的集合类型，没有排序语义有效。

## 集合合并的限制

您不能合并不同的集合类型(例如 `Map` 和 `List`)。如果尝试这样做，则会抛出一个适当的 `Exception`。`merge` 属性必须在较低的继承的子定义中指定。在父集合定义上指定 `merge` 属性是多余的，不会导致所需的合并。

## Strongly-typed collection

随着 Java 5 中泛型类型的引入，您可以使用强类型集合。也就是说，可以声明 `Collection` 类型，使其只能包含(例如) `String` 元素。如果使用 Spring 将强类型的 `Collection` 依赖项注入到 bean 中，则可以利用 Spring 的类型转换支持，以便将强类型 `Collection` 实例的元素转换为适当的类型，然后再添加到 `Collection`。以下 Java 类和 bean 定义显示了如何执行此操作：

```
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>
```

当准备注入 `something` bean 的 `accounts` 属性时，可以通过反射获得有关强类型 `Map<String, Float>` 的元素类型的泛型信息。因此，Spring 的类型转换基础结构将各种值元素识别为 `Float` 类型，并且字符串值(`9.99`, `2.75` 和 `3.99`)被转换为实际的 `Float` 类型。

## 空字符串值和空字符串

Spring 将属性等的空参数视为空 `String`。以下基于 XML 的配置元数据片段将 `email` 属性设置为空的 `String` 值(“”)。

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

前面的示例等效于以下 Java 代码：

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

`<null/>` 元素处理 `null` 个值。以下清单显示了一个示例：

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

前面的配置等效于下面的 Java 代码：

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

## 具有 `p` 名称空间的 XML 快捷方式

`p-namespace` 允许您使用 `bean` 元素的属性(而不是嵌套的 `<property/>` 元素)来描述协作 Bean 的属性值，或同时使用这两者。

Spring 支持基于 XML Schema 定义的可扩展配置格式 [with namespaces](#)。本章讨论的 [beans](#) 配置格式在 XML Schema 文档中定义。但是，[p](#) 命名空间未在 XSD 文件中定义，仅存在于 Spring 的核心中。

下面的示例显示了两个 XML 代码段(第一个使用标准 XML 格式，第二个使用 [p](#) 命名空间)，它们可以解析为相同的结果：

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

    <bean name="classic" class="com.example.ExampleBean">
        <property name="email" value="[emailprotected]" />
    </bean>

    <bean name="p-namespace" class="com.example.ExampleBean"
          p:email="[emailprotected]" />
</beans>
```

该示例显示了 [p](#) 命名空间中 Bean 定义中名为 [email](#) 的属性。这告诉 Spring 包含一个属性声明。如前所述，[p](#) 名称空间没有架构定义，因此可以将属性名称设置为属性名称。

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"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">

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

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

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

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，而第二个 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` 是属性名称，而 `-ref` 部分表示这不是一个直接值，而是对另一个 bean 的引用。

### iNote

`p` 命名空间不如标准 XML 格式灵活。例如，用于声明属性引用的格式与以 `Ref` 结尾的属性发生冲突，而标准 XML 格式则没有。我们建议您仔细选择方法，并与团队成员进行交流，以避免同时使用这三种方法生成 XML 文档。

## 具有 c-namespace 的 XML 快捷方式

与 [具有 p-命名空间的 XML 快捷方式](#) 相似，在 Spring 3.1 中引入的 c-namespace 允许使用内联属性来配置构造函数参数，而不是嵌套的 `constructor-arg` 元素。

以下示例使用 `c:` 命名空间执行与 [基于构造函数的依赖注入](#) 相同的操作：

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

    <bean id="thingOne" class="x.y.ThingTwo"/>
    <bean id="thingTwo" class="x.y.ThingThree"/>

    <!-- traditional declaration -->
    <bean id="thingOne" class="x.y.ThingOne">
        <constructor-arg ref="thingTwo"/>
        <constructor-arg ref="thingThree"/>
        <constructor-arg value="[emailprotected]"/>
    </bean>

    <!-- c-namespace declaration -->
    <bean id="thingOne" class="x.y.ThingOne" c:thingTwo-ref="thingTwo" c:thingThree-ref="thingThree">
        <!-- configuration using c-namespace -->
    </bean>
</beans>
```

`c:` 名称空间使用与 `p:` 相同的约定(对于 Bean 引用，尾随 `-ref`)以其名称设置构造函数参数。

同样，即使未在 XSD 模式中定义它(也存在于 Spring 内核中)也需要声明它。

对于极少数情况下无法使用构造函数自变量名称的情况(通常，如果字节码是在没有调试信息的情况下编译的)，可以对参数索引使用后备，如下所示：

```
<!-- c-namespace index declaration -->
<bean id="thingOne" class="x.y.ThingOne" c:_0-ref="thingTwo" c:_1-ref="thingThree"/>
```

### iNote

由于 XML 语法的原因，索引符号要求前导 `_` 的存在，因为 XML 属性名称不能以数字开头 (即使某些 IDE 允许)。

实际上，构造函数解析[mechanism](#)在匹配参数方面非常有效，因此，除非您确实需要，否则我们建议在整个配置过程中使用名称符号。

### 复合属性名称

设置 `bean` 属性时，可以使用复合属性名称或嵌套属性名称，只要路径中除最终属性名称之外的所有组件都不是 `null` 即可。考虑以下 `bean` 定义：

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

`something` `bean` 具有 `fred` 属性，该属性具有 `bob` 属性，该属性具有 `sammy` 属性，并且最终的 `sammy` 属性被设置为 `123` 的值。为了使它起作用，在构造 `bean` 之后，`something` 的 `fred` 属性和 `fred` 的 `bob` 属性一定不能为 `null`。否则，将抛出 `NullPointerException`。

### 1.4.3. 使用依赖

如果一个 `bean` 是另一个 `bean` 的依赖项，则通常意味着将一个 `bean` 设置为另一个 `bean` 的属性。通常，您可以使用基于 XML 的配置元数据中的[`<ref/> element`](#)完成此操作。但是，有时 `bean` 之间的依赖性不太直接。一个示例是何时需要触发类中的静态初始值设定项，例如用于数据库驱动程序注册。`depends-on` 属性可以在初始化使用此元素的 `bean` 之前显式强制初始化一个或多个 `bean`

。以下示例使用 `depends-on` 属性表示对单个 bean 的依赖关系：

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

要表示对多个 bean 的依赖性，请提供一个 bean 名称列表作为 `depends-on` 属性的值(逗号，空格和分号是有效的分隔符)：

```
<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" />
```

#### ①Note

`depends-on` 属性既可以指定初始化时间依赖性，也可以仅在 [singleton](#) bean 的情况下指定相应的销毁时间依赖性。与给定 bean 定义 `depends-on` 关系的从属 bean 首先被销毁，然后再销毁给定 bean 本身。因此，`depends-on` 也可以控制关闭 Sequences。

#### 1.4.4. 懒初始化 bean

默认情况下，作为初始化过程的一部分，[ApplicationContext](#) 实现会急于创建和配置所有 [singleton](#) bean。通常，这种预初始化是可取的，因为与数小时甚至数天后相比，会立即发现配置或周围环境中的错误。如果不希望使用此行为，则可以通过将 bean 定义标记为延迟初始化来防止单例 bean 的预实例化。延迟初始化的 bean 告诉 IoC 容器在首次请求时而不是在启动时创建一个 bean 实例。

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

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

当 `ApplicationContext` 消耗了前面的配置时，`ApplicationContext` 启动时就不会急于预先实例化 `lazy` bean，而 `not.lazy` Bean 则会急于预先实例化。

但是，当延迟初始化的 bean 是未延迟初始化的单例 bean 的依赖项时，`ApplicationContext` 将在启动时创建延迟初始化的 bean，因为它必须满足单例的依赖关系。延迟初始化的 bean 被注入到其他未延迟初始化的单例 bean 中。

您还可以使用 `<beans/>` 元素上的 `default-lazy-init` 属性在容器级别控制延迟初始化，以下示例显示：

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

## 1.4.5. 自动布线合作者

Spring 容器可以自动装配协作 bean 之间的关系。您可以通过检查 `ApplicationContext` 的内容来让 Spring 自动为您的 bean 解决协作者(其他 bean)。自动装配具有以下优点：

- 自动装配可以大大减少指定属性或构造函数参数的需要。(在此方面，其他机制(例如 Bean 模板[在本章其他地方讨论](#))也很有价值。)
- 随着对象的 Developing，自动装配可以更新配置。例如，如果您需要向类中添加一个依赖项，则无需修改配置即可自动满足该依赖项。因此，自动装配在开发过程中特别有用，而不必担心当代码库变得更稳定时切换到显式接线的选择。

使用基于 XML 的配置元数据时(请参见[Dependency Injection](#))，可以使用 `<bean/>` 元素的 `autowire` 属性为 bean 定义指定自动装配模式。自动装配功能具有四种模式。您可以为每个 bean 指定自动装配，因此可以选择要自动装配的装配。下表描述了四种自动装配模式：

表 2. 自动装配模式

Mode	Explanation
<code>no</code>	(默认)无自动装配。 Bean 引用必须由 <code>ref</code> 元素定义。对于大型部署，建议不要更改默认设置，因为明确指定协作者可以提供更好的控制和清晰度。在某种程度上，它记录了系统的结构。
<code>byName</code>	按属性名称自动布线。 Spring 寻找与需要自动装配的属性同名的 bean。例如，如果一个 bean 定义被设置为按名称自动装配，并且包含一个 <code>master</code> 属性(即，它具有 <code>setMaster(..)</code> 方法)，那么 Spring 将查找一个名为 <code>master</code> 的 bean 定义并使用它来设置属性。
<code>byType</code>	如果容器中恰好存在一个该属性类型的 bean，则使该属性自动装配。如果存在多个错误，则会引发致命异常，这表明您可能不对该 bean 使用 <code>byType</code> 自动装配。如果没有匹配的 bean，则什么也不会发生(未设置该属性)。
<code>constructor</code>	类似于 <code>byType</code> ，但适用于构造函数参数。如果容器中不存在构造函数参数类型的一个 bean，则将引发致命错误。

在 `byType` 或 `constructor` 自动装配模式下，您可以连接阵列和键入的集合。在这种情况下，将提供容器中与期望类型匹配的所有自动装配候选，以满足相关性。如果预期的密钥类型为 `String`，则可以自动连接强类型的 `Map` 实例。自动连接的 `Map` 实例的值包含与期望的类型匹配的所有 bean 实例，并且 `Map` 实例的键包含相应的 Bean 名称。

## 自动装配的局限性和缺点

当在项目中一致使用自动装配时，自动装配效果最佳。如果通常不使用自动装配，那么使用开发人员仅连接一个或两个 bean 定义可能会使开发人员感到困惑。

考虑自动装配的局限性和缺点：

- `property` 和 `constructor-arg` 设置中的显式依赖项始终会覆盖自动装配。您不能自动连接简单属性，例如基元，`Strings` 和 `Classes` (以及此类简单属性的数组)。此限制是设计使然。
- 自动装配不如显式接线精确。尽管如前所述，Spring 还是小心避免在可能产生意外结果的模棱两可的情况下进行猜测。`SpringManagement` 的对象之间的关系不再明确记录。
- 接线信息可能不适用于可能从 Spring 容器生成文档的工具。
- 容器内的多个 `bean` 定义可能与要自动装配的 `setter` 方法或构造函数参数指定的类型匹配。对于数组，集合或 `Map` 实例，这不一定是问题。但是，对于需要单个值的依赖项，不会任意解决此歧义。如果没有唯一的 `bean` 定义可用，则引发异常。

在后一种情况下，您有几种选择：

- 放弃自动布线，转而使用明确的布线。
- 如[next section](#)中所述，通过将其 `autowire-candidate` 属性设置为 `false` 来避免自动装配 `bean` 定义。
- 通过将其 `<bean/>` 元素的 `primary` 属性设置为 `true`，将单个 `bean` 定义指定为主要候选对象。
- 如[基于 Comments 的容器配置](#)中所述，通过基于 Comments 的配置实现更细粒度的控件。

## 从自动装配中排除 Bean

在每个 `bean` 的基础上，您可以从自动装配中排除一个 `bean`。使用 Spring 的 XML 格式，将

`<bean/>` 元素的 `autowire-candidate` 属性设置为 `false`。容器使特定的 `bean` 定义对于自动装配基础结构不可用(包括 Comments 样式配置，例如[@Autowired](#))。

### Note

`autowire-candidate` 属性旨在仅影响基于类型的自动装配。它不会影响按名称显示的显式引用，即使未将指定的 Bean 标记为自动装配候选，该名称也可得到解析。因此，如果名称匹配，按名称自动装配仍会注入 Bean。

您还可以基于与 Bean 名称的模式匹配来限制自动装配候选。顶级 `<beans/>` 元素在其 `default-autowire-candidates` 属性内接受一个或多个模式。例如，要将自动装配候选状态限制为名称以 `Repository` 结尾的任何 bean，请提供值 `*Repository`。要提供多种模式，请在以逗号分隔的列表中定义它们。Bean 定义的 `autowire-candidate` 属性的 `true` 或 `false` 的显式值始终优先。对于此类 bean，模式匹配规则不适用。

这些技术对于您不希望通过自动装配将其注入其他 bean 的 bean 非常有用。这并不意味着排除的 bean 本身不能使用自动装配进行配置。相反，bean 本身不是自动装配其他 bean 的候选对象。

## 1.4.6. 方法注入

在大多数应用场景中，容器中的大多数 bean 是 [singletons](#)。当单例 Bean 需要与另一个单例 Bean 协作或非单例 Bean 需要与另一个非单例 Bean 协作时，通常可以通过将一个 Bean 定义为另一个 Bean 的属性来处理依赖性。当 bean 的生命周期不同时会出现问题。假设单例 bean A 需要使用非单例(原型)bean B，也许在 A 的每个方法调用上都使用。容器仅创建一次单例 bean A，因此只有一次机会来设置属性。每次需要一个容器时，容器都无法为 bean A 提供一个新的 bean B 实例。

一个解决方案是放弃某些控制反转。您可以通过实现 [ApplicationContextAware](#) 接口来使 bean A 知道容器，并在每次 bean A 需要它时对容器进行 `getBean("B")` 调用询问(通常是新的)bean B 实例。以下示例显示了此方法：

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

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

```

public class CommandManager implements ApplicationContextAware {

    private ApplicationContext applicationContext;

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

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

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

```

前面的内容是不理想的，因为业务代码知道并耦合到 Spring 框架。方法注入是 Spring IoC 容器的一项高级功能，使您可以干净地处理此用例。

您可以[在此博客条目](#)中阅读有关方法注入动机的更多信息。

## 查找方法注入

查找方法注入是容器重写容器 Management 的 Bean 上的方法并返回容器中另一个命名 Bean 的查找结果的能力。查找通常涉及原型 bean，如[上一节](#)中所述。Spring 框架通过使用从 CGLIB 库生成字节码来动态生成覆盖该方法的子类来实现此方法注入。

### iNote

- 为了使此动态子类起作用，Spring bean 容器子类的类也不能为 `final`，并且要覆盖的方法也不能为 `final`。
- 对具有 `abstract` 方法的类进行单元测试需要您自己对该类进行子类化，并提供 `abstract` 方法的存根实现。
- 组件扫描也需要具体方法，这需要具体的类别。

- 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，因此无法创建运行时生成的子类在飞行中。

对于前面的代码片段中的 `CommandManager` 类，Spring 容器将动态覆盖 `createCommand()` 方法的实现。`CommandManager` 类没有任何 Spring 依赖项，如重做的示例所示：

```
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(在本例中为 `CommandManager`)的 Client 端类中，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 `abstract`，则动态生成的子类将实现该方法。否则，动态生成的子类将覆盖原始类中定义的具体方法。考虑以下示例：

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

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

每当需要 `myCommand` bean 的新实例时，标识为 `commandManager` 的 bean 就会调用其自己的 `createCommand()` 方法。如果确实需要 `myCommand` bean 作为原型，则必须小心。如果它是 `singleton`，则每次都返回 `myCommand` bean 的相同实例。

另外，在基于 Comments 的组件模型中，您可以通过 `@Lookup` Comments 声明一个查找方法，如以下示例所示：

```
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，更惯用的是，您可以依赖于目标 bean 根据查找方法的声明的返回类型来解析：

```
public abstract class CommandManager {  
  
    public Object process(Object commandState) {  
        MyCommand command = createCommand();  
        command.setState(commandState);  
        return command.execute();  
    }  
  
    @Lookup  
    protected abstract MyCommand createCommand();  
}
```

请注意，通常应使用具体的存根实现声明此类带 Comments 的查找方法，以便它们与 Spring 的组件扫描规则兼容，在默认情况下抽象类将被忽略。此限制不适用于显式注册或显式导入的 Bean 类。

### Tip

访问范围不同的目标 bean 的另一种方法是 `ObjectFactory` / `Provider` 注入点。参见 [范围 bean 作为依赖项](#)。

您可能还会发现 `ServiceLocatorFactoryBean` (在 `org.springframework.beans.factory.config` 包中) 很有用。

## 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种方法实现替换托管 bean 中的任意方法。您可以放心地跳过本节的其余部分，直到您 `true` 需要此功能为止。

借助基于 XML 的配置元数据，您可以使用 `replaced-method` 元素将现有的方法实现替换为已部署的 Bean。考虑下面的类，它具有一个名为 `computeValue` 的方法，我们想重写该方法：

```
public class MyValueCalculator {  
    public String computeValue(String input) {  
        // some real code...  
    }  
    // some other methods...  
}
```

实现 `org.springframework.beans.factory.support.MethodReplacer` 接口的类提供了新的方法定义，如以下示例所示：

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

用于部署原始类并指定方法覆盖的 Bean 定义类似于以下示例：

```
<bean id="myValueCalculator" class="x.y.z.MyValueCalculator">  
    <!-- arbitrary method replacement -->  
    <replaced-method name="computeValue" replacer="replacementComputeValue">
```

```

<arg-type>String</arg-type>
</replaced-method>
</bean>

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

```

您可以在 `<replaced-method/>` 元素中使用一个或多个 `<arg-type/>` 元素来指示要覆盖的方法的方法签名。仅当方法重载且类中存在多个变体时，才需要对参数签名。为了方便起见，参数的类型字符串可以是完全限定类型的子字符串。例如，以下所有都匹配 `java.lang.String`：

```

java.lang.String
String
Str

```

因为参数的数量通常足以区分每个可能的选择，所以通过让您仅键入与参数类型匹配的最短字符串，此快捷方式可以节省很多 Importing。

## 1.5. bean 范围

创建 bean 定义时，将创建一个配方来创建该 bean 定义所定义的类的实际实例。bean 定义是配方的想法很重要，因为它意味着与类一样，您可以从一个配方中创建许多对象实例。

您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。这种方法功能强大且灵活，因为您可以选择通过配置创建的对象的范围，而不必在 Java 类级别上烘烤对象的范围。可以将 Bean 定义为部署在多个范围之一中。

Spring 框架支持六个范围，其中只有在使用网络感知 `ApplicationContext` 时才可用。您也可以创建[自定义范围](#)。

下表描述了受支持的范围：

表 3. Bean 作用域

Scope	Description
<a href="#">singleton</a>	(默认) 将每个 Spring IoC 容器的单个 bean 定义范围限定为单个对象实例

Scope	Description
	◦
<a href="#">prototype</a>	将单个 bean 定义的作用域限定为任意数量的对象实例。
<a href="#">request</a>	将单个 bean 定义的范围限定为单个 HTTP 请求的生命周期。也就是说，每个 HTTP 请求都有一个在单个 bean 定义后面创建的 bean 实例。仅在可感知网络的 Spring <code>ApplicationContext</code> 中有效。
<a href="#">session</a>	将单个 bean 定义的范围限定为 HTTP <code>Session</code> 的生命周期。仅在可感知网络的 Spring <code>ApplicationContext</code> 上下文中有效。
<a href="#">application</a>	将单个 bean 定义的范围限定为 <code>ServletContext</code> 的生命周期。仅在可感知网络的 Spring <code>ApplicationContext</code> 上下文中有效。
<a href="#">websocket</a>	将单个 bean 定义的范围限定为 <code>WebSocket</code> 的生命周期。仅在可感知网络的 Spring <code>ApplicationContext</code> 上下文中有效。

### iNote

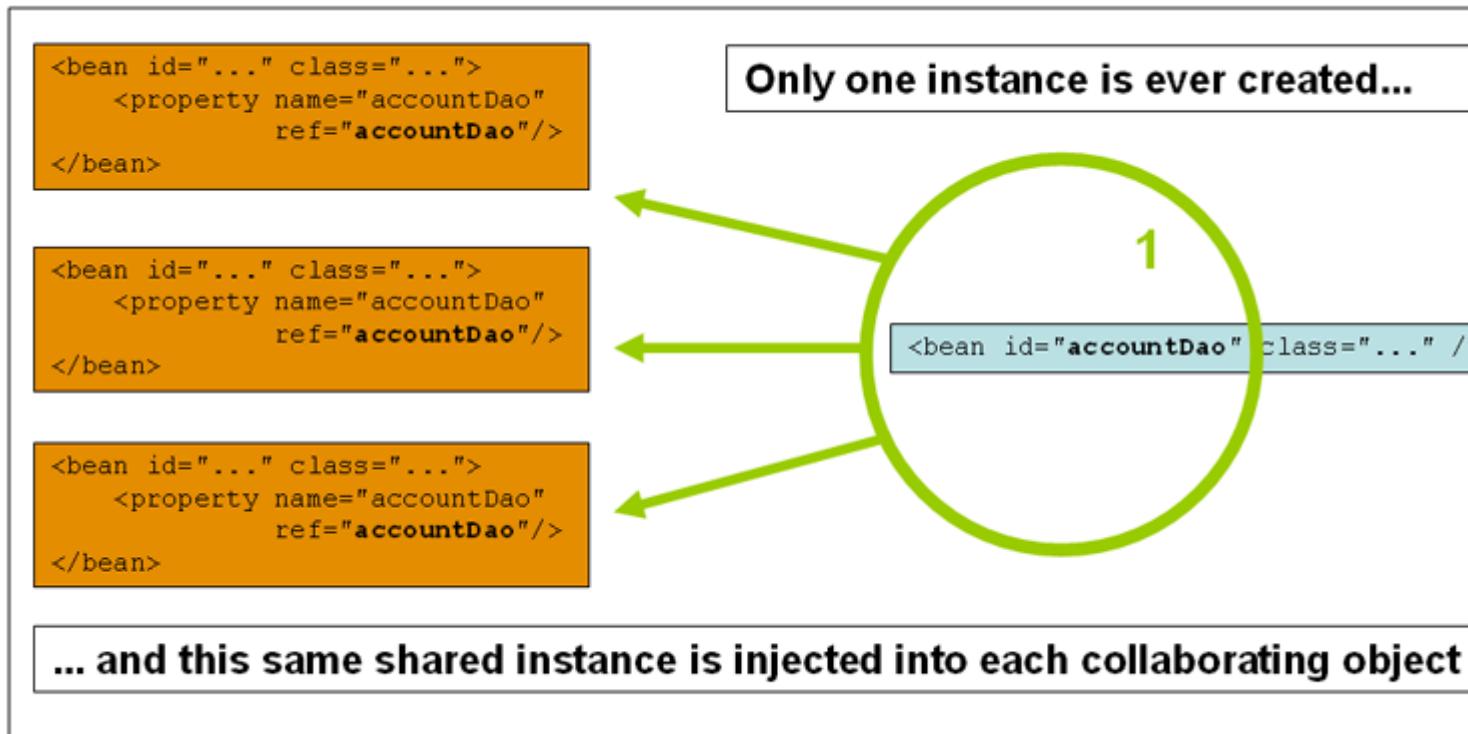
从 Spring 3.0 开始，线程作用域可用，但默认情况下未注册。有关更多信息，请参见[SimpleThreadScope](#)的文档。有关如何注册此范围或任何其他自定义范围的说明，请参见[使用自定义范围](#)。

## 1.5.1. 单例范围

仅 Management 一个 singleton bean 的一个共享实例，并且所有对具有 ID 或与该 bean 定义相匹

配的 ID 的 bean 的请求都会导致该特定的 bean 实例由 Spring 容器返回。

换句话说，当您定义一个 bean 定义并且其作用域为单例时，Spring IoC 容器将为该 bean 定义所定义的对象创建一个实例。该单个实例存储在此类单例 bean 的高速缓存中，并且对该命名 bean 的所有后续请求和引用都返回该高速缓存的对象。下图显示了单例作用域如何工作：



Spring 的 singleton bean 的概念与“四人帮”(GoF)模式一书中定义的 singleton 模式不同。GoF 单例对对象的范围进行硬编码，以使每个 ClassLoader 只能创建一个特定类的一个实例。最好将 Spring 单例的范围描述为每个容器和每个 bean。这意味着，如果您在单个 Spring 容器中为特定类定义一个 bean，则 Spring 容器将创建该 bean 定义所定义的类的一个且只有一个实例。 Singleton 范围是 Spring 中的默认范围。要将 bean 定义为 XML 中的单例，可以定义 bean，如以下示例所示：

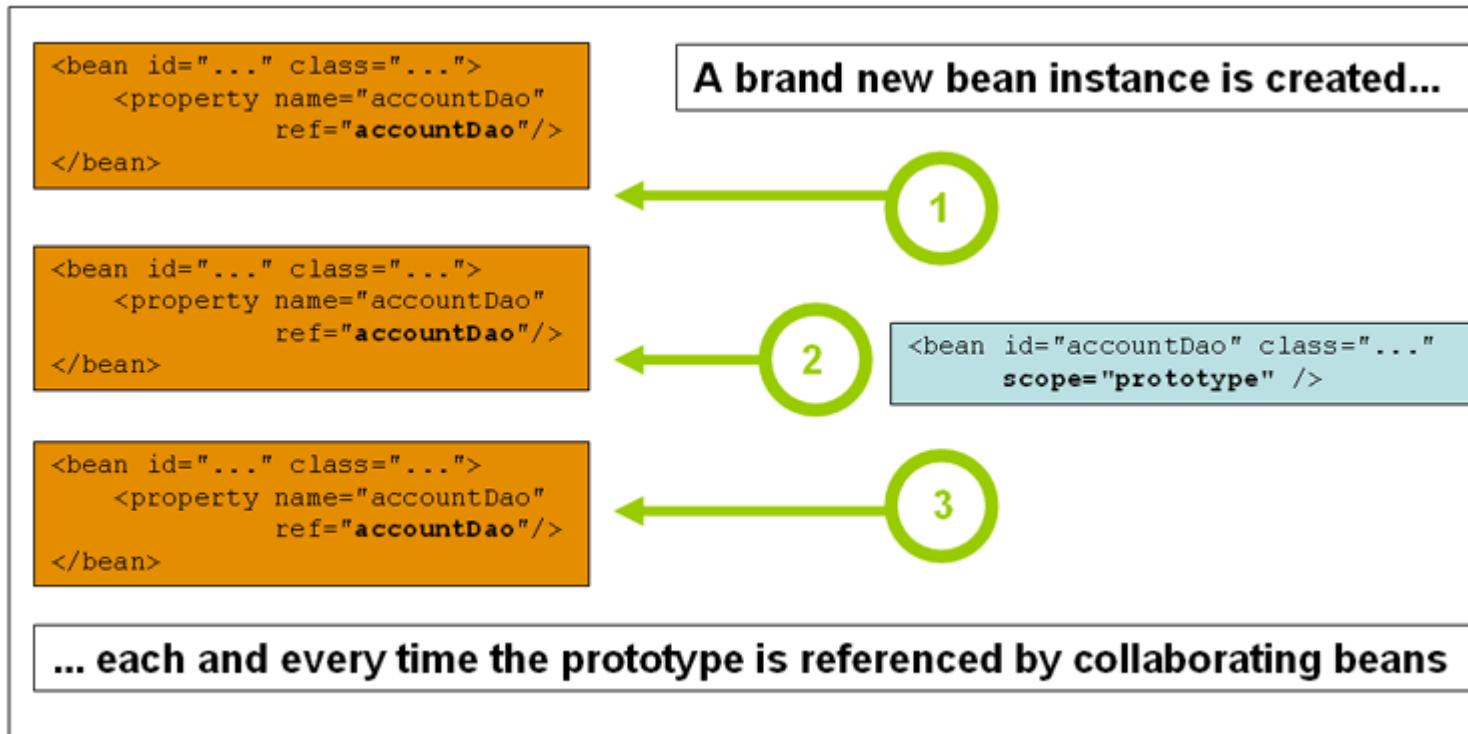
```
<bean id="accountService" class="com.something.DefaultAccountService"/>
<!-- the following is equivalent, though redundant (singleton scope is the default) -->
<bean id="accountService" class="com.something.DefaultAccountService" scope="singleton">
```

## 1.5.2. 原型范围

每次对特定 bean 提出请求时，bean 部署的非单一原型范围都会导致创建一个新 bean 实例。也就

是说，将 Bean 注入到另一个 Bean 中，或者您可以通过容器上的 `getBean()` 方法调用来请求它。通常，应将原型作用域用于所有有状态 Bean，将单例作用域用于 StatelessBean。

下图说明了 Spring 原型范围：



(数据访问对象(DAO)通常不配置为原型，因为典型的 DAO 不拥有任何对话状态。对于我们而言，重用单例图的核心更为容易。)

以下示例将 bean 定义为 XML 原型：

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

与其他作用域相反，Spring 不 Management 原型 Bean 的完整生命周期。容器将实例化，配置或组装原型对象，然后将其交给 Client 端，而无需对该原型实例的进一步记录。因此，尽管在不考虑范围的情况下在所有对象上都调用了初始化生命周期回调方法，但在原型的情况下，不会调用已配置的销毁生命周期回调。Client 端代码必须清除原型作用域内的对象，并释放原型 Bean 拥有的昂贵资源。要使 Spring 容器释放由原型作用域的 bean 占用的资源，请尝试使用自定义[bean post-processor](#)，该自变量\_1 包含对需要清理的 bean 的引用。

在某些方面，Spring 容器在原型作用域 bean 方面的角色是 Java `new` 运算符的替代。超过该时间

点的所有生命周期 Management 必须由 Client 端处理。(有关 Spring 容器中 bean 的生命周期的详细信息, 请参见[Lifecycle Callbacks](#)。)

### 1.5.3. 具有原型 Bean 依赖关系的 Singleton Bean

当您使用对原型 bean 有依赖性的单例作用域 Bean 时, 请注意, 依赖关系在实例化时已解决。因此, 如果将依赖项原型的 bean 依赖项注入到单例范围的 bean 中, 则将实例化新的原型 bean, 然后将依赖项注入到单例 bean 中。原型实例是曾经提供给单例范围的 bean 的唯一实例。

但是, 假设您希望单例作用域的 bean 在运行时重复获取原型作用域的 bean 的新实例。您不能将原型作用域的 bean 依赖项注入到您的单例 bean 中, 因为当 Spring 容器实例化单例 bean 并解析并注入其依赖项时, 该注入仅发生一次。如果在运行时多次需要一个原型 bean 的新实例, 请参见[Method Injection](#)

### 1.5.4. 请求, 会话, 应用程序和 WebSocket 范围

`request`, `session`, `application` 和 `websocket` 范围仅在使用 Web 感知的 Spring `ApplicationContext` 实现(例如 `XmlWebApplicationContext`)时可用。如果将这些作用域与常规的 Spring IoC 容器(例如 `ClassPathXmlApplicationContext`)一起使用, 则会引发抱怨未知 bean 作用域的 `IllegalStateException`。

#### 初始 Web 配置

为了支持 `request`, `session`, `application` 和 `websocket` 级别(网络范围的 Bean)的 Bean 范围界定, 在定义 Bean 之前, 需要一些较小的初始配置。(对于标准范围 `singleton` 和 `prototype`, 不需要此初始设置。)

如何完成此初始设置取决于您的特定 Servlet 环境。

如果实际上在 Spring `DispatcherServlet` 处理的请求中访问 Spring Web MVC 中的作用域 Bean, 则不需要特殊的设置。`DispatcherServlet` 已经公开了所有相关状态。

如果您使用 Servlet 2.5 Web 容器，并且在 Spring 的 `DispatcherServlet` 之外处理请求(例如，当使用 JSF 或 Struts 时)，则需要注册

```
org.springframework.web.context.request.RequestContextListener
```

`ServletRequestListener`。对于 Servlet 3.0，可以使用 `WebApplicationInitializer` 接口以编程方式完成此操作。或者，或者对于较旧的容器，将以下声明添加到 Web 应用程序的 `web.xml` 文件中：

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

另外，如果您的监听器设置存在问题，请考虑使用 Spring 的 `RequestContextFilter`。过滤器 Map 取决于周围的 Web 应用程序配置，因此您必须适当地对其进行更改。以下清单显示了 Web 应用程序的过滤器部分：

```
<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` 都做完全相同的事情，即将 HTTP 请求对象绑定到正在为该请求提供服务的 `Thread` 上。这使得在请求链和会话范围内的 Bean 可以在调用链的更下游使用。

## Request scope

考虑以下 XML 配置来定义 bean:

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

Spring 容器通过为每个 HTTP 请求使用 `loginAction` bean 定义来创建 `LoginAction` bean 的新实例。也就是说，`loginAction` bean 的作用域是 HTTP 请求级别。您可以根据需要更改创建实例的内部状态，因为从同一 `loginAction` bean 定义创建的其他实例看不到这些状态更改。它们特定于单个请求。当请求完成处理时，将限制作用于该请求的 Bean。

使用 Comments 驱动的组件或 Java 配置时，可以使用 `@RequestScope` Comments 将组件分配给 `request` 范围。以下示例显示了如何执行此操作：

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

## Session Scope

考虑以下 XML 配置来定义 bean:

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

在单个 HTTP `Session` 的生存期内，Spring 容器通过使用 `userPreferences` bean 定义来创建 `UserPreferences` bean 的新实例。换句话说，`userPreferences` bean 的作用域实际上是 HTTP `Session` 级别。与请求范围的 Bean 一样，您可以根据需要任意更改所创建实例的内部状态，因为知道其他 HTTP `Session` 实例(也使用从相同 `userPreferences` Bean 定义创建的实例)不会看到这些状态更改，因为它们特定于单个 HTTP `Session`。当最终丢弃 HTTP `Session` 时，也将丢弃作用于该特定 HTTP `Session` 的 bean。

使用 Comments 驱动的组件或 Java 配置时，可以使用 `@SessionScope` Comments 将组件分配给

`session` 范围。

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

## Application Scope

考虑以下 XML 配置来定义 bean:

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

Spring 容器通过对整个 Web 应用程序使用 `appPreferences` bean 定义来创建

`AppPreferences` bean 的新实例。也就是说，`appPreferences` bean 的作用域为 `ServletContext` 级别，并存储为常规 `ServletContext` 属性。这有点类似于 Spring 单例 bean，但是有两个重要的区别：它是每个 `ServletContext` 而不是每个 Spring'ApplicationContext'(在任何给定的 Web 应用程序中可能都有多个)，并且实际上是公开的，因此可见为 `ServletContext` 属性。

使用 Comments 驱动的组件或 Java 配置时，可以使用 `@ApplicationScope` Comments 将组件分配给 `application` 范围。以下示例显示了如何执行此操作：

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

## 作用域 Bean 作为依赖项

Spring IoC 容器不仅 Management 对象(bean)的实例化，而且还 Management 协作者(或依赖项)的连接。如果要将(例如)HTTP 请求范围的 Bean 注入(例如)另一个作用域更长的 Bean，则可以选择注入 AOP 代理来代替已定义范围的 Bean。也就是说，您需要注入一个代理对象，该对象公开与

范围对象相同的公共接口，但也可以从相关范围(例如 HTTP 请求)中检索实际目标对象，并将方法调用委托给该真实对象。

### iNote

您还可以在范围为 `singleton` 的 bean 之间使用 `<aop:scoped-proxy/>`，然后引用将通过可序列化的中间代理，因此可以在反序列化时重新获得目标单例 bean。

当针对范围为 `prototype` 的 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新的目标实例，然后将该调用转发到该目标实例。

同样，作用域代理不是以生命周期安全的方式从较短的作用域访问 bean 的唯一方法。您也可以将注入点(即构造器或 setter 参数或自动装配的字段)声明为

`ObjectFactory<MyTargetBean>`，从而允许 `getObject()` 调用在需要时每次按需检索当前实例-holding 无需保留该实例或对其进行存储分别。

作为扩展变体，您可以声明 `ObjectProvider<MyTargetBean>`，它提供了几个附加的访问变体，包括 `getIfAvailable` 和 `getIfUnique`。

JSR-330 的这种变体称为 `Provider`，并且每次检索尝试都使用

`Provider<MyTargetBean>` 声明和相应的 `get()` 调用。有关 JSR-330 总体的更多详细信息，请参见[here](#)。

以下示例中的配置仅一行，但是了解其背后的“原因”和“方式”很重要：

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

    <!-- an HTTP Session-scoped bean exposed as a proxy -->
    <bean id="userPreferences" class="com.something.UserPreferences" scope="session">
        <!-- instructs the container to proxy the surrounding bean -->
        <aop:scoped-proxy/> (1)
    </bean>
</beans>
```

```

</bean>

<!-- 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>

```

- (1) 定义代理的行。

要创建这样的代理，请将子 `<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 `request`，`session` 和自定义范围级别定义的 bean 定义需要 `<aop:scoped-proxy/>` 元素？考虑以下单例 bean 定义，并将其与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，单例 bean(`userManager`)注入了对 HTTP `Session` 范围的 bean(`userPreferences`)的引用。这里的重点是 `userManager` bean 是单例的：每个容器仅实例化一次，并且它的依赖项(在这种情况下，仅一个 `userPreferences` bean)也仅注入一次。这意味着 `userManager` bean 仅在完全相同的 `userPreferences` 对象(即最初与之注入对象)上操作。

将寿命较短的作用域 bean 注入寿命较长的作用域 bean 时，这不是您想要的行为(例如，将 HTTP `Session` 范围的协作 bean 作为依赖项注入到 singleton bean 中)。相反，您只需要一个 `userManager` 对象，并且在 HTTP `Session` 的生存期内，您需要一个特定于 HTTP `Session` 的 `userPreferences` 对象。因此，容器创建了一个对象，该对象公开了与 `UserPreferences` 类完全相同的公共接口(理想情况下是 `UserPreferences` 实例的对象)，该对象可以从范围机制(HTTP 请求，`Session` 等)中获取实际的 `UserPreferences` 对象。。容器将此代理对象注入到

`userManager` bean 中，而后者不知道此 `UserPreferences` 引用是代理。在此示例中，当 `UserManager` 实例在注入依赖项的 `UserPreferences` 对象上调用方法时，实际上是在代理上调用方法。然后，代理从 HTTP `Session` (在这种情况下) 获取真实的 `UserPreferences` 对象，并将方法调用委托给检索到的真实的 `UserPreferences` 对象。

因此，将 `request-` 和 `session-scoped` bean 注入到协作对象中时，您需要以下(正确和完整)配置，如以下示例所示：

```
<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>
```

## 选择要创建的代理类型

默认情况下，当 Spring 容器为使用 `<aop:scoped-proxy/>` 元素标记的 bean 创建代理时，将创建基于 CGLIB 的类代理。

### iNote

CGLIB 代理仅拦截公共方法调用！不要在此类代理上调用非公共方法。它们没有被委派给实际的作用域目标对象。

另外，您可以通过将 `<aop:scoped-proxy/>` 元素的 `proxy-target-class` 属性的值指定为 `false` 来配置 Spring 容器为此类作用域的 bean 创建基于标准 JDK 接口的代理。使用基于 JDK 接口的代理意味着您不需要应用程序 Classpath 中的其他库即可影响此类代理。但是，这也意味着作用域 bean 的类必须实现至少一个接口，并且作用域 bean 注入到其中的所有协作者都必须通过其接口之一引用该 bean。以下示例显示基于接口的代理：

```
<!-- DefaultUserPreferences implements the UserPreferences interface -->
<bean id="userPreferences" class="com.stuff.DefaultUserPreferences" scope="session">
```

```
<aop:scoped-proxy proxy-target-class="false" />
</bean>

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

有关选择基于类或基于接口的代理的更多详细信息，请参见[Proxifying Mechanisms](#)。

### 1.5.5. 自定义范围

Bean 作用域机制是可扩展的。您可以定义自己的范围，甚至重新定义现有范围，尽管后者被认为 是不好的做法，并且您不能覆盖内置的 `singleton` 和 `prototype` 范围。

#### 创建自定义范围

要将自定义范围集成到 Spring 容器中，您需要实现

`org.springframework.beans.factory.config.Scope` 接口，本节对此进行了介绍。有关如何 实现自己的范围的想法，请参阅 Spring Framework 本身提供的 `Scope` 实现和 `Scope` javadoc，它 们详细说明了需要实现的方法。

`Scope` 接口有四种方法可以从范围中获取对象，从范围中删除对象，然后销毁它们。

例如，会话范围实现返回会话范围的 Bean(如果不存在，则该方法将其绑定到会话以供将来参考之 后，将返回该 Bean 的新实例)。以下方法从基础范围返回对象：

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

会话范围的实现，例如，从基础会话中删除了会话范围的 bean。应该返回该对象，但是如果找 不到具有指定名称的对象，则可以返回 null。以下方法从基础范围内删除该对象：

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

以下方法注册在销毁作用域或销毁作用域中的指定对象时作用域应执行的回调：

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

有关销毁回调的更多信息，请参见[javadoc](#)或 Spring 范围实现。

以下方法获取基础范围的会话标识符：

```
String getConversationId()
```

每个范围的标识符都不相同。对于会话范围的实现，此标识符可以是会话标识符。

## 使用自定义范围

在编写并测试一个或多个自定义 `Scope` 实现之后，您需要使 Spring 容器意识到您的新作用域。以下方法是在 Spring 容器中注册新的 `Scope` 的中心方法：

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

此方法在 `ConfigurableBeanFactory` 接口上声明，该接口可通过 Spring 附带的大多数具体 `ApplicationContext` 实现上的 `BeanFactory` 属性使用。

`registerScope(...)` 方法的第一个参数是与范围关联的唯一名称。Spring 容器本身中的此类名称示例为 `singleton` 和 `prototype`。`registerScope(...)` 方法的第二个参数是您希望注册和使用的自定义 `Scope` 实现的实际实例。

假设您编写了自定义 `Scope` 实现，然后如以下示例中所示进行注册。

### iNote

下一个示例使用 `SimpleThreadScope`，它已包含在 Spring 中，但默认情况下未注册。这些说明与您自己的自定义 `Scope` 实现相同。

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

然后，您可以创建符合自定义 `Scope` 范围规则的 bean 定义，如下所示：

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

使用自定义 `Scope` 实现，您不仅可以通过程序注册该范围。您还可以通过使用

`CustomScopeConfigurer` 类以声明方式进行 `Scope` 注册，如以下示例所示：

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

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

### iNote

当您将 `<aop:scoped-proxy/>` 放在 `FactoryBean` 实现中时，作用域是工厂 bean 本身，而不是 `getObject()` 返回的对象。

## 1.6. 自定义 bean 的性质

Spring 框架提供了许多接口，可用于自定义 Bean 的性质。本节将它们分组如下：

- [Lifecycle Callbacks](#)

- [ApplicationContextAware 和 BeanNameAware](#)
- [其他感知接口](#)

### 1.6.1. 生命周期回调

要与容器对 bean 生命周期的 Management 进行交互，可以实现 Spring `InitializingBean` 和 `DisposableBean` 接口。容器对前者调用 `afterPropertiesSet()`，对后者调用 `destroy()`，以使 Bean 在初始化和销毁 Bean 时执行某些操作。

#### Tip

通常，在现代 Spring 应用程序中，JSR-250 `@PostConstruct` 和 `@PreDestroy` Comments 被认为是接收生命周期回调的最佳实践。使用这些 Comments 意味着您的 bean 没有耦合到特定于 Spring 的接口。有关详细信息，请参见[使用@PostConstruct 和@PreDestroy](#)。

如果您不想使用 JSR-250 注解，但仍然想删除耦合，请考虑使用 `init-method` 和 `destroy-method` 对象定义元数据。

在内部，Spring 框架使用 `BeanPostProcessor` 实现来处理它可以找到的任何回调接口并调用适当的方法。如果您需要自定义功能或其他生命周期行为，Spring 默认不提供，则您可以自己实现 `BeanPostProcessor`。有关更多信息，请参见[集装箱延伸点](#)。

除了初始化和销毁回调之外，Spring 托管的对象还可以实现 `Lifecycle` 接口，以便这些对象可以在容器自身生命周期的驱动下参与启动和关闭过程。

本节介绍了生命周期回调接口。

## Initialization Callbacks

`org.springframework.beans.factory.InitializingBean` 接口允许容器在容器上设置了所有

必需的属性后，bean 可以执行初始化工作。`InitializingBean` 接口指定一个方法：

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

我们建议您不要使用 `InitializingBean` 接口，因为它不必要地将代码耦合到 Spring。另外，我们建议使用 `@PostConstruct` 注解或指定 POJO 初始化方法。对于基于 XML 的配置元数据，我们可以使用 `init-method` 属性指定具有无参数签名的方法的名称。通过 Java 配置，可以使用 `@Bean` 的 `initMethod` 属性。参见 [接收生命周期回调](#)。考虑以下示例：

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

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

前面的示例与下面的示例(由两个清单组成)几乎具有完全相同的效果：

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

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

但是，前面两个示例中的第一个示例并未将代码耦合到 Spring。

## Destruction Callbacks

实现 `org.springframework.beans.factory.DisposableBean` 接口后，当包含 bean 的容器被销毁时，bean 可以获取回调。`DisposableBean` 接口指定一个方法：

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

我们建议您不要使用 `DisposableBean` 回调接口，因为它不必要地将代码耦合到 Spring。另外，我们建议使用 `@PreDestroy` 注解或指定 bean 定义支持的通用方法。使用基于 XML 的配置元数据时，可以在 `<bean/>` 上使用 `destroy-method` 属性。通过 Java 配置，可以使用 `@Bean` 的 `destroyMethod` 属性。参见 [接收生命周期回调](#)。考虑以下定义：

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

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

前面的定义与下面的定义几乎具有完全相同的效果：

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

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

但是，前面两个定义中的第一个没有将代码耦合到 Spring。

### Tip

您可以为 `<bean>` 元素的 `destroy-method` 属性分配特殊的 `(inferred)` 值，该值指示 Spring 自动检测特定 bean 类上的公共 `close` 或 `shutdown` 方法。(因此，实现 `java.lang.AutoCloseable` 或 `java.io.Closeable` 的任何类都将匹配)您还可以在 `<beans>` 元素的 `default-destroy-method` 属性上设置此特殊的 `(inferred)` 值，以将此行为应用于整个 bean 组(请参见[默认初始化和销毁方法](#))。请注意，这是 Java 配置的默认行为。

## 默认初始化和销毁方法

当您编写不使用特定于 Spring 的 `InitializingBean` 和 `DisposableBean` 回调接口的初始化和销毁方法回调时，通常会编写诸如 `init()`，`initialize()`，`dispose()` 等名称的方法。理想情况下，此类生命周期回调方法的名称应在整个项目中标准化，以便所有开发人员都使用相同的方法名称并确保一致性。

您可以将 Spring 容器配置为“查找”命名的初始化，并销毁每个 bean 上的回调方法名称。这意味着，作为应用程序开发人员，您可以编写应用程序类并使用名为 `init()` 的初始化回调，而不必为每个 bean 定义配置 `init-method="init"` 属性。Spring IoC 容器在创建 bean 时(并根据标准生命周期回调协定[described previously](#))调用该方法。此功能还对初始化和销毁方法回调强制执行一致的命名约定。

假设您的初始化回调方法命名为 `init()`，而 `destroy` 回调方法命名为 `destroy()`。然后，您的类类似于以下示例中的类：

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

然后，您可以在类似于以下内容的 Bean 中使用该类：

```
<beans default-init-method="init">  
  
    <bean id="blogService" class="com.something.DefaultBlogService">  
        <property name="blogDao" ref="blogDao" />  
    </bean>  
  
</beans>
```

顶层 `<beans/>` 元素属性上 `default-init-method` 属性的存在会导致 Spring IoC 容器将 Bean 类上称为 `init` 的方法识别为初始化方法回调。创建和组装 bean 时，如果 bean 类具有此类方法，则会在适当的时间调用它。

您可以通过使用顶级 `<beans/>` 元素上的 `default-destroy-method` 属性类似地(在 XML 中)配置 `destroy` 方法回调。

如果现有的 Bean 类已经具有按惯例命名的回调方法，则可以通过使用 `<bean/>` 本身的 `init-method` 和 `destroy-method` 属性指定(在 XML 中)方法名称来覆盖默认值。

Spring 容器保证在为 bean 提供所有依赖项后立即调用配置的初始化回调。因此，在原始 bean 引用上调用了初始化回调，这意味着 AOP 拦截器等尚未应用于 bean。首先完全创建目标 bean，然后应用带有其拦截器链的 AOP 代理(例如)。如果目标 Bean 和代理分别定义，则您的代码甚至可以绕过代理与原始目标 Bean 进行交互。因此，将拦截器应用于 `init` 方法将是不一致的，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。

## 组合生命周期机制

从 Spring 2.5 开始，您可以使用三个选项来控制 Bean 生命周期行为：

- [InitializingBean](#) 和 [DisposableBean](#) 回调接口
- 自定义 `init()` 和 `destroy()` 方法
- [@PostConstruct](#) 和 [@PreDestroy](#) 注解。您可以结合使用这些机制来控制给定的 bean。

### iNote

如果为一个 bean 配置了多个生命周期机制，并且为每个机制配置了不同的方法名称，则将按照此 Comments 后列出的 Sequences 执行每个已配置的方法。但是，如果为多个生命周期机制中的多个生命周期配置了相同的方法名称(例如，对于初始化方法使用 `init()`)，则

该方法将执行一次，如 [preceding section](#) 中所述。

为同一个 bean 配置的具有不同初始化方法的多种生命周期机制如下：

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

销毁方法的调用 Sequences 相同：

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

## 启动和关闭回调

`Lifecycle` 接口为具有自己的生命周期要求(例如启动和停止某些后台进程)的任何对象定义基本方法：

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

任何 SpringManagement 的对象都可以实现 `Lifecycle` 接口。然后，当 `ApplicationContext` 本身接收到启动和停止 signal 时(例如，对于运行时的停止/重新启动场景)，它将把这些调用级联到在该上下文中定义的所有 `Lifecycle` 实现。它通过委托 `LifecycleProcessor` 来完成此任务，如以下清单所示：

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

请注意，`LifecycleProcessor` 本身是 `Lifecycle` 接口的扩展。它还添加了两种其他方法来响应正在刷新和关闭的上下文。

### Tip

请注意，常规 `org.springframework.context.Lifecycle` 接口是用于显式启动和停止通知的普通协议，并不意味着在上下文刷新时自动启动。为了对特定 bean 的自动启动进行精细控制(包括启动阶段)，请考虑实现 `org.springframework.context.SmartLifecycle`。

另外，请注意，不能保证会在销毁之前发出停止通知。在常规关闭时，在传播常规销毁回调之前，所有 `Lifecycle` bean 都首先收到停止通知。但是，在上下文生存期内的热刷新或中止的刷新尝试中，仅调用 `destroy` 方法。

启动和关闭调用的 Sequences 可能很重要。如果任何两个对象之间存在“依赖”关系，则依赖方在其依赖之后开始，而在依赖之前停止。但是，有时直接依赖项是未知的。您可能只知道某种类型的对象应该先于另一种类型的对象开始。在这些情况下，`SmartLifecycle` 接口定义了另一个选项，即在其超级接口 `Phased` 上定义的 `getPhase()` 方法。以下清单显示了 `Phased` 接口的定义：

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

以下清单显示了 `SmartLifecycle` 接口的定义：

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

```
}
```

启动时，相位最低的对象首先启动。停止时，遵循相反的 Sequences。因此，实现 `SmartLifecycle` 且其 `getPhase()` 方法返回 `Integer.MIN_VALUE` 的对象将是第一个启动且最后一个停止的对象。在频谱的另一端，相位值 `Integer.MAX_VALUE` 表示该对象应最后启动并首先停止(可能是因为它取决于正在运行的其他进程)。考虑相位值时，重要的是要知道，任何未实现 `SmartLifecycle` 的“正常” `Lifecycle` 对象的默认相位是 `0`。因此，任何负相位值都表明对象应在这些标准组件之前开始(并在它们之后停止)。对于任何正相位值，反之亦然。

`SmartLifecycle` 定义的 `stop` 方法接受回调。在该实现的关闭过程完成之后，任何实现都必须调用该回调的 `run()` 方法。这将在必要时启用异步关闭，因为 `LifecycleProcessor` 接口的默认实现 `DefaultLifecycleProcessor` `await` 每个阶段内的对象组的超时值，以调用该回调。默认的每阶段超时为 30 秒。您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，则定义以下内容即可：

```
<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。后者驱动关闭过程，就好像已明确调用 `stop()` 一样，但是在上下文关闭时会发生。另一方面，“刷新”回调启用 `SmartLifecycle` bean 的另一个功能。刷新上下文时(在所有对象都被实例化和初始化之后)，该回调将被调用。那时，默认生命周期处理器将检查每个 `SmartLifecycle` 对象的 `isAutoStartup()` 方法返回的布尔值。如果 `true`，则在该点启动该对象，而不是 `await` 上下文或它自己的 `start()` 方法的显式调用(与上下文刷新不同，对于标准的上下文实现，上下文启动不会自动发生)。`phase` 值和任何“依赖”关系确定启动 Sequences，如前所述。

## 在非 Web 应用程序中正常关闭 Spring IoC 容器

## iNote

本节仅适用于非 Web 应用程序。Spring 的基于 Web 的 `ApplicationContext` 实现已经具有适当的代码，可以在相关 Web 应用程序关闭时正常关闭 Spring IoC 容器。

如果您在非 Web 应用程序环境中(例如，在富 Client 端桌面环境中)使用 Spring 的 IoC 容器，请向 JVM 注册一个关闭钩子。这样做可以确保正常关机，并在您的 Singleton bean 上调用相关的 `destroy` 方法，以便释放所有资源。您仍然必须正确配置和实现这些 `destroy` 回调。

要注册关闭钩子，请调用在 `ConfigurableApplicationContext` 接口上声明的

`registerShutdownHook()` 方法，如以下示例所示：

```
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` 创建实现

对该 `ApplicationContext` 的引用。以下清单显示了 `ApplicationContextAware` 接口的定义：

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

因此，bean 可以通过 `ApplicationContext` 接口或通过将引用转换为该接口的已知子类(例如 `ConfigurableApplicationContext`，以公开其他功能)来以编程方式操纵创建它们的 `ApplicationContext`。一种用途是通过编程方式检索其他 bean。有时，此功能很有用。但是，通常应避免使用它，因为它将代码耦合到 Spring，并且不遵循控制反转样式，在该样式中，将协作者作为属性提供给 bean。`ApplicationContext` 的其他方法提供对文件资源的访问，发布应用程序事件以及对 `MessageSource` 的访问。这些附加功能在 [ApplicationContext 的其他功能](#) 中描述。

从 Spring 2.5 开始，自动装配是获得对 `ApplicationContext` 的引用的另一种选择。“传统” `constructor` 和 `byType` 自动装配模式(如[Autowiring Collaborators](#) 中所述)可以分别为构造函数参数或 `setter` 方法参数提供 `ApplicationContext` 类型的依赖项。为了获得更大的灵活性，包括能够自动连接字段和使用多个参数方法，请使用基于 `Comments` 的新自动装配功能。如果这样做，`ApplicationContext` 将自动连接到期望使用 `ApplicationContext` 类型的字段，构造函数自变量或方法参数中(如果有问题的字段，构造函数或方法带有 `@Autowired` `Comments`)。有关更多信息，请参见[Using @Autowired](#)。

当 `ApplicationContext` 创建实现 `org.springframework.beans.factory.BeanNameAware` 接口的类时，该类将获得对在其关联的对象定义中定义的名称的引用。以下清单显示了 `BeanNameAware` 接口的定义：

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

在填充常规 bean 属性之后但在初始化回调(例如 `InitializingBean`，`afterPropertiesSet` 或自定义 `init-method`)之前调用该回调。

### 1.6.3. 其他感知接口

除了 `ApplicationContextAware` 和 `BeanNameAware` (已讨论`earlier`)之外, Spring 还提供了 `Aware` 接口范围, 这些接口使 Bean 向容器指示它们需要某种基础结构依赖性。通常, 该名称很好地表明了依赖项类型。下表总结了最重要的 `Aware` 接口:

表 4. 感知接口

Name	Injected Dependency	Explained in...
<code>ApplicationContextAware</code>	声明 <code>ApplicationContext</code> 。	<a href="#">ApplicationContextAware</a> 和 <a href="#">BeanNameAware</a>
<code>ApplicationEventPublisherAware</code>	附件 <code>ApplicationContext</code> 的事件发布者。	<a href="#">ApplicationContext 的其他功能</a>
<code>BeanClassLoaderAware</code>	类加载器, 用于加载 Bean 类。	<a href="#">Instantiating Beans</a>
<code>BeanFactoryAware</code>	声明 <code>BeanFactory</code> 。	<a href="#">ApplicationContextAware</a> 和 <a href="#">BeanNameAware</a>
<code>BeanNameAware</code>	声明 bean 的名称。	<a href="#">ApplicationContextAware</a> 和 <a href="#">BeanNameAware</a>
<code>BootstrapContextAware</code>	运行容器的资源适配器 <code>BootstrapContext</code> 。 通常仅在支持 JCA 的 <code>ApplicationContext</code>	<a href="#">JCA CCI</a>

Name	Injected Dependency	Explained in...
<code>LoadTimeWeaverAware</code>	实例中可用。 定义的编织器，用于在加载时处理类定义。	<a href="#">在 Spring Framework 中使用 AspectJ 进行加载时编织</a>
<code>MessageSourceAware</code>	解决消息的已配置策略 (支持参数化和国际化)。	<a href="#">ApplicationContext 的其他功能</a>
<code>NotificationPublisherAware</code>	Spring JMX 通知发布者 。	<a href="#">Notifications</a>
<code>ResourceLoaderAware</code>	配置的加载程序，用于对资源的低级别访问。	<a href="#">Resources</a>
<code>ServletConfigAware</code>	当前容器运行的 <code>ServletConfig</code> 。仅在可感知网络的 Spring <code>ApplicationContext</code> 中有效。	<a href="#">Spring MVC</a>
<code>ServletContextAware</code>	当前容器运行的 <code>ServletContext</code> 。仅在可感知网络的 Spring <code>ApplicationContext</code> 中有效。	<a href="#">Spring MVC</a>

再次注意，使用这些接口会将您的代码与 Spring API 绑定在一起，并且不遵循“控制反转”样式。因此，我们建议将它们用于需要以编程方式访问容器的基础结构 Bean。

## 1.7. Bean 定义继承

Bean 定义可以包含许多配置信息，包括构造函数参数，属性值和特定于容器的信息，例如初始化方法，静态工厂方法名称等。子 bean 定义从父定义继承配置数据。子定义可以覆盖某些值或根据需要添加其他值。使用父 bean 和子 bean 定义可以节省很多 Importing。实际上，这是一种模板形式。

如果您以编程方式使用 `ApplicationContext` 接口，则子 Bean 定义由 `ChildBeanDefinition` 类表示。大多数用户不在此级别上与他们合作。相反，它们在 `ClassPathXmlApplicationContext` 之类的类中声明性地配置 bean 定义。使用基于 XML 的配置元数据时，可以通过使用 `parent` 属性(将父 bean 指定为该属性的值)来指示子 bean 定义。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"> (1)
    <property name="name" value="override"/>
    <!-- the age property value of 1 will be inherited from parent -->
</bean>
```

- (1) 请注意 `parent` 属性。

如果未指定子 bean 定义，则使用父定义中的 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(也就是说，它必须接受父类的属性值)。

子 bean 定义从父对象继承范围，构造函数参数值，属性值和方法替代，并可以选择添加新值。您指定的任何范围，初始化方法，`destroy` 方法或 `static` 工厂方法设置都会覆盖相应的父设置。

其余设置始终从子定义中获取：依赖项，自动装配模式，依赖项检查，单例和惰性初始化。

前面的示例通过使用 `abstract` 属性将父 bean 定义显式标记为抽象。如果父定义未指定类，则需要将父 bean 定义显式标记为 `abstract`，如以下示例所示：

```
<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，并且还被明确标记为 `abstract`。当定义为 `abstract` 时，它只能用作纯模板 bean 定义，用作子定义的父定义。尝试单独使用此类 `abstract` 父 Bean(将其称为另一个 bean 的 ref 属性)或使用父 Bean ID 进行显式 `getBean()` 调用会返回错误。同样，容器的内部 `preInstantiateSingletons()` 方法将忽略定义为抽象的 bean 定义。

### 1 Note

`ApplicationContext` 默认情况下预先实例化所有单例。因此，重要的是(至少对于单例 bean)，如果您有一个(父)bean 定义仅打算用作模板，并且此定义指定了一个类，则必须确保设置\* `abstract` 属性为 `true`\*，否则应用程序上下文将实际(尝试)预先实例化 `abstract` bean。

## 1.8. 集装箱延伸点

通常，应用程序开发人员不需要将 `ApplicationContext` 实现类作为子类。相反，可以通过插入特殊集成接口的实现来扩展 Spring IoC 容器。接下来的几节描述了这些集成接口。

### 1.8.1. 使用 BeanPostProcessor 自定义 Bean

`BeanPostProcessor` 接口定义了回调方法，您可以实现这些回调方法，以提供自己的(或覆盖容器的默认值)实例化逻辑，依赖项解析逻辑等。如果您想在 Spring 容器完成实例化，配置和初始化 bean 之后实现一些自定义逻辑，则可以插入一个或多个 `BeanPostProcessor` 实现。

您可以配置多个 `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 Sequences。仅当 `BeanPostProcessor` 实现 `Ordered` 接口时，才可以设置此属性。如果您编写自己的 `BeanPostProcessor`，则也应该考虑实现 `Ordered` 接口。有关更多详细信息，请参见[BeanPostProcessor](#)和[Ordered](#)接口的 javadoc。另请参见[BeanPostProcessor 实例的编程注册](#)上的 Comments。

### iNote

`BeanPostProcessor` 个实例在 bean(或对象)实例上运行。也就是说，Spring IoC 容器实例化一个 bean 实例，然后 `BeanPostProcessor` 个实例完成其工作。

`BeanPostProcessor` 个实例按容器划分范围。仅在使用容器层次结构时，这才有意义。如果在一个容器中定义 `BeanPostProcessor`，则它将仅对该容器中的 bean 进行后处理。换句话说，一个容器中定义的 Bean 不会被另一个容器中定义的 `BeanPostProcessor` 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)，您需要使用 `BeanFactoryPostProcessor`，如[使用 BeanFactoryPostProcessor 自定义配置元数据](#)中所述。

`org.springframework.beans.factory.config.BeanPostProcessor` 接口恰好由两个回调方法组成。当此类被注册为容器的后处理器时，对于容器创建的每个 bean 实例，后处理器都会在容器初始化方法(例如 `InitializingBean.afterPropertiesSet()`，任何已声明的 `init` 之后)之前从容器获取回调。方法，并在任何 bean 初始化回调之后调用。后处理器可以对 bean 实例执行任何操作，包括完全忽略回调。Bean 后处理器通常检查回调接口，或者可以用代理包装 Bean。一些 Spring AOP 基础结构类被实现为 bean 后处理器，以提供代理包装逻辑。

`ApplicationContext` 自动检测实现 `BeanPostProcessor` 接口的配置元数据中定义的所有 bean

- `ApplicationContext` 将这些 bean 注册为后处理器，以便以后在 bean 创建时可以调用它们
- Bean 后处理器可以以与其他任何 Bean 相同的方式部署在容器中。

请注意，在配置类上使用 `@Bean` 工厂方法声明 `BeanPostProcessor` 时，工厂方法的返回类型应该是实现类本身或至少是 `org.springframework.beans.factory.config.BeanPostProcessor` 接口，从而清楚地表明该 bean 的后处理器性质。否则，`ApplicationContext` 无法在完全创建之前按类型自动检测它。由于 `BeanPostProcessor` 需要提早实例化以便应用于上下文中其他 bean 的初始化，因此这种提早类型检测至关重要。

## ① Programmatically registering BeanPostProcessor instances

推荐的 `BeanPostProcessor` 注册方法是通过 `ApplicationContext` 自动检测(如前所述)，但是您可以使用 `addBeanPostProcessor` 通过 `ConfigurableBeanFactory` 以编程方式注册它们。当您需要在注册之前评估条件逻辑，甚至需要跨层次结构的上下文复制 Bean 后处理器时，这将非常有用。但是请注意，以编程方式添加的 `BeanPostProcessor` 实例不遵守 `Ordered` 接口。在这里，注册的 Sequences 决定了执行的 Sequences。另请注意，以编程方式注册的 `BeanPostProcessor` 实例始终在通过自动检测注册的实例之前进行处理，而不考虑任何明确的 Sequences。

## ② BeanPostProcessor instances and AOP auto-proxying

实现 `BeanPostProcessor` 接口的类是特殊的，并且容器对它们的处理方式有所不同。它们直接引用的所有 `BeanPostProcessor` 实例和 Bean 在启动时都会实例化，作为 `ApplicationContext` 特殊启动阶段的一部分。接下来，以排序方式注册所有 `BeanPostProcessor` 实例，并将其应用于容器中的所有其他 bean。由于 AOP 自动代理本

身是由 `BeanPostProcessor` 实现的，因此 `BeanPostProcessor` 实例或它们直接引用的 bean 都不适合进行自动代理，因此，没有编织的方面。

对于任何此类 bean，您应该看到一条参考日志消息：`Bean someBean is not eligible for getting processed by all BeanPostProcessor interfaces (for example: not eligible for auto-proxying)`。

如果您通过自动装配或 `@Resource` 将 Bean 连接到 `BeanPostProcessor`（可能会退回到自动装配），则 Spring 在搜索类型匹配的依赖项候选对象时可能会访问意外的 bean，因此使它们不符合自动代理或其他类型的条件。 Bean 后处理。例如，如果您有一个用 `@Resource` Comments 的依赖项，其中字段或设置器名称不直接与 bean 的声明名称相对应，并且不使用 name 属性，那么 Spring 将访问其他 bean 以按类型匹配它们。

以下示例显示了如何在 `ApplicationContext` 中写入，注册和使用 `BeanPostProcessor` 实例。

## 示例：Hello World，BeanPostProcessor 风格

第一个示例说明了基本用法。该示例显示了一个自定义 `BeanPostProcessor` 实现，该实现调用容器创建的每个 bean 的 `toString()` 方法，并将结果字符串打印到系统控制台。

以下清单显示了自定义 `BeanPostProcessor` 实现类的定义：

```
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;
    }
}
```

以下 `beans` 元素使用 `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
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/lang
        http://www.springframework.org/schema/lang/spring-lang.xsd">

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

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

</beans>
```

注意 `InstantiationTracingBeanPostProcessor` 是如何定义的。它甚至没有名称，并且因为它是一个 Bean，所以可以像注入其他任何 Bean 一样对其进行依赖注入。（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](#) 的一章中有详细介绍。）

以下 Java 应用程序运行上述代码和配置：

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

public final class Boot {

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

前面的应用程序的输出类似于以下内容：

```
Bean 'messenger' created : org.springframework.scripting.groovy.GroovyMesse[emailprotected]
```

## 示例：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。一个示例是 Spring 的“实现，该实现随 Spring 发行版一起提供，并确保使用(任意)注解标记的 bean 上的 JavaBean 属性实际上(配置为)依赖注入了一个值。

### 1.8.2. 使用 BeanFactoryPostProcessor 自定义配置元数据

我们要看的下一个扩展点是

`org.springframework.beans.factory.config.BeanFactoryPostProcessor`。此接口的语义与 `BeanPostProcessor` 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。

您可以配置多个 `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 Sequences。但是，仅当 `BeanFactoryPostProcessor` 实现 `Ordered` 接口时才能设置此属性。如果您编写自己的 `BeanFactoryPostProcessor`，则也应该考虑实现 `Ordered` 接口。有关更多详细信息，请参见 [BeanFactoryPostProcessor](#) 和 [Ordered](#) 接口的 javadoc。

#### iNote

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` (在[使用 BeanPostProcessor 自定义 Bean](#)前面进行了介绍)。尽管在

技术上可以在 `BeanFactoryPostProcessor` 内使用 bean 实例(例如，通过使用

`BeanFactory.getBean()`)，但这样做会导致 bean 实例化过早，从而违反了标准容器的生

命周期。这可能会导致负面影响，例如绕过 bean 后处理。

此外，每个容器都限制了 `BeanFactoryPostProcessor` 个实例的作用域。仅在使用容器层

次结构时才有意义。如果在一个容器中定义 `BeanFactoryPostProcessor`，则仅将其应用于该容器中的 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，即使两个容器都属于同一层次结构也是如此。

在 `ApplicationContext` 内部声明 Bean 工厂后处理器时，将自动执行该处理器，以便将更改应用于定义容器的配置元数据。Spring 包含许多 `sched` 定义的 bean 工厂后处理器，例如 `PropertyOverrideConfigurer` 和 `PropertyPlaceholderConfigurer`。您还可以使用自定义 `BeanFactoryPostProcessor`，例如，注册自定义属性编辑器。

`ApplicationContext` 自动检测实现 `BeanFactoryPostProcessor` 接口的部署到其中的所有 bean。它在适当的时候将这些 bean 用作 bean 工厂的后处理器。您可以像部署其他任何 bean 一样部署这些后处理器 bean。

### iNote

与 `BeanPostProcessor` 一样，您通常不希望将 `BeanFactoryPostProcessor` 配置为延迟初始化。如果没有其他 bean 引用 `Bean(Factory)PostProcessor`，则该后处理器将完全不会被实例化。因此，将其标记为延迟初始化将被忽略，即使您在 `<beans />` 元素的声明中将 `default-lazy-init` 属性设置为 `true`，也将急切地实例化 `Bean(Factory)PostProcessor`。

## 示例：类名替换 `PropertyPlaceholderConfigurer`

您可以使用标准 Java `Properties` 格式，使用 `PropertyPlaceholderConfigurer` 来从单独文件中的 bean 定义外部化属性值。这样做使部署应用程序的人员可以自定义特定于环境的属性，例如数据库 URL 和密码，而不会为修改容器的一个或多个主要 XML 定义文件带来复杂性或风险。

考虑以下基于 XML 的配置元数据片段，其中定义了带有占位符值的 `DataSource`：

```
<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>
```

该示例显示了从外部 `Properties` 文件配置的属性。在运行时，将

`PropertyPlaceholderConfigurer` 应用于替换数据源的某些属性的元数据。将要替换的值指定为 `${property-name}` 形式的占位符，该形式遵循 Ant 和 log4j 和 JSP EL 样式。

实际值来自标准 Java `Properties` 格式的另一个文件：

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

因此，在运行时将 `${jdbc.username}` 字符串替换为值“sa”，并且其他与属性文件中的键匹配的占位符也是如此。`PropertyPlaceholderConfigurer` 检查 bean 定义的大多数属性和属性中的占位符。此外，您可以自定义占位符前缀和后缀。

使用 Spring 2.5 中引入的 `context` 名称空间，您可以使用专用配置元素配置属性占位符。您可以在 `location` 属性中以逗号分隔列表的形式提供一个或多个位置，如以下示例所示：

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

`PropertyPlaceholderConfigurer` 不仅在您指定的 `Properties` 文件中查找属性。默认情况下，如果无法在指定的属性文件中找到属性，则还会检查 Java `System` 属性。您可以通过使用以下三个受支持的整数值之一设置配置程序的 `systemPropertiesMode` 属性来自定义此行为：

- `never` (0): 从不检查系统属性。
- `fallback` (1): 检查系统属性是否在指定的属性文件中不可解析。这是默认值。
- `override` (2): 在尝试指定的属性文件之前, 请先检查系统属性。这使系统属性可以覆盖任何其他属性源。

有关更多信息, 请参见[PropertyPlaceholderConfigurer](#) javadoc。

### Tip

您可以使用 `PropertyPlaceholderConfigurer` 来替换类名, 这在您必须在运行时选择特定的实现类时有时很有用。以下示例显示了如何执行此操作:

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

如果无法在运行时将类解析为有效的类, 则将要创建的 bean 的解析将失败, 即在非延迟初始化 bean 的 `ApplicationContext` 的 `preInstantiateSingletons()` 阶段。

## 示例 : PropertyOverrideConfigurer

`PropertyOverrideConfigurer` 是另一个 Bean 工厂后处理程序, 类似于 `PropertyPlaceholderConfigurer`, 但是与后者不同, 原始定义对于 Bean 属性可以具有默认值或完全没有值。如果覆盖的 `Properties` 文件没有某个 bean 属性的条目, 则使用默认的上下文定义。

注意, bean 定义不知道会被覆盖, 因此从 XML 定义文件中不能立即看出正在使用覆盖配置器。如

果有多个 `PropertyOverrideConfigurer` 实例为同一个 bean 属性定义了不同的值，则由于覆盖机制，最后一个实例将获胜。

属性文件配置行采用以下格式：

```
beanName.property=value
```

下面的清单显示了格式的示例：

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

此示例文件可与包含定义为 `dataSource` 且具有 `driver` 和 `url` 属性的 Bean 的容器定义一起使用。

只要路径的每个组成部分(最终属性被覆盖)之外的所有组成部分都已经为非空(可能是由构造函数初始化)，则也支持复合属性名。在以下示例中，将 `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
```

### ① Note

指定的替代值始终是 Literals 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 Spring 2.5 中引入的 `context` 名称空间，可以使用专用配置元素配置属性覆盖，如以下示例所示：

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

## 1.8.3. 使用 FactoryBean 自定义实例化逻辑

您可以为本身就是工厂的对象实现 `org.springframework.beans.factory.FactoryBean` 接口。

`FactoryBean` 接口是可插入 Spring IoC 容器的实例化逻辑的点。如果您有复杂的初始化代码，而不是(可能)冗长的 XML，可以用 Java 更好地表达，则可以创建自己的 `FactoryBean`，在该类中编写复杂的初始化，然后将自定义 `FactoryBean` 插入容器。

`FactoryBean` 界面提供了三种方法：

- `Object getObject()`：返回此工厂创建的对象的实例。实例可以共享，具体取决于该工厂是否返回单例或原型。
- `boolean isSingleton()`：如果此 `FactoryBean` 返回单例，则返回 `true`，否则返回 `false`。
- `Class getObjectType()`：返回由 `getObject()` 方法或 `null` 返回的对象类型(如果事先未知)。

`FactoryBean` 概念和界面在 Spring Framework 中的许多地方都使用过。Spring 本身附带了 `FactoryBean` 接口的 50 多种实现。

当您需要向容器请求一个实际的 `FactoryBean` 实例本身而不是它生成的 bean 时，请在调用 `ApplicationContext` 的 `getBean()` 方法时在 bean 的 `id` 前面加上一个&符号(`&`)。因此，对于给定的 `id myBean` 的 `FactoryBean`，在容器上调用 `getBean("myBean")` 返回 `FactoryBean` 的乘积，而调用 `getBean("&myBean")` 则返回 `FactoryBean` 实例本身。

## 1.9. 基于 Comments 的容器配置

Comments 在配置 Spring 方面比 XML 更好吗？

基于 Comments 的配置的引入提出了一个问题，即这种方法是否比 XML“更好”。简短的答案是“取决于情况”。长话短说，每种方法都有其优缺点，通常，由开发人员决定哪种策略更适合他们。由于定义方式的不同，Comments 在声明中提供了很多上下文，从而使配置更短，更简洁。但是，XML 擅长连接组件而不接触其源代码或重新编译它们。一些开发人员更喜欢将布线放置在靠近源

的位置，而另一些开发人员则认为带 Comments 的类不再是 POJO，而且，该配置变得分散且难以控制。

无论选择如何，Spring 都可以容纳两种样式，甚至可以将它们混合在一起。值得指出的是，通过其 [JavaConfig](#) 选项，Spring 允许以非侵入方式使用 Comments，而无需接触目标组件的源代码，并且就工具而言，[Spring 工具套件](#) 支持所有配置样式。

基于 Comments 的配置提供了 XML 设置的替代方法，该配置依赖字节码元数据来连接组件，而不是尖括号声明。通过使用相关类，方法或字段声明上的 Comments，开发人员无需使用 XML 来描述 bean 的连接，而是将配置移入组件类本身。如[示例：RequiredAnnotationBeanPostProcessor](#) 中所述，结合使用 [BeanPostProcessor](#) 和 Comments 是扩展 Spring IoC 容器的常用方法。例如，Spring 2.0 引入了使用[@Required](#)Comments 强制执行必需属性的可能性。Spring 2.5 使遵循相同的通用方法来驱动 Spring 的依赖注入成为可能。本质上，[@Autowired](#) 注解提供了与 [Autowiring Collaborators](#) 中描述的功能相同的功能，但具有更细粒度的控制和更广泛的适用性。Spring 2.5 还添加了对 JSR-250 注解的支持，例如 [@PostConstruct](#) 和 [@PreDestroy](#)。Spring 3.0 添加了对 [javax.inject](#) 软件包(例如 [@Inject](#) 和 [@Named](#))中包含的 JSR-330(Java 依赖性注入)Comments 的支持。有关这些 Comments 的详细信息，请参见[relevant section](#)。

### iNote

Comments 注入在 XML 注入之前执行。因此，XML 配置将覆盖通过两种方法连接的属性的 Comments。

与往常一样，您可以将它们注册为单独的 bean 定义，但是也可以通过在基于 XML 的 Spring 配置中包含以下标记来隐式注册它们(请注意包含 [context](#) 名称空间)：

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

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

<context:annotation-config/>

</beans>
```

(隐式注册的后处理器包括[AutowiredAnnotationBeanPostProcessor](#),  
[CommonAnnotationBeanPostProcessor](#), [PersistenceAnnotationBeanPostProcessor](#)和上述  
[RequiredAnnotationBeanPostProcessor](#)。)

#### Note

`<context:annotation-config/>` 仅在定义它的相同应用程序上下文中查找 bean 上的 Comments。这意味着，如果您将 `<context:annotation-config/>` 放在 `WebApplicationContext` 中而不是 `DispatcherServlet`，则它仅检查控制器中的 `@Autowired` bean，而不检查服务。有关更多信息，请参见[The DispatcherServlet](#)。

### 1.9.1. @Required

`@Required` Comments 适用于 bean 属性设置器方法，如以下示例所示：

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

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

    // ...
}
```

此 Comments 指示必须在配置时通过 bean 定义中的显式属性值或通过自动装配来填充受影响的 bean 属性。如果受影响的 bean 属性尚未填充，则容器将引发异常。这允许急切和显式的故障，避免以后再出现 `NullPointerException` 个实例等。我们仍然建议您将 `assert` 放入 bean 类本身中（例如，放入 `init` 方法中）。这样做会强制执行那些必需的引用和值，即使您在容器外部使用该类也

是如此。

## 1.9.2. 使用@Autowired

### iNote

在本节中包含的示例中，可以使用 JSR 330 的 `@Inject` Comments 代替 Spring 的 `@Autowired` Comments。有关更多详细信息，请参见[here](#)。

您可以将 `@Autowired` Comments 应用于构造函数，如以下示例所示：

```
public class MovieRecommender {  
  
    private final CustomerPreferenceDao customerPreferenceDao;  
  
    @Autowired  
    public MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {  
        this.customerPreferenceDao = customerPreferenceDao;  
    }  
  
    // ...  
}
```

### iNote

从 Spring Framework 4.3 开始，如果目标 bean 仅定义一个构造函数作为开始，则不再需要在此类构造函数上使用 `@Autowired` Comments。但是，如果有几个构造函数可用，则必须至少 `Comments` 一个，以告诉容器使用哪个构造函数。

您还可以将 `@Autowired` Comments 应用于“传统” setter 方法，如以下示例所示：

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

```
    // ...
}
```

您还可以将 `Comments` 应用于具有任意名称和多个参数的方法，如以下示例所示：

```
public class MovieRecommender {

    private MovieCatalog movieCatalog;

    private CustomerPreferenceDao customerPreferenceDao;

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

    // ...
}
```

您也可以将 `@Autowired` 应用于字段，甚至将其与构造函数混合使用，如以下示例所示：

```
public class MovieRecommender {

    private final CustomerPreferenceDao customerPreferenceDao;

    @Autowired
    private MovieCatalog movieCatalog;

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

    // ...
}
```

### Tip

确保目标组件(例如 `MovieCatalog` 或 `CustomerPreferenceDao`)由用于 `@Autowired`

`Comments` 的注入点的类型一致地声明。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 `Classpath` 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，您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，请考虑在工厂方法中声明最具体的返回类型（至少根据引用您的 bean 的注入点的要求具体声明）。

您还可以通过将 Comments 添加到需要该类型数组的字段或方法中，来提供

ApplicationContext 中所有特定类型的 bean，如以下示例所示：

```
public class MovieRecommender {  
  
    @Autowired  
    private MovieCatalog[] movieCatalogs;  
  
    // ...  
}
```

如以下示例所示，这同样适用于类型化集合：

```
public class MovieRecommender {  
  
    private Set<MovieCatalog> movieCatalogs;  
  
    @Autowired  
    public void setMovieCatalogs(Set<MovieCatalog> movieCatalogs) {  
        this.movieCatalogs = movieCatalogs;  
    }  
  
    // ...  
}
```

### Tip

如果希望数组或列表中的项按特定 Sequences 排序，则目标 bean 可以实现

org.springframework.core.Ordered 接口或使用 @Order 或标准 @Priority

Comments。否则，它们的 Sequences 将遵循容器中相应目标 bean 定义的注册 Sequences。

您可以在目标类级别和 @Bean 方法上声明 @Order 注解，可能通过单个 bean 定义来声明（如果使用同一 bean 类的多个定义）。@Order 值可能会影响注入点的优先级，但是要注意，它们不会影响单例启动 Sequences，这是由依赖关系和 @DependsOn 声明确定的正交关注

点。

请注意，标准 `javax.annotation.Priority` `Comments` 在 `@Bean` 级别不可用，因为无法在方法上声明它。对于每种类型，可以通过 `@Order` 值与 `@Primary` 组合在单个 bean 上对其语义进行建模。

只要预期的密钥类型为 `String`，即使是键入的 `Map` 实例也可以自动装配。 `Map` 值包含所有预期类型的 bean，并且键包含相应的 bean 名称，如以下示例所示：

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

默认情况下，只要有零个候选 bean 可用，自动装配就会失败。默认行为是将带 `Comments` 的方法，构造函数和字段视为指示所需的依赖项。在下面的示例中，您可以按照说明更改此行为：

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

### iNote

每个类仅可以将一个带 `Comments` 的构造函数标记为必需，但可以对多个非必需的构造函数进行 `Comments`。在这种情况下，每个候选对象都将被考虑在内，并且 Spring 使用最贪婪的构造函数，其依赖关系可以得到满足-即具有最多参数的构造函数。

建议在 `@Required` Comments 上使用 `@Autowired` 的必需属性。 `required` 属性表示自动装配不需要该属性。如果无法自动装配该属性，则将其忽略。另一方面，`@Required` 更强大，因为它可以强制执行通过容器支持的任何方式设置的属性。如果未注入任何值，则会引发相应的异常。

另外，您可以通过 Java 8 的 `java.util.Optional` 表示特定依赖项的非必需性质，如以下示例所示：

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

从 Spring Framework 5.0 开始，您还可以使用 `@Nullable` Comments(在任何包中是任何一种形式，例如，来自 JSR-305 的 `javax.annotation.Nullable`)：

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

您也可以将 `@Autowired` 用于众所周知的可解决依赖项：`BeanFactory`，`ApplicationContext`，`Environment`，`ResourceLoader`，`ApplicationEventPublisher` 和 `MessageSource`。这些接口及其扩展接口(例如 `ConfigurableApplicationContext` 或 `ResourcePatternResolver`)将自动解析，而无需进行特殊设置。下面的示例自动连接 `ApplicationContext` 对象：

```
public class MovieRecommender {
```

```
@Autowired  
private ApplicationContext context;  
  
public MovieRecommender() {  
}  
  
// ...  
}
```

### iNote

`@Autowired` , `@Inject` , `@Resource` 和 `@Value` 注解由 Spring `BeanPostProcessor` 实现处理。这意味着您不能在自己的 `BeanPostProcessor` 或 `BeanFactoryPostProcessor` 类型(如果有)中应用这些 Comments。必须使用 XML 或 Spring `@Bean` 方法显式“连接”这些类型。

### 1.9.3. 使用`@Primary` 微调基于 Comments 的自动装配

由于按类型自动布线可能会导致多个候选对象，因此通常有必要对选择过程进行更多控制。实现此目的的一种方法是使用 Spring 的 `@Primary` Comments。`@Primary` 表示当多个 bean 可以自动连接到单值依赖项的候选对象时，应优先考虑特定的 bean。如果候选中恰好存在一个主 bean，则它将成为自动装配的值。

考虑以下将 `firstMovieCatalog` 定义为主 `MovieCatalog` 的配置：

```
@Configuration  
public class MovieConfiguration {  
  
    @Bean  
    @Primary  
    public MovieCatalog firstMovieCatalog() { ... }  
  
    @Bean  
    public MovieCatalog secondMovieCatalog() { ... }  
  
    // ...  
}
```

使用前面的配置，下面的 `MovieRecommender` 与 `firstMovieCatalog` 自动连接：

```
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  
                           http://www.springframework.org/schema/beans/spring-beans.xsd  
                           http://www.springframework.org/schema/context  
                           http://www.springframework.org/schema/context/spring-context.xsd">  
  
    <context:annotation-config/>  
  
    <bean class="example.SimpleMovieCatalog" primary="true">  
        <!-- inject any dependencies required by this bean -->  
    </bean>  
  
    <bean class="example.SimpleMovieCatalog">  
        <!-- inject any dependencies required by this bean -->  
    </bean>  
  
    <bean id="movieRecommender" class="example.MovieRecommender"/>  
  
</beans>
```

#### 1.9.4. 使用限定符对基于 Comments 的自动装配进行微调

当可以确定一个主要候选对象时，`@Primary` 是在几种情况下按类型使用自动装配的有效方法。当您需要对选择过程进行更多控制时，可以使用 Spring 的 `@Qualifier` 注解。您可以将限定符值与特定的参数相关联，从而缩小类型匹配的范围，以便为每个参数选择特定的 bean。在最简单的情况下，这可以是简单的描述性值，如以下示例所示：

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

您还可以在各个构造函数参数或方法参数上指定 `@Qualifier` 注解，如以下示例所示：

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

以下示例显示了相应的 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  
                           http://www.springframework.org/schema/beans/spring-beans.xsd  
                           http://www.springframework.org/schema/context  
                           http://www.springframework.org/schema/context/spring-context.xsd">  
  
<context:annotation-config/>  
  
<bean class="example.SimpleMovieCatalog">  
    <qualifier value="main"/> (1)  
  
    <!-- inject any dependencies required by this bean -->  
</bean>  
  
<bean class="example.SimpleMovieCatalog">  
    <qualifier value="action"/> (2)  
  
    <!-- inject any dependencies required by this bean -->  
</bean>  
  
<bean id="movieRecommender" class="example.MovieRecommender"/>  
  
</beans>
```

- (1) 具有 `main` 限定符值的 Bean 与限定有相同值的构造函数自变量连接。
- (2) 具有 `action` 限定符值的 Bean 与限定有相同值的构造函数参数连接。

对于后备匹配，bean 名称被视为默认的限定符值。因此，您可以用 `id` 或 `main` 定义 bean，而不

是嵌套的限定符元素，从而得到相同的匹配结果。但是，尽管您可以使用此约定按名称引用特定的 bean，但 `@Autowired` 基本上是关于带有可选语义限定符的类型驱动的注入。这意味着，即使带有 Bean 名称后备的限定符值，在类型匹配集中也始终具有狭窄的语义。它们没有在语义上表示对唯一 bean `id` 的引用。好的限定符值为 `main` 或 `EMEA` 或 `persistent`，它们表示特定组件的特性，它们独立于 Bean `id`，如果是匿名 Bean 定义(例如上例中的定义)，则可以自动生成这些组件。

限定词也适用于类型化的集合，如前面所述(例如，`Set<MovieCatalog>`)。在这种情况下，根据声明的限定符，将所有匹配的 bean 作为集合注入。这意味着限定词不必是唯一的。相反，它们构成了过滤标准。例如，您可以使用相同的限定符值“action”定义多个 `MovieCatalog` bean，所有这些都注入到以 `@Qualifier("action")` `Comments` 的 `Set<MovieCatalog>` 中。

### Tip

在类型匹配的候选对象中，让限定符值针对目标 bean 名称进行选择，在注入点不需要 `@Qualifier` `Comments`。如果没有其他解析度指示符(例如限定词或主标记)，则对于非唯一依赖性情况，Spring 将注入点名称(即字段名称或参数名称)与目标 Bean 名称进行匹配，然后选择同名候选人(如果有)。

就是说，如果您打算按名称表示 `Comments` 驱动的注入，则不要主要使用 `@Autowired`，即使它能够在类型匹配的候选对象中按 bean 名称进行选择。而是使用 JSR-250 `@Resource` `Comments`，该 `Comments` 的语义定义是通过其唯一名称来标识特定目标组件，而声明的类型与匹配过程无关。`@Autowired` 具有不同的语义：按类型选择候选 Bean 之后，仅在那些类型选择的候选中考虑指定的 `String` 限定符值(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 `Map` 或数组类型的 bean，使用 `@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 版本开始，只要元素类

型信息保留在 `@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，您可以使用限定符值在同类型的集合中进行选择，如上一段所述。

从 4.3 开始，`@Autowired` 还考虑了自我引用以进行注入(即，引用回当前注入的 Bean)。请注意，自我注入是一个后备。对其他组件的常规依赖始终优先。从这个意义上说，自我推荐不参与常规的候选人选择，因此尤其是绝不是主要的。相反，它们总是以最低优先级结束。实际上，您应该仅将自我引用用作最后的手段(例如，通过 bean 的事务代理在同一实例上调用其他方法)。考虑在这种情况下将受影响的方法分解为单独的委托 Bean。或者，您可以使用 `@Resource`，它可以通过其唯一名称获取返回到当前 bean 的代理。

`@Autowired` 适用于字段，构造函数和多参数方法，从而允许在参数级别缩小限定符 `Comments` 的范围。相反，仅具有单个参数的字段和 bean 属性设置器方法支持 `@Resource`。因此，如果注入目标是构造函数或多参数方法，则应坚持使用限定符。

您可以创建自己的自定义限定符 `Comments`。为此，请定义一个 `Comments` 并在定义中提供 `@Qualifier` `Comments`，如以下示例所示：

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

然后，您可以在自动连接的字段和参数上提供自定义限定符，如以下示例所示：

```
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，您可以提供有关候选 bean 定义的信息。您可以将 `<qualifier/>` 标记添加为 `<bean/>` 标记的子元素，然后指定 `type` 和 `value` 以匹配您的自定义限定符 Comments。该类型与 Comments 的完全限定的类名匹配。另外，为方便起见，如果不存在名称冲突的风险，则可以使用简短的类名。下面的示例演示了两种方法：

```
<?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  
                           http://www.springframework.org/schema/beans/spring-beans.xsd  
                           http://www.springframework.org/schema/context  
                           http://www.springframework.org/schema/context/spring-context.xsd">  
  
<context:annotation-config/>  
  
<bean class="example.SimpleMovieCatalog">  
    <qualifier type="Genre" value="Action"/>  
    <!-- inject any dependencies required by this bean -->  
</bean>  
  
<bean class="example.SimpleMovieCatalog">  
    <qualifier type="example.Genre" value="Comedy"/>  
    <!-- inject any dependencies required by this bean -->  
</bean>  
  
<bean id="movieRecommender" class="example.MovieRecommender"/>  
  
</beans>
```

在 [Classpath 扫描和托管组件](#) 中，您可以看到基于 Comments 的替代方法，以 XML 形式提供限定符元数据。具体来说，请参见 [提供带 Comments 的限定符元数据](#)。

在某些情况下，使用没有值的 Comments 就足够了。当 Comments 用于更一般的用途并且可以应用于几种不同类型的依赖项时，这将很有用。例如，您可以提供一个脱机目录，当没有 Internet 连接可用时，可以对其进行搜索。首先，定义简单的 Comments，如以下示例所示：

```
@Target({ElementType.FIELD, ElementType.PARAMETER})  
@Retention(RetentionPolicy.RUNTIME)
```

```
@Qualifier  
public @interface Offline {  
}
```

然后将 **Comments** 添加到要自动装配的字段或属性，如以下示例所示：

```
public class MovieRecommender {  
  
    @Autowired  
    @Offline (1)  
    private MovieCatalog offlineCatalog;  
  
    // ...  
}
```

- (1) 此行添加了 `@Offline` **Comments**。

现在，**bean** 定义只需要一个限定符 `type`，如以下示例所示：

```
<bean class="example.SimpleMovieCatalog">  
    <qualifier type="Offline"/> (1)  
    <!-- inject any dependencies required by this bean -->  
</bean>
```

- (1) 此元素指定限定符。

您还可以定义自定义限定符注解，除了简单的 `value` 属性之外，还可以接受命名属性。如果随后在要自动装配的字段或参数上指定了多个属性值，则 **bean** 定义必须与所有此类属性值匹配才能被视为自动装配候选。例如，请考虑以下 **Comments** 定义：

```
@Target({ElementType.FIELD, ElementType.PARAMETER})  
@Retention(RetentionPolicy.RUNTIME)  
@Qualifier  
public @interface MovieQualifier {  
  
    String genre();  
  
    Format format();  
}
```

在这种情况下，`Format` 是一个枚举，定义如下：

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

要自动装配的字段将用定制限定符进行 Comments，并包括两个属性 `genre` 和 `format` 的值，如以下示例所示：

```
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` 定义应包含匹配的限定符值。此示例还演示了可以使用 `bean` 元属性代替 `<qualifier/>` 元素。如果可用，则 `<qualifier/>` 元素及其属性优先，但是如果不存在此类限定符，则自动装配机制将退回到 `<meta/>` 标记内提供的值，如以下示例中的最后两个 `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  
                           http://www.springframework.org/schema/beans/spring-beans.xsd  
                           http://www.springframework.org/schema/context  
                           http://www.springframework.org/schema/context/spring-context.xsd">  
  
    <context:annotation-config/>  
  
    <bean class="example.SimpleMovieCatalog">  
        <qualifier type="MovieQualifier">  
            <attribute key="format" value="VHS"/>  
            <attribute key="genre" value="Action"/>  
        </qualifier>  
    </bean>
```

```

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

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

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

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

</beans>

```

## 1.9.5. 将泛型用作自动装配限定符

除了 `@Qualifier` 注解，您还可以将 Java 泛型类型用作资格的隐式形式。例如，假设您具有以下配置：

```

@Configuration
public class MyConfiguration {

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

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

```

假设前面的 bean 实现了通用接口(即 `Store<String>` 和 `Store<Integer>` )，则可以

`@Autowired` `Store` 接口，并且通用接口用作限定符，如以下示例所示：

```

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

```

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

自动连接列表，`Map` 实例和数组时，通用限定符也适用。以下示例将自动连接通用 `List`：

```
// 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 `BeanFactoryPostProcessor`，即使您没有使用 Spring 的

`@Qualifier` Comments 对您自己的自定义限定符 Comments 类型进行注册，您也可以使用它们。以下示例显示了如何使用 `CustomAutowireConfigurer`：

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

`AutowireCandidateResolver` 通过以下方式确定自动装配的候选对象：

- 每个 bean 定义的 `autowire-candidate` 值
- `<beans/>` 元素上可用的任何 `default-autowire-candidates` 模式
- `@Qualifier` Comments 和在 `CustomAutowireConfigurer` 中注册的所有自定义 Comments 的存在

当多个 bean 可以作为自动装配候选者时，确定“主要”的步骤如下：如果候选者中恰好有一个 bean 定义具有 `primary` 属性设置为 `true`，则将其选中。

## 1.9.7. 用@Resource 注入

Spring 还通过在字段或 bean 属性设置器方法上使用 JSR-250 `@Resource` Comments 来支持注入

。这是 Java EE 5 和 6 中的常见模式(例如，在 JSF 1.2 托管 Bean 或 JAX-WS 2.0 端点中)。Spring 也为 SpringManagement 的对象支持此模式。

`@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，如以下示例所示：

```
public class SimpleMovieLister {  
  
    private MovieFinder movieFinder;  
  
    @Resource(name="myMovieFinder") (1)  
    public void setMovieFinder(MovieFinder movieFinder) {  
        this.movieFinder = movieFinder;  
    }  
}
```

- (1) 此行注入 `@Resource`。

如果未明确指定名称，则默认名称是从字段名称或 `setter` 方法派生的。如果是字段，则采用字段名称。在使用 `setter` 方法的情况下，它采用 `bean` 属性名称。以下示例将名为 `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;  
    }  
}
```

### Note

Comments 提供的名称由 `CommonAnnotationBeanPostProcessor` 知道的

`ApplicationContext` 解析为 `bean` 名称。如果您显式配置 Spring 的

[SimpleJndiBeanFactory](#), 则可以通过 JNDI 解析名称。但是, 我们建议您依靠默认行为并使用 Spring 的 JNDI 查找功能来保留间接级别。

在未使用 `@Resource` 且未指定显式名称且与 `@Autowired` 类似的特殊情况下, `@Resource` 查找主类型匹配而不是特定的命名 bean, 并解析众所周知的可解决依赖项: `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, 然后回退到类型 `CustomerPreferenceDao` 的主类型匹配:

```
public class MovieRecommender {

    @Resource
    private CustomerPreferenceDao customerPreferenceDao;

    @Resource
    private ApplicationContext context; (1)

    public MovieRecommender() {
    }

    // ...
}
```

- (1) `context` 字段是根据已知的可解决依赖项类型 `ApplicationContext` 注入的。

## 1.9.8. 使用`@PostConstruct` 和`@PreDestroy`

`CommonAnnotationBeanPostProcessor` 不仅可以识别 `@Resource` Comments, 还可以识别 JSR-250 生命周期 Comments。在 Spring 2.5 中引入了对这些 Comments 的支持, 为[initialization callbacks](#)和[destruction callbacks](#)中描述的 Comments 提供了另一种选择。假设 `CommonAnnotationBeanPostProcessor` 已在 Spring `ApplicationContext` 中注册, 则在生命周期的同一点与相应的 Spring 生命周期接口方法或显式声明的回调方法一起调用带有这些注解之一的方法。在以下示例中, 缓存在初始化时预先填充, 并在销毁时清除:

```
public class CachingMovieLister {  
  
    @PostConstruct  
    public void populateMovieCache() {  
        // populates the movie cache upon initialization...  
    }  
  
    @PreDestroy  
    public void clearMovieCache() {  
        // clears the movie cache upon destruction...  
    }  
}
```

### iNote

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](#)。

## 1.10. Classpath 扫描和托管组件

本章中的大多数示例都使用 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 Comments 的容器配置](#))演示了如何通过源级 Comments 提供许多配置元数据。但是，即使在这些示例中，“基本” bean 定义也已在 XML 文件中明确定义，而 Comments 仅驱动依赖项注入。本节介绍了通过扫描 Classpath 来隐式检测候选组件的选项。候选组件是与过滤条件匹配的类，并在容器中注册了相应的 Bean 定义。这消除了使用 XML 进行 bean 注册的需要。相反，您可以使用 Comments(例如 `@Component`)，AspectJ 类型表达式或您自己的自定义过滤条件来选择哪些类已向容器注册了 bean 定义。

### iNote

从 Spring 3.0 开始，Spring JavaConfig 项目提供的许多功能是核心 Spring Framework 的一部分。这使您可以使用 Java 而不是使用传统的 XML 文件来定义 bean。请查看

`@Configuration`，`@Bean`，`@Import` 和 `@DependsOn` 注解，以获取有关如何使用这些新功能的示例。

### 1.10.1. `@Component` 和其他构造型 Comments

`@Repository` Comments 是满足存储库的角色或构造型(也称为数据访问对象或 DAO)的任何类的标记。如[Exception Translation](#)中所述，此标记的用途是自动翻译异常。

Spring 提供了进一步的构造型 Comments：`@Component`，`@Service` 和 `@Controller`。

`@Component` 是任何 SpringManagement 的组件的通用构造型。`@Repository`，`@Service` 和 `@Controller` 是 `@Component` 的特化，用于更具体的用例(分别在持久层，服务层和表示层中)。

因此，您可以使用 `@Component` Comments 组件类，但是通过使用 `@Repository`，`@Service` 或 `@Controller` Comments 组件类，则您的类更适合于通过工具进行处理或与方面相关联。例如，这些构造型 Comments 成为切入点的理想目标。`@Repository`，`@Service` 和 `@Controller` 在 Spring 框架的 Future 发行版中还可包含其他语义。因此，如果在服务层使用 `@Component` 或 `@Service` 之间进行选择，则 `@Service` 显然是更好的选择。同样，如前所述，`@Repository` 已被支持作为持久层中自动异常转换的标记。

## 1.10.2. 使用元 Comments 和组合 Comments

Spring 提供的许多 Comments 都可以在您自己的代码中用作元 Comments。元 Comments 是可以应用于另一个 Comments 的 Comments。例如，提到的[earlier](#)的 `@Service` Comments 使用 `@Component` 进行元 Comments，如以下示例所示：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component(1)
public @interface Service {

    // ...
}
```

- (1) `Component` 导致 `@Service` 的处理方式与 `@Component` 相同。

您还可以结合使用元 Comments 来创建“组合 Comments”。例如，Spring MVC 的 `@RestController` Comments 由 `@Controller` 和 `@ResponseBody` 组成。

此外，组合 Comments 可以选择从元 Comments 中重新声明属性，以允许自定义。当您只想公开元 Comments 属性的子集时，这可能特别有用。例如，Spring 的 `@SessionScope` Comments 将作用域名称硬编码为 `session`，但仍允许自定义 `proxyMode`。以下清单显示了 `SessionScope` 注解的定义：

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

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

}
```

然后，您可以使用 `@SessionScope` 而不用声明 `proxyMode`，如下所示：

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

您还可以覆盖 `proxyMode` 的值，如以下示例所示：

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

有关更多详细信息，请参见[SpringComments 编程模型](#) Wiki 页面。

### 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，并向 `ApplicationContext` 注册相应的 `BeanDefinition` 实例。

例如，以下两个类别有资格进行这种自动检测：

```
@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` 属性是两个类的公共父包。(或者，您可以指定一个逗号分隔，分号分隔或空格分隔的列表，其中包括每个类的父包。)

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

### iNote

为简洁起见，前面的示例可能使用了 Comments 的 `value` 属性(即

```
@ComponentScan("org.example") )。
```

以下替代方法使用 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:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context.xsd">

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

```

```
</beans>
```

## Tip

`<context:component-scan>` 的使用隐式启用 `<context:annotation-config>` 的功能。使用 `<context:component-scan>` 时通常不需要包含 `<context:annotation-config>` 元素。

## Note

扫描 Classpath 包需要在 Classpath 中存在相应的目录条目。使用 Ant 构建 JAR 时, 请确保未激活 JAR 任务的仅文件开关。另外, 在某些环境中(例如, JDK 1.7.0\_45 及更高版本上的独立应用程序(在清单中要求“受信任的库”设置)), 见

<http://stackoverflow.com/questions/19394570/java-jre-7u45-breaks-classloader-getresources>), 可能不会基于安全策略公开 Classpath 目录。

在 JDK 9 的模块路径(Jigsaw)上, Spring 的 Classpath 扫描通常可以按预期进行。但是, 请确保将组件类导出到 `module-info` Descriptors 中。如果您希望 Spring 调用类的非公共成员, 请确保它们已“打开”(即, 它们在 `module-info` Descriptors 中使用 `opens` 声明而不是 `exports` 声明)。

此外, 当您使用 component-scan 元素时, `AutowiredAnnotationBeanPostProcessor` 和 `CommonAnnotationBeanPostProcessor` 都隐式包含在内。这意味着两个组件将被自动检测并连接在一起, 而所有这些都不需要 XML 中提供任何 bean 配置元数据。

## Note

您可以通过包含 `Comments` 设置属性 `false` 来禁用

`AutowiredAnnotationBeanPostProcessor` 和 `CommonAnnotationBeanPostProcessor` 的注册。

#### 1.10.4. 使用过滤器自定义扫描

默认情况下，唯一检测到的候选组件是用 `@Component`，`@Repository`，`@Service`，`@Controller` `Comments` 的类或本身用 `@Component` `Comments` 的定制 `Comments`。但是，您可以通过应用自定义过滤器来修改和扩展此行为。将它们添加为 `@ComponentScan` 注解的 `includeFilters` 或 `excludeFilters` 参数(或 `component-scan` 元素的 `include-filter` 或 `exclude-filter` 子元素)。每个过滤器元素都需要 `type` 和 `expression` 属性。下表描述了过滤选项：

表 5. 过滤器类型

Filter Type	Example Expression	Description
annotation (default)	<code>org.example.SomeAnnotation</code>	在目标组件的类型级别上存在的 <code>Comments</code> 。
assignable	<code>org.example.SomeClass</code>	目标组件可分配给(扩展或实现)的类(或接口)。
aspectj	<code>org.example..*Service+</code>	目标组件要匹配的 AspectJ 类型表达式。
regex	<code>org\\.example\\Default.*</code>	要与目标组件类名称匹配的正则表达式。

Filter Type	Example Expression	Description
custom	<code>org.example.MyTypeFilter</code>	<code>org.springframework.core.type.TypeFilter</code> 接口的自定义实现。

以下示例显示了忽略所有 `@Repository` 注解并使用“存根”存储库的配置：

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

以下清单显示了等效的 XML：

```
<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>
```

### iNote

您还可以通过在 `Comments` 上设置 `useDefaultFilters=false` 或通过提供 `use-default-filters="false"` 作为 `<component-scan/>` 元素的属性来禁用默认过滤器。实际上，这将禁用对带有 `@Component`，`@Repository`，`@Service`，`@Controller` 或 `@Configuration` `Comments` 的类的自动检测。

## 1.10.5. 在组件中定义 Bean 元数据

Spring 组件还可以将 bean 定义元数据贡献给容器。您可以使用与 `@Configuration` 带

Comments 的类中定义 Bean 元数据相同的 `@Bean` Comments 来执行此操作。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
    }
}
```

上一类是 Spring 组件，该组件的 `doWork()` 方法中包含特定于应用程序的代码。但是，它也提供了具有工厂方法的 bean 定义，该工厂方法引用方法 `publicInstance()`。`@Bean` Comments 标识工厂方法和其他 bean 定义属性，例如通过 `@Qualifier` Comments 的限定符值。可以指定的其他方法级别 Comments 是 `@Scope`，`@Lazy` 和自定义限定符 Comments。

### Tip

除了用于组件初始化的角色外，还可以将 `@Lazy` Comments 放置在标有 `@Autowired` 或 `@Inject` 的注入点上。在这种情况下，它导致注入了惰性解析代理。

如前所述，支持自动连线的字段和方法，并自动支持 `@Bean` 方法的自动装配。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");
    }
}
```

```

// 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);
}
}

```

该示例将 `String` 方法参数 `country` 自动连接到另一个名为 `privateInstance` 的 bean 上 `age` 属性的值。Spring Expression Language 元素通过符号 `#{ <expression> }` 定义属性的值。对于 `@Value` Comments，表达式解析器已预先配置为在解析表达式文本时查找 bean 名称。

从 Spring Framework 4.3 开始，您还可以声明类型为 `InjectionPoint` (或其更具体的子类：  
`DependencyDescriptor`) 的工厂方法参数，以访问触发当前 bean 创建的请求注入点。注意，这仅适用于实际创建 bean 实例，而不适用于注入现有实例。因此，此功能对原型范围的 bean 最有意义。对于其他作用域，factory 方法仅在给定作用域中看到触发创建新 bean 实例的注入点(例如，触发创建惰性单例 bean 的依赖项)。在这种情况下，可以将提供的注入点元数据与语义一起使用。以下示例显示了如何使用 `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` 类中相应方法的处理方式不同。不同之处在于，CGLIB 并未增强 `@Component` 类来拦截方法和字段的调用。CGLIB 代理是调用 `@Configuration` 类中 `@Bean` 方法中的方法或字段中的字段的方法，用于创建 Bean 元数据引用以协作对象。此类方法不是用普通的 Java 语义调用的，而是通过容器进行的，以提供通常的生命周期 Management 和 Spring bean 的代理，即使通过编程调用 `@Bean` 方法引用其他 bean 时也是如此。相反，在普通 `@Component` 类内的 `@Bean` 方法中调用方法或字段具有标准 Java 语义，而无需特殊的 CGLIB 处理或其他约束。

### 1 Note

您可以将 `@Bean` 方法声明为 `static`，从而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。在定义后处理器 Bean(例如，类型 `BeanFactoryPostProcessor` 或 `BeanPostProcessor`)时，这特别有意义，因为此类 Bean 在容器生命周期的早期进行了初始化，并且应避免在那时触发配置的其他部分。

由于技术限制，对静态 `@Bean` 方法的调用永远不会被容器拦截，即使在 `@Configuration` 类中也是如此(如本节前面所述)，由于技术限制：CGLIB 子类只能覆盖非静态方法。因此，直接调用另一个 `@Bean` 方法具有标准的 Java 语义，从而导致直接从工厂方法本身直接返回一个独立的实例。

`@Bean` 方法的 Java 语言可见性不会对 Spring 容器中的最终 bean 定义产生直接影响。您可以随意声明自己的工厂方法，以适合非 `@Configuration` 类，也可以随处声明静态方法。但是，`@Configuration` 类中的常规 `@Bean` 方法必须是可重写的一即，不得将它们声明为 `private` 或 `final`。

还可以在给定组件或配置类的 Base Class 上以及在由组件或配置类实现的接口中声明的 Java 8 默认方法上找到 `@Bean` 方法。这为组合复杂的配置安排提供了很大的灵 Active，从 Spring 4.2 开始，通过 Java 8 默认方法甚至可以实现多重继承。

最后，单个类可以为同一个 bean 保留多个 `@Bean` 方法，这取决于在运行时可用的依赖关系，从而可以使用多个工厂方法。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：在构造时将选择具有最大可满足依赖关系数量的变量，类似于容器在多个 `@Autowired` 构造函数之间进行选择的方式。

## 1.10.6. 命名自动检测的组件

在扫描过程中自动检测到某个组件时，其 bean 名称由该扫描器已知的 `BeanNameGenerator` 策略生成。默认情况下，任何包含名称 `value` 的 Spring 构造型 `Comments`(`@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 `Comments` 不包含名称 `value` 或任何其他检测到的组件(例如，由自定义过滤器发现的组件)，则缺省 bean 名称生成器将返回不使用大写字母的非限定类名称。例如，如果检测到以下组件类，则名称将为 `myMovieLister` 和 `movieFinderImpl`：

```
@Service("myMovieLister")
public class SimpleMovieLister {
    // ...
}
```

```
@Repository
public class MovieFinderImpl implements MovieFinder {
    // ...
}
```

### • Note

如果不依赖默认的 Bean 命名策略，则可以提供自定义 Bean 命名策略。首先，实现 `BeanNameGenerator` 接口，并确保包括默认的 no-arg 构造函数。然后，在配置扫描器时提供完全限定的类名，如以下示例 `Comments` 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>
```

通常，请考虑在其他组件可能对其进行显式引用时，使用注解指定名称。另一方面，只要容器负责接线，自动生成的名称就足够了。

### 1.10.7. 提供自动检测组件的范围

通常，与 SpringManagement 的组件一样，自动检测到的组件的默认且最常见的范围是 `singleton`。但是，有时您需要由 `@Scope` Comments 指定的其他范围。您可以在注解中提供范围的名称，如以下示例所示：

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

#### iNote

`@Scope` Comments 仅在具体的 bean 类(对于带 Comments 的组件)或工厂方法(对于 `@Bean` 方法)上进行内省。与 XML bean 定义相反，没有 bean 定义继承的概念，并且在类级别的继承层次结构与元数据目的无关。

有关特定于 Web 的作用域的详细信息，例如 Spring 上下文中的“请求”或“会话”，请参见[请求，会话，应用程序和 WebSocket 范围](#)。与这些范围的预构建 Comments 一样，您也可以使用 Spring 的元 Comments 方法来编写自己的作用域 Comments：例如，使用 `@Scope("prototype")` 进行元 Comments 的自定义 Comments，也可能会声明自定义范围代理模式。

## iNote

要提供用于范围解析的自定义策略，而不是依赖于基于 `Comments` 的方法，您可以实现 `ScopeMetadataResolver` 接口。确保包括默认的无参数构造函数。然后，可以在配置扫描程序时提供完全限定的类名，如以下 `Comments` 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.MySc
```

使用某些非单作用域时，可能有必要为作用域对象生成代理。推理论在 [范围 bean 作为依赖项](#) 中描述。为此，在 `component-scan` 元素上可以使用 `scoped-proxy` 属性。三个可能的值是：`no`，

`interfaces` 和 `targetClass`。例如，以下配置产生标准的 JDK 动态代理：

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

```
<beans>  
    <context:component-scan base-package="org.example" scoped-proxy="interfaces" />  
</beans>
```

### 1.10.8. 提供带 `Comments` 的限定符元数据

使用 [限定符对基于 `Comments` 的自动装配进行微调](#) 中讨论了 `@Qualifier` `Comments`。该部分中的示例演示了 `@Qualifier` 注解和自定义限定符注解的使用，以在解析自动装配候选时提供细粒度的控制。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 `qualifier` 或 `meta` 子元素，在候选 `bean` 定义上提供了限定符元数据。当依靠 Classpath 扫描

来自动检测组件时，可以在候选类上为限定符元数据提供类型级别的 Comments。下面的三个示例演示了此技术：

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

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

```
@Component
@Offline
public class CachingMovieCatalog implements MovieCatalog {
    // ...
}
```

### iNote

与大多数基于 Comments 的替代方法一样，请记住，Comments 元数据绑定到类定义本身，而 XML 的使用允许相同类型的多个 bean 提供其限定符元数据的变体，因为该元数据是按 instance 而不是按类。

## 1.10.9. 生成候选组件的索引

尽管 Classpath 扫描非常快，但可以通过在编译时创建候选静态列表来提高大型应用程序的启动性能。在这种模式下，应用程序的所有模块都必须使用此机制，因为当 `ApplicationContext` 检测到这样的索引时，它将自动使用它而不是扫描 Classpath。

要生成索引，请向每个包含组件的模块添加附加依赖关系，这些组件是组件扫描指令的目标。以下示例显示了如何使用 Maven 进行操作：

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

```
<version>5.1.3.RELEASE</version>
<optional>true</optional>
</dependency>
</dependencies>
```

以下示例显示了如何使用 Gradle 进行操作：

```
dependencies {
    compileOnly("org.springframework:spring-context-indexer:5.1.3.RELEASE")
}
```

该过程将生成 jar 文件中包含的 `META-INF/spring.components` 文件。

### iNote

在 IDE 中使用此模式时，必须将 `spring-context-indexer` 注册为 Comments 处理器，以确保在更新候选组件时索引是最新的。

### Tip

当在 Classpath 上找到 `META-INF/spring.components` 时，索引将自动启用。如果某些库(或用例)的索引部分可用，但无法为整个应用程序构建，则可以通过将 `spring.index.ignore` 设置为 `true` 来回退到常规的 Classpath 安排(好像根本没有索引)。系统属性或 Classpath 根目录下的 `spring.properties` 文件中。

## 1.11. 使用 JSR 330 标准 Comments

从 Spring 3.0 开始，Spring 提供对 JSR-330 标准 Comments(依赖注入)的支持。这些 Comments 的扫描方式与 SpringComments 的扫描方式相同。要使用它们，您需要在 Classpath 中有相关的 jar。

### iNote

如果使用 Maven，则 `javax.inject` 工件在标准 Maven 存储库(

<http://repo1.maven.org/maven2/javax/inject/javax.inject/1/>中可用。您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. @Inject 和@Named 的依赖注入

可以使用 `@javax.inject.Inject` 代替 `@Autowired`，如下所示：

```
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，您可以在字段级别，方法级别和构造函数参数级别使用 `@Inject`。此外

，您可以将注入点声明为 `Provider`，以允许按需访问范围较小的 bean，或者通过

`Provider.get()` 调用来懒惰地访问其他 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(...);
    ...
}
}
```

如果要为应注入的依赖项使用限定名称，则应使用 `@Named` 注解，如以下示例所示：

```
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。这在这里更加适用，因为 `@Inject` 没有 `required` 属性。以下一对示例显示了如何使用 `@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. @Named 和@ManagedBean：@ComponentComments 的标准等效项

代替 `@Component`，可以使用 `@javax.inject.Named` 或 `javax.annotation.ManagedBean`，如以下示例所示：

```
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` 是很常见的。`@Named` 可以类似的方式使用，如以下示例所示：

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

@Named
public class SimpleMovieLister {

    private MovieFinder movieFinder;

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

    // ...
}
```

使用 `@Named` 或 `@ManagedBean` 时，可以使用与使用 SpringComments 完全相同的方式来使用组件扫描，如以下示例所示：

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

```
public class AppConfig {  
    ...  
}
```

### ①Note

与 `@Component` 相反, JSR-330 `@Named` 和 JSR-250 `ManagedBean` **Comments** 是不可组合的。您应该使用 Spring 的构造型模型来构建自定义组件 **Comments**。

### 1.11.3. JSR-330 标准 Comments 的局限性

当使用标准 **Comments** 时, 您应该知道某些重要功能不可用, 如下表所示:

表 6. Spring 组件模型元素与 JSR-330 变体

Spring	javax.inject.*	javax.inject 限制/Comments
<code>@Autowired</code>	<code>@Inject</code>	<code>@Inject</code> 没有“必需”属性 可以与 Java 8 的 <code>Optional</code> 一起使用。
<code>@Component</code>	<code>@Named/@ManagedBean</code>	JSR-330 不提供可组合的模型 仅提供一种识别命名组件 的方法。
<code>@Scope("singleton")</code>	<code>@Singleton</code>	JSR-330 的默认范围类似于 Spring 的 <code>prototype</code> 。但 是, 为了使其与 Spring 的常 规默认设置保持一致, 默认 情况下, 在 Spring 容器中声

Spring	javax.inject.*	javax.inject 限制/Comments
		<p>明的 JSR-330 bean 为 <code>singleton</code>。为了使用 <code>singleton</code> 以外的范围，您应该使用 Spring 的 <code>@Scope</code> 注解。 <code>javax.inject</code> 还提供 <code>@ScopeComments</code>。但是，此仅用于创建自己的 Comments。</p>
@Qualifier	@ Qualifier/@ Named	<p><code>javax.inject.Qualifier</code> 只是用于构建自定义限定符的元 Comments。可以通过 <code>javax.inject.Named</code> 关联具体的 <code>String</code> 限定词(如带有值的 Spring 的 <code>@Qualifier</code>)。</p>
@Value	-	no equivalent
@Required	-	no equivalent
@Lazy	-	no equivalent
ObjectFactory	Provider	<code>javax.inject.Provider</code>

Spring	javax.inject.*	javax.inject 限制/Comments
		<p>是 Spring 的 <code>ObjectFactory</code> 的直接替代方法，只是使用较短的 <code>get()</code> 方法名。它也可以与 Spring 的 <code>@Autowired</code> 或未 <code>Comments</code> 的构造函数和 <code>setter</code> 方法结合使用。</p>

## 1.12. 基于 Java 的容器配置

本节介绍如何在 Java 代码中使用 `Comments` 来配置 Spring 容器。它包括以下主题：

- [基本概念：@Bean 和@Configuration](#)
- [使用 AnnotationConfigApplicationContext 实例化 Spring 容器](#)
- [使用@BeanComments](#)
- [使用@Configuration 注解](#)
- [组成基于 Java 的配置](#)
- [Bean 定义配置文件](#)
- [PropertySource Abstraction](#)
- [Using @PropertySource](#)
- [声明中的占位符解析](#)

### 1.12.1. 基本概念：@Bean 和@Configuration

Spring 的新 Java 配置支持中的主要工件是 `@Configuration` Comments 的类和 `@Bean` Comments 的方法。

`@Bean` Comments 用于指示方法实例化，配置和初始化要由 Spring IoC 容器 Management 的新对象。对于熟悉 Spring 的 `<beans/>` XML 配置的人来说，`@Bean` Comments 与 `<bean/>` 元素具有相同的作用。您可以对任何 Spring `@Component` 使用 `@Bean` Comments 的方法。但是，它们最常与 `@Configuration` bean 一起使用。

用 `@Configuration` Comments 类表示该类的主要目的是作为 Bean 定义的来源。此外，`@Configuration` 类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` 类的内容如下：

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

上面的 `AppConfig` 类等效于下面的 Spring `<beans/>` XML：

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

完整的`@Configuration` 与“精简”`@Bean` 模式？

当在未使用 `@Configuration` Comments 的类中声明 `@Bean` 方法时，它们被称为以“精简”模式进行处理。在 `@Component` 或什至在普通的旧类中声明的 Bean 方法被认为是“精简版”，其中包含类的主要目的不同，而 `@Bean` 方法在那里具有某种优势。例如，服务组件可以通过每个适用的组件类上的附加 `@Bean` 方法向容器公开 Management 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 `@Configuration` 不同，lite `@Bean` 方法无法声明 Bean 之间的依赖关系。取而代之的是，它们在其包含组件的内部状态上运行，并且可以选择地在它们可以声明的参数上运行。因此，此类 `@Bean` 方法不应调用其他 `@Bean` 方法。实际上，每个此类方法仅是用于特定 bean 引用的工厂方法，而没有任何特殊的运行时语义。这里的积极副作用是，不必在运行时应用 CGLIB 子类，因此在类设计方面没有任何限制(即，包含类可能为 `final` 等)。

在常见情况下，将在 `@Configuration` 类中声明 `@Bean` 方法，以确保始终使用“完全”模式，因此跨方法引用将重定向到容器的生命周期 Management。这样可以防止通过常规 Java 调用意外地调用同一 `@Bean` 方法，从而有助于减少在“精简”模式下运行时难以追查的细微错误。

以下各节将对 `@Bean` 和 `@Configuration` Comments 进行深入讨论。但是，首先，我们介绍了通过基于 Java 的配置使用创建 spring 容器的各种方法。

## 1.12.2. 使用 AnnotationConfigApplicationContext 实例化 Spring 容器

以下各节记录了 Spring 3.0 中引入的 Spring 的 `AnnotationConfigApplicationContext`。这种通用的 `ApplicationContext` 实现不仅可以接受 `@Configuration` 类作为 Importing，而且还可以接受普通 `@Component` 类和带有 JSR-330 元数据 Comments 的类。

当提供 `@Configuration` 类作为 Importing 时，`@Configuration` 类本身被注册为 bean 定义，并且该类中所有已声明的 `@Bean` 方法也被注册为 bean 定义。

当提供 `@Component` 和 JSR-330 类时，它们将注册为 bean 定义，并且假定在必要时在这些类中使用了诸如 `@Autowired` 或 `@Inject` 之类的 DI 元数据。

### Simple Construction

与实例化 `ClassPathXmlApplicationContext` 时将 Spring XML 文件用作 Importing 的方式几乎相同，实例化 `AnnotationConfigApplicationContext` 时可以将 `@Configuration` 类用作

Importing。如下面的示例所示，这允许完全不使用 XML 来使用 Spring 容器：

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

如前所述，`AnnotationConfigApplicationContext` 不限于仅使用 `@Configuration` 个类。可以将任何 `@Component` 或 JSR-330 带 Comments 的类作为 Importing 提供给构造函数，如以下示例所示：

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

前面的示例假定 `MyServiceImpl`，`Dependency1` 和 `Dependency2` 使用 Spring 依赖项注入 Comments，例如 `@Autowired`。

## 使用寄存器以编程方式构建容器(Class<?>...)

您可以使用 no-arg 构造函数实例化 `AnnotationConfigApplicationContext`，然后使用 `register()` 方法对其进行配置。以编程方式构建 `AnnotationConfigApplicationContext` 时，此方法特别有用。以下示例显示了如何执行此操作：

```
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 ...) 启用组件扫描

要启用组件扫描，您可以如下 Comments 您的 `@Configuration` 类：

```
@Configuration  
@ComponentScan(basePackages = "com.acme") (1)  
public class AppConfig {  
    ...  
}
```

- (1) 此 `Comments` 启用组件扫描。

### Tip

有经验的 Spring 用户可能熟悉 Spring 的 `context:` 名称空间中的等效 XML 声明，如以下示例所示：

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

在前面的示例中，扫描 `com.acme` 包以查找带有 `@Component` `Comments` 的任何类，并将这些类注册为容器内的 Spring bean 定义。`AnnotationConfigApplicationContext` 公开 `scan(String...)` 方法以允许相同的组件扫描功能，如以下示例所示：

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

### Note

请记住，`@Configuration` 类与 `@Component` 是 [meta-annotated](#)，因此它们是组件扫描的候选对象。在前面的示例中，假定在 `com.acme` 包（或下面的任何包）中声明了 `AppConfig`，则在对 `scan()` 的调用期间将其拾取。在 `refresh()` 上，将处理其所有 `@Bean` 方法并将其实现注册为容器内的 bean 定义。

## 通过 AnnotationConfigWebApplicationContext 支持 Web 应用程序

AnnotationConfigWebApplicationContext 可提供 AnnotationConfigApplicationContext 的 WebApplicationContext 变体。在配置 Spring ContextLoaderListener servlet 聆听器, Spring MVC DispatcherServlet 等时, 可以使用此实现。以下 web.xml 片段配置了典型的 Spring MVC Web 应用程序(请注意 contextClass context-param 和 init-param 的使用):

```
<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.AnnotationConfigWebApplicationConte
        </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-
    </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.AnnotationConfigWebApplicationC
            </param-value>
        </init-param>
        <!-- Again, config locations must consist of one or more comma- or space-delimi
            and fully-qualified @Configuration classes -->
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>com.acme.web.MvcConfig</param-value>
        </init-param>
    </servlet>

    <!-- map all requests for /app/* to the dispatcher servlet -->
    <servlet-mapping>
```

```
<servlet-name>dispatcher</servlet-name>
<url-pattern>/app/*</url-pattern>
</servlet-mapping>
</web-app>
```

### 1.12.3. 使用@BeanComments

`@Bean` 是方法级 Comments，是 XML `<bean/>` 元素的直接类似物。Comments 支持 `<bean/>` 提供的某些属性，例如：\* `init-method` \* `destroy-method` \* `autowiring` \* `name`。

您可以在 `@Configuration` Comments 的类或 `@Component` Comments 的类中使用 `@Bean` Comments。

#### 声明一个 Bean

要声明 bean，可以使用 `@Bean` Comments 对方法进行 Comments。您可以使用此方法在指定为该方法的返回值的类型的 `ApplicationContext` 内注册 bean 定义。默认情况下，Bean 名称与方法名称相同。以下示例显示了 `@Bean` 方法声明：

```
@Configuration
public class AppConfig {

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

前面的配置与下面的 Spring XML 完全等效：

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

这两个声明使一个名为 `transferService` 的 bean 在 `ApplicationContext` 中可用，并绑定到 `TransferServiceImpl` 类型的对象实例，如以下文本图像所示：

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

您还可以使用接口(或 Base Class)返回类型声明 `@Bean` 方法，如以下示例所示：

```
@Configuration
public class AppConfig {

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

但是，这将提前类型预测的可见性限制为指定的接口类型(`TransferService`)。然后，使用只对容器知道一次的完整类型(`TransferServiceImpl`)，实例化受影响的单例 bean。非惰性单例 bean 根据其声明 Sequences 实例化，因此您可能会看到不同的类型匹配结果，具体取决于另一个组件何时尝试通过未声明的类型进行匹配(例如 `@Autowired TransferServiceImpl`，该实例仅在实例化 `transferService` bean 时才解析。)

### Tip

如果您 pass 语句的服务接口一致地引用类型，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由其实现类型潜在引用的组件，声明可能的最具体的返回类型(至少与引用您的 bean 的注入点所要求的具体类型一样)更为安全。

## Bean Dependencies

带有 `@Bean` Comments 的方法可以具有任意数量的参数，这些参数描述构建该 bean 所需的依赖关系。例如，如果我们的 `TransferService` 要求 `AccountRepository`，则可以使用方法参数实现该依赖关系，如以下示例所示：

```
@Configuration
public class AppConfig {

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

```
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几乎相同。有关更多详细信息，请参见[相关部分](#)。

## 接收生命周期回调

任何使用 `@Bean` Comments 定义的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` Comments。有关更多详细信息，请参见[JSR-250 annotations](#)。

还完全支持常规的 Spring [lifecycle](#) 回调。如果 bean 实现 `InitializingBean`，

`DisposableBean` 或 `Lifecycle`，则容器将调用它们各自的方法。

也完全支持 `*Aware` 接口的标准集合(例如[BeanFactoryAware](#), [BeanNameAware](#),

[MessageSourceAware](#), [ApplicationContextAware](#)等)。

`@Bean` Comments 支持指定任意的初始化和销毁回调方法，就像 `bean` 元素上的 Spring XML 的 `init-method` 和 `destroy-method` 属性一样，如以下示例所示：

```
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();
}
```

### iNote

默认情况下，使用 Java 配置定义的具有公共 `close` 或 `shutdown` 方法的 bean 会自动通过销毁回调注册。如果您有一个公共的 `close` 或 `shutdown` 方法，并且您不希望在容器关闭时调用它，则可以将 `@Bean(destroyMethod="" )` 添加到 bean 定义中以禁用默认的 `(inferred)` 模式。

默认情况下，您可能要对通过 JNDI 获取的资源执行此操作，因为其生命周期是在应用程序外部进行 Management 的。特别是，请确保始终对 `DataSource` 进行操作，因为这在 Java EE 应用程序服务器上是有问题的。

以下示例显示了如何防止 `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，通常使用程序化 JNDI 查找，方法是使用 Spring 的 `JndiTemplate` 或 `JndiLocatorDelegate` 帮助器，或者直接使用 JNDI `InitialContext` 用法，而不使用 `JndiObjectFactoryBean` 变体(这将迫使您将返回类型声明为 `FactoryBean` 类型，而不是实际的目标。类型，使其更难以在打算引用此处提供的资源的其他 `@Bean` 方法中用于交叉引用调用。

对于前面 Comments 中的示例中的 `BeanOne`，在构造期间直接调用 `init()` 方法同样有效，如以下示例所示：

```
@Configuration  
public class AppConfig {  
  
    @Bean  
    public BeanOne beanOne() {  
        BeanOne beanOne = new BeanOne();  
        beanOne.init();  
        return beanOne;  
    }  
  
    // ...  
}
```

### Tip

当您直接使用 Java 工作时，您可以对对象执行任何操作，而不必总是依赖于容器生命周期。

## 指定 Bean 范围

Spring 包含 `@Scope` 注解，以便您可以指定 bean 的范围。

### 使用@Scope 注解

您可以指定使用 `@Bean` [Comments](#) 定义的 bean 应该具有特定范围。您可以使用 [Bean Scopes](#) 部分中指定的任何标准范围。

默认范围是 `singleton`，但是您可以使用 `@Scope` 注解覆盖它，如以下示例所示：

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

### @Scope 和 scoped-proxy

Spring 提供了一种通过[scoped proxies](#)处理范围内的依赖项的便捷方法。使用 XML 配置时创建此类代理的最简单方法是 `<aop:scoped-proxy/>` 元素。使用 `@Scope` [Comments](#) 在 Java 中配置

bean 可以提供与 `proxyMode` 属性等效的支持。缺省值为无代理(`ScopedProxyMode.NO`)，但是您可以指定 `ScopedProxyMode.TARGET_CLASS` 或 `ScopedProxyMode.INTERFACES`。

如果使用 Java 从 XML 参考文档(请参阅[scoped proxies](#))将作用域代理示例移植到我们的 `@Bean`，则它类似于以下内容：

```
// 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 命名

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，如以下示例所示：

```
@Configuration
public class AppConfig {

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

## Bean Aliasing

如[Naming Beans](#)中所述，有时希望为单个 bean 提供多个名称，否则称为 bean 别名。`@Bean` 注解的 `name` 属性为此目的接受一个 String 数组。以下示例说明如何为 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 Description

有时，提供有关 bean 的更详细的文本描述会很有帮助。当出于监视目的而暴露(可能通过 JMX)bean 时，这尤其有用。

要将说明添加到 `@Bean`，可以使用[@Description](#)注解，如以下示例所示：

```
@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 注解

`@Configuration` 是类级别的 Comments，指示对象是 Bean 定义的源。`@Configuration` 类通过公共 `@Bean` 带 Comments 的方法声明 Bean。对 `@Configuration` 类的 `@Bean` 方法的调用也可以用于定义 Bean 间的依赖关系。有关一般介绍，请参见[基本概念：@Bean 和@Configuration](#)。

### 注入 Bean 间的依赖关系

当 bean 相互依赖时，表示这种依赖关系就像让一个 bean 方法调用另一个一样简单，如以下示例所示：

```
@Configuration
public class AppConfig {

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

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

在前面的示例中，`beanOne` 通过构造函数注入接收对 `beanTwo` 的引用。

### iNote

仅当在 `@Configuration` 类中声明 `@Bean` 方法时，此声明 bean 间依赖性的方法才有效。

您不能通过使用普通 `@Component` 类来声明 Bean 间的依赖关系。

## 查找方法注入

如前所述，查找方法注入是您不应该使用的高级功能。在单例作用域的 bean 依赖于原型作用域的 bean 的情况下，这很有用。将 Java 用于这种类型的配置为实现此模式提供了自然的方法。以下示例显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，您可以创建 `CommandManager` 的子类，在该子类中，抽象

`createCommand()` 方法被覆盖，从而可以查找新的(原型)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 command() overridden
    // to return a new prototype Command object
    return new CommandManager() {
        protected Command createCommand() {
            return asyncCommand();
        }
    }
}

```

## 有关基于 Java 的配置在内部如何工作的更多信息

考虑下面的示例，该示例显示了一个被两次调用的 `@Bean` `Comments` 方法：

```

@Configuration
public class AppConfig {

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

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

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

```

`clientDao()` 在 `clientService1()` 中被调用过一次，在 `clientService2()` 中被调用过一次。由于此方法创建了 `ClientDaoImpl` 的新实例并返回它，因此通常希望有两个实例(每个服务一个)。那绝对是有问题的：在 Spring 中，实例化的 bean 默认具有 `singleton` 范围。这就是神奇的地方：所有 `@Configuration` 类在启动时都使用 `CGLIB` 子类化。在子类中，子方法在调用父方法并创建新实例之前，首先检查容器中是否有任何缓存(作用域)的 bean。

### iNote

根据 bean 的范围，行为可能有所不同。我们在这里谈论单例。

### iNote

从 Spring 3.2 开始，不再需要将 CGLIB 添加到您的 Classpath 中，因为 CGLIB 类已经在

`org.springframework.cglib` 下重新打包并直接包含在 `spring-core` JAR 中。

### Tip

由于 CGLIB 在启动时会动态添加功能，因此存在一些限制。特别是，配置类不能是最终的。

但是，从 4.3 版本开始，配置类中允许使用任何构造函数，包括对默认注入使用

`@Autowired` 或单个非默认构造函数声明。

如果您希望避免任何 CGLIB 施加的限制，请考虑在非 `@Configuration` 类(例如，在普通

`@Component` 类上)声明 `@Bean` 方法。然后，不会拦截 `@Bean` 方法之间的跨方法调用，因

此您必须专门依赖那里的构造函数或方法级别的依赖项注入。

## 1.12.5. 组成基于 Java 的配置

Spring 的基于 Java 的配置功能使您可以编写注解，这可以降低配置的复杂性。

### 使用@导入 Comments

就像 Spring XML 文件中使用 `<import/>` 元素来帮助模块化配置一样，`@Import` Comments 允

许从另一个配置类中加载 `@Bean` 定义，如以下示例所示：

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

```
}

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

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

现在，无需在实例化上下文时同时指定 `ConfigA.class` 和 `ConfigB.class`，只需显式提供 `ConfigB`，如以下示例所示：

```
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，而不是要求您在构造过程中记住大量的 `@Configuration` 类。

### Tip

从 Spring Framework 4.2 开始，`@Import` 还支持引用常规组件类，类似于 `AnnotationConfigApplicationContext.register` 方法。如果要通过使用一些配置类作为入口点来显式定义所有组件，从而避免组件扫描，则此功能特别有用。

## 注入对导入的@Bean 定义的依赖关系

前面的示例有效，但过于简单。在大多数实际情况下，Bean 在配置类之间相互依赖。使用 XML 时，这不是问题，因为不涉及任何编译器，并且您可以声明 `ref="someBean"` 并信任 Spring 在容器初始化期间进行处理。使用 `@Configuration` 类时，Java 编译器会在配置模型上施加约束，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。与[我们已经讨论过了一样](#)，[@Bean](#) 方法可以具有任意数量的描述 Bean 依赖关系的参数。考虑以下具有多个 [@Configuration](#) 类的更实际的场景，每个类均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
    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相同的结果。请记住，[@Configuration](#) 类最终仅是容器中的另一个 bean：这意味着它们可以利用 [@Autowired](#) 和 [@Value](#) 注入以及与任何其他 bean 相同的其他功能。

### ⚠️ Warning

确保以这种方式注入的依赖项只是最简单的一种。[@Configuration](#) 类是在上下文初始化

期间非常早地处理的，并且强制以这种方式注入依赖项可能导致意外的早期初始化。如上例所示，请尽可能使用基于参数的注入。

另外，要特别注意 `BeanPostProcessor` 和 `BeanFactoryPostProcessor` 到 `@Bean` 的定义。通常应将它们声明为 `static @Bean` 方法，而不触发其包含的配置类的实例化。否则，`@Autowired` 和 `@Value` 不适用于配置类本身，因为它太早被创建为 Bean 实例。

以下示例说明如何将一个 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);
```

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

## Tip

从 Spring Framework 4.3 开始，仅支持 `@Configuration` 类中的构造方法注入。还要注意，如果目标 bean 仅定义一个构造函数，则无需指定 `@Autowired`。在前面的示例中，`RepositoryConfig` 构造函数上不需要 `@Autowired`。

完全合格的 importbean，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化，但是确切地确定自动装配的 Bean 定义在何处声明仍然有些模棱两可。例如，当开发人员查看 `ServiceConfig` 时，您如何确切知道 `@Autowired AccountRepository` 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](#)提供的工具可以呈现图形，显示所有连线的方式，这可能就是您所需要的。另外，您的 Java IDE 可以轻松找到 `AccountRepository` 类型的所有声明和使用，并快速向您显示返回该类型的 `@Bean` 方法的位置。

如果这种歧义是不可接受的，并且您希望从 IDE 内直接从一个 `@Configuration` 类导航到另一个 `@Configuration` 类，请考虑自动装配配置类本身。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());  
    }  
}
```

在上述情况下，定义 `AccountRepository` 是完全明确的。但是，`ServiceConfig` 现在与

`RepositoryConfig` 紧密耦合。那是权衡。通过使用基于接口或基于抽象类的 `@Configuration`

类，可以稍微缓解这种紧密耦合。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 {

    @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` 松散耦合，并且内置的 IDE 工具仍然

有用：您可以轻松地获得 `RepositoryConfig` 实现的类型层次结构。这样，对 `@Configuration`

类及其依赖项进行导航变得与导航基于接口的代码的通常过程没有什么不同。

## Tip

如果要影响某些 Bean 的启动创建 Sequences, 请考虑将其中一些声明为 `@Lazy` (用于首次访问而不是在启动时创建)或 `@DependsOn` 声明其他某些 Bean(确保在当前 Bean 之前创建了特定的其他 Bean), 而不是后者的直接依赖项所暗示的含义)。

## 有条件地包含@Configuration 类或@Bean 方法

基于某些任意系统状态, 有条件地启用或禁用完整的 `@Configuration` 类甚至单个 `@Bean` 方法通常很有用。一个常见的示例是仅在 Spring `Environment` 中启用了特定概要文件时才使用 `@Profile` Comments 来激活 bean(有关详细信息, 请参见[Bean 定义配置文件](#))。

`@Profile` Comments 实际上是通过使用更灵活的称为[@Conditional](#)的 Comments 来实现的。

`@Conditional` 注解指示在注册 `@Bean` 之前应参考的

`org.springframework.context.annotation.Condition` 特定实现。

`Condition` 接口的实现提供了一个 `matches(...)` 方法, 该方法返回 `true` 或 `false`。例如, 以下清单显示了用于 `@Profile` 的实际 `Condition` 实现:

```
@Override
public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
    if (context.getEnvironment() != null) {
        // Read the @Profile annotation attributes
        MultiValueMap<String, Object> attrs = metadata.getAllAnnotationAttributes(Profile.class);
        if (attrs != null) {
            for (Object value : attrs.get("value")) {
                if (context.getEnvironment().acceptsProfiles(((String[]) value)))
                    return true;
            }
        }
        return false;
    }
    return true;
}
```

有关更多详细信息, 请参见[@Conditional](#) javadoc。

## 结合 Java 和 XML 配置

Spring 的 `@Configuration` 类支持并非旨在 100% 完全替代 Spring XML。某些工具(例如 Spring XML 名称空间)仍然是配置容器的理想方法。在使用 XML 方便或有必要的情况下，您可以选择：通过使用 `ClassPathXmlApplicationContext` 以“以 XML 为中心”的方式实例化容器，或通过使用 `AnnotationConfigApplicationContext` 和以“以 Java 为中心”的方式实例化容器。

`@ImportResource` Comments 以根据需要导入 XML。

### 以 XML 为中心的`@Configuration` 类的使用

最好从 XML 引导 Spring 容器，并以即席方式包含 `@Configuration` 类。例如，在使用 Spring XML 的大型现有代码库中，根据需要创建 `@Configuration` 类并从现有 XML 文件中包含它们很容易。在本节的后面，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` 类声明为纯 Spring `<bean/>` 元素

请记住，`@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 `<context:annotation-config/>` 已打开，因此容器会识别 `@Configuration` 注解并正确处理 `AppConfig` 中声明的 `@Bean` 方法。

以下示例显示了 Java 中的普通配置类：

```
@Configuration
public class AppConfig {

    @Autowired
    private DataSource dataSource;

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

```

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

```

以下示例显示了示例 `system-test-config.xml` 文件的一部分：

```

<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>

```

以下示例显示了一个可能的 `jdbc.properties` 文件：

```

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

```

```

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

```

### iNote

在 `system-test-config.xml` 文件中，`AppConfig` `<bean/>` 没有声明 `id` 元素。尽管这样做是可以接受的，但是由于没有其他 `bean` 曾经引用过它，因此这是不必要的，并且不太可能通过名称从容器中显式获取。类似地，`DataSource` `bean` 只能按类型自动装配，因此并不需要严格要求显式 `bean id`。

使用\<>选择 `@Configuration` 个类

因为 `@Configuration` 使用 `@Component` 进行元 Comments，所以 `@Configuration` Comments 的类自动成为组件扫描的候选对象。使用与上一个示例中描述的场景相同的场景，我们可以重新定义 `system-test-config.xml` 以利用组件扫描的优势。请注意，在这种情况下，我们无需显式声明 `<context:annotation-config/>`，因为 `<context:component-scan/>` 启用相同的功能。

以下示例显示了修改后的 `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>
```

## @Configuration 以类为中心的 XML 与@ImportResource 的使用

在 `@Configuration` 类是配置容器的主要机制的应用程序中，仍然可能有必要至少使用一些 XML。在这些情况下，您可以使用 `@ImportResource` 并仅定义所需的 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。以下示例(包括配置类，定义 Bean 的 XML 文件，属性文件和 `main` 类)显示了如何使用 `@ImportResource` Comments 来实现按需使用 XML 的“以 Java 为中心”的配置：

```
@Configuration
@ImportResource("classpath:/com/acme/properties-config.xml")
public class AppConfig {

    @Value("${jdbc.url}")
    private String url;

    @Value("${jdbc.username}")
    private String username;

    @Value("${jdbc.password}")
    private String password;
```

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

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

```
jdbc.properties  
jdbc.url=jdbc:hsqldb:hsq://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);  
    // ...  
}
```

## 1.13. 环境抽象

[Environment](#) 接口是集成在容器中的抽象，用于对应用程序环境的两个关键方面进行建模：[profiles](#) 和 [properties](#)。

概要文件是仅在给定概要文件处于活动状态时才向容器注册的 Bean 定义的命名逻辑组。可以将 Bean 分配给概要文件，无论是以 XML 定义还是带有 Comments。[Environment](#) 对象与配置文件相关的作用是确定当前哪些配置文件(如果有)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。

属性在几乎所有应用程序中都起着重要作用，并且可能源自多种来源：属性文件，JVM 系统属性，系统环境变量，JNDI，Servlet 上下文参数，即席 [Properties](#) 对象，[Map](#) 对象等等。

[Environment](#) 对象相对于属性的作用是为用户提供方便的服务界面，用于配置属性源并从中解析属性。

### 1.13.1. Bean 定义配置文件

Bean 定义配置文件在核心容器中提供了一种机制，该机制允许在不同环境中注册不同的 Bean。

“环境”一词对不同的用户可能具有不同的含义，并且此功能可以帮助解决许多用例，包括：

- 在开发中针对内存中的数据源进行工作，而不是在进行 QA 或生产时从 JNDI 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视基础结构。
- 为 ClientA 和 ClientB 部署注册 bean 的自定义实现。

考虑实际应用中需要 `DataSource` 的第一个用例。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，假设该应用程序的数据源已在生产应用程序服务器的 JNDI 目录中注册，请考虑如何将该应用程序部署到 QA 或生产环境中。现在，我们的 `dataSource` bean 看起来像下面的清单：

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

问题是根据当前环境在使用这两种变体之间进行切换。随着时间的流逝，Spring 用户已经设计出了多种方法来完成此任务，通常依赖于系统环境变量和包含  `${placeholder}`  令牌的 XML

`<import/>` 语句的组合，这些语句根据环境变量的值解析为正确的配置文件路径。Bean 定义配置文件是一项核心容器功能，可提供此问题的解决方案。

如果我们概括前面特定于环境的 Bean 定义示例中所示的用例，那么最终需要在某些上下文中而不是在其他上下文中注册某些 Bean 定义。您可能会说您要在情况 A 中注册一个特定的 bean 定义配置文件，在情况 B 中注册一个不同的配置文件。我们首先更新配置以反映这种需求。

## Using @Profile

`@ProfileComments` 使您可以指示一个或多个指定配置文件处于活动状态时有资格注册的组件。使用前面的示例，我们可以如下重写 `dataSource` 配置：

```
@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");
    }
}
```

### iNote

如前所述，对于 `@Bean` 方法，通常选择使用程序化 JNDI 查找，方法是使用 Spring 的

`JndiTemplate` / `JndiLocatorDelegate` 帮助器或前面显示的直接 JNDI

`InitialContext` 用法，而不使用 `JndiObjectFactoryBean` 变体，这将迫使您将返回类型声明为 `FactoryBean` 类型。

配置文件字符串可以包含简单的配置文件名称(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`)。概要文件表达式中支持以下运算符：

- ! : 配置文件的逻辑“非”
- & : 配置文件的逻辑“与”
- | : 配置文件的逻辑“或”

### iNote

不使用括号不能混合使用 & 和 | 运算符。例如， production & us-east | eu-central 不是有效的表达式。它必须表示为 production & (us-east | eu-central)。

您可以将 @Profile 用作 [meta-annotation](#)，以创建自定义的合成 Comments。以下示例定义了一个自定义 @Production 注解，您可以将其用作 @Profile("production") 的替代品：

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

### Tip

如果 @Configuration 类用 @Profile 标记，则与该类关联的所有 @Bean 方法和 @Import Comments 都将被绕过，除非一个或多个指定的配置文件处于活动状态。如果 @Component 或 @Configuration 类标记为 @Profile({"p1", "p2"})，则除非已激活配置文件'p1'或'p2'，否则不会注册或处理该类。如果给定的配置文件以 NOT 运算符( ! )为前缀，则仅在该配置文件未激活时才注册带 Comments 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，则会进行注册。

@Profile 也可以在方法级别声明为仅包含配置类的一个特定 Bean(例如，针对特定 Bean 的替代

变体), 如以下示例所示:

```
@Configuration
public class AppConfig {

    @Bean("dataSource")
    @Profile("development") (1)
    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") (2)
    public DataSource jndiDataSource() throws Exception {
        Context ctx = new InitialContext();
        return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");
    }
}
```

- (1) `standaloneDataSource` 方法仅在 `development` Profile 中可用。
- (2) `jndiDataSource` 方法仅在 `production` Profile 中可用。

### iNote

对于 `@Bean` 方法上的 `@Profile`, 可能适用特殊情况: 对于具有相同 Java 方法名称的 `@Bean` 方法重载(类似于构造函数重载), 必须在所有重载方法上一致声明 `@Profile` 条件。如果条件不一致, 则仅重载方法中第一个声明的条件很重要。因此, `@Profile` 不能用于选择具有特定自变量签名的重载方法。在创建时, 相同 `bean` 的所有工厂方法之间的解析都遵循 Spring 的构造函数解析算法。

如果要定义具有不同概要文件条件的备用 Bean, 请使用 `@Bean name` 属性使用不同的 Java 方法名称来指向相同的 Bean 名称, 如前面的示例所示。如果参数签名都相同(例如, 所有变体都具有 no-arg 工厂方法), 则这是首先在有效 Java 类中表示这种排列的唯一方法(因为只能有一个特定名称和参数签名的方法)。

## XML Bean 定义配置文件

XML 对应项是 `<beans>` 元素的 `profile` 属性。我们前面的示例配置可以用两个 XML 文件重写，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>
```

也可以避免在同一文件中拆分和嵌套 `<beans/>` 元素，如以下示例所示：

```
<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="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` 已被限制为仅允许这些元素作为文件中的最后一个元素。这应该有助于提供灵 Active，而不会引起 XML 文件混乱。

## iNote

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 `!` 运算符取消配置文件。也可以通过嵌套配置文件来应用逻辑“和”，如以下示例所示：

```
<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>
```

在前面的示例中，如果 `production` 和 `us-east` 配置文件都处于活动状态，则 `dataSource` bean 被公开。

## 激活 Profile

现在，我们已经更新了配置，我们仍然需要指示 Spring 哪个配置文件处于活动状态。如果我们现在启动示例应用程序，将会看到一个 `NoSuchBeanDefinitionException` 抛出，因为容器找不到名为 `dataSource` 的 Spring bean。

可以通过多种方式来激活配置文件，但是最直接的方法是针对通过 `ApplicationContext` 可用的 `Environment` API 以编程方式进行配置。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，您还可以通过 `spring.profiles.active` 属性以声明方式激活概要文件，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` 来指定，甚至可以作为 JNDI 中

的条目来指定(请参阅[PropertySource Abstraction](#))。在集成测试中，可以使用 `spring-test` 模块中的 `@ActiveProfiles` 注解来声明活动配置文件(请参阅[使用环境配置文件进行上下文配置](#))。

请注意，配置文件不是“非此即彼”的命题。您可以一次激活多个配置文件。您可以通过编程方式为 `setActiveProfiles()` 方法提供多个配置文件名称，该方法接受 `String...` varargs。以下示例激活多个配置文件：

```
ctx.getEnvironment().setActiveProfiles("profile1", "profile2");
```

以声明方式，`spring.profiles.active` 可以接受以逗号分隔的配置文件名称列表，如以下示例所示：

```
-Dspring.profiles.active="profile1,profile2"
```

## Default Profile

默认配置文件表示默认情况下启用的配置文件。考虑以下示例：

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

如果没有任何配置文件处于活动状态，则创建 `dataSource`。您可以看到这是为一个或多个 bean 提供默认定义的一种方法。如果启用了任何配置文件，则默认配置文件将不适用。

您可以通过在 `Environment` 上使用 `setDefaultProfiles()` 或 `pass` 语句 `spring.profiles.default` 属性来更改默认配置文件的名称。

## 1.13.2. PropertySource 抽象

Spring 的 `Environment` 抽象提供了对属性源的可配置层次结构的搜索操作。考虑以下清单：

```
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);
```

在前面的代码片段中，我们看到了一种询问 Spring 是否为当前环境定义 `my-property` 属性的高级方法。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，而 Spring 的 `StandardEnvironment` 配置了两个 `PropertySource` 对象—一个代表 JVM 系统属性集 (`System.getProperties()`) 和一个代表系统环境变量集 (`System.getenv()`)。

### iNote

这些默认属性源针对 `StandardEnvironment` 存在，供独立应用程序使用。

`StandardServletEnvironment` 填充了其他默认属性源，包括 `servlet` 配置和 `servlet` 上下文参数。它可以选择启用 `IndiPropertySource`。有关详细信息，请参见 javadoc。

具体来说，当您使用 `StandardEnvironment` 时，如果在运行时存在 `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`。

### Tip

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")` 的过程中在两个地方都同时设置了 `my-property` 属性，则系统属性值“wins”并返回。请注意，属性值不会合并，而是会被前面的条目完全覆盖。

对于常见的 `StandardServletEnvironment`，完整层次结构如下，最高优先级条目位于顶部：

- `ServletConfig` 参数(如果适用, 例如, 在 `DispatcherServlet` 上下文的情况下)
- `ServletContext` 参数(`web.xml` 上下文参数条目)
- JNDI 环境变量( `java:comp/env/` 个条目)
- JVM 系统属性( `-D` 个命令行参数)
- JVM 系统环境(os 环境变量)

最重要的是, 整个机制是可配置的。也许您具有要集成到此搜索中的自定义属性源。为此, 请实现并实例化自己的 `PropertySource` 并将其添加到当前 `Environment` 的 `PropertySources` 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, 在搜索中添加了具有最高优先级的 `MyPropertySource`。如果它包含 `my-property` 属性, 则会检测到并返回该属性, 而支持其他 `PropertySource` 中的任何 `my-property` 属性。[MutablePropertySources API](#) 公开了许多方法, 这些方法可以精确地控制属性源集。

### 1.13.3. 使用@PropertySource

[@PropertySource](#)Comments 为将 `PropertySource` 添加到 Spring 的 `Environment` 提供了一种方便的声明性机制。

给定名为 `app.properties` 的文件, 其中包含键值对 `testbean.name=myTestBean`, 以下 `@Configuration` 类使用 `@PropertySource`, 从而对 `testBean.getName()` 的调用返回 `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 \${...} 占位符都是根据已针对该环境注册的一组属性源来解析的，如以下示例所示：

```
@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;
    }
}
```

假定 my.placeholder 存在于已注册的属性源之一(例如，系统属性或环境变量)中，则占位符将解析为相应的值。如果不是，则使用 default/path 作为默认值。如果未指定默认值并且无法解析属性，则会引发 IllegalArgumentException。

### iNote

根据 Java 8 约定，@PropertySource Comments 是可重复的。但是，所有此类

@PropertySource 注解都需要在同一级别上声明，可以直接在配置类上声明，也可以在同  
一自定义注解中声明为元注解。不建议将直接 Comments 和元 Comments 混合使用，因为  
直接 Comments 会有效地覆盖元 Comments。

## 1.13.4. 声明中的占位符解析

从历史上看，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。这已不再是这种情况。由于 `Environment` 抽象集成在整个容器中，因此很容易通过它路由占位符的解析。这意味着您可以按照自己喜欢的任何方式配置解析过程。您可以更改搜索系统属性和环境变量的优先级，也可以完全删除它们。您还可以根据需要将自己的属性源添加到组合中。

具体来说，只要在 `Environment` 中可用，以下语句无论在 `customer` 属性的定义位置如何都有效：

```
<beans>
    <import resource="com/bank/service/${customer}-config.xml"/>
</beans>
```

## 1.14. 注册一个 LoadTimeWeaver

`LoadTimeWeaver` 被 Spring 使用，以在将类加载到 Java 虚拟机(JVM)中时对其进行动态转换。

要启用加载时编织，可以将 `@EnableLoadTimeWeaving` 添加到 `@Configuration` 类之一，如以下示例所示：

```
@Configuration
@EnableLoadTimeWeaving
public class AppConfig { }
```

另外，对于 XML 配置，可以使用 `context:load-time-weaver` 元素：

```
<beans>
    <context:load-time-weaver/>
</beans>
```

一旦为 `ApplicationContext` 配置，该 `ApplicationContext` 内的任何 bean 都可以实现 `LoadTimeWeaverAware`，从而接收到对加载时编织器实例的引用。这与 [Spring 的 JPA 支持](#) 结合使用特别有用，在这种情况下，JPA 类转换可能需要进行加载时编织。有关更多详细信息，请查阅

[LocalContainerEntityManagerFactoryBean](#) javadoc。有关 AspectJ 加载时编织的更多信息，请参见在 Spring Framework 中使用 AspectJ 进行加载时编织。

## 1.15. ApplicationContext 的其他功能

如[chapter introduction](#)中所讨论，`org.springframework.beans.factory` 包提供了用于 Management 和操作 bean 的基本功能，包括以编程方式。`org.springframework.context` 包添加了[ApplicationContext](#)接口，该接口扩展了 `BeanFactory` 接口，并扩展了其他接口以提供更多面向应用程序框架的样式的附加功能。许多人以完全声明性的方式使用 `ApplicationContext`，甚至没有以编程方式创建它，而是依靠诸如 `ContextLoader` 之类的支持类来自动实例化 `ApplicationContext` 作为 Java EE 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` 界面访问 i18n 样式的消息。
- 通过 `ResourceLoader` 界面访问资源，例如 URL 和文件。
- 事件发布，即通过使用 `ApplicationEventPublisher` 接口发布给实现 `ApplicationListener` 接口的 bean。
- 加载多个(分层)上下文，使每个上下文都通过 `HierarchicalBeanFactory` 接口集中在一个特定的层上，例如应用程序的 Web 层。

### 1.15.1. 使用 MessageSource 进行国际化

`ApplicationContext` 接口扩展了名为 `MessageSource` 的接口，因此提供了国际化(“i18n”)功能。Spring 还提供了 `HierarchicalMessageSource` 接口，该接口可以分层解析消息。这些接口一起提供了 Spring 影响消息解析的基础。这些接口上定义的方法包括：

- `String getMessage(String code, Object[] args, String default, Locale loc)`

：用于从 `MessageSource` 检索消息的基本方法。如果找不到针对指定语言环境的消息，则使用默认消息。使用标准库提供的 `MessageFormat` 功能，传入的所有参数都将成为替换值。

- `String getMessage(String code, Object[] args, Locale loc)` : 与以前的方法基本相同，但有一个区别：不能指定默认消息。如果找不到该消息，则抛出 `NoSuchMessageException`。
- `String getMessage(MessageSourceResolvable resolvable, Locale locale)` : 前面方法中使用的所有属性也都包装在名为 `MessageSourceResolvable` 的类中，您可以将其与该方法一起使用。

加载 `ApplicationContext` 时，它将自动搜索上下文中定义的 `MessageSource` bean。Bean 必须具有名称 `messageSource`。如果找到了这样的 bean，则对先前方法的所有调用都将委派给消息源。如果找不到消息源，则 `ApplicationContext` 尝试查找包含同名 bean 的父对象。如果是这样，它将使用该 bean 作为 `MessageSource`。如果 `ApplicationContext` 找不到任何消息源，则实例化一个空的 `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` 以便进行嵌套消息传递。`StaticMessageSource` 很少使用，但提供了将消息添加到源中的编程方式。以下示例显示

`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>
```

该示例假定您在 Classpath 中定义了三个名为 `format`，`exceptions` 和 `windows` 的资源包。任何解析消息的请求都通过 JDK 标准的 `ResourceBundle` 对象解析消息来处理。就本示例而言，假定上述两个资源束文件的内容如下：

```
# in format.properties  
message=Alligators rock!
```

```
# in exceptions.properties  
argument.required=The {0} argument is required.
```

下一个示例显示了执行 `MessageSource` 功能的程序。请记住，所有 `ApplicationContext` 实现也是 `MessageSource` 实现，因此可以转换为 `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);  
}
```

以上程序的结果输出如下：

```
Alligators rock!
```

总而言之，`MessageSource` 是在名为 `beans.xml` 的文件中定义的，该文件位于 Classpath 的根目录下。`messageSource` bean 定义通过其 `basename` 属性引用了许多资源包。列表中传递给 `basename` 属性的三个文件以 Classpath 的根文件形式存在，分别称为 `format.properties`，`exceptions.properties` 和 `windows.properties`。

下一个示例显示了传递给消息查找的参数。这些参数将转换为 `String` 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.ResourceBundleM
```

```

</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);
    }
}

```

调用 `execute()` 方法得到的结果如下：

```
The userDao argument is required.
```

关于国际化(“i18n”), Spring 的各种 `MessageSource` 实现遵循与标准 JDK  `ResourceBundle` 相同的语言环境解析和后备规则。简而言之，并 `continue` 前面定义的示例 `messageSource`，如果要根据英国(`en-GB`)语言环境解析消息，则可以分别创建名为 `format_en_GB.properties`, `exceptions_en_GB.properties` 和 `windows_en_GB.properties` 的文件。

通常，语言环境解析由应用程序的周围环境 `Management`。在以下示例中，手动指定了针对其解析(英国)消息的语言环境：

```
# in exceptions_en_GB.properties
argument.required=Ebagum lad, the {0} argument is required, I say, required.
```

```

public static void main(final String[] args) {
    MessageSource resources = new ClassPathXmlApplicationContext("beans.xml");
    String message = resources.getMessage("argument.required",
        new Object [] {"userDao"}, "Required", Locale.UK);

```

```
        System.out.println(message);
    }
```

运行上述程序的结果输出如下：

```
Ebagum lad, the 'userDao' argument is required, I say, required.
```

您还可以使用 `MessageSourceAware` 界面获取对已定义的任何 `MessageSource` 的引用。创建和配置 Bean 时，在实现 `MessageSourceAware` 接口的 `ApplicationContext` 中定义的任何 Bean 都会被注入应用程序上下文的 `MessageSource`。

#### ①Note

作为 `ResourceBundleMessageSource` 的替代方法，Spring 提供了 `ReloadableResourceBundleMessageSource` 类。此变体支持相同的 Binding 文件格式，但比基于标准 JDK 的 `ResourceBundleMessageSource` 实现更灵活。特别是，它允许从任何 Spring 资源位置(不仅从 Classpath)读取文件，并支持热重载 Binding 属性文件(同时在它们之间进行有效缓存)。有关详细信息，请参见[ReloadableResourceBundleMessageSource javadoc](#)。

## 1.15.2. 标准和自定义事件

`ApplicationContext` 中的事件处理是通过 `ApplicationEvent` 类和 `ApplicationListener` 接口提供的。如果将实现 `ApplicationListener` 接口的 bean 部署到上下文中，则每次将 `ApplicationEvent` 发布到 `ApplicationContext` 时，都会通知该 bean。本质上，这是标准的 Observer 设计模式。

#### ②Tip

从 Spring 4.2 开始，事件基础结构得到了显着改进，并提供了[annotation-based model](#) 以及

发布任何任意事件的能力(即， 对象不一定从 `ApplicationEvent` 扩展)。发布此类对象后， 我们会为您包装一个事件。

下表描述了 Spring 提供的标准事件：

表 7. 内置事件

Event	Explanation
<code>ContextRefreshedEvent</code>	<p>在初始化或刷新 <code>ApplicationContext</code> 时发布(例如， 通过使用 <code>ConfigurableApplicationContext</code> 接口上的 <code>refresh()</code> 方法)。在这里，“已初始化”是指所有 Bean 都已加载，检测到并激活了后处理器 Bean，已预先实例化单例，并且已准备好使用 <code>ApplicationContext</code> 对象。只要尚未关闭上下文，只要选定的 <code>ApplicationContext</code> 实际上支持这种“热”刷新，就可以多次触发刷新。例如，<code>XmlWebApplicationContext</code> 支持热刷新，但 <code>GenericApplicationContext</code> 不支持。</p>
<code>ContextStartedEvent</code>	<p>在 <code>ConfigurableApplicationContext</code> 界面上使用 <code>start()</code> 方法启动 <code>ApplicationContext</code> 时发布。在这里，“启动”是指所有 <code>Lifecycle</code> bean 都收到一个明确的启动 signal。通常，此 signal 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，尚未在初始化时启动的组件)。</p>

Event	Explanation
<code>ContextStoppedEvent</code>	<p>在 <code>ConfigurableApplicationContext</code> 接口上使用 <code>stop()</code> 方法停止 <code>ApplicationContext</code> 时发布。此处，“已停止”表示所有 <code>Lifecycle</code> bean 都收到一个明确的停止 signal。停止的上下文可以通过 <code>start()</code> 调用重新启动。</p>
<code>ContextClosedEvent</code>	<p>在 <code>ConfigurableApplicationContext</code> 接口上使用 <code>close()</code> 方法关闭 <code>ApplicationContext</code> 时发布。此处，“封闭”表示所有单例 bean 都被破坏。封闭的情境到了生命的尽头。无法刷新或重新启动。</p>
<code>RequestHandledEvent</code>	<p>一个特定于 Web 的事件，告诉所有 Bean HTTP 请求已得到服务。请求完成后，将发布此事件。此事件仅适用于使用 Spring 的 <code>DispatcherServlet</code> 的 Web 应用程序。</p>

您还可以创建和发布自己的自定义事件。以下示例显示了一个简单的类，该类扩展了 Spring 的

`ApplicationEvent` Base Class:

```
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...
}
```

```
}
```

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 Spring bean 来完成的。以下示例显示了此类：

```
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 `ApplicationEventPublisherAware` 并自动调用 `setApplicationEventPublisher()`。实际上，传入的参数是 Spring 容器本身。您正在通过其 `ApplicationEventPublisher` 接口与应用程序上下文进行交互。

要接收自定义 `ApplicationEvent`，您可以创建一个实现 `ApplicationListener` 的类并将其注册为 Spring Bean。以下示例显示了此类：

```
public class BlackListNotifier implements ApplicationListener<BlackListEvent> {  
  
    private String notificationAddress;  
  
    public void setNotificationAddress(String notificationAddress) {  
        this.notificationAddress = notificationAddress;  
    }  
  
    public void onApplicationEvent(BlackListEvent event) {  
        // notify appropriate parties via notificationAddress...  
    }  
}
```

```
    }  
}
```

注意，`ApplicationListener` 通常用您的自定义事件的类型(上一示例中的 `BlackListEvent`)进行参数化。这意味着 `onApplicationEvent()` 方法可以保持类型安全，从而避免了向下转换的任何需要。您可以根据需要注册任意数量的事件侦听器，但是请注意，默认情况下，事件侦听器会同步接收事件。这意味着 `publishEvent()` 方法将阻塞，直到所有侦听器都已完成对事件的处理为止。这种同步和单线程方法的一个优点是，当侦听器收到事件时，如果有可用的事务上下文，它将在发布者的事务上下文内部进行操作。如果需要其他用于事件发布的策略，请参见 Spring 的 [ApplicationEventMulticaster](#) 接口的 javadoc。

以下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>[emailprotected]</value>  
            <value>[emailprotected]</value>  
            <value>[emailprotected]</value>  
        </list>  
    </property>  
</bean>  
  
<bean id="blackListNotifier" class="example.BlackListNotifier">  
    <property name="notificationAddress" value="[emailprotected]" />  
</bean>
```

将所有内容放在一起，当调用 `emailService` bean 的 `sendEmail()` 方法时，如果有任何电子邮件消息应列入黑名单，则会发布 `BlackListEvent` 类型的自定义事件。`blackListNotifier` bean 注册为 `ApplicationListener` 并接收 `BlackListEvent`，此时它可以通知适当的参与者。

### iNote

Spring 的事件机制旨在在同一应用程序上下文内在 Spring bean 之间进行简单的通信。但是，对于更复杂的企业集成需求，单独维护的[Spring Integration](#)项目为构建基于众所周知的 Spring 编程模型的事件驱动的轻量级[pattern-oriented](#)体系结构提供了完整的支持。

## 基于 Comments 的事件监听器

从 Spring 4.2 开始，您可以使用 `EventListener` 注解在托管 Bean 的任何公共方法上注册事件监听器。`BlackListNotifier` 可以重写如下：

```
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...  
    }  
}
```

方法签名再次声明其监听的事件类型，但是这次使用灵活的名称并且没有实现特定的监听器接口。只要实际事件类型在其实现层次结构中解析您的通用参数，也可以通过通用类型来缩小事件类型。

如果您的方法应该监听多个事件，或者您要完全不使用任何参数来定义它，则事件类型也可以在 Comments 本身上指定。以下示例显示了如何执行此操作：

```
@EventListener({ContextStartedEvent.class, ContextRefreshedEvent.class})  
public void handleContextStart() {  
    ...  
}
```

也可以通过使用定义 [SpEL expression](#) 的 Comments 的 `condition` 属性来添加其他运行时过滤，该属性应匹配以针对特定事件实际调用该方法。

以下示例显示了仅当事件的 `content` 属性等于 `my-event` 时，才可以重写我们的通知程序以进行调用：

```
@EventListener(condition = "#blEvent.content == 'my-event'")  
public void processBlackListEvent(BlackListEvent blEvent) {  
    // notify appropriate parties via notificationAddress...  
}
```

每个 `SpEL` 表达式都针对专用上下文进行评估。下表列出了可用于上下文的项目，以便您可以将它

们用于条件事件处理：

表 8. *Event SpEL* 可用的元数据

Name	Location	Description	Example
Event	root object	实际的 <code>ApplicationEvent</code> 。	<code>#root.event</code>
Arguments array	root object	用于调用目标的参数(作为 数组)。	<code>#root.args[0]</code>
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 ) , 则参数名称也可以在 <code>#a&lt;#arg&gt;</code> 下获得, 其中 <code>#arg</code> 代表参数索引(从 0 开始)。	<code>#b1Event</code> 或 <code>#a0</code> (您也可以使用 <code>#p0</code> 或 <code>#p&lt;#arg&gt;</code> 表示 法作为别名)

请注意, 即使您的方法签名实际上引用了已发布的任意对象, `#root.event` 也使您可以访问基础事件。

如果由于处理另一个事件而需要发布一个事件, 则可以更改方法签名以返回应发布的事件, 如以下示例所示:

```
@EventListener  
public ListUpdateEvent handleBlackListEvent(BlackListEvent event) {  
    // notify appropriate parties via notificationAddress and  
    // then publish a ListUpdateEvent...  
}
```

## iNote

[asynchronous listeners](#) 不支持此功能。

此新方法为上述方法处理的每个 `BlackListEvent` 发布一个新的 `ListUpdateEvent`。如果您需要发布多个事件，则可以返回 `Collection` 事件。

## Asynchronous Listeners

如果希望特定的侦听器异步处理事件，则可以重用[常规@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  
}
```

使用异步事件时，请注意以下限制：

- 如果事件监听器抛出 `Exception`，则它不会传播到调用者。有关更多详细信息，请参见 [AsyncUncaughtExceptionHandler](#)。
- 此类事件侦听器无法发送答复。如果您需要发送另一个事件作为处理结果，请注入 [ApplicationEventPublisher](#) 以手动发送事件。

## Ordering Listeners

如果需要先调用一个侦听器，则可以将 `@Order` 注解添加到方法声明中，如以下示例所示：

```
@EventListener  
@Order(42)  
public void processBlackListEvent(BlackListEvent event) {  
    // notify appropriate parties via notificationAddress...  
}
```

## Generic Events

您还可以使用泛型来进一步定义事件的结构。考虑使用 `EntityCreatedEvent<T>`，其中 `T` 是已

创建的实际实体的类型。例如，您可以创建以下监听器定义以仅接收\_4 的 `EntityCreatedEvent`

```
:  
  
    @EventListener  
    public void onPersonCreated(EntityCreatedEvent<Person> event) {  
        ...  
    }
```

由于类型擦除，只有在触发的事件解析了事件监听器所依据的通用参数(即诸如 `class`

`PersonCreatedEvent extends EntityCreatedEvent<Person> { ... }`)时，此方法才起作用。

在某些情况下，如果所有事件都遵循相同的结构，这可能会变得很乏味(就像前面示例中的事件一样)。在这种情况下，您可以实现 `ResolvableTypeProvider` 以指导框架超出运行时环境提供的范围。以下事件显示了如何执行此操作：

```
public class EntityCreatedEvent<T> extends ApplicationEvent implements ResolvableTypeProvider {  
  
    public EntityCreatedEvent(T entity) {  
        super(entity);  
    }  
  
    @Override  
    public ResolvableType getResolvableType() {  
        return ResolvableType.forClassWithGenerics(getClass(), ResolvableType.forInstantiationByType());  
    }  
}
```

### Tip

这不仅适用于 `ApplicationEvent`，而且适用于您作为事件发送的任何任意对象。

## 1.15.3. 方便地访问低级资源

为了最佳使用和理解应用程序上下文，您应该熟悉 Spring 的 `Resource` 抽象，如[Resources](#)中所述。

应用程序上下文是 `ResourceLoader`，可用于加载 `Resource` 个对象。`Resource` 本质上是 JDK `java.net.URL` 类的功能更丰富的版本。实际上，`Resource` 的实现在适当的地方包装了

`java.net.URL` 的实例。 `Resource` 可以从几乎任何位置以透明方式获取低级资源，包括从 `Classpath`、文件系统位置，可使用标准 URL 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，则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。

您可以配置部署到应用程序上下文中的 Bean，以实现特殊的回调接口 `ResourceLoaderAware`，以便在初始化时使用应用程序上下文本身作为 `ResourceLoader` 进行自动回调。您还可以公开 `Resource` 类型的属性，以用于访问静态资源。它们像其他任何属性一样注入其中。您可以将这些 `Resource` 属性指定为简单的 `String` 路径，并在部署 bean 时依靠特殊的 JavaBean `PropertyEditor` (由上下文自动注册) 将这些文本字符串转换为实际的 `Resource` 对象。

提供给 `ApplicationContext` 构造函数的一个或多个位置路径实际上是资源字符串，并且根据特定的上下文实现以简单的形式对其进行适当处理。例如 `ClassPathXmlApplicationContext` 将简单的位置路径视为 `Classpath` 位置。您也可以使用带有特殊前缀的位置路径(资源字符串)来强制从 `Classpath` 或 URL 中加载定义，而不管实际的上下文类型如何。

#### 1.15.4. Web 应用程序的便捷 ApplicationContext 实例化

您可以使用 `ContextLoader` 声明性地创建 `ApplicationContext` 实例。当然，您也可以使用 `ApplicationContext` 实现之一以编程方式创建 `ApplicationContext` 实例。

您可以使用 `ContextLoaderListener` 注册 `ApplicationContext`，如以下示例所示：

```
<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，那么侦听器将使用 `/WEB-`

`META-INF/applicationContext.xml` 作为默认值。当参数确实存在时，侦听器将使用 `sched` 定义的定界符(逗号，分号和空格)来分隔 `String`，并将这些值用作搜索应用程序上下文的位置。还支持 Ant 风格的路径模式。示例为 `/WEB-INF/*Context.xml` (对于名称以 `Context.xml` 结尾并且位于 `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 Spring ApplicationContext 部署为 Java EE RAR 文件

可以将 Spring `ApplicationContext` 部署为 RAR 文件，并将上下文及其所有必需的 Bean 类和库 JAR 封装在 Java EE RAR 部署单元中。这等效于引导独立的 `ApplicationContext` (仅托管在 Java EE 环境中)能够访问 Java EE 服务器功能。对于部署无头 WAR 文件的情况，RAR 部署是一种更自然的选择-实际上，这种 WAR 文件没有任何 HTTP 入口点，仅用于在 Java EE 环境中引导 Spring `ApplicationContext`。

对于不需要 HTTP 入口点而仅由消息端点和计划的作业组成的应用程序上下文，RAR 部署是理想的选择。在这样的上下文中，Bean 可以使用应用程序服务器资源，例如 JTA 事务 Management 器，与 JNDI 绑定的 JDBC `DataSource` 实例和 JMS `ConnectionFactory` 实例，还可以在平台的 JMX 服务器上注册-整个过程都通过 Spring 的标准事务 Management 以及 JNDI 和 JMX 支持工具进行。应用程序组件还可以通过 Spring 的 `TaskExecutor` 抽象与应用程序服务器的 JCA `WorkManager` 进行交互。

有关 RAR 部署中涉及的配置详细信息，请参见[SpringContextResourceAdapter](#)类的 javadoc。

对于将 Spring ApplicationContext 作为 Java EE RAR 文件的简单部署：

- 将所有应用程序类打包到 RAR 文件(这是具有不同文件 extensions 的标准 JAR 文件)中。将所有必需的库 JAR 添加到 RAR 归档文件的根目录中。添加一个 `META-INF/ra.xml` 部署 Descriptors(如[SpringContextResourceAdapter](#) 的 javadoc 所示)和相应的 Spring XML bean 定义文件(通常为 `META-INF/applicationContext.xml`)。

- 将生成的 RAR 文件拖放到应用程序服务器的部署目录中。

### iNote

此类 RAR 部署单元通常是独立的。它们不会将组件暴露给外界，甚至不会暴露给同一应用程序的其他模块。与基于 RAR 的 `ApplicationContext` 的交互通常通过与其他模块共享的 JMS 目标进行。例如，基于 RAR 的 `ApplicationContext` 还可以计划一些作业或对文件系统(或类似文件)中的新文件做出反应。如果需要允许从外部进行同步访问，则可以(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。

## 1.16. bean 工厂

`BeanFactory` API 为 Spring 的 IoC 功能提供了基础。它的特定 Contract 主要用于与 Spring 的其他部分以及相关的第三方框架集成，并且其 `DefaultListableBeanFactory` 实现是更高级别 `GenericApplicationContext` 容器中的关键委托。

`BeanFactory` 和相关接口(例如 `BeanFactoryAware`，`InitializingBean`，`DisposableBean`)是其他框架组件的重要集成点。通过不需要任何 Comments 甚至反射，它们可以在容器及其组件之间进行非常有效的交互。应用程序级 Bean 可以使用相同的回调接口，但通常更喜欢通过 Comments 或通过程序配置进行声明式依赖注入。

请注意，核心 `BeanFactory` API 级别及其 `DefaultListableBeanFactory` 实现不对配置格式或要使用的任何组件 Comments 进行假设。所有这些风味都是通过 extensions(例如 `XmlBeanDefinitionReader` 和 `AutowiredAnnotationBeanPostProcessor`)引入的，并以共享的 `BeanDefinition` 对象作为核心元数据表示形式进行操作。这就是使 Spring 的容器如此灵活和可扩展的本质。

### 1.16.1. BeanFactory 或 ApplicationContext ?

本节说明 `BeanFactory` 和 `ApplicationContext` 容器级别之间的区别以及对引导的影响。

除非有充分的理由，否则应使用 `ApplicationContext`，除非 `GenericApplicationContext` 及其子类 `AnnotationConfigApplicationContext` 作为自定义引导的常见实现，否则应使用它们。

对于所有常见目的，这些都是 Spring 核心容器的主要入口点：加载配置文件，触发 Classpath 扫描，以编程方式注册 Bean 定义和带 Comments 的类，以及(从 5.0 版本开始)注册功能性 Bean 定义。

因为 `ApplicationContext` 包含 `BeanFactory` 的所有功能，所以通常建议在纯 `BeanFactory` 上使用，除非需要完全控制 Bean 处理的情况。在 `ApplicationContext` (例如 `GenericApplicationContext` 实现)内，按惯例(即，按 Bean 名称或 Bean 类型(尤其是后处理器))检测到几种 Bean，而普通的 `DefaultListableBeanFactory` 则与任何特殊的 Bean 无关。

对于许多扩展的容器功能(例如 Comments 处理和 AOP 代理)，[BeanPostProcessor 扩展点](#)是必不可少的。如果仅使用普通 `DefaultListableBeanFactory`，则默认情况下不会检测到此类后处理器并将其激活。这种情况可能令人困惑，因为您的 bean 配置实际上并没有错。而是在这种情况下，需要通过其他设置完全引导容器。

下表列出了 `BeanFactory` 和 `ApplicationContext` 接口和实现提供的功能。

表 9. 功能列表

Feature	<code>BeanFactory</code>	<code>ApplicationContext</code>
Bean instantiation/wiring	Yes	Yes
集成生命周期 Management	No	Yes
自动 <code>BeanPostProcessor</code> 注册	No	Yes

Feature	BeanFactory	ApplicationContext
自动 BeanFactoryPostProcessor 注册	No	Yes
方便的 MessageSource 访问权限(用于内部化)	No	Yes
内置的 ApplicationEvent 发布机制	No	Yes

要使用 DefaultListableBeanFactory 显式注册 bean 后处理器，您需要以编程方式调用 addBeanPostProcessor ，如以下示例所示：

```
DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
// populate the factory with bean definitions

// now register any needed BeanPostProcessor instances
factory.addBeanPostProcessor(new AutowiredAnnotationBeanPostProcessor());
factory.addBeanPostProcessor(new MyBeanPostProcessor());

// now start using the factory
```

要将 BeanFactoryPostProcessor 应用于普通 DefaultListableBeanFactory ，需要调用其 postProcessBeanFactory 方法，如以下示例所示：

```
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，显式的注册步骤都是不方便的，这就是为什么在 Spring 支持的应用程序中，各种

`ApplicationContext` 变体优于普通 `DefaultListableBeanFactory` 的原因，尤其是在典型企业设置中依靠 `BeanFactoryPostProcessor` 和 `BeanPostProcessor` 实例扩展容器功能时。

### iNote

`AnnotationConfigApplicationContext` 已注册了所有通用 `Comments` 后处理器，并且可以通过诸如 `@EnableTransactionManagement` 之类的配置 `Comments` 在幕后引入其他处理器。在 Spring 基于 `Comments` 的配置模型的抽象级别上，`bean` 后处理器的概念仅是内部容器详细信息。

## 2. Resources

本章介绍了 Spring 如何处理资源以及如何在 Spring 中使用资源。它包括以下主题：

- [Introduction](#)
- [资源接口](#)
- [内置资源实现](#)
- [The ResourceLoader](#)
- [ResourceLoaderAware 接口](#)
- [资源依赖](#)
- [应用程序上下文和资源路径](#)

### 2.1. Introduction

不幸的是，Java 的标准 `java.net.URL` 类和用于各种 URL 前缀的标准处理程序不足以满足所有对低级资源的访问。例如，没有标准化的 `URL` 实现可用于访问需要从 Classpath 或相对于 `ServletContext` 获得的资源。尽管可以为专用的 `URL` 前缀注册新的处理程序(类似于现有的针对

`http:` 的前缀的处理程序), 但这通常相当复杂, 并且 `URL` 接口仍然缺少某些理想的功能, 例如用于检查是否存在 `URL` 的方法。指向的资源。

## 2.2. 资源接口

Spring 的 `Resource` 接口旨在成为一种功能更强大的接口, 用于抽象化对低级资源的访问。以下清单显示了 `Resource` 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();  
}
```

如 `Resource` 接口的定义所示, 它扩展了 `InputStreamSource` 接口。以下清单显示了 `InputStreamSource` 接口的定义:

```
public interface InputStreamSource {  
    InputStream getInputStream() throws IOException;  
}
```

`Resource` 界面中一些最重要的方法是:

- `getInputStream()`: 找到并打开资源, 返回 `InputStream` 以从资源中读取。预计每次调用都返回一个新的 `InputStream`。呼叫者有责任关闭流。
- `exists()`: 返回 `boolean`, 指示此资源是否实际以物理形式存在。

- `isOpen()` : 返回一个 `boolean` , 指示此资源是否代表具有打开流的句柄。如果为 `true` , 则不能多次读取 `InputStream` , 必须仅读取一次, 然后关闭 `InputStream` 以避免资源泄漏。对于所有常规资源实现, 返回 `false` , 但 `InputStreamResource` 除外。
- `getDescription()` : 返回此资源的描述, 用于在处理资源时输出错误。这通常是标准文件名或资源的实际 URL。

其他方法可让您获得代表资源的实际 `URL` 或 `File` 对象(如果基础实现兼容并且支持该功能)。

当需要资源时, Spring 本身广泛使用 `Resource` 抽象作为许多方法签名中的参数类型。某些 Spring API 中的其他方法(例如, 各种 `ApplicationContext` 实现的构造函数)采用 `String` , 其以无装饰或简单的形式用于创建适合于该上下文实现的 `Resource` , 或者通过 `String` 路径上的特殊前缀, 让调用者指定必须创建并使用特定的 `Resource` 实现。

虽然 `Resource` 接口在 Spring 和 Spring 上经常使用, 但在您自己的代码中单独用作通用实用工具类来访问资源实际上非常有用, 即使您的代码不知道或不关心其他任何代码 Spring 的一部分。虽然这会将您的代码耦合到 Spring, 但实际上仅将其耦合到这套 Util 小类, 它们可以更强大地替代 `URL` , 并且可以认为等同于您将为此使用的任何其他库。

#### Note

`Resource` 抽象不能代替功能。它尽可能地包装它。例如, 一个 `UrlResource` 包裹一个 `URL`, 并使用包裹的 `URL` 来完成其工作。

## 2.3. 内置资源实现

Spring 包含以下 `Resource` 实现:

- [UrlResource](#)

- [ClassPathResource](#)
- [FileSystemResource](#)
- [ServletContextResource](#)
- [InputStreamResource](#)
- [ByteArrayResource](#)

### 2.3.1. UrlResource

`UrlResource` 包装 `java.net.URL`， 并且可用于访问通常可通过 URL 访问的任何对象，例如文件，HTTP 目标，FTP 目标等。所有 URL 都具有标准化的 `String` 表示形式，以便使用适当的标准化前缀来指示另一种 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。

Java 代码通过显式使用 `UrlResource` 构造函数来创建 `UrlResource`，但是在调用带有 `String` 自变量表示路径的 API 方法时，通常会隐式创建 `UrlResource`。对于后一种情况，JavaBeans `PropertyEditor` 最终决定要创建哪种类型的 `Resource`。如果路径字符串包含众所周知的前缀（例如 `classpath:`），则会为该前缀创建一个适当的专用 `Resource`。但是，如果无法识别前缀，则假定该字符串是标准 URL 字符串并创建 `UrlResource`。

### 2.3.2. ClassPathResource

此类表示应从 Classpath 获取的资源。它使用线程上下文类加载器，给定的类加载器或给定的类来加载资源。

如果 Classpath 资源驻留在文件系统中，而不是驻留在 jar 中并且尚未（通过 servlet 引擎或任何环境将其扩展到）文件系统的 Classpath 资源不驻留在文件系统中，则此 `Resource` 实现支持以 `java.io.File` 解析。为了解决这个问题，各种 `Resource` 实现始终支持将分辨率作为

`java.net.URL`。

Java 代码通过显式使用 `ClassPathResource` 构造函数来创建 `ClassPathResource`，但是在调用带有 `String` 自变量表示路径的 API 方法时，通常会隐式创建 `ClassPathResource`。对于后一种情况，JavaBeans `PropertyEditor` 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

这是 `java.io.File` 和 `java.nio.file.Path` 句柄的 `Resource` 实现。它支持 `File` 和 `URL` 分辨率。

### 2.3.4. ServletContextResource

这是 `ServletContext` 资源的 `Resource` 实现，用于解释相关 Web 应用程序根目录中的相对路径。

它始终支持流访问和 URL 访问，但仅在扩展 Web 应用程序 Files 并且资源实际位于文件系统上时才允许 `java.io.File` 访问。它是在文件系统上扩展还是在文件系统上进行扩展，或者直接从 JAR 或其他类似数据库(可以想到的)访问，实际上取决于 Servlet 容器。

### 2.3.5. InputStreamResource

`InputStreamResource` 是给定 `InputStream` 的 `Resource` 实现。仅当没有特定的 `Resource` 实现适用时才应使用它。特别是，在可能的情况下，最好使用 `ByteArrayResource` 或任何基于文件的 `Resource` 实现。

与其他 `Resource` 实现相反，这是一个已经打开的资源的 `Descriptors`。因此，它从 `isOpen()` 返回 `true`。如果您需要将资源 `Descriptors` 保留在某个地方，或者需要多次读取流，请不要使用它。

## 2.3.6. ByteArrayResource

这是给定字节数组的 `Resource` 实现。它为给定的字节数组创建一个 `ByteArrayInputStream`。

这对于从任何给定的字节数组加载内容而无需使用一次性 `InputStreamResource` 很有用。

## 2.4. 资源加载器

`ResourceLoader` 接口旨在由可以返回(即加载) `Resource` 实例的对象实现。以下清单显示了

`ResourceLoader` 接口定义：

```
public interface ResourceLoader {  
    Resource getResource(String location);  
}
```

所有应用程序上下文都实现 `ResourceLoader` 接口。因此，所有应用程序上下文都可用于获取 `Resource` 个实例。

当您在特定的应用程序上下文上调用 `getResource()` 时，并且指定的位置路径没有特定的前缀时，您将获得适合该特定应用程序上下文的 `Resource` 类型。例如，假设针对

`ClassPathXmlApplicationContext` 实例执行了以下代码段：

```
Resource template = ctx.getResource("some/resource/path/myTemplate.txt");
```

针对 `ClassPathXmlApplicationContext`，该代码返回 `ClassPathResource`。如果对 `FileSystemXmlApplicationContext` 实例执行了相同的方法，它将返回 `FileSystemResource`。对于 `WebApplicationContext`，它将返回 `ServletContextResource`。类似地，它将为每个上下文返回适当的对象。

结果，您可以以适合特定应用程序上下文的方式加载资源。

另一方面，您也可以通过指定特殊的 `classpath:` 前缀来强制使用 `ClassPathResource`，而与应

用程序上下文类型无关，如下例所示：

```
Resource template = ctx.getResource("classpath:some/resource/path/myTemplate.txt");
```

同样，您可以通过指定任何标准 `java.net.URL` 前缀来强制使用 `UrlResource`。以下一对示例使用 `file` 和 `http` 前缀：

```
Resource template = ctx.getResource("file:///some/resource/path/myTemplate.txt");
```

```
Resource template = ctx.getResource("http://myhost.com/resource/path/myTemplate.txt");
```

下表总结了将 `String` 个对象转换为 `Resource` 个对象的策略：

表 10. 资源字符串

Prefix	Example	Explanation
classpath:	<code>classpath:com/myapp/config.xml</code>	从 Classpath 加载。
file:	<code>file:///data/config.xml</code>	从文件系统作为 <code>URL</code> 加载。 另请参见 <a href="#">FileSystemResource</a> <a href="#">Caveats</a> 。
http:	<code>http://myserver/logo.png</code>	载入为 <code>URL</code> 。
(none)	<code>/data/config.xml</code>	取决于基础的 <code>ApplicationContext</code> 。

## 2.5. ResourceLoaderAware 接口

`ResourceLoaderAware` 接口是一个特殊的标记接口，用于标识期望提供 `ResourceLoader` 参考

的对象。以下清单显示了 `ResourceLoaderAware` 接口的定义：

```
public interface ResourceLoaderAware {  
    void setResourceLoader(ResourceLoader resourceLoader);  
}
```

当一个类实现 `ResourceLoaderAware` 并部署到应用程序上下文中(作为 Spring 托管 bean)时，该类被应用程序上下文识别为 `ResourceLoaderAware`。然后，应用程序上下文调用 `setResourceLoader(ResourceLoader)`，将其自身作为参数(请记住，Spring 中的所有应用程序上下文都实现 `ResourceLoader` 接口)。

由于 `ApplicationContext` 是 `ResourceLoader`，因此 bean 也可以实现 `ApplicationContextAware` 接口并直接使用提供的应用程序上下文来加载资源。但是，通常，如果需要的话，最好使用专用的 `ResourceLoader` 接口。该代码将仅耦合到资源加载接口(可以视为 Util 接口)，而不耦合到整个 Spring `ApplicationContext` 接口。

从 Spring 2.5 开始，您可以依靠 `ResourceLoader` 的自动装配来替代实现 `ResourceLoaderAware` 接口。现在，“传统” `constructor` 和 `byType` 自动装配模式(如 [Autowiring Collaborators](#) 中所述)能够分别为构造函数参数或 `setter` 方法参数提供 `ResourceLoader` 类型的依赖项。为了获得更大的灵 Active(包括自动装配字段和多个参数方法的能力)，请考虑使用基于 `Comments` 的自动装配功能。在这种情况下，只要有问题的字段，构造函数或方法带有 `@Autowired` `Comments`，就将 `ResourceLoader` 自动连接到期望 `ResourceLoader` 类型的字段，构造函数参数或方法参数中。有关更多信息，请参见[Using @Autowired](#)。

## 2.6. 资源依赖

如果 Bean 本身将通过某种动态过程来确定并提供资源路径，那么对于 Bean 来说，使用 `ResourceLoader` 接口加载资源可能是有意义的。例如，考虑加载某种模板，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，则有必要完全消除对 `ResourceLoader` 接口的使用，让 Bean 公开所需的 `Resource` 属性，并期望将其注入其中。

然后注入这些属性很简单，因为所有应用程序上下文都注册并使用了特殊的 JavaBeans `PropertyEditor`，可以将 `String` 路径转换为 `Resource` 对象。因此，如果 `myBean` 具有类型为 `Resource` 的模板属性，则可以为该资源配置一个简单的字符串，如下所示：

```
<bean id="myBean" class="...">>
    <property name="template" value="some/resource/path/myTemplate.txt"/>
</bean>
```

请注意，资源路径没有前缀。因此，由于应用程序上下文本身将用作 `ResourceLoader`，因此根据上下文的确切类型，资源本身是通过 `ClassPathResource`、`FileSystemResource` 或 `ServletContextResource` 加载的。

如果需要强制使用特定的 `Resource` 类型，则可以使用前缀。以下两个示例显示了如何强制 `ClassPathResource` 和 `UrlResource`（后者用于访问文件系统文件）：

```
<property name="template" value="classpath:some/resource/path/myTemplate.txt">
```

```
<property name="template" value="file:///some/resource/path/myTemplate.txt"/>
```

## 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. 构造应用程序上下文

应用程序上下文构造函数(针对特定的应用程序上下文类型)通常采用字符串或字符串数组作为资源

的位置路径，例如构成上下文定义的 XML 文件。

当这样的位置路径没有前缀时，从该路径构建并用于加载 Bean 定义的特定 `Resource` 类型取决于特定的应用程序上下文，并且适用于特定的应用程序上下文。例如，考虑以下示例，该示例创建一个 `ClassPathXmlApplicationContext`：

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("conf/appContext.xml");
```

Bean 定义是从 Classpath 加载的，因为使用了 `ClassPathResource`。但是，请考虑以下示例，该示例创建 `FileSystemXmlApplicationContext`：

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("conf/appContext.xml");
```

现在，bean 定义是从文件系统位置(在这种情况下，是相对于当前工作目录)加载的。

请注意，在位置路径上使用特殊的 Classpath 前缀或标准 URL 前缀会覆盖为加载定义而创建的默认类型 `Resource`。考虑以下示例：

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("classpath:conf/appContext.xml");
```

使用 `FileSystemXmlApplicationContext` 从 Classpath 中加载 bean 定义。但是，它仍然是 `FileSystemXmlApplicationContext`。如果随后将其用作 `ResourceLoader`，则所有未前缀的路径仍将视为文件系统路径。

## 构造 `ClassPathXmlApplicationContext` 实例—快捷方式

`ClassPathXmlApplicationContext` 公开了许多构造函数以启用方便的实例化。基本思想是，您只能提供一个字符串数组，该字符串数组仅包含 XML 文件本身的文件名(不包含前导路径信息)，还提供 `Class`。`ClassPathXmlApplicationContext` 然后从提供的类中导出路径信息。

请考虑以下目录布局：

```
com/
  foo/
    services.xml
    daos.xml
  MessengerService.class
```

以下示例显示了如何实例化由名为 `services.xml` 和 `daos.xml` (位于 Classpath 中) 的文件中定义的 bean 组成的 `ClassPathXmlApplicationContext` 实例：

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

有关各种构造函数的详细信息，请参见[ClassPathXmlApplicationContext](#) javadoc。

## 2.7.2. 应用程序上下文构造函数资源路径中的通配符

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(如前所述)，每个路径都具有到目标 `Resource` 的一对一 Map，或者可以包含特殊的“`classpath*:`”前缀或内部 Ant。样式的正则表达式(通过使用 Spring 的 `PathMatcher` Util 进行匹配)。后者都是有效的通配符。

这种机制的一种用途是当您需要进行组件样式的应用程序组装时。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 `classpath*:` 的相同路径创建最终应用程序上下文时，所有组件片段都会被自动拾取。

请注意，此通配符特定于在应用程序上下文构造函数中使用资源路径(或在直接使用 `PathMatcher` Util 类层次结构时使用)，并在构造时解决。它与 `Resource` 类型本身无关。您不能使用 `classpath*:` 前缀构造实际的 `Resource`，因为资源一次仅指向一个资源。

### Ant-style Patterns

路径位置可以包含 Ant 样式的模式，如以下示例所示：

```
/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 Ant 样式的模式时，解析程序将遵循更复杂的过程来尝试解析通配符。它为到达最后一个非通配符段的路径生成一个 `Resource`，并从中获取 URL。如果此 URL 不是 `jar:` URL 或特定于容器的变体(例如 WebLogic 中的 `zip:`，WebSphere 中的 `wsjar` 等)，则从中获得 `java.io.File` 并将其用于遍历文件系统来解析通配符。对于 jar URL，解析器可以从中获取 `java.net.JarURLConnection` 或手动解析 jar URL，然后遍历 jar 文件的内容来解析通配符。

## 对便携性的影响

如果指定的路径已经是一个文件 URL(由于基 `ResourceLoader` 是一个文件系统，则是隐式的，或者是明确的)，则保证通配符可以完全可移植的方式工作。

如果指定的路径是 Classpath 位置，则解析器必须通过进行 `ClassLoader.getResource()` 调用来获取最后一个非通配符路径段 URL。由于这只是路径的一个节点(而不是末尾的文件)，因此实际上(在 `ClassLoader` javadoc 中)未定义到底返回了哪种 URL。实际上，它始终是代表目录的 `java.io.File` (Classpath 资源解析为文件系统位置)或某种 jar URL(Classpath 资源解析为 jar 位置)。尽管如此，此操作仍存在可移植性问题。

如果为最后一个非通配符段获取了 jar URL，则解析程序必须能够从中获取

`java.net.JarURLConnection` 或手动解析 jar URL，以便能够遍历 jar 的内容并解析通配符。这在大多数环境中确实有效，但在其他环境中则无效，因此我们强烈建议您在依赖特定环境之前，对来自 jars 的资源的通配符解析进行彻底测试。

## Classpath\*：前缀

构造基于 XML 的应用程序上下文时，位置字符串可以使用特殊的 `classpath*:` 前缀，如以下示例所示：

```
ApplicationContext ctx =  
    new ClassPathXmlApplicationContext("classpath*:conf/appContext.xml");
```

这个特殊的前缀指定必须获取与给定名称匹配的所有 Classpath 资源(内部，这实际上是通过调用

`ClassLoader.getResources(...)` 发生的), 然后合并以形成最终的应用程序上下文定义。

### iNote

通配符 Classpath 依赖于基础类加载器的 `getResources()` 方法。由于当今大多数应用程序服务器提供其自己的类加载器实现, 因此行为可能有所不同, 尤其是在处理 jar 文件时。检查 `classpath*` 是否有效的简单测试是使用类加载器从 Classpath

```
getClass().getClassLoader().getResources("<someFileInsideTheJar>")
```

 的 jar 中加载文件。尝试对具有相同名称但位于两个不同位置的文件进行此测试。如果返回了不合适的结果, 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。

您还可以在其余的位置路径(例如 `classpath*:META-INF/*-beans.xml`)中将 `classpath*:` 前缀与 `PathMatcher` 模式结合使用。在这种情况下, 解析策略非常简单: 在最后一个非通配符路径段上使用 `ClassLoader.getResources()` 调用, 以获取类加载器层次结构中的所有匹配资源, 然后从每个资源中分离出与上述相同的 `PathMatcher` 解析策略: 用于通配符子路径。

### 与通配符有关的其他说明

请注意, 当 `classpath*:` 与 Ant 样式的模式结合使用时, 除非模式文件实际存在于目标文件系统中, 否则至少在模式启动之前, 它至少与一个根目录可靠地配合使用。这意味着诸如 `classpath*:*.*xml` 之类的模式可能不会从 jar 文件的根目录检索文件, 而只会从扩展目录的根目录检索文件。

Spring 检索 Classpath 条目的能力源于 JDK 的 `ClassLoader.getResources()` 方法, 该方法仅返回文件系统位置中的空字符串(指示可能要搜索的根目录)。Spring 也会在 jar 文件中评估 `URLClassLoader` 运行时配置和 `java.class.path` 清单, 但这不能保证会导致可移植行为。

### iNote

扫描 Classpath 包需要在 Classpath 中存在相应的目录条目。使用 Ant 构建 JAR 时, 请勿激活 JAR 任务的仅文件开关。另外, 在某些环境中, Classpath 目录可能不会基于安全策略公开-例如, 在 JDK 1.7.0\_45 及更高版本上的独立应用程序(要求在清单中设置“受信任的库” .请参阅<http://stackoverflow.com/questions/19394570/java-jre-7u45-breaks-classloader-getresources>)。

在 JDK 9 的模块路径(Jigsaw)上, Spring 的 Classpath 扫描通常可以按预期进行。强烈建议在此处将资源放入专用目录中, 以避免在搜索 jar 文件根级别时遇到上述可移植性问题。

如果要搜索的根包在多个 Classpath 位置中可用, 则不能保证具有 `classpath:` 资源的 Ant 样式模式会找到匹配的资源。考虑以下资源位置示例:

```
com/mycompany/package1/service-context.xml
```

现在考虑某人可能用来尝试找到该文件的 Ant 样式的路径:

```
classpath:com/mycompany/**/service-context.xml
```

这样的资源可能只在一个位置, 但是当使用诸如前面示例的路径尝试解析它时, 解析器将根据 `getResource("com/mycompany")`; 返回的(第一个)URL 进行处理。如果此基本包节点存在于多个类加载器位置, 则实际的最终资源可能不存在。因此, 在这种情况下, 您应该更喜欢使用具有相同 Ant 样式模式的 `classpath*:`, 该模式将搜索包含根包的所有 Classpath 位置。

### 2.7.3. FileSystemResource 警告

未连接到 `FileSystemApplicationContext` 的 `FileSystemResource` (也就是说, 当 `FileSystemApplicationContext` 不是实际的 `ResourceLoader` 时)将按您期望的那样处理绝对路径和相对路径。相对路径是相对于当前工作目录的, 而绝对路径是相对于文件系统的根的。

但是, 出于向后兼容性(历史)的原因, 当 `FileSystemApplicationContext` 为 `ResourceLoader` 时, 情况会有所变化。`FileSystemApplicationContext` 强制所有附加的

`FileSystemResource` 实例将所有位置路径都视为相对位置，无论它们是否以前斜杠开头。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");
```

以下示例也是等效的(即使它们有所不同也有意义，因为一种情况是相对的，另一种情况是绝对的)：

```
FileSystemXmlApplicationContext ctx = ...;
ctx.getResource("some/resource/path/myTemplate.txt");
```

```
FileSystemXmlApplicationContext ctx = ...;
ctx.getResource("/some/resource/path/myTemplate.txt");
```

实际上，如果您需要真实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，而应通过使用 `file:` URL 前缀来强制使用 `UrlResource`。以下示例显示了如何执行此操作：

```
// 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");
```

## 3. 验证，数据绑定和类型转换

考虑将验证作为业务逻辑是有利有弊，Spring 提供了一种验证(和数据绑定)设计，但并不排除其中任何一个。具体来说，验证不应与网络层绑定，并且应该易于本地化，并且应该可以插入任何可用的验证器。考虑到这些问题，Spring 提供了一个 `Validator` 接口，该接口既基本又可以在应用程序的每一层使用。

数据绑定对于使用户 Importing 动态绑定到应用程序(或用于处理用户 Importing 的任何对象)域模型很有用。 Spring 提供了恰当地命名为 `DataBinder` 的名称。 `Validator` 和 `DataBinder` 组成了 `validation` 程序包，该程序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，在很多地方都使用过。但是，您可能不需要直接使用 `BeanWrapper`。但是，由于这是参考文档，因此我们认为可能需要进行一些解释。我们将在本章中解释 `BeanWrapper`，因为如果您将要使用它，则在尝试将数据绑定到对象时最有可能使用它。

Spring 的 `DataBinder` 和较低级别的 `BeanWrapper` 都使用 `PropertyEditorSupport` 实现来解析和格式化属性值。`PropertyEditor` 和 `PropertyEditorSupport` 接口是 JavaBeans 规范的一部分，本章还将对此进行说明。Spring 3 引入了 `core.convert` 软件包，该软件包提供了常规的类型转换工具，以及用于格式化 UI 字段值的高级“格式”软件包。您可以将这些软件包用作 `PropertyEditorSupport` 实现的更简单替代方案。本章还将对它们进行讨论。

## JSR-303/JSR-349 Bean 验证

从 4.0 版本开始，Spring 框架支持 Bean 验证 1.0(JSR-303)和 Bean 验证 1.1(JSR-349)，以支持设置并使其适应 Spring 的 `Validator` 接口。

应用程序可以选择一次全局启用 Bean 验证(如[Spring Validation](#)中所述)，并将其专用于所有验证需求。

应用程序还可以为每个 `DataBinder` 实例注册其他 Spring `Validator` 实例，如[配置一个 DataBinder](#)中所述。这对于在不使用 Comments 的情况下插入验证逻辑可能很有用。

### 3.1. 使用 Spring 的 Validator 接口进行验证

Spring 具有 `Validator` 接口，可用于验证对象。`Validator` 接口通过使用 `Errors` 对象工作，因此验证器可以在验证时向 `Errors` 对象报告验证失败。

考虑以下小型数据对象的示例：

```
public class Person {  
  
    private String name;  
    private int age;  
  
    // the usual getters and setters...  
}
```

下一个示例通过实现 `org.springframework.validation.Validator` 接口的以下两种方法来提供 `Person` 类的验证行为：

- `supports(Class)`：此 `Validator` 是否可以验证提供的 `Class` 的实例？
- `validate(Object, org.springframework.validation.Errors)`：验证给定的对象，并在验证错误的情况下，将其注册到给定的 `Errors` 对象。

实现 `Validator` 非常简单，尤其是当您知道 Spring 框架还提供的 `ValidationUtils` helper 类时。以下示例为 `Person` 实例实现 `Validator`：

```
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(如果它是 `null` 或空字符串)。看看[ValidationUtils javadoc](#)，看看它提供了除先前显示的示例之外的功能。

虽然可以实现单个 `Validator` 类来验证丰富对象中的每个嵌套对象，但是最好将每个嵌套类的验证逻辑封装在自己的 `Validator` 实现中。一个“丰富”对象的简单示例是一个 `Customer`，它由两个 `String` 属性(名字和第二个名称)和一个复杂的 `Address` 对象组成。`Address` 对象可以独立于 `Customer` 对象使用，因此已实现了不同的 `AddressValidator`。如果希望 `CustomerValidator` 重用 `AddressValidator` 类中包含的逻辑而不求助于复制和粘贴，则可以在 `CustomerValidator` 中依赖注入或实例化 `AddressValidator`，如以下示例所示：

```
public class CustomerValidator implements Validator {  
  
    private final Validator addressValidator;  
  
    public CustomerValidator(Validator addressValidator) {  
        if (addressValidator == null) {  
            throw new IllegalArgumentException("The supplied [Validator] is " +  
                "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());  
        } finally {  
            errors.popNestedPath();  
        }  
    }  
}
```

验证错误会报告给传递给验证器的 `Errors` 对象。对于 Spring Web MVC，可以使用

`<spring:bind/>` 标签检查错误消息，但是也可以自己检查 `Errors` 对象。有关其提供的方法的更多信息，请参见 [javadoc](#)。

## 3.2. 将代码解析为错误消息

我们介绍了数据绑定和验证。本节介绍与验证错误相对应的输出消息。在preceding section所示的示例中，我们拒绝了`name` 和 `age` 字段。如果要使用`MessageSource` 输出错误消息，可以使用拒绝字段时提供的错误代码(在这种情况下为“名称”和“年龄”)进行输出。当您从`Errors` 接口调用(直接或间接使用`ValidationUtils` 类) `rejectValue` 或其他 `reject` 方法时，基础实现不仅会注册您传入的代码，还会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，使用`DefaultMessageCodesResolver`，例如，它不仅使用您提供的代码注册一条消息，而且还注册包含您传递给`reject`方法的字段名称的消息。因此，如果您使用`rejectValue("age", "too.darn.old")` 拒绝字段，除了`too.darn.old` 代码，Spring 还会注册`too.darn.old.age` 和 `too.darn.old.age.int`(第一个包含字段名称，第二个包含字段类型)。这样做是为了方便开发人员在定位错误消息时提供帮助。

有关`MessageCodesResolver` 和默认策略的更多信息，可以分别在[MessageCodesResolver](#)和[DefaultMessageCodesResolver](#)的 javadoc 中找到。

## 3.3. Bean 操作和 BeanWrapper

`org.springframework.beans` 软件包遵循 JavaBeans 标准。JavaBean 是具有默认无参数构造函数的类，并且遵循命名约定，其中(例如)名为`bingoMadness` 的属性将具有`setter`方法`setBingoMadness(..)` 和`getter`方法`getBingoMadness()`。有关 JavaBean 和规范的更多信息，请参见[javabeans](#)。

Bean 包中的一个非常重要的类是`BeanWrapper` 接口及其相应的实现(`BeanWrapperImpl`)。正如从 Javadoc 引用的那样，`BeanWrapper` 提供了以下功能：设置和获取属性值(单独或批量)，获取属性 Descriptors 以及查询属性以确定它们是否可读或可写。此外，`BeanWrapper` 还支持嵌套属性，从而可以将子属性上的属性设置为无限深度。`BeanWrapper` 还支持添加标准 JavaBean

`PropertyChangeListeners` 和 `VetoableChangeListeners` 的能力，而无需在目标类中支持代码。最后但并非最不重要的一点是，`BeanWrapper` 支持设置索引属性。`BeanWrapper` 通常不直接由应用程序代码使用，而由 `DataBinder` 和 `BeanFactory` 使用。

`BeanWrapper` 的工作方式部分地由其名称表示：它包装一个 bean 以对该 bean 执行操作，例如设置和检索属性。

### 3.3.1. 设置和获取基本和嵌套属性

设置和获取属性的方法是使用 `setProperty`，`setProperties`，  
`getProperty` 和 `getProperties` 方法，这些方法带有一些重载的变体。Springs javadoc 更详细地描述了它们。JavaBeans 规范具有用于指示对象属性的约定。下表显示了这些约定的一些示例：

表 11. 属性示例

Expression	Explanation
<code>name</code>	表示与 <code>name</code> 或 <code>isName()</code> 和 <code>setName(..)</code> 方法相对应的属性 <code>name</code> 。
<code>account.name</code>	表示与 <code>getAccount().setName()</code> 或 <code>getAccount().getName()</code> 方法相对应的属性 <code>account</code> 的嵌套属性 <code>name</code> 。
<code>account[2]</code>	指示索引属性 <code>account</code> 的 * third * 元素。索引属性可以是 <code>array</code> ， <code>list</code> 或其他自然排序的集合。

Expression	Explanation
<code>account[COMPANYNAME]</code>	表示由 <code>account</code> 属性的 <code>Map</code> 键索引的 <code>COMPANYNAME</code> 条目的值。

(如果您不打算直接使用 `BeanWrapper`，那么下一部分对您而言并不重要.如果仅使用 `DataBinder` 和 `BeanFactory` 及其默认实现，则应跳到[关于 PropertyEditors 的部分。](#) )

以下两个示例类使用 `BeanWrapper` 来获取和设置属性：

```
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;
    }
}
```

```
public class Employee {

    private String name;

    private float salary;

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public float getSalary() {
```

```

        return salary;
    }

    public void setSalary(float salary) {
        this.salary = salary;
    }
}

```

以下代码段显示了如何检索和操纵实例化的 `Companies` 和 `Employees` 的某些属性的一些示例：

```

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。以不同于对象本身的方式表示属性可能很方便。例如，`Date` 可以以人类可读的方式表示(如 `String` : `'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(或者更好的是，转换以人类可读的形式 Importing 的任何日期)回到 `Date` 个对象)。通过注册 `java.beans.PropertyEditor` 类型的自定义编辑器可以实现此行为。在 `BeanWrapper` 上或在特定的 IoC 容器中注册自定义编辑器(如上一章所述)，使它具有如何将属性转换为所需类型的知识。有关 `PropertyEditor` 的更多信息，请参见 [Oracle 的 java.beans 包的 javadoc](#)。

在 Spring 中使用属性编辑的几个示例：

- 通过使用 `PropertyEditor` 实现在 bean 上设置属性。当使用 `java.lang.String` 作为在 XML 文件中声明的某些 bean 的属性值时，Spring(如果相应属性的设置器具有 `Class` 参数) 将

使用 `ClassEditor` 尝试将参数解析为 `Class` 对象。

- 在 Spring 的 MVC 框架中，通过使用各种 `PropertyEditor` 实现来解析 HTTP 请求参数，您可以在 `CommandController` 的所有子类中手动绑定这些实现。

Spring 具有许多内置的 `PropertyEditor` 实现，以简化生活。它们都位于

`org.springframework.beans.propertyeditors` 包中。默认情况下，大多数(但不是全部，如下表所示)由 `BeanWrapperImpl` 注册。如果可以通过某种方式配置属性编辑器，则仍可以注册自己的变体以覆盖默认变体。下表描述了 Spring 提供的各种 `PropertyEditor` 实现：

表 12. 内置 `PropertyEditor` 实现

Class	Explanation
<code>ByteArrayPropertyEditor</code>	字节数组的编辑器。将字符串转换为其相应的字节表示形式。默认情况下由 <code>BeanWrapperImpl</code> 注册。
<code>ClassEditor</code>	将表示类的字符串解析为实际类，反之亦然。如果找不到类，则会抛出 <code>IllegalArgumentException</code> 。默认情况下，由 <code>BeanWrapperImpl</code> 注册。
<code>CustomBooleanEditor</code>	<code>Boolean</code> 属性的可定制属性编辑器。默认情况下，由 <code>BeanWrapperImpl</code> 注册，但是可以通过将其自定义实例注册为自定义编辑器来覆盖。
<code>CustomCollectionEditor</code>	集合的属性编辑器，可将任何源 <code>Collection</code> 转换为给定目标 <code>Collection</code> 类型。

Class	Explanation
<code>CustomDateEditor</code>	<p><code>java.util.Date</code> 的可定制属性编辑器，支持定制 <code>DateFormat</code>。默认未注册。必须根据需要以适当的格式进行用户注册。</p>
<code>CustomNumberEditor</code>	<p>任何 <code>Number</code> 子类(例如 <code>Integer</code>，<code>Long</code>，<code>Float</code> 或 <code>Double</code>)的可定制属性编辑器。默认情况下，由 <code>BeanWrapperImpl</code> 注册，但是可以通过将其自定义实例注册为自定义编辑器来覆盖。</p>
<code>FileEditor</code>	<p>将字符串解析为 <code>java.io.File</code> 个对象。默认情况下，由 <code>BeanWrapperImpl</code> 注册。</p>
<code>InputStreamEditor</code>	<p>单向属性编辑器，它可以获取字符串并产生(通过中间的 <code>ResourceEditor</code> 和 <code>Resource</code>) <code>InputStream</code>，以便可以将 <code>InputStream</code> 属性直接设置为字符串。请注意，默认用法不会为您关闭 <code>InputStream</code>。默认情况下，由 <code>BeanWrapperImpl</code> 注册。</p>
<code>LocaleEditor</code>	<p>可以将字符串解析为 <code>Locale</code> 对象，反之亦然(字符串格式为 <code>[country][variant]</code>，与 <code>Locale</code> 的 <code>toString()</code> 方法相同)。默认情况下，由 <code>BeanWrapperImpl</code> 注册。</p>

Class	Explanation
<code>PatternEditor</code>	可以将字符串解析为 <code>java.util.regex.Pattern</code> 个对象，反之亦然。
<code>PropertiesEditor</code>	可以将字符串(格式为 <code>java.util.Properties</code> 类的 javadoc 中定义的格式)转换为 <code>Properties</code> 对象。默认情况下，由 <code>BeanWrapperImpl</code> 注册。
<code>StringTrimmerEditor</code>	修剪字符串的属性编辑器。(可选)允许将空字符串转换为 <code>null</code> 值。默认情况下未注册—必须是用户注册的。 ◦
<code>URLEditor</code>	可以将 URL 的字符串表示形式解析为实际的 <code>URL</code> 对象。默认情况下，由 <code>BeanWrapperImpl</code> 注册。

Spring 使用 `java.beans.PropertyEditorManager` 来设置可能需要的属性编辑器的搜索路径。搜索路径还包括 `sun.bean.editors`，其中 `sun.bean.editors` 包括针对 `Font`，`Color` 等类型和大多数基本类型的 `PropertyEditor` 实现。还要注意，如果标准 JavaBeans 基础结构与它们处理的类在相同的程序包中并且与该类具有相同的名称并附加了 `Editor`，则自动发现 `PropertyEditor` 类(无需显式注册它们)。例如，可能具有以下类和包结构，足以识别 `SomethingEditor` 类并将其用作 `Something` 类型的属性的 `PropertyEditor`。

```
com
  chank
    pop
      Something
        SomethingEditor // the PropertyEditor for the Something class
```

请注意，您也可以在此处使用标准的 `BeanInfo` JavaBeans 机制(在某种程度上已描述[here](#))。下面的示例使用 `BeanInfo` 机制使用关联类的属性显式注册一个或多个 `PropertyEditor` 实例：

```
com
  chank
    pop
      Something
        SomethingBeanInfo // the BeanInfo for the Something class
```

所引用的 `SomethingBeanInfo` 类的以下 Java 源代码将 `CustomNumberEditor` 与 `Something` 类的 `age` 属性相关联：

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

## 注册其他自定义 `PropertyEditor` 实现

当将 `bean` 属性设置为字符串值时，Spring IoC 容器最终使用标准 JavaBeans `PropertyEditor` 实现将这些字符串转换为属性的复杂类型。Spring 预注册了许多自定义 `PropertyEditor` 实现(例如，将表示为字符串的类名转换为 `Class` 对象)。另外，Java 的标准 JavaBeans `PropertyEditor` 查找机制允许为类的 `PropertyEditor` 适当命名，并与它提供支持的类放在同一包中，以便可以自动找到它。

如果需要注册其他自定义 `PropertyEditors`，则可以使用几种机制。最手动的方法(通常不方便或不建议使用)是使用 `ConfigurableBeanFactory` 接口的 `registerCustomEditor()` 方法(假设您

有 `BeanFactory` 引用)。另一种(稍微方便些)的机制是使用称为 `CustomEditorConfigurer` 的特殊 bean 工厂后处理器。尽管您可以将 Bean 工厂后处理器与 `BeanFactory` 实现一起使用，但是 `CustomEditorConfigurer` 具有嵌套的属性设置，因此我们强烈建议您将 `CustomEditorConfigurer` 与 `ApplicationContext` 一起使用，在这里您可以以与其他任何 Bean 类似的方式部署它，并且可以将其放置在自动检测并应用。

请注意，所有 bean 工厂和应用程序上下文通过使用 `BeanWrapper` 来处理属性转换，都会自动使用许多内置的属性编辑器。`BeanWrapper` 寄存器的标准属性编辑器在[上一节](#)中列出。此外，`ApplicationContexts` 还重写或添加其他编辑器，以适合于特定应用程序上下文类型的方式处理资源查找。

标准 JavaBeans `PropertyEditor` 实例用于将以字符串表示的属性值转换为该属性的实际复杂类型。您可以使用 Bean 工厂后处理器 `CustomEditorConfigurer` 来方便地向 `ApplicationContext` 添加对其他 `PropertyEditor` 实例的支持。

考虑以下示例，该示例定义了一个名为 `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;
    }
}
```

正确设置之后，我们希望能够将 type 属性分配为字符串，PropertyEditor 会将其转换为实际的 ExoticType 实例。以下 bean 定义显示了如何构建这种关系：

```
<bean id="sample" class="example.DependsOnExoticType">
    <property name="type" value="aNameForExoticType"/>
</bean>
```

PropertyEditor 的实现可能类似于以下内容：

```
// converts string representation to ExoticType object
package example;

public class ExoticTypeEditor extends PropertyEditorSupport {

    public void setAsText(String text) {
        setValue(new ExoticType(text.toUpperCase()));
    }
}
```

最后，以下示例显示了如何使用 CustomEditorConfigurer 向 ApplicationContext 注册新的 PropertyEditor，然后便可以根据需要使用它了：

```
<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="customEditors">
        <map>
            <entry key="example.ExoticType" value="example.ExoticTypeEditor"/>
        </map>
    </property>
</bean>
```

## Using PropertyEditorRegistrar

向 Spring 容器注册属性编辑器的另一种机制是创建并使用 PropertyEditorRegistrar。当需要在几种不同情况下使用同一组属性编辑器时，此接口特别有用。您可以编写相应的注册商，并在每种情况下重复使用它。PropertyEditorRegistrar 实例与名为 PropertyEditorRegistry 的接口配合使用，该接口由 Spring BeanWrapper (和 DataBinder ) 实现。

PropertyEditorRegistrar 实例与 CustomEditorConfigurer (描述为 [here](#)) 结合使用时特别方便，后者公开了称为 setPropertyEditorRegistrars(..) 的属性。以这种方式添加到

`CustomEditorConfigurer` 的 `PropertyEditorRegistrar` 实例可以轻松地与 `DataBinder` 和 Spring MVC 控制器共享。此外，它避免了在自定义编辑器上进行同步的需要：

`PropertyEditorRegistrar` 有望为每次 bean 创建尝试创建新的 `PropertyEditor` 实例。

以下示例显示了如何创建自己的 `PropertyEditorRegistrar` 实现：

```
package com.foo.editors.spring;

public final class CustomPropertyEditorRegistrar implements PropertyEditorRegistrar {

    public void registerCustomEditors(PropertyEditorRegistry registry) {

        // it is expected that new PropertyEditor instances are created
        registry.registerCustomEditor(ExoticType.class, new ExoticTypeEditor());

        // you could register as many custom property editors as are required here...
    }
}
```

另请参见 `org.springframework.beans.support.ResourceEditorRegistrar` 以获取示例

`PropertyEditorRegistrar` 的实现。请注意，在实现 `registerCustomEditors(..)` 方法时，它如何创建每个属性编辑器的新实例。

下一个示例显示了如何配置 `CustomEditorConfigurer` 并将 `CustomPropertyEditorRegistrar` 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](#) 的用户而言，这与本章的重点有所偏离)，将 `PropertyEditorRegistrars` 与数据绑定 `Controllers` (例如 `SimpleFormController`) 结合使用会非常方便。下面的示例在 `initBinder(..)` 方法的实现中使用 `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 `PropertyEditor` 注册样式可以导致代码简洁(`initBinder(..)` 的实现只有一行长), 并且可以将通用的 `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。

## 3.4. Spring 类型转换

Spring 3 引入了 `core.convert` 软件包, 该软件包提供了常规的类型转换系统。该系统定义了一个用于实现类型转换逻辑的 SPI 和一个用于在运行时执行类型转换的 API。在 Spring 容器中, 您可以将此系统用作 `PropertyEditor` 实现的替代方法, 以将外部化的 bean 属性值字符串转换为所需的属性类型。您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. 转换器 SPI

如以下接口定义所示, 用于实现类型转换逻辑的 SPI 非常简单且具有强类型。

```

package org.springframework.core.convert.converter;

public interface Converter<S, T> {

    T convert(S source);
}

```

要创建自己的转换器, 请实现 `Converter` 界面, 并将 `S` 作为要转换的类型, 并将 `T` 参数化为要转换的类型。如果需要将 `S` 的集合或数组转换为 `T` 的数组或集合, 您也可以透明地应用此类转换

器，前提是还已注册了委派的数组或集合转换器(默认情况下 `DefaultConversionService` 这样做)。

对于每次 `convert(S)` 的调用，保证源参数不为 `null`。如果转换失败，您的 `Converter` 可能会引发任何未经检查的异常。具体来说，它应该抛出 `IllegalArgumentException` 以报告无效的源值。注意确保 `Converter` 实现是线程安全的。

为方便起见，在 `core.convert.support` 包中提供了几种转换器实现。这些包括从字符串到数字以及其他常见类型的转换器。下面的清单显示了 `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. 使用 `ConverterFactory`

当需要集中整个类层次结构的转换逻辑时(例如，从 `String` 转换为 `java.lang.Enum` 对象时)，可以实现 `ConverterFactory`，如以下示例所示：

```
package org.springframework.core.convert.converter;

public interfaceConverterFactory<S, R> {

    <T extends R> Converter<S, T> getConverter(Class<T> targetType);
}
```

参数化 `S` 为您要转换的类型，参数化 `R` 为定义可以转换为的类的“范围”的基本类型。然后实现 `getConverter(Class)`，其中 `T` 是 `R` 的子类。

以 `StringToEnum` `ConverterFactory` 为例：

```

package org.springframework.core.convert.support;

final class StringToEnumConverterFactory implementsConverterFactory<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

当您需要复杂的 `Converter` 实现时，请考虑使用 `GenericConverter` 界面。

`GenericConverter` 具有比 `Converter` 更灵活但类型不那么强的签名，支持

`GenericConverter` 在多种源类型和目标类型之间进行转换。另外，`GenericConverter` 提供了实现转换逻辑时可以使用的源字段和目标字段上下文。这种上下文允许类型转换由字段 `Comments` 或在字段签名上声明的通用信息驱动。以下清单显示了 `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`，让 `getConvertibleTypes()` 返回支持的源→目标类型对。然后实现 `convert(Object, TypeDescriptor, TypeDescriptor)` 以包含您的转换逻辑。源 `TypeDescriptor` 提供对包含正在转换的值的源字段的访问。目标 `TypeDescriptor` 提供对要设置转换值的目标字段的访问。

`GenericConverter` 的一个很好的例子是在 Java 数组和集合之间进行转换的转换器。这样的 `ArrayToCollectionConverter` 会检查声明目标集合类型的字段以解析集合的元素类型。这样就可以在将集合设置到目标字段上之前，将源数组中的每个元素转换为集合元素类型。

### Note

由于 `GenericConverter` 是更复杂的 SPI 接口，因此仅应在需要时使用它。支持 `Converter` 或 `ConverterFactory` 以满足基本类型转换需求。

## Using ConditionalGenericConverter

有时，您希望 `Converter` 仅在满足特定条件下运行。例如，您可能只想在目标字段上存在特定 Comments 时才运行 `Converter`，或者仅在目标类上定义了特定方法(例如 `static valueOf` 方法)时才运行 `Converter`。`ConditionalGenericConverter` 是 `GenericConverter` 和 `ConditionalConverter` 接口的并集，可用于定义以下自定义匹配条件：

```
public interface ConditionalConverter {  
    boolean matches(TypeDescriptor sourceType, TypeDescriptor targetType);  
}  
  
public interface ConditionalGenericConverter extends GenericConverter, ConditionalConve
```

`ConditionalGenericConverter` 的一个很好的例子是 `EntityConverter`，它在持久性实体标识符和实体引用之间转换。仅当目标实体类型声明静态查找器方法(例如 `findAccount(Long)`)时，此类 `EntityConverter` 才可能匹配。您可以在 `matches(TypeDescriptor, TypeDescriptor)` 的实现中执行这种 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。转换器通常在以

下外观界面后面执行：

```
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` 实现还实现 `ConverterRegistry`，该 `ConverterRegistry` 提供了用于注册转换器的 SPI。在内部，`ConversionService` 实现委派其注册的转换器执行类型转换逻辑。

`core.convert.support` 软件包中提供了一种强大的 `ConversionService` 实现。

`GenericConversionService` 是适用于大多数环境的通用实现。`ConversionServiceFactory` 提供了一个方便的工厂来创建通用的 `ConversionService` 配置。

### 3.4.5. 配置 `ConversionService`

`ConversionService` 是 Stateless 对象，旨在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`)配置一个 `ConversionService` 实例。Spring 需要 `ConversionService` 并在框架需要执行类型转换时使用它。您也可以将此 `ConversionService` 注入到您的任何 bean 中并直接调用它。

#### ① Note

如果没有 `ConversionService` 向 Spring 注册，则使用基于 `PropertyEditor` 的原始系统。

要向 Spring 注册默认的 `ConversionService`，请添加以下 Bean 定义，并将 `id` 的

`conversionService`：

```
<bean id="conversionService"
      class="org.springframework.context.support.ConversionServiceFactoryBean"/>
```

默认的 `ConversionService` 可以在字符串，数字，枚举，集合，Map 和其他常见类型之间进行转换。要用您自己的自定义转换器补充或覆盖默认转换器，请设置 `converters` 属性。属性值可以实现 `Converter`，`ConverterFactory` 或 `GenericConverter` 接口中的任何一个。

```
<bean id="conversionService"
      class="org.springframework.context.support.ConversionServiceFactoryBean">
    <property name="converters">
      <set>
        <bean class="example.MyCustomConverter"/>
      </set>
    </property>
</bean>
```

在 Spring MVC 应用程序中通常使用 `ConversionService`。请参阅 Spring MVC 章节中的[转换和格式化](#)。

在某些情况下，您可能希望在转换过程中应用格式设置。有关使用

`FormattingConversionServiceFactoryBean` 的详细信息，请参见[FormatterRegistry SPI](#)。

### 3.4.6. 以编程方式使用 `ConversionService`

要以编程方式使用 `ConversionService` 实例，可以像对其他任何 bean 一样注入对该实例的引用。以下示例显示了如何执行此操作：

```
@Service
public class MyService {

    @Autowired
    public MyService(ConversionService conversionService) {
        this.conversionService = conversionService;
    }

    public void doIt() {
```

```
        this.conversionService.convert(...)  
    }  
}
```

对于大多数使用情况，可以使用指定 `targetType` 的 `convert` 方法，但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 `Integer` 的 `List` 转换为 `String` 的 `List`，则需要提供源类型和目标类型的正式定义。

幸运的是，`TypeDescriptor` 提供了各种选项来使操作变得如此简单，如以下示例所示：

```
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)));
```

请注意，`DefaultConversionService` 自动注册适用于大多数环境的转换器。这包括集合转换器，标量转换器和基本的 `Object` 到 `String` 转换器。您可以使用 `DefaultConversionService` 类上的静态 `addDefaultConverters` 方法将相同的转换器注册到任何 `ConverterRegistry` 中。

值类型的转换器可重用于数组和集合，因此，无需使用特定的转换器即可将 `s` 的 `Collection` 转换为 `T` 的 `Collection`，前提是需要标准的集合处理。

## 3.5. Spring 字段格式

如上一节所述，`core.convert` 是通用类型转换系统。它提供了统一的 `ConversionService` API 以及强类型的 `Converter` 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 Expression Language(SpEL) 和 `DataBinder` 都使用此系统绑定字段值。例如，当 SpEL 需要将 `Short` 强制转换为 `Long` 才能完成

`expression.setValue(Object bean, Object value)` 尝试时，`core.convert` 系统将执行强制转换。

现在考虑典型 Client 端环境(例如 Web 或桌面应用程序)的类型转换要求。在这种环境中，通常您会从 `String` 转换为支持 Client 端回发过程，而从 `String` 转换为支持视图渲染过程。另外，您通常需要本地化 `String` 值。更为通用的 `core.convert Converter` SPI 不能直接解决此类格式化要求。为了直接解决这些问题，Spring 3 引入了一个方便的 `Formatter` SPI，它为 Client 端环境提供了一种简单且健壮的替代 `PropertyEditor` 实现的方法。

通常，当您需要实现通用类型转换逻辑时(例如，在 `java.util.Date` 和 `java.lang.Long` 之间进行转换)，可以使用 `Converter` SPI。在 Client 端环境(例如 Web 应用程序)中工作并且需要解析和打印本地化的字段值时，可以使用 `Formatter` SPI。`ConversionService` 为两个 SPI 提供统一的类型转换 API。

### 3.5.1. 格式化器 SPI

用于实现字段格式逻辑的 `Formatter` SPI 非常简单且类型严格。以下清单显示了 `Formatter` 接口定义：

```
package org.springframework.format;

public interface Formatter<T> extends Printer<T>, Parser<T> { }
```

`Formatter` 来自 `Printer` 和 `Parser` 构件块接口。以下清单显示了这两个接口的定义：

```
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;
}
```

要创建自己的 `Formatter`，请实现前面显示的 `Formatter` 接口。将 `T` 参数化为您希望格式化的

对象类型(例如 `java.util.Date`)。实现 `print()` 操作以打印 `T` 的实例以在 Client 端语言环境中显示。实现 `parse()` 操作, 以从 Client 端语言环境返回的格式化表示形式解析 `T` 的实例。如果解析尝试失败, 则您的 `Formatter` 应该抛出 `ParseException` 或 `IllegalArgumentException`。注意确保 `Formatter` 实现是线程安全的。

为了方便起见, `format` 子软件包提供了 `Formatter` 个实现。`number` 包提供 `NumberStyleFormatter`, `CurrencyStyleFormatter` 和 `PercentStyleFormatter` 来格式化使用 `java.text.NumberFormat` 的 `java.lang.Number` 对象。`datetime` 包提供了 `DateFormatter`, 用于将 `java.util.Date` 对象格式化为 `java.text.DateFormat`。`datetime.joda` 包基于[Joda-Time library](#)提供了全面的日期时间格式支持。

以下 `DateFormatter` 是示例 `Formatter` 的实现:

```
package org.springframework.format.datetime;

public final class DateFormatter implements Formatter<Date> {

    private String pattern;

    public DateFormatter(String pattern) {
        this.pattern = pattern;
    }

    public String print(Date date, Locale locale) {
        if (date == null) {
            return "";
        }
        return getDateFormat(locale).format(date);
    }

    public Date parse(String formatted, Locale locale) throws ParseException {
        if (formatted.length() == 0) {
            return null;
        }
        return getDateFormat(locale).parse(formatted);
    }

    protected DateFormat getDateFormat(Locale locale) {
        DateFormat dateFormat = new SimpleDateFormat(this.pattern, locale);
        dateFormat.setLenient(false);
        return dateFormat;
    }
}
```

### 3.5.2. Comments 驱动的格式

可以通过字段类型或 Comments 配置字段格式。要将 Comments 绑定到 [Formatter](#), 请实现

[AnnotationFormatterFactory](#)。以下清单显示了 [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);
}
```

要创建一个实现：。将 A 参数化为您希望与格式逻辑关联的字段 [annotationType](#)，例如

[org.springframework.format.annotation.DateTimeFormat](#)。。让 [getFieldTypes\(\)](#) 返回可以在其上使用 Comments 的字段类型。。让 [getPrinter\(\)](#) 返回 [Printer](#) 以打印带 Comments 的字段的值。。让 [getParser\(\)](#) 返回 [Parser](#) 来解析带 Comments 字段的 [clientValue](#)。

下面的示例 [AnnotationFormatterFactory](#) 实现将 [@NumberFormat](#) Comments 绑定到格式化程序，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);
    }
}
```

```

public Parser<Number> getParser(NumberFormat annotation, Class<?> fieldType) {
    return configureFormatterFrom(annotation, fieldType);
}

private Formatter<Number> configureFormatterFrom(NumberFormat annotation, Class<?>
    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();
        }
    }
}
}

```

要触发格式，可以使用`@NumberFormatComments` 字段，如以下示例所示：

```

public class MyModel {

    @NumberFormat(style=Style.CURRENCY)
    private BigDecimal decimal;
}

```

## 格式 CommentsAPI

`org.springframework.format.annotation` 软件包中存在可移植格式 CommentsAPI。您可以

使用 `@NumberFormat` 格式化 `java.lang.Number` 字段，使用 `@DateTimeFormat` 格式化

`java.util.Date`，`java.util.Calendar`，`java.util.Long` 或 Joda-Time 字段。

下面的示例使用 `@DateTimeFormat` 格式化 `java.util.Date` 作为 ISO 日期(yyyy-MM-dd)：

```

public class MyModel {

    @DateTimeFormat(iso=ISO.DATE)
    private Date date;
}

```

## 3.5.3. FormatterRegistry SPI

`FormatterRegistry` 是用于注册格式器和转换器的 SPI。 `FormattingConversionService` 是适用于大多数环境的 `FormatterRegistry` 的实现。您可以使用 `FormattingConversionServiceFactoryBean` 以编程方式或声明方式将此实现配置为 Spring bean。因为此实现还实现了 `ConversionService`，所以您可以直接配置它以与 Spring 的 `DataBinder` 和 Spring Expression Language(SpEL)一起使用。

以下清单显示了 `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);

    void addFormatterForFieldType(Formatter<?> formatter);

    void addFormatterForAnnotation(AnnotationFormatterFactory<?, ?> factory);
}
```

如前面的清单所示，您可以按字段类型或注解注册格式化程序。

`FormatterRegistry` SPI 使您可以集中配置格式设置规则，而不必在控制器之间复制此类配置。例如，您可能需要强制以某种方式设置所有日期字段的格式或以某种方式设置带有特定注解的字段的格式。使用共享的 `FormatterRegistry`，您一次定义这些规则，并在需要格式化时将它们应用。

### 3.5.4. FormatterRegistrar SPI

`FormatterRegistrar` 是用于通过 `FormatterRegistry` 注册格式器和转换器的 SPI。以下清单显示了其接口定义：

```
package org.springframework.format;

public interface FormatterRegistrar {

    void registerFormatters(FormatterRegistry registry);
```

```
}
```

为给定的格式类别(例如日期格式)注册多个相关的转换器和格式器时, `FormatterRegistrar` 很有用。在声明式注册不充分的情况下它也很有用。例如, 当格式化程序需要在不同于其自己的 `<T>` 的特定字段类型下进行索引时, 或者在注册 `Printer` / `Parser` 对时。下一节将提供有关转换器和格式化程序注册的更多信息。

### 3.5.5. 在 Spring MVC 中配置格式

请参阅 Spring MVC 章节中的[转换和格式化](#)。

## 3.6. 配置全局日期和时间格式

默认情况下, 未使用 `@DateTimeFormat Comments` 的日期和时间字段是使用 `DateFormat.SHORT` 样式从字符串转换的。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, 您需要确保 Spring 不注册默认格式器。相反, 您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, 具体取决于您使用的是 Joda-Time 库。

例如, 以下 Java 配置注册全局 `yyyyMMdd` 格式(此示例不依赖于 Joda-Time 库):

```
@Configuration
public class AppConfig {

    @Bean
    public FormattingConversionService conversionService() {

        // Use the DefaultFormattingConversionService but do not register defaults
        DefaultFormattingConversionService conversionService = new DefaultFormattingCon

        // Ensure @NumberFormat is still supported
        conversionService.addFormatterForFieldAnnotation(new NumberFormatAnnotationForm

        // Register date conversion with a specific global format
    }
}
```

```

        DateFormatRegistrar registrar = new DateFormatRegistrar();
        registrar.setFormatter(new DateFormat("yyyyMMdd"));
        registrar.registerFormatters(conversionService);

        return conversionService;
    }
}

```

如果您喜欢基于 XML 的配置，则可以使用 [FormattingConversionServiceFactoryBean](#)。以下示例显示了如何执行此操作(这次使用 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
           http://www.springframework.org/schema/beans/spring-beans.xsd>

<bean id="conversionService" class="org.springframework.format.support.FormattingCo
    <property name="registerDefaultFormatters" value="false" />
    <property name="formatters">
        <set>
            <bean class="org.springframework.format.number.NumberFormatAnnotationFo
                </set>
            </property>
            <property name="formatterRegistrars">
                <set>
                    <bean class="org.springframework.format.datetime.joda.JodaTimeFormatter
                        <property name="dateFormatter">
                            <bean class="org.springframework.format.datetime.joda.DateTimeF
                                <property name="pattern" value="yyyyMMdd"/>
                            </bean>
                        </property>
                    </bean>
                </set>
            </property>
        </bean>
    </beans>

```

## iNote

Joda-Time 提供了单独的不同类型来表示 `date`，`time` 和 `date-time` 值。

`JodaTimeFormatterRegistrar` 的 `dateFormatter`，`timeFormatter` 和 `dateTimeFormatter` 属性应用于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便捷方法。

## iNote

如果您使用 Spring MVC，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` 类并覆盖 `mvcConversionService()` 方法。对于 XML，应使用 `mvc:annotation-driven` 元素的 `conversion-service` 属性。有关详情，请参见[转换和格式化](#)。

## 3.7. Spring 验证

Spring 3 对其验证支持进行了一些增强。首先，完全支持 JSR-303 Bean 验证 API。其次，当以编程方式使用时，Spring 的 `DataBinder` 可以验证对象并绑定到它们。第三，Spring MVC 支持以声明方式验证 `@Controller` Importing。

### 3.7.1. JSR-303 Bean 验证 API 概述

JSR-303 标准化了 Java 平台的验证约束声明和元数据。通过使用此 API，您可以使用声明性验证约束来 Comments 域模型属性，并且运行时会强制执行它们。您可以使用许多内置约束。您还可以定义自己的自定义约束。

考虑以下示例，该示例显示了具有两个属性的简单 `PersonForm` 模型：

```
public class PersonForm {  
    private String name;  
    private int age;  
}
```

JSR-303 允许您针对此类属性定义声明性验证约束，如以下示例所示：

```
public class PersonForm {  
  
    @NotNull  
    @Size(max=64)  
    private String name;  
  
    @Min(0)  
    private int age;  
}
```

当 JSR-303 验证程序验证此类的实例时，将强制执行这些约束。

有关 JSR-303 和 JSR-349 的一般信息，请参见[Bean 验证网站](#)。有关默认参考实现的特定功能的信息，请参见[Hibernate Validator](#) 文档。要学习如何将 bean 验证提供程序设置为 Spring bean，请 continue 阅读。

### 3.7.2. 配置 Bean 验证提供程序

Spring 提供了对 Bean 验证 API 的全面支持。这包括对将 JSR-303 或 JSR-349 Bean 验证提供程序引导为 Spring Bean 的便捷支持。这样，您就可以在应用程序中需要验证的地方注入

`javax.validation.ValidatorFactory` 或 `javax.validation.Validator`。

您可以使用 `LocalValidatorFactoryBean` 将默认的 Validator 配置为 Spring Bean，如以下示例所示：

```
<bean id="validator"
      class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean" />
```

前面示例中的基本配置触发 Bean 验证以使用其默认引导机制进行初始化。诸如 Hibernate Validator 之类的 JSR-303 或 JSR-349 提供程序应预期存在于 Classpath 中并被自动检测到。

#### 注入验证器

`LocalValidatorFactoryBean` 实现 `javax.validation.ValidatorFactory` 和 `javax.validation.Validator` 以及 Spring 的 `org.springframework.validation.Validator`。您可以将对这些接口之一的引用注入需要调用验证逻辑的 bean 中。

如果您希望直接使用 Bean Validation API，则可以插入对 `javax.validation.Validator` 的引用，如以下示例所示：

```
import javax.validation.Validator;
@Service
public class MyService {
```

```
@Autowired  
private Validator validator;
```

如果您的 bean 需要使用 Spring Validation API, 则可以注入对

`org.springframework.validation.Validator` 的引用, 如以下示例所示:

```
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: \* `@Constraint` Comments, 用于声明约束及其可配置属性。  
◦ \* `javax.validation.ConstraintValidator` 接口的实现, 用于实现约束的行为。

要将声明与实现相关联, 每个 `@Constraint` 注解都引用一个对应的 `ConstraintValidator` 实现类。在运行时, 当域模型中遇到约束 Comments 时, `ConstraintValidatorFactory` 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`。这使您的自定义 `ConstraintValidators` 像任何其他 Spring bean 一样受益于依赖项注入。

以下示例显示了一个自定义 `@Constraint` 声明, 后跟一个关联的 `ConstraintValidator` 实现, 该实现使用 Spring 进行依赖项注入:

```
@Target({ElementType.METHOD, ElementType.FIELD})  
@Retention(RetentionPolicy.RUNTIME)  
@Constraint(validatedBy=MyConstraintValidator.class)  
public @interface MyConstraint {  
}
```

```
import javax.validation.ConstraintValidator;

public class MyConstraintValidator implements ConstraintValidator {

    @Autowired;
    private Foo aDependency;

    ...
}
```

如前面的示例所示，`ConstraintValidator` 实现可以像其他任何 Spring bean 一样具有其依赖关系 `@Autowired`。

## Spring 驱动的方法验证

您可以通过 `MethodValidationPostProcessor` bean 定义将 Bean Validation 1.1(以及作为自定义扩展，还包括 Hibernate Validator 4.3)支持的方法验证功能通过

`MethodValidationPostProcessor` bean 定义集成到 Spring 上下文中，如下所示：

```
<bean class="org.springframework.validation.beanvalidation.MethodValidationPostProcessor"
```

为了有资格通过 Spring 驱动的方法验证，所有目标类都需要使用 Spring 的 `@Validated` Comments 进行 Comments。 (可选地，您也可以声明要使用的验证组.)有关 Hibernate Validator 和 Bean Validation 1.1 提供程序的设置详细信息，请参见[MethodValidationPostProcessor](#) javadoc。

## 其他配置选项

在大多数情况下，默认的 `LocalValidatorFactoryBean` 配置就足够了。从消息插值到遍历解析，有许多用于各种 Bean 验证构造的配置选项。有关这些选项的更多信息，请参见[LocalValidatorFactoryBean](#) javadoc。

### 3.7.3. 配置一个 DataBinder

从 Spring 3 开始，您可以使用 `Validator` 配置 `DataBinder` 实例。配置完成后，您可以通过调用

`binder.validate()` 来调用 `Validator`。任何验证 `Errors` 都会自动添加到 Binder 的 `BindingResult` 中。

下面的示例显示在绑定到目标对象之后，如何以编程方式使用 `DataBinder` 来调用验证逻辑：

```
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();
```

您还可以通过 `dataBinder.addValidators` 和 `dataBinder.replaceValidators` 配置具有多个 `Validator` 实例的 `DataBinder`。当将全局配置的 bean 验证与在 `DataBinder` 实例上本地配置的 Spring `Validator` 结合使用时，这很有用。参见[\[validation-mvc-configuring\]](#)。

### 3.7.4. Spring MVC 3 验证

请参阅 Spring MVC 章节中的[Validation](#)。

## 4. Spring 表达式语言(SpEL)

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。语言论法类似于 Unified EL，但提供了其他功能，最著名的是方法调用和基本的字符串模板功能。

尽管还有其他几种 Java 表达式语言可用-OGNL, MVEL 和 JBoss EL, 仅举几例-Spring 表达式语言的创建是为了向 Spring 社区提供一种受良好支持的表达式语言，该语言可用于以下版本中的所有产品 Spring 投资组合。它的语言功能由 Spring 产品组合中的项目要求所驱动，包括基于 Eclipse 的 Spring Tool Suite 中代码完成支持的工具要求。也就是说，SpEL 基于与技术无关的 API，如果需要，可以将其他表达语言实现集成在一起。

虽然 SpEL 是 Spring 产品组合中表达评估的基础，但它并不直接与 Spring 绑定，可以独立使用。为了自成一体，本章中的许多示例都将 SpEL 当作一种独立的表达语言来使用。这需要创建一些自举基础结构类，例如解析器。大多数 Spring 用户不需要处理这种基础结构，而只能编写表达式字符串进行评估。这种典型用法的一个示例是将 SpEL 集成到创建 XML 或基于 Comments 的 Bean 定义中，如[表达式支持，用于定义 bean 定义](#)所示。

本章介绍了表达语言，其 API 和语言语法的功能。在许多地方，`Inventor` 和 `Society` 类用作表达式评估的目标对象。这些类声明和用于填充它们的数据在本章末尾列出。

表达式语言支持以下功能：

- Literal expressions
- 布尔运算符和关系运算符
- Regular expressions
- Class expressions
- 访问属性，数组，列表和 Map
- 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

## 4.1. Evaluation

本节介绍 SpEL 接口的简单用法及其表达语言。完整的语言参考可以在[Language Reference](#) 中找到。

以下代码介绍了 SpEL API 来评估 Literals 字符串表达式 `Hello World`。

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("'Hello World'"); (1)
String message = (String) exp.getValue();
```

- (1) 消息变量的值为 `'Hello World'`。

您最可能使用的 SpEL 类和接口位于 `org.springframework.expression` 包及其子包中，例如 `spel.support`。

`ExpressionParser` 接口负责解析表达式字符串。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 `Literals`。`Expression` 接口负责评估先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 支持多种功能，例如调用方法，访问属性和调用构造函数。

在以下方法调用示例中，我们在字符串 `Literals` 上调用 `concat` 方法：

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("'Hello World'.concat('!'))); (1)
String message = (String) exp.getValue();
```

- (1) `message` 的值现在是“Hello World!”。

以下调用 JavaBean 属性的示例将调用 `String` 属性 `Bytes`：

```
ExpressionParser parser = new SpelExpressionParser();

// invokes 'getBytes()'
Expression exp = parser.parseExpression("'Hello World'.bytes"); (1)
byte[] bytes = (byte[]) exp.getValue();
```

- (1) 这行将 `Literals` 转换为字节数组。

SpEL 还通过使用标准的点符号(例如 `prop1.prop2.prop3`)和属性值的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 `Literals` 的长度：

```
ExpressionParser parser = new SpelExpressionParser();

// invokes 'getBytes().length'
Expression exp = parser.parseExpression("'Hello World'.bytes.length"); (1)
int length = (Integer) exp.getValue();
```

- (1) `'Hello World'.bytes.length` 给出 `Literals` 的长度。

可以调用 `String` 的构造函数，而不是使用字符串 `Literals`，如以下示例所示：

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("new String('hello world').toUpperCase()"); (1)
String message = exp.getValue(String.class);
```

- (1) 从原义构造一个新的 `String` 并使其大写。

注意使用通用方法：`public <T> T getValue(Class<T> desiredResultType)`。使用此方法无需将表达式的值强制转换为所需的结果类型。如果无法将值强制转换为 `T` 类型或无法使用已注册的类型转换器进行转换，则将引发 `EvaluationException`。

SpEL 的更常见用法是提供一个表达式字符串，该字符串针对特定对象实例(称为根对象)进行评估。

下面的示例演示如何从 `Inventor` 类的实例检索 `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"); (1)
String name = (String) exp.getValue(tesla);
// name == "Nikola Tesla"

exp = parser.parseExpression("name == 'Nikola Tesla'");
boolean result = exp.getValue(tesla, Boolean.class);
// result == true
```

- (1) 将 `name` 解析为表达式。

#### 4.1.1. 了解评估上下文

评估表达式以解析属性，方法或字段并帮助执行类型转换时，使用 `EvaluationContext` 接口。

Spring 提供了两种实现。

- `SimpleEvaluationContext`：针对不需要全部 SpEL 语言语法范围且应受到有意义限制的表达式类别，公开了 SpEL 基本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过滤器。
- `StandardEvaluationContext`：公开 SpEL 语言功能和配置选项的全部集合。您可以使用它来指定默认的根对象，并配置每个可用的评估相关策略。

`SimpleEvaluationContext` 设计为仅支持 SpEL 语言语法的一部分。它不包括 Java 类型引用，构造函数和 Bean 引用。它还要求您明确选择对表达式中的属性和支持级别的方法。默认情况下，`create()` 静态工厂方法仅启用对属性的读取访问。您还可以获取构建器来配置所需的确切支持级别，并针对以下一种或某些组合：

- 仅自定义 `PropertyAccessor` (无反射)
- 只读访问的数据绑定属性
- 读写的数据绑定属性

## Type Conversion

默认情况下，SpEL 使用 Spring core(

`org.springframework.core.convert.ConversionService`)中可用的转换服务。此转换服务随附有许多用于常见转换的内置转换器，但它也是完全可扩展的，因此您可以在类型之间添加自定义转换。此外，它是泛型感知的。这意味着，当您在表达式中使用泛型类型时，SpEL 会尝试进行转换以维护遇到的任何对象的类型正确性。

在实践中这意味着什么？假设使用 `setValue()` 的赋值被用来设置 `List` 属性。该属性的类型实际上 是 `List<Boolean>`。SpEL 意识到列表中的元素需要先转换为 `Boolean`，然后才能放入其中。

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);

EvaluationContext context = SimpleEvaluationContext().forReadOnlyDataBinding().build();

// false is passed in here as a string. SpEL and the conversion service
// correctly recognize that it needs to be a Boolean and convert it
parser.parseExpression("booleanList[0]").setValue(context, simple, "false");

// b is false
Boolean b = simple.booleanList.get(0);
```

### 4.1.2. 解析器配置

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。配置对象控制某些表达式组件的行为。例如，如果您索引到数组或集合中并且指定索引处的元素是 `null`，则可以自动创建该元素。当使用由属性引用链组成的表达式时，这很有用。如果索引

到数组或列表中并指定的索引超出了数组或列表当前大小的末尾，则可以自动增长数组或列表以容纳该索引。下面的示例演示如何自动增加列表：

```
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

Spring Framework 4.1 包含一个基本的表达式编译器。通常对表达式进行解释，这样可以在评估过程中提供很大的动态灵 Active，但不能提供最佳性能。对于偶尔使用表达式，这很好，但是，当与其他组件(如 Spring Integration)一起使用时，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。在评估期间，编译器会生成一个真实的 Java 类，该类体现了表达式行为，并使用该类来实现更快的表达式评估。由于缺少在表达式周围 Importing 内容的信息，因此编译器在执行编译时会使用在表达式的解释求值过程中收集的信息。例如，它不仅仅从表达式中就知道属性引用的类型，而是在第一次解释求值时就知道它是什么。当然，如果各种表达式元素的类型随时间变化，则基于此信息进行编译可能会在以后引起麻烦。因此，编译最适合类型信息在重复求值时不会改变的表达式。

考虑以下基本表达式：

```
someArray[0].someProperty.someOtherProperty < 0.1
```

因为前面的表达式涉及数组访问，一些属性取消引用和数字运算，所以性能提升可能非常明显。在一个示例中，进行了 50000 次迭代的微基准测试，使用解释器评估需要 75 毫秒，而使用表达式的

编译版本仅需要 3 毫秒。

## Compiler Configuration

默认情况下不打开编译器，但是您可以通过两种不同的方式之一来打开它。当 SpEL 用法嵌入到另一个组件中时，可以使用解析器配置过程([discussed earlier](#))或使用系统属性来将其打开。本节讨论这两个选项。

编译器可以在 `org.springframework.expression.spel.SpelCompilerMode` 枚举中捕获的三种模式之一进行操作。模式如下：

- `OFF` (默认)：编译器已关闭。
- `IMMEDIATE`：在立即模式下，将尽快编译表达式。通常是在第一次解释评估之后。如果编译的表达式失败(通常是由类型更改，如前所述)，则表达式求值的调用者将收到异常。
- `MIXED`：在混合模式下，表达式会随着时间静默在解释模式和编译模式之间切换。经过一定数量的解释运行后，它们会切换到编译形式，如果编译形式出了问题(例如，如前所述的类型更改)，则表达式会自动再次切换回解释形式。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。

之所以存在 `IMMEDIATE` 模式，是因为 `MIXED` 模式可能会导致具有副作用的表达式出现问题。如果已编译的表达式在部分成功后就崩溃了，则它可能已经完成了影响系统状态的操作。如果发生这种情况，调用者可能不希望它在解释模式下静默地重新运行，因为表达式的一部分可能运行了两次。

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);
```

当指定编译器模式时，还可以指定一个类加载器(允许传递 null)。编译的表达式是在提供的任何子类加载器中定义的。重要的是要确保，如果指定了类加载器，则它可以查看表达式评估过程中涉及的所有类型。如果未指定类加载器，则使用默认的类加载器(通常是在表达式求值期间运行的线程的上下文类加载器)。

第二种配置编译器的方法是将 SpEL 嵌入到其他组件中，并且可能无法通过配置对象进行配置。在这些情况下，可以使用系统属性。您可以将 `spring.expression.compiler.mode` 属性设置为 `SpelCompilerMode` 枚举值之一(`off`，`immediate` 或 `mixed`)。

## Compiler Limitations

从 Spring Framework 4.1 开始，已经有了基本的编译框架。但是，该框架尚不支持编译每种表达式。最初的重点是可能在性能关键型上下文中使用的通用表达式。目前无法编译以下类型的表达式：

- 涉及赋值的表达
- 表达式依赖转换服务
- 使用自定义解析器或访问器的表达式
- 使用选择或投影的表达式

将来会编译更多类型的表达。

## 4.2. Bean 定义中的表达式

您可以将 SpEL 表达式与基于 XML 或基于 Comments 的配置元数据一起使用，以定义

`BeanDefinition` 实例。在这两种情况下，定义表达式的语法均为 `#{ <expression string> }` 形式。

### 4.2.1. XML 配置

可以使用表达式来设置属性或构造函数的参数值，如以下示例所示：

```
<bean id="numberGuess" class="org.springframework.samples.NumberGuess">
```

```
<property name="randomNumber" value="#{ T(java.lang.Math).random() * 100.0 }"/>  
<!-- other properties -->  
</bean>
```

**systemProperties** 变量是 **sched** 定义的，因此您可以在表达式中使用它，如以下示例所示：

```
<bean id="taxCalculator" class="org.springframework.samples.TaxCalculator">  
    <property name="defaultLocale" value="#{ systemProperties['user.region'] }"/>  
  
    <!-- other properties -->  
</bean>
```

请注意，在这种情况下，您不必在 **sched** 定义变量前加上 **#** 符号。

您还可以按名称引用其他 **bean** 属性，如以下示例所示：

```
<bean id="numberGuess" class="org.springframework.samples.NumberGuess">  
    <property name="randomNumber" value="#{ T(java.lang.Math).random() * 100.0 }"/>  
  
    <!-- other properties -->  
</bean>  
  
<bean id="shapeGuess" class="org.springframework.samples.ShapeGuess">  
    <property name="initialShapeSeed" value="#{ numberGuess.randomNumber }"/>  
  
    <!-- other properties -->  
</bean>
```

## 4.2.2. Comments 配置

若要指定默认值，可以在字段、方法以及方法或构造函数参数上放置 **@Value** 注解。

下面的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() {  
        return this.defaultLocale;  
    }
```

```
}
```

以下示例显示了等效的但使用属性设置器方法的示例：

```
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;
    }

}
```

自动装配的方法和构造函数也可以使用 `@Value` 注解，如以下示例所示：

```
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;
    }

    // ...
}
```

## 4.3. 语言参考

本节描述了 Spring Expression Language 的工作方式。它涵盖以下主题：

- [Literal Expressions](#)
- [属性, 数组, 列表, Map 和索引器](#)
- [Inline Lists](#)
- [Inline Maps](#)
- [Array Construction](#)
- [Methods](#)
- [Operators](#)
- [Types](#)
- [Constructors](#)
- [Variables](#)
- [Functions](#)
- [Bean References](#)
- [三元运算符\(If-Then-Else\)](#)
- [Elvis operator](#)
- [安全导航操作员](#)

### 4.3.1. Literals 表达

支持的 Literals 表达式的类型为字符串, 数值(int, 实数, 十六进制), 布尔值和 null。字符串由单引号引起。要将单引号本身放在字符串中, 请使用两个单引号字符。

以下清单显示了 Literals 的简单用法。通常, 它们不是像这样孤立地使用, 而是作为更复杂的表达

式的一部分使用-例如，在逻辑比较运算符的一侧使用 **Literals**。

```
ExpressionParser parser = new SpelExpressionParser();

// evals to "Hello World"
String helloWorld = (String) parser.parseExpression("'Hello World'").getValue();

double avogadrosNumber = (Double) parser.parseExpression("6.0221415E+23").getValue();

// evals to 2147483647
int maxValue = (Integer) parser.parseExpression("0x7FFFFFFF").getValue();

boolean trueValue = (Boolean) parser.parseExpression("true").getValue();

Object nullValue = parser.parseExpression("null").getValue();
```

数字支持使用负号，指数符号和小数点。默认情况下，使用 **Double.parseDouble()** 解析实数。

### 4.3.2. 属性，数组，列表，Map 和索引器

使用属性引用进行导航很容易。为此，请使用句点来指示嵌套的属性值。**Inventor** 类的实例

**pupin** 和 **tesla** 填充了[示例中使用的类](#)部分中列出的数据。要向下导航并获取特斯拉的出生年份和普平的出生城市，我们使用以下表达式：

```
// evals to 1856
int year = (Integer) parser.parseExpression("Birthdate.Year + 1900").getValue(context);

String city = (String) parser.parseExpression("placeOfBirth.City").getValue(context);
```

属性名称的首字母允许不区分大小写。数组和列表的内容通过使用方括号表示法获得，如以下示例所示：

```
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);
```

通过在方括号内指定 **Literals** 键值可以获取 **Map** 的内容。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 **Literals**：

```
// 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. 内联列表

您可以使用 **{}** 表示法直接在表达式中表达列表。

```
// 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);
```

**{}** 本身表示一个空列表。出于性能原因，如果列表本身完全由固定的 **Literals** 组成，则会创建一个常量列表来表示表达式(而不是在每次求值时都构建一个新列表)。

### 4.3.4. 内联 Map

您也可以使用 **{key:value}** 表示法在表达式中直接表达 **Map**。以下示例显示了如何执行此操作：

```
// 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:{year:1856,month:July,day:10}}").getValue(context);
```

`{ : }` 本身就是一张空的 Map。出于性能原因，如果 Map 表本身由固定的 Literals 或其他嵌套的常量结构(列表或 Map 表)组成，则会创建一个常量 Map 表来表示该表达式(而不是在每次求值时都构建一个新的 Map 表)。Map 键的引用是可选的。上面的示例不使用带引号的键。

### 4.3.5. 阵列构造

您可以使用熟悉的 Java 语法来构建数组，可以选择提供一个初始化程序以在构造时填充该数组。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，当前无法提供初始化程序。

### 4.3.6. Methods

您可以使用典型的 Java 编程语法来调用方法。您还可以在 Literals 上调用方法。还支持变量参数。

下面的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

Spring Expression Language 支持以下几种运算符：

- [Relational Operators](#)
- [Logical Operators](#)
- [Mathematical Operators](#)

- 赋值运算符

## Relational Operators

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)。

以下清单显示了一些运算符示例:

```
// evaluates to true
boolean trueValue = parser.parseExpression("2 == 2").getValue(Boolean.class);

// evaluates to false
boolean falseValue = parser.parseExpression("2 < -5.0").getValue(Boolean.class);

// evaluates to true
boolean trueValue = parser.parseExpression("'black' < 'block'").getValue(Boolean.class)
```

### iNote

与 `null` 的大于和小于比较遵循一个简单的规则: `null` 被视为无(不是零)。结果, 任何其他值始终大于 `null` (`x > null` 始终为 `true`), 并且其他任何值都不小于零(`x < null` 总是 `false`)。

如果您更喜欢数字比较, 请避免基于数字的 `null` 比较, 而反对与零的比较(例如 `x > 0` 或 `x < 0`)。

除了标准的关系运算符外, SpEL 还支持 `instanceof` 和基于正则表达式的 `matches` 运算符。以下清单显示了两个示例:

```
// 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);
```

## ⚠Warning

请注意基本类型，因为它们会立即被包装为包装类型，因此，按预期，`1 instanceof T(int)` 的值为 `false`，而 `1 instanceof T(Integer)` 的值为 `true`。

每个符号运算符也可以指定为纯字母等效项。这样可以避免使用的符号对于嵌入表达式的文档类型具有特殊含义的问题(例如在 XML 文档中)。等效的 Literals 是：

- `lt` ( `<` )
- `gt` ( `>` )
- `le` ( `<=` )
- `ge` ( `>=` )
- `eq` ( `==` )
- `ne` ( `!=` )
- `div` ( `/` )
- `mod` ( `%` )
- `not` ( `!` ).

所有的文本运算符都不区分大小写。

## Logical Operators

SpEL 支持以下逻辑运算符：

- `and`
- `or`

- **not**

下面的示例显示如何使用逻辑运算符

```
// -- 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);
```

## Mathematical Operators

您可以在数字和字符串上使用加法运算符。您只能对数字使用减法、乘法和除法运算符。您还可以使用模数(%)和指数幂(^)运算符。强制执行标准运算符优先级。以下示例显示了正在使用的 `math` 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); //
```

```

// 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

```

## 赋值运算符

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。下面的清单显示了使用赋值运算符的两种方法：

```

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. Types

您可以使用特殊的 `T` 运算符来指定 `java.lang.Class` (类型)的实例。静态方法也可以通过使用此运算符来调用。`StandardEvaluationContext` 使用 `TypeLocator` 查找类型，而 `StandardTypeLocator` (可以替换)是在了解 `java.lang` 程序包的情况下构建的。这意味着对 `java.lang` 中的类型的 `T()` 引用不需要完全限定，但所有其他类型引用都必须是完全限定的。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. Constructors

您可以使用 `new` 运算符来调用构造函数。除了基本类型(`int`, `float` 等等)和 `String` 之外，您都应使用完全限定的类名。下面的示例演示如何使用 `new` 运算符调用构造函数：

```
Inventor einstein = p.parseExpression(  
    "new org.springframework.samples.spel.inventor.Inventor('Albert Einstein', 'German')")  
    .getValue(Inventor.class);  
  
//create new inventor instance within add method of List  
p.parseExpression(  
    "Members.add(new org.springframework.samples.spel.inventor.Inventor(  
        'Albert Einstein', 'German'))").getValue(societyContext);
```

### 4.3.10. Variables

您可以使用 `#variableName` 语法在表达式中引用变量。通过在 `EvaluationContext` 实现上使用 `setVariable` 方法来设置变量。以下示例显示了如何使用变量：

```
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 变量

始终定义 `#this` 变量，该变量指向当前评估对象(针对不合格的引用，将对其进行解析)。始终定义 `#root` 变量，并引用根上下文对象。尽管 `#this` 可能随表达式的组成部分的求值而变化，但 `#root` 始终引用根。以下示例显示了如何使用 `#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. Functions

您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。该功能通过 `EvaluationContext` 注册。下面的示例演示如何注册用户定义的函数：

```
Method method = ...;

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();
context.setVariable("myFunction", method);
```

例如，考虑以下用于反转字符串的 Util 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，您可以注册并使用前面的方法，如以下示例所示：

```
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 参考

如果评估上下文已使用 `bean` 解析器配置，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：

```
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，应改为在 Bean 名称前加上 `&` 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. 三元运算符(If-Then-Else)

您可以使用三元运算符在表达式内部执行 if-then-else 条件逻辑。以下清单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` 导致返回字符串值 `'falseExp'`。一个更现实的示例如下：

```
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，请参阅关于 Elvis 运算符的下一部分。

### 4.3.14. Elvisoperator

Elvis 运算符是三元运算符语法的简化，并且以[Groovy](#)语言使用。使用三元运算符语法，通常必须将变量重复两次，如以下示例所示：

```
String name = "Elvis Presley";
String displayName = (name != null ? name : "Unknown");
```

取而代之的是，您可以使用 **Elvis** 运算符(其命名类似于 Elvis 的发型)。以下示例显示了如何使用 **Elvis** 运算符：

```
ExpressionParser parser = new SpelExpressionParser();

String name = parser.parseExpression("name?:'Unknown'").getValue(String.class);
System.out.println(name); // 'Unknown'
```

以下清单显示了一个更复杂的示例：

```
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,
System.out.println(name); // Nikola Tesla

tesla.setName(null);
name = parser.parseExpression("Name?:'Elvis Presley'").getValue(context, tesla, String.
System.out.println(name); // Elvis Presley
```

### iNote

您可以使用 **Elvis** 运算符在表达式中应用默认值。以下示例显示了如何在 **@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**，否则将注入 25.

## 4.3.15. 安全导航操作员

安全导航操作符用于避免 **NullPointerException**，并且来自[Groovy](#)语言。通常，当您引用一个对象时，可能需要在访问该对象的方法或属性之前验证其是否为 **null**。为了避免这种情况，安全导航运算符返回 **null** 而不是引发异常。下面的示例演示如何使用安全导航操作符：

```
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. collections 选择

选择是一种强大的表达语言功能，可让您通过从源集合中选择条目来将其转换为另一个集合。

选择使用 `.?[selectionExpression]` 的语法。它过滤集合并返回一个包含原始元素子集的新集合。例如，通过选择，我们可以轻松地获得塞尔维亚发明者的列表，如以下示例所示：

```
List<Inventor> list = (List<Inventor>) parser.parseExpression(
    "Members.? [Nationality == 'Serbian']").getValue(societyContext);
```

在列表和 Map 上都可以选择。对于列表，将针对每个单独的列表元素评估选择标准。针对 Map，针对每个 Map 条目 (Java 类型 `Map.Entry` 的对象) 评估选择标准。每个 Map 条目都有其键和值，可作为属性进行访问以供选择。

以下表达式返回一个新 Map，该 Map 由原始 Map 中条目值小于 27 的那些元素组成：

```
Map newMap = parser.parseExpression("map.? [value < 27]").getValue();
```

除了返回所有选定的元素外，您只能检索第一个或最后一个值。要获得与选择匹配的第一个条目，语法为 `.^ [selectionExpression]`。要获得最后的匹配选择，语法为 `.${selectionExpression}`。

### 4.3.17. 集合投影

投影使集合可以驱动子表达式的求值，结果是一个新的集合。投影的语法为

`.! [projectionExpression]`。例如，假设我们有一个发明家列表，但想要他们出生的城市的列

表。实际上，我们希望为发明人列表中的每个条目评估“placeOfBirth.city”。下面的示例使用投影来做到这一点：

```
// returns ['Smiljan', 'Idvor']
List placesOfBirth = (List)parser.parseExpression("Members.! [placeOfBirth.city]");
```

您还可以使用 `Map` 来驱动投影，在这种情况下，将根据 `Map` 中的每个条目(以 Java `Map.Entry` 表示)来评估投影表达式。跨 `Map` 的投影结果是一个列表，其中包含针对每个 `Map` 条目的投影表达式的评估。

### 4.3.18. 表达式模板

表达式模板允许将 `Literals` 文本与一个或多个评估块混合。每个评估块均以您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，如以下示例所示：

```
String randomPhrase = parser.parseExpression(
    "random number is #{T(java.lang.Math).random()}" ,
    new TemplateParserContext().getValue(String.class);

// evaluates to "random number is 0.7038186818312008"
```

通过将 `Literals` 文本 `'random number is '` 与计算 `#{}` 定界符内的表达式的结果(在这种情况下，是调用 `random()` 方法的结果)进行连接来评估字符串。`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;
    }
}
```

## 4.4. 示例中使用的类

本节列出了本章示例中使用的类。

### 例子 1. Inventor.java

```
package org.springframework.samples.spel.inventor;

import java.util.Date;
import java.util.GregorianCalendar;

public class Inventor {

    private String name;
    private String nationality;
    private String[] inventions;
    private Date birthdate;
    private PlaceOfBirth placeOfBirth;

    public Inventor(String name, String nationality) {
        GregorianCalendar c= new GregorianCalendar();
        this.name = name;
        this.nationality = nationality;
        this.birthdate = c.getTime();
    }

    public Inventor(String name, Date birthdate, String nationality) {
        this.name = name;
        this.nationality = nationality;
        this.birthdate = birthdate;
    }

    public Inventor() {
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getNationality() {
        return nationality;
    }

    public void setNationality(String nationality) {
        this.nationality = nationality;
    }

    public Date getBirthdate() {
        return birthdate;
    }

    public void setBirthdate(Date birthdate) {
```

```
        this.birthdate = birthdate;
    }

    public PlaceOfBirth getPlaceOfBirth() {
        return placeOfBirth;
    }

    public void setPlaceOfBirth(PlaceOfBirth placeOfBirth) {
        this.placeOfBirth = placeOfBirth;
    }

    public void setInventions(String[] inventions) {
        this.inventions = inventions;
    }

    public String[] getInventions() {
        return inventions;
    }
}
```

### 例子 2. PlaceOfBirth.java

```
package org.springframework.samples.spel.inventor;

public class PlaceOfBirth {

    private String city;
    private String country;

    public PlaceOfBirth(String city) {
        this.city=city;
    }

    public PlaceOfBirth(String city, String country) {
        this(city);
        this.country = country;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String s) {
        this.city = s;
    }

    public String getCountry() {
        return country;
    }

    public void setCountry(String country) {
        this.country = country;
    }
}
```

### 例子 3. Society.java

```
package org.springframework.samples.spel.inventor;

import java.util.*;

public class Society {

    private String name;

    public static String Advisors = "advisors";
    public static String President = "president";

    private List<Inventor> members = new ArrayList<Inventor>();
    private Map officers = new HashMap();

    public List getMembers() {
        return members;
    }

    public Map getOfficers() {
        return officers;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public boolean isMember(String name) {
        for (Inventor inventor : members) {
            if (inventor.getName().equals(name)) {
                return true;
            }
        }
        return false;
    }
}
```

## 5. 使用 Spring 进行面向方面的编程

面向方面的编程(AOP)通过提供另一种思考程序结构的方式来补充面向对象的编程(OOP)。OOP 中模块化的关键单元是类，而在 AOP 中模块化是方面。方面使关注点(例如事务 Management)的模块化可以跨越多种类型和对象。(这种关注在 AOP 文献中通常被称为“跨领域”关注。)

Spring 的关键组件之一是 AOP 框架。尽管 Spring IoC 容器不依赖于 AOP(这意味着您不需要使用 AOP)，但 AOP 是对 Spring IoC 的补充，以提供功能非常强大的中间件解决方案。

Spring 2.0 AOP

Spring 2.0 引入了一种使用 [schema-based approach](#) 或 [@AspectJ](#) 注解样式来编写自定义方面的更简单，更强大的方法。这两种样式都提供了完全类型化的建议，并使用了 [AspectJ](#) 切入点语言，同时仍使用 Spring AOP 进行编织。

本章讨论基于 Spring 2.0 模式和基于[@AspectJ](#) 的 AOP 支持。[接下来的章节](#) 中讨论了 Spring 1.2 应用程序中常见的较低级别的 AOP 支持。

AOP 在 Spring Framework 中用于：

- 提供声明性企业服务，尤其是作为 EJB 声明性服务的替代品。最重要的此类服务是[声明式 TransactionManagement](#)。
- 让用户实现自定义方面，以 AOP 补充其对 OOP 的使用。

#### iNote

如果您只对通用声明性服务或其他预打包的声明性中间件服务(例如池)感兴趣，则无需直接使用 Spring AOP，并且可以跳过本章的大部分内容。

## 5.1. AOP 概念

让我们首先定义一些重要的 AOP 概念和术语。这些术语不是特定于 Spring 的。不幸的是，AOP 术语并不是特别直观。但是，如果 Spring 使用其自己的术语，那将更加令人困惑。

- 方面：涉及多个类别的关注点的模块化。事务 Management 是企业 Java 应用程序中横切关注的一个很好的例子。在 Spring AOP 中，方面是通过使用常规类([schema-based approach](#))或使用 [@Aspect](#) 注解([@AspectJ style](#))Comments 的常规类来实现的。
- 连接点：在程序执行过程中的一点，例如方法的执行或异常的处理。在 Spring AOP 中，连接点始终代表方法的执行。
- 建议：方面在特定的连接点处采取的操作。不同类型的建议包括“周围”，“之前”和“之后”建议。(建议类型将在后面讨论)包括 Spring 在内的许多 AOP 框架都将建议建模为拦截器，并在连接点周围维护一系列拦截器。

- 切入点：与连接点匹配的谓词。建议与切入点表达式关联，并在与该切入点匹配的任何连接点处运行(例如，执行具有特定名称的方法)。切入点表达式匹配的连接点的概念是 AOP 的核心，并且 Spring 默认使用 AspectJ 切入点表达语言。
- 简介：代表类型声明其他方法或字段。Spring AOP 允许您向任何建议的对象引入新的接口(和相应的实现)。例如，您可以使用简介使 Bean 实现 `IsModified` 接口，以简化缓存。(在 AspectJ 社区中，介绍被称为类型间声明。)
- 目标对象：一个或多个方面建议的对象。也称为“建议对象”。由于 Spring AOP 是使用运行时代理实现的，因此该对象始终是代理对象。
- AOP 代理：由 AOP 框架创建的一个对象，用于实现方面协定(建议方法执行等)。在 Spring Framework 中，AOP 代理是 JDK 动态代理或 CGLIB 代理。
- 编织：将方面与其他应用程序类型或对象链接以创建建议的对象。这可以在编译时(例如，使用 AspectJ 编译器)，加载时或在运行时完成。像其他纯 Java AOP 框架一样，Spring AOP 在运行时执行编织。

Spring AOP 包括以下类型的建议：

- 在建议之前：在连接点之前运行的建议，但是它不能阻止执行流程前进到连接点(除非它引发异常)。
- 返回建议后：在连接点正常完成后要运行的建议(例如，如果方法返回而没有引发异常)。
- 抛出建议后：如果方法因抛出异常而退出，则执行建议。
- 建议之后(最终)：无论连接点退出的方式如何(正常或特殊返回)，均应执行建议。
- 围绕建议：围绕连接点的建议，例如方法调用。这是最有力的建议。周围建议可以在方法调用之前和之后执行自定义行为。它还负责选择是返回连接点还是通过返回其自身的返回值或引发异常来捷径建议的方法执行。

围绕建议是最通用的建议。由于 Spring AOP 与 AspectJ 一样，提供了各种建议类型，因此我们建议您使用功能最弱的建议类型，以实现所需的行为。例如，如果您只需要使用方法的返回值更新缓存，则最好使用返回后的建议而不是周围的建议，尽管周围的建议可以完成相同的事情。使用最具

体的建议类型可以提供更简单的编程模型，并减少出错的可能性。例如，您不需要在用于周围建议的 `JoinPoint` 上调用 `proceed()` 方法，因此，您不会失败。

在 Spring 2.0 中，所有建议参数都是静态类型的，因此您可以使用适当类型(例如，从方法执行返回值的类型)而不是 `Object` 数组的建议参数。

切入点匹配的连接点的概念是 AOP 的关键，它与仅提供拦截功能的旧技术有所不同。切入点使建议的目标独立于面向对象的层次结构。例如，您可以将提供声明性事务 Management 的环绕建议应用于跨越多个对象(例如服务层中的所有业务操作)的一组方法。

## 5.2. SpringAOP 能力和目标

Spring AOP 是用纯 Java 实现的。不需要特殊的编译过程。Spring AOP 不需要控制类加载器的层次结构，因此适合在 Servlet 容器或应用程序服务器中使用。

Spring AOP 当前仅支持方法执行连接点(建议在 Spring Bean 上执行方法)。尽管可以在不破坏核心 Spring AOP API 的情况下添加对字段拦截的支持，但并未实现字段拦截。如果需要建议字段访问和更新连接点，请考虑使用诸如 AspectJ 之类的语言。

Spring AOP 的 AOP 方法不同于大多数其他 AOP 框架。目的不是提供最完整的 AOP 实现(尽管 Spring AOP 相当强大)。相反，其目的是在 AOP 实现和 Spring IoC 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，通常将 Spring Framework 的 AOP 功能与 Spring IoC 容器结合使用。通过使用常规 `bean` 定义语法来配置方面(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。使用 Spring AOP 无法轻松或高效地完成某些事情，例如建议非常细粒度的对象(通常是域对象)。在这种情况下，AspectJ 是最佳选择。但是，我们的经验是，Spring AOP 为企业 Java 应用程序中适合 AOP 的大多数问题提供了出色的解决方案。

Spring AOP 从未努力与 AspectJ 竞争以提供全面的 AOP 解决方案。我们认为，基于代理的框架(如 Spring AOP)和成熟的框架(如 AspectJ)都是有价值的，它们是互补的，而不是竞争。Spring 将 AspectJ 无缝集成了 Spring AOP 和 IoC，以在基于 Spring 的一致应用程序架构中支持 AOP 的所有使用。这种集成不会影响 Spring AOP API 或 AOP Alliance API。Spring AOP 仍然向后兼容。有

关于 Spring AOP API 的讨论, 请参见[接下来的章节](#)。

### iNote

Spring 框架的中心宗旨之一是非侵入性。这是一个想法, 不应强迫您将特定于框架的类和接口引入业务或域模型。但是, 在某些地方, Spring Framework 确实为您提供了将特定于 Spring Framework 的依赖项引入代码库的选项。提供此类选项的理由是, 在某些情况下, 以这种方式阅读或编码某些特定功能可能会变得更加容易。但是, Spring 框架(几乎)总是为您提供选择: 您可以自由地就哪个选项最适合您的特定用例或场景做出明智的决定。

与本章相关的一种选择是选择哪种 AOP 框架(以及哪种 AOP 样式)。您可以选择 AspectJ 和 / 或 Spring AOP。您还可以选择@AspectJComments 样式方法或 Spring XML 配置样式方法。本章选择首先介绍@AspectJ 风格的方法这一事实不应被视为表明 Spring 团队比 Spring XML 配置风格更喜欢@AspectJComments 风格的方法。

有关每种样式的“为什么和为什么”的更完整讨论, 请参见[选择要使用的 AOP 声明样式](#)。

## 5.3. AOP 代理

Spring AOP 默认将标准 JDK 动态代理用于 AOP 代理。这使得可以代理任何接口(或一组接口)。

Spring AOP 也可以使用 CGLIB 代理。这对于代理类而不是接口是必需的。默认情况下, 如果业务对象未实现接口, 则使用 CGLIB。由于对接口而不是对类进行编程是一种好习惯, 因此业务类通常实现一个或多个业务接口。在某些情况下(可能极少发生), 您需要建议未在接口上声明的方法, 或者需要将代理对象作为具体类型传递给方法, 则可以使用[强制使用 CGLIB](#)。

掌握 Spring AOP 是基于代理的这一事实非常重要。有关此实现细节实际含义的彻底检查, 请参见[了解 AOP 代理](#)。

## 5.4. @AspectJ 支持

@AspectJ 是一种将方面声明为带有 Comments 的常规 Java 类的样式。@AspectJ 样式是[AspectJ project](#)作为 AspectJ 5 版本的一部分引入的。Spring 使用 AspectJ 提供的用于切入点解析和匹配的

库来解释与 AspectJ 5 相同的 Comments。但是，AOP 运行时仍然是纯 Spring AOP，并且不依赖于 AspectJ 编译器或编织器。

#### iNote

使用 AspectJ 编译器和 weaver 可以使用完整的 AspectJ 语言，并将在[在 Spring 应用程序中使用 AspectJ](#)中进行讨论。

### 5.4.1. 启用@AspectJ 支持

要在 Spring 配置中使用@AspectJ 方面，您需要启用 Spring 支持以基于@AspectJ 方面配置 Spring AOP，并根据这些方面是否建议对它们进行自动代理。通过自动代理，我们的意思是，如果 Spring 确定一个或多个方面建议一个 bean，它会自动为该 bean 生成一个代理来拦截方法调用并确保按需执行建议。

可以使用 XML 或 Java 样式的配置来启用@AspectJ 支持。无论哪种情况，都需要确保 AspectJ 的 `aspectjweaver.jar` 库位于应用程序的 Classpath(版本 1.8 或更高版本)上。该库在 AspectJ 发行版的 `lib` 目录中或从 Maven Central 存储库中可用。

#### 通过 Java 配置启用@AspectJ 支持

要使用 Java `@Configuration` 启用@AspectJ 支持，请添加 `@EnableAspectJAutoProxy` 注解，如以下示例所示：

```
@Configuration  
@EnableAspectJAutoProxy  
public class AppConfig {  
}
```

#### 通过 XML 配置启用@AspectJ 支持

要通过基于 XML 的配置启用@AspectJ 支持，请使用 `aop:aspectj-autoproxy` 元素，如以下示例所示：

```
<aop:aspectj-autoproxy/>
```

假设您使用[基于 XML 模式的配置](#)中所述的架构支持。有关如何在 `aop` 名称空间中导入标签的信息，请参见[AOP 模式](#)。

### 5.4.2. 声明一个方面

启用`@Aspect` 支持后，Spring 会自动检测到在应用程序上下文中使用`@Aspect` 方面(具有`@Aspect` 注解)的类定义的任何 bean，并用于配置 Spring AOP。接下来的两个示例显示了一个不太有用方面所需的最小定义。

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>
```

这两个示例中的第二个示例显示了 `NotVeryUsefulAspect` 类定义，该类定义带有

`org.aspectj.lang.annotation.Aspect` Comments；

```
package org.xyz;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class NotVeryUsefulAspect {
```

方面(带有 `@Aspect` Comments 的类)可以具有方法和字段，与任何其他类相同。它们还可以包含切入点、建议和引入(类型间)声明。

#### 1 Autodetecting aspects through component scanning

您可以将方面类注册为 Spring XML 配置中的常规 bean，也可以通过 Classpath 扫描来自动

检测它们-与其他任何 SpringManagement 的 bean 一样。但是, 请注意, `@Aspect`

Comments 不足以在 Classpath 中进行自动检测。为此, 您需要添加一个单独的 `@Component` 注解(或者, 或者, 按照 Spring 的组件扫描程序的规则, 有条件的自定义构造型注解)。

### ①Advising aspects with other aspects?

在 Spring AOP 中, 方面本身不能成为其他方面的建议目标。类上的 `@Aspect` Comments 将其标记为一个方面, 因此将其从自动代理中排除。

### 5.4.3. 声明切入点

切入点确定了感兴趣的连接点, 从而使我们能够控制执行建议的时间。Spring AOP 仅支持 Spring Bean 的方法执行连接点, 因此您可以将切入点视为与 Spring Bean 上的方法执行相匹配。切入点声明由两部分组成: 一个包含名称和任何参数的签名, 以及一个切入点表达式, 该切入点表达式准确确定我们感兴趣的方法执行。在 AOP 的`@Aspect` 注解样式中, 常规方法定义提供了切入点签名。, 并通过使用 `@Pointcut` Comments 指示切入点表达式(用作切入点签名的方法必须具有 `void` 返回类型)。

一个示例可能有助于使切入点签名和切入点表达式之间的区别变得清晰。下面的示例定义一个名为 `anyOldTransfer` 的切入点, 该切入点与任何名为 `transfer` 的方法的执行相匹配:

```
@Pointcut("execution(* transfer(..))")// the pointcut expression
private void anyOldTransfer() {}// the pointcut signature
```

形成 `@Pointcut` 注解的值的切入点表达式是一个常规的 AspectJ 5 切入点表达式。有关 AspectJ 的切入点语言的完整讨论, 请参见[AspectJ 编程指南](#)(以及 extensions[AspectJ 5 开发人员的笔记本](#))或有关 AspectJ 的其中一本书(例如 Colyer 等人的\* Eclipse AspectJ 或 AspectJ in Action \*), 由 Ramnivas Laddad)。

## 支持的切入点指示符

Spring AOP 支持以下在切入点表达式中使用的 AspectJ 切入点指示符(PCD):

- `execution` : 用于匹配方法执行的连接点。这是使用 Spring AOP 时要使用的主要切入点指示符。
- `within` : 将匹配限制为某些类型内的连接点(使用 Spring AOP 时，在匹配类型内声明的方法的执行)。
- `this` : 将匹配限制为连接点(使用 Spring AOP 时方法的执行)，其中 bean 引用(Spring AOP 代理)是给定类型的实例。
- `target` : 将目标对象(正在代理的应用程序对象)是给定类型的实例的连接点(使用 Spring AOP 时，方法的执行)限制为匹配。
- `args` : 将参数限制为给定类型的实例的连接点(使用 Spring AOP 时方法的执行)限制匹配。
- `@target` : 将执行对象的类具有给定类型的 Comments 的连接点(使用 Spring AOP 时，方法的执行)限制为匹配。
- `@args` : 限制匹配的连接点(使用 Spring AOP 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 Comments。
- `@within` : 将匹配限制为具有给定 Comments 的类型的连接点(使用 Spring AOP 时，使用给定 Comments 的类型中声明的方法的执行)。
- `@annotation` : 将匹配限制为连接点的主题(在 Spring AOP 中正在执行的方法)具有给定 Comments 的连接点。

## 其他切入点类型

完整的 AspectJ 切入点语言支持 Spring 中不支持的其他切入点指示符：`call`，`get`，`set`，  
`preinitialization`，`staticinitialization`，`initialization`，`handler`，

`adviceexecution`, `withincode`, `cflow`, `cflowbelow`, `if`, `@this` 和 `@withincode`。在 Spring AOP 解释的切入点表达式中使用这些切入点指示符会导致抛出 `IllegalArgumentException`。

Spring AOP 支持的切入点指示符集合可能会在将来的版本中扩展，以支持更多的 AspectJ 切入点指示符。

因为 Spring AOP 将匹配限制为仅方法执行连接点，所以前面对切入点指示符的讨论所提供的定义比 AspectJ 编程指南中的定义要窄。此外，AspectJ 本身具有基于类型的语义，并且在执行连接点处 `this` 和 `target` 引用同一对象：执行该方法的对象。Spring AOP 是基于代理的系统，可区分代理对象本身(绑定到 `this`)和代理后面的目标对象(绑定到 `target`)。

### iNote

由于 Spring 的 AOP 框架基于代理的性质，因此根据定义，不会拦截目标对象内的调用。对于 JDK 代理，只能拦截代理上的公共接口方法调用。使用 CGLIB 时，将拦截代理上的公共方法和受保护方法(甚至在必要时对程序包可见的方法)。但是，通常应通过公共签名设计通过代理进行的常见交互。

请注意，切入点定义通常与任何拦截方法匹配。如果严格地将切入点设置为仅公开使用，即使在 CGLIB 代理方案中通过代理存在潜在的非公开交互作用，也需要相应地进行定义。

如果您的拦截需要在目标类中包括方法调用甚至构造函数，请考虑使用 Spring 驱动的[原生 AspectJ 编织](#)而不是 Spring 的基于代理的 AOP 框架。这构成了具有不同 Feature 的 AOP 使用模式，因此请确保在做出决定之前先熟悉编织。

Spring AOP 还支持名为 `bean` 的附加 PCD。使用 PCD，可以将连接点的匹配限制为特定的命名 Spring Bean 或一组命名 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 构建了一些命名约定, 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 也可以与 `&&` (和), `||` (或)和 `!` (否定)运算符一起使用。

### iNote

`bean` PCD 仅在 Spring AOP 中受支持, 而在本机 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, 不适用于 `@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 受其限制)上运行。基于实例的切入点指示符是 Spring 基于代理的 AOP 框架及其与 Spring bean 工厂的紧密集成的一种特殊功能, 在该工厂中自然而直接地通过名称来标识特定的 bean。

## 组合切入点表达式

您可以组合切入点表达式, 可以使用 `&&`, `||` 和 `!` 进行组合。您也可以按名称引用切入点表达式。以下示例显示了三个切入点表达式:

```
@Pointcut("execution(public * *(..))")  
private void anyPublicOperation() {} (1)  
  
@Pointcut("within(com.xyz.someapp.trading..*)")  
private void inTrading() {} (2)  
  
@Pointcut("anyPublicOperation() && inTrading()")  
private void tradingOperation() {} (3)
```

- (1) `anyPublicOperation` 匹配方法执行联接点是否表示任何公共方法的执行。
- (2) `inTrading` 如果 Transaction 模块中有方法执行则匹配。
- (3) `tradingOperation` 匹配, 如果方法执行代表 Transaction 模块中的任何公共方法。

最佳实践是从较小的命名组件中构建更复杂的切入点表达式, 如先前所示。按名称引用切入点时, 将应用常规的 Java 可见性规则(您可以看到相同类型的私有切入点, 层次结构中受保护的切入点, 任何位置的公共切入点, 等等)。可见性不影响切入点匹配。

## 共享通用切入点定义

在使用企业应用程序时，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。我们建议为此定义一个“SystemArchitecture”方面，以捕获常见的切入点表达式。这样的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() {}

}

```

您可以在需要切入点表达式的任何地方引用在这样的方面中定义的切入点。例如，要使服务层具有事务性，您可以编写以下内容：

```

<aop:config>
    <aop:advisor
        pointcut="com.xyz.someapp.SystemArchitecture.businessService()"
        advice-ref="tx-advice"/>
</aop:config>

<tx:advice id="tx-advice">
    <tx:attributes>
        <tx:method name="*" propagation="REQUIRED" />
    </tx:attributes>
</tx:advice>

```

[基于架构的 AOP 支持](#) 中讨论了 `<aop:config>` 和 `<aop:advisor>` 元素。`Transaction` 元素在 [Transaction Management](#) 中讨论。

## Examples

Spring AOP 用户可能最常使用 `execution` 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
    throws-pattern?)
```

除了返回类型模式(前面的代码段中为 `ret-type-pattern`)，名称模式和参数模式以外的所有其他部分都是可选的。返回类型模式确定该方法的返回类型必须是什么才能使连接点匹配。 `*` 最常用作返回类型模式。它匹配任何返回类型。仅当方法返回给定类型时，完全合格的类型名称才匹配。名称模式与方法名称匹配。您可以将 `*` 通配符用作名称模式的全部或一部分。如果指定了声明类型模式，请在其末尾添加 `.` 并将其连接到名称模式组件。参数模式稍微复杂一些：`()` 匹配不带参数的方法，而 `(...)` 匹配任意数量(零个或多个)的参数。`(*)` 模式与采用任何类型的一个参数的方法

法匹配。`(* ,String)` 与采用两个参数的方法匹配。第一个可以是任何类型，而第二个必须是 `String`。有关更多信息，请查阅 AspectJ 编程指南的[Language Semantics](#)部分。

以下示例显示了一些常用的切入点表达式：

- 任何公共方法的执行：

```
execution(public * *(..))
```

- 名称以 `set` 开头的任何方法的执行：

```
execution(* set*(..))
```

- `AccountService` 接口定义的任何方法的执行：

```
execution(* com.xyz.service.AccountService.*(..))
```

- `service` 软件包中定义的任何方法的执行：

```
execution(* com.xyz.service.*.*(..))
```

- 服务包或其子包之一中定义的任何方法的执行：

```
execution(* com.xyz.service..*.*(..))
```

- 服务包中的任何连接点(仅在 Spring AOP 中执行方法)：

```
within(com.xyz.service.*)
```

- 服务包或其子包之一中的任何连接点(仅在 Spring AOP 中执行方法)：

```
within(com.xyz.service..*)
```

- 代理实现 `AccountService` 接口的任何连接点(仅在 Spring AOP 中执行方法)：

```
this(com.xyz.service.AccountService)
```

### iNote

“this”通常以绑定形式使用。有关如何在建议正文中使代理对象可用的信息，请参阅[Declaring Advice](#)上的部分。

- 目标对象实现 `AccountService` 接口的任何连接点(仅在 Spring AOP 中执行方法):

```
target(com.xyz.service.AccountService)
```

### iNote

“目标”通常以绑定形式使用。有关如何使建议对象在建议正文中可用的信息，请参见[Declaring Advice](#)部分。

- 任何采用单个参数且运行时传递的参数为 `Serializable` 的连接点(仅在 Spring AOP 中是方法执行):

```
args(java.io.Serializable)
```

### iNote

“args”更通常以绑定形式使用。有关如何使方法参数在建议正文中可用的信息，请参见[Declaring Advice](#)部分。

请注意，此示例中给出的切入点不同于 `execution(* *(java.io.Serializable))`。如果在运行时传递的参数为 `Serializable`，则 args 版本匹配，如果方法签名声明单个类型为 `Serializable` 的参数，则执行版本匹配。

- 目标对象带有 `@Transactional` 注解的任何连接点(仅在 Spring AOP 中执行方法):

```
@target(org.springframework.transaction.annotation.Transactional)
```

### iNote

您也可以在绑定形式中使用“@target”。有关如何使 Comments 对象在建议正文中可用的信息，请参见[Declaring Advice](#)部分。

- 目标对象的声明类型具有 `@Transactional` Comments 的任何连接点(仅在 Spring AOP 中是方法执行):

```
@within(org.springframework.transaction.annotation.Transactional)
```

### iNote

您也可以在绑定形式中使用“@within”。有关如何使 Comments 对象在建议正文中可用的信息，请参见[Declaring Advice](#)部分。

- 执行方法带有 `@Transactional` 注解的任何连接点(仅在 Spring AOP 中是方法执行):

```
@annotation(org.springframework.transaction.annotation.Transactional)
```

### iNote

您也可以在绑定形式中使用“@annotation”。有关如何使 Comments 对象在建议正文中可用的信息，请参见[Declaring Advice](#)部分。

- 任何采用单个参数且传递的参数的运行时类型具有 `@Classified` Comments 的连接点(仅在 Spring AOP 中是方法执行)。

```
@args(com.xyz.security.Classified)
```

## iNote

您也可以在绑定形式中使用“@args”。请参见[Declaring Advice](#)部分，如何在建议正文中使Comments对象可用。

- 名为`tradeService`的Spring bean上的任何连接点(仅在Spring AOP中执行方法):

```
bean(tradeService)
```

- Spring Bean上具有与通配符表达式`*Service`匹配的名称的任何连接点(仅在Spring AOP中是方法执行):

```
bean(*Service)
```

## 编写好的切入点

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(静态或动态)匹配给定的切入点是一个昂贵的过程。(动态匹配意味着无法从静态分析中完全确定匹配，并且在代码中进行测试以确定在运行代码时是否存在实际匹配)。首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，切入点以DNF(析取范式)重写，并且对切入点的组件进行排序，以便首先检查那些较便宜的组件。这意味着您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Sequences提供它们。

但是，AspectJ只能使用所告诉的内容。为了获得最佳的匹配性能，您应该考虑他们试图达到的目标，并在定义中尽可能缩小匹配的搜索空间。现有的指示符自然分为三类之一：同类，作用域和上下文：

- 亲切的指示者选择一种特定的连接点：`execution`，`get`，`set`，`call`和`handler`。
- 作用域指定者选择一组感兴趣的连接点(可能是多种)：`within`和`withincode`
- 上下文指示符根据以下上下文进行匹配(并可选地绑定)：`this`，`target`和`@annotation`

编写正确的切入点至少应包括前两种类型(种类和作用域)。您可以包括上下文指示符以根据连接点

上下文进行匹配，也可以绑定该上下文以在建议中使用。仅提供同类的标识符或仅提供上下文的标识符是可行的，但是由于额外的处理和分析，可能会影响编织性能(使用的时间和内存)。范围指定者的匹配非常快，使用它们的使用意味着 AspectJ 可以非常迅速地消除不应进一步处理的连接点组。一个好的切入点应该始终包括一个切入点。

## 5.4.4. 宣告建议

建议与切入点表达式关联，并且在切入点匹配的方法执行之前，之后或周围运行。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

您可以使用 `@Before` Comments 在方面中在建议之前声明：

```
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() {
        // ...
    }
}
```

如果使用就地切入点表达式，则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() {
        // ...
    }
}
```

### 返回建议后

返回建议后，当匹配的方法执行正常返回时，运行建议。您可以使用 `@AfterReturning` 注解进行

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() {
        // ...
    }

}
```

### iNote

您可以在同一方面内拥有多个建议声明(以及其他成员)。在这些示例中，我们仅显示单个建议声明，以集中每个建议的效果。

有时，您需要在建议正文中访问返回的实际值。您可以使用 `@AfterReturning` 的形式绑定返回值以获取该访问权限，如以下示例所示：

```
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` 方法中的参数名称相对应。当方法执行返回时，该返回值将作为相应的参数值传递到通知方法。`returning` 子句还将匹配仅限制为返回指定类型值(在这种情况下为 `Object`，该值与任何返回值匹配)的那些方法执行。

请注意，返回建议后使用时，不可能返回完全不同的参考。

## 提出建议后

抛出建议后，当匹配的方法执行通过抛出异常退出时运行建议。您可以使用 `@AfterThrowing` 注解进行声明，如以下示例所示：

```
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() {
        // ...
    }

}
```

通常，您希望通知仅在引发给定类型的异常时才运行，并且您通常还需要访问通知正文中的异常。您可以使用 `throwing` 属性来限制匹配(如果需要)(否则，请使用 `Throwable` 作为异常类型)，并将抛出的异常绑定到 `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` 方法中的参数名称相对应。当通过抛出异常退出方法执行时，该异常将作为相应的参数值传递给通知方法。`throwing` 子句还将匹配仅限制为抛出指定类型(在本例中为 `DataAccessException`)的异常的方法执行。

## (最后)建议后

当匹配的方法执行退出时，通知(最终)运行。通过使用 `@After` Comments 进行声明。之后必须准

备处理正常和异常返回条件的建议。它通常用于释放资源和类似目的。以下示例显示了最终建议后的用法：

```
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

最后一种建议是围绕建议。围绕建议在匹配方法的执行过程中“围绕”运行。它有机会在方法执行之前和之后进行工作，并确定何时，如何以及什至根本不执行该方法。如果需要以线程安全的方式(例如，启动和停止计时器)在方法执行之前和之后共享状态，则通常使用绕行建议。始终使用最不符合要求的建议形式(即，在建议可以使用之前，不要在建议周围使用)。

周围的建议通过使用 `@Around` Comments 来声明。咨询方法的第一个参数必须为

`ProceedingJoinPoint` 类型。在建议的正文中，在 `ProceedingJoinPoint` 上调用 `proceed()` 会使底层方法执行。`proceed` 方法也可以传入 `Object[]`。数组中的值用作方法执行时的参数。

### ①Note

对于由 AspectJ 编译器编译的周围建议，使用 `Object[]` 调用时 `proceed` 的行为与 `proceed` 的行为稍有不同。对于使用传统的 AspectJ 语言编写的环绕通知，传递给 `proceed` 的参数数量必须与传递给环绕通知的参数数量(而不是基础连接点采用的参数数量)相匹配，并且传递给值的值必须与给定的参数位置会取代该值绑定到的实体的连接点处的原始值(不要担心，如果这现在没有意义)。Spring 采取的方法更简单，并且更适合其基于代理的，仅执行的语义。如果您编译为 Spring 编写的`@AspectJ` 方面，并将 `proceed` 与 AspectJ 编译器和 weaver 的参数一起使用，则只需要意识到这种区别。有一种方法可以编写在

Spring AOP 和 AspectJ 之间 100% 兼容的方面，这在[以下关于建议参数的部分](#)中进行了讨论。  
。

以下示例显示了如何使用周围建议：

```
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;
    }
}
```

周围建议返回的值是该方法的调用者看到的返回值。例如，如果一个简单的缓存方面有一个值，则可以从缓存中返回一个值；如果没有，则调用 `proceed()`。请注意，`proceed` 可能在周围建议的正文中被调用一次，多次或完全不被调用。所有这些都是合法的。

## Advice Parameters

Spring 提供了完全类型化的建议，这意味着您可以在建议签名中声明所需的参数（如我们先前在返回和抛出示例中所见），而不是一直使用 `Object[]` 数组。我们将在本节的后面部分介绍如何使参数和其他上下文值可用于建议主体。首先，我们看一下如何编写通用建议，以了解该建议当前建议的方法。

### 访问当前的 JoinPoint

任何通知方法都可以将类型 `org.aspectj.lang.JoinPoint` 的参数声明为第一个参数（请注意，在周围的通知中必须声明类型 `JoinPoint` 的子类 `ProceedingJoinPoint` 的第一个参数）。

`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()`：返回建议使用的方法的描述。
- `toString()`：打印有关所建议方法的有用描述。

有关详情，请参见[javadoc](#)。

## 将参数传递给建议

我们已经看到了如何绑定返回的值或异常值(在返回之后和引发建议之后使用)。要使参数值可用于建议正文，可以使用 `args` 的绑定形式。如果在 `args` 表达式中使用参数名称代替类型名称，则在调用建议时会将相应参数的值作为参数值传递。一个例子应该使这一点更清楚。假设您要建议执行以 `Account` 对象作为第一个参数的 DAO 操作，并且您需要访问建议正文中的帐户。您可以编写以下内容：

```
@Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation() && args(account,...)")
public void validateAccount(Account account) {
    // ...
}
```

切入点表达式的 `args(account,...)` 部分有两个作用。首先，它将匹配限制为仅方法采用至少一个参数并且传递给该参数的参数是 `Account` 的实例的方法执行。其次，它通过 `account` 参数使实际的 `Account` 对象可用于建议。

编写此文件的另一种方法是声明一个切入点，当切入点 `Account` 对象值与连接点匹配时，该切入点“提供”，然后从建议中引用命名切入点。如下所示：

```
@Pointcut("com.xyz.myapp.SystemArchitecture.dataAccessOperation() && args(account,...)")
private void accountDataAccessOperation(Account account) {}

@Before("accountDataAccessOperation(account)")
```

```
public void validateAccount(Account account) {  
    // ...  
}
```

有关更多详细信息，请参见 [AspectJ 编程指南](#)。

代理对象(`this`)，目标对象(`target`)和 `Comments`(`@within`，`@target`，`@annotation` 和 `@args`)都可以以类似的方式绑定。接下来的两个示例显示如何匹配使用 `@Auditable` `Comments` 的方法的执行并提取审核代码：

这两个示例中的第一个显示了 `@Auditable` 注解的定义：

```
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.METHOD)  
public @interface Auditable {  
    AuditCode value();  
}
```

两个示例中的第二个示例显示与 `@Auditable` 方法的执行相匹配的建议：

```
@Before("com.xyz.lib.Pointcuts.anyPublicMethod() && @annotation(auditable)")  
public void audit(Auditable auditable) {  
    AuditCode code = auditable.value();  
    // ...  
}
```

## 建议参数和泛型

Spring AOP 可以处理类声明和方法参数中使用的泛型。假设您具有如下通用类型：

```
public interface Sample<T> {  
    void sampleGenericMethod(T param);  
    void sampleGenericCollectionMethod(Collection<T> param);  
}
```

您可以通过在要拦截方法的参数类型中键入 `advice` 参数，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  
}
```

这种方法不适用于通用集合。因此，您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  
}
```

为了使这项工作有效，我们将不得不检查集合中的每个元素，这是不合理的，因为我们也无法决定通常如何处理 `null` 值。要实现类似目的，您必须将参数键入 `Collection<?>` 并手动检查元素的类型。

## 确定参数名称

通知调用中的参数绑定依赖于切入点表达式中使用的名称与通知和切入点方法签名中声明的参数名称的匹配。通过 Java 反射无法获得参数名称，因此 Spring AOP 使用以下策略来确定参数名称：

- 如果用户已明确指定参数名称，则使用指定的参数名称。建议和切入点 `Comments` 都具有可选的 `argNames` 属性，您可以使用该属性来指定带 `Comments` 方法的参数名称。这些参数名称在运行时可用。下面的示例演示如何使用 `argNames` 属性：

```
@Before(value="com.xyz.lib.Pointcuts.anyPublicMethod() && target(bean) && @annotation(a  
    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，如果您修改前面的建议以接收连接点对象，则 `argNames` 属性不需要包括它：

```
@Before(value="com.xyz.lib.Pointcuts.anyPublicMethod() && target(bean) && @annotation(a  
    argNames="bean,auditable")  
public void audit(JoinPoint jp, Object bean, Auditable auditable) {  
    AuditCode code = auditable.value();  
    // ... use code, bean, and jp  
}
```

对于不收集任何其他连接点上下文的建议实例，对 `JoinPoint`，`ProceedingJoinPoint` 和

`JoinPoint.StaticPart` 类型的第一个参数进行特殊处理特别方便。在这种情况下，您可以省略 `argNames` 属性。例如，以下建议无需声明 `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，如果未指定 `'argNames'` 属性，Spring AOP 将查看该类的调试信息，并尝试从局部变量表中确定参数名称。只要已使用调试信息(至少 `'_-g:vars'`)编译了类，就存于此信息。启用此标志时进行编译的结果是：(1)您的代码稍微易于理解(逆向工程)，(2)类文件的大小略大(通常无关紧要)，(3)删除未使用的本地代码的优化变量不适用于您的编译器。换句话说，通过启用该标志，您应该不会遇到任何困难。

### iNote

如果即使没有调试信息，AspectJ 编译器(ajc)都已编译@AspectJ 方面，则无需添加 `argNames` 属性，因为编译器会保留所需的信息。

- 如果在没有必要调试信息的情况下编译了代码，Spring AOP 会尝试推断绑定变量与参数的配对(例如，如果切入点表达式中仅绑定了一个变量，并且 advice 方法仅接受一个参数，则配对很明显)。如果在给定可用信息的情况下变量的绑定不明确，则会抛出 `AmbiguousBindingException`。
- 如果以上所有策略均失败，则抛出 `IllegalArgumentException`。

## 处理参数

前面我们提到过，我们将描述如何使用在 Spring AOP 和 AspectJ 上始终有效的参数编写 `proceed` 调用。解决方案是确保建议签名按 Sequences 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，无论如何都要进行此绑定(如上例所示)。

## Advice Ordering

当多条建议都希望在同一连接点上运行时会发生什么？Spring AOP 遵循与 AspectJ 相同的优先级规则来确定建议执行的 Sequences。优先级最高的建议首先“在途中”运行(因此，给定两条优先建议，则优先级最高的建议首先运行)。从连接点“出路”中，优先级最高的建议将最后运行(因此，给定两条后置通知，优先级最高的建议将第二次运行)。

当在不同方面定义的两条建议都需要在同一连接点上运行时，除非另行指定，否则执行 Sequences 是不确定的。您可以通过指定优先级来控制执行 Sequences。通过在 Aspect 类中实现

`org.springframework.core.Ordered` 接口或使用 `Order` 注解对其进行 Comments，可以通过普通的 Spring 方法来完成。给定两个方面，从 `Ordered.getValue()` 返回较低值(或 Comments 值)的方面具有较高的优先级。

当在相同方面定义的两条建议都需要在同一连接点上运行时，其 Sequences 是未定义的(因为无法通过反射来获取 javac 编译类的声明 Sequences)。考虑将这些建议方法折叠成每个方面类中每个连接点的一个建议方法，或将建议重构为单独的方面类，您可以在方面级别进行 Order。

## 5.4.5. Introductions

简介(在 AspectJ 中称为类型间声明)使方面可以声明建议对象实现给定的接口，并代表那些对象提供该接口的实现。

您可以使用 `@DeclareParents` Comments 进行介绍。此注解用于声明匹配类型具有新的父代(因此而得名)。例如，在给定名为 `UsageTracked` 的接口和该接口名为 `DefaultUsageTracked` 的实现的情况下，以下方面声明服务接口的所有实现者也都实现 `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();  
    }  
}
```

要实现的接口由带 `Comments` 的字段的类型确定。 `@DeclareParents` 注解的 `value` 属性是 `AspectJ` 类型的模式。任何匹配类型的 `bean` 都实现 `UsageTracked` 接口。请注意，在前面示例的建议中，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`，则应编写以下内容：

```
UsageTracked usageTracked = (UsageTracked) context.getBean("myService");
```

## 5.4.6. 方面实例化模型

### iNote

这是一个高级主题。如果您刚开始使用 AOP，则可以放心地跳过它，直到以后。

默认情况下，应用程序上下文中每个方面都有一个实例。AspectJ 将此称为单例实例化模型。可以使用备用生命周期来定义方面。Spring 支持 AspectJ 的 `perthis` 和 `pertarget` 实例化模型(当前不支持 `percflow`, `percflowbelow`, 和 `pertypewithin`)。

您可以通过在 `@Aspect` 注解中指定 `perthis` 子句来声明 `perthis` 方面。考虑以下示例：

```
@Aspect("perthis(com.xyz.myapp.SystemArchitecture.businessService())")  
public class MyAspect {  
  
    private int someState;
```

```
@Before(com.xyz.myapp.SystemArchitecture.businessService())
public void recordServiceUsage() {
    // ...
}
```

在前面的示例中，`'perthis'` 子句的作用是为每个执行业务服务的唯一服务对象(每个与切入点表达式匹配的联接点绑定到“`this`”的唯一对象)创建一个方面实例。方面实例是在服务对象上首次调用方法时创建的。当服务对象超出范围时，方面将超出范围。在创建方面实例之前，其中的任何建议都不会执行。创建方面实例后，在其中声明的建议将在匹配的连接点处执行，但仅当服务对象是与此方面相关联的对象时才执行。有关 `per` 子句的更多信息，请参见 [AspectJ 编程指南](#)。

`pertarget` 实例化模型的工作方式与 `perthis` 完全相同，但是它为匹配的连接点处的每个唯一目标对象创建一个方面实例。

#### 5.4.7. AOP 示例

既然您已经了解了所有组成部分是如何工作的，那么我们可以将它们放在一起做一些有用的事情。

有时由于并发问题(例如，死锁失败者)，业务服务的执行可能会失败。如果重试该操作，则很可能在下一次尝试中成功。对于适合在这种情况下重试的业务服务(不需要为解决冲突而需要返回给用户的幂等操作)，我们希望透明地重试该操作以避免 Client 端看到

`PessimisticLockingFailureException`。这项要求清楚地跨越了服务层中的多个服务，因此非常适合通过一个方面实施。

因为我们想重试该操作，所以我们需要使用“周围”建议，以便我们可以多次调用 `proceed`。以下清单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);
        throw lockFailureException;
    }
}

```

请注意，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建议(每次重试时都希望有新的事务)。 `maxRetries` 和 `order` 属性均由 Spring 配置。主要动作发生在 `doConcurrentOperation` 周围建议中。请注意，目前，我们将重试逻辑应用于每个 `businessService()`。我们尝试 `continue`，如果失败并失败了 `PessimisticLockingFailureException`，我们将重试，除非我们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.ConcurrentOper
    <property name="maxRetries" value="3"/>
    <property name="order" value="100"/>
</bean>

```

为了优化方面，使其仅重试幂等操作，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 `Comments` 来 `Comments` 服务操作的实现。方面的更改仅重试幂等操作涉及精简切入点表达式，以便只有 `@Idempotent` 个操作匹配，如下所示：

```
@Around("com.xyz.myapp.SystemArchitecture.businessService() && " +
        "@annotation(com.xyz.myapp.service.Idempotent)")
public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {
    ...
}
```

## 5.5. 基于架构的 AOP 支持

如果您更喜欢基于 XML 的格式，Spring 还支持使用新的 `aop` 名称空间标签定义方面。支持与使用 `@AspectJ` 样式时完全相同的切入点表达式和建议类型。因此，在本节中，我们将重点放在新语法上，并使 Reader 参考上一节([@AspectJ support](#))中的讨论，以了解编写切入点表达式和建议参数的绑定。

要使用本节中描述的 `aop` 名称空间标签，您需要导入 `spring-aop` 模式，如[基于 XML 模式的配置](#)中所述。有关如何在 `aop` 名称空间中导入标签的信息，请参见[AOP 模式](#)。

在您的 Spring 配置中，所有方面和顾问元素都必须放在 `<aop:config>` 元素内(在应用程序上下文配置中可以有多个 `<aop:config>` 元素)。`<aop:config>` 元素可以包含切入点，顾问和方面元素(请注意，这些元素必须按此 `Sequences` 声明)。

### ⚠Warning

`<aop:config>` 样式的配置大量使用了 Spring 的[auto-proxying](#)机制。如果您已经通过使用 `BeanNameAutoProxyCreator` 或类似方法来使用显式自动代理，则可能会导致问题(例如未编制建议)。推荐的用法模式是仅使用 `<aop:config>` 样式或仅 `AutoProxyCreator` 样式

， 并且不要混合使用。

### 5.5.1. 声明一个方面

使用模式支持时，方面是在 Spring 应用程序上下文中定义为 Bean 的常规 Java 对象。状态和行为在对象的字段和方法中捕获，切入点和建议信息在 XML 中捕获。

您可以使用`\<>`元素声明一个方面，并使用`ref` 属性引用该支持 bean，如以下示例所示：

```
<aop:config>
  <aop:aspect id="myAspect" ref="aBean">
    ...
  </aop:aspect>
</aop:config>

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

支持方面(在这种情况下为 `aBean`)的 bean 当然可以像配置任何其他 Spring bean 一样进行配置并注入依赖项。

### 5.5.2. 声明切入点

您可以在 `<aop:config>` 元素中声明命名的切入点，从而使切入点定义在多个方面和顾问程序之间共享。

可以定义代表服务层中任何业务服务的执行的切入点：

```
<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 support](#) 中所述的 AspectJ 切入点表达式语言。如果使用基于架构的声明样式，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：

```
<aop:config>

    <aop:pointcut id="businessService"
        expression="com.xyz.myapp.SystemArchitecture.businessService()" />

</aop:config>
```

假定您具有[共享通用切入点定义](#)中所述的 `SystemArchitecture` 外观。

然后，在方面中声明切入点与声明顶级切入点非常相似，如以下示例所示：

```
<aop:config>

    <aop:aspect id="myAspect" ref="aBean">

        <aop:pointcut id="businessService"
            expression="execution(* com.xyz.myapp.service.*.*(..))" />

        ...

    </aop:aspect>

</aop:config>
```

与`@Aspect`方面几乎相同，使用基于架构的定义样式声明的切入点可以收集连接点上下文。例如，以下切入点收集 `this` 对象作为连接点上下文，并将其传递给建议：

```
<aop:config>

    <aop:aspect id="myAspect" ref="aBean">

        <aop:pointcut id="businessService"
            expression="execution(* com.xyz.myapp.service.*.*(..)) && this(service)" />

        <aop:before pointcut-ref="businessService" method="monitor" />

        ...

    </aop:aspect>

</aop:config>
```

必须声明通知，以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) {
    ...
}
```

组合切入点子表达式时，XML 文档中的 `&&` 很尴尬，因此可以分别使用 `and`，`or` 和 `not` 关键字代替 `&&`，`||` 和 `!`。例如，上一个切入点可以更好地编写如下：

```
<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"/>

        ...

    </aop:aspect>
</aop:config>
```

请注意，以这种方式定义的切入点由其 XML `id` 引用，并且不能用作命名切入点以形成复合切入点。因此，基于架构的定义样式中的命名切入点支持比@AspectJ 样式所提供的更受限制。

### 5.5.3. 宣告建议

基于模式的 AOP 支持使用与@AspectJ 样式相同的五种建议，并且它们具有完全相同的语义。

#### Before Advice

在运行匹配的方法之前，建议运行之前。使用`\<>`元素在 `<aop:aspect>` 内部声明它，如以下示例所示：

```
<aop:aspect id="beforeExample" ref="aBean">

    <aop:before
        pointcut-ref="dataAccessOperation"
        method="doAccessCheck"/>

    ...

</aop:aspect>
```

在这里，`dataAccessOperation` 是在最高(`<aop:config>`)级别定义的切入点的 `id`。要定义切入点内联，请用 `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@AspectJ 样式的讨论中所指出的那样，使用命名的切入点可以显着提高代码的可读性。

`method` 属性标识提供建议正文的方法(`doAccessCheck`)。必须为包含建议的 Aspect 元素所引用的 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 Aspect Bean 上的 `doAccessCheck` 方法。

## 返回建议后

返回的建议在匹配的方法执行正常完成时运行。在 `<aop:aspect>` 内部以与建议之前相同的方式声明它。以下示例显示了如何声明它：

```
<aop:aspect id="afterReturningExample" ref="aBean">  
    <aop:after-returning  
        pointcut-ref="dataAccessOperation"  
        method="doAccessCheck"/>  
  
    ...  
</aop:aspect>
```

与@AspectJ 样式一样，您可以在建议正文中获取返回值。为此，使用 `returning` 属性指定返回值应传递到的参数的名称，如以下示例所示：

```
<aop:aspect id="afterReturningExample" ref="aBean">  
    <aop:after-returning  
        pointcut-ref="dataAccessOperation"  
        returning="RetVal"  
        method="doAccessCheck"/>  
  
    ...  
</aop:aspect>
```

`doAccessCheck` 方法必须声明一个名为 `retval` 的参数。该参数的类型以与 `@AfterReturning` 相同的方式约束匹配。例如，您可以声明方法签名，如下所示：

```
public void doAccessCheck(Object retVal) { ... }
```

## 提出建议后

抛出建议后，当匹配的方法执行通过抛出异常退出时执行建议。通过使用掷后元素在 `<aop:aspect>` 内部声明它，如以下示例所示：

```
<aop:aspect id="afterThrowingExample" ref="aBean">

    <aop:after-throwing
        pointcut-ref="dataAccessOperation"
        method="doRecoveryActions" />

    ...
</aop:aspect>
```

与`@AspectJ` 样式一样，您可以在通知正文中获取引发的异常。为此，请使用 `throwing` 属性指定异常应传递到的参数的名称，如以下示例所示：

```
<aop:aspect id="afterThrowingExample" ref="aBean">

    <aop:after-throwing
        pointcut-ref="dataAccessOperation"
        throwing="dataAccessEx"
        method="doRecoveryActions" />

    ...
</aop:aspect>
```

`doRecoveryActions` 方法必须声明一个名为 `dataAccessEx` 的参数。该参数的类型以与 `@AfterThrowing` 相同的方式约束匹配。例如，方法签名可以声明如下：

```
public void doRecoveryActions(DataAccessException dataAccessEx) { ... }
```

## (最后)建议后

无论最终如何执行匹配的方法，建议(最终)都会运行。您可以使用 `after` 元素对其进行声明，如以

下示例所示：

```
<aop:aspect id="afterFinallyExample" ref="aBean">

    <aop:after
        pointcut-ref="dataAccessOperation"
        method="doReleaseLock"/>

    ...

</aop:aspect>
```

## Around Advice

最后一种建议是围绕建议。围绕建议在匹配的方法执行过程中“围绕”运行。它有机会在方法执行之前和之后进行工作，并确定何时，如何以及什至根本不执行该方法。周围建议通常用于以线程安全的方式(例如，启动和停止计时器)在方法执行之前和之后共享状态。始终使用最不强大的建议形式，以满足您的要求。如果建议可以完成这项工作，请不要在建议周围使用。

您可以使用 `aop:around` 元素在建议周围进行声明。咨询方法的第一个参数必须为 `ProceedingJoinPoint` 类型。在建议的正文中，在 `ProceedingJoinPoint` 上调用 `proceed()` 会导致基础方法执行。`proceed` 方法也可以用 `Object[]` 调用。数组中的值用作方法执行时的参数。有关使用 `Object[]` 调用 `proceed` 的说明，请参见 [Around Advice](#)。以下示例显示了如何在 XML 中围绕建议进行声明：

```
<aop:aspect id="aroundExample" ref="aBean">

    <aop:around
        pointcut-ref="businessService"
        method="doBasicProfiling"/>

    ...

</aop:aspect>
```

`doBasicProfiling` 通知的实现可以与 `@Aspect` 示例完全相同(当然要减去 Comments)，如以下示例所示：

```
public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {
    // start stopwatch
    Object retVal = pjp.proceed();
```

```
// stop stopwatch  
return retVal;  
}
```

## Advice Parameters

基于架构的声明样式以与 @AspectJ 支持相同的方式支持完全类型的建议，即通过名称与建议方法参数匹配切入点参数。有关详情，请参见 [Advice Parameters](#)。如果您希望显式指定建议方法的参数名称(不依赖于先前描述的检测策略)，则可以通过使用建议元素的 `arg-names` 属性来实现，该属性与 `argNames` 属性的处理方式相同。建议 Comments(如 [确定参数名称](#) 中所述)。以下示例显示如何在 XML 中指定参数名称：

```
<aop:before  
    pointcut="com.xyz.lib.Pointcuts.anyPublicMethod() and @annotation(auditable)"  
    method="audit"  
    arg-names="auditable"/>
```

`arg-names` 属性接受逗号分隔的参数名称列表。

以下基于 XSD 的方法中涉及程度稍高的示例显示了一些与一些强类型参数结合使用的建议：

```
package x.y.service;  
  
public interface PersonService {  
  
    Person getPerson(String personName, int age);  
}  
  
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的参数，第一个恰好是用于进行方法调用的连接点。此参数的存在表明 `profile(..)` 用作 `around` 建议，如以下示例所示：

```
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 Throwa
        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，以下示例 XML 配置影响了特定连接点的上述建议的执行：

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans http://www.springframework.org/sche
           http://www.springframework.org/schema/aop http://www.springframework.org/schem

<!-- 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>

```

考虑以下驱动程序脚本：

```

import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.ClassPathXmlApplicationContext;
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 Boot 类，我们将在标准输出上获得类似于以下内容的输出：

```
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)上执行多个建议时，排序规则如[Advice Ordering](#)中所述。方面之间的优先级是通过将 `Order` 注解添加到支持方面的 Bean 或通过使 Bean 实现 `Ordered` 接口来确定的。

### 5.5.4. Introductions

简介(在 AspectJ 中称为类型间声明)使方面可以声明建议的对象实现给定的接口，并代表那些对象提供该接口的实现。

您可以使用 `aop:aspect` 内的 `aop:declare-parents` 元素进行介绍。您可以使用 `aop:declare-parents` 元素来声明匹配类型具有新的父代(因此而得名)。例如，给定名为 `UsageTracked` 的接口和该名为 `DefaultUsageTracked` 的接口的实现，以下方面声明服务接口的所有实现者也都实现 `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 类型的模式。任何匹配类型的 bean 都实现 `UsageTracked` 接口。请注意，在前面示例的建议中，服务 Bean 可以直接用作 `UsageTracked` 接口的实现。要以编程方式访问 bean，可以编写以下代码：

```
UsageTracked usageTracked = (UsageTracked) context.getBean("myService");
```

## 5.5.5. 方面实例化模型

模式定义方面唯一受支持的实例化模型是单例模型。在将来的版本中可能会支持其他实例化模型。

## 5.5.6. Advisors

“顾问”的概念来自 Spring 中定义的 AOP 支持，并且在 AspectJ 中没有直接等效的概念。顾问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 [Spring 的建议类型](#) 中描述的建议接口之一。顾问可以利用 AspectJ 切入点表达式。

Spring 通过 `<aop:advisor>` 元素支持顾问程序概念。您通常会看到它与事务建议结合使用，事务建议在 Spring 中也有自己的名称空间支持。以下示例显示顾问程序：

```
<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>
```

除了前面的示例中使用的 `pointcut-ref` 属性，还可以使用 `pointcut` 属性来内联定义切入点表达式。

要定义顾问程序的优先级，以便该建议书可以参与 Order，请使用 `order` 属性来定义顾问程序的 `Ordered` 值。

### 5.5.7. AOP 模式示例

本节显示了使用模式支持重写时来自[AOP 示例](#)的并发锁定失败重试示例的外观。

有时由于并发问题(例如，死锁失败者)，业务服务的执行可能会失败。如果重试该操作，则很可能在下一次尝试中成功。对于适合在这种情况下重试的业务服务(不需要为解决冲突而需要返回给用户的幂等操作)，我们希望透明地重试该操作以避免 Client 端看到

`PessimisticLockingFailureException`。这项要求清楚地跨越了服务层中的多个服务，因此非常适合通过一个方面实施。

因为我们想重试该操作，所以我们需要使用“周围”建议，以便我们可以多次调用 `proceed`。以下清单显示了基本方面的实现(这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() {
        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;
    }

}

```

请注意，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建议(每次重试时都希望有新的事务)。`maxRetries` 和 `order` 属性均由 Spring 配置。主要动作发生在 `doConcurrentOperation` 周围建议方法中。我们尝试 `continue`。如果我们失败了 `PessimisticLockingFailureException`，我们将重试，除非我们用尽了所有的重试尝试。

### Note

该类与@AspectJ 示例中使用的类相同，但是除去了 Comments。

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，目前我们假设所有业务服务都是幂等的。如果不是这种情况，我们可以通过引入 `@Idempotent` Comments 并使用该 Comments 来 Comments 服务操作的实现，来改进方面，使其仅重试 true 的幂等操作，如以下示例所示：

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Idempotent {
    // marker annotation
}
```

方面的更改仅重试幂等操作涉及精简切入点表达式，以便只有 `@Idempotent` 个操作匹配，如下所示：

```
<aop:pointcut id="idempotentOperation"
    expression="execution(* com.xyz.myapp.service.*.*(..)) and
    @annotation(com.xyz.myapp.service.Idempotent)"/>
```

## 5.6. 选择要使用的 AOP 声明样式

一旦确定方面是实现给定需求的最佳方法，您如何在使用 Spring AOP 或 AspectJ 以及在 Aspect 语言(代码)样式，@ AspectJ 注解样式或 Spring XML 样式之间做出选择？这些决定受许多因素影响，包括应用程序需求，开发工具以及团队对 AOP 的熟悉程度。

### 5.6.1. Spring AOP 还是 Full AspectJ ?

使用最简单的方法即可。Spring AOP 比使用完整的 AspectJ 更简单，因为不需要在开发和构建过程中引入 AspectJ 编译器/编织器。如果您只需要建议在 Spring bean 上执行操作，则 Spring AOP 是正确的选择。如果您需要建议不受 Spring 容器 Management 的对象(通常是域对象)，则需要使用 AspectJ。如果您希望建议除简单方法执行以外的连接点(例如，字段 get 或设置连接点等)，则还需要使用 AspectJ。

使用 AspectJ 时，可以选择 AspectJ 语言语法(也称为“代码样式”)或@AspectJComments 样式。显然，如果您不使用 Java 5，那么将为您做出选择：使用代码样式。如果方面在您的设计中起着重

要作用，并且您能够将[AspectJ 开发工具\(AJDT\)](#)插件用于 Eclipse，则 AspectJ 语言语法是首选。它更干净，更简单，因为该语言是专为编写方面而设计的。如果您不使用 Eclipse 或只有少数几个方面在您的应用程序中不起作用，那么您可能要考虑使用@AspectJ 样式，在 IDE 中坚持常规 Java 编译，并向其中添加方面编织阶段您的构建脚本。

## 5.6.2. @AspectJ 或 Spring AOP 的 XML？

如果选择使用 Spring AOP，则可以选择@AspectJ 或 XML 样式。有各种折衷考虑。

XML 样式可能是现有 Spring 用户最熟悉的，并且得到了 true 的 POJO 的支持。当使用 AOP 作为配置企业服务的工具时，XML 可能是一个不错的选择(一个很好的测试是您是否将切入点表达式视为配置的一部分，您可能希望独立更改)。使用 XML 样式，可以说从您的配置中可以更清楚地了解系统中存在哪些方面。

XML 样式有两个缺点。首先，它没有完全将要解决的需求的实现封装在一个地方。DRY 原则说，系统中的任何知识都应该有单一，明确，Authority 的表示形式。当使用 XML 样式时，关于如何实现需求的知识会在配置文件中的后备 bean 类的声明和 XML 中分散。当您使用@AspectJ 样式时，此信息将封装在一个单独的模块中：方面。其次，与@AspectJ 样式相比，XML 样式在表达能力上有更多限制：仅支持“单例”方面实例化模型，并且无法组合以 XML 声明的命名切入点。例如，使用@AspectJ 样式，您可以编写如下内容：

```
@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，您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 方法的缺点是无法通过组合这些定义来定义 `accountPropertyAccess` 切入点。

`@AspectJ` 样式支持其他实例化模型和更丰富的切入点组合。它具有将方面保持为模块化单元的优势。它还具有的优点是，Spring AOP 和 AspectJ 都可以理解`@AspectJ` 方面。因此，如果您以后决定需要 AspectJ 的功能来实现其他要求，则可以轻松地迁移到基于 AspectJ 的方法。总而言之，只要您拥有比简单地配置企业服务更多的功能，Spring 团队就会喜欢`@AspectJ` 样式。

## 5.7. 混合方面类型

通过使用自动代理支持，模式定义的 `<aop:aspect>` 方面，`<aop:advisor>` 声明的顾问程序，甚至是在同一配置中使用 Spring 1.2 样式定义的代理和拦截器，完全可以混合使用`@AspectJ` 样式方面的所有这些都是通过使用相同的基础支持机制实现的，并且可以毫无困难地共存。

## 5.8. 代理机制

Spring AOP 使用 JDK 动态代理或 CGLIB 创建给定目标对象的代理。（只要有选择，首选 JDK 动态代理）。

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。代理了由目标类型实现的所有接口。如果目标对象未实现任何接口，则将创建 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，而不仅是由其接口实现的方法），都可以这样做。但是，您应该考虑以下问题：

- 不能建议 `final` 方法，因为它们不能被覆盖。
- 从 Spring 3.2 开始，不再需要将 CGLIB 添加到您的项目 Classpath 中，因为 CGLIB 类在 `org.springframework` 下重新打包并直接包含在 spring-core JAR 中。这意味着基于 CGLIB 的代理支持“有效”，就像 JDK 动态代理始终具有一样。
- 从 Spring 4.0 开始，由于 CGLIB 代理实例是通过 Objenesis 创建的，因此不再调用代理对象的构造函数两次。仅当您的 JVM 不允许绕过构造函数时，您才可能从 Spring 的 AOP 支持中看到两次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` 自动代理支持时强制 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" />
```

### ①Note

多个 `<aop:config>` 节在运行时折叠到一个统一的自动代理创建器中, 该创建器将应用 `<aop:config>` 节中的任何(通常来自不同 XML bean 定义文件)指定的\* strong \*代理设置。这也适用于 `<tx:annotation-driven/>` 和 `<aop:aspectj-autoproxy/>` 元素。明确地说, 在 `<tx:annotation-driven/>`, `<aop:aspectj-autoproxy/>` 或 `<aop:config/>` 元素上使用 `proxy-target-class="true"` 会强制对所有三个元素\*使用 CGLIB 代理。

## 5.8.1. 了解 AOP 代理

Spring AOP 是基于代理的。在编写自己的方面或使用 Spring Framework 随附的任何基于 Spring AOP 的方面之前, 掌握最后一条语句实际含义的语义至关重要。

首先考虑您有一个普通的, 未经代理的, 没有什么特别的, 直接的对象引用的情况, 如以下代码片段所示:

```
public class SimplePojo implements Pojo {
    public void foo() {
        // this next method invocation is a direct call on the 'this' reference
```

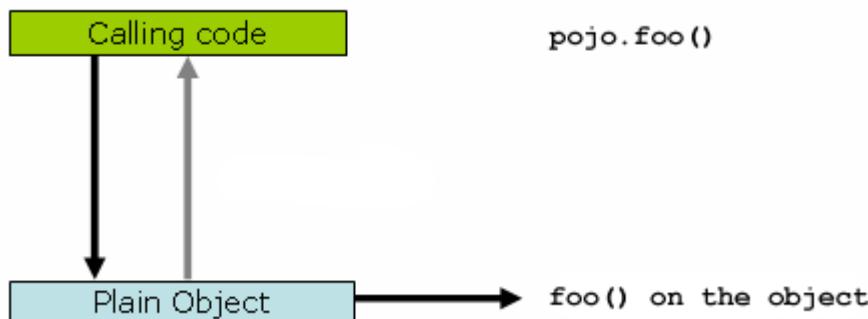
```

        this.bar();
    }

    public void bar() {
        // some logic...
    }
}

```

如果在对象引用上调用方法，则直接在该对象引用上调用该方法，如下图和清单所示：



```

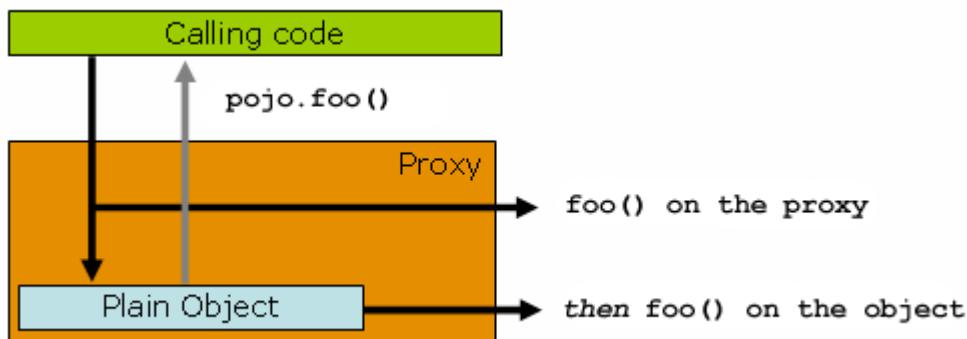
public class Main {

    public static void main(String[] args) {
        Pojo pojo = new SimplePojo();

        // this is a direct method call on the 'pojo' reference
        pojo.foo();
    }
}

```

当 Client 端代码具有的引用是代理时，情况会稍有变化。考虑以下图表和代码片段：



```

public class Main {

    public static void main(String[] args) {
        ProxyFactory factory = new ProxyFactory(new SimplePojo());
        factory.addInterface(Pojo.class);
    }
}

```

```

        factory.addAdvice(new RetryAdvice());
        Pojo pojo = (Pojo) factory.getProxy();
        // this is a method call on the proxy!
        pojo.foo();
    }
}

```

这里要理解的关键是，`Main` 类的 `main(..)` 方法内部的 Client 端代码具有对代理的引用。这意味着该对象引用上的方法调用是代理上的调用。结果，代理可以委派给与该特定方法调用相关的所有拦截器(建议)。但是，一旦调用最终到达目标对象(在这种情况下为 `SimplePojo`)，则为

`this.bar()` 或 `this.foo()`，则将针对 `this` 引用而不是对 `this` 引用调用它可能对其自身进行的任何方法调用。代理。这具有重要的意义。这意味着自调用不会导致与方法调用相关的建议得到执行的机会。

好吧，那么该怎么办？最好的方法(此处宽松地使用术语“最好”)是重构代码，以免发生自调用。这确实需要您做一些工作，但这是最好的，侵入性最小的方法。下一种方法绝对可怕，我们正要指出这一点，恰恰是因为它是如此可怕。您可以(对我们来说是痛苦的)完全将类中的逻辑与 Spring AOP 绑定在一起，如以下示例所示：

```

public class SimplePojo implements Pojo {

    public void foo() {
        // this works, but... gah!
        ((Pojo) AopContext.currentProxy()).bar();
    }

    public void bar() {
        // some logic...
    }
}

```

这将您的代码完全耦合到 Spring AOP，并且使类本身意识到在 AOP 上下文中使用它这一事实，而 AOP 却是事实。创建代理时，还需要一些其他配置，如以下示例所示：

```

public class Main {

    public static void main(String[] args) {
        ProxyFactory factory = new ProxyFactory(new SimplePojo());
        factory.addInterface(Pojo.class);
    }
}

```

```
factory.addAdvice(new RetryAdvice());
factory.setExposeProxy(true);

Pojo pojo = (Pojo) factory.getProxy();

// this is a method call on the proxy!
pojo.foo();
}

}
```

最后，必须注意，AspectJ 没有此自调用问题，因为它不是基于代理的 AOP 框架。

## 5.9. 以编程方式创建@AspectJ 代理

除了使用 `<aop:config>` 或 `<aop:aspectj-autoproxy>` 声明配置中的各个方面外，还可以通过编程方式创建建议目标对象的代理。有关 Spring 的 AOP API 的完整详细信息，请参见[next chapter](#)。在这里，我们要重点介绍通过使用@AspectJ 方面自动创建代理的功能。

您可以使用 `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。此类的基本用法非常简单，如以下示例所示：

```
// 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
factory.addAspect(usageTracker);

// now get the proxy object...
MyInterfaceType proxy = factory.getProxy();
```

有关更多信息，请参见[javadoc](#)。

## 5.10. 在 Spring 应用程序中使用 AspectJ

到目前为止，本章介绍的所有内容都是纯 Spring AOP。在本节中，我们将研究如果您的需求超出了 Spring AOP 所提供的功能，那么如何使用 AspectJ 编译器或 weaver 代替 Spring AOP 或除 Spring AOP 之外使用。

Spring 附带了一个小的 AspectJ 方面库，该库在您的发行版中可以作为 `spring-aspects.jar` 独立使用。您需要将其添加到 Classpath 中才能使用其中的方面。使用 AspectJ 通过 Spring 依赖注入域对象和 AspectJ 的其他 Spring 方面讨论了该库的内容以及如何使用它。使用 Spring IoC 配置 AspectJ Aspects 讨论如何依赖注入使用 AspectJ 编译器编织的 AspectJ 方面。最后，在 Spring Framework 中使用 AspectJ 进行加载时编织介绍了使用 AspectJ 的 Spring 应用程序的加载时编织。

### 5.10.1. 使用 AspectJ 通过 Spring 依赖注入域对象

Spring 容器实例化并配置在您的应用程序上下文中定义的 bean。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 Comments 驱动的方面，该方面利用此功能允许任何对象的依赖项注入。该支架旨在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` 运算符或通过 ORM 工具以数据库查询的方式通过程序创建的。

`@Configurable` Comments 将一个类标记为符合 Spring 驱动的配置。在最简单的情况下，您可以将其纯粹用作标记 Comments，如以下示例所示：

```
package com.xyz.myapp.domain;

import org.springframework.beans.factory.annotation.Configurable;

@Configurable
public class Account {
    // ...
}
```

当以这种方式用作标记接口时，Spring 通过使用具有与完全限定类型名称(`com.xyz.myapp.domain.Account`)同名的 bean 定义(通常为原型作用域)来配置带 Comments 类型的新实例(在本例中为 `Account`)。由于 Bean 的默认名称是其类型的完全限定名称，因此声明原型定义的便捷方法是省略 `id` 属性，如以下示例所示：

```
<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，则可以直接在注解中这样做，如以下示例所示：

```
package com.xyz.myapp.domain;

import org.springframework.beans.factory.annotation.Configurable;

@Configurable("account")
public class Account {
    // ...
}
```

Spring 现在查找名为 account 的 bean 定义，并将其用作配置新 Account 实例的定义。

您也可以使用自动装配来避免完全指定专用的 bean 定义。要让 Spring 应用自动装配，请使用

@Configurable 注解的 autowire 属性。您可以分别按类型或名称指定

@Configurable(autowire=Autowire.BY\_TYPE) 或

@Configurable(autowire=Autowire.BY\_NAME) 自动布线。或者，从 Spring 2.5 开始，最好在字段或方法级别使用 @Autowired 或 @Inject 为 @Configurable bean 指定显式的，Comments 驱动的依赖项注入(有关更多详细信息，请参见[基于 Comments 的容器配置](#))。

最后，您可以使用 dependencyCheck 属性(例如

@Configurable(autowire=Autowire.BY\_NAME,dependencyCheck=true) )为新创建和配置的对象中的对象引用启用 Spring 依赖项检查。如果此属性设置为 true，则 Spring 在配置后验证是否已设置所有属性(不是基元或集合)。

请注意，单独使用 Comments 不会执行任何操作。Comments 中存在的是 spring-

aspects.jar 中的 AnnotationBeanConfigurerAspect。从本质上讲，方面说：“在从带有

@Configurable Comments 的类型的新的对象的初始化返回之后，根据 Comments 的属性使用 Spring 配置新创建的对象”。在这种情况下，“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)以及正在反序列化(例如，通过[readResolve\(\)](#))的 Serializable 对象。

## iNote

上段中的关键短语之一是“本质上”。对于大多数情况，“从新对象的初始化返回后”的确切语义是可以的。在这种情况下，“初始化之后”是指在构造对象之后注入依赖项。这意味着该依赖项不可在类的构造函数体中使用。如果要在构造函数主体执行之前注入依赖项，从而可以在构造函数主体中使用这些依赖项，则需要在 `@Configurable` 声明中定义此变量，如下所示：

```
@Configurable(preConstruction=true)
```

您可以在[AspectJ 编程指南](#)的 AspectJ 在本附录中找到有关各种切入点类型的语言语义的更多信息。

为此，必须将带 `Comments` 的类型与 AspectJ 编织器编织在一起。您可以使用构建时 Ant 或 Maven 任务来执行此操作(例如，参见[\\_\\_](#))，也可以使用加载时编织(请参见[在 Spring Framework 中使用 AspectJ 进行加载时编织](#))。`AnnotationBeanConfigurerAspect` 本身需要由 Spring 配置(以获取对将用于配置新对象的 Bean 工厂的引用)。如果使用基于 Java 的配置，则可以将

`@EnableSpringConfigured` 添加到任何 `@Configuration` 类中，如下所示：

```
@Configuration  
@EnableSpringConfigured  
public class AppConfig {  
}
```

如果您喜欢基于 XML 的配置，Spring [context namespace](#) 定义了一个方便的 `context:spring-configured` 元素，您可以按以下方式使用它：

```
<context:spring-configured/>
```

在配置方面之前创建的 `@Configurable` 个对象的实例导致向调试日志发出一条消息，并且未进行任何对象配置。一个示例可能是 Spring 配置中的 bean，当它由 Spring 初始化时会创建域对象。在这种情况下，可以使用 `depends-on` bean 属性来手动指定该 bean 取决于配置方面。下面的示

例演示如何使用 `depends-on` 属性：

```
<bean id="myService"
      class="com.xzy.myapp.service.MyService"
      depends-on="org.springframework.beans.factory.aspectj.AnnotationBeanConfigurerAspect"

      <!-- ... -->

</bean>
```

### iNote

除非您真的想在运行时依赖它的语义，否则不要通过 bean configurer 方面激活

`@Configurable` 处理。特别是，请确保不要在已通过容器注册为常规 Spring bean 的 bean 类上使用 `@Configurable`。这样做将导致两次初始化，一次是通过容器，一次是通过方面。

## 单元测试`@Configurable` 对象

`@Configurable` 支持的目标之一是实现域对象的独立单元测试，而不会遇到与硬编码查找相关的困难。如果 AspectJ 尚未编织 `@Configurable` 类型，则 Comments 在单元测试期间不起作用。您可以在被测对象中设置模拟或存根属性引用，然后照常进行。如果 `@Configurable` 类型是 AspectJ 编织的，您仍然可以像往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，指示该对象尚未由 Spring 配置。

## 处理多个应用程序上下文

用于实现 `@Configurable` 支持的 `AnnotationBeanConfigurerAspect` 是 AspectJ 单例方面。单例方面的范围与 `static` 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，如果您在同一个类加载器层次结构中定义多个应用程序上下文，则需要考虑在哪里定义 `@EnableSpringConfigured` bean 以及在哪里将 `spring-aspects.jar` 放在 Classpath 上。

考虑一个典型的 Spring Web 应用程序配置，该配置具有一个共享的父应用程序上下文，该上下文

定义了通用的业务服务，支持那些服务所需的一切，以及每个 `Servlet` 的一个子应用程序上下文(其中包含该 `Servlet` 的特定定义)。所有这些上下文共存于相同的类加载器层次结构中，因此

`AnnotationBeanConfigurerAspect` 只能保留对其中一个的引用。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。这定义了您可能想注入域对象的服务。结果是，您无法使用`@Configurable` 机制来配置域对象，该域对象引用的是在子(特定于 `Servlet` 的)上下文中定义的 bean 的引用(无论如何，这可能不是您想做的事情)。

在同一容器中部署多个 Web 应用程序时，请确保每个 Web 应用程序使用自己的类加载器(例如，将 `spring-aspects.jar` 放在 '`WEB-INF/lib`' 中)加载 `spring-aspects.jar` 中的类型。如果 `spring-aspects.jar` 仅添加到容器级的 Classpath 中(并因此由共享的父类加载器加载)，则所有 Web 应用程序都共享相同方面实例(这可能不是您想要的)。

## 5.10.2. AspectJ 的其他 Spring 方面

除了 `@Configurable` 方面之外，`spring-aspects.jar` 还包含一个 AspectJ 方面，您可以使用它来驱动 Spring 的事务 Management，该事务 Management 使用 `@Transactional` Comments 进行 Comments 的类型和方法。这主要适用于希望在 Spring 容器之外使用 Spring Framework 的事务支持的用户。

解释 `@Transactional` Comments 的方面是 `AnnotationTransactionAspect`。使用此方面时，必须 Comments 实现类(或该类中的方法或两者)，而不是 Comments 该类所实现的接口(如果有)。AspectJ 遵循 Java 的规则，即不继承接口上的 Comments。

类上的 `@Transactional` Comments 指定用于执行该类中任何公共操作的默认事务语义。

类内方法上的 `@Transactional` Comments 将覆盖类 Comments(如果存在)给出的默认事务语义。可以标注任何可见性的方法，包括私有方法。直接 Comments 非公共方法是执行此类方法而获得事务划分的唯一方法。

Tip

从 Spring Framework 4.2 开始，`spring-aspects` 提供了类似的方面，为标准 `javax.transaction.Transactional` Comments 提供了完全相同的功能。检查 `JtaAnnotationTransactionAspect` 了解更多详细信息。

对于希望使用 Spring 配置和事务 Management 支持但又不想(或不能)使用 Comments 的 AspectJ 程序员，`spring-aspects.jar` 还包含 `abstract` 个方面，您可以扩展它们以提供自己的切入点定义。有关更多信息，请参见 `AbstractBeanConfigurerAspect` 和 `AbstractTransactionAspect` 方面的资源。例如，以下摘录显示了如何编写方面来使用与完全限定的类名匹配的原型 Bean 定义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);  
}
```

### 5.10.3. 使用 Spring IoC 配置 AspectJ Aspects

当您将 AspectJ 方面与 Spring 应用程序一起使用时，既自然又希望能够使用 Spring 配置这些方面。AspectJ 运行时本身负责方面的创建，并且通过 Spring 配置 AspectJ 创建的方面的方法取决于方面所使用的 AspectJ 实例化模型(`per-xxx` 子句)。

AspectJ 的大多数方面都是单例方面。这些方面的配置很容易。您可以创建一个正常引用外观类型并包含 `factory-method="aspectOf"` bean 属性的 bean 定义。这可以确保 Spring 通过向 AspectJ 索要长宽比实例，而不是尝试自己创建实例来获得长宽比实例。下面的示例演示如何使用 `factory-method="aspectOf"` 属性：

```
<bean id="profiler" class="com.xyz.profiler.Profiler"
      factory-method="aspectOf" (1)

      <property name="profilingStrategy" ref="jamonProfilingStrategy" />
</bean>
```

- (1) 请注意 `factory-method="aspectOf"` 属性

非单一方面很难配置。但是，可以通过创建原型 bean 定义并使用 `spring-aspects.jar` 的

`@Configurable` 支持来配置方面实例(一旦 AspectJ 运行时创建了 bean)来实现。

如果您有一些要与 AspectJ 编织的@AspectJ 方面(例如，对域模型类型使用加载时编织)以及要与 Spring AOP 一起使用的其他@AspectJ 方面，那么这些方面都已在 Spring 中配置，您需要告诉 Spring AOP @AspectJ 自动代理支持，应使用配置中定义的@AspectJ 方面的确切子集进行自动代理。您可以使用 `<aop:aspectj-autoproxy/>` 声明中的一个或多个 `<include/>` 元素来完成此操作。每个 `<include/>` 元素都指定一个名称模式，并且只有名称与至少一个模式匹配的 bean 才可用于 Spring AOP 自动代理配置。以下示例显示了如何使用 `<include/>` 元素：

```
<aop:aspectj-autoproxy>
  <aop:include name="thisBean"/>
  <aop:include name="thatBean"/>
</aop:aspectj-autoproxy>
```

#### iNote

不要被 `<aop:aspectj-autoproxy/>` 元素的名称所迷惑。使用它可以创建 Spring AOP 代理。此处使用了@AspectJ 样式的声明，但是不涉及 AspectJ 运行时。

### 5.10.4. 在 Spring Framework 中使用 AspectJ 进行加载时编织

加载时编织(LTW)是指在将 AspectJ 方面加载到应用程序的类文件中时将它们编织到 Java 虚拟机(JVM)中的过程。本部分的重点是在 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 的详细信息以及仅使用 AspectJ 配置 LTW(完全不涉及 Spring)的详细信息，请参见[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精细的控制。“香草” AspectJ LTW 通过使用 Java(5)代理来实现，该代理通过在启动 JVM 时指定 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 允许您以 `ClassLoader` 为基础打开 LTW，它的粒度更细，并且在“单 JVM-多应用程序”环境中(例如在典型的应用程序服务器中发现)更有意义。环境)。

此外，在某些环境中，此支持可实现加载时编织，而无需对添加 `-`

`javaagent:path/to/aspectjweaver.jar` 或 `-`

`javaagent:path/to/org.springframework.instrument-{version}.jar` (以前称为 `spring-agent.jar`) 所需的应用服务器的启动脚本进行任何修改。开发人员可以修改构成应用程序上下文的一个或多个文件，以实现加载时编织，而不必依赖通常负责部署配置(例如启动脚本)的 Management 员。

现在，销售工作已经结束，让我们首先浏览一个使用 Spring 的 AspectJ LTW 的快速示例，然后详细介绍示例中引入的元素。有关完整示例，请参见[Petclinic sample 申请](#)。

## 第一个例子

假设您是一位负责诊断系统中某些性能问题的原因的应用程序开发人员。与其使用分析工具，不如使用一个简单的分析方面，使我们能够快速获得一些性能 Metrics。然后，我们可以立即在该特定区域应用更细粒度的分析工具。

### iNote

此处提供的示例使用 XML 配置。您还可以将[Java configuration](#) 配置和使用@AspectJ。具体来说，您可以使用 `@EnableLoadTimeWeaving` Comments 替代 `<context:load-time-weaver/>` (有关详细信息，请参见[below](#))。

下面的示例显示了配置方面，它不是花哨的-它是基于时间的探查器，它使用@AspectJ 样式的方面声明：

```

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，以通知 AspectJ 编织者我们要将

`ProfilingAspect` 编织到类中。此文件约定，即在 JavaClasspath 上名为 `META-INF/aop.xml` 的文件，是标准 AspectJ。下面的示例显示 `aop.xml` 文件：

```

<!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN" "http://www.eclipse.org/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 `continue` 进行配置中特定于 Spring 的部分。我们需要配置一个

`LoadTimeWeaver` (稍后说明)。此加载时织布器是必不可少的组件，负责将一个或多个 `META-`

`META-INF/aop.xml` 文件中的方面配置编织到应用程序的类中。好处是，它不需要很多配置(您可以指定一些其他选项，但是稍后会详细介绍)，如以下示例所示：

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

    <!-- a service object; we will be profiling its methods -->
    <bean id="entitlementCalculationService"
          class="foo.StubEntitlementCalculationService"/>

    <!-- this switches on the load-time weaving -->
    <context:load-time-weaver/>
</beans>
```

现在，所有必需的构件(方面，`META-INF/aop.xml` 文件和 Spring 配置)都就位了，我们可以使用 `main(...)` 方法创建以下驱动程序类，以演示实际的 LTW：

```
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。本节的引言确实说过，可以使用 Spring 以 `ClassLoader` 为基础选择性地打开 LTW，这是事实。但是，在此示例中，我们使用 Java 代理(Spring 随附)打开 LTW。我们使用以下命令运行前面显示的 `Main` 类：

```
java -javaagent:C:/projects/foo/lib/global/spring-instrument.jar foo.Main
```

`-javaagent` 是用于指定和启用代理来检测在 [JVM 上运行的程序](#) 的标志。Spring 框架附带了这样的代理 `InstrumentationSavingAgent`，该代理打包在 `spring-instrument.jar` 中，在上一示例中，该代理作为 `-javaagent` 自变量的值提供。

`Main` 程序的执行输出类似于下一个示例。（我在 `calculateEntitlement()` 实现中引入了 `Thread.sleep(...)` 语句，以便探查器实际上捕获的不是 0 毫秒（`01234` 毫秒不是 AOP 引入的开销）。以下清单显示了运行探查器时得到的输出：

```
Calculating entitlement

StopWatch 'ProfilingAspect': running time (millis) = 1234
-----
ms      %      Task name
-----
01234  100%  calculateEntitlement
```

由于此 LTW 是通过使用成熟的 AspectJ 来实现的，因此我们不仅限于建议 Spring Bean。`Main` 程序的以下细微变化会产生相同的结果：

```
package foo;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public final class Main {
    public static void main(String[] args) {
        new ClassPathXmlApplicationContext("beans.xml", Main.class);
        EntitlementCalculationService entitlementCalculationService =
            new StubEntitlementCalculationService();
        // the profiling aspect will be 'woven' around this method execution
        entitlementCalculationService.calculateEntitlement();
    }
}
```

注意，在前面的程序中，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。剖析建议仍会被应用。

诚然，这个例子很简单。但是，在前面的示例中已经介绍了 Spring 对 LTW 支持的基础，本节的其余部分详细解释了每一位配置和用法的“原因”。

## iNote

在此示例中使用的 `ProfilingAspect` 可能是基本的，但很有用。这是开发时方面的一个很好的示例，开发人员可以在开发过程中使用它，然后轻松地将其从部署到 UAT 或 Producing 的应用程序构建中排除。

## Aspects

您在 LTW 中使用的方面必须是 AspectJ 方面。您可以使用 AspectJ 语言本身来编写它们，也可以使用@AspectJ 风格来编写方面。这样，您的方面就是有效的 AspectJ 和 Spring AOP 方面。此外，编译的方面类需要在 Classpath 上可用。

### 'META-INF/aop.xml'

通过使用 JavaClasspath 上的一个或多个 `META-INF/aop.xml` 文件(直接或通常在 jar 文件中)来配置 AspectJ LTW 基础结构。

该文件的结构和内容在 LTW 部分[AspectJ 参考文档](#)中详细介绍。由于 `aop.xml` 文件是 100% AspectJ，因此在此不再赘述。

### 必需的库(JARS)

至少，您需要以下库来使用 Spring Framework 对 AspectJ LTW 的支持：

- `spring-aop.jar` (2.5 版或更高版本，以及所有强制性依赖项)
- `aspectjweaver.jar` (1.6.8 版或更高版本)

如果您使用[Spring 提供的代理程序可实现检测](#)，则还需要：

- `spring-instrument.jar`

## Spring Configuration

Spring 的 LTW 支持中的关键组件是 `LoadTimeWeaver` 接口(在

`org.springframework.instrument.classloading` 包中), 以及 Spring 发行版附带的众多实现

- `LoadTimeWeaver` 负责在运行时将一个或多个

`java.lang.instrument.ClassFileTransformers` 添加到 `ClassLoader`, 这为各种有趣的应用

程序打开了大门, 其中之一恰好是方面的 LTW。

### Tip

如果您不熟悉运行时类文件转换的概念, 请在 `continue` 之前先查看

`java.lang.instrument` 软件包的 javadoc API 文档。虽然该文档并不全面, 但是至少您可

以看到关键的接口和类(在您阅读本节时作为参考)。

为特定的 `ApplicationContext` 配置 `LoadTimeWeaver` 就像添加一行一样容易。(请注意, 您几乎肯定需要使用 `ApplicationContext` 作为您的 Spring 容器—通常, `BeanFactory` 是不够的, 因为 LTW 支持使用 `BeanFactoryPostProcessors`.)

要启用 Spring Framework 的 LTW 支持, 您需要配置 `LoadTimeWeaver`, 通常通过使用

`@EnableLoadTimeWeaving` 注解来完成, 如下所示:

```
@Configuration  
@EnableLoadTimeWeaving  
public class AppConfig {  
}
```

另外, 如果您更喜欢基于 XML 的配置, 请使用 `<context:load-time-weaver/>` 元素。请注意, 该元素是在 `context` 名称空间中定义的。以下示例显示了如何使用 `<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
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

<context:load-time-weaver/>

</beans>

```

前面的配置会自动为您定义并注册许多 LTW 特定的基础结构 Bean，例如 `LoadTimeWeaver` 和 `AspectJWeavingEnabler`。缺省 `LoadTimeWeaver` 是 `DefaultContextLoadTimeWeaver` 类，它将尝试装饰自动检测到的 `LoadTimeWeaver`。“自动检测到”的 `LoadTimeWeaver` 的确切类型取决于您的运行时环境。下表总结了各种 `LoadTimeWeaver` 实现：

\*表 13. DefaultContextLoadTimeWeaver LoadTimeWeavers \*

Runtime Environment	<code>LoadTimeWeaver</code> 实施
在 Oracle 的 <a href="#">WebLogic</a> 中运行	<code>WebLogicLoadTimeWeaver</code>
在 Oracle 的 <a href="#">GlassFish</a> 中运行	<code>GlassFishLoadTimeWeaver</code>
在 <a href="#">Apache Tomcat</a> 中运行	<code>TomcatLoadTimeWeaver</code>
在 Red Hat 的 <a href="#">JBoss AS</a> 或 <a href="#">WildFly</a> 中运行	<code>JBossLoadTimeWeaver</code>
在 IBM 的 <a href="#">WebSphere</a> 中运行	<code>WebSphereLoadTimeWeaver</code>
JVM 从 Spring  <code>InstrumentationSavingAgent ( java -</code> <code>javaagent:path/to/spring-</code>	<code>InstrumentationLoadTimeWeaver</code>

<p>Runtime Environment</p> <p><code>instrument.jar</code>)开始回退，期望基础 ClassLoader 遵循通用约定(例如适用于</p> <p><code>TomcatInstrumentableClassLoader</code> 和 <code>Resin</code>)</p>	<p><code>LoadTimeWeaver</code> 实施</p> <p><code>ReflectiveLoadTimeWeaver</code></p>
--	--

请注意，该表仅列出了使用 `DefaultContextLoadTimeWeaver` 时自动检测到的 `LoadTimeWeavers`。您可以确切指定要使用的 `LoadTimeWeaver` 实现。

要使用 Java 配置指定特定的 `LoadTimeWeaver`，请实现 `LoadTimeWeavingConfigurer` 接口并覆盖 `getLoadTimeWeaver()` 方法。以下示例指定 `ReflectiveLoadTimeWeaver`：

```
@Configuration
@EnableLoadTimeWeaving
public class AppConfig implements LoadTimeWeavingConfigurer {

    @Override
    public LoadTimeWeaver getLoadTimeWeaver() {
        return new ReflectiveLoadTimeWeaver();
    }
}
```

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/>` 元素上的 `weaver-class` 属性的值。同样，以下示例指定 `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
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context.xsd">
```

```

<context:load-time-weaver
    weaver-class="org.springframework.instrument.classloading.ReflectiveLoadTim
</beans>

```

以后可以使用众所周知的名称 `LoadTimeWeaver` 从 Spring 容器中检索由配置定义和注册的 `LoadTimeWeaver`。请记住，`LoadTimeWeaver` 仅作为 Spring 的 LTW 基础结构添加一个或多个 `ClassFileTransformers` 的机制而存在。执行 LTW 的实际 `ClassFileTransformer` 是 `ClassPreProcessorAgentAdapter` (来自 `org.aspectj.weaver.loadtime` 程序包)类。有关更多详细信息，请参见 `ClassPreProcessorAgentAdapter` 类的类级 javadoc，因为实际上如何实现编织的细节不在本文档的讨论范围之内。

还需要讨论配置的最后一个属性：`aspectjWeaving` 属性(如果使用 XML，则为 `aspectj-weaving`)。此属性控制是否启用 LTW。它接受三个可能的值之一，如果属性不存在，则默认值为 `autodetect`。下表总结了三个可能的值：

表 14. AspectJ 编织属性值

Annotation Value	XML Value	Explanation
<code>ENABLED</code>	<code>on</code>	AspectJ 正在编织，并且在加载时适当地编织了方面。
<code>DISABLED</code>	<code>off</code>	LTW 已关闭。加载时不会编织任何方面。
<code>AUTODETECT</code>	<code>autodetect</code>	如果 Spring LTW 基础结构可以找到至少一个 <code>META-INF/aop.xml</code> 文件，则表示 AspectJ 编织已开始。否则，它关闭。这是默认值。

## Environment-specific Configuration

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 支持时所需的所有其他设置和配置。

### Tomcat

从历史上看，[Apache Tomcat](#)的默认类加载器不支持类转换，因此 Spring 提供了增强的实现来满足此需求。名为 `TomcatInstrumentableClassLoader` 的加载程序可在 Tomcat 6.0 及更高版本上运行。

#### Tip

不要在 Tomcat 8.0 及更高版本上定义 `TomcatInstrumentableClassLoader`。相反，让 Spring 通过 `TomcatLoadTimeWeaver` 策略自动使用 Tomcat 的新本机 `InstrumentableClassLoader` 工具。

如果仍然需要使用 `TomcatInstrumentableClassLoader`，则可以为每个 Web 应用程序分别进行注册，如下所示：

- 将 `org.springframework.instrument.tomcat.jar` 复制到 `$CATALINA_HOME/lib`，其中 `$CATALINA_HOME` 代表 Tomcat 安装的根目录
- 通过编辑 Web 应用程序上下文文件，指示 Tomcat 使用自定义类加载器(而不是默认值)，如以下示例所示：

```
<Context path="/myWebApp" docBase="/my/webApp/location">
    <Loader
        loaderClass="org.springframework.instrument.classloading.tomcat.TomcatInstrumen
    </Context>
```

Apache Tomcat 6.0 支持多个上下文位置：

- 服务器配置文件：`$CATALINA_HOME/conf/server.xml`

- 默认上下文配置：`$CATALINA_HOME/conf/context.xml`，这会影响所有已部署的 Web 应用程序
- 每个 Web 应用程序配置，可以在`$CATALINA_HOME/conf/[engineName]/[hostname]/[webapp]-context.xml`部署在服务器端，也可以在`META-INF/context.xml`嵌入在 Web 应用程序存档中

为了提高效率，我们建议使用嵌入式的逐个 Web 应用程序配置样式，因为它只影响使用自定义类加载器的应用程序，并且不需要对服务器配置进行任何更改。有关可用上下文位置的更多详细信息，请参见 Tomcat 6.0.x [documentation](#)。

或者，考虑使用 Spring 提供的通用 VM 代理，该代理在 Tomcat 的启动脚本中指定(本节前面已描述)。这使得检测对所有已部署的 Web 应用程序均可用，无论它们运行在 `ClassLoader` 上。

## WebLogic，WebSphere，Resin，GlassFish 和 JBoss

最新版本的 WebLogic Server(版本 10 和更高版本)，IBM WebSphere Application Server(版本 7 和更高版本)，Resin(版本 3.1 和更高版本)以及 JBoss(版本 6.x 或更高版本)提供了 `ClassLoader` 并能够进行本地检测。Spring 的本地 LTW 利用此类 `ClassLoader` 实现来实现 AspectJ 编织。您可以通过激活加载时编织来启用 LTW，如[described earlier](#)。具体来说，您无需修改启动脚本即可添加`-javaagent:path/to/spring-instrument.jar`。

请注意，具有 GlassFish 工具功能的 `ClassLoader` 仅在其 EAR 环境中可用。对于 GlassFish Web 应用程序，请遵循 Tomcat 设置说明[outlined earlier](#)。

请注意，在 JBoss 6.x 上，您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将名为`WEB-INF/jboss-scanning.xml`的文件添加到您的工件中，其中包含以下内容：

```
<scanning xmlns="urn:jboss:scanning:1.0"/>
```

在不支持或现有 `LoadTimeWeaver` 实现不支持的环境中需要类检测时, JDK 代理可以是唯一的解决方案。对于这种情况, Spring 提供了 `InstrumentationLoadTimeWeaver`, 这需要一个 Spring 特定(但非常通用)的 VM 代理 `org.springframework.instrument-{version}.jar` (以前称为 `spring-agent.jar`)。

要使用它, 您必须通过提供以下 JVM 选项来使用 Spring 代理启动虚拟机:

```
-javaagent:/path/to/org.springframework.instrument-{version}.jar
```

请注意, 这需要修改 VM 启动脚本, 这可能会阻止您在应用程序服务器环境中使用它(取决于您的操作策略)。此外, JDK 代理会检测整个 VM, 这可能会很昂贵。

出于性能原因, 我们建议仅在目标环境(例如[Jetty](#))没有(或不支持)专用 LTW 的情况下才使用此配置。

## 5.11. 更多资源

可以在[AspectJ website](#)上找到有关 AspectJ 的更多信息。

- Adrian Colyer 等人的《Eclipse AspectJ \*》。等(Addison-Wesley, 2005 年)为 AspectJ 语言提供了全面的介绍和参考。

强烈推荐 Ramnivas Laddad(Manning, 2009)出版的《AspectJ in Action \*》第二版。本书的重点是 AspectJ, 但在一定程度上探讨了许多通用的 AOP 主题。

## 6. Spring AOP API

上一章使用`@AspectJ` 和基于模式的方面定义描述了 Spring 对 AOP 的支持。在本章中, 我们讨论较低级别的 Spring AOP API 和通常在 Spring 1.2 应用程序中使用的 AOP 支持。对于新应用程序, 我们建议使用上一章中介绍的 Spring 2.0 和更高版本的 AOP 支持。但是, 当您使用现有应用程序(或阅读书籍和文章)时, 可能会遇到 Spring 1.2 样式的示例。Spring 5 仍然与 Spring 1.2 向后兼容, Spring 5 完全支持本章中描述的所有内容。

## 6.1. Spring 中的 Pointcut API

本节描述了 Spring 如何处理关键切入点概念。

### 6.1.1. Concepts

Spring 的切入点模型使切入点重用不受建议类型的影响。您可以使用相同的切入点来定位不同的建议。

`org.springframework.aop.Pointcut` 界面是中央界面，用于将建议定向到特定的类和方法。完整的界面如下：

```
public interface Pointcut {  
    ClassFilter getClassFilter();  
    MethodMatcher getMethodMatcher();  
}
```

将 `Pointcut` 接口分为两部分，可以重用类和方法匹配的部分以及细粒度的合成操作(例如与另一个方法匹配器执行“联合”)。

`ClassFilter` 接口用于将切入点限制为给定的一组目标类。如果 `matches()` 方法始终返回 `true`，则匹配所有目标类。以下清单显示了 `ClassFilter` 接口定义：

```
public interface ClassFilter {  
    boolean matches(Class clazz);  
}
```

`MethodMatcher` 界面通常更重要。完整的界面如下：

```
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。创建 AOP 代理时可以执行此评估，以避免需要对每个方法调用进行测试。如果给定方法的两个参数的 `matches` 方法返回 `true`，而用于 `MethodMatcher` 的 `isRuntime()` 的方法返回 `true`，则在每次方法调用时都会调用三个参数的 `match` 方法。这样，切入点就可以在执行目标建议之前立即查看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`。在这种情况下，永远不会调用三参数 `matches` 方法。

### Tip

如果可能，请尝试使切入点成为静态，从而在创建 AOP 代理时允许 AOP 框架缓存切入点评估的结果。

## 6.1.2. 切入点的操作

Spring 支持切入点上的操作(特别是联合和相交)。

联合表示两个切入点都匹配的方法。交集是指两个切入点都匹配的方法。联合通常更有用。您可以使用 `org.springframework.aop.support.Pointcuts` 类中的静态方法或同一包中的 `ComposablePointcut` 类来编写切入点。但是，使用 AspectJ 切入点表达式通常是一种更简单的办法。

## 6.1.3. AspectJ 表达切入点

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，该切入点使用 AspectJ 提供的库来解析 AspectJ 切入点表达式字符串。

有关支持的 AspectJ 切入点 Primitives 的讨论，请参见[previous chapter](#)。

## 6.1.4. 便捷切入点实现

Spring 提供了几种方便的切入点实现。您可以直接使用其中一些。其他的则应归入特定于应用程序的切入点中。

### Static Pointcuts

静态切入点是基于方法和目标类的，不能考虑方法的参数。静态切入点足以满足大多数用途，并且最好。首次调用方法时，Spring 只能评估一次静态切入点。之后，无需在每次方法调用时再次评估切入点。

本节的其余部分描述了 Spring 附带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种明显方法是正则表达式。除了 Spring 之外，还有几个 AOP 框架使之成为可能。`org.springframework.aop.support.JdkRegexpMethodPointcut` 是通用正则表达式切入点，它使用 JDK 中的正则表达式支持。

使用 `JdkRegexpMethodPointcut` 类，可以提供模式字符串列表。如果其中任何一个匹配，则切入点的值为 `true`。（因此，结果实际上是这些切入点的并集。）

以下示例显示了如何使用 `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` 的便捷类，它使我们也可以引用

`Advice`（请记住 `Advice` 可以是拦截器，而不是建议，引发建议等）。在后台，Spring 使用

`JdkRegexpMethodPointcut`。使用 `RegexpMethodPointcutAdvisor` 简化了接线，因为一个

bean 封装了切入点和建议，如以下示例所示：

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

您可以将 `RegexpMethodPointcutAdvisor` 与任何 `Advice` 类型一起使用。

## Attribute-driven Pointcuts

静态切入点的一种重要类型是元数据驱动的切入点。这使用元数据属性的值(通常是源级别的元数据)。

## Dynamic pointcuts

动态切入点比静态切入点更昂贵。它们考虑了方法参数以及静态信息。这意味着必须在每次方法调用时对它们进行评估，并且由于参数会有所不同，因此无法缓存结果。

主要示例是 `control flow` 切入点。

### 控制流切入点

Spring 控制流切入点在概念上类似于 AspectJ `cflow` 切入点，但功能较弱。(当前无法指定一个切入点在与另一个切入点匹配的连接点下执行。)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，则可能会触发。通过使用 `org.springframework.aop.support.ControlFlowPointcut` 类指定控制流切入点。

#### • Note

与其他动态切入点相比，控制流切入点在运行时进行评估要昂贵得多。在 Java 1.4 中，成本

大约是其他动态切入点的五倍。

## 6.1.5. 切入点超类

Spring 提供了有用的切入点超类，以帮助您实现自己的切入点。

因为静态切入点最有用，所以您可能应该子类 `StaticMethodMatcherPointcut`。这仅需要实现一个抽象方法(尽管您可以覆盖其他方法以自定义行为)。以下示例显示了如何对 `StaticMethodMatcherPointcut` 进行子类化：

```
class TestStaticPointcut extends StaticMethodMatcherPointcut {  
    public boolean matches(Method m, Class targetClass) {  
        // return true if custom criteria match  
    }  
}
```

动态切入点也有超类。

在 Spring 1.0 RC2 及更高版本中，您可以将自定义切入点与任何建议类型一起使用。

## 6.1.6. 自定义切入点

由于 Spring AOP 中的切入点是 Java 类，而不是语言功能(如 AspectJ)，因此可以声明自定义切入点，无论是静态还是动态。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 切入点表达语言。

### ①Note

更高版本的 Spring 可能提供对 JAC offered 提供的“语义切入点”的支持，例如，“更改目标对象中实例变量的所有方法”。

## 6.2. Spring 咨询 API

现在，我们可以检查 Spring AOP 如何处理建议。

## 6.2.1. 咨询生命周期

每个建议都是一个 Spring bean。建议实例可以在所有建议对象之间共享，或者对于每个建议对象都是唯一的。这对应于每个类或每个实例的建议。

每班建议最常用。适用于一般建议，例如 Transaction 顾问。这些不依赖于代理对象的状态或添加新状态。它们仅作用于方法和参数。

每个实例的建议都适合引入，以支持混合。在这种情况下，建议将状态添加到代理对象。

您可以在同一 AOP 代理中混合使用共享建议和基于实例的建议。

## 6.2.2. Spring 的建议类型

Spring 提供了几种建议类型，并且可以扩展以支持任意建议类型。本节介绍基本概念和标准建议类型。

### 围绕建议进行拦截

Spring 中最基本的建议类型是围绕建议的拦截。

对于使用方法拦截的建议，Spring 符合 AOP `Alliance` 接口。实现 `MethodInterceptor` 并围绕建议实现的类也应实现以下接口：

```
public interface MethodInterceptor extends Interceptor {  
    Object invoke(MethodInvocation invocation) throws Throwable;  
}
```

`invoke()` 方法的 `MethodInvocation` 参数公开了正在调用的方法，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。

以下示例显示了一个简单的 `MethodInterceptor` 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;
    }
}
```

请注意对 `MethodInvocation` 的 `proceed()` 方法的调用。这沿着拦截器链向下到达连接点。大多数拦截器都调用此方法并返回其返回值。但是，`MethodInterceptor` 就像任何周围的建议一样，可以返回不同的值或引发异常，而不是调用 `proceed` 方法。但是，您没有充分的理由就不想这样做。

### 1 Note

`MethodInterceptor` 实现提供与其他符合 AOP Alliance 要求的 AOP 实现的互操作性。本节其余部分讨论的其他建议类型将实现常见的 AOP 概念，但以特定于 Spring 的方式。尽管使用最具体的建议类型有一个优势，但是如果可能想在另一个 AOP 框架中运行方面，请坚持使用 `MethodInterceptor`。请注意，切入点当前无法在框架之间互操作，并且 AOP Alliance 当前未定义切入点接口。

## Before Advice

一种更简单的建议类型是事前建议。不需要 `MethodInvocation` 对象，因为它仅在进入方法之前被调用。

先行建议的主要优点是无需调用 `proceed()` 方法，因此，不会无意中未能沿拦截器链 `continue` 前进。

以下清单显示了 `MethodBeforeAdvice` 界面：

```
public interface MethodBeforeAdvice extends BeforeAdvice {
    void before(Method m, Object[] args, Object target) throws Throwable;
}
```

(尽管通常的对象适用于字段拦截，并且 Spring 不太可能实现它，但 Spring 的 API 设计允许先于字

段咨询。)

请注意，返回类型为 `void`。通知可以在联接点执行之前插入自定义行为，但不能更改返回值。如果之前的建议引发异常，它将中止拦截器链的进一步执行。异常会传播回拦截器链。如果未选中它或在调用的方法的签名上，则将其直接传递给 Client 端。否则，它将被 AOP 代理包装在未经检查的异常中。

以下示例显示了 Spring 中的 `before` 建议，该建议计算所有方法调用：

```
public class CountingBeforeAdvice implements MethodBeforeAdvice {  
  
    private int count;  
  
    public void before(Method m, Object[] args, Object target) throws Throwable {  
        ++count;  
    }  
  
    public int getCount() {  
        return count;  
    }  
}
```

### Tip

在将建议与任何切入点一起使用之前。

## Throws Advice

如果联接点引发异常，则在联接点返回之后调用引发通知。Spring 提供类型化的抛出建议。请注意，这意味着 `org.springframework.aop.ThrowsAdvice` 接口不包含任何方法。它是一个标签接口，用于标识给定对象实现了一个或多个类型化的 `throws` 通知方法。这些应采用以下形式：

```
afterThrowing([Method, args, target], subclassOfThrowable)
```

仅最后一个参数是必需的。方法签名可以具有一个或四个参数，具体取决于建议方法是否对该方法和参数感兴趣。接下来的两个清单显示了类，它们是引发建议的示例。

如果抛出 `RemoteException` (包括来自子类)，则调用以下建议：

```
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，下一个示例声明四个参数，以便可以访问被调用的方法，方法参数和目标对象。如果抛出 `ServletException`，则调用以下建议：

```
public class ServletThrowsAdviceWithArguments implements ThrowsAdvice {  
    public void afterThrowing(Method m, Object[] args, Object target, ServletException ex) throws Throwable {  
        // Do something with all arguments  
    }  
}
```

最后一个示例说明如何在处理 `RemoteException` 和 `ServletException` 的单个类中使用这两种方法。可以将任意数量的引发建议方法组合到一个类中。以下清单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) throws Throwable {  
        // Do something with all arguments  
    }  
}
```

### iNote

如果 `throws-advice` 方法本身引发异常，则它将覆盖原始异常(也就是说，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 `throws-advice` 方法抛出一个已检查的异常，则它必须与目标方法的已声明异常匹配，因此在某种程度上与特定的目标方法签名耦合。请勿抛出与目标方法签名不兼容的未声明检查异常！

### Tip

抛出建议可以与任何切入点一起使用。

## 返回建议后

在 Spring 中返回通知后，必须实现 `org.springframework.aop.AfterReturningAdvice` 接口，以下清单显示了该接口：

```
public interface AfterReturningAdvice extends Advice {  
    void afterReturning(Object returnValue, Method m, Object[] args, Object target)  
        throws Throwable;  
}
```

After After Returning 建议可以访问返回值(它不能修改)，调用的方法，方法的参数和目标。

返回建议后的以下内容将计数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;  
    }  
}
```

该建议不会更改执行路径。如果抛出异常，则会将其抛出拦截器链，而不是返回值。

### Tip

返回后，建议可以与任何切入点一起使用。

## Introduction Advice

Spring 将介绍建议视为一种特殊的拦截建议。

简介需要 `IntroductionAdvisor` 和 `IntroductionInterceptor` 来实现以下接口：

```
public interface IntroductionInterceptor extends MethodInterceptor {  
    boolean implementsInterface(Class intf);  
}
```

从 AOP Alliance `MethodInterceptor` 接口继承的 `invoke()` 方法必须实现介绍。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()`。

简介建议不能与任何切入点一起使用，因为它仅适用于类，而不适用于方法级别。您只能通过 `IntroductionAdvisor` 使用介绍建议，该建议具有以下方法：

```
public interface IntroductionAdvisor extends Advisor, IntroductionInfo {  
    ClassFilter getClassFilter();  
  
    void validateInterfaces() throws IllegalArgumentException;  
}  
  
public interface IntroductionInfo {  
    Class[] getInterfaces();  
}
```

没有 `MethodMatcher`，因此也没有 `Pointcut` 与介绍建议相关联。只有类过滤是合乎逻辑的。

`getInterfaces()` 方法返回此顾问程序引入的接口。

在内部使用 `validateInterfaces()` 方法来查看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，并假设我们想为一个或多个对象引入以下接口：

```
public interface Lockable {  
    void lock();  
    void unlock();  
    boolean locked();  
}
```

这说明了混合。我们希望能够将建议对象强制转换为 `Lockable`，无论它们的类型如何，并调用锁定和解锁方法。如果调用 `lock()` 方法，则希望所有的 `setter` 方法都抛出 `LockedException`。因

此，我们可以添加一个方面，使对象在不了解对象的情况下不可变：AOP 的一个很好的例子。

首先，我们需要一个 `IntroductionInterceptor` 来完成繁重的工作。在这种情况下，我们扩展了 `org.springframework.aop.support.DelegatingIntroductionInterceptor` 便利类。我们可以直接实现 `IntroductionInterceptor`，但是在大多数情况下最好使用 `DelegatingIntroductionInterceptor`。

`DelegatingIntroductionInterceptor` 的设计宗旨是将引入的接口委派给引入接口的实际实现，从而隐藏使用拦截的方式。您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)为 `this`。因此，在下一个示例中，委托是

`DelegatingIntroductionInterceptor` 的 `LockMixin` 子类。给定一个委托(默认情况下为自身)，`DelegatingIntroductionInterceptor` 实例将查找由委托实现的所有接口(`IntroductionInterceptor` 除外)，并支持针对其中任何接口的介绍。诸如 `LockMixin` 之类的子类可以调用 `suppressInterface(Class intf)` 方法来禁止不应公开的接口。但是，无论 `IntroductionInterceptor` 准备支持多少个接口，`IntroductionAdvisor` 使用的控件都会控制实际公开哪些接口。引入的接口隐藏了目标对同一接口的任何实现。

因此，`LockMixin` 扩展了 `DelegatingIntroductionInterceptor` 并实现了 `Lockable` 本身。超类会自动选择支持 `Lockable` 进行介绍，因此我们无需指定。我们可以通过这种方式引入任意数量的接口。

请注意 `locked` 实例变量的使用。这有效地将附加状态添加到目标对象中保存的状态。

下面的示例显示示例 `LockMixin` 类：

```
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，您无需覆盖 `invoke()` 方法。通常，`DelegatingIntroductionInterceptor` 实现(如果引入了 `delegate` 方法，则调用 `delegate` 方法，否则 `continue` 向连接点前进)。在当前情况下，我们需要添加一个检查：如果处于锁定模式，则不能调用任何 `setter` 方法。

所需的简介仅需要保存一个不同的 `LockMixin` 实例并指定所引入的接口(在这种情况下，只需 `Lockable`)。一个更复杂的示例可能引用了引入拦截器(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。以下示例显示了 `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此顾问程序，因为它不需要配置。(但是，如果没有 `IntroductionAdvisor`，就不可能使用 `IntroductionInterceptor`)像通常的介绍一样，顾问程序必须是按实例的，因为它是有状态的。对于每个建议对象，我们需要 `LockMixinAdvisor` 的不同实例，因此需要 `LockMixin`。顾问程序包含建议对象状态的一部分。

我们可以像其他任何顾问一样，通过使用 `Advised.addAdvisor()` 方法或 XML 配置中的(推荐方式)以编程方式应用此顾问。下面讨论的所有代理创建选择，包括“自动代理创建器”，都可以正确处理介绍和有状态的混合。

## 6.3. Spring 的 Advisor API

在 Spring 中，顾问程序是一个方面，其中仅包含与切入点表达式关联的单个建议对象。

除了介绍的特殊情况外，任何顾问都可以与任何建议一起使用。

`org.springframework.aop.support.DefaultPointcutAdvisor` 是最常用的顾问类。可以与 `MethodInterceptor`，`BeforeAdvice` 或 `ThrowsAdvice` 一起使用。

可以在同一 AOP 代理中的 Spring 中混合使用顾问和建议类型。例如，您可以在一个代理配置中使用围绕建议的拦截，抛出建议以及在建议之前。Spring 自动创建必要的拦截器链。

## 6.4. 使用 ProxyFactoryBean 创建 AOP 代理

如果将 Spring IoC 容器(`ApplicationContext` 或 `BeanFactory`)用于业务对象(应该是!)，则要使用 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。)

### Note

Spring AOP 支持还在后台使用了工厂 bean。

在 Spring 中创建 AOP 代理的基本方法是使用

`org.springframework.aop.framework.ProxyFactoryBean`。这样可以完全控制切入点，任何适用的建议及其 Sequences。但是，如果您不需要这样的控制，则有一些更简单的选项是可取的。

### 6.4.1. Basics

像其他 Spring `FactoryBean` 实现一样，`ProxyFactoryBean` 引入了一个间接级别。如果定义一个名为 `foo` 的 `ProxyFactoryBean`，则引用 `foo` 的对象看不到 `ProxyFactoryBean` 实例本身，而是看到通过 `ProxyFactoryBean` 中的 `getObject()` 方法的实现创建的对象。此方法创建一个包装目标对象的 AOP 代理。

使用 `ProxyFactoryBean` 或另一个 IoC 感知类创建 AOP 代理的最重要好处之一是，IoC 也可以 Management 建议和切入点。这是一项强大的功能，可实现某些其他 AOP 框架难以实现的方法。例如，受益于依赖注入提供的所有可插入性，建议本身可以引用应用程序对象(目标之外，目标应该在任何 AOP 框架中可用)。

## 6.4.2. JavaBean 属性

与 Spring 提供的大多数 `FactoryBean` 实现一样，`ProxyFactoryBean` 类本身就是 JavaBean。其属性用于：

- 指定您要代理的目标。
- 指定是否使用 CGLIB(稍后介绍，另请参见[基于 JDK 和 CGLIB 的代理](#))。

一些关键属性是从 `org.springframework.aop.framework.ProxyConfig` (Spring 中所有 AOP 代理工厂的超类)继承的。这些关键属性包括：

- `proxyTargetClass` : `true` (如果要代理目标类，而不是目标类的接口)。如果此属性值设置为 `true`，那么将创建 CGLIB 代理(但也请参见[基于 JDK 和 CGLIB 的代理](#))。
- `optimize` : 控制是否将积极的优化应用于通过 CGLIB 创建的代理。除非您完全了解相关的 AOP 代理如何处理优化，否则不要随意使用此设置。当前仅用于 CGLIB 代理。它对 JDK 动态代理无效。
- `frozen` : 如果代理配置为 `frozen`，则不再允许更改配置。这是一个轻微的优化，对于在您不希望调用者在创建代理后能够(通过 `Advised` 接口)操纵代理的情况下很有用。此属性的默认值为 `false`，因此允许进行更改(例如添加其他建议)。
- `exposeProxy` : 确定是否应在 `ThreadLocal` 中公开当前代理，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`，则目标可以使用 `AopContext.currentProxy()` 方法。

`ProxyFactoryBean` 特有的其他属性包括：

- `proxyInterfaces` : `String` 接口名称的数组。如果未提供，则使用目标类的 CGLIB 代理(另请参见[基于 JDK 和 CGLIB 的代理](#))。
- `interceptorNames` : 要应用的 `Advisor`，拦截器或其他建议名称的 `String` 数组。

Sequences 很重要，先到先得。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，包括祖先工厂中的 bean 名称。您不能在这里提及 bean 引用，因为这样做会导致 `ProxyFactoryBean` 忽略建议的单例设置。

您可以在拦截器名称后加上星号(`*`)。这样做会导致应用所有顾问 Bean，其名称以要应用星号的部分开头。您可以在[使用“全局”顾问](#)中找到使用此功能的示例。

- 单例：无论调用 `getObject()` 方法的频率如何，工厂是否应返回单个对象。几种 `FactoryBean` 实现提供了这种方法。默认值为 `true`。如果要使用状态通知(例如，对于状态混合)，请使用原型建议以及 `false` 的单例值。

### 6.4.3. 基于 JDK 和 CGLIB 的代理

本部分是有关 `ProxyFactoryBean` 如何选择为特定目标对象(将被代理)创建基于 JDK 的代理或基于 CGLIB 的代理的 Authority 性文档。

#### **Note**

在 Spring 的 1.2.x 版和 2.0 版之间，`ProxyFactoryBean` 创建基于 JDK 或 CGLIB 的代理的行为发生了变化。`ProxyFactoryBean` 现在在自动检测接口方面表现出与 `TransactionProxyFactoryBean` 类类似的语义。

如果要代理的目标对象的类(以下简称为目标类)没有实现任何接口，则将创建基于 CGLIB 的代理。

这是最简单的情况，因为 JDK 代理是基于接口的，并且没有接口意味着甚至无法进行 JDK 代理。您可以插入目标 bean 并通过设置 `interceptorNames` 属性来指定拦截器列表。请注意，即使 `ProxyFactoryBean` 的 `proxyTargetClass` 属性已设置为 `false`，也会创建基于 CGLIB 的代理。（这样做没有任何意义，最好将其从 Bean 定义中删除，因为它充其量是多余的，并且在最糟的情况下会造成混淆。）

如果目标类实现一个(或多个)接口，则创建的代理类型取决于 `ProxyFactoryBean` 的配置。

如果 `ProxyFactoryBean` 的 `proxyTargetClass` 属性已设置为 `true`，则会创建基于 CGLIB 的代理。这是有道理的，并且符合最小惊讶原则。即使已将 `ProxyFactoryBean` 的 `proxyInterfaces` 属性设置为一个或多个完全限定的接口名称，但 `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 `proxyInterfaces` 属性中指定的所有接口。如果目标类恰好实现了比 `proxyInterfaces` 属性中指定的接口更多的接口，那就很好了，但是这些附加接口不会由返回的代理实现。

如果尚未设置 `ProxyFactoryBean` 的 `proxyInterfaces` 属性，但是目标类确实实现了一个(或多个)接口，则 `ProxyFactoryBean` 自动检测到目标类实际上确实实现了至少一个接口以及基于 JDK 的代理被构建。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。但是，它的工作量大大减少，而且不容易出现印刷错误。

#### 6.4.4. 代理接口

考虑一个实际的 `ProxyFactoryBean` 的简单示例。此示例涉及：

- 代理的目标 bean。这是示例中的 `personTarget` bean 定义。

- `Advisor` 和 `Interceptor` 用于提供建议。
- 一个 AOP 代理 bean 定义，用于指定目标对象(`personTarget` bean)，代理接口以及要应用的建议。

以下清单显示了示例：

```
<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"/>
    <property name="interceptorNames">
        <list>
            <value>myAdvisor</value>
            <value>debugInterceptor</value>
        </list>
    </property>
</bean>
```

请注意，`interceptorNames` 属性具有 `String` 列表，其中包含当前工厂中的拦截器或顾问程序的 Bean 名称。您可以在返回之前，之后使用顾问程序，拦截器并引发建议对象。顾问的 Sequences 很重要。

### **Note**

您可能想知道为什么列表不包含 bean 引用。这样做的原因是，如果 `ProxyFactoryBean` 的 `singleton` 属性设置为 `false`，则它必须能够返回独立的代理实例。如果任何顾问本身就是原型，则需要返回一个独立的实例，因此必须能够从工厂获得原型的实例。保持引用是不够的。

可以使用前面显示的 `person` bean 定义代替 `Person` 实现，如下所示：

```
Person person = (Person) factory.getBean("person");
```

与普通 Java 对象一样，在同一 IoC 上下文中的其他 bean 可以表达对此的强类型依赖性。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>
```

在此示例中，`PersonUser` 类公开了 `Person` 类型的属性。就其而言，可以透明地使用 AOP 代理代替“真实的”人实现。但是，其类将是动态代理类。可以将其强制转换为 `Advised` 接口(稍后讨论)。

您可以使用匿名内部 bean 隐藏目标和代理之间的区别。只有 `ProxyFactoryBean` 定义不同。该建议仅出于完整性考虑。以下示例显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 的优点是只有一个类型为 `Person` 的对象。如果我们希望防止应用程序上下文的用户获取对未建议对象的引用，或者需要避免使用 Spring IoC 自动装配的任何歧义，这将非常有

用。可以说，还有一个优点是 `ProxyFactoryBean` 定义是独立的。但是，有时能够从工厂获得未经建议的目标实际上可能是一个优势(例如，在某些测试方案中)。

## 6.4.5. 代理类

如果您需要代理一类，而不是一个或多个接口，该怎么办？

想象一下，在我们之前的示例中，没有 `Person` 接口。我们需要建议一个名为 `Person` 的类，该类未实现任何业务接口。在这种情况下，您可以将 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` 的 `proxyTargetClass` 属性设置为 `true`。尽管最好对接口而不是对类进行编程，但是在处理遗留代码时，建议未实现接口的类的功能可能会很有用。(通常，Spring 并不是规定性的。虽然可以轻松地应用良好实践，但可以避免强制采用特定方法。)

如果需要，即使您有接口，也可以在任何情况下强制使用 CGLIB。

CGLIB 代理通过在运行时生成目标类的子类来工作。Spring 配置此生成的子类以将方法调用委托给原始目标。子类用于实现 Decorator 模式，并编织在建议中。

CGLIB 代理通常应对用户透明。但是，有一些问题要考虑：

- 不能建议 `Final` 方法，因为它们不能被覆盖。
- 无需将 CGLIB 添加到您的 Classpath 中。从 Spring 3.2 开始，CGLIB 被重新打包并包含在 `spring-core` JAR 中。换句话说，基于 CGLIB 的 AOP 就像 JDK 动态代理一样“开箱即用”。

CGLIB 代理和动态代理之间几乎没有性能差异。从 Spring 1.0 开始，动态代理要快一些。但是，将来可能会改变。在这种情况下，性能不应作为决定性的考虑因素。

## 6.4.6. 使用“全局”顾问

通过在拦截器名称后附加一个星号，所有具有与该星号之前的部分匹配的 Bean 名称的顾问程序都将添加到顾问程序链中。如果您需要添加标准的“全局”顾问程序集，这可能会派上用场。以下示例定义了两个全局顾问程序：

```
<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.PerformanceMon
```

## 6.5. 简洁的代理定义

特别是在定义事务代理时，您可能会得到许多类似的代理定义。使用父子 bean 定义和子 bean 定义以及内部 bean 定义可以使代理定义更加简洁明了。

首先，我们为代理创建父模板，bean 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>
```

它本身从未实例化，因此实际上可能是不完整的。然后，每个需要创建的代理都是一个子 bean 定义，它将代理的目标包装为内部 bean 定义，因为无论如何该目标都不会单独使用。以下示例显示了这样的子 bean：

```
<bean id="myService" parent="txProxyTemplate">
<property name="target">
    <bean class="org.springframework.samples.MyServiceImpl">
    </bean>
</property>
</bean>
```

您可以从父模板覆盖属性。在以下示例中，我们将覆盖事务传播设置：

```
<bean id="mySpecialService" parent="txProxyTemplate">
<property name="target">
    <bean class="org.springframework.samples.MySpecialServiceImpl">
    </bean>
```

```
</property>
<property name="transactionAttributes">
    <props>
        <prop key="get*>">PROPAGATION_REQUIRED,readOnly</prop>
        <prop key="find*>">PROPAGATION_REQUIRED,readOnly</prop>
        <prop key="load*>">PROPAGATION_REQUIRED,readOnly</prop>
        <prop key="store*>">PROPAGATION_REQUIRED</prop>
    </props>
</property>
</bean>
```

请注意，在父 bean 的示例中，我们通过将 `abstract` 属性设置为 `true` 来明确标记父 bean 定义为抽象，如[previously](#)所述，因此它实际上可能没有实例化。默认情况下，应用程序上下文(但不是简单的 bean 工厂)会预先实例化所有单例。因此，重要的是(至少对于单例 bean)，如果您有一个(父)bean 定义仅打算用作模板，并且此定义指定了一个类，则必须确保将 `abstract` 属性设置为 `true`。否则，应用程序上下文实际上会尝试对其进行实例化。

## 6.6. 使用 ProxyFactory 以编程方式创建 AOP 代理

使用 Spring 以编程方式创建 AOP 代理很容易。这使您可以使用 Spring AOP，而无需依赖 Spring IoC。

由目标对象实现的接口将被自动代理。以下清单显示了使用一个拦截器和一个顾问程序为目标对象创建代理的过程：

```
ProxyFactory factory = new ProxyFactory(myBusinessInterfaceImpl);
factory.addAdvice(myMethodInterceptor);
factory.addAdvisor(myAdvisor);
MyBusinessInterface tb = (MyBusinessInterface) factory.getProxy();
```

第一步是构造类型为 `org.springframework.aop.framework.ProxyFactory` 的对象。您可以用目标对象创建此对象，如前面的示例中所示，或指定要在备用构造函数中代理的接口。

您可以添加建议(使用拦截器作为一种特殊的建议)，建议程序，或同时添加两者，并在 `ProxyFactory` 的生命周期内对其进行操作。如果添加

`IntroductionInterceptionAroundAdvisor`，则可以使代理实现其他接口。

`ProxyFactory` (从 `AdvisedSupport` 继承) 上还有便捷的方法，可让您添加其他建议类型，例如 `before` 并引发建议。`AdvisedSupport` 是 `ProxyFactory` 和 `ProxyFactoryBean` 的超类。

### Tip

在大多数应用程序中，将 AOP 代理创建与 IoC 框架集成在一起是最佳实践。通常，建议您使用 AOP 从 Java 代码外部化配置。

## 6.7. 操作建议对象

无论创建 AOP 代理，都可以使用 `org.springframework.aop.framework.Advised` 界面对其进行操作。任何 AOP 代理都可以强制转换为该接口，无论它实现了哪个其他接口。该界面包括以下方法：

```
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()` 方法针对已添加到工厂的每个顾问程序，拦截器或其他建议类型返回 `Advisor`。如果添加了 `Advisor`，则在此索引处返回的顾问程序就是您添加的对象。如果添加了拦截器或其他建议类型，Spring 会将其包装在带有指向始终返回 `true` 的切入点的顾问程序中。因此，如果添加了 `MethodInterceptor`，则为此索引返回的顾问程序是 `DefaultPointcutAdvisor`，它返

回 `MethodInterceptor` 以及与所有类和方法匹配的切入点。

`addAdvisor()` 方法可用于添加任何 `Advisor`。通常，拥有切入点和建议的顾问是通用的

`DefaultPointcutAdvisor`，您可以将其与任何建议或切入点一起使用(但不能用于介绍)。

默认情况下，即使已创建代理，也可以添加或删除顾问程序或拦截器。唯一的限制是不可能添加或删除介绍顾问，因为工厂中的现有代理不会显示界面更改。(您可以从工厂获取新的代理来避免此问题。)

以下示例显示了将 AOP 代理投射到 `Advised` 接口并检查和处理其建议：

```
Advised advised = (Advised) myObject;
Advisor[] advisors = advised.getAdvisors();
int oldAdvisorCount = advisors.length;
System.out.println(oldAdvisorCount + " advisors");

// Add an advice like an interceptor without a pointcut
// Will match all proxied methods
// Can use for interceptors, before, after returning or throws advice
advised.addAdvice(new DebugInterceptor());

// Add selective advice using a pointcut
advised.addAdvisor(new DefaultPointcutAdvisor(mySpecialPointcut, myAdvice));

assertEquals("Added two advisors", oldAdvisorCount + 2, advised.getAdvisors().length);
```

### ①Note

尽管无疑存在合法的使用案例，但是是否建议(无双关语)修改 Producing 的业务对象的建议值得怀疑。但是，它在开发中(例如在测试中)非常有用。有时我们发现以拦截器或其他建议的形式添加测试代码，并进入我们要测试的方法调用中非常有用。(例如，建议可以进入为该方法创建的事务内部，也许可以在将事务标记为回滚之前运行 SQL 以检查数据库是否已正确更新。)

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`，而通过添加或删除来修改建议的任何尝试都将导致 `AopConfigException`。冻结建议对象状态的功能在某些情况下很有用(例如，防止调用代码删除安全拦截器)。如果已知不需要修改运行时建议，则在 Spring 1.1 中也可以使用它来进行积极的优化。

## 6.8. 使用“自动代理”功能

到目前为止，我们已经考虑过使用 `ProxyFactoryBean` 或类似的工厂 bean 来显式创建 AOP 代理。

Spring 还允许我们使用“自动代理” Bean 定义，该定义可以自动代理选定的 Bean 定义。这是在 Spring 的“bean 后处理器”基础结构上构建的，该基础结构允许在容器加载时修改任何 bean 定义。

在此模型中，您在 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。这使您可以声明有资格进行自动代理的目标。您无需使用 `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 的自动代理创建器。
- 自动代理创建的一种特殊情况，值得单独考虑：由源级元数据属性驱动的自动代理创建。

### 6.8.1. 自动代理 Bean 定义

本节介绍了 `org.springframework.aop.framework.autoproxy` 软件包提供的自动代理创建者。

#### BeanNameAutoProxyCreator

`BeanNameAutoProxyCreator` 类是 `BeanPostProcessor`，它会自动为名称与 Literals 值或通配符匹配的 bean 创建 AOP 代理。以下示例显示了如何创建 `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 `interceptorNames` 属性而不是拦截器列表，以允许原型顾问程序具有正确的行为。命名为“拦截器”的可以是顾问或任何建议类型。

通常，与自动代理一样，使用 `BeanNameAutoProxyCreator` 的要点是将相同的配置一致地应用于多个对象，并且配置量最少。将声明式事务应用于多个对象是一种流行的选择。

名称匹配的 Bean 定义(例如上例中的 `jdkMyBean` 和 `onlyJdk`)是带有目标类的普通旧 Bean 定义。

- `BeanNameAutoProxyCreator` 自动创建一个 AOP 代理。相同的建议适用于所有匹配的 bean。注意，如果使用了顾问程序(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，功能更强大的自动代理创建者是 `DefaultAdvisorAutoProxyCreator`。这可以在当前上下文中自动应用合格的顾问程序，而无需在自动代理顾问程序的 Bean 定义中包括特定的 Bean 名称。它具有与 `BeanNameAutoProxyCreator` 相同的一致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，因为必须有一个评估的切入点，以检查每个建议是否符合候选 bean 定义。

`DefaultAdvisorAutoProxyCreator` 自动评估每个顾问中包含的切入点，以查看它应应用于每个业务对象的建议(如果有)(例如示例中的 `businessObject1` 和 `businessObject2`)。

这意味着可以将任意数量的顾问程序自动应用于每个业务对象。如果在任何顾问程序中没有切入点与业务对象中的任何方法匹配，则该对象不会被代理。当为新的业务对象添加 Bean 定义时，如有必要，它们会自动被代理。

通常，自动代理的优点是使调用者或依赖者无法获得不建议的对象。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“inner bean”惯用语也提供了这一好处。)

以下示例创建一个 `DefaultAdvisorAutoProxyCreator` bean 和本节中讨论的其他元素：

```
<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"/>
```

如果要将相同的建议一致地应用于许多业务对象，则 `DefaultAdvisorAutoProxyCreator` 非常有用。基础结构定义到位后，您可以添加新的业务对象，而无需包括特定的代理配置。您还可以轻松地添加其他方面(例如，跟踪或性能监视方面)，而对配置的更改最少。

`DefaultAdvisorAutoProxyCreator` 支持过滤(通过使用命名约定，以便仅评估某些顾问程序，从而允许在同一工厂中使用多个不同配置的 AdvisorAutoProxyCreators)和排序。如果存在问题，顾问可以实现 `org.springframework.core.Ordered` 接口以确保正确的排序。前面示例中使用的 `TransactionAttributeSourceAdvisor` 具有可配置的 Sequences 值。默认设置为无序。

## 6.9. 使用 TargetSource 实现

Spring 提供了 `org.springframework.aop.TargetSource` 接口中表示的 `TargetSource` 的概念。该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，都会向 `TargetSource` 实现请求目标实例。

使用 Spring AOP 的开发人员通常不需要直接使用 `TargetSource` 实现，但这提供了支持池化，热插拔和其他复杂目标的强大方法。例如，池 `TargetSource` 可以通过使用池来 Management 实例，从而为每次调用返回不同的目标实例。

如果未指定 `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(与您

期望的一样)。

本节的其余部分描述了 Spring 随附的标准目标源以及如何使用它们。

### Tip

使用自定义目标源时，目标通常需要是原型而不是单例 bean 定义。这样，Spring 可以在需要时创建一个新的目标实例。

## 6.9.1. 可热交换的目标源

`org.springframework.aop.target.HotSwappableTargetSource` 的存在是为了允许 AOP 代理服务器的目标切换，同时允许呼叫者保留对其的引用。

更改目标源的目标会立即生效。`HotSwappableTargetSource` 是线程安全的。

您可以使用 `HotSwappableTargetSource` 上的 `swap()` 方法更改目标，如以下示例所示：

```
HotSwappableTargetSource swapper = (HotSwappableTargetSource) beanFactory.getBean("swapper");
Object oldTarget = swapper.swap(newTarget);
```

以下示例显示了必需的 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>
```

前面的 `swap()` 调用更改了可交换 bean 的目标。拥有对该 bean 的引用的 Client 端不知道更改，但立即开始达到新目标。

尽管此示例未添加任何建议(使用 `TargetSource` 无需添加建议)，但任何 `TargetSource` 均可与任意建议结合使用。

## 6.9.2. 汇集目标源

使用池目标源提供了与 Stateless 会话 EJB 相似的编程模型，在 Stateless 会话 EJB 中，维护了相同实例的池，方法调用将释放池中的对象。

Spring 池和 SLSB 池之间的关键区别在于，Spring 池可以应用于任何 POJO。通常，与 Spring 一样，可以以非侵入性方式应用此服务。

Spring 提供了对 Commons Pool 2.2 的支持，该池提供了相当有效的池实现。您需要在应用程序的 Classpath 上使用 `commons-pool` Jar 才能使用此功能。您还可以将

`org.springframework.aop.target.AbstractPoolingTargetSource` 子类化以支持任何其他池化 API。

### iNote

还支持 Commons Pool 1.5，但从 Spring Framework 4.2 开始不推荐使用。

以下清单显示了一个示例配置：

```
<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(上例中为 `businessObjectTarget`)必须是原型。这使

`PoolingTargetSource` 实现可以创建目标的新实例以根据需要扩展池。有关其属性的信息，请参见 [AbstractPoolingTargetSource 的 javadoc](#) 和您希望使用的具体子类。`maxSize` 是最基本的，  
并且始终保证存在。

在这种情况下，`myInterceptor` 是需要在同一 IoC 上下文中定义的拦截器的名称。但是，您无需指定拦截器即可使用池。如果只希望池化而没有其他建议，则根本不要设置 `interceptorNames` 属性。

您可以将 Spring 配置为能够将任何池化对象投射到

`org.springframework.aop.target.PoolingConfig` 接口，该接口通过介绍来公开有关池的配置和当前大小的信息。您需要定义类似于以下内容的顾问程序：

```
<bean id="poolConfigAdvisor" class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
    <property name="targetObject" ref="poolTargetSource"/>
    <property name="targetMethod" value="getPoolingConfigMixin"/>
</bean>
```

通过在 `AbstractPoolingTargetSource` 类上调用便捷方法(因此使用

`MethodInvokingFactoryBean`)可获得此顾问程序。该顾问的名称(此处为 `poolConfigAdvisor`)必须位于公开池对象的 `ProxyFactoryBean` 中的拦截器名称列表中。

演员表的定义如下：

```
PoolingConfig conf = (PoolingConfig) beanFactory.getBean("businessObject");
System.out.println("Max pool size is " + conf.getMaxSize());
```

### iNote

通常不需要合并 Stateless 服务对象。我们不认为它应该是默认选择，因为大多数 Stateless 对象自然是线程安全的，并且如果缓存了资源，实例池会成问题。

通过使用自动代理，可以简化池。您可以设置任何自动代理创建者使用的 `TargetSource` 实现。

## 6.9.3. 原型目标源

设置“原型”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。尽管在现代 JVM 中创建新对象的成本并不高，但是连接新对象(满足其 IoC 依赖关系)的成本可能会更高。因此，没有充分的理由就不应使用此方法。

为此，您可以按如下所示修改 `poolTargetSource` 定义(为清楚起见，我们也更改了名称)：

```
<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。`TargetSource` 实现中使用继承来确保命名的一致性。与池化目标源一样，目标 Bean 必须是原型 Bean 定义。

#### 6.9.4. ThreadLocal 目标源

如果您需要为每个传入请求(每个线程)创建一个对象，则 `ThreadLocal` 目标源很有用。

`ThreadLocal` 的概念提供了 JDK 范围的功能，可以透明地将资源与线程一起存储。设置

`ThreadLocalTargetSource` 与其他类型的目标源所说明的几乎相同，如以下示例所示：

```
<bean id="threadlocalTargetSource" class="org.springframework.aop.target.ThreadLocalTargetSource">
    <property name="targetBeanName" value="businessObjectTarget"/>
</bean>
```

##### ①Note

`ThreadLocal` 实例在多线程和多类加载器环境中使用不正确时会遇到严重问题(可能导致内存泄漏)。您应该始终考虑将 `threadlocal` 包装在其他一些类中，并且切勿直接使用

`ThreadLocal` 本身(包装类中除外)。另外，您应始终记住正确设置和取消设置线程本地资源

(在后者中仅涉及对 `ThreadLocal.set(null)` 的调用)。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出现问题。Spring 的 `ThreadLocal` 支持为您做到了这一点

，应该始终考虑使用 `ThreadLocal` 实例，而无需其他适当的处理代码。

### 6.10. 定义新的建议类型

Spring AOP 被设计为可扩展的。尽管目前在内部使用拦截实现策略，但是除了在建议周围，在建议之前，抛出建议和返回建议之后进行拦截之外，还可以支持任意建议类型。

`org.springframework.aop.framework.adapter` 软件包是 SPI 软件包，可在不更改核心框架的情况下添加对新的自定义建议类型的 support。自定义 `Advice` 类型的唯一限制是它必须实现 `org.aopalliance.aop.Advice` 标记接口。

有关更多信息，请参见[org.springframework.aop.framework.adapter](#) javadoc。

## 7. Null-safety

尽管 Java 不允许您使用其类型系统来表示空安全性，但 Spring Framework 现在在 `org.springframework.lang` 包中提供了以下 Comments，以使您可以声明 API 和字段的空性：

- [@NotNull](#): 表示特定参数，返回值或字段不能为 `null` 的 Comments(在使用 `@NotNullApi` 和 `@NotNullFields` 的参数和返回值上不需要)。
- [@Nullable](#): 表示特定参数，返回值或字段可以是 `null` 的 Comments。
- [@NotNullApi](#): 程序包级别的 Comments，它声明非 null 为参数和返回值的默认行为。
- [@NotNullFields](#): 程序包级别的 Comments，它声明非 null 为字段的默认行为。

Spring Framework 自身利用了这些 Comments，但是它们也可以在任何基于 Spring 的 Java 项目中使用，以声明 null 安全的 API 和可选的 null 安全的字段。尚不支持泛型类型参数，varargs 和数组元素的可空性，但应在即将发布的版本中使用它们，有关最新信息，请参见[SPR-15942](#)。可以在 Spring Framework 版本之间(包括次要版本)对可空性声明进行微调。方法主体内部使用的类型的可空性超出了此功能的范围。

### iNote

像 Reactor 或 Spring Data 这样的库提供使用此功能的空安全 API。

## 7.1. 用例

除了为 Spring Framework API 可空性提供显式声明之外，IDE(例如 IDEA 或 Eclipse)还可以使用这些 Comments 来提供与空安全性相关的有用警告，从而避免在运行时出现 `NullPointerException`。

由于 Kotlin 本机支持[null-safety](#)，因此它们还用于使 Kotlin 项目中的 Spring API 空安全。[Kotlin 支持文档](#)中提供了更多详细信息。

## 7.2. JSR 305 元 Comments

SpringComments 使用[JSR 305](#)Comments(休眠但分布广泛的 JSR)进行元 Comments。JSR 305 元 Comments 使工具供应商(如 IDEA 或 Kotlin)以通用方式提供了空安全性支持，而无需对 SpringComments 进行硬编码支持。

既不需要也不建议在项目 Classpath 中添加 JSR 305 依赖项以利用 Spring 空安全 API。只有诸如在其代码库中使用空安全 Comments 的基于 Spring 的库之类的项目才应添加具有 `compileOnly` Gradle 配置或 Maven `provided` 范围的 `com.google.code.findbugs:jsr305:3.0.2` 以避免编译警告。

## 8. 数据缓冲区和编解码器

Java NIO 提供了 `ByteBuffer`，但是许多库在顶部构建了自己的字节缓冲区 API，特别是对于网络操作，其中重用缓冲区和/或使用直接缓冲区对于性能有好处。例如，Netty 具有 `ByteBuf` 层次结构，Undertow 使用 XNIO，Jetty 使用具有要释放的回调的池字节缓冲区，依此类推。`spring-core` 模块提供了一组抽象，可与各种字节缓冲区 API 配合使用，如下所示：

- [DataBufferFactory](#) 抽象了数据缓冲区的创建。
- [DataBuffer](#) 代表字节缓冲区，可以是 [pooled](#)。
- [DataBufferUtils](#) 提供了用于数据缓冲区的 Util 方法。
- [Codecs](#) 将流数据缓冲区流解码或编码为更高级别的对象。

## 8.1. DataBufferFactory

`DataBufferFactory` 用于通过以下两种方式之一创建数据缓冲区：

- 分配一个新的数据缓冲区，可以选择预先指定容量(如果知道的话)，即使 `DataBuffer` 的实现可以按需增长和缩小，该容量也会更有效。
- 包装现有的 `byte[]` 或 `java.nio.ByteBuffer`，该\_或\_用 `DataBuffer` 实现装饰给定数据，并且不涉及分配。

请注意，`WebFlux` 应用程序不会直接创建 `DataBufferFactory`，而是通过 `Client` 端的 `ServerHttpResponse` 或 `ClientHttpRequest` 访问它。工厂的类型取决于基础 `Client` 端或服务器，例如 `NettyDataBufferFactory` 用于 Reactor Netty，`DefaultDataBufferFactory` 用于其他。

## 8.2. DataBuffer

`DataBuffer` 界面提供与 `java.nio.ByteBuffer` 类似操作，但还带来了一些其他好处，其中一些是受 Netty `ByteBuf` 启发的。以下是部分好处清单：

- 具有独立位置的读取和写入，即不需要调用 `flip()` 在读取和写入之间交替。
- 与 `java.lang.StringBuilder` 一样，容量可按需扩展。
- 通过 `PooledDataBuffer` 合并缓冲和引用计数。
- 查看缓冲区为 `java.nio.ByteBuffer`，`InputStream` 或 `OutputStream`。
- 确定给定字节的索引或最后一个索引。

## 8.3. PooledDataBuffer

如 Javadoc `ByteBuffer` 中所述，字节缓冲区可以是直接的也可以是非直接的。直接缓冲区可以驻留

在 Java 堆之外，从而无需复制本机 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，这导致了缓冲池的想法。

`PooledDataBuffer` 是 `DataBuffer` 的扩展，有助于进行引用计数，这对于字节缓冲区池至关重要。它是如何工作的？分配 `PooledDataBuffer` 时，参考计数为 1. 调用 `retain()` 会使计数递增，而调用 `release()` 会使计数递减。只要计数大于 0，就保证不会释放缓冲区。当计数减少到 0 时，可以释放池中的缓冲区，这实际上意味着将为缓冲区保留的内存返回到内存池。

请注意，与其直接在 `PooledDataBuffer` 上进行操作，不如在大多数情况下，最好使用 `DataBufferUtils` 中的便捷方法，仅当它是 `PooledDataBuffer` 的实例时，才对 `DataBuffer` 应用发布或保留。

## 8.4. DataBufferUtils

`DataBufferUtils` 提供了许多 Util 方法来对数据缓冲区进行操作：

- 将数据缓冲区流连接到单个缓冲区中，可能具有零个副本，例如通过复合缓冲区(如果底层字节缓冲区 API 支持)。
- 将 `InputStream` 或 NIO `Channel` 转换为 `Flux<DataBuffer>`，反之亦然将 `Publisher<DataBuffer>` 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。
- 从字节流中跳过或获取，直到特定的字节数为止。

## 8.5. Codecs

`org.springframework.core.codec` 软件包提供以下策略接口：

- `Encoder` 将 `Publisher<T>` 编码为数据缓冲区流。
- `Decoder` 将 `Publisher<DataBuffer>` 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` 模块添加了 Jackson JSON, Jackson Smile, JAXB2, 协议缓冲区以及其他编码器和解码器。请参阅 “WebFlux”部分中的[Codecs](#)。

## 8.6. 使用 DataBuffer

使用数据缓冲区时，必须特别小心以确保释放缓冲区，因为它们可能是[pooled](#)。我们将使用编解码器来说明其工作原理，但这些概念会更普遍地应用。让我们看看编解码器必须在内部执行哪些操作来 Management 数据缓冲区。

`Decoder` 是创建高级对象之前最后读取 Importing 数据缓冲区的方法，因此它必须按以下方式释放它们：

- 如果 `Decoder` 只是读取每个 Importing 缓冲区并准备立即释放它，则可以通过`DataBufferUtils.release(dataBuffer)` 这样做。
- 如果 `Decoder` 使用 `Flux` 或 `Mono` 运算符(例如 `flatMap`, `reduce`)以及内部预取和缓存数据项的其他运算符，或者正在使用 `filter`, `skip` 以及其他省略项的运算符，则必须将`doOnDiscard(PooledDataBuffer.class, DataBufferUtils::release)` 添加到组合中链以确保此类缓冲区在被丢弃之前被释放，也可能是由于错误或取消 `signal` 而导致的。
- 如果 `Decoder` 以任何其他方式保留在一个或多个数据缓冲区上，则它必须确保在完全读取时释放它们，或者在读取和释放缓存的数据缓冲区之前发生错误或取消 `signal` 的情况下。

请注意，`DataBufferUtils#join` 提供了一种安全有效的方法来将数据缓冲区流聚合到单个数据缓冲区中。同样，`skipUntilByteCount` 和 `takeUntilByteCount` 是供解码器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。因此，`Encoder` 没什么事要做。但是，如果在向缓冲区填充数据时发生序列化错误，则 `Encoder` 必须小心释放数据缓冲区。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` 的使用者负责释放其接收的数据缓冲区。在 WebFlux 应用程序中，`Encoder` 的输出用于写入 HTTP 服务器响应或 Client 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 Client 端的责任。请求。

请注意，在 Netty 上运行时，[解决缓冲区泄漏问题](#)有调试选项。

## 9. Appendix

### 9.1. XML 模式

附录的此部分列出了与核心容器相关的 XML 模式。

#### 9.1.1. 实用模式

顾名思义，`util` 标签处理常见的 Util 配置问题，例如配置集合，引用常量等。要在 `util` 模式中使用标签，您需要在 Spring XML 配置文件的顶部具有以下序言(代码段中的文本引用了正确的模式，以便您可以使用 `util` 名称空间中的标签)：

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

## Using <util:constant/>

考虑以下 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>
```

前面的配置使用 Spring `FactoryBean` 实现(`FieldRetrievingFactoryBean`)将 Bean 上 `isolation` 属性的值设置为 `java.sql.Connection.TRANSACTION_SERIALIZABLE` 常量的值。这一切都很好，但是很冗长，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 的版本更加简洁，清楚地表达了开发人员的意图(“注入此常量值”), 并且读起来更好：

```
<bean id="..." class="...">
    <property name="isolation">
        <util:constant static-field="java.sql.Connection.TRANSACTION_SERIALIZABLE"/>
    </property>
</bean>
```

## 根据字段值设置 Bean 属性或构造函数参数

[FieldRetrievingFactoryBean](#) 是 `FactoryBean`，它检索 `static` 或非静态字段值。它通常用于检索 `public` `static` `final` 常量，然后可用于为另一个 bean 设置属性值或构造函数参数。

下面的示例显示如何通过使用 `staticField` 属性来显示 `static` 字段：

```
<bean id="myField"
      class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean">
    <property name="staticField" value="java.sql.Connection.TRANSACTION_SERIALIZABLE"/>
</bean>
```

还有一个便利用法表格，其中将 `static` 字段指定为 bean 名称，如以下示例所示：

```
<bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE"
```

```
class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean" />
```

这的确意味着 Bean `id` 不再是任何选择(因此，引用它的其他任何 Bean 也必须使用这个较长的名称)，但是这种形式的定义非常简洁，可以很方便地用作内部对象。 `bean`，因为不必为 `bean` 引用指定 `id`，如以下示例所示：

```
<bean id="..." class="...">
    <property name="isolation">
        <bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE"
              class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean" />
    </property>
</bean>
```

您还可以访问另一个 `bean` 的非静态(实例)字段，如[FieldRetrievingFactoryBean](#)类的 API 文档中所述。

在 Spring 中，很容易将枚举值作为属性或构造函数参数注入到 `bean` 中。实际上，您不必做任何事情或不了解 Spring 内部知识(甚至不必了解诸如 `FieldRetrievingFactoryBean` 之类的类)。以下示例枚举显示了注入枚举值的难易程度：

```
package javax.persistence;

public enum PersistenceContextType {
    TRANSACTION,
    EXTENDED
}
```

现在考虑以下类型为 `PersistenceContextType` 的 `setter` 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>
```

## Using <util:property-path/>

考虑以下示例：

```
<!-- 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' -->
<bean id="testBean.age" class="org.springframework.beans.factory.config.PropertyPathFac...
```

前面的配置使用 Spring `FactoryBean` 实现(`PropertyPathFactoryBean`)创建一个名为 `testBean.age` 的 Bean(类型 `int`)，该 Bean 的值等于 `testBean` bean 的 `age` 属性。

现在考虑以下示例，该示例添加了一个 `<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，它将获取名为 `testBean` 的 bean 的 `age` 属性。该 `age` 属性的值为 `10`。

## 使用\<>设置 Bean 属性或构造函数参数

`PropertyPathFactoryBean` 是 `FactoryBean`，它评估给定目标对象上的属性路径。可以直接指

定目标对象，也可以通过 `bean` 名称指定目标对象。然后，您可以在另一个 `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` 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>
```

在以下示例中，针对内部 `bean` 评估路径：

```
<!-- 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>
```

还有一种快捷方式，其中 Bean 名称是属性路径。以下示例显示了快捷方式表格：

```
<!-- results in 10, which is the value of property 'age' of bean 'person' -->
<bean id="person.age"
      class="org.springframework.beans.factory.config.PropertyPathFactoryBean" />
```

这种形式的确意味着在 Bean 名称中别无选择。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，如以下示例所示：

```
<bean id="..." class="...">
    <property name="age">
        <bean id="person.age"
```

```
        class="org.springframework.beans.factory.config.PropertyPathFactoryBean">
    </property>
</bean>
```

您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，但有时可能会有用。有关此功能的更多信息，请参见 [javadoc](#)。

## Using <util:properties/>

考虑以下示例：

```
<!-- 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>
```

前面的配置使用 Spring [FactoryBean](#) 实现([PropertiesFactoryBean](#))来实例化具有从提供的 [Resource](#) 位置加载的值的 [java.util.Properties](#) 实例)。

以下示例使用 [util:properties](#) 元素进行更简洁的表示：

```
<!-- creates a java.util.Properties instance with values loaded from the supplied location -->
<util:properties id="jdbcConfiguration" location="classpath:com/foo/jdbc-production.properties"/>
```

## Using <util:list/>

考虑以下示例：

```
<!-- 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>[emailprotected]</value>
            <value>[emailprotected]</value>
            <value>[emailprotected]</value>
            <value>[emailprotected]</value>
        </list>
    </property>
</bean>
```

前面的配置使用 Spring [FactoryBean](#) 实现([ListFactoryBean](#))创建 [java.util.List](#) 实例

, 并使用从提供的 `sourceList` 中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">
    <value>[emailprotected]</value>
    <value>[emailprotected]</value>
    <value>[emailprotected]</value>
    <value>[emailprotected]</value>
</util:list>
```

您还可以使用 `<util:list/>` 元素上的 `list-class` 属性来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`, 则可以使用以下配置:

```
<util:list id="emails" list-class="java.util.LinkedList">
    <value>[emailprotected]</value>
    <value>[emailprotected]</value>
    <value>[emailprotected]</value>
    <value>d'[emailprotected]</value>
</util:list>
```

如果没有提供 `list-class` 属性, 则容器选择 `List` 实现。

## Using `<util:map/>`

考虑以下示例:

```
<!-- 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="[emailprotected]" />
            <entry key="raskolnikov" value="[emailprotected]" />
            <entry key="stavrogin" value="[emailprotected]" />
            <entry key="porfiry" value="[emailprotected]" />
        </map>
    </property>
</bean>
```

前面的配置使用 Spring `FactoryBean` 实现(`MapFactoryBean`)来创建 `java.util.Map` 实例, 该实例使用从提供的 '`sourceMap`' 中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="[emailprotected]"/>
    <entry key="raskolnikov" value="[emailprotected]"/>
    <entry key="stavrogin" value="[emailprotected]"/>
    <entry key="porfiry" value="[emailprotected]"/>
</util:map>
```

您还可以使用 `<util:map/>` 元素上的 `'map-class'` 属性来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`，则可以使用以下配置：

```
<util:map id="emails" map-class="java.util.TreeMap">
    <entry key="pechorin" value="[emailprotected]"/>
    <entry key="raskolnikov" value="[emailprotected]"/>
    <entry key="stavrogin" value="[emailprotected]"/>
    <entry key="porfiry" value="[emailprotected]"/>
</util:map>
```

如果没有提供 `'map-class'` 属性，则容器选择 `Map` 实现。

## Using `<util:set/>`

考虑以下示例：

```
<!-- 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>[emailprotected]</value>
            <value>[emailprotected]</value>
            <value>[emailprotected]</value>
            <value>[emailprotected]</value>
        </set>
    </property>
</bean>
```

前面的配置使用 Spring `FactoryBean` 实现(`SetFactoryBean`)来创建 `java.util.Set` 实例，该实例使用从提供的 `sourceSet` 中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>[emailprotected]</value>
    <value>[emailprotected]</value>
    <value>[emailprotected]</value>
    <value>[emailprotected]</value>
</util:set>
```

您还可以使用 `<util:set/>` 元素上的 `set-class` 属性来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`，则可以使用以下配置：

```
<util:set id="emails" set-class="java.util.TreeSet">
    <value>[emailprotected]</value>
    <value>[emailprotected]</value>
    <value>[emailprotected]</value>
    <value>[emailprotected]</value>
</util:set>
```

如果没有提供 `set-class` 属性，则容器选择 `Set` 实现。

### 9.1.2. aop 模式

`aop` 标签处理在 Spring 中配置 AOP 的所有事情，包括 Spring 自己的基于代理的 AOP 框架以及 Spring 与 AspectJ AOP 框架的集成。这些标签在标题为 [Spring 面向方面的编程](#) 的章节中全面介绍。

为了完整起见，要在 `aop` 模式中使用标签，您需要在 Spring XML 配置文件的顶部具有以下前导（代码段中的文本引用了正确的模式，以便 `aop` 名称空间中的标签为提供给您）：

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

    <!-- bean definitions here -->
</beans>
```

### 9.1.3. 上下文模式

`context` 标签处理与管道相关的 `ApplicationContext` 配置，即通常不是对最终用户重要的 bean，而是在 Spring 中完成大量“艰巨”工作的 bean，例如 `BeanfactoryPostProcessors`。以下代码段引用了正确的架构，以便您可以使用 `context` 名称空间中的元素：

```
<?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 http://www.springframework.org/schema/beans.xsd
        http://www.springframework.org/schema/context http://www.springframework.org/schema/context.xsd">
    <!-- bean definitions here -->
</beans>
```

## Using `<property-placeholder/>`

此元素激活 `${...}` 占位符的替换，这些占位符针对指定的属性文件(作为 Spring 资源位置)解析。此元素是为您设置 `PropertyPlaceholderConfigurer` 的便捷机制。如果您需要对 `PropertyPlaceholderConfigurer` 的更多控制，则可以自己明确定义。

## Using `<annotation-config/>`

此元素激活 Spring 基础结构以检测 Bean 类中的 Comments：

- Spring 的 `@Required` 和 `@Autowired`
- JSR 250 的 `@PostConstruct`，`@PreDestroy` 和 `@Resource` (如果有)
- JPA 的 `@PersistenceContext` 和 `@PersistenceUnit` (如果有)。

或者，您可以选择为这些 Comments 显式激活单独的 `BeanPostProcessors`。

### iNote

该元素不会激活对 Spring 的 `@TransactionalComments` 的处理。您可以为此使用 `<tx:annotation-driven/>` 元素。

## Using <component-scan/>

[基于 Comments 的容器配置](#) 中对此元素进行了详细说明。

## Using <load-time-weaver/>

[在 Spring Framework 中使用 AspectJ 进行加载时编织](#) 中对此元素进行了详细说明。

## Using <spring-configured/>

[使用 AspectJ 通过 Spring 依赖注入域对象](#) 中对此元素进行了详细说明。

## Using <mbean-export/>

[配置基于 Comments 的 MBean 导出](#) 中对此元素进行了详细说明。

### 9.1.4. Bean 模式

最后但并非最不重要的一点是，我们在 `beans` 模式中具有元素。自框架诞生之初，这些元素就一直出现在 Spring。此处未显示 `beans` 模式中各种元素的示例，因为它们在[依赖关系和配置详细](#)中(并且实际上在整个[chapter](#)中)已进行了全面介绍。

请注意，您可以将零个或多个键值对添加到 `<bean/>` XML 定义中。使用此额外的元数据进行的操作(如果有的话)完全取决于您自己的自定义逻辑(因此，通常只有在您按照标题为[XML 模式创作](#)的附录中所述编写自己的自定义元素时才能使用)。

以下示例在周围的 `<bean/>` 上下文中显示了 `<meta/>` 元素(请注意，由于没有任何逻辑来解释它，因此元数据实际上是毫无用处的)。

```
<?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 http://www.springframework.org/sche
<bean id="foo" class="x.y.Foo">
    <meta key="cacheName" value="foo"/> (1)
    <property name="name" value="Rick"/>
</bean>
```

```
</beans>
```

- (1) 这是示例 `meta` 元素

在前面的示例中，您可以假设存在一些逻辑，这些逻辑消耗了 `bean` 的定义，并构建了一些使用提供的元数据的缓存基础结构。

## 9.2. XML 模式创作

从 2.0 版开始，Spring 提供了一种机制，可以将基于架构的扩展添加到基本 Spring XML 格式中，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，以及如何将此类解析器集成到 Spring IoC 容器中。

为了方便使用架构感知的 XML 编辑器编写配置文件，Spring 的可扩展 XML 配置机制基于 XML Schema。如果您不熟悉标准 Spring 发行版随附的 Spring 当前的 XML 配置扩展，则应首先阅读名为 [\[xsd-config\]](#) 的附录。

要创建新的 XML 配置扩展，请执行以下操作：

- [Author](#) XML 模式，用于描述您的自定义元素。
- [Code](#) 自定义 `NamespaceHandler` 实现。
- [Code](#) 一个或多个 `BeanDefinitionParser` 实现(这是完成实际工作的地方)。
- [Register](#) 使用 Spring 的新工作。

对于一个统一的示例，我们创建一个 XML 扩展(一个自定义 XML 元素)，该扩展使我们可以配置 `SimpleDateFormat` 类型的对象(来自 `java.text` 包)。完成后，我们将能够如下定义

`SimpleDateFormat` 类型的 bean 定义：

```
<myns:dateFormat id="dateFormat"
    pattern="yyyy-MM-dd HH:mm"
    lenient="true" />
```

(我们将在本附录后面提供更多详细的示例。第一个简单示例的目的是引导您完成制作自定义扩展程序的基本步骤。)

## 9.2.1. 编写架构

创建用于 Spring 的 IoC 容器的 XML 配置扩展首先要编写 XML Schema 来描述扩展。对于我们的示例，我们使用以下架构来配置 `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"> (1)
          <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>
```

- (1) 所指示的行包含所有可识别标签的扩展基础(这意味着它们具有 `id` 属性，我们可以将其用作容器中的 bean 标识符)。我们可以使用此属性，因为我们导入了 Spring 提供的 `beans` 名称空间。

前面的架构使我们可以使用 `<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：

```
<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，上述 XML 片段与以下 XML 片段基本相同：

```
<bean id="dateFormat" class="java.text.SimpleDateFormat">
    <constructor-arg value="yyyy-HH-dd HH:mm"/>
    <property name="lenient" value="true"/>
</bean>
```

前面两个片段中的第二个片段在容器中创建了一个 bean(以类型 `SimpleDateFormat` 的名称 `dateFormat` 标识), 并设置了两个属性。

### iNote

创建配置格式的基于模式的方法允许与具有模式识别 XML 编辑器的 IDE 紧密集成。通过使用正确编写的架构, 可以使用自动完成功能来让用户在枚举中定义的多个配置选项之间进行选择。

## 9.2.2. 编码 NamespaceHandler

除了模式, 我们还需要一个 `NamespaceHandler` 来解析 Spring 在解析配置文件时遇到的该特定名称空间的所有元素。对于此示例, `NamespaceHandler` 应该负责 `myns:dateformat` 元素的解析。

`NamespaceHandler` 界面具有三种方法:

- `init()` : 允许 `NamespaceHandler` 初始化, 并且在使用处理器之前由 Spring 调用。
- `BeanDefinition parse(Element, ParserContext)` : 当 Spring 遇到顶级元素(未嵌套在 bean 定义或其他命名空间中)时调用。此方法本身可以注册 Bean 定义, 返回 Bean 定义或两者。
- `BeanDefinitionHolder decorate(Node, BeanDefinitionHolder, ParserContext)` : 当 Spring 遇到另一个名称空间的属性或嵌套元素时调用。例如, 一个或多个 bean 定义的修饰与 [Spring 支持的范围](#)一起使用。我们首先突出显示一个简单的示例, 而不使用装饰, 然后在一个更高级的示例中显示装饰。

尽管您可以为整个名称空间编写自己的 `NamespaceHandler` (并因此提供解析名称空间中每个元素的代码)，但是通常情况下，Spring XML 配置文件中的每个顶级 XML 元素都产生一个 bean 定义(例如在我们的示例中，单个 `<myns:datatype />` 元素导致单个 `SimpleDateFormat` bean 定义)。Spring 提供了许多支持这种情况的便利类。在下面的示例中，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("datatype", new SimpleDateFormatBeanDefinitionParser());
    }

}
```

您可能会注意到，此类中实际上没有很多解析逻辑。的确，`NamespaceHandlerSupport` 类具有内置的委托概念。它支持注册任意数量的 `BeanDefinitionParser` 实例，当需要解析其命名空间中的元素时，可以委托该实例注册。这种清晰的关注点分离使 `NamespaceHandler` 处理其命名空间中所有自定义元素的解析编排，同时委派 `BeanDefinitionParsers` 来完成 XML 解析的繁琐工作。这意味着每个 `BeanDefinitionParser` 仅包含解析单个自定义元素的逻辑，正如我们在下一步中看到的那样。

### 9.2.3. 使用 BeanDefinitionParser

如果 `NamespaceHandler` 遇到 Map 到特定 bean 定义解析器(在这种情况下为 `datatype`)的 XML 元素，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(因此也可以访问其子元素)，以便我们可以解析自定义 XML 内容，如以下示例所示：

```
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; (2)
    }

    protected void doParse(Element element, BeanDefinitionBuilder bean) {
        // this will never be null since the schema explicitly requires that a value be
        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));
        }
    }
}

```

- (1) 我们使用 Spring 提供的 `AbstractSingleBeanDefinitionParser` 来处理创建单个 `BeanDefinition` 的许多基本工作。
- (2) 我们为 `AbstractSingleBeanDefinitionParser` 超类提供了我们的单个 `BeanDefinition` 表示的类型。

在这种简单的情况下，这就是我们要做的全部。 `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. 注册处理器和架构

编码完成。剩下要做的就是让 Spring XML 解析基础结构了解我们的自定义元素。为此，我们在两个特殊用途的属性文件中注册了自定义 `namespaceHandler` 和自定义 XSD 文件。这些属性文件都放置在应用程序的 `META-INF` 目录中，例如，可以与二进制类一起分发到 JAR 文件中。Spring XML 解析基础结构通过使用这些特殊的属性文件来自动选择您的新扩展，以下两部分将详细介绍其格式。

## Writing META-INF/spring.handlers

名为 `spring.handlers` 的属性文件包含 XML 模式 URI 到名称空间处理器类的 Map。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.MyNamespaceHandle
```

(`:` 字符是 Java 属性格式的有效分隔符，因此 URI 中的 `:` 字符需要用反斜杠转义。)

键值对的第一部分(键)是与您的自定义名称空间扩展关联的 URI，并且需要与您的自定义 XSD 架构中指定的 `targetNamespace` 属性值完全匹配。

## Writing 'META-INF/spring.schemas'

名为 `spring.schemas` 的属性文件包含 XML 架构位置(与架构声明一起引用，在使用该架构作为 `xsi:schemaLocation` 属性的一部分的 XML 文件中)到 Classpath 资源。需要该文件来防止 Spring 绝对使用默认的 `EntityResolver`，该默认 `EntityResolver` 需要 Internet 访问才能检索架构文件。如果您在此属性文件中指定 Map，Spring 将在 Classpath 上搜索架构(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，必须对 `:` 字符进行转义。)

鼓励您在 Classpath 上的 `NamespaceHandler` 和 `BeanDefinitionParser` 类旁边部署 XSD 文件。

### 9.2.5. 在 Spring XML 配置中使用自定义扩展

使用您自己实现的定制扩展与使用 Spring 提供的“定制”扩展之一没有什么不同。以下示例在 Spring XML 配置文件中使用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 http://www.springframework.org/schema/beans.xsd
        http://www.mycompany.com/schema/myns http://www.mycompany.com/schema/myns/myns.xsd">

    <!-- as a top-level bean -->
    <myns:dateFormat id="defaultDateFormat" pattern="yyyy-MM-dd HH:mm" lenient="true"/>

    <bean id="jobDetailTemplate" abstract="true">
        <property name="dateFormat">
            <!-- as an inner bean -->
            <myns:dateFormat pattern="HH:mm MM-dd-yyyy" />
        </property>
    </bean>

</beans>

```

- (1) 我们的自定义 bean。

## 9.2.6. 更详细的例子

本节提供一些更详细的自定义 XML 扩展示例。

### 在自定义元素中嵌套自定义元素

本节中的示例显示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 http://www.springframework.org/schema/beans.xsd
        http://www.foo.com/schema/component http://www.foo.com/schema/component/component.xsd">

    <foo:component id="bionic-family" name="Bionic-1">
        <foo:component name="Mother-1">
            <foo:component name="Karate-1" />
            <foo:component name="Sport-1" />
        </foo:component>
        <foo:component name="Rock-1" />
    </foo:component>

</beans>

```

前面的配置将自定义扩展相互嵌套。 `<foo:component />` 元素实际配置的类是 `Component` 类(在下一个示例中显示)。注意 `Component` 类如何不公开 `components` 属性的 `setter` 方法。这使得很

难(或几乎不可能)通过使用 `setter` 注入为 `Component` 类配置 bean 定义。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;
    }

}
```

解决此问题的典型方法是创建一个自定义 `FactoryBean`，该自定义 `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);
        }
    }
    return this.parent;
}

public Class<Component> getObjectType() {
    return Component.class;
}

public boolean isSingleton() {
    return true;
}

}

```

这很好用，但是向最终用户暴露了很多 Spring 管道。我们要做的是编写一个自定义 extensions，以隐藏所有此 Spring 管道。如果我们坚持使用[前面描述的步骤](#)，那么我们首先创建 XSD 模式以定义自定义标记的结构，如以下清单所示：

```

<?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>

```

在[前面描述的过程](#)之后，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 `BeanDefinition` 来描述 `ComponentFactoryBean`。以下清单显示了我们的自定义 `BeanDefinitionParser`：

```
package com.foo;

import org.springframework.beans.factory.config.BeanDefinition;
import org.springframework.beans.factory.support.AbstractBeanDefinition;
import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.support.ManagedList;
import org.springframework.beans.factory.xml.AbstractBeanDefinitionParser;
import org.springframework.beans.factory.xml.ParserContext;
import org.springframework.util.xml.DomUtils;
import org.w3c.dom.Element;

import java.util.List;

public class ComponentBeanDefinitionParser extends AbstractBeanDefinitionParser {

    protected AbstractBeanDefinition parseInternal(Element element, ParserContext parser)
        return parseComponentElement(element);
    }

    private static AbstractBeanDefinition parseComponentElement(Element element) {
        BeanDefinitionBuilder factory = BeanDefinitionBuilder.rootBeanDefinition(Compon
        factory.addPropertyValue("parent", parseComponent(element));

        List<Element> childElements = DomUtils.getChildElementsByTagName(element, "comp
        if (childElements != null && childElements.size() > 0) {
            parseChildComponents(childElements, factory);
        }

        return factory.getBeanDefinition();
    }

    private static BeanDefinition parseComponent(Element element) {
        BeanDefinitionBuilder component = BeanDefinitionBuilder.rootBeanDefinition(Comp
        component.addPropertyValue("name", element.getAttribute("name")));
        return component.getBeanDefinition();
    }

    private static void parseChildComponents(List<Element> childElements, BeanDefinition
        ManagedList<BeanDefinition> children = new ManagedList<BeanDefinition>(childEle
        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，将各种工件注册到 Spring XML 基础结构中，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
```

## “常规”元素上的自定义属性

编写自己的自定义解析器和关联的工件并不难。但是，有时这不是正确的选择。考虑一个需要将元数据添加到已经存在的 `bean` 定义的场景。在这种情况下，您当然不需要编写自己的整个自定义扩展。相反，您只想向现有的 `bean` 定义元素添加一个附加属性。

作为另一个示例，假设您为访问集群[JCache](#)的服务对象(它不知道)定义了一个 `bean` 定义，并且您想确保在周围的集群中急切启动命名的 `JCache` 实例。以下清单显示了这样的定义：

```
<bean id="checkingAccountService" class="com.foo.DefaultCheckingAccountService"  
      jcache:cache-name="checking.account">  
    <!-- other dependencies here... -->  
</bean>
```

然后，当解析 `'jcache:cache-name'` 属性时，我们可以创建另一个 `BeanDefinition`。然后，此 `BeanDefinition` 为我们初始化命名的 `JCache`。我们还可以为

`'checkingAccountService'` 修改现有的 `BeanDefinition`，以便它依赖于此新的 `JCache` 初始化 `BeanDefinition`。以下清单显示了我们的 `JCacheInitializer`：

```
package com.foo;  
  
public class JCacheInitializer {  
  
    private String name;  
  
    public JCacheInitializer(String name) {  
        this.name = name;
```

```
}

public void initialize() {
    // lots of JCache API calls to initialize the named cache...
}

}
```

现在我们可以进入自定义扩展了。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`：

```
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 BeanDefinitionDecorat

    private static final String[] EMPTY_STRING_ARRAY = new String[0];

    public BeanDefinitionHolder decorate(Node source, BeanDefinitionHolder holder,
        ParserContext ctx) {
        String initializerBeanName = registerJCacheInitializer(source, ctx);
        createDependencyOnJCacheInitializer(holder, initializerBeanName);
        return holder;
    }

    private void createDependencyOnJCacheInitializer(BeanDefinitionHolder holder,
        String initializerBeanName) {
        AbstractBeanDefinition definition = ((AbstractBeanDefinition) holder.getBeanDef
        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(
                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，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/jcache=com.foo.JCacheNamespaceHandler
```

```
# in 'META-INF/spring.schemas'
http://www.foo.com/schema/jcache/jcache.xsd=com/foo/jcache.xsd
```

# Testing

---

本章涵盖 Spring 对集成测试的支持以及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)确实确实使单元测试和集成测试更加容易(因为在类上存在 `setter` 方法和适当的构造函数,使得它们在测试中更容易连接在一起,而不必设置服务定位器注册表和类似结构)。

## 1. Spring 测试简介

---

测试是企业软件开发的组成部分。本章重点介绍 IoC 原则为[unit testing](#)带来的增值,以及 Spring 框架对[integration testing](#)的支持所带来的好处。(对企业中测试的彻底处理不在本参考手册的范围之内。)

## 2. 单元测试

---

与传统的 Java EE 开发相比,依赖注入应该使您的代码对容器的依赖程度降低。组成应用程序的 POJO 应该可以在 JUnit 或 TestNG 测试中进行测试,并使用 `new` 运算符实例化对象,而无需使用 Spring 或任何其他容器。您可以使用[mock objects](#)(结合其他有价值的测试技术)来单独测试代码。如果您遵循 Spring 的体系结构建议,那么代码库的最终分层和组件化将使单元测试更加容易。例如,您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性数据。

true 的单元测试通常运行非常快,因为没有可设置的运行时基础架构。将 true 的单元测试作为开发方法的一部分可以提高生产率。您可能不需要测试章节的这一部分来帮助您为基于 IoC 的应用程序编写有效的单元测试。但是,对于某些单元测试方案, Spring 框架提供了模拟对象和测试支持类,本章对此进行了介绍。

### 2.1. 模拟对象

Spring 包含许多专用于模拟的软件包:

- [Environment](#)
- [JNDI](#)
- [Servlet API](#)
- [Spring WebReactive](#)

### 2.1.1. Environment

`org.springframework.mock.env` 软件包包含 `Environment` 和 `PropertySource` 抽象的模拟实现(请参见[Bean 定义配置文件](#)和[PropertySource Abstraction](#))。 `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很有用。

### 2.1.2. JNDI

`org.springframework.mock.jndi` 软件包包含 JNDI SPI 的实现, 可用于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` 实例在测试代码中与在 Java EE 容器中绑定到相同的 JNDI 名称, 则可以在测试场景中重用应用程序代码和配置, 而无需进行修改。

### 2.1.3. Servlet API

`org.springframework.mock.web` 软件包包含一组全面的 Servlet API 模拟对象, 可用于测试 Web 上下文, 控制器和过滤器。这些模拟对象针对 Spring 的 Web MVC 框架使用, 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。

#### Tip

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 测试框架构建在模拟 Servlet API 对象的基础上, 为 Spring MVC 提供了集成测试框架。  
。参见[Spring MVC 测试框架](#)。

## 2.1.4. Spring WebReactive

`org.springframework.mock.http.server.reactive` 程序包包含用于 WebFlux 应用程序的 `ServerHttpRequest` 和 `ServerHttpResponse` 的模拟实现。

`org.springframework.mock.web.server` 软件包包含一个模拟 `ServerWebExchange`，该模拟 `ServerWebExchange` 取决于那些模拟请求和响应对象。

`MockServerHttpRequest` 和 `MockServerHttpResponse` 都从与特定于服务器的实现相同的抽象 Base Class 扩展，并与它们共享行为。例如，模拟请求一旦创建便是不可变的，但是您可以使用 `ServerHttpRequest` 中的 `mutate()` 方法来创建修改后的实例。

为了使模拟响应正确实现写约定并返回写完成句柄(即 `Mono<Void>`)，默认情况下，它使用 `Flux` 和 `cache().then()` 来缓冲数据并将其用于测试中的 `assert`。应用程序可以设置自定义写入功能(例如，测试无限流)。

[WebTestClient](#) 构建在模拟请求和响应的基础上，以提供对不使用 HTTP 服务器的 WebFlux 应用程序测试的支持。Client 端还可以用于正在运行的服务器的端到端测试。

## 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：

- [通用测试工具](#)
- [Spring MVC 测试 Util](#)

### 2.2.1. 通用测试工具

`org.springframework.test.util` 软件包包含几个通用 Util，可用于单元和集成测试。

`ReflectionTestUtils` 是基于反射的 Util 方法的集合。您可以在测试方案中使用这些方法，在这些情况下，当测试应用程序代码是否需要更改常量的值，设置非 `public` 字段，调用非 `public`

setter 方法或调用非 `public` 配置或生命周期回调方法时，用例如下：

- ORM 框架(例如 JPA 和 Hibernate)宽容 `private` 或 `protected` 字段访问，而不是针对域实体中的属性的 `public` setter 方法。
- Spring 对 Comments(例如 `@Autowired`，`@Inject` 和 `@Resource`)的支持，这些 Comments 为 `private` 或 `protected` 字段，setter 方法和配置方法提供依赖项注入。
- 将 `@PostConstruct` 和 `@PreDestroy` 之类的 Comments 用于生命周期回调方法。

[AopTestUtils](#)是与 AOP 相关的 Util 方法的集合。您可以使用这些方法来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，如果您已通过使用 EasyMock 或 Mockito 之类的库将 bean 配置为动态模拟，并且该模拟包装在 Spring 代理中，则可能需要直接访问基础模拟以配置对它的期望并执行验证。有关 Spring 的核心 AOPUtil，请参见[AopUtils](#)和[AopProxyUtils](#)。

## 2.2.2. Spring MVC 测试 Util

`org.springframework.test.web` 软件包包含[ModelAndViewAssert](#)，您可以将它们与 JUnit，TestNG 或任何其他测试框架结合使用，以进行处理 Spring MVC  `ModelAndView` 对象的单元测试。

### Unit testing Spring MVC Controllers

要对作为 POJO 的 Spring MVC `Controller` 类进行单元测试，请使用

`ModelAndViewAssert` 结合 `MockHttpServletRequest`，`MockHttpSession` 等，以此类推。为了对 Spring MVC 和 REST `Controller` 类以及 Spring MVC 的 `WebApplicationContext` 配置进行全面的集成测试，请改用[Spring MVC 测试框架](#)。

## 3.集成测试

本节(本章其余部分)涵盖了 Spring 应用程序的集成测试。它包括以下主题：

- [Overview](#)
- [集成测试的目标](#)
- [JDBC 测试支持](#)
- [Annotations](#)
- [Spring TestContext 框架](#)
- [Spring MVC 测试框架](#)
- [PetClinic Example](#)

## 3.1. Overview

能够执行一些集成测试而无需部署到应用程序服务器或连接到其他企业基础结构，这一点很重要。这样做可以测试以下内容：

- Spring IoC 容器上下文的正确接线。
- 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 Map 之类的东西。

Spring 框架为 `spring-test` 模块中的集成测试提供了一流的支持。实际的 JAR 文件的名称可能包括发行版，也可能采用长 `org.springframework.test` 格式，具体取决于从何处获取(说明请参见 [依赖 Management 部分](#))。该库包含 `org.springframework.test` 软件包，该软件包包含用于与 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。此类测试的运行速度比单元测试慢，但比依赖于部署到应用程序服务器的等效 Selenium 测试或远程测试快得多。

在 Spring 2.5 和更高版本中，以 Comments 驱动的[Spring TestContext 框架](#)形式提供了单元和集成测试支持。TestContext 框架与实际使用的测试框架无关，该框架允许在各种环境(包括 JUnit，TestNG 和其他环境)中对测试进行检测。

## 3.2. 集成测试的目标

Spring 的集成测试支持具有以下主要目标：

- 在两次测试之间 Management Spring IoC 容器缓存。
- 提供测试夹具实例的依赖注入。
- 提供适合集成测试的 transaction management。
- 提供 Spring 特定的 Base Class 来帮助开发人员编写集成测试。

接下来的几节描述了每个目标，并提供了有关实现和配置详细信息的链接。

### 3.2.1. 上下文 Management 和缓存

Spring TestContext Framework 提供了 Spring `ApplicationContext` 实例和 `WebApplicationContext` 实例的一致加载以及这些上下文的缓存。支持加载上下文的缓存很重要，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才能实例化。例如，具有 50 到 100 个 HibernateMap 文件的项目可能需要 10 到 20 秒钟来加载 Map 文件，并且在每个测试夹具中运行每个测试之前要承担该费用，这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(通常是在 Classpath 中)或用于配置应用程序的带 `Comments` 类的数组。这些位置或类与 `web.xml` 或其他用于生产部署的配置文件中指定的位置或类相同或相似。

默认情况下，配置的 `ApplicationContext` 加载后将重新用于每个测试。因此，每个测试套件仅产生一次安装成本，并且随后的测试执行要快得多。在这种情况下，术语“测试套件”是指所有测试都在同一 JVM 中运行，例如，所有测试都从给定项目或模块的 Ant, Maven 或 Gradle 构建运行。在不太可能的情况下，测试破坏了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` 框架配置为重新加载配置并重建应用程序上下文，然后再执行下一个测试。

参见[Context Management](#)和[Context Caching](#)与 TestContext 框架。

### 3.2.2. 测试夹具的依赖注入

当 TestContext 框架加载您的应用程序上下文时，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，可以通过在应用程序上下文中使用预配置的 bean 来设置测试装置。此处的一个强大好处是您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 SpringManagement 的对象图，事务代理， `DataSource` 实例等)，从而避免了为单个测试复制复杂的测试夹具设置的需要案件。

例如，考虑一个场景，其中我们有一个类(`HibernateTitleRepository`)，该类实现 `Title` 域实体的数据访问逻辑。我们要编写集成测试来测试以下方面：

- Spring 配置：基本上，与 `HibernateTitleRepository` bean 的配置有关的所有内容正确无误吗？
- HibernateMap 文件配置：是否正确 Map 了所有内容，并且是否有正确的延迟加载设置？
- `HibernateTitleRepository` 的逻辑：此类的配置实例是否按预期执行？

请参阅[TestContext framework](#)依赖测试夹具的依赖项注入。

### 3.2.3. TransactionManagement

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状态的影响。即使使用开发数据库，对状态的更改也可能会影响以后的测试。此外，无法在事务外部执行(或验证)许多操作(例如，插入或修改持久数据)。

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。您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，则对象将根据其配置的事务语义正确运行。此外，如果测试方法在为测试 Management 的事务中运行时删除选定表的内容，则该事务将默认回滚，并且数据库将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，为测试提供了事务支持。

如果您要提交事务(这不常见，但在希望特定测试填充或修改数据库时偶尔有用)，则可以通过使用 [@CommitComments](#) 告诉 `TestContext` 框架来提交事务，而不是回滚。

请参阅使用 [TestContext framework](#) 进行 TransactionManagement。

### 3.2.4. 集成测试支持类

Spring `TestContext Framework` 提供了几个 `abstract` 支持类，这些类简化了集成测试的编写。

这些基础测试类为测试框架提供了定义明确的钩子，还提供了方便的实例变量和方法，使您可以访问：

- `ApplicationContext`，用于执行显式的 `bean` 查找或测试整个上下文的状态。
- `JdbcTemplate`，用于执行 SQL 语句来查询数据库。您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，并且 Spring 确保此类查询在与应用程序代码相同的事務范围内运行。与 ORM 工具结合使用时，请务必避免使用 [false positives](#)。

此外，您可能希望使用针对项目的实例变量和方法来创建自己的自定义，应用程序级超类。

请参阅 [TestContext framework](#) 的支持类。

### 3.3. JDBC 测试支持

`org.springframework.test.jdbc` 软件包包含 `JdbcTestUtils`，这是 JDBC 相关 Util 功能的集合，旨在简化标准数据库测试方案。具体地说，`JdbcTestUtils` 提供了以下静态 Util 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` 子句从给定表中删除行。

- `dropTables(...)` : 删除指定的表。

## Tip

[AbstractTransactionalJUnit4SpringContextTests](#) 和

[AbstractTransactionalTestNGSpringContextTests](#) 提供了方便的方法，这些方法委托给

`JdbcTestUtils` 中的上述方法。

`spring-jdbc` 模块提供了对配置和启动嵌入式数据库的支持，您可以在与数据库交互的集成测试中使用它。有关详细信息，请参见[嵌入式数据库支持](#)和[使用嵌入式数据库测试数据访问逻辑](#)。

## 3.4. Annotations

本节介绍了在测试 Spring 应用程序时可以使用的 Comments。它包括以下主题：

- [Spring 测试 Comments](#)
- [标准 Comments 支持](#)
- [Spring JUnit 4 测试 Comments](#)
- [Spring JUnit Jupiter 测试 Comments](#)
- [测试的元 Comments 支持](#)

### 3.4.1. Spring 测试 Comments

Spring 框架提供了以下特定于 Spring 的 Comments 集，您可以在单元测试和集成测试中将它们与 `TestContext` 框架结合使用。有关更多信息，请参见相应的 javadoc，包括默认属性值，属性别名和其他详细信息。

Spring 的测试 Comments 包括以下内容：

- [@BootstrapWith](#)

- [@ContextConfiguration](#)
- [@WebAppConfiguration](#)
- [@ContextHierarchy](#)
- [@ActiveProfiles](#)
- [@TestPropertySource](#)
- [@DirtiesContext](#)
- [@TestExecutionListeners](#)
- [@Commit](#)
- [@Rollback](#)
- [@BeforeTransaction](#)
- [@AfterTransaction](#)
- [@Sql](#)
- [@SqlConfig](#)
- [@SqlGroup](#)

## @BootstrapWith

`@BootstrapWith` 是类级别的 Comments，可用于配置如何引导 Spring TestContext Framework

。具体来说，您可以使用 `@BootstrapWith` 指定自定义 `TestContextBootstrapper`。有关更多详细信息，请参见[引导 TestContext 框架](#)部分。

## @ContextConfiguration

`@ContextConfiguration` 定义了类级别的元数据，用于确定如何加载和配置

`ApplicationContext` 以进行集成测试。具体地说，`@ContextConfiguration` 声明了用于加载

上下文的应用程序上下文资源 `locations` 或带 `Comments` 的 `classes`。

资源位置通常是位于 `Classpath` 中的 XML 配置文件或 Groovy 脚本，而带 `Comments` 的类通常是在 `@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 `Comments` 的类可以是组件类，依此类推。

以下示例显示了一个 `@ContextConfiguration` `Comments`，该 `Comments` 引用了 XML 文件：

```
@ContextConfiguration("/test-config.xml") (1)
public class XmlApplicationContextTests {
    // class body...
}
```

- (1) 引用 XML 文件。

以下示例显示了一个 `@ContextConfiguration` `Comments`，该 `Comments` 引用一个类：

```
@ContextConfiguration(classes = TestConfig.class) (1)
public class ConfigClassApplicationContextTests {
    // class body...
}
```

- (1) 提及类。

作为声明资源位置或带 `Comments` 的类的替代方法或补充，您可以使用

`@ContextConfiguration` 声明 `ApplicationContextInitializer` 类。以下示例显示了这种情况：

```
@ContextConfiguration(initializers = CustomContextIntializer.class) (1)
public class ContextInitializerTests {
    // class body...
}
```

- (1) 声明一个初始化程序类。

您也可以选择使用 `@ContextConfiguration` 来声明 `ContextLoader` 策略。但是请注意，由于默认加载器支持 `initializers` 以及资源 `locations` 或带 `Comments` 的 `classes`，因此通常不需要显式配置加载器。

以下示例同时使用位置和装载程序：

```
@ContextConfiguration(locations = "/test-context.xml", loader = CustomContextLoader.class)
public class CustomLoaderXmlApplicationContextTests {
    // class body...
}
```

- (1) 同时配置位置和自定义加载程序。

#### iNote

`@ContextConfiguration` 为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提供支持。

有关更多详细信息，请参见[Context Management](#) 和 `@ContextConfiguration` javadocs。

## @WebAppConfiguration

`@WebAppConfiguration` 是类级别的 Comments，您可以用来声明为集成测试加载的 `ApplicationContext` 应该是 `WebApplicationContext`。在测试类上仅存在 `@WebAppConfiguration` 可以确保为测试加载 `WebApplicationContext`，并使用默认值 `"file:src/main/webapp"` 作为 Web 应用程序根目录的路径(即资源基础路径)。资源基础路径在幕后用于创建 `MockServletContext`，它用作测试 `WebApplicationContext` 的 `ServletContext`。

以下示例显示了如何使用 `@WebAppConfiguration` 注解：

```
@ContextConfiguration
@WebAppConfiguration(1)
public class WebAppTests {
    // class body...
}
```

- (1) `@WebAppConfiguration` Comments。

要覆盖默认值，可以使用隐式 `value` 属性指定其他基础资源路径。 `classpath:` 和 `file:` 资源前缀均受支持。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 Classpath 资源：

```
@ContextConfiguration  
@WebAppConfiguration("classpath:test-web-resources") (1)  
public class WebAppTests {  
    // class body...  
}
```

- (1) 指定 Classpath 资源。

请注意，必须在单个测试类中或在测试类层次结构中将 `@WebAppConfiguration` 与 `@ContextConfiguration` 结合使用。有关更多详细信息，请参见[@WebAppConfiguration](#) javadoc。

## @ContextHierarchy

`@ContextHierarchy` 是类级别的注解，用于定义 `ApplicationContext` 实例的层次结构以进行集成测试。`@ContextHierarchy` 应该用一个或多个 `@ContextConfiguration` 实例的列表声明，每个实例定义上下文层次结构中的一个级别。以下示例演示了在单个测试类中使用 `@ContextHierarchy` (也可以在测试类层次结构中使用 `@ContextHierarchy`)：

```
@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给定级别的上下文层次结构的配置，则必须通过在类层次结构中每个相应级别上为 `@ContextConfiguration` 的 `name` 属性提供相同的值来显式命名该级别。有关更多示例，请参见[Context Hierarchies](#)和[@ContextHierarchy](#) javadoc。

## @ActiveProfiles

`@ActiveProfiles` 是类级别的 Comments，用于声明在加载 `ApplicationContext` 进行集成测试时应激活哪些 bean 定义配置文件。

以下示例表明 `dev` Profile 应处于活动状态：

```
@ContextConfiguration  
@ActiveProfiles("dev") (1)  
public class DeveloperTests {  
    // class body...  
}
```

- (1) 表示 `dev` Profile 应处于活动状态。

以下示例表明 `dev` 和 `integration` 配置文件均应处于活动状态：

```
@ContextConfiguration  
@ActiveProfiles({"dev", "integration"}) (1)  
public class DeveloperIntegrationTests {  
    // class body...  
}
```

- (1) 指示 `dev` 和 `integration` Profile 应处于活动状态。

### iNote

`@ActiveProfiles` 默认情况下支持继承超类声明的活动 bean 定义配置文件。您还可以通过实现自定义[ActiveProfilesResolver](#)并使用 `@ActiveProfiles` 的 `resolver` 属性对其进行编程，从而以编程方式解析活动 bean 定义概要文件。

有关示例和更多详细信息，请参见[使用环境配置文件进行上下文配置](#)和[@ActiveProfiles](#) javadoc。

## @TestPropertySource

`@TestPropertySource` 是类级别的注解，可用于配置属性文件和内联属性的位置，以将其添加到 `Environment` 中的 `PropertySources` 集合中，以便为集成测试加载 `ApplicationContext`。

测试属性源的优先级高于从 `os` 环境或 `Java` 系统属性以及应用程序通过 `@PropertySource` 或以编程方式添加的属性源加载的属性。因此，测试属性源可用于选择性覆盖系统和应用程序属性源中定义的属性。此外，内联属性比从资源位置加载的属性具有更高的优先级。

下面的示例演示如何从 `Classpath` 声明属性文件：

```
@ContextConfiguration  
@TestPropertySource("/test.properties") (1)  
public class MyIntegrationTests {  
    // class body...  
}
```

- (1) 从 `Classpath` 根目录中的 `test.properties` 获取属性。

下面的示例演示如何声明内联属性：

```
@ContextConfiguration  
@TestPropertySource(properties = { "timezone = GMT", "port: 4242" }) (1)  
public class MyIntegrationTests {  
    // class body...  
}
```

- (1) 声明 `timezone` 和 `port` 属性。

## @DirtiesContext

`@DirtiesContext` 表示基础 Spring `ApplicationContext` 在执行测试期间已被弄脏(即，测试以某种方式修改或破坏了它，例如，通过更改单例 bean 的状态)，因此应将其关闭。当应用程序上下文标记为脏时，会将其从测试框架的缓存中删除并关闭。因此，将为需要上下文具有相同配置元数据的任何后续测试重建基础 Spring 容器。

您可以将 `@DirtiesContext` 用作同一类或类层次结构中的类级别和方法级别的 `Comments`。在这

种情况下，取决于配置的 `methodMode` 和 `classMode`，在任何此类带 `Comments` 的方法之前或之后以及当前测试类之前或之后，`ApplicationContext` 被标记为脏。

以下示例说明了在各种配置情况下何时弄脏上下文：

- 在当前测试类之前，在类模式设置为 `BEFORE_CLASS` 的类上声明时。

```
@DirtiesContext(classMode = BEFORE_CLASS) (1)
public class FreshContextTests {
    // some tests that require a new Spring container
}
```

- (1) 在当前测试类之前弄脏上下文。
- 在当前测试类之后，当在类模式设置为 `AFTER_CLASS` 的类上声明时(即默认的类模式)。

```
@DirtiesContext (1)
public class ContextDirtyingTests {
    // some tests that result in the Spring container being dirtied
}
```

- (1) 在当前测试类之后弄脏上下文。
- 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) (1)
public class FreshContextTests {
    // some tests that require a new Spring container
}
```

- (1) 在每种测试方法之前弄乱上下文。
- 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` 的类上声明时

```
@DirtiesContext(classMode = AFTER_EACH_TEST_METHOD) (1)
public class ContextDirtyingTests {
    // some tests that result in the Spring container being dirtied
}
```

- (1) 在每种测试方法之后弄脏上下文。
- 在当前测试之前，在方法模式设置为 `BEFORE_METHOD` 的方法上声明时。

```
@DirtiesContext(methodMode = BEFORE_METHOD) (1)
@Test
public void testProcessWhichRequiresFreshAppCtx() {
    // some logic that requires a new Spring container
}
```

- (1) 在当前测试方法之前弄脏上下文。
- 当前测试之后，当在将方法模式设置为 `AFTER_METHOD` 的方法上声明时(即默认方法模式)。

```
@DirtiesContext (1)
@Test
public void testProcessWhichDirtiesAppCtx() {
    // some logic that results in the Spring container being dirtied
}
```

- (1) 在当前测试方法之后弄脏上下文。

如果在测试中使用 `@DirtiesContext` 且上下文使用 `@ContextHierarchy` 配置为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缓存。默认情况下，使用穷举算法清除上下文缓存，不仅包括当前级别，还包括共享当前测试共有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于矫 kill 过正，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) (1)
    public void test() {
        // some logic that results in the child context being dirtied
}
```

```
    }  
}
```

- (1) 使用当前级别的算法。

有关 `EXHAUSTIVE` 和 `CURRENT_LEVEL` 算法的更多详细信息, 请参见 [DirtiesContext.HierarchyMode javadoc](#)。

## @TestExecutionListeners

`@TestExecutionListeners` 定义了用于配置 `TestExecutionListener` 实现的类级元数据, 该实现应向 `TestContextManager` 注册。通常, `@TestExecutionListeners` 与 `@ContextConfiguration` 结合使用。

下面的示例演示如何注册两个 `TestExecutionListener` 实现:

```
@ContextConfiguration  
@TestExecutionListeners({CustomTestExecutionListener.class, AnotherTestExecutionListener.class})  
public class CustomTestExecutionListenerTests {  
    // class body...  
}
```

- (1) 注册两个 `TestExecutionListener` 实现。

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, 请参见 [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。您可以使用 `@Commit` 作为 `@Rollback(false)` 的直接替代品, 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 `Comments`。

以下示例显示了如何使用 `@Commit` 注解:

```
@Commit (1)
@Test
public void testProcessWithoutRollback() {
    // ...
}
```

- (1) 将测试结果提交到数据库。

## @Rollback

`@Rollback` 指示在测试方法完成后是否应回退事务测试方法的事务。如果为 `true`，则事务回滚。否则，将提交事务(另请参见[@Commit](#))。即使未明确声明 `@Rollback`，Spring TestContext Framework 中集成测试的回滚默认为 `true`。

当声明为类级别的 Comments 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。当声明为方法级别的 Comments 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。

下面的示例使测试方法的结果不回滚(即，结果已提交到数据库)：

```
@Rollback(false) (1)
@Test
public void testProcessWithoutRollback() {
    // ...
}
```

- (1) 不要回退结果。

## @BeforeTransaction

`@BeforeTransaction` 表示已 Comments 的 `void` 方法应在事务启动之前运行，对于已配置为使用 Spring 的 `@Transactional` Comments 在事务内运行的测试方法。从 Spring Framework 4.3 开始，`@BeforeTransaction` 方法不需要为 `public`，并且可以在基于 Java 8 的接口默认方法中声明。

以下示例显示了如何使用 `@BeforeTransaction` 注解：

```
@BeforeTransaction (1)
void beforeTransaction() {
    // logic to be executed before a transaction is started
}
```

- (1) 在 Transaction 前运行此方法。

## @AfterTransaction

`@AfterTransaction` 表示 Comments `void` 方法应在事务结束后运行，对于已配置为使用 Spring 的 `@Transactional` Comments 在事务内运行的测试方法。从 Spring Framework 4.3 开始，`@AfterTransaction` 方法不需要为 `public`，并且可以在基于 Java 8 的接口默认方法中声明。

```
@AfterTransaction (1)
void afterTransaction() {
    // logic to be executed after a transaction has ended
}
```

- (1) 在事务处理后运行此方法。

## @Sql

`@Sql` 用于 Comments 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"}) (1)
public void userTest {
    // execute code that relies on the test schema and test data
}
```

- (1) 为此测试运行两个脚本。

有关更多详细信息，请参见[使用`@Sql`声明式执行 SQL 脚本](#)。

## @SqlConfig

`@SqlConfig` 定义用于确定如何解析和运行用 `@Sql` 注解配置的 SQL 脚本的元数据。以下示例显

示了如何使用它：

```
@Test
@Sql(
    scripts = "/test-user-data.sql",
    config = @SqlConfig(commentPrefix = "`", separator = "@@") (1)
)
public void userTest {
    // execute code that relies on the test data
}
```

- (1) 在 SQL 脚本中设置 Comments 前缀和分隔符。

## @SqlGroup

`@SqlGroup` 是一个容器 Comments，它聚合了几个 `@Sql` Comments。您可以本地使用 `@SqlGroup` 来声明多个嵌套的 `@Sql` Comments，也可以将其与 Java 8 对可重复 Comments 的支持结合使用，其中 `@Sql` 可以在同一类或方法上声明多次，从而隐式生成此容器 Comments。下面的示例显示如何声明一个 SQL 组：

```
@Test
@SqlGroup({ (1)
    @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
}
```

- (1) 声明一组 SQL 脚本。

### 3.4.2. 标准 Comments 支持

Spring TestContext Framework 的所有配置的标准语义都支持以下 Comments。请注意，这些 Comments 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) 是否存在 JSR-250

- `@ManagedBean` (`javax.annotation`) 是否存在 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`

## ①JSR-250 Lifecycle Annotations

在 Spring TestContext Framework 中，可以在 `ApplicationContext` 中配置的任何应用程序组件上以标准语义使用 `@PostConstruct` 和 `@PreDestroy`。但是，这些生命周期 `Comments` 在实际测试类中的使用受到限制。

如果测试类中的方法用 `@PostConstruct` `Comments`，则该方法在基础测试框架的 `before` 方法之前运行(例如，用 JUnit Jupiter 的 `@BeforeEach` `Comments` 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` `Comments`，则该方法将永远不会运行。因此，在测试类中，建议您使用来自基础测试框架的测试生命周期回调，而不是 `@PostConstruct` 和 `@PreDestroy`。

### 3.4.3. Spring JUnit 4 测试 Comments

以下 `Comments` 仅与 [SpringRunner](#), [Spring 的 JUnit 4 规则](#) 或 [Spring 的 JUnit 4 支持类](#) 结合使用时受支持：

- [`@IfProfileValue`](#)

- [@ProfileValueSourceConfiguration](#)
- [@Timed](#)
- [@Repeat](#)

## @IfProfileValue

`@IfProfileValue` 表示已为特定测试环境启用带 Comments 的测试。如果已配置的 `ProfileValueSource` 为提供的 `name` 返回匹配的 `value`，则启用测试。否则，测试将被禁用，并且实际上将被忽略。

您可以在类级别、方法级别或两者上应用 `@IfProfileValue`。对于该类或其子类中的任何方法，`@IfProfileValue` 的类级用法优先于方法级用法。具体来说，如果在类级别和方法级别都启用了测试，则将启用该测试。缺少 `@IfProfileValue` 意味着测试已隐式启用。这类似于 JUnit 4 的 `@Ignore` Comments 的语义，除了 `@Ignore` 的存在总是会禁用测试。

以下示例显示了带有 `@IfProfileValue` 注解的测试：

```
@IfProfileValue(name="java.vendor", value="Oracle Corporation") (1)
@Test
public void testProcessWhichRunsOnlyOnOracleJvm() {
    // some logic that should run only on Java VMs from Oracle Corporation
}
```

- (1) 仅当 Java 供应商是“Oracle Corporation”时才运行此测试。

或者，您可以使用 `@IfProfileValue` 列表(具有 OR 语义)来配置 `@IfProfileValue`，以实现对 JUnit 4 环境中的测试组的类似于 TestNG 的支持。考虑以下示例：

```
@IfProfileValue(name="test-groups", values={"unit-tests", "integration-tests"}) (1)
@Test
public void testProcessWhichRunsForUnitOrIntegrationTestGroups() {
    // some logic that should run only for unit and integration test groups
}
```

- (1) 对单元测试和集成测试运行此测试。

## @ProfileValueSourceConfiguration

`@ProfileValueSourceConfiguration` 是类级别的 Comments，它指定在检索通过 `@IfProfileValue` Comments 配置的配置文件值时要使用的 `ProfileValueSource` 类型。如果未为测试声明 `@ProfileValueSourceConfiguration`，则默认使用 `SystemProfileValueSource`。以下示例显示了如何使用 `@ProfileValueSourceConfiguration`：

```
@ProfileValueSourceConfiguration(CustomProfileValueSource.class) (1)
public class CustomProfileValueSourceTests {
    // class body...
}
```

- (1) 使用自定义配置文件值源。

## @Timed

`@Timed` 表示带 Comments 的测试方法必须在指定的时间段(以毫秒为单位)内完成执行。如果文本执行时间超过指定的时间段，则测试将失败。

该时间段包括运行测试方法本身，测试的任何重复(请参见 `@Repeat`)以及测试夹具的设置或拆除。

以下示例显示了如何使用它：

```
@Timed(millis = 1000) (1)
public void testProcessWithOneSecondTimeout() {
    // some logic that should not take longer than 1 second to execute
}
```

- (1) 将测试时间设置为一秒。

Spring 的 `@Timed` Comments 与 JUnit 4 的 `@Test(timeout=...)` 支持具有不同的语义。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 `Thread` 中执行测试方法)，如果测试花费的时间太长，`@Test(timeout=...)` 将抢先失败。另一方面，Spring 的 `@Timed` 不会抢先通过测试，而是在失败之前 `await` 测试完成。

## @Repeat

`@Repeat` 表示必须重复运行带 Comments 的测试方法。Comments 中指定了要执行测试方法的次数。

重复执行的范围包括测试方法本身的执行以及测试夹具的任何安装或拆除。以下示例显示了如何使用 `@Repeat` 注解：

```
@Repeat(10) (1)
@Test
public void testProcessRepeatedly() {
    // ...
}
```

- (1) 重复此测试十次。

### 3.4.4. Spring JUnit Jupiter 测试 Comments

以下 Comments 仅在与 [SpringExtension](#) 和 JUnit Jupiter(即 JUnit 5 中的编程模型)结合使用时才受支持：

- [@SpringJUnitConfig](#)
- [@SpringJUnitWebConfig](#)
- [@EnabledIf](#)
- [@DisabledIf](#)

## @SpringJUnitConfig

`@SpringJUnitConfig` 是组合的 Comments，它将 JUnit Jupiter 的 `@ExtendWith(SpringExtension.class)` 和 Spring TestContext Framework 的 `@ContextConfiguration` 组合在一起。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 Comments 的类。

以下示例显示如何使用 `@SpringJUnitConfig` 注解指定配置类：

```
@SpringJUnitConfig(TestConfig.class) (1)
class ConfigurationClassJUnitJupiterSpringTests {
    // class body...
}
```

- (1) 指定配置类别。

以下示例显示如何使用 `@SpringJUnitConfig` 注解指定配置文件的位置：

```
@SpringJUnitConfig(locations = "/test-config.xml") (1)
class XmlJUnitJupiterSpringTests {
    // class body...
}
```

- (1) 指定配置文件的位置。

有关更多详细信息，请参见[Context Management](#)以及[@SpringJUnitConfig](#)和

[@ContextConfiguration](#) 的 javadoc。

## @SpringJUnitWebConfig

`@SpringJUnitWebConfig` 是一个组合 Comments，它将来自 JUnit Jupiter 的

`@ExtendWith(SpringExtension.class)` 与来自 Spring TestContext Framework 的

`@ContextConfiguration` 和 `@WebAppConfiguration` 组合在一起。您可以在类级别使用它代替

`@ContextConfiguration` 和 `@WebAppConfiguration`。关于配置选项，

`@ContextConfiguration` 和 `@SpringJUnitWebConfig` 之间的唯一区别是您可以使用

`@SpringJUnitWebConfig` 中的 `value` 属性声明带 Comments 的类 bu。此外，仅可以使用

`@SpringJUnitWebConfig` 中的 `resourcePath` 属性来覆盖 `@WebAppConfiguration` 中的

`value` 属性。

以下示例显示如何使用 `@SpringJUnitWebConfig` 注解指定配置类：

```
@SpringJUnitWebConfig(TestConfig.class) (1)
class ConfigurationClassJUnitJupiterSpringWebTests {
    // class body...
}
```

- (1) 指定配置类别。

以下示例显示如何使用 `@SpringJUnitWebConfig` 注解指定配置文件的位置：

```
@SpringJUnitWebConfig(locations = "/test-config.xml") (1)
class XmlJUnitJupiterSpringWebTests {
    // class body...
}
```

- (1) 指定配置文件的位置。

有关更多详细信息，请参见[Context Management](#)以及[@SpringJUnitWebConfig](#),  
[@ContextConfiguration](#)和[@WebAppConfiguration](#)的 javadoc。

## @EnabledIf

`@EnabledIf` 用于表示已 Comments 的 JUnit Jupiter 测试类或测试方法已启用，如果提供的 `expression` 评估为 `true`，则应运行 `@EnabledIf`。具体来说，如果表达式的计算结果为 `Boolean.TRUE` 或等于 `true` 的 `String` (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意一种：

- [Spring 表达语言\(SpEL\)](#)表达式。例如：

```
@EnabledIf( "#{systemProperties['os.name'].toLowerCase().contains('mac')}" )
```

- Spring [Environment](#)中可用属性的占位符。例如：

```
@EnabledIf( "${smoke.tests.enabled}" )
```

- LiteralsLiterals。例如： `@EnabledIf("true")`

但是请注意，不是属性占位符动态解析的结果的文本 Literals 的实际值为零，因为

`@EnabledIf("false")` 等效于 `@Disabled`，而 `@EnabledIf("true")` 在逻辑上是没有意义的。

您可以使用 `@EnabledIf` 作为元 Comments 来创建自定义的组合 Comments。例如，您可以创建一个自定义 `@EnabledOnMac` Comments，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 Comments 的 JUnit Jupiter 测试类或测试方法已禁用，并且如果提供的 `expression` 计算结果为 `true`，则不应执行 `@DisabledIf`。具体来说，如果表达式的计算结果为 `Boolean.TRUE` 或等于 `true` 的 `String` (忽略大小写)，则会禁用测试。当在类级别应用时，该类中的所有测试方法也会自动禁用。

表达式可以是以下任意一种：

- Spring 表达语言(SpEL)表达式。例如：

```
@DisabledIf("#{systemProperties['os.name'].toLowerCase().contains('mac')}")
```

- Spring Environment 中可用属性的占位符。例如：

```
@DisabledIf("${smoke.tests.disabled}")
```

- LiteralsLiterals。例如： `@DisabledIf("true")`

但是请注意，不是属性占位符动态解析的结果的文本 `Literals` 的实际值为零，因为

`@DisabledIf("true")` 等效于 `@Disabled`，而 `@DisabledIf("false")` 在逻辑上是没有意义的。

您可以使用 `@DisabledIf` 作为元 Comments 来创建自定义的组合 Comments。例如，您可以创

建一个自定义 `@DisabledOnMac` Comments，如下所示：

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@DisabledIf(
    expression = "#{systemProperties['os.name'].toLowerCase().contains('mac')}",
    reason = "Disabled on Mac OS"
)
public @interface DisabledOnMac {}
```

### 3.4.5. 测试的元 Comments 支持

您可以将大多数与测试相关的 Comments 用作[meta-annotations](#)，以创建自定义的组合 Comments，并减少整个测试套件中的配置重复。

您可以将以下各项与[TestContext framework](#)一起用作元 Comments。

- `@BootstrapWith`
- `@ContextConfiguration`
- `@ContextHierarchy`
- `@ActiveProfiles`
- `@TestPropertySource`
- `@DirtiesContext`
- `@WebAppConfiguration`
- `@TestExecutionListeners`
- `@Transactional`
- `@BeforeTransaction`
- `@AfterTransaction`
- `@Commit`
- `@Rollback`

- `@Sql`
- `@SqlConfig`
- `@SqlGroup`
- `@Repeat` (仅 *JUnit 4* 支持)
- `@Timed` (仅 *JUnit 4* 支持)
- `@IfProfileValue` (仅 *JUnit 4* 支持)
- `@ProfileValueSourceConfiguration` (仅 *JUnit 4* 支持)
- `@SpringJUnitConfig` (仅 *JUnit Jupiter* 支持)
- `@SpringJUnitWebConfig` (仅 *JUnit Jupiter* 支持)
- `@EnabledIf` (仅 *JUnit Jupiter* 支持)
- `@DisabledIf` (仅 *JUnit Jupiter* 支持)

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 { }
```

如果发现我们在基于 *JUnit 4* 的测试套件中重复了前面的配置，则可以通过引入一个自定义的组合 *Comments* 来减少重复，该 *Comments* 集中了 Spring 的通用测试配置，如下所示：

```
@Target(ElementType.TYPE)
```

```
@Retention(RetentionPolicy.RUNTIME)
@ContextConfiguration({"app-config.xml", "test-data-access-config.xml"})
@ActiveProfiles("dev")
@Transactional
public @interface TransactionalDevTestConfig { }
```

然后，我们可以使用定制的 `@TransactionalDevTestConfig` Comments 来简化基于单个 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 { }
```

如果我们使用 JUnit Jupiter 编写测试，则可以进一步减少代码重复，因为 JUnit 5 中的 Comments 也可以用作元 Comments。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 { }
```

如果发现我们正在基于 JUnit Jupiter 的测试套件中重复上述配置，则可以通过引入一个自定义的组合 Comments 来减少重复，该 Comments 集中了 Spring 和 JUnit Jupiter 的通用测试配置，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，我们可以使用自定义的 `@TransactionalDevTestConfig` Comments 来简化基于单个 JUnit

Jupiter 的测试类的配置，如下所示：

```
@TransactionalDevTestConfig  
class OrderRepositoryTests { }  
  
@TransactionalDevTestConfig  
class UserRepositoryTests { }
```

由于 JUnit Jupiter 支持将 `@Test`，`@RepeatedTest`，`ParameterizedTest` 等作为元 Comments 使用，因此您也可以在测试方法级别创建自定义的组合 Comments。例如，如果我们希望创建一个组合的 Comments，将 JUnit Jupiter 的 `@Test` 和 `@Tag` Comments 与 Spring 的 `@Transactional` Comments 相结合，则可以创建 `@TransactionalIntegrationTest` Comments，如下所示：

```
@Target(ElementType.METHOD)  
@Retention(RetentionPolicy.RUNTIME)  
@Transactional  
@Tag("integration-test") // org.junit.jupiter.api.Tag  
@Test // org.junit.jupiter.api.Test  
public @interface TransactionalIntegrationTest { }
```

然后，我们可以使用自定义的 `@TransactionalIntegrationTest` Comments 简化基于 JUnit Jupiter 的各个测试方法的配置，如下所示：

```
@TransactionalIntegrationTest  
void saveOrder() { }  
  
@TransactionalIntegrationTest  
void deleteOrder() { }
```

有关更多详细信息，请参见[SpringComments 编程模型](#) Wiki 页面。

## 3.5. Spring TestContext 框架

Spring TestContext Framework(位于 `org.springframework.test.context` 包中)提供了由 Comments 驱动的通用单元和集成测试支持，这些支持与所使用的测试框架无关。TestContext 框架还非常重视约定优于配置，您可以通过基于 Comments 的配置覆盖合理的默认值。

除了通用的测试基础结构之外，`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 `abstract` 个支持类。此外，Spring 为 JUnit 4 提供了自定义 JUnit `Runner` 和自定义 JUnit `Rules` 以及为 JUnit Jupiter 提供了自定义 `Extension`，可让您编写所谓的 POJO 测试类。POJO 测试类不需要扩展特定的类层次结构，例如 `abstract` 支持类。

下一节概述了 `TestContext` 框架的内部。如果您只对使用框架感兴趣，而对使用自己的自定义侦听器或自定义加载程序进行扩展不感兴趣，请直接进入配置([context management](#), [dependency injection](#), [transaction management](#)), [support classes](#)和[annotation support](#)部分。

### 3.5.1. 关键抽象

该框架的核心由 `TestContextManager` 类和 `TestContext`，`TestExecutionListener` 和 `SmartContextLoader` 接口组成。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)。`TestContextManager` 依次 Management 一个 `TestContext`，该 `TestContext` 保留当前测试的上下文。`TestContextManager` 还随着测试的进行更新并委托 `TestExecutionListener` 实现来更新 `TestContext` 的状态，该实现通过提供依赖项注入，Management 事务等手段来检测实际的测试执行。`SmartContextLoader` 负责为给定的测试类加载 `ApplicationContext`。有关更多信息和各种实现的示例，请参见[javadoc](#)和 Spring 测试套件。

#### TestContext

`TestContext` 封装了要在其中执行测试的上下文(与所使用的实际测试框架无关)，并为其负责的测试实例提供了上下文 Management 和缓存支持。`TestContext` 还委托 `SmartContextLoader` 来加载 `ApplicationContext` (如果有要求)。

#### TestContextManager

`TestContextManager` 是 Spring TestContext Framework 的主要入口点，并负责 Management 单个 `TestContext` 并在定义明确的测试执行点向每个注册的 `TestExecutionListener` 发 signal 通知事件：

- 在特定测试框架的任何“上课之前”或“全部之前”方法之前。
- 测试实例后处理。
- 在特定测试框架的任何“之前”或“每个之前”方法之前。
- 立即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立即将测试拆解。
- 在特定测试框架的任何“之后”或“每个之后”方法之后。
- 在特定测试框架的任何“课后”或“毕竟”方法之后。

## TestExecutionListener

`TestExecutionListener` 定义用于对注册侦听器的 `TestContextManager` 发布的测试执行事件做出反应的 API。参见 [TestExecutionListener Configuration](#)。

## Context Loaders

`ContextLoader` 是 Spring 2.5 中引入的策略接口，用于加载 `ApplicationContext` 以便由 Spring TestContext Framework Management 的集成测试。您应该实现 `SmartContextLoader` 而不是此接口，以为带 `Comments` 的类，活动的 Bean 定义配置文件，测试属性源，上下文层次结构和 `WebApplicationContext` 提供支持。

`SmartContextLoader` 是 Spring 3.1 中引入的 `ContextLoader` 接口的扩展。

`SmartContextLoader` SPI 取代了 Spring 2.5 中引入的 `ContextLoader` SPI。具体来说，`SmartContextLoader` 可以选择处理资源位置，带 `Comments` 的类或上下文初始化器。此外，`SmartContextLoader` 可以在其加载的上下文中设置活动 bean 定义概要文件并测试属性源。

Spring 提供了以下实现：

- `DelegatingSmartContextLoader`：两个默认加载器之一，它在内部委托 `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 在 Classpath 上时才启用 Groovy 支持。
- `WebDelegatingSmartContextLoader`：两个默认加载器之一，它在内部委托 `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。仅当测试类上存在 `@WebAppConfiguration` 时才使用 Web `ContextLoader`。仅当 Groovy 在 Classpath 上时才启用 Groovy 支持。
- `AnnotationConfigContextLoader`：从带 Comments 的类中加载标准 `ApplicationContext`。
- `AnnotationConfigWebContextLoader`：从带 Comments 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 属性文件中加载标准 `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希望更改默认的 `ContextLoader`，实现自定义的 `TestContext` 或 `ContextCache`，扩展默认的 `ContextCustomizerFactory` 和 `TestExecutionListener` 实现集，等等。为了对 `TestContext` 框架的运行方式进行低级控制，Spring 提供了自举策略。

`TestContextBootstrapper` 定义了用于引导 `TestContext` 框架的 SPI。`TestContextManager` 使用 `TestContextBootstrapper` 来加载当前测试的 `TestExecutionListener` 实现并构建其 Management 的 `TestContext`。您可以直接使用 `@BootstrapWith` 或作为元 `Comments`，为测试类(或测试类层次结构)配置自定义引导策略。如果没有通过使用 `@BootstrapWith` 显式配置引导程序，则根据 `@WebAppConfiguration` 的存在使用 `DefaultTestContextBootstrapper` 或 `WebTestContextBootstrapper`。

由于 `TestContextBootstrapper` SPI 将来可能会更改(以适应新要求)，因此我们强烈建议实现者不要直接实现此接口，而应扩展 `AbstractTestContextBootstrapper` 或其具体子类之一。

### 3.5.3. TestExecutionListener 配置

Spring 提供了以下 `TestExecutionListener` 实现，这些实现默认情况下按以下 Sequences 注册

:

- `ServletTestExecutionListener`：为 `WebApplicationContext` 配置 Servlet API 模拟。
- `DirtiesContextBeforeModesTestExecutionListener`：处理“之前”模式的 `@DirtiesContext` `Comments`。
- `DependencyInjectionTestExecutionListener`：为测试实例提供依赖项注入。

- `DirtiesContextTestExecutionListener`：处理“之后”模式的 `@DirtiesContext` `Comments`。
- `TransactionalTestExecutionListener`：为事务测试执行提供默认回滚语义。
- `SqlScriptsTestExecutionListener`：运行使用 `@Sql` 注解配置的 SQL 脚本。

## 注册自定义 `TestExecutionListener` 实现

您可以使用 `@TestExecutionListeners` 注解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，请参见[annotation support](#)和[@TestExecutionListeners](#)的 javadoc。

## 自动发现默认的 `TestExecutionListener` 实现

通过使用 `@TestExecutionListeners` 注册自定义 `TestExecutionListener` 实现非常适合在有限的测试方案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 `TestExecutionListener` 实现来解决此问题。

具体来说，`spring-test` 模块在其 `META-INF/spring.factories` 属性文件的 `org.springframework.test.context.TestExecutionListener` 键下声明了所有核心默认的 `TestExecutionListener` `+749+` 实现贡献到默认侦听器列表中。

## `OrderTestExecutionListener` 实现

当 `TestContext` 框架通过[aforementioned](#) `SpringFactoriesLoader` 机制发现默认的 `TestExecutionListener` 实现时，实例化的侦听器将通过使用 Spring 的 `AnnotationAwareOrderComparator` 进行排序，该样式会使用 Spring 的 `Ordered` 接口和 `@Order` `Comments` 进行排序。`AbstractTestExecutionListener` 和 Spring 提供的所有默认

的 `TestExecutionListener` 实现都以适当的值实现 `Ordered`。因此，第三方框架和开发人员应通过实现 `Ordered` 或声明 `@Order` 来确保按默认 Sequences 注册其默认的 `TestExecutionListener` 实现。有关为每个核心侦听器分配哪些值的详细信息，请参见 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方案中，这有效地迫使开发人员手动声明除任何自定义侦听器之外的所有默认侦听器。下面的 Lists 演示了这种配置样式：

```
@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已注册了哪些侦听器。此外，默认的侦听器集可以在各个发行版之间进行更改，例如，在 Spring Framework 4.1 中引入了 `SqlScriptsTestExecutionListener`，在 Spring Framework 4.2 中引入了 `DirtiesContextBeforeModesTestExecutionListener`。此外，像 Spring Security 这样的第三方框架通过使用上述[自动发现机制](#)来注册自己的默认 `TestExecutionListener` 实现。

为了避免必须了解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` 表示应将本地声明的侦听器与默认侦听器合并。合并算法可确保从列表中删除重复项，并确保根据 `AnnotationAwareOrderComparator` 的语义对合并的侦听器的结果集进行排序，如

[OrderTestExecutionListener 实现](#) 中所述。如果监听器实现 `Ordered` 或带有 `@Order` 注解，则它可以影响将其与默认值合并的位置。否则，合并时，本地声明的监听器将追加到默认监听器列表中。

例如，如果上一个示例中的 `MyCustomTestExecutionListener` 类将其 `order` 值(例如 `500`)配置为小于 `ServletTestExecutionListener` (恰好是 `1000`)的 Sequences，那么 `MyCustomTestExecutionListener` 可以自动与默认列表合并在 `ServletTestExecutionListener` 的前面，并且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 {  
    // class body...  
}
```

### 3.5.4. 上下文 Management

每个 `TestContext` 为其负责的测试实例提供上下文 Management 和缓存支持。测试实例不会自动获得对配置的 `ApplicationContext` 的访问权限。但是，如果测试类实现 `ApplicationContextAware` 接口，则将对 `ApplicationContext` 的引用提供给测试实例。请注意，`AbstractJUnit4SpringContextTests` 和 `AbstractTestNGSpringContextTests` 实现 `ApplicationContextAware`，因此自动提供对 `ApplicationContext` 的访问。

#### >{@Autowired ApplicationContext}

作为实现 `ApplicationContextAware` 接口的替代方法，您可以通过字段或 `setter` 方法上的 `@Autowired` Comments 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 (1)  
    private ApplicationContext applicationContext;  
  
    // class body...  
}
```

- (1) 注入 `ApplicationContext`。

同样，如果将测试配置为加载 `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 (1)  
@ContextConfiguration  
public class MyWebAppTest {  
  
    @Autowired (2)  
    private WebApplicationContext wac;  
  
    // class body...  
}
```

- (1) 配置 `WebApplicationContext`。
- (2) 注入 `WebApplicationContext`。

通过 `@Autowired` 提供的依赖关系注入由

`DependencyInjectionTestExecutionListener` 提供，默认情况下已配置

`DependencyInjectionTestExecutionListener` (请参见[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的类或实现特定的接口来配置其应用程序上下文。而是通过在类级别声明 `@ContextConfiguration` `Comments` 来实现配置。如果您的测试类未明确声明应用程序上下文资源位置或带 `Comments` 的类，则已配置的 `ContextLoader` 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 `Comments` 的类，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。

以下各节说明如何通过使用 XML 配置文件, Groovy 脚本, 带 Comments 的类(通常为 `@Configuration` 类)或上下文初始化器, 使用 Spring 的 `@ContextConfiguration` 注解配置测试 `ApplicationContext`。另外, 您可以针对高级用例实现并配置自己的自定义 `SmartContextLoader`。

- [使用 XML 资源进行上下文配置](#)
- [使用 Groovy 脚本进行上下文配置](#)
- [带 Comments 类的上下文配置](#)
- [混合 XML, Groovy 脚本和带 Comments 的类](#)
- [使用上下文初始化器进行上下文配置](#)
- [上下文配置继承](#)
- [使用环境配置文件进行上下文配置](#)
- [具有测试属性源的上下文配置](#)
- [加载 WebApplicationContext](#)
- [Context Caching](#)
- [Context Hierarchies](#)

## 使用 XML 资源进行上下文配置

要使用 XML 配置文件为测试加载 `ApplicationContext`, 请用 `@ContextConfiguration` Comments 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。普通或相对路径(例如 `context.xml`)被视为相对于定义测试类的包的 Classpath 资源。以斜杠开头的路径被视为绝对 Classpath 位置(例如 `/org/example/config.xml`)。按原样使用表示资源 URL 的路径(即以 `classpath:`, `file:`, `http:` 等为前缀的路径)。

```
@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"}) (1)
public class MyTest {
    // class body...
}
```

- (1) 将 `locations` 属性设置为 XML 文件列表。

`@ContextConfiguration` 通过标准 Java `value` 属性支持 `locations` 属性的别名。因此，如果不需要在 `@ContextConfiguration` 中声明其他属性，则可以使用以下示例中演示的速记格式，省略 `locations` 属性名称的声明并声明资源位置：

```
@RunWith(SpringRunner.class)
@ContextConfiguration({"app-config.xml", "test-config.xml"}) (1)
public class MyTest {
    // class body...
}
```

- (1) 在不使用 `location` 属性的情况下指定 XML 文件。

如果您从 `@ContextConfiguration` 注解中省略了 `locations` 和 `value` 属性，则 `TestContext` 框架将尝试检测默认的 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。如果您的类名为 `com.example.MyTest`，则 `GenericXmlContextLoader` 从 `"classpath:com/example/MyTest-context.xml"` 加载您的应用程序上下文。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 (1)
public class MyTest {
    // class body...
}
```

- (1) 从默认位置加载配置。

## 使用 Groovy 脚本进行上下文配置

要使用使用[Groovy Bean 定义 DSL](#)的 Groovy 脚本为测试加载 `ApplicationContext`，可以用 `@ContextConfiguration` `Comments` 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针对[XML 配置文件](#)描述的语义相同。

### Enabling Groovy script support

如果 Groovy 位于 Classpath 上，则会自动启用对使用 Groovy 脚本在 Spring TestContext Framework 中加载 `ApplicationContext` 的支持。

下面的示例显示如何指定 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"}) (1)
public class MyTest {
    // class body...
}
```

- (1) 指定 Groovy 配置文件的位置。

如果您从 `@ContextConfiguration` 注解中省略了 `locations` 和 `value` 属性，则 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。如果您的类名为 `com.example.MyTest`，那么 Groovy 上下文加载器将从 `"classpath:com/example/MyTestContext.groovy"` 加载您的应用程序上下文。下面的示例演示如何使用默认值：

```
package com.example;

@RunWith(SpringRunner.class)
// ApplicationContext will be loaded from
// "classpath:com/example/MyTestContext.groovy"
@ContextConfiguration(1)
public class MyTest {
    // class body...
}
```

- (1) 从默认位置加载配置。

## ⌚ Declaring XML configuration and Groovy scripts simultaneously

您可以使用 `@ContextConfiguration` 的 `locations` 或 `value` 属性同时声明 XML 配置文件和 Groovy 脚本。如果到已配置资源位置的路径以 `.xml` 结尾，则使用 `XmlBeanDefinitionReader` 加载它。否则，将使用 `GroovyBeanDefinitionReader` 加载它。

以下 Lists 显示了如何在集成测试中将两者结合在一起：

```
@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...
}
```

## 带 Comments 类的上下文配置

要使用带 Comments 的类(请参见[基于 Java 的容器配置](#))为测试加载 `ApplicationContext`，可以使用 `@ContextConfiguration` `Comments` 测试类，并使用包含对带 Comments 的类的引用的数组来配置 `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}) (1)
public class MyTest {
    // class body...
```

```
}
```

- (1) 指定带 Comments 的类。

## 💡Annotated Classes

术语“带 Comments 的类”可以指以下任何一种：

- 用 `@Configuration` Comments 的类。
- 组件(即带有 `@Component`，`@Service`，`@Repository` 或其他构造型 Comments 的类)。
- 带有 `javax.inject` 注解的 JSR-330 兼容类。
- 包含 `@Bean` 个方法的任何其他类。

有关带 Comments 的类的配置和语义的更多信息，请参见[@Configuration](#)和[@Bean](#)的 javadoc，尤其要注意 `@Bean` Lite Mode 的讨论。

如果从 `@ContextConfiguration` 注解中省略 `classes` 属性，则 `TestContext` 框架将尝试检测默认配置类的存在。具体来说，`AnnotationConfigContextLoader` 和 `AnnotationConfigWebContextLoader` 检测到满足[@Configuration](#) javadoc 中指定的配置类实现要求的测试类的所有 `static` 嵌套类。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 `static` 嵌套配置类。在下面的示例中，`OrderServiceTest` 类声明了一个名为 `Config` 的 `static` 嵌套配置类，该类将自动用于为测试类加载 `ApplicationContext`：

```
@RunWith(SpringRunner.class)
// ApplicationContext will be loaded from the
// static nested Config class
@Configuration(1)
public class OrderServiceTest {

    static class Config {
```

```

// this bean will be injected into the OrderServiceTest class
@Bean
public OrderService orderService() {
    OrderService orderService = new OrderServiceImpl();
    // set properties, etc.
    return orderService;
}
}

@Autowired
private OrderService orderService;

@Test
public void testOrderService() {
    // test the orderService
}

}

```

- (1) 从嵌套类加载配置信息。

## 混合 XML，Groovy 脚本和带 Comments 的类

有时可能需要混合使用 XML 配置文件，Groovy 脚本和带 Comments 的类(通常为

`@Configuration` 类)来为测试配置 `ApplicationContext`。例如，如果在 Producing 使用 XML 配置，则可以决定要使用 `@Configuration` 类为测试配置特定的 Spring 托管组件，反之亦然。

此外，某些第三方框架(例如 Spring Boot)提供了一流的支持，可同时从不同类型的资源(例如 XML 配置文件，Groovy 脚本和 `@Configuration` 类)中加载 `ApplicationContext`。过去，Spring 框架不支持此标准部署。因此，Spring 框架在 `spring-test` 模块中提供的大多数

`SmartContextLoader` 实现对于每个测试上下文仅支持一种资源类型。但是，这并不意味着您不能同时使用两者。一般规则的一个 exception 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同时支持 `locations` 和 `classes` 到 `@ContextConfiguration` 的声明，并且，借助 `TestContext` 框架中的标准测试支持，您可以选择以下选项。

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，并且其中一个必须包含或导入另一个。例如，在 XML 或 Groovy 脚本中，可以通过使用

组件扫描或将它们定义为普通的 Spring bean 来包含 `@Configuration` 类，而在 `@Configuration` 类中，可以使用 `@ImportResource` 导入 XML 配置文件或 Groovy 脚本。请注意，此行为在语义上等同于您在生产环境中配置应用程序的方式：在生产配置中，您定义了一组 XML 或 Groovy 资源位置或一组 `@Configuration` 类，从中加载了生产 `ApplicationContext`，但是您仍然拥有自由包括或导入其他类型的配置。

## 使用上下文初始化程序进行上下文配置

要使用上下文初始化器为测试配置 `ApplicationContext`，请使用 `@ContextConfiguration` `Comments` 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来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` 类型必须与使用中的 `SmartContextLoader` 创建的 `ApplicationContext` 类型(通常为 `GenericApplicationContext`)兼容。此外，初始化程序的调用 `Sequences` 取决于它们是实现 Spring 的 `Ordered` 接口还是用 Spring 的 `@Order` `Comments` 或标准 `@Priority` `Comments` 进行 `Comments`。下面的示例演示如何使用初始化程序：

```
@RunWith(SpringRunner.class)
// ApplicationContext will be loaded from TestConfig
// and initialized by TestAppCtxInitializer
@Configuration(
    classes = TestConfig.class,
    initializers = TestAppCtxInitializer.class) (1)
public class MyTest {
    // class body...
}
```

- (1) 通过使用配置类和初始化程序来指定配置。

您也可以完全省略 `@ContextConfiguration` 中的 XML 配置文件，Groovy 脚本或带 `Comments` 的类的声明，而仅声明 `ApplicationContextInitializer` 类，然后这些类负责在上下文中注册 Bean(例如，通过编程方式从 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) (1)
public class MyTest {
    // class body...
}
```

- (1) 仅使用初始化程序来指定配置。

## 上下文配置继承

`@ContextConfiguration` 支持布尔值 `inheritLocations` 和 `inheritInitializers`，它们指示是否应继承资源位置或超类声明的带 `Comments` 的类和上下文初始化器。这两个标志的默认值为 `true`。这意味着测试类将继承资源位置或带 `Comments` 的类以及任何超类声明的上下文初始化器。具体地说，将测试类的资源位置或带 `Comments` 的类追加到超类声明的资源位置或带 `Comments` 的类的列表中。同样，将给定测试类的初始化程序添加到由测试超类定义的初始化程序集。因此，子类可以选择扩展资源位置，带 `Comments` 的类或上下文初始化器。

如果 `@ContextConfiguration` 中的 `inheritLocations` 或 `inheritInitializers` 属性设置为 `false`，则用于测试类影子的资源位置或带 `Comments` 的类以及上下文初始化器分别有效地替换由超类定义的配置。

在下一个使用 XML 资源位置的示例中，从 `base-config.xml` 和 `extended-config.xml` 依次加载 `ExtendedTest` 的 `ApplicationContext`。因此，`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") (1)
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") (2)  
public class ExtendedTest extends BaseTest {  
    // class body...  
}
```

- (1) 在超类中定义的配置文件。
- (2) 在子类中定义的配置文件。

同样，在下一个使用带 `Comments` 的类的示例中，从 `BaseConfig` 和 `ExtendedConfig` 类按该 `Sequences` 加载 `ExtendedTest` 的 `ApplicationContext`。因此，`ExtendedConfig` 中定义的 Bean 可以覆盖(即替换) `BaseConfig` 中定义的 Bean。以下示例显示一个类如何扩展另一个类，并同时使用其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) (1)  
public class BaseTest {  
    // class body...  
}  
  
// ApplicationContext will be loaded from BaseConfig and ExtendedConfig  
@ContextConfiguration(classes = ExtendedConfig.class) (2)  
public class ExtendedTest extends BaseTest {  
    // class body...  
}
```

- (1) 在超类中定义的配置类。
- (2) 在子类中定义的配置类。

在使用上下文初始化程序的下一个示例中，通过使用 `BaseInitializer` 和 `ExtendedInitializer` 初始化 `ExtendedTest` 的 `ApplicationContext`。但是请注意，初始化程序的调用 `Sequences` 取决于它们是实现 Spring 的 `Ordered` 接口还是用 Spring 的 `@Order` `Comments` 或标准 `@Priority` `Comments` 进行 `Comments`。下面的示例显示一个类如何扩展另一个类，并同时使用其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) (1)
public class BaseTest {
    // class body...
}

// ApplicationContext will be initialized by BaseInitializer
// and ExtendedInitializer
@ContextConfiguration(initializers = ExtendedInitializer.class) (2)
public class ExtendedTest extends BaseTest {
    // class body...
}

```

- (1) 在超类中定义的初始化程序。
- (2) 子类中定义的初始化程序。

## 使用环境配置文件进行上下文配置

Spring 3.1 在框架中引入了对环境和概要文件(AKA“ bean 定义概要文件” )概念的一流支持，并且可以配置集成测试以针对各种测试场景激活特定的 bean 定义概要文件。这可以通过用

`@ActiveProfiles` Comments 对测试类进行 Comments 并提供在加载 `ApplicationContext` 进行测试时应激活的配置文件列表来实现。

### iNote

您可以将 `@ActiveProfiles` 与新 `SmartContextLoader` SPI 的任何实现一起使用，但是较旧的 `ContextLoader` SPI 的实现不支持 `@ActiveProfiles`。

考虑两个具有 XML 配置和 `@Configuration` 类的示例：

```

<!-- 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")
public class TransferServiceTest {

    @Autowired
    private TransferService transferService;

    @Test
    public void testTransferService() {
        // test the transferService
    }
}

```

运行 `TransferServiceTest` 时，其 `ApplicationContext` 是从 Classpath 根目录中的 `app-config.xml` 配置文件中加载的。如果检查 `app-config.xml`，则可以看到 `accountRepository` bean 对 `dataSource` bean 有依赖性。但是，`dataSource` 没有定义为顶级 bean。而是，`dataSource` 定义了 3 次：在 `production` 配置文件中，在 `dev` 配置文件中和在 `default` 配置文件中。

通过用 `@ActiveProfiles("dev")` Comments `TransferServiceTest`，我们指示 Spring TestContext Framework 加载 `ApplicationContext` 并将活动配置文件设置为 `{"dev"}`。结果，创建了一个嵌入式数据库，并用测试数据填充了该数据库，并通过对开发 `DataSource` 的引用连接 `accountRepository` bean。这可能是我们在集成测试中想要的。

将 bean 分配给 `default` 概要文件有时很有用。仅当没有专门激活其他配置文件时，才包含默认配置文件中的 Bean。您可以使用它来定义要在应用程序的默认状态下使用的“后备”bean。例如，您可以显式提供 `dev` 和 `production` 配置文件的数据源，但是当两者都不处于活动状态时，将内存中数据源定义为默认数据源。

以下代码 Lists 演示了如何使用 `@Configuration` 类而不是 XML 来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();
    }
}
```

```
@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 配置分为四个独立的 `@Configuration` 类：

- `TransferServiceConfig`：使用 `@Autowired` 通过依赖项注入获取 `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，则为默认的嵌入式数据库定义 `dataSource`。

与基于 XML 的配置示例一样，我们仍然使用 `@ActiveProfiles("dev")` Comments

`TransferServiceTest`，但是这次我们使用 `@ContextConfiguration` Comments 指定所有四个配置类。测试类的主体本身保持完全不变。

通常在给定项目中跨多个测试类使用一组概要文件。因此，为避免重复声明 `@ActiveProfiles` 注解，可以在 Base Class 上声明一次 `@ActiveProfiles`，并且子类会自动从 Base Class 继承 `@ActiveProfiles` 配置。在以下示例中，`@ActiveProfiles` 的声明(以及其他 Comments)已移至抽象超类 `AbstractIntegrationTest`：

```
package com.bank.service;

@RunWith(SpringRunner.class)
@ContextConfiguration(classes = {
    TransferServiceConfig.class,
    StandaloneDataConfig.class,
    JndiDataConfig.class,
    DefaultDataConfig.class})
@ActiveProfiles("dev")
public abstract class AbstractIntegrationTest {
```

```
package com.bank.service;

// "dev" profile inherited from superclass
public class TransferServiceTest extends AbstractIntegrationTest {

    @Autowired
    private TransferService transferService;

    @Test
    public void testTransferService() {
        // test the transferService
    }
}
```

`@ActiveProfiles` 还支持 `inheritProfiles` 属性，该属性可用于禁用活动配置文件的继承，如下示例所示：

```
package com.bank.service;

// "dev" profile overridden with "production"
@ActiveProfiles(profiles = "production", inheritProfiles = false)
public class ProductionTransferServiceTest extends AbstractIntegrationTest {
    // test body
}
```

此外，有时有必要以编程方式而不是声明方式来解析测试的活动配置文件，例如，基于：

- 当前的 OS。
- 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。
- 自定义类级别 Comments 的存在。
- Other concerns.

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` 并通过使用 `@ActiveProfiles` 的 `resolver` 属性对其进行注册。有关更多信息，请参见相应的[javadoc](#)。以下示例演示了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};
    }
}
```

## 具有测试属性源的上下文配置

Spring 3.1 在框架中引入了对具有属性源层次结构的环境的概念的一流支持。从 Spring 4.1 开始，您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` 注解相反，您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。将这些测试属性源添加到 `Environment` 中的 `PropertySources` 集合中，以供为 `Comments` 的集成测试加载的 `ApplicationContext`。

### iNote

您可以将 `@TestPropertySource` 与 `SmartContextLoader` SPI 的任何实现一起使用，但是较旧的 `ContextLoader` SPI 的实现不支持 `@TestPropertySource`。

`SmartContextLoader` 的实现通过 `MergedContextConfiguration` 中的 `getPropertySourceLocations()` 和 `getPropertySourceProperties()` 方法访问合并的测试属性源值。

## 声明测试属性源

您可以使用 `@TestPropertySource` 的 `locations` 或 `value` 属性来配置测试属性文件。

支持传统属性文件格式和基于 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 `Resource`。纯路径(例如 `"test.properties"`)被视为相对于定义测试类的程序包的 Classpath 资源。以斜杠开头的路径被视为绝对 Classpath 资源(例如 `"/org/example/test.xml"`)。通过使用指定的资源协议来加载引用 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 `*/.properties`)：每个位置都必须精确评估一个 `.properties` 或 `.xml` 资源。

以下示例使用测试属性文件：

```
@ContextConfiguration  
@TestPropertySource("/test.properties") (1)  
public class MyIntegrationTests {  
    // class body...  
}
```

- (1) 指定具有绝对路径的属性文件。

您可以使用 `@TestPropertySource` 的 `properties` 属性以键值对的形式配置内联属性，如下面的示例所示。将所有键值对作为具有最高优先级的单个测试 `PropertySource` 添加到封闭的 `Environment` 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`

下面的示例设置两个内联属性：

```
@ContextConfiguration  
@TestPropertySource(properties = {"timezone = GMT", "port: 4242"}) (1)  
public class MyIntegrationTests {  
    // class body...  
}
```

- (1) 通过使用键值语法的两种变体来设置两个属性。

## 默认属性文件检测

如果 `@TestPropertySource` 被声明为空 `Comments`(即 `locations` 或 `properties` 属性没有显式值)，则会尝试检测相对于声明该 `Comments` 的类的默认属性文件。例如，如果带 `Comments`

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，则会引发 `IllegalStateException`。

## Precedence

测试属性源的优先级高于从 os 环境、Java 系统属性或应用程序通过使用 `@PropertySource` 声明性地添加的属性源加载的属性。因此，测试属性源可用于选择性覆盖系统和应用程序属性源中定义的属性。此外，内联属性比从资源位置加载的属性具有更高的优先级。

在下一个示例中，`timezone` 和 `port` 属性以及 `"/test.properties"` 中定义的任何属性都将覆盖系统和应用程序属性源中定义的同名属性。此外，如果 `"/test.properties"` 文件为 `timezone` 和 `port` 属性定义了条目，则这些条目将被使用 `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同时指定属性：

```
@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` 支持布尔值 `inheritLocations` 和 `inheritProperties`，它们指示是否应继承属性文件和超类声明的内联属性的资源位置。这两个标志的默认值为 `true`。这意味着测试类将继承任何超类声明的位置和内联属性。具体来说，将测试类的位置和内联属性附加到超类声明的位置和内联属性中。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 shade(即，覆盖)之前出现的相同名称的属性。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为影子并有效替换超类定义的配置。

在下一个示例中，仅通过将 `base.properties` 文件用作测试属性源来加载

`ApplicationContext` for `BaseTest`。相反，通过将 `base.properties` 和 `extended.properties` 文件用作测试属性源位置来加载\_5 的 `ApplicationContext`。下面的示例显示如何通过使用 `properties` 文件在子类及其父类中定义属性：

```
@TestPropertySource("base.properties")
@ContextConfiguration
public class BaseTest {
    // ...
}

@TestPropertySource("extended.properties")
@ContextConfiguration
public class ExtendedTest extends BaseTest {
    // ...
}
```

在下一个示例中，仅使用内联的 `key1` 属性加载 `BaseTest` 的 `ApplicationContext`。相反，使用内联的 `key1` 和 `key2` 属性来加载\_5 的 `ApplicationContext`。下面的示例演示如何通过使用内联属性在子类及其父类中定义属性：

```
@TestPropertySource(properties = "key1 = value1")
@ContextConfiguration
public class BaseTest {
    // ...
}

@TestPropertySource(properties = "key2 = value2")
@ContextConfiguration
public class ExtendedTest extends BaseTest {
    // ...
}
```

## 加载 WebApplicationContext

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`，可以用 `@WebAppConfiguration` `Comments` 各自的测试类。

测试类上 `@WebAppConfiguration` 的存在指示 `TestContext` 框架(TCF)应该为集成测试加载

`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)。如果您熟悉 Maven 项目中 Web 应用程序的目录结构，则知道 `src/main/webapp` 是 WAR 根目录的默认位置。如果需要覆盖此默认值，则可以提供 `@WebAppConfiguration` Comments 的备用路径(例如 `@WebAppConfiguration("src/test/webapp")`)。如果您希望从 Classpath 而不是文件系统中引用基本资源路径，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` 实现的支持相当。使用 `WebApplicationContext` 进行测试时，可以通过使用 `@ContextConfiguration` 来声明 XML 配置文件，Groovy 脚本或 `@Configuration` 类。您还可以自由使用任何其他测试 Comments，例如 `@ActiveProfiles`，`@TestExecutionListeners`，`@Sql`，`@Rollback` 等。

本节中的其余示例显示了用于加载 `WebApplicationContext` 的各种配置选项。以下示例显示了 `TestContext` 框架对配置约定的支持：

#### 例子 1. 约定

```
@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` Comments 测试类而未指定资源基本路径，则资源路径实际上默认为 `file:src/main/webapp`。同样，如果您声明 `@ContextConfiguration` 而不指定资源

`locations`，带 `Comments` 的 `classes` 或上下文 `initializers`，则 Spring 会尝试使用约定 (即 `WacTests-context.xml` 与 `WacTests` 类或静态嵌套 `@Configuration` 类放在同一包中) 来检测配置的存在。

下面的示例演示如何使用 `@WebAppConfiguration` 显式声明资源基础路径和使用 `@ContextConfiguration` 声明 XML 资源位置：

### 例子 2. 默认资源语义

```
@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 `Comments` 的路径的语义不同。默认情况下，

`@WebAppConfiguration` 资源路径基于文件系统，而 `@ContextConfiguration` 资源位置基于 Classpath。

下面的示例显示，我们可以通过指定 Spring 资源前缀来覆盖两个 `Comments` 的默认资源语义：

### 例子 3. 显式资源语义

```
@RunWith(SpringRunner.class)

// classpath resource
@WebAppConfiguration("classpath:test-web-resources")

// file system resource
@ContextConfiguration("file:src/main/webapp/WEB-INF/servlet-config.xml")

public class WacTests {
    //...
}
```

将本示例中的 `Comments` 与上一个示例进行对比。

## 使用网络模拟

为了提供全面的 Web 测试支持, Spring 3.2 引入了默认情况下启用的 `ServletTestExecutionListener`。在针对 `WebApplicationContext` 进行测试时, 此 `TestExecutionListener` 通过在每种测试方法之前使用 Spring Web 的 `RequestContextHolder` 来 设置默认的线程本地状态, 并根据 `@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, 它将清除线程本地状态。

加载 `WebApplicationContext` 进行测试后, 您可能会发现需要与 Web 模拟进行交互, 例如, 设置测试夹具或在调用 Web 组件后执行声明。以下示例显示可以将哪些模拟自动连接到您的测试实例。请注意, `WebApplicationContext` 和 `MockServletContext` 都缓存在测试套件中, 而其他模拟则由 `ServletTestExecutionListener` 按测试方法进行 Management。

### 例子 4.注入模拟

```
@WebAppConfiguration
@ContextConfiguration
public class WacTests {

    @Autowired
    WebApplicationContext wac; // cached

    @Autowired
    MockServletContext servletContext; // cached

    @Autowired
    MockHttpSession session;

    @Autowired
    MockHttpServletRequest request;

    @Autowired
    MockHttpServletResponse response;

    @Autowired
    ServletWebRequest webRequest;

    //...
}
```

## Context Caching

一旦 TestContext 框架为测试加载 `ApplicationContext` (或 `WebApplicationContext`)，该上下文将被缓存并重新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，重要的是要了解“唯一”和“测试套件”的含义。

`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密钥，在该密钥下缓存上下文。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`)
- `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 框架将加载相应的 `ApplicationContext` 并将其存储在 `static` 上下文缓存中，该缓存位于仅基于那些位置的键下

。因此，如果 `TestClassB` 也为`其位置(通过继承显式或隐式)定义 { "app-config.xml", "test-config.xml" }`，但未`定义 @WebAppConfiguration`，不同的 `ContextLoader`，不同的活动配置文件，不同的上下文初始化器，不同的测试属性源或不同的父上下文，则相同的 `ApplicationContext` 由两个测试类共享。这意味着加载应用程序上下文的设置成本仅发生一次（每个测试套件），并且随后的测试执行要快得多。

## ①Test suites and forked processes

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 `static` 变量中。换句话说，如果测试是在单独的进程中执行的，则在每次测试执行之间都会清除静态缓存，从而有效地禁用了缓存机制。

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 中以组的形式执行所有测试来实现。同样，在使用诸如 Ant, Maven 或 Gradle 之类的构建框架执行测试时，确保该构建框架不会在测试之间进行派生很重要。例如，如果将 Maven Surefire 插件的 `forkMode` 设置为 `always` 或 `pertest`，则 TestContext 框架将无法在测试类之间缓存应用程序上下文，因此，构建过程的运行速度将大大降低。

从 Spring Framework 4.3 开始，上下文缓存的大小限制为默认的最大大小 32.只要达到最大大小，就会使用最近最少使用(LRU)驱逐策略来驱逐和关闭陈旧的上下文。您可以通过设置名为 `spring.test.context.cache.maxSize` 的 JVM 系统属性，从命令行或构建脚本中配置最大大小。或者，您可以使用 `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该套件花费不必要的长时间执行，因此准确地知道已加载和缓存了多少个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，测试破坏了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），可以用 `@DirtiesContext` `Comments` 测试类或测试方法（请参见 `@DirtiesContext` 的讨论）[Spring 测试 Comments](#)）。这指示 Spring 在运行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，

`DirtiesContextBeforeModesTestExecutionListener` 和

`DirtiesContextTestExecutionListener` 提供了对 `@DirtiesContext` `Comments` 的支持，默  
认情况下已启用它们。

## Context Hierarchies

在编写依赖于已加载的 Spring `ApplicationContext` 的集成测试时，通常足以针对单个上下文进  
行测试。但是，有时需要对 `ApplicationContext` 个实例的层次结构进行测试是有益的，甚至有  
必要进行测试。例如，如果您正在开发 Spring MVC Web 应用程序，则通常具有由 Spring 的  
`ContextLoaderListener` 加载的根 `WebApplicationContext` 和由 Spring 的  
`DispatcherServlet` 加载的子级 `WebApplicationContext`。这导致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，并在特定于 Web 的组件的子上下文中使用。在  
Spring Batch 应用程序中可以找到另一个用例，在该应用程序中，您经常有一个父上下文为共享批  
处理基础结构提供配置，而子上下文为特定批处理作业的配置提供配置。

从 Spring Framework 3.2.2 开始，您可以通过在单个测试类上或在测试类层次结构中使用  
`@ContextHierarchy` 注解声明上下文配置来编写使用上下文层次结构的集成测试。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  
`Comments` 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。

本节中其余的基于 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。

具有上下文层次结构的单个测试类

`ControllerIntegrationTests` `pass` 语句一个上下文层次结构来表示 Spring MVC Web 应用程  
序的典型集成测试场景，该上下文层次结构包含两个级别，一个层次用于根  
`WebApplicationContext` (使用 `TestAppConfig` `@Configuration` 类加载)，一个层次用于调

度程序 servlet `WebApplicationContext` (使用 `WebApplicationContext` 加载)。 `WebConfig`

`@Configuration` 类)。自动连接到测试实例中的 `WebApplicationContext` 是用于子上下文(即, 层次结构中的最低上下文)的那个。以下 Lists 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

此示例中的测试类在测试类层次结构中定义了上下文层次结构。 `AbstractWebTests` 在 Spring 支

持的 Web 应用程序中声明根 `WebApplicationContext` 的配置。但是请注意,

`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` 的声明一个), 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 Lists 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 {}
```

## 具有合并上下文层次结构配置的类层次结构

本示例中的类显示了使用命名层次结构级别的目的，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 `parent` 和 `child`。`ExtendedTests` 扩展 `BaseTests` 并通过确保 `@ContextConfiguration` 的 `name` 属性中声明的名称都为 `child` 来指示 Spring TestContext Framework 合并 `child` 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`，一个用于 `{"/user-config.xml", "/order-config.xml"}`。与前面的示例一样，将从 `/app-config.xml` 加载的应用程序上下文设置为从 `/user-config.xml` 和 `{"/user-config.xml", "/order-config.xml"}` 加载的上下文的父上下文。以下 Lists 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")
)
public class ExtendedTests extends BaseTests {}
```

## 具有覆盖的上下文层次结构配置的类层次结构

与前面的示例相反，此示例演示如何通过将 `@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，并且其父级设置为从 `/app-config.xml` 加载的上下文。以下 Lists 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 {}
```

### ➊ Dirtying a context within a context hierarchy

如果在上下文被配置为上下文层次结构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缓存。有关更多详细信息，请参见[Spring 测  
试 Comments](#)和[@DirtiesContext](#) javadoc 中有关 `@DirtiesContext` 的讨论。

### 3.5.5. 测试夹具的依赖注入

使用 `DependencyInjectionTestExecutionListener` (默认情况下已配置)时，测试实例的依赖项  
从使用 `@ContextConfiguration` 配置的应用程序上下文中的 bean 注入。您可以使用 setter 注入  
, 字段注入或同时使用这两种方式, 这取决于您选择的 Comments 以及是否将它们放置在 setter  
方法或字段中。为了与 Spring 2.5 和 3.0 中引入的 Comments 支持保持一致, 可以使用 Spring 的  
`@Autowired` Comments 或 JSR 330 中的 `@Inject` Comments。

#### Tip

`TestContext` 框架不检测测试实例的实例化方式。因此，对于构造函数使用 `@Autowired` 或  
`@Inject` 对测试类无效。

因为 `@Autowired` 用于执行[按类型自动布线](#)，所以如果您具有相同类型的多个 bean 定义，那么对  
于那些特定的 bean，您将不能依靠这种方法。在这种情况下，您可以将 `@Autowired` 与  
`@Qualifier` 结合使用。从 Spring 3.0 开始，您还可以选择将 `@Inject` 与 `@Named` 结合使用。另  
外，如果您的测试类可以访问其 `ApplicationContext`，则可以通过使用(例如)对

`applicationContext.getBean("titleRepository")` 的调用来执行显式查找。

如果您不希望将依赖项注入应用于测试实例，请不要使用 `@Autowired` 或 `@Inject` Comments 字段或设置方法。或者，可以通过用 `@TestExecutionListeners` 显式配置您的类并从侦听器列表中省略 `DependencyInjectionTestExecutionListener.class` 来完全禁用依赖项注入。

请考虑测试[Goals](#)部分中概述的 `HibernateTitleRepository` 类的方案。接下来的两个代码 Lists 演示了在字段和 setter 方法上使用 `@Autowired` 的方法。在所有示例代码 Lists 之后显示了应用程序上下文配置。

### iNote

以下代码 Lists 中的依赖项注入行为并非特定于 JUnit4. 相同的 DI 技术可以与任何测试框架结合使用。

以下示例调用了静态 assert 方法，例如 `assertNotNull()`，但未在调用前加上 `Assert`。

在这种情况下，假定该方法已通过示例中未显示的 `import static` 声明正确导入。

第一个代码 Lists 显示了使用 `@Autowired` 进行字段注入的测试类的基于 JUnit 4 的实现：

```
@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，您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 Lists 使用 `@ContextConfiguration` Comments(即 `repository-config.xml` )引用的相同 XML 上下文文件。下面显示了此配置：

```

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

    <!-- this bean will be injected into the HibernateTitleRepositoryTests class -->
    <bean id="titleRepository" class="com.foo.repository.hibernate.HibernateTitleRepository">
        <property name="sessionFactory" ref="sessionFactory"/>
    </bean>

    <bean id="sessionFactory" class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
        <!-- configuration elided for brevity -->
    </bean>
</beans>

```

### iNote

如果从 Spring 提供的测试 Base Class 扩展而来，而该 Base Class 恰巧在其 `setter` 方法之上使用 `@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，您可以重写 `setter` 方法并使用 `@Qualifier` 注解指示特定的目标 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，从而将类型匹配的范围缩小到特定的 bean。其值与相应 `<bean>` 定义中的 `<qualifier>` 声明匹配。Bean 名称用作后备限定符值，因此您也可以在该名称中有效地指向特定的 Bean(如先前所示，假设 `myDataSource` 是 Bean `id`)。

### 3.5.6. 测试请求和会话范围的 Bean

早年以来，Spring 就一直支持[请求范围和会话范围的 bean](#)。从 Spring 3.2 开始，您可以按照以下步骤测试请求范围和会话范围的 bean：

- 通过用 `@WebAppConfiguration` `Comments` 测试类，确保为测试加载了 `WebApplicationContext`。
- 将模拟请求或会话注入到您的测试实例中，并根据需要准备测试夹具。
- 调用从已配置的 `WebApplicationContext` 中检索到的 Web 组件(带有依赖项注入)。
- 对模拟执行 assert。

下一个代码片段显示了登录用例的 XML 配置。注意，`userService` bean 与请求范围内的 `loginAction` bean 有依赖性。另外，通过使用[SpEL expressions](#)实例化 `LoginAction`，该 [SpEL expressions](#)从当前 HTTP 请求中检索用户名和密码。在我们的测试中，我们想通过 `TestContext` 框架 `Management` 的模拟来配置这些请求参数。以下 Lists 显示了此用例的配置：

例子 5. 请求范围的 bean 配置

```

<beans>

    <bean id="userService" class="com.example.SimpleUserService"
          c:loginAction-ref="loginAction"/>

    <bean id="loginAction" class="com.example.LoginAction"
          c:username="#{request.getParameter('user')}"
          c:password="#{request.getParameter('pswd')}"
          scope="request">
        <aop:scoped-proxy/>
    </bean>

</beans>

```

在 `RequestScopedBeanTests` 中，我们将 `UserService` (即被测对象) 和 `MockHttpServletRequest` 都注入到我们的测试实例中。在 `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` (即，我们只在其中设置了参数)。然后，我们可以根据用户名和密码的已知 Importing 对结果进行 assert。以下 Lists 显示了如何执行此操作：

#### 例子 6. 请求范围的 bean 测试

```

@RunWith(SpringRunner.class)
@ContextConfiguration
@WebAppConfiguration
public class RequestScopedBeanTests {

    @Autowired UserService userService;
    @Autowired MockHttpServletRequest request;

    @Test
    public void requestScope() {
        request.setParameter("user", "enigma");
        request.setParameter("pswd", "$pr!ng");

        LoginResults results = userService.loginUser();
        // assert results
    }
}

```

以下代码段类似于我们之前针对请求范围的 Bean 看到的代码段。但是，这次，`userService` bean 对会话范围的 `userPreferences` bean 具有依赖性。注意，`UserPreferences` bean 是通

过使用 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，我们需要在由 `TestContext` 框架 `Management` 的模拟会话中配置主题。以下示例显示了如何执行此操作：

### 例子 7.会话范围的 bean 配置

```
<beans>

    <bean id="userService" class="com.example.SimpleUserService"
          c:userPreferences-ref="userPreferences" />

    <bean id="userPreferences" class="com.example.UserPreferences"
          c:theme="#{session.getAttribute('theme')}"
          scope="session">
        <aop:scoped-proxy/>
    </bean>

</beans>
```

在 `SessionScopedBeanTests` 中，我们将 `UserService` 和 `MockHttpSession` 注入到我们的测试实例中。在 `sessionScope()` 测试方法中，我们通过在提供的 `MockHttpSession` 中设置预期的 `theme` 属性来设置测试夹具。在 `userService` 上调用 `processUserPreferences()` 方法时，可以确保用户服务可以访问当前 `MockHttpSession` 的会话作用域 `userPreferences`，并且可以基于配置的主题对结果执行 `assert`。以下示例显示了如何执行此操作：

### 例子 8.会话范围的 bean 测试

```
@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. TransactionManagement

在 `TestContext` 框架中，事务由 `TransactionalTestExecutionListener` Management，默认情况下配置为 `TransactionalTestExecutionListener`，即使您没有在测试类上显式声明 `@TestExecutionListeners`。但是，要启用对事务的支持，必须在 `ApplicationContext` 中配置一个 `platformTransactionManager` bean，该 Bean 加载了 `@ContextConfiguration` 语义 (稍后将提供更多详细信息)。另外，必须在测试的类或方法级别声明 Spring 的 `@Transactional` 注解。

#### Test-managed Transactions

测试 Management 的事务是使用 `TransactionalTestExecutionListener` 声明式 Management 或使用 `TestTransaction` 以编程方式 Management 的事务(稍后描述)。您不应将此类事务与 Spring 托管的事务(由 `ApplicationContext` 加载以供测试直接由 Spring 直接 Management 的事务)或应用程序托管的事务(由测试调用的应用程序代码中以编程方式 Management 的事务)混淆。SpringManagement 的事务和应用程序 Management 的事务通常参与测试 Management 的事务。但是，如果使用 `REQUIRED` 或 `SUPPORTS` 以外的任何传播类型配置了 SpringManagement 的事务或应用程序 Management 的事务，则应格外小心(有关详细信息，请参见[transaction propagation](#)的讨论)。

#### 启用和禁用 Transaction

用 `@Transactional` Comments 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会自动回滚。如果测试类用 `@Transactional` Comments，则该类层次结构中的每个测试方法都在事务中运行。未用 `@Transactional` Comments 的测试方法(在类或方法级别)不在事务内运行。此外，用 `@Transactional` Comments 但将 `propagation` 类型设置为 `NOT_SUPPORTED` 的测试不在事务内运行。

请注意，[AbstractTransactionalJUnit4SpringContextTests](#) 和

[AbstractTransactionalTestNGSpringContextTests](#) 已预先配置为在类级别提供事务支持。

下面的示例演示了为基于 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
        sessionFactory.getCurrentSession().flush();
        assertEquals(count + 1, countRowsInTable("user"));
    }

    protected int countRowsInTable(String tableName) {
        return Jdbc TestUtils.countRowsInTable(this.jdbcTemplate, tableName);
    }

    protected void assertEquals(int expected) {
        assertEquals("Number of rows in the [user] table.", expected, countRowsInTable("user"));
    }
}
```

如[事务回滚和提交行为](#)中所述，在 [createUser\(\)](#) 方法运行后无需清理数据库，因为

[TransactionalTestExecutionListener](#) 会自动回滚对数据库所做的任何更改。有关其他示例，[请参见 PetClinic Example](#)。

## 事务回滚和提交行为

默认情况下，测试 Transaction 将在测试完成后自动回滚；但是，可以通过 [@Commit](#) 和

`@Rollback` `Comments` 声明性地配置事务提交和回滚行为。有关更多详细信息，请参见 [annotation support](#) 部分中的相应条目。

## 程序化 TransactionManagement

从 Spring Framework 4.1 开始，您可以使用 `TestTransaction` 中的静态方法以编程方式与测试 Management 的事务进行交互。例如，您可以在测试方法中，方法之前和之后使用 `TestTransaction` 来启动或结束当前的测试 Management 的事务，或配置当前的测试 Management 的事务以进行回滚或提交。启用 `TransactionalTestExecutionListener` 时，将自动提供对 `TestTransaction` 的支持。

以下示例演示了 `TestTransaction` 的某些功能。有关更多详细信息，请参见 [TestTransaction](#) 的 javadoc。

```
@ContextConfiguration(classes = TestConfig.class)
public class ProgrammaticTransactionManagementTests extends
    AbstractTransactionalJUnit4SpringContextTests {

    @Test
    public void transactionalTest() {
        // assert initial state in test database:
        assertEquals(2, countRowsInTable("user"));

        // changes to the database will be committed!
        TestTransaction.flagForCommit();
        TestTransaction.end();
        assertFalse(TestTransaction.isActive());
        assertEquals(0, countRowsInTable("user"));

        TestTransaction.start();
        // perform other actions against the database that will
        // be automatically rolled back after the test completes...
    }

    protected void assertEquals(int expected) {
        assertEquals("Number of rows in the [user] table.", expected, countRowsInTable("user"));
    }
}
```

## 在事务外运行代码

有时，您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之后验证预期的事务提交行为(如果测试已配置为提交事务)。

`TransactionalTestExecutionListener` 完全支持这种情况的 `@BeforeTransaction` 和 `@AfterTransaction` 注解。您可以使用这些 `Comments` 之一 `Comments` 测试类中的任何 `void` 方法或测试接口中的任何 `void` 默认方法，并且 `TransactionalTestExecutionListener` 确保您的事务之前或之后事务方法在适当的时间运行。

### Tip

任何之前的方法(例如，以 JUnit Jupiter 的 `@BeforeEach` `Comments` 的方法)和任何之后的方法(例如以 JUnit Jupiter 的 `@AfterEach` `Comments` 的方法)都在事务内运行。此外，对于未配置为在事务内运行的测试方法，不会运行带有 `@BeforeTransaction` 或 `@AfterTransaction` `Comments` 的方法。

## 配置事务 Management 器

`TransactionalTestExecutionListener` 期望在 Spring `ApplicationContext` 中为测试定义 `PlatformTransactionManager` bean。如果测试的 `ApplicationContext` 中有 `PlatformTransactionManager` 的多个实例，则可以使用 `@Transactional("myTxMgr")` 或 `@Transactional(transactionManager = "myTxMgr")` 声明限定符，或者 `TransactionManagementConfigurer` 可以由 `@Configuration` 类实现。有关在测试 `ApplicationContext` 中用于查找事务 Management 器的算法的详细信息，请查阅 [TestContextTransactionUtils.retrieveTransactionManager\(\) 的 Javadoc](#)。

## 演示所有与 Transaction 相关的 Comments

以下基于 JUnit 4 的示例显示了一个虚拟的集成测试方案，该方案突出显示了所有与事务相关的 `Comments`。该示例并不是为了演示最佳实践，而是为了演示如何使用这些 `Comments`。有关更多信息和配置示例，请参见[annotation support](#)部分。[@Sql 的事务 Management](#)包含一个其他示例

，该示例使用 `@Sql` 以默认事务回滚语义对声明性 SQL 脚本执行。以下示例以粗体显示了相关

Comments：

```
@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
    }
}
```

## ①Avoid false positives when testing ORM code

当您测试操纵 `Hibernate` 会话或 `JPA` 持久性上下文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：您的测试通过了，但是相同的代码在实际的生产环境中引发异常。请注意，这适用于任何维护内存中工作单元的 ORM 框架。在下面的基于 `Hibernate` 的示例测试案例中，一种方法演示了误报，而另一种方法正确地公开了刷新会话的结果：

```
// ...
```

```

@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();
}

// ...

```

以下示例显示了 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 脚本来修改数据库架构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，为

“初始化”嵌入式或现有数据库提供支持。有关详情，请参见[嵌入式数据库支持](#)和[使用嵌入式数据库测试数据访问逻辑](#)。

尽管在加载 `ApplicationContext` 时一次初始化数据库以进行测试非常有用，但是有时在集成测试过程中能够修改数据库是必不可少的。以下各节说明在集成测试期间如何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.AbstractTransactionalJUnit4SpringContextTest`
- `org.springframework.test.context.testng.AbstractTransactionalTestNGSpringContextTest`

`ScriptUtils` 提供了用于处理 SQL 脚本的静态 Util 方法的集合，并且主要供框架内部使用。但是，如果您需要完全控制 SQL 脚本的解析和执行方式，则 `ScriptUtils` 可能比稍后介绍的其他一些替代方法更适合您的需求。有关详细信息，请参见 `ScriptUtils` 中的[javadoc](#)。

`ResourceDatabasePopulator` 提供了基于对象的 API，可通过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` 提供了用于配置在解析和运行脚本时使用的字符编码，语句分隔符，Comments 定界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，请参见[javadoc](#)。要运行 `ResourceDatabasePopulator` 中配置的脚本，可以调用 `populate(Connection)` 方法以针对 `java.sql.Connection` 执行填充程序，也可以调用 `execute(DataSource)` 方法以针对 `javax.sql.DataSource` 执行填充程序。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` 执行这些脚本：

```
@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` 内部委派给 `ScriptUtils` 来解析和运行 SQL 脚本。同样，[AbstractTransactionalJUnit4SpringContextTests](#) 和 [AbstractTransactionalTestNGSpringContextTests](#) 中的 `executeSqlScript(..)` 方法内部使用 `ResourceDatabasePopulator` 运行 SQL 脚本。有关各种详细信息，请参见 javadoc 中的各种 `executeSqlScript(..)` 方法。

## 使用`@Sql` 声明式执行 SQL 脚本

除了上述用于以编程方式运行 SQL 脚本的机制之外，您还可以在 Spring TestContext Framework 中声明性地配置 SQL 脚本。具体而言，您可以在测试类或测试方法上声明 `@Sql` 注解，以配置应该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，方法级别的声明将覆盖类级别的声明，并且 `SqlScriptsTestExecutionListener` 提供了对 `@Sql` 的支持，默认情况下已启用。

## 路径资源语义

每个路径都被解释为 Spring `Resource`。纯路径(例如 `"schema.sql"`)被视为相对于定义测试类的程序包的 Classpath 资源。以斜杠开头的路径被视为绝对 Classpath 资源(例如 `"/org/example/schema.sql"`)。通过使用指定的资源协议来加载引用 URL 的路径(例如，以 `classpath:`，`file:`，`http:` 为前缀的路径)。

以下示例显示如何在基于 JUnit Jupiter 的集成测试类中的类级别和方法级别使用 `@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
    }
}

```

## 默认脚本检测

如果未指定任何 SQL 脚本，则根据声明 `@Sql` 的位置来尝试检测 `default` 脚本。如果无法检测到默认值，则会引发 `IllegalStateException`。

- 类级别的声明：如果带 `Comments` 的测试类为 `com.example.MyTest`，则相应的默认脚本为

`classpath:com/example/MyTest.sql`。

- 方法级别的声明：如果带 `Comments` 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，但使用不同的语法配置，不同的错误处理规则或每组不同的执行阶段，则可以声明 `@Sql` 的多个实例。使用 Java 8，您可以将 `@Sql` 用作可重复 `Comments`。否则，您可以使用 `@SqlGroup` 注解作为显式容器来声明 `@Sql` 的多个实例。

以下示例显示了如何将 `@Sql` 用作 Java 8 的可重复 `Comments`：

```

@Test
@Sql/scripts = "/test-schema.sql", config = @SqlConfig(commentPrefix = "`"))
@Sql("/test-user-data.sql")

```

```
public void userTest {  
    // execute code that uses the test schema and test data  
}
```

在前面的示例中提出的方案中，`test-schema.sql` 脚本对单行 Comments 使用了不同的语法。

以下示例与前面的示例相同，除了 `@Sql` 声明在 `@SqlGroup` 内分组在一起，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一组特定的脚本(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，分别从 `Sql.TransactionMode` 和 `Sql.ExecutionPhase` 静态导入 `ISOLATED` 和 `AFTER_TEST_METHOD`。

## 使用`@SqlConfig` 进行脚本配置

您可以使用 `@SqlConfig` 注解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的 `Comments` 时，`@SqlConfig` 充当测试类层次结构中所有 SQL 脚本的全局配置。通过使用 `@Sql` 注解的 `config` 属性直接声明时，`@SqlConfig` 用作封闭 `@Sql` 注解中声明的 SQL 脚本的本地配置。`@SqlConfig` 中的每个属性都有一个隐式默认值，该默认值记录在相应属性的 `javadoc` 中。不幸的是，由于 Java 语言规范中为 `Comments` 属性定义了规则，因此无法将 `null` 的值分配给 `Comments` 属性。因此，为了支持对继承的全局配置的覆盖，`@SqlConfig` 属性的显式默认值为 `" "` (对于字符串)或 `DEFAULT` (对于枚举)。这种方法允许 `@SqlConfig` 的局部声明通过提供 `" "` 或 `DEFAULT` 以外的值来选择性地覆盖 `@SqlConfig` 的全局声明中的各个属性。只要本地 `@SqlConfig` 属性未提供 `" "` 或 `DEFAULT` 以外的显式值，就会继承全局 `@SqlConfig` 属性。因此，显式本地配置将覆盖全局配置。

`@Sql` 和 `@SqlConfig` 提供的配置选项与 `ScriptUtils` 和 `ResourceDatabasePopulator` 支持的配置选项等效，但是 `<jdbc:initialize-database/>` XML 名称空间元素提供的配置选项的超集。有关详细信息，请参见 [@Sql](#) 和 [@SqlConfig](#) 中的各个属性的 `javadoc`。

## `@Sql` 的 TransactionManagement

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，根据 `transactionMode` 属性的配置值，SQL 脚本可以在没有事务的情况下运行，而可以在现有的 `SpringManagement` 的事务中运行(例如，由 `TransactionalTestExecutionListener` Management 的事务以 `@Transactional` `Comments`)，也可以在隔离的事务中运行。在 `@SqlConfig` 中，并且在测试的 `ApplicationContext` 中存在 `PlatformTransactionManager`。但是，作为最低要求，测试的 `ApplicationContext` 中必须存在 `javax.sql.DataSource`。

如果 `SqlScriptsTestExecutionListener` 用来检测 `DataSource` 和

`PlatformTransactionManager` 并推断事务语义的算法不符合您的需求，则可以通过设置 `@SqlConfig` 的 `dataSource` 和 `transactionManager` 属性来指定显式名称。此外，您可以通过设置 `@SqlConfig` 的 `transactionMode` 属性来控制事务传播行为(例如，是否应在隔离的事务中运行脚本)。尽管对 `@Sql` 事务 Management 的所有受支持选项的详尽讨论不在本参考手册的范围内，但是`@SqlConfig`和`SqlScriptsTestExecutionListener`的 javadoc 提供了详细信息，并且以下示例显示了使用 JUnit Jupiter 和 `@Sql` 的事务测试的典型测试方案。：

```
@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:
        assertEquals(2, countRowsInTable("user"));
        // execute code that uses the test data...
    }

    int countRowsInTable(String tableName) {
        return Jdbc TestUtils.countRowsInTable(this.jdbcTemplate, tableName);
    }

    void assertEquals(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(在测试方法中或在 `/test-data.sql` 脚本中)都会由 `TransactionalTestExecutionListener` 自动回滚(有关详细信息，请参见[transaction management](#))。

### 3.5.9. 并行测试执行

当使用 Spring TestContext Framework 时，Spring Framework 5.0 引入了对在单个 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，而无需更改测试代码或配置。

### Tip

有关如何设置并行测试执行的详细信息，请参见您的测试框架，构建工具或 IDE 的文档。

请记住，在您的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，对于何时不并行执行测试，Spring 团队提供了以下一般准则。

如果测试符合以下条件，请勿并行执行测试：

- 使用 Spring 的 `@DirtiesContext` 支持。
- 使用 JUnit 4 的 `@FixMethodOrder` 支持或旨在确保测试方法按特定 Sequences 运行的任何测试框架功能。但是请注意，如果整个测试类都是并行执行的，则此方法不适用。
- 更改共享服务或系统(如数据库，消息代理，文件系统等)的状态。这适用于内存系统和外部系统。
  -

### Tip

如果并行测试执行失败，并指出当前测试的 `ApplicationContext` 不再处于活动状态，则通常意味着 `ApplicationContext` 已从另一个线程中的 `ContextCache` 中删除。

这可能是由于使用 `@DirtiesContext` 或由于从 `ContextCache` 自动驱逐。如果 `@DirtiesContext` 是罪魁祸首，则您需要找到一种避免使用 `@DirtiesContext` 的方法，或者从并行执行中排除此类测试。如果已超过 `ContextCache` 的最大大小，则可以增加缓存的最大大小。有关详细信息，请参见[context caching](#)上的讨论。

### Warning

仅当基础 `TestContext` 实现提供了副本构造函数时，才可以在 Spring TestContext Framework 中并行执行测试，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 Framework 支持类

本节描述了支持 Spring TestContext Framework 的各种类。

#### Spring JUnit 4 Runner

Spring TestContext Framework 通过自定义运行程序(在 JUnit 4.12 或更高版本上受支持)提供了与 JUnit 4 的完全集成。通过使用 `@RunWith(SpringJUnit4ClassRunner.class)` 或更短的 `@RunWith(SpringRunner.class)` Comments 测试类，开发人员可以实现基于 JUnit 4 的标准单元测试和集成测试，并同时获得 TestContext 框架的好处，例如支持加载应用程序上下文，测试实例的依赖注入，事务性测试。方法执行，等等。如果您想将 Spring TestContext Framework 与备用运行程序(例如 JUnit 4 的 `Parameterized` 运行程序)或第三方运行程序(例如 `MockitoJUnitRunner`)一起使用，则可以选择使用[Spring 对 JUnit 规则的支持](#)。

以下代码 Lists 显示了配置测试类以与自定义 Spring `Runner` 一起运行的最低要求：

```
@RunWith(SpringRunner.class)
@TestExecutionListeners({})
public class SimpleTest {

    @Test
    public void testMethod() {
        // execute test logic...
    }
}
```

在前面的示例中，`@TestExecutionListeners` 配置有一个空列表以禁用默认侦听器，否则将需要通过 `@ContextConfiguration` 配置 `ApplicationContext`。

## Spring JUnit 4 规则

`org.springframework.test.context.junit4.rules` 软件包提供以下 JUnit 4 规则(在 JUnit 4.12 或更高版本上受支持):

- `SpringClassRule`
- `SpringMethodRule`

`SpringClassRule` 是支持 Spring TestContext Framework 的类级功能的 JUnit `TestRule`，而 `SpringMethodRule` 是支持 Spring TestContext Framework 的实例级和方法级的功能的 JUnit `MethodRule`。

与 `SpringRunner` 相比，Spring 的基于规则的 JUnit 支持具有独立于任何 `org.junit.runner.Runner` 实现的优点，因此可以与现有的替代运行程序(例如 JUnit 4 的 `Parameterized`)或第三方运行程序(例如 `MockitoJUnitRunner`)。

为了支持 TestContext 框架的全部功能，必须将 `SpringClassRule` 与 `SpringMethodRule` 组合在一起。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` 软件包为基于 JUnit 4 的测试用例提供了以下支

持类(在 JUnit 4.12 或更高版本上受支持):

- `AbstractJUnit4SpringContextTests`
- `AbstractTransactionalJUnit4SpringContextTests`

`AbstractJUnit4SpringContextTests` 是抽象的基础测试类，它在 JUnit 4 环境中将 Spring

TestContext Framework 与显式的 `ApplicationContext` 测试支持集成在一起。扩展

`AbstractJUnit4SpringContextTests` 时，可以访问 `protected ApplicationContext` 实例变量，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。

此类期望在 `ApplicationContext` 中定义 `javax.sql.DataSource` bean 和 `PlatformTransactionManager` bean。扩展

`AbstractTransactionalJUnit4SpringContextTests` 时，可以访问 `protected`

`JdbcTemplate` 实例变量，该实例变量可用于运行 SQL 语句来查询数据库。您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，并且 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，请务必避免使用 [false positives](#)。如 [JDBC 测试支持](#) 中所述，`AbstractTransactionalJUnit4SpringContextTests` 还提供了便捷方法，这些方法通过使用上述 `JdbcTemplate` 委派给 `JdbcTestUtils` 中的方法。此外，

`AbstractTransactionalJUnit4SpringContextTests` 提供了 `executeSqlScript(..)` 方法，用于针对已配置的 `DataSource` 运行 SQL 脚本。

### Tip

这些类为扩展提供了便利。如果您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](#) 来配置自己的自定义测试

类。

## JUnit Jupiter 的 SpringExtension

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)` Comments 测试类，您可以实现基于 JUnit Jupiter 的标准单元测试和集成测试，并同时获得 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，事务性测试方法执行等。

此外，得益于 JUnit Jupiter 中丰富的扩展 API，Spring 可以提供除 Spring 支持的 JUnit 4 和 TestNG 功能集之外的以下功能：

- 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，请参见 [SpringExtension 的依赖注入](#)。
- 基于 SpEL 表达式，环境变量，系统属性等，对[有条件的测试执行](#)的强大支持。有关更多详细信息和示例，请参见[Spring JUnit Jupiter 测试 Comments](#)中 `@EnabledIf` 和 `@DisabledIf` 的文档。
- 定制组合 Comments，结合了 Spring 和 JUnit Jupiter 的 Comments。有关更多详细信息，请参见[测试的元 Comments 支持](#)中的 `@TransactionalDevTestConfig` 和 `@TransactionalIntegrationTest` 示例。

以下代码 Lists 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...
    }
}
```

由于您还可以将 JUnit 5 中的 `Comments` 用作元 `Comments`, 因此 Spring 可以提供

`@SpringJUnitConfig` 和 `@SpringJUnitWebConfig` 组成的 `Comments`, 以简化测试 `ApplicationContext` 和 JUnit Jupiter 的配置。

以下示例使用 `@SpringJUnitConfig` 减少上一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...
    }
}
```

同样, 以下示例使用 `@SpringJUnitWebConfig` 创建一个 `WebApplicationContext` 以便与 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, 请参见[Spring JUnit Jupiter 测试 Comments](#)中 `@SpringJUnitConfig` 和 `@SpringJUnitWebConfig` 的文档。

## 使用 `SpringExtension` 进行依赖项注入

`SpringExtension` 实现了 JUnit Jupiter 的[ParameterResolver](#)扩展 API, 该 API 使 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项注入。

具体来说, `SpringExtension` 可以将来自测试 `ApplicationContext` 的依赖项注入到以

```
@BeforeAll , @AfterAll , @BeforeEach , @AfterEach , @Test , @RepeatedTest ,  
@ParameterizedTest 等标记的测试构造函数和方法中。
```

## Constructor Injection

如果 JUnit Jupiter 测试类的构造函数中的参数类型为 `ApplicationContext` (或其子类型), 或者使用 `@Autowired`, `@Qualifier` 或 `@Value` 进行 Comments 或元 Comments, 则 Spring 会为该特定参数的值注入相应的值。测试 `ApplicationContext` 中的 bean。如果所有参数都应由 Spring 提供, 则也可以用 `@Autowired` 直接 Comments 测试构造函数。

### ⚠ Warning

如果测试类的构造函数本身带有 `@Autowired` Comments, 则 Spring 负责解析构造函数中的 \* all \* 参数。因此, 在 JUnit Jupiter 中注册的其他 `ParameterResolver` 都无法解析此类构造函数的参数。

在以下示例中, Spring 将从 `TestConfig.class` 加载的 `ApplicationContext` 注入 `OrderService` bean 到 `OrderServiceIntegrationTests` 构造函数中。

```
@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, 此功能使测试依赖项为 `final`, 因此不可变。

## Method Injection

如果 JUnit Jupiter 测试方法或测试生命周期回调方法中的参数类型为 `ApplicationContext` (或其子类型), 或者使用 `@Autowired`, `@Qualifier` 或 `@Value` 进行 Comments 或元 Comments , 则 Spring 会使用以下命令为该特定参数注入值测试的 `ApplicationContext` 对应的 bean。

在以下示例中, Spring 将从 `TestConfig.class` 加载的 `ApplicationContext` 注入 `OrderService` 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
    }
}
```

由于 JUnit Jupiter 中 `ParameterResolver` 支持的强大功能, 您不仅可以从 Spring 中而且从 JUnit Jupiter 本身或其他第三方扩展中也可以将多个依赖项注入到单个方法中。

下面的示例演示如何让 Spring 和 JUnit Jupiter 同时将依赖项注入到 `placeOrderRepeatedly()` 测试方法中。

```
@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, 通过使用 JUnit Jupiter 中的 `@RepeatedTest` , 测试方法可以访问 `RepetitionInfo` 。

## TestNG 支持类

`org.springframework.test.context.testng` 软件包为基于 TestNG 的测试用例提供以下支持类:

- `AbstractTestNGSpringContextTests`
- `AbstractTransactionalTestNGSpringContextTests`

`AbstractTestNGSpringContextTests` 是抽象的基础测试类，该类将 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 `protected ApplicationContext` 实例变量，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。此类期望在 `ApplicationContext` 中定义 `javax.sql.DataSource` bean 和 `PlatformTransactionManager` bean。扩展 `AbstractTransactionalTestNGSpringContextTests` 时，可以访问 `protected JdbcTemplate` 实例变量，该实例变量可用于执行 SQL 语句来查询数据库。您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，并且 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，请务必避免使用 [false positives](#)。如 [JDBC 测试支持](#) 中所述，`AbstractTransactionalTestNGSpringContextTests` 还提供了便捷方法，这些方法通过使用上述 `JdbcTemplate` 委派给 `JdbcTestUtils` 中的方法。此外，`AbstractTransactionalTestNGSpringContextTests` 提供了 `executeSqlScript(..)` 方法，用于针对已配置的 `DataSource` 运行 SQL 脚本。

### Tip

这些类为扩展提供了便利。如果您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` 等来配置自己的自定义测试类，并使用 `TestContextManager` 手动检测您的测试类。有关如何检测您的测试类的示例

， 请参见 `AbstractTestNGSpringContextTests` 的源代码。

## 3.6. Spring MVC 测试框架

Spring MVC 测试框架提供了一流的支持，可以使用可与 JUnit、TestNG 或任何其他测试框架一起使用的流畅 API 测试 Spring MVC 代码。它是基于 `spring-test` 模块的 [Servlet API 模拟对象](#) 构建的，因此不使用正在运行的 Servlet 容器。它使用 `DispatcherServlet` 来提供完整的 Spring MVC 运行时行为，并支持通过 `TestContext` 框架加载实际的 Spring 配置以及独立模式，在该模式下，您可以手动实例化控制器并一次对其进行测试。

Spring MVC Test 还为使用 `RestTemplate` 的代码提供 Client 端支持。Client 端测试模拟服务器响应，并且不使用运行中的服务器。

### Tip

Spring Boot 提供了一个选项，可以编写包括运行中的服务器在内的完整的端到端集成测试。如果这是您的目标，请参见[Spring Boot 参考页](#)。有关容器外测试与端到端集成测试之间的区别的更多信息，请参见[容器外测试与端到端集成测试之间的差异](#)。

### 3.6.1. 服务器端测试

您可以使用 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，将其注入模拟或存根依赖性，然后调用其方法(根据需要传递 `MockHttpServletRequest`，`MockHttpServletResponse` 和其他方法)。但是，在编写这样的单元测试时，仍有很多未经测试的内容：例如，请求 Map，数据绑定，类型转换，验证等等。此外，也可以调用其他控制器方法(例如 `@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` 生成响应来提供一种

测试控制器的有效方法。

Spring MVC Test 构建在 `spring-test` 模块中熟悉的 [Servlet API 的“模拟”实现](#) 的基础上。这允许执行请求和生成响应，而无需在 `Servlet` 容器中运行。在大多数情况下，一切都应像在运行时一样工作，但有一些值得注意的异常，如[容器外测试与端到端集成测试之间的差异](#) 中所述。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"));
    }
}
```

前面的测试依赖于 `TestContext` 框架的 `WebApplicationContext` 支持从与测试类位于同一包中的 XML 配置文件中加载 Spring 配置，但是还支持基于 Java 和基于 Groovy 的配置。参见这些[sample tests](#)。

`MockMvc` 实例用于对 `/accounts/1` 执行 `GET` 请求，并验证结果响应的状态为 200，Content Type 为 `application/json`，响应主体具有名为 `name` 的 JSON 属性，值为 `Lee`。Jayway [JsonPath project](#) 支持 `jsonPath` 语法。本文档后面将讨论用于验证执行请求结果的许多其他选项。

## Static Imports

示例 [preceding section](#) 中的 FluentAPI 需要一些静态导入，例如 `MockMvcRequestBuilders.*`，

`MockMvcResultMatchers.*` 和 `MockMvcBuilders.*`。查找这些类的一种简单方法是搜索与 `MockMvc*` 匹配的类型。如果使用 Eclipse 或基于 Eclipse 的 Spring Tool Suite, 请确保在 Java→ 编辑器→Content Assist→收藏夹下的 Eclipse 首选项中将它们添加为“收藏的静态成员”。这样, 您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。检查对静态成员的代码完成支持。

## Setup Choices

创建 `MockMvc` 实例有两个主要选项。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` 注入到测试中以用于构建 `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();
    }

    // ...
}
```

您的第二个选择是在不加载 Spring 配置的情况下手动创建控制器实例。而是自动创建基本的默认配置, 该配置与 MVC `JavaConfig` 或 MVC 命名空间大致相当。您可以在一定程度上对其进行自定义。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();
    }
}
```

```
// ...  
}
```

您应该使用哪个设置选项？

`webAppContextSetup` 加载您的实际 Spring MVC 配置，从而进行更完整的集成测试。由于

`TestContext` 框架缓存了已加载的 Spring 配置，因此即使您在测试套件中引入了更多测试，它也有助于保持测试的快速运行。此外，您可以通过 Spring 配置将模拟服务注入控制器中，以 continue 专注于测试 Web 层。下面的示例使用 Mockito 声明一个模拟服务：

```
<bean id="accountService" class="org.mockito.Mockito" factory-method="mock">  
    <constructor-arg value="org.example.AccountService"/>  
</bean>
```

然后，您可以将模拟服务注入测试中，以设置和验证您的期望，如以下示例所示：

```
@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` 稍微接近单元测试。它一次测试一个控制器。您可以手动注入具有模拟依赖项的控制器，并且不涉及加载 Spring 配置。这样的测试更多地集中在样式上，使查看被测试的控制器，是否需要任何特定的 Spring MVC 配置等工作变得更加容易。`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“集成与单元测试”的 Arguments 一样，没有正确或错误的答案。但是，使用 `standaloneSetup` 确实意味着需要进行额外的 `webAppContextSetup` 测试，以验证您的 Spring

MVC 配置。另外，您可以使用 `webAppContextSetup` 编写所有测试，以便始终针对实际的 Spring MVC 配置进行测试。

## Setup Features

无论您使用哪种 MockMvc 构建器，所有 `MockMvcBuilder` 实现都提供一些常见且非常有用的功能。例如，您可以为所有请求声明 `Accept` Headers，并在所有响应中期望状态为 200 以及 `Content-Type` Headers，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)可以预先打包安装说明，例如 `MockMvcConfigurer` 中的那些。

Spring Framework 具有一个这样的内置实现，可以帮助保存和重用跨请求的 HTTP 会话。您可以按以下方式使用它：

```
// static import of SharedHttpSessionConfigurer.sharedHttpSession

MockMvc mockMvc = MockMvcBuilders.standaloneSetup(new TestController())
    .apply(sharedHttpSession())
    .build();

// Use mockMvc to perform requests...
```

有关所有 MockMvc 构建器功能的列表，请参见[ConfigurableMockMvcBuilder](#)的 javadoc 或使用 IDE 探索可用选项。

## Performing Requests

您可以使用任何 HTTP 方法执行请求，如以下示例所示：

```
mockMvc.perform(post("/hotels/{id}", 42).accept(MediaType.APPLICATION_JSON));
```

您还可以执行内部使用 `MockMultipartHttpServletRequest` 的文件上传请求，这样就不会对

Multipart 请求进行实际的解析。相反，您必须将其设置为类似于以下示例：

```
mockMvc.perform(multipart("/doc").file("a1", "ABC".getBytes("UTF-8")));
```

您可以使用 URI 模板样式指定查询参数，如以下示例所示：

```
mockMvc.perform(get("/hotels?thing={thing}", "somewhere"));
```

您还可以添加代表查询或表单参数的 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 Servlet 请求参数并且没有显式检查查询字符串(通常是这种情况)，那么使用哪个选项都没有关系。但是请记住，随 URI 模板提供的查询参数将被解码，而通过 `param(...)` 方法提供的请求参数应已被解码。

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。如果必须使用完整的请求 URI 进行测试，请确保相应地设置 `contextPath` 和 `servletPath`，以便请求 Map 起作用，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` 将很麻烦。相反，您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();  
    }  
}
```

前面的属性会影响通过 `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都指定它们。

## Defining Expectations

您可以通过在执行请求后追加一个或多个 `.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 `assert` 验证响应的属性(例如, 响应状态, Headers 和内容)。这些是要 `assert` 的最重要的结果。

第二类 `assert` 超出了响应范围。这些 `assert` 使您可以检查 Spring MVC 的特定方面, 例如哪种控制器方法处理了请求, 是否引发和处理异常, 模型的内容是什么, 选择了哪种视图, 添加了哪些闪存属性, 等等。它们还使您可以检查 Servlet 的特定方面, 例如请求和会话属性。

以下测试 `assert` 绑定或验证失败:

```
mockMvc.perform(post("/persons"))
    .andExpect(status().isOk())
    .andExpect(model().attributeHasErrors("person"));
```

很多时候, 编写测试时, 转储已执行请求的结果很有用。您可以按照以下步骤操作, 其中

`print()` 是从 `MockMvcResultHandlers` 的静态导入:

```
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 `log()` 方法和 `print()` 方法的两个其他变体, 一个变量接受 `OutputStream`, 另一个变量接受 `Writer`。例如, 调用 `print(System.err)` 将结果数据打印到 `System.err`, 而调用 `print(myWriter)` 则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，则可以调用 `log()` 方法，该方法将结果数据记录为

`org.springframework.test.web.servlet.result` 记录类别下的一条 `DEBUG` 消息。

在某些情况下，您可能希望直接访问结果并验证否则无法验证的内容。可以通过在所有其他期望值之后附加 `.andReturn()` 来实现，如以下示例所示：

```
MvcResult mvcResult = mockMvc.perform(post("/persons")).andExpect(status().isOk()).andReturn()  
// ...
```

如果所有测试都重复相同的期望，则在构建 `MockMvc` 实例时可以一次设置共同的期望，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，通常会应用共同的期望，并且在不创建单独的 `MockMvc` 实例的情况下不能将其覆盖。

当 JSON 响应内容包含使用 [Spring HATEOAS](#) 创建的超媒体链接时，您可以使用 JsonPath 表达式来验证生成的链接，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，您可以使用 XPath 表达式来验证结果链接：

```
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 Registrations

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(...))
```

已注册的过滤器通过 `spring-test` 的 `MockFilterChain` 调用，最后一个过滤器委托给 `DispatcherServlet`。

## 容器外测试与端到端集成测试之间的区别

如前所述，Spring MVC Test 是基于 `spring-test` 模块的 Servlet API 模拟对象构建的，并且不使用正在运行的 Servlet 容器。因此，与在实际 Client 端和服务器上运行的完整端到端集成测试相比，存在一些重要差异。

考虑这一点的最简单方法是从空白 `MockHttpServletRequest` 开始。您添加到其中的内容就是请求的内容。可能令您感到惊讶的是，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 呈现。而是将“转发”和“重定向” URL 保存在 `MockHttpServletResponse` 中，并且可以按预期进行 assert。

这意味着，如果您使用 JSP，则可以验证将请求转发到的 JSP 页面，但不会呈现 HTML。换句话说，不调用 JSP。但是请注意，所有其他不依赖转发的呈现技术(例如 Thymeleaf 和 Freemarker)都可以按预期将 HTML 呈现到响应主体。通过 `@ResponseBody` 方法呈现 JSON, XML 和其他格式时也是如此。

或者，您可以考虑使用 `@WebIntegrationTest` 从 Spring Boot 获得完整的端到端集成测试支持。参见[Spring Boot 参考指南](#)。

每种方法都各有利弊。从经典的单元测试到全面的集成测试，Spring MVC Test 中提供的选项在规模上是不同的。可以肯定的是，Spring MVC Test 中的所有选项都不属于经典单元测试的类别，但与之接近。例如，您可以通过将模拟服务注入到控制器中来隔离 Web 层，在这种情况下，您仅通过 `DispatcherServlet` 来测试 Web 层，但使用实际的 Spring 配置，因为您可能会与数据访问层隔离地对其之上的层进行测试。此外，您可以使用独立设置，一次只关注一个控制器，然后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，此类测试是服务器端的，因此您可以检查使用了哪个处理器，如果使用 `HandlerExceptionResolver` 处理了异常，则模型的内容是什

么，绑定错误是什么？还有其他细节。这意味着编写期望值更容易，因为服务器不是一个黑盒子，就像通过实际的 `HttpClient` 端对其进行测试时那样。通常，这是经典单元测试的一个优势：编写、推理和调试更容易，但不能代替完全集成测试的需要。同时，重要的是不要忽视以下事实：响应是最重要的检查对象。简而言之，即使在同一项目中，这里也存在多种测试样式和测试策略的空间。

## 其他服务器端测试示例

框架自己的测试包括[许多 samples 测试](#)，旨在展示如何使用 Spring MVC Test。您可以浏览这些示例以获取进一步的想法。另外，[spring-mvc-showcase](#)项目具有基于 Spring MVC Test 的完整测试范围。

### 3.6.2. HtmlUnit 集成

Spring 提供了[MockMvc](#)和[HtmlUnit](#)之间的集成。使用基于 HTML 的视图时，这简化了执行端到端测试的过程。通过此集成，您可以：

- 使用[HtmlUnit](#), [WebDriver](#)和[Geb](#)之类的工具轻松测试 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。

#### Note

`MockMvc` 使用不依赖 Servlet 容器的模板技术(例如 Thymeleaf, FreeMarker 等)，但不适用于 JSP，因为它们依赖 Servlet 容器。

## 为什么要进行 HtmlUnit 集成？

想到的最明显的问题是“我为什么需要这个？”通过探索一个非常基本的示例应用程序，最好找到答案。假设您有一个 Spring MVC Web 应用程序，该应用程序支持对 `Message` 对象的 CRUD 操作

。该应用程序还支持所有消息的分页。您将如何进行测试？

使用 Spring MVC Test，我们可以轻松地测试是否能够创建 `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"));
```

如果我们要测试允许我们创建消息的表单视图怎么办？例如，假设我们的表单类似于以下代码段：

```
<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());
```

此测试有一些明显的缺点。如果我们更新控制器以使用参数 `message` 而不是 `text`，则即使 HTML 表单与控制器不同步，我们的表单测试也会 `continue` 通过。为了解决这个问题，我们可以结合以下两个测试：

```
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"));
```

这样可以减少测试不正确通过的风险，但是仍然存在一些问题：

- 如果页面上有多个表单怎么办？诚然，我们可以更新 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：字段是正确的类型吗？是否启用了字段？等等。
- 另一个问题是我们正在做我们期望的两倍的工作。我们必须首先验证视图，然后使用刚刚验证的相同参数提交视图。理想情况下，可以一次完成所有操作。
- 最后，我们仍然无法解释某些事情。例如，如果表单具有我们也希望测试的 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与其他资源交互的组合。例如，表单视图的结果用作用户创建消息的 Importing。另外，我们的表单视图可以潜在地使用影响页面行为的其他资源，例如 JavaScript 验证。

## 对救援进行集成测试？

为了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。考虑测试允许我们翻阅消息的视图。我们可能需要以下测试：

- 我们的页面是否向用户显示通知，以指示消息为空时没有可用结果？
- 我们的页面是否正确显示一条消息？
- 我们的页面是否正确支持分页？

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。这带来了许多其他挑战：

- 确保数据库中包含正确的消息可能很繁琐。（考虑外键约束。）
- 测试可能会变慢，因为每次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。
- 对诸如自动生成的 id，时间戳等项目进行 assert 可能很困难。

这些挑战并不意味着我们应该完全放弃端到端集成测试。相反，我们可以通过重构详细的测试以使运行速度更快，更可靠且没有副作用的模拟服务来减少端到端集成测试的数量。然后，我们可以实施少量 true 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。

## ImportingHtmlUnit 集成

那么，如何在测试页面的交互性之间保持平衡，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

要将 MockMvc 与 HtmlUnit 集成时，有很多选择：

- [MockMvc 和 HtmlUnit](#)：如果要使用原始的 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](#)：如果要使用 Groovy 进行测试，简化开发并在集成和端到端测试之间重用代码，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，请确保您已包含对 `net.sourceforge.htmlunit:htmlunit` 的测试依赖项。为了将 HtmlUnit 与 Apache HttpComponents 4.5 一起使用，您需要使用 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();
}
```

### iNote

这是使用 `MockMvcWebClientBuilder` 的简单示例。有关高级用法, 请参见[Advanced MockMvcWebClientBuilder](#)。

这样可以确保在服务器上引用 `localhost` 的所有 URL 都定向到我们的 `MockMvc` 实例, 而无需 true 的 HTTP 连接。通常, 通过使用网络连接来请求其他任何 URL。这使我们可以轻松测试 CDN 的使用。

## MockMvc 和 HtmlUnit 的用法

现在, 我们可以像往常一样使用 `HtmlUnit`, 而无需将应用程序部署到 `Servlet` 容器。例如, 我们可以请求视图创建以下消息:

```
HtmlPage createMsgFormPage = webClient.getPage("http://localhost/messages/form");
```

### iNote

默认上下文路径为 `" "`。或者, 我们可以指定上下文路径, 如[Advanced MockMvcWebClientBuilder](#)中所述。

一旦有了对 `HtmlPage` 的引用, 我们就可以填写表格并将其提交以创建一条消息, 如以下示例所示:

```
HtmlForm form = createMsgFormPage.getHtmlElementById("messageForm");
HtmlTextInput summaryInput = createMsgFormPage.getHtmlElementById("summary");
summaryInput.setValueAttribute("Spring Rocks");
HtmlTextArea textInput = createMsgFormPage.getHtmlElementById("text");
textInput.setText("In case you didn't know, Spring Rocks!");
HtmlSubmitInput submit = form.getOneHtmlElementByAttribute("input", "type", "submit");
HtmlPage newMessagePage = submit.click();
```

最后，我们可以验证是否成功创建了新消息。以下 assert 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!");
```

前面的代码通过多种方式对我们的[MockMvc test](#)进行了改进。首先，我们不再需要显式验证表单，然后创建类似于表单的请求。相反，我们要求表单，将其填写并提交，从而大大减少了开销。

另一个重要因素是[HtmlUnit 使用 Mozilla Rhino 引擎](#)评估 JavaScript。这意味着我们还可以在页面内测试 JavaScript 的行为。

有关使用 HtmlUnit 的其他信息，请参见[HtmlUnit documentation](#)。

## Advanced MockMvcWebClientBuilder

在到目前为止的示例中，我们已通过 Spring TestContext Framework 为我们加载的

`WebApplicationContext` 构建一个 `WebClient`，从而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();
}
```

另外，我们可以通过分别配置 `MockMvc` 实例并将其提供给 `MockMvcWebClientBuilder` 来执行完全相同的设计，如下所示：

```
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();
```

这比较冗长，但是通过使用 `MockMvc` 实例构建 `WebClient`，我们可以轻而易举地拥有 `MockMvc` 的全部功能。

### Tip

有关创建 `MockMvc` 实例的其他信息，请参见 [Setup Choices](#)。

## 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，我们在 `Selenium` [WebDriver](#) 中使用其他抽象使事情变得更加容易。

### 为什么要使用 WebDriver 和 MockMvc？

我们已经可以使用 `HtmlUnit` 和 `MockMvc`，那么为什么要使用 `WebDriver`? `Selenium` `WebDriver` 提供了一个非常优雅的 API，使我们可以轻松地组织代码。为了更好地显示其工作原理，我们在本

节中探索一个示例。

### iNote

尽管是[Selenium](#)的一部分，WebDriver 并不需要 Selenium Server 来运行测试。

假设我们需要确保正确创建一条消息。测试涉及找到 HTML 表单 Importing 元素，将其填写并做出各种 assert。

这种方法会导致大量单独的测试，因为我们也想测试错误情况。例如，如果只填写表格的一部分，我们要确保得到一个错误。如果我们填写整个表格，那么新创建的消息将在之后显示。

如果将其中一个字段命名为“summary”，则我们可能会在测试中的多个位置重复以下内容：

```
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？这样做将迫使我们更新所有测试以纳入此更改。这违反了 DRY 原理，因此理想情况下，我们应将此代码提取到其自己的方法中，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);
}
```

这样做可以确保我们在更改 UI 时不必更新所有测试。

我们甚至可以更进一步，并将此逻辑放在代表我们当前所在的 `HtmlPage` 的 `Object` 内，如以下示例所示：

```
public class CreateMessagePage {

    final HtmlPage currentPage;

    final HtmlTextInput summaryInput;
```

```

final HtmlSubmitInput submit;

public CreateMessagePage(HtmlPage currentPage) {
    this.currentPage = currentPage;
    this.summaryInput = currentPage.getHtmlElementById("summary");
    this.submit = currentPage.getHtmlElementById("submit");
}

public <T> T createMessage(String summary, String text) throws Exception {
    setSummary(summary);

    HtmlPage result = submit.click();
    boolean error = CreateMessagePage.at(result);

    return (T) (error ? new CreateMessagePage(result) : new ViewMessagePage(result));
}

public void setSummary(String summary) throws Exception {
    summaryInput.setValueAttribute(summary);
}

public static boolean at(HtmlPage page) {
    return "Create Message".equals(page.getTitleText());
}
}

```

以前，此模式称为[页面对象模式](#)。虽然我们当然可以使用 `HtmlUnit` 做到这一点，但 `WebDriver` 提供了一些我们在以下各节中探讨的工具，以使该模式的实现更加容易。

## MockMvc 和 WebDriver 设置

要将 Selenium WebDriver 与 Spring MVC Test 框架一起使用，请确保您的项目包含对

`org.seleniumhq.selenium:selenium-htmlunit-driver` 的测试依赖项。

我们可以使用 `MockMvcHtmlUnitDriverBuilder` 轻松创建一个与 MockMvc 集成的 Selenium WebDriver，如以下示例所示：

```

@Autowired
WebApplicationContext context;

WebDriver driver;

@Before
public void setup() {
    driver = MockMvcHtmlUnitDriverBuilder
        .webAppContextSetup(context)
        .build();
}

```

## iNote

这是使用 `MockMvcHtmlUnitDriverBuilder` 的简单示例。有关更多高级用法，请参见 [Advanced MockMvcHtmlUnitDriverBuilder](#)。

前面的示例确保在服务器上引用 `localhost` 的所有 URL 都定向到我们的 `MockMvc` 实例，而无需 `true` 的 HTTP 连接。通常，通过使用网络连接来请求其他任何 URL。这使我们可以轻松测试 CDN 的使用。

## MockMvc 和 WebDriver 的用法

现在，我们可以像往常一样使用 WebDriver，而无需将应用程序部署到 Servlet 容器。例如，我们可以请求视图创建以下消息：

```
CreateMessagePage page = CreateMessagePage.to(driver);
```

然后，我们可以填写表格并提交以创建一条消息，如下所示：

```
ViewMessagePage viewMessagePage =
    page.createMessage(ViewMessagePage.class, expectedSummary, expectedText);
```

利用 Page Object Pattern，这可以改善 [HtmlUnit test](#) 的设计。正如我们在 [为什么要使用 WebDriver 和 MockMvc?](#) 中提到的，我们可以将 Page Object Pattern 与 HtmlUnit 一起使用，但使用 WebDriver 则容易得多。考虑以下 `CreateMessagePage` 实现：

```
public class CreateMessagePage
    extends AbstractPage { (1)

    (2)
    private WebElement summary;
    private WebElement text;

    (3)
    @FindBy(css = "input[type=submit]")
    private WebElement submit;

    public CreateMessagePage(WebDriver driver) {
        super(driver);
    }
```

```

public <T> T createMessage(Class<T> resultPage, String summary, String details) {
    this.summary.sendKeys(summary);
    this.text.sendKeys(details);
    this.submit.click();
    return PageFactory.initElements(driver, resultPage);
}

public static CreateMessagePage to(WebDriver driver) {
    driver.get("http://localhost:9990/mail/messages/form");
    return PageFactory.initElements(driver, CreateMessagePage.class);
}
}

```

- (1) `CreateMessagePage` 扩展了 `AbstractPage`。我们不会详细介绍 `AbstractPage`，但是总而言之，它包含所有页面的通用功能。例如，如果我们的应用程序具有导航栏，全局错误消息和其他功能，则可以将此逻辑放置在共享位置。
- (2) 我们对感兴趣的 HTML 页面的每个部分都有一个成员变量。这些类型为 `WebElement`。

WebDriver 的 `PageFactory` 通过自动解析每个 `WebElement`，使我们从 `CreateMessagePage` 的 HtmlUnit 版本中删除了大量的代码。`PageFactory#initElements(WebDriver,Class<T>)` 方法通过使用字段名称并通过 HTML 页面中元素的 `id` 或 `name` 查找来自动解析每个 `WebElement`。

- (3) 我们可以使用 `@FindBy annotation` 覆盖默认的查找行为。我们的示例说明了如何使用 `@FindBy` 注解通过 `css` 选择器(`input [type = submit]`)查找提交按钮。

最后，我们可以验证是否成功创建了新消息。以下 assert 使用 `AssertJ` 库：

```

assertThat(viewMessagePage.getMessage()).isEqualTo(expectedMessage);
assertThat(viewMessagePage.getSuccess()).isEqualTo("Successfully created a new message")

```

我们可以看到 `ViewMessagePage` 使我们能够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，我们可以在声明中使用富域对象。

最后，我们一定不要忘记在测试完成后关闭 `WebDriver` 实例，如下所示：

```
@After
public void destroy() {
    if (driver != null) {
        driver.close();
    }
}
```

有关使用 `WebDriver` 的其他信息，请参见 [Selenium WebDriver documentation](#)。

## Advanced MockMvcHtmlUnitDriverBuilder

在到目前为止的示例中，我们已通过 `Spring TestContext Framework` 为我们加载的 `WebApplicationContext` 构建一个 `WebDriver`，从而以最简单的方式使用了 `MockMvcHtmlUnitDriverBuilder`。在此重复此方法，如下所示：

```
@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();
}
```

另外，我们可以通过分别配置 `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();
```

这比较冗长，但是通过使用 `MockMvc` 实例构建 `WebDriver`，我们可以轻而易举地拥有 `MockMvc` 的全部功能。

### Tip

有关创建 `MockMvc` 实例的其他信息，请参见 [Setup Choices](#)。

## MockMvc 和 Geb

在上一节中，我们了解了如何在 WebDriver 中使用 MockMvc。在本节中，我们使用 [Geb](#) 来使我们的测试甚至更接近 Groovy-er。

### 为什么选择 Geb 和 MockMvc？

Geb 由 WebDriver 支持，因此它提供了许多从 WebDriver 获得的 [same benefits](#)。但是，Geb 通常为我们处理一些样板代码使事情变得更加轻松。

### MockMvc 和 Geb 设置

我们可以轻松地使用使用 MockMvc 的 Selenium WebDriver 来初始化 Geb `Browser`，如下所示：

```
def setup() {
    browser.driver = MockMvcHtmlUnitDriverBuilder
        .webAppContextSetup(context)
        .build()
}
```

### iNote

这是使用 `MockMvcHtmlUnitDriverBuilder` 的简单示例。有关更多高级用法，请参见 [Advanced MockMvcHtmlUnitDriverBuilder](#)。

这样可以确保在服务器上引用 `localhost` 的所有 URL 都定向到我们的 `MockMvc` 实例，而无需 `true` 的 HTTP 连接。通常，通过使用网络连接来请求其他任何 URL。这使我们可以轻松测试 CDN 的使用。

## MockMvc 和 Geb 的使用

现在，我们可以像往常一样使用 Geb 了，而无需将应用程序部署到 Servlet 容器中。例如，我们可以请求视图创建以下消息：

```
to CreateMessagePage
```

然后，我们可以填写表格并提交以创建一条消息，如下所示：

```
when:
form.summary = expectedSummary
form.text = expectedMessage
submit.click(ViewMessagePage)
```

找不到的所有无法识别的方法调用或属性访问或引用都将转发到当前页面对象。这消除了我们直接使用 WebDriver 时需要的许多样板代码。

与直接使用 WebDriver 一样，这通过使用页面对象模式改进了 [HtmlUnit test](#) 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 则更加容易。考虑

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`，但是总而言之，它包含所有页面的通用功能。我们定义一个可在其中找到此页面的 URL。这使我们可以导航到页面，如下所示：

```
to CreateMessagePage
```

我们还有一个 `at` 闭包，它确定我们是否在指定的页面上。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 `assert` 我们在正确的页面上的原因，如下所示：

```
then:  
at CreateMessagePage  
errors.contains('This field is required.')
```

### iNote

我们在闭包中使用一个 `assert`，以便我们可以确定在错误的页面上哪里出错了。

接下来，我们创建一个 `content` 闭包，以指定页面内所有感兴趣的区域。我们可以使用 [jQuery-ish 导航器 API](#) 选择感兴趣的内容。

最后，我们可以验证是否成功创建了新消息，如下所示：

```
then:  
at ViewMessagePage  
success == 'Successfully created a new message'  
id  
date
```

```
summary == expectedSummary  
message == expectedMessage
```

有关如何充分利用 Geb 的更多详细信息, 请参见[奇书](#)用户手册。

### 3.6.3. Client 端 REST 测试

您可以使用 Client 端测试来测试内部使用 `RestTemplate` 的代码。这个想法是声明预期的请求并提供“存根”响应, 以便您可以专注于隔离测试代码(即, 不运行服务器)。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` (Client 端 REST 测试的中心类) 使用自定义 `ClientHttpRequestFactory` 配置 `RestTemplate`, 该自定义 `ClientHttpRequestFactory` `assert` 实际请求超出期望并返回“存根”响应。在这种情况下, 我们希望有一个对 `/greeting` 的请求, 并希望返回一个包含 `text/plain` 内容的 200 响应。我们可以根据需要定义其他预期的请求和存根响应。当我们定义期望的请求和存根响应时, `RestTemplate` 可以照常在 Client 端代码中使用。在测试结束时, 可以使用 `mockServer.verify()` 来验证是否满足所有期望。

默认情况下, 请求应按声明的期望 `Sequences` 进行。您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(以便)以找到给定请求的匹配项。这意味着允许请求以任何 `Sequences` 出现。以下示例使用 `ignoreExpectOrder`:

```
server = MockRestServiceServer.bindTo(restTemplate).ignoreExpectOrder(true).build();
```

即使默认情况下无 `Sequences` 请求, 每个请求也只能执行一次。`expect` 方法提供了一个重载的

变量，该变量接受一个 `ExpectedCount` 参数，该参数指定一个计数范围(例如 `once` , `manyTimes` , `max` , `min` , `between` 等)。以下示例使用 `times` :

```
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，当未设置 `ignoreExpectOrder` 时(默认设置)，因此，请求应按声明 `Sequences` 进行，则该 `Sequences` 仅适用于任何预期请求中的第一个。例如，如果期望 “/something”两次，然后是 “/somewhere”三次，那么在请求 “/somewhere”之前应该先请求 “/something”，但是除了随后的 “/” 东西” 和 “/某处”，请求可以随时发出。

作为上述所有方法的替代，Client 端测试支持还提供 `ClientHttpRequestFactory` 实现，您可以将其配置为 `RestTemplate` 以将其绑定到 `MockMvc` 实例。这样就可以使用实际的服务器端逻辑来处理请求，而无需运行服务器。以下示例显示了如何执行此操作：

```
MockMvc mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).build();
this.restTemplate = new RestTemplate(new MockMvcClientHttpRequestFactory(mockMvc));

// Test code that uses the above RestTemplate ...
```

## Static Imports

与服务器端测试一样，用于 Client 端测试的 FluentAPI 需要进行一些静态导入。通过搜索 `MockRest*` 可以轻松找到它们。Eclipse 用户应在 Java→编辑器→Content Assist→收藏夹下的 Eclipse 首选项中将 `MockRestRequestMatchers.*` 和 `MockRestResponseCreators.*` 作为“最喜欢的静态成员”添加。这样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。检查对静态成员的代码完成的支持。

## Client 端 REST 测试的更多示例

Spring MVC Test 自己的测试包括[example tests](#)个 Client 端 REST 测试。

## 3.7. WebTestClient

`WebTestClient` 是围绕[WebClient](#)的薄壳，用于执行请求并公开专用的 Fluent 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。

### Tip

Kotlin 用户：请参阅[this section](#)与 `WebTestClient` 的使用有关。

### 3.7.1. Setup

要创建 `WebTestClient`，您必须选择多个服务器设置选项之一。实际上，您是在配置要绑定到的 WebFlux 应用程序，还是使用 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();
```

前面的示例加载[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，可以在没有 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定制默认 WebFlux Java 配置。

#### 绑定到 Router 功能

以下示例显示了如何从[RouterFunction](#)设置服务器：

```
RouterFunction<?> route = ...
client = WebTestClient.bindToRouterFunction(route).build();
```

在内部，配置将传递到 `RouterFunctions.toWebHandler`。通过使用模拟请求和响应对象，可以在没有 HTTP 服务器的情况下测试生成的 WebFlux 应用程序。

## 绑定到 ApplicationContext

以下示例显示了如何从应用程序的 Spring 配置或其一部分来设置服务器：

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = WebConfig.class) (1)
public class MyTests {

    @Autowired
    private ApplicationContext context; (2)

    private WebTestClient client;

    @Before
    public void setUp() {
        client = WebTestClient.bindToApplicationContext(context).build(); (3)
    }
}
```

- (1) 指定要加载的配置
- (2) 注入配置
- (3) 创建 `WebTestClient`

在内部，配置将传递到 `WebHttpHandlerBuilder` 以构建请求处理链。有关更多详细信息，请参见 [WebHandler API](#)。通过使用模拟请求和响应对象，可以在没有 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使您可以连接到正在运行的服务器：

```
client = WebTestClient.bindToServer().baseUrl("http://localhost:8080").build();
```

## Client Builder

除了前面描述的服务器设置选项之外，您还可以配置 Client 端选项，包括基本 URL，默认 Headers，Client 端过滤器等。这些选项可在 `bindToServer` 之后使用。对于所有其他服务器，您需要使用

`configureClient()` 从服务器配置过渡到 Client 端配置，如下所示：

```
client = WebTestClient.bindToController(new TestController())
    .configureClient()
    .baseUrl("/test")
    .build();
```

### 3.7.2. 写作测试

`WebTestClient` 通过使用 `exchange()` 提供与 [WebClient](#) 相同的 API。`exchange()` 之后是链接的 API 工作流，用于验证响应。

通常，首先声明响应状态和 Headers，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，您指定如何解码和使用响应主体：

- `expectBody(Class<T>)`：解码为单个对象。
- `expectBodyList(Class<T>)`：将对象解码并收集到 `List<T>`。
- `expectBody()`：将 [JSON Content](#) 解码为 `byte[]` 或为空的正文。

然后，您可以为主体使用内置的 assert。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);
```

您还可以超越内置的 assert 并创建自己的 assert，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)...
});
```

您还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();
```

### Tip

当您需要使用泛型解码为目标类型时，请寻找接受[ParameterizedTypeReference](#)而不是  
`Class<T>`的重载方法。

## No Content

如果响应中没有内容(或者您不在乎)，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 `assert` 没有响应内容，则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，响应以 `byte[]` 的形式使用。这对于原始内容声明很有用。例如，您可以使用[JSONAssert](#)来验证 JSON 内容，如下所示：

```
client.get().uri("/persons/1")
    .exchange()
    .expectStatus().isOk()
    .expectBody()
    .json("{ \"name\": \"Jane\" }")
```

您还可以使用[JSONPath](#)表达式，如下所示：

```
client.get().uri("/persons")
    .exchange()
    .expectStatus().isOk()
    .expectBody()
    .jsonPath("$.name").isEqualTo("Jane")
    .jsonPath("$.name").isEqualTo("Jason");
```

## Streaming Responses

要测试无限流(例如 `"text/event-stream"` 或 `"application/stream+json"`)，需要在响应状态和 Headers 声明之后立即退出链接的 API(通过使用 `returnResult`)，如以下示例所示：

```
FluxExchangeResult<MyEvent> result = client.get().uri("/events")
    .accept(TEXT_EVENT_STREAM)
    .exchange()
    .expectStatus().isOk()
    .returnResult(MyEvent.class);
```

现在，您可以使用 `Flux<T>`，在它们到达时 `assert` 已解码的对象，然后在达到测试目标时在某个时候取消。我们建议使用 `reactor-test` 模块中的 `StepVerifier` 进行此操作，如以下示例所示：

```
Flux<Event> eventFux = result.getResponseBody();

StepVerifier.create(eventFux)
    .expectNext(person)
    .expectNextCount(4)
    .consumeNextWith(p -> ...)
    .thenCancel()
    .verify();
```

## Request Body

当涉及到构建请求时，`WebTestClient` 提供了与 `WebClient` 相同的 API，实现主要是简单的传

递。有关如何准备带有正文的请求的示例，请参见[WebClient documentation](#)，包括提交表单数据，分段请求等。

## 3.8. PetClinic 示例

[GitHub](#) 上提供的 PetClinic 应用程序显示了 JUnit 4 环境中 Spring TestContext Framework 的几个功能。大多数测试功能包含在 `AbstractClinicTests` 中，下面列出了部分功能：

```
import static org.junit.Assert.assertEquals;
// import ...

@ContextConfiguration (1)
public abstract class AbstractClinicTests extends AbstractTransactionalJUnit4SpringContextTests {

    @Autowired (3)
    protected Clinic clinic;

    @Test
    public void getVets() {
        Collection<Vet> vets = this.clinic.getVets();
        assertEquals("JDBC query must show the same number of vets",
                    super.countRowsInTable("VETS"), vets.size()); (4)
        Vet v1 = EntityUtils.getById(vets, Vet.class, 2);
        assertEquals("Leary", v1.getLastName());
        assertEquals(1, v1.getNrOfSpecialties());
        assertEquals("radiology", (v1.getSpecialties().get(0)).getName());
        // ...
    }

    // ...
}
```

- (1) 从默认位置 `AbstractClinicTests-context.xml` 加载应用程序上下文。
- (2) 此测试用例扩展了 `AbstractTransactionalJUnit4SpringContextTests` 类，从该类继承了 Dependency Injection(通过 `DependencyInjectionTestExecutionListener`)和事务行为(通过 `TransactionalTestExecutionListener`)的配置。
- (3) 依赖注入通过 `@Autowired` 语义设置了 `clinic` 实例变量(正在测试的应用程序对象)。
- (4) `getVets()` 方法说明了如何使用继承的 `countRowsInTable()` 方法来轻松验证给定表中的行数，从而验证所测试的应用程序代码的正确行为。这样可以进行更强大的测试，并减少对精确测试数据的依赖。例如，您可以在数据库中添加其他行而不破坏测试。

像许多使用数据库的集成测试一样，`AbstractClinicTests` 中的大多数测试取决于测试案例运行之前数据库中已存在的最小数据量。或者，您可以在测试用例设置的测试夹具中填充数据库(同样，在与测试相同的事务中)。

PetClinic 应用程序支持三种数据访问技术：JDBC，Hibernate 和 JPA。pass 语句

`@ContextConfiguration` 没有任何特定的资源位置，`AbstractClinicTests` 类将从默认位置 `AbstractClinicTests-context.xml` 加载其应用程序上下文，该默认位置声明一个公共 `DataSource`。子类指定必须声明 `PlatformTransactionManager` 和 `Clinic` 的具体实现的其他上下文位置。

例如，PetClinic 测试的 Hibernate 实现包含以下实现。对于此示例，`HibernateClinicTests` 不包含一行代码。我们只需要声明 `@ContextConfiguration`，并且测试是从

`AbstractClinicTests` 继承的。因为声明 `@ContextConfiguration` 时没有任何特定的资源位置，所以 Spring TestContext Framework 从 `AbstractClinicTests-context.xml` 中定义的所有 bean(即继承位置)和 `HibernateClinicTests-context.xml` 加载应用程序上下文，而 `HibernateClinicTests-context.xml` 可能会覆盖 `AbstractClinicTests-context.xml` 中定义的 bean。以下 Lists 显示了 `HibernateClinicTests` 类的定义：

```
@ContextConfiguration (1)
public class HibernateClinicTests extends AbstractClinicTests { }
```

- (1) 从 `AbstractClinicTests-context.xml` 和 `HibernateClinicTests-context.xml` 加载应用程序上下文。

在大型应用程序中，Spring 配置通常分为多个文件。因此，通常在所有特定于应用程序的集成测试的通用 Base Class 中指定配置位置。这样的 Base Class 还可以添加有用的实例变量(自然而然地由 Dependency Injection 填充)，例如在使用 Hibernate 的应用程序中为 `SessionFactory`。

集成测试中应尽可能具有与部署环境中完全相同的 Spring 配置文件。一个可能的不同点是数据库连接池和事务基础结构。如果要部署到功能完善的应用程序服务器，则可能使用其连接池(可通过

JNDI 获得)和 JTA 实现。因此，在 Producing，可以将 `JndiObjectFactoryBean` 或 `<jee:jndi-lookup>` 用作 `DataSource` 和 `JtaTransactionManager`。JNDI 和 JTA 在容器外集成测试中不可用，因此您应使用 Commons DBCP `BasicDataSource` 和 `DataSourceTransactionManager` 或 `HibernateTransactionManager` 之类的组合。您可以将此变异行为分解为一个 XML 文件，可以在应用程序服务器和“本地”配置之间进行选择，该配置与所有其他配置分开，在测试和生产环境之间不会有所不同。另外，我们建议您使用属性文件进行连接设置。有关示例，请参见 PetClinic 应用程序。

## 4. 其他资源

---

有关测试的更多信息，请参见以下资源：

- [JUnit](#): “面向 Java 的面向程序员的测试框架”。由 Spring Framework 在其测试套件中使用。
- [TestNG](#): 受 JUnit 启发的测试框架，它对 Comments，测试组，数据驱动的测试，分布式测试和其他功能提供了额外的支持。
- [AssertJ](#): “Java 的 assert”，包括对 Java 8 lambda，流和其他功能的支持。
- [Mock Objects](#): 维基百科中的文章。
- [MockObjects.com](#): 专门用于模拟对象的网站，一种用于在测试驱动的开发中改进代码设计的技术。
- [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来提供模拟对象(以及通过类扩展的对象)。”由 Spring Framework 在其测试套件中使用。
- [JMMock](#): 支持使用模拟对象进行 Java 代码的测试驱动开发的库。
- [DbUnit](#): JUnit 扩展(也可与 Ant 和 Maven 一起使用)，针对数据库驱动的项目，除其他外，它使数据库在测试运行之间进入已知状态。
- [The Grinder](#): Java 负载测试框架。

# Data Access

---

参考文档的这一部分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。

详细介绍 Spring 全面的事务 Management 支持，然后全面介绍了 Spring 框架所集成的各种数据访问框架和技术。

## 1. TransactionManagement

---

全面的事务支持是使用 Spring Framework 的最令人信服的原因。Spring 框架为事务 Management 提供了一致的抽象，具有以下优点：

- 跨不同事务 API(例如 Java 事务 API(JTA), JDBC, Hibernate 和 Java Persistence API(JPA))的一致编程模型。
- 支持声明式 [TransactionManagement](#)。
- [programmatic](#) 事务 Management 的 API 比 JTA 等复杂的事务 API 更简单。
- 与 Spring 的数据访问抽象的出色集成。

以下各节描述了 Spring 框架的事务功能和技术：

- [Spring 框架的事务支持模型的优点](#) 描述了为什么您将使用 Spring Framework 的事务抽象而不是 EJB 容器 Management 的事务(CMT)或选择通过专有 API(例如 Hibernate)驱动本地事务的原因。
- [了解 Spring Framework 事务抽象](#) 概述了核心类，并描述了如何从各种来源配置和获取 [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 TransactionManagement](#) 描述了对声明式事务 Management 的支持。
- [程序化 TransactionManagement](#) 涵盖了对程序化(即显式编码)事务 Management 的支持。

- [Transaction 绑定事件](#)描述了如何在事务中使用应用程序事件。

(本章还讨论了最佳做法[应用服务器集成](#)和[常见问题的解决方案](#)。)

## 1.1. Spring 框架的事务支持模型的优点

传统上，Java EE 开发人员在事务 Management 中有两种选择：全局或本地事务，这两者都有很大的局限性。下两节将回顾全局和本地事务 Management，然后讨论 Spring 框架的事务 Management 支持如何解决全局和本地事务模型的局限性。

### 1.1.1. Global Transaction

全局事务使您可以使用多个事务资源，通常是关系数据库和消息队列。应用服务器通过 JTA Management 全局事务，该 JTA 是繁琐的 API(部分是由于其异常模型)。此外，通常需要从 JNDI 派生 JTA [UserTransaction](#)，这意味着您还需要使用 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 Management 的事务)。CMT 是声明式事务 Management 的一种形式(与程序性事务 Management 不同)。EJB CMT 消除了与事务相关的 JNDI 查找的需要，尽管使用 EJB 本身必须使用 JNDI。它消除了编写 Java 代码来控制事务的大部分(但不是全部)需求。重大缺点是 CMT 与 JTA 和应用程序服务器环境相关联。而且，仅当选择在 EJB 中(或至少在事务性 EJB 幕后之后)实现业务逻辑时，此功能才可用。通常，EJB 的缺点很大，以至于这不是一个有吸引力的主张，尤其是面对声明式事务 Management 的强制选择时。

### 1.1.2. 本地 Transaction

本地事务是特定于资源的，例如与 JDBC 连接关联的事务。本地事务可能更易于使用，但有一个明显的缺点：它们不能跨多个事务资源工作。例如，使用 JDBC 连接 Management 事务的代码不能在全局 JTA 事务中运行。由于应用程序服务器不参与事务 Management，因此它无法帮助确保多个资源之间的正确性。(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侵入了编程模型。

### 1.1.3. Spring 框架的一致编程模型

Spring 解决了 Global 和本地 Transaction 的弊端。它使应用程序开发人员可以在任何环境中使用一致的编程模型。您只需编写一次代码，即可从不同环境中的不同事务 Management 策略中受益。Spring 框架提供了声明式和程序化事务 Management。大多数用户喜欢声明式事务 Management，在大多数情况下我们建议这样做。

通过程序化事务 Management，开发人员可以使用 Spring Framework 事务抽象，该抽象可以在任何基础事务基础架构上运行。使用首选的声明性模型，开发人员通常很少或几乎不编写与事务 Management 相关的代码，因此，它们不依赖于 Spring Framework 事务 API 或任何其他事务 API。

您需要用于事务 Management 的应用服务器吗？

Spring Framework 的事务 Management 支持更改了有关企业 Java 应用程序何时需要应用程序服务器的传统规则。

特别是，您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，即使您的应用服务器具有强大的 JTA 功能，您也可以决定 Spring 框架的声明式事务比 EJB CMT 提供更多的功能和更高效的编程模型。

通常，仅当您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，而这并不是许多应用程序所必需的。许多高端应用程序使用单个高度可扩展的数据库(例如 Oracle RAC)来代替。独立事务 Management 器(例如[Atomikos Transactions](#)和[JOTM](#))是其他选项。当然，您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 使您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。不再使用 EJB CMT 或 JTA 的唯一选择是使用本地事务(例如 JDBC 连接上的事务)编写代码，并且如果您需要该代码在全局的，容器 Management 的事务中运行，则面临大量的返工。使用 Spring Framework，仅需要更改配置文件中的某些 Bean 定义(而不是代码)。

## 1.2. 了解 Spring 框架事务抽象

Spring 事务抽象的关键是事务策略的概念。

`org.springframework.transaction.PlatformTransactionManager` 界面定义了一种

Transaction 策略，以下 Lists 显示了该策略：

```
public interface PlatformTransactionManager {  
    TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException;  
    void commit(TransactionStatus status) throws TransactionException;  
    void rollback(TransactionStatus status) throws TransactionException;  
}
```

尽管您可以从应用程序代码中使用 [programmatically](#)，但它主要是服务提供商接口(SPI)。由于 `PlatformTransactionManager` 是一个接口，因此可以根据需要轻松对其进行模拟或存根。它与诸如 JNDI 之类的查找策略无关。`PlatformTransactionManager` 实现的定义与 Spring Framework IoC 容器中的任何其他对象(或 bean)一样。单独使用此好处，即使使用 JTA，Spring 框架事务也成为有价值的抽象。与直接使用 JTA 相比，您可以更轻松地测试事务代码。

同样，根据 Spring 的哲学，未选中可由 `PlatformTransactionManager` 接口的任何方法抛出的 `TransactionException` (即，它扩展了 `java.lang.RuntimeException` 类)。Transaction 基础架构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`。突出的一点是，不强制开发人员这样做。

`getTransaction(...)` 方法根据 `TransactionDefinition` 参数返回 `TransactionStatus` 对象。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：

- 传播：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，如果在已存在事务上下文的情况下执行事务方法，则可以指定行为。例如，代码可以在现有事务中 `continue` 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 熟悉的所有事务传播选项。要了解有关 Spring 中事务传播的语义，请参阅[Transaction Propagation](#)。

- 隔离度：此事务与其他事务的工作隔离的程度。例如，此事务能否看到其他事务未提交的写入？
- 超时：超时之前该事务运行了多长时间，并被基础事务基础结构自动回滚。
- 只读状态：当代码读取但不修改数据时，可以使用只读事务。在某些情况下，例如使用 Hibernate 时，只读事务可能是有用的优化。

这些设置反映了标准的 `Transaction` 概念。如有必要，请参考讨论事务隔离级别和其他核心事务概念的资源。了解这些概念对于使用 Spring Framework 或任何事务 Management 解决方案至关重要。

`TransactionStatus` 界面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

概念应该很熟悉，因为它们对于所有事务 API 都是通用的。以下 Lists 显示了

`TransactionStatus` 界面：

```
public interface TransactionStatus extends SavepointManager {
    boolean isNewTransaction();
    boolean hasSavepoint();
    void setRollbackOnly();
    boolean isRollbackOnly();
    void flush();
    boolean isCompleted();
}
```

无论您在 Spring 中选择声明式还是程序化事务 Management，定义正确的

`PlatformTransactionManager` 实现都是绝对必要的。通常，您可以通过依赖注入来定义此实现

。

`PlatformTransactionManager` 实现通常需要了解其工作环境：JDBC、JTA、Hibernate 等。以

下示例显示了如何定义本地 `PlatformTransactionManager` 实现(在这种情况下，使用纯 JDBC)。

您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="c
    <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 定义引用了 `DataSource` 定义。它应类似于以下示例：

```
<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionMa
    <property name="dataSource" ref="dataSource"/>
</bean>
```

如果在 Java EE 容器中使用 JTA，则将通过 JNDI 获得的 `DataSource` 容器与 Spring 的 `JtaTransactionManager` 一起使用。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="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/jee
           http://www.springframework.org/schema/jee/spring-jee.xsd">

    <jee:jndi-lookup id="dataSource" jndi-name="jdbc/jpetstore"/>

    <bean id="txManager" class="org.springframework.transaction.jta.JtaTransactionManag
        <!-- other <bean/> definitions here -->
    </beans>
```

`JtaTransactionManager` 不需要了解 `DataSource`（或任何其他特定资源），因为它使用了容器的全局事务 Management 基础结构。

### iNote

`dataSource` bean 的先前定义使用 `jee` 名称空间中的 `<jndi-lookup/>` 标记。有关更多信息，请参见 [JEE 模式](#)。

您还可以轻松使用 Hibernate 本地事务，如以下示例所示。在这种情况下，您需要定义一个 Hibernate `LocalSessionFactoryBean`，您的应用程序代码可以使用它来获取 Hibernate `Session` 实例。

`DataSource` bean 定义与先前显示的本地 JDBC 示例相似，因此在以下示例中未显示。

### 1 Note

如果 `DataSource` (由任何非 JTA 事务 Management 器使用)通过 JNDI 查找并由 Java EE 容器 Management，则它应该是非事务性的，因为 Spring 框架(而不是 Java EE 容器) Management 事务。

在这种情况下，`txManager` bean 是 `HibernateTransactionManager` 类型。就像 `DataSourceTransactionManager` 需要引用 `DataSource` 一样，`HibernateTransactionManager` 也需要引用 `SessionFactory`。以下示例声明 `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>
```

如果您使用 Hibernate 和 Java EE 容器 Management 的 JTA 事务，则应使用与前面的 JDBC JTA 示例相同的 `JtaTransactionManager`，如以下示例所示：

```
<bean id="txManager" class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

### iNote

如果您使用 JTA，则无论使用哪种数据访问技术(无论是 JDBC, Hibernate JPA 或任何其他受支持的技术)，事务 Management 器定义都应该看起来相同。这是由于 JTA 事务是全局事务，它可以征用任何事务资源。

在所有这些情况下，无需更改应用程序代码。您可以仅通过更改配置来更改事务的 Management 方式，即使更改意味着从本地事务转移到全局事务，反之亦然。

## 1.3. 将资源与事务同步

现在应该清楚如何创建不同的事务 Management 器，以及如何将它们链接到需要同步到事务的相关资源(例如，`DataSourceTransactionManager` 到 JDBC `DataSource`，`HibernateTransactionManager` 到 Hibernate `SessionFactory` 等等)。本节描述应用程序代码(通过使用诸如 JDBC, Hibernate 或 JPA 之类的持久性 API 直接或间接)确保如何正确创建，重用和清理这些资源。本节还讨论了如何通过相关的 `PlatformTransactionManager` (可选)触发事务同步。

### 1.3.1. 高级同步方法

首选方法是使用 Spring 基于最高级别模板的持久性集成 API 或将本机 ORM API 与具有事务感知功能的工厂 bean 或代理一起使用，以 Management 本机资源工厂。这些支持事务的解决方案在内部处理资源的创建和重用，清理，资源的可选事务同步以及异常 Map。因此，用户数据访问代码不必解决这些任务，而可以完全专注于非样板持久性逻辑。通常，您使用本机 ORM API 或通过使用 `JdbcTemplate` 采取模板方法进行 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` (对于 JDBC), `EntityManagerFactoryUtils` (对于 JPA),  
`SessionFactoryUtils` (对于 Hibernate) 等类存在于较低级别。当您希望应用程序代码直接处理本机持久性 API 的资源类型时, 可以使用这些类来确保获得正确的 Spring FrameworkManagement 的实例, 事务(可选)同步, 并且流程中发生的异常是正确 Map 到一致的 API。

例如, 在 JDBC 的情况下, 可以使用 Spring 的

`org.springframework.jdbc.datasource.DataSourceUtils` 类代替传统的 JDBC 方法, 即在 `DataSource` 上调用 `getConnection()` 方法, 如下所示:

```
Connection conn = DataSourceUtils.getConnection(dataSource);
```

如果现有事务已经有与其同步(链接)的连接, 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)同步到任何现有事务, 并可供该同一事务中的后续重用使用。如前所述, 任何

`SQLException` 都包装在 Spring Framework `CannotGetJdbcConnectionException` 中, Spring Framework `CannotGetJdbcConnectionException` 是未经检查的

`DataAccessException` 类型的层次结构之一。与从 `SQLException` 轻松获得的信息相比, 这种方法为您提供的信息更多, 并确保了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 Management 的情况下也可以使用(事务同步是可选的), 因此无论是否使用 Spring 进行事务 Management, 都可以使用它。

当然, 一旦使用了 Spring 的 JDBC 支持, JPA 支持或 Hibernate 支持, 您通常不希望不使用 `DataSourceUtils` 或其他帮助程序类, 因为与直接使用相关的 API 相比, 通过 Spring 抽象进行工作会更快乐。例如, 如果您使用 Spring `JdbcTemplate` 或 `jdbc.object` 包来简化 JDBC 的使用, 则正确的连接检索将在后台进行, 并且您无需编写任何特殊代码。

### 1.3.3. TransactionAwareDataSourceProxy

最低级别是 `TransactionAwareDataSourceProxy` 类。这是目标 `DataSource` 的代理, 该代理包

装了目标 `DataSource` 以增加对 SpringManagement 的事务的了解。在这方面，它类似于 Java EE 服务器提供的事务性 JNDI `DataSource`。

您几乎永远不需要或不想使用此类，除非必须调用现有代码并通过标准的 JDBC `DataSource` 接口实现。在这种情况下，该代码可能可用，但参与了 SpringManagement 的事务。您可以使用前面提到的高级抽象来编写新代码。

## 1.4. 声明式 TransactionManagement

### iNote

大多数 Spring Framework 用户选择声明式事务 Management。此选项对应用程序代码的影响最小，因此与无创轻量级容器的 IDEA 最一致。

Spring 面向方面的编程(AOP)使 Spring 框架的声明式事务 Management 成为可能。但是，由于事务方面的代码随 Spring 框架发行版一起提供并且可以以样板方式使用，因此通常不必理解 AOP 概念即可有效地使用此代码。

Spring 框架的声明式事务 Management 与 EJB CMT 相似，因为您可以指定事务行为(或缺少事务行为)，直至单个方法级别。如有必要，您可以在事务上下文中进行 `setRollbackOnly()` 调用。两种类型的事务 Management 之间的区别是：

- 与绑定到 JTA 的 EJB CMT 不同，Spring 框架的声明式事务 Management 可在任何环境中工作。它可以通过使用 JDBC, JPA 或 Hibernate 通过调整配置文件来处理 JTA 事务或本地事务。
- 您可以将 Spring Framework 声明式事务 Management 应用于任何类，而不仅限于 EJB 之类的特殊类。
- Spring 框架提供了声明性的 [rollback rules](#)，此功能没有 EJB 等效项。提供了对回滚规则的编程和声明性支持。
- Spring Framework 允许您使用 AOP 自定义事务行为。例如，在事务回滚的情况下，您可以插入自定义行为。您还可以添加任意建议以及事务建议。使用 EJB CMT，除了

`setRollbackOnly()` 之外，您不能影响容器的事务 Management。

- Spring 框架不像高端应用程序服务器那样支持跨远程调用传播事务上下文。如果需要此功能，建议您使用 EJB。但是，在使用这种功能之前，请仔细考虑，因为通常情况下，您不希望事务跨越远程调用。

TransactionProxyFactoryBean 在哪里？

Spring 2.0 及更高版本中的声明式事务配置与 Spring 的早期版本有很大不同。主要区别在于不再需要配置 `TransactionProxyFactoryBean` bean。

Spring 2.0 之前的配置样式仍然是 100% 有效的配置。将新的 `<tx:tags/>` 视为代表您定义的 `TransactionProxyFactoryBean` bean。

回滚规则的概念很重要。他们让您指定哪些异常(和可抛出对象)应引起自动回滚。您可以在配置中而不是在 Java 代码中声明性地指定。因此，尽管您仍然可以在 `TransactionStatus` 对象上调用 `setRollbackOnly()` 来回滚当前事务，但大多数情况下，您可以指定

`MyApplicationException` 必须始终导致回滚的规则。此选项的主要优点是业务对象不依赖于事务基础结构。例如，他们通常不需要导入 Spring 事务 API 或其他 Spring API。

尽管 EJB 容器的默认行为会在系统异常(通常是运行时异常)时自动回滚事务，但是 EJB CMT 不会在应用程序异常(即 `java.rmi.RemoteException` 以外的已检查异常)时自动回滚事务。尽管 Spring 声明式事务 Management 的默认行为遵循 EJB 约定(仅针对未检查的异常会自动回滚)，但自定义此行为通常很有用。

### 1.4.1. 了解 Spring 框架的声明式事务实现

仅仅告诉您使用 `@Transactional` 对类进行注解，将

`@EnableTransactionManagement` 添加到您的配置中，并希望您了解其全部工作原理是不够的。

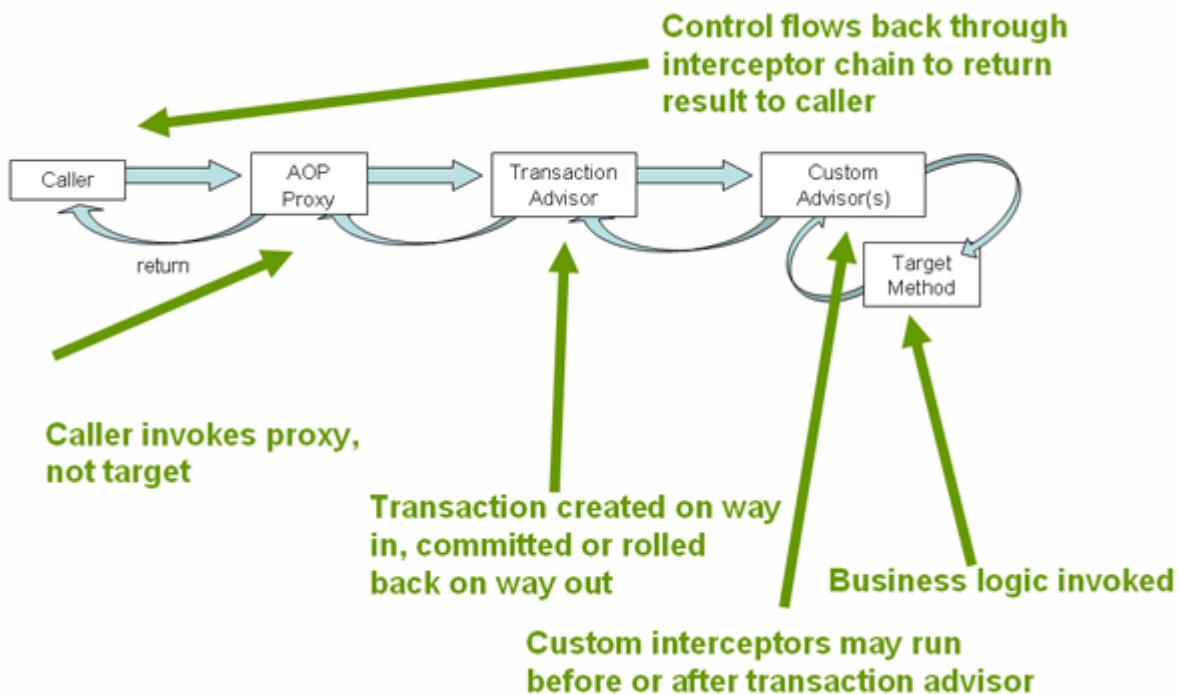
为了提供更深入的理解，本节介绍了在发生与事务相关的问题时，Spring 框架的声明式事务基础结构的内部工作方式。

关于 Spring Framework 的声明式事务支持，要把握的最重要的概念是启用此支持通过 AOP 代理， 并且事务通知由元数据(当前基于 XML 或基于 Comments)驱动。 AOP 与事务元数据的结合产生了一个 AOP 代理，该代理使用 `TransactionInterceptor` 结合适当的 `PlatformTransactionManager` 实现来驱动方法调用周围的事务。

### ①Note

Spring AOP 包含在[AOP 部分](#)中。

下图显示了在事务代理上调用方法的概念视图：



## 1.4.2. 声明式事务实现示例

考虑以下接口及其附带的实现。本示例使用 `Foo` 和 `Bar` 类作为占位符，以便您可以专注于事务使用而不关注特定的域模型。就本示例而言，`DefaultFooService` 类在每个已实现方法的主体中引发 `UnsupportedOperationException` 实例的事实是很好的。该行为使您可以看到已创建事务，然后响应 `UnsupportedOperationException` 实例而回滚了事务。以下 Lists 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();
    }

}
```

假定 `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上下文中执行，而其他方法 `insertFoo(Foo)` 和 `updateFoo(Foo)` 必须在具有 `read`-语义的事务上下文中执行写语义。以下几节将详细说明以下配置：

```
<!-- from the file 'context.xml' -->
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
```

```

http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd">

<!-- this is the service object that we want to make transactional -->
<bean id="fooService" class="x.y.service.DefaultFooService"/>

<!-- 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。它假定您要使服务对象 `fooService` bean 成为事务性的。要应用的事务语义封装在 `<tx:advice/>` 定义中。 `<tx:advice/>` 的定义为“从 `get` 开始的所有方法都将在只读事务的上下文中执行，而所有其他方法将以默认的事务语义执行”。`<tx:advice/>` 标记的 `transaction-manager` 属性设置为将要驱动事务的 `PlatformTransactionManager` bean 的名称(在本例中为 `txManager` bean)。

## Tip

如果要连接的 `PlatformTransactionManager` 的 bean 名称具有 `transactionManager`，则可以在事务通知(`<tx:advice/>`)中省略 `transaction-manager` 属性。如果要连接的 `PlatformTransactionManager` bean 具有其他名称，则必须如上例所示显式使用 `transaction-manager` 属性。

`<aop:config/>` 定义可确保 `txAdvice` bean 定义的事务建议在程序的适当位置执行。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，使用顾问将切入点与 `txAdvice` 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，请参见[AOP 部分](#)。

一个普遍的要求是使整个服务层具有事务性。最好的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>
```

## Note

在前面的示例中，假定您的所有服务接口都在 `x.y.service` 包中定义。有关更多详细信息，请参见[AOP 部分](#)。

现在我们已经分析了配置，您可能会问自己：“所有这些配置实际上是做什么的？”

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。代理配置有事务建议，以便在代理上调用适当的方法时，根据与该方法相关联的事务配置，事务将被启动，挂起，标

记为只读等。考虑下面的程序，该程序测试驱动前面显示的配置：

```
public final class Boot {  
  
    public static void main(final String[] args) throws Exception {  
        ApplicationContext ctx = new ClassPathXmlApplicationContext("context.xml", Boot.class);  
        FooService fooService = (FooService) ctx.getBean("fooService");  
        fooService.insertFoo(new Foo());  
    }  
}
```

运行先前程序的输出应类似于以下内容(为清晰起见，Log4J 输出和由 DefaultFooService 类的 insertFoo(..)方法抛出的 UnsupportedOperationException 的堆栈跟踪已被截断)：

```
<!-- the Spring container is starting up... -->  
[AspectJInvocationContextExposingAdvisorAutoProxyCreator] - Creating implicit proxy for  
  
<!-- 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 [[emailprotected]] for JDBC transaction  
  
<!-- the insertFoo(..) method from DefaultFooService throws an exception... -->  
[RuleBasedTransactionAttribute] - Applying rules to determine whether transaction should roll back  
[TransactionInterceptor] - Invoking rollback for transaction on x.y.service.FooService.insertFoo  
  
<!-- and the transaction is rolled back (by default, RuntimeException instances cause rollback by default) -->  
[DataSourceTransactionManager] - Rolling back JDBC transaction on Connection [[emailprotected]]  
[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()  
<!-- AOP infrastructure stack trace elements removed for clarity -->  
at $Proxy0.insertFoo(Unknown Source)  
at Boot.main(Boot.java:11)
```

### 1.4.3. 回滚声明式事务

上一节概述了如何在应用程序中声明性地指定类(通常是服务层类)的事务设置的基础。本节介绍如何以一种简单的声明性方式控制事务的回滚。

向 Spring Framework 的事务基础结构指示要回滚事务的推荐方法是从事务上下文中当前正在执行的代码中抛出 `Exception`。Spring 框架的事务基础结构代码会捕获未处理的 `Exception`，因为它会使调用栈冒泡，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将事务标记为回滚。也就是说，当抛出的异常是 `RuntimeException` 的实例或子类时。（默认情况下，`Error` 个实例也会导致回滚）。从事务方法引发的检查异常不会导致默认配置中的回滚。

您可以准确配置哪些 `Exception` 类型将事务标记为回滚，包括已检查的异常。以下 XML 代码段演示了如何为选中的，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>
```

如果您不希望在引发异常时回滚事务，则还可以指定“无回滚规则”。下面的示例告诉 Spring 框架的事务基础结构，即使面对未处理的 `InstrumentNotFoundException`，也要提交伴随的事务：

```
<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到异常并咨询已配置的回滚规则以确定是否将事务标记为回滚时，最强的匹配规则获胜。因此，在以下配置的情况下，除

`InstrumentNotFoundException` 之外的任何异常都会导致附带事务的回滚：

```
<tx:advice id="txAdvice">
  <tx:attributes>
    <tx:method name="*" rollback-for="Throwable" no-rollback-for="InstrumentNotFoundException"/>
  </tx:attributes>
</tx:advice>
```

您还可以通过编程方式指示所需的回滚。尽管很简单，但是此过程具有很大的侵入性，并将您的代码紧密耦合到 Spring Framework 的事务基础结构。下面的示例演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();
    }
}

```

强烈建议您尽可能使用声明性方法进行回滚。如果您绝对需要它，则可以使用程序化回滚，但是面对一个干净的基于 POJO 的体系结构，它的用法就不那么理想了。

#### 1.4.4. 为不同的 Bean 配置不同的事务语义

考虑以下场景：您有许多服务层对象，并且您希望对每个对象应用完全不同的事务配置。您可以通  
过定义具有不同 `pointcut` 和 `advice-ref` 属性值的不同 `<aop:advisor/>` 元素来做到这一点。

作为比较，首先假定所有服务层类都在根 `x.y.service` 包中定义。要使所有在该包(或子包)中定义  
的类实例且名称以 `Service` 结尾的 bean 具有默认的事务配置，可以编写以下代码：

```

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

    <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 pack
    <bean id="barManager" class="x.y.service.SimpleBarManager"/> <!-- (doesn't end in '
```

```

<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>

```

以下示例说明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
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/tx
           http://www.springframework.org/schema/tx/spring-tx.xsd
           http://www.springframework.org/schema/aop
           http://www.springframework.org/schema/aop/spring-aop.xsd">

<aop:config>

    <aop:pointcut id="defaultServiceOperation"
                   expression="execution(* x.y.service.*Service.*(..))" />

    <aop:pointcut id="noTxServiceOperation"
                   expression="execution(* x.y.service.ddl.DefaultDdlManager.*(..))" />

    <aop:advisor pointcut-ref="defaultServiceOperation" advice-ref="defaultTxAdvice" />
    <aop:advisor pointcut-ref="noTxServiceOperation" advice-ref="noTxAdvice" />

</aop:config>

<!-- this bean will be transactional (see the 'defaultServiceOperation' pointcut) -->
<bean id="fooService" class="x.y.service.DefaultFooService"/>

<!-- this bean will also be transactional, but with totally different transactional settings -->
<bean id="anotherFooService" class="x.y.service.ddl.DefaultDdlManager"/>

<tx:advice id="defaultTxAdvice">
    <tx:attributes>
        <tx:method name="get*" read-only="true" />
        <tx:method name="*" />
    </tx:attributes>
</tx:advice>

<tx:advice id="noTxAdvice">
    <tx:attributes>
        <tx:method name="*" propagation="NEVER" />
    </tx:attributes>

```

```

</tx:attributes>
</tx:advice>

<!-- other transaction infrastructure beans such as a PlatformTransactionManager omitted -->

</beans>

```

## 1.4.5. <tx:advice/>设置

本节总结了您可以使用 `<tx:advice/>` 标签指定的各种事务设置。默认的 `<tx:advice/>` 设置为：

:

- propagation setting 是 `REQUIRED`.
- 隔离级别为 `DEFAULT`.
- 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，则默认为无。
- 任何 `RuntimeException` 都会触发回滚，而所有选中的 `Exception` 都不会触发。

您可以更改这些默认设置。下表总结了嵌套在 `<tx:advice/>` 和 `<tx:attributes/>` 标签中的 `<tx:method/>` 标签的各种属性：

表 1.<tx:method/>设置

Attribute	Required?	Default	Description
<code>name</code>	Yes		与事务属性关联的方法名称。通配符(*)可用于将事务属性设置与多种方法(例如 <code>get*</code> , <code>handle*</code> 等)相关联。
<code>propagation</code>	No	<code>REQUIRED</code>	事务传播行为。

Attribute	Required?	Default	Description
<code>isolation</code>	No	<code>DEFAULT</code>	事务隔离级别。仅适用于 <code>REQUIRED</code> 或 <code>REQUIRES_NEW</code> 的传播设置。
<code>timeout</code>	No	-1	事务超时(秒)。仅适用于传播 <code>REQUIRED</code> 或 <code>REQUIRES_NEW</code> 。
<code>read-only</code>	No	<code>false</code>	读写与只读事务。仅适用于 <code>REQUIRED</code> 或 <code>REQUIRES_NEW</code> 。
<code>rollback-for</code>	No		触发回滚的 <code>Exception</code> 个实例的逗号分隔的 <code>com.foo.MyBusinessException,ServiceException</code>
<code>no-rollback-for</code>	No		逗号分隔的 <code>Exception</code> 个实例列表，不会触发回滚。如 <code>com.foo.MyBusinessException,ServiceException</code>

## 1.4.6. 使用@Transactional

除了基于 XML 的声明式方法进行事务配置外，还可以使用基于 Comments 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，因为原本打算以事务方式使用的代码几乎总是以这种方式部署。

### iNote

还支持使用标准的 `javax.transaction.Transactional` Comments 来代替 Spring 自己的 Comments。请参阅 JTA 1.2 文档以获取更多详细信息。

使用 `@Transactional` 注解提供的易用性将通过一个示例得到最好的说明，下面将对此进行说明。

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);

}
```

在上面的类级别使用，`Comments` 指示声明类(及其子类)的所有方法的默认值。另外，每种方法都可以单独 `Comments`。注意，类级别的 `Comments` 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，以参与子类级别的 `Comments`。

当一个 POJO 类(例如上面的一个)在 Spring 上下文中定义为 bean 时，您可以通过

`@Configuration` 类中的 `@EnableTransactionManagement` `Comments` 使 bean 实例具有事务性。有关详细信息，请参见[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
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/tx
           http://www.springframework.org/schema/tx/spring-tx.xsd
           http://www.springframework.org/schema/aop
           http://www.springframework.org/schema/aop/spring-aop.xsd">

<!-- this is the service object that we want to make transactional -->
<bean id="fooService" class="x.y.service.DefaultFooService"/>

<!-- enable the configuration of transactional behavior based on annotations -->
<tx:annotation-driven transaction-manager="txManager"/><!-- a PlatformTransactionMa

<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransacti
```

```
<!-- (this dependency is defined somewhere else) -->
<property name="dataSource" ref="dataSource"/>
</bean>

<!-- other <bean/> definitions here -->

</beans>
```

- (1) 使 bean 实例具有事务性的行。

### Tip

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` Comments 应用于具有公开可见性的方法。如果使用 `@Transactional` Comments 对受保护的，私有的或程序包可见的方法进行 Comments，则不会引发任何错误，但是带 Comments 的方法不会显示已配置的事务设置。如果需要 Comments 非公共方法，请考虑使用 AspectJ(稍后描述)。

您可以将 `@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，仅存在 `@Transactional` Comments 不足以激活事务行为。`@Transactional` Comments 仅仅是元数据，可以被 `@Transactional` 感知的某些运行时基础结构使用，并且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。

### Tip

Spring 团队建议您仅使用 `@Transactional` 对具体类(以及具体类的方法)进行 `Comments`, 而不是对接口进行 `Comments`。您当然可以在接口(或接口方法)上放置 `@Transactional` 注解, 但这仅在您使用基于接口的代理时才可以预期地起作用。Java 注解不从接口继承的事实意味着, 如果您使用基于类的代理(`proxy-target-class="true"`)或基于编织的方面(`mode="aspectj"`), 则代理和编织基础结构无法识别事务设置, 并且该对象是没有包装在 `Transaction` 代理中。

### 1 Note

在代理模式(默认设置)下, 仅拦截通过代理传入的外部方法调用。这意味着即使调用的方法标记为 `@Transactional`, 自调用(实际上是目标对象内的方法调用目标对象的另一种方法)也不会导致实际事务。另外, 必须完全初始化代理以提供预期的行为, 因此您不应在初始化代码(即 `@PostConstruct`)中依赖此功能。

如果您希望自调用也与事务包装在一起, 请考虑使用 AspectJ 模式(请参见下表中的 `mode` 属性)。

在这种情况下, 首先没有代理。而是编织目标类(即, 修改其字节码), 以将 `@Transactional` 转换为任何方法上的运行时行为。

表 2. `Comments` 驱动的事务设置

XML Attribute	Annotation Attribute	Default
<code>transaction-</code> <code>manager</code>	不适用(请参见 <a href="#">TransactionManagementConfigurer</a> javadoc)	<code>transactionManager</code>

XML Attribute	Annotation Attribute	Default
	mode	proxy
proxy-target-class	proxyTargetClass	false

XML Attribute	Annotation Attribute	Default
order	order	Ordered.LOWEST_PRECEDENCE

XML Attribute	Annotation Attribute	Default

### iNote

处理 `@Transactional` 注解的默认建议模式是 `proxy`，该模式仅允许通过代理拦截呼叫。同一类内的本地调用无法以这种方式被拦截。对于更高级的侦听模式，请考虑结合编译时或加载时编织切换到 `aspectj` 模式。

### iNote

`proxy-target-class` 属性控制为使用 `@Transactional` `Comments` 标注的类创建哪种类型的事务代理。如果 `proxy-target-class` 设置为 `true`，则会创建基于类的代理。如果 `proxy-target-class` 是 `false` 或省略了属性，则将创建基于标准 JDK 接口的代理。（有关不同代理类型的讨论，请参见[\[aop-proxying\]](#)。）

### iNote

`@EnableTransactionManagement` 和 `<tx:annotation-driven/>` 仅在定义它们的相同应用程序上下文中的 Bean 上寻找 `@Transactional`。这意味着，如果将 `Comments` 驱动的配置放在\_6 的 `WebApplicationContext` 中，则仅在控制器而不是服务中检查 `@Transactional` bean。有关更多信息，请参见[MVC](#)。

在评估方法的事务设置时，最派生的位置优先。在下面的示例中，使用只读事务的设置在类级别 `Comments` `DefaultFooService` 类，但是同一类中 `updateFoo(Foo)` 方法上的 `@Transactional` `Comments` 优先于在类级别定义的事务设置。

```

@Transactional(readOnly = true)
public class DefaultFooService implements FooService {

    public Foo getFoo(String fooName) {
        // do something
    }

    // these settings have precedence for this method
    @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
    public void updateFoo(Foo foo) {
        // do something
    }
}

```

## @Transactional Settings

`@Transactional` Comments 是元数据，它指定接口，类或方法必须具有事务语义(例如，“在调用此方法时启动一个全新的只读事务，并挂起任何现有事务”)。默认的 `@Transactional` 设置如下：

- 传播设置为 `PROPAGATION_REQUIRED`。
- 隔离级别为 `ISOLATION_DEFAULT`。
- 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，则默认为无。
- 任何 `RuntimeException` 都会触发回滚，而所有选中的 `Exception` 都不会触发。

您可以更改这些默认设置。下表总结了 `@Transactional` 注解的各种属性：

表 3. `@Transaction` 设置

Property	Type	Description
<code>value</code>	<code>String</code>	可选的限定词，指定要使用的事务 Management 器。

Property	Type	Description
<u>propagation</u>	<p><code>enum</code> :</p> <p><code>Propagation</code></p>	<p>可选的传播设置。</p>
<u>isolation</u>	<p><code>enum</code> : <code>Isolation</code></p>	<p>可选的隔离级别。仅适用于 <code>REQUIRED</code> 或 <code>REQUIRES_NEW</code> 的传播值。</p>
<u>timeout</u>	<p><code>int</code> (以秒为单位)</p>	<p>可选的事务超时。仅适用于 <code>REQUIRED</code> 或 <code>REQUIRES_NEW</code> 的传播值。</p>
<u>readOnly</u>	<p><code>boolean</code></p>	<p>读写与只读事务。仅适用于 <code>REQUIRED</code> 或 <code>REQUIRES_NEW</code> 的值。</p>
<u>rollbackFor</u>	<p><code>Class</code> 个对象的数组，必须从 <code>Throwable.</code> 派生</p>	<p>必须引起回滚的异常类的可选数组。</p>
<u>rollbackForClassName</u>	<p>类名数组。这些类必须源自 <code>Throwable.</code></p>	<p>必须引起回滚的异常类名称的可选数组。</p>

Property	Type	Description
<code>noRollbackFor</code>	<code>Class</code> 个对象的数组 ，必须从 <code>Throwable</code> 派生	不能导致回滚的异常类的可选数组。
<code>noRollbackForClassName</code>	<code>String</code> 类名称的数组，必须从 <code>Throwable</code> 派生	不能引起回滚的异常类名称的可选数组。

当前，您无法对事务名称进行显式控制，其中“名称”是指显示在事务监视器(如果适用)(例如，WebLogic 的事务监视器)和日志输出中的事务名称。对于声明性事务，事务名称始终是完全合格的类名称。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，则事务的名称应为：`com.example.BusinessService.handlePayment`。

## 具有@Transactional 的多个事务 Management 器

大多数 Spring 应用程序仅需要一个事务 Management 器，但是在某些情况下，您可能需要在一个应用程序中使用多个独立的事务 Management 器。您可以使用 `@Transactional` 注解的 `value` 属性来指定要使用的 `PlatformTransactionManager` 的标识。这可以是 bean 名称，也可以是事务 Management 器 bean 的限定符值。例如，使用限定符表示法，可以在应用程序上下文中将以下 Java 代码与以下事务 Management 器 bean 声明进行组合：

```
public class TransactionalService {

    @Transactional("order")
    public void setSomething(String name) { ... }

    @Transactional("account")
    public void doSomething() { ... }
}
```

以下 Lists 显示了 bean 声明：

```
<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` 上的两种方法在单独的事务 Management 器下运行，并以 `order` 和 `account` 限定词进行区分。如果未找到特别限定的 `PlatformTransactionManager` bean，仍将使用默认的 `<tx:annotation-driven>` 目标 bean 名称 `transactionManager`。

## 自定义快捷方式 Comments

如果发现您在许多不同的方法上重复使用 `@Transactional` 的相同属性，则可以使用 [Spring 的元 Comments](#) 支持为特定用例定义自定义快捷方式 Comments。例如，考虑以下 Comments 定义：

```
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Transactional("order")
public @interface OrderTx {
}

@Transactional("account")
public @interface AccountTx { }
```

前面的 Comments 使我们可以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，我们使用了语法来定义事务 Management 器限定符，但是我们还可以包括传播

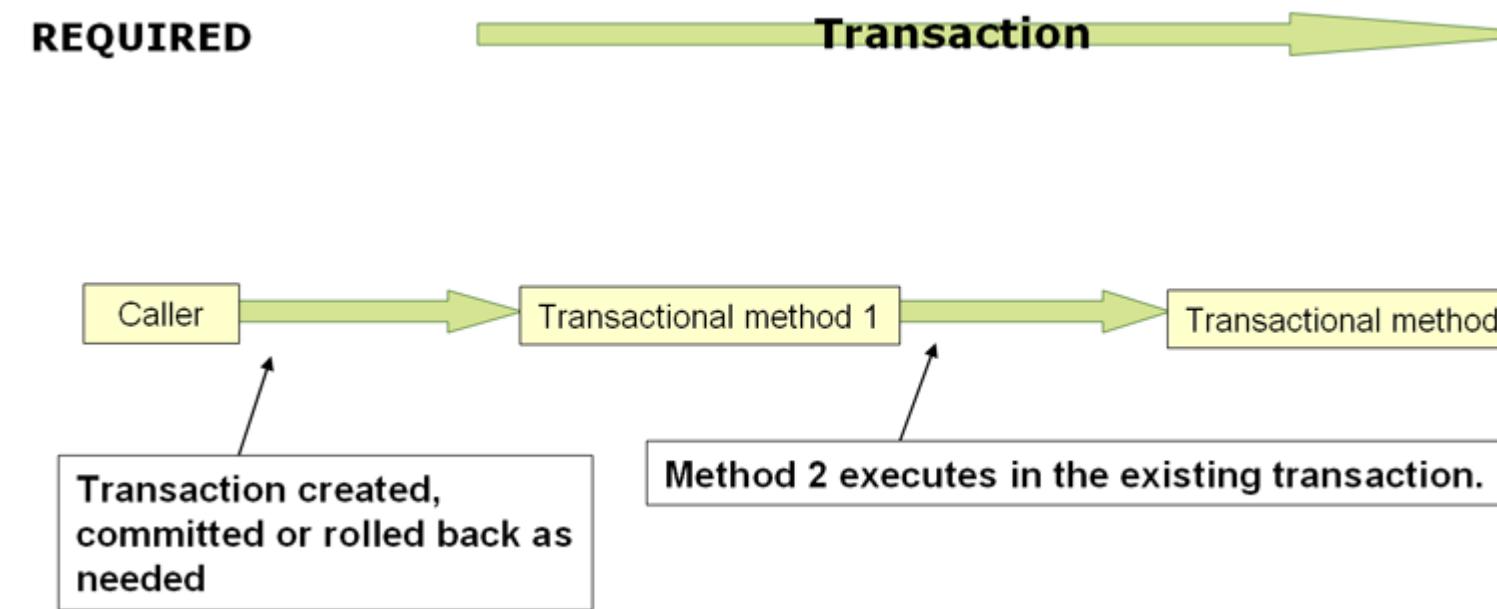
行为，回滚规则，超时和其他功能。

### 1.4.7. Transaction 传播

本节描述了 Spring 中事务传播的一些语义。请注意，本节不是对事务传播的适当介绍。相反，它详细介绍了有关 Spring 中事务传播的一些语义。

在 SpringManagement 的事务中，请注意物理事务和逻辑事务之间的差异，以及传播设置如何应于此差异。

#### Understanding PROPAGATION\_REQUIRED



`PROPAGATION_REQUIRED` 强制执行物理事务，如果尚不存在，则在当前范围内本地执行，或参与为较大范围定义的现有“外部”事务。这是同一线程(例如，委派给几种存储库方法的服务立面，所有基础资源都必须参与服务级事务的服务立面)的优良默认设置。

#### iNote

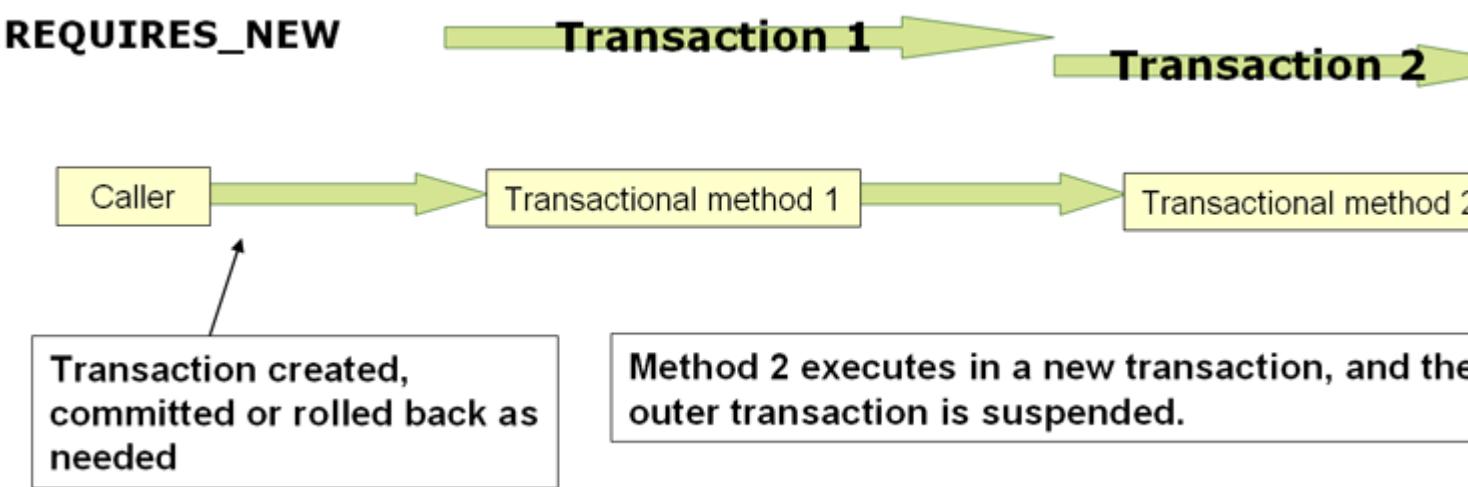
默认情况下，参与的事务将加入外部作用域的 Feature，而忽略本地隔离级别，超时值或只读标志(如果有)。如果要在参与具有不同隔离级别的现有事务时拒绝隔离级别声明，请考虑在事务 Management 器上将 `validateExistingTransactions` 标志切换为 `true`。这种非

宽容模式还拒绝只读不匹配(即，内部读写事务试图参与只读外部作用域)。

当传播设置为 `PROPAGATION_REQUIRED` 时，将为应用该设置的每种方法创建一个逻辑事务作用域。每个这样的逻辑事务作用域可以单独确定仅回滚状态，而外部事务作用域在逻辑上独立于内部事务作用域。在标准 `PROPAGATION_REQUIRED` 行为的情况下，所有这些范围都 Map 到同一物理事务。因此，内部事务范围内设置的仅回滚标记确实会影响外部事务实际提交的机会。

但是，在内部事务范围内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(由内部事务范围默默触发)是意外的。此时将抛出一个对应的 `UnexpectedRollbackException`。这是预期的行为，因此事务调用者永远不会被误认为是在确实未执行提交的情况下进行的。因此，如果内部事务(外部调用者不知道)将事务无提示地标记为仅回滚，则外部调用者仍会调用 `commit`。外部呼叫者需要接收 `UnexpectedRollbackException` 来明确指示已执行回滚。

## Understanding PROPAGATION\_REQUIRE\_NEW



与 `PROPAGATION_REQUIRED` 相比，`PROPAGATION_REQUIRE_NEW` 始终对每个受影响的事务范围使用独立的物理事务，而不参与外部范围的现有事务。在这种安排中，基础资源事务是不同的，因此可以独立地提交或回滚，而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其自己的隔离级别，超时和只读设置，而不继承外部事务的 Feature。

## Understanding PROPAGATION\_NESTED

`PROPAGATION_NESTED` 使用具有多个可还原到的保存点的单个物理事务。这种部分回滚使内部事务范围触发其范围的回滚，尽管某些操作已回滚，但外部事务仍能够 `continue` 物理事务。此设置通常 Map 到 JDBC 保存点，因此仅适用于 JDBC 资源事务。参见 Spring 的 [DataSourceTransactionManager](#)。

## 1.4.8. Transaction 事务咨询

假设您要执行事务性操作和一些基本的分析建议。您如何在 `<tx:annotation-driven/>` 的情况下实现此目的？

调用 `updateFoo(Foo)` 方法时，您想要查看以下操作：

- 配置的外观方面开始。
- Transaction 建议执行。
- 建议对象上的方法执行。
- 事务提交。
- 分析方面报告整个事务方法调用的确切持续时间。

### ①Note

本章不涉及任何详细的 AOP 解释(除非它适用于事务)。有关 AOP 配置和一般 AOP 的详细介绍，请参见[AOP](#)。

以下代码显示了前面讨论的简单配置方面：

```
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;
}
}

```

建议的排序是通过 `Ordered` 界面控制的。有关建议 Order 的完整详细信息，请参见[Advice ordering](#)。

以下配置创建一个 `fooService` bean，该 bean 具有按所需 Sequences 应用的概要分析和事务方面：

```

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

    <bean id="fooService" class="x.y.service.DefaultFooService"/>

    <!-- this is the aspect -->
    <bean id="profiler" class="x.y.SimpleProfiler">
        <!-- execute before the transactional advice (hence the lower order number) -->
        <property name="order" value="1"/>
    </bean>

    <tx:annotation-driven transaction-manager="txManager" order="200"/>

    <aop:config>

```

```

<!-- this advice will execute around the transactional advice -->
<aop:aspect id="profilingAspect" ref="profiler">
    <aop:pointcut id="serviceMethodWithReturnValue"
        expression="execution(!void x.y..*Service.*(..))"/>
    <aop:around method="profile" pointcut-ref="serviceMethodWithReturnValue"
    </aop:aspect>
</aop:config>

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
    <property name="url" value="jdbc:oracle:thin:@rj-t42:1521:elvis"/>
    <property name="username" value="scott"/>
    <property name="password" value="tiger"/>
</bean>

<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>

</beans>

```

您可以用类似的方式配置任意数量的其他方面。

下面的示例创建与前两个示例相同的设置，但是使用纯 XML 声明性方法：

```

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

    <bean id="fooService" class="x.y.service.DefaultFooService"/>

    <!-- the profiling advice -->
    <bean id="profiler" class="x.y.SimpleProfiler">
        <!-- execute before the transactional advice (hence the lower order number) -->
        <property name="order" value="1"/>
    </bean>

    <aop:config>
        <aop:pointcut id="entryPointMethod" expression="execution(* x.y..*Service.*(..))"/>
        <!-- will execute after the profiling advice (c.f. the order attribute) -->

        <aop:advisor advice-ref="txAdvice" pointcut-ref="entryPointMethod" order="2"/>
        <!-- order value is higher than the profiling aspect -->

        <aop:aspect id="profilingAspect" ref="profiler">
            <aop:pointcut id="serviceMethodWithReturnValue"
                expression="execution(!void x.y..*Service.*(..))"/>
            <aop:around method="profile" pointcut-ref="serviceMethodWithReturnValue"/>
        </aop:aspect>
    </aop:config>

```

```
</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 -->

</beans>
```

先前配置的结果是一个 `fooService` bean，该 bean 依次具有概要分析和事务方面的内容。如果您希望性能分析建议在进来的事务处理建议之后和之后的事务处理建议之前执行，则可以交换性能分析方面 Bean 的 `order` 属性的值，以使其高于事务处理建议的订单值。

您可以以类似方式配置其他方面。

#### 1.4.9. 在 AspectJ 中使用@Transactional

您还可以通过 AspectJ 方面在 Spring 容器之外使用 Spring Framework 的 `@Transactional` 支持。为此，首先使用 `@Transactional` Comments 对您的类(以及可选的类的方法)进行 Comments，然后将您的应用程序与 `spring-aspects.jar` 文件中定义的

`org.springframework.transaction.aspectj.AnnotationTransactionAspect` 链接(编织)。

您还必须使用事务 Management 器配置方面。您可以使用 Spring Framework 的 IoC 容器来进行依赖注入方面。配置事务 Management 方面的最简单方法是使用 `<tx:annotation-driven/>` 元素并将 `mode` 属性指定为 `aspectj`，如[Using @Transactional](#)中所述。因为这里我们专注于在 Spring 容器之外运行的应用程序，所以我们向您展示了如何以编程方式进行操作。

#### iNote

在 `continue` 之前，您可能需要分别阅读[Using @Transactional](#)和[AOP](#)。

以下示例显示了如何创建事务 Management 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  
AnnotationTransactionAspect.aspectOf().setTransactionManager(txManager);
```

### iNote

使用此方面时，必须 `Comments` 实现类(或该类中的方法或两者)，而不是 `Comments` 该类所实现的接口(如果有)。AspectJ 遵循 Java 的规则，即不继承接口上的 `Comments`。

类上的 `@Transactional` `Comments` 指定用于执行该类中任何公共方法的默认事务语义。

类中方法上的 `@Transactional` `Comments` 将覆盖类 `Comments`(如果存在)给出的默认事务语义。您可以 `Comments` 任何方法，而不管可见性如何。

要使用 `AnnotationTransactionAspect` 编织应用程序，您必须使用 AspectJ 来构建应用程序(请参见[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，请参见[在 Spring Framework 中使用 AspectJ 进行加载时编织](#)。

## 1.5. 程序化 TransactionManagement

Spring 框架通过使用以下两种方式提供程序化事务 Management 的方法：

- `TransactionTemplate`。
- 直接执行 `PlatformTransactionManager`。

Spring 团队通常建议使用 `TransactionTemplate` 进行程序化事务 Management。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(例如 `JdbcTemplate`)相同的方法。它使用一种回调方法(使应用程序代码不必进行样板获取和释放事务性资源), 并生成意向驱动的代码, 因为您 的代码仅专注于您要执行的操作。

### iNote

如以下示例所示, 使用 `TransactionTemplate` 绝对可以使您与 Spring 的事务基础结构和 API 耦合。程序化事务 Management 是否适合您的开发需求是您必须自己做的决定。

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, 您可以编写 `TransactionCallback` 实现(通常表示为匿名内部类), 其中包含您需要在事务上下文中执行的代码。然后, 您可以将自定义 `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, 则可以将便捷的 `TransactionCallbackWithoutResult` 类与匿名类一起使用 , 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 Transaction 设置

您可以以编程方式或配置方式在 `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` 个实例具有[默认 Transaction 设置](#)。以下示例显示了针对特定 `TransactionTemplate` 的 Transaction 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);

        // the transaction settings can be set here explicitly if so desired
        this.transactionTemplate.setIsolationLevel(TransactionDefinition.ISOLATION_READ_COMMITTED);
        this.transactionTemplate.setTimeout(30); // 30 seconds
        // and so forth...
    }
}
```

以下示例通过使用 Spring XML 配置来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，您可以根据需要将 `sharedTransactionTemplate` 注入到尽可能多的服务中。

最后，`TransactionTemplate` 类的实例是线程安全的，因为该实例不维护任何对话状态。

`TransactionTemplate` 实例确实会维持配置状态。因此，尽管许多类可以共享一个 `TransactionTemplate` 的单个实例，但是如果一个类需要使用具有不同设置(例如，不同的隔离级别)的 `TransactionTemplate`，则需要创建两个不同的 `TransactionTemplate` 实例。

### 1.5.2. 使用 PlatformTransactionManager

您也可以直接使用 `org.springframework.transaction.PlatformTransactionManager` 来 Management 您的 Transaction。为此，请通过 bean 引用将您使用的 `PlatformTransactionManager` 的实现传递给 bean。然后，通过使用 `TransactionDefinition` 和 `TransactionStatus` 对象，您可以启动事务，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. 在程序性和声明性事务 Management 之间进行选择

仅当您执行少量 Transaction 操作时，程序 TransactionManagement 通常是一个好主意。例如

, 如果您的 Web 应用程序仅要求对某些更新操作进行事务处理, 则可能不希望通过使用 Spring 或任何其他技术来设置事务代理。在这种情况下, 使用 `TransactionTemplate` 可能是一种很好的方法。能够显式设置事务名称也是只能通过使用编程方法进行事务 Management 来完成的事情。

另一方面, 如果您的应用程序具有大量事务操作, 则声明式事务 Management 通常是值得的。它使事务 Management 脱离业务逻辑, 并且不难配置。当使用 Spring 框架而不是 EJB CMT 时, 声明式事务 Management 的配置成本大大降低了。

## 1.7. Transaction 绑定事件

从 Spring 4.2 开始, 事件的侦听器可以绑定到事务的某个阶段。典型的示例是在事务成功完成后处理事件。这样, 当当前事务的结果实际上对侦听器很重要时, 便可以更加灵活地使用事件。

您可以使用 `@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`。这样做时, 默认情况下, 侦听器绑定到事务的提交阶段。

下一个示例显示了此概念。假设某个组件发布了一个订单创建的事件, 并且我们想要定义一个侦听器, 该侦听器仅在发布该事件的事务成功提交后才应处理该事件。以下示例设置了这样的事件侦听器:

```
@Component
public class MyComponent {

    @TransactionalEventListener
    public void handleOrderCreatedEvent(CreationEvent<Order> creationEvent) {
        ...
    }
}
```

`@TransactionalEventListener` `Comments` 公开了 `phase` 属性, 该属性使您可以自定义应将侦听器绑定到的事务阶段。有效阶段为 `BEFORE_COMMIT`, `AFTER_COMMIT` (默认), `AFTER_ROLLBACK` 和 `AFTER_COMPLETION`, 这些阶段汇总事务完成(提交或回滚)。

如果没有事务在运行, 则根本不会调用侦听器, 因为我们无法遵守所需的语义。但是, 您可以通过将 `Comments` 的 `fallbackExecution` 属性设置为 `true` 来覆盖该行为。

## 1.8. 应用服务器特定的集成

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，该位置随应用程序服务器的不同而不同。可以访问 JTA `TransactionManager` 来增强事务语义-特别是支持事务挂起。有关详细信息，请参见[JtaTransactionManager](#) javadoc。

Spring 的 `JtaTransactionManager` 是在 Java EE 应用程序服务器上运行的标准选择，并且已知可以在所有普通服务器上运行。诸如事务挂起之类的高级功能也可以在许多服务器(包括 GlassFish, JBoss 和 Geronimo)上运行，而无需任何特殊配置。但是，为了完全支持事务暂停和进一步的高级集成，Spring 包括用于 WebLogic Server 和 WebSphere 的特殊适配器。这些适配器将在以下各节中讨论。

对于包括 WebLogic Server 和 WebSphere 在内的标准方案，请考虑使用方便的 `<tx:jta-transaction-manager/>` 配置元素。配置后，此元素将自动检测基础服务器，并选择可用于平台的最佳事务 Management 器。这意味着您无需显式配置服务器特定的适配器类(如以下各节所述)。而是自动选择它们，并以标准 `JtaTransactionManager` 作为默认后备。

### 1.8.1. IBM WebSphere

在 WebSphere 6.1.0.9 和更高版本上，推荐使用的 Spring JTA 事务 Management 器是 `WebSphereUowTransactionManager`。这个特殊的适配器使用 IBM 的 `UOWManager` API，WebSphere Application Server 6.1.0.9 和更高版本中提供了该 API。使用此适配器，IBM 正式支持 Spring 驱动的事务挂起(由 `PROPAGATION_REQUIRE_NEW` 发起的挂起和 `continue`)。

### 1.8.2. Oracle WebLogic 服务器

在 WebLogic Server 9.0 或更高版本上，通常将使用 `WebLogicJtaTransactionManager` 而不是 stock `JtaTransactionManager` 类。常规 `JtaTransactionManager` 的特定于 WebLogic 的特殊

子类在标准 JTA 语义之外，支持 WebLogicManagement 的事务环境中 Spring 事务定义的全部功能。功能包括事务名称，每个事务的隔离级别以及在所有情况下正确恢复事务。

## 1.9. 常见问题的解决方案

本节介绍一些常见问题的解决方案。

### 1.9.1. 对特定的数据源使用错误的事务 Management 器

根据您选择的 Transaction 技术和要求，使用正确的 `PlatformTransactionManager` 实现。如果使用得当，Spring 框架仅提供了直接且可移植的抽象。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(或其中的[应用服务器器特定的子类](#))。否则，事务基础结构将尝试在诸如容器 `DataSource` 实例之类的资源上执行本地事务。这样的本地事务是没有意义的，好的应用服务器会将它们视为错误。

## 1.10. 更多资源

有关 Spring 框架的事务支持的更多信息，请参见：

- [带有和不带有 XA 的 Spring 中的分布式事务](#)是 JavaWorld 的演示文稿，其中 Spring 的 David Syer 指导您完成在 Spring 应用程序中进行分布式事务处理的七个模式，其中三个具有 XA 模式，四个具有 XA 模式。
- [Java 事务设计策略](#)是 InfoQ 提供的一本书，详细介绍了 Java 事务。它还包括有关如何通过 Spring Framework 和 EJB3 配置和使用事务的并行示例。

## 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)。这使您可以轻松地在上述持久性技术之间进行切换，还使您无需担心捕获每种技术特有的异常即可进行编码。

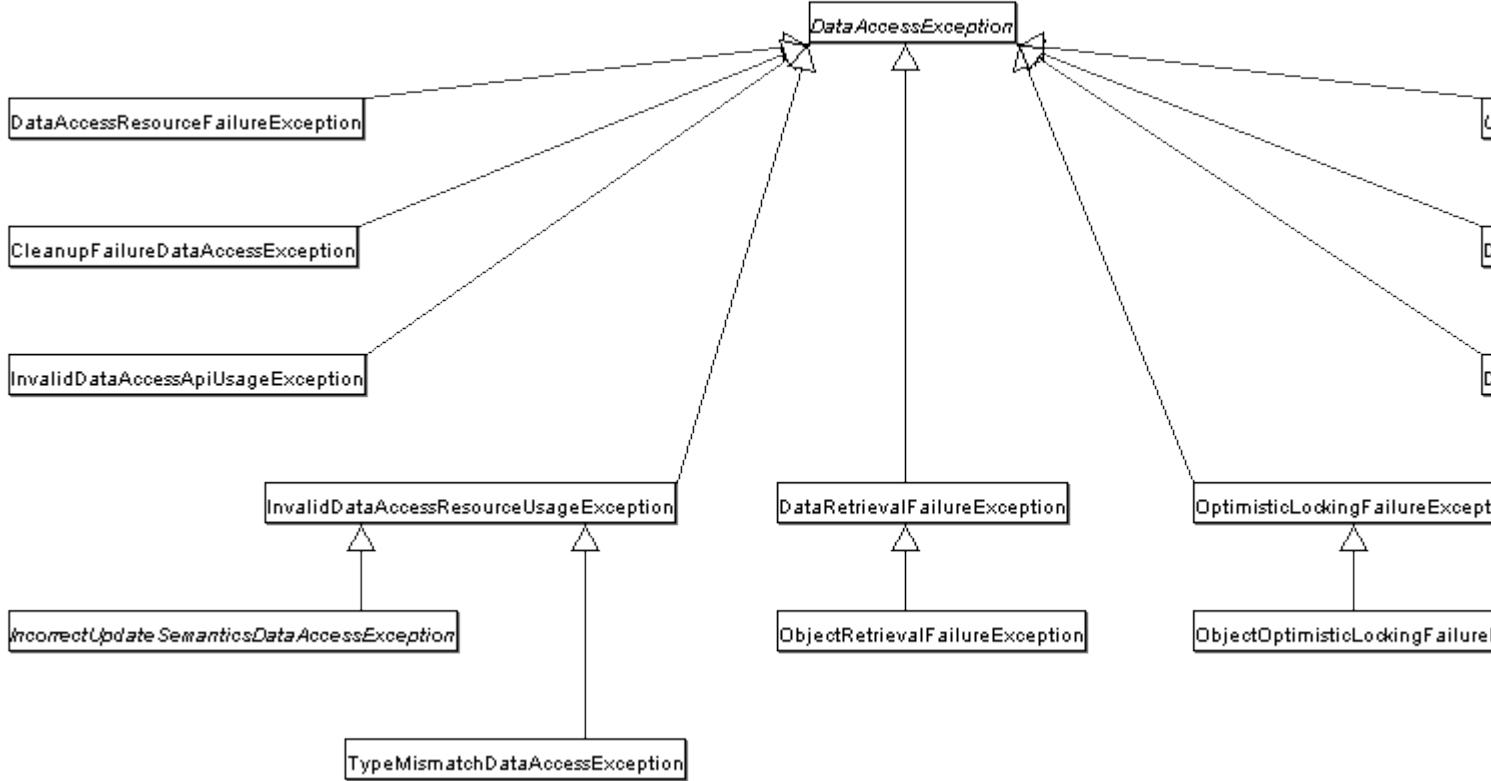
## 2.1. 一致的异常层次结构

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(以 `DataAccessException` 作为根异常)的便捷转换。这些异常包装了原始异常，因此您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，您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。(尽管您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，这意味着您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。如果您使用基于拦截器的类，则应用程序必须关心处理 `HibernateExceptions` 和 `PersistenceExceptions` 本身，最好分别委派 `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 `PersistenceExceptions`，它们也可能被抛出(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，图像中详细说明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 Comments

确保您的数据访问对象(DAO)或存储库提供异常翻译的最佳方法是使用 `@Repository` 注解。此

Comments 还使组件扫描支持可以查找和配置 DAO 和存储库，而不必为其提供 XML 配置条目。以下示例显示了如何使用 `@Repository` 注解：

```

@Repository (1)
public class SomeMovieFinder implements MovieFinder {
    // ...
}
  
```

- (1) `@Repository` Comments。

根据使用的持久性技术，任何 DAO 或存储库实现都需要访问持久性资源。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;  
  
    // ...  
}
```

如果您使用传统的 **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;  
    }  
  
    // ...  
}
```

我们在此显示的最后一个示例是对典型 **JDBC** 支持的。您可以将 **DataSource** 注入到初始化方法中，在该方法中，您可以使用 **DataSource** 创建 **JdbcTemplate** 和其他数据访问支持类(例如 **SimpleJdbcCall** 等)。以下示例自动连接 **DataSource**：

```
@Repository  
public class JdbcMovieFinder implements MovieFinder {  
  
    private JdbcTemplate jdbcTemplate;  
  
    @Autowired  
    public void init(DataSource dataSource) {  
        this.jdbcTemplate = new JdbcTemplate(dataSource);  
    }  
  
    // ...  
}
```

### iNote

有关如何配置应用程序上下文以利用这些 **Comments** 的详细信息，请参见每种持久性技术的

特定介绍。

### 3. 使用 JDBC 进行数据访问

下表中概述的操作序列可能最好地显示了 Spring Framework JDBC 抽象提供的值。该表显示了 Spring 负责哪些操作，哪些操作是您的责任。

表 4. Spring JDBC-谁做什么？

Action	Spring	You
定义连接参数。		X
打开连接。	X	
指定 SQL 语句。		X
声明参数并提供参数值		X
准备并执行该语句。	X	
设置循环以遍历结果(如果有)。	X	
进行每次迭代的工作。		X
处理任何异常。	X	
Handle transactions.	X	

Action	Spring	You
关闭连接，语句和结果集。	X	

Spring 框架负责所有可能使 JDBC 成为乏味的 API 的低级细节。

### 3.1. 选择一种用于 JDBC 数据库访问的方法

您可以选择几种方法来构成 JDBC 数据库访问的基础。除了 `JdbcTemplate` 的三种风格之外，新的 `SimpleJdbcInsert` 和 `SimpleJdbcCall` 方法还优化了数据库元数据，并且 RDBMS Object 样式采用了一种更加面向对象的方法，类似于 JDO Query 设计。一旦开始使用这些方法之一，您仍然可以混合搭配以包含来自其他方法的功能。所有方法都需要兼容 JDBC 2.0 的驱动程序，某些高级功能需要 JDBC 3.0 驱动程序。

- `JdbcTemplate` 是经典且最受欢迎的 Spring JDBC 方法。这种“最低级别”的方法以及所有其他方法都在幕后使用了 `JdbcTemplate`。
- `NamedParameterJdbcTemplate` 包装 `JdbcTemplate` 以提供命名参数，而不是传统的 JDBC `?` 占位符。当您有多个 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，因此您只需要提供表或过程的名称，并提供与列名称匹配的参数 Map 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`)要求您在初始化数据访问层期间创建可重用且线程安全的对象。此方法以 JDO Query 为模型，其中您定义查询字符串，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。

### 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。参见[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 Util 类，用于方便的 `DataSource` 访问和各种简单的 `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.springframework.jdbc.datasource.embedded` 的子程序包支持使用 Java 数据库引擎(例如 HSQL, H2 和 Derby)创建嵌入式数据库。参见[控制数据库连接](#)和[嵌入式数据库支持](#)。
- `object` : `org.springframework.jdbc.object` 软件包包含一些类，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。参见[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，但此方法由 JDO 建模。较高级别的 JDBC 抽象取决于 `org.springframework.jdbc.core` 包中的较低级别的抽象。
- `support` : `org.springframework.jdbc.support` 包提供 `SQLException` 转换功能和一些 Util 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，这使您可以选择捕获可从中恢复的异常，同时将其他异常传播到调用方。参见[Using SQLExceptionTranslator](#)。

### 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：

- [Using JdbcTemplate](#)
- [Using NamedParameterJdbcTemplate](#)
- [Using SQLExceptionTranslator](#)
- [Running Statements](#)
- [Running Queries](#)
- [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，这有助于您避免常见的错误，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` 个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。（请参阅[一致的异常层次结构](#)。）

在代码中使用 `JdbcTemplate` 时，只需实现回调接口，即可为它们明确定义 Contract。给定 `JdbcTemplate` 类提供的 `Connection`，`PreparedStatementCreator` 回调接口将创建一条准备好的语句，提供 SQL 和任何必要的参数。`CallableStatementCreator` 接口(创建可调用语句)也是如此。`RowCallbackHandler` 接口从 `ResultSet` 的每一行提取值。

您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。

### iNote

`DataSource` 应该始终配置为 Spring IoC 容器中的 bean。在第一种情况下，将 Bean 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 都以 `DEBUG` 级别记录在与模板实例的标准类名相对应的类别下(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 `JdbcTemplate` 用法的一些示例。这些示例不是 `JdbcTemplate` 公开的所有功能的详尽列表。见服务员[javadoc](#)。

## Querying (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 `String`：

```
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.setFirst_name(rs.getString("first_name"));
            actor.setLast_name(rs.getString("last_name"));
            return 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;
        }
});
```

如果最后两个代码段确实存在于同一应用程序中，则删除两个 `RowMapper` 匿名内部类中存在的重复并将它们提取到单个类(通常是 `static` 嵌套类)中，然后可以引用该重复是有意义的。根据需要使用 DAO 方法。例如，最好编写以下代码片段，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;
    }
}
```

## 使用 `JdbcTemplate` 更新(`INSERT`，`UPDATE` 和 `DELETE`)

您可以使用 `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。

下面的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：

```
this.jdbcTemplate.update(  
    "update t_actor set last_name = ? where id = ?" ,  
    "Banjo", 5276L);
```

下面的示例删除一个条目：

```
this.jdbcTemplate.update(  
    "delete from actor where id = ?" ,  
    Long.valueOf(actorId));
```

## 其他 **JdbcTemplate** 操作

您可以使用 `execute(..)` 方法来运行任意 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))");
```

下面的示例调用一个存储过程：

```
this.jdbcTemplate.update(  
    "call SUPPORT.REFRESH_ACTORS_SUMMARY(?)" ,  
    Long.valueOf(unionId));
```

[covered later](#) 是更复杂的存储过程支持。

## **JdbcTemplate** 最佳做法

`JdbcTemplate` 类的实例一旦配置便是线程安全的。这很重要，因为这意味着您可以配置

`JdbcTemplate` 的单个实例，然后将该共享引用安全地注入到多个 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 `DataSource` 的设置器中创建 `JdbcTemplate`。这将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...  
}
```

以下示例显示了相应的 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:context="http://www.springframework.org/schema/context"  
       xsi:schemaLocation="  
           http://www.springframework.org/schema/beans  
           http://www.springframework.org/schema/beans/spring-beans.xsd  
           http://www.springframework.org/schema/context  
           http://www.springframework.org/schema/context/spring-context.xsd">  
  
    <bean id="corporateEventDao" class="com.example.JdbcCorporateEventDao">  
        <property name="dataSource" ref="dataSource"/>  
    </bean>  
  
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">  
        <property name="driverClassName" value="${jdbc.driverClassName}" />  
        <property name="url" value="${jdbc.url}" />  
        <property name="username" value="${jdbc.username}" />  
        <property name="password" value="${jdbc.password}" />  
    </bean>  
  
    <context:property-placeholder location="jdbc.properties" />  
  
</beans>
```

显式配置的替代方法是使用组件扫描和 `Comments` 支持进行依赖项注入。在这种情况下，可以用 `@Repository` `Comments` 该类(这使其成为组件扫描的候选对象)，并用 `@Autowired` `Comments` `DataSource` `setter` 方法。以下示例显示了如何执行此操作：

```
@Repository (1)  
public class JdbcCorporateEventDao implements CorporateEventDao {
```

```

private JdbcTemplate jdbcTemplate;

@Autowired (2)
public void setDataSource(DataSource dataSource) {
    this.jdbcTemplate = new JdbcTemplate(dataSource); (3)
}

// JDBC-backed implementations of the methods on the CorporateEventDao follow...
}

```

- (1) 用 `@Repository` `Comments` 类。
- (2) 用 `@Autowired` `Comments` `DataSource` setter 方法。
- (3) 用 `DataSource` 创建一个新的 `JdbcTemplate`。

以下示例显示了相应的 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:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context.xsd">

    <!-- Scans within the base package of the application for @Component classes to con
    <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" />

```

如果使用 Spring 的 `JdbcDaoSupport` 类，并且各种 JDBC 支持的 DAO 类都从该类扩展，则您的子类将从 `JdbcDaoSupport` 类继承 `setDataSource(...)` 方法。您可以选择是否从此类继承。提供 `JdbcDaoSupport` 类只是为了方便。

无论您选择使用(或不使用)以上哪种模板初始化样式，都无需在每次运行 SQL 时都创建一个新的

`JdbcTemplate` 类实例。配置完成后，`JdbcTemplate` 实例是线程安全的。如果您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` 实例，然后需要多个不同配置的 `JdbcTemplate` 实例。

### 3.3.2. 使用 `NamedParameterJdbcTemplate`

`NamedParameterJdbcTemplate` 类增加了对使用命名参数编程 JDBC 语句的支持，这与仅使用经典占位符 (`'?'`) 参数进行编程的 JDBC 相反。`NamedParameterJdbcTemplate` 类包装 `JdbcTemplate` 并委托包装的 `JdbcTemplate` 完成其许多工作。本节仅描述 `NamedParameterJdbcTemplate` 类的与 `JdbcTemplate` 本身不同的区域，即使用命名参数对 JDBC 语句进行编程。以下示例显示了如何使用 `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 `namedParameters` 变量（类型 `MapSqlParameterSource`）的相应值中使用了命名参数符号。

或者，您可以使用基于 `Map` 的样式将命名参数及其对应的值传递给

`NamedParameterJdbcTemplate` 实例。`NamedParameterJdbcOperations` 公开并由 `NamedParameterJdbcTemplate` 类实现的其余方法遵循类似的模式，此处不再赘述。

下面的示例说明基于 `Map` 的样式的用法：

```
// some JDBC-backed DAO class...
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

public void setDataSource(DataSource dataSource) {
    this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);
}

public int countOfActorsByFirstName(String firstName) {

    String sql = "select count(*) from T_ACTOR where first_name = :first_name";

    Map<String, String> namedParameters = Collections.singletonMap("first_name", firstName);

    return this.namedParameterJdbcTemplate.queryForObject(sql, namedParameters, Integer.class);
}
```

与 `NamedParameterJdbcTemplate` 相关联(并且存在于同一 Java 包中)的一项不错的功能是 `SqlParameterSource` 接口。您已经在以前的代码片段之一(`MapSqlParameterSource` 类)中看到了此接口的实现示例。`SqlParameterSource` 是 `NamedParameterJdbcTemplate` 的命名参数值的来源。`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)，并使用包装的 `JavaBean` 的属性作为命名参数值的源。

以下示例显示了典型的 `JavaBean`:

```
public class Actor {

    private Long id;
    private String firstName;
    private String lastName;

    public String getFirstName() {
        return this.firstName;
    }

    public String getLastName() {
        return this.lastName;
    }

    public Long getId() {
        return this.id;
    }
}
```

```
// setters omitted...
}
```

下面的示例使用 `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`。

另请参阅 [JdbcTemplate 最佳做法](#)，以获取有关在应用程序上下文中使用 `NamedParameterJdbcTemplate` 类的准则。

### 3.3.3. 使用 SQLExceptionTranslator

`SQLExceptionTranslator` 是要由可以在 `SQLExceptions` 和 Spring 自己的 `org.springframework.dao.DataAccessException` 之间进行转换的类实现的接口，而在数据访问策略方面则不可知。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 `SQLExceptionTranslator` 的实现，默认情况下使用。此实现使用特定的供应商代码。它比 `SQLState` 实现更为精确。错误代码转换基于 JavaBean

类型类 `SQLErrorCode` 中保存的代码。此类由 `SQLErrorCodeFactory` 创建和填充，`SQLErrorCodeFactory` (顾名思义)是基于名为 `sql-error-codes.xml` 的配置文件的内容创建 `SQLErrorCode` 的工厂。该文件使用供应商代码填充，并且基于 `DatabaseMetaData` 中的 `DatabaseProductName` 填充。使用您正在使用的实际数据库的代码。

`SQLErrorCodeSQLExceptionTranslator` 按以下 Sequences 应用匹配规则：

- 子类实现的任何自定义转换。通常，将使用提供的具体 `SQLErrorCodeSQLExceptionTranslator`，因此该规则不适用。仅当您确实提供了子类实现时，它才适用。
- 作为 `SQLErrorCode` 类的 `customSqlExceptionTranslator` 属性提供的 `SQLExceptionTranslator` 接口的任何自定义实现。
- 搜索 `CustomSQLErrorCodeTranslation` 类的实例列表(为 `SQLErrorCode` 类的 `customTranslations` 属性提供)，以查找匹配项。
- 错误代码匹配被应用。
- 使用后备翻译器。`SQLExceptionSubclassTranslator` 是默认的后备翻译器。如果此翻译不可用，则下一个后备翻译器是 `SQLStateSQLExceptionTranslator`。

### iNote

默认情况下，使用 `SQLErrorCodeFactory` 来定义 `Error` 代码和自定义异常翻译。在 Classpath 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 元数据中的数据库名称找到匹配的 `SQLErrorCode` 实例。

您可以扩展 `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。若要使用此自定义转换器，必须通过 `setExceptionTranslator` 方法将其传递给 `JdbcTemplate`，并且必须将 `JdbcTemplate` 用于需要该转换器的所有数据访问处理。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 location  
    CustomSQLErrorCodesTranslator tr = new CustomSQLErrorCodesTranslator();  
    tr.setDataSource(dataSource);  
    this.jdbcTemplate.setExceptionTranslator(tr);  
  
}  
  
public void updateShippingCharge(long orderId, long pct) {  
    // use the prepared JdbcTemplate for this update  
    this.jdbcTemplate.update("update orders" +  
        " set shipping_charge = shipping_charge * ? / 100" +  
        " where id = ?", pct, orderId);  
}
```

自定义转换器会传递一个数据源，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。您需要 `DataSource` 和 `JdbcTemplate`，包括 `JdbcTemplate` 随附的便捷方法。以下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;

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` 转换为作为参数传入的 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(..)`，它返回 `List`，其中每个元素都是 `Map`，其中每一列都包含一个条目，并使用列名作为键。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}]
```

### 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。您可以将参数值作为 varargs 或作为对象数组传递。因此，您应该在原始包装器类中显式包装原始器，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。该方法以 `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
            PreparedStatement ps = connection.prepareStatement(INSERT_SQL, new String[] []
                ps.setString(1, name);
                return ps;
    }
),
keyHolder);

// keyHolder.getKey() now contains the generated key
```

## 3.4. 控制数据库连接

本节内容包括：

- [Using DataSource](#)
- [Using DataSourceUtils](#)
- [Implementing SmartDataSource](#)
- [Extending AbstractDataSource](#)
- [Using SingleConnectionDataSource](#)
- [Using DriverManagerDataSource](#)
- [Using TransactionAwareDataSourceProxy](#)
- [Using DataSourceTransactionManager](#)

### 3.4.1. 使用数据源

Spring 通过 `DataSource` 获得与数据库的连接。 `DataSource` 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 Management 问题。作为开发人员，您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 Management 员的责任

。您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，您可以从 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。

本节使用 Spring 的 `DriverManagerDataSource` 实现，稍后将介绍其他一些实现。

### iNote

您只能将 `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`：

- 通常会获得 JDBC 连接，因此获得与 `DriverManagerDataSource` 的连接。
- 指定 JDBC 驱动程序的标准类名，以便 `DriverManager` 可以加载驱动程序类。
- 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。)
- 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:hsq://localhost:");
dataSource.setUsername("sa");
dataSource.setPassword("");
```

以下示例显示了相应的 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"/>
```

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="c
  <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"/>
```

以下示例显示了 C3P0 配置：

```
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource" destroy-method="c
  <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 `static` 个方法以从 JNDI 获取连接并在必要时关闭连接。它支持例如 `DataSourceTransactionManager` 的线程绑定连接。

### 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，以便使用它的类查询给定操作后是否应关闭连接。当您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 `abstract` Base Class。它实现了所有 `DataSource` 实现通用的代码。如果编写自己的 `DataSource` 实现，则应扩展 `AbstractDataSource` 类。

### 3.4.5. 使用 SingleConnectionDataSource

`SingleConnectionDataSource` 类是 `SmartDataSource` 接口的实现，该接口包装单个 `Connection`，每次使用后都不会关闭。这不是多线程功能。

如果假定共享连接(例如使用持久性工具)时，任何 Client 端代码都调用 `close`，则应将 `suppressClose` 属性设置为 `true`。此设置将返回用于封装物理连接的关闭抑制代理。请注意，您不能再将此对象转换为本地 Oracle `Connection` 或类似的对象。

`SingleConnectionDataSource` 主要是测试班。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

`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很有用，可以作为 Spring IoC 容器中的 `DataSource` bean 或与简单的 JNDI 环境结合使用。池假设 `Connection.close()` 调用将关闭连接，因此任何 `DataSource` 感知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-dbc`)也是如此容易，以至总是总是比 `DriverManagerDataSource` 更好地使用这样的连接池。

### 3.4.7. 使用 TransactionAwareDataSourceProxy

`TransactionAwareDataSourceProxy` 是目标 `DataSource` 的代理。代理包装该目标 `DataSource` 以增加对 SpringManagement 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 `DataSource`。

#### iNote

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，您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 Management 抽象来编写自己的新代码，例如 `JdbcTemplate` 或 `DataSourceUtils`。

有关更多详细信息，请参见[TransactionAwareDataSourceProxy javadoc](#)。

### 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。

需要应用程序代码才能通过 `DataSourceUtils.getConnection(DataSource)` 而不是 Java EE 的 标准 `DataSource.getConnection` 检索 JDBC 连接。它引发未检查的 `org.springframework.dao` 异常，而不是已检查的 `SQLExceptions`。所有框架类(例如 `JdbcTemplate`)都隐式使用此策略。如果不与该事务 Management 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，应用程序代码必须对每个创建的语句使用 `JdbcTemplate` 或调用 `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。只要您坚持要求的连接查找模式，则在两者之间进行切换仅是配置问题。JTA 不支持自定义隔离级别。

## 3.5. JDBC 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

通过实现一个特殊接口 `BatchPreparedStatementSetter` 的两个方法，并将该实现作为 `batchUpdate` 方法调用中的第二个参数传入，可以完成 `JdbcTemplate` 批处理。您可以使用 `getBatchSize` 方法提供当前批处理的大小。您可以使用 `setValues` 方法设置准备好的语句的参数。该方法称为您在 `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，您可以使用 `InterruptibleBatchPreparedStatementSetter` 界面，一旦 Importing 源用尽，您就可以中断批处理。`isBatchExhausted` 方法可让您发出批处理结束的 signal。

### 3.5.2. 具有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。框架循环这些值，并使用内部准备好的语句设置器。API 会有所不同，具体取决于您是否使用命名参数。对于命名参数，您提供一个 `SqlParameterSource` 数组，每个批次成员一个条目。您可以使用 `SqlParameterSourceUtils.createBatch` 便捷方法创建此数组，传入一个 Bean 样式对象(带有与参数相对应的 getter 方法)，`String`-keyed `Map` 实例(包含对应的参数作为值)或它们的混合的数组。

以下示例显示使用命名参数的批处理更新：

```
public class JdbcActorDao implements ActorDao {  
  
    private NamedParameterTemplate namedParameterJdbcTemplate;  
  
    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  
            SqlParameterSourceUtils.createBatch(actors));  
    }  
  
    // ... additional methods  
}
```

对于使用经典 `?` 占位符的 SQL 语句，您传入一个包含带有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具有一个条目，并且它们的 Sequences 必须与 SQL 语句中定义的 Sequences 相同。

除使用经典的 JDBC `?` 占位符外，以下示例与上述示例相同：

```
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`。

### iNote

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 `null` 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，您应该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 Classpath 根目录中的 `spring.properties` 文件中)，例如，如 Oracle 12c(SPR)所述-16139)。

或者，您可以考虑通过“BatchPreparedStatementSetter”(如前所示)，通过给基于“

List<Object[]>"的调用的显式类型数组，通过在自定义“MapSqlParameterSource”上的“registerSqlType”调用来显式指定相应的 JDBC 类型。'实例，或者通过' BeanPropertySqlParameterSource'从 Java 声明的属性类型派生 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

前面的批处理更新示例处理的批处理太大，以至于您想将它们分成几个较小的批处理。您可以通过多次调用 `batchUpdate` 方法来使用前面提到的方法，但是现在有一个更方便的方法。除了 SQL 语句外，此方法还使用 `Collection` 个对象，这些对象包含参数，每个批次要进行的更新次数以及 `ParameterizedPreparedStatementSetter` 来设置已准备语句的参数值。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` 数组的数组，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。顶级数组的长度指示已执行的批处理数，第二级数组的长度指示该批处

理中的更新数。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`。

## 3.6. 使用 SimpleJdbc 类简化 JDBC 操作

`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，但是如果您愿意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 `SimpleJdbcInsert` 类。而是可以创建一个新实例并使用 `withTableName` 方法设置表名称。此类的配置方法遵循 `fluid` 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  
}
```

此处使用的 `execute` 方法将普通 `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内容, 但是它没有传递 `id`, 而是检索自动生成的密钥并将其设置在新的 `Actor` 对象上。当创建 `SimpleJdbcInsert` 时, 除了指定表名之外, 还使用 `usingGeneratedKeyColumns` 方法指定生成的键列的名称。以下 `Lists` 显示了它的工作方式:

```
public class JdbcActorDao implements ActorDao {  
  
    private JdbcTemplate jdbcTemplate;  
    private SimpleJdbcInsert insertActor;  
  
    public void setDataSource(DataSource dataSource) {  
        this.jdbcTemplate = new JdbcTemplate(dataSource);  
        this.insertActor = new SimpleJdbcInsert(dataSource)  
            .withTableName("t_actor")  
            .usingGeneratedKeyColumns("id");  
    }  
  
    public void add(Actor actor) {  
        Map<String, Object> parameters = new HashMap<String, Object>(2);  
        parameters.put("first_name", actor.getFirstName());  
        parameters.put("last_name", actor.getLastName());  
        Number newId = insertActor.executeAndReturnKey(parameters);  
        actor.setId(newId.longValue());  
    }  
  
    // ... additional methods  
}
```

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`, 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 Base Class。如果您有多个自动生成的列或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

您可以通过使用 `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，但这并不是最方便使用的类。Spring 提供了

`SqlParameterSource` 接口的几种实现，您可以代替使用它们。第一个是

`BeanPropertySqlParameterSource`，如果您有一个包含值的 JavaBean 兼容类，这是一个非常

方便的类。它使用相应的 `getter` 方法提取参数值。以下示例显示了如何使用

`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");
    }

    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`，但提供了更方便的 `addValue` 方法，可以将其链接。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 Importing 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，也可以声明没有自动 Map 到 Java 类的参数(例如 `ARRAY` 或 `STRUCT`)。第一个示例显示了一个简单过程，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 `actor` 条目，并以 `out` 参数的形式返回 `first_name`，`last_name` 和 `birth_date` 列。以下 Lists 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演员的 `id`。 `out` 参数返回从表读取的数据。

您可以采用类似于声明 `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(除了 `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
}

```

您为执行调用而编写的代码涉及创建一个包含 `IN` 参数的 `SqlParameterSource`。您必须为 `Importing` 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，而另一些数据库使用小写或指定的大写。

`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任何 `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，使用在存储过程中声明的 `out` 参数的名称也很重要。同样，结果 `Map` 中存储的 `out` 参数名称的大小写与数据库中 `out` 参数名称的大小写匹配，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的 `out` 参数名称的情况下发生冲突。

### 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 `declareParameters` 方法创建和配置 `SimpleJdbcCall` 来实现，该方法采用可变数量的 `SqlParameter` 对象作为 Importing。有关如何定义 `SqlParameter` 的详细信息，请参见[next](#)

[section](#)。

### iNote

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 的存储函数的元数据查找。

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，可以将方法

`withoutProcedureColumnMetaDataAccess` 作为声明的一部分进行调用。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` 以指定要包含在给定签名中的 IN 参数名称列表。

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。第二个示例显式指定所有详细信息，而不是依赖于元数据。

## 3.6.7. 如何定义 `SqlParameters`

要为 `SimpleJdbc` 类以及 RDBMS 操作类(在[将 JDBC 操作建模为 Java 对象](#)中介绍)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(在[Understanding SqlQuery](#)中介绍)，可以将 IN 参数用于存储过程调用和查询。

第二行(带有 `SqlOutParameter`)声明要在存储过程调用中使用的 `out` 参数。还有 `SqlInOutParameter` 表示 `InOut` 参数(为过程提供 IN 值并返回值的参数)。

#### ①Note

仅声明为 `SqlParameter` 和 `SqlInOutParameter` 的参数用于提供 Importing 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 Importing 值。

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，您可以提供 `RowMapper` 来处理从 `REF` 游标返回的行的 Map。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，以指示您要对函数进行调用，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返回值，这意味着您不必从结果 Map 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)基于名为 `get_actor_name` 的存储函数, 该函数返回参与者的全名:

```
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 = 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
}
```

所使用的 `executeFunction` 方法返回一个 `String`, 其中包含该函数调用的返回值。

### 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 `out` 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。使用 `SimpleJdbcCall`, 您可以使用 `returningResultSet` 方法并声明

`RowMapper` 实现用于特定参数。如果在结果处理期间返回了结果集，则没有定义任何名称，因此返回的结果必须与声明 `RowMapper` 实现的 `Sequences` 匹配。指定的名称仍用于将处理后的结果列表存储在从 `execute` 语句返回的结果 `Map` 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`。因为您要 Map 到的类遵循 JavaBean 规则，所以可以使用通过在 `newInstance` 方法中传入要 Map 的必需类而创建的 `BeanPropertyRowMapper`。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<Actor> getActorsList() {
        Map<String, Object> map = procReadAllActors.execute(new HashMap<String, Object>(0));
        return (List<Actor>) map.get("actors");
    }

    // ... additional methods
}
```

`execute` 调用传入空的 `Map`，因为该调用没有任何参数。然后从结果图中检索参与者列表，并将其返回给调用者。

## 3.7. 将 JDBC 操作建模为 Java 对象

`org.springframework.jdbc.object` 软件包包含一些类，这些类使您可以以更加面向对象的方式访问数据库。例如，您可以执行查询并将结果作为包含业务对象的列表返回，该业务对象的关系列数据 Map 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。

#### iNote

许多 Spring 开发人员认为，下面描述的各种 RDBMS 操作类(但[StoredProcedure](#)类除外)通常可以被直接 `JdbcTemplate` 调用替换。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，如果通过使用 RDBMS 操作类获得可测量的价值，则应 continue 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(...)` 方法来提供 `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。`SqlQuery` 类很少直接使用，因为 `MappingSqlQuery` 子类为将行 Map 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，其中具体子类必须实现抽象 `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，该查询将数据从 `t_actor` 关系 Map 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;
}

}

```

该类扩展了用 `Actor` 类型参数化的 `MappingSqlQuery`。此 Client 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 `DataSource` 和应执行以检索此查询的行的 SQL 调用超类上的构造函数。该 SQL 用于创建 `PreparedStatement`，因此它可以包含在执行期间要传递的任何参数的占位符。您必须使用 `SqlParameter` 传递的 `declareParameter` 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，以便可以准备该语句并在以后运行。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) {
    this.actorMappingQuery = new ActorMappingQuery(dataSource);
}

public Customer getCustomer(Long id) {
    return actorMappingQuery.findObject(id);
}

```

前面示例中的方法使用传入的 `id` 作为唯一参数来检索 Client。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 `execute` 方法之一，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 `update(..)` 方法，类似于查询对象的 `execute(..)` 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，不必继承 `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 `abstract`，并且其各种 `execute(..)` 方法具有 `protected` 访问权限，除了通过提供更严格的键入的子类之外，其他都禁止使用。

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：

```
new SqlParameter("in_id", Types.NUMERIC),  
new SqlOutParameter("out_first_name", Types.VARCHAR),
```

SQL 类型使用 `java.sql.Types` 常量指定。

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(在[Understanding SqlCommand](#)中覆盖)的查询。

第二行(带有 `SqlOutParameter`)声明了要在存储过程调用中使用的 `out` 参数。还有

`SqlInOutParameter` 表示 `InOut` 参数(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，您可以提供 `RowMapper` 来处理从 `REF` 游标返回的行的 `Map`。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 数据库随附的函数(`sysdate()`)

。要使用存储过程功能，您必须创建一个扩展 `StoredProcedure` 的类。在此的示例

`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。本示例没有 `Importing` 参数，但是使用 `SqlOutParameter` 类将输出参数声明为日期类型。`execute()` 方法运行该过程，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(在本例中为一个)都有一个条目。以下 `Lists` 显示了我们自定义 `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;
        }
    }
}

```

下面的 `StoredProcedure` 示例具有两个输出参数(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 TitleMap());
        declareParameter(new SqlOutParameter("genres", OracleTypes.CURSOR, new GenreMap());
        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 `RowMapper` 实现实例中传递在 `TitlesAndGenresStoredProcedure` 构造函数中使用的 `declareParameter(..)` 方法的重载变体。这是重用现有功能的非常方便且强大的方法。

接下来的两个示例提供了两个 `RowMapper` 实现的代码。

对于提供的 `ResultSet` 中的每一行，`TitleMapper` 类将 `ResultSet` Map 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;
    }
}

```

对于提供的 `ResultSet` 中的每一行，`GenreMapper` 类将 `ResultSet` Map 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"));
    }
}

```

要将参数传递给 RDBMS 中定义中具有一个或多个 Importing 参数的存储过程，可以编写一个强类型的 `execute(..)` 方法，该方法将委派给超类中的非类型 `execute(Map)` 方法，如以下示例所示：

```
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);
    }
}
```

## 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 `SqlParameter` 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。所有这些方法都将 `LobHandler` 接口的实现用于 LOB(大对象)数据的实际 Management。`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 LOBImporting 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 : `getBlobAsCharacterStream` 和 `setBlobAsCharacterStream`

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 `LobCreator`，我们可以用来设置 SQL 插入语句中 LOB 列的值。

对于此示例，我们假设存在一个变量 `lobHandler`，该变量已设置为 `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(
    "INSERT INTO lob_table (id, a_clob, a_blob) VALUES (?, ?, ?)",
    new AbstractLobCreatingPreparedStatementCallback(lobHandler) { (1)
        protected void setValues(PreparedStatement ps, LobCreator lobCreator) throws SQ
            ps.setLong(1, 1L);
            lobCreator.setBlobAsCharacterStream(ps, 2, clobReader, (int)clobIn.length());
            lobCreator.setBlobAsBinaryStream(ps, 3, blobIs, (int)blobIn.length()); (3)
        }
    );
}

blobIs.close();
clobReader.close();
```

- (1) 传递 `lobHandler` (在此示例中) 为普通 `DefaultLobHandler`。
- (2) 使用方法 `setBlobAsCharacterStream` 传递 CLOB 的内容。
- (3) 使用方法 `setBlobAsBinaryStream` 传入 BLOB 的内容。

### ①Note

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数，则

`DefaultLobHandler` 使用不带 `length` 参数的 `set-stream` 方法的 JDBC 4.0 变体。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，您使用具有相同实例变量 `lobHandler` 的

`JdbcTemplate` 和对 `DefaultLobHandler` 的引用。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"); (1)
        results.put("CLOB", clobText);
        byte[] blobBytes = lobHandler.getBlobAsBytes(rs, "a_blob"); (2)
        results.put("BLOB", blobBytes);
        return results;
    }
});
```

- (1) 使用方法 `getClobAsString` 检索 CLOB 的内容。
- (2) 使用方法 `getBlobAsBytes` 检索 BLOB 的内容。

### 3.8.3. 传入 IN 子句的值列表

SQL 标准允许根据包含变量值列表的表达式选择行。一个典型的例子是 `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。

## iNote

传递许多值时要小心。 JDBC 标准不能保证 `in` 表达式列表可以使用 100 个以上的值。各种数据库都超过了此数目，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 `java.util.List` 对象数组。该列表可以支持为 `in` 子句定义的多个表达式，例如 `select * from T_ACTOR where (id, last_name) in ((1, 'Johnson'), (2, 'Harrop'))`。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。为了适应这些类型，Spring 提供了一个 `SqlReturnType` 来处理它们(从存储过程调用中返回)，并提供 `SqlTypeValue` 并将它们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 `getSqlTypeValue`)。此接口用作 `SqlOutParameter` 声明的一部分。以下示例显示返回声明为 `ITEM_TYPE` 类型的用户的 Oracle `STRUCT` 对象的值：

```
public class TestItemStoredProcedure extends StoredProcedure {  
  
    public TestItemStoredProcedure(DataSource dataSource) {  
        ...  
        declareParameter(new SqlOutParameter("item", OracleTypes.STRUCT, "ITEM_TYPE",  
            new SqlReturnType() {  
                public Object getSqlTypeValue(CallableStatement cs, int colIndx, int sqlTy  
                    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。以下示例创建一个 `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 `execute` 调用的 `Importing` 参数的 `Map`。

`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 具有自己的内部 `ARRAY` 类, 您可以使用 `SqlTypeValue` 创建 Oracle `ARRAY` 的实例, 并使用 Java `ARRAY` 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;
    }
};
```

## 3.9. 嵌入式数据库支持

`org.springframework.jdbc.datasource.embedded` 软件包提供对嵌入式 Java 数据库引擎的支持。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，启动时间短，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` 名称空间中使用 `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，该数据库使用来自 Classpath 根目录中 `schema.sql` 和 `test-data.sql` 资源的 SQL 进行填充。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 对于 Spring 容器可用，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，可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()
```

有关所有支持的选项的更多详细信息，请参见[用于 EmbeddedDatabaseBuilder 的 javadoc](#)。

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)
            .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：

- [Using HSQL](#)
- [Using H2](#)
- [Using Derby](#)

#### Using 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。要显式指定 HSQL，请将 `embedded-database` 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。

#### Using H2

Spring 支持 H2 数据库。要启用 H2, 请将 `embedded-database` 标签的 `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。

## Using Derby

Spring 支持 Apache Derby 10.5 及更高版本。要启用 Derby, 请将 `embedded-database` 标签的

`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, 如果要创建在测试套件中共享的嵌入式数据库, 请考虑使用[Spring TestContext 框架](#)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[以编程方式创建嵌入式数据库](#)中所述。以下 Lists 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( /* ... */ );  
    }  
  
    @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)中的多个测试场景中重复使用相应的配置，则这很容易发生。例如，集成测试针对 `ApplicationContext` 配置仅在哪个 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。

此类错误的根本原因是，如果没有另外指定，Spring 的 `EmbeddedDatabaseFactory` (由 `<jdbc:embedded-database>` XML 名称空间元素和 `EmbeddedDatabaseBuilder` 用于 Java 配置内部使用)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与 Bean 的 `id` 相同的名称(通常为 `dataSource` 之类的名称)。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。要启用生成名称的使用，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。

- 实施 `DataSourceFactory` 以支持新的 `DataSource` 实施，例如用于 Management 嵌入式数据库连接的连接池。

我们鼓励您在 [jira.spring.io](#) 向 Spring 社区提供扩展。

## 3.10. 初始化数据源

`org.springframework.jdbc.datasource.init` 软件包为初始化现有的 `DataSource` 提供支持。嵌入式数据库支持提供了一个为应用程序创建和初始化 `DataSource` 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，则可以使用 `spring-jdbc` 命名空间中的 `initialize-database` 标记：

```
<jdbc:initialize-database data-source="dataSource">
    <jdbc:script location="classpath:com/foo/sql/db-schema.sql"/>
    <jdbc:script location="classpath:com/foo/sql/db-test-data.sql"/>
</jdbc:initialize-database>
```

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。脚本位置也可以是带有通配符的模式，该通配符具有用于 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 Sequences 运行。

数据库初始化程序的默认行为是无条件运行提供的脚本。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。通过遵循首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}"> (1)
    <jdbc:script location="..." />
</jdbc:initialize-database>
```

- (1) 从名为 `INITIALIZE_DATABASE` 的系统属性获取 `enabled` 的值。

控制现有数据会发生什么的第二种选择是更加容忍失败。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` 语句将因此失败。因此，失败的 SQL `DROP` 语句将被忽略，但其他失败将导致异常。如果您的 SQL 方言不支持 `DROP ... IF EXISTS` (或类似值)，但是您希望在重新创建它之前无条件地删除所有测试数据，这将很有用。在这种情况下，第一个脚本通常是一组 `DROP` 语句，然后是一组 `CREATE` 语句。

`ignore-failures` 选项可以设置为 `NONE` (默认设置)，`DROPS` (忽略失败的放置) 或 `ALL` (忽略所有失败)。

如果脚本中根本没有 `;` 字符，则每个语句应用 `;` 或换行符分隔。您可以全局控制该脚本，也可以逐个脚本控制脚本，如以下示例所示：

```
<jdbc:initialize-database data-source="dataSource" separator="@@"> (1)
    <jdbc:script location="classpath:com/myapp/sql/db-schema.sql" separator=";" /> (2)
    <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>
```

- (1) 将分隔符脚本设置为 `@@`。
- (2) 将 `db-schema.sql` 的分隔符设置为 `;`。

在此示例中，两个 `test-data` 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。

数据库初始化程序取决于 `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-method`，组件中的 `@PostConstruct` 方法或实现 `InitializingBean` 的组件中的 `afterPropertiesSet()` 方法)。如果其他 bean 依赖于相同的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。一个常见的示例是一个高速缓存，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。有关如何实现此目的的一些建议包括：

- 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 `autoStartup` 标志来自动启动 `SmartLifecycle`，并且可以通过在封闭上下文中调用 `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` 随时可供使用(在初始化所有 bean 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 Sequences 初始化 bean。通过采用 XML 配置中一组 `<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，您可以轻松地进行安排。
- 将 `DataSource` 和使用它的业务组件分开，并通过将它们放在单独的 `ApplicationContext` 实例中来控制其启动 Sequences(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，但可以更广泛地应用。

## 4. 对象关系 Map(ORM)数据访问

---

本节介绍使用对象关系 Map(ORM)时的数据访问。

### 4.1. Spring ORM 简介

Spring Framework 支持与 Java Persistence API(JPA)集成，并支持用于资源 Management，数据访问对象(DAO)实现和事务策略的本地 Hibernate。例如，对于 Hibernate，提供了一流的支持以及一些便捷的 IoC 功能，这些功能可以解决许多典型的 Hibernate 集成问题。您可以通过“依赖关系注入”为 OR(对象关系)Map 工具配置所有受支持的功能。他们可以参与 Spring 的资源和事务 Management，并且符合 Spring 的通用事务和 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，因为所有内容都是作为一组可重用的 JavaBean 设计的。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：

- 更轻松的测试。Spring 的 IoC 方法使交换 Hibernate `SessionFactory` 实例，JDBC `DataSource` 实例，事务 Management 器和 Map 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的测试变得容易得多。

- **通用数据访问异常.** 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 Management.** 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 Management 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` 公开当前的 `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。
- **集成的事务 Management.** 您可以通过 `@Transactional` Comments 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)样式方法拦截器包装 ORM 代码。在这两种情况下，都为您处理事务语义和异常处理(回滚等)。如资源与 TransactionManagement中所述，您还可以交换各种事务 Management 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(例如声明式事务)在本地事务和 JTA 之间交换。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。

### Tip

要获得更全面的 ORM 支持，包括对 MongoDB 等替代数据库技术的支持，您可能需要查看 Spring Data 项目套件。如果您是 JPA 用户，则<https://spring.io>的使用 JPA 访问数据的入门

指南提供了很好的介绍。

## 4.2. ORM 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](#) 部分提供更多详细信息，并在具体上下文中显示这些功能和配置。

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 - 不再对数据访问或事务策略依赖业务服务，不再进行硬编码资源查找，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，并尽可能避免容器依赖性。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 意识的纯 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：数据访问模板，数据访问对象，事务 Management 器，使用数据访问对象和事务 Management 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. 资源与 TransactionManagement

典型的业务应用程序中充斥着重复的资源 Management 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，[previous section](#) 中提到的 `JdbcTemplate` 类提供了连接处理和 `SQLException` 到 `DataAccessException` 层次结构的正确转换，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，请参见[next section](#) 以获取相同的异常翻译好处。

在事务 Management 方面，`JdbcTemplate` 类通过相应的 Spring 事务 Management 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 Management 器提供了 Hibernate 和 JPA 支持以及 JTA 支持。有关 Transaction 支持的

详细信息, 请参见[Transaction Management](#)章。

## 4.2.2. exception 翻译

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。DAO 抛出 `HibernateException` 或 `PersistenceException` 的子类, 具体取决于技术。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, 除非他们希望依赖于持久性技术自身的异常结构。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(或两者都使用)的应用程序是可接受的。但是, Spring 允许通过 `@Repository` Comments 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...
}
```

```
<beans>

    <!-- Exception translation bean post processor -->
    <bean class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor" />

    <bean id="myProductDao" class="product.ProductDaoImpl" />

</beans>
```

后处理器自动查找所有异常翻译器(`PersistenceExceptionTranslator` 接口的实现), 并建议所有带有 `@Repository` 注解标记的 bean, 以便发现的翻译器可以拦截并对抛出的异常应用适当的翻译。

总而言之, 您可以基于纯持久性技术的 API 和 Comments 来实现 DAO, 同时仍可受益于 SpringManagement 的事务, 依赖项注入以及对 Spring 的自定义异常层次结构的透明异常转换(如果需要)。

## 4.3. Hibernate

我们从 Spring 环境中的 [Hibernate 5](#) 开始，使用它来演示 Spring 集成 ORMap 器所采用的方法。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。

### iNote

从 Spring Framework 5.0 开始，Spring 需要 Hibernate ORM 4.3 或更高版本来支持 JPA，甚至需要 Hibernate ORM 5.0 来针对本机 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 `sched` 义实例的引用，如[next section](#) 中的 DAO 定义所示。

XML 应用程序上下文定义的以下摘录显示了如何在其之上设置 JDBC `DataSource` 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:hsq1://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">
            <refBean name="hibernateProperties"/>
        </property>
    </bean>
```

```
<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` (通常由应用服务器 Management)只是配置问题，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>` 来访问位于 JNDI 的 `SessionFactory`，以检索并公开它。但是，这通常在 EJB 上下文之外并不常见。

### iNote

Spring 还提供了 `LocalSessionFactoryBuilder` 变体，可与 `@Bean` 样式配置和编程设置(不涉及 `FactoryBean`)无缝集成。

`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，这样的本地 Hibernate 设置还可以在本地 Hibernate 访问旁边公开用于标准 JPA 交互的 JPA `EntityManagerFactory`。有关详情，请参见[JPA 的本机 Hibernate 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 本身每个事务 Management 一个当前的 Session。这大致相当于 Spring 对每个事务同步一个 Hibernate Session。基于普通的 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();  
    }  
}
```

该样式与 Hibernate 参考文档和示例的样式类似，不同之处在于将 SessionFactory 保留在实例变量中。我们强烈建议在 Hibernate 的 CaveatEmptor 示例应用程序中的老式 static 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 static 变量中保留任何资源。）

前面的 DAO 示例遵循依赖项注入模式。它很好地适合于 Spring IoC 容器，就像针对 Spring 的 HibernateTemplate 进行编码一样。您还可以在纯 Java 中设置这种 DAO（例如，在单元测试中）。为此，请对其进行实例化并使用所需的工厂参考调用 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 会抛出普通的 `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 事务策略的 Hibernate 的 `SessionFactory.getCurrentSession()` 方法，甚至返回 `HibernateTransactionManager`，返回当前的 `SpringManagement` 的事务 `Session`。该方法的标准行为仍然是返回与正在进行的 JTA 事务关联的当前 `Session` (如果有)。无论您使用 Spring 的 `JtaTransactionManager`，EJB 容器 Management 的事务(CMT)还是 JTA，此行为均适用。

总之，您可以基于普通的 Hibernate API 实现 DAO，同时仍然能够参与 `SpringManagement` 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，该支持使您可以用 AOP 事务拦截器替换 Java 代码中的显式事务划分 API 调用。您可以使用 Java 注解或 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 true 价值。

#### iNote

在 continue 之前，我们强烈建议您阅读[声明式 TransactionManagement](#)(如果您尚未阅读的话)。

您可以使用 `@Transactional` Comments 对服务层进行 Comments，并指示 Spring 容器查找这些 Comments 并为这些带 Comments 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
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/tx
           http://www.springframework.org/schema/tx/spring-tx.xsd
           http://www.springframework.org/schema/aop
           http://www.springframework.org/schema/aop/spring-aop.xsd">

    <!-- SessionFactory, DataSource, etc. omitted -->

    <bean id="transactionManager"
          class="org.springframework.orm.hibernate5.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory"/>
    </bean>

    <tx:annotation-driven/>

    <bean id="myProductService" class="product.SimpleProductService">
        <property name="productDao" ref="myProductDao"/>
    </bean>

</beans>

```

#### 4.3.4. 程序化 Transaction 划分

您可以在应用程序的较高级别中划分事务，而这些较低级别的数据访问服务可以跨越任意数量的操作。

作。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。同样，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 Management 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` 仅限于回调中的未检查的异常。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` 触发回滚。默认情况下，`TransactionInterceptor` 的行为方式相同，但允许每个方法配置可回退策略。

### 4.3.5. TransactionManagement 策略

`TransactionTemplate` 和 `TransactionInterceptor` 都将实际的事务处理委托给 `PlatformTransactionManager` 实例(对于内部使用 `ThreadLocal` `Session` 可以是 `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 `PlatformTransactionManager` 实现。从本机 Hibernate 事务 Management 切换到 JTA(例如，当面对某些应用程序部署的分布式事务要求时)仅是配置问题。您可以用 Spring 的 JTA 事务实现替换 Hibernate 事务 Management 器。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 Management API。

对于跨多个 Hibernate 会话工厂的分布式事务，可以将 `JtaTransactionManager` 作为事务策略与多个 `LocalSessionFactoryBean` 定义结合使用。然后，每个 DAO 都将一个特定的 `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 和任意数量的会话工厂之间划分事务。

`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 Management 器查找或 JCA 连接器(如果您不使用 EJB 来启动事务)。

`HibernateTransactionManager` 可以将 Hibernate JDBC `Connection` 导出为特定 `DataSource` 的普通 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，而完全无需 JTA，前提是您仅访问一个数据库。如果已通过 `LocalSessionFactoryBean` 类的 `dataSource` 属性使用 `DataSource` 设置了传入的 `SessionFactory`，则 `HibernateTransactionManager` 自动将 Hibernate 事务公开为 JDBC 事务。或者，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 Management 的资源和本地定义的资源

您可以在容器 Management 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 Management 基础结构的附加价值，但不会增加实际价值。

Spring 的事务支持未绑定到容器。当配置了 JTA 以外的任何策略时，事务支持也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 `EJBStateless` 会话 Bean 驱动事务时，即使您仅访问单个数据库并且仅使用 `Stateless` 会话 Bean 通过容器 Management 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 的，由 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` 以及本地 JDBC `DataSource` 一起工作。因此，当您具有分布式事务需求时，仅需要使用 Spring 的 JTA 事务策略。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)中运行，例如 Tomcat, Resin 甚至是普通 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。

考虑到所有问题，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 Pseudo 应用程序服务器警告

在某些具有非常严格的 `XADatasource` 实现的 JTA 环境中(当前仅某些 WebLogic Server 和 WebSphere 版本)，在不考虑该环境的 JTA `PlatformTransactionManager` 对象的情况下配置 Hibernate 时，虚假警告或异常会显示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.
```

您可以通过使 Hibernate 知道与之同步的 JTA `PlatformTransactionManager` 实例(以及 Spring)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 JTA `PlatformTransactionManager` 对象(大概是通过 `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定义此 JTA 的 bean 的引用 `PlatformTransactionManager` 实例作为 `LocalSessionFactoryBean`. Spring 的 `jtaTransactionManager` 属性的值，然后使该对象可用于 Hibernate。
- 更可能的是，您还没有 JTA `PlatformTransactionManager` 实例，因为 Spring 的 `JtaTransactionManager` 可以自己找到它。因此，您需要配置 Hibernate 以直接查找 JTA `PlatformTransactionManager`。通过在 Hibernate 配置中配置特定于应用程序服务器的 `TransactionManagerLookup` 类来执行此操作，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的 Sequences。

如果未配置 Hibernate 来识别 JTA `PlatformTransactionManager`，则在 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 Management 器通过 `afterCompletion` 回调对其进行回调。
- 在其他活动中，这种同步可以触发 Spring 通过 Hibernate 的 `afterTransactionCompletion` 回调(用于清除 Hibernate 缓存)来回调到 Hibernate，然后在 Hibernate 会话上进行显式 `close()` 调用，这将导致 Hibernate 尝试 `close()` JDBC Connection。
- 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` 感知能力时，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 Management 器通过 `beforeCompletion` 回调回调了该事务。
- Spring 知道，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 `Session`，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 Management 器通过 `afterCompletion` 回调调用了该事务，并可以正确清除其缓存。

## 4.4. JPA

可以在 `org.springframework.orm.jpa` 软件包下获得的 Spring JPA 以类似于与 Hibernate 集成的方式为 [Java 持久性 API](#) 提供全面的支持，同时知道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` 的方式，供应用程序用来获取实体 Management 器。

- [Using LocalEntityManagerFactoryBean](#)
- [从 JNDI 获取 EntityManagerFactory](#)
- [Using LocalContainerEntityManagerFactoryBean](#)

#### Using 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>
```

JPA 部署的这种形式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，从而允许使用不同于服务器默认值的提供程序。

从 JNDI(例如在 Java EE 环境中)获取 `EntityManagerFactory` 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 服务器自动检测持久性单元(实际上是应用程序 jar 中的 `META-INF/persistence.xml` 个文件)和 Java EE 部署 Descriptors 中的 `persistence-unit-ref` 个条目(例如 `web.xml`)，并为这些持久性单元定义环境命名上下文位置。

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 DataSource` 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`，通过依赖性注入将其传递给应用程序对象，并 Management 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` 注解中)。

## Using LocalContainerEntityManagerFactoryBean

您可以将此选项用于基于 Spring 的应用程序环境中的完整 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。

## iNote

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。有关详情，请参见[用于 JPA 交互的本地 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>
```

以下示例显示了一个典型的 `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>
```

## iNote

`<exclude-unlisted-classes/>` 快捷方式指示不应进行对带 Comments 的实体类的扫描。显式的“true”值(`<exclude-unlisted-classes>true</exclude-unlisted-`

`classes/>`)也表示不进行扫描。 `<exclude-unlisted-classes>false</exclude-`

`unlisted-classes/>` 确实触发了扫描。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 `DataSource` 的链接，同时支持本地和全局事务，等等。但是，它还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，则具有可编织类加载器的可用性。

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` 定义上指定一个自定义 `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 Descriptors。因为 Java EE 服务器仅查找默认的 `META-INF/persistence.xml` 文件，所以它忽略了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。休眠是一个没有的例子。如果您的提供程序不需要代理，或者您有其他选择，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，可让 JPA `ClassTransformer` 实例以特定方式插入，具体取决于环境是 Web 容器还是应用程序服务器。通过 `agent` 钩子 `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。

有关 `LoadTimeWeaver` 实现及其设置的更多信息, 请参见 AOP 章节中的[Spring configuration](#), 这

些实现是通用的还是针对各种平台(例如 Tomcat, WebLogic, GlassFish, Resin 和 JBoss)定制的。

如[Spring configuration](#)中所述, 您可以使用 `context:load-time-weaver` XML 元素的

`@EnableLoadTimeWeaving` 注解来配置上下文范围内的 `LoadTimeWeaver`。所有 JPA

`LocalContainerEntityManagerFactoryBean` 实例都会自动拾取这样的全局编织器。以下示例显

示了设置加载时织构的首选方法, 该方法可自动检测平台(WebLogic

, GlassFish, Tomcat, Resin, JBoss 或 VM 代理), 并将织构自动传播到所有可识别织构的 bean

:

```
<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, 这一点尤其重要, 因为 JPA 转换器仅应用于类加载器级别, 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 Classpath 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, 这可能是昂贵的。

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 Classpath 中搜索 `META-INF/persistence.xml` 个文件.)以下示例配置多个位置:

```
<bean id="pum" class="org.springframework.orm.jpa.persistenceunit.DefaultPersistenceUnitManager">
    <persistenceUnits>
        <persistenceUnit name="PU1" transaction-type="JTA"/>
        <persistenceUnit name="PU2" transaction-type="JTA"/>
    </persistenceUnits>
</bean>
```

```

<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(通过允许持久化单元选择的 `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` 在内部创建和使用一个。

## Background Bootstrapping

`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` 代理可以注入到其他应用程序组件中，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`)，这些调用将阻塞，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

##### 1 Note

尽管 `EntityManagerFactory` 个实例是线程安全的, 但 `EntityManager` 个实例不是线程安全的。根据 JPA 规范定义, 注入的 JPA `EntityManager` 的行为类似于从应用程序服务器的 JNDI 环境中获取的 `EntityManager`。它将所有调用委派给当前事务 `EntityManager` (如果有)。否则, 它会退回到每个操作新创建的 `EntityManager`, 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`, 可以针对不带任何 Spring 依赖关系的普通 JPA 编写代码。如果启用了 `PersistenceAnnotationBeanPostProcessor`, 则 Spring 可以在字段和方法级别理解 `@PersistenceUnit` 和 `@PersistenceContext` 注解。以下示例显示了使用 `@PersistenceUnit` `Comments` 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) {  
        EntityManager em = this.emf.createEntityManager();  
        try {  
            Query query = em.createQuery("from Product as p where p.category = ?1");  
            query.setParameter(1, category);  
            return query.getResultList();  
        }  
        finally {  
            if (em != null) {  
                em.close();  
            }  
        }  
    }  
}
```

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 Comments 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>
```

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 Comments 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，它总是在工厂中创建一个新的 `EntityManager`。您可以通过请求注入事务 `EntityManager`（也称为“共享的 EntityManager”，因为它是实际事务 `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` Comments 具有一个名为 `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 `PersistenceContextType.EXTENDED` 是完全不同的事情。这导致所谓的 `EntityManager` 扩展，它不是线程安全的，因此不能在并发访问的组件(例如 SpringManagement 的单例 bean)中使用。扩展的 `EntityManager` 实例仅应用于有状态的组件中，例如，驻留在会话中的状态组件，而 `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 site 级进样

您可以在类中的字段或方法上应用指示依赖项注入的 Comments(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 Comments 简洁明了，易于使用，而方法级 Comments 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。

那么类级 Comments 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 SpringManagement 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 Comments 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。而且，由于可以理解 JPA 注解，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 JPATransaction

#### ① Note

我们强烈建议您阅读[声明式 TransactionManagement](#)(如果您尚未阅读的话), 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 的本机事务支持来进行本地事务。Spring 的 `JpaTransactionManager` 提供了针对任何常规 JDBC 连接池(无需 XA 要求)的本地 JDBC 事务中已知的许多功能(例如, 特定于事务的隔离级别和资源级别的只读优化)。

Spring JPA 还允许已配置的 `JpaTransactionManager` 将 JPA 事务公开给访问同一 `DataSource` 的 JDBC 访问代码, 只要注册的 `JpaDialect` 支持底层 JDBC `Connection` 的检索。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[next section](#)。

#### ①Note

作为直接替代方案, 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。有关详情, 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` 实现通常可以以特定于供应商的方式启用 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 `Connection` (用于公开基于 JDBC 的 DAO)
- `PersistenceExceptions` 到 Spring `DataAccessExceptions` 的高级翻译

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。

### Tip

`JpaVendorAdapter` 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。

分别指定 `HibernateJpaVendorAdapter` 或 `EclipseLinkJpaVendorAdapter` 是分别为 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设计用于与 Spring 驱动的事务 Management 一起使用(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 Management 设置 JPA

作为 `JpaTransactionManager` 的替代方法，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，您还需要采取其他步骤：

- 底层的 JDBC 连接池必须具有 XA 功能，并与事务协调器集成。这在 Java EE 环境中通常很简单，通过 JNDI 公开了另一种 `DataSource`。有关详细信息，请参见您的应用程序服务器文档。类似地，独立事务协调器通常带有特殊的 XA 集成 `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 `on-close`，它与 Hibernate 5.0 中 Hibernate 自己的默认值匹配，但在 5.1/5.2 中不再匹配。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 自己的默认值。有关 WebLogic 的相关说明，请参见[Hibernate 虚 Pseudo 应用程序服务器警告](#)。
- 或者，考虑从应用程序服务器本身获取 `EntityManagerFactory` (即，通过 JNDI 查找而不是本地声明的 `LocalContainerEntityManagerFactoryBean`)。服务器提供的 `EntityManagerFactory` 可能需要在服务器配置中进行特殊定义(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。

#### 4.4.6. 用于 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 `LocalSessionFactoryBean` 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` 本机实现 JPA 的 `EntityManagerFactory` 接口，而 Hibernate `Session` 句柄本机则是 JPA `EntityManager`。Spring 的 JPA 支持工具会自动检测本地 Hibernate 会话。

因此，这种本机 Hibernate 设置可以在许多情况下替代标准 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大的 Hibernate 集成和更大的配置灵活性，因为它不受 JPA 引导 Contract 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` 表示本机 Hibernate 设置，并与 `@Bean` 样式配置(不涉及 `FactoryBean`)无缝集成。

### iNote

`LocalSessionFactoryBean` 和 `LocalSessionFactoryBuilder` 支持后台引导，就像 JPA `LocalContainerEntityManagerFactoryBean` 一样。有关简介，请参见 [Background Bootstrapping](#)。

在 `LocalSessionFactoryBean` 上，可以通过 `bootstrapExecutor` 属性使用。在程序化 `LocalSessionFactoryBuilder` 上，重载的 `buildSessionFactory` 方法采用引导执行程序参数。

## 5. 使用对象 XMLMap 器编组 XML

### 5.1. Introduction

本章描述了 Spring 的 Object-XML Mapping 支持。对象 XMLMap(简称 O-XMap)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-XMap 领域，编组负责将对象(图形)序列化为 XML。以类似的方式，解组器将 XML 反序列化为对象图。该 XML 可以采用 DOM 文档，Importing 或输出流或 SAX 处理程序的形式。

使用 Spring 满足 O/XMap 需求的一些好处是：

- 易于配置

- [Consistent Interfaces](#)
- [一致的异常层次结构](#)

### 5.1.1. 易于配置

Spring 的 bean 工厂使配置编组器变得容易，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-XMap 通过两个全局接口 [Marshaller](#) 和 [Unmarshaller](#) 进行操作。这些抽象使您可以相对轻松地切换 O-XMap 框架，而对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 执行的编组和某些由 Castor 执行的编组)进行 XML 编组，从而使您可以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-XMap 工具的异常到其自己的异常层次的转换，并以 [XmlMappingException](#) 作为根异常。这些运行时异常包装了原始异常，因此不会丢失任何信息。

## 5.2. 马歇尔与非马歇尔

如[introduction](#)中所述，编组器将对象序列化为 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. 了解马歇尔

Spring 在 [org.springframework.oxm.Marshaller](#) 接口之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;  
}
```

**Marshaller** 接口有一个主要方法，该方法将给定对象封装给给定

`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：

Result implementation	包装 XML 表示形式
<code>DOMResult</code>	<code>org.w3c.dom.Node</code>
<code>SAXResult</code>	<code>org.xml.sax.ContentHandler</code>
<code>StreamResult</code>	<code>java.io.File</code> , <code>java.io.OutputStream</code> 或 <code>java.io.Writer</code>

### iNote

尽管 `marshal()` 方法接受一个普通对象作为其第一个参数，但是大多数 **Marshaller** 实现无法处理任意对象。相反，必须将对象类 Map 到 Map 文件中，用 Comments 标记，在编辑器中注册或具有公共 Base Class。请参阅本章后面的部分，以确定 O-X 技术如何 Management 此问题。

## 5.2.2. 了解解组器

与 **Marshaller** 类似，我们具有 `org.springframework.oxm.Unmarshaller` 界面，以下 Lists 显示了该界面：

```
public interface Unmarshaller {
```

```

    /**
     * Unmarshal the given provided Source into an object graph.
     */
    Object unmarshal(Source source) throws XmlMappingException, IOException;
}

```

该接口还有一个方法，该方法从给定的 `javax.xml.transform.Source` (`XMLImporting` 抽象) 中读取并返回读取的对象。与 `Result` 一样，`Source` 是具有三种具体实现的标记接口。每个表都包装了不同的 XML 表示形式，如下表所示：

Source implementation	包装 XML 表示形式
<code>DOMSource</code>	<code>org.w3c.dom.Node</code>
<code>SAXSource</code>	<code>org.xml.sax.InputSource</code> 和 <code>org.xml.sax.XMLReader</code>
<code>StreamSource</code>	<code>java.io.File</code> ， <code>java.io.InputStream</code> 或 <code>java.io.Reader</code>

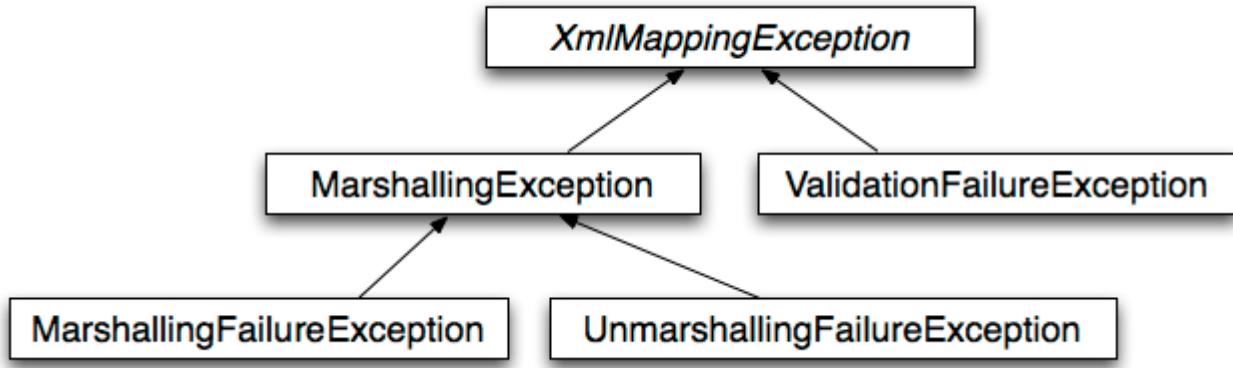
即使有两个单独的编组接口 (`Marshaller` 和 `Unmarshaller`)，Spring-WS 中的所有实现都在一个类中实现。这意味着您可以连接一个编组类，并在 `applicationContext.xml` 中将其称为编组类和解编组。

### 5.2.3. 了解 `XmlMappingException`

Spring 使用 `XmlMappingException` 作为根异常将底层 O-XMap 工具中的异常转换为它自己的异常层次。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，`MarshallingFailureException` 和 `UnmarshallingFailureException` 提供了编组和解组操作之间的区别，即使底层的 O-XMap 工具没有这样做。

O-XMap 异常层次结构如下图所示：



### 5.3. 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。除了主要方法外，该类还有两个方法：`saveSettings()` 将设置 bean 保存到名为 `settings.xml` 的文件中，并且 `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：

```
import java.io.FileInputStream;  
import java.io.FileOutputStream;  
import java.io.IOException;  
import javax.xml.transform.stream.StreamResult;  
import javax.xml.transform.stream.StreamSource;  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
import org.springframework.oxm.Marshaller;  
import org.springframework.oxm.Unmarshaller;  
  
public class Application {
```

```

private static final String FILE_NAME = "settings.xml";
private Settings settings = new Settings();
private Marshaller marshaller;
private Unmarshaller unmarshaller;

public void setMarshaller(Marshaller marshaller) {
    this.marshaller = marshaller;
}

public void setUnmarshaller(Unmarshaller unmarshaller) {
    this.unmarshaller = unmarshaller;
}

public void saveSettings() throws IOException {
    FileOutputStream os = null;
    try {
        os = new FileOutputStream(FILE_NAME);
        this.marshaller.marshal(settings, new StreamResult(os));
    } finally {
        if (os != null) {
            os.close();
        }
    }
}

public void loadSettings() throws IOException {
    FileInputStream is = null;
    try {
        is = new FileInputStream(FILE_NAME);
        this.settings = (Settings) this.unmarshaller.unmarshal(new StreamSource(is));
    } finally {
        if (is != null) {
            is.close();
        }
    }
}

public static void main(String[] args) throws IOException {
    ApplicationContext appContext =
        new ClassPathXmlApplicationContext("applicationContext.xml");
    Application application = (Application) appContext.getBean("application");
    application.saveSettings();
    application.loadSettings();
}
}

```

`Application` 要求同时设置 `marshaller` 和 `unmarshaller` 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 `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"/>
```

## 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" (1)
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/oxm http://www.springframework.org/schema/oxm/s
```

- (1) 引用 `oxm` 模式。
- (2) 指定 `oxm` 模式位置。

当前，该模式使以下元素可用：

- [jaxb2-marshaller](#)
- [jibx-marshaller](#)
- [castor-marshaller](#)

每个标签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.airli
```

## 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 Comments 的 Java 类生成模式的方法。

Spring 遵循[马歇尔与非马歇尔](#)中描述的 `Marshaller` 和 `Unmarshaller` 接口，将 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` 类同时实现 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

`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">
```

或者，您可以使用 `class-to-be-bound` 子元素提供要绑定到编组的类的列表：

```
<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
<code>id</code>	编组的 ID	No
<code>contextPath</code>	JAXB 上下文路径	No

## 5.6. Castor

Castor XMLMap 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 Map 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` 实现 `Marshaller` 和 `Unmarshaller` 接口。可以如下进行连接：

```
<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，但可能有必要对其进行更多控制。您可以通过使用 CastorMap 文件来获得更多控制。有关更多信息，请参见[Castor XMLMap](#)。

您可以使用 `mappingLocation` 资源属性来设置 Map，在以下示例中使用 Classpath 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` 标签配置 `org.springframework.oxm.castor.CastorMarshaller`，如以下示例所示：

```
<oxm:castor-marshaller id="marshaller" mapping-location="classpath:org/springframework/
```

您可以通过两种方式配置 marshaller 实例：通过指定 Map 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)来确定存在其相应 XMLDescriptors 类的 Java POJO。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
<code>id</code>	编组的 ID	No

Attribute	Description	Required
<code>encoding</code>	用于从 XML 解组的编码	No
<code>target-class</code>	POJO 的 Java 类名，可为其使用 XML 类 Descriptors(通过代码生成生成)	No
<code>target-package</code>	Java 包名称，用于标识包含 POJO 及其对应的 Castor XMLDescriptors 类的包(由 XML 模式生成的 代码生成)	No
<code>mapping-location</code>	Castor XMLMap 文件的位置	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从 XML 到 XML 的类实例的转换。

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 Importing 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 `Flights` 类：

```
<beans>
```

```
<bean id="jibxFlightsMarshaller" class="org.springframework.oxm.jibx.JibxMarshaller">
    <property name="targetClass">org.springframework.oxm.jibx.Flights</property>
</bean>
...
</beans>
```

为单个类配置了 `JibxMarshaller`。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` 标签配置 `org.springframework.oxm.jibx.JibxMarshaller`，如以下示例所示：

```
<oxm:jibx-marshaller id="marshaller" target-class="org.springframework.ws.samples.airli...
```

下表描述了可用的属性：

Attribute	Description	Required
<code>id</code>	编组的 ID	No
<code>target-class</code>	该编组的目标类	Yes
<code>bindingName</code>	该编组器使用的绑定名称	No

## 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 Map 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。

## 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。为了进一步自定义 XML，可以设置一个别名 Map，该 Map 由 Map 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>
```

### ⚠ Warning

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)中解组 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>
```

这样做可确保只有注册的类才有资格进行编组。

此外，您可以注册 [custom converters](#) 以确保只能解组受支持的类。除了显式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。

### 💡 Note

请注意，XStream 是 XML 序列化库，而不是数据绑定库。因此，它具有有限的名称空间支持。结果，它非常不适合在 Web 服务中使用。

## 6. Appendix

### 6.1. XML 模式

附录的此部分列出了用于数据访问的 XML 模式，包括以下内容：

- [tx 模式](#)
- [jdbc 模式](#)

#### 6.1.1. tx 模式

**tx** 标签用于在 Spring 的全面事务支持中配置所有这些 bean。这些标签在标题为 [Transaction Management](#) 的章节中介绍。

#### Tip

我们强烈建议您查看 Spring 发行版随附的 '[spring-tx.xsd](#)' 文件。该文件包含用于 Spring 事务配置的 XML 模式，并涵盖 **tx** 名称空间中的所有各个元素，包括属性默认值和类似信息。该文件已内联记录，因此，出于遵守 DRY(请勿重复自己)原则的考虑，此处不再重复信息。

为了完整起见，要使用 **tx** 模式中的元素，您需要在 Spring XML 配置文件的顶部具有以下序言。

以下代码段中的文本引用了正确的架构，以便您可以使用 **tx** 名称空间中的标记：

```
<?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" (1)
```

```
xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans
http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop

<!-- bean definitions here -->

</beans>
```

- (1) 声明 `tx` 名称空间的用法。
- (2) 指定位置(以及其他架构位置)。

#### iNote

通常，当您使用 `tx` 名称空间中的元素时，也会同时使用 `aop` 名称空间中的元素(因为 Spring 中的声明式事务支持是使用 AOP 实现的)。前面的 XML 代码段包含引用 `aop` 模式所需的相关行，以便 `aop` 命名空间中的元素可供您使用。

### 6.1.2. jdbc 模式

`jdbc` 元素使您可以快速配置嵌入式数据库或初始化现有数据源。这些元素分别记录在[嵌入式数据库支持](#)和[初始化数据源](#)中。

要使用 `jdbc` 模式中的元素，您需要在 Spring XML 配置文件的顶部具有以下序言。以下代码段中的文本引用了正确的架构，以便您可以使用 `jdbc` 名称空间中的元素：

```
<?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:jdbc="http://www.springframework.org/schema/jdbc" (1)
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/jdbc http://www.springframework.org/schema/jdbc

<!-- bean definitions here -->

</beans>
```

- (1) 声明 `jdbc` 名称空间的用法。

- (2) 指定位置(以及其他架构位置)。

# Web on Servlet 堆栈

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 Support](#)和[WebSocket Support](#)。有关反应式堆栈 Web 应用程序, 请参见[网上反应堆](#)。

## 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 Framework 中。正式名称 “Spring Web MVC”来自其源模块的名称([spring-webmvc](#)), 但更通常称为 “Spring MVC”。

与 Spring Web MVC Parallel, 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。[next section](#) 涵盖了 Spring WebFlux。

有关基线信息以及与 Servlet 容器和 Java EE 版本范围的兼容性, 请参见 Spring Framework [Wiki](#)。

### 1.1. DispatcherServlet

[与 Spring WebFlux 中的相同](#)

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, 其中中央 [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。

和其他 [Servlet](#) 一样, 需要根据 [Servlet](#) 规范通过使用 Java 配置或在 [web.xml](#) 中进行声明和 Map。反过来, [DispatcherServlet](#) 使用 Spring 配置发现请求 Map, 视图解析, 异常处理[and more](#) 所需的委托组件。

以下 Java 配置示例注册并初始化 `DispatcherServlet`，它由 Servlet 容器自动检测(请参见 [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/*");  
    }  
}
```

### iNote

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>

<servlet-mapping>
    <servlet-name>app</servlet-name>
    <url-pattern>/app/*</url-pattern>
</servlet-mapping>

</web-app>
```

### iNote

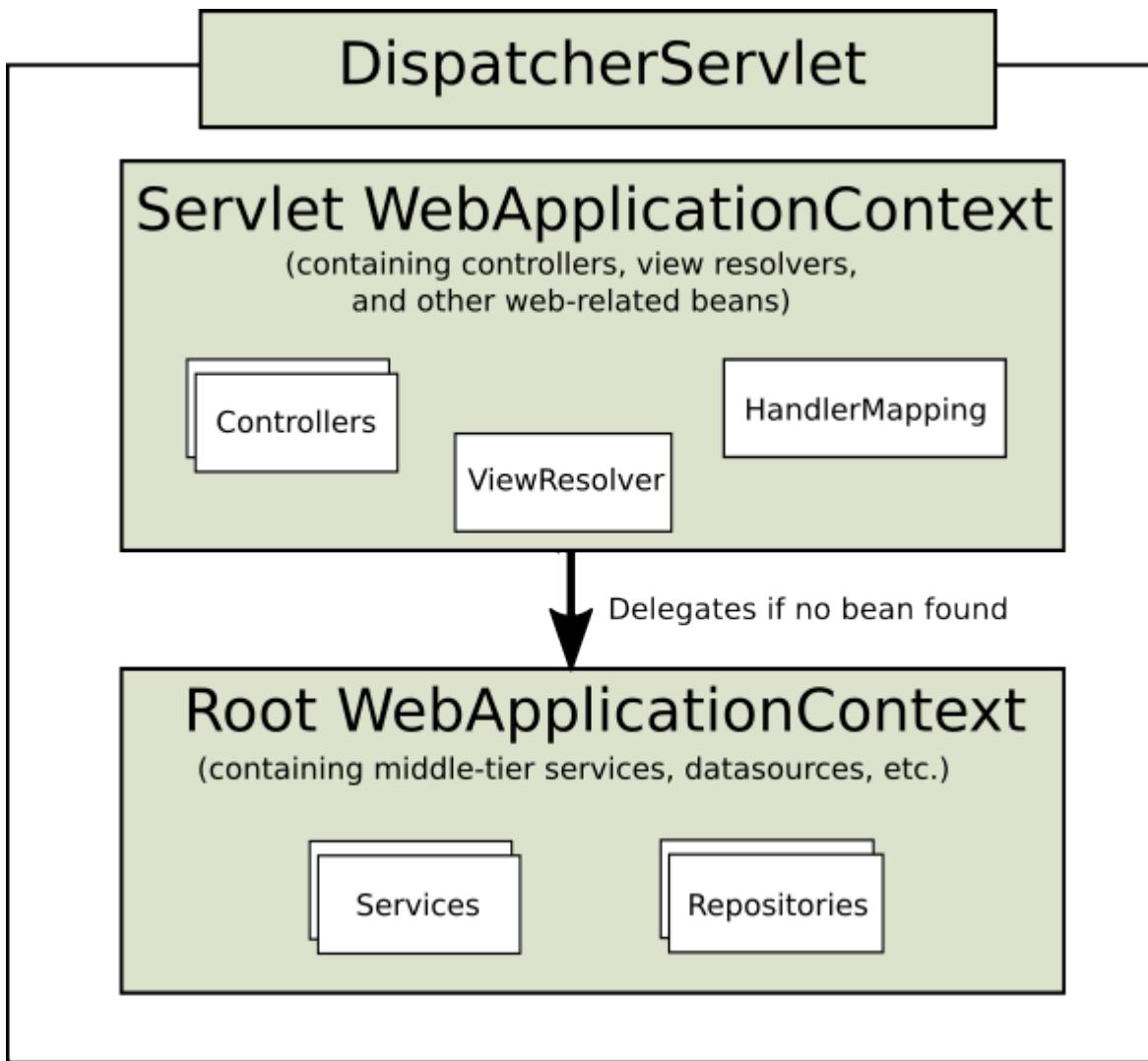
Spring Boot 遵循不同的初始化 Sequences。Spring Boot 并没有陷入 Servlet 容器的生命周期，而是使用 Spring 配置来引导自身和嵌入式 Servlet 容器。在 Spring 配置中检测到 `Filter` 和 `Servlet` 声明，并在 Servlet 容器中注册。有关更多详细信息，请参见[Spring Boot 文档](#)。

## 1.1.1. 上下文层次

`DispatcherServlet` 期望其自己的配置为 `WebApplicationContext` (纯 `ApplicationContext` 的 `extensions`)。`WebApplicationContext` 具有到 `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)。

对于许多应用程序来说，只有一个 `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` 跨多个 `DispatcherServlet` (或其他 `Servlet`) 实例共享，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，可以在 `Servlet` 特定子 `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：



以下示例配置 `WebApplicationContext` 层次结构：

```

public class MyWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class<?>[] { RootConfig.class };
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class<?>[] { App1Config.class };
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/app1/*" };
    }
}
    
```

Tip

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 `getServletConfigClasses()` 的 `getRootConfigClasses()` 和 `null` 返回所有配置。

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>
```

### Tip

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

与 Spring WebFlux 中的相同

`DispatcherServlet` 委托特殊 bean 处理请求并呈现适当的响应。所谓“特殊 bean”，是指实现

WebFlux 框架 Contract 的 SpringManagement 的 `Object` 实例。这些通常带有内置 Contract，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 `DispatcherHandler` 检测到的特殊 bean:

Bean type	Explanation
<code>HandlerMapping</code>	将请求与 <a href="#">interceptors</a> 列表一起 Map 到处理程序，以进行预处理和后期处理。Map 基于某些条件，具体取决于 <code>HandlerMapping</code> 实现。

`HandlerMapping` 的两个主要实现是 `RequestMappingHandlerMapping` (支持 `@RequestMapping` 带 Comments 的方法) 和 `SimpleUrlHandlerMapping` (将 URI 路径模式的显式注册维护到处理程序)。

| `HandlerAdapter` | 帮助 `DispatcherServlet` 调用 Map 到请求的处理程序，无论实际如何调用该处理程序。例如，调用带 Comments 的控制器需要解析 Comments。`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 Map 到处理程序，HTML 错误视图或其他目标。参见[Exceptions](#)。

| [ViewResolver](#) | 解析从处理程序返回到实际 `View` 的基于逻辑 `String` 的视图名称，使用该视图名称呈现给响应。参见[View Resolution](#) 和 [View Technologies](#)。

| [LocaleResolver](#), [LocaleContextResolver](#) | 解析 Client 端正在使用的 `Locale` 以及可能的时区，以便能够提供国际化的视图。参见[Locale](#)。

| [ThemeResolver](#) | 解决 Web 应用程序可以使用的主题，例如提供个性化的布局。参见[Themes](#)。

| [MultipartResolver](#) | 用于借助一些 Multipart 解析库来分析 Multipart 请求(例如，浏览器表单文件上载)的抽象。参见[Multipart Resolver](#)。

| [FlashMapManager](#) | 存储和检索“Importing”和“输出”[FlashMap](#)，它们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。参见[Flash Attributes](#)。

### 1.1.3. Web MVC 配置

[与 Spring WebFlux 中的相同](#)

应用程序可以声明处理请求所必需的[特殊 bean](#)类中列出的基础结构 bean。

[DispatcherServlet](#) 为每个特殊 bean 检查 [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，并提供更高级别的配置回调 API 对其进行自定义。

#### Note

Spring Boot 依靠 MVC Java 配置来配置 Spring MVC，并提供许多额外的方便选项。

### 1.1.4. Servlet 配置

在 Servlet 3.0 环境中，您可以选择以编程方式配置 Servlet 容器，以替代方式或与 [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",
            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。名为 `AbstractDispatcherServletInitializer` 的

`WebApplicationInitializer` 的抽象 Base Class 实现通过覆盖方法来指定 `servletMap` 和

`DispatcherServlet` 配置的位置，从而使 `DispatcherServlet` 的注册更加容易。

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
    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` 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 ctxt = new XmlWebApplicationContext();
        ctxt.setConfigLocation("/WEB-INF/spring/dispatcher-config.xml");
        return ctxt;
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
}
```

`AbstractDispatcherServletInitializer` 还提供了一种方便的方法来添加 `Filter` 实例，并

将它们自动 Map 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 Map 到 `DispatcherServlet`。

`AbstractDispatcherServletInitializer` 的受 `isAsyncSupported` 保护的方法提供了一个位置，以对 `DispatcherServlet` 及其 Map 的所有过滤器启用异步支持。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

与 [Spring WebFlux 中的相同](#)

`DispatcherServlet` 处理请求的方式如下：

- 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，它是在 `DispatcherServlet.WEB_APPLICATION_CONTEXT_ATTRIBUTE` 键下绑定的。
- 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，则可以忽略它。

- 如果指定 Multipart 文件解析器，则将检查请求中是否有 Multipart。如果找到 Multipart，则将请求包装在 `MultipartHttpServletRequest` 中，以供流程中的其他元素进一步处理。有关 Multipart 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 Comments 的控制器，可以呈现响应(在 `HandlerAdapter` 内)，而不返回视图。
- 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[Exceptions](#)。

Spring `DispatcherServlet` 还支持 Servlet API 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 Map 并测试找到的处理程序是否实现 `LastModified` 接口。如果是这样，则 `LastModified` 接口的 `long getLastModified(request)` 方法的值返回给 Client 端。

您可以通过向 `web.xml` 文件中的 `Servlet` 声明中添加 `Servlet` 初始化参数(`init-param` 元素)来自定义 `DispatcherServlet` 实例。下表列出了受支持的参数：

表 1. `DispatcherServlet` 初始化参数

Parameter	Explanation
<code>contextClass</code>	实现 <code>ConfigurableWebApplicationContext</code> 的类，将由此 <code>Servlet</code> 实例化并在本地配置。默

Parameter	Explanation
	<p>认情况下，使用 <code>XmlWebApplicationContext</code>。</p>
<code>contextConfigLocation</code>	<p>传递给上下文实例的字符串(由 <code>contextClass</code> 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。</p>
<code>namespace</code>  <code>throwExceptionIfNoHandlerFound</code>	<p><code>WebApplicationContext</code> 的命名空间。默认认为 <code>[servlet-name]-servlet</code>。</p> <p>在找不到请求处理器时是否抛出 <code>NoHandlerFoundException</code>。然后可以使用户 <code>HandlerExceptionResolver</code> 捕获异常(例如，通过使用 <code>@ExceptionHandler</code> 控制器方法)，然后将其作为其他任何异常进行处理。</p>

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，这些拦截器在您要将特定功能应用于某些请求时非常有用-例如检查主体。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 Active 以执行所有类型的预处理和后处理：

- `preHandle(..)`：在执行实际处理程序之前
- `postHandle(..)`：执行处理程序后
- `afterCompletion(..)`：完成完整的请求后

`preHandle(..)` 方法返回布尔值。您可以使用此方法来中断或 `continue` 执行链的处理。当此方法返回 `true` 时，处理程序执行链 `continue`。当它返回 `false` 时，`DispatcherServlet` 假定拦截器本身已经处理了请求(例如，提供了适当的视图)，并且不会 `continue` 执行执行链中的其他拦截器和实际处理程序。

有关如何配置拦截器的示例，请参见 MVC 配置部分中的[Interceptors](#)。您还可以通过使用各个 `HandlerMapping` 实现上的设置器直接注册它们。

请注意，`postHandle` 在 `@ResponseBody` 和 `ResponseEntity` 方法中用处不大，因为在 `HandlerAdapter` 内和 `postHandle` 之前将响应写入和提交。这意味着对响应进行任何更改为时已晚，例如添加额外的 Headers。对于此类情况，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

与 Spring WebFlux 中的相同

如果在请求 Map 期间发生异常或从请求处理程序(例如 `@Controller`)引发异常，则

`DispatcherServlet` 委托 `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：

表 2. `HandlerExceptionResolver` 实现

<code>HandlerExceptionResolver</code>	Description
<code>SimpleMappingExceptionResolver</code>	异常类名称和错误视图名称之间的 Map。对于在浏览器应用程序中呈现错误页面很有用。
<a href="#">DefaultHandlerExceptionResolver</a>	解决了 Spring MVC 引发的异常，并将它们 Map 到 HTTP 状态代码。另请参见 <code>ResponseEntityExceptionHandler</code> 和 <a href="#">REST API exception</a> 。
<code>ResponseStatusExceptionResolver</code>	使用 <code>@ResponseStatus</code> Comments 解决异常，并根据 Comments 中的值将其 Map 到 HTTP 状态代码。
<code>ExceptionHandlerExceptionResolver</code>	通过调用 <code>@Controller</code> 或 <code>@ControllerAdvice</code> 类中的 <code>@ExceptionHandler</code> 方法来解决异常。参见 <a href="#">@ExceptionHandler methods</a> 。

## 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 Contract 规定它可以返回：

- `ModelAndView` 指向错误视图。
- 如果在解析程序中处理了异常，则为空  `ModelAndView`。
- `null` 如果仍未解决异常，则供以后的解析器尝试，并且，如果异常仍在末尾，则允许其冒泡到 Servlet 容器。

[MVC Config](#) 自动为默认的 Spring MVC 异常，`@ResponseStatus` 带 Comments 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 `continue` 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` 中声明一个错误页面 Map。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，当异常冒出气泡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，可能将其 Map 到 `@Controller`，可以实现该错误以返回带有模型的错误视图名称或呈现 JSON 响应，如以下示例所示：

```
@RestController
public class ErrorController {
```

```

    @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;
    }
}

```

### Tip

Servlet API 没有提供在 Java 中创建错误页面 Map 的方法。但是，您可以同时使用 `WebApplicationInitializer` 和最小的 `web.xml`。

## 1.1.8. 查看分辨率

[与 Spring WebFlux 中的相同](#)

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，而无需将您绑定到特定的视图技术。`ViewResolver` 提供了视图名称和实际视图之间的 Map。`View` 解决了在移交给特定视图技术之前的数据准备问题。

下表提供了有关 `ViewResolver` 层次结构的更多详细信息：

表 3. `ViewResolver` 实现

ViewResolver	Description
<code>AbstractCachingViewResolver</code>	<p><code>AbstractCachingViewResolver</code> 缓存视图实例所解析的子类。缓存可以提高某些视图技术的性能。您可以通过将 <code>cache</code> 属性设置为 <code>false</code> 来关闭缓存。此外，如果必须在运行时刷新某个视图(例如，修改 FreeMarker 模</p>

ViewResolver	Description
	<p>板时), 则可以使用 <code>removeFromCache(String viewName, Locale loc)</code> 方法。</p>
<code>XmlViewResolver</code>	<p><code>ViewResolver</code> 的实现, 该实现接受用 XML 编写的配置文件, 该配置文件的 DTD 与 Spring 的 XML bean 工厂相同。默认配置文件为 <code>/WEB-INF/views.xml</code>。</p>
<code> ResourceBundleViewResolver</code>	<p><code>ViewResolver</code> 的实现使用 <code>ResourceBundle</code> 中的 bean 定义(由包基本名称指定)。对于应该解析的每个视图, 它将属性 <code>[viewname].(class)</code> 的值用作视图类, 并将属性 <code>[viewname].url</code> 的值用作视图 URL。您可以在<a href="#">View Technologies</a>的章节中找到示例。</p>
<code>UrlBasedViewResolver</code>	<p><code>ViewResolver</code> 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 Map 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, 而不需要任意 Map, 则这是适当的。</p>
<code>InternalResourceViewResolver</code>	<p>方便的 <code>UrlBasedViewResolver</code> 子类, 支持</p>

ViewResolver	Description
	<p><code>InternalResourceView</code> (实际上是 Servlet 和 JSP) 以及 <code>JstlView</code> 和 <code>TilesView</code> 之类的子类。您可以使用 <code>setViewClass(...)</code> 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<a href="#">UrlBasedViewResolver javadoc</a>。</p>
<code>FreeMarkerViewResolver</code>	<p><code>UrlBasedViewResolver</code> 的便捷子类, 支持 <code>FreeMarkerView</code> 及其自定义子类。</p>
<code>ContentNegotiatingViewResolver</code>	<p><code>ViewResolver</code> 接口的实现, 该接口根据请求文件名或 <code>Accept</code> Headers 解析视图。参见<a href="#">Content Negotiation</a>。</p>

## Handling

[与 Spring WebFlux 中的相同](#)

您可以 `pass` 语句多个解析器 bean 以及必要时通过设置 `order` 属性以指定 Sequences 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` 来指示找不到该视图。但是, 对于 JSP 和 `InternalResourceViewResolver` 而言, 弄清 JSP 是否存在的唯一方法是通过 `RequestDispatcher` 进行调度。因此, 必须始终将 `InternalResourceViewResolver` 配置为在视图解析器的总体 Sequences 中排在最后。

配置视图分辨率就像在 Spring 配置中添加 `ViewResolver` bean 一样简单。[MVC Config](#) 为 [View Resolvers](#) 和添加无逻辑的 [View Controllers](#) 提供了专用的配置 API，这对于不带控制器逻辑的 HTML 模板呈现非常有用。

## Redirecting

### [与 Spring WebFlux 中的相同](#)

视图名称中特殊的 `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 上下文重定向，而名称(例如 `redirect:http://myhost.com/some/arbitrary/path`)重定向到绝对 URL。

请注意，如果控制器方法用 `@ResponseStatus` `Comments`，则 `Comments` 值优先于 `RedirectView` 设置的响应状态。

## Forwarding

您还可以为视图名称使用特殊的 `forward:` 前缀，这些名称最终由 `UrlBasedViewResolver` 和子类解析。这将创建一个 `InternalResourceView`，它执行 `RequestDispatcher.forward()`。因此，此前缀对于 `InternalResourceViewResolver` 和 `InternalResourceView` (对于 JSP)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，您也可以改为链接多个视图解析器。

## Content Negotiation

### [与 Spring WebFlux 中的相同](#)

`ContentNegotiatingViewResolver` 本身不会解析视图，而是委派给其他视图解析器，并选择类似于 Client 端请求的表示形式的视图。可以从 `Accept` Headers 或查询参数(例如

`" /path?format=pdf "` )确定表示形式。

`ContentNegotiatingViewResolver` 通过将请求媒体类型与与每个 `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` 来处理请求。列表中具有兼容 `Content-Type` 的第一个 `View` 将表示形式返回给 Client 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例 `Views`, 该单例 `Views` 可以呈现当前资源的适当表示形式, 而与逻辑视图名称无关。 `Accept` Headers 可以包含通配符(例如 `text/*`), 在这种情况下, `Content-Type` 为 `text/xml` 的 `View` 是兼容的匹配。

有关配置详细信息, 请参见[MVC Config](#)下的[View Resolvers](#)。

### 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。

`DispatcherServlet` 使您可以使用 Client 端的语言环境自动解析邮件。这是通过 `LocaleResolver` 个对象完成的。

收到请求时, `DispatcherServlet` 查找语言环境解析器, 如果找到一个, 它将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 Map 上附加一个拦截器(有关处理程序 Map 拦截器的更多信息, 请参见[Interception](#))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, 并以常规方式在您的应用程序上下文中配置。 Spring 包含以下选择的语言环境解析器。

- [Time Zone](#)
- [Header Resolver](#)

- [Cookie Resolver](#)
- [Session Resolver](#)
- [Locale Interceptor](#)

## Time Zone

除了获取 Client 的语言环境外，了解其时区通常也很有用。 `LocaleContextResolver` 界面提供了对 `LocaleResolver` 的扩展，该扩展使解析器可以提供更丰富的 `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`。Spring 的 `ConversionService` 注册的任何 Date/Time `Converter` 和 `Formatter` 对象都会自动使用时区信息。

## Header Resolver

此语言环境解析器检查 Client 端(例如，Web 浏览器)发送的请求中的 `accept-language` Headers。通常，此头字段包含 Client 端 os 的语言环境。请注意，此解析器不支持时区信息。

## Cookie Resolver

此语言环境解析器检查 Client 端上可能存在的 `Cookie`，以查看是否指定了 `Locale` 或 `TimeZone`。如果是这样，它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 shutdown)-->
    <property name="cookieMaxAge" value="100000"/>
</bean>
```

下表描述了属性 `CookieLocaleResolver`：

表 4. `CookieLocaleResolver` 属性

Property	Default	Description
<code>cookieName</code>	类别名称 LOCALE	Cookie 的名称
<code>cookieMaxAge</code>	Servlet 容器默认	Cookie 在 Client 端上保留的最长时间。如果指定了 <code>-1</code> ，则 cookie 将不会保留。它仅在 Client 端关闭浏览器之前可用。
<code>cookiePath</code>	/	将 Cookie 的可见性限制在您网站的特定部分。指定 <code>cookiePath</code> 时，该 cookie 仅对该路径及其下方的路径可见。

## Session Resolver

`SessionLocaleResolver` 可让您从可能与用户请求关联的会话中检索 `Locale` 和 `TimeZone`。

与 `CookieLocaleResolver` 相反，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 Management 机制(例如 Spring Session 项目)没有直接关系。该

`SessionLocaleResolver` 对当前 `HttpServletRequest` 评估并修改了相应的 `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。它在请求中检测到一个参数，并相应地更改语言环境，从而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参数 `siteLanguage` 的所有

`*.view` 资源的调用现在会更改语言环境。因此，例如，对 URL

`http://www.sf.net/home.view?siteLanguage=nl` 的请求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>
</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(通常是样式表和图像)的集合，这些资源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，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 Classpath 的根目录加载属性文件。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
```

属性的键是引用视图代码中主题元素的名称。对于 JSP，通常使用 `spring:theme` 自定义标签来完成此操作，该标签与 `spring:message` 标签非常相似。以下 JSP 片段使用上一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="" type="text/css"/>  
    </head>  
    <body style="background=<spring:theme code='background' />">  
        ...  
    </body>  
</html>
```

默认情况下，`ResourceBundleThemeSource` 使用空的基本名称前缀。结果，从 Classpath 的根加载属性文件。因此，您可以将 `cool.properties` 主题定义放在 Classpath 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，我们可能有一个 `/WEB-INF/classes/cool_nl.properties`，它引用了带有荷兰 Literals 的特殊背景图像。

## Resolving Themes

定义主题后，如 [preceding section](#) 中所述，您可以决定使用哪个主题。`DispatcherServlet` 查找名为 `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` 大致相同。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：

表 5. `ThemeResolver` 实现

Class	Description
FixedThemeResolver	选择通过使用 <code>defaultThemeName</code> 属性设置的固定主题。
SessionThemeResolver	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 Client 端的 cookie 中。

Spring 还提供了一个 `ThemeChangeInterceptor`，该主题可以使用简单的 `request` 参数在每个请求上更改主题。

### 1.1.11. Multipart 解析器

[与 Spring WebFlux 中的相同](#)

`org.springframework.web.multipart` 包中的 `MultipartResolver` 是一种用于解析包括文件上传在内的 Multipart 请求的策略。有一种基于[Commons FileUpload](#)的实现，另一种基于 Servlet 3.0 Multipart 请求解析。

要启用 Multipart 处理，您需要在 `DispatcherServlet` Spring 配置中声明名称为 `multipartResolver` 的 `MultipartResolver` bean。`DispatcherServlet` 检测到它并将其应用于传入的请求。收到 Content Type 为 `multipart/form-data` 的 POST 时，解析程序将解析内容并将当前 `HttpServletRequest` 包装为 `MultipartHttpServletRequest` 以提供对已解析部分的访问权限，除了将其公开为请求参数之外。

#### Apache Commons FileUpload

要使用 Apache Commons `FileUpload`，可以配置名称为 `multipartResolver` 的类型 `CommonsMultipartResolver` 的 bean。您还需要 `commons-fileupload` 作为对 Classpath 的依

赖。

## Servlet 3.0

需要通过 Servlet 容器配置启用 Servlet 3.0 Multipart 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` 的类型为

`StandardServletMultipartResolver` 的 bean。

### 1.1.12. Logging

[与 Spring WebFlux 中的相同](#)

Spring MVC 中的 DEBUG 级别的日志被设计为紧凑，最少且人性化的。它侧重于高价值的信息，这些信息一遍又一遍地有用，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。如果发现任何不符合既定目标的东西，请告诉我们。

## Sensitive Data

## [与 Spring WebFlux 中的相同](#)

调试和跟踪日志记录可能会记录敏感信息。这就是为什么默认情况下屏蔽请求参数和 Headers，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使用 Java 配置来执行此操作：

```
public class MyInitializer
    extends AbstractAnnotationConfigDispatcherServletInitializer {

    @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");
    }
}
```

## 1.2. Filters

### [与 Spring WebFlux 中的相同](#)

`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. 表格数据

浏览器只能通过 HTTP GET 或 HTTP POST 提交表单数据，但非浏览器 Client 端也可以使用 HTTP PUT, PATCH 和 DELETE。Servlet API 需要 `ServletRequest.getParameter*()` 个方法才能仅对 HTTP POST 支持表单字段访问。

`spring-web` 模块提供 `FormContentFilter` 来拦截 Content Type 为 `application/x-www-form-urlencoded` 的 HTTP PUT, PATCH 和 DELETE 请求，从请求的正文中读取表单数据，并包装 `ServletRequest` 以通过 `ServletRequest.getParameter*()` 系列方法使表单数据可用。

### 1.2.2. 转发的标题

[与 Spring WebFlux 中的相同](#)

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 Client 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 `Forwarded` HTTPHeaders，代理可用来提供有关原始请求的信息。还有其他非标准 Headers，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 `Forwarded` Headers 修改请求的主机，端口和方案，然后删除这些 Headers。

对于转发的 Headers，存在安全方面的考虑，因为应用程序无法知道 Headers 是由代理添加的，还是由恶意 Client 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 `Forwarded` Headers 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 Headers。

### 1.2.3. 浅 ETag

`ShallowEtagHeaderFilter` 过滤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。Client 端下一次发送时，将执行相同的操作，但还会将计算值与 `If-None-Match` 请求 Headers 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，但不能节省 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。参见[HTTP Caching](#)。

该过滤器具有一个 `writeWeakETag` 参数，该参数将过滤器配置为写入弱 ETag，类似于以下代码：

`w/ "02a2d595e6ed9a0b24f027f2b63b134d6"` (在[RFC 7232 第 2.3 节](#)中定义)。

## 1.2.4. CORS

[与 Spring WebFlux 中的相同](#)

Spring MVC 通过控制器上的 Comments 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 `CorsFilter` 必须在 Spring Security 的过滤器链之前 Order。

有关更多详细信息，请参见[CORS](#)和[CORS Filter](#)部分。

## 1.3. 带 Comments 的控制器

[与 Spring WebFlux 中的相同](#)

Spring MVC 提供了基于 Comments 的编程模型，其中 `@Controller` 和 `@RestController` 组件使用 Comments 来表达请求 Map，请求 Importing，异常处理等。带 Comments 的控制器具有灵活的方法签名，无需扩展 Base Class 或实现特定的接口。以下示例显示了由 Comments 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。

### Tip

[spring.io](#) 上的指南和教程使用本节中介绍的基于 `Comments` 的编程模型。

## 1.3.1. Declaration

与 Spring WebFlux 中的相同

您可以使用 Servlet 的 `WebApplicationContext` 中的标准 Spring bean 定义来定义控制器 bean

- `@Controller` 原型允许自动检测，与 Spring 对在 Classpath 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 `Comments` 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 {

    // ...
}
```

下面的示例显示与前面的示例等效的 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
```

```
http://www.springframework.org/schema/beans/spring-beans.xsd  
http://www.springframework.org/schema/context  
http://www.springframework.org/schema/context/spring-context.xsd">  
  
<context:component-scan base-package="org.example.web"/>  
  
<!-- ... -->  
  
</beans>
```

`@RestController` 是一个 [composed annotation](#)，它本身会用 `@Controller` 和 `@ResponseBody` 进行元 Comments，以表示其每个方法都继承了类型级别 `@ResponseBody` Comments 的控制器，因此直接将其写入响应主体(与视图分辨率和 HTML 模板渲染相比)。

## AOP Proxies

在某些情况下，许多人需要在运行时用 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` 注解。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. 请求 Map

[与 Spring WebFlux 中的相同](#)

您可以使用 `@RequestMapping` 注解将请求 Map 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，Headers 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 Map，也可以在方法级别使用它来缩小到特定的端点 Map。

也有 `@RequestMapping` 的 HTTP 方法特定的快捷方式：

- `@GetMapping`
- `@PostMapping`

- `@PutMapping`
- `@DeleteMapping`
- `@PatchMapping`

之所以提供快捷方式是[Custom Annotations](#)，是因为可以说，大多数控制器方法应该 Map 到特定的 HTTP 方法，而不是使用 `@RequestMapping`（默认情况下，该方法与所有 HTTP 方法匹配）。同时，在类级别仍需要 `@RequestMapping` 来表示共享 Map。

以下示例具有类型和方法级别的 Map：

```
@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

[与 Spring WebFlux 中的相同](#)

您可以使用以下 glob 模式和通配符来 Map 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 变量将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 support。参见[Type Conversion](#) 和 [DataBinder](#)。

您可以显式命名 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"`，以下方法将提取名称，版本和文件 extensions：

```
@GetMapping("/{name:[a-z-]+}-{version:\\d\\.\\d\\.\\d}{ext:\\.[a-z]+}")
public void handle(@PathVariable String version, @PathVariable String ext) {
    // ...
}
```

URI 路径模式还可以嵌入  `${...}` 占位符，这些占位符在启动时可以通过针对本地，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。

### iNote

Spring MVC 使用 `PathMatcher` 协定和 `spring-core` 的 `AntPathMatcher` 实现实现 URI 路径匹配。

## Pattern Comparison

[与 Spring WebFlux 中的相同](#)

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。这是通过使用

`AntPathMatcher.getPatternComparator(String path)` 完成的，该

`AntPathMatcher.getPatternComparator(String path)` 查找更具体的模式。

如果模式的 URI 变量数较少，单个通配符计为 1，双通配符计为 2，则模式的特异性较低。给定分数，则选择较长的模式。给定相同的分数和长度，将选择 URI 变量比通配符更多的模式。

默认 Map 模式(`/**`)被排除在评分之外，并且始终排在最后。另外，前缀模式(例如 `/public/**`)被认为比其他没有双通配符的模式更具体。

有关完整的详细信息，请参见[AntPathMatcher](#)中的[AntPatternComparator](#)，并请记住您可以自定义[PathMatcher](#)实现。请参阅配置部分中的[Path Matching](#)。

## Suffix Match

默认情况下，Spring MVC 执行 `.*` 后缀模式匹配，以便 Map 到 `/person` 的控制器也隐式 Map 到 `/person.*`。然后，文件 extensions 用于解释请求的 Content Type 以用于响应(即，代替 `Accept Headers`)—例如 `/person.pdf`，`/person.xml` 等。

当浏览器过去曾经发送难以理解的 `Accept Headers` 时，以这种方式使用文件 extensions 是必要的。目前，这已不再是必须的，使用 `Accept Headers` 应该是首选。

随着时间的流逝，文件 extensions 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。

要完全禁用文件 extensions，必须设置以下两项：

- `useSuffixPatternMatching(false)`，请参阅[PathMatchConfigurer](#)
- `favorPathExtension(false)`，请参阅[ContentNegotiationConfigurer](#)

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 extensions 附带的大多数问题。另外，如果您必须使用文件 extensions，请考虑通过[ContentNegotiationConfigurer](#)的 `mediaTypes` 属性将它们限制为显式注册的 extensions 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 Importing(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 Content Type，Client 端可以通过 URL 路径扩展请求这些 Content Type。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` Headers，以建议一个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 extensions 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 extensions 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 extensions 以进行内容协商，以避免为这些 extensions 添加 `Content-Disposition` Headers。参见[Content Types](#)。

有关 RFD 的其他建议，请参见[CVE-2015-5211](#)。

## 消耗媒体类型

## 与 Spring WebFlux 中的相同

您可以根据请求的 `Content-Type` 缩小请求 Map，如下例所示：

```
@PostMapping(path = "/pets", consumes = "application/json") (1)
public void addPet(@RequestBody Pet pet) {
    // ...
}
```

- (1) 使用 `consumes` 属性按 Content Type 缩小 Map 范围。

`consumes` 属性还支持否定表达式-例如，`!text/plain` 表示 `text/plain` 以外的任何 Content Type。

您可以在类级别声明共享的 `consumes` 属性。但是，与大多数其他请求 Map 属性不同，在类级使用时，方法级 `consumes` 属性会覆盖而不是扩展类级声明。

### Tip

`MediaType` 为常用的媒体类型(例如 `APPLICATION_JSON_VALUE` 和 `APPLICATION_XML_VALUE`)提供常数。

## 可生产的媒体类型

### 与 Spring WebFlux 中的相同

您可以根据 `Accept` 请求 Headers 和控制器方法生成的 Content Type 列表来缩小请求 Map，如下例所示：

```
@GetMapping(path = "/pets/{petId}", produces = "application/json; charset=UTF-8") (1)
@ResponseBody
public Pet getPet(@PathVariable String petId) {
    // ...
}
```

- (1) 使用 `produces` 属性按 Content Type 缩小 Map 范围。

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“文本/纯文本”之外的任何 Content Type。

### iNote

对于 JSONContent Type，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 Map 属性不同，在类级使用时，方法级 `produces` 属性会覆盖而不是扩展类级声明。

### Tip

`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

与 [Spring WebFlux 中的相同](#)

您可以根据请求参数条件来缩小请求 Map。您可以测试是否存在请求参数(`myParam`)，是否存在请求参数(`!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") (1)
public void findPet(@PathVariable String petId) {
    // ...
}
```

- (1) 测试 `myParam` 是否等于 `myValue`。

您还可以将其与请求 Headers 条件一起使用，如以下示例所示：

```
@GetMapping(path = "/pets", headers = "myHeader=myValue") (1)
public void findPet(@PathVariable String petId) {
```

```
// ...  
}
```

- (1) 测试 `myHeader` 是否等于 `myValue`。

### Tip

您可以将 `Content-Type` 和 `Accept` 与 Headers 条件进行匹配，但最好使用 [consumes](#) 和 [produces](#)。

## HTTP HEAD，选项

[与 Spring WebFlux 中的相同](#)

`@GetMapping` (和 `@RequestMapping(method=HttpMethod.GET)`) 透明地支持 HTTP HEAD 以进行请求 Map。控制器方法不需要更改。`javax.servlet.http.HttpServlet` 中应用的响应包装器确保将 `Content-Length` Headers 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 Map 到并支持 HTTP HEAD。像处理 HTTP GET 一样处理 HTTP HEAD 请求，不同的是，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 `Allow` 响应 Headers 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`，`Allow` Headers 设置为 `GET, HEAD, POST, PUT, PATCH, DELETE, OPTIONS`。控制器方法应始终声明受支持的 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 Map 到 HTTP HEAD 和 HTTP OPTIONS，但这在通常情况下不是必需的。

## Custom Annotations

[与 Spring WebFlux 中的相同](#)

Spring MVC 支持使用 [composed annotations](#) 进行请求 Map。这些注解本身用

`@RequestMapping` 进行元注解，并组成它们以更狭窄，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 Comments 的示例。之所以提供它们，是因为大多数控制器方法应该 Map 到特定的 HTTP 方法，而不是使用 `@RequestMapping`，默认情况下，`@RequestMapping` 匹配所有 HTTP 方法。如果需要组合 Comments 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 Map 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

[与 Spring WebFlux 中的相同](#)

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用例，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 han
        throws NoSuchMethodException {

        RequestMappingInfo info = RequestMappingInfo
            .paths("/user/{id}").methods(RequestMethod.GET).build(); (2)

        Method method = UserHandler.class.getMethod("getUser", Long.class); (3)

        mapping.registerMapping(info, handler, method); (4)
    }
}
```

- (1) 注入目标处理程序和控制器的处理程序 Map。
- (2) 准备请求 Map 元数据。
- (3) 获取处理程序方法。
- (4) 添加注册。

### 1.3.3. 处理程序方法

[与 Spring WebFlux 中的相同](#)

`@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。

#### Method Arguments

[与 Spring WebFlux 中的相同](#)

下表描述了受支持的控制器方法参数。任何参数均不支持 Reactive 类型。

支持 JDK 8 的 `java.util.Optional` 作为方法参数，并与具有 `required` 属性(例如 `@RequestParam`，`@RequestHeader` 等)的注解结合使用，该注解等效于 `required=false`。

控制器方法参数	Description
<code>WebRequest</code> ， <code>NativeWebRequest</code>	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。
<code>javax.servlet.ServletRequest</code> ， <code>javax.servlet.ServletResponse</code>	选择任何特定的请求或响应类型，例如 <code>ServletRequest</code> ， <code>HttpServletRequest</code> 或 Spring 的 <code>MultipartRequest</code> ， <code>MultipartHttpServletRequest</code> 。

控制器方法参数	Description
<code>javax.servlet.http.HttpSession</code>	<p>强制会话的存在。因此，这样的参数永远不会是 <code>null</code>。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 <code>RequestMappingHandlerAdapter</code> 实例的 <code>synchronizeOnSession</code> 标志设置为 <code>true</code>。</p>
<code>javax.servlet.http.PushBuilder</code>	<p>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 Client 端不支持 HTTP/2 功能，则注入的 <code>PushBuilder</code> 实例可以为 <code>null</code>。</p>
<code>java.security.Principal</code>	<p>当前通过身份验证的用户-可能是特定的 <code>Principal</code> 实现类(如果已知)。</p>
<code>HttpMethod</code>	<p>请求的 HTTP 方法。</p>
<code>java.util.Locale</code>	<p>当前的请求语言环境，由最具体的 <code>LocaleResolver</code> (实际上是配置的 <code>LocaleResolver</code> 或 <code>LocaleContextResolver</code>)确定。</p>
<code>java.util.TimeZone</code> + <code>java.time.ZoneId</code>	<p>与当前请求关联的时区，由 <code>LocaleContextResolver</code> 确定。</p>

控制器方法参数	Description
<code>java.io.InputStream</code> , <code>java.io.Reader</code>	用于访问 Servlet API 公开的原始请求正文。
<code>java.io.OutputStream</code> , <code>java.io.Writer</code>	用于访问 Servlet API 公开的原始响应正文。
<code>@PathVariable</code>	用于访问 URI 模板变量。参见 <a href="#">URI patterns</a> 。 。
<code>@MatrixVariable</code>	用于访问 URI 路径段中的名称/值对。参见 <a href="#">Matrix Variables</a> 。
<code>@RequestParam</code>	用于访问 Servlet 请求参数，包括 Multipart 文件。参数值将转换为声明的方法参数类型。参见 <a href="#">@RequestParam</a> 和 <a href="#">Multipart</a> 。

请注意，对于简单的参数值，使用 `@RequestParam` 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 Headers。Headers 值将转换为声明的方法参数类型。参见[@RequestHeader](#)。

| `@CookieValue` | 用于访问 cookie。Cookies 值将转换为声明的方法参数类型。参见[@CookieValue](#)。

| `@RequestBody` | 用于访问 HTTP 请求正文。通过使用 `HttpMessageConverter` 实现，主体内容将转换为声明的方法参数类型。参见[@RequestBody](#)。

- | `HttpEntity<B>` | 用于访问请求 Headers 和正文。主体用 `HttpMessageConverter` 转换。参见[HttpEntity](#)。
- | `@RequestPart` | 要访问 `multipart/form-data` 请求中的 Component, 请用 `HttpMessageConverter` 转换 Component 的主体。参见[Multipart](#)。
- | `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(即附加到查询字符串)的情况下使用的属性，以及将 Flash 属性临时存储直到重定向后的请求。参见[Redirect Attributes](#)和[Flash Attributes](#)。
- | `@ModelAttribute` | 用于访问应用了数据绑定和验证的模型中的现有属性(如果不存在，则进行实例化)。参见[@ModelAttribute](#)以及[Model](#)和[DataBinder](#)。

请注意，使用 `@ModelAttribute` 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` | 用于访问命令对象的验证和数据绑定(即 `@ModelAttribute` 参数)中的错误或 `@RequestBody` 或 `@RequestPart` 参数的验证中的错误。您必须在经过验证的方法参数后立即声明 `Errors` 或 `BindingResult` 参数。
- | `SessionStatus` 类级 `@SessionAttributes` | 用于标记表单处理完成，将触发清除通过类级 `@SessionAttributes` `Comments` 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 `servletMap` 的 `Literals` 部分的 URL。参见[URL Links](#)。
- | `@SessionAttribute` | 用于访问任何会话属性，与由于类级别 `@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](#)。

|任何其他参数|如果方法参数与该表中的任何较早值都不匹配，并且是简单类型(由 [BeanUtils#isSimpleProperty](#) 确定，则将其解析为 `@RequestParam`。否则，将其解析为 `@ModelAttribute`。

## Return Values

[与 Spring WebFlux 中的相同](#)

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 Reactive 类型。

控制器方法返回值	Description
<code>@ResponseBody</code>	返回值通过 <code>HttpMessageConverter</code> 实现进行转换，并写入响应中。参见 <a href="#">@ResponseBody</a> 。
<code>HttpEntity&lt;B&gt;</code> , <code>ResponseEntity&lt;B&gt;</code>	指定完整响应(包括 <code>HTTPHeaders</code> 和正文)的返回值将通过 <code>HttpMessageConverter</code> 实现进行转换并写入响应中。参见 <a href="#"> ResponseEntity</a> 。
<code>HttpHeaders</code>	用于返回不包含标题的响应。
<code>String</code>	用 <code>ViewResolver</code> 实现解析的视图名称，并与通过命令对象和 <code>@ModelAttribute</code> 方法确定的隐式模型一起使用。处理程序方法还可以 pass 语句 <code>Model</code> 参数(请参见 <a href="#">Explicit Registrations</a> )以编程方式丰富模型。
<code>View</code>	用于与隐式模型一起渲染的 <code>View</code> 实例-通过命

控制器方法返回值	Description
	令对象和 <code>@ModelAttribute</code> 方法确定。处理程序方法还可以 pass 语句 <code>Model</code> 参数(请参见 <a href="#">Explicit Registrations</a> )以编程方式丰富模型。
<code>java.util.Map</code> , <code>org.springframework.ui.Model</code>	要添加到隐式模型的属性，视图名称通过 <code>RequestToViewNameTranslator</code> 隐式确定。
<code>@ModelAttribute</code>	要添加到模型的属性，视图名称通过 <code>RequestToViewNameTranslator</code> 隐式确定。

请注意，`@ModelAttribute` 是可选的。请参见此表末尾的“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` | 具有 `void` 返回类型(或 `null` 返回值)的方法，如果它也具有 `ServletResponse` , `OutputStream` 参数或 `@ResponseStatus` Comments，则认为已完全处理了响应。如果控制器对 `ETag` 或 `lastModified` 时间戳进行了肯定检查，则也是如此(有关详细信息，请参见 [Controllers](#))。

如果以上所有条件都不成立，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 控制器指示默认的视图名称选择。

- | `DeferredResult<V>` | 从任何线程异步生成任何上述返回值，例如由于某些事件或回调的结果。
  - 。参见[Asynchronous Requests](#)和[DeferredResult](#)。
- | `Callable<V>` | 在 Spring MVCManagement 的线程中异步产生上述任何返回值。参见 [Asynchronous Requests](#)和[Callable](#)。

| `ListenableFuture<V>` , `java.util.concurrent.CompletionStage<V>` ,  
`java.util.concurrent.CompletableFuture<V>` | 替代 `DeferredResult` , 以方便使用(例如  
, 当基础服务返回其中之一时)。

| `ResponseBodyEmitter` , `SseEmitter` | 使用 `HttpMessageConverter` 实现异步发出对象流  
以将其写入响应。也支持作为 `ResponseEntity` 的主体。参见[Asynchronous Requests](#)和[HTTP Streaming](#)。

| `StreamingResponseBody` | 异步写入响应 `OutputStream` 。也支持作为 `ResponseEntity` 的  
主体。参见[Asynchronous Requests](#)和[HTTP Streaming](#)。

| Reactive 类型-反应器, RxJava 或其他通过 `ReactiveAdapterRegistry` | 替代

`DeferredResult` 的多值流(例如 `Flux` , `Observable` )收集到 `List` 。

对于流方案(例如 `text/event-stream` , `application/json+stream` ), 将使用 `SseEmitter` 和  
`ResponseBodyEmitter` 代替, 其中 `ServletOutputStream` 阻塞 I/O 在 Spring  
MVCManagement 的线程上执行, 并且在每次写入完成时施加反压。  
参见[Asynchronous Requests](#)和[Reactive Types](#)。

| 任何其他返回值|与该表中的任何较早值不匹配且为 `String` 或 `void` 的任何返回值均被视为视图  
名称(适用于通过 `RequestToViewNameTranslator` 进行默认视图名称选择), 前提是该操作不简单  
类型, 由[BeanUtils#isSimpleProperty](#)确定。简单类型的值仍然无法解析。

## Type Conversion

与 [Spring WebFlux](#) 中的相同

如果声明的参数不是 `String` , 则表示基于 `String` 的请求 Importing 的某些带 Comments 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` 等)。您可以通过 `WebDataBinder` (请参见[DataBinder](#))或通过向 `FormattingConversionService` 注册 `Formatters` 来定制类型转换。参见[Spring 字段格式](#)。

## Matrix Variables

[与 Spring WebFlux 中的相同](#)

[RFC 3986](#)讨论路径段中的名称/值对。在 Spring MVC 中，基于 Tim Berners-Lee 的["old post"](#)，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(例如

`color:red;color:green;color:blue`)来指定多个值。

如果期望 URL 包含矩阵变量，则控制器方法的请求 Map 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 Sequences 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
}
```

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中的歧义。下面的示例演示如何做到这一点：

```
// 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，您需要使用

`removeSemicolonContent=false` 到 [Path Matching](#) 来设置 `UrlPathHelper`。在 MVC XML 名称空间中，可以设置 `<mvc:annotation-driven enable-matrix-variables="true"/>`。

## @RequestParam

[与 Spring WebFlux 中的相同](#)

您可以使用 `@RequestParam` 注解将 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) { (1)
        Pet pet = this.clinic.loadPet(petId);
        model.addAttribute("pet", pet);
        return "petForm";
    }
}
```

```
    }  
    // ...  
}
```

- (1) 使用 `@RequestParam` 绑定 `petId`。

默认情况下，使用此注解的方法参数是必需的，但是您可以通过将 `@RequestParam` 注解的 `required` 标志设置为 `false` 或通过 `java.util.Optional` 包装器声明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 `String`，则会自动应用类型转换。参见[Type Conversion](#)。

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` 注解声明为 `Map<String, String>` 或 `MultiValueMap<String, String>`，而未在注解中指定参数名，则将使用每个给定参数名的请求参数值填充 Map。

请注意，使用 `@RequestParam` 是可选的(例如，设置其属性)。默认情况下，任何简单值类型(由 [BeanUtils#isSimpleProperty](#) 确定)且未由其他任何参数解析器解析的参数都将被视为带有 `@RequestParam` Comments。

## `@RequestHeader`

[与 Spring WebFlux 中的相同](#)

您可以使用 `@RequestHeader` 注解将请求 Headers 绑定到控制器中的方法参数。

考虑以下带有 Headers 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

下面的示例获取 `Accept-Encoding` 和 `Keep-Alive` Headers 的值：

```
@GetMapping("/demo")
public void handle(
    @RequestHeader("Accept-Encoding") String encoding, (1)
    @RequestHeader("Keep-Alive") long keepAlive) { (2)
    //...
}
```

- (1) 获取 `Accept-Encoding` Headers 的值。
- (2) 获取 `Keep-Alive` Headers 的值。

如果目标方法的参数类型不是 `String`，则将自动应用类型转换。参见[Type Conversion](#)。

在 `Map<String, String>`，`MultiValueMap<String, String>` 或 `HttpHeaders` 参数上使用 `@RequestHeader` Comments 时，将使用所有 Headers 值填充 Map。

### Tip

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，带有 `@RequestHeader("Accept")` Comments 的方法参数可以是 `String` 类型，也可以是 `String[]` 或 `List<String>` 类型。

## @CookieValue

[与 Spring WebFlux 中的相同](#)

您可以使用 `@CookieValue` 注解将 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) { (1)
    //...
}
```

- (1) 获取 `JSESSIONID` cookie 的值。

如果目标方法的参数类型不是 `String`，那么将自动应用类型转换。参见[Type Conversion](#)。

## @ModelAttribute

[与 Spring WebFlux 中的相同](#)

您可以在方法参数上使用 `@ModelAttribute` Comments 来访问模型中的属性，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) { } (1)
```

- (1) 绑定 `Pet` 的实例。

上面的 `Pet` 实例解析如下：

- 从模型(如果已使用[Model](#)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。
- 从带有与 Servlet 请求参数匹配的参数的“主要构造函数”的调用开始。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。

虽然通常使用[Model](#)来为模型填充属性，但另一种替代方法是将 `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[Validation](#)。有关自定义数据绑定的更多信息，请参见[DataBinder](#)。

数据绑定可能会导致错误。默认情况下，引发 `BindException`。但是，要检查 controller 方法中的此类错误，可以在 `@ModelAttribute` 旁边紧接着添加 `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) { (1)
    if (result.hasErrors()) {
        return "petForm";
    }
    // ...
}
```

- (1) 在 `@ModelAttribute` 旁边添加一个 `BindingResult`。

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，您可以将 `Model` 注入控制器并直接访问它，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")
```

```
public String update(@Valid AccountUpdateForm form, BindingResult result,
    @ModelAttribute(binding=false) Account account) { (1)
    // ...
}
```

- (1) 设置 `@ModelAttribute(binding=false)`。

通过添加 `javax.validation.Valid` Comments 或 Spring 的 `@Validated` Comments(ee Bean validation 和 Spring validation), 可以在数据绑定之后自动应用验证。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";
    }
    // ...
}
```

- (1) 验证 `Pet` 实例。

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)且未由任何其他参数解析器解析的参数都将被视为以

`@ModelAttribute` Comments。

## @SessionAttributes

与 [Spring WebFlux 中的相同](#)

`@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 Comments，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名称或模型属性类型。

下面的示例使用 `@SessionAttributes` 注解：

```
@Controller
@SessionAttributes("pet") (1)
public class EditPetForm {
```

```
// ...  
}
```

- (1) 使用 `@SessionAttributes` Comments。

在第一个请求上，将名称为 `pet` 的模型属性添加到模型时，该属性会自动提升为 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") (1)  
public class EditPetForm {  
  
    // ...  
  
    @PostMapping("/pets/{id}")  
    public String handle(Pet pet, BindingResult errors, SessionStatus status) {  
        if (errors.hasErrors()) {  
            // ...  
        }  
        status.setComplete(); (2)  
        // ...  
    }  
}  
}
```

- (1) 将 `Pet` 值存储在 Servlet 会话中。
- (2) 从 Servlet 会话中清除 `Pet` 值。

## @SessionAttribute

[与 Spring WebFlux 中的相同](#)

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)Management)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` Comments，作为以下示例显示：

```
@RequestMapping("/")  
public String handle(@SessionAttribute User user) { (1)  
    // ...  
}
```

- (1) 使用 `@SessionAttribute` Comments。

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` 注入控制器方法。

为了将模型属性临时存储在会话中作为控制器工作流的一部分，请考虑使用[@SessionAttributes](#) 中所述的 `@SessionAttributes`。

## **@RequestAttribute**

[与 Spring WebFlux 中的相同](#)

与 `@SessionAttribute` 相似，您可以使用 `@RequestAttribute` 注解来访问先前创建的预先存在的请求属性(例如，由 Servlet `Filter` 或 `HandlerInterceptor` 创建)：

```
@GetMapping("/")
public String handle(@RequestAttribute Client client) { (1)
    // ...
}
```

- (1) 使用 `@RequestAttribute` Comments。

## **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 `Comments` 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)。为了避免此类属性出现在 URL 中的可能性，`@RequestMapping` 方法可以声明类型

`RedirectAttributes` 的参数，并使用它来指定确切的属性以供 `RedirectView` 使用。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，您可以使用该标志来表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，控制器方法应声明类型为 `RedirectAttributes` 的属性，或者，如果没有声明，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，建议将其设置为 `true`。

请注意，展开重定向 URL 时，本请求中的 URI 模板变量将自动变为可用，并且您需要通过 `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闪存属性。与其他重定向属性不同，Flash 属性保存在 HTTP 会话中(因此不会出现在 URL 中)。有关更多信息，请参见[Flash Attributes](#)。

## 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 Management `FlashMap` 实例。

Flash 属性支持始终处于“启用”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，都有一个具有从上一个请求(如果有)传递的属性的“Importing”`FlashMap`，和一个具有为后续请求保存的属性的“输出”`FlashMap`。通过

`RequestContextUtils` 中的静态方法，可以从 Spring MVC 中的任何位置访问这两个 `FlashMap` 实例。

带 `Comments` 的控制器通常不需要直接与 `FlashMap` 一起使用。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，在重定向之后，来自“Importing” `FlashMap` 的属性会自动添加到服务目标 URL 的控制器的 `Model` 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 Flash 属性。

为了减少发生此类问题的可能性，`RedirectView` 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“Importing” `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，但是可以通过重定向 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

### 与 Spring WebFlux 中的相同

在 `MultipartResolver` 成为 `enabled` 之后，将解析 `multipart/form-data` 的 POST 请求的内容并将其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>` 可以为同一参数名称解析多个文件。

如果将 `@RequestParam` 注解声明为 `Map<String, MultipartFile>` 或

`MultivalueMap<String, MultipartFile>`，而未在注解中指定参数名称，则将使用每个给定参数名的 `Multipart` 文件填充 `Map`。

### iNote

使用 `Servlet 3.0 Multipart` 解析时，您还可以将 `javax.servlet.http.Part` 而不是 Spring 的 `MultipartFile` 声明为方法参数或集合值类型。

您还可以将 `Multipart` 内容用作绑定到 [command object](#) 的数据的一部分。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  
            return "redirect:uploadSuccess";  
        }  
        return "redirect:uploadFailure";  
    }  
}
```

在 RESTful 服务方案中，也可以从非浏览器 Client 端提交 `Multipart` 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 `String` 来访问“元数据”部分，但您可能希望将其从 JSON 反序列化(类似于 `@RequestBody`)。用 [HttpMessageConverter](#) 转换后，使用 `@RequestPart` 注解来访问 Multipart：

```
@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` Comments，这两种 Comments 都会导致应用标准 Bean 验证。默认情况下，验证错误会导致 `MethodArgumentNotValidException`，它变成 400(BAD\_REQUEST)响应。或者，您可以通过 `Errors` 或 `BindingResult` 参数在控制器内部本地处理验证错误，如以下示例所示：

```
@PostMapping( "/")
public String handle(@Valid @RequestPart("meta-data") MetaData metadata,
                     BindingResult result) {
    // ...
}
```

## `@RequestBody`

与 [Spring WebFlux 中的相同](#)

您可以使用 `@RequestBody` 注解将请求正文读取并通过 `HttpMessageConverter` 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 [Message Converters](#) 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` Comments 结合使用，这两种 Comments 都会导致应用标准 Bean 验证。默认情况下，验证错误会导致 `MethodArgumentNotValidException`，它变成 400(BAD\_REQUEST) 响应。或者，您可以通过 `Errors` 或 `BindingResult` 参数在控制器内部本地处理验证错误，如以下示例所示：

```
@PostMapping("/accounts")
public void handle(@Valid @RequestBody Account account, BindingResult result) {
    // ...
}
```

## HttpEntity

[与 Spring WebFlux 中的相同](#)

`HttpEntity` 与使用 `@RequestBody` 大致相同，但是它基于一个容器对象，该对象公开了请求 Headers 和正文。以下 Lists 显示了一个示例：

```
@PostMapping("/accounts")
public void handle(HttpEntity<Account> entity) {
    // ...
}
```

## @ResponseBody

[与 Spring WebFlux 中的相同](#)

您可以在方法上使用 `@ResponseBody` 注解，以使返回值通过 `HttpMessageConverter` 序列化到响

应主体。以下 `Lists` 显示了一个示例：

```
@GetMapping("/accounts/{id}")
@ResponseBody
public Account handle() {
    // ...
}
```

`@ResponseBody` 在类级别上也受支持，在这种情况下，它被所有控制器方法继承。这就是 `@RestController` 的效果，它不过是用 `@Controller` 和 `@ResponseBody` 标记的元 Comments。

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 [Asynchronous Requests](#) 和 [Reactive Types](#)。

您可以使用 [MVC Config](#) 的 [Message Converters](#) 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。有关详情，请参见 [Jackson JSON](#)。

## ResponseEntity

[与 Spring WebFlux 中的相同](#)

`ResponseEntity` 类似于 [@ResponseBody](#)，但具有状态和标题。例如：

```
@GetMapping("/something")
public ResponseEntity<String> handle() {
    String body = ... ;
    String etag = ... ;
    return ResponseEntity.ok().eTag(etag).build(body);
}
```

Spring MVC 支持使用单个值 [reactive type](#) 异步生成 `ResponseEntity` 和/或主体的单值和多值反应类型。

## Jackson JSON

Spring 提供了对 Jackson JSON 库的支持。

## Jackson 序列化视图

[与 Spring WebFlux 中的相同](#)

Spring MVC 为[Jackson 的序列化视图](#)提供了内置支持，该支持仅渲染 `Object` 中所有字段的一部分。要将它与 `@ResponseBody` 或  `ResponseEntity` 控制器方法一起使用，可以使用 Jackson 的 `@JsonView` 注解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;
        }
    }
}
```

### Note

`@JsonView` 允许一组视图类，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。

对于依赖视图分辨率的控制器，您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

[与 Spring WebFlux 中的相同](#)

您可以使用 `@ModelAttribute` 注解：

- 在 `method argument` in `@RequestMapping` 方法中，从模型创建或访问 `Object` 并将其通过 `WebDataBinder` 绑定到请求。
- 作为 `@Controller` 或 `@ControllerAdvice` 类中的方法级 `Comments`，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。

`@ModelAttribute` 方法也可以通过 `@ControllerAdvice` 在控制器之间共享。有关更多详细信息，请参见[Controller Advice](#)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，但 `@ModelAttribute` 本身或与请求正文相关的任何东西除外。

以下示例显示了 `@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);  
}
```

### iNote

如果未明确指定名称，则根据 `Object` 类型选择默认名称，如[Conventions](#)的 javadoc 中所述。您始终可以通过使用重载的 `addAttribute` 方法或通过 `@ModelAttribute` 上的 `name` 属性(用于返回值)来分配显式名称。

您也可以将 `@ModelAttribute` 用作 `@RequestMapping` 方法的方法级 Comments，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，因为这是 HTML 控制器中的默认行为，除非返回值是 `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

[与 Spring WebFlux 中的相同](#)

`@Controller` 或 `@ControllerAdvice` 类可以具有 `@InitBinder` 个方法来初始化

`WebDataBinder` 的实例，而这些实例又可以：

- 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, Headers, 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` 一个方法可以注册特定于控制器的 `java.bean.PropertyEditor` 或 Spring `Converter` 和 `Formatter` 组件。此外, 您可以使用 [MVC config](#) 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, 但 `@ModelAttribute` (命令对象)参数除外。通常, 它们使用 `WebDataBinder` 参数(用于注册)和 `void` 返回值声明。以下 Lists 显示了一个示例：

```
@Controller
public class FormController {

    @InitBinder (1)
    public void initBinder(WebDataBinder binder) {
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        dateFormat.setLenient(false);
        binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, false));
    }

    // ...
}
```

- (1) 定义 `@InitBinder` 方法。

另外, 当您通过共享 `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 (1)
```

```
protected void initBinder(WebDataBinder binder) {
    binder.addCustomFormatter(new DateFormatter("yyyy-MM-dd"));
}

// ...
}
```

- (1) 在自定义格式化程序上定义 `@InitBinder` 方法。

### 1.3.6. Exceptions

[与 Spring WebFlux 中的相同](#)

`@Controller` 和 `@ControllerAdvice` 类可以具有 `@ExceptionHandler` 个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，直接抛出 `IOException`)或顶级包装异常中的直接原因(例如，在 `IllegalStateException` 内包装的 `IOException`)相匹配。

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。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` 用于根据异常对引发的异常类型的深度进行排序。

另外，`Comments` 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) {
    // ...
}
```

## iNote

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，通常以实际的 `FileSystemException` 或 `RemoteException` 实例作为参数来调用该方法，因为这两个实例都从 `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。

`handle(Exception)` 变体中的行为甚至更简单。在包装方案中，总是使用包装器异常来调用此方法，在这种情况下，实际匹配的异常将通过 `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。

通常，我们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，每个方法都通过其签名匹配单个特定的异常类型。

在多 `@ControllerAdvice` 的安排中，我们建议在以相应 Sequences 优先的 `@ControllerAdvice` 上声明您的主根异常 Map。尽管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` 方法实现可以选择以放弃原始形式重新处理给定异常实例的方法。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决方案链中传播, 就像给定的 `@ExceptionHandler` 方法最初不会匹配一样。

Spring MVC 中对 `@ExceptionHandler` 方法的支持构建在 `DispatcherServlet` 级别的 [HandlerExceptionResolver](#) 机制上。

## Method Arguments

`@ExceptionHandler` 方法支持以下参数:

Method argument	Description
<code>Exception type</code>	用于访问引发的异常。
<code>HandlerMethod</code>	用于访问引发异常的控制器方法。
<code>WebRequest</code> , <code>NativeWebRequest</code>	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。
<code>javax.servlet.ServletRequest</code> , <code>javax.servlet.ServletResponse</code>	选择任何特定的请求或响应类型(例如 <code>ServletRequest</code> 或 <code>HttpServletRequest</code> 或 Spring 的 <code>MultipartRequest</code> 或 <code>MultipartHttpServletRequest</code> )。
<code>javax.servlet.http.HttpSession</code>	强制会话的存在。因此, 这样的参数永远不会有是 <code>null</code> 。

请注意，会话访问不是线程安全的。如果允许多个请求同时访问会话，请考虑将 `RequestMappingHandlerAdapter` 实例的 `synchronizeOnSession` 标志设置为 `true`。

| `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 `LocaleContextResolver` 确定。

| `java.io.OutputStream` , `java.io.Writer` | 用于访问原始响应主体，如 Servlet API 所公开。

| `java.util.Map` , `org.springframework.ui.Model` ,  
`org.springframework.ui.ModelMap` | 用于访问模型以进行错误响应。总是空的。

| `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)和 flash 属性，这些属性将临时存储直到重定向后的请求。参见[Redirect Attributes](#)和[Flash Attributes](#)。

| `@SessionAttribute` | 用于访问任何会话属性，与由于类级别 `@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](#)。

## Return Values

`@ExceptionHandler` 方法支持以下返回值：

Return value	Description
<code>@ResponseBody</code>	返回值通过 <code>HttpMessageConverter</code> 个实例

Return value	Description
	<p>进行转换并写入响应中。参见 <a href="#">@ResponseBody</a>。</p>
<pre>HttpEntity&lt;B&gt; , ResponseEntity&lt;B&gt;</pre>	<p>返回值指定完整的响应(包括 HttpHeaders 和正文)通过 <a href="#">HttpMessageConverter</a> 实例进行转换并写入响应中。参见<a href="#"> ResponseEntity</a>。</p>
<pre>String</pre>	<p>要通过 <a href="#">ViewResolver</a> 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 <a href="#">@ModelAttribute</a> 方法确定。处理程序方法还可以pass 语句 <a href="#">Model</a> 参数(如前所述)以编程方式丰富模型。</p>
<pre>View</pre>	<p>通过命令对象和 <a href="#">@ModelAttribute</a> 方法确定的用于与隐式模型一起渲染的 <a href="#">View</a> 实例。处理程序方法还可以pass 语句 <a href="#">Model</a> 参数(先前描述)来以编程方式丰富模型。</p>
<pre>java.util.Map , org.springframework.ui.Model</pre>	<p>通过 <a href="#">RequestToViewNameTranslator</a> 隐式确定的视图名称将添加到隐式模型的属性。</p>
<pre>@ModelAttribute</pre>	<p>通过 <a href="#">RequestToViewNameTranslator</a> 隐式确定的视图名称将添加到模型的属性。</p>

请注意，`@ModelAttribute` 是可选的。请参见此表末尾的“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` | 具有 `void` 返回类型(或 `null` 返回值)的方法，如果它也具有 `ServletResponse`，

`OutputStream` 自变量或 `@ResponseStatus` Comments，则认为已完全处理了响应。如果控制器对 `ETag` 或 `lastModified` 时间戳进行了肯定检查，则也是如此(有关详细信息，请参见 [Controllers](#))。

如果以上所有条件都不成立，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。

| 其他任何返回值 | 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，则仍然无法解决。

## REST API 异常

[与 Spring WebFlux 中的相同](#)

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` 可以使用具有 `ResponseEntity` 返回值的 `@ExceptionHandler` 方法来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。

在响应正文中实现具有错误详细信息的全局异常处理的应用程序应考虑扩展  [ResponseEntityExceptionHandler](#)，该扩展为 Spring MVC 引发的异常提供处理，并提供用于自定义响应正文的钩子。要使用此功能，请创建 `ResponseEntityExceptionHandler` 的子类，并用 `@ControllerAdvice` 对其进行 Comments，覆盖必要的方法，然后将其声明为 Spring bean。

### 1.3.7. 控制器建议

[与 Spring WebFlux 中的相同](#)

通常，`@ExceptionHandler`，`@InitBinder` 和 `@ModelAttribute` 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[component scanning](#)将此类注册为 Spring Bean。`@RestControllerAdvice` 也是标有 `@ControllerAdvice` 和 `@ResponseBody` 的元 Comments，从本质上讲，这意味着 `@ExceptionHandler` 方法通过消息转换(与视图分辨率或模板渲染)呈现给响应主体。

启动时，`@RequestMapping` 和 `@ExceptionHandler` 方法的基础结构类将检测 `@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\*本地方法之后(来自 `@Controller`)被应用。相比之下，全局 `@ModelAttribute` 和 `@InitBinder` 方法在本地方法之前被应用。

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注解上的属性将其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 `selectors` 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。

## 1.4. URI 链接

[与 Spring WebFlux 中的相同](#)

本节描述了 Spring 框架中可用于 URI 的各种选项。

### 1.4.1. UriComponents

Spring MVC 和 Spring WebFlux

`UriComponentsBuilder` 帮助从具有变量的 URI 模板构建 URI，如以下示例所示：

```
UriComponents uriComponents = UriComponentsBuilder
    .fromUriString("http://example.com/hotels/{hotel}") (1)
    .queryParam("q", "{q}") (2)
    .encode() (3)
    .build(); (4)

URI uri = uriComponents.expand("Westin", "123").toUri(); (5)
```

- (1) 带有 URI 模板的静态工厂方法。
- (2) 添加或替换 URI 组件。
- (3) 请求对 URI 模板和 URI 变量进行编码。
- (4) 构建 `UriComponents`。
- (5) 展开变量并获得 `URI`。

可以将前面的示例合并为一个链，并使用 `buildAndExpand` 进行缩短，如以下示例所示：

```
URI uri = UriComponentsBuilder
    .fromUriString("http://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("http://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("http://example.com/hotels/{hotel}?q={q}")
    .build("Westin", "123");
```

## 1.4.2. UriBuilder

Spring MVC 和 Spring WebFlux

[UriComponentsBuilder](#) 实现 [UriBuilder](#)。您可以依次创建 [UriBuilder](#) 和 [UriBuilderFactory](#)。[UriBuilderFactory](#) 和 [UriBuilder](#) 一起提供了一种可插入的机制，用于基于共享配置(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。

以下示例显示了如何配置 [RestTemplate](#)：

```
// import org.springframework.web.util.DefaultUriBuilderFactory.EncodingMode;

String baseUrl = "http://example.org";
DefaultUriBuilderFactory factory = new DefaultUriBuilderFactory(baseUrl);
factory.setEncodingMode(EncodingMode.TEMPLATE_AND_VARIABLES);

RestTemplate restTemplate = new RestTemplate();
restTemplate.setUriTemplateHandler(factory);
```

以下示例配置 [WebClient](#)：

```
// import org.springframework.web.util.DefaultUriBuilderFactory.EncodingMode;

String baseUrl = "http://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，而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 = "http://example.com";
DefaultUriBuilderFactory uriBuilderFactory = new DefaultUriBuilderFactory(baseUrl);

URI uri = uriBuilderFactory.uriString("/hotels/{hotel}")
    .queryParam("q", "{q}")
    .build("Westin", "123");
```

### 1.4.3. URI 编码

Spring MVC 和 Spring WebFlux

`UriComponentsBuilder` 公开了两个级别的编码选项：

- [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\(\)](#): 在扩展 URI 变量后\*编码 URI 组件。

这两个选项都使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。

#### Tip

考虑 “;” ， 它在路径上是合法的，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：

```
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 = "http://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) 来预编码 URI 模板，并在扩展时严格编码 URI 变量。
- `VALUES_ONLY`：不对 URI 模板进行编码，而是在将它们扩展到模板之前通过 `UriUtils#encodeUriVariables` 对 URI 变量进行严格编码。
- `URI_COMPONENTS`：在扩展 URI 变量之后\*，使用与较早列表中第二个选项相对应的 `UriComponents#encode()` 来编码 URI 组件值。

- **NONE**：未应用编码。

出于历史原因和向后兼容性，`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. 相对 Servlet 请求

您可以使用 `ServletUriComponentsBuilder` 来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...

ServletUriComponentsBuilder ucb = ServletUriComponentsBuilder.fromServletMapping(request)
    .path("/accounts").build()
```

#### Note

从 5.1 开始，`ServletUriComponentsBuilder` 会忽略 `Forwarded` 和 `X-Forwarded-*` Headers 中的信息，这些 Headers 指定了 Client 端起源的地址。考虑使用 [ForwardedHeaderFilter](#) 提取和使用或丢弃此类 Headers。

## 1.4.5. 链接到控制器

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](#) 来填充任何剩余的 URI 变量，例如从类型级请求 Map 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 `MvcUriComponentsBuilder.on` 静态导入)：

```
UriComponents uriComponents = MvcUriComponentsBuilder
    .fromMethodCall(on(BookingController.class).getBooking(21)).buildAndExpand(42);

URI uri = uriComponents.encode().toUri();
```

### Note

当控制器方法签名可用于 `fromMethodCall` 链接创建时，它们的设计受到限制。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 `final`。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` 或甚至是普通 `Object` (返回值为 `String`)。

较早的示例在 `MvcUriComponentsBuilder` 中使用静态方法。在内部，他们依靠 `ServletUriComponentsBuilder` 从当前请求的方案，主机，端口，上下文路径和 `servlet` 路径准备基本 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，从请求路径中删除并需要重新设置的语言环境前缀)。插入链接)。

在这种情况下，您可以使用接受 `fromXXX` 的静态 `fromXXX` 重载方法来使用基本 URL。或者，您可以使用基本 URL 创建 `MvcUriComponentsBuilder` 的实例，然后使用基于实例的 `withXXX` 方法。例如，以下 `Lists` 使用 `withMethodCall`：

```
UriComponentsBuilder base = ServletUriComponentsBuilder.fromCurrentContextPath().path("MvcUriComponentsBuilder builder = MvcUriComponentsBuilder.relativeTo(base);
builder.withMethodCall(on(BindingController.class).getBooking(21)).buildAndExpand(42);

URI uri = uriComponents.encode().toUri();
```

#### iNote

从 5.1 开始，`MvcUriComponentsBuilder` 会忽略 `Forwarded` 和 `X-Forwarded-*` Headers 中的信息，这些 Headers 指定了 Client 端起源的地址。考虑使用 [ForwardedHeaderFilter](#) 提取和使用或丢弃此类 Headers。

## 1.4.6. 视图中的链接

在 Thymeleaf, FreeMarker 或 JSP 之类的视图中，可以通过引用每个请求 Map 的隐式或显式分配的名称来构建到带 Comments 的控制器的链接。

考虑以下示例：

```
@RequestMapping("/people/{id}/addresses")
public class PersonAddressController {

    @RequestMapping("/{country}")
    public ResponseEntity 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)，但是可以轻松定义自己的函数或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，其默认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`”)。如果存在名称冲突，则可以使用 `@RequestMapping(name="...")` 分配一个明确的名称或实现自己的 `HandlerMethodMappingNamingStrategy`。

## 1.5. 异步请求

[与 WebFlux 相比](#)

Spring MVC 与 Servlet 3.0 异步请求 [processing](#) 进行了广泛的集成：

- 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 [stream](#) 多个值，包括 [SSE](#) 和 [raw data](#)。
- 控制器可以使用反应式 Client 端，并返回 [reactive types](#) 进行响应处理。

### 1.5.1. DeferredResult

[与 WebFlux 相比](#)

一旦异步请求处理功能在 Servlet 容器中为 [enabled](#), 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

#### 与 WebFlux 相比

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 [configured](#) [TaskExecutor](#) 运行给定的任务来获取返回值。

### 1.5.3. Processing

#### 与 WebFlux 相比

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。这样做的主要效

果是 Servlet(以及所有过滤器)可以退出，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`，您可以将其用于对异步处理的进一步控制。例如，它提供了 `dispatch` 方法，与 Servlet API 的转发类似，不同之处在于，它允许应用程序恢复 Servlet 容器线程上的请求处理。
- `ServletRequest` 提供对当前 `DispatcherType` 的访问，您可以使用它们区分处理初始请求，异步调度，转发和其他调度程序类型。

`DeferredResult` 处理如下：

- 控制器返回一个 `DeferredResult` 并将其保存在一些可以访问它的内存队列或列表中。
- 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` 处理如下：

- 控制器返回 `Callable`。
- Spring MVC 调用 `request.startAsync()` 并将 `Callable` 提交给 `TaskExecutor` 以便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` 异步生成的返回值恢复处理。

有关进一步的背景知识，您还可以阅读[博客文章](#)，它介绍了 Spring MVC 3.2 中的异步请求处理支持。

## Exception Handling

使用 `DeferredResult` 时，可以选择是呼叫  `setResult` 还是  `setErrorResult`，但有 `exception`。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，该异常将通过常规的异常处理机制(例如，调用 `@ExceptionHandler` 方法)进行处理。

当您使用 `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。

## Interception

`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更深入地集成(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](#)。`Callable` 可以替代 `WebAsyncTask`，后者提供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)，在此期间再

次 Map 控制器，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 [Reactive Types](#) 作为控制器方法中的返回值。Spring MVC 甚至支持流式传输，包括 Reactive 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)。

另一个根本的区别是，Spring MVC 在控制器方法参数中不支持异步或 Reactive 类型(例如

`@RequestBody`，`@RequestPart` 等)，也没有对异步和 Reactive 类型作为模型属性的任何明确支持。Spring WebFlux 确实支持所有这些。

#### 1.5.4. HTTP 流

##### [与 Spring WebFlux 中的相同](#)

您可以将 `DeferredResult` 和 `Callable` 用作单个异步返回值。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。

##### Objects

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` 的正文，以自定义响应的状态和标题。  
。

当 `emitter` 抛出  `IOException` 时(例如，如果远程 Client 端离开了)，应用程序不负责清理连接，并且不应调用 `emitter.complete` 或 `emitter.completeWithError`。取而代之的是，Servlet 容器自动启动  `AsyncListener` 错误通知，Spring MVC 在其中发出 `completeWithError` 调用。该调用依次对应用程序执行最后的  `ASYNC` 分派，在此期间 Spring MVC 调用配置的异常解析器并完成请求。

## SSE

`SseEmitter` (`ResponseBodyEmitter` 的子类)提供对[Server-Sent Events](#)的支持，其中根据 W3C SSE 规范格式化从服务器发送的事件。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();
```

尽管 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 messaging](#)和[SockJS fallback](#)传输(包括 SSE)一起使用，这些传输针对广泛的浏览器。

另请参阅[previous section](#)以获取有关异常处理的说明。

## Raw Data

有时，绕过消息转换并直接流式传输到响应 `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` 中的正文，以自定义响应的状态和标题。

### 1.5.5. 反应类型

#### 与 Spring WebFlux 中的相同

Spring MVC 支持在控制器中使用反应式 Client 端库(另请参阅 WebFlux 部分中的[Reactive Libraries](#))。这包括 `spring-webflux` 中的  `WebClient` 以及其他内容，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。

Reactive 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)。
- 类似于使用 `ResponseBodyEmitter` 或 `SseEmitter`，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` (Reactor)或 `Observable` (RxJava)。应用程序还可以返回 `Flux<ServerSentEvent>` 或

`Observable<ServerSentEvent>`。

- 类似于使用 `DeferredResult<List<?>>`，可以适应具有任何其他媒体类型(例如 `application/json`)的多值流。

### Tip

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，支持了反作用背压，但响应的写仍处于阻塞状态，并通过[configured TaskExecutor](#) 在单独的线程上执行，以避免阻塞上游源(例如 `WebClient` 返回的 `Flux`)。默认情况下，`SimpleAsyncTaskExecutor` 用于阻止写入，但是在负载下不适合。如果您打算使用 Reactive 类型进行流式传输，则应使用[MVC configuration](#)来配置任务 Actuator。

## 1.5.6. Disconnects

[与 Spring WebFlux 中的相同](#)

当远程 Client 端离开时，Servlet API 不提供任何通知。因此，在通过[SseEmitter](#)或\<<mvc-ann-async-reactive-types,reactive types>流式传输到响应时，定期发送数据很重要，因为如果 Client 端断开连接，写入将失败。发送可以采取空(仅 Comment)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[通过 WebSocket 进行 STOMP](#)或带有[SockJS](#)的 WebSocket)。

## 1.5.7. Configuration

[与 WebFlux 相比](#)

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 Container

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 Map 以处理 `ASYNC` `javax.servlet.DispatcherType`。

在 Java 配置中，当您使用 `AbstractAnnotationConfigDispatcherServletInitializer` 初始化 Servlet 容器时，这是自动完成的。

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 Map。

## 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` 用于在与[Reactive Types](#)流式传输时阻止写操作，并用于执行从控制器方法返回的 `Callable` 实例。如果您使用反应式类型进行流式处理或具有返回 `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` 个实施和 `CallableProcessingInterceptor` 个实施。

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。

## 1.6. CORS

### 与 Spring WebFlux 中的相同

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. Introduction

#### 与 Spring WebFlux 中的相同

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)是由[most browsers](#)实现的[W3C specification](#)，它使您可以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

#### 与 Spring WebFlux 中的相同

CORS 规范区分飞行前，简单和实际要求。要了解 CORS 的工作原理，您可以阅读[this article](#)，或者阅读规范以获得更多详细信息。

Spring MVC `HandlerMapping` 实现为 CORS 提供内置支持。成功将请求 Map 到处理程序后，

`HandlerMapping` 个实现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 Headers。

为了启用跨域请求(即 `Origin` Headers 存在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，则飞行前请求将被拒绝。没有将 `CORSHeaders` 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。

每个 `HandlerMapping` 可以分别是 [configured](#)，并具有基于 URL 模式的 `CorsConfiguration`

Map。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 Map，这导致将单个全局 Map 传递给所有 `HandlerMapping` 实例。

您可以将 `HandlerMapping` 级别的全局 CORS 配置与更细粒度的处理器级别的 CORS 配置结合使用。例如，带 `Comments` 的控制器可以使用类或方法级别的 `@CrossOrigin` `Comments`(其他处理器可以实现 `CorsConfigurationSource`)。

整体配置和局部配置的组合规则通常是相加的，例如，所有全局和所有本地来源。对于仅接受单个值的那些属性(例如 `allowCredentials` 和 `maxAge`)，局部变量将覆盖全局值。有关更多详细信息，请参见[CorsConfiguration#combine\(CorsConfiguration\)](#)。

### Tip

要从源中了解更多信息或进行高级自定义，请查看后面的代码：

- `CorsConfiguration`
- `CorsProcessor` , `DefaultCorsProcessor`
- `AbstractHandlerMapping`

### 1.6.3. @CrossOrigin

[与 Spring WebFlux 中的相同](#)

`@CrossOrigin`Comments 启用带 `Comments` 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) {
    // ...
}
```

默认情况下，`@CrossOrigin` 允许：

- All origins.
- All headers.
- 控制器方法 Map 到的所有 HTTP 方法。

默认情况下未启用 `allowedCredentials`，因为它构建了一个信任级别，该级别公开了敏感的用户特定信息(例如 cookie 和 CSRF 令牌)，仅应在适当的地方使用。

`maxAge` 设置为 30 分钟。

`@CrossOrigin` 在类级别也受支持，并且被所有方法继承，如以下示例所示：

```
@CrossOrigin(origins = "http://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使用 `@CrossOrigin`，如以下示例所示：

```
@CrossOrigin(maxAge = 3600)
@RestController
@RequestMapping("/account")
public class AccountController {
```

```
@CrossOrigin("http://domain2.com")
@GetMapping("/{id}")
public Account retrieve(@PathVariable Long id) {
    // ...
}

@Override
@DeleteMapping("/{id}")
public void remove(@PathVariable Long id) {
    // ...
}
```

## 1.6.4. 全局配置

### 与 Spring WebFlux 中的相同

除了细粒度的控制器方法级别配置之外，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` Map。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.
- All headers.
- `GET`，`HEAD` 和 `POST` 方法。

默认情况下未启用 `allowedCredentials`，因为它构建了一个信任级别，该级别公开了敏感的用户特定信息(例如 cookie 和 CSRF 令牌)，仅应在适当的地方使用。

`maxAge` 设置为 30 分钟。

## Java Configuration

### 与 Spring WebFlux 中的相同

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("http://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="http://domain1.com, http://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="http://domain1.com" />

</mvc:cors>
```

### 1.6.5. CORS 过滤器

[与 Spring WebFlux 中的相同](#)

您可以通过内置的[CorsFilter](#)来应用 CORS 支持。

#### Note

如果您尝试将 `CorsFilter` 与 Spring Security 一起使用，请记住 Spring Security 的 CORS 具有[built-in support](#)。

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);
config.addAllowedOrigin("http://domain1.com");
config.addAllowedHeader("*");
config.addAllowedMethod("*");

UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
source.registerCorsConfiguration("/**", config);

CorsFilter filter = new CorsFilter(source);
```

## 1.7. 网络安全

[与 Spring WebFlux 中的相同](#)

[Spring Security](#) 项目为保护 Web 应用程序免受恶意攻击提供支持。请参阅 Spring Security 参考文档，包括：

- [Spring MVC 安全性](#)
- [Spring MVC 测试支持](#)
- [CSRF protection](#)
- [安全响应 Headers](#)

[HDIV](#) 是另一个与 Spring MVC 集成的 Web 安全框架。

## 1.8. HTTP 缓存

[与 Spring WebFlux 中的相同](#)

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 Headers 以及随后的条件请求 Headers(例如 `Last-Modified` 和 `ETag`)展开。`Cache-Control` 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` Headers 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** Headers 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

[与 Spring WebFlux 中的相同](#)

**CacheControl** 支持配置与 **Cache-Control** Headers 相关的设置，并且在许多地方都作为参数接受

:

- [WebContentInterceptor](#)
- [WebContentGenerator](#)
- [Controllers](#)
- [Static Resources](#)

[RFC 7234](#) 描述了 **Cache-Control** 响应 Headers 的所有可能的指令，而 **CacheControl** 类型采用

了面向用例的方法，该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** 的值不会生成 **Cache-Control** 响应头。

- `0` 值可防止使用 `'Cache-Control: no-store'` 指令进行缓存。
- `n > 0` 值使用 `'Cache-Control: max-age=n'` 指令将给定响应缓存 `n` 秒。

## 1.8.2. Controllers

与 Spring WebFlux 中的相同

控制器可以添加对 HTTP 缓存的显式支持。我们建议这样做，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 Headers 进行比较。控制器可以将 `ETag` Headers 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
        .body(book);
}
```

如果与条件请求 Headers 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` Headers 将添加到响应中。

您还可以在控制器中针对条件请求 Headers 进行检查，如以下示例所示：

```
@RequestMapping
public String myHandleMethod(WebRequest webRequest, Model model) {
    long eTag = ... (1)

    if (request.checkNotModified(eTag)) {
        return null; (2)
    }

    model.addAttribute(...); (3)
    return "myViewName";
}
```

- (1) 特定于应用程序的计算。

- (2) 响应已设置为 304(NOT\_MODIFIED)-没有进一步处理。
- (3) continue 进行请求处理。

可以使用三种变体来检查针对 `eTag` 值和 `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`，您可以改为将响应设置为 409(PRECONDITION\_FAILED)，以防止并发修改。

### 1.8.3. 静态资源

[与 Spring WebFlux 中的相同](#)

您应该为静态资源提供 `Cache-Control` 和条件响应 Headers，以实现最佳性能。请参阅有关配置 [Static Resources](#) 的部分。

### 1.8.4. ETag 过滤器

您可以使用 `ShallowEtagHeaderFilter` 添加根据响应内容计算的“浅” `eTag` 值，从而节省带宽，但不节省 CPU 时间。参见[Shallow ETag](#)。

## 1.9. 查看技术

[与 Spring WebFlux 中的相同](#)

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

[与 Spring WebFlux 中的相同](#)

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 Management。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

[与 Spring WebFlux 中的相同](#)

[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 Configuration

[与 Spring WebFlux 中的相同](#)

以下示例显示了如何将 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-configure>
    <mvc:template-loader-path location="/WEB-INF/freemarker"/>
</mvc:freemarker-configure>
```

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一示例所示的 `FreeMarkerConfigurer` 指定的目录中。给定前面的配置，如果您的控制器返回视图名称 `welcome`，则解析器将查找 `/WEB-INF/freemarker/welcome.ftl` 模板。

## FreeMarker Configuration

### 与 Spring WebFlux 中的相同

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 `Configuration` 对象(由 `SpringManagement`)

- `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。

## Form Handling

Spring 提供了一个供 JSP 使用的标签库，其中包含 `<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 Importing 元素本身的附加便利宏。

### 绑定宏

两种语言都在 `spring-webmvc.jar` 文件中维护了一组标准宏，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从模板内直接调用的宏。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。

### Simple Binding

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 Importing 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>
        ...
        <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')，后跟句点和命令对象上的字段名称希望绑定。您还可以使用嵌套字段，例如 `command.address.street`。`bind` 宏假定 `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，并且应尽可能使用这些宏。下一节将对它们进行说明。

## Input macros

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 Importing 字段，并且您可以将它们与简单的 HTML 混合或匹配，或者直接调用我们前面强调的 `spring` 绑定宏。

下表可用的宏显示了每个 FTL 定义和参数列表：

表 6.宏定义表

macro	FTL definition
<code>message</code> (根据 <code>code</code> 参数从资源包中输出字符串)	<code>&lt;@spring.message code/&gt;</code>
<code>messageText</code> (根据 <code>code</code> 参数从资源包中输出一个字符串，回退到默认参数的值)	<code>\ &lt;@spring.messageText code, text/&gt;</code>
<code>url</code> (使用应用程序的上下文根作为相对 URL 的前缀)	<code>&lt;@spring.url relativeUrl/&gt;</code>

macro	FTL definition
<p><code>formInput</code> (用于收集用户 Importing 的标准 Importing 字段)</p>	<pre>\&lt;@spring.formInput path, attributes, fieldType/&gt;</pre>
<p><code>formHiddenInput</code> (用于提交非用户 Importing 的隐藏 Importing 字段)</p>	<pre>\&lt;@spring.formHiddenInput path, attributes/&gt;</pre>
<p><code>formPasswordInput</code> (用于收集密码的标准 Importing 字段.请注意，此类型的字段中不会填充任何值.)</p>	<pre>\&lt;@spring.formPasswordInput path, attributes/&gt;</pre>
<p><code>formTextarea</code> (大文本字段，用于收集长而自由格式的文本 Importing)</p>	<pre>\&lt;@spring.formTextarea path, attributes/&gt;</pre>
<p><code>formSingleSelect</code> (选项的下拉框，用于选择一个必需的值)</p>	<pre>\&lt;@spring.formSingleSelect path, options, attributes/&gt;</pre>
<p><code>formMultiSelect</code> (选项列表框，允许用户选择 0 个或多个值)</p>	<pre>\&lt;@spring.formMultiSelect path, options, attributes/&gt;</pre>
<p><code>formRadioButtons</code> (一组单选按钮，可从可用选项中进行单个选择)</p>	<pre>\&lt;@spring.formRadioButtons path, options separator, attributes/&gt;</pre>
<p><code>formCheckboxes</code> (一组允许选择 0 个或多个值的复选框)</p>	<pre>\&lt;@spring.formCheckboxes path, options, separator, attributes/&gt;</pre>

macro	FTL definition
<code>formCheckbox</code> (单个复选框)	\ <@spring.formCheckbox path, attributes/>
<code>showErrors</code> (简化显示绑定字段的验证错误)	\ <@spring.showErrors separator, classOrStyle/>

- 在 FTL(FreeMarker) 中，实际上并不需要 `formHiddenInput` 和 `formPasswordInput`，因为您可以使用普通的 `formInput` 宏，将 `hidden` 或 `password` 指定为 `fieldType` 参数的值。

以上任何宏的参数都具有一致的含义：

- `path`：要绑定的字段的名称(即“`command.name`”)
- `options`：可以在 Importing 字段中选择的所有可用值中的 `Map`。Map 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 Map 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 Map 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 Map，可以将 `SortedMap` (例如 `TreeMap`) 与合适的 `Comparator` 一起使用，对于要按插入 Sequences 返回值的任意 Map，请使用 `LinkedHashMap` 或 `commons-collections` 的 `LinkedMap`。
- `separator`：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 `<br>` )。
- `attributes`：HTML 标记本身中将包含一个附加的任意标记字符串或文本。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)，则错误将被包装在 `<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)。如果两种语言之间存在用法差异，请在 `Comments` 中进行说明。

## Input Fields

`formInput` 宏带有 `path` 参数(`command.name`)和一个附加的 `attributes` 参数(在接下来的示例中为空)。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。绑定将保持有效，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` 宏带有分隔符参数(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 `showErrors` 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，并在提交表单后在该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 `textarea` 的 `rows` 和 `cols` 属性。

## Selection Fields

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 Importing:

- `formSingleSelect`
- `formMultiSelect`
- `formRadioButtons`
- `formCheckboxes`

四个宏中的每个宏都接受一个 `Map` 选项，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“伦敦”，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 Lists 显示了示例：

```
...
Town:  
<@spring.formRadioButtons "command.address.town", cityMap, "" /><br><br>
```

上面的 Lists 呈现了一行单选按钮，其中一个用于 `cityMap` 中的每个值，并使用一个分隔符 `..`。没有提供其他属性(缺少该宏的最后一个参数)。`cityMap` 对 Map 中的每个键值对使用相同的 `String`。Map 键是表单实际作为 POSTed 请求参数提交的键。Map 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)，则可以使用适当的键创建代码 Map，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，该代码会生成输出，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

前面描述的表单宏的默认用法导致 HTML 元素符合 HTML 4.01，并且使用 `web.xml` 文件中定义的 HTML 转义的默认值，如 Spring 的绑定支持所使用。要使元素符合 XHTML 或覆盖默认的 HTML 转义值，可以在模板(或模型中对模板可见的位置)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值 `true`，如以下示例所示：

```
<!-- for FreeMarker -->
<#assign xhtmlCompliant = true>
```

处理完此指令后，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 -->
```

### 1.9.3. Groovy 标记

[Groovy 标记模板引擎](#)主要用于生成类似 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。

#### Note

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. 脚本视图

[与 Spring WebFlux 中的相同](#)

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
<a href="#">Handlebars</a>	<a href="#">Nashorn</a>
<a href="#">Mustache</a>	<a href="#">Nashorn</a>
<a href="#">React</a>	<a href="#">Nashorn</a>
<a href="#">EJS</a>	<a href="#">Nashorn</a>
<a href="#">ERB</a>	<a href="#">JRuby</a>
<a href="#">String templates</a>	<a href="#">Jython</a>

Scripting Library	Scripting Engine
<a href="#">Kotlin 脚本模板</a>	<a href="#">Kotlin</a>

## Tip

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

### [与 Spring WebFlux 中的相同](#)

您需要在 Classpath 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。
- 应该添加 [JRuby](#) 作为 Ruby 支持的依赖项。
- 应该添加 [Jython](#) 作为对 Python 支持的依赖。
- 为了支持 Kotlin 脚本，应添加 `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 [this example](#)。

您需要具有脚本模板库。一种针对 Javascript 的方法是通过 [WebJars](#)。

## Script Templates

### [与 Spring WebFlux 中的相同](#)

您可以声明一个 `ScriptTemplateConfigurer` bean，以指定要使用的脚本引擎，要加载的脚本文档，要调用的函数以渲染模板等等。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>

<mvc:script-template-configure engine-name="nashorn" render-object="Mustache" render-f
    <mvc:script location="mustache.js"/>
</mvc:script-template-configure>

```

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`：用于访问应用程序上下文，语言环境，模板加载器和 URL 的 [RenderingContext](#)(自 5.0 开始)

`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，而 [polyfill](#) 则需要模拟一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;
    }
}
```

### iNote

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，需要将 `sharedEngine` 属性设置为 `false`，例如 Nashorn 上运行的 Handlebars 或 React。在这种情况下，由于 [this bug](#)，因此需要 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 Management 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);
}
```

查看 Spring Framework 单元测试[Java](#)和[resources](#)，以获取更多配置示例。

## 1.9.5. JSP 和 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 `InternalResourceViewResolver` 或

`ResourceBundleViewResolver`  bean。

`ResourceBundleViewResolver`  依靠属性文件来定义 Map 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 uses (views.properties in WEB-INF/classes):
welcome.(class)=org.springframework.web.servlet.view.JstlView
welcome.url=/WEB-INF/jsp/welcome.jsp

productList.(class)=org.springframework.web.servlet.view.JstlView
productList.url=/WEB-INF/jsp/productlist.jsp
```

`InternalResourceBundleViewResolver` 也可用于 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 Client 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 Java 标准标记库时，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 提供了将请求参数与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。

`spring.tld` 标签库 Descriptors(TLD) 包含在 `spring-webmvc.jar` 中。有关单个标签的全面参考，请浏览 [API reference](#) 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记，用于处理表单元素。每个标签都支持与其对应的 HTML 标签对等物的属性集，从而使标签熟悉且使用直观。标记生成的 HTML 符合 HTML 4.01/XHTML 1.0.

与其他表单/Importing 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，并查看有关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 Comments。

## Configuration

表单标签库 Binding 在 `spring-webmvc.jar` 中。库 Descriptors 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" %>
```

其中 `form` 是要用于此库中标签的标签名称前缀。

## 表单标签

此标记呈现 HTML 'form' 元素，并向内部标记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 `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。 `continue` 阅读以查看有关如何将内部标签与 `form` 标签一起使用的更复杂的示例。

下面的 `Lists` 显示了生成的 HTML，它看起来像标准格式：

```

<form method="POST">


|                                             |                                                     |
|---------------------------------------------|-----------------------------------------------------|
| First Name:                                 | <input name="firstName" type="text" value="Harry"/> |
| Last Name:                                  | <input name="lastName" type="text" value="Potter"/> |
| <input type="submit" value="Save Changes"/> |                                                     |


</form>

```

前面的 JSP 假定表单支持对象的变量名称为 `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">


|                                             |                                |
|---------------------------------------------|--------------------------------|
| First Name:                                 | <form:input path="firstName"/> |
| Last Name:                                  | <form:input path="lastName"/>  |
| <input type="submit" value="Save Changes"/> |                                |


</form:form>

```

## Importing 标签

默认情况下，此标记呈现具有绑定值和 `type='text'` 的 HTML `input` 元素。有关此标记的示例，请参见[表单标签](#)。您还可以使用特定于 HTML5 的类型，例如 `email`，`tel`，`date` 等。

## 复选框标签

此标签呈现 `type` 设置为 `checkbox` 的 HTML `input` 标签。

假设我们的 `User` 具有首选项，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() {  
        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="DADA"/>
    </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>

```

`checkbox` 标记有三种方法，应该可以满足您所有复选框的需求。

- 方法一：当绑定值为 `java.lang.Boolean` 时，如果绑定值为 `true`，则 `input(checkbox)` 被标记为 `checked`。 `value` 属性对应于 `setValue(Object)` `value` 属性的解析值。
- 方法二：当界限值是 `array` 或 `java.util.Collection` 类型时，如果界限 `Collection` 中存在已配置的 `setValue(Object)` 值，则 `input(checkbox)` 被标记为 `checked`。
- 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(checkbox)` 标记为 `checked`。

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="DADA"/>
        <input type="hidden" value="1" name="_preferences.interests"/>
    </td>
</tr>

```

```
</td>
</tr>
```

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` 标签遵循现有的 Spring 约定，其中为每个复选框包括一个带有下划线(`_`)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，并且我希望与表单数据绑定的对象能够反映复选框的状态，无论如何。”

## 复选框标签

此标签呈现 `type` 设置为 `checkbox` 的多个 HTML `input` 标签。

本节以上一个 `checkbox` 标签节的示例为基础。有时，您宁愿不必在 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 `checkboxes` 标记的目的。您可以传入 `items` 属性中包含可用选项的 `Array`，`List` 或 `Map`。通常，`bound` 属性是一个集合，因此它可以容纳用户选择的多个值。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 `List`，可以用作模型属性，其中包含要从中选择的值的字符串。如果使用 `Map`，则将 `Map` 条目键用作值，并将 `Map` 条目的值用作要显示的标签。您也可以使用自定义对象，在其中可以使用 `itemValue` 提供值的属性名称，并通过 `itemLabel` 提供标签。

## 单选按钮标记

此标记呈现 `type` 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:radio button path="sex" value="M"/>
        Female: <form:radio button path="sex" value="F"/>
    </td>
</tr>
```

## 单选按钮标记

此标记呈现 `type` 设置为 `radio` 的多个 HTML `input` 元素。

与 [checkboxes tag](#) 一样，您可能希望将可用选项作为运行时变量传递。为此，您可以使用 `radiobuttons` 标签。您传入 `Array`，`List` 或 `Map`，它们包含 `items` 属性中的可用选项。如果使用 `Map`，则将 `Map` 条目键用作值，并将 `Map` 条目的值用作要显示的标签。您还可以使用一个自定义对象，在其中您可以使用 `itemValue` 提供值的属性名称，并通过 `itemLabel` 提供标签，如以下示例所示：

```
<tr>
    <td>Sex:</td>
    <td><form:radio buttons path="sex" items="${sexOptions}"/></td>
</tr>
```

## 密码标签

此标记呈现具有设置为 `password` 且具有绑定值的 HTML `input` 标记。

```
<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。如果您确实希望显示密码值，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>
```

## 选择标签

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的 `option` 和 `options` 标签。

假设 `User` 拥有技能列表。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 `User's` 技能是草药学，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>
```

## 选项标签

此标记呈现 HTML `option` 元素。它基于绑定值设置 `selected`。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 `User's` 房屋位于格兰芬多，则“房屋”行的 HTML 源代码如下：

```

<tr>
    <td>House:</td>
    <td>
        <select name="house">
            <option value="Gryffindor" selected="selected">Gryffindor</option> (1)
            <option value="Hufflepuff">Hufflepuff</option>
            <option value="Ravenclaw">Ravenclaw</option>
            <option value="Slytherin">Slytherin</option>
        </select>
    </td>
</tr>

```

- (1) 请注意添加了 `selected` 属性。

## 选项标签

此标记呈现 HTML `option` 元素的列表。它根据绑定值设置 `selected` 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 `User` 居住在英国，则“国家/locale”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> (1)
<option value="US">United States</option>
</select>
</td>
</tr>
```

- (1) 请注意添加了 `selected` 属性。

如前面的示例所示，`option` 标记与 `options` 标记的组合用法会生成相同的标准 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` 属性通常由项对象的集合或数组填充。`itemValue` 和 `itemLabel` 引用这些项目对象的 `bean` 属性(如果已指定)。否则，项目对象本身将变成字符串。或者，您可以指定项目的 `Map`，在这种情况下，`Map` 键将解释为选项值，并且 `Map` 值对应于选项标签。如果也恰好指定了 `itemValue` 或 `itemLabel` (或两者)，则 `item value` 属性应用于 `Map` 键，`item label` 属性应用于 `Map` 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>
```

## 隐藏的标签

该标签呈现 HTML `input` 标签，其中 `type` 设置为 `hidden` 并具有绑定值。要提交未绑定的隐藏值，请使用 `type` 设置为 `hidden` 的 HTML `input` 标记。以下 HTML 显示了其典型输出：

```
<form:hidden path="house" />
```

如果我们选择将 `house` 值提交为隐藏值，则 HTML 如下所示：

```
<input name="house" type="hidden" value="Gryffindor"/>
```

## 错误代码

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  
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "lastName", "required", "Field  
    }  
}
```

`form.jsp` 可能如下：

```
<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>
            <%-- 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` 字段关联的所有错误。
- 如果省略 `path`，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>

```

```
        </tr>
    </table>
</form:form>
```

HTML 将如下所示：

```
<form method="POST">
    <span name=".errors" class="errorBox">Field is required.
Field is required.</span>
    <table>
        <tr>
            <td>First Name:</td>
            <td><input name="firstName" type="text" value="" /></td>
            <td><span name="firstName.errors">Field is required.</span></td>
        </tr>

        <tr>
            <td>Last Name:</td>
            <td><input name="lastName" type="text" value="" /></td>
            <td><span name="lastName.errors">Field is required.</span></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
</form>
```

`spring-form.tld` 标签库 Descriptors(TLD)包含在 `spring-webmvc.jar` 中。有关单个标签的全面参考，请浏览[API reference](#)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 Importing 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 `method` 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”samples：

```
<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 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 Tags

Spring 表单标签库允许 Importing 动态属性，这意味着您可以 Importing 任何 HTML5 特定的属性。

`input` 标签形式支持 Importing `text` 以外的 `type` 属性。这旨在允许呈现新的 HTML5 特定 Importing 类型，例如 `email`，`date`，`range` 等。请注意，由于 `text` 是默认类型，因此不需要 Importing `type='text'`。

## 1.9.6. Tiles

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。本节将广泛地描述如何执行此操作。

### iNote

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。

### Dependencies

为了能够使用 Tiles，您必须在项目中添加对 Tiles 3.0.1 或更高版本以及[它的传递依赖](#)的依赖。

### Configuration

为了能够使用 Tiles，您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<http://tiles.apache.org>)。在 Spring，可以使用 `TilesConfigurer` 完成此操作。以下

示例 `ApplicationContext` 配置显示了如何执行此操作：

```
<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，与 Spring 所使用的任何其他视图技术一样，您必须具有 `ViewResolver`。您可以使用 `UrlBasedViewResolver` 和

`ResourceBundleViewResolver` 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>
```

使用前面的配置，`tiles_fr_FR.xml` 用于具有 `fr_FR` 语言环境的请求，默认情况下使用 `tiles.xml`。

### iNote

由于下划线用于指示语言环境，因此我们建议不要在 Tiles 定义的文件名中使用下划线。

## UrlBasedViewResolver

`UrlBasedViewResolver` 为要解析的每个视图实例化给定的 `viewClass`。以下 bean 定义了

`UrlBasedViewResolver`：

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.UrlBasedViewResolver">
    <property name="viewClass" value="org.springframework.web.servlet.view.tiles3.TilesView"/>
</bean>
```

## ResourceBundleViewResolver

`ResourceBundleViewResolver` 必须提供一个属性文件，其中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)：

```
<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，您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。

您可以基于指定的准备器类，通过应用 Spring 的容器回调以及应用配置的 Spring `BeanPostProcessor`，来指定 `SimpleSpringPreparerFactory` 自动连接 `ViewPreparer` 实例。如果已激活 Spring 的上下文范围内的 `Comments` 配置，则会自动检测并应用 `ViewPreparer` 类中的 `Comments`。请注意，这与默认的 `PreparerFactory` 一样，期望 Tiles 定义文件中的准备程序类。

您可以指定 `SpringBeanPreparerFactory` 来对指定的准备器名称(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)。下面的示例显示如何在 `TilesConfigurer` bean 上定义一个 `SpringBeanPreparerFactory` 属性集：

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

```

        <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.SpringBeanPreparerFactor

</bean>

```

## 1.9.7. RSS 和 Atom

`AbstractAtomFeedView` 和 `AbstractRssFeedView` 都继承自 `AbstractFeedView` Base Class， 并分别用于提供 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，并可以选择覆盖 `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
    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
}

}

```

下例显示了类似的要求，适用于实现 `AbstractRssFeedView`：

```

public class SampleContentAtomView extends AbstractRssFeedView {

    @Override
    protected void buildFeedMetadata(Map<String, Object> model,
                                     Channel feed, HttpServletRequest request) {
        // implementation omitted
    }
}

```

```
    @Override  
    protected List<Item> buildFeedItems(Map<String, Object> model,  
                                         HttpServletRequest request, HttpServletResponse response) throws Exception  
        // implementation omitted  
    }  
}
```

`buildFeedItems()` 和 `buildFeedEntries()` 方法传递 HTTP 请求，以防您需要访问区域设置。

仅针对 Cookie 或其他 `HTTPHeaders` 的设置传入 HTTP 响应。方法返回后，该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 [entry](#)。

## 1.9.8. PDF 和 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。

### 文档视图简介

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 Content Type 进行流传输，以(有希望)使 Client 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 Classpath 中，要生成 PDF，需要添加(最好是)OpenPDF 库。

#### iNote

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 1.0.5)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  
        HttpServletRequest request, HttpServletResponse response) throws Exception  
  
        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)或作为处理程序方法的 `view` 实例返回此类视图。

## Excel Views

从 Spring Framework 4.2 开始，

`org.springframework.web.servlet.view.document.AbstractXlsView` 作为 Excel 视图的

Base Class 提供。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()` 作为中心模板方法，控制器

能够从外部定义(按名称)或从 `handler` 方法作为 `view` 实例返回这种视图。

## 1.9.9. Jackson

[与 Spring WebFlux 中的相同](#)

Spring 提供了对 Jackson JSON 库的支持。

### 基于 Jackson 的 JSON 视图

[与 Spring WebFlux 中的相同](#)

`MappingJackson2JsonView` 使用 Jackson 库的 `ObjectMapper` 将响应内容呈现为 JSON。默认情

况下，模型 Map 的所有内容(特定于框架的类除外)均编码为 JSON。对于需要过滤 Map 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。您还可以使用 `extractValueFromSingleKeyModel` 属性来直接提取和序列化单键模型中的值，而不是将其作为模型属性的 Map。

您可以根据需要使用 Jackson 提供的 Comments 来自定义 JSONMap。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

[与 Spring WebFlux 中的相同](#)

`MappingJackson2XmlView` 使用 [Jackson XML 扩展的 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，那么它将自动序列化。

您可以根据需要使用 JAXB 或 Jackson 提供的 Comments 自定义 XMLMap。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 的情况，您需要为特定类型提供序列化器和反序列化器。

## 1.9.10. XML 编组

`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 属性来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` 软件包中功能的更多信息，请参见 [使用 O/XMap 器编组 XML](#)。

## 1.9.11. XSLT 视图

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

自然地处理 XML，或者如果您的模型可以轻松转换为 XML，那么 XSLT 可以作为视图技术的不错选择。下一节说明如何将 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 Map 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[Annotated Controllers](#)。XSLT 控制器将单词列表转换为准备转换的简单 XML 文档。

## Beans

配置是简单的 Spring Web 应用程序的标准配置：MVC 配置必须定义 `XsltViewResolver` bean 和常规 `MVCComments` 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词生成逻辑的控制器。

控制器逻辑封装在 `@Controller` 类中，其中 `handler` 方法的定义如下：

```
@Controller
public class XsltController {

    @RequestMapping("/")
    public String home(Model model) throws Exception {
        Document document = DocumentBuilderFactory.newInstance().newDocumentBuilder().n...
```

```

        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 ModelMap 中。请注意，您还可以将 XML 文件作为 `Resource` 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 ManagementDOMification 流程时是一种危险。

## Transformation

最后，`XsltViewResolver` 解析“主” 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` 目录中的 `.war` 文件中，并以 `.xslt` 文件 `extensions` 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" />

    <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>
```

## 1.10. MVC 配置

[与 Spring WebFlux 中的相同](#)

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对其进行自定义的配置 API。

有关配置 API 中没有的更高级的自定义设置，请参见[高级 Java 配置](#)和[高级 XML 配置](#)。

您不需要了解由 MVC Java 配置和 MVC 名称空间创建的基础 bean。如果您想了解更多信息，请参阅[特殊 bean 类](#)和[Web MVC 配置](#)。

### 1.10.1. 启用 MVC 配置

[与 Spring WebFlux 中的相同](#)

在 Java 配置中，可以使用 `@EnableWebMvc` 注解来启用 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  
           http://www.springframework.org/schema/beans/spring-beans.xsd  
           http://www.springframework.org/schema/mvc  
           http://www.springframework.org/schema/mvc/spring-mvc.xsd">  
  
    <mvc:annotation-driven/>  
  
</beans>
```

前面的示例注册了许多 Spring MVC [infrastructure beans](#)，并适应了 Classpath 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

[与 Spring WebFlux 中的相同](#)

在 Java 配置中，您可以实现 `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. 类型转换

[与 Spring WebFlux 中的相同](#)

默认情况下，安装了 `Number` 和 `Date` 类型的格式化程序，包括对 `@NumberFormat` 和 `@DateTimeFormat` 注解的支持。如果 Classpath 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
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/mvc
           http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <mvc:annotation-driven conversion-service="conversionService"/>

    <bean id="conversionService"
          class="org.springframework.format.support.FormattingConversionServiceFactoryBean">
        <property name="converters">
            <set>
                <bean class="org.example.MyConverter" />
            </set>
        </property>
        <property name="formatters">
            <set>
                <bean class="org.example.MyFormatter" />
                <bean class="org.example.MyAnnotationFormatterFactory" />
            </set>
        </property>
        <property name="formatterRegistrars">
            <set>
                <bean class="org.example.MyFormatterRegistrar" />
            </set>
        </property>
    </bean>
</beans>
```

## iNote

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

[与 Spring WebFlux 中的相同](#)

默认情况下, 如果[Bean Validation](#)存在于 Classpath 中(例如, Hibernate Validator), 则  
`LocalValidatorFactoryBean` 被注册为全局[Validator](#), 以便与控制器方法参数上的 `@Valid` 和  
`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
@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
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/mvc
           http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <mvc:annotation-driven validator="globalValidator" />

</beans>
```

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());
    }

}
```

### Tip

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("/admin/**");
        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>
    <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 Type

[与 Spring WebFlux 中的相同](#)

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` Headers, URL 路径扩展, 查询参数等)。

默认情况下, 将首先检查 URL 路径 extensions, 即将 `json`, `xml`, `rss` 和 `atom` 注册为已知 extensions(取决于 Classpath 依赖项)。 `Accept` Headers 被第二次检查。

考虑将这些默认值仅更改为 `Accept` Headers, 并且, 如果必须使用基于 URL 的 Content Type 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[Suffix Match](#)和[后缀匹配和 RFD](#)。

在 Java 配置中, 您可以自定义请求的 Content Type 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.ContentNegotiationManager">
    <property name="mediaTypes">
        <value>
            json=application/json
            xml=application/xml
        </value>
    </property>
</bean>
```

## 1.10.7. 讯息转换器

## 与 Spring WebFlux 中的相同

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)来自定义 Java 配置中的 `HttpMessageConverter`。

以下示例使用自定义的 `ObjectMapper` 而非默认的添加了 XML 和 Jackson JSON 转换器：

```
@Configuration
@EnableWebMvc
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.createXmlMapp
    }
}
```

在前面的示例中，使用[Jackson2ObjectMapperBuilder](#)来为

`MappingJackson2HttpMessageConverter` 和 `MappingJackson2XmlHttpMessageConverter` 创建启用缩进的通用配置，自定义的日期格式和[jackson-module-parameter-names](#)的注册，这增加了对访问参数名称的支持(Java 8 中添加的功能)。

此构建器自定义 Jackson 的默认属性，如下所示：

- [DeserializationFeature.FAIL\\_ON\\_UNKNOWN\\_PROPERTIES](#)已禁用。
- [MapperFeature.DEFAULT\\_VIEW\\_INCLUSION](#)已禁用。

如果在 Classpath 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`。

### iNote

启用具有 Jackson XML 支持的缩进除了 [jackson-dataformat-xml](#) 之外还需要 [woodstox-core-asl](#) 个依赖。

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">
    <property name="indentOutput" value="true" />
    <property name="simpleDateFormat" value="yyyy-MM-dd" />
    <property name="modulesToInstall" value="com.fasterxml.jackson.module.paramnames.ParameterNamesModule" />
</bean>

<bean id="xmlMapper" parent="objectMapper" p:createXmlMapper="true" />
```

## 1.10.8. 查看控制器

这是定义被调用时立即转发到视图的 `ParameterizableViewController` 的快捷方式。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。

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，实现了与上述示例相同的操作，但使用 XML

:

```
<mvc:view-controller path="/" view-name="home"/>
```

## 1.10.9. 查看解析器

[与 Spring WebFlux 中的相同](#)

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 渲染的默认 `view` 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-configure>
    <mvc:template-loader-path location="/freemarker" />
</mvc:freemarker-configure>
```

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. 静态资源

[与 Spring WebFlux 中的相同](#)

此选项提供了一种方便的方式来从基于 `Resource` 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 Classpath 下相对于 `/public` 查找和提供静态资源。这些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。还将评估 `Last-Modified`

Headers，并返回 **304** 状态代码(如果存在)。

以下 Lists 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" />
```

另请参见对静态资源的 [HTTP 缓存支持](#)。

资源处理器还支持[ResourceResolver](#)实现和[ResourceTransformer](#)实现的链，您可以使用它们来创建工具链以使用优化的资源。

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，其中有一些值得注意的 exception，例如与模块加载器一起使用的 JavaScript 资源。

以下示例显示了如何在 Java 配置中使用 [VersionResourceResolver](#)：

```
@Configuration  
@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>
        <mvc:resource-cache/>
        <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 `ResourceUrlProvider` bean，以便可以将其注入其他对象。对于 Thymeleaf, JSP, FreeMarker 以及其他依赖于 `HttpServletResponse#encodeURL` 的 URL 标记，您也可以使用 `ResourceUrlEncodingFilter` 使重写透明。

请注意，同时使用 `EncodedResourceResolver` (例如，用于提供压缩或 brotli 编码的资源) 和 `VersionedResourceResolver` 时，必须按此 Sequences 注册它们。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](#) 也受 `WebJarsResourceResolver` 支持，并且当 `org.webjars:webjars-locator` 存在于 Classpath 中时会自动注册。解析程序可以重写 URL 以包括 jar 的版本，还可以与不带版本的传入 URL 进行匹配，例如 `/jquery/jquery.min.js` 到 `/jquery/1.2.0/jquery.min.js`。

## 1.10.11. 默认 Servlet

Spring MVC 允许将 `DispatcherServlet` Map 到 `/` (从而覆盖了容器默认 Servlet 的 Map)，同时仍允许容器默认 Servlet 处理静态资源请求。它将 `DefaultServletHttpRequestHandler` 配置为 `/**` 的 URLMap，并且相对于其他 URLMap 具有最低的优先级。

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 `Sequences` 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` (即 `Integer.MAX_VALUE`) 的值。

以下示例显示了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/>
```

覆盖 / `ServletMap` 的警告是， 默认 `Servlet` 的 `RequestDispatcher` 必须通过名称而不是通过路径来检索。 `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`。如果已使用其他名称自定义配置了默认 `Servlet`，或者在默认 `Servlet` 名称未知的情况下使用了不同的 `Servlet` 容器，则必须显式提供默认 `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. 路径匹配

### 与 Spring WebFlux 中的相同

您可以自定义与路径匹配和 URL 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：

```
<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. 高级 Java 配置

与 Spring WebFlux 中的相同

@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 WebMvcConfigurer 实现自定义该配置。

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 Base Class 中的 bean 声明，并且在 Classpath 上仍然可以具有任意数量的其他 WebMvcConfigurer 实现。

### 1.10.14. 高级 XML 配置

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 生命周期钩子，如以下示例所示：

```
@Component
public class MyPostProcessor implements BeanPostProcessor {

    public Object postProcessBeforeInitialization(Object bean, String name) throws BeansException {
        // ...
    }
}
```

注意，您需要将 `MyPostProcessor` 声明为 bean，或者以 XML 形式显式声明，或者通过 `<component-scan/>` 声明来检测它。

## 1.11. HTTP/2

[与 Spring WebFlux 中的相同](#)

需要 Servlet 4 容器支持 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 Client 端，并且受[method argument](#) 到 `@RequestMapping` 方法支持。

## 2. RESTClient 端

本节描述了 Client 端对 REST 端点的访问选项。

### 2.1. RestTemplate

`RestTemplate` 是执行 HTTP 请求的同步 Client 端。它是原始的 Spring RESTClient 端，并在基础 `HttpClient` 端库上公开了简单的模板方法 API。

#### iNote

从 5.0 开始，无阻塞，Reactive `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。

有关详情，请参见[REST Endpoints](#)。

## 2.2. WebClient

`WebClient` 是用于执行 HTTP 请求的非阻塞，响应式 Client 端。它是在 5.0 中引入的，并提供了 `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：

- Non-blocking I/O.
- Reactive 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。
- 从服务器向上流或向下流。

有关更多详细信息，请参见[WebClient](#)。

## 3. Testing

---

[与 Spring WebFlux 相同](#)

本节总结了 `spring-test` 中可用于 Spring MVC 应用程序的选项。

- Servlet API 模拟：用于单元测试控制器，过滤器和其他 Web 组件的 Servlet APIContracts 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

Comments)测试带 Comments 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 测试](#)。

- Client 端 REST: `spring-test` 提供了一个 `MockRestServiceServer`，您可以将其用作模拟服务器来测试内部使用 `RestTemplate` 的 Client 端代码。有关更多详细信息，请参见[Client 端 REST 测试](#)。
- `WebTestClient`：用于测试 WebFlux 应用程序，但也可以用于通过 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 Client 端，非常适合测试异步和流传输方案。

## 4. WebSockets

### [与 Spring WebFlux 中的相同](#)

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。

### 4.1. WebSocket 介绍

WebSocket 协议[RFC 6455](#)提供了一种标准化方法，可以通过单个 TCP 连接在 Client 端和服务器之间构建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` Headers 进行升级，或者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 (1)
Connection: Upgrade (2)
Sec-WebSocket-Key: Uc919TMkWGbhFD2qnFH1tg==
Sec-WebSocket-Protocol: v10.stomp, v11.stomp
Sec-WebSocket-Version: 13
Origin: http://localhost:8080
```

- (1) `Upgrade` Headers。

- (2) 使用 `Upgrade` 连接。

具有 WebSocket 支持的服务器代替通常的 200 状态代码，返回类似于以下内容的输出：

```
HTTP/1.1 101 Switching Protocols (1)
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: 1qVdfYHU9hP014JYYNxF623Gzn0=
Sec-WebSocket-Protocol: v10.stomp
```

- (1) 协议切换

成功握手后，HTTP 升级请求的基础 TCP 套接字将保持打开状态，Client 端和服务器均可 continue 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 与 WebSocket

尽管 WebSocket 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，Client 端访问那些 URL，即请求-响应样式。服务器根据 HTTP URL，方法和 Headers 将请求路由到适当的处理程序。

相比之下，在 WebSockets 中，初始连接通常只有一个 URL。随后，所有应用程序消息在同一 TCP 连接上流动。这指向了完全不同的异步，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 Client 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。

WebSocketClient 端和服务器可以通过 HTTP 握手请求上的 `Sec-WebSocket-Protocol` Headers

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. 何时使用 WebSockets

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 流或长时间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，但是每隔几分钟这样做是完全可以的。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` Headers，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。

## 4.2. WebSocket API

[与 Spring WebFlux 中的相同](#)

Spring 框架提供了一个 WebSocket API，可用于编写处理 WebSocket 消息的 Client 端和服务器端应用程序。

### 4.2.1. WebSocketHandler

[与 Spring WebFlux 中的相同](#)

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 Map 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
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/websocket
           http://www.springframework.org/schema/websocket/spring-websocket.xsd">

    <websocket:handlers>
        <websocket:mapping path="/myHandler" handler="myHandler"/>
    </websocket:handlers>

    <bean id="myHandler" class="org.springframework.samples.MyHandler"/>

</beans>

```

以下示例供 Spring MVC 应用程序使用，并且应包含在[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。

## 4.2.2. WebSocket 握手

### [与 Spring WebFlux 中的相同](#)

定制初始 HTTP WebSocket 握手请求的最简单方法是通过 [HandshakeInterceptor](#)，它公开了在“之前”和“之后”握手的方法。您可以使用此类拦截器来排除握手或使任何属性对 [WebSocketSession](#) 可用。下面的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
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/websocket
           http://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.HttpSessionHand...
            </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执行 WebSocket 握手步骤的 `DefaultHandshakeHandler`，包括验证 Client 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，请参见[Deployment](#))。Java 配置和 XML 名称空间均可配置自定义 `HandshakeHandler`。

### Tip

Spring 提供了一个 `WebSocketHandlerDecorator` Base Class，您可以使用该 Base Class 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` 的 WebSocket 会话，这表明服务器错误。

### 4.2.3. Deployment

易于将 Spring WebSocket API 集成到 Spring MVC 应用程序中，其中 `DispatcherServlet` 服务于 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 Classpath 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 的一个重大限制，即使在 JSR-356 运行时中运行，Spring 的 WebSocket 支持使用特定于服务器的 `RequestUpgradeStrategy` 实现解决。Tomcat

, Jetty, GlassFish, WebLogic, WebSphere 和 Undertow(和 WildFly)目前存在此类策略。

### iNote

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](#)处进行跟踪。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。

另一个要考虑的因素是, 希望支持 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。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 />` 元素选择性地启用或禁用 Web 片段(和 SCI 扫描), 如以下示例所示显示:

```
<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
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  version="3.0">

  <absolute-ordering/>

</web-app>
```

然后, 您可以按名称有选择地启用 Web 片段, 例如 Spring 自己的

`SpringServletContainerInitializer`, 该片段提供对 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
    http://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. 服务器配置

[与 Spring WebFlux 中的相同](#)

每个基础的 WebSocket 引擎都公开控制运行时 Feature 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
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/websocket
           http://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>
```

### Note

对于 Client 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`，然后通过 WebSocket

Java 配置将其插入 Spring 的 `DefaultHandshakeHandler`。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)));
    }

}
```

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
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/websocket
           http://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. 允许的来源

[与 Spring WebFlux 中的相同](#)

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 Client 端。没有任何措施可以阻止其他类型的 Client 端修改 `Origin` Headers 值(有关更多详细信息，请参见[RFC 6454：Web 起源概念](#))。

三种可能的行为是：

- 仅允许同源请求(默认)：在此模式下，启用 SockJS 时，Iframe HTTP 响应 Headers `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的来源。因此，启用此模式时，不支持 IE6 和 IE7.
- 允许指定来源列表：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 至 IE9.
- 允许所有来源：要启用此模式，您应提供 `*` 作为允许的来源值。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 {  
  
    @Override  
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {  
        registry.addHandler(myHandler(), "/myHandler").setAllowedOrigins("http://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  
           http://www.springframework.org/schema/beans/spring-beans.xsd  
           http://www.springframework.org/schema/websocket  
           http://www.springframework.org/schema/websocket/spring-websocket.xsd">  
  
    <websocket:handlers allowed-origins="http://mydomain.com">  
        <websocket:mapping path="/myHandler" handler="myHandler" />  
    </websocket:handlers>  
  
    <bean id="myHandler" class="org.springframework.samples.MyHandler"/>  
  
</beans>
```

## 4.3. SockJS 后备

在公共 Internet 上，控件外部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** Headers，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，然后再尝试使用基于 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(以及 Client 端)支持。

### 4.3.1. Overview

SockJS 的目标是让应用程序使用 WebSocket API，但在运行时需要时使用非 WebSocket 替代方法

, 而无需更改应用程序代码。

SockJS 包括：

- 以可执行文件[narrated tests](#)的形式定义的[SockJS protocol](#)。
- [SockJS JavaScriptClient 端](#) Client 端库, 供浏览器使用。
- SockJS 服务器实现, 包括 Spring Framework `spring-websocket` 模块中的一个。
- `spring-websocket` 模块中的 SockJS JavaClient 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 client](#)页。传输分为三大类: WebSocket, HTTP 流和 HTTP 长轮询。有关这些类别的概述, 请参见[此博客文章](#)。

SockJSClient 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}
```

where:

- `{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 Client 端消息的一个长时间运行的

请求，以及对 Client 端到服务器消息的其他 HTTP POST 请求。长轮询与长轮询类似，不同之处在于长轮询在每次服务器到 Client 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` (“打开” 框架)，消息以 `a[ "message1" , "message2" ]` (JSON 编码数组)发送，如果在 25 秒内没有消息流(默认情况下)，则以字母 `h` (“心跳” 框架)发送消息，和字母 `c` (“关闭” 框架)以关闭会话。

要了解更多信息，请在浏览器中运行示例并查看 HTTP 请求。SockJSClient 端允许修复传输列表，因此可以一次查看每个传输。SockJSClient 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[narrated test](#)。

### 4.3.2. 启用 SockJS

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
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/websocket
           http://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](#)(1.0.x 版)。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。请参见[sockjs-client](#)页和浏览器支持的传输类型列表。Client 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 和 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。

SockJSClient 端通过使用 Microsoft 的[XDomainRequest](#)在 IE 8 和 9 中支持 Ajax/XHR 流。这适用于所有域，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 SockJSClient 端可用于多种服务器类型(不仅是 Java 服务器类型)，因此需要知道 cookie 是否重要。如果是这样，则 SockJSClient 端更喜欢 Ajax/XHR 进行流传输。否则，它依赖于基于 iframe 的技术。

SockJSClient 端发出的第一个 `/info` 请求是对可能影响 Client 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具有粘性会话的群集)。Spring 的 SockJS 支持包括名为 `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该功能。如果您的应用程序不需要它，则可以关闭此选项，然后 SockJSClient 端应在 IE 8 和 9 中选择 `xdr-streaming`。

如果您确实使用基于 iframe 的传输，请记住，可以通过将 HTTP 响应 Headers `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 [clickjacking](#)。

#### iNote

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 Headers，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` Headers 设置的详细信息，请参见 Spring Security 文档的 [默认安全标题](#)。您还可以看到[SEC-2501](#)以获得其他背景。

如果您的应用程序添加了 `X-Frame-Options` 响应 Headers(应如此！)并依赖于基于 `iframe` 的传输，则需要将 Headers 值设置为 `SAMEORIGIN` 或 `ALLOW-FROM <origin>`。Spring SockJS 支持还需要知道 `SockJSClient` 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 `SockJSClient` 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
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @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。

#### iNote

在最初的开发过程中，请启用 SockJSClient 端 `devel` 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 client](#)页。

#### 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，您可以使用该属性来自定义频率。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。对于公共 Internet 应用程序，此 25 秒值与以下[IETF recommendation](#)一致。

##### Note

在 WebSocket 和 SockJS 上使用 STOMP 时，如果 STOMPClient 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` 来安排心跳任务。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 端断开连接

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 `continue` 写入另一个线程的响应。

一个特定的问题是，Servlet API 不会为已离开的 Client 端提供通知。参见[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(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 Client 端断开连接。

## iNote

结果，由于 Client 端已断开连接，可能会发生网络 I/O 故障，这可能会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 Client 端断开连接(特定于每个服务器)的此类网络故障，并使用专用日志类别 `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 和 CORS

如果您允许跨域请求(请参阅[Allowed Origins](#))，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 CORSHeaders 的存在，否则将自动添加 CORSHeaders。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` 将跳过这一部分。

也可以通过在 Spring 的 `SockJsService` 中设置 `suppressCors` 属性来禁用这些 CORSHeaders 的添加。

SockJS 需要以下 Headers 和值：

- `Access-Control-Allow-Origin`：从 `Origin` 请求 Headers 的值初始化。
- `Access-Control-Allow-Credentials`：始终设置为 `true`。
- `Access-Control-Request-Headers`：从等效请求 Headers 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 JavaClient 端，无需使用浏览器即可连接到远程 SockJS 端点。当需要通过公共网络在两个服务器之间进行双向通信时(这是网络代理可以阻止使用 WebSocket 协议的地方)，这特别有用。SockJS JavaClient 端对于测试也非常有用(例如，模拟大量并发用户)。

SockJS JavaClient 端支持 `websocket`，`xhr-streaming` 和 `xhr-polling` 传输。其余的仅在浏览器中使用才有意义。

您可以使用以下方式配置 `WebSocketTransport`：

- 在 JSR-356 运行时中为 `StandardWebSocketClient`。
- `JettyWebSocketClient` 通过使用 Jetty 9 本机 WebSocket API。
- Spring 的 `WebSocketClient` 的任何实现。

顾名思义，`XhrTransport` 支持 `xhr-streaming` 和 `xhr-polling`，因为从 Client 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 SockJSClient 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");
```

### iNote

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 Classpath 上。另外，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 `HTTPClient` 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) (1)
            .setHttpMessageCacheSize(1000) (2)
            .setDisconnectDelay(30 * 1000); (3)
    }

    // ...
}
```

- (1) 将 `streamBytesLimit` 属性设置为 512KB(默认值为 128KB — `128 * 1024`)。
- (2) 将 `httpMessageCacheSize` 属性设置为 1,000(默认值为 `100`)。
- (3) 将 `disconnectDelay` 属性设置为 30 属性秒(默认值为 5 秒— `5 * 1000`)。

## 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, Client 端和服务器可以协商子协议(即高级消息协议), 以在 WebSocket 上使用该协议来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, 但是无论哪种方式, Client 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协议)最初是为脚本语言(例如 Ruby, Python 和 Perl)创建的, 用于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 Lists 显示了 STOMP 帧的结构:

```
COMMAND
header1:value1
header2:value2

Body^@
```

Client 端可以使用 `SEND` 或 `SUBSCRIBE` 命令发送或订阅消息, 以及 `destination` Headers, 该 Headers 描述消息的内容以及应由谁接收。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 Client 端, 或者将消息发送到服务器以请求执行某些工作。

当您使用 Spring 的 STOMP 支持时, Spring WebSocket 应用程序将充当 Client 端的 STOMP 代理。消息被路由到 `@Controller` 消息处理方法或简单的内存中代理, 该代理跟踪订阅并向订阅的用户 Broadcast 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 Broadcast 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 `WebSocketClient` 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。

下面的示例显示了一个订阅以接收股票报价的 Client 端, 服务器可能会定期发出该股票报价(例如, 通过计划的任务, 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 Client 端发送的 Transaction 请求，服务器可以通过 `@MessageMapping` 方法处理该 Transaction 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 `ClientBroadcastTransaction` 确认消息和详细信息。

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 服务器来定义它们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)消息交换。

STOMP 服务器可以使用 `MESSAGE` 命令向所有订户 Broadcast 消息。以下示例显示了服务器向订阅的 Client 端发送股票报价的服务器：

```
MESSAGE
message-id:nxahk1f6-1
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 Client 端订阅，并且服务器消息的 `subscription-id` Headers 必须与 Client 端订阅的 `id` Headers 匹配。

前面的概述旨在提供对 STOMP 协议的最基本的了解。我们建议您完整阅读协议[specification](#)。

## 4.4.2. Benefits

与使用原始 WebSocket 相比，使用 STOMP 作为子协议可以使 Spring 框架和 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 STOMPClient 端，包括 Spring 框架中的[Java client](#)。
- 您可以(可选)使用消息代理(例如 RabbitMQ, ActiveMQ 和其他代理)来 Management 订阅和 Broadcast 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 Headers 将消息路由到它们，而对于给定的连接，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. 启用 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：

```
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(); (1)
    }

    @Override
    public void configureMessageBroker(MessageBrokerRegistry config) {
        config.setApplicationDestinationPrefixes("/app"); (2)
        config.enableSimpleBroker("/topic", "/queue"); (3)
    }
}
```

- (1) `/portfolio` 是 WebSocket(或 SockJS)Client 端需要连接到 WebSocket 握手的端点的

HTTP URL。

- (2) 目标 Headers 以 `/app` 开头的 STOMP 消息被路由到 `@Controller` 类中的 `@MessageMapping` 方法。
- (3) 使用内置的消息代理进行订阅和 Broadcast，并将目标 Headers 以“`/topic` or `/queue`”开头的消息路由到代理。

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
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/websocket
        http://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>
```

### ①Note

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 Consumer)。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 Producing 使用多年，但不再维护。目前，[JSteunou/webstomp-client](#)是该库中最活跃，Developing 最快的继承者。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 连接(没有 SockJS)，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 `login` 和 `passcode` Headers。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 broker](#)和[Authentication](#)。

有关更多示例代码，请参见：

- [使用 WebSocket 构建交互式 Web 应用程序](#) —入门指南。
- [Stock Portfolio](#) —一个示例应用程序。

#### 4.4.4. WebSocket 服务器

要配置基础的 WebSocket 服务器，请使用[Server Configuration](#)中的信息。对于 Jetty，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());  
    }  
  
    @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. 消息流

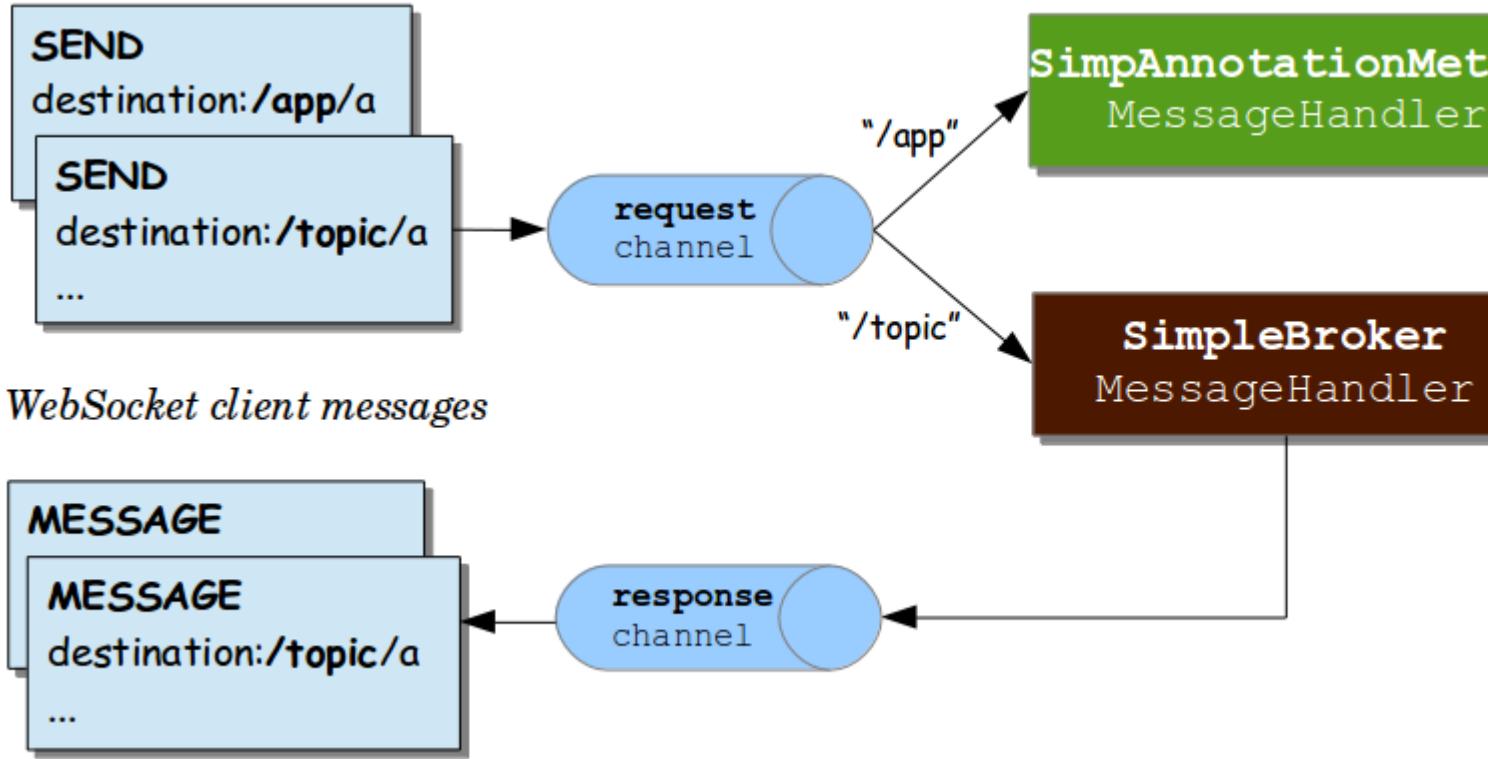
公开 STOMP 端点后，Spring 应用程序将成为已连接 Client 端的 STOMP 代理。本节描述服务器端的消息流。

`spring-messaging` 模块包含对源自 [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 框架中，以便在许多 [Spring projects](#) 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 Contract。
- [MessageChannel](#)：发送消息的 Contract，该消息使生产者和 Consumer 之间的耦合松散。
- [SubscribableChannel](#)： [MessageChannel](#) 和 [MessageHandler](#) 个订阅者。
- [ExecutorSubscribableChannel](#)： [SubscribableChannel](#) 使用 [Executor](#) 传递邮件。

Java 配置(即 `@EnableWebSocketMessageBroker`)和 XML 名称空间配置(即

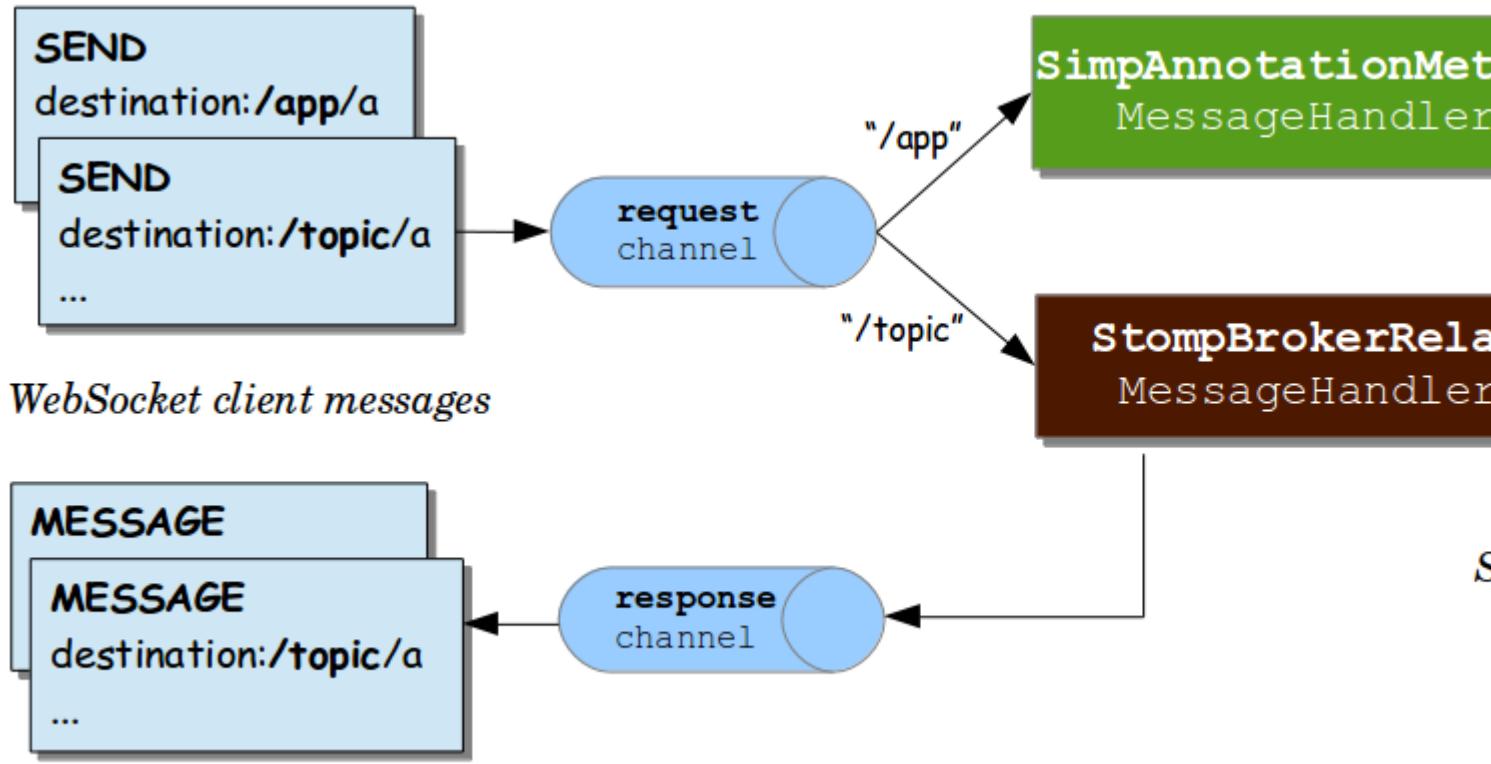
`<websocket:message-broker>`)都使用前面的组件来组装消息工作流。下图显示了启用简单内置消息代理时使用的组件：



上图显示了三个消息通道：

- `clientInboundChannel`：用于传递从 WebSocketClient 端收到的消息。
- `clientOutboundChannel`：用于向 WebSocketClient 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 Management 订阅和 Broadcast 消息时使用的组件：



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到订阅的 Client 端。

从 WebSocket 连接接收到消息后，消息将被解码为 STOMP 帧，转换为 Spring `Message` 表示形式，然后发送至 `clientInboundChannel` 进行进一步处理。例如，目标 Headers 以 `/app` 开头的 STOMP 消息可以路由到带 `Comments` 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 Client 端的 STOMP 消息的带 `Comments` 的 `@Controller` 可以通过 `brokerChannel` 向消息代理发送消息，并且代理通过 `clientOutboundChannel` 将消息 Broadcast 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 Client 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 Broadcast 到订阅的 Client 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：

- Client 端连接到 `http://localhost:8080/portfolio`，一旦构建了 WebSocket 连接，STOMP 帧就开始在其上流动。
- Client 端发送带有目 Headers `/topic/greeting` 的 SUBSCRIBE 帧。收到并解码后，该消息将发送到 `clientInboundChannel`，然后路由到消息代理，该代理存储 Client 端订阅。
- Client 端向 `/app/greeting` 发送一个 aSEND 帧。`/app` 前缀有助于将其路由到带 Comments 的控制器。除去 `/app` 前缀后，目标的其余 `/greeting` 部分将 Map 到 `GreetingController` 中的 `@MessageMapping` 方法。
- 从 `GreetingController` 返回的值被转换为带有返回值和默认目 Headers `/topic/greeting` (从 Importing 目标中用 `/app` 替换为 `/topic` 的源目标)的有效负载的 Spring `Message`。结果消息将发送到 `brokerChannel` 并由消息代理处理。
- 消息代理查找所有匹配的订户，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 `Comments` 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. 带 `Comments` 的控制器

应用程序可以使用带 `Comments` 的 `@Controller` 类来处理来自 Client 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` `Comments` 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 Map。

默认情况下，Map 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 Map 目标约定，如[点作为分隔符](#)中所述。

##### 支持的方法参数

下表描述了方法参数：

Method argument	Description
<code>Message</code>	用于访问完整的消息。

Method argument	Description
<code>MessageHeaders</code>	用于访问 <code>Message</code> 中的标题。
<code>MessageHeaderAccessor</code> , <code>SimpMessageHeaderAccessor</code> 和 <code>StompHeaderAccessor</code>	用于通过类型化访问器方法访问 Headers。
<code>@Payload</code>	为了访问消息的有效负载，由配置的 <code>MessageConverter</code> 转换(例如，从 JSON 转换)。

不需要此 Comments，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 `@Validated` Comments 有效负载参数，以使有效负载参数得到自动验证。

| `@Header` | 用于访问特定的 Headers 值-必要时还可以使用

`org.springframework.core.convert.converter.Converter` 进行类型转换。

| `@Headers` | 用于访问消息中的所有标题。此自变量必须可分配给 `java.util.Map`。

| `@DestinationVariable` | 用于访问从消息目标提取的模板变量。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，来自 `@MessageMapping` 方法的返回值通过匹配的 `MessageConverter` 序列化为有效负载，并以 `Message` 的形式发送到 `brokerChannel`，并从那里 Broadcast 给订户。出站消息

的目的地与入站消息的目的地相同，但前缀为 `/topic`。

您可以使用 `@SendTo` 和 `@SendToUser` `Comments` 来自定义输出消息的目的地。`@SendTo` 用于自定义目标目的地或指定多个目的地。`@SendToUser` 用于将输出消息仅定向到与 Importing 消息关联的用户。参见[User Destinations](#)。

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，它们将作为类中方法的默认值。但是，请记住，任何方法级别的 `@SendTo` 或 `@SendToUser` `Comments` 都将在类级别覆盖所有此类 `Comments`。

消息可以异步处理，并且 `@MessageMapping` 方法可以返回 `ListenableFuture`，  
`CompletableFuture` 或 `CompletionStage`。

请注意，`@SendTo` 和 `@SendToUser` 仅仅是一种便利，等同于使用 `SimpMessagingTemplate` 发送消息。如有必要，对于更高级的方案，`@MessageMapping` 方法可以直接使用 `SimpMessagingTemplate`。这可以代替返回值，也可以附加于返回值。参见[Sending Messages](#)。

## `@SubscribeMapping`

`@SubscribeMapping` 与 `@MessageMapping` 类似，但是将 Map 范围缩小到仅订阅消息。它支持与 `@MessageMapping` 相同的[method arguments](#)。但是，对于返回值，默认情况下，将消息直接发送到 Client 端(通过 `clientOutboundChannel`，作为对订阅的响应)，而不发送给代理(通过 `brokerChannel`，作为对匹配的订阅的 Broadcast)。添加 `@SendTo` 或 `@SendToUser` 会覆盖此行为，而是发送给代理。

什么时候有用？假定代理 Map 到 `/topic` 和 `/queue`，而应用程序控制器 Map 到 `/app`。在此设置中，代理将所有要重复 Broadcast 的 `/topic` 和 `/queue` 订阅存储起来，并且不需要应用程序参与。Client 端还可以预订某个 `/app` 目的地，并且控制器可以响应该预订而返回一个值，而无

需 broker 参与，而无需再次存储或使用该预订(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 Map 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 broker 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 Broadcast 时通知目标，则 Client 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 client](#)，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 `brokerChannel` 上的[to register](#) 和 `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` 个方法来处理 `@MessageMapping` 个方法中的异常。如果要访问异常实例，则可以在注解本身中声明异常，也可以通过方法参数声明异常。下面的示例通过方法参数声明异常：

```
@Controller
public class MyController {

    // ...

    @MessageExceptionHandler
    public ApplicationError handleException(MyException exception) {
        // ...
        return appError;
    }
}
```

`@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)。如果要使此类方法更全局地应用(跨控制器)，则可以在标有 `@ControllerAdvice` 的类中声明它们。这与 Spring MVC 中可用的[类似的支持](#)相当。

#### 4.4.7. 传送讯息

如果要从应用程序的任何部分向连接的 Client 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. 简单 broker

内置的简单消息代理处理来自 Client 端的订阅请求，将其存储在内存中，并将消息 Broadcast 到具有匹配目标的已连接 Client 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。

## iNote

应用程序还可以使用点分隔(而不是斜杠分隔)目标。参见[点作为分隔符](#)。

如果配置了任务计划程序，则简单代理支持[STOMP heartbeats](#)。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})
            .setTaskScheduler(this.messageBrokerTaskScheduler);

        // ...
    }
}
```

### 4.4.9. 外部 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)，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 文档以获取您所选择的消息代理(例如[RabbitMQ](#), [ActiveMQ](#)等)，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
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/websocket
           http://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，它构建到代理的 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 Client 端。本质上，它充当双向转发消息的“中继”。

### iNote

将 `io.projectreactor.netty:reactor-netty` 和 `io.netty:netty-all` 依赖项添加到您的项目中以进行 TCP 连接 Management。

此外，应用程序组件(例如 HTTP 请求处理方法，业务服务等)还可以将消息发送到代理中继(如 [Sending Messages](#) 中所述)，以将消息 Broadcast 到订阅的 WebSocketClient 端。

实际上，代理中继可实现健壮且可伸缩的消息 Broadcast。

## 4.4.10. 连接到 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 帧 `login` 和 `passcode` Headers)。这在 XML 名称空间和 Java 配置中都公开为 `systemLogin` 和 `systemPasscode` 属性，，默认值为 `guest` 和 `guest`。

STOMP 代理中继还为每个连接的 `WebSocketClient` 端创建一个单独的 TCP 连接。您可以配置用于代表 Client 端创建的所有 TCP 连接的 STOMP 凭据。这在 XML 名称空间和 Java 配置中均以默认值 `guest` and `guest` 的 `clientLogin` and `clientPasscode`` 属性公开。

### iNote

STOMP 代理中继始终在代表 Client 端转发给代理的每个 `CONNECT` 帧上设置 `login` 和 `passcode` Headers。因此，`WebSocketClient` 端无需设置这些 Headers。他们被忽略。如 [Authentication](#) 部分所述，`WebSocketClient` 端应改为依靠 HTTP 身份验证来保护 `WebSocket` 端点并构建 Client 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 `continue` 尝试重新连接，直到成功。

当与代理的“系统”连接丢失并重新构建时，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，`Broadcast` 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 `host` Headers，并且很有用(例如，在构建 TCP 连接的实际主机不同于提供基于云的 STOMP 服务的主机的云环境中)。

#### 4.4.11. 点作为分隔符

将消息路由到 `@MessageMapping` 方法时，它们与 `AntPathMatcher` 匹配。默认情况下，模式应使用斜杠(`/`)作为分隔符。这是 Web 应用程序中的一种良好约定，类似于 HTTP URL。但是，如果您更习惯于消息传递约定，则可以切换到使用点(`.`)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
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    // ...

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.setPathMatcher(new AntPathMatcher("."));
        registry.enableStompBrokerRelay("/queue", "/topic");
        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
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/websocket
        http://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("erd")
public class RedController {

    @MessageMapping("blue.{green}")
    public void handleGreen(@DestinationVariable String green) {
        // ...
    }
}

```

Client 端现在可以向 `/app/red.blue.green123` 发送消息。

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 Headers 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 后备情况下，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。

因此，对于 WebSocket 握手或 SockJS HTTP 传输请求，通常已经有一个通过

`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，并随后与通过该用户 Headers 在该会话上传的所有 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，就不需要做任何事情。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 `Message` 流上标记用户 Headers 通过应用程序。

请注意，STOMP 协议在 `CONNECT` 帧上确实具有 `login` 和 `passcode` Headers。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 Headers，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 会话包含经过身份验证的用户。

#### iNote

Spring Security 提供了[WebSocket 子协议授权](#)，该[WebSocket 子协议授权](#)使用 `ChannelInterceptor` 来基于消息中的用户 Headers 授权消息。另外，Spring Session 提供了一个[WebSocket integration](#)，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. 令牌认证

[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 的会话并不总是最合适的选择(例如，在不维护服务器端会话的应用程序中或在通常使用 Headers 进行身份验证的移动应用程序中)。

[WebSocket 协议, RFC 6455](#)“没有规定服务器在 WebSocket 握手期间可以对 Client 端进行身份验证的任何特定方式。”但是，实际上，浏览器 Client 端只能使用标准身份验证 Headers(即基本 HTTP 身份验证)或 cookie，而不能(例如)提供自定义 Headers。同样，SockJS JavaScriptClient 端也不提供通过 SockJS 传输请求发送 HttpHeaders 的方法。参见[sockjs-Client 端问题 196](#)。相反，它确实允许发送可用于发送令牌的查询参数，但这有其自身的缺点(例如，令牌可能会无意中与服务器日志中的 URL 一起记录)。

#### Note

前面的限制适用于基于浏览器的 Client 端，不适用于基于 Spring Java 的 STOMPClient 端，该 Client 端确实支持通过 WebSocket 和 SockJS 请求发送 Headers。

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选择。他们可能更喜欢在 STOMP 消息传递协议级别使用 Headers 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 STOMPClient 端在连接时传递身份验证头。
- 使用 `ChannelInterceptor` 处理身份验证 Headers。

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 `Message` 上设置用户 Headers。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);
        }
        return message;
    }
}
}

```

另外, 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 Order 身份验证 `ChannelInterceptor config`。最好通过在其自己的 `WebSocketMessageBrokerConfigurer` 实现中声明用 `@Order(Ordered.HIGHEST_PRECEDENCE + 99)` 标记的自定义拦截器来完成。

#### 4.4.14. 用户目的地

应用程序可以发送针对特定用户的消息, Spring 的 STOMP 支持为此识别前缀 `/user/` 的目标。例如, Client 端可能订阅了 `/user/queue/position-updates` 目标。该目的地由 `UserDestinationMessageHandler` 处理, 并转换为用户会话唯一的目的地(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, 消息可以发送到诸如 `/user/{username}/queue/position-updates` 之类的目的地, 而该目的地又被 `UserDestinationMessageHandler` 转换为一个或多个目的地, 每个与用户相关联的会话都需要一个目的地。这使应用程序内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。Comments 和消息传递模板也支持此功能。

消息处理方法可以通过 `@SendToUser` Comments 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一个公共目的地), 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` 来实现，如下示例所示：

```
@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;
    }
}
```

### iNote

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` `Comments` 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`。(如果要使用 `@Qualifier` 进行限定，则 Bean 名称为 `"brokerMessagingTemplate"`.)下面的示例演示了如何实现：

```

@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());
    }
}

```

### **i**Note

将用户目标与外部消息代理一起使用时，应查看代理文档以了解如何 Management 非活动队列，以便在用户会话结束时，将删除所有唯一的用户队列。例如，当您使用诸如

`/exchange/amq.direct/position-updates` 之类的目标时，RabbitMQ 将创建自动删除

队列。因此，在这种情况下，Client 端可以订阅

`/user/exchange/amq.direct/position-updates`。同样，ActiveMQ 具有[configuration options](#) 用于清除非活动目标。

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仍然无法解析。在这种情况下，可以将目标配置为 Broadcast 未解析的消息，以便其他服务器可以尝试。这可以通过 Java 配置中 `MessageBrokerRegistry` 的 `userDestinationBroadcast` 属性和 XML 中 `message-broker` 元素的 `user-destination-broadcast` 属性来完成。

## 4.4.15. 消息 Sequences

来自代理的消息将发布到 `clientOutboundChannel`，然后从那里写入 WebSocket 会话。由于该通道由 `ThreadPoolExecutor` 支持，因此将在不同的线程中处理消息，并且 Client 端接收到的结果序列可能与发布的确切 Sequences 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);
    }

}
```

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
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/websocket
           http://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 Client 端会话中的消息将一次发布到 `clientOutboundChannel`, 因此可以保证发布 Sequences。请注意, 这会产生很小的性能开销, 因此, 只有在需要时才应启用它。

#### 4.4.16. Events

几个 `ApplicationContext` 事件已发布, 可以通过实现 Spring 的 `ApplicationListener` 接口来接收:

- `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构建与代理的“系统”连接。结果, 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 Client 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 Client 端发送的所有自定义 Headers。这对于跟踪 Client 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。此时，可以认为 STOMP 会话已完全构建。
- `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 可能已经从 Client 端发送，或者它可能在 WebSocket 会话关闭时自动生成。在某些情况下，每个会话多次发布该事件。组件应与多个断开事件有关。

#### **Note**

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，Client 端连接不会自动重新连接。假设启用了心跳，则 Client 端通常会注意到代理在 10 秒内没有响应。Client 需要实现自己的重新连接逻辑。

### 4.4.17. Interception

[Events](#) 提供 STOMP 连接生命周期的通知，但不提供每条 Client 端消息的通知。应用程序还可以注册 `ChannelInterceptor` 来拦截处理链中任何部分的任何消息。以下示例显示了如何拦截来自 Client 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` 访问有关消息的信息，如以下示例所示：

```
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。对于发送到通道的每个消息，一次调用 `ChannelInterceptor` 时，`ExecutorChannelInterceptor` 在订阅该通道消息的每个 `MessageHandler` 的线程中提供了钩子。

请注意，与前面所述的 `SesionDisconnectEvent` 一样，`DISCONNECT` 消息可以来自 Client 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。组件应与多个断开事件有关。

## 4.4.18. STOMPClient 端

Spring 提供了一个基于 `WebSocket` 的 `STOMPClient` 端和一个基于 `TCP` 的 `STOMPClient` 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，您可以将`_`替换为 `SockJsClient`，因为这也是 `WebSocketClient` 的实现。

`SockJsClient` 可以使用 `WebSocket` 或基于 `HTTP` 的传输作为后备。有关更多详细信息，请参见 [SockJsClient](#)。

接下来，您可以构建连接并为 `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);
```

会话准备就绪时，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) {
        // ...
    }
}
```

构建会话后，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。`subscribe` 方法需要用于订阅消息的处理程序，并返回 `Subscription` 句柄，您可以使用该句柄取消订阅。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 `Object` 类型，如以下示例所示：

```
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，您可以将 `WebSocketStompClient` 配置为 `TaskScheduler`，并可以选择自定义心跳间隔(写不活动(导致发送心跳)需要 10 秒，读不活动(可以关闭连接)需要 10 秒)。

#### iNote

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 Client 端时，请考虑关闭心跳 `signal`，因为每个连接都会调度自己的心跳任务，并且并未针对同一台计算机上运行的大量 Client 端进行优化。

STOMP 协议还支持回执，在该回执中，Client 端必须添加 `receipt` 报头，服务器在处理完发送或订阅后将以 RECEIPT 帧响应。为了支持这一点，`StompSession` 提供了 `setAutoReceipt(boolean)`，该 `setAutoReceipt(boolean)` 会在每个后续的 `send` 或 `subscription` 事件上添加 `receipt` Headers。或者，您也可以手动将收据 Headers 添加到 `StompHeaders`。发送和订阅都返回 `Receiptable` 的实例，您可以使用该实例来注册接收成功和失败的回调。要使用此功能，您必须为 Client 端配置 `TaskScheduler` 以及收据过期之前的时间(默认认为 15 秒)。

请注意，`StompSessionHandler` 本身是 `StompFrameHandler`，除了用于处理来自消息的异常的 `handleException` 回调和用于 `ConnectionLostException` 的传输级错误的 `handleTransportError` 之外，它还可以处理 ERROR 帧。

## 4.4.19. WebSocket 范围

每个 WebSocket 会话都有一个属性 Map。该 Map 作为 Headers 附加到入站 Client 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 `SpringManagement` 的 bean。您可以将 WebSocket 作用域的 bean 注入控制器和在 `clientInboundChannel` 上注册的任何通道拦截器中。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
    }

    // ...

    @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 首次从控制器访问它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，没有万灵药。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，以进行线程池支持的异步执行。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](#)。

最明显的起点是配置支持 `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 `Comments` 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 `await` 数据库或其他外部系统，则可能需要增加线程池大小。

##### Note

`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增加到超出核心池大小的范围。

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，所有操作都与向 `WebSocketClient` 端发送消息有关。如果 Client 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。

尽管可以预测 `clientInboundChannel` 的工作量(毕竟，这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 Client 端时允许发送多长时间以及可以缓冲多少数据。

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 Client 端。同时，所有其他消息都将被缓冲，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
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/websocket
           http://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，STOMPCClient 端(例如 JavaScript [webstomp-client](#)等)会在 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);
    }

    // ...
}
```

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
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/websocket
           http://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 Broadcast 的消息可以通过代理 Broadcast 到通过任何其他应用程序实例连接的 WebSocketClient 端。

## 4.4.21. Monitoring

当您使用 `@EnableWebSocketMessageBroker` 或 `<websocket:message-broker>` 时，关键基础架构组件会自动收集统计信息和计数器，这些统计信息和计数器可提供对应用程序内部状态的重要了解。该配置还声明了一个类型为 `WebSocketMessageBrokerStats` 的 bean，该 bean 在一个地方收集所有可用信息，并且默认情况下，每 30 分钟在 `INFO` 级别收集一次。该 bean 可以通过 Spring 的 `MBeanExporter` 导出到 JMX，以便在运行时查看(例如，通过 JDK 的 `jconsole`)。以下列表总结了可用信息：

- Client 端 WebSocket 会话
  - ◦ Current
    - 指示当前有多少个 Client 端会话，并且通过 WebSocket 与 HTTP 流和轮询 SockJS 会话进一步细分计数。
  - Total
    - 指示已构建的会话总数。
  - Abnormally Closed
    - ◦ Connect Failures
      - 已构建但在 60 秒内未收到任何消息后关闭的会话。这通常表示代理或网络问题。
    - 超过发送限制
      - 超过配置的发送超时或发送缓冲区限制后，会话将关闭，慢 Client 端可能会发生这种情况(请参阅上一节)。
  - Transport Errors
    - 传输错误(例如无法读取或写入 WebSocket 连接或 HTTP 请求或响应)后，会话关闭。
  - STOMP Frames
    - 已处理的 CONNECT, CONNECTED 和 DISCONNECT 帧的总数，指示在 STOMP 级别

上连接了多少个 Client 端。请注意，当会话异常关闭或 Client 端未发送 DISCONNECT 帧而关闭时，DISCONNECT 计数可能会降低。

- STOMPbroker 接力
  - ◦ TCP Connections
    - 指示与代理构建了代表 Client 端 WebSocket 会话的 TCP 连接数。这应该等于 Client 端 WebSocket 会话的数量 1 个用于从应用程序内部发送消息的附加共享“系统”连接◦
    -
  - STOMP Frames
    - 代表 Client 端转发到代理或从代理接收的 CONNECT, CONNECTED 和 DISCONNECT 帧总数。请注意，无论 Client 端 WebSocket 会话如何关闭，DISCONNECT 帧都会发送到代理。因此，较低的 DISCONNECT 帧计数表示代理正在主动关闭连接(可能是由于未及时到达的心跳，无效的 Importing 帧或其他问题)。
- Client 入站通道
  - 来自支持 `clientInboundChannel` 的线程池的统计信息，可深入了解传入消息处理的运行状况。此处排队的任务表明该应用程序可能太慢而无法处理消息。如果存在 I/O 绑定的任务(例如，慢速的数据库查询，对第三方 REST API 的 HTTP 请求等)，请考虑增加线程池的大小。
- Client 出站通道
  - 来自支持 `clientOutboundChannel` 的线程池的统计信息，该统计信息可深入了解向 Client 端 Broadcast 消息的运行状况。此处排队的任务表明 Client 端太慢而无法使用消息。解决此问题的一种方法是增加线程池大小，以容纳并发慢速 Client 端的预期数量。另一个选择是减少发送超时和发送缓冲区大小限制(请参阅上一节)。
- SockJS 任务计划程序
  - 来自 SockJS 任务调度程序的线程池的统计信息，用于发送心跳。请注意，在 STOMP 级别

协商心跳时，将禁用 SockJS 心跳。

## 4.4.22. Testing

当您使用 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 `Comments` 的消息处理方法。第二种是编写涉及运行 Client 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，但是编写和维护它们的工作也更多。

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 `Comments`。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 `Comments` 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，并使用它发送要由控制器方法处理的消息。
- 手动设置调用控制器(即 `SimpAnnotationMethodMessageHandler`)并将控制器消息直接传递给它所需的最低 Spring 框架基础结构。

[股票投资组合测试](#)示例应用程序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 `WebSocketClient` 端连接到该服务器，该 Client 端发送包含 STOMP 帧的 WebSocket 消息。[股票投资组合测试](#)示例应用程序还通过使用 Tomcat 作为嵌入式 WebSocket 服务器和用于测试目的的简单 `STOMPClient` 端，演示了这种方法。

## 5.其他 Web 框架

本章详细介绍了 Spring 与第三方 Web 框架的集成。

Spring Framework 的核心价值主张之一是启用\* choice \*。在一般意义上，Spring 不会强迫您使用或购买任何特定的体系结构，技术或方法(尽管它肯定比其他建议更重要)。可以自由选择与开发人员及其开发团队最相关的架构，技术或方法，这在 Web 领域最明显，在 Web 领域，Spring 提供了自己的 Web 框架([Spring MVC](#))，而与此同时，提供与许多流行的第三方 Web 框架的集成。

## 5.1. 通用配置

在深入研究每个受支持的 Web 框架的集成细节之前，让我们首先看一下并非针对任何一个 Web 框架的 Spring 配置。(本节同样适用于 Spring 自己的 Web 框架 Spring MVC.)

Spring 的轻量级应用程序模型拥护的一个概念(需要一个更好的词)是分层体系结构的概念。请记住，在“经典”分层体系结构中，Web 层只是许多层之一。它充当服务器端应用程序的入口点之一，并且委派服务层中定义的服务对象(外观)，以满足特定于业务(与表示技术无关)的用例。在 Spring 中，这些服务对象，任何其他特定于业务的对象，数据访问对象和其他对象都存在于不同的“业务上下文”中，其中不包含 Web 或表示层对象(表示对象，例如 Spring MVC 控制器，通常是在不同的“展示环境”中进行配置)。本节详细介绍如何配置包含应用程序中所有“Business Bean”的 Spring 容器([WebApplicationContext](#))。

continue 讲细节，您要做的就是在 Web 应用程序的标准 Java EE servlet [web.xml](#) 文件中声明一个[ContextLoaderListener](#)，并添加 [contextConfigLocation](#) `<context-param>` 部分(在同一文件中)，该部分定义了要设置的 Spring XML 配置文件集。加载。

请考虑以下 `<listener>` 配置：

```
<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

进一步考虑以下 `<context-param>` 配置：

```
<context-param>
```

```
<param-name>contextConfigLocation</param-name>
<param-value>/WEB-INF/applicationContext*.xml</param-value>
</context-param>
```

如果未指定 `contextConfigLocation` 上下文参数，则 `ContextLoaderListener` 查找要加载的名为 `/WEB-INF/applicationContext.xml` 的文件。加载上下文文件后，Spring 将根据 bean 定义创建一个 [WebApplicationContext](#) 对象，并将其存储在 Web 应用程序的 `ServletContext` 中。

所有 Java Web 框架都构建在 Servlet API 的基础上，因此您可以使用以下代码段来访问由 `ContextLoaderListener` 创建的“业务上下文” `ApplicationContext`。

以下示例显示了如何获取 `WebApplicationContext`：

```
WebApplicationContext ctx = WebApplicationContextUtils.getWebApplicationContext(servlet
```

[WebApplicationContextUtils](#) 类是为了方便起见，因此您无需记住 `ServletContext` 属性的名称。

如果 `WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE` 键下的对象不存在，[则其 `getWebApplicationContext\(\)` 方法返回 `null`](#)。最好不要使用 `getRequiredWebApplicationContext()` 方法，而不要冒在应用程序中使用 `NullPointerExceptions` 的风险。`ApplicationContext` 丢失时，此方法将引发异常。

一旦有了对 `WebApplicationContext` 的引用，就可以按其名称或类型检索 bean。大多数开发人员都按名称检索 bean，然后将其转换为实现的接口之一。

幸运的是，本节中的大多数框架都具有更简单的查找 bean 的方法。它们不仅使从 Spring 容器中获取 bean 变得容易，而且还使您可以在其控制器上使用依赖项注入。每个 Web 框架部分都有其特定集成策略的更多详细信息。

## 5.2. JSF

JavaServer Faces(JSF)是 JCP 的基于组件的标准，事件驱动的 Web 用户界面框架。从 Java EE 5 开始，它是 Java EE 总括的正式组成部分。

对于流行的 JSF 运行时以及流行的 JSF 组件库, 请查看[Apache MyFaces 项目](#)。MyFaces 项目还提供了常见的 JSF 扩展, 例如[MyFaces Orchestra](#)(基于 Spring 的 JSF 扩展, 提供了丰富的对话范围支持)。

#### iNote

Spring Web Flow 2.0 通过其新构建的 Spring Faces 模块提供了丰富的 JSF 支持, 既可用于以 JSF 为中心的用法(如本节所述), 又可用于以 Spring 为中心的用法(在 Spring MVC 调度程序中使用 JSF 视图)。有关详情, 请参见[Spring Web Flow 网站](#)。

Spring 的 JSF 集成中的关键元素是 JSF `ELResolver` 机制。

### 5.2.1. Spring Bean 解析器

`SpringBeanFacesELResolver` 是符合 JSF 1.2 的 `ELResolver` 实现, 与 JSF 1.2 和 JSP 2.1 使用的标准 Unified EL 集成在一起。作为 `SpringBeanVariableResolver`, 它首先委派给 Spring 的“业务上下文” `WebApplicationContext`, 然后委派给基础 JSF 实现的默认解析器。

在配置方面, 您可以在 JSF `faces-context.xml` 文件中定义 `SpringBeanFacesELResolver`, 如以下示例所示:

```
<faces-config>
    <application>
        <el-resolver>org.springframework.web.jsf.el.SpringBeanFacesELResolver</el-resolver>
        ...
    </application>
</faces-config>
```

### 5.2.2. 使用 FacesContextUtils

将属性 Map 到 `faces-config.xml` 中的 bean 时, 自定义 `VariableResolver` 效果很好, 但是有时您可能需要显式地获取 bean。[FacesContextUtils](#) 类使此操作变得容易。它与 `WebApplicationContextUtils` 类似, 除了它采用 `FacesContext` 参数而不是

`ServletContext` 参数。

以下示例显示了如何使用 `FacesContextUtils` :

```
ApplicationContext ctx = FacesContextUtils.getWebApplicationContext(FacesContext.getCurrentContext());
```

## 5.3. Apache Struts 2.x

[Struts](#) 由 Craig McClanahan 发明，是由 Apache Software Foundation 托管的开源项目。当时，它极大地简化了 JSP/Servlet 编程范例，并赢得了许多使用专有框架的开发人员的青睐。它简化了编程模型，它是开源的(因此像啤酒一样是免费的)，并且它具有庞大的社区，这使该项目得以 Developing 并在 Java Web 开发人员中广受欢迎。

查看 Struts [Spring Plugin](#)，了解 Struts 随附的内置 Spring 集成。

## 5.4. 挂毯 5.x

[Tapestry](#) 是“用于在 Java 中创建动态，健壮，高度可伸缩的 Web 应用程序的面向组件的框架。”

尽管 Spring 拥有自己的[强大的网页层](#)，但是通过将 Tapestry 用于 Web 用户界面并将 Spring 容器用于底层，来构建企业 Java 应用程序具有许多独特的优势。

有关更多信息，请参见 Tapestry 的专用[Spring 集成模块](#)。

## 5.5. 更多资源

以下链接提供了有关本章中描述的各种 Web 框架的更多资源。

- [JSF 主页](#)
- [Struts 主页](#)
- [Tapestry 主页](#)

# 网上反应堆

文档的此部分涵盖对基于[Reactive Streams](#) API 构建的反应堆式 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。各章介绍[Spring WebFlux](#)框架，响应式[WebClient](#)，对[testing](#)的支持和[reactive libraries](#)。对于 Servlet 堆栈 Web 应用程序，请参见[Web on Servlet 堆栈](#)。

# 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 容器构建的。Reactive 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 框架都镜像其源模块的名称([spring-webmvc](#)和[spring-webflux](#))，并在 Spring 框架中并排共存。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两个模块，例如，带有响应 [WebClient](#) 的 Spring MVC 控制器。

## 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，Contract 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，因为在异步，非阻塞空间中构建良好的服务器(例如 Netty)。

答案的另一部分是函数式编程。就像在 Java 5 中添加 Comments 会 Creating 机会(例如带 Comments 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 APICreating 机会。这对于无阻塞的应用程序和延续样式的 API(如 [CompletableFuture](#) 和 [ReactiveX](#)所流行)的好处是，它们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 Comments 的控制器一起提供功能性的 Web 端点。

### 1.1.1. 定义 “Reactive”

我们谈到了“非阻塞性”和“功能性”，但是 Reactive 是什么意思？

术语“Reactive”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 Reactive 的，因为我们现在正处于操作完成或数据可用时对通知进行反应的方式，而不是被阻塞。

我们 Spring 团队还有另一个重要机制与“Reactive”相关联，这是不阻塞背压的机制。在同步，命令式代码中，阻塞调用是强制调用者 `await` 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[small spec](#)(在 Java 9 中也是[adopted](#))，它定义了具有反压力的异步组件之间的交互。例如，数据存储库(充当[Publisher](#))可以生成 HTTP 服务器(充当[Subscriber](#))然后可以写入响应的数据。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。

#### iNote

##### 常见问题：发布商不能放慢脚步怎么办？

反应流的目的仅仅是构建机制和边界。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. ReactiveAPI

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。这就是反应式库发挥的作用。

[Reactor](#)是 Spring WebFlux 的反应库选择。它通过与 [ReactiveX 运算符词汇](#)对齐的一组丰富的运算符，提供[Mono](#)和[Flux](#) API 类型，以处理 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 要求 Reactor 作为核心依赖项，但是它可以通过 Reactive Streams 与其他反应式库互操作。通常，WebFlux API 接受纯 `Publisher` 作为 Importing，在内部将其适应于 Reactor 类型，使用该类型，然后返回 `Flux` 或 `Mono` 作为输出。因此，您可以传递任何 `Publisher` 作为 Importing，并且可以对输出应用操作，但是您需要调整输出以与另一个反应式库一起使用。只要可行(例如，带 `Comments` 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[Reactive Libraries](#)。

### 1.1.3. 编程模型

`spring-web` 模块包含 Spring WebFlux 基础的反应式基础，包括 HTTP 抽象，用于受支持服务器的 Reactive 流[adapters](#), [codecs](#)以及与 Servlet API 类似但具有非阻塞 Contract 的核心[WebHandler API](#)。

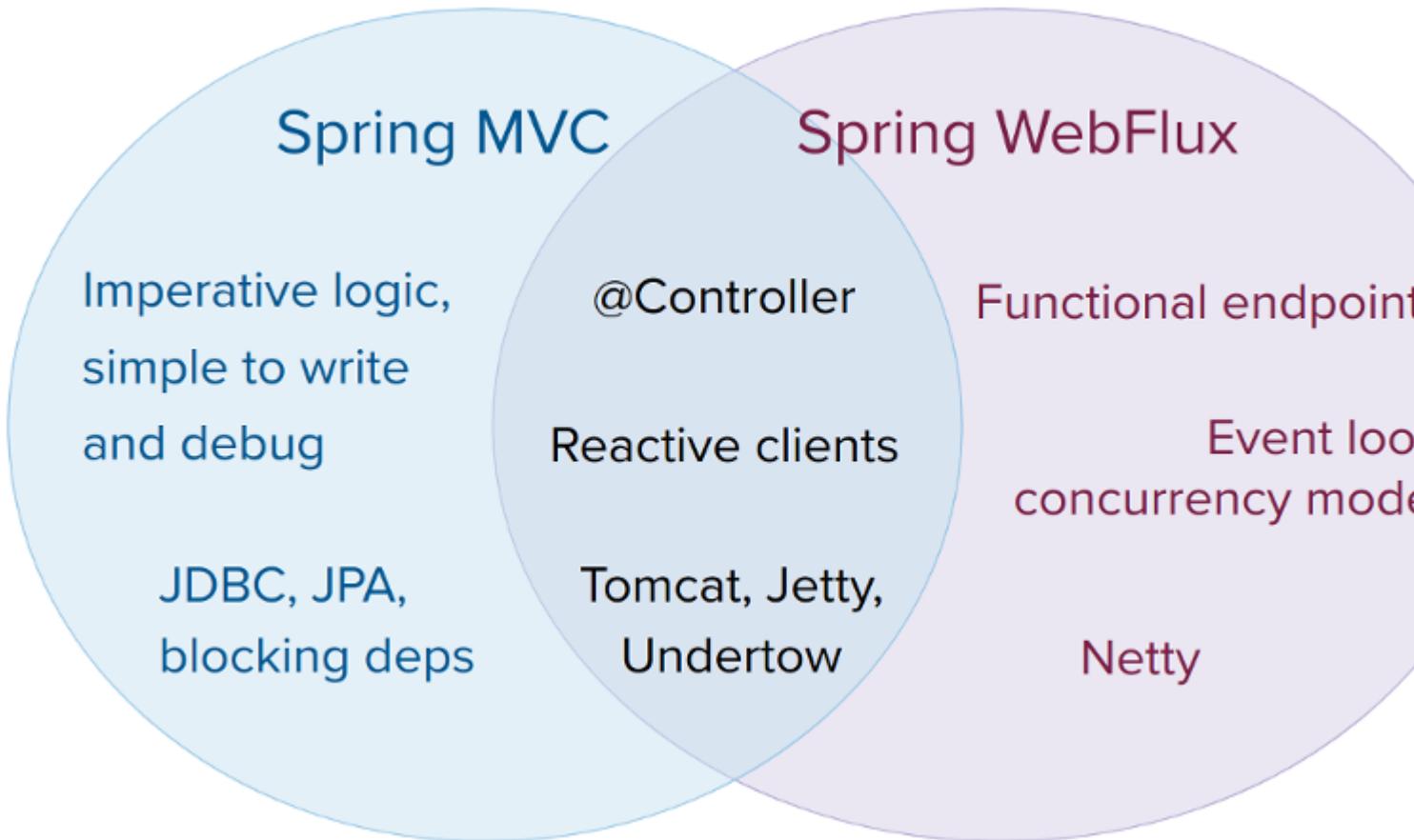
在此基础上，Spring WebFlux 提供了两种编程模型的选择：

- [Annotated Controllers](#): 与 Spring MVC 一致，并基于 `spring-web` 模块中的相同 `Comments`。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 Reactive `@RequestBody` 参数。
- [Functional Endpoints](#): 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 Util，应用程序可以使用它们来路由和处理请求。带 `Comments` 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 `Comments` 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，但构建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。
- 如果您已经在购买非阻塞式 Web 堆栈，Spring WebFlux 可以在该领域提供与其他应用程序相同的执行模型优势，还可以选择服务器(Netty, Tomcat, Jetty, Undertow 和 Servlet 3.1 容器)。编程模型(带 Comments 的控制器和功能性 Web 端点)，以及反应式库(Reactor, RxJava 或其他)的选择。
- 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 Comments 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 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，但是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, [or other](#))。每个呼叫的 await 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，功能和声明式编程的过渡过程中的学习曲线很陡。在没有完全切换的情况下启动的实际方法是使用电抗 [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。所有服务器都适用于低级别的[common API](#)，因此跨服务器可以支持更高级别的[programming models](#)。

Spring WebFlux 不具有内置支持来启动或停止服务器。但是，很容易用几行代码从 Spring 配置中 [assemble](#)一个应用程序和[WebFlux infrastructure](#)和[run it](#)。

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 Client 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，并在需要时允许应用程序直接使用 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

表演具有许多 Feature 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。

Reactive 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. 并发模型

Spring MVC 和 Spring WebFlux 都支持带 Comments 的控制器，但是并发模型和默认的阻塞和线程假设存在关键差异。

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(事件循环工作者)来处理请求。

### Tip

“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(而是依赖于回调)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

如果确实需要使用阻止库怎么办？Reactor 和 RxJava 都提供 `publishOn` 运算符以 `continue` 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。

### Mutable State

在 Reactor 和 RxJava 中，您可以通过运算符声明逻辑，然后在运行时形成反应式管道，在该管道

中，数据在不同的阶段被 Sequences 处理。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并发调用。

## Threading Model

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 容器可能以更多线程开始(例如，Tomcat 上为 10)，以支持 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 `reactor-http-nio-`)。但是，如果 Client 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 提供了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 `Scheduler` 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 框架不支持启动和停止[servers](#)。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，如果使用 Spring Boot，请检查每个服务器的 Spring Boot 配置选项。您可以直接[configure](#) `WebClient`。对于所有其他库，请参阅其各自的文档。

## 1.2. 反应堆芯

`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](#): 具有非阻塞 I/O 和响应流反压力的 HTTP 请求处理的基本协定，以及 Reactor Netty, Undertow, Tomcat, Jetty 和任何 Servlet 3.1 容器的适配器。
  - [WebHandler API](#): 用于请求处理的更高级别的通用 Web API，在此之上构建了具体的编程模型，例如带 Comments 的控制器和功能端点。
- 对于 Client 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[Jetty HttpClient](#)的适配器。应用程序中使用的较高级别[WebClient](#)以此基本 Contract 为基础。
- 对于 Client 端和服务器，[codecs](#)用于对 HTTP 请求和响应内容进行序列化和反序列化。

### 1.2.1. HttpHandler

[HttpHandler](#)是具有单个方法的简单 Contract，用于处理请求和响应。它是故意最小的，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	使用的服务器 API	反应式流支持
Netty	Netty API	<a href="#">Reactor Netty</a>
Undertow	Undertow API	spring-web: 向响应流 bridge 过渡
Tomcat	Servlet 3.1 非阻塞 I/O; Tomcat API 读写 ByteBuffers 与 byte []	spring-web: Servlet 3.1 非阻塞 I/O 到响应流 bridge
Jetty	Servlet 3.1 非阻塞 I/O; Jetty API 编写 ByteBuffers 与 byte []	spring-web: Servlet 3.1 非阻塞 I/O 到响应流 bridge

Server name	使用的服务器 API	反应式流支持
Servlet 3.1 容器	Servlet 3.1 非阻塞 I/O	spring-web: Servlet 3.1 非阻塞 I/O 到响应流 bridge

下表描述了服务器依赖性(另请参见[supported versions](#)):

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
Jetty	org.eclipse.jetty	jetty-server, jetty-servlet

下面的代码段显示了在每个服务器 API 中使用 `HttpHandler` 适配器:

## Reactor Netty

```
HttpHandler handler = ...
ReactorHttpHandlerAdapter adapter = new ReactorHttpHandlerAdapter(handler);
HttpServer.create(host, port).newHandler(adapter).block();
```

## 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 容器

要作为 WAR 部署到任何 Servlet 3.1 容器，您可以扩展 [AbstractReactiveWebInitializer](#) 并将其包含在 WAR 中。该类用 `ServletHttpHandlerAdapter` 包装 `HttpHandler` 并将其注册为 `Servlet`。

### 1.2.2. WebHandler API

`org.springframework.web.server` 包构建在 [HttpHandlerContracts](#) 的基础上，以提供通用 Web API，以通过多个 [WebExceptionHandler](#)、多个 [WebFilter](#) 和单个 [WebHandler](#) 组件链处理请求。可以通过简单地指向组件为 [auto-detected](#) 的 Spring `ApplicationContext` 和/或通过向构建器注册组件来将链与 `WebHttpHandlerBuilder` 放在一起。

`HttpHandler` 的抽象目标很简单，而 `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。
- Request attributes.
- 解决了 `Locale` 或 `Principal` 的请求。
- 访问已解析和缓存的表单数据。
- Multipart 数据的抽象。
- and more..

## 特殊 bean 类型

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>	<code>WebExceptionHandler</code>	0..N	提供对 WebHandler 的详细信息
<any>	<code>WebFilter</code>	0..N	在其余和之后的信息，
<code>webHandler</code>	<code>WebHandler</code>	1	请求的
<code>webSessionManager</code>	<code>WebSessionManager</code>	0..1	通过 Session 的 WebSessionManager

Bean name	Bean type	Count	Description
			器。 I
			情况下
			用于访
			析表单
serverCodecConfigurer	ServerCodecConfigurer	0..1	Server 数据。
			Server
			认情况
			Local
localeContextResolver	LocaleContextResolver	0..1	Server Accept
			默认情
			对于处
forwardedHeaderTransformer	ForwardedHeaderTransformer	0..1	提取和 。默认

## Form Data

`ServerWebExchange` 公开了以下访问表单数据的方法：

```
Mono<MultiValueMap<String, String>> formData();
```

`DefaultServerWebExchange` 使用配置的 `HttpMessageReader` 将表单数据(`application/x-www-form-urlencoded`)解析为 `MultiValueMap`。缺省情况下，`FormHttpMessageReader` 配

置为由 `ServerCodecConfigurer` bean 使用(请参见[Web 处理程序 API](#))。

## Multipart Data

[与 Spring MVC 中的相同](#)

`ServerWebExchange` 公开了以下访问 Multipart 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`。目前, [Synchronous NIO](#)[Multipart](#) 是唯一受支持的第三方库, 也是我们知道的用于非阻塞解析 Multipart 请求的唯一库。通过 `ServerCodecConfigurer` bean(请参阅[Web 处理程序 API](#))启用了它。

要以流方式解析 Multipart 数据, 可以改用 `HttpMessageReader<Part>` 返回的 `Flux<Part>`。

例如, 在带 `Comments` 的控制器中, 使用 `@RequestPart` 意味着按名称对单个部分进行 `Map` 类访问, 因此需要完整地解析 Multipart 数据。相反, 您可以使用 `@RequestBody` 将内容解码为 `Flux<Part>` 而不收集为 `MultiValueMap`。

## Forwarded Headers

[与 Spring MVC 中的相同](#)

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, 从 Client 端的角度来看, 要创建指向正确的主机, 端口和方案的链接是一个挑战。

[RFC 7239](#) 定义 `Forwarded` HTTPHeaders, 代理可用来提供有关原始请求的信息。还有其他非标准 Headers, 包括 `X-Forwarded-Host`, `X-Forwarded-Port`, `X-Forwarded-Proto`, `X-Forwarded-Ssl` 和 `X-Forwarded-Prefix`。

`ForwardedHeaderTransformer` 是一个组件, 可根据转发的 Headers 修改请求的主机, 端口和方

案，然后删除这些 Headers。您可以将其声明为名称为 `forwardedHeaderTransformer` 的 bean，并且它是 [detected](#) 并且已使用。

对于转发的 Headers，存在安全方面的考虑，因为应用程序无法知道 Headers 是由代理添加的，还是由恶意 Client 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流量的原因。您也可以使用 `removeOnly=true` 配置 `ForwardedHeaderTransformer`，在这种情况下，它会删除但不使用标题。

#### 1 Note

在 5.1 版中，`ForwardedHeaderFilter` 已弃用并被 `ForwardedHeaderTransformer` 取代，因此可以在创建交换之前更早处理转发的 Headers。如果仍然配置了过滤器，则将其从过滤器列表中删除，并使用 `ForwardedHeaderTransformer` 代替。

### 1.2.3. Filters

#### 与 Spring MVC 中的相同

在 [WebHandler API](#) 中，您可以使用 `WebFilter` 在其余过滤器处理链和目标 `WebHandler` 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

#### 与 Spring MVC 中的相同

Spring WebFlux 通过控制器上的 Comments 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`，该参数必须在 Spring Security 的过滤器链之前 Order。

有关更多详细信息，请参见 [CORS](#) 和 [CORS WebFilter](#) 部分。

## 1.2.4. Exceptions

[与 Spring MVC 中的相同](#)

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
<code>ResponseStatusExceptionHandler</code>	通过将响应设置为异常的 HTTP 状态代码，提供对 <a href="#">ResponseStatusException</a> 类型的异常的处理。
<code>WebFluxResponseStatusExceptionHandler</code>	<code>ResponseStatusExceptionHandler</code> extensions，也可以确定任何异常上的 <code>@ResponseStatus</code> 注解的 HTTP 状态代码。

该处理程序在[WebFlux Config](#)中声明。

## 1.2.5. Codecs

[与 Spring MVC 中的相同](#)

`spring-web` 和 `spring-core` 模块提供了对通过非阻塞 I/O(具有 Reactive Streams 背压)在高级对象之间来回串行化和反序列化字节内容的支持。以下介绍了此支持：

- [Encoder](#) 和 [Decoder](#) 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。
- [Encoder](#) 可以用 [EncoderHttpMessageWriter](#) 包裹以使其适合在 Web 应用程序中使用，而 [Decoder](#) 可以用 [DecoderHttpMessageReader](#) 包裹。
- [DataBuffer](#) 提取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 Core”部分中的[数据缓冲区和编解码器](#)。

[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 Jackson JSON，Jackson Smile，JAXB2，Protocol Buffers 和其他编码器和解码器，以及用于表单数据，Multipart 内容，服务器发送的事件以及其他内容的纯 Web HTTP 消息读取器和写入器实现。

[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](#)，每个\_代表一个 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，只要为完整格式的对象接收到足够的字节，每个

`TokenBuffer` 就会传递给 `ObjectMapper`。Importing 的内容可以是 JSON 数组，如果 Content Type 为 “application/stream json”，则为 [line-delimited JSON](#)。

`Jackson2Encoder` 的工作方式如下：

- 对于单个价值发布者(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 [line-delimited JSON](#) 格式分别对每个值进行编码，写入和刷新。
- 对于 SSE，每个事件都调用 `Jackson2Encoder`，并且刷新输出以确保传递时没有延迟。

### iNote

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>` 渲染 JSON 数组，请使用 `Flux#collectToList()` 并编码 `Mono<List<String>>`。

## 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 [Form Data](#)。

使用 `getFormData()` 后，将无法再从请求正文中读取原始原始内容。因此，与从原始请求主体读取数据相比，应用程序应始终通过 `ServerWebExchange` 访问缓存的表单数据。

## Multipart

`MultipartHttpMessageReader` 和 `MultipartHttpMessageWriter` 支持对“Multipart/表单数据”内容进行解码和编码。依次将 `MultipartHttpMessageReader` 委派给另一个 `HttpMessageReader` 以便实际解析为 `Flux<Part>`，然后将这些部分简单地收集到 `MultiValueMap` 中。目前，[Synchronous NIO Multipart](#) 用于实际解析。

在可能需要从多个位置访问 Multipart 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[Web Handler API](#) 部分中的 [Multipart Data](#)。

使用 `getMultipartData()` 后，将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 Map 地访问 Component，否则必须依靠 `SynchronousPartHttpMessageReader` 来一次性访问 `Flux<Part>`。

## Streaming

[与 Spring MVC 中的相同](#)

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，以便尽早而不是稍后可靠地检测到断开连接的 Client 端。这样的发送可以是仅 Comments 的空 SSE 事件，也可以是有效用作心跳的任何其他“无操作”数据。

## DataBuffer

`DataBuffer` 是 WebFlux 中字节缓冲区的表示形式。参考的 Spring Core 部分在[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引用计数，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 应用程序通常无需关心此类问题，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[Using DataBuffer](#)部分。

## 1.2.6. Logging

### [与 Spring MVC 中的相同](#)

Spring WebFlux 中的 DEBUG 级别日志记录旨在紧凑，最小化并且对用户友好。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 相同的原理(例如，也不应成为 firehose)，但可用于调试任何问题。此外，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

`ServerWebExchange#getLogPrefix()` 提供了基于该 ID 的完全格式化的前缀。在 `WebClient`

端，日志 ID 存储在 `ClientRequest` 属性([LOG\\_ID\\_ATTRIBUTE](#))中，而

`ClientRequest#logPrefix()` 提供了完整格式的前缀。

### Sensitive Data

### [与 Spring MVC 中的相同](#)

`DEBUG` 和 `TRACE` 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针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 Client 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();
```

## 1.3. DispatcherHandler

[与 Spring MVC 中的相同](#)

与 Spring MVC 类似，Spring WebFlux 围绕前端控制器模式进行设计，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](#) 以访问其运行的上下文。如果以 [webHandler](#) 的 bean 名称声明了 [DispatcherHandler](#)，则依次由[WebHttpHandlerBuilder](#)发现，该\_组合了一个请求处理链，如[WebHandler API](#)中所述。

WebFlux 应用程序中的 Spring 配置通常包含：

- [DispatcherHandler](#)，名称为 [webHandler](#)
- [WebFilter](#) 和 [WebExceptionHandler](#) bean
- [DispatcherHandler 特殊 bean](#)

- Others

配置已分配给 `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` 准备与 [server adapter](#) 一起使用。

### 1.3.1. 特殊 bean 类

[与 Spring MVC 中的相同](#)

`DispatcherHandler` 委托特殊 bean 处理请求并呈现适当的响应。所谓“特殊 bean”，是指实现 `WebFlux` 框架 Contract 的 `SpringManagement` 的 `Object` 实例。这些通常带有内置 Contract，但是您可以自定义它们的属性，扩展它们或替换它们。

下表列出了 `DispatcherHandler` 检测到的特殊 bean。请注意，在较低级别还检测到其他一些 Bean(请参阅 Web Handler API 中的 [特殊 bean 类](#))。

Bean type	Explanation
<code>HandlerMapping</code>	将请求 Map 到处理程序。Map 基于某些条件，具体细节因 <code>HandlerMapping</code> 实现方式(带 Comments 的控制器，简单的 URL 模式 Map 等)而异。

`HandlerMapping` 主要实现是\_用于 `@RequestMapping` 带 Comments 的方法，`RouterFunctionMapping` 用于功能性端点路由，`SimpleUrlHandlerMapping` 用于 URI 路径模式和 `WebHandler` 实例的显式注册。  
| `HandlerAdapter` | 帮助 `DispatcherHandler` 调用 Map 到请求的处理程序，而不管该处理程

序的实际调用方式如何。例如，调用带 Comments 的控制器需要解析 Comments。

`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。参见[Result Handling](#)。

### 1.3.2. WebFlux 配置

[与 Spring MVC 中的相同](#)

应用程序可以声明处理请求所需的基础结构 bean(在[Web 处理程序 API](#)和[DispatcherHandler](#)下列出)。但是，在大多数情况下，[WebFlux Config](#)是最佳起点。它声明了所需的 bean，并提供了更高级别的配置回调 API 来对其进行自定义。

#### ①Note

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

[与 Spring MVC 中的相同](#)

`DispatcherHandler` 处理请求的方式如下：

- 要求每个 `HandlerMapping` 查找匹配的处理程序，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，该处理程序将执行返回的值公开为 `HandlerResult`。
- 通过直接写入响应或使用视图进行渲染，将 `HandlerResult` 赋予适当的 `HandlerResultHandler` 以完成处理。

### 1.3.4. 结果处理

通过 `HandlerAdapter` 调用处理程序的返回值与 `HandlerResult` 一起包装为 `HandlerResult`，并附加到其他上下文中，并传递给要求支持它的第一个 `HandlerResultHandler`。下表显示了可用的 `HandlerResultHandler` 实现，所有实现均在 [WebFlux Config](#) 中声明：

结果处理程序类型	Return Values	Default Order
<code> ResponseEntityResultHandler</code>	<code> ResponseEntity</code> ，通常来自 <code>@Controller</code> 个实例。	0
<code> ServerResponseResultHandler</code>	<code> ServerResponse</code> ，通常来自功能端点。	0
<code> ResponseBodyResultHandler</code>	处理来自 <code>@ResponseBody</code> 个方法或 <code>@RestController</code> 个类的返回值。	100
<code> ViewResolutionResultHandler</code>	<code> CharSequence</code> ， <a href="#">View</a> , <a href="#">Model</a> , <a href="#">Map</a> ， <a href="#">Rendering</a> 或任何其他 <code>Object</code> 被视为模型属性。	
	另请参见 <a href="#">View Resolution</a> 。	

结果处理器类型	Return Values	Default Order
<code>Integer.MAX_VALUE</code>		

### 1.3.5. Exceptions

#### 与 Spring MVC 中的相同

从 `HandlerAdapter` 返回的 `HandlerResult` 可以公开基于某些特定于处理器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：

- 处理器(例如 `@Controller`)调用失败。
- 通过 `HandlerResultHandler` 处理处理器返回值失败。

只要在从处理器返回的反应类型产生任何数据项之前发生错误 signal，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 `HandlerExceptionResolver` 的支持基于此。这通常不重要。但是，请记住，在 WebFlux 中，不能使用 `@ControllerAdvice` 处理在选择处理器之前发生的异常。

另请参见“带 Comments 的控制器”部分中的[Managing Exceptions](#)或 WebHandler API 部分中的[Exceptions](#)。

### 1.3.6. 查看分辨率

#### 与 Spring MVC 中的相同

视图分辨率使您可以使用 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，通过使用 `ViewResolver` 实例将 String(代表逻辑视图名称)Map 到 `View` 实例的专用 `HandlerResultHandler` 支持视图解析。`View` 然后用于呈现响应。

## Handling

### 与 Spring MVC 中的相同

传递给 `ViewResolutionResultHandler` 的 `HandlerResult` 包含处理程序的返回值和包含请求处理过程中添加的属性的模型。返回值将作为以下值之一进行处理：

- `String` , `CharSequence` : 通过已配置的 `ViewResolver` 实现的列表解析为 `View` 的逻辑视图名称。
- `void` : 根据请求路径选择默认视图名称，减去前斜杠和后斜杠，然后将其解析为 `View`。当未提供视图名称(例如，返回了模型属性)或异步返回值(例如，`Mono` 已完成为空)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` : 要添加到请求模型的额外模型属性。
- 任何其他：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。除非存在处理程序方法 `@ModelAttribute` 注解，否则使用 conventions从类名称派生属性名称。

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提供用于视图分辨率的专用配置 API。

有关与 Spring WebFlux 集成的视图技术的更多信息，请参见[View Technologies](#)。

## Redirecting

### 与 Spring MVC 中的相同

视图名称中特殊的 `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` 或 `Rendering.redirectTo("abc").build()` 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。诸如 `redirect:/some/resource` 之类的视图名称是相对于当前应用程序的，而诸如 `redirect:http://example.com/arbitrary/path` 之类的视图名称则重定向到绝对 URL。

## Content Negotiation

### 与 Spring MVC 中的相同

`ViewResolutionResultHandler` 支持内容协商。它将请求媒体类型与每个选定的 `View` 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 `View`。

为了支持 JSON 和 XML 之类的媒体类型，Spring WebFlux 提供了 `HttpMessageWriterView`，这是一个特殊的 `View`，它通过 [HttpMessageWriter](#) 呈现。通常，您可以通过 [WebFlux Configuration](#) 将它们配置为默认视图。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。

## 1.4. 带 Comments 的控制器

### 与 Spring MVC 中的相同

Spring WebFlux 提供了一个基于 Comments 的编程模型，其中 `@Controller` 和 `@RestController` 组件使用 Comments 来表达请求 Map，请求 Importing，处理异常等。带 Comments 的控制器具有灵活的方法签名，无需扩展 Base Class 或实现特定的接口。

以下 Lists 显示了一个基本示例：

```
@RestController
public class HelloController {
```

```
    @GetMapping("/hello")
    public String handle() {
        return "Hello WebFlux";
    }
}
```

在前面的示例中，该方法返回要写入响应主体的 `String`。

### 1.4.1. @Controller

#### 与 Spring MVC 中的相同

您可以使用标准的 Spring bean 定义来定义控制器 bean。`@Controller` 原型允许自动检测，并且与 Spring 常规支持保持一致，以支持在 Classpath 中检测 `@Component` 类并为其自动注册 Bean 定义。它还充当带 `Comments` 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") (1)
public class WebConfig {

    // ...
}
```

- (1) 扫描 `org.example.web` 软件包。

`@RestController` 是本身由 `@Controller` 和 `@ResponseBody` 进行元 `Comments` 的 [composed annotation](#)，表示其每个方法都继承了类型级别 `@ResponseBody` `Comments` 的控制器，因此直接将其写入响应主体(与视图分辨率和 HTML 模板渲染相比)。

### 1.4.2. 请求 Map

#### 与 Spring MVC 中的相同

`@RequestMapping` 注解用于将请求 Map 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，Headers 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 Map，也可以在方法级别使用它来缩小到特定的端点 Map。

也有 `@RequestMapping` 的 HTTP 方法特定的快捷方式：

- `@GetMapping`
- `@PostMapping`
- `@PutMapping`
- `@DeleteMapping`
- `@PatchMapping`

提供前面的 Comments 是 [Custom Annotations](#)，因为可以说，大多数控制器方法应该 Map 到特定的 HTTP 方法，而不是使用 `@RequestMapping`，默认情况下，`@RequestMapping` 匹配所有 HTTP 方法。同时，在类级别仍需要 `@RequestMapping` 来表示共享 Map。

以下示例使用类型和方法级别的 Map：

```
@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

[与 Spring MVC 中的相同](#)

您可以使用全局模式和通配符来 Map 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}") (1)
public class OwnerController {

    @GetMapping("/pets/{petId}") (2)
    public Pet findPet(@PathVariable Long ownerId, @PathVariable Long petId) {
        // ...
    }
}
```

- **(1)** 类级 **URIMap**。
- **(2)** 方法级 **URIMap**。

**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 support。参见[Type Conversion](#)和[DataBinder](#)。

**URI** 变量可以显式命名(例如 `@PathVariable("customId")`)，但是如果名称相同，则可以省略该详细信息，并使用调试信息或 Java 8 上的 `-parameters` 编译器标志编译代码。

语法 `{ *varName }` 声明了一个与零个或多个剩余路径段匹配的 **URI** 变量。例如，

`/resources/{ *path }` 匹配所有文件 `/resources/`，并且 `"path"` 变量捕获完整的相对路径。

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`，以下方法将提取名称，版本和文件 extensions：

```
@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

`PropertyPlaceholderConfigurer` 会针对本地，系统，环境和其他属性源进行解析。您可以使用它来例如基于某些外部配置参数化基本 URL。

### 1 Note

Spring WebFlux 使用 `PathPattern` 和 `PathPatternParser` 来获得 URI 路径匹配支持。这两个类都位于 `spring-web` 中，并且专门设计用于 Web 应用程序中的 HTTP URL 路径，在 Web 应用程序中，在运行时会匹配大量 URI 路径模式。

Spring WebFlux 不支持后缀模式匹配-与 Spring MVC 不同，在 Spring MVC 中，诸如 `/person` 的 Map 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

### 与 Spring MVC 中的相同

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 `PathPattern.SPECIFICITY_COMPARATOR` 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, 则选择较长的模式。

## 消耗媒体类型

### 与 Spring MVC 中的相同

您可以根据请求的 `Content-Type` 缩小请求 Map, 如下例所示:

```
@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 `text/plain` 以外的任何 Content Type。

您可以在类级别声明共享的 `consumes` 属性。但是, 与大多数其他请求 Map 属性不同, 在类级别使用时, 方法级别的 `consumes` 属性会覆盖而不是扩展类级别的声明。

### Tip

`MediaType` 提供常用媒体类型的常量, 例如 `APPLICATION_JSON_VALUE` 和 `APPLICATION_XML_VALUE`。

## 可生产的媒体类型

### 与 Spring MVC 中的相同

您可以根据 `Accept` 请求 Headers 和控制器方法生成的 Content Type 列表来缩小请求 Map, 如下例所示:

```
@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 `text/plain` 以外的任何 Content Type。

### iNote

对于 JSONContent Type，即使[RFC7159](#)明确指出“未为此注册定义任何字符集参数”，也应指定 UTF-8 `charset`，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 Map 属性不同，在类级别使用时，方法级别的 `produces` 属性会覆盖而不是扩展类级别的声明。

### Tip

`MediaType` 提供常用媒体类型的常量，例如。`APPLICATION_JSON_UTF8_VALUE`，`APPLICATION_XML_VALUE`。

## 参数和标题

### 与 Spring MVC 中的相同

您可以根据查询参数条件来缩小请求 Map。您可以测试是否存在查询参数(`myParam`)，查询参数是否不存在(`!myParam`)或特定值(`myParam=myValue`)。以下示例测试具有值的参数：

```
@GetMapping(path = "/pets/{petId}", params = "myParam=myValue") (1)
public void findPet(@PathVariable String petId) {
    // ...
}
```

- (1) 检查 `myParam` 等于 `myValue`。

您还可以将其与请求 Headers 条件一起使用，如以下示例所示：

```
@GetMapping(path = "/pets", headers = "myHeader=myValue") (1)
```

```
public void findPet(@PathVariable String petId) {  
    // ...  
}
```

- (1) 检查 `myHeader` 等于 `myValue`。

## HTTP HEAD，选项

[与 Spring MVC 中的相同](#)

`@GetMapping` 和 `@RequestMapping(method=HttpMethod.GET)` 透明地支持 HTTP HEAD，以进行请求 Map。控制器方法无需更改。`HttpHandler` 服务器适配器中应用的响应包装器可确保将 `Content-Length` Headers 设置为写入的字节数，而无需实际写入响应。

默认情况下，通过将 `Allow` 响应 Headers 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`，`Allow` Headers 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，使用 HTTP 方法特定的变体 `@GetMapping`，`@PostMapping` 等)。

您可以将 `@RequestMapping` 方法显式 Map 到 HTTP HEAD 和 HTTP OPTIONS，但这在通常情况下不是必需的。

## Custom Annotations

[与 Spring MVC 中的相同](#)

Spring WebFlux 支持使用[composed annotations](#)进行请求 Map。这些注解本身用 `@RequestMapping` 进行元注解，并组成它们以更狭窄，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 Comments 的示例。之所以提供它们，是因为大多数控制器方法应该 Map 到特定的 HTTP 方法，而不是使用 `@RequestMapping`，默认情况下，`@RequestMapping` 匹配所有 HTTP 方法。如果需要组合 Comments 的示例，请查看如何声明它们。

Spring WebFlux 还支持具有自定义请求匹配逻辑的自定义请求 Map 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 `custom` 属性并返回自己的 `RequestCondition`。

## Explicit Registrations

### 与 Spring MVC 中的相同

您可以以编程方式注册 Handler 方法，这些方法可用于动态注册或高级用例，例如同一处理程序在不同 URL 下的不同实例。以下示例显示了如何执行此操作：

```
@Configuration
public class MyConfig {

    @Autowired
    public void setHandlerMapping(RequestMappingHandlerMapping mapping, UserHandler han
        throws NoSuchMethodException {

        RequestMappingInfo info = RequestMappingInfo
            .paths("/user/{id}").methods(RequestMethod.GET).build(); (2)

        Method method = UserHandler.class.getMethod("getUser", Long.class); (3)

        mapping.registerMapping(info, handler, method); (4)
    }
}
```

- (1) 注入目标处理程序和控制器的处理程序 Map。
- (2) 准备请求 Map 元数据。
- (3) 获取处理程序方法。
- (4) 添加注册。

### 1.4.3. 处理程序方法

### 与 Spring MVC 中的相同

`@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。

## Method Arguments

与 Spring MVC 中的相同

下表显示了受支持的控制器方法参数。

需要解析 I/O(例如，读取请求正文)的参数支持 Reactive 类型(Reactor, RxJava, [or other](#))。这在“描述”列中进行了标记。不需要阻塞的参数不应使用 Reactive 类型。

支持 JDK 1.8 的 `java.util.Optional` 作为方法参数，并与具有 `required` 属性(例如

`@RequestParam`，`@RequestHeader` 等)的注解结合使用，并且与 `required=false` 等效。

控制器方法参数	Description
<code>ServerWebExchange</code>	访问完整的 <code>ServerWebExchange</code> 容器，以获取 HTTP 请求和响应，请求和会话属性， <code>checkNotModified</code> 方法等。
<code>ServerHttpRequest</code> ， <code>ServerHttpResponse</code>	访问 HTTP 请求或响应。
<code>WebSession</code>	访问会话。除非添加了属性，否则这不会强制开始新的会话。支持反应类型。
<code>java.security.Principal</code>	当前经过身份验证的用户-可能是特定的 <code>Principal</code> 实现类(如果已知)。支持

控制器方法参数	Description
	反应类型。
<code>org.springframework.http.HttpMethod</code>	请求的 HTTP 方法。
<code>java.util.Locale</code>	当前的请求语言环境，实际上是由最具体的 <code>LocaleResolver</code> 确定的，即已配置的 <code>LocaleResolver</code> / <code>LocaleContextResolver</code> 。
<code>java.util.TimeZone</code> + <code>java.time.ZoneId</code>	与当前请求关联的时区，由 <code>LocaleContextResolver</code> 确定。
<code>@PathVariable</code>	用于访问 URI 模板变量。参见 <a href="#">URI Patterns</a> 。
<code>@MatrixVariable</code>	用于访问 URI 路径段中的名称/值对。参见 <a href="#">Matrix Variables</a> 。
<code>@RequestParam</code>	用于访问 Servlet 请求参数。参数值将转换为声明的方法参数类型。参见 <a href="#">@RequestParam</a> 。

请注意，使用 `@RequestParam` 是可选的，例如用于设置其属性。请参阅此表后面的“其他任何参数”。

- | `@RequestHeader` | 用于访问请求 Headers。Headers 值将转换为声明的方法参数类型。参见 [@RequestHeader](#)。
- | `@CookieValue` | 用于访问 cookie。Cookie 值将转换为声明的方法参数类型。参见 [@CookieValue](#)。
- | `@RequestBody` | 用于访问 HTTP 请求正文。正文内容通过使用 `HttpMessageReader` 实例转换为声明的方法参数类型。支持反应类型。参见 [@RequestBody](#)。
- | `HttpEntity<B>` | 用于访问请求 Headers 和正文。主体使用 `HttpMessageReader` 个实例进行转换。支持反应类型。参见 [HttpEntity](#)。
- | `@RequestPart` | 用于访问 `multipart/form-data` 请求中的 Component。支持反应类型。参见 [Multipart Content](#) 和 [Multipart Data](#)。
- | `java.util.Map` , `org.springframework.ui.Model` 和 `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暴露给模板。
- | `@ModelAttribute` | 用于访问应用了数据绑定和验证的模型中的现有属性(如果不存在，则进行实例化)。参见 [@ModelAttribute](#) 以及 [Model](#) 和 [DataBinder](#)。  
请注意，使用 `@ModelAttribute` 是可选的，例如用于设置其属性。请参阅此表后面的“其他任何参数”。
- | `Errors` , `BindingResult` | 用于访问验证和命令对象(即 `@ModelAttribute` 自变量)的数据绑定错误或 `@RequestBody` 或 `@RequestPart` 自变量验证的错误。必须在经过验证的方法参数后立即声明 `Errors` 或 `BindingResult` 参数。
- | `SessionStatus` 类级 `@SessionAttributes` | 用于标记表单处理完成，将触发清除通过类级 `@SessionAttributes` `Comments` 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，方案和路径的 URL。参见 [URI Links](#)。

| `@SessionAttribute` | 用于访问任何会话属性-与由于类级别 `@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](#)。

| 任何其他自变量|如果方法自变量与以上任何一个都不匹配, 则默认情况下, 如果它是由[BeanUtils#isSimpleProperty](#)确定的简单类型, 则默认解析为 `@RequestParam`; 否则, 解析为 `@ModelAttribute`。

## Return Values

[与 Spring MVC 中的相同](#)

下表显示了受支持的控制器方法返回值。请注意, 所有返回值通常都支持 Reactor, RxJava, [or other](#)之类的库中的反应类型。

控制器方法返回值	Description
<code>@ResponseBody</code>	返回值通过 <code>HttpMessageWriter</code> 个实例进行编码, 并写入响应中。参见 <a href="#">@ResponseBody</a> 。
<code>HttpEntity&lt;B&gt;</code> , <code> ResponseEntity&lt;B&gt;</code>	返回值指定完整的响应, 包括 <code>HTTPHeaders</code> , 并且正文通过 <code>HttpMessageWriter</code> 实例进行编码并写入响应中。参见 <a href="#"> ResponseEntity</a> 。
<code> HttpHeaders</code>	用于返回不包含标题的响应。
<code> String</code>	用 <code> ViewResolver</code> 实例解析并与隐式模型一起使用的视图名称(通过命令对象和

控制器方法返回值	Description
	<p><code>@ModelAttribute</code> 方法确定)。处理器方法还可以 pass 语句 <code>Model</code> 参数(描述为 <a href="#">earlier</a>)以编程方式丰富模型。</p>
<code>View</code>	<p>用于与隐式模型一起渲染的 <code>View</code> 实例-通过命令对象和 <code>@ModelAttribute</code> 方法确定。处理器方法还可以 pass 语句 <code>Model</code> 参数(描述为 <a href="#">earlier</a>)以编程方式丰富模型。</p>
<code>java.util.Map</code> , <code>org.springframework.ui.Model</code>	要添加到隐式模型的属性，视图名称根据请求路径隐式确定。
<code>@ModelAttribute</code>	要添加到模型的属性，视图名称根据请求路径隐式确定。

请注意，`@ModelAttribute` 是可选的。请参阅本表后面的“其他任何返回值”。

| `Rendering` | 用于模型和视图渲染方案的 API。

| `void` | 具有 `void`，可能是异步的方法(例如 `Mono<Void>`)，返回类型(或 `null` 返回值)的方法，如果它也具有 `ServerHttpResponse`，`ServerWebExchange` 参数或 `@ResponseStatus` `Comments`，则认为已完全处理了响应。如果控制器对 `ETag` 或 `lastModified` 时间戳进行了肯定检查，则也是如此。// TODO：有关详细信息，请参见[Controllers](#)。

如果以上所有条件都不成立，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。

| `Flux<ServerSentEvent>` , `Observable<ServerSentEvent>` 或其他响应式类型|发送服务器发送的事件。当仅需要写入数据时，可以省略 `ServerSentEvent` 包装器(但是，必须通过 `produces` 属性在 Map 中请求或声明 `text/event-stream` )。

|任何其他返回值|如果返回值与以上任何一个都不匹配，则默认情况下将其视为视图名称，如果是 `String` 或 `void` (适用默认视图名称选择)或模型属性除非它是简单的类型(如 [BeanUtils#isSimpleProperty](#) 所确定)，否则将其添加到模型中，在这种情况下，它仍未解析。

## Type Conversion

[与 Spring MVC 中的相同](#)

如果参数声明为 `String` 以外的其他内容，则表示基于字符串的请求 Importing 的某些带 Comments 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(例如 `int` , `long` , `Date` 等)。可以通过 `WebDataBinder` (请参见[\[mvc-ann-initbinder\]](#))或通过向 `FormattingConversionService` 注册 `Formatters` (请参见[Spring 字段格式](#))来自定义类型转换。

## Matrix Variables

[与 Spring MVC 中的相同](#)

[RFC 3986](#)讨论路径段中的名称/值对。在 Spring WebFlux 中，基于 Tim Berners-Lee 的["old post"](#)，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来指定多个值，例如 `"color=red;color=green;color=blue"` 。

与 Spring MVC 不同，在 WebFlux 中，URL 中是否存在矩阵变量不会影响请求 Map。换句话说，您不需要使用 URI 变量来屏蔽变量内容。就是说，如果要从控制器方法访问矩阵变量，则需要将 URI 变量添加到期望矩阵变量的路径段中。以下示例显示了如何执行此操作：

```
// GET /pets/42;q=11;r=22

@GetMapping("/pets/{petId}")
public void findPet(@PathVariable String petId, @MatrixVariable int q) {

    // petId == 42
    // q == 11
}
```

鉴于所有路径段都可以包含矩阵变量，因此有时可能需要消除矩阵变量应位于哪个路径变量的歧义，如以下示例所示：

```
// 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
}
```

您可以定义一个矩阵变量，可以将其定义为可选变量并指定一个默认值，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，请使用 [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]
}
```

## @RequestParam

[与 Spring MVC 中的相同](#)

您可以使用 `@RequestParam` 注解将查询参数绑定到控制器中的方法参数。以下代码段显示了用法：

```
@Controller
@RequestMapping("/pets")
public class EditPetForm {

    // ...

    @GetMapping
    public String setupForm(@RequestParam("petId") int petId, Model model) { (1)
        Pet pet = this.clinic.loadPet(petId);
        model.addAttribute("pet", pet);
        return "petForm";
    }

    // ...
}
```

- (1) 使用 `@RequestParam`。

### Tip

Servlet API 的“请求参数”概念将查询参数，表单数据和 Multipart 合并为一个。但是，在 WebFlux 中，每个对象都是通过 `ServerWebExchange` 单独访问的。虽然 `@RequestParam` 仅绑定到查询参数，但是您可以使用数据绑定将查询参数，表单数据和 Multipart 应用于 [command object](#)。

默认情况下，使用 `@RequestParam` `Comments` 的方法参数是必需的，但是您可以通过将

`@RequestParam` 的必需标志设置为 `false` 或通过 `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 `String`，则会自动应用类型转换。参见[\[mvc-ann-typeconversion\]](#)。

在 `Map<String, String>` 或 `MultiValueMap<String, String>` 参数上声明 `@RequestParam` `Comments` 时，将使用所有查询参数填充 Map。

请注意，使用 `@RequestParam` 是可选的，例如用于设置其属性。默认情况下，任何简单值类型(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已用 `@RequestParam` `Comments`。

## `@RequestHeader`

[与 Spring MVC 中的相同](#)

您可以使用 `@RequestHeader` 注解将请求 Headers 绑定到控制器中的方法参数。

以下示例显示了带有 Headers 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

下面的示例获取 `Accept-Encoding` 和 `Keep-Alive` Headers 的值：

```
@GetMapping("/demo")
public void handle(
    @RequestHeader("Accept-Encoding") String encoding, (1)
    @RequestHeader("Keep-Alive") long keepAlive) { (2)
    //...
}
```

- (1) 获取 `Accept-Encoding` Headers 的值。
- (2) 获取 `Keep-Alive` Headers 的值。

如果目标方法参数类型不是 `String`，则会自动应用类型转换。参见[\[mvc-ann-typeconversion\]](#)。

在 `Map<String, String>` , `MultiValueMap<String, String>` 或 `HttpHeaders` 参数上使用 `@RequestHeader` Comments 时，将使用所有 Headers 值填充 Map。

### Tip

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，带有 `@RequestHeader("Accept")` Comments 的方法参数可以是 `String` 类型，也可以是 `String[]` 或 `List<String>` 类型。

## @CookieValue

[与 Spring MVC 中的相同](#)

您可以使用 `@CookieValue` 注解将 HTTP cookie 的值绑定到控制器中的方法参数。

以下示例显示了一个带有 cookie 的请求：

```
JSESSIONID=415A4AC178C59DACE0B2C9CA727CDD84
```

下面的代码示例演示如何获取 cookie 值：

```
@GetMapping("/demo")
public void handle(@CookieValue("JSESSIONID") String cookie) { (1)
    //...
}
```

- (1) 获取 Cookie 值。

如果目标方法参数类型不是 `String`，则会自动应用类型转换。参见[\[mvc-ann-typeconversion\]](#)。

## @ModelAttribute

[与 Spring MVC 中的相同](#)

您可以在方法参数上使用 `@ModelAttribute` 注解来访问模型中的属性，或将其实例化(如果不存在)。 `model` 属性还覆盖了查询参数的值和名称与字段名称匹配的表单字段。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。下面的示例绑定 `Pet` 的实例：

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public String processSubmit(@ModelAttribute Pet pet) { } (1)
```

- (1) 绑定 `Pet` 的实例。

上例中的 `Pet` 实例按以下方式解析：

- 从模型(如果已通过[Model](#)添加)。
- 从 HTTP 会话通过[@SessionAttributes](#)。
- 从默认构造函数的调用开始。
- 从带有匹配合查询参数或表单字段的参数的“主要构造函数”的调用开始。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。

获取模型属性实例后，将应用数据绑定。`WebExchangeDataBinder` 类将查询参数和表单字段的名称与目标 `Object` 上的字段名称匹配。在必要时应用类型转换后，将填充匹配字段。有关数据绑定(和验证)的更多信息，请参见[Validation](#)。有关自定义数据绑定的更多信息，请参见[DataBinder](#)。

数据绑定可能会导致错误。默认情况下，引发 `WebExchangeBindException`，但是，要检查 controller 方法中的此类错误，可以在 `@ModelAttribute` 的紧后面添加 `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) { (1)
    if (result.hasErrors()) {
        return "petForm";
    }
    // ...
}
```

- (1) 添加 `BindingResult`。

您可以在数据绑定之后通过添加 `javax.validation.Valid` Comments 或 Spring 的

`@Validated` Comments 来自动应用验证(另请参见[Bean validation](#)和[Spring validation](#))。以下示例使用 `@Valid` 注解：

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public String processSubmit(@Valid @ModelAttribute("pet") Pet pet, BindingResult result)
    if (result.hasErrors()) {
        return "petForm";
    }
    // ...
}
```

- (1) 在模型属性参数上使用 `@Valid`。

与 Spring MVC 不同，Spring WebFlux 在模型 `supports` 中支持反应类型，例如 `Mono<Account>` 或 `io.reactivex.Single<Account>`。您可以声明带有或不带有 Reactive 类型包装器的 `@ModelAttribute` 参数，并将根据需要将其解析为实际值。但是，请注意，要使用 `BindingResult` 参数，您必须在 `@ModelAttribute` 参数之前声明它而不使用反应式类型包装器，如先前所示。另外，您可以通过反应式处理任何错误，如以下示例所示：

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public Mono<String> processSubmit(@Valid @ModelAttribute("pet") Mono<Pet> petMono) {
    return petMono
        .flatMap(pet -> {
            // ...
        })
        .onErrorResume(ex -> {
            // ...
        });
}
```

请注意，使用 `@ModelAttribute` 是可选的，例如用于设置其属性。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已用 `@ModelAttribute` Comments。

## **@SessionAttributes**

[与 Spring MVC 中的相同](#)

`@SessionAttributes` 用于在两次请求之间的 `WebSession` 中存储模型属性。它是类型级别的

**Comments**, 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名称或模型属性类型。

考虑以下示例：

```
@Controller  
@SessionAttributes("pet") (1)  
public class EditPetForm {  
    // ...  
}
```

- (1) 使用 `@SessionAttributes` Comments。

在第一个请求中，将名称为 `pet` 的模型属性添加到模型后，该属性会自动提升为 `WebSession` 并保存在其中。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") (1)  
public class EditPetForm {  
  
    // ...  
  
    @PostMapping("/pets/{id}")  
    public String handle(Pet pet, BindingResult errors, SessionStatus status) { (2)  
        if (errors.hasErrors()) {  
            // ...  
        }  
        status.setComplete();  
        // ...  
    }  
}
```

- (1) 使用 `@SessionAttributes` Comments。

- (2) 使用 `SessionStatus` 变量。

## **@SessionAttribute**

[与 Spring MVC 中的相同](#)

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)Management)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`。Comments, 作为以下示例显示：

```
@GetMapping("/")
public String handle(@SessionAttribute User user) { (1)
    // ...
}
```

- (1) 使用 `@SessionAttribute`。

对于需要添加或删除会话属性的用例，请考虑将 `WebSession` 注入控制器方法。

要将模型属性临时存储在会话中作为控制器工作流的一部分，请考虑使用 `SessionAttributes`，如 [@SessionAttributes](#) 中所述。

## @RequestAttribute

[与 Spring MVC 中的相同](#)

与 `@SessionAttribute` 相似，您可以使用 `@RequestAttribute` 注解来访问先前创建的预先存在的请求属性(例如 `WebFilter`)，如以下示例所示：

```
@GetMapping("/")
public String handle(@RequestAttribute Client client) { (1)
    // ...
}
```

- (1) 使用 `@RequestAttribute`。

## Multipart Content

[与 Spring MVC 中的相同](#)

如[Multipart Data](#)中所述，`ServerWebExchange` 提供对 Multipart 内容的访问。在控制器中处理文件上传表单(例如，从浏览器)的最佳方法是通过将数据绑定到[command object](#)，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) {
        // ...
    }

}

```

您还可以在 RESTful 服务方案中从非浏览器 Client 端提交 Multipart 请求。以下示例将文件与 JSON 一起使用：

```

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 `@RequestPart` 来访问各个部分，如以下示例所示：

```

@PostMapping("/")
public String handle(@RequestPart("meta-data") Part metadata, (1)
                     @RequestPart("file-data") FilePart file) { (2)
    // ...
}

```

- (1) 使用 `@RequestPart` 获取元数据。
- (2) 使用 `@RequestPart` 获取文件。

要反序列化原始 Component 的内容(例如，类似于 `@RequestBody` 的 JSON)，可以声明一个具体的目标 `Object`，而不是 `Part`，如以下示例所示：

```
@PostMapping( "/ " )
public String handle(@RequestPart("meta-data") MetaData metadata) { (1)
    // ...
}
```

- (1) 使用 `@RequestPart` 获取元数据。

您可以将 `@RequestPart` 与 `javax.validation.Valid` 或 Spring 的 `@Validated` 注解结合使用，这将导致应用标准 Bean 验证。默认情况下，验证错误会导致 `WebExchangeBindException`，它变成 400(`BAD_REQUEST`)响应。或者，您可以通过 `Errors` 或 `BindingResult` 参数在控制器内部本地处理验证错误，如以下示例所示：

```
@PostMapping( "/ " )
public String handle(@Valid @RequestPart("meta-data") MetaData metadata, (1)
    BindingResult result) { (2)
    // ...
}
```

- (1) 使用 `@Valid` Comments。
- (2) 使用 `BindingResult` 参数。

要以 `MultivalueMap` 的形式访问所有 Multipart 数据，可以使用 `@RequestBody`，如以下示例所示：

```
@PostMapping( "/ " )
public String handle(@RequestBody Mono<MultiValueMap<String, Part>> parts) { (1)
    // ...
}
```

- (1) 使用 `@RequestBody`。

要以流方式 Sequences 访问 Multipart 数据，可以将 `@RequestBody` 与 `Flux<Part>` 结合使用，如以下示例所示：

```
@PostMapping( "/")
public String handle(@RequestBody Flux<Part> parts) { (1)
    // ...
}
```

- (1) 使用 `@RequestBody`。

## @RequestBody

### 与 Spring MVC 中的相同

您可以使用 `@RequestBody` 注解将请求正文读取并通过 [HttpMessageReader](#) 反序列化为 `Object`

- 。以下示例使用 `@RequestBody` 参数：

```
@PostMapping( "/accounts")
public void handle(@RequestBody Account account) {
    // ...
}
```

与 Spring MVC 不同，在 WebFlux 中，`@RequestBody` 方法参数支持响应类型以及完全无阻塞的读取和(Client 端到服务器)流。以下示例使用 `Mono`：

```
@PostMapping( "/accounts")
public void handle(@RequestBody Mono<Account> account) {
    // ...
}
```

您可以使用 [WebFlux Config](#) 的 [HTTP 消息编解码器](#) 选项来配置或自定义消息阅读器。

您可以将 `@RequestBody` 与 `javax.validation.Valid` 或 Spring 的 `@Validated` 注解结合使用，这将导致应用标准 Bean 验证。默认情况下，验证错误会导致 `WebExchangeBindException`，它变成 400(`BAD_REQUEST`)响应。或者，您可以通过 `Errors` 或 `BindingResult` 参数在控制器内本地处理验证错误。以下示例使用 `BindingResult` 参数：

```
@PostMapping( "/accounts")
public void handle(@Valid @RequestBody Account account, BindingResult result) {
    // ...
}
```

```
}
```

## HttpEntity

### 与 Spring MVC 中的相同

`HttpEntity` 与使用 `@RequestBody` 大致相同，但基于一个容器对象，该对象公开了请求 Headers 和正文。以下示例使用 `HttpEntity`：

```
@PostMapping("/accounts")
public void handle(HttpEntity<Account> entity) {
    // ...
}
```

## @ResponseBody

### 与 Spring MVC 中的相同

您可以在方法上使用 `@ResponseBody` 注解，以使返回值通过 `HttpMessageWriter` 序列化到响应主体。以下示例显示了如何执行此操作：

```
@GetMapping("/accounts/{id}")
@ResponseBody
public Account handle() {
    // ...
}
```

`@ResponseBody` 在类级别上也受支持，在这种情况下，它被所有控制器方法继承。这就是 `@RestController` 的效果，它不过是用 `@Controller` 和 `@ResponseBody` 标记的元 Comments。

`@ResponseBody` 支持反应式类型，这意味着您可以返回 Reactor 或 RxJava 类型，并将它们产生的异步值呈现给响应。有关更多详细信息，请参见 [Streaming](#) 和 [JSON rendering](#)。

您可以将 `@ResponseBody` 方法与 JSON 序列化视图结合使用。有关详情，请参见 [Jackson JSON](#)。

您可以使用 [WebFlux Config](#) 的 [HTTP 消息编解码器](#) 选项来配置或自定义消息编写。

## ResponseEntity

[与 Spring MVC 中的相同](#)

`ResponseEntity` 类似于 [@ResponseBody](#)，但具有状态和标题。例如：

```
@GetMapping("/something")
public ResponseEntity<String> handle() {
    String body = ... ;
    String etag = ... ;
    return ResponseEntity.ok().eTag(etag).build(body);
}
```

WebFlux 支持使用单个值 [reactive type](#) 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。

### Jackson 序列化视图

[与 Spring MVC 中的相同](#)

Spring WebFlux 提供对 [Jackson 的序列化视图](#) 的内置支持，该支持仅渲染  `Object` 中所有字段的一部分。要将它与  `@ResponseBody` 或  `ResponseEntity` 控制器方法一起使用，可以使用 Jackson 的  `@JsonView` 注解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;
}
}

```

### iNote

`@JsonView` 允许一组视图类，但每个控制器方法只能指定一个。如果需要激活多个视图，  
请使用复合界面。

## 1.4.4. Model

### 与 Spring MVC 中的相同

您可以使用 `@ModelAttribute` 注解：

- 在 `method argument` in `@RequestMapping` 方法上，可以从模型创建或访问对象，并通过 `WebDataBinder` 将其绑定到请求。
- 作为 `@Controller` 或 `@ControllerAdvice` 类中的方法级 Comments，有助于在任何 `@RequestMapping` 方法调用之前初始化模型。
- 在 `@RequestMapping` 方法上将其返回值标记为模型属性。

本节讨论 `@ModelAttribute` 方法，或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。

`@ModelAttribute` 方法也可以通过 `@ControllerAdvice` 在控制器之间共享。有关更多详细信息，[请参见 Controller Advice 部分](#)。

`@ModelAttribute` 方法具有灵活的方法签名。它们支持许多与 `@RequestMapping` 方法相同的参数(`@ModelAttribute` 本身以及与请求主体相关的任何东西除外)。

下面的示例使用 `@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);  
}
```

### iNote

如果未明确指定名称，则根据类型选择默认名称，如[Conventions](#)的 javadoc 中所述。您始终可以使用重载的 `addAttribute` 方法或通过 `@ModelAttribute` 上的 `name` 属性(用于返回值)来分配显式名称。

与 Spring MVC 不同，Spring WebFlux 在模型中显式支持响应类型(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，前提是声明了不带包装的 `@ModelAttribute` 参数，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) {
    // ...
}
```

此外，任何具有 **Reactive** 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` 方法的方法级 **Comments**，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，因为这是 **HTML** 控制器中的默认行为，除非返回值是 `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;
}
```

## 1.4.5. DataBinder

[与 Spring MVC 中的相同](#)

`@Controller` 或 `@ControllerAdvice` 类可以具有 `@InitBinder` 方法，以初始化 `WebDataBinder` 的实例。这些依次用于：

- 将请求参数(即表单数据或查询)绑定到模型对象。
- 将基于 `String` 的请求值(例如请求参数，路径变量，Headers，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` 个方法可以注册特定于控制器的 `java.bean.PropertyEditor` 或 Spring

`Converter` 和 `Formatter` 组件。此外，您可以使用[WebFlux Java 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，但 `@ModelAttribute` (命令对象)参数除外。通常，它们使用 `WebDataBinder` 参数声明(用于注册)和 `void` 返回值。以下示例使用 `@InitBinder` 注解：

```
@Controller
public class FormController {

    @InitBinder (1)
    public void initBinder(WebDataBinder binder) {
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        dateFormat.setLenient(false);
        binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, false));
    }

    // ...
}
```

- (1) 使用 `@InitBinder` Comments。

或者，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` 实例，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)
    }

    // ...
}
```

- (1) 添加自定义格式程序(在本例中为 `DateFormatter` )。

## 1.4.6. Management 异常

## 与 Spring MVC 中的相同

`@Controller` 和 `@ControllerAdvice` 类可以具有 `@ExceptionHandler` 个方法来处理控制器方法中的异常。下面的示例包括这样的处理程序方法：

```
@Controller
public class SimpleController {

    // ...

    @ExceptionHandler (1)
    public ResponseEntity<String> handle(IOException ex) {
        // ...
    }
}
```

- (1) 声明 `@ExceptionHandler`：

该异常可以与正在传播的顶级异常(即直接抛出 `IOException`)匹配，也可以与顶级包装程序异常中的直接原因匹配(例如，将 `IOException` 包裹在 `IllegalStateException` 内)。

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，`Comments` 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，并在优先级为 `@ControllerAdvice` 的情况下以相应的 `Sequences` 声明您的主根异常 Map。有关详情，请参见 [MVC 部分](#)。

### **i**Note

WebFlux 中的 `@ExceptionHandler` 方法与 `@RequestMapping` 方法支持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` for `@RequestMapping` 方法提供了对 Spring WebFlux 中 `@ExceptionHandler` 方法的支持。有关更多详细信息，请参见 [DispatcherHandler](#)。

## REST API 异常

## 与 Spring MVC 中的相同

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` 可以使用具有  `ResponseEntity` 返回值的 `@ExceptionHandler` 方法来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。

### **Note**

请注意，Spring WebFlux 与 Spring MVC  `ResponseEntityExceptionHandler` 不具有等效项，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. 控制器建议

### 与 Spring MVC 中的相同

通常，`@ExceptionHandler`，`@InitBinder` 和 `@ModelAttribute` 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[component scanning](#)将此类注册为 Spring Bean。`@RestControllerAdvice` 也是标有 `@ControllerAdvice` 和 `@ResponseBody` 的元 Comments，从本质上讲，这意味着 `@ExceptionHandler` 方法通过消息转换(与视图分辨率或模板渲染)呈现给响应主体。

启动时，`@RequestMapping` 和 `@ExceptionHandler` 方法的基础结构类将检测 `@ControllerAdvice` 类型的 Spring bean，并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\*\*本地方法之前被应用。

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 `Comments` 中的属性将其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 {}
```

前面的 `selectors` 会在运行时进行评估，如果广泛使用它们，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。

## 1.5. 功能端点

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 `Comments` 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. Overview

在 `WebFlux.fn` 中，HTTP 请求使用 `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 `Comments` 的编程模型中 `@RequestMapping` 方法的主体。

传入的请求会通过 `RouterFunction` 路由到处理函数，该函数接受 `ServerRequest` 并返回延迟的 `HandlerFunction` (即 `Mono<HandlerFunction>`)。当 Router 功能匹配时，返回处理程序功能。否则为空 `Mono`。`RouterFunction` 等效于 `@RequestMapping` `Comments`，但主要区别在于

Router 功能不仅提供数据，还提供行为。

`RouterFunctions.route()` 提供了一个 Router 构建器，可简化 Router 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` 并通过内置的[server adapters](#)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

`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。请求和响应都对体流提供 [Reactive Streams](#) 背压。请求主体用 Reactor `Flux` 或 `Mono` 表示。响应主体由任何响应流 `Publisher` 表示，包括 `Flux` 和 `Mono`。有关更多信息，请参见 [Reactive Libraries](#)。

## ServerRequest

`ServerRequest` 提供对 HTTP 方法，URI，Headers 和查询参数的访问，而通过 `body` 方法提供对正文的访问。

以下示例将请求正文提取到 `Mono<String>`：

```
Mono<String> string = request.bodyToMono(String.class);
```

以下示例将主体提取到 `Flux<Person>`，其中 `Person` 对象是从某种序列化形式(例如 JSON 或 XML)解码的：

```
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 `ServerRequest.body(BodyExtractor)` 接受 `BodyExtractor` 功能策略界面。Util 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 的形式访问 Multipart 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 `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带有 `Location` Headers 且不包含主体的 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"));
```

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，该类在与基于 Comments 的应用程序中的作用与 `@Controller` 类似。例如，以下类公开了一个响应式 `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) { (1)  
        Flux<Person> people = repository.allPeople();
```

```

        return ok().contentType(APPLICATION_JSON).body(people, Person.class);
    }

    public Mono<ServerResponse> createPerson(ServerRequest request) { (2)
        Mono<Person> person = request.bodyToMono(Person.class);
        return ok().build(repository.savePerson(person));
    }

    public Mono<ServerResponse> getPerson(ServerRequest request) { (3)
        int personId = Integer.valueOf(request.pathVariable("id"));
        return repository.getPerson(personId)
            .flatMap(person -> ok().contentType(APPLICATION_JSON).body(fromObject(person)
                .switchIfEmpty(ServerResponse.notFound().build()));
    }
}

```

- (1) `listPeople` 是一个处理函数，它以 JSON 格式返回存储库中找到的所有 `Person` 对象。
- (2) `createPerson` 是一个处理函数，用于存储请求正文中包含的新 `Person`。注意 `PersonRepository.savePerson(Person)` 返回 `Mono<Void>`：当从请求中读取并存储此人时，空 `Mono` 发出完成 signal。因此，当收到完成 signal 时(即，保存 `Person` 时)，我们使用 `build(Publisher<Void>)` 方法发送响应。
- (3) `getPerson` 是一个处理函数，它返回一个由 `id` path 变量标识的人。我们从存储库中检索该 `Person`，并创建一个 JSON 响应(如果找到)。如果找不到，我们使用 `switchIfEmpty(Mono<T>)` 返回 404 Not Found 响应。

### 1.5.3. RouterFunction

Router 功能用于将请求路由到相应的 `HandlerFunction`。通常，您不是自己编写 Router 功能，而是使用 `RouterFunctions` 实用工具类上的方法创建一个。`RouterFunctions.route()` (无参数)为您提供了流畅的生成器来创建 Router 功能，而 `RouterFunctions.route(RequestPredicate, HandlerFunction)` 提供了直接的方式来创建 Router。

通常，建议使用 `route()` 构建器，因为它为典型的 Map 方案提供了便捷的快捷方式，而无需发现静态导入。例如，Router 功能构建器提供了 `GET(String, HandlerFunction)` 方法来为 GET 请求

求创建 Map；和 `POST(String, HandlerFunction)` (用于 POST)。

除了基于 HTTP 方法的 Map 外，路由构建器还提供了一种在 Map 到请求时引入其他谓词的方法。

对于每个 HTTP 方法，都有一个以 `RequestPredicate` 作为参数的重载变体，但是可以表示其他约束。

## Predicates

您可以编写自己的 `RequestPredicate`，但是 `RequestPredicates` Util 类根据请求路径，HTTP 方法，Content Type 等提供常用的实现。以下示例使用请求谓词基于 `Accept Headers` 创建约束：

```
RouterFunction<ServerResponse> route = RouterFunctions.route()
    .GET("/hello-world", accept(MediaType.TEXT_PLAIN),
        request -> Response.ok().body(fromObject("Hello World")));
```

您可以使用以下方法将多个请求谓词组合在一起：

- `RequestPredicate.and(RequestPredicate)` 一两者都必须匹配。
- `RequestPredicate.or(RequestPredicate)` 一都可以匹配。

`RequestPredicates` 中的许多谓词组成。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 `accept` 谓词组合在一起。

## Routes

Router 功能按 Sequences 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 Comments 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 Router 功能生成器时，所有定义的路由都组成一个 `RouterFunction`，从

`RouterFunction` 返回。还有其他方法可以将多个 Router 功能组合在一起：

- `add(RouterFunction)` 在 `RouterFunctions.route()` 构建器上
- `RouterFunction.and(RouterFunction)`
- `RouterFunction.andRoute(RequestPredicate, HandlerFunction)` 一嵌套 `RouterFunctions.route()` 的 `RouterFunction.and()` 的快捷方式。

以下示例显示了四种 Route 的组成：

```
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.RouterFunction.*;
import static org.springframework.web.reactive.function.server.ServerResponse.*;

PersonRepository repository = ...
PersonHandler handler = new PersonHandler(repository);

RouterFunction<ServerResponse> otherRoute = ...

RouterFunction<ServerResponse> route = route()
    .GET("/person/{id}", accept(APPLICATION_JSON), handler::getPerson) (1)
    .GET("/person", accept(APPLICATION_JSON), handler::listPeople) (2)
    .POST("/person", handler::createPerson) (3)
    .add(otherRoute) (4)
    .build();
```

- (1) 具有与 JSON 匹配的 `Accept Headers` 的 `GET /person/{id}` 被路由到 `PersonHandler.getPerson`
- (2) 具有与 JSON 匹配的 `Accept Headers` 的 `GET /person` 被路由到 `PersonHandler.listPeople`
- (3) `POST /person` 没有其他谓词被 Map 到 `PersonHandler.createPerson`，并且
- (4) `otherRoute` 是在其他地方创建并添加到所构建路由的 Router 功能。

## Nested Routes

一组 Router 功能通常具有共享谓词，例如共享路径。在上面的示例中，共享谓词将是与其中三个路由使用的 `/person` 匹配的路径谓词。使用 `Comments` 时，您可以通过使用 Map 到 `/person` 的类型级别 `@RequestMapping` `Comments` 来删除此重复项。在 `WebFlux.fn` 中，可以通过

Router 功能构建器上的 `path` 方法共享路径谓词。例如，可以通过以下方式使用嵌套路由来改进上面示例的最后几行：

```
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，`path` 的第二个参数是使用 Router 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。上面的代码仍然包含一些共享的 `Accept` -header 谓词形式的重复项。通过结合使用 `nest` 和 `accept` 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. 运行服务器

如何在 HTTP 服务器中运行 Router 功能？一个简单的选项是使用以下方法之一将 Router 功能转换为 `HttpHandler`：

- `RouterFunctions.toHttpHandler(RouterFunction)`
- `RouterFunctions.toHttpHandler(RouterFunction, HandlerStrategies)`

然后，您可以按照[HttpHandler](#)中有关服务器特定的说明，将返回的 `HttpHandler` 与许多服务器适配器一起使用。

Spring Boot 也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，该设置使用 Spring 配置来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 `RouterFunction`。
- `HandlerFunctionAdapter`：简单的适配器，它使 `DispatcherHandler` 调用 Map 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 `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 Comments 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() {
        // ...
    }

    // ...

    @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. 过滤处理程序功能

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 `Comments`，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) (1)
                .header("X-RequestHeader", "Value")
                .build()))
        .POST("/person", handler::createPerson))
    .after((request, response) -> logResponse(response)) (2)
    .build();
```

- (1) 添加自定义请求 Headers 的 `before` 过滤器仅应用于两条 GET 路由。
- (2) 记录响应的 `after` 过滤器应用于所有路由，包括嵌套路由。

Router 构建器上的 `filter` 方法采用 `HandlerFilterFunction`：此函数采用 `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，假设我们有一个 `SecurityManager` 可以确定是否允许特定路径。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();
```

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 Router 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 Router 功能。

### iNote

通过专用的 [CorsWebFilter](#) 为功能端点提供 CORS 支持。

## 1.6. URI 链接

[与 Spring MVC 中的相同](#)

本节描述了 Spring 框架中用于准备 URI 的各种选项。

### 1.6.1. UriComponents

Spring MVC 和 Spring WebFlux

`UriComponentsBuilder` 帮助从具有变量的 URI 模板构建 URI，如以下示例所示：

```
UriComponents uriComponents = UriComponentsBuilder  
.fromUriString("http://example.com/hotels/{hotel}") (1)  
.queryParam("q", "{q}") (2)  
.encode() (3)  
.build(); (4)  
  
URI uri = uriComponents.expand("Westin", "123").toUri(); (5)
```

- (1) 带有 URI 模板的静态工厂方法。
- (2) 添加或替换 URI 组件。
- (3) 请求对 URI 模板和 URI 变量进行编码。
- (4) 构建 `UriComponents`。
- (5) 展开变量并获得 `URI`。

可以将前面的示例合并为一个链，并使用 `buildAndExpand` 进行缩短，如以下示例所示：

```
URI uri = UriComponentsBuilder
    .fromUriString("http://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("http://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("http://example.com/hotels/{hotel}?q={q}")
    .build("Westin", "123");
```

## 1.6.2. UriBuilder

Spring MVC 和 Spring WebFlux

`UriComponentsBuilder` 实现 `UriBuilder`。您可以依次创建 `UriBuilder` 和 `UriBuilderFactory`。`UriBuilderFactory` 和 `UriBuilder` 一起提供了一种可插入的机制，用于基于共享配置(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。

以下示例显示了如何配置 `RestTemplate`：

```
// import org.springframework.web.util.DefaultUriBuilderFactory.EncodingMode;

String baseUrl = "http://example.org";
DefaultUriBuilderFactory factory = new DefaultUriBuilderFactory(baseUrl);
factory.setEncodingMode(EncodingMode.TEMPLATE_AND_VARIABLES);

RestTemplate restTemplate = new RestTemplate();
restTemplate.setUriTemplateHandler(factory);
```

以下示例配置 `WebClient`：

```
// import org.springframework.web.util.DefaultUriBuilderFactory.EncodingMode;

String baseUrl = "http://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，而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 = "http://example.com";
DefaultUriBuilderFactory uriBuilderFactory = new DefaultUriBuilderFactory(baseUrl);

URI uri = uriBuilderFactory.uriString("/hotels/{hotel}")
    .queryParam("q", "{q}")
    .build("Westin", "123");
```

### 1.6.3. URI 编码

Spring MVC 和 Spring WebFlux

`UriComponentsBuilder` 公开了两个级别的编码选项：

- [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\(\)](#): 在扩展 URI 变量后\*编码 URI 组件。

这两个选项都使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。

### Tip

考虑 “;” ， 它在路径上是合法的，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：

```
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 = "http://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) 来预编码 URI 模板，并在扩展时严格编码 URI 变量。
- `VALUES_ONLY`：不对 URI 模板进行编码，而是在将它们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 变量之后\*，使用与较早列表中第二个选项相对应的 `UriComponents#encode()` 来编码 URI 组件值。
- `NONE`：未应用编码。

出于历史原因和向后兼容性，`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`。

## 1.7. CORS

[与 Spring MVC 中的相同](#)

Spring WebFlux 使您可以处理 CORS(跨源资源共享)。本节介绍如何执行此操作。

### 1.7.1. Introduction

## 与 Spring MVC 中的相同

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)是由[most browsers](#)实现的[W3C specification](#)，它使您可以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

#### 与 Spring MVC 中的相同

CORS 规范区分飞行前，简单和实际要求。要了解 CORS 的工作原理，您可以阅读[this article](#)，或者阅读规范以获得更多详细信息。

Spring WebFlux `HandlerMapping` 实现为 CORS 提供内置支持。成功将请求 Map 到处理程序后，`HandlerMapping` 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，验证并设置了所需的 CORS 响应 Headers。

为了启用跨域请求(即 `Origin` Headers 存在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，则飞行前请求将被拒绝。没有将 CORSHeaders 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。

每个 `HandlerMapping` 可以分别是[configured](#)，并具有基于 URL 模式的 `CorsConfiguration` Map。在大多数情况下，应用程序使用 WebFlux Java 配置声明此类 Map，从而导致将单个全局 Map 传递给所有 `HadlerMappping` 实现。

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 `Comments` 的控制器可以使用类或方法级别的 `@CrossOrigin` `Comments`(其他处理程序可以实现 `CorsConfigurationSource`)。

整体配置和局部配置的组合规则通常是相加的，例如，所有全局和所有本地来源。对于只能接受单个值的那些属性(例如 `allowCredentials` 和 `maxAge`)，局部变量将覆盖全局值。有关更多详细信息，请参见[CorsConfiguration#combine\(CorsConfiguration\)](#)。

### Tip

要从源中了解更多信息或进行高级自定义，请参阅：

- `CorsConfiguration`
- `CorsProcessor` 和 `DefaultCorsProcessor`
- `AbstractHandlerMapping`

### 1.7.3. @CrossOrigin

[与 Spring MVC 中的相同](#)

`@CrossOriginComments` 启用带 `Comments` 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 Mono<Account> retrieve(@PathVariable Long id) {
        // ...
    }

    @DeleteMapping("/{id}")
    public Mono<Void> remove(@PathVariable Long id) {
        // ...
    }
}
```

默认情况下，`@CrossOrigin` 允许：

- All origins.
- All headers.
- 控制器方法 Map 到的所有 HTTP 方法。

默认情况下未启用 `allowedCredentials`，因为它构建了一个信任级别，该级别公开了敏感的用户特定信息(例如 cookie 和 CSRF 令牌)，仅在适当的地方使用。

`maxAge` 设置为 30 分钟。

`@CrossOrigin` 在类级别也受支持，并且被所有方法继承。下面的示例指定一个特定的域并将

`maxAge` 设置为一个小时：

```
@CrossOrigin(origins = "http://domain2.com", maxAge = 3600)
@RestController
@RequestMapping("/account")
public class AccountController {

    @GetMapping("/{id}")
    public Mono<Account> retrieve(@PathVariable Long id) {
        // ...
    }

    @DeleteMapping("/{id}")
    public Mono<Void> remove(@PathVariable Long id) {
        // ...
    }
}
```

您可以在类和方法级别上都使用 `@CrossOrigin`，如以下示例所示：

```
@CrossOrigin(maxAge = 3600) (1)
@RestController
@RequestMapping("/account")
public class AccountController {

    @CrossOrigin("http://domain2.com") (2)
    @GetMapping("/{id}")
    public Mono<Account> retrieve(@PathVariable Long id) {
        // ...
    }

    @DeleteMapping("/{id}")
    public Mono<Void> remove(@PathVariable Long id) {
        // ...
    }
}
```

- (1) 在类使用 `@CrossOrigin`。
- (2) 在方法级别使用 `@CrossOrigin`。

## 1.7.4. 全局配置

[与 Spring MVC 中的相同](#)

除了细粒度的控制器方法级配置之外，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` Map。但是，大多数应用程序都使用 WebFlux Java 配置来执行此操作。

默认情况下，全局配置启用以下功能：

- All origins.
- All headers.
- `GET` , `HEAD` 和 `POST` 方法。

默认情况下未启用 `allowedCredentials`，因为它构建了一个信任级别，该级别公开了敏感的用户特定信息(例如 cookie 和 CSRF 令牌)，仅在适当的地方使用。

`maxAge` 设置为 30 分钟。

要在 WebFlux Java 配置中启用 CORS，可以使用 `CorsRegistry` 回调，如以下示例所示：

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {

        registry.addMapping("/api/**")
            .allowedOrigins("http://domain2.com")
            .allowedMethods("PUT", "DELETE")
            .allowedHeaders("header1", "header2", "header3")
            .exposedHeaders("header1", "header2")
            .allowCredentials(true).maxAge(3600);

        // Add more mappings...
    }
}
```

## 1.7.5. CORS WebFilter

## [与 Spring MVC 中的相同](#)

您可以通过内置的[CorsWebFilter](#)来应用 CORS 支持，这很适合[functional endpoints](#)。

要配置过滤器，可以声明一个 `CorsWebFilter` bean 并将 `CorsConfigurationSource` 传递给其构造函数，如以下示例所示：

```
@Bean
CorsWebFilter corsFilter() {

    CorsConfiguration config = new CorsConfiguration();

    // Possibly...
    // config.applyPermitDefaultValues()

    config.setAllowCredentials(true);
    config.addAllowedOrigin("http://domain1.com");
    config.addAllowedHeader("*");
    config.addAllowedMethod("*");

    UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration("/**", config);

    return new CorsWebFilter(source);
}
```

## 1.8. 网络安全

### [与 Spring MVC 中的相同](#)

[Spring Security](#) 项目为保护 Web 应用程序免受恶意攻击提供支持。请参阅 [Spring Security 参考文档](#)，包括：

- [WebFlux Security](#)
- [WebFlux 测试支持](#)
- [CSRF Protection](#)
- [安全响应 Headers](#)

## 1.9. 查看技术

### [与 Spring MVC 中的相同](#)

Spring WebFlux 中视图技术的使用是可插入的。是否决定使用 Thymeleaf, FreeMarker 或其他某种视图技术，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

#### [与 Spring MVC 中的相同](#)

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。Thymeleaf 提供了广泛的功能集，并且正在积极地开发和维护。有关更完整的介绍，请参见 [Thymeleaf 项目主页](#)。

Thymeleaf 与 Spring WebFlux 的集成由 Thymeleaf 项目 Management。该配置涉及一些 Bean 声明，例如 `SpringResourceTemplateResolver`，`SpringWebFluxTemplateEngine` 和 `ThymeleafReactiveViewResolver`。有关更多详细信息，请参阅 [Thymeleaf+Spring](#) 和 WebFlux 集成 [announcement](#)。

### 1.9.2. FreeMarker

#### [与 Spring MVC 中的相同](#)

[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 WebFlux 与 FreeMarker 模板一起使用。

#### **View Configuration**

#### [与 Spring MVC 中的相同](#)

以下示例显示了如何将 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("classpath:/templates");
        return configurer;
    }
}

```

您的模板需要存储在 `FreeMarkerConfigurer` 指定的目录中，如上例所示。给定上述配置，如果您的控制器返回视图名称 `welcome`，则解析器将查找

`classpath:/templates/freemarker/welcome.ftl` 模板。

## FreeMarker Configuration

### 与 Spring MVC 中的相同

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 `Configuration` 对象(由 `SpringManagement`)

- `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
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. 脚本视图

[与 Spring MVC 中的相同](#)

Spring 框架具有内置的集成，可以将 Spring WebFlux 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。下表显示了我们在不同脚本引擎上测试过的模板库：

Scripting Library	Scripting Engine
<a href="#">Handlebars</a>	<a href="#">Nashorn</a>
<a href="#">Mustache</a>	<a href="#">Nashorn</a>
<a href="#">React</a>	<a href="#">Nashorn</a>
<a href="#">EJS</a>	<a href="#">Nashorn</a>
<a href="#">ERB</a>	<a href="#">JRuby</a>
<a href="#">String templates</a>	<a href="#">Jython</a>
<a href="#">Kotlin 脚本模板</a>	<a href="#">Kotlin</a>

#### Tip

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

## 与 Spring MVC 中的相同

您需要在 Classpath 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。
- 应该添加 [JRuby](#) 作为 Ruby 支持的依赖项。
- 应该添加 [Jython](#) 作为对 Python 支持的依赖。
- 为了支持 Kotlin 脚本，应添加 `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 [this example](#)。

您需要具有脚本模板库。一种针对 Javascript 的方法是通过 [WebJars](#)。

## Script Templates

### 与 Spring MVC 中的相同

您可以声明一个 `ScriptTemplateConfigurer` bean，以指定要使用的脚本引擎，要加载的脚本文件，要调用的函数以渲染模板等等。以下示例使用 Mustache 模板和 Nashorn JavaScript 引擎：

```
@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("mustache.js");
        configurer.setRenderObject("Mustache");
        configurer.setRenderFunction("render");
        return configurer;
    }
}
```

`render` 函数使用以下参数调用：

- `String template`：模板内容
- `Map model`：视图模型
- `RenderingContext renderingContext`：用于访问应用程序上下文，语言环境，模板加载器和 URL 的 [RenderingContext](#)(自 5.0 开始)

`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，[Handlebars](#) 需要在使用模板之前先对其进行编译，并且需要 [polyfill](#) 才能模拟服务器端脚本引擎中不可用的某些浏览器功能。下面的示例演示如何设置自定义渲染功能：

```
@Configuration
@EnableWebMvc
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;
    }
}
```

### iNote

当使用非线程安全脚本引擎和非并发设计的模板库(例如在 Nashorn 上运行的 Handlebars 或 React)时，需要将 `sharedEngine` 属性设置为 `false`。在这种情况下，由于[this bug](#)，因此需要 Java 8u60 或更高版本。

`polyfill.js` 仅定义 Handlebars 正常运行所需的 `window` 对象，如以下代码片段所示：

```
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和重用缓存的模板或预编译的模板。这可以在脚本端以及您需要的任何自定义(例如，Management 模板引擎配置)上完成。以下示例显示了如何编译模板：

```
function render(template, model) {
    var compiledTemplate = Handlebars.compile(template);
    return compiledTemplate(model);
}
```

查看 Spring Framework 单元测试[Java](#)和[resources](#)，以获取更多配置示例。

## 1.9.4. JSON 和 XML

[与 Spring MVC 中的相同](#)

出于[Content Negotiation](#)的目的，根据 Client 端请求的 Content Type，能够在使用 HTML 模板呈现模型或使用其他格式(例如 JSON 或 XML)呈现模型之间进行切换非常有用。为此，Spring WebFlux 提供了 `HttpMessageWriterView`，您可以使用 `HttpMessageWriterView` 从 `spring-web` 插入任何可用的[Codecs](#)，例如 `Jackson2JsonEncoder`，`Jackson2SmileEncoder` 或 `Jaxb2XmlEncoder`。

与其他视图技术不同，`HttpMessageWriterView` 不需要 `ViewResolver`，而是将[configured](#)作为默认视图。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 Content Type 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，可以使用要渲染的模型属性的名称配置 `HttpMessageWriterView`。如果模型仅包含一个属性，则使用该属性。

## 1.10. HTTP 缓存

[与 Spring MVC 中的相同](#)

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 Headers 和后续的条件请求 Headers(例如 `Last-Modified` 和 `ETag` )展开。 `Cache-Control` 建议私有(例如浏览器)和公共(例如代理)缓存如何缓存和重复使用响应。 `ETag` Headers 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` Headers 的更复杂的后继者。

本节描述了 Spring WebFlux 中可用的 HTTP 缓存相关选项。

### 1.10.1. CacheControl

[与 Spring MVC 中的相同](#)

`CacheControl` 支持配置与 `Cache-Control` Headers 相关的设置, 并且在许多地方都作为参数接受:

- [Controllers](#)
- [Static Resources](#)

[RFC 7234](#) 描述了 `Cache-Control` 响应 Headers 的所有可能的指令, 而 `CacheControl` 类型采用面向用例的方法, 该方法着重于常见方案, 如以下示例所示:

```
// 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
```

### 1.10.2. Controllers

[与 Spring MVC 中的相同](#)

控制器可以添加对 HTTP 缓存的显式支持。我们建议这样做，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 Headers 进行比较。控制器可以将 `ETag`

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
        .body(book);
}
```

如果与条件请求 Headers 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` Headers 将添加到响应中。

您还可以在控制器中针对条件请求 Headers 进行检查，如以下示例所示：

```
@RequestMapping
public String myHandleMethod(ServerWebExchange exchange, Model model) {
    long eTag = ... (1)

    if (exchange.checkNotModified(eTag)) {
        return null; (2)
    }

    model.addAttribute(...); (3)
    return "myViewName";
}
```

- (1) 特定于应用程序的计算。
- (2) 响应已设置为 304(NOT\_MODIFIED)。无需进一步处理。
- (3) `continue` 进行请求处理。

可以使用三种变体来检查针对 `ETag` 值和 `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`，您可以改为将响应设置为 409(PRECONDITION\_FAILED)以防止并发修改。

### 1.10.3. 静态资源

[与 Spring MVC 中的相同](#)

您应该为静态资源提供 `Cache-Control` 和条件响应 Headers，以实现最佳性能。请参阅有关配置 [Static Resources](#) 的部分。

## 1.11. WebFlux 配置

[与 Spring MVC 中的相同](#)

WebFlux Java 配置声明使用带 Comments 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 配置创建的底层 bean。但是，如果您想了解它们，则可以在 `WebFluxConfigurationSupport` 中查看它们，或在[特殊 bean 类](#)中阅读有关它们的更多信息。

要获得配置 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](#)获得对配置的完全控制。

### 1.11.1. 启用 WebFlux 配置

[与 Spring MVC 中的相同](#)

您可以在 Java 配置中使用 `@EnableWebFlux` Comments，如以下示例所示：

```
@Configuration  
@EnableWebFlux  
public class WebConfig {  
}
```

前面的示例注册了许多 Spring WebFlux [infrastructure beans](#)，并适应了 classpath 上可用的依赖项(对于 JSON, XML 等)。

### 1.11.2. WebFlux 配置 API

[与 Spring MVC 中的相同](#)

在 Java 配置中，您可以实现 `WebFluxConfigurer` 接口，如以下示例所示：

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    // Implement configuration methods...

}
```

### 1.11.3. 转换，格式化

[与 Spring MVC 中的相同](#)

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` 注解的支持。如果 Classpath 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) {
        // ...
    }
}
```

#### iNote

有关何时使用 `FormatterRegistrar` 实现的更多信息，请参见[FormatterRegistrar SPI](#)和 `FormattingConversionServiceFactoryBean`。

### 1.11.4. Validation

[与 Spring MVC 中的相同](#)

默认情况下，如果 [Bean Validation](#) 存在于 Classpath(例如，Hibernate Validator)上，则

`LocalValidatorFactoryBean` 被注册为全局 `validator`，以便与 `@Controller` 方法参数上的 `@Valid` 和 `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());
    }

}
```

### Tip

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 配置中声明的那个冲突。

## 1.11.5. Content Type 解析器

### 与 Spring MVC 中的相同

您可以配置 Spring WebFlux 如何根据请求确定 `@Controller` 实例的请求媒体类型。默认情况下，仅选中 `Accept Headers`，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 Content Type 解析：

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void configureContentTypeResolver(RequestedContentTypeResolverBuilder builder) {
        // ...
    }
}
```

## 1.11.6. HTTP 消息编解码器

[与 Spring MVC 中的相同](#)

以下示例显示如何自定义如何读取和写入请求和响应正文：

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @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读取器和写入器，自定义默认读取器或完全替换默认读取器。

对于 Jackson JSON 和 XML，请考虑使用[Jackson2ObjectMapperBuilder](#)，它使用以下属性自定义 Jackson 的默认属性：

- `DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES` 已禁用。
- `MapperFeature.DEFAULT_VIEW_INCLUSION` 已禁用。

如果在 Classpath 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.nio.file.Path` 之类的 Java 7 类型。
- [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. 查看解析器

### [与 Spring MVC 中的相同](#)

以下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` 具有用于与 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;
    }
}
```

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);
    }
}
```

要支持[Content Negotiation](#)并通过视图分辨率(除 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[Codecs](#)。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));
    }

    // ...
}
```

有关与 Spring WebFlux 集成的视图技术的更多信息，请参见[View Technologies](#)。

## 1.11.8. 静态资源

### 与 Spring MVC 中的相同

此选项提供了一种方便的方式来从基于[Resource](#)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 Classpath 上查找和提供与 [/static](#) 相关的静态资源。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。[Last-Modified](#) Headers 也会被评估，如果存在，则返回 [304](#) 状态码。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));  
    }  
}
```

资源处理器还支持[ResourceResolver](#)个实现和[ResourceTransformer](#)个实现的链，可用于创建工具链以使用优化的资源。

您可以将 [VersionResourceResolver](#) 用于基于资源，固定应用程序版本或其他信息计算出的 MD5 哈希的版本化资源 URL。[ContentVersionStrategy](#) (MD5 哈希)是一个不错的选择，但有一些值得注意的 exception(例如与模块加载器一起使用的 JavaScript 资源)。

以下示例显示了如何在 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 [ResourceUrlProvider](#)，以便可以将其注入其他对象。

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，当同时使用 [EncodedResourceResolver](#) (例如，Gzip，Brotli 编码)和

`VersionedResourceResolver` 时，必须按该 `Sequences` 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](#) 也受 `WebJarsResourceResolver` 支持，并且当 `org.webjars:webjars-locator` 存在于 Classpath 中时会自动注册。解析器可以重写 URL 以包括 jar 的版本，并且还可以与没有版本的传入 URL 匹配(例如 `/jquery/jquery.min.js` 到 `/jquery/1.2.0/jquery.min.js`)。

## 1.11.9. 路径匹配

[与 Spring MVC 中的相同](#)

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](#)。以下示例显示了如何使用 `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));
    }
}
```

### Tip

Spring WebFlux 依赖于名为 `RequestPath` 的请求路径的解析表示形式，以访问已解码的路径段值，并删除了分号内容(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 也不支持后缀模式匹配，这与 Spring MVC 不同，在 Spring MVC 中，我们也[recommend](#)不再依赖它。

## 1.11.10. 高级配置模式

[与 Spring MVC 中的相同](#)

@EnableWebFlux 导入 DelegatingWebFluxConfiguration :

- 为 WebFlux 应用程序提供默认的 Spring 配置
- 检测并委托 WebFluxConfigurer 个实现来自定义该配置。

对于高级模式，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 WebConfig 中，但是现在您还可以覆盖 Base Class 中的 bean 声明，并且在 Classpath 上仍然具有任意数量的其他 WebMvcConfigurer 实现。

## 1.12. HTTP/2

[与 Spring MVC 中的相同](#)

需要 Servlet 4 容器支持 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 Client 端。

## 2. WebClient

Spring WebFlux 包括一个用于 HTTP 请求的 Reactive 非阻塞 WebClient。Client 端具有功能性，

Fluent 的 API，具有用于声明式组合的反应式类型，请参见[Reactive Libraries](#)。WebFluxClient 端和服务器依靠相同的非阻塞[codecs](#)对请求和响应内容进行编码和解码。

在内部 `WebClient` 委托给 HTTPClient 端库。默认情况下，它使用[Reactor Netty](#)，内置了对 Jetty [reactive HttpClient](#) 的支持，其他的可以通过 `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)`

上面的方法使用具有默认设置的 Reactor Netty `HttpClient`，并期望

`io.projectreactor.netty:reactor-netty` 位于 Classpath 中。

您还可以将 `WebClient.builder()` 与其他选项一起使用：

- `uriBuilderFactory`：自定义 `UriBuilderFactory` 用作基本 URL。
- `defaultHeader`：每个请求的标题。
- `defaultCookie`：每个请求的 cookie。
- `defaultRequest`：`Consumer` 自定义每个请求。
- `filter`：针对每个请求的 Client 端过滤器。
- `exchangeStrategies`：HTTP 消息读取器/写入器定制。
- `clientConnector`：HTTPClient 端库设置。

以下示例配置[HTTP codecs](#)：

```
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, filterD
```

### 2.1.1. 反应堆净值

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` 参与 `reactor.netty.http.HttpResources` 拥有的全局 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，作为 WAR 部署的 Spring MVC 应用程序），则可以声明 `ReactorResourceFactory` 和 `globalResources=true`（默认值）类型的 Spring 托管 bean，以确保 Reactor Netty 全局 Spring

`ApplicationContext` 关闭时，资源将关闭，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 NettyClient 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); (1)
    return factory;
}

@Bean
public WebClient webClient() {

    Function<HttpClient, HttpClient> mapper = client -> {
        // Further customizations...
    };

    ClientHttpConnector connector =
        new ReactorClientHttpConnector(resourceFactory(), mapper); (2)

    return WebClient.builder().clientConnector(connector).build(); (3)
}
```

- (1) 创建独立于全局资源的资源。
- (2) 将 `ReactorClientHttpConnector` 构造函数与资源工厂一起使用。
- (3) 将连接器插入 `WebClient.Builder`。

## 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;
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

以下示例显示了如何自定义 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(`Executor`，`ByteBufferPool`，`Scheduler`)，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 JettyClient 端(和服务器)的多个实例之间共享资源，并 pass 语句类型为 `JettyResourceFactory` 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); (1)

    return WebClient.builder().clientConnector(connector).build(); (2)
}
```

- (1) 将 `JettyClientHttpConnector` 构造函数与资源工厂一起使用。

- (2) 将连接器插入 `WebClient.Builder`。

## 2.2. `retrieve()`

`retrieve()` 方法是获取响应正文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("http://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);
```

默认情况下，带有 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` 方法来自定义产生的异常，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()`

`exchange()` 方法比 `retrieve` 方法提供更多的控制权。以下示例与 `retrieve()` 等效，但也提

供对 `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));
```

请注意(与 `retrieve()` 不同)，对于 `exchange()`，没有 `4xx` 和 `5xx` 响应的自动错误 `signal`。您必须检查状态码并决定如何进行。

### ⚠Warning

使用 `exchange()` 时，必须始终使用 `ClientResponse` 的任何 `body` 或 `toEntity` 方法，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. 请求正文

可以从 `Object` 编码请求主体，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 = ... ;  
  
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. 表格数据

要发送表单数据，您可以提供 `MultiValueMap<String, String>` 作为正文。请注意，内容会由 `FormHttpMessageWriter` 自动设置为 `application/x-www-form-urlencoded`。以下示例显示了如何使用 `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 数据

要发送 Multipart 数据，您需要提供一个 `MultiValueMap<String, ?>`，其值要么是代表

Component 内容的 `Object` 实例，要么是代表 Component 内容和头的 `HttpEntity` 实例。

`MultipartBodyBuilder` 提供了方便的 API 以准备 Multipart 请求。以下示例显示了如何创建

`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`。Content Type 是根据要序列化的

`HttpMessageWriter` 自动确定的，对于 `Resource` 则根据文件 extensions 自动确定。如有必要，您可以通过重载的生成器 `part` 方法之一显式提供 `MediaType` 供每个 Component 使用。

准备好 `MultiValueMap` 之后，将其传递给 `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，该值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` 的替代方法，您还可以通过内置的 `BodyInserters` 提供内联样式

的 Multipart 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 端过滤器

您可以通过 `WebClient.Builder` 注册 Client 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：

```
// static import of ExchangeFilterFunctions.basicAuthentication

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，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("http://example.org/")
    .attribute("myAttribute", "...")
    .retrieve()
    .bodyToMono(Void.class);

}
```

您也可以复制现有的 `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：

```
// static import of ExchangeFilterFunctions.basicAuthentication

WebClient client = webClient.mutate()
    .filters(filterList -> {
        filterList.add(0, basicAuthentication("user", "password"));
    })
    .build();
```

## 2.6. 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[static-server](#)示例。

## 3. WebSockets

[与 Servlet 堆栈中的相同](#)

参考文档的此部分涵盖对反应堆 WebSocket 消息传递的支持。

### 3.1. WebSocket 介绍

WebSocket 协议[RFC 6455](#)提供了一种标准化方法，可以通过单个 TCP 连接在 Client 端和服务器之间构建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` Headers 进行升级，或者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 (1)
Connection: Upgrade (2)
Sec-WebSocket-Key: Uc9l9TMkWGbhFD2qnFH1tg==
Sec-WebSocket-Protocol: v10.stomp, v11.stomp
Sec-WebSocket-Version: 13
Origin: http://localhost:8080
```

- (1) `Upgrade` Headers。

- (2) 使用 `Upgrade` 连接。

具有 WebSocket 支持的服务器代替通常的 200 状态代码，返回类似于以下内容的输出：

```
HTTP/1.1 101 Switching Protocols (1)
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: 1qVdfYHU9hP014JYYNxF623Gzn0=
Sec-WebSocket-Protocol: v10.stomp
```

- (1) 协议切换

成功握手后，HTTP 升级请求的基础 TCP 套接字将保持打开状态，Client 端和服务器均可 continue 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 与 WebSocket

尽管 WebSocket 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，Client 端访问那些 URL，即请求-响应样式。服务器根据 HTTP URL，方法和 Headers 将请求路由到适当的处理程序。

相比之下，在 **WebSockets** 中，初始连接通常只有一个 URL。随后，所有应用程序消息在同一 TCP 连接上流动。这指向了完全不同的异步，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 Client 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。

**WebSocketClient** 端和服务器可以通过 HTTP 握手请求上的 **Sec-WebSocket-Protocol** Headers 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. 何时使用 **WebSockets**

**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 流或长时间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，但是每隔几分钟这样做是完全可以的。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** Headers，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**

[与 Servlet 堆栈中的相同](#)

Spring 框架提供了一个 **WebSocket API**，可用于编写处理 **WebSocket** 消息的 Client 端和服务器端应用程序。

### 3.2.1. **Server**

[与 Servlet 堆栈中的相同](#)

要创建 WebSocket 服务器，您可以先创建一个 `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，您可以将其 Map 到 URL 并添加 `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：

WebSocketSession 方法	Description
<code>Flux&lt;WebSocketMessage&gt; receive()</code>	提供对入站消息流的访问，并在关闭连接后完成。
<code>Mono&lt;Void&gt;</code> <code>send(Publisher&lt;WebSocketMessage&gt;)</code>	获取传出消息的源，编写消息，然后返回 <code>Mono&lt;Void&gt;</code> ，该源完成并写入后即完成。

`WebSocketHandler` 必须将入站和出站流组成一个统一的流，并返回 `Mono<Void>` 以反映该流的完成。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。
- 在选定的位置，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“响应流” signal 会终止活动。入站流接收完成或错误 signal，而出站流接收取消 signal。

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()                               (1)
            .doOnNext(message -> {
                // ...
            })                                         (2)
            .concatMap(message -> {
                // ...
            })                                         (3)
            .then();                                     (4)
    }
}
```

- (1) 访问入站消息流。

- **(2)** 对每条消息进行处理。
- **(3)** 执行使用消息内容的嵌套异步操作。
- **(4)** 返回一个 `Mono<Void>`，它在接收完成时会完成。

### Tip

对于嵌套的异步操作，您可能需要在使用池化数据缓冲区(例如 Netty)的基础服务器上调用 `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()                                (1)
            .doOnNext(message -> {
                // ...
            })
            .concatMap(message -> {
                // ...
            })
            .map(value -> session.textMessage("Echo " + value));      (2)

        return session.send(output);
    }
}
```

- **(1)** 处理入站消息流。
- **(2)** 创建出站消息，从而产生组合流。
- **(3)** 返回 `Mono<Void>`，但我们 `continue` 接收时未完成。

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
    public Mono<Void> handle(WebSocketSession session) {
        Mono<Void> input = session.receive()                                (1)
```

```

        .doOnNext(message -> {
            // ...
        })
        .concatMap(message -> {
            // ...
        })
        .then();

Flux<String> source = ... ;
Mono<Void> output = session.send(source.map(session::textMessage)); (2)

return Mono.zip(input, output).then(); (3)
}
}

```

- (1) 处理入站消息流。
- (2) 发送传出邮件。
- (3) 加入流并返回一个 `Mono<Void>`，当任何一个流结束时，该 `Mono<Void>` 完成。

### 3.2.3. DataBuffer

`DataBuffer` 是 WebFlux 中字节缓冲区的表示形式。参考的 Spring Core 部分在[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引用计数，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 Importing 数据缓冲区以确保它们不被释放，则必须使用 `DataBufferUtils.retain(dataBuffer)`，然后在使用缓冲区时使用

```
DataBufferUtils.release(dataBuffer)。
```

### 3.2.4. Handshake

[与 Servlet 堆栈中的相同](#)

`WebSocketHandlerAdapter` 代表 `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，然后将

`RequestUpgradeStrategy` 用于所使用的服务器。当前，内置了对 Reactor Netty

，Tomcat，Jetty 和 Undertow 的支持。

`HandshakeWebSocketService` 公开了 `sessionAttributePredicate` 属性，该属性允许设置 `Predicate<String>` 以从 `WebSession` 提取属性并将其插入 `WebSocketSession` 的属性。

### 3.2.5. 服务器配置

与 Servlet 堆栈中的相同

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。

### 3.2.6. CORS

与 Servlet 堆栈中的相同

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 `CorsConfiguraiton` 并带有允许的来源，Headers 和其他详细信息。如果您不能这样做，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` 方法将它们组合在一起。

### 3.2.7. Client

Spring WebFlux 为 Reactor Netty, Tomcat, Jetty, Undertow 和标准 Java(即 JSR-356)的实现提供了 `WebSocketClient` 抽象。

### iNote

TomcatClient 端实际上 是标准 JavaClient 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 Client 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());
```

某些 Client 端(例如 Jetty)实现 `Lifecycle`，并且需要先停止然后启动，然后才能使用它们。所有 Client 端都具有与基础 WebSocketClient 端的配置有关的构造器选项。

## 4. Testing

在 Spring MVC 中相同

`spring-test` 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 WebReactive](#)。

[WebTestClient](#) 构建在这些模拟请求和响应对象的基础上，以提供对不使用 HTTP 服务器的 WebFlux 应用程序测试的支持。您也可以使用 `WebTestClient` 进行端到端集成测试。

## 5. Reactive Library

`spring-webflux` 依赖 `reactor-core`，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 Importing。`Flux` 与 `Mono` 的使用很重要，因为它有助于表达基数，例如，是期望单个还是多个异步值，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 Comments 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[ReactiveAdapterRegistry](#)提供了对反应库和其他异步类型的可插入支持。该注册表具有对 RxJava 和 `CompletableFuture` 的内置支持，但您也可以注册其他注册表。

对于功能性 API(例如[Functional Endpoints](#), `WebClient` 等)，WebFlux API 的一般规则适用于返回值 `Flux` 和 `Mono` 和响应流 `Publisher` 作为 Importing。当提供 `Publisher` 时，无论是自定义的还是来自其他 Reactive 库的，都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)` 包裹它，而不用传递原始 `Publisher`。

例如，给定的 `Publisher` 不是 `Mono`，Jackson JSON 消息编写器需要多个值。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。

## Integration

参考文档的这一部分涵盖了 Spring Framework 与许多 Java EE(及相关)技术的集成。

## 1. 使用 Spring 进行远程处理和 Web 服务

Spring 具有集成类，用于通过各种技术来远程支持。远程支持简化了由常规(Spring)POJO 实施的启用远程服务的开发。当前，Spring 支持以下远程技术：

- **远程方法调用(RMI)**：通过使用 `RmiProxyFactoryBean` 和 `RmiServiceExporter`，Spring

支持传统的 RMI(具有 `java.rmi.Remote` 接口和 `java.rmi.RemoteException`)以及通过 RMI 调用者进行透明远程处理(具有任何 Java 接口)。

- **Spring 的 HTTP 调用程序**： Spring 提供了一种特殊的远程处理策略，该策略允许通过 HTTP 进行 Java 序列化，从而支持任何 Java 接口(就像 RMI 调用程序一样)。相应的支持类别为 `HttpInvokerProxyFactoryBean` 和 `HttpInvokerServiceExporter`。
- **Hessian**：通过使用 Spring 的 `HessianProxyFactoryBean` 和 `HessianServiceExporter`，您可以通过 Cauchy 提供的基于 HTTP 的轻量级二进制协议透明地公开您的服务。
- **JAX-WS**： Spring 通过 JAX-WS(Java EE 5 和 Java 6 中引入的 JAX-RPC 的继承者)为 Web 服务提供远程支持。
- **JMS**：通过 `JmsInvokerServiceExporter` 和 `JmsInvokerProxyFactoryBean` 类支持使用 JMS 作为基础协议进行远程处理。
- **AMQP**： Spring AMQP 项目支持使用 AMQP 作为基础协议进行远程处理。

在讨论 Spring 的远程功能时，我们使用以下域模型和相应的服务：

```
public class Account implements Serializable{  
    private String name;  
  
    public String getName(){  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

```
public interface AccountService {  
  
    public void insertAccount(Account account);  
  
    public List<Account> getAccounts(String name);  
}
```

```
// the implementation doing nothing at the moment
public class AccountServiceImpl implements AccountService {

    public void insertAccount(Account acc) {
        // do something...
    }

    public List<Account> getAccounts(String name) {
        // do something...
    }

}
```

本节首先使用 RMI 将服务公开给远程 Client 端，然后再谈谈使用 RMI 的缺点。然后 continue 以使用 Hessian 作为协议的示例。

## 1.1. 使用 RMI 公开服务

通过使用 Spring 对 RMI 的支持，您可以通过 RMI 基础结构透明地公开服务。进行了此设置之后，除了不存在对安全上下文传播或远程事务传播的标准支持这一事实之外，您基本上具有与远程 EJB 相似的配置。当您使用 RMI 调用程序时，Spring 确实为此类附加调用上下文提供了钩子，因此，例如，您可以插入安全框架或自定义安全凭证。

### 1.1.1. 使用 RmiServiceExporter 导出服务

使用 `RmiServiceExporter`，我们可以将 `AccountService` 对象的接口公开为 RMI 对象。可以使用 `RmiProxyFactoryBean` 来访问该接口，或者在传统 RMI 服务的情况下可以通过普通 RMI 来访问该接口。`RmiServiceExporter` 明确支持通过 RMI 调用程序公开任何非 RMI 服务。

我们首先必须在 Spring 容器中设置服务。以下示例显示了如何执行此操作：

```
<bean id="accountService" class="example.AccountServiceImpl">
    <!-- any additional properties, maybe a DAO? -->
</bean>
```

接下来，我们必须使用 `RmiServiceExporter` 公开我们的服务。以下示例显示了如何执行此操作：

:

```
<bean class="org.springframework.remoting.rmi.RmiServiceExporter">
    <!-- does not necessarily have to be the same name as the bean to be exported -->
    <property name="serviceName" value="AccountService"/>
    <property name="service" ref="accountService"/>
    <property name="serviceInterface" value="example.AccountService"/>
    <!-- defaults to 1099 -->
    <property name="registryPort" value="1199"/>
</bean>
```

在前面的示例中，我们覆盖了 RMI 注册表的端口。通常，您的应用服务器还维护一个 RMI 注册表，因此最好不要干涉该注册表。此外，服务名称用于绑定服务。因此，在前面的示例中，服务绑定在 `'rmi://HOST:1199/AccountService'` 处。稍后，我们将使用此 URL 链接到 Client 端的服务。

#### iNote

`servicePort` 属性已被省略(默认为 0)。这意味着将使用匿名端口与服务进行通信。

### 1.1.2. 在 Client 端链接服务

我们的 Client 是一个简单的对象，它使用 `AccountService` 来 Management 帐户，如以下示例所示：

```
public class SimpleObject {

    private AccountService accountService;

    public void setAccountService(AccountService accountService) {
        this.accountService = accountService;
    }

    // additional methods using the accountService
}
```

为了在 Client 端上链接服务，我们创建了一个单独的 Spring 容器，其中包含以下简单对象和服务链接配置位：

```
<bean class="example.SimpleObject">
    <property name="accountService" ref="accountService"/>
</bean>
```

```
<bean id="accountService" class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
    <property name="serviceUrl" value="rmi://HOST:1199/AccountService"/>
    <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

这就是我们要在 Client 端上支持远程帐户服务所需要做的一切。 Spring 透明地创建一个调用程序， 并通过 `RmiServiceExporter` 远程启用帐户服务。在 Client 端， 我们使用 `RmiProxyFactoryBean` 将其链接。

## 1.2. 使用 Hessian 通过 HTTP 远程调用服务

Hessian 提供了一个基于 HTTP 的二进制远程协议。它由 Caucho 开发， 您可以在 <http://www.caucho.com> 上找到有关 Hessian 本身的更多信息。

### 1.2.1. 为 Hessian 连接 DispatcherServlet

Hessian 通过 HTTP 进行通信， 并通过使用自定义 servlet 进行通信。通过使用 Spring 的 `DispatcherServlet` 原理(请参阅[\[webmvc#mvc-servlet\]](#))， 我们可以连接这样的 servlet 以公开您的服务。首先， 我们必须在应用程序中创建一个新的 servlet， 如以下 `web.xml` 的摘录所示：

```
<servlet>
    <servlet-name>remoting</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>remoting</servlet-name>
    <url-pattern>/remoting/*</url-pattern>
</servlet-mapping>
```

如果您熟悉 Spring 的 `DispatcherServlet` 原理， 那么您可能知道现在必须在 `WEB-INF` 目录中创建一个名为 `remoting-servlet.xml` (在 Servlet 名称之后)的 Spring 容器配置资源。下一节将使用应用程序上下文。

或者， 考虑使用 Spring 的简单 `HttpRequestHandlerServlet`。这样做使您可以将远程导出程序定义嵌入到根应用程序上下文中(默认情况下，在 `WEB-INF/applicationContext.xml` 中)， 而单

个 Servlet 定义则指向特定的导出程序 bean。在这种情况下，每个 servlet 名称都必须与其目标导出器的 bean 名称相匹配。

### 1.2.2. 使用 HessianServiceExporter 公开您的 Bean

在新创建的名为 `remoting-servlet.xml` 的应用程序上下文中，我们创建一个

`HessianServiceExporter` 以导出我们的服务，如以下示例所示：

```
<bean id="accountService" class="example.AccountServiceImpl">
    <!-- any additional properties, maybe a DAO? -->
</bean>

<bean name="/AccountService" class="org.springframework.remoting.caucho.HessianServiceExporter">
    <property name="service" ref="accountService"/>
    <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

现在，我们准备在 Client 端链接服务。没有指定显式处理程序 Map(将请求 URLMap 到服务)，因此我们使用 `BeanNameUrlHandlerMapping`。因此，该服务将在包含 `DispatcherServlet` 实例的 Map(如先前定义) `http://HOST:8080/remoting/AccountService` 中通过其 bean 名称指示的 URL 导出。

另外，您可以在根应用程序上下文中(例如，在 `WEB-INF/applicationContext.xml` 中)创建 `HessianServiceExporter`，如以下示例所示：

```
<bean name="accountExporter" class="org.springframework.remoting.caucho.HessianServiceExporter">
    <property name="service" ref="accountService"/>
    <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

在后一种情况下，您应该在 `web.xml` 中为此导出程序定义一个相应的 servlet，最终结果相同：导出程序 Map 到 `/remoting/AccountService` 的请求路径。注意，servlet 名称需要与目标导出器的 bean 名称匹配。以下示例显示了如何执行此操作：

```
<servlet>
    <servlet-name>accountExporter</servlet-name>
    <servlet-class>org.springframework.web.context.support.HttpRequestHandlerServlet</servlet-class>
```

```
</servlet>

<servlet-mapping>
    <servlet-name>accountExporter</servlet-name>
    <url-pattern>/remoting/AccountService</url-pattern>
</servlet-mapping>
```

### 1.2.3. 在 Client 端上链接服务

通过使用 `HessianProxyFactoryBean`，我们在 Client 端链接服务。与 RMI 示例相同的原理适用。我们创建一个单独的 bean 工厂或应用程序上下文，并通过使用 `AccountService` 来 Management 帐户来提及 `SimpleObject` 所在的以下 bean，如以下示例所示：

```
<bean class="example.SimpleObject">
    <property name="accountService" ref="accountService"/>
</bean>

<bean id="accountService" class="org.springframework.remoting.caucho.HessianProxyFactor
    <property name="serviceUrl" value="http://remotehost:8080/remoting/AccountService"/>
    <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

### 1.2.4. 将 HTTP 基本身份验证应用于通过 Hessian 公开的服务

Hessian 的优点之一是我们可以轻松地应用 HTTP 基本身份验证，因为这两种协议都是基于 HTTP 的。例如，可以通过使用 `web.xml` 安全功能来应用常规的 HTTP 服务器安全性机制。通常，您无需在此处使用每个用户的安全凭证。相反，您可以使用在 `HessianProxyFactoryBean` 级别定义的共享凭据(类似于 JDBC `DataSource`)，如以下示例所示：

```
<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping">
    <property name="interceptors" ref="authorizationInterceptor"/>
</bean>

<bean id="authorizationInterceptor"
      class="org.springframework.web.servlet.handler.UserRoleAuthorizationInterceptor">
    <property name="authorizedRoles" value="administrator,operator"/>
</bean>
```

在前面的示例中，我们明确提到 `BeanNameUrlHandlerMapping` 并设置了一个拦截器，以仅让 Management 员和操作员调用此应用程序上下文中提到的 bean。

## iNote

前面的示例未显示灵活的安全基础结构。有关安全性的更多选项，请查看位于 <http://projects.spring.io/spring-security/> 的 Spring Security 项目。

## 1.3. 使用 HTTP 调用程序公开服务

与 Hessian 相反，Spring HTTP 调用程序都是轻量级协议，它们使用自己的苗条序列化机制，并使用标准 Java 序列化机制通过 HTTP 公开服务。如果您的参数和返回类型是无法通过使用 Hessian 使用的序列化机制进行序列化的复杂类型，则这将具有巨大的优势(选择远程处理技术时，请参阅下一节以获得更多注意事项)。

在幕后，Spring 使用 JDK 或 Apache [HttpComponents](#) 提供的标准功能来执行 HTTP 调用。如果您需要更高级且更容易使用的功能，请使用后者。有关更多信息，请参见 [hc.apache.org/httpcomponents-client-ga/](http://hc.apache.org/httpcomponents-client-ga/)。

## ⚠Warning

注意由于不安全的 Java 反序列化而导致的漏洞：在反序列化步骤中，操纵的 Importing 流可能导致服务器上有害的代码执行。因此，请勿将 HTTP 调用方终结点暴露给不受信任的 Client 端。而是仅在您自己的服务之间公开它们。通常，强烈建议您改用其他任何消息格式（例如 JSON）。

如果您担心由 Java 序列化引起的安全漏洞，请考虑在核心 JVM 级别上使用通用序列化筛选器机制，该机制最初是为 JDK 9 开发的，但同时又移植到 JDK 8、7 和 6. 参见 [https://blogs.oracle.com/java-platform-group/entry/incoming\\_filter\\_serialization\\_data\\_a](https://blogs.oracle.com/java-platform-group/entry/incoming_filter_serialization_data_a) 和 <http://openjdk.java.net/jeps/290>。

### 1.3.1. 公开服务对象

为服务对象设置 HTTP 调用程序基础结构与使用 Hessian 进行设置的方法非常相似。由于 Hessian 支持提供了 [HessianServiceExporter](#)，Spring 的 [HttpInvoker](#) 支持提供了

```
org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter。
```

为了在 Spring Web MVC `DispatcherServlet` 中公开 `AccountService` (前面提到), 需要在调度程序的应用程序上下文中进行以下配置, 如以下示例所示:

```
<bean name="/AccountService" class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
    <property name="service" ref="accountService"/>
    <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

如[关于黑森 State 的部分](#)所述, 此类导出程序定义通过 `DispatcherServlet` 实例的标准 Map 工具公开。

另外, 您可以在根应用程序上下文中(例如, 在 `'WEB-INF/applicationContext.xml'` 中)创建 `HttpInvokerServiceExporter`, 如以下示例所示:

```
<bean name="accountExporter" class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
    <property name="service" ref="accountService"/>
    <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

另外, 您可以在 `web.xml` 中为此导出程序定义一个相应的 servlet, 该 servlet 名称与目标导出程序的 bean 名称匹配, 如以下示例所示:

```
<servlet>
    <servlet-name>accountExporter</servlet-name>
    <servlet-class>org.springframework.web.context.support.HttpRequestHandlerServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>accountExporter</servlet-name>
    <url-pattern>/remoting/AccountService</url-pattern>
</servlet-mapping>
```

### 1.3.2. 在 Client 端链接服务

同样, 从 Client 端链接服务与使用 Hessian 时的方式非常相似。通过使用代理, Spring 可以将您对 HTTP POST 请求的调用转换为指向导出服务的 URL。以下示例显示如何配置此安排:

```
<bean id="httpInvokerProxy" class="org.springframework.remoting.httpinvoker.HttpInvokerProxy">
    <property name="serviceUrl" value="http://remotehost:8080/remoting/AccountService"/>
    <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

如前所述，您可以选择要使用的 `HTTPClient` 端。默认情况下，`HttpInvokerProxy` 使用 JDK 的 `HTTP` 功能，但是您也可以通过设置 `httpInvokerRequestExecutor` 属性来使用 Apache `HttpComponents` Client 端。以下示例显示了如何执行此操作：

```
<property name="httpInvokerRequestExecutor">
    <bean class="org.springframework.remoting.httpinvoker.HttpComponentsHttpInvokerRequestExecutor"/>
</property>
```

## 1.4. Web Services

Spring 提供了对标准 Java Web 服务 API 的全面支持：

- 使用 JAX-WS 公开 Web 服务
- 使用 JAX-WS 访问 Web 服务

除了在 Spring Core 中对 JAX-WS 的库存支持之外，Spring 产品组合还具有[SpringWeb Service](#)，这是一种针对 Contract 优先，文档驱动的 Web 服务的解决方案。

### 1.4.1. 使用 JAX-WS 公开基于 Servlet 的 Web 服务

Spring 为 JAX-WS servlet 端点实现提供了一个方便的 Base Class：

`SpringBeanAutowiringSupport`。为了公开 `AccountService`，我们扩展 Spring 的 `SpringBeanAutowiringSupport` 类并在此处实现我们的业务逻辑，通常将调用委派给业务层。我们使用 Spring 的 `@Autowired` 注解来表达对 `SpringManagement` 的 bean 的依赖。以下示例显示了扩展 `SpringBeanAutowiringSupport` 的类：

```
/**
 * JAX-WS compliant AccountService implementation that simply delegates
 * to the AccountService implementation in the root web application context.
```

```

/*
 * This wrapper class is necessary because JAX-WS requires working with dedicated
 * endpoint classes. If an existing service needs to be exported, a wrapper that
 * extends SpringBeanAutowiringSupport for simple Spring bean autowiring (through
 * the @Autowired annotation) is the simplest JAX-WS compliant way.
 *
 * This is the class registered with the server-side JAX-WS implementation.
 * In the case of a Java EE 5 server, this would simply be defined as a servlet
 * in web.xml, with the server detecting that this is a JAX-WS endpoint and reacting
 * accordingly. The servlet name usually needs to match the specified WS service name.
 *
 * The web service engine manages the lifecycle of instances of this class.
 * Spring bean references will just be wired in here.
 */
import org.springframework.web.context.support.SpringBeanAutowiringSupport;

@WebService(serviceName="AccountService")
public class AccountServiceEndpoint extends SpringBeanAutowiringSupport {

    @Autowired
    private AccountService biz;

    @WebMethod
    public void insertAccount(Account acc) {
        biz.insertAccount(acc);
    }

    @WebMethod
    public Account[] getAccounts(String name) {
        return biz.getAccounts(name);
    }
}

```

我们的 `AccountServiceEndpoint` 需要在与 Spring 上下文相同的 Web 应用程序中运行，以允许访问 Spring 的设施。在 Java EE 5 环境中，默认情况下就是这种情况，使用用于 JAX-WS servlet 端点部署的标准协定。有关详细信息，请参见各种 Java EE 5 Web 服务教程。

### 1.4.2. 使用 JAX-WS 导出独立的 Web 服务

Oracle JDK 随附的内置 JAX-WS 提供程序通过使用 JDK 中也包含的内置 HTTP 服务器来支持 Web 服务公开。Spring 的 `SimpleJaxWsServiceExporter` 检测到 Spring 应用程序上下文中所有带有 `@WebService` Comments 的 bean，并将它们通过默认的 JAX-WS 服务器(JDK HTTP 服务器)导出。

在这种情况下，端点实例被定义和 Management 为 Spring Bean 本身。它们已在 JAX-WS 引擎中注册，但是它们的生命周期取决于 Spring 应用程序上下文。这意味着您可以将 Spring 功能(例如显

式依赖项注入)应用于端点实例。通过 `@Autowired` 进行 `Comments` 驱动的注入也有效。以下示例显示了如何定义这些 bean:

```
<bean class="org.springframework.remoting.jaxws.SimpleJaxWsServiceExporter">
    <property name="baseAddress" value="http://localhost:8080/" />
</bean>

<bean id="accountServiceEndpoint" class="example.AccountServiceEndpoint">
    ...
</bean>

...
```

`AccountServiceEndpoint` 可以但不必从 Spring 的 `SpringBeanAutowiringSupport` 派生，因为此示例中的端点是完全由 `SpringManagement` 的 bean。这意味着端点实现可以如下所示(不声明任何超类，并且仍然采用 Spring 的 `@Autowired` 配置 `Comments`):

```
@WebService(serviceName="AccountService")
public class AccountServiceEndpoint {

    @Autowired
    private AccountService biz;

    @WebMethod
    public void insertAccount(Account acc) {
        biz.insertAccount(acc);
    }

    @WebMethod
    public List<Account> getAccounts(String name) {
        return biz.getAccounts(name);
    }
}
```

### 1.4.3. 使用 JAX-WS RI 的 Spring 支持导出 Web 服务

作为 GlassFish 项目的一部分开发的 Oracle JAX-WS RI，将 Spring 支持作为其 JAX-WS Commons 项目的一部分。这允许将 JAX-WS 端点定义为 `SpringManagement` 的 bean，类似于 [previous section](#) 中讨论的独立模式，但这次是在 Servlet 环境中。

**iNote**

这在 Java EE 5 环境中不可移植。它主要用于将 JAX-WS RI 嵌入为 Web 应用程序一部分的非 EE 环境，例如 Tomcat。

与导出基于 servlet 的端点的标准样式不同之处在于，端点实例本身的生命周期由 SpringManagement，并且在 `web.xml` 中仅定义了一个 JAX-WS servlet。使用标准的 Java EE 5 样式(如前所示)，每个服务端点都有一个 servlet 定义，每个端点通常委派给 Spring Bean(如前所述，通过使用 `@Autowired`)。

有关设置和使用方式的详细信息，请参见<https://jax-ws-commons.java.net/spring/>。

#### 1.4.4. 使用 JAX-WS 访问 Web 服务

Spring 提供了两个工厂 bean 来创建 JAX-WS Web 服务代理，即

`LocalJaxWsServiceFactoryBean` 和 `JaxWsPortProxyFactoryBean`。前者只能返回一个 JAX-WS 服务类供我们使用。后者是完整版本，可以返回实现我们的业务服务接口的代理。在以下示例中，我们再次使用 `JaxWsPortProxyFactoryBean` 为 `AccountService` 端点创建代理：

```
<bean id="accountWebService" class="org.springframework.remoting.jaxws.JaxWsPortProxyFactoryBean">
    <property name="serviceInterface" value="example.AccountService"/> (1)
    <property name="wsdlDocumentUrl" value="http://localhost:8888/AccountServiceEndpoint"/>
    <property name="namespaceUri" value="http://example//"/>
    <property name="serviceName" value="AccountService"/>
    <property name="portName" value="AccountServiceEndpointPort"/>
</bean>
```

- (1) 其中 `serviceInterface` 是 Client 使用的我们的业务界面。

`wsdlDocumentUrl` 是 WSDL 文件的 URL。Spring 在启动时需要使用它来创建 JAX-WS 服务。

`namespaceUri` 对应于.wsdl 文件中的 `targetNamespace`。`serviceName` 对应于.wsdl 文件中的服务名称。`portName` 对应于.wsdl 文件中的端口名称。

访问 Web 服务很容易，因为我们有一个供其使用的 bean 工厂，它将它公开为名为 `AccountService` 的接口。以下示例说明了如何在 Spring 中进行连接：

```
<bean id="client" class="example.AccountClientImpl">
    ...
    <property name="service" ref="accountWebService"/>
</bean>
```

从 Client 端代码，我们可以像访问普通类一样访问 Web 服务，如以下示例所示：

```
public class AccountClientImpl {

    private AccountService service;

    public void setService(AccountService service) {
        this.service = service;
    }

    public void foo() {
        service.insertAccount(...);
    }
}
```

### iNote

上面的内容略有简化，因为 JAX-WS 需要使用 `@WebService`，`@SOAPBinding` `etcComments` 对端点接口和实现类进行 Comments。这意味着您不能(轻松)使用纯 Java 接口和实现类作为 JAX-WS 端点工件。您需要首先对它们进行 Comments。查看 JAX-WS 文档以获取有关这些需求的详细信息。

## 1.5. 通过 JMS 公开服务

您还可以通过使用 JMS 作为基础通信协议来透明地公开服务。Spring 框架中的 JMS 远程支持非常基本。它在 `same thread` 上和同一非事务 `Session` 中发送和接收。结果，吞吐量取决于实现方式。请注意，这些单线程和非事务性约束仅适用于 Spring 的 JMS 远程支持。请参阅[JMS\(Java 消息服务\)](#)以获取有关 Spring 对基于 JMS 的消息传递的丰富支持的信息。

服务器和 Client 端均使用以下接口：

```
package com.foo;

public interface CheckingAccountService {
```

```
    public void cancelAccount(Long accountId);  
}
```

在服务器端使用上述接口的以下简单实现：

```
package com.foo;  
  
public class SimpleCheckingAccountService implements CheckingAccountService {  
  
    public void cancelAccount(Long accountId) {  
        System.out.println("Cancelling account [" + accountId + "]");  
    }  
}
```

以下配置文件包含在 Client 机和服务器上共享的 JMS 基础结构 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"  
       xsi:schemaLocation="http://www.springframework.org/schema/beans  
                           http://www.springframework.org/schema/beans/spring-beans.xsd">  
  
    <bean id="connectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">  
        <property name="brokerURL" value="tcp://ep-t43:61616"/>  
    </bean>  
  
    <bean id="queue" class="org.apache.activemq.command.ActiveMQQueue">  
        <constructor-arg value="mmm"/>  
    </bean>  
  
</beans>
```

### 1.5.1. 服务器端配置

在服务器上，您需要公开使用 `JmsInvokerServiceExporter` 的服务对象，如以下示例所示：

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
       xsi:schemaLocation="http://www.springframework.org/schema/beans  
                           http://www.springframework.org/schema/beans/spring-beans.xsd">  
  
    <bean id="checkingAccountService" class="org.springframework.jms.remoting.JmsInvokerServiceExporter">  
        <property name="serviceInterface" value="com.foo.CheckingAccountService"/>  
        <property name="service">  
            <bean class="com.foo.SimpleCheckingAccountService"/>  
        </property>
```

```

</bean>

<bean class="org.springframework.jms.listener.SimpleMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory" />
    <property name="destination" ref="queue" />
    <property name="concurrentConsumers" value="3" />
    <property name="messageListener" ref="checkingAccountService" />
</bean>

</beans>

```

```

package com.foo;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Server {

    public static void main(String[] args) throws Exception {
        new ClassPathXmlApplicationContext(new String[]{"com/foo/server.xml", "com/foo/beans.xml"});
    }
}

```

## 1.5.2. Client 端配置

Client 端只需要创建一个 Client 端代理即可实现约定的接口([CheckingAccountService](#))。

以下示例定义了可以注入到其他 Client 端对象中的 Bean(代理负责通过 JMS 将调用转发到服务器端对象):

```

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

    <bean id="checkingAccountService"
          class="org.springframework.jms.remoting.JmsInvokerProxyFactoryBean">
        <property name="serviceInterface" value="com.foo.CheckingAccountService" />
        <property name="connectionFactory" ref="connectionFactory" />
        <property name="queue" ref="queue" />
    </bean>

</beans>

```

```

package com.foo;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

```

```
public class Client {  
  
    public static void main(String[] args) throws Exception {  
        ApplicationContext ctx = new ClassPathXmlApplicationContext(  
            new String[] {"com/foo/client.xml", "com/foo/jms.xml"});  
        CheckingAccountService service = (CheckingAccountService) ctx.getBean("checking  
        service");  
        service.cancelAccount(new Long(10));  
    }  
}
```

## 1.6. AMQP

有关更多信息，请参见[Spring AMQP 参考指南的“使用 AMQP 进行远程处理”部分](#)。

### iNote

远程接口未实现自动检测

对于远程接口，不会自动检测已实现接口的主要原因是为了避免为远程调用者打开太多门。

目标对象可能实现内部回调接口，例如 `InitializingBean` 或 `DisposableBean`，而这些接口将不会向调用者公开。

在本地情况下，提供具有目标所实现的所有接口的代理通常无关紧要。但是，当导出远程服务时，应公开特定的服务接口，并提供用于远程使用的特定操作。除了内部回调接口之外，目标还可以实现多个业务接口，其中只有一个用于远程公开。由于这些原因，我们需要指定这样的服务接口。

这是配置便利性与内部方法意外暴露风险之间的折衷方案。始终指定服务接口并不会花费太多精力，这使您可以安全地控制特定方法的使用。

## 1.7. 选择技术时的注意事项

这里介绍的每种技术都有其缺点。选择一种技术时，应仔细考虑您的需求，公开的服务以及通过网络发送的对象。

使用 RMI 时，除非通过隧道传送 RMI 流量，否则无法通过 HTTP 协议访问对象。RMI 是一个重量级协议，因为它支持全对象序列化，当您使用需要通过网络进行序列化的复杂数据模型时，RMI 非

常重要。但是，RMI-JRMP 绑定到 JavaClient 端。它是 Java 到 Java 的远程解决方案。

如果您需要基于 HTTP 的远程处理而且还依赖 Java 序列化，那么 Spring 的 HTTP 调用程序是一个不错的选择。它与 RMI 调用程序共享基本的基础结构，但使用 HTTP 作为传输。请注意，HTTP 调用程序不仅限于 Java 到 Java 远程处理，还不仅限于 Client 端和服务器端的 Spring。（后者也适用于非 RMI 接口的 Spring RMI 调用程序。）

在异类环境中运行时，Hessian 可能会提供重要的价值，因为它们明确允许使用非 JavaClient 端。但是，非 Java 支持仍然有限。已知的问题包括 Hibernate 对象的序列化以及延迟初始化的集合。如果您有这样的数据模型，请考虑使用 RMI 或 HTTP 调用程序而不是 Hessian。

JMS 可用于提供服务集群，并使 JMS 代理负责负载平衡，发现和自动故障转移。默认情况下，Java 序列化用于 JMS 远程处理，但是 JMS 提供程序可以使用其他机制进行线路格式化，例如 XStream，以使服务器可以用其他技术实现。

最后但并非最不重要的一点是，EJB 具有优于 RMI 的优势，因为它支持基于标准角色的身份验证和授权以及远程事务传播。尽管核心 Spring 并没有提供 RMI 调用程序或 HTTP 调用程序来支持安全上下文传播，但也有可能。Spring 仅提供用于插入第三方或自定义解决方案的钩子。

## 1.8. REST 端点

Spring 框架提供了两种选择来调用 REST 端点：

- [Using RestTemplate](#): 具有同步模板方法 API 的原始 Spring RESTClient 端。
- [WebClient](#): 一种非阻塞的，Reactive 的替代方案，它支持同步和异步以及流方案。

### ① Note

从 5.0 开始，无阻塞，响应式 `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。

## 1.8.1. 使用 RestTemplate

`RestTemplate` 通过 `HttpClient` 端库提供了更高级别的 API。它使在一行中轻松调用 REST 端点变得容易。它公开了以下几组重载方法：

表 1. `RestTemplate` 方法

Method group	Description
<code>getForObject</code>	通过 GET 检索表示形式。
<code>getForEntity</code>	通过使用 GET 检索 <code> ResponseEntity</code> (即状态, 标题和正文)。
<code>headForHeaders</code>	通过使用 HEAD 检索资源的所有 Headers。
<code>postForLocation</code>	通过使用 POST 创建新资源, 并从响应中返回 <code> Location</code> Headers。
<code>postForObject</code>	通过使用 POST 创建新资源, 并从响应中返回表示形式。
<code>postForEntity</code>	通过使用 POST 创建新资源, 并从响应中返回表示形式。
<code>put</code>	通过使用 PUT 创建或更新资源。
<code>patchForObject</code>	通过使用 PATCH 更新资源, 并从响应中返回表示形式。请注意, JDK <code> HttpURLConnection</code> 不支持 <code> PATCH</code> , 但是 Apache <code>HttpComponents</code> 和其他支持。

Method group	Description
<code>delete</code>	使用 <code>DELETE</code> 删除指定 URI 处的资源。
<code>optionsForAllow</code>	通过使用 <code>ALLOW</code> 检索资源的允许的 HTTP 方法。
<code>exchange</code>	前述方法的通用性强(且不那么固执)版本，可在需要时提供额外的灵 Active。它接受 <code>RequestEntity</code> (包括 HTTP 方法, URL, Headers 和正文作为 Importing), 并返回 <code>ResponseEntity</code> 。

这些方法允许使用 `ParameterizedTypeReference` 而不是 `Class` 来指定具有泛型的响应类型。

| `execute` | 执行请求的最通用方法，完全控制通过回调接口进行的请求准备和响应提取。

## Initialization

默认构造函数使用 `java.net.HttpURLConnection` 执行请求。您可以使用

`ClientHttpRequestFactory` 实现切换到其他 HTTP 库。内置支持以下内容：

- Apache HttpComponents
- Netty
- OkHttp

例如，要切换到 Apache HttpComponents，可以使用以下命令：

```
RestTemplate template = new RestTemplate(new HttpComponentsClientHttpRequestFactory());
```

每个 `ClientHttpRequestFactory` 都公开特定于基础 `HTTPClient` 端库的配置选项，例如用于凭证

, 连接池和其他详细信息。

## Tip

请注意, 访问表示错误的响应状态(例如 401)时, HTTP 请求的 `java.net` 实现可能引发异常。如果这是一个问题, 请切换到另一个 `HttpClient` 端库。

## URIs

许多 `RestTemplate` 方法都接受 URI 模板和 URI 模板变量, 它们可以作为 `String` 变量参数或 `Map<String, String>`。

下面的示例使用一个 `String` 变量参数:

```
String result = restTemplate.getForObject(  
    "http://example.com/hotels/{hotel}/bookings/{booking}", String.class, "42", "21")
```

以下示例使用 `Map<String, String>`:

```
Map<String, String> vars = Collections.singletonMap("hotel", "42");  
  
String result = restTemplate.getForObject(  
    "http://example.com/hotels/{hotel}/rooms/{hotel}", String.class, vars);
```

请注意, URI 模板是自动编码的, 如以下示例所示:

```
restTemplate.getForObject("http://example.com/hotel list", String.class);  
  
// Results in request to "http://example.com/hotel%20list"
```

您可以使用 `RestTemplate` 的 `uriTemplateHandler` 属性来自定义 URI 的 encodings。或者, 您可以准备 `java.net.URI` 并将其传递到接受 `URI` 的 `RestTemplate` 方法之一。

有关使用和编码 URI 的更多详细信息, 请参见[URI Links](#)。

## Headers

您可以使用 `exchange()` 方法来指定请求 Headers，如以下示例所示：

```
String uriTemplate = "http://example.com/hotels/{hotel}";
URI uri = UriComponentsBuilder.fromUriString(uriTemplate).build(42);

RequestEntity<Void> requestEntity = RequestEntity.get(uri)
    .header("MyRequestHeader", "MyValue")
    .build();

ResponseEntity<String> response = template.exchange(requestEntity, String.class);

String responseHeader = response.getHeaders().getFirst("MyResponseHeader");
String body = response.getBody();
```

您可以通过许多返回  `ResponseEntity` 的 `RestTemplate` 方法变体来获取响应 Headers。

## Body

在 `HttpMessageConverter` 的帮助下，通过 `RestTemplate` 方法传递和返回的对象将与原始内容进行转换。

在 POST 上，Importing 对象被序列化到请求主体，如以下示例所示：

```
URI location = template.postForLocation("http://example.com/people", person);
```

您无需显式设置请求的 Content-TypeHeaders。在大多数情况下，您可以找到基于源 `Object` 类型的兼容消息转换器，并且所选消息转换器会相应地设置 Content Type。如有必要，可以使用 `exchange` 方法显式提供 `Content-Type` 请求 Headers，从而影响选择哪个消息转换器。

在 GET 上，响应的主体反序列化为输出 `Object`，如以下示例所示：

```
Person person = restTemplate.getForObject("http://example.com/people/{id}", Person.class);
```

不需要明确设置请求的 `Accept` Headers。在大多数情况下，可以根据预期的响应类型找到兼容的消息转换器，这有助于填充 `Accept` Headers。如有必要，可以使用 `exchange` 方法显式提供 `Accept` Headers。

默认情况下，`RestTemplate` 注册所有内置的 [message converters](#)，具体取决于有助于确定存在哪些可选转换库的 Classpath 检查。您还可以将消息转换器设置为显式使用。

## Message Conversion

[与 Spring WebFlux 中的相同](#)

`spring-web` 模块包含 `HttpMessageConverter` 协定，用于通过 `InputStream` 和 `OutputStream` 读写 HTTP 请求和响应的正文。`HttpMessageConverter` 实例用于 Client 端(例如 `RestTemplate`)和服务器端(例如 Spring MVC REST 控制器)。

框架中提供了主要媒体(MIME)类型的具体实现，默认情况下，它们在 Client 端的 `RestTemplate` 和服务器端的 `RequestMethodHandlerAdapter` 注册(请参见[配置消息转换器](#))。

以下各节介绍了 `HttpMessageConverter` 的实现。对于所有转换器，都使用默认的媒体类型，但是您可以通过设置 `supportedMediaTypes` bean 属性来覆盖它。下表描述了每种实现：

表 2. `HttpMessageConverter` 实现

MessageConverter	Description
<code>StringHttpMessageConverter</code>	可以从 HTTP 请求和响应读取和写入 <code>String</code> 实例的 <code>HttpMessageConverter</code> 实现。默认情况下，此转换器支持所有文本媒体类型( <code>text/*</code> )并以 <code>Content-Type</code> 或 <code>text/plain</code> 写入。
<code>FormHttpMessageConverter</code>	可以从 HTTP 请求和响应中读取和写

MessageConverter	Description
	<p>入表单数据的 <code>HttpMessageConverter</code> 实现。默认情况下，此转换器读取和写入 <code>application/x-www-form-urlencoded</code> 媒体类型。从 <code>MultiValueMap&lt;String, String&gt;</code> 读取表格数据并将其写入。</p>
<code>ByteArrayHttpMessageConverter</code>	<p>可以从 HTTP 请求和响应读取和写入字节数组的 <code>HttpMessageConverter</code> 实现。默认情况下，此转换器支持所有媒体类型 (<code>*/*</code>) 并以 <code>Content-Type application/octet-stream</code> 写入。您可以通过设置 <code>supportedMediaTypes</code> 属性并覆盖 <code>getContentType(byte[])</code> 来覆盖它。</p>
<code>MarshallingHttpMessageConverter</code>	<p>可以使用包中的 Spring 的 <code>Marshaller</code> 和 <code>Unmarshaller</code> 抽象来读取和写入</p>

MessageConverter	Description
	<p>XML 的 <code>HttpMessageConverter</code> 实现。该转换器需要使用 <code>Marshaller</code> 和 <code>Unmarshaller</code> 才能使用。您可以通过构造函数或 <code>bean</code> 属性注入它们。默认情况下，此转换器支持 <code>text/xml</code> 和 <code>application/xml</code>。</p>
<code>MappingJackson2HttpMessageConverter</code>	<p>可以使用 Jackson 的 <code>ObjectMapper</code> 读取和写入 JSON 的 <code>HttpMessageConverter</code> 实现。您可以根据需要使用 Jackson 提供的 <code>Comments</code> 来自定义 JSONMap。当您需要进一步控制时(对于需要为特定类型提供自定义 JSON 序列化器 / 反序列化器的情况)，可以通过 <code>ObjectMapper</code> 属性注入自定义 <code>ObjectMapper</code>。默认情况下，此转换器支持 <code>application/json</code>。</p>
<code>MappingJackson2XmlHttpMessageConverter</code>	<p>可以使用 <a href="#">Jackson XML extensions</a> <code>XmlMapper</code> 读写 XML 的 <code>HttpMessageConverter</code> 实现。您可以根据需要使用 JAXB 或 Jackson</p>

MessageConverter	Description
	<p>提供的 Comments 来自定义 XMLMap。当您需要进一步控制时(对于需要为特定类型提供自定义 XML 序列化器/反序列化器的情况), 可以通过 <code>ObjectMapper</code> 属性注入自定义 <code>XmlMapper</code>。默认情况下, 此转换器支持 <code>application/xml</code>。</p>
<code>SourceHttpMessageConverter</code>	<p>可以从 HTTP 请求和响应中读取和写入 <code>javax.xml.transform.Source</code> 的 <code>HttpMessageConverter</code> 实现。仅支持 <code>DOMSource</code>, <code>SAXSource</code> 和 <code>StreamSource</code>。默认情况下, 此转换器支持 <code>text/xml</code> 和 <code>application/xml</code>。</p>
<code>BufferedImageHttpMessageConverter</code>	<p>可以从 HTTP 请求和响应中读取和写入 <code>java.awt.image.BufferedImage</code> 的 <code>HttpMessageConverter</code> 实现。该转换器读取和写入 Java I/O API 支持的媒体类型。</p>

您可以指定[Jackson JSON 视图](#)来仅序列化对象属性的一个子集，如以下示例所示：

```
MappingJacksonValue value = new MappingJacksonValue(new User("eric", "7!jd#h23"));
value.setSerializationView(User.WithoutPasswordView.class);

RequestEntity<MappingJacksonValue> requestEntity =
    RequestEntity.post(new URI("http://example.com/user")).body(value);

ResponseEntity<String> response = template.exchange(requestEntity, String.class);
```

## Multipart

要发送 Multipart 数据，您需要提供一个 `MultiValueMap<String, ?>`，其值要么是代表

Component 内容的 `Object` 实例，要么是代表 Component 内容和头的 `HttpEntity` 实例。

`MultipartBodyBuilder` 提供了一个方便的 API 来准备 Multipart 请求，如以下示例所示：

```
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`。Content Type 是根据要序列化的

`HttpMessageConverter` 自动确定的，对于 `Resource` 则根据文件 extensions 自动确定。如有必要，您可以通过重载的生成器 `part` 方法之一显式提供 `MediaType` 供每个 Component 使用。

`MultiValueMap` 准备就绪后，您可以将其传递给 `RestTemplate`，如以下示例所示：

```
MultipartBodyBuilder builder = ...;
template.postForObject("http://example.com/upload", builder.build(), Void.class);
```

如果 `MultiValueMap` 包含至少一个非 `String` 值，该值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`。当您使用 `MultipartBodyBuilder` 来确保 `HttpEntity` 包装器时，总是如此。

## 1.8.2. 使用 AsyncRestTemplate(不建议使用)

`AsyncRestTemplate` 已弃用。对于所有您可能考虑使用 `AsyncRestTemplate` 的用例，请改用 [WebClient](#)。

# 2.企业 JavaBeans(EJB)集成

作为轻量级容器，Spring 通常被认为是 EJB 的替代品。我们确实相信，对于许多(即使不是大多数)应用程序和用例，Spring 作为容器，结合其在事务，ORM 和 JDBC 访问领域的丰富支持功能，比通过 EJB 实现等效功能是更好的选择。容器和 EJB。

但是，请务必注意，使用 Spring 不会阻止您使用 EJB。实际上，Spring 使访问 EJB 以及在其中实现 EJB 和功能变得更加容易。另外，使用 Spring 访问 EJB 提供的服务可以使这些服务的实现稍后在本地 EJB，远程 EJB 或 POJO(普通旧 Java 对象)变体之间透明切换，而不必更改 Client 端代码。

在本章中，我们将研究 Spring 如何帮助您访问和实现 EJB。当访问 Stateless 会话 Bean(SLSB)时，Spring 提供了特殊的值，因此我们从讨论这个主题开始。

## 2.1. 访问 EJB

本节介绍如何访问 EJB。

### 2.1.1. Concepts

要在本地或远程 Stateless 会话 Bean 上调用方法，Client 端代码通常必须执行 JNDI 查找以获取(本地或远程)EJB Home 对象，然后对该对象使用 `create` 方法调用以获取实际的(本地或远程)Bean。)EJB 对象。然后在 EJB 上调用一种或多种方法。

为了避免重复的低级代码，许多 EJB 应用程序都使用服务定位器和业务委托模式。这些比在整个 Client 端代码中喷射 JNDI 查找要好，但是它们的常规实现有很多缺点：

- 通常，使用 EJB 的代码取决于 Service Locator 或 Business Delegate 单例，使其难以测试。
- 在不使用业务委托的情况下使用服务定位器模式的情况下，应用程序代码仍然最终必须在 EJB

`home` 上调用 `create()` 方法并处理产生的异常。因此，它仍然与 EJB API 和 EJB 编程模型的复杂性联系在一起。

- 实现业务委托模式通常会导致大量的代码重复，我们必须编写许多在 EJB 上调用相同方法的方法。

Spring 的方法是允许创建和使用代理对象(通常在 Spring 容器内配置)，这些代理对象充当无代码的业务委托。除非您在此类代码中实际添加了实际价值，否则您无需在手动编码的业务委托中编写另一个 Service Locator，另一个 JNDI 查找或重复方法。

## 2.1.2. 访问本地 SLSB

假设我们有一个需要使用本地 EJB 的 Web 控制器。我们遵循最佳实践，并使用 EJB 业务方法接口模式，以便 EJB 的本地接口扩展了非 EJB 特定的业务方法接口。我们将此业务方法界面称为 `MyComponent`。以下示例显示了这样的接口：

```
public interface MyComponent {  
    ...  
}
```

使用业务方法接口模式的主要原因之一是确保本地接口中的方法签名与 bean 实现类之间的同步是自动的。另一个原因是，如果有必要的话，以后可以使我们更轻松地切换到服务的 POJO(普通旧 Java 对象)实现。我们还需要实现本地 `home` 接口，并提供实现 `SessionBean` 和 `MyComponent` 业务方法接口的实现类。现在，将 Web 层控制器连接到 EJB 实现所需要的唯一 Java 编码是在控制器上公开 `MyComponent` 类型的 `setter` 方法。这会将引用另存为控制器中的实例变量。以下示例显示了如何执行此操作：

```
private MyComponent myComponent;  
  
public void setMyComponent(MyComponent myComponent) {  
    this.myComponent = myComponent;  
}
```

随后，我们可以在控制器中的任何业务方法中使用此实例变量。现在，假设我们从 Spring 容器中获取控制器对象，我们可以(在相同上下文中)配置 `LocalStatelessSessionProxyFactoryBean` 实

例，它是 EJB 代理对象。我们配置代理，并使用以下配置条目设置控制器的 `myComponent` 属性：

```
<bean id="myComponent"
      class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">
    <property name="jndiName" value="ejb/myBean"/>
    <property name="businessInterface" value="com.mycom.MyComponent"/>
</bean>

<bean id="myController" class="com.mycom.myController">
    <property name="myComponent" ref="myComponent"/>
</bean>
```

尽管没有强迫您使用 AOP 概念来欣赏结果，但仍要依靠 Spring AOP 框架在幕后进行大量工作。

`myComponent` bean 定义为 EJB 创建了一个代理，该代理实现了业务方法接口。EJB 本地主目录在启动时被缓存，因此只有一个 JNDI 查找。每次调用 EJB 时，代理都会在本地 EJB 上调用 `classname` 方法，并在 EJB 上调用相应的业务方法。

`myController` bean 定义将控制器类的 `myComponent` 属性设置为 EJB 代理。

或者(最好在许多此类代理定义的情况下)，考虑在 Spring 的“jee”名称空间中使用 `<jee:local-slsb>` 配置元素。以下示例显示了如何执行此操作：

```
<jee:local-slsb id="myComponent" jndi-name="ejb/myBean"
                 business-interface="com.mycom.MyComponent"/>

<bean id="myController" class="com.mycom.myController">
    <property name="myComponent" ref="myComponent"/>
</bean>
```

这种 EJB 访问机制极大地简化了应用程序代码。Web 层代码(或其他 EJBClient 端代码)与 EJB 的使用无关。要用 POJO 或模拟对象或其他测试存根替换该 EJB 引用，我们可以更改 `myComponent` bean 定义而无需更改任何 Java 代码。此外，作为应用程序的一部分，我们不必编写任何一行 JNDI 查找或其他 EJB 管道代码。

实际应用中的基准和经验表明，这种方法的性能开销(涉及目标 EJB 的反射调用)是最小的，并且在常规使用中是无法检测到的。请记住，无论如何我们都希望对 EJB 进行细粒度的调用，因为与应用程序服务器中的 EJB 基础结构相关联的成本很高。

关于 JNDI 查找有一个警告。在 bean 容器中，此类通常最好用作单例(没有理由使其成为原型)。但是，如果该 bean 容器预先实例化了单例(与各种 XML ApplicationContext 变体一样)，则在 EJB 容器加载目标 EJB 之前加载 bean 容器时可能会出现问题。这是因为 JNDI 查找是在此类的 init() 方法中执行的，然后进行了缓存，但是 EJB 尚未绑定到目标位置。解决方案是不预先实例化该工厂对象，而是让它在首次使用时创建。在 XML 容器中，您可以使用 lazy-init 属性来控制它。

尽管大多数 Spring 用户都不感兴趣，但是那些使用 EJB 进行编程 AOP 的用户可能希望查看 LocalSlsbInvokerInterceptor。

### 2.1.3. 访问远程 SLSB

除了使用 SimpleRemoteStatelessSessionProxyFactoryBean 或 <jee:remote-slsb> 配置元素外，访问远程 EJB 与访问本地 EJB 基本相同。当然，无论是否使用 Spring，远程调用语义都适用：调用另一台计算机上另一台 VM 中的对象上的方法时，有时在使用情况和故障处理方面必须区别对待。

与非 Spring 方法相比，Spring 的 EJBClient 端支持增加了另一个优势。通常，在本地或远程调用 EJB 之间轻松地来回切换 EJBClient 端代码是有问题的。这是因为远程接口方法必须声明它们抛出 RemoteException，而 Client 端代码必须对此进行处理，而本地接口方法则不需要。通常需要修改为需要移至远程 EJB 的本地 EJB 编写的 Client 端代码，以添加对远程异常的处理，为需要移至本地 EJB 的远程 EJB 编写的 Client 端代码可以保持不变，但可以执行以下操作：许多不必要的远程异常处理，或进行修改以删除该代码。使用 Spring 远程 EJB 代理，您不能在业务方法接口和实现 EJB 代码中声明任何抛出的 RemoteException，具有相同的远程接口(除了它确实抛出

RemoteException)，并且依靠代理来动态地处理两个接口就像它们一样。也就是说，Client 端代码不必处理已检查的 RemoteException 类。在 EJB 调用期间抛出的任何实际 RemoteException 都将重新抛出为未经检查的 RemoteAccessException 类，该类是 RuntimeException 的子类。然后，您可以在本地 EJB 或远程 EJB(甚至纯 Java 对象)实现之间随意切换目标服务，而无需了解或

关心 Client 端代码。当然，这是可选的：没有什么可以阻止您在业务界面中声明

`RemoteException`。

## 2.1.4. 访问 EJB 2.x SLSB 与 EJB 3 SLSB

通过 Spring 访问 EJB 2.x 会话 Bean 和 EJB 3 会话 Bean 在很大程度上是透明的。Spring 的 EJB 访问器(包括 `<jee:local-slsb>` 和 `<jee:remote-slsb>` 设施)在运行时透明地适应实际组件。它们会处理一个 Home 接口(如果找到)(EJB 2.x 样式)，或者在没有可用 Home 接口(EJB 3 样式)的情况下执行直接组件调用。

注意：对于 EJB 3 会话 Bean，您还可以有效地使用 `JndiObjectFactoryBean` / `<jee:jndi-lookup>`，因为公开了完全可用的组件引用以用于在那里的普通 JNDI 查找。定义明确的 `<jee:local-slsb>` 或 `<jee:remote-slsb>` 查找可提供一致且更明确的 EJB 访问配置。

## 3. JMS(Java 消息服务)

Spring 提供了一个 JMS 集成框架，该框架简化了 JMS API 的使用，就像 Spring 对 JDBC API 的集成一样。

JMS 可以大致分为两个功能区域，即消息的产生和使用。`JmsTemplate` 类用于消息生成和同步消息接收。对于类似于 Java EE 的消息驱动 bean 样式的异步接收，Spring 提供了许多消息侦听器容器，可用于创建消息驱动 POJO(MDP)。Spring 还提供了一种声明式方法来创建消息侦听器。

`org.springframework.jms.core` 软件包提供了使用 JMS 的核心功能。它包含 JMS 模板类，该类通过处理资源的创建和释放来简化 JMS 的使用，就像 `JdbcTemplate` 对于 JDBC 一样。Spring 模板类共有的设计原则是提供帮助器方法来执行常用操作，并且对于更复杂的用法，将处理任务的本质委托给用户实现的回调接口。JMS 模板遵循相同的设计。这些类提供了各种方便的方法，用于发送消息，同步使用消息以及向用户公开 JMS 会话和消息生成器。

`org.springframework.jms.support` 软件包提供 `JMSException` 翻译功能。转换将已检查的

`JMSEException` 层次结构转换为未检查的异常的镜像层次结构。如果存在选中的 `javax.jms.JMSEException` 的任何提供程序特定的子类，则将此异常包装在未选中的 `UncategorizedJmsException` 中。

`org.springframework.jms.support.converter` 包提供 `MessageConverter` 抽象以在 Java 对象和 JMS 消息之间进行转换。

`org.springframework.jms.support.destination` 包提供了用于 ManagementJMS 目的地的各种策略，例如为 JNDI 中存储的目的地提供服务定位器。

`org.springframework.jms.annotation` 软件包提供了必要的基础结构，以通过使用 `@JmsListener` 支持 `Comments` 驱动的侦听器端点。

`org.springframework.jms.config` 软件包为 `jms` 名称空间提供了解析器实现，并提供了 Java config 支持以配置侦听器容器和创建侦听器端点。

最后，`org.springframework.jms.connection` 软件包提供了适用于独立应用程序的 `ConnectionFactory` 的实现。它还包含用于 JMS 的 Spring `PlatformTransactionManager` 的实现(巧妙地名为 `JmsTransactionManager`)。这允许将 JMS 作为事务资源无缝集成到 Spring 的事务 Management 机制中。

## 3.1. 使用 Spring JMS

本节描述如何使用 Spring 的 JMS 组件。

### 3.1.1. 使用 JmsTemplate

`JmsTemplate` 类是 JMS 核心软件包中的中心类。由于它在发送或同步接收消息时处理资源的创建和释放，因此它简化了 JMS 的使用。

使用 `JmsTemplate` 的代码仅需要实现回调接口，即可为其提供明确定义的高级 Contract。当给

`JmsTemplate` 中的调用代码提供 `Session` 时, `MessageCreator` 回调接口会创建一条消息。为了允许更复杂地使用 JMS API, `SessionCallback` 提供了 JMS 会话, 而 `ProducerCallback` 公开了 `Session` 和 `MessageProducer` 对。

JMS API 公开了两种类型的发送方法, 一种采用交付模式, 优先级和生存时间作为服务质量(QOS)参数, 另一种不采用 QOS 参数并使用默认值。由于 `JmsTemplate` 有许多发送方法, 因此设置 QOS 参数已作为 bean 属性公开, 以避免重复发送方法。同样, 使用 `setReceiveTimeout` 属性设置同步接收调用的超时值。

某些 JMS 提供程序允许通过 `ConnectionFactory` 的配置来 Management 默认 QOS 值的设置。这样做的结果是, 对 `MessageProducer` 实例的 `send` 方法(`send(Destination destination, Message message)`)的调用使用与 JMS 规范中指定的 QOS 默认值不同的 QOS 默认值。为了提供对 QOS 值的一致 Management, 因此, 必须通过将布尔属性 `isExplicitQosEnabled` 设置为 `true` 来专门使 `JmsTemplate` 使用其自己的 QOS 值。

为方便起见, `JmsTemplate` 还公开了一个基本的请求-答复操作, 该操作允许发送消息并 `await` 为该操作一部分而创建的临时队列的答复。

### Tip

`JmsTemplate` 类的实例一旦配置便是线程安全的。这很重要, 因为这意味着您可以配置 `JmsTemplate` 的单个实例, 然后将该共享引用安全地注入多个协作者中。需要明确的是, `JmsTemplate` 是有状态的, 因为它保持对 `ConnectionFactory` 的引用, 但是此状态不是会话状态。

从 Spring Framework 4.1 开始, `JmsMessagingTemplate` 构建在 `JmsTemplate` 的基础上, 并提供了与消息传递抽象(即 `org.springframework.messaging.Message`)的集成。这使您可以创建以通用方式发送的消息。

### 3.1.2. Connections

`JmsTemplate` 要求引用 `ConnectionFactory`。`ConnectionFactory` 是 JMS 规范的一部分，并且是使用 JMS 的入口点。Client 端应用程序使用它作为工厂来创建与 JMS 提供程序的连接，并封装各种配置参数，其中许多是特定于供应商的，例如 SSL 配置选项。

当在 EJB 中使用 JMS 时，供应商提供 JMS 接口的实现，以便它们可以参与声明式事务 Management 并执行连接和会话的池化。为了使用此实现，Java EE 容器通常要求您在 EJB 或 Servlet 部署 Descriptors 中将 JMS 连接工厂声明为 `resource-ref`。为了确保在 EJB 内的 `JmsTemplate` 中使用这些功能，Client 端应用程序应确保引用了 `ConnectionFactory` 的托管实现。

#### 缓存消息传递资源

标准 API 涉及创建许多中间对象。要发送消息，请执行以下“API”遍历：

```
ConnectionFactory->Connection->Session->MessageProducer->send
```

在 `ConnectionFactory` 和 `Send` 操作之间，创建并销毁了三个中间对象。为了优化资源使用并提高性能，Spring 提供了 `ConnectionFactory` 的两种实现。

#### Using SingleConnectionFactory

Spring 提供了 `ConnectionFactory` 接口 `SingleConnectionFactory` 的实现，该接口在所有 `createConnection()` 调用中返回相同的 `Connection`，而忽略对 `close()` 的调用。这对于测试和独立环境很有用，因此同一连接可用于可能跨越任意数量事务的多个 `JmsTemplate` 调用。

`SingleConnectionFactory` 引用了通常来自 JNDI 的标准 `ConnectionFactory`。

#### Using CachingConnectionFactory

`CachingConnectionFactory` 扩展了 `SingleConnectionFactory` 的功能，并添加了 `Session`

, `MessageProducer` 和 `MessageConsumer` 实例的缓存。初始缓存大小设置为 `1`。您可以使用 `sessionCacheSize` 属性来增加缓存的会话数。请注意, 由于根据会话的确认模式缓存会话, 因此实际缓存的会话数大于该数量, 因此, 当 `sessionCacheSize` 设置为 `one` 时, 最多可以有四个缓存的会话实例(每个确认模式一个)。`MessageProducer` 和 `MessageConsumer` 实例被缓存在它们自己的会话中, 并且在缓存时还考虑了生产者和使用者的唯一属性。MessageProducers 将根据其目的地进行缓存。基于由目标, `selectors`, `noLocal` 传递标志和持久订阅名称(如果创建持久使用者)组成的键来缓存 MessageConsumers。

### 3.1.3. 目的地 Management

目标是 `ConnectionFactory` 实例, 是可以在 JNDI 中存储和检索的 `JMSManagement` 的对象。在配置 Spring 应用程序上下文时, 可以使用 JNDI `JndiObjectFactoryBean` factory 类或 `<jee:jndi-lookup>` 对对象对 JMS 目标的引用执行依赖项注入。但是, 如果应用程序中有大量目标, 或者 JMS 提供程序具有独特的高级目标 Management 功能, 则此策略通常很麻烦。这种高级目标 Management 的示例包括动态目标的创建或对目标的分层名称空间的支持。`JmsTemplate` 将目标名称的解析委托给实现 `DestinationResolver` 接口的 JMS 目标对象。

`DynamicDestinationResolver` 是 `JmsTemplate` 使用的默认实现, 并且可以解析动态目标。还提供 `JndiDestinationResolver` 充当 JNDI 中包含的目的地的服务定位器, 并且可以选择退回到 `DynamicDestinationResolver` 中包含的行为。

通常, 仅在运行时才知道 JMS 应用程序中使用的目的地, 因此, 在部署应用程序时无法通过 Management 方式创建。这通常是因为在交互的系统组件之间存在共享的应用程序逻辑, 这些组件根据已知的命名约定在运行时创建目标。即使创建动态目标不属于 JMS 规范的一部分, 但大多数供应商都提供了此功能。动态目标是使用用户定义的名称创建的, 该名称将它们与临时目标区分开来, 并且通常未在 JNDI 中注册。各个提供者之间用于创建动态目的地的 API 有所不同, 因为与目的地关联的属性是特定于供应商的。但是, 供应商有时会做出一个简单的实现选择, 就是忽略 JMS 规范中的警告, 并使用方法 `TopicSession createTopic(String topicName)` 或

`QueueSession` `createQueue(String queueName)` 方法来创建具有默认目标属性的新目标。根据供应商的实现，`DynamicDestinationResolver` 然后还可以创建一个物理目标，而不是仅解决一个物理目标。

布尔属性 `pubSubDomain` 用于在知道正在使用哪个 JMS 域的情况下配置 `JmsTemplate`。默认情况下，此属性的值为 `false`，指示将使用点对点域 `Queues`。此属性(由 `JmsTemplate` 使用)通过 `DestinationResolver` 接口的实现确定动态目标解析的行为。

您还可以通过属性 `defaultDestination` 将 `JmsTemplate` 配置为默认目标。默认目标是带有不引用特定目标的发送和接收操作。

### 3.1.4. 消息监听器容器

在 EJB 世界中，JMS 消息最常见的用途之一是驱动消息驱动的 bean(MDB)。Spring 提供了一种解决方案，以不将用户绑定到 EJB 容器的方式创建消息驱动的 POJO(MDP)。(有关 Spring 对 MDP 支持的详细介绍，请参见[异步接收：消息驱动的 POJO](#)。)从 Spring Framework 4.1 开始，可以用

`@JmsListener` `Comments` 端点方法，请参见[Comments 驱动的监听器端点](#)以获取更多详细信息。

消息监听器容器用于从 JMS 消息队列接收消息，并驱动注入到其中的 `MessageListener`。监听器容器负责消息接收的所有线程，并分派到监听器中进行处理。消息监听器容器是 MDP 与消息传递提供程序之间的中介，并负责注册接收消息，参与事务，资源获取和释放，异常转换等。这使您可以编写与接收消息(并可能对其进行响应)相关的(可能很复杂的)业务逻辑，并将样板 JMS 基础结构问题委托给框架。

Spring 附带了两个标准的 JMS 消息监听器容器，每个容器都有其专门的功能集。

- [SimpleMessageListenerContainer](#)
- [DefaultMessageListenerContainer](#)

#### Using SimpleMessageListenerContainer

此消息侦听器容器是两种标准样式中的简单容器。它在启动时创建固定数量的 JMS 会话和使用者，使用标准 JMS `MessageConsumer.setMessageListener()` 方法注册侦听器，并将其留给 JMS 提供者执行侦听器回调。此变体不允许动态适应运行时需求或参与外部 Management 的事务。在兼容性方面，它非常接近独立 JMS 规范的精神，但通常与 Java EE 的 JMS 限制不兼容。

### ⓘ Note

尽管 `SimpleMessageListenerContainer` 不允许参与外部 Management 的事务，但它支持本机 JMS 事务。要启用此功能，可以将 `sessionTransacted` 标志切换为 `true`，或者在 XML 名称空间中将 `acknowledge` 属性设置为 `transacted`。然后，从您的侦听器抛出的异常会导致回滚，并重新传递消息。或者，考虑使用 `CLIENT_ACKNOWLEDGE` 模式，该模式在出现异常的情况下也可以重新传送，但不使用事务处理的 `Session` 实例，因此在事务协议中不包括任何其他 `Session` 操作(例如发送响应消息)。

### 💡 Tip

默认的 `AUTO_ACKNOWLEDGE` 模式不能提供适当的可靠性保证。当侦听器执行失败时(由于提供者在侦听器调用之后会自动确认每条消息，没有异常要传播到提供者)，或者在侦听器容器关闭时(您可以通过设置 `acceptMessagesWhileStopping` 标志进行配置)，消息可能会丢失。确保出于可靠性需求(例如，为了可靠的队列处理和持久的主题订阅)使用事务处理的会话。

## Using DefaultMessageListenerContainer

大多数情况下使用此消息侦听器容器。与 `SimpleMessageListenerContainer` 相比，此容器变体允许动态适应运行时需求，并且能够参与外部 Management 的事务。配置为 `JtaTransactionManager` 时，每个接收到的消息都将注册到 XA 事务中。结果，处理可以利用 XA 事务语义。该侦听器容器在对 JMS 提供程序的低要求，高级功能(例如参与外部 Management 的事务)以及与 Java EE 环境的兼容性之间取得了良好的平衡。

您可以自定义容器的缓存级别。请注意，当未启用缓存时，将为每个消息接收创建一个新的连接和一个新的会话。将此内容与具有高负载的非持久订阅结合使用可能会导致消息丢失。在这种情况下，请确保使用适当的缓存级别。

当代理关闭时，此容器还具有可恢复的功能。默认情况下，简单的 `BackOff` 实现每五秒钟重试一次。您可以为更细粒度的恢复选项指定自定义 `BackOff` 实现。有关示例，请参见 [api-spring-framework/util/backoff/ExponentialBackOff.html](#) [ +516+ ]。

#### ❶ Note

与其同级 ([SimpleMessageListenerContainer](#))一样，

`DefaultMessageListenerContainer` 支持本机 JMS 事务，并允许自定义确认模式。如果对您的方案可行，则强烈建议在外部 Management 的事务上使用此方法，也就是说，如果 JVM 死亡，您可以偶尔接收重复消息。业务逻辑中的自定义重复消息检测步骤可以解决这种情况，例如以业务实体存在检查或协议表检查的形式。任何这样的安排都比其他安排更为有效：用 XA 事务包装整个处理过程(通过将 `DefaultMessageListenerContainer` 配置为 `JtaTransactionManager` )来覆盖 JMS 消息的接收以及消息侦听器中业务逻辑的执行(包括数据库操作等)。

#### 💡 Tip

默认的 `AUTO_ACKNOWLEDGE` 模式不能提供适当的可靠性保证。当侦听器执行失败时(由于提供者在侦听器调用之后会自动确认每条消息，没有异常要传播到提供者)，或者在侦听器容器关闭时(您可以通过设置 `acceptMessagesWhileStopping` 标志进行配置)，消息可能会丢失。确保出于可靠性需求(例如，为了可靠的队列处理和持久的主题订阅)使用事务处理的会话。

### 3.1.5. TransactionManagement

Spring 提供了一个 `JmsTransactionManager`，用于 Management 单个 JMS

`ConnectionFactory` 的事务。如[数据访问一章的事务 Management 部分](#)所述，这使 JMS 应用程序可以利用 Spring 的托管事务功能。`JmsTransactionManager` 执行本地资源事务，将来自指定 `ConnectionFactory` 的 JMS 连接/会话对绑定到线程。`JmsTemplate` 自动检测此类 `Transaction` 资源并相应地对其进行操作。

在 Java EE 环境中，`ConnectionFactory` 汇集了 `Connection` 和 `Session` 实例，因此可以有效地在事务之间重用这些资源。在独立环境中，使用 Spring 的 `SingleConnectionFactory` 会导致共享 JMS `Connection`，每个事务都有自己的独立 `Session`。或者，考虑使用提供程序专用的池适配器，例如 ActiveMQ 的 `PooledConnectionFactory` 类。

您还可以将 `JmsTemplate` 与 `JtaTransactionManager` 和具有 XA 功能的 JMS `ConnectionFactory` 结合使用来执行分布式事务。请注意，这需要使用 JTA 事务 Management 器以及正确的 XA 配置的 `ConnectionFactory`。（检查您的 Java EE 服务器或 JMS 提供程序的文档。）

使用 JMS API 从 `Connection` 创建 `Session` 时，在托管和非托管事务环境中重用代码可能会造成混淆。这是因为 JMS API 只有一个工厂方法来创建 `Session`，并且它需要事务和确认模式的值。在托管环境中，设置这些值是环境的事务基础结构的责任，因此，供应商对 JMS `Connection` 的包装将忽略这些值。在非托管环境中使用 `JmsTemplate` 时，可以通过使用属性 `sessionTransacted` 和 `sessionAcknowledgeMode` 来指定这些值。当您将 `PlatformTransactionManager` 与 `JmsTemplate` 一起使用时，始终为模板提供事务 `JMS Session`。

## 3.2. 发送信息

`JmsTemplate` 包含许多发送消息的便捷方法。发送方法使用 `javax.jms.Destination` 对象指定目标，其他方法通过在 JNDI 查找中使用 `String` 指定目标。不使用目标参数的 `send` 方法使用默认目标。

以下示例使用 `MessageCreator` 回调从提供的 `Session` 对象创建文本消息：

```
import javax.jms.ConnectionFactory;
import javax.jms.JMSEException;
import javax.jms.Message;
import javax.jms.Queue;
import javax.jms.Session;

import org.springframework.jms.core.MessageCreator;
import org.springframework.jms.core.JmsTemplate;

public class JmsQueueSender {

    private JmsTemplate jmsTemplate;
    private Queue queue;

    public void setConnectionFactory(ConnectionFactory cf) {
        this.jmsTemplate = new JmsTemplate(cf);
    }

    public void setQueue(Queue queue) {
        this.queue = queue;
    }

    public void simpleSend() {
        this.jmsTemplate.send(this.queue, new MessageCreator() {
            public Message createMessage(Session session) throws JMSEException {
                return session.createTextMessage("hello queue world");
            }
        });
    }
}
```

在前面的示例中，`JmsTemplate` 是通过将引用传递给 `ConnectionFactory` 来构造的。或者，提供零参数构造函数和 `connectionFactory`，它们可用于以 JavaBean 样式(使用 `BeanFactory` 或纯 Java 代码)构造实例。或者，考虑从 Spring 的 `JmsGatewaySupport` 便捷 Base Class 派生，该 Base Class 为 JMS 配置提供了预构建的 bean 属性。

`send(String destinationName, MessageCreator creator)` 方法使您可以使用目标的字符串名称发送消息。如果这些名称已在 JNDI 中注册，则应将模板的 `destinationResolver` 属性设置为 `JndiDestinationResolver` 的实例。

如果您创建了 `JmsTemplate` 并指定了默认目的地，则 `send(MessageCreator c)` 会向该目的地发送一条消息。

### 3.2.1. 使用消息转换器

为了方便域模型对象的发送，`JmsTemplate` 具有各种发送方法，这些方法将 Java 对象作为消息数据内容的参数。`JmsTemplate` 中的重载方法 `convertAndSend()` 和 `receiveAndConvert()` 方法将转换过程委托给 `MessageConverter` 接口的实例。该接口定义了一个简单的协定，可以在 Java 对象和 JMS 消息之间进行转换。默认实现(`SimpleMessageConverter`)支持 `String` 和 `TextMessage`，`byte[]` 和 `BytesMessage` 以及 `java.util.Map` 和 `MapMessage` 之间的转换。通过使用转换器，您和您的应用程序代码可以专注于通过 JMS 发送或接收的业务对象，而不必担心如何将其表示为 JMS 消息。

沙箱当前包含一个 `MapMessageConverter`，它使用反射在 JavaBean 和 `MapMessage` 之间进行转换。您可能自己实现的其他流行实现选择是使用现有 XML 编组程序包(例如 JAXB, Castor 或 XStream)创建代表对象的 `TextMessage` 的转换器。

为了适应消息属性，Headers 和正文的设置，这些属性通常不能封装在转换器类中，因此 `MessagePostProcessor` 接口使您可以在转换消息之后但在发送消息之前对其进行访问。以下示例显示了将 `java.util.Map` 转换为消息后如何修改消息头和属性：

```
public void sendWithConversion() {
    Map map = new HashMap();
    map.put("Name", "Mark");
    map.put("Age", new Integer(47));
    jmsTemplate.convertAndSend("testQueue", map, new MessagePostProcessor() {
        public Message postProcessMessage(Message message) throws JMSException {
            message.setIntProperty("AccountID", 1234);
            message.setJMSCorrelationID("123-00001");
            return message;
        }
    });
}
```

这将导致以下形式的消息：

```
MapMessage={
  Header={
    ... standard headers ...
  CorrelationID={123-00001}
```

```
    }
    Properties={
        AccountID={Integer:1234}
    }
    Fields={
        Name={String:Mark}
        Age={Integer:47}
    }
}
```

### 3.2.2. 使用 SessionCallback 和 ProducerCallback

尽管发送操作涵盖了许多常见的使用场景，但是您有时可能希望对 JMS `Session` 或 `MessageProducer` 执行多个操作。`SessionCallback` 和 `ProducerCallback` 分别暴露 JMS `Session` 和 `Session / MessageProducer` 对。`JmsTemplate` 上的 `execute()` 方法执行这些回调方法。

## 3.3. 接收讯息

这描述了如何在 Spring 中使用 JMS 接收消息。

### 3.3.1. 同步接收

虽然 JMS 通常与异步处理相关联，但是您可以同步使用消息。重载的 `receive(..)` 方法提供了此功能。在同步接收期间，调用线程将阻塞，直到消息可用为止。这可能是危险的操作，因为调用线程可能会无限期地被阻塞。`receiveTimeout` 属性指定接收者在放弃 `await` 消息之前应该 `await` 多长时间。

### 3.3.2. 异步接收：消息驱动的 POJO

#### iNote

Spring 还通过使用 `@JmsListener` `Comments` 支持带 `Comments` 的侦听器端点，并提供了开放的基础结构以编程方式注册端点。到目前为止，这是设置异步接收器的最便捷方法。有关更多详细信息，请参见[启用侦听器端点 `Comments`](#)。

消息驱动 POJO(MDP)以类似于 EJB 世界中的消息驱动 Bean(MDB)的方式充当 JMS 消息的接收者。

MDP 的一个限制(但请参见[Using MessageListenerAdapter](#))是它必须实现

`javax.jms.MessageListener` 接口。请注意，如果您的 POJO 在多个线程上接收消息，则重要的  
是要确保您的实现是线程安全的。

以下示例显示了 MDP 的简单实现：

```
import javax.jms.JMSEException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

public class ExampleListener implements MessageListener {

    public void onMessage(Message message) {
        if (message instanceof TextMessage) {
            try {
                System.out.println(((TextMessage) message).getText());
            }
            catch (JMSEException ex) {
                throw new RuntimeException(ex);
            }
        }
        else {
            throw new IllegalArgumentException("Message must be of type TextMessage");
        }
    }
}
```

实现 `MessageListener` 之后，就可以创建消息侦听器容器了。

以下示例显示如何定义和配置 Spring 附带的消息侦听器容器之一(在本例中为

`DefaultMessageListenerContainer` )：

```
<!-- this is the Message Driven POJO (MDP) -->
<bean id="messageListener" class="jmsexample.ExampleListener"/>

<!-- and this is the message listener container -->
<bean id="jmsContainer" class="org.springframework.jms.listener.DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="destination" ref="destination"/>
    <property name="messageListener" ref="messageListener"/>
</bean>
```

有关每个实现所支持功能的完整说明，请参见各种消息侦听器容器(所有这些都实现  
[MessageListenerContainer](#))的 Spring javadoc。

### 3.3.3. 使用 SessionAwareMessageListener 接口

`SessionAwareMessageListener` 接口是特定于 Spring 的接口，它提供与 JMS `MessageListener` 接口相似的协定，但还使消息处理方法可以访问从中接收 `Message` 的 JMS `Session`。以下 Lists 显示了 `SessionAwareMessageListener` 接口的定义：

```
package org.springframework.jms.listener;

public interface SessionAwareMessageListener {

    void onMessage(Message message, Session session) throws JMSException;
}
```

如果希望 MDP 能够响应任何接收到的消息(使用 `onMessage(Message, Session)` 方法中提供的 `Session`)，则可以选择让 MDP 实现此接口(优先于标准 JMS `MessageListener` 接口)。Spring 附带的所有消息侦听器容器实现都支持实现 `MessageListener` 或 `SessionAwareMessageListener` 接口的 MDP。实现 `SessionAwareMessageListener` 的类带有警告，然后通过接口将它们绑定到 Spring。是否使用它的选择完全由您作为应用程序开发人员或架构师来决定。

请注意，`SessionAwareMessageListener` 接口的 `onMessage(...)` 方法抛出 `JMSException`。与标准 JMS `MessageListener` 接口相反，在使用 `SessionAwareMessageListener` 接口时，Client 端代码负责处理所有引发的异常。

### 3.3.4. 使用 MessageListenerAdapter

`MessageListenerAdapter` 类是 Spring 异步消息传递支持中的最后一个组件。简而言之，它使您几乎可以将任何类公开为 MDP(尽管存在一些约束)。

考虑以下接口定义：

```
public interface MessageDelegate {
```

```
void handleMessage(String message);  
void handleMessage(Map message);  
void handleMessage(byte[] message);  
void handleMessage(Serializable message);  
}
```

请注意，尽管该接口既未扩展 `MessageListener` 也未扩展 `SessionAwareMessageListener` 接口，但仍可以通过使用 `MessageListenerAdapter` 类将其用作 MDP。还要注意如何根据各种 `Message` 类型的内容来强类型化各种消息处理方法，它们可以接收和处理。

现在考虑 `MessageDelegate` 接口的以下实现：

```
public class DefaultMessageDelegate implements MessageDelegate {  
    // implementation elided for clarity...  
}
```

特别要注意的是，`MessageDelegate` 接口(`DefaultMessageDelegate` 类)的先前实现完全没有 JMS 依赖性。这确实是一个 POJO，我们可以通过以下配置将其变成 MDP：

```
<!-- this is the Message Driven POJO (MDP) -->  
<bean id="messageListener" class="org.springframework.jms.listener.adapter.MessageListenerAdapter">  
    <constructor-arg>  
        <bean class="jmsexample.DefaultMessageDelegate"/>  
    </constructor-arg>  
</bean>  
  
<!-- and this is the message listener container... -->  
<bean id="jmsContainer" class="org.springframework.jms.listener.DefaultMessageListenerContainer">  
    <property name="connectionFactory" ref="connectionFactory"/>  
    <property name="destination" ref="destination"/>  
    <property name="messageListener" ref="messageListener"/>  
</bean>
```

下一个示例显示另一个 MDP，它只能处理接收 JMS `TextMessage` 消息。请注意，实际上是如何将消息处理方法称为 `receive` (`MessageListenerAdapter` 中的消息处理方法的名称默认为 `handleMessage`)，但是它是可配置的(如本节后面所述)。还请注意 `receive(..)` 方法是如何强类型 Importing 的，以便仅接收和响应 JMS `TextMessage` 消息。以下 Lists 显示了

`TextMessageDelegate` 接口的定义：

```
public interface TextMessageDelegate {  
    void receive(TextMessage message);  
}
```

下面的 Lists 显示了实现 `TextMessageDelegate` 接口的类：

```
public class DefaultTextMessageDelegate implements TextMessageDelegate {  
    // implementation elided for clarity...  
}
```

话务员 `MessageListenerAdapter` 的配置如下：

```
<bean id="messageListener" class="org.springframework.jms.listener.adapter.MessageListene  
<constructor-arg>  
    <bean class="jmsexample.DefaultTextMessageDelegate"/>  
</constructor-arg>  
<property name="defaultListenerMethod" value="receive"/>  
<!-- we don't want automatic message context extraction -->  
<property name="messageConverter">  
    <null/>  
</property>  
</bean>
```

请注意，如果 `messageListener` 收到的类型不是 `TextMessage` 的 JMS `Message`，则会抛出 `IllegalStateException` (随后将其吞咽)。 `MessageListenerAdapter` 类的另一个功能是，如果处理程序方法返回非无效值，则自动发送回响应 `Message` 的功能。考虑以下接口和类：

```
public interface ResponsiveTextMessageDelegate {  
    // notice the return type...  
    String receive(TextMessage message);  
}
```

```
public class DefaultResponsiveTextMessageDelegate implements ResponsiveTextMessageDeleg  
    // implementation elided for clarity...  
}
```

如果将 `DefaultResponsiveTextMessageDelegate` 与 `MessageListenerAdapter` 结合使用，则

从 `'receive(..)'` 方法的执行返回的任何非 `null` 值都将(在默认配置中)转换为 `TextMessage`。

然后将结果 `TextMessage` 发送到原始 `Message` 的 JMS `Reply-To` 属性或

`MessageListenerAdapter` 上设置的默认 `Destination` (如果已配置)的 JMS `Reply-To` 属性中

定义的 `Destination` (如果存在)。如果未找到 `Destination`，则会引发

`InvalidDestinationException` (请注意，该异常不会被吞没，并且会在调用堆栈中传播)。

### 3.3.5. 处理事务中的消息

在事务中调用消息侦听器仅需要重新配置侦听器容器。

您可以通过侦听器容器定义上的 `sessionTransacted` 标志激活本地资源事务。然后，每个消息侦

听器调用都在活动的 JMS 事务中运行，并且在侦听器执行失败的情况下回退消息接收。(通过

`SessionAwareMessageListener`)发送响应消息是同一本地事务的一部分，但是任何其他资源操

作(例如数据库访问)都是独立运行的。这通常需要在侦听器实现中进行重复消息检测，以解决数据  
库处理已提交但消息处理未能提交的情况。

考虑以下 bean 定义：

```
<bean id="jmsContainer" class="org.springframework.jms.listener.DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="destination" ref="destination"/>
    <property name="messageListener" ref="messageListener"/>
    <property name="sessionTransacted" value="true"/>
</bean>
```

要参与外部 Management 的事务，您需要配置一个事务 Management 器并使用支持外部

Management 的事务(通常为 `DefaultMessageListenerContainer`)的侦听器容器。

要为 XA 事务参与配置消息侦听器容器，您需要配置 `JtaTransactionManager` (默认情况下，它委

派给 Java EE 服务器的事务子系统)。请注意，底层的 JMS `ConnectionFactory` 必须具有 XA 功能

， 并已向您的 JTA 事务协调器正确注册。(检查 Java EE 服务器的 JNDI 资源配置)这使消息接收和

(例如)数据库访问成为同一事务的一部分(具有统一的提交语义，但以 XA 事务日志开销为代价)。

以下 bean 定义创建一个事务 Management 器：

```
<bean id="transactionManager" class="org.springframework.transaction.jta.JtaTransaction
```

然后，我们需要将其添加到我们之前的容器配置中。容器负责其余的工作。以下示例显示了如何执行此操作：

```
<bean id="jmsContainer" class="org.springframework.jms.listener.DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="destination" ref="destination"/>
    <property name="messageListener" ref="messageListener"/>
    <property name="transactionManager" ref="transactionManager"/> (1)
</bean>
```

- (1) 我们的 TransactionManager。

## 3.4. 支持 JCA 消息端点

从 2.5 版开始，Spring 还提供了对基于 JCA 的 `MessageListener` 容器的支持。

`JmsMessageEndpointManager` 尝试根据提供者的 `ResourceAdapter` 类名自动确定

`ActivationSpec` 类名。因此，通常可以提供 Spring 的通用 `JmsActivationSpecConfig`，如以

下示例所示：

```
<bean class="org.springframework.jms.listener.endpoint.JmsMessageEndpointManager">
    <property name="resourceAdapter" ref="resourceAdapter"/>
    <property name="activationSpecConfig">
        <bean class="org.springframework.jms.listener.endpoint.JmsActivationSpecConfig">
            <property name="destinationName" value="myQueue"/>
        </bean>
    </property>
    <property name="messageListener" ref="myMessageListener" />
</bean>
```

或者，您可以使用给定的 `ActivationSpec` 对象设置 `JmsMessageEndpointManager`。

`ActivationSpec` 对象也可以来自 JNDI 查找(使用 `<jee:jndi-lookup>`)。以下示例显示了如何执行此操作：

```
<bean class="org.springframework.jms.listener.endpoint.JmsMessageEndpointManager">
    <property name="resourceAdapter" ref="resourceAdapter"/>
```

```
<property name="activationSpec">
    <bean class="org.apache.activemq.ra.ActiveMQActivationSpec">
        <property name="destination" value="myQueue" />
        <property name="destinationType" value="javax.jms.Queue" />
    </bean>
</property>
<property name="messageListener" ref="myMessageListener" />
</bean>
```

使用 Spring 的 `ResourceAdapterFactoryBean`，您可以在本地配置目标 `ResourceAdapter`，如以下示例所示：

```
<bean id="resourceAdapter" class="org.springframework.jca.support.ResourceAdapterFactoryBean">
    <property name="resourceAdapter">
        <bean class="org.apache.activemq.ra.ActiveMQResourceAdapter">
            <property name="serverUrl" value="tcp://localhost:61616" />
        </bean>
    </property>
    <property name="workManager">
        <bean class="org.springframework.jca.work.SimpleTaskWorkManager" />
    </property>
</bean>
```

指定的 `WorkManager` 也可以指向特定于环境的线程池-通常通过 `SimpleTaskWorkManager` 实例的 `asyncTaskExecutor` 属性来指向。如果您碰巧使用多个适配器，请考虑为所有 `ResourceAdapter` 实例定义一个共享线程池。

在某些环境(例如 WebLogic 9 或更高版本)中，您可以(通过使用 `<jee:jndi-lookup>`)从 JNDI 获取整个 `ResourceAdapter` 对象。然后，基于 Spring 的消息侦听器可以与服务器托管的 `ResourceAdapter` 进行交互，该服务器也使用服务器的内置 `WorkManager`。

有关更多详细信息，请参见[JmsMessageEndpointManager](#), [JmsActivationSpecConfig](#)和[ResourceAdapterFactoryBean](#)的 javadoc。

Spring 还提供了与 JMS 无关的通用 JCA 消息端点 Management 器：

`org.springframework.jca.endpoint.GenericMessageEndpointManager`。该组件允许使用任何消息侦听器类型(例如 CCI `MessageListener`)和任何特定于提供程序的 `ActivationSpec` 对象。请参阅 JCA 提供程序的文档以了解连接器的实际功能，并请参阅

[GenericMessageEndpointManager](#) javadoc 以获取特定于 Spring 的配置详细信息。

#### iNote

基于 JCA 的消息端点 Management 与 EJB 2.1 消息驱动 Bean 非常相似。它使用相同的基础资源提供者 Contract。与 EJB 2.1 MDB 一样，您也可以在 Spring 上下文中使用 JCA 提供程序支持的任何消息侦听器接口。尽管如此，Spring 仍为 JMS 提供了明确的“便利”支持，因为 JMS 是 JCA 端点 Management 协定中最常用的端点 API。

## 3.5. Comments 驱动的侦听器端点

异步接收消息的最简单方法是使用带 Comments 的侦听器端点基础结构。简而言之，它使您可以将托管 Bean 的方法公开为 JMS 侦听器端点。以下示例显示了如何使用它：

```
@Component
public class MyService {

    @JmsListener(destination = "myDestination")
    public void processOrder(String data) { ... }
}
```

前面示例的想法是，只要 `javax.jms.Destination` `myDestination` 上有消息可用，就相应地调用 `processOrder` 方法(在这种情况下，使用 JMS 消息的内容，类似于[MessageListenerAdapter](#)提供的内容)。

带 Comments 的终结点基础结构通过使用 `JmsListenerContainerFactory` 在幕后为每种带 Comments 的方法创建一个消息侦听器容器。此类容器未针对应用程序上下文进行注册，但可以通过使用 `JmsListenerEndpointRegistry` bean 进行轻松定位以进行 Management。

#### Tip

`@JmsListener` 是 Java 8 上的可重复 Comments，因此您可以通过向其添加其他 `@JmsListener` 声明来将多个 JMS 目标与同一方法相关联。

### 3.5.1. 启用侦听器端点 Comments

要启用对 `@JmsListener` `Comments` 的支持，可以将 `@EnableJms` 添加到 `@Configuration` 类之一，如以下示例所示：

```
@Configuration  
@EnableJms  
public class AppConfig {  
  
    @Bean  
    public DefaultJmsListenerContainerFactory jmsListenerContainerFactory() {  
        DefaultJmsListenerContainerFactory factory = new DefaultJmsListenerContainerFactory();  
        factory.setConnectionFactory(connectionFactory());  
        factory.setDestinationResolver(destinationResolver());  
        factory.setSessionTransacted(true);  
        factory.setConcurrency("3-10");  
        return factory;  
    }  
}
```

默认情况下，基础结构将查找名为 `jmsListenerContainerFactory` 的 bean 作为工厂用来创建消息侦听器容器的源。在这种情况下(并忽略了 JMS 基础结构设置)，您可以使用三个线程的核心轮询大小和十个线程的最大池大小来调用 `processOrder` 方法。

您可以自定义用于每个 `Comments` 的侦听器容器工厂，也可以通过实现

`JmsListenerConfigurer` 接口来配置显式默认值。仅当至少一个端点在没有特定容器工厂的情况下注册时，才需要使用默认值。有关详细信息和示例，请参见实现 [JmsListenerConfigurer](#) 的类的 javadoc。

如果您更喜欢 [XML configuration](#)，则可以使用 `<jms:annotation-driven>` 元素，如以下示例所示：

```
<jms:annotation-driven/>  
  
<bean id="jmsListenerContainerFactory"  
      class="org.springframework.jms.config.DefaultJmsListenerContainerFactory">  
    <property name="connectionFactory" ref="connectionFactory"/>  
    <property name="destinationResolver" ref="destinationResolver"/>  
    <property name="sessionTransacted" value="true"/>  
    <property name="concurrency" value="3-10"/>  
</bean>
```

### 3.5.2. 程序化端点注册

`JmsListenerEndpoint` 提供 JMS 端点的模型，并负责为该模型配置容器。除了 `JmsListener` `Comments` 检测到的端点外，该基础结构还允许您以编程方式配置端点。以下示例显示了如何执行此操作：

```
@Configuration
@EnableJms
public class AppConfig implements JmsListenerConfigurer {

    @Override
    public void configureJmsListeners(JmsListenerEndpointRegistrar registrar) {
        SimpleJmsListenerEndpoint endpoint = new SimpleJmsListenerEndpoint();
        endpoint.setId("myJmsEndpoint");
        endpoint.setDestination("anotherQueue");
        endpoint.setMessageListener(message -> {
            // processing
        });
        registrar.registerEndpoint(endpoint);
    }
}
```

在前面的示例中，我们使用了 `SimpleJmsListenerEndpoint`，它提供了实际的 `MessageListener` 进行调用。但是，您也可以构建自己的端点变体来描述自定义调用机制。

请注意，您可以完全跳过 `@JmsListener` 的使用，而只能通过 `JmsListenerConfigurer` 以编程方式注册您的端点。

### 3.5.3. 带 `Comments` 的端点方法签名

到目前为止，我们已经在端点中注入了一个简单的 `String`，但实际上它可以具有非常灵活的方法签名。在以下示例中，我们将其重写为使用自定义 Headers 注入 `Order`：

```
@Component
public class MyService {

    @JmsListener(destination = "myDestination")
    public void processOrder(Order order, @Header("order_type") String orderType) {
        ...
    }
}
```

您可以在 JMS 监听器端点中注入的主要元素如下：

- 原始的 `javax.jms.Message` 或其任何子类(前提是它与传入的消息类型匹配)。
- `javax.jms.Session` 用于对本机 JMS API 的可选访问(例如，用于发送自定义回复)。
- `org.springframework.messaging.Message` 代表传入的 JMS 消息。请注意，此消息同时包含自定义 Headers 和标准 Headers(由 `JmsHeaders` 定义)。
- `@Header`-带 Comments 的方法参数，用于提取特定的 Headers 值，包括标准的 JMSHeaders。
- 一个带有 `@Headers` Comments 的参数，还必须可以将其分配给 `java.util.Map` 才能访问所有 Headers。
- 不是受支持的类型(`Message` 或 `Session`)之一的非 Comments 元素被视为有效负载。您可以通过对用 `@Payload` Comments 参数来使其明确。您还可以通过添加额外的 `@Valid` 来启用验证。

注入 Spring 的 `Message` 抽象的能力特别有用，它可以受益于存储在特定于传输的消息中的所有信息，而无需依赖于特定于传输的 API。以下示例显示了如何执行此操作：

```
@JmsListener(destination = "myDestination")
public void processOrder(Message<Order> order) { ... }
```

`DefaultMessageHandlerMethodFactory` 提供了方法参数的处理，您可以进一步对其进行自定义以支持其他方法参数。您也可以在那里自定义转换和验证支持。

例如，如果我们想在处理 `Order` 之前确保其有效，则可以使用 `@Valid` Comments 有效负载并配置必要的验证器，如以下示例所示：

```
@Configuration
@EnableJms
public class AppConfig implements JmsListenerConfigurer {
```

```

@Override
public void configureJmsListeners(JmsListenerEndpointRegistrar registrar) {
    registrar.setMessageHandlerMethodFactory(myJmsHandlerMethodFactory());
}

@Bean
public DefaultMessageHandlerMethodFactory myHandlerMethodFactory() {
    DefaultMessageHandlerMethodFactory factory = new DefaultMessageHandlerMethodFactory();
    factory.setValidator(myValidator());
    return factory;
}
}

```

### 3.5.4. 反应 Management

[MessageListenerAdapter](#) 中的现有支持已使您的方法具有非 `void` 返回类型。在这种情况下，调用的结果将封装在 `javax.jms.Message` 中，该 `javax.jms.Message` 要么在原始消息的 `JMSReplyTo` 头中指定的目标中发送，要么在侦听器上配置的默认目标中发送。现在，您可以使用消息传递抽象的 `@SendTo` `Comments` 设置该默认目标。

假设我们的 `processOrder` 方法现在应该返回 `OrderStatus`，我们可以将其编写为自动发送响应，如以下示例所示：

```

@JmsListener(destination = "myDestination")
@SendTo("status")
public OrderStatus processOrder(Order order) {
    // order processing
    return status;
}

```

#### Tip

如果您有几种带有 `@JmsListener` `Comments` 的方法，则还可以将 `@SendTo` `Comments` 放置在类级别以共享默认的答复目标。

如果需要以与传输无关的方式设置其他 Headers，则可以使用类似于以下方法来返回 `Message`：

```

@JmsListener(destination = "myDestination")

```

```
@SendTo("status")
public Message<OrderStatus> processOrder(Order order) {
    // order processing
    return MessageBuilder
        .withPayload(status)
        .setHeader("code", 1234)
        .build();
}
```

如果需要在运行时计算响应目标，则可以将响应封装在 `JmsResponse` 实例中，该实例还提供要在运行时使用的目标。我们可以如下重写前一个示例：

```
@JmsListener(destination = "myDestination")
public JmsResponse<Message<OrderStatus>> processOrder(Order order) {
    // order processing
    Message<OrderStatus> response = MessageBuilder
        .withPayload(status)
        .setHeader("code", 1234)
        .build();
    return JmsResponse.forQueue(response, "status");
}
```

最后，如果您需要为响应指定一些 QoS 值，例如优先级或生存时间，则可以相应地配置 `JmsListenerContainerFactory`，如以下示例所示：

```
@Configuration
@EnableJms
public class AppConfig {

    @Bean
    public DefaultJmsListenerContainerFactory jmsListenerContainerFactory() {
        DefaultJmsListenerContainerFactory factory = new DefaultJmsListenerContainerFactory();
        factory.setConnectionFactory(connectionFactory());
        QosSettings replyQosSettings = new QosSettings();
        replyQosSettings.setPriority(2);
        replyQosSettings.setTimeToLive(10000);
        factory.setReplyQosSettings(replyQosSettings);
        return factory;
    }
}
```

## 3.6. JMS 命名空间支持

Spring 提供了一个 XML 名称空间来简化 JMS 配置。要使用 JMS 命名空间元素，您需要引用 JMS 模式，如以下示例所示：

```

<?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:jms="http://www.springframework.org/schema/jms" (1)
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans http://www.springframework.org/
        http://www.springframework.org/schema/jms http://www.springframework.org/sch
        <!-- bean definitions here -->
</beans>

```

- (1) 引用 JMS 模式。

命名空间由三个顶级元素组成：`<annotation-driven/>`，`<listener-container/>` 和`<jca-listener-container/>`。`<annotation-driven/>` 启用[Comments 驱动的侦听器端点](#)的使用。`<listener-container/>` 和`<jca-listener-container/>` 定义共享侦听器容器配置，并且可以包含`<listener/>` 子元素。以下示例显示了两个侦听器的基本配置：

```

<jms:listener-container>

    <jms:listener destination="queue.orders" ref="orderService" method="placeOrder"/>

    <jms:listener destination="queue.confirmations" ref="confirmationLogger" method="log"/>

</jms:listener-container>

```

前面的示例等效于创建两个不同的侦听器容器 bean 定义和两个不同的[MessageListenerAdapter](#) bean 定义，如[Using MessageListenerAdapter](#)所示。除了前面示例中显示的属性之外，`listener` 元素还可以包含多个可选属性。下表描述了所有可用属性：

表 3. JMS`<listener>`元素的属性

Attribute	Description
<code>id</code>	托管侦听器容器的 Bean 名称。如果未指定，则会自动生成一个 Bean 名称。

Attribute	Description
<code>destination</code> (必填)	该侦听器的目标名称，通过 <code>DestinationResolver</code> 策略解析。
<code>ref</code> (必填)	处理程序对象的 <code>bean</code> 名称。
<code>method</code>	要调用的处理程序方法的名称。如果 <code>ref</code> 属性指向 <code>MessageListener</code> 或 Spring <code>SessionAwareMessageListener</code> ，则可以省略此属性。
<code>response-destination</code>	向其发送响应消息的默认响应目标的名称。如果请求消息中不包含 <code>JMSReplyTo</code> 字段，则应用此方法。此目标的类型由侦听器容器的 <code>response-destination-type</code> 属性确定。请注意，这仅适用于具有返回值的侦听器方法，为此，每个结果对象都将转换为响应消息。
<code>subscription</code>	持久订阅的名称(如果有)。
<code>selector</code>	此侦听器的可选消息 <code>selectors</code> 。
<code>concurrency</code>	要启动此侦听器的并发会话或使用者的数量。该值可以是表示最大数的简单数字(例如 <code>5</code> )，也可以是指示下限和上限的范围(例如 <code>3-5</code> )。请注意，指定的最小值仅是一个提示，在运行时可能会被忽略。默认值为容器提供的值。

`<listener-container/>` 元素还接受几个可选属性。这允许自定义各种策略(例如

`taskExecutor` 和 `destinationResolver`) 以及基本的 JMS 设置和资源引用。通过使用这些属性，您可以定义高度自定义的侦听器容器，同时仍然受益于命名空间的便利性。

您可以通过指定要通过 `factory-id` 属性公开的 Bean 的 `id` 来自动将此类设置公开为 `JmsListenerContainerFactory`，如以下示例所示：

```
<jms:listener-container connection-factory="myConnectionFactory"
    task-executor="myTaskExecutor"
    destination-resolver="myDestinationResolver"
    transaction-manager="myTransactionManager"
    concurrency="10">

    <jms:listener destination="queue.orders" ref="orderService" method="placeOrder"/>

    <jms:listener destination="queue.confirmations" ref="confirmationLogger" method="log"/>
</jms:listener-container>
```

下表描述了所有可用属性。有关各个属性的更多详细信息，请参见

[AbstractMessageListenerContainer](#) 的类级 javadoc 及其具体子类。Javadoc 还讨论了事务选择和消息重新交付方案。

表 4. JMS`<listener-container>`元素的属性

Attribute	Description
<code>container-type</code>	此侦听器容器的类型。可用选项为 <code>default</code> ， <code>simple</code> ， <code>default102</code> 或 <code>simple102</code> (默认选项为 <code>default</code> )。
<code>class</code>	自定义侦听器容器实现类，作为完全限定的类名。根据 <code>container-type</code> 属性，默认值为 Spring 的标准 <code>DefaultMessageListenerContainer</code> 或 <code>SimpleMessageListenerContainer</code> 。

Attribute	Description
<code>factory-id</code>	<p>用指定的 <code>id</code> 公开此元素定义为 <code>JmsListenerContainerFactory</code> 的设置，以便可以将其与其他端点重用。</p>
<code>connection-factory</code>	<p>对 JMS <code>ConnectionFactory</code> bean 的引用(默认 bean 名称为 <code>connectionFactory</code>)。</p>
<code>task-executor</code>	<p>对 JMS 倾听器调用程序的 Spring <code>TaskExecutor</code> 的引用。</p>
<code>destination-resolver</code>	<p>对解决 JMS <code>Destination</code> 实例的 <code>DestinationResolver</code> 策略的引用。</p>
<code>message-converter</code>	<p>对将 JMS 消息转换为倾听器方法参数的 <code>MessageConverter</code> 策略的引用。默认值为 <code>SimpleMessageConverter</code>。</p>
<code>error-handler</code>	<p>对 <code>ErrorHandler</code> 策略的引用，该策略用于处理 <code>MessageListener</code> 执行期间可能发生的任何未捕获的异常。</p>
<code>destination-type</code>	<p>此倾听器的 JMS 目标类型：<code>queue</code>，<code>topic</code>，<code>durableTopic</code>，<code>sharedTopic</code> 或 <code>sharedDurableTopic</code>。这可能会启用容器的 <code>pubSubDomain</code>，<code>subscriptionDurable</code> 和 <code>subscriptionShared</code> 属性。默认值为 <code>queue</code> (将禁用这三个属性)。</p>

Attribute	Description
<code>response-destination-type</code>	响应的 JMS 目标类型: <code>queue</code> 或 <code>topic</code> 。默认值为 <code>destination-type</code> 属性的值。
<code>client-id</code>	此侦听器容器的 <code>JMSClient</code> 端 ID。使用持久订阅时必须指定它。
<code>cache</code>	JMS 资源的缓存级别: <code>none</code> , <code>connection</code> , <code>session</code> , <code>consumer</code> 或 <code>auto</code> 。默认情况下( <code>auto</code> )，缓存级别实际上是 <code>consumer</code> ，除非已指定外部事务 Management 器。在这种情况下，有效的默认值为 <code>none</code> (假设 Java EE 风格的事务 Management，其中给定的 <code>ConnectionFactory</code> 是 XA 感知的池)。
<code>acknowledge</code>	本机 JMS 确认模式: <code>auto</code> , <code>client</code> , <code>dups-ok</code> 或 <code>transacted</code> 。值 <code>transacted</code> 激活本地处理的 <code>Session</code> 。或者，您可以指定 <code>transaction-manager</code> 属性，如表中稍后所述。默认值为 <code>auto</code> 。
<code>transaction-manager</code>	对外部 <code>PlatformTransactionManager</code> 的引用(通常是基于 XA 的事务协调器，例如 Spring 的 <code>JtaTransactionManager</code> )。如果未指定，则使用本机确认(请参见 <code>acknowledge</code> 属性)。
<code>concurrency</code>	每个侦听器启动的并发会话或使用者的数量。它可以是表示最大数

Attribute	Description
	<p>的简单数字(例如 <code>5</code>)，也可以是指示下限和上限的范围(例如 <code>3-5</code>)。请注意，指定的最小值只是一个提示，在运行时可能会被忽略。默认值为 <code>1</code>。如果是主题侦听器或队列 Sequences 很重要，则应将并发限制为 <code>1</code>。考虑将其提高到一般队列。</p>
<code>prefetch</code>	<p>加载到单个会话中的最大消息数。请注意，增加此数字可能会导致并发 Consumer 饥饿。</p>
<code>receive-timeout</code>	<p>用于接听电话的超时时间(以毫秒为单位)。默认值为 <code>1000</code> (一秒)。<code>-1</code> 表示没有超时。</p>
<code>back-off</code>	<p>指定用于计算两次恢复尝试间隔的 <code>BackOff</code> 实例。如果 <code>BackOffExecution</code> 实现返回 <code>BackOffExecution#STOP</code>，则侦听器容器不会进一步尝试恢复。设置此属性时，将忽略 <code>recovery-interval</code> 值。默认值为 <code>FixedBackOff</code>，间隔为 5000 毫秒(即五秒)。</p>
<code>recovery-interval</code>	<p>指定两次恢复尝试之间的时间间隔(以毫秒为单位)。它提供了一种方便的方法来创建具有指定间隔的 <code>FixedBackOff</code>。有关更多恢复选项，请考虑改为指定 <code>BackOff</code> 实例。缺省值为 5000 毫秒(即 5 秒)。</p>
<code>phase</code>	<p>此容器应在其中启动和停止的生命周期阶段。值越低，此容器启动的越早，而容器停止的越晚。默认值为 <code>Integer.MAX_VALUE</code>，这</p>

Attribute	Description
	意味着容器将尽可能晚地启动，并尽快停止。

配置具有 `jms` 模式支持的基于 JCA 的侦听器容器非常相似，如以下示例所示：

```
<jms:jca-listener-container resource-adapter="myResourceAdapter"
    destination-resolver="myDestinationResolver"
    transaction-manager="myTransactionManager"
    concurrency="10">

    <jms:listener destination="queue.orders" ref="myMessageListener" />

</jms:jca-listener-container>
```

下表描述了 JCA 变体的可用配置选项：

表 5. `JMS<jca-listener-container>` 元素的属性

Attribute	Description
<code>factory-id</code>	用指定的 <code>id</code> 公开此元素定义为 <code>JmsListenerContainerFactory</code> 的设置，以便可以将其与其他端点重用。
<code>resource-adapter</code>	对 JCA <code>ResourceAdapter</code> bean 的引用(默认 bean 名称为 <code>resourceAdapter</code> )。
<code>activation-spec-factory</code>	对 <code>JmsActivationSpecFactory</code> 的引用。默认设置是自动检测 JMS 提供程序及其 <code>ActivationSpec</code> 类(请参见 <a href="#">DefaultJmsActivationSpecFactory</a> )。

Attribute	Description
<code>destination-resolver</code>	对解决 JMS <code>Destinations</code> 的 <code>DestinationResolver</code> 策略的引用。
<code>message-converter</code>	对将 JMS 消息转换为侦听器方法参数的 <code>MessageConverter</code> 策略的引用。默认值为 <code>SimpleMessageConverter</code> 。
<code>destination-type</code>	此侦听器的 JMS 目标类型: <code>queue</code> , <code>topic</code> , <code>durableTopic</code> , <code>sharedTopic</code> 或 <code>sharedDurableTopic</code> 。这可能会启用容器的 <code>pubSubDomain</code> , <code>subscriptionDurable</code> 和 <code>subscriptionShared</code> 属性。默认值为 <code>queue</code> (将禁用这三个属性)。
<code>response-destination-type</code>	响应的 JMS 目标类型: <code>queue</code> 或 <code>topic</code> 。默认值为 <code>destination-type</code> 属性的值。
<code>client-id</code>  <code>acknowledge</code>	此侦听器容器的 <code>JMSClient</code> 端 ID。使用持久订阅时需要指定它。  本机 JMS 确认模式: <code>auto</code> , <code>client</code> , <code>dups-ok</code> 或 <code>transacted</code> 。值 <code>transacted</code> 激活本地处理的 <code>Session</code> 。或者, 您可以指定后面描述的 <code>transaction-manager</code> 属性。默认值为 <code>auto</code> 。

Attribute	Description
<code>transaction-manager</code>	对 Spring <code>JtaTransactionManager</code> 或 <code>javax.transaction.TransactionManager</code> 的引用，用于为每个传入消息启动 XA 事务。如果未指定，则使用本机确认(请参见 <code>acknowledge</code> 属性)。
<code>concurrency</code>	每个侦听器启动的并发会话或使用者的数量。它可以是表示最大数的简单数字(例如 <code>5</code> )，也可以是指示下限和上限的范围(例如 <code>3-5</code> )。请注意，指定的最小值只是一个提示，通常在运行时使用 JCA 侦听器容器时将被忽略。预设值为 <code>1</code> .
<code>prefetch</code>	加载到单个会话中的最大消息数。请注意，增加此数字可能会导致并发 Consumer 饥饿。

## 4. JMX

Spring 中的 JMX(JavaManagement 扩展)支持提供的功能使您可以轻松，透明地将 Spring 应用程序集成到 JMX 基础结构中。

### JMX?

本章不是 JMX 的介绍。它没有试图解释为什么您可能要使用 JMX。如果您不熟悉 JMX，请参阅本章末尾的[Further Resources](#)。

具体来说，Spring 的 JMX 支持提供了四个核心功能：

- 将任何 Spring bean 自动注册为 JMX MBean。
- 一种用于控制 beanManagement 界面的灵活机制。

- 通过远程 JSR-160 连接器以声明方式公开 MBean。
- 本地和远程 MBean 资源的简单代理。

这些功能旨在在不将应用程序组件耦合到 Spring 或 JMX 接口和类的情况下起作用。实际上，在大多数情况下，您的应用程序类无需了解 Spring 或 JMX 即可利用 Spring JMX 功能。

## 4.1. 将您的 Bean 导出到 JMX

Spring 的 JMX 框架的核心类是 `MBeanExporter`。此类负责获取您的 Spring bean 并向 JMX `MBeanServer` 注册它们。例如，考虑以下类：

```
package org.springframework.jmx;

public class JmxTestBean implements IJmxTestBean {

    private String name;
    private int age;
    private boolean isSuperman;

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public int add(int x, int y) {
        return x + y;
    }

    public void dontExposeMe() {
        throw new RuntimeException();
    }
}
```

要将此 Bean 的属性和方法公开为 MBean 的属性和操作，可以在配置文件中配置 `MBeanExporter` 类的实例并传入 Bean，如以下示例所示：

```

<beans>
    <!-- this bean must not be lazily initialized if the exporting is to happen -->
    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter" lazy-init="true">
        <property name="beans">
            <map>
                <entry key="bean:name=testBean1" value-ref="testBean"/>
            </map>
        </property>
    </bean>
    <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>
</beans>

```

前面的配置片段中相关的 bean 定义是 `exporter` bean。 `beans` 属性准确地告诉 `MBeanExporter` 您必须将哪个 bean 导出到 JMX `MBeanServer`。在默认配置中，`beans` Map 中每个条目的键用作相应条目值所引用的 Bean 的 `ObjectName`。您可以按照 [控制您的 Bean 的 ObjectName 实例](#) 中所述更改此行为。

使用此配置，`testBean` bean 在 `ObjectName` `bean:name=testBean1` 下作为 MBean 公开。

默认情况下，bean 的所有 `public` 属性都作为属性公开，所有 `public` 方法(从 `Object` 类继承的方法除外)都作为操作公开。

### iNote

`MBeanExporter` 是 `Lifecycle` bean(请参阅[启动和关机回调](#))。默认情况下，MBean 在应用程序生命周期中尽可能晚地导出。您可以通过设置 `autoStartup` 标志来配置导出发生的 `phase` 或禁用自动注册。

## 4.1.1. 创建一个 MBeanServer

[preceding section](#) 中显示的配置假定该应用程序正在一个(并且只有一个) `MBeanServer` 已运行的环境中运行。在这种情况下，Spring 尝试找到正在运行的 `MBeanServer` 并将您的 bean 注册到该服务器(如果有)。当您的应用程序在具有自己的 `MBeanServer` 的容器(例如 Tomcat 或 IBM

WebSphere)中运行时，此行为很有用。

但是，此方法在独立环境中或在不提供 `MBeanServer` 的容器中运行时无用。为了解决这个问题，您可以通过在配置中添加 `org.springframework.jmx.support.MBeanServerFactoryBean` 类的实例来声明性地创建 `MBeanServer` 实例。您还可以通过将 `MBeanExporter` 实例的 `server` 属性的值设置为 `MBeanServerFactoryBean` 返回的 `MBeanServer` 值来确保使用特定的 `MBeanServer`，如以下示例所示：

```
<beans>
```

```
    <bean id="mbeanServer" class="org.springframework.jmx.support.MBeanServerFactoryBean">
        <!--
            this bean needs to be eagerly pre-instantiated in order for the exporting to occur;
            this means that it must not be marked as lazily initialized
        -->
        <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
            <property name="beans">
                <map>
                    <entry key="bean:name=testBean1" value-ref="testBean"/>
                </map>
            </property>
            <property name="server" ref="mbeanServer"/>
        </bean>

        <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
            <property name="name" value="TEST"/>
            <property name="age" value="100"/>
        </bean>
    </beans>
```

在前面的示例中，`MBeanServer` 的实例由 `MBeanServerFactoryBean` 创建，并通过 `server` 属性提供给 `MBeanExporter`。当您提供自己的 `MBeanServer` 实例时，`MBeanExporter` 不会尝试查找正在运行的 `MBeanServer` 并使用提供的 `MBeanServer` 实例。为了使其正常工作，您必须在 Classpath 上具有 JMX 实现。

#### 4.1.2. 重用现有的 MBeanServer

如果未指定服务器，则 `MBeanExporter` 尝试自动检测正在运行的 `MBeanServer`。在大多数仅使用一个 `MBeanServer` 实例的环境中，这是可行的。但是，当存在多个实例时，导出器可能选择了

错误的服务器。在这种情况下，应使用 `MBeanServer` `agentId` 指示要使用的实例，如以下示例所示：

```
<beans>
    <bean id="mbeanServer" class="org.springframework.jmx.support.MBeanServerFactoryBea
        <!-- indicate to first look for a server -->
        <property name="locateExistingServerIfPossible" value="true" />
        <!-- search for the MBeanServer instance with the given agentId -->
        <property name="agentId" value="MBeanServer_instance_agentId"/>
    </bean>
    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="server" ref="mbeanServer"/>
        ...
    </bean>
</beans>
```

对于平台或现有 `MBeanServer` 具有通过查找方法检索到的动态(或未知) `agentId` 的情况，应使用 [factory-method](#)，如以下示例所示：

```
<beans>
    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="server">
            <!-- Custom MBeanServerLocator -->
            <bean class="platform.package.MBeanServerLocator" factory-method="locateMBea
        </property>
    </bean>

    <!-- other beans here -->

</beans>
```

### 4.1.3. 延迟初始化的 MBean

如果为 Bean 配置了 `MBeanExporter`，并且也为延迟初始化配置了该 `MBeanExporter`，则 `MBeanExporter` 不会破坏该协定，并且避免实例化该 Bean。相反，它向 `MBeanServer` 注册了一个代理，并推迟从容器获取 Bean，直到对该代理进行第一次调用为止。

### 4.1.4. 自动注册 MBean

通过 `MBeanExporter` 导出并且已经是有效 MBean 的所有 bean 都将直接在 `MBeanServer` 上注册，而无需 Spring 的进一步干预。通过将 `autodetect` 属性设置为 `true`，可以使

`MBeanExporter` 自动检测 MBean，如以下示例所示：

```
<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="autodetect" value="true"/>
</bean>

<bean name="spring:mbean=true" class="org.springframework.jmx.export.TestDynamicMBean"/>
```

在前面的示例中，名为 `spring:mbean=true` 的 bean 已经是有效的 JMX MBean，并由 Spring 自动注册。缺省情况下，自动检测到 JMX 注册的 bean 的 Bean 名称用作 `ObjectName`。您可以覆盖此行为，如[控制您的 Bean 的 ObjectName 实例](#)中所述。

#### 4.1.5. 控制注册行为

考虑以下情形：Spring `MBeanExporter` 尝试通过使用 `ObjectName bean:name=testBean1` 向 `MBeanServer` 注册 `MBean`。如果 `MBean` 实例已经在同一 `ObjectName` 下注册，则默认行为是失败(并抛出 `InstanceAlreadyExistsException`)。

您可以精确控制将 `MBean` 注册到 `MBeanServer` 时发生的情况。当注册过程发现 `MBean` 已经在同一 `ObjectName` 下注册时，Spring 的 JMX 支持允许三种不同的注册行为来控制注册行为。下表总结了这些注册行为：

表 6. 注册行为

Registration behavior	Explanation
<code>FAIL_ON_EXISTING</code>	这是默认的注册行为。如果 <code>MBean</code> 实例已经在同一 <code>ObjectName</code> 下注册，则不注册正在注册的 <code>MBean</code> ，并抛出 <code>InstanceAlreadyExistsException</code> 。现有的 <code>MBean</code> 不受影响。

Registration behavior	Explanation
<code>IGNORE_EXISTING</code>	<p>如果 <code>MBean</code> 实例已经在同一 <code>ObjectName</code> 下注册，则正在注册的 <code>MBean</code> 不会被注册。现有的 <code>MBean</code> 不受影响，并且不会抛出 <code>Exception</code>。这在多个应用程序要在共享 <code>MBeanServer</code> 中共享一个公共 <code>MBean</code> 的设置中很有用。</p>
<code>REPLACE_EXISTING</code>	<p>如果 <code>MBean</code> 实例已经在同一 <code>ObjectName</code> 下注册，那么先前已注册的现有 <code>MBean</code> 将被取消注册，而新的 <code>MBean</code> 会在其位置注册(新的 <code>MBean</code> 有效地替换先前的实例)。</p>

上表中的值定义为 `RegistrationPolicy` 类上的枚举。如果要更改默认注册行为，则需要将 `MBeanExporter` 定义上的 `registrationPolicy` 属性的值设置为这些值之一。

下面的示例显示如何从默认注册行为更改为 `REPLACE_EXISTING` 行为：

```

<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="beans">
            <map>
                <entry key="bean:name=testBean1" value-ref="testBean"/>
            </map>
        </property>
        <property name="registrationPolicy" value="REPLACE_EXISTING" />
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST" />
        <property name="age" value="100" />
    </bean>

</beans>

```

## 4.2. 控制 Bean 的 Management 接口

在 preceding section 的示例中，您几乎无法控制 bean 的 Management 界面。每个导出 bean 的所有 public 属性和方法分别作为 JMX 属性和操作公开。为了对已导出的 bean 的哪些属性和方法实际上作为 JMX 属性和操作公开而进行更细粒度的控制，Spring JMX 提供了一种全面且可扩展的机制来控制 bean 的 Management 接口。

#### 4.2.1. 使用 MBeanInfoAssembler 接口

在幕后，MBeanExporter 代表

org.springframework.jmx.export assembler.MBeanInfoAssembler 接口的实现，该接口负责定义每个公开的 bean 的 Management 接口。默认实现

org.springframework.jmx.export assembler.SimpleReflectiveMBeanInfoAssembler 定义了一个 Management 接口，该接口公开了所有公共属性和方法(如您在前面几节的示例中所看到的)。Spring 提供了 MBeanInfoAssembler 接口的两个附加实现，使您可以使用源级元数据或任何任意接口来控制生成的 Management 接口。

#### 4.2.2. 使用源级元数据：JavaComments

通过使用 MetadataMBeanInfoAssembler，您可以通过使用源级元数据来定义 bean 的 Management 接口。元数据的读取由

org.springframework.jmx.export.metadata.JmxAttributeSource 接口封装。Spring JMX 提供了一个使用 JavaComments 的默认实现，即

org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource。您必须为 MetadataMBeanInfoAssembler 配置 JmxAttributeSource 接口的实现实例，才能使其正常运行(没有默认值)。

要标记要导出到 JMX 的 bean，应使用 ManagedResource Comments 对 bean 类进行 Comments。您必须使用 @注解标记要公开的每个方法，并使用 ManagedAttribute 注解标记希望公开的每个属性。标记属性时，可以省略 getter 或 setter 的 Comments，以分别创建只写或只读属性。

## iNote

带有 `ManagedResource` Comments 的 Bean 必须是公共的，公开操作或属性的方法也必须是公共的。

以下示例显示了我们在[创建一个 MBeanServer](#)中使用的 `IJmxTestBean` 类的带 Comments 版本：

```
package org.springframework.jmx;

import org.springframework.jmx.export.annotation.ManagedResource;
import org.springframework.jmx.export.annotation.ManagedOperation;
import org.springframework.jmx.export.annotation.ManagedAttribute;

@ManagedResource(
    objectName="bean:name=testBean4",
    description="My Managed Bean",
    log=true,
    logFile="jmx.log",
    currencyTimeLimit=15,
    persistPolicy="OnUpdate",
    persistPeriod=200,
    persistLocation="foo",
    persistName="bar")
public class AnnotationTestBean implements IJmxTestBean {

    private String name;
    private int age;

    @ManagedAttribute(description="The Age Attribute", currencyTimeLimit=15)
    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @ManagedAttribute(description="The Name Attribute",
        currencyTimeLimit=20,
        defaultValue="bar",
        persistPolicy="OnUpdate")
    public void setName(String name) {
        this.name = name;
    }

    @ManagedAttribute(defaultValue="foo", persistPeriod=300)
    public String getName() {
        return name;
    }

    @ManagedOperation(description="Add two numbers")
    @ManagedOperationParameters({
        @ManagedOperationParameter(name = "x", description = "The first number"),
        @ManagedOperationParameter(name = "y", description = "The second number")})
}
```

```

public int add(int x, int y) {
    return x + y;
}

public void dontExposeMe() {
    throw new RuntimeException();
}

}

```

在前面的示例中，您可以看到 `JmxTestBean` 类标记有 `ManagedResource` Comments，并且此 `ManagedResource` Comments 配置了一组属性。这些属性可用于配置 `MBeanExporter` 生成的 MBean 的各个方面，稍后将在[源级元数据类型](#)中进行详细说明。

`age` 和 `name` 属性都带有 `ManagedAttribute` Comments，但是，在 `age` 属性的情况下，仅标记了吸气剂。这导致这两个属性都作为属性包含在 Management 界面中，但是 `age` 属性是只读的。

最后，`add(int, int)` 方法带有 `ManagedOperation` 属性标记，而 `dontExposeMe()` 方法则没有。当您使用 `MetadataMBeanInfoAssembler` 时，这将导致 Management 界面仅包含一个操作(`add(int, int)`)。

以下配置显示了如何配置 `MBeanExporter` 以使用 `MetadataMBeanInfoAssembler`：

```

<beans>
    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="assembler" ref="assembler"/>
        <property name="namingStrategy" ref="namingStrategy"/>
        <property name="autodetect" value="true"/>
    </bean>

    <bean id="jmxAttributeSource"
          class="org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource">
        <!-- will create management interface using annotation metadata -->
        <bean id="assembler"
              class="org.springframework.jmx.export.assembler.MetadataMBeanInfoAssembler">
            <property name="attributeSource" ref="jmxAttributeSource"/>
        </bean>

        <!-- will pick up the ObjectName from the annotation -->
        <bean id="namingStrategy"
              class="org.springframework.jmx.export.naming.MetadataNamingStrategy">

```

```

<property name="attributeSource" ref="jmxAttributeSource"/>
</bean>

<bean id="testBean" class="org.springframework.jmx.AnnotationTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
</bean>
</beans>

```

在前面的示例中，已为 `MetadataMBeanInfoAssembler` bean 配置了

`AnnotationJmxDataSource` 类的实例，并通过汇编程序属性将其传递给 `MBeanExporter`

。这是为 Spring 公开的 MBean 利用元数据驱动的 Management 接口所需要的全部。

### 4.2.3. 源级元数据类型

下表描述了可在 Spring JMX 中使用的源级别元数据类型：

表 7. 源级元数据类型

Purpose	Annotation	Annotation Type
将 <code>Class</code> 的所有实例标记为 JMX 托管资源。	<code>@ManagedResource</code>	Class
将方法标记为 JMX 操作。	<code>@ManagedOperation</code>	Method
将一个 getter 或 setter 标记为 JMX 属性的一半。	<code>@ManagedAttribute</code>	方法(仅 getter 和 setter)

Purpose	Annotation	Annotation Type
定义操作参数的描述。	@ManagedOperationParameter 和 @ManagedOperationParameters	Method

下表描述了可在这些源级元数据类型上使用的配置参数：

表 8. 源级元数据参数

Parameter	Description	Applies to
ObjectName	<p><code>MetadataNamingStrategy</code></p> <p>用于确定托管资源的 <code>ObjectName</code>。</p>	ManagedResource
description	<p>设置资源、属性或操作的友好描述。</p>	<code>ManagedResource</code> ， <code>ManagedAttribute</code> ， <code>ManagedOperation</code> 或 <code>ManagedOperationParameter</code>
currencyTimeLimit	<p>设置 <code>currencyTimeLimit</code> Descriptors 字段的值。</p>	<code>ManagedResource</code> 或 <code>ManagedAttribute</code>
defaultValue	<p>设置 <code>defaultValue</code> Descriptors 字段的值。</p>	ManagedAttribute
log	设置 <code>log</code> Descriptors 字段	ManagedResource

Parameter	Description	Applies to
	的值。	
<code>logFile</code>	设置 <code>logFile</code> Descriptors 字段的值。	<code>ManagedResource</code>
<code>persistPolicy</code>	设置 <code>persistPolicy</code> Descriptors 字段的值。	<code>ManagedResource</code>
<code>persistPeriod</code>	设置 <code>persistPeriod</code> Descriptors 字段的值。	<code>ManagedResource</code>
<code>persistLocation</code>	设置 <code>persistLocation</code> Descriptors 字段的值。	<code>ManagedResource</code>
<code>persistName</code>	设置 <code>persistName</code> Descriptors 字段的值。	<code>ManagedResource</code>
<code>name</code>	设置操作参数的显示名称。	<code>ManagedOperationParameter</code>
<code>index</code>	设置操作参数的索引。	<code>ManagedOperationParameter</code>

#### 4.2.4. 使用 AutodetectCapableMBeanInfoAssembler 接口

为了进一步简化配置，Spring 包含了 `AutodetectCapableMBeanInfoAssembler` 接口，该接口扩展了 `MBeanInfoAssembler` 接口以添加对自动检测 MBean 资源的支持。如果用

`AutodetectCapableMBeanInfoAssembler` 的实例配置 `MBeanExporter`，则可以在包含 Bean 的情况下“投票”，以暴露给 JMX。

`AutodetectCapableMBeanInfo` 接口的唯一实现是 `MetadataMBeanInfoAssembler`，该投票表决以包括所有带有 `ManagedResource` 属性标记的 bean。在这种情况下，默认方法是将 Bean 名称用作 `ObjectName`，这将导致与以下内容类似的配置：

```
<beans>
```

```
  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <!-- notice how no 'beans' are explicitly configured here -->
    <property name="autodetect" value="true"/>
    <property name="assembler" ref="assembler"/>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

  <bean id="assembler" class="org.springframework.jmx.export.assembler.MetadataMBeanInfoAssembler">
    <property name="attributeSource">
      <bean class="org.springframework.jmx.export.annotation.AnnotationJmxAttributeBuilder"/>
    </property>
  </bean>
</beans>
```

请注意，在上述配置中，没有将任何 bean 传递给 `MBeanExporter`。但是，`JmxTestBean` 仍被注册，因为它已被标记为 `ManagedResource` 属性，并且 `MetadataMBeanInfoAssembler` 检测到该问题并对其进行投票以将其包括在内。这种方法的唯一问题是 `JmxTestBean` 的名称现在具有商业意义。您可以通过更改控制您的 Bean 的 `ObjectName` 实例中定义的 `ObjectName` 创建的默认行为来解决此问题。

#### 4.2.5. 使用 Java 接口定义 Management 接口

除了 `MetadataMBeanInfoAssembler` 之外，Spring 还包括

`InterfaceBasedMBeanInfoAssembler`，它使您可以基于一组接口中定义的一组方法来约束公开的方法和属性。

尽管公开 MBean 的标准机制是使用接口和简单的命名方案，但是

`InterfaceBasedMBeanInfoAssembler` 扩展了此功能，它消除了对命名约定的需要，使您可以使用多个接口，并且不再需要 bean 来实现 MBean 接口。

考虑以下接口，该接口用于为我们先前显示的 `JmxTestBean` 类定义 Management 接口：

```
public interface IJmxTestBean {  
    public int add(int x, int y);  
    public long myOperation();  
    public int getAge();  
    public void setAge(int age);  
    public void setName(String name);  
    public String getName();  
}
```

该接口定义在 JMX MBean 上作为操作和属性公开的方法和属性。以下代码显示了如何配置 Spring JMX 以使用此接口作为 Management 接口的定义：

```
<beans>  
  
<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">  
    <property name="beans">  
        <map>  

```

在前面的示例中，在为任何 bean 构造 Management 接口时，将

`InterfaceBasedMBeanInfoAssembler` 配置为使用 `IJmxTestBean` 接口。理解由

`InterfaceBasedMBeanInfoAssembler` 处理的 bean 不需要实现用于生成 JMXManagement 接口的接口，这一点很重要。

在上述情况下，`IJmxTestBean` 接口用于为所有 bean 构造所有 Management 接口。在许多情况下，这不是理想的行为，并且您可能希望对不同的 bean 使用不同的接口。在这种情况下，您可以通过 `interfaceMappings` 属性传递 `InterfaceBasedMBeanInfoAssembler Properties` 实例，其中每个条目的键是 Bean 名称，每个条目的值是一个用逗号分隔的接口名称列表，用于该 Bean。

如果没有通过 `managedInterfaces` 或 `interfaceMappings` 属性指定 Management 接口，则 `InterfaceBasedMBeanInfoAssembler` 会在 Bean 上进行反映，并使用该 Bean 所实现的所有接口来创建 Management 接口。

#### 4.2.6. 使用 `MethodNameBasedMBeanInfoAssembler`

`MethodNameBasedMBeanInfoAssembler` 使您可以指定作为属性和操作公开给 JMX 的方法名称的列表。以下代码显示了示例配置：

```
<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
        <map>
            <entry key="bean:name=testBean5" value-ref="testBean"/>
        </map>
    </property>
    <property name="assembler">
        <bean class="org.springframework.jmx.export.assembler.MethodNameBasedMBeanInfoAssembler">
            <property name="managedMethods">
                <value>add,myOperation,getName,setName,getAge</value>
            </property>
        </bean>
    </property>
    </bean>
```

在前面的示例中，您可以看到 `add` 和 `myOperation` 方法公开为 JMX 操作，而 `getName()`，`setName(String)` 和 `getAge()` 公开为 JMX 属性的适当一半。在前面的代码中，方法 Map 适用于 JMX 公开的 bean。要控制每个 bean 的方法公开，可以使用

`MethodNameMBeanInfoAssembler` 的 `methodMappings` 属性将 bean 名称 Map 到方法名称列表。

## 4.3. 控制您的 Bean 的 `ObjectName` 实例

在幕后，`MBeanExporter` 委派 `ObjectNamingStrategy` 的实现，以为其注册的每个 bean 获得 `ObjectName` 实例。默认情况下，默认实现 `KeyNamingStrategy` 使用 `beans` `Map` 的键作为 `ObjectName`。此外，`KeyNamingStrategy` 可以将 `beans` `Map` 的键 Map 到 `Properties` 文件(或多个文件)中的条目以解析 `ObjectName`。除了 `KeyNamingStrategy` 之外，Spring 还提供了两个附加的 `ObjectNamingStrategy` 实现：`IdentityNamingStrategy` (基于 Bean 的 JVM 身份构建 `ObjectName`) 和 `MetadataNamingStrategy` (使用源级元数据获取 `ObjectName`)。

### 4.3.1. 从属性读取 `ObjectName` 实例

您可以配置自己的 `KeyNamingStrategy` 实例，并将其配置为从 `Properties` 实例读取 `ObjectName` 实例，而不必使用 Bean 键。`KeyNamingStrategy` 尝试使用与 bean 密钥相对应的密钥在 `Properties` 中定位条目。如果未找到任何条目，或者 `Properties` 实例是 `null`，则使用 bean 密钥本身。

以下代码显示了 `KeyNamingStrategy` 的示例配置：

```
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="beans">
            <map>
                <entry key="testBean" value-ref="testBean"/>
            </map>
        </property>
        <property name="namingStrategy" ref="namingStrategy"/>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>
```

```

<bean id="namingStrategy" class="org.springframework.jmx.export.naming.KeyNamingStr
  <property name="mappings">
    <props>
      <prop key="testBean">bean:name=testBean1</prop>
    </props>
  </property>
  <property name="mappingLocations">
    <value>names1.properties,names2.properties</value>
  </property>
</bean>

</beans>

```

前面的示例将 `KeyNamingStrategy` 实例与 `Properties` 实例配置在一起，该实例是从 `mapping` 属性定义的 `Properties` 实例和位于 `mappings` 属性定义的路径中的属性文件合并而成的。在此配置中，`testBean` bean 被赋予 `bean:name=testBean1` 的 `ObjectName`，因为这是 `Properties` 实例中的条目，该条目具有与 bean 密钥相对应的密钥。

如果在 `Properties` 实例中找不到任何条目，则将 bean 键名用作 `ObjectName`。

### 4.3.2. 使用元数据命名策略

`MetadataNamingStrategy` 使用每个 bean 上 `ManagedResource` 属性的 `objectName` 属性来创建 `ObjectName`。以下代码显示了 `MetadataNamingStrategy` 的配置：

```

<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="testBean" value-ref="testBean"/>
      </map>
    </property>
    <property name="namingStrategy" ref="namingStrategy"/>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

  <bean id="namingStrategy" class="org.springframework.jmx.export.naming.MetadataNami
    <property name="attributeSource" ref="attributeSource"/>
  </bean>

```

```
<bean id="attributeSource"
      class="org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource">
</beans>
```

如果没有为 `ManagedResource` 属性提供 `objectName`，那么将使用以下格式创建 `ObjectName`：  
\*: \* [完全合格的软件包名称]: type = [short-classname], name = [bean-name] \*。例如，为以下 bean 生成的 `ObjectName` 将是 `com.example:type=MyClass,name=myBean`：

```
<bean id="myBean" class="com.example.MyClass"/>
```

### 4.3.3. 配置基于 Comments 的 MBean 导出

如果您更喜欢使用[基于 Comments 的方法](#)定义 Management 界面，则可以使用 `MBeanExporter` 的便利子类：`AnnotationMBeanExporter`。在定义该子类的实例时，您不再需要 `namingStrategy`，`assembler` 和 `attributeSource` 配置，因为它始终使用基于 `JavaComments` 的标准元数据(也始终启用自动检测)。实际上，`@EnableMBeanExport` `@Configuration` Comments 支持更简单的语法，而不是定义 `MBeanExporter` bean，如以下示例所示：

```
@Configuration
@EnableMBeanExport
public class AppConfig { }
```

如果您更喜欢基于 XML 的配置，则 `<context:mbean-export/>` 元素具有相同的目的，并在以下列表中显示：

```
<context:mbean-export/>
```

如有必要，可以提供对特定 MBean `server` 的引用，并且 `defaultDomain` 属性(`AnnotationMBeanExporter` 的属性)接受生成的 MBean `ObjectName` 域的备用值。如上一示例

所示，它用于代替上一章节[MetadataNamingStrategy](#)中描述的标准软件包名称：

```
@EnableMBeanExport(server="myMBeanServer", defaultDomain="myDomain")
@Configuration
ContextConfiguration {
}
```

以下示例显示了与前面的基于 Comments 的示例等效的 XML：

```
<context:mbean-export server="myMBeanServer" default-domain="myDomain"/>
```

### ⚠Warning

请勿将基于接口的 AOP 代理与 bean 类中的 JMXComments 的自动检测结合使用。基于接口的代理“隐藏”目标类，它也隐藏了 JMXManagement 的资源 Comments。因此，在这种情况下，您应该使用目标类代理(通过在 `<aop:config/>`，`<tx:annotation-driven/>` 等上设置'proxy-target-class'标志)。否则，启动时可能会静默忽略您的 JMX bean。

## 4.4. 使用 JSR-160 连接器

对于远程访问，Spring JMX 模块在 `org.springframework.jmx.support` 包中提供了两个 `FactoryBean` 实现，用于创建服务器端和 Client 端连接器。

### 4.4.1. 服务器端连接器

要使 Spring JMX 创建，启动和公开 JSR-160 `JMXConnectorServer`，可以使用以下配置：

```
<bean id="serverConnector" class="org.springframework.jmx.support.ConnectorServerFactory"
```

默认情况下，`ConnectorServerFactoryBean` 创建绑定到

`service:jmx:jmxmp://localhost:9875` 的 `JMXConnectorServer`。因此，

`serverConnector` bean 通过 localhost 上的 JMXMP 协议(端口 9875)向 Client 端公开本地

`MBeanServer`。请注意，JSR 160 规范将 JMXMP 协议标记为可选。当前，主要的开源 JMX 实现 MX4J 和 JDK 随附的实现不支持 JMXMP。

要指定另一个 URL 并向 `MBeanServer` 注册 `JMXConnectorServer` 本身，可以分别使用 `serviceUrl` 和 `ObjectName` 属性，如以下示例所示：

```
<bean id="serverConnector"
      class="org.springframework.jmx.support.ConnectorServerFactoryBean">
    <property name="objectName" value="connector:name=rmi"/>
    <property name="serviceUrl"
              value="service:jmx:rmi://localhost/jndi/rmi://localhost:1099/myconnector"/>
</bean>
```

如果设置了 `ObjectName` 属性，Spring 会在 `ObjectName` 下自动将连接器注册到 `MBeanServer`。以下示例显示了创建 `JMXConnector` 时可以传递给 `ConnectorServerFactoryBean` 的完整参数集：

```
<bean id="serverConnector"
      class="org.springframework.jmx.support.ConnectorServerFactoryBean">
    <property name="objectName" value="connector:name=iiop"/>
    <property name="serviceUrl"
              value="service:jmx:iiop://localhost/jndi/iiop://localhost:900/myconnector"/>
    <property name="threaded" value="true"/>
    <property name="daemon" value="true"/>
    <property name="environment">
      <map>
        <entry key="someKey" value="someValue"/>
      </map>
    </property>
</bean>
```

请注意，在使用基于 RMI 的连接器时，需要启动查找服务（`tnameserv` 或 `rmiregistry`）才能完成名称注册。如果您使用 Spring 通过 RMI 为您导出远程服务，则 Spring 已经构造了一个 RMI 注册表。如果没有，您可以使用以下配置片段轻松启动注册表：

```
<bean id="registry" class="org.springframework.remoting.rmi.RmiRegistryFactoryBean">
  <property name="port" value="1099"/>
</bean>
```

#### 4.4.2. Client 端连接器

要为启用了远程 JSR-160 的 `MBeanServer` 创建 `MBeanServerConnection`，可以使用 `MBeanServerConnectionFactoryBean`，如以下示例所示：

```
<bean id="clientConnector" class="org.springframework.jmx.support.MBeanServerConnection">
    <property name="serviceUrl" value="service:jmx:rmi://localhost/jndi/rmi://localhost/>
</bean>
```

#### 4.4.3. 通过 Hessian 或 SOAP 的 JMX

JSR-160 允许扩展 Client 端与服务器之间进行通信的方式。前面各节中显示的示例使用 JSR-160 规范(IIOP 和 JRMP)和(可选)JMXMP 所需的基于 RMI 的强制实现。通过使用其他提供程序或 JMX 实现(例如[MX4J](#))，可以通过简单的 HTTP 或 SSL 以及其他协议利用 SOAP 或 Hessian 等协议，如以下示例所示：

```
<bean id="serverConnector" class="org.springframework.jmx.support.ConnectorServerFactory">
    <property name="objectName" value="connector:name=burlap"/>
    <property name="serviceUrl" value="service:jmx:burlap://localhost:9874"/>
</bean>
```

在前面的示例中，我们使用了 MX4J 3.0.0. 有关更多信息，请参见官方 MX4J 文档。

### 4.5. 通过代理访问 MBean

Spring JMX 使您可以创建代理，以将调用重新路由到在本地或远程 `MBeanServer` 中注册的 MBean。这些代理为您提供了一个标准的 Java 接口，您可以通过它与 MBean 进行交互。以下代码显示了如何为在本地 `MBeanServer` 中运行的 MBean 配置代理：

```
<bean id="proxy" class="org.springframework.jmx.access.MBeanProxyFactoryBean">
    <property name="objectName" value="bean:name=testBean"/>
    <property name="proxyInterface" value="org.springframework.jmx.IJmxTestBean"/>
</bean>
```

在前面的示例中，您可以看到为在 `bean:name=testBean` 的 `ObjectName` 下注册的 MBean 创建了代理。代理实现的接口集由 `proxyInterfaces` 属性控制，将这些接口上的方法和属性 Map 到 MBean 上的操作和属性的规则与 `InterfaceBasedMBeanInfoAssembler` 使用的规则相同。

`MBeanProxyFactoryBean` 可以创建可通过 `MBeanServerConnection` 访问的任何 MBean 的代理

。默认情况下，已找到并使用了本地 `MBeanServer`，但是您可以覆盖它并提供一个

`MBeanServerConnection` 指向远程 `MBeanServer`，以适应指向远程 MBean 的代理：

```
<bean id="clientConnector"
      class="org.springframework.jmx.support.MBeanServerConnectionFactoryBean">
    <property name="serviceUrl" value="service:jmx:rmi://remotehost:9875"/>
</bean>

<bean id="proxy" class="org.springframework.jmx.access.MBeanProxyFactoryBean">
    <property name="objectName" value="bean:name=testBean"/>
    <property name="proxyInterface" value="org.springframework.jmx.IJmxTestBean"/>
    <property name="server" ref="clientConnector"/>
</bean>
```

在前面的示例中，我们创建一个 `MBeanServerConnection`，该 `MBeanServerConnection` 指向

使用 `MBeanServerConnectionFactoryBean` 的远程计算机。然后，此 `MBeanServerConnection`

通过 `server` 属性传递到 `MBeanProxyFactoryBean`。创建的代理通过此

`MBeanServerConnection` 将所有调用转发到 `MBeanServer`。

## 4.6. Notifications

Spring 的 JMX 产品包括对 JMX 通知的全面支持。

### 4.6.1. 注册侦听器以接收通知

Spring 的 JMX 支持使您可以轻松地将任意数量的 `NotificationListeners` 注册到任意数量的

MBean(这包括 Spring 的 `MBeanExporter` 导出的 MBean 和通过其他机制注册的 MBean)。例如

，考虑一种情况，即每次目标 MBean 的属性发生更改时，都希望(通过 `Notification`)被告知。

以下示例将通知写入控制台：

```
package com.example;

import javax.management.AttributeChangeNotification;
import javax.management.Notification;
import javax.management.NotificationFilter;
```

```

import javax.management.NotificationListener;

public class ConsoleLoggingNotificationListener
    implements NotificationListener, NotificationFilter {

    public void handleNotification(Notification notification, Object handback) {
        System.out.println(notification);
        System.out.println(handback);
    }

    public boolean isNotificationEnabled(Notification notification) {
        return AttributeChangeNotification.class.isAssignableFrom(notification.getClass());
    }
}

```

以下示例将 `ConsoleLoggingNotificationListener` (在前面的示例中定义)添加到

`notificationListenerMappings` :

```

<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="beans">
            <map>
                <entry key="bean:name=testBean1" value-ref="testBean"/>
            </map>
        </property>
        <property name="notificationListenerMappings">
            <map>
                <entry key="bean:name=testBean1">
                    <bean class="com.example.ConsoleLoggingNotificationListener"/>
                </entry>
            </map>
        </property>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

</beans>

```

使用先前的配置，每次从目标 MBean(`bean:name=testBean1`)BroadcastJMX `Notification` 时

，都会通知通过 `notificationListenerMappings` 属性注册为侦听器的

`ConsoleLoggingNotificationListener` bean。然后

`ConsoleLoggingNotificationListener` bean 可以响应 `Notification` 采取其认为适当的任何操作。

您还可以使用纯 bean 名称作为导出的 bean 与侦听器之间的链接，如以下示例所示：

```
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="beans">
            <map>
                <entry key="bean:name=testBean1" value-ref="testBean"/>
            </map>
        </property>
        <property name="notificationListenerMappings">
            <map>
                <entry key="testBean">
                    <bean class="com.example.ConsoleLoggingNotificationListener"/>
                </entry>
            </map>
        </property>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

</beans>
```

如果要为封闭的 MBeanExporter 导出的所有 bean 注册一个 NotificationListener 实例，则可以使用特殊的通配符( \* )作为 notificationListenerMappings 属性 Map 中条目的键，如以下示例所示：

```
<property name="notificationListenerMappings">
    <map>
        <entry key="*">
            <bean class="com.example.ConsoleLoggingNotificationListener"/>
        </entry>
    </map>
</property>
```

如果需要进行相反操作(即，针对 MBean 注册多个不同的侦听器)，则必须使用 notificationListeners list 属性(而不是 notificationListenerMappings 属性)。这次，我们配置 NotificationListenerBean 实例，而不是为单个 MBean 配置 NotificationListener。

- NotificationListenerBean 将 NotificationListener 和要注册的 ObjectName (或 ObjectNames )封装在 MBeanServer 中。 NotificationListenerBean 还封装了许多其他属性，例如 NotificationFilter 和可以在高级 JMX 通知场景中使用的任意 handback 对象。

使用 `NotificationListenerBean` 实例时的配置与先前介绍的配置没有很大不同，如以下示例所示：

```
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="beans">
            <map>
                <entry key="bean:name=testBean1" value-ref="testBean"/>
            </map>
        </property>
        <property name="notificationListeners">
            <list>
                <bean class="org.springframework.jmx.export.NotificationListenerBean">
                    <constructor-arg>
                        <bean class="com.example.ConsoleLoggingNotificationListener"/>
                    </constructor-arg>
                    <property name="mappedObjectNames">
                        <list>
                            <value>bean:name=testBean1</value>
                        </list>
                    </property>
                </bean>
            </list>
        </property>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

</beans>
```

前面的示例等效于第一个通知示例。那么，假设我们想在每次引发 `Notification` 时得到一个递归对象，并且还希望通过提供 `NotificationFilter` 来过滤掉无关的 `Notifications`。以下示例实现了这些目标：

```
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="beans">
            <map>
                <entry key="bean:name=testBean1" value-ref="testBean1"/>
                <entry key="bean:name=testBean2" value-ref="testBean2"/>
            </map>
        </property>
        <property name="notificationListeners">
            <list>
                <bean class="org.springframework.jmx.export.NotificationListenerBean">
                    <constructor-arg ref="customerNotificationListener"/>
                    <property name="mappedObjectNames">
```

```

        <list>
            <!-- handles notifications from two distinct MBeans -->
            <value>bean:name=testBean1</value>
            <value>bean:name=testBean2</value>
        </list>
    </property>
    <property name="handback">
        <bean class="java.lang.String">
            <constructor-arg value="This could be anything..."/>
        </bean>
    </property>
    <property name="notificationFilter" ref="customerNotificationListener" />
</bean>
</list>
</property>
</bean>

<!-- implements both the NotificationListener and NotificationFilter interfaces -->
<bean id="customerNotificationListener" class="com.example.ConsoleLoggingNotificationListener" />

<bean id="testBean1" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
</bean>

<bean id="testBean2" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="ANOTHER TEST"/>
    <property name="age" value="200"/>
</bean>

</beans>

```

(有关移交对象是什么，实际上 `NotificationFilter` 是什么的完整讨论，请参阅 JMX 规范(1.2)中名为“JMX 通知模型”的部分。)

## 4.6.2. 发布通知

Spring 不仅提供注册支持以接收 `Notifications`，而且还提供发布 `Notifications` 的支持。

### **Note**

这部分实际上仅与已通过 `MBeanExporter` 公开为 MBean 的 Spring 托管 Bean 有关。任何现有的用户定义的 MBean 都应使用标准的 JMX API 进行通知发布。

Spring 的 JMX 通知发布支持中的关键接口是 `NotificationPublisher` 接口(在

`org.springframework.jmx.export.notification` 包中定义)。任何要通过 `MBeanExporter` 实

例作为 MBean 导出的 bean 都可以实现相关的 `NotificationPublisherAware` 接口来访问 `NotificationPublisher` 实例。 `NotificationPublisherAware` 接口通过一个简单的 `setter` 方法将 `NotificationPublisher` 的实例提供给实现 Bean，然后该 bean 可以使用它来发布 `Notifications`。

如[NotificationPublisher](#)接口的 javadoc 中所述，通过 `NotificationPublisher` 机制发布事件的托管 bean 不负责通知侦听器的状态 Management。Spring 的 JMX 支持负责处理所有 JMX 基础结构问题。作为应用程序开发人员，您需要做的就是实现 `NotificationPublisherAware` 接口并使用提供的 `NotificationPublisher` 实例开始发布事件。注意，`NotificationPublisher` 是在已将托管 bean 注册到 `MBeanServer` 之后设置的。

使用 `NotificationPublisher` 实例非常简单。您创建一个 JMX `Notification` 实例(或适当的 `Notification` 子类的实例)，使用与要发布的事件相关的数据填充通知，并在 `NotificationPublisher` 实例上调用 `sendNotification(Notification)` 并传入 `Notification`。

在以下示例中，每次调用 `add(int, int)` 操作时，`JmxTestBean` 的导出实例都会发布 `NotificationEvent`：

```
package org.springframework.jmx;

import org.springframework.jmx.export.notification.NotificationPublisherAware;
import org.springframework.jmx.export.notification.NotificationPublisher;
import javax.management.Notification;

public class JmxTestBean implements IJmxTestBean, NotificationPublisherAware {

    private String name;
    private int age;
    private boolean isSuperman;
    private NotificationPublisher publisher;

    // other getters and setters omitted for clarity

    public int add(int x, int y) {
        int answer = x + y;
        publisher.sendNotification(new Notification("JmxTestBean", "AddEvent", null));
    }
}
```

```

        this.publisher.sendNotification(new Notification("add", this, 0));
    return answer;
}

public void dontExposeMe() {
    throw new RuntimeException();
}

public void setNotificationPublisher(NotificationPublisher notificationPublisher) {
    this.publisher = notificationPublisher;
}

}

```

`NotificationPublisher` 界面和使一切正常运行的机制是 Spring 的 JMX 支持的更好的功能之一。但是，它确实带有将类耦合到 Spring 和 JMX 的代价。和往常一样，这里的建议要务实。如果您需要 `NotificationPublisher` 提供的功能，并且可以接受到 Spring 和 JMX 的耦合，则可以这样做。

## 4.7. 更多资源

本节包含有关 JMX 的更多资源的链接：

- 甲骨文的[JMX homepage](#)。
- [JMX specification](#)(JSR-000003)。
- [JMX 远程 API 规范](#)(JSR-000160)。
- [MX4J homepage](#)。 (MX4J 是各种 JMX 规范的开源实现.)

## 5. JCA CCI

---

Java EE 提供了一个规范来标准化对企业信息系统(EIS)的访问：JCA(Java EE 连接器体系结构)。该规范分为两个不同的部分：

- 连接器提供程序必须实现的 SPI(服务提供程序接口)。这些接口构成了可以部署在 Java EE 应用程序服务器上的资源适配器。在这种情况下，服务器将 Management 连接池，事务和安全性(托管模式)。应用服务器还负责 Management 配置，该配置保存在 Client 端应用程序外部。连

接器也可以在没有应用程序服务器的情况下使用。在这种情况下，应用程序必须直接对其进行配置(非托管模式)。

- 应用程序可用于与连接器进行交互并由此与 EIS 通信的 CCI(通用 Client 端接口)。还提供了用于本地事务划分的 API。

Spring CCI 支持的目的是使用 Spring Framework 的常规资源和事务 Management 工具，提供类似典型的 Spring 样式访问 CCI 连接器。

### **Note**

连接器的 Client 端始终不使用 CCI。某些连接器公开了自己的 API，提供了 JCA 资源适配器以使用 Java EE 容器的系统协定(连接池，全局事务和安全性)。Spring 没有为此类特定于连接器的 API 提供特殊支持。

## 5.1. 配置 CCI

本节介绍如何配置公共 Client 端接口(CCI)。它包括以下主题：

- [Connector Configuration](#)
- [Spring 中的 ConnectionFactory 配置](#)
- [配置 CCI 连接](#)
- [使用单个 CCI 连接](#)

### 5.1.1. 连接器配置

使用 JCA CCI 的基本资源是 `ConnectionFactory` 接口。您使用的连接器必须提供此接口的实现。

要使用连接器，可以将其部署在应用程序服务器上，并从服务器的 JNDI 环境(托管模式)中获取 `ConnectionFactory`。连接器必须打包为 RAR 文件(资源适配器 Files)，并包含 `ra.xml` 文件来描述其部署特性。资源的实际名称是在部署时指定的。要在 Spring 中访问它，可以使用 Spring 的

`JndiObjectFactoryBean` 或 `<jee:jndi-lookup>` 通过其 JNDI 名称获取工厂。

使用连接器的另一种方法是将其嵌入到您的应用程序中(非托管模式), 而不使用应用程序服务器来部署和配置它。 Spring 提供了通过称为(`LocalConnectionFactoryBean`)的`FactoryBean` 实现将连接器配置为 Bean 的可能性。通过这种方式, 您只需要在 Classpath 中使用连接器库(不需要 RAR 文件和 `ra.xml` Descriptors)。如有必要, 必须从连接器的 RAR 文件中提取该库。

一旦可以访问 `ConnectionFactory` 实例, 就可以将其注入到组件中。这些组件可以根据普通的 CCI API 进行编码, 也可以使用 Spring 的支持类进行 CCI 访问(例如 `CciTemplate` )。

#### iNote

在非托管模式下使用连接器时, 您将无法使用全局事务, 因为资源永远不会在当前线程的当前全局事务中被征用或除名。该资源不知道任何可能正在运行的全局 Java EE 事务。

### 5.1.2. Spring 中的 `ConnectionFactory` 配置

要连接到 EIS, 需要从应用程序服务器(如果处于托管模式)或直接从 Spring(如果处于非托管模式)获取 `ConnectionFactory`。

在托管模式下, 您可以从 JNDI 访问 `ConnectionFactory`。其属性在应用程序服务器中配置。以下示例显示了如何执行此操作:

```
<jee:jndi-lookup id="eciConnectionFactory" jndi-name="eis/cicseci"/>
```

在非托管模式下, 必须将要在 Spring 的配置中使用的 `ConnectionFactory` 配置为 JavaBean。

`LocalConnectionFactoryBean` 类提供了这种设置样式, 传入了连接器的 `ManagedConnectionFactory` 实现, 公开了应用程序级 CCI `ConnectionFactory`。以下示例显示了如何执行此操作:

```
<bean id="eciManagedConnectionFactory" class="com.ibm.connector2.cics.ECIManagedConnect
```

```
<property name="serverName" value="TXSERIES"/>
<property name="connectionURL" value="tcp://localhost//"/>
<property name="portNumber" value="2006"/>
</bean>

<bean id="eciConnectionFactory" class="org.springframework.jca.support.LocalConnectionF
    <property name="managedConnectionFactory" ref="eciManagedConnectionFactory"/>
</bean>
```

### iNote

您不能直接实例化特定的 `ConnectionFactory`。您需要为连接器完成

`ManagedConnectionFactory` 接口的相应实现。该接口是 JCA SPI 规范的一部分。

## 5.1.3. 配置 CCI 连接

通过 JCA CCI，您可以使用连接器的 `ConnectionSpec` 实现来配置到 EIS 的连接。要配置其属性，您需要使用专用适配器 `ConnectionSpecConnectionFactoryAdapter` 包装目标连接工厂。您可以使用 `connectionSpec` 属性(作为内部 bean)配置专用的 `ConnectionSpec`。

此属性不是必需的，因为 CCI `ConnectionFactory` 接口定义了两种不同的方法来获取 CCI 连接。

您通常可以在应用程序服务器(处于托管模式下)或相应的本地 `ManagedConnectionFactory` 实现上配置某些 `ConnectionSpec` 属性。以下 Lists 显示了 `ConnectionFactory` 接口定义的相关部分：

```
public interface ConnectionFactory implements Serializable, Referenceable {
    ...
    Connection getConnection() throws ResourceException;
    Connection getConnection(ConnectionSpec connectionSpec) throws ResourceException;
    ...
}
```

Spring 提供了一个 `ConnectionSpecConnectionFactoryAdapter`，您可以指定一个

`ConnectionSpec` 实例用于给定工厂上的所有操作。如果指定了适配器的 `connectionSpec` 属性，则适配器将 `getConnection` 变量与 `ConnectionSpec` 参数一起使用。否则，适配器将使用不带该参数的变量。以下示例显示了如何配置 `ConnectionSpecConnectionFactoryAdapter`：

```

<bean id="managedConnectionFactory"
      class="com.sun.connector.cciblackbox.CciLocalTxManagedConnectionFactory">
    <property name="connectionURL" value="jdbc:hsqldb:hsqldb://localhost:9001"/>
    <property name="driverName" value="org.hsqldb.jdbcDriver"/>
</bean>

<bean id="targetConnectionFactory"
      class="org.springframework.jca.support.LocalConnectionFactoryBean">
    <property name="managedConnectionFactory" ref="managedConnectionFactory"/>
</bean>

<bean id="connectionFactory"
      class="org.springframework.jca.cci.connection.ConnectionSpecConnectionFactoryAd
<property name="targetConnectionFactory" ref="targetConnectionFactory"/>
<property name="connectionSpec">
    <bean class="com.sun.connector.cciblackbox.CciConnectionSpec">
        <property name="user" value="sa"/>
        <property name="password" value="" />
    </bean>
</property>
</bean>

```

## 5.1.4. 使用单个 CCI 连接

如果要使用单个 CCI 连接，Spring 会提供另一个 `ConnectionFactory` 适配器来 Management 此连接。 `SingleConnectionFactory` 适配器类延迟打开单个连接，并在应用程序关闭时销毁该 bean 时将其关闭。此类公开了具有相应行为的特殊 `Connection` 代理，它们均共享相同的基础物理连接。以下示例显示如何使用 `SingleConnectionFactory` 适配器类：

```

<bean id="eciManagedConnectionFactory"
      class="com.ibm.connector2.cics.ECIManagedConnectionFactory">
    <property name="serverName" value="TEST"/>
    <property name="connectionURL" value="tcp://localhost:/" />
    <property name="portNumber" value="2006"/>
</bean>

<bean id="targetEciConnectionFactory"
      class="org.springframework.jca.support.LocalConnectionFactoryBean">
    <property name="managedConnectionFactory" ref="eciManagedConnectionFactory"/>
</bean>

<bean id="eciConnectionFactory"
      class="org.springframework.jca.cci.connection.SingleConnectionFactory">
    <property name="targetConnectionFactory" ref="targetEciConnectionFactory"/>
</bean>

```

### 1 Note

无法使用 `ConnectionSpec` 直接配置此 `ConnectionFactory` 适配器。如果您需要特定 `ConnectionSpec` 的单个连接，则可以使用 `SingleConnectionFactory` 与之联系的中介 `ConnectionSpecConnectionFactoryAdapter`。

## 5.2. 使用 Spring 的 CCI 访问支持

本节描述如何使用 Spring 对 CCI 的支持来实现各种目的。它包括以下主题：

- [Record Conversion](#)
- [Using CciTemplate](#)
- [使用 DAO 支持](#)
- [自动输出记录生成](#)
- [CciTemplate 交互摘要](#)
- [直接使用 CCI 连接和交互](#)
- [CciTemplate 用法示例](#)

### 5.2.1. 记录转换

Spring 的 JCA CCI 支持的目的之一是为处理 CCI 记录提供便利的设施。您可以指定策略来创建记录并从 Logging 提取数据，以用于 Spring 的 `CciTemplate`。如果您不想直接在应用程序中使用记录，本节中描述的接口将策略配置为使用 Importing 和输出记录。

要创建 Importing `Record`，可以使用 `RecordCreator` 接口的专用实现。以下 Lists 显示了 `RecordCreator` 接口定义：

```
public interface RecordCreator {  
    Record createRecord(RecordFactory recordFactory) throws ResourceException, DataAcce
```

```
}
```

`createRecord(...)` 方法接收 `RecordFactory` 实例作为参数，该实例与所使用的 `ConnectionFactory` 的 `RecordFactory` 相对应。您可以使用此引用来创建 `IndexedRecord` 或 `MappedRecord` 实例。下面的示例演示如何使用 `RecordCreator` 接口以及索引或 Map 记录：

```
public class MyRecordCreator implements RecordCreator {  
  
    public Record createRecord(RecordFactory recordFactory) throws ResourceException {  
        IndexedRecord input = recordFactory.createIndexedRecord("input");  
        input.add(new Integer(id));  
        return input;  
    }  
  
}
```

您可以使用输出 `Record` 从 EIS 接收数据。因此，您可以将 `RecordExtractor` 接口的特定实现传递给 Spring 的 `CciTemplate`，以从输出 `Record` 提取数据。以下 Lists 显示了 `RecordExtractor` 接口定义：

```
public interface RecordExtractor {  
  
    Object extractData(Record record) throws ResourceException, SQLException, DataAccessException;  
}
```

以下示例显示了如何使用 `RecordExtractor` 界面：

```
public class MyRecordExtractor implements RecordExtractor {  
  
    public Object extractData(Record record) throws ResourceException {  
        CommAreaRecord commAreaRecord = (CommAreaRecord) record;  
        String str = new String(commAreaRecord.toByteArray());  
        String field1 = str.substring(0,6);  
        String field2 = str.substring(6,1);  
        return new OutputObject(Long.parseLong(field1), field2);  
    }  
}
```

## 5.2.2. 使用 CciTemplate

`CciTemplate` 是核心 CCI 支持包(`org.springframework.jca.cci.core`)的中心类。由于它处理资源的创建和释放，因此它简化了 CCI 的使用。这有助于避免常见错误，例如忘记始终关闭连接。它关心连接和交互对象的生命周期，让应用程序代码专注于从应用程序数据生成 Importing 记录并从输出 Logging 提取应用程序数据。

JCA CCI 规范定义了两种不同的方法来调用 EIS 上的操作。CCI `Interaction` 接口提供了两个 `execute` 方法签名，如以下 Lists 所示：

```
public interface javax.resource.cci.Interaction {  
    ...  
    boolean execute(InteractionSpec spec, Record input, Record output) throws ResourceException;  
    Record execute(InteractionSpec spec, Record input) throws ResourceException;  
    ...  
}
```

根据所调用的模板方法，`CciTemplate` 知道在交互中要调用哪个 `execute` 方法。无论如何，必须正确初始化 `InteractionSpec` 实例。

您可以通过两种方式使用 `CciTemplate.execute(..)`：

- 具有直接的 `Record` 参数。在这种情况下，您需要传递 CCIImporting 记录，并且返回的对象是相应的 CCI 输出记录。
- 对于应用程序对象，通过使用记录 Map。在这种情况下，您需要提供相应的 `RecordCreator` 和 `RecordExtractor` 实例。

对于第一种方法，使用模板的以下方法(直接与 `Interaction` 接口上的方法相对应)：

```
public class CciTemplate implements CciOperations {  
    public Record execute(InteractionSpec spec, Record inputRecord)  
        throws DataAccessException { ... }  
    public void execute(InteractionSpec spec, Record inputRecord, Record outputRecord)
```

```
    throws DataAccessException { ... }  
}
```

使用第二种方法，我们需要指定记录创建和记录提取策略作为参数。使用的接口是[上一节有关记录转换](#)中描述的接口。下面的 Lists 显示了相应的 `CciTemplate` 方法：

```
public class CciTemplate implements CciOperations {  
  
    public Record execute(InteractionSpec spec,  
                         RecordCreator inputCreator) throws DataAccessException {  
        // ...  
    }  
  
    public Object execute(InteractionSpec spec, Record inputRecord,  
                         RecordExtractor outputExtractor) throws DataAccessException {  
        // ...  
    }  
  
    public Object execute(InteractionSpec spec, RecordCreator creator,  
                         RecordExtractor extractor) throws DataAccessException {  
        // ...  
    }  
}
```

除非在模板上设置了 `outputRecordCreator` 属性(请参见下一节)，否则每个方法都将使用两个参数调用 `InteractionSpec` 和 Importing `Record` 来调用 CCI `Interaction` 的相应 `execute` 方法。它接收输出 `Record` 作为其返回值。

`CciTemplate` 还提供了通过 `createIndexRecord(..)` 和 `createMappedRecord(..)` 方法在 `RecordCreator` 实现之外创建 `IndexRecord` 和 `MappedRecord` 的方法。您可以在 DAO 实现中使用它来创建 `Record` 实例以传递到相应的 `CciTemplate.execute(..)` 方法中。以下 Lists 显示了 `CciTemplate` 接口定义：

```
public class CciTemplate implements CciOperations {  
  
    public IndexedRecord createIndexRecord(String name) throws DataAccessException { ... }  
    public MappedRecord createMappedRecord(String name) throws DataAccessException { ... }  
}
```

### 5.2.3. 使用 DAO 支持

Spring 的 CCI 支持为 DAO 提供了一个抽象类，支持注入 `ConnectionFactory` 或 `CciTemplate` 实例。该类的名称是 `CciDaoSupport`。它提供了简单的 `setConnectionFactory` 和 `setCciTemplate` 方法。在内部，此类为传入的 `ConnectionFactory` 创建 `CciTemplate` 实例，并将其暴露给子类中的具体数据访问实现。以下示例显示了如何使用 `CciDaoSupport`：

```
public abstract class CciDaoSupport {  
  
    public void setConnectionFactory(ConnectionFactory connectionFactory) {  
        // ...  
    }  
  
    public ConnectionFactory getConnectionFactory() {  
        // ...  
    }  
  
    public void setCciTemplate(CciTemplate cciTemplate) {  
        // ...  
    }  
  
    public CciTemplate getCciTemplate() {  
        // ...  
    }  
  
}
```

### 5.2.4. 自动输出记录生成

如果您使用的连接器仅支持将 Importing 和输出记录作为参数的 `Interaction.execute(...)` 方法（也就是说，它需要传递所需的输出记录而不是返回适当的输出记录），则可以将 `CciTemplate` 的 `outputRecordCreator` 属性设置为收到响应后，将自动生成输出记录，以供 JCA 连接器填充。然后将该记录返回给模板的调用者。

此属性保存用于该目的的 [RecordCreator interface](#) 的实现。您必须直接在 `CciTemplate` 上指定 `outputRecordCreator` 属性。以下示例显示了如何执行此操作：

```
cciTemplate.setOutputRecordCreator(new EciOutputRecordCreator());
```

或者(建议使用这种方法), 在 Spring 配置中, 如果将 `CciTemplate` 配置为专用 bean 实例, 则可以按以下方式定义 bean:

```
<bean id="eciOutputRecordCreator" class="eci.EciOutputRecordCreator"/>

<bean id="cciTemplate" class="org.springframework.jca.cci.core.CciTemplate">
    <property name="connectionFactory" ref="eciConnectionFactory"/>
    <property name="outputRecordCreator" ref="eciOutputRecordCreator"/>
</bean>
```

### ① Note

由于 `CciTemplate` 类是线程安全的, 因此通常将其配置为共享实例。

## 5.2.5. CciTemplate 交互摘要

下表总结了 `CciTemplate` 类的机制以及在 CCI `Interaction` 接口上调用的相应方法:

表 9. 交互执行方法的用法

<code>CciTemplate</code> 方法签名	<code>CciTemplate</code> 属性	<code>execute</code>
<code>Record</code> <code>execute(InteractionSpec,</code> <code>Record)</code>	<code>Not set</code>	<code>Record</code> <code>execute(</code> <code>Record)</code>
<code>Record</code> <code>execute(InteractionSpec,</code> <code>Record)</code>	<code>Set</code>	<code>boolean</code> <code>execute(</code> <code>Record,</code>
无效执行( <code>InteractionSpec</code> , 记录, 记录)	<code>Not set</code>	无效执行( <code>,</code> , 记录)

CciTemplate 方法签名	CciTemplate outputRecordCreator 属性	execute
<pre>void execute(InteractionSpec, Record, Record)</pre>	Set	<pre>void execute(InteractionSpec, Record, Record)</pre>
<pre>Record execute(InteractionSpec, RecordCreator)</pre>	Not set	<pre>Record execute(InteractionSpec, Record)</pre>
<pre>Record execute(InteractionSpec, RecordCreator)</pre>	Set	<pre>void execute(InteractionSpec, Record)</pre>
<pre>Record execute(InteractionSpec, Record, RecordExtractor)</pre>	Not set	<pre>Record execute(InteractionSpec, Record)</pre>
<pre>Record execute(InteractionSpec, Record, RecordExtractor)</pre>	Set	<pre>void execute(InteractionSpec, Record, Record)</pre>
<pre>Record execute(InteractionSpec, RecordCreator, RecordExtractor)</pre>	Not set	<pre>Record execute(InteractionSpec, Record)</pre>
Record	Set	void

<code>CciTemplate</code> 方法签名	<code>CciTemplate</code> <code>outputRecordCreator</code> 属性	<code>execute</code>
<pre>execute(InteractionSpec, RecordCreator, RecordExtractor)</pre>		<pre>execute( Record,</pre>

## 5.2.6. 直接使用 CCI 连接和交互

`CciTemplate` 还允许您以与 `JdbcTemplate` 和 `JmsTemplate` 相同的方式直接处理 CCI 连接和交互。例如，当您要对 CCI 连接或交互执行多个操作时，此功能很有用。

`ConnectionCallback` 接口提供 CCI `Connection` 作为自变量(以对其执行自定义操作)以及创建 `Connection` 的 CCI `ConnectionFactory`。后者可能很有用(例如，获取关联的 `RecordFactory` 实例并创建索引/Map 记录)。以下 Lists 显示了 `ConnectionCallback` 接口定义：

```
public interface ConnectionCallback {
    Object doInConnection(Connection connection, ConnectionFactory connectionFactory)
        throws ResourceException, SQLException, DataAccessException;
}
```

`InteractionCallback` 接口提供 CCI `Interaction` (以对其执行自定义操作)以及相应的 CCI `ConnectionFactory`。以下 Lists 显示了 `InteractionCallback` 接口定义：

```
public interface InteractionCallback {
    Object doInInteraction(Interaction interaction, ConnectionFactory connectionFactory)
        throws ResourceException, SQLException, DataAccessException;
}
```

### iNote

`InteractionSpec` 对象可以在多个模板调用之间共享，也可以在每个回调方法中重新创建。这完全取决于 DAO 的实现。

### 5.2.7. CciTemplate 用法示例

在本部分中，我们将说明 `CciTemplate` 通过 IBM CICS ECI 连接器以 ECI 模式访问 CICS 的用法。

首先，我们必须对 CCI `InteractionSpec` 进行一些初始化，以指定要访问的 CICS 程序以及如何与之交互，如以下示例所示：

```
ECIInteractionSpec interactionSpec = new ECIInteractionSpec();
interactionSpec.setFunctionName("MYPROG");
interactionSpec.setInteractionVerb(ECIInteractionSpec.SYNC_SEND_RECEIVE);
```

然后程序可以通过 Spring 的模板使用 CCI 并指定自定义对象和 CCI `Records` 之间的 Map，如以下示例所示：

```
public class MyDaoImpl extends CciDaoSupport implements MyDao {

    public OutputObject getData(InputObject input) {
        ECIInteractionSpec interactionSpec = ...;

        OutputObject output = (ObjectOutput) getCCiTemplate().execute(interactionSpec,
            new RecordCreator() {
                public Record createRecord(RecordFactory recordFactory) throws ResourceException {
                    return new CommAreaRecord(input.toString().getBytes());
                }
            },
            new RecordExtractor() {
                public Object extractData(Record record) throws ResourceException {
                    CommAreaRecord commAreaRecord = (CommAreaRecord)record;
                    String str = new String(commAreaRecord.toByteArray());
                    String field1 = str.substring(0,6);
                    String field2 = str.substring(6,1);
                    return new OutputObject(Long.parseLong(field1), field2);
                }
            });
        return output;
    }
}
```

如前所述，您可以使用回调直接在 CCI 连接或交互上工作。以下示例显示了如何执行此操作：

```
public class MyDaoImpl extends CciDaoSupport implements MyDao {  
  
    public OutputObject getData(InputObject input) {  
        ObjectOutput output = (ObjectOutput) getCciTemplate().execute(  
            new ConnectionCallback() {  
                public Object doInConnection(Connection connection,  
                    ConnectionFactory factory) throws ResourceException {  
  
                    // do something...  
  
                }  
            } );  
        return output;  
    }  
}
```

### iNote

对于 `ConnectionCallback`，使用的 `Connection` 由 `CciTemplate` Management 和关闭，但是回调实现必须 Management 在连接上创建的所有交互。

对于更具体的回调，您可以实现 `InteractionCallback`。如果这样做，传入的 `Interaction` 将由 `CciTemplate` Management 和关闭。以下示例显示了如何执行此操作：

```
public class MyDaoImpl extends CciDaoSupport implements MyDao {  
  
    public String getData(String input) {  
        ECIInteractionSpec interactionSpec = ...;  
        String output = (String) getCciTemplate().execute(interactionSpec,  
            new InteractionCallback() {  
                public Object doInInteraction(Interaction interaction,  
                    ConnectionFactory factory) throws ResourceException {  
                    Record input = new CommAreaRecord(inputString.getBytes());  
                    Record output = new CommAreaRecord();  
                    interaction.execute(holder.getInteractionSpec(), input, output);  
                    return new String(output.toByteArray());  
                }  
            } );  
        return output;  
    }  
}
```

对于前面的示例，在非托管模式下，所涉及的 Spring Bean 的相应配置可能类似于以下示例：

```

<bean id="managedConnectionFactory" class="com.ibm.connector2.cics.ECIManagedConnection"
    <property name="serverName" value="TXSERIES"/>
    <property name="connectionURL" value="local:"/>
    <property name="userName" value="CICSUSER"/>
    <property name="password" value="CICS"/>
</bean>

<bean id="connectionFactory" class="org.springframework.jca.support.LocalConnectionFactory"
    <property name="managedConnectionFactory" ref="managedConnectionFactory"/>
</bean>

<bean id="component" class="mypackage.MyDaoImpl">
    <property name="connectionFactory" ref="connectionFactory"/>
</bean>

```

在托管模式下(即，在 Java EE 环境中)，配置可能类似于以下示例：

```

<jee:jndi-lookup id="connectionFactory" jndi-name="eis/cicseci"/>

<bean id="component" class="MyDaoImpl">
    <property name="connectionFactory" ref="connectionFactory"/>
</bean>

```

## 5.3. 将 CCI 访问建模为操作对象

`org.springframework.jca.cci.object` 软件包包含支持类，这些支持类使您可以以不同的方式访问 EIS：通过可重用的操作对象，类似于 Spring 的 JDBC 操作对象(请参见[数据访问一章的 JDBC 部分](#))。这通常封装了 CCI API。应用程序级 Importing 对象被传递给操作对象，因此它可以构造 Importing 记录，然后将接收到的记录数据转换为应用程序级输出对象并返回它。

### iNote

这种方法在内部基于 `CciTemplate` 类和 `RecordCreator` 或 `RecordExtractor` 接口，重用了 Spring 核心 CCI 支持的机制。

### 5.3.1. 使用 MappingRecordOperation

`MappingRecordOperation` 本质上与 `CciTemplate` 执行相同的工作，但代表一个特定的，预先配置的操作作为对象。它提供了两种模板方法来指定如何将 Importing 对象转换为 Importing 记录以及如何将输出记录转换为输出对象(记录 Map)：

- `createInputRecord(...)`：指定如何将 Importing 对象转换为 Importing Record
- `extractOutputData(...)`：指定如何从输出 Record 中提取输出对象

以下 Lists 显示了这些方法的签名：

```
public abstract class MappingRecordOperation extends EisOperation {
    ...
    protected abstract Record createInputRecord(RecordFactory recordFactory,
                                                Object inputObject) throws ResourceException, DataAccessException {
        // ...
    }

    protected abstract Object extractOutputData(Record outputRecord)
                                                throws ResourceException, SQLException, DataAccessException {
        // ...
    }

    ...
}
```

之后，要执行 EIS 操作，您需要使用单个 `execute` 方法，传入应用程序级 Importing 对象并接收应用程序级输出对象作为结果。以下示例显示了如何执行此操作：

```
public abstract class MappingRecordOperation extends EisOperation {
    ...
    public Object execute(Object inputObject) throws DataAccessException {
    }
    ...
}
```

与 `CciTemplate` 类相反，此 `execute(...)` 方法没有 `InteractionSpec` 作为参数。相反，`InteractionSpec` 对操作是全局的。您必须使用以下构造函数实例化具有特定 `InteractionSpec` 的操作对象。以下示例显示了如何执行此操作：

```
InteractionSpec spec = ...;
MyMappingRecordOperation eisOperation = new MyMappingRecordOperation(getConnectionFactory)
    ...
}
```

### 5.3.2. 使用 MappingCommAreaOperation

一些连接器使用基于 COMMAREA 的记录，该记录表示一个字节数组，其中包含要发送到 EIS 的参数以及它返回的数据。Spring 提供了一个特殊的操作类，可以直接在 COMMAREA 上工作而不是在记录上工作。`MappingCommAreaOperation` 类扩展了 `MappingRecordOperation` 类以提供此特殊的 COMMAREA 支持。它隐式地使用 `CommAreaRecord` 类作为 Importing 和输出记录类型，并提供了两种新方法将 Importing 对象转换为 ImportingCOMMAREA 并将输出 COMMAREA 转换为输出对象。以下 Lists 显示了相关的方法签名：

```
public abstract class MappingCommAreaOperation extends MappingRecordOperation {  
    ...  
  
    protected abstract byte[] objectToBytes(Object inObject)  
        throws IOException, DataAccessException;  
  
    protected abstract Object bytesToObject(byte[] bytes)  
        throws IOException, DataAccessException;  
  
    ...  
}
```

### 5.3.3. 自动输出记录生成

由于每个 `MappingRecordOperation` 子类内部都基于 `CciTemplate`，因此可以使用与 `CciTemplate` 相同的自动生成输出记录的方式。每个操作对象都提供相应的 `setOutputRecordCreator(...)` 方法。有关更多信息，请参见[自动输出记录生成](#)。

### 5.3.4. Summary

操作对象方法以与 `CciTemplate` 类相同的方式使用记录。

表 10. 交互执行方法的用法

<code>MappingRecordOperation</code> 方法签名	<code>MappingRecordOperation</code> 属性 <code>outputRecordCreator</code>
<code>Object execute(Object)</code>	未设置
<code>Object execute(Object)</code>	已设置

### 5.3.5. `MappingRecordOperation` 用法示例

在本节中，我们显示如何使用 `MappingRecordOperation` 通过 Blackbox CCI 连接器访问数据库。

#### ① Note

该连接器的原始版本由 Java EE SDK(1.3 版)提供，可以从 Oracle 获得。

首先，您必须对 CCI `InteractionSpec` 进行一些初始化以指定要执行的 SQL 请求。在以下示例中，我们直接定义将请求的参数转换为 CCI 记录的方法以及将 CCI 结果记录转换为 `Person` 类的实例的方法：

```
public class PersonMappingOperation extends MappingRecordOperation {

    public PersonMappingOperation(ConnectionFactory connectionFactory) {
        setConnectionFactory(connectionFactory);
        CciInteractionSpec interactionSpec = new CciConnectionSpec();
        interactionSpec.setSql("select * from person where person_id=?");
        setInteractionSpec(interactionSpec);
    }

    protected Record createInputRecord(RecordFactory recordFactory,
                                      Object inputObject) throws ResourceException {
        Integer id = (Integer) inputObject;
        IndexedRecord input = recordFactory.createIndexedRecord("input");
        input.add(new Integer(id));
    }
}
```

```

        return input;
    }

protected Object extractOutputData(Record outputRecord)
    throws ResourceException, SQLException {
    ResultSet rs = (ResultSet) outputRecord;
    Person person = null;
    if (rs.next()) {
        Person person = new Person();
        person.setId(rs.getInt("person_id"));
        person.setLastName(rs.getString("person_last_name"));
        person.setFirstName(rs.getString("person_first_name"));
    }
    return person;
}
}

```

然后，应用程序可以使用人员标识符作为参数来执行操作对象。请注意，您可以将操作对象设置为共享实例，因为它是线程安全的。下面以人员标识符作为参数执行操作对象：

```

public class MyDaoImpl extends CciDaoSupport implements MyDao {

    public Person getPerson(int id) {
        PersonMappingOperation query = new PersonMappingOperation(getConnectionFactory());
        Person person = (Person) query.execute(new Integer(id));
        return person;
    }
}

```

在非托管模式下，Spring Bean 的相应配置如下：

```

<bean id="managedConnectionFactory"
      class="com.sun.connector.cciblackbox.CciLocalTxManagedConnectionFactory">
    <property name="connectionURL" value="jdbc:hsqldb:hsq://localhost:9001"/>
    <property name="driverName" value="org.hsqldb.jdbcDriver"/>
</bean>

<bean id="targetConnectionFactory"
      class="org.springframework.jca.support.LocalConnectionFactoryBean">
    <property name="managedConnectionFactory" ref="managedConnectionFactory"/>
</bean>

<bean id="connectionFactory"
      class="org.springframework.jca.cci.connection.ConnectionSpecConnectionFactoryAd
<property name="targetConnectionFactory" ref="targetConnectionFactory"/>
<property name="connectionSpec">
    <bean class="com.sun.connector.cciblackbox.CciConnectionSpec">
        <property name="user" value="sa"/>
        <property name="password" value="" />
    </bean>
</property>
</bean>

<bean id="component" class="MyDaoImpl">

```

```
<property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

在托管模式下(即，在 Java EE 环境中)，配置可以如下：

```
<jee:jndi-lookup id="targetConnectionFactory" jndi-name="eis/blackbox"/>

<bean id="connectionFactory"
      class="org.springframework.jca.cci.connection.ConnectionSpecConnectionFactoryAd
<property name="targetConnectionFactory" ref="targetConnectionFactory"/>
<property name="connectionSpec">
    <bean class="com.sun.connector.cciblackbox.CciConnectionSpec">
        <property name="user" value="sa"/>
        <property name="password" value="" />
    </bean>
</property>
</bean>

<bean id="component" class="MyDaoImpl">
    <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

### 5.3.6. MappingCommAreaOperation 用法示例

在本部分中，我们将说明如何使用 [MappingCommAreaOperation](#) 的用法通过 IBM CICS ECI 连接器以 ECI 模式访问 CICS。

首先，我们需要初始化 CCI [InteractionSpec](#) 以指定要访问哪个 CICS 程序以及如何与之交互，如以下示例所示：

```
public abstract class EciMappingOperation extends MappingCommAreaOperation {

    public EciMappingOperation(ConnectionFactory connectionFactory, String programName)
        setConnectionFactory(connectionFactory);
        ECIInteractionSpec interactionSpec = new ECIInteractionSpec(),
        interactionSpec.setFunctionName(programName);
        interactionSpec.setInteractionVerb(ECIInteractionSpec.SYNC_SEND_RECEIVE);
        interactionSpec.setCommareaLength(30);
        setInteractionSpec(interactionSpec);
        setOutputRecordCreator(new EciOutputRecordCreator());
    }

    private static class EciOutputRecordCreator implements RecordCreator {
        public Record createRecord(RecordFactory recordFactory) throws ResourceException
            return new CommAreaRecord();
        }
    }
}
```

然后，我们可以继承抽象的 `EciMappingOperation` 类，以指定自定义对象和 `Records` 之间的 Map，如以下示例所示：

```
public class MyDaoImpl extends CciDaoSupport implements MyDao {  
  
    public OutputObject getData(Integer id) {  
        EciMappingOperation query = new EciMappingOperation(getConnectionFactory(), "MY  
  
        protected abstract byte[] objectToBytes(Object inObject) throws IOException;  
        Integer id = (Integer) inObject;  
        return String.valueOf(id);  
    }  
  
    protected abstract Object bytesToObject(byte[] bytes) throws IOException;  
    String str = new String(bytes);  
    String field1 = str.substring(0,6);  
    String field2 = str.substring(6,1);  
    String field3 = str.substring(7,1);  
    return new OutputObject(field1, field2, field3);  
    }  
});  
  
    return (OutputObject) query.execute(new Integer(id));  
}  
  
}
```

在非托管模式下，Spring Bean 的相应配置如下：

```
<bean id="managedConnectionFactory" class="com.ibm.connector2.cics.ECIManagedConnection  
    <property name="serverName" value="TXSERIES"/>  
    <property name="connectionURL" value="local:"/>  
    <property name="userName" value="CICSUSER"/>  
    <property name="password" value="CICS" />  
</bean>  
  
<bean id="connectionFactory" class="org.springframework.jca.support.LocalConnectionFacto  
    <property name="managedConnectionFactory" ref="managedConnectionFactory" />  
</bean>  
  
<bean id="component" class="MyDaoImpl">  
    <property name="connectionFactory" ref="connectionFactory" />  
</bean>
```

在托管模式下(即，在 Java EE 环境中)，配置可以如下：

```
<jee:jndi-lookup id="connectionFactory" jndi-name="eis/cicseci" />  
  
<bean id="component" class="MyDaoImpl">  
    <property name="connectionFactory" ref="connectionFactory" />  
</bean>
```

## 5.4. Transactions

JCA 为资源适配器指定了多个级别的事务支持。资源适配器支持的事务类型在其 `ra.xml` 文件中指定。本质上，有三个选项：无(例如，使用 CICS EPI 连接器)，本地事务(例如，使用 CICS ECI 连接器)和全局事务(例如，使用 IMS 连接器)。以下示例配置了全局选项：

```
<connector>
    <resourceadapter>
        <!-- <transaction-support>NoTransaction</transaction-support> -->
        <!-- <transaction-support>LocalTransaction</transaction-support> -->
        <transaction-support>XATransaction</transaction-support>
    <resourceadapter>
<connector>
```

对于全局事务，您可以使用 Spring 的通用事务基础结构来划分事务，以 `JtaTransactionManager` 作为后端(委托给下面的 Java EE 服务器的分布式事务处理协调器)。

对于单个 CCI `ConnectionFactory` 上的本地事务，Spring 为 CCI 提供了一种特定的事务 `Management` 策略，类似于 JDBC 的 `DataSourceTransactionManager`。CCI API 定义了本地事务对象和相应的本地事务划分方法。Spring 的 `CciLocalTransactionManager` 以完全符合 Spring 的通用 `PlatformTransactionManager` 抽象的方式执行此类本地 CCI 事务。以下示例配置 `CciLocalTransactionManager`：

```
<jee:jndi-lookup id="eciConnectionFactory" jndi-name="eis/cicseci"/>

<bean id="eciTransactionManager"
      class="org.springframework.jca.cci.connection.CciLocalTransactionManager">
    <property name="connectionFactory" ref="eciConnectionFactory"/>
</bean>
```

您可以将这两种事务策略与 Spring 的任何事务划分工具一起使用，无论是声明式还是程序式。这是 Spring 通用的 `PlatformTransactionManager` 抽象的结果，该抽象将事务划分与实际执行策略分离。您可以根据需要在 `JtaTransactionManager` 和 `CciLocalTransactionManager` 之间切换，从而保持事务划分不变。

有关 SpringTransaction 功能的更多信息, 请参见[Transaction Management](#)。

## 6. Email

---

本节介绍如何使用 Spring Framework 发送电子邮件。

### Library dependencies

为了使用 Spring Framework 的电子邮件库, 以下 JAR 必须位于应用程序的 Classpath 中:

- [JavaMail](#) 库

该库可以在 Web 上免费使用, 例如在 Maven Central 中为 `com.sun.mail:javax.mail`。

Spring 框架提供了一个有用的 Util 库, 用于发送电子邮件, 使您不受底层邮件系统的限制, 并负责代表 Client 端进行低级资源处理。

`org.springframework.mail` 软件包是 Spring 框架的电子邮件支持的根级软件包。发送电子邮件的中央接口是 `MailSender` 接口。封装 `from` 和 `to` (以及许多其他邮件)之类的简单邮件的属性的简单值对象是 `SimpleMailMessage` 类。该软件包还包含已检查异常的层次结构, 该层次结构提供了比较低级别的邮件系统异常更高的抽象级别, 根异常为 `MailException`。有关富邮件异常层次结构的更多信息, 请参见[javadoc](#)。

`org.springframework.mail.javamail.JavaMailSender` 接口向 `MailSender` 接口(从中继承)提供了特殊的 JavaMail 功能, 例如 MIME 消息支持。`JavaMailSender` 还提供了称为 `org.springframework.mail.javamail.MimeMessagePreparator` 的回调接口, 用于准备 `MimeMessage`。

### 6.1. Usage

假设我们有一个名为 `OrderManager` 的业务接口, 如以下示例所示:

```
public interface OrderManager {  
    void placeOrder(Order order);  
}
```

进一步假设我们有一个要求，说明需要生成带有订单号的电子邮件消息并将其发送给下订单的 Client。

### 6.1.1. MailSender 和 SimpleMailMessage 的基本用法

以下示例显示了有人下订单时如何使用 `MailSender` 和 `SimpleMailMessage` 发送电子邮件：

```
import org.springframework.mail.MailException;  
import org.springframework.mail.MailSender;  
import org.springframework.mail.SimpleMailMessage;  
  
public class SimpleOrderManager implements OrderManager {  
  
    private MailSender mailSender;  
    private SimpleMailMessage templateMessage;  
  
    public void setMailSender(MailSender mailSender) {  
        this.mailSender = mailSender;  
    }  
  
    public void setTemplateMessage(SimpleMailMessage templateMessage) {  
        this.templateMessage = templateMessage;  
    }  
  
    public void placeOrder(Order order) {  
  
        // Do the business calculations...  
  
        // Call the collaborators to persist the order...  
  
        // Create a thread safe "copy" of the template message and customize it  
        SimpleMailMessage msg = new SimpleMailMessage(this.templateMessage);  
        msg.setTo(order.getCustomer().getEmailAddress());  
        msg.setText(  
            "Dear " + order.getCustomer().getFirstName()  
            + order.getCustomer().getLastName()  
            + ", thank you for placing order. Your order number is "  
            + order.getOrderNumber());  
        try{  
            this.mailSender.send(msg);  
        }  
        catch (MailException ex) {  
            // simply log it and go on...  
            System.err.println(ex.getMessage());  
        }  
    }  
}
```

```
}
```

以下示例显示了上述代码的 bean 定义：

```
<bean id="mailSender" class="org.springframework.mail.javamail.JavaMailSenderImpl">
    <property name="host" value="mail.mycompany.com"/>
</bean>

<!-- this is a template message that we can pre-load with default state -->
<bean id="templateMessage" class="org.springframework.mail.SimpleMailMessage">
    <property name="from" value="[emailprotected]"/>
    <property name="subject" value="Your order"/>
</bean>

<bean id="orderManager" class="com.mycompany.businessapp.support.SimpleOrderManager">
    <property name="mailSender" ref="mailSender"/>
    <property name="templateMessage" ref="templateMessage"/>
</bean>
```

## 6.1.2. 使用 JavaMailSender 和 MimeMessagePreparator

本节描述了使用 `MimeMessagePreparator` 回调接口的 `OrderManager` 的另一种实现。在下面的示例中，`mailSender` 属性的类型为 `JavaMailSender`，因此我们可以使用 JavaMail `MimeMessage` 类：

```
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

import javax.mail.internet.MimeMessage;
import org.springframework.mail.MailException;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.mail.javamail.MimeMessagePreparator;

public class SimpleOrderManager implements OrderManager {

    private JavaMailSender mailSender;

    public void setMailSender(JavaMailSender mailSender) {
        this.mailSender = mailSender;
    }

    public void placeOrder(final Order order) {
        // Do the business calculations...
        // Call the collaborators to persist the order...

        MimeMessagePreparator preparator = new MimeMessagePreparator() {
            public void prepare(MimeMessage mimeMessage) throws Exception {

```

```

        mimeMessage.setRecipient(Message.RecipientType.TO,
            new InternetAddress(order.getCustomer().getEmailAddress())));
        mimeMessage.setFrom(new InternetAddress("[emailprotected]"));
        mimeMessage.setText("Dear " + order.getCustomer().getFirstName() + " "
            + order.getCustomer().getLastName() + ", thanks for your order.
            Your order number is " + order.getOrderNumber() + ".");
    }
}

try {
    this.mailSender.send(preparator);
}
catch (MailException ex) {
    // simply log it and go on...
    System.err.println(ex.getMessage());
}
}

}

```

### **Note**

邮件代码是一个横切关注点，很可能是重构为[自定义 Spring AOP 方面](#)的候选者，然后可以在 `OrderManager` 目标上的适当连接点处执行。

Spring Framework 的邮件支持随附于标准 JavaMail 实现。有关更多信息，请参见相关的 Javadoc。  
。

## 6.2. 使用 JavaMail MimeMessageHelper

处理 JavaMail 消息时非常方便的类是

`org.springframework.mail.javamail.MimeMessageHelper`，这使您不必使用冗长的 JavaMail API。使用 `MimeMessageHelper`，很容易创建 `MimeMessage`，如以下示例所示：

```

// of course you would use DI in any real-world cases
JavaMailSenderImpl sender = new JavaMailSenderImpl();
sender.setHost("mail.host.com");

MimeMessage message = sender.createMimeMessage();
MimeMessageHelper helper = new MimeMessageHelper(message);
helper.setTo("[emailprotected]");
helper.setText("Thank you for ordering!");

sender.send(message);

```

## 6.2.1. 发送附件和内联资源

Multipart 电子邮件允许同时使用附件和内联资源。内联资源的示例包括您要在邮件中使用但不希望显示为附件的图像或样式表。

### Attachments

下面的示例向您展示如何使用 `MimeMessageHelper` 发送带有单个 JPEG 图像附件的电子邮件：

```
JavaMailSenderImpl sender = new JavaMailSenderImpl();
sender.setHost("mail.host.com");

MimeMessage message = sender.createMimeMessage();

// use the true flag to indicate you need a multipart message
MimeMessageHelper helper = new MimeMessageHelper(message, true);
helper.setTo("[emailprotected]");

helper.setText("Check out this image!");

// let's attach the infamous windows Sample file (this time copied to c:/)
FileSystemResource file = new FileSystemResource(new File("c:/Sample.jpg"));
helper.addAttachment("CoolImage.jpg", file);

sender.send(message);
```

### Inline Resources

下面的示例向您展示如何使用 `MimeMessageHelper` 发送带有嵌入式图像的电子邮件：

```
JavaMailSenderImpl sender = new JavaMailSenderImpl();
sender.setHost("mail.host.com");

MimeMessage message = sender.createMimeMessage();

// use the true flag to indicate you need a multipart message
MimeMessageHelper helper = new MimeMessageHelper(message, true);
helper.setTo("[emailprotected]");

// use the true flag to indicate the text included is HTML
helper.setText("<html><body><img src='cid:identifier1234'></body></html>", true);

// let's include the infamous windows Sample file (this time copied to c:/)
FileSystemResource res = new FileSystemResource(new File("c:/Sample.jpg"));
helper.addInline("identifier1234", res);

sender.send(message);
```

## ⚠Warning

内联资源通过使用指定的 `Content-ID` (在上例中为 `identifier1234`) 添加到 `MimeMessage`。添加文本和资源的 `Sequences` 非常重要。确保首先添加文本，然后添加资源。如果您正相反进行操作，则此操作无效。

### 6.2.2. 使用模板库创建电子邮件内容

上一节中显示的示例中的代码通过使用诸如 `message.setText(...)` 之类的方法调用显式创建了电子邮件的内容。这对于简单的情况很好，并且在上述示例的上下文中也可以，其目的是向您展示 API 的基本知识。

但是，在典型的企业应用程序中，由于多种原因，开发人员通常不使用以前显示的方法来创建电子邮件的内容：

- 用 Java 代码创建基于 HTML 的电子邮件内容很繁琐且容易出错。
- 显示逻辑和业务逻辑之间没有明确区分。
- 更改电子邮件内容的显示结构需要编写 Java 代码，重新编译，重新部署等。

通常，解决这些问题的方法是使用模板库(例如 FreeMarker)来定义电子邮件内容的显示结构。这使您的代码只能执行创建要在电子邮件模板中呈现的数据并发送电子邮件的任务。当您的电子邮件的内容变得相当复杂时，这绝对是一种最佳实践，而且，借助 Spring Framework 的 FreeMarker 支持类，它变得非常容易实现。

## 7.任务执行和计划

Spring 框架分别通过 `TaskExecutor` 和 `TaskScheduler` 接口为任务的异步执行和调度提供了抽象。Spring 还提供了那些接口的实现，这些接口在应用程序服务器环境中支持线程池或委托给 CommonJ。最终，在公共接口后面使用这些实现可以抽象化 Java SE 5, Java SE 6 和 Java EE 环境之间的差异。

Spring 还具有集成类，以支持 `Timer` (自 1.3 起成为 JDK 的一部分) 和 Quartz Scheduler (<http://quartz-scheduler.org>) 进行调度。您可以使用 `FactoryBean` 分别设置对 `Timer` 或 `Trigger` 实例的可选引用来设置这两个调度程序。此外，还提供了 Quartz Scheduler 和 `Timer` 的便捷类，使您可以调用现有目标对象的方法(类似于正常的 `MethodInvokingFactoryBean` 操作)。

## 7.1. Spring TaskExecutor 抽象

执行程序是线程池概念的 JDK 名称。“执行程序”的命名是由于不能保证基础实现实际上是一个池。执行程序可能是单线程的，甚至是同步的。Spring 的抽象隐藏了 Java SE 和 Java EE 环境之间的实现细节。

Spring 的 `TaskExecutor` 界面与 `java.util.concurrent.Executor` 界面相同。实际上，最初，其存在的主要原因是在使用线程池时抽象出对 Java 5 的需求。该接口具有单个方法(`execute(Runnable task)`)，该方法根据线程池的语义和配置接受要执行的任务。

`TaskExecutor` 最初是为了在需要时为其他 Spring 组件提供线程池抽象而创建的。

`ApplicationEventMulticaster`，JMS 的 `AbstractMessageListenerContainer` 和 Quartz 集成之类的组件都使用 `TaskExecutor` 抽象来池化线程。但是，如果您的 bean 需要线程池行为，则也可以根据自己的需要使用此抽象。

### 7.1.1. TaskExecutor 类型

Spring 包含许多 `TaskExecutor` 的预构建实现。您极有可能无需实现自己的方法。Spring 提供的变体如下：

- `SyncTaskExecutor`：此实现不会异步执行调用。而是，每个调用都在调用线程中进行。它主要用于不需要多线程的情况下，例如在简单的测试案例中。
- `SimpleAsyncTaskExecutor`：此实现不重用任何线程。而是，它为每次调用启动一个新线程

。但是，它确实支持并发限制，该限制会阻止超出限制的所有调用，直到释放插槽为止。如果您正在寻找 true 的池，请参阅 `ThreadPoolTaskExecutor`，在此列表的后面。

- `ConcurrentTaskExecutor`：此实现是 `java.util.concurrent.Executor` 实例的适配器。还有一个替代方法(`ThreadPoolTaskExecutor`)，它将 `Executor` 配置参数公开为 bean 属性。很少需要直接使用 `ConcurrentTaskExecutor`。但是，如果 `ThreadPoolTaskExecutor` 不够灵活，无法满足您的需求，则可以选择 `ConcurrentTaskExecutor`。
- `ThreadPoolTaskExecutor`：最常使用此实现。它公开了用于配置 `java.util.concurrent.ThreadPoolExecutor` 的 bean 属性，并将其包装在 `TaskExecutor` 中。如果您需要适应另一种 `java.util.concurrent.Executor`，我们建议您使用 `ConcurrentTaskExecutor` 代替。
- `WorkManagerTaskExecutor`：此实现使用 CommonJ `WorkManager` 作为其支持服务提供者，并且是在 Spring 应用程序上下文中的 WebLogic 或 WebSphere 上设置基于 CommonJ 的线程池集成的中心便利类。
- `DefaultManagedTaskExecutor`：此实现在兼容 JSR-236 的运行时环境(例如 Java EE 7 应用程序服务器)中使用 JNDI 获得的 `ManagedExecutorService`，为此替换了 CommonJ WorkManager。

## 7.1.2. 使用 TaskExecutor

Spring 的 `TaskExecutor` 实现用作简单的 JavaBean。在以下示例中，我们定义一个使用

`ThreadPoolTaskExecutor` 异步打印出一组消息的 bean：

```
import org.springframework.core.task.TaskExecutor;
public class TaskExecutorExample {
    private class MessagePrinterTask implements Runnable {
```

```

private String message;

public MessagePrinterTask(String message) {
    this.message = message;
}

public void run() {
    System.out.println(message);
}
}

private TaskExecutor taskExecutor;

public TaskExecutorExample(TaskExecutor taskExecutor) {
    this.taskExecutor = taskExecutor;
}

public void printMessages() {
    for(int i = 0; i < 25; i++) {
        taskExecutor.execute(new MessagePrinterTask("Message" + i));
    }
}
}

```

如您所见，您不必将 `Runnable` 添加到队列中，而是从池中检索线程并自行执行。然后 `TaskExecutor` 使用其内部规则来决定何时执行任务。

要配置 `TaskExecutor` 使用的规则，我们公开了简单的 `bean` 属性：

```

<bean id="taskExecutor" class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor">
    <property name="corePoolSize" value="5"/>
    <property name="maxPoolSize" value="10"/>
    <property name="queueCapacity" value="25"/>
</bean>

<bean id="taskExecutorExample" class="TaskExecutorExample">
    <constructor-arg ref="taskExecutor"/>
</bean>

```

## 7.2. Spring TaskScheduler 抽象

除了 `TaskExecutor` 抽象之外，Spring 3.0 引入了 `TaskScheduler`，它具有多种用于计划任务在将来某个 Moment 运行的方法。以下 Lists 显示了 `TaskScheduler` 接口定义：

```

public interface TaskScheduler {
    ScheduledFuture schedule(Runnable task, Trigger trigger);
}

```

```
ScheduledFuture schedule(Runnable task, Instant startTime);

ScheduledFuture schedule(Runnable task, Date startTime);

ScheduledFuture scheduleAtFixedRate(Runnable task, Instant startTime, Duration period);

ScheduledFuture scheduleAtFixedRate(Runnable task, Date startTime, long period);

ScheduledFuture scheduleAtFixedRate(Runnable task, Duration period);

ScheduledFuture scheduleAtFixedRate(Runnable task, long period);

ScheduledFuture scheduleWithFixedDelay(Runnable task, Instant startTime, Duration delay);

ScheduledFuture scheduleWithFixedDelay(Runnable task, Date startTime, long delay);

ScheduledFuture scheduleWithFixedDelay(Runnable task, Duration delay);

ScheduledFuture scheduleWithFixedDelay(Runnable task, long delay);

}
```

最简单的方法是名为 `schedule` 的方法，该方法只包含 `Runnable` 和 `Date`。这将导致任务在指定时间后运行一次。所有其他方法都可以安排任务重复运行。固定速率和固定延迟方法用于简单的定期执行，但是接受 `Trigger` 的方法更加灵活。

### 7.2.1. 触发界面

`Trigger` 接口实质上受 JSR-236 的启发，该 JSR-236 在 Spring 3.0 之前尚未正式实现。

`Trigger` 的基本思想是可以根据过去的执行结果甚至任意条件来确定执行时间。如果这些确定确实考虑了先前执行的结果，则该信息在 `TriggerContext` 内可用。`Trigger` 接口本身非常简单，如以下 Lists 所示：

```
public interface Trigger {

    Date nextExecutionTime(TriggerContext triggerContext);
}
```

`TriggerContext` 是最重要的部分。它封装了所有相关数据，如有必要，将来可以扩展。

`TriggerContext` 是一个接口(默认情况下使用 `SimpleTriggerContext` 实现)。以下 Lists 显示了 `Trigger` 实现的可用方法。

```
public interface TriggerContext {  
    Date lastScheduledExecutionTime();  
    Date lastActualExecutionTime();  
    Date lastCompletionTime();  
}
```

## 7.2.2. 触发实施

Spring 提供了 `Trigger` 接口的两种实现。最有趣的是 `CronTrigger`。它启用了基于 cron 表达式的任务调度。例如，以下任务计划在每小时的 15 分钟后运行，但仅在工作日的 9 到 5 个“工作时间”内运行：

```
scheduler.schedule(task, new CronTrigger("0 15 9-17 * * MON-FRI"));
```

另一种实现是 `PeriodicTrigger`，它接受固定的时间段，可选的初始延迟值和布尔值，以指示该时间段应被解释为固定速率还是固定延迟。由于 `TaskScheduler` 接口已经定义了用于以固定速率或固定延迟计划任务的方法，因此应尽可能直接使用这些方法。`PeriodicTrigger` 实现的价值在于您可以在依赖 `Trigger` 抽象的组件中使用它。例如，允许周期性触发器，基于 cron 的触发器，甚至自定义触发器实现可互换使用可能很方便。这样的组件可以利用依赖注入的优势，以便您可以在外部配置 `Triggers`，因此可以轻松地对其进行修改或扩展。

## 7.2.3. TaskScheduler 的实现

像 Spring 的 `TaskExecutor` 抽象一样，`TaskScheduler` 安排的主要好处是应用程序的调度需求与部署环境分离。当部署到不应由应用程序本身直接创建线程的应用程序服务器环境时，此抽象级别特别重要。对于这种情况，Spring 提供了 `TimerManagerTaskScheduler`，它委派给 WebLogic 或 WebSphere 上的 CommonJ `TimerManager`，而最新的 `DefaultManagedTaskScheduler` 则委派给 Java EE 7 环境中的 JSR-236 `ManagedScheduledExecutorService`。两者通常都配置有 JNDI 查找。

每当不需要外部线程 Management 时，一个更简单的选择就是在应用程序中进行本地 `ScheduledExecutorService` 设置，可以通过 Spring 的 `ConcurrentTaskScheduler` 进行调整。为了方便起见，Spring 还提供了 `ThreadPoolTaskScheduler`，它在内部委托 `ScheduledExecutorService` 来提供与 `ThreadPoolTaskExecutor` 相似的通用 bean 样式配置。这些变体也适用于宽松的应用程序服务器环境中的本地嵌入式线程池设置，尤其是在 Tomcat 和 Jetty 上。

## 7.3. 计划和异步执行的 Comments 支持

Spring 为任务调度和异步方法执行提供 Comments 支持。

### 7.3.1. 启用计划 Comments

要启用对 `@Scheduled` 和 `@Async` 注解的支持，可以将 `@EnableScheduling` 和 `@EnableAsync` 添加到您的 `@Configuration` 类中，如以下示例所示：

```
@Configuration  
@EnableAsync  
@EnableScheduling  
public class AppConfig {  
}
```

您可以选择与应用程序相关的 Comments。例如，如果只需要支持 `@Scheduled`，则可以省略 `@EnableAsync`。为了获得更细粒度的控制，您可以另外实现 `SchedulingConfigurer` 接口和 / 或 `AsyncConfigurer` 接口。有关完整的详细信息，请参见 [SchedulingConfigurer](#) 和 [AsyncConfigurer](#) javadoc。

如果您更喜欢 XML 配置，则可以使用 `<task:annotation-driven>` 元素，如以下示例所示：

```
<task:annotation-driven executor="myExecutor" scheduler="myScheduler"/>  
<task:executor id="myExecutor" pool-size="5"/>  
<task:scheduler id="myScheduler" pool-size="10"/>
```

请注意，使用前面的 XML，提供了执行程序引用来处理与带有 `@Async` 注解的方法相对应的那些任务，并且提供了调度程序引用来 Management 用 `@Scheduled` Comments 的那些方法。

#### iNote

处理 `@Async` 注解的默认建议模式是 `proxy`，该模式仅允许通过代理拦截呼叫。同一类内的本地调用无法以这种方式被拦截。对于更高级的侦听模式，请考虑结合编译时或加载时编织切换到 `aspectj` 模式。

### 7.3.2. @Scheduled 注解

您可以将 `@Scheduled` Comments 以及触发器元数据添加到方法中。例如，以下方法每隔五秒钟以固定的延迟被调用一次，这意味着该时间段是从每个先前调用的完成时间开始计算的：

```
@Scheduled(fixedDelay=5000)
public void doSomething() {
    // something that should execute periodically
}
```

如果需要固定汇率执行，则可以更改 Comments 中指定的属性名称。每五秒钟调用一次以下方法（在每次调用的连续开始时间之间测量）：

```
@Scheduled(fixedRate=5000)
public void doSomething() {
    // something that should execute periodically
}
```

对于固定延迟和固定速率的任务，可以通过指示在第一次执行该方法之前要 `await` 的毫秒数来指定初始延迟，如以下 `fixedRate` 示例所示：

```
@Scheduled(initialDelay=1000, fixedRate=5000)
public void doSomething() {
    // something that should execute periodically
}
```

如果简单的定期调度不足以表现出来，则可以提供 cron 表达式。例如，以下仅在工作日执行：

```
@Scheduled(cron="*/5 * * * * MON-FRI")
public void doSomething() {
    // something that should execute on weekdays only
}
```

### Tip

您还可以使用 `zone` 属性指定解析 cron 表达式的时区。

请注意，要调度的方法必须具有空返回值，并且不能期望任何参数。如果该方法需要与应用程序上下文中的其他对象进行交互，则通常将通过依赖项注入来提供这些对象。

### Note

从 Spring Framework 4.3 开始，任何范围的 bean 都支持 `@Scheduled` 方法。

确保不要在运行时初始化同一 `@Scheduled` `Comments` 类的多个实例，除非您确实希望为每个此类实例计划回调。与此相关，请确保不要在用 `@Scheduled` `Comments` 并已在容器中注册为常规 Spring Bean 的 bean 类上使用 `@Configurable`。否则，您将获得双重初始化(一次通过容器，一次通过 `@Configurable` 方面)，从而导致每个 `@Scheduled` 方法被调用两次。

### 7.3.3. @Async 注解

您可以在方法上提供 `@Async` 注解，以便异步调用该方法。换句话说，调用者在调用后立即返回，而方法的实际执行发生在已提交给 Spring `TaskExecutor` 的任务中。在最简单的情况下，可以将 `Comments` 应用于返回 `void` 的方法，如以下示例所示：

```
@Async
void doSomething() {
    // this will be executed asynchronously
}
```

与用 `@Scheduled` CommentsComments 的方法不同，这些方法可以使用参数，因为它们在运行时由调用方以“常规”方式调用，而不是从容器 Management 的计划任务中调用。例如，以下代码是 `@Async` 注解的合法应用：

```
@Async  
void doSomething(String s) {  
    // this will be executed asynchronously  
}
```

即使返回值的方法也可以异步调用。但是，要求此类方法具有 `Future` 类型的返回值。这仍然提供了异步执行的好处，以便调用者可以在对该 `Future` 调用 `get()` 之前执行其他任务。下面的示例演示如何在返回值的方法上使用 `@Async`：

```
@Async  
Future<String> returnSomething(int i) {  
    // this will be executed asynchronously  
}
```

### Tip

`@Async` 方法不仅可以声明常规 `java.util.concurrent.Future` 返回类型，还可以声明 Spring 的 `org.springframework.util.concurrent.ListenableFuture`，或者声明为 Spring 4.2 的 JDK 8 的 `java.util.concurrent.CompletableFuture`，以与异步任务进行更丰富的交互并通过进一步的处理步骤立即进行组合。

您不能将 `@Async` 与 `@PostConstruct` 之类的生命周期回调结合使用。要异步初始化 Spring Bean，当前必须使用一个单独的初始化 Spring Bean，然后在目标上调用 `@Async` 带 Comments 的方法，如以下示例所示：

```
public class SampleBeanImpl implements SampleBean {  
  
    @Async  
    void doSomething() {  
        // ...  
    }  
}
```

```

    }

}

public class SampleBeanInitializer {

    private final SampleBean bean;

    public SampleBeanInitializer(SampleBean bean) {
        this.bean = bean;
    }

    @PostConstruct
    public void initialize() {
        bean.doSomething();
    }

}

```

### iNote

`@Async` 没有直接的 XML 等效项，因为此类方法应首先设计用于异步执行，而不是在外部重新声明为异步。但是，您可以结合使用自定义切入点，通过 Spring AOP 手动设置 Spring 的 `AsyncExecutionInterceptor`。

#### 7.3.4. @Async 执行者资格

默认情况下，在方法上指定 `@Async` 时，使用的执行程序是[在启用异步支持时配置](#)，即，如果您使用的是 XML 或 `AsyncConfigurer` 实现(如果有)，则为“Comments 驱动”元素。但是，当需要指示在执行给定方法时应使用默认值以外的 Actuator 时，可以使用 `@Async` 注解的 `value` 属性。以下示例显示了如何执行此操作：

```

@Async("otherExecutor")
void doSomething(String s) {
    // this will be executed asynchronously by "otherExecutor"
}

```

在这种情况下，`"otherExecutor"` 可以是 Spring 容器中任何 `Executor` bean 的名称，也可以是与任何 `Executor` 关联的限定符的名称(例如，由 `<qualifier>` 元素或 Spring 的 `@Qualifier` Comments 指定)。

### 7.3.5. @Async 的异常 Management

当 `@Async` 方法具有 `Future` 类型的返回值时，很容易 Management 在方法执行期间引发的异常，因为在 `Future` 结果上调用 `get` 时会引发此异常。但是，对于 `void` 返回类型，该异常未被捕获并且无法发送。您可以提供 `AsyncUncaughtExceptionHandler` 来处理此类异常。以下示例显示了如何执行此操作：

```
public class MyAsyncUncaughtExceptionHandler implements AsyncUncaughtExceptionHandler {  
  
    @Override  
    public void handleUncaughtException(Throwable ex, Method method, Object... params)  
        // handle exception  
    }  
}
```

默认情况下，仅记录异常。您可以使用 `AsyncConfigurer` 或 `<task:annotation-driven/>` XML 元素定义自定义 `AsyncUncaughtExceptionHandler`。

## 7.4. 任务命名空间

从 3.0 版开始，Spring 包含用于配置 `TaskExecutor` 和 `TaskScheduler` 实例的 XML 名称空间。它还提供了一种方便的方法来配置要通过触发器安排的任务。

### 7.4.1. “调度程序”元素

以下元素创建具有指定线程池大小的 `ThreadPoolTaskScheduler` 实例：

```
<task:scheduler id="scheduler" pool-size="10"/>
```

为 `id` 属性提供的值用作池中线程名称的前缀。`scheduler` 元素相对简单。如果不提供 `pool-size` 属性，则默认线程池只有一个线程。调度程序没有其他配置选项。

### 7.4.2. 执行者元素

下面创建一个 `ThreadPoolTaskExecutor` 实例：

```
<task:executor id="executor" pool-size="10"/>
```

与 [previous section](#) 中显示的调度程序一样，为 `id` 属性提供的值用作池中线程名称的前缀。就池大小而言，`executor` 元素比 `scheduler` 元素支持更多的配置选项。一方面，`ThreadPoolTaskExecutor` 的线程池本身更具可配置性。执行者的线程池不仅可以具有单个大小，而且可以具有不同的核心和最大大小值。如果提供单个值，则执行程序具有固定大小的线程池(核心大小和最大大小相同)。但是，`executor` 元素的 `pool-size` 属性也接受 `min-max` 形式的范围。下面的示例将 `5` 的最小值设置为 `25` 的最大值：

```
<task:executor  
    id="executorWithPoolSizeRange"  
    pool-size="5-25"  
    queue-capacity="100"/>
```

在前面的配置中，还提供了 `queue-capacity` 值。还应根据执行者的队列容量来考虑线程池的配置。有关池大小和队列容量之间关系的完整描述，请参见[ThreadPoolExecutor](#)的文档。主要思想是，在提交任务时，如果活动线程数当前小于核心大小，则执行程序首先尝试使用空闲线程。如果已达到核心大小，则只要尚未达到其容量，就将任务添加到队列中。只有这样，如果达到队列的容量，执行程序才创建超出核心大小的新线程。如果还达到了最大大小，则执行者拒绝任务。

默认情况下，队列是无界的，但这很少是所需的配置，因为如果在所有池线程都忙时将足够的任务添加到该队列中，它将导致 `OutOfMemoryErrors`。此外，如果队列是无界的，则最大大小完全无效。由于执行程序总是在创建超出核心大小的新线程之前尝试队列，因此队列必须具有有限的容量，线程池才能超出核心大小(这就是为什么固定大小的池是使用时唯一明智的情况无限队列)。

如上所述，考虑拒绝任务的情况。默认情况下，当任务被拒绝时，线程池执行程序将抛出 `TaskRejectedException`。但是，拒绝策略实际上是可配置的。使用默认拒绝策略(即 `AbortPolicy` 实现)时会引发异常。对于在高负载下可以跳过某些任务的应用程序，您可以配置 `DiscardPolicy` 或 `DiscardOldestPolicy`。对于需要在重负载下限制提交的任务的应用程序

, 另一个很好的选择是 `CallerRunsPolicy`。该策略不会引发异常或放弃任务, 而是强制调用提交方法的线程运行任务本身。这个想法是这样的调用者在运行该任务时很忙, 无法立即提交其他任务。因此, 它提供了一种在保持线程池和队列限制的同时限制传入负载的简单方法。通常, 这使执行程序可以“赶上”它正在处理的任务, 从而释放队列, 池中或两者中的某些容量。您可以从 `executor` 元素上 `rejection-policy` 属性可用值的枚举中选择任何一个。

下面的示例显示一个 `executor` 元素, 该元素具有许多用于指定各种行为的属性:

```
<task:executor  
    id="executorWithCallerRunsPolicy"  
    pool-size="5-25"  
    queue-capacity="100"  
    rejection-policy="CALLER_RUNS" />
```

最后, `keep-alive` 设置确定线程终止之前可以保持空闲状态的时间限制(以秒为单位)。如果当前池中的线程数超过核心数, 则在 `await` 此时间而不处理任务之后, 多余的线程将被终止。时间值为零会导致多余的线程在执行任务后立即终止, 而不会在任务队列中保留后续工作。下面的示例将 `keep-alive` 值设置为两分钟:

```
<task:executor  
    id="executorWithKeepAlive"  
    pool-size="5-25"  
    keep-alive="120" />
```

### 7.4.3. “计划任务”元素

Spring 任务名称空间最强大的功能是支持配置要在 Spring Application Context 中调度的任务。这遵循类似于 Spring 中其他“方法调用者”的方法, 例如 JMS 命名空间提供的用于配置消息驱动的 POJO 的方法。基本上, `ref` 属性可以指向任何 `SpringManagement` 的对象, 而 `method` 属性提供要在该对象上调用的方法的名称。以下 Lists 显示了一个简单的示例:

```
<task:scheduled-tasks scheduler="myScheduler">  
    <task:scheduled ref="beanA" method="methodA" fixed-delay="5000" />  
</task:scheduled-tasks>  
  
<task:scheduler id="myScheduler" pool-size="10" />
```

调度程序由外部元素引用，并且每个单独的任务都包括其触发元数据的配置。在前面的示例中，该元数据定义了一个具有固定延迟的周期性触发器，该延迟指示了每个任务执行完成后要 `await` 的毫秒数。另一个选项是 `fixed-rate`，指示应该执行该方法的频率，而不管以前的执行需要花费多长时间。此外，对于 `fixed-delay` 和 `fixed-rate` 任务，您都可以指定一个“`initial-delay`”参数，指示首次执行该方法之前要 `await` 的毫秒数。为了获得更多控制，您可以改为提供 `cron` 属性。

以下示例显示了这些其他选项：

```
<task:scheduled-tasks scheduler="myScheduler">
    <task:scheduled ref="beanA" method="methodA" fixed-delay="5000" initial-delay="1000"/>
    <task:scheduled ref="beanB" method="methodB" fixed-rate="5000"/>
    <task:scheduled ref="beanC" method="methodC" cron="*/5 * * * * MON-FRI"/>
</task:scheduled-tasks>

<task:scheduler id="myScheduler" pool-size="10"/>
```

## 7.5. 使用 Quartz Scheduler

Quartz 使用 `Trigger`，`Job` 和 `JobDetail` 对象来实现各种作业的调度。有关 Quartz 的基本概念，请参见<http://quartz-scheduler.org>。为了方便起见，Spring 提供了两个类，这些类简化了在基于 Spring 的应用程序中使用 Quartz 的过程。

### 7.5.1. 使用 JobDetailFactoryBean

Quartz `JobDetail` 对象包含运行作业所需的所有信息。Spring 提供了 `JobDetailFactoryBean`，它提供了用于 XML 配置目的的 bean 样式的属性。考虑以下示例：

```
<bean name="exampleJob" class="org.springframework.scheduling.quartz.JobDetailFactoryBean">
    <property name="jobClass" value="example.ExampleJob" />
    <property name="jobDataAsMap">
        <map>
            <entry key="timeout" value="5" />
        </map>
    </property>
</bean>
```

作业详细信息配置包含运行作业所需的所有信息(`ExampleJob`)。超时在作业数据 Map 中指定。

作业数据 Map 可通过 `JobExecutionContext` (在执行时传递给您)获得，但是 `JobDetail` 也会从

Map 到作业实例属性的作业数据中获取其属性。因此，在以下示例中，`ExampleJob` 包含一个名为 `timeout` 的 bean 属性，并且 `JobDetail` 自动将其应用：

```
package example;

public class ExampleJob extends QuartzJobBean {

    private int timeout;

    /**
     * Setter called after the ExampleJob is instantiated
     * with the value from the JobDetailFactoryBean (5)
     */
    public void setTimeout(int timeout) {
        this.timeout = timeout;
    }

    protected void executeInternal(JobExecutionContext ctx) throws JobExecutionException
        // do the actual work
    }

}
```

您也可以使用作业数据 Map 中的所有其他属性。

#### iNote

通过使用 `name` 和 `group` 属性，您可以分别修改作业的名称和组。默认情况下，作业的名称与 `JobDetailFactoryBean` (在上面的示例中为 `exampleJob`) 的 bean 名称匹配。

### 7.5.2. 使用 MethodInvokingJobDetailFactoryBean

通常，您只需要在特定对象上调用方法。通过使用 `MethodInvokingJobDetailFactoryBean`，您可以完成此操作，如以下示例所示：

```
<bean id="jobDetail" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
    <property name="targetObject" ref="exampleBusinessObject" />
    <property name="targetMethod" value="doIt" />
</bean>
```

前面的示例导致在 `exampleBusinessObject` 方法上调用 `doIt` 方法，如以下示例所示：

```
public class ExampleBusinessObject {  
    // properties and collaborators  
  
    public void doIt() {  
        // do the actual work  
    }  
}
```

```
<bean id="exampleBusinessObject" class="examples.ExampleBusinessObject"/>
```

通过使用 `MethodInvokingJobDetailFactoryBean`，您无需创建仅调用方法的单行作业。您只需要创建实际的业务对象并连接详细对象即可。

缺省情况下，Quartz Jobs 是 Stateless 的，从而导致作业相互干扰的可能性。如果为相同的 `JobDetail` 指定两个触发器，则有可能在第一个作业完成之前启动第二个。如果 `JobDetail` 类实现 `Stateful` 接口，则不会发生。在第一个作业完成之前，第二个作业没有开始。要将 `MethodInvokingJobDetailFactoryBean` 产生的作业设为非并发，请将 `concurrent` 标志设置为 `false`，如以下示例所示：

```
<bean id="jobDetail" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">  
    <property name="targetObject" ref="exampleBusinessObject"/>  
    <property name="targetMethod" value="doIt"/>  
    <property name="concurrent" value="false"/>  
</bean>
```

#### iNote

默认情况下，作业将以并发方式运行。

### 7.5.3. 通过使用触发器和 `SchedulerFactoryBean` 来连接作业

我们已经创建了工作详细信息和工作。我们还回顾了便捷 bean，该 bean 使您可以在特定对象上调用方法。当然，我们仍然需要自己安排工作。这是通过使用触发器和 `SchedulerFactoryBean` 完成的。Quartz 中有几个触发器可用，Spring 提供了两个 Quartz `FactoryBean` 实现，它们带有方便的默认值：`CronTriggerFactoryBean` 和 `SimpleTriggerFactoryBean`。

触发器需要安排。Spring 提供了一个 `SchedulerFactoryBean`，它公开了要设置为属性的触发器

- `SchedulerFactoryBean` 通过这些触发器计划实际的作业。

以下 Lists 同时使用了 `SimpleTriggerFactoryBean` 和 `CronTriggerFactoryBean`：

```
<bean id="simpleTrigger" class="org.springframework.scheduling.quartz.SimpleTriggerFact
    <!-- see the example of method invoking job above -->
    <property name="jobDetail" ref="jobDetail"/>
    <!-- 10 seconds -->
    <property name="startDelay" value="10000"/>
    <!-- repeat every 50 seconds -->
    <property name="repeatInterval" value="50000"/>
</bean>

<bean id="cronTrigger" class="org.springframework.scheduling.quartz.CronTriggerFactoryB
    <property name="jobDetail" ref="exampleJob"/>
    <!-- run every morning at 6 AM -->
    <property name="cronExpression" value="0 0 6 * * ?"/>
</bean>
```

前面的示例设置了两个触发器，一个触发器每隔 50 秒运行一次，启动延迟为 10 秒，另一个触发器每天清晨 6 点运行。要完成所有工作，我们需要设置 `SchedulerFactoryBean`，如以下示例所示

：

```
<bean class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
    <property name="triggers">
        <list>
            <ref bean="cronTrigger"/>
            <ref bean="simpleTrigger"/>
        </list>
    </property>
</bean>
```

`SchedulerFactoryBean` 提供了更多属性，例如作业详细信息使用的 `calendar`，用于自定义 Quartz 的属性以及其他属性。有关更多信息，请参见 [SchedulerFactoryBean javadoc](#)。

## 8.缓存抽象

从 3.1 版开始，Spring 框架提供了对将缓存透明添加到现有 Spring 应用程序的支持。与 [transaction](#) 支持类似，缓存抽象允许一致使用各种缓存解决方案，而对代码的影响最小。

从 Spring 4.1 开始，通过支持 [JSR-107 annotations](#) 和更多自定义选项，显着改善了缓存抽象。

## 8.1. 了解缓存抽象

### 缓存与缓冲区

术语“缓冲区”和“缓存”倾向于互换使用。但是请注意，它们代表不同的事物。传统上，缓冲区用作快速实体和慢速实体之间的数据的中间临时存储。由于一方必须 `await` 另一方(这会影响性能)，因此缓冲区允许一次移动整个数据块，而不是一小段地移动数据块，从而缓解了这种情况。数据只能从缓冲区写入和读取一次。此外，缓冲区对于至少一个知道缓冲区的一方是可见的。

另一方面，根据定义，缓存是隐藏的，任何一方都不知道会发生缓存。它还可以提高性能，但是可以通过快速读取多次相同数据来实现。

您可以找到有关缓冲区和缓存[here](#)的区别进一步说明。

缓存抽象的核心是将缓存应用于 Java 方法，从而根据缓存中可用的信息减少执行次数。也就是说，每次调用目标方法时，抽象都会应用一种缓存行为，该行为检查给定参数是否已经执行了该方法。如果已执行，则返回缓存的结果，而不必执行实际的方法。如果尚未执行该方法，则将执行该方法，并将结果缓存并返回给用户，以便下次调用该方法时，将返回缓存的结果。这样，对于给定的一组参数，昂贵的方法(无论是受 CPU 限制还是与 IO 绑定)只能执行一次，结果可以重复使用，而不必再次实际执行该方法。缓存逻辑是透明地应用的，不会对调用方造成任何干扰。

### Tip

该方法仅适用于保证无论给定 Importing(或参数)执行多少次都返回相同输出(结果)的方法。

缓存抽象提供了其他与缓存相关的操作，例如更新缓存内容或删除一个或所有条目的能力。如果高速缓存处理在应用程序过程中可能更改的数据，则这些功能很有用。

与 Spring Framework 中的其他服务一样，缓存服务是一种抽象(不是缓存实现)，并且需要使用实际的存储来存储缓存数据-也就是说，抽象使您不必编写缓存逻辑，但是没有提供实际的数据存储。

`org.springframework.cache.Cache` 和 `org.springframework.cache.CacheManager` 接口实现了这种抽象。

Spring 提供了这种抽象的一些实现：基于 JDK `java.util.concurrent.ConcurrentMap` 的缓存

, [Ehcache 2.x](#), Gemfire 缓存, [Caffeine](#)和符合 JSR-107 的缓存(例如 Ehcache 3.x)。有关插入其他缓存存储区和提供程序的更多信息, 请参见[插入不同的后端缓存](#)。

### Tip

对于多线程和多进程环境, 缓存抽象没有特殊处理, 因为此类功能由缓存实现处理。。

如果您具有多进程环境(即, 一个应用程序部署在多个节点上), 则需要相应地配置缓存提供程序。根据您的用例, 在几个节点上复制相同数据就足够了。但是, 如果在应用程序过程中更改数据, 则可能需要启用其他传播机制。

高速缓存特定项直接等同于通过程序化高速缓存交互找到的典型“如果找不到, 然后 `continue` 处理并放入”代码块。没有应用锁, 几个线程可能会尝试同时加载同一项目。驱逐同样如此。如果多个线程试图同时更新或逐出数据, 则可以使用陈旧数据。某些缓存提供程序在该区域提供高级功能。有关更多详细信息, 请参见缓存提供程序的文档。

要使用缓存抽象, 您需要注意两个方面:

- 缓存声明: 确定需要缓存的方法及其策略。
- 缓存配置: 数据存储和读取的后备缓存。

## 8.2. 基于声明式 Comments 的缓存

对于缓存声明, Spring 的缓存抽象提供了一组 JavaComments:

- `@Cacheable` : 触发缓存填充。
- `@CacheEvict` : 触发缓存逐出。
- `@CachePut` : 在不影响方法执行的情况下更新缓存。
- `@Caching` : 重新组合要在方法上应用的多个缓存操作。
- `@CacheConfig` : 在类级别共享一些与缓存有关的常见设置。

## 8.2.1. @Cacheable 注解

顾名思义，您可以使用 `@Cacheable` 来划分可缓存的方法-即将结果存储在缓存中的方法，以便在后续调用(具有相同参数)时返回缓存中的值，而不会必须实际执行该方法。Comments 声明以最简单的形式要求与带 Comments 的方法关联的缓存名称，如以下示例所示：

```
@Cacheable("books")
public Book findBook(ISBN isbn) { ... }
```

在前面的代码段中，`findBook` 方法与名为 `books` 的缓存关联。每次调用该方法时，都会检查缓存以查看调用是否已执行并且不必重复。虽然在大多数情况下，仅声明一个缓存，但是 Comments 允许指定多个名称，以便使用多个缓存。在这种情况下，在执行方法之前检查每个缓存-如果命中了至少一个缓存，则返回关联的值。

### Note

即使未实际执行缓存的方法，所有其他不包含该值的缓存也会被更新。

以下示例在 `findBook` 方法上使用 `@Cacheable`：

```
@Cacheable({"books", "isbns"})
public Book findBook(ISBN isbn) { ... }
```

### 默认密钥生成

由于缓存本质上是键值存储，因此每次调用缓存的方法都需要转换为适合缓存访问的键。缓存抽象使用基于以下算法的简单 `KeyGenerator`：

- 如果没有给出参数，则返回 `SimpleKey.EMPTY`。
- 如果仅给出一个参数，则返回该实例。
- 如果给出了一个以上的参数，则返回一个包含所有参数的 `SimpleKey`。

只要参数具有自然键并实现有效的 `hashCode()` 和 `equals()` 方法，该方法就适用于大多数用例

。如果不是这种情况，则需要更改策略。

要提供其他默认密钥生成器，您需要实现

`org.springframework.cache.interceptor.KeyGenerator` 接口。

### iNote

随着 Spring 4.0 的发布，默认的密钥生成策略发生了变化。Spring 的早期版本使用密钥生成策略，该策略对于多个关键参数仅考虑参数 `hashCode()` 而不考虑 `equals()`。这可能会导致意外的按键冲突(有关背景，请参见[SPR-10237](#))。对于这种情况，新的 `SimpleKeyGenerator` 使用复合键。

如果要 `continue` 使用以前的关键策略，则可以配置已弃用的

`org.springframework.cache.interceptor.DefaultKeyGenerator` 类或创建基于哈希的自定义 `KeyGenerator` 实现。

## 自定义密钥生成声明

由于缓存是通用的，因此目标方法很可能具有各种签名，这些签名无法轻易 Map 到缓存结构的顶部。当目标方法具有多个参数时，只有其中一些参数适合缓存(而其余参数仅由方法逻辑使用)，这往往会变得很明显。考虑以下示例：

```
@Cacheable("books")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)
```

乍一看，虽然两个 `boolean` 参数会影响书的查找方式，但它们对缓存没有用处。此外，如果两者中只有一个重要而另一个不重要怎么办？

在这种情况下，使用 `@Cacheable` 注解可以指定如何通过其 `key` 属性生成密钥。您可以使用[SpEL](#) 来选择感兴趣的参数(或其嵌套属性)，执行操作甚至调用任意方法，而无需编写任何代码或实现任何接口。这是在[default generator](#) 上推荐的方法，因为随着代码库的增长，方法的签名趋向于完全不同。虽然默认策略可能适用于某些方法，但很少适用于所有方法。

以下示例是各种 SpEL 声明(如果您不熟悉 SpEL, 请帮忙并阅读[Spring 表达语言](#)):

```
@Cacheable(cacheNames="books", key="#isbn")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)

@Cacheable(cacheNames="books", key="#isbn.rawNumber")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)

@Cacheable(cacheNames="books", key="T(someType).hash(#isbn)")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)
```

前面的代码片段显示了选择某个参数, 其属性之一甚至是任意(静态)方法是多么容易。

如果负责生成密钥的算法过于具体或需要共享, 则可以在操作上定义一个自定义 `keyGenerator`。

为此, 请指定要使用的 `KeyGenerator` bean 实现的名称, 如以下示例所示:

```
@Cacheable(cacheNames="books", keyGenerator="myKeyGenerator")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)
```

### iNote

`key` 和 `keyGenerator` 参数是互斥的, 并且同时指定这两个参数的操作将导致异常。

## 默认缓存分辨率

缓存抽象使用简单的 `CacheResolver`, 该 `CacheResolver` 通过使用配置的 `CacheManager` 检索在操作级别定义的缓存。

要提供其他默认缓存解析器, 您需要实现

`org.springframework.cache.interceptor.CacheResolver` 接口。

## 自定义缓存解析

默认的缓存分辨率非常适合使用单个 `CacheManager` 并且没有复杂的缓存分辨率要求的应用程序。

对于使用多个缓存 Management 器的应用程序, 可以将 `cacheManager` 设置为用于每个操作, 如以下示例所示:

```
@Cacheable(cacheNames="books", cacheManager="anotherCacheManager") (1)
public Book findBook(ISBN isbn) {...}
```

- (1) 指定 `anotherCacheManager`。

您还可以完全以替换`key generation`的方式替换 `CacheResolver`。对于每个缓存操作，都要求解决该问题，让实现实际上可以根据运行时参数来解析要使用的缓存。以下示例显示如何指定

`CacheResolver`：

```
@Cacheable(cacheResolver="runtimeCacheResolver") (1)
public Book findBook(ISBN isbn) {...}
```

- (1) 指定 `CacheResolver`。

### iNote

从 Spring 4.1 开始，缓存 `Comments` 的 `value` 属性不再是必需的，因为 `CacheResolver` 可以提供此特定信息，而不管 `Comments` 的内容如何。

与 `key` 和 `keyGenerator` 类似，`cacheManager` 和 `cacheResolver` 参数是互斥的，并且同时指定这两个参数的操作将导致异常。因为 `CacheResolver` 实现会忽略自定义 `CacheManager`。这可能不是您所期望的。

## Synchronized Caching

在多线程环境中，可能会为同一参数同时调用某些操作(通常是在启动时)。默认情况下，缓存抽象不会锁定任何内容，并且可能会多次计算相同的值，从而破坏了缓存的目的。

对于那些特殊情况，您可以使用 `sync` 属性来指示基础缓存提供程序在计算值时锁定缓存条目。结果，只有一个线程正在忙于计算该值，而其他线程则被阻塞，直到在缓存中更新该条目为止。下面的示例演示如何使用 `sync` 属性：

```
@Cacheable(cacheNames="foos", sync=true) (1)
public Foo executeExpensiveOperation(String id) { ... }
```

- (1) 使用 `sync` 属性。

### 1 Note

这是一项可选功能，您最喜欢的缓存库可能不支持它。核心框架提供的所有 `CacheManager` 实现都支持它。有关更多详细信息，请参见缓存提供程序的文档。

## Conditional Caching

有时，一种方法可能并不总是适合缓存(例如，它可能取决于给定的参数)。高速缓存注解通过 `condition` 参数支持此类功能，该参数采用 `SpEL` 表达式，该表达式被评估为 `true` 或 `false`。如果为 `true`，则将缓存该方法。如果不是，它的行为就好像未缓存该方法一样(也就是说，无论缓存中使用什么值或使用了什么参数，每次都会执行该方法)。例如，仅当参数 `name` 的长度短于 32 时，才缓存以下方法：

```
@Cacheable(cacheNames="book", condition="#name.length() < 32") (1)
public Book findBook(String name)
```

- (1) 在 `@Cacheable` 上设置条件。

除了 `condition` 参数之外，您还可以使用 `unless` 参数来否决将值添加到缓存中。与 `condition` 不同，`unless` 表达式是在调用方法之后求值的。为了扩展前面的示例，也许我们只想缓存平装书，如以下示例所示：

```
@Cacheable(cacheNames="book", condition="#name.length() < 32", unless="#result.hardback")
public Book findBook(String name)
```

- (1) 使用 `unless` 属性来阻止精装书。

缓存抽象支持 `java.util.Optional`，仅当其存在时才将其内容用作缓存值。`#result` 始终引

用业务实体，而不引用受支持的包装器，因此可以将以下示例重写为：

```
@Cacheable(cacheNames="book", condition="#name.length() < 32", unless="#result?.hardbac  
public Optional<Book> findBook(String name)
```

请注意，`result` 仍指 `Book` 而不是 `Optional`。因为可能是 `null`，所以我们应该使用安全的导航操作符。

## 可用的缓存 SpEL 评估上下文

每个 `SpEL` 表达式针对专用的 [context](#) 求值。除了内置参数外，该框架还提供了与缓存相关的专用元数据，例如参数名称。下表描述了可用于上下文的项目，以便您可以将其用于键和条件计算：

表 11. Cache SpEL 可用元数据

Name	Location	Description	Example
<code>methodName</code>	Root object	被调用方法的名称	<code>#root.methodName</code>
<code>method</code>	Root object	被调用的方法	<code>#root.method.name</code>
<code>target</code>	Root object	被调用的目标对象	<code>#root.target</code>
<code>targetClass</code>	Root object	被调用目标的类	<code>#root.targetClass</code>
<code>args</code>	Root object	用于调用目标的参数 (作为数组)	<code>#root.args[0]</code>

Name	Location	Description	Example
	Root object	执行当前方法所依据的缓存集合	#root.caches[0].name
Argument name	Evaluation context	<p>任何方法参数的名称。如果名称不可用(可能是由于没有调试信息), 则参数名称在 <code>#a&lt;#arg&gt;</code> 下也可用, 其中 <code>#arg</code> 代表参数索引(从 0 开始)。</p>	<p>#iban 或 #a0 (您也可以使用 #p0 或 #p&lt;#arg&gt; 表示法作为别名)。</p>
result	Evaluation context	<p>方法调用的结果(要缓存的值)。仅在 <code>unless</code> 个表达式, <code>cache put</code> 个表达式(用于计算 <code>key</code>)或 <code>cache evict</code> 个表达式(当 <code>beforeInvocation</code> 为 <code>false</code> 时)中可用。对于受支持的包装器(例如 <code>Optional</code>) , <code>#result</code> 指的是实际对象, 而不是包</p>	#result

Name	Location	Description	Example
		装器。	

## 8.2.2. @CachePut 注解

当需要在不影响方法执行的情况下更新缓存时，可以使用 `@CachePut` 注解。也就是说，该方法始终执行，并将其结果放入缓存中(根据 `@CachePut` 选项)。它支持与 `@Cacheable` 相同的选项，应用于缓存填充，而不是方法流优化。以下示例使用 `@CachePut` 注解：

```
@CachePut(cacheNames="book", key="#isbn")
public Book updateBook(ISBN isbn, BookDescriptor descriptor)
```

### Tip

强烈建议不要在同一方法上使用 `@CachePut` 和 `@Cacheable` Comments，因为它们具有不同的行为。后者导致通过使用缓存跳过方法执行，而前者则强制执行以便执行缓存更新。这会导致意外的行为，并且，除了特定的极端情况(例如具有相互排斥条件的 Comments)外，应避免此类声明。还请注意，此类条件不应依赖于结果对象(即 `#result` 变量)，因为这些条件已预先验证以确认排除。

## 8.2.3. @CacheEvict 注解

缓存抽象不仅允许缓存存储的填充，还允许逐出。此过程对于从缓存中删除陈旧或未使用的数据很有用。与 `@Cacheable` 相反，`@CacheEvict` 划分了执行高速缓存逐出的方法(即，用作触发从高速缓存中删除数据的触发器的方法)。与其同级类似，`@CacheEvict` 要求指定一个或多个受操作影响的缓存，允许指定自定义缓存和键解析或条件，并具有一个额外的参数(`allEntries`)，该参数指示是否需要在整个缓存范围内逐出而不是只是逐项驱逐(基于密钥)。下面的示例从 `books` 缓存中逐出所有条目：

```
@CacheEvict(cacheNames="books", allEntries=true) (1)
public void loadBooks(InputStream batch)
```

- (1) 使用 `allEntries` 属性从缓存中逐出所有条目。

当需要清除整个缓存区域时，此选项非常有用。而不是逐出每个条目(由于效率低下，这将花费很长时间)，因此一次操作会删除所有条目，如前面的示例所示。请注意，该框架会忽略此方案中指定的任何键，因为它不适用(整个高速缓存被驱逐，而不仅仅是一个条目)。

您还可以使用 `beforeInvocation` 属性指示驱逐应该发生在(默认之后)还是方法执行之前。前者提供与其余注解相同的语义：方法成功完成后，将对缓存执行操作(在这种情况下为逐出)。如果该方法未执行(可能已被缓存)或引发了异常，则不会发生逐出。后者(`beforeInvocation=true`)导致逐出总是在调用该方法之前发生。在不需要将逐出与方法结果联系在一起的情况下，这很有用。

请注意，`void` 方法可与 `@CacheEvict` 一起使用-由于这些方法充当触发器，因此将忽略返回值(因为它们不会与缓存交互)。`@Cacheable` 并非如此，`@Cacheable` 向缓存中添加或更新数据，因此需要结果。

#### 8.2.4. @Caching 注解

有时，需要指定多个相同类型的 Comments(例如 `@CacheEvict` 或 `@CachePut`)，例如，因为不同缓存之间的条件或键表达式不同。`@Caching` 允许在同一方法上使用多个嵌套的 `@Cacheable`，`@CachePut` 和 `@CacheEvict` 注解。以下示例使用两个 `@CacheEvict` Comments：

```
@Caching(evict = { @CacheEvict("primary"), @CacheEvict(cacheNames="secondary", key="#p0
public Book importBooks(String deposit, Date date)
```

#### 8.2.5. @CacheConfig 注解

到目前为止，我们已经看到缓存操作提供了许多自定义选项，并且您可以为每个操作设置这些选项。但是，如果某些自定义选项适用于该类的所有操作，则配置它们可能很繁琐。例如，指定用于类的每个高速缓存操作的高速缓存的名称可以由单个类级定义代替。这是 `@CacheConfig` 发挥作用的

地方。以下示例使用 `@CacheConfig` 设置缓存的名称：

```
@CacheConfig("books") (1)
public class BookRepositoryImpl implements BookRepository {

    @Cacheable
    public Book findBook(ISBN isbn) { ... }
}
```

- (1) 使用 `@CacheConfig` 设置缓存名称。

`@CacheConfig` 是类级别的注解，它允许共享缓存名称，自定义 `KeyGenerator`，自定义 `CacheManager` 和自定义 `CacheResolver`。将此 `Comments` 放在类上不会打开任何缓存操作。

操作级别的自定义始终会覆盖在 `@CacheConfig` 上设置的自定义。因此，这为每个缓存操作提供了三个定制级别：

- 全局配置，可用于 `CacheManager`，`KeyGenerator`。
- 在 Class 上，使用 `@CacheConfig`。
- 在操作级别。

## 8.2.6. 启用缓存 Comments

重要的是要注意，即使声明缓存 `Comments` 并不会自动触发它们的动作-就像 Spring 中的许多事情一样，必须声明性地启用该功能(这意味着如果您怀疑应该归因于缓存，则可以通过删除来禁用它仅一个配置行，而不是代码中的所有 `Comments`)。

要启用缓存 `Comments`，请将 `Comments` `@EnableCaching` 添加到您的 `@Configuration` 类中：

```
@Configuration
@EnableCaching
public class AppConfig { }
```

另外，对于 XML 配置，您可以使用 `cache:annotation-driven` 元素：

```

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:cache="http://www.springframework.org/schema/cache"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans.xsd
        http://www.springframework.org/schema/cache http://www.springframework.org/schema/cache.xsd">
    <cache:annotation-driven/>
</beans>

```

`cache:annotation-driven` 元素和 `@EnableCaching` `Comments` 都允许您指定各种选项，这些选项影响通过 AOP 将缓存行为添加到应用程序的方式。该配置有意与 [@Transactional](#) 相似。

### iNote

处理缓存 `Comments` 的默认建议模式为 `proxy`，该模式仅允许通过代理拦截呼叫。同一类内的本地调用无法以这种方式被拦截。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。

### iNote

有关实现 `CachingConfigurer` 所需的高级自定义(使用 Java 配置)的更多详细信息，请参见 [javadoc](#)。

表 12. 缓存 `Comments` 设置

XML Attribute	Annotation Attribute	Default	Description
<code>cache-</code> <code>manager</code>	不适用(请参见 <a href="#">CachingConfigurer</a> javadoc)	<code>cacheManager</code>	要使用的缓 Manageme 称。使用此 Manageme 台初始化默

XML Attribute	Annotation Attribute	Default	Description
			<p>CacheResource</p> <p>果未设置，将使用 <code>cacheManager</code> 属性。</p> <p>cacheManager 属性提供了更精细地管理缓存的能力。如果未设置，将使用 <code>cacheManager</code> 属性。</p> <p>Management</p> <p>分辨率，请参考 <code>cache-resolver</code> 属性。</p>
<code>cache-resolver</code>			<p>用来解析后端缓存的 Bean 名称。</p> <p>bean 是必需的，且必须是实现 <code>CacheResolver</code> 接口的类。</p> <p>“cache-m</p> <p>性的替代方</p>
<code>key-generator</code>			<p>要使用的自定义键生成器的名称。</p>
<code>error-handler</code>			<p>要使用的自定义错误处理器。</p> <p>认情况下，所有操作期间有异常都将被捕获并由 Client 端处理。</p>

XML Attribute	Annotation Attribute	Default	Description
mode	mode	proxy	<p>缺省模式(<a href="#">F</a>)</p> <p>用 Spring 的处理要 Combean(遵循什么, 如前所述通过代理传用)。相反,</p> <p><a href="#">aspectj</a>)</p> <p>Spring 的 A 方面编织受, 修改目标应用于任何调用。 Aspect要在 Classp</p> <p><a href="#">spring-</a></p> <p><a href="#">aspects.</a>)</p> <p>加载时编织 编织)。(有加载时编织, 请参见<a href="#">Spring configuration</a>)</p>
proxy-target-class	proxyTargetClass	false	<p>仅适用于代制为使用</p> <p><a href="#">@Cacheable</a></p> <p><a href="#">@CacheEvict</a></p>

XML Attribute	Annotation Attribute	Default	Description
			<p>Comments 的类创建哪 存代理。如 <code>proxy-target-class</code> <code>true</code>， 则 类的代理。 <code>proxy-target-class</code> 是 省略了属性 基于标准 JD 理。 (有关不 型的详细检 <a href="#">Proxying M</a> 。 )</p>
<code>order</code>	<code>order</code>	Ordered.LOWEST_PRECEDENCE	<p>定义应用于 <code>@Cacheable</code> <code>@CacheEvict</code> Comments 的缓存建议 Sequences OrderAOP 规则的更多 <a href="#">Advice O</a> )没有指定的</p>

XML Attribute	Annotation Attribute	Default	Description
			着 AOP 子系统 议的 Sequence

### ①Note

`<cache:annotation-driven/>` 仅在定义它的相同应用程序上下文中的 Bean 上寻找

`@Cacheable/@CachePut/@CacheEvict/@Caching`。这意味着，如果将

`<cache:annotation-driven/>` 放在 `WebApplicationContext` 中，而将

`DispatcherServlet` 放在 `_5` 中，则它仅在控制器中检查 bean，而在服务中检查 bean

。有关更多信息，请参见[MVC 部分](#)。

### 方法可见性和缓存 Comments

使用代理时，应仅将缓存 Comments 应用于具有公共可见性的方法。如果使用这些 Comments 对受保护的、私有的或程序包可见的方法进行 Comments，则不会引发任何错误，但是带 Comments 的方法不会显示已配置的缓存设置。如果您需要 Comments 非公共方法，请考虑使用 AspectJ(请参阅本节的其余部分)，因为它会更改字节码本身。

### ②Tip

Spring 建议您仅使用 `@Cache*` Comments 对具体类(以及具体类的方法)进行 Comments

，而不是对接口进行 Comments。您当然可以在接口(或接口方法)上放置 `@Cache*` 注解

，但这仅在您使用基于接口的代理时才可以预期地起作用。Java 注解不是从接口继承的事实

意味着，如果您使用基于类的代理(`proxy-target-class="true"`)或基于编织的方面(

`mode="aspectj"`)，则代理和编织基础结构无法识别缓存设置，并且该对象是没有包装在

缓存代理中，那肯定是不好的。

## iNote

在代理模式(默认)下，仅拦截通过代理传入的外部方法调用。这意味着即使调用的方法标记为 `@Cacheable`，自调用(实际上是目标对象内的方法调用目标对象的另一种方法)也不会导致实际的缓存。在这种情况下，请考虑使用 `aspectj` 模式。另外，必须完全初始化代理以提供预期的行为，因此您不应在初始化代码(即 `@PostConstruct`)中依赖此功能。

### 8.2.7. 使用自定义 Comments

自定义 Comments 和 AspectJ

该功能仅适用于基于代理的方法，但可以通过使用 AspectJ 进行一些额外的工作来启用。

`spring-aspects` 模块仅为标准 Comments 定义一个方面。如果定义了自己的 Comments，则还需要为其定义一个方面。查看 `AnnotationCacheAspect` 为例。

缓存抽象使您可以使用自己的 Comments 来标识哪种方法触发缓存填充或逐出。作为模板机制，这非常方便，因为它消除了重复缓存注解声明的需求，如果指定了键或条件或代码库中不允许外部导入(`org.springframework`)，则这特别有用。与其它 [stereotype](#) 注解类似，可以将

`@Cacheable`，`@CachePut`，`@CacheEvict` 和 `@CacheConfig` 用作 [meta-annotations](#)(即可以 Comments 其他注解的注解)。在以下示例中，我们用自己的自定义 Comments 替换了通用的 `@Cacheable` 声明：

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
@Cacheable(cacheNames="books", key="#isbn")
public @interface SlowService { }
```

在前面的示例中，我们定义了自己的 `SlowService` Comments，该 Comments 本身被 `@Cacheable` Comments。现在我们可以替换以下代码：

```
@Cacheable(cacheNames="books", key="#isbn")
```

```
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)
```

以下示例显示了自定义注解，我们可以用其替换前面的代码：

```
@SlowService  
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)
```

即使 `@SlowService` 不是 SpringComments，容器也会在运行时自动获取其声明并理解其含义。

请注意，如[earlier](#)所述，需要启用 Comments 驱动的行为。

## 8.3. JCache(JSR-107)注解

从 4.1 版开始，缓存抽象完全支持 JCache 标准 Comments：`@CacheResult`，`@CachePut`，

`@CacheRemove` 和 `@CacheRemoveAll` 以及 `@CacheDefaults`，`@CacheKey` 和 `@CacheValue`

随播。您可以使用这些 Comments，而无需将缓存存储迁移到 JSR-107。内部实现使用 Spring 的缓存抽象，并提供符合规范的默认 `CacheResolver` 和 `KeyGenerator` 实现。换句话说，如果您已经在使用 Spring 的缓存抽象，则可以切换到这些标准 Comments，而无需更改缓存存储(或配置)。

### 8.3.1. 功能摘要

对于那些熟悉 Spring 缓存 Comments 的人，下表描述了 SpringComments 与 JSR-107 副本之间的主要区别：

表 13. Spring 和 JSR-107 缓存 Comments

Spring	JSR-107	Remark
<code>@Cacheable</code>	<code>@CacheResult</code>	相当相似。  <code>@CacheResult</code> 可以缓存特定的异常并强制执行该方法，而

Spring	JSR-107	Remark
		不管缓存的内容如何。 。
<code>@CachePut</code>	<code>@CachePut</code>	当 Spring 使用方法调用的结果更新缓存时, JCache 要求将其作为参数传递给 <code>@CacheValue</code> 。由于存在这种差异, JCache 允许在实际方法调用之前或之后更新缓存。
<code>@CacheEvict</code>	<code>@CacheRemove</code>	相当相似。当方法调用导致异常时, <code>@CacheRemove</code> 支持条件驱逐。
<code>@CacheEvict(allEntries=true)</code>	<code>@CacheRemoveAll</code>	参见 <code>@CacheRemove</code> 。
<code>@CacheConfig</code>	<code>@CacheDefaults</code>	让您以类似的方式配置相同的概念。

JCache 具有 `javax.cache.annotation.CacheResolver` 的概念, 该概念与 Spring 的 `CacheResolver` 接口相同, 只是 JCache 仅支持单个缓存。默认情况下, 一个简单的实现根据

`Comments` 中声明的名称检索要使用的缓存。应该注意的是，如果 `Comments` 中未指定缓存名称，`则会自动生成一个默认值。有关更多信息，请参见 @CacheResult#cacheName\(\) 的 javadoc。`

`CacheResolver` 实例由 `CacheResolverFactory` 检索。可以为每个缓存操作自定义工厂，如以下示例所示：

```
@CacheResult(cacheNames="books", cacheResolverFactory=MyCacheResolverFactory.class) (1)
public Book findBook(ISBN isbn)
```

- (1) 为此操作定制工厂。

### ①Note

对于所有引用的类，Spring 尝试查找具有给定类型的 bean。如果存在多个匹配项，那么将创建一个新实例，并可以使用常规 bean 生命周期回调，例如依赖项注入。

密钥是由 `javax.cache.annotation.CacheKeyGenerator` 生成的，其目的与 Spring 的 `KeyGenerator` 相同。默认情况下，将考虑所有方法参数，除非至少一个参数用 `@CacheKey`。这类似于 Spring 的[自定义密钥生成声明](#)。例如，以下是相同的操作，一个使用 Spring 的抽象，另一个使用 JCache：

```
@Cacheable(cacheNames="books", key="#isbn")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)

@CacheResult(cacheName="books")
public Book findBook(@CacheKey ISBN isbn, boolean checkWarehouse, boolean includeUsed)
```

您还可以在操作上指定 `CacheKeyResolver`，类似于指定 `CacheResolverFactory` 的方式。

JCache 可以 Management 带 `Comments` 的方法引发的异常。这样可以防止更新缓存，但也可以将异常缓存为失败的指示，而不必再次调用该方法。假设如果 ISBN 的结构无效，则抛出 `InvalidISBNNotFoundException`。这是一个永久性的失败(使用这样的参数无法检索任何书籍)。以下内容缓存了该异常，以便使用相同的无效 ISBN 进行的进一步调用直接引发该缓存的异常，而不是再次调用该方法：

```
@CacheResult(cacheName="books", exceptionCacheName="failures"
    cachedExceptions = InvalidIsbnNotFoundException.class)
public Book findBook(ISBN isbn)
```

### 8.3.2. 启用 JSR-107 支持

除了启用 Spring 的声明性 Comments 支持外，您无需执行任何其他操作即可启用 JSR-107 支持。

如果 Classpath 中同时存在 JSR-107 API 和 `spring-context-support` 模块，则

`@EnableCaching` 和 `cache:annotation-driven` 元素都会自动启用 JCache 支持。

#### iNote

根据您的用例，选择基本上是您的选择。您甚至可以通过在某些服务器上使用 JSR-107 API 并在其他服务器上使用 Spring 自己的 Comments 来混合和匹配服务。但是，如果这些服务影响相同的缓存，则应使用一致且相同的密钥生成实现。

## 8.4. 基于声明式 XML 的缓存

如果不能使用 Comments(可能是由于无法访问源代码或没有外部代码)，则可以使用 XML 进行声明式缓存。因此，您可以在外部指定目标方法和缓存指令，而不是 Comments 用于缓存的方法(类似于声明式事务 Management[advice](#))。上一节中的示例可以转换为以下示例：

```
<!-- the service we want to make cacheable -->
<bean id="bookService" class="x.y.service.DefaultBookService"/>

<!-- cache definitions -->
<cache:advice id="cacheAdvice" cache-manager="cacheManager">
    <cache:caching cache="books">
        <cache:cacheable method="findBook" key="#isbn"/>
        <cache:cache-evict method="loadBooks" all-entries="true"/>
    </cache:caching>
</cache:advice>

<!-- apply the cacheable behavior to all BookService interfaces -->
<aop:config>
    <aop:advisor advice-ref="cacheAdvice" pointcut="execution(* x.y.BookService.*(..))" />
</aop:config>

<!-- cache manager definition omitted -->
```

在前面的配置中，`bookService` 已设置为可缓存。要应用的缓存语义封装在 `cache:advice` 定义中，这将导致 `findBooks` 方法用于将数据放入缓存，而 `loadBooks` 方法用于将数据逐出。两种定义都针对 `books` 缓存。

`aop:config` 定义通过使用 AspectJ 切入点表达式将缓存建议应用于程序中的适当点(更多信息可在[Spring 面向方面的编程](#)中获得)。在前面的示例中，考虑了 `BookService` 中的所有方法，并将缓存建议应用于它们。

声明式 XML 缓存支持所有基于 Comments 的模型，因此在两者之间移动应该相当容易。此外，两者都可以在同一应用程序内使用。基于 XML 的方法不会涉及目标代码。但是，它本质上比较冗长。当处理具有用于缓存的重载方法的类时，确定合适的方法确实需要付出额外的努力，因为 `method` 参数不是很好的判别器。在这些情况下，您可以使用 AspectJ 切入点来挑选目标方法并应用适当的缓存功能。但是，通过 XML，更容易应用程序包或组或接口范围的缓存(同样，由于 AspectJ 切入点的缘故)和创建类似模板的定义(就像我们在前面的示例中一样，通过 `cache:definitions` 定义了目标缓存) `cache` 属性)。

## 8.5. 配置缓存存储

缓存抽象提供了几种存储集成选项。要使用它们，您需要声明一个适当的 `CacheManager` (控制和 Management `Cache` 实例的实体，该实体可用于检索这些实例以进行存储)。

### 8.5.1. 基于 JDK ConcurrentHashMap 的缓存

基于 JDK 的 `Cache` 实现位于 `org.springframework.cache.concurrent` 包下。它使您可以将 `ConcurrentHashMap` 用作后备 `Cache` 存储。以下示例显示了如何配置两个缓存：

```
<!-- simple cache manager -->
<bean id="cacheManager" class="org.springframework.cache.support.SimpleCacheManager">
    <property name="caches">
        <set>
            <bean class="org.springframework.cache.concurrent.ConcurrentMapCacheFactory"
            <bean class="org.springframework.cache.concurrent.ConcurrentMapCacheFactory"
```

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

前面的代码段使用 `SimpleCacheManager` 为名为 `default` 和 `books` 的两个嵌套

`ConcurrentMapCache` 实例创建 `CacheManager`。请注意，名称是直接为每个缓存配置的。

由于缓存是由应用程序创建的，因此绑定到其生命周期，使其适合于基本用例，测试或简单的应用程序。缓存可以很好地扩展并且非常快，但是它不提供任何 Management，持久性功能或驱逐 Contract。

### 8.5.2. 基于 Ehcache 的缓存

#### ①Note

Ehcache 3.x 完全符合 JSR-107，并且不需要专用支持。

Ehcache 2.x 实现位于 `org.springframework.cache.ehcache` 包中。同样，要使用它，您需要声明适当的 `CacheManager`。以下示例显示了如何执行此操作：

```
<bean id="cacheManager"
      class="org.springframework.cache.ehcache.EhCacheCacheManager" p:cache-manager-r
      <!-- EhCache library setup -->
<bean id="ehcache"
      class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean" p:config-lo
```

此设置在 Spring IoC 内部引导 ehcache 库(通过 `ehcache` bean)，然后将其连接到专用的 `CacheManager` 实现中。请注意，完整的 ehcache 特定配置是从 `ehcache.xml` 读取的。

### 8.5.3. Caffeine 缓存

Caffeine 是对 Guava 缓存的 Java 8 重写，其实现位于 `org.springframework.cache.caffeine` 包中，并提供对 Caffeine 多个功能的访问。

下面的示例配置一个 `CacheManager` 来按需创建缓存：

```
<bean id="cacheManager"
      class="org.springframework.cache.caffeine.CaffeineCacheManager"/>
```

您还可以提供要显式使用的缓存。在这种情况下，`Manager` 只能提供那些。以下示例显示了如何执行此操作：

```
<bean id="cacheManager" class="org.springframework.cache.caffeine.CaffeineCacheManager"
  <property name="caches">
    <set>
      <value>default</value>
      <value>books</value>
    </set>
  </property>
</bean>
```

Caffeine `CacheManager` 还支持自定义 `Caffeine` 和 `CacheLoader`。有关这些的更多信息，请参见 [Caffeine documentation](#)。

#### 8.5.4. 基于 GemFire 的缓存

GemFire 是面向内存，磁盘支持，弹性可伸缩，连续可用，活动(具有内置的基于模式的订阅通知)，全局复制的数据库，并提供功能齐全的边缘缓存。有关如何将 GemFire 用作 `CacheManager` (以及更多)的更多信息，请参见 [Spring Data GemFire 参考文档](#)。

#### 8.5.5. JSR-107 缓存

Spring 的缓存抽象也可以使用符合 JSR-107 的缓存。JCache 实现位于

`org.springframework.cache.jcache` 包中。

同样，要使用它，您需要声明适当的 `CacheManager`。以下示例显示了如何执行此操作：

```
<bean id="cacheManager"
      class="org.springframework.cache.jcache.JCacheCacheManager"
      p:cache-manager-ref="jCacheManager"/>

<!-- JSR-107 cache manager setup -->
<bean id="jCacheManager" .../>
```

## 8.5.6. 在没有后备存储的情况下处理缓存

有时，在切换环境或进行测试时，您可能具有缓存声明而未配置实际的后备缓存。由于这是无效的配置，因此在运行时会引发异常，因为缓存基础结构无法找到合适的存储。在这种情况下，可以删除简单的伪高速缓存，而不执行高速缓存，而不是删除高速缓存声明(这可能很乏味)，即不强制执行高速缓存的方法，即每次都执行高速缓存的方法。以下示例显示了如何执行此操作：

```
<bean id="cacheManager" class="org.springframework.cache.support.CompositeCacheManager">
    <property name="cacheManagers">
        <list>
            <ref bean="jdkCache" />
            <ref bean="gemfireCache" />
        </list>
    </property>
    <property name="fallbackToNoOpCache" value="true" />
</bean>
```

前面的 `CompositeCacheManager` 链接了多个 `CacheManager` 实例，并通过 `fallbackToNoOpCache` 标志为未配置的缓存 Management 器处理的所有定义添加了一个无操作缓存。也就是说，在 `jdkCache` 或 `gemfireCache` 中未找到的每个缓存定义(在示例中较早配置)均由不存储任何信息的无操作缓存处理，导致每次执行目标方法。

## 8.6. 插入不同的后端缓存

显然，有很多缓存产品可以用作后备存储。要插入它们，您需要提供 `CacheManager` 和 `Cache` 的实现，因为不幸的是，没有可用的替代标准。这听起来可能比实际要难，因为在实践中，这些类往往是简单的[adapters](#)，就像 `ehcache` 类一样，它们将缓存抽象框架 Map 到存储 API 的顶部。大多数 `CacheManager` 类可以使用 `org.springframework.cache.support` 包中的类(例如 `AbstractCacheManager`)，它负责样板代码，仅保留实际的 Map)。我们希望，及时提供与 Spring 集成的库可以弥补这一小的配置空白。

## 8.7. 如何设置 TTL/TTI /驱逐策略/ XXX 功能？

直接通过您的缓存提供程序。缓存抽象是一种抽象，而不是缓存实现。您使用的解决方案可能支持

其他解决方案不支持的各种数据策略和不同的拓扑(例如, JDK `ConcurrentHashMap` —在缓存抽象中暴露出来将是无用的, 因为没有后备支持)。此类功能应通过后备缓存(配置时)或通过其本机 API 直接控制。

## 9. Appendix

### 9.1. XML 模式

附录的此部分列出了与集成技术有关的 XML 模式。

#### 9.1.1. jee 模式

`jee` 元素处理与 Java EE(Java 企业版)配置有关的问题, 例如查找 JNDI 对象和定义 EJB 引用。

要使用 `jee` 模式中的元素, 您需要在 Spring XML 配置文件的顶部具有以下序言。以下代码段中的文本引用了正确的架构, 以便您可以使用 `jee` 名称空间中的元素:

```
<?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:jee="http://www.springframework.org/schema/jee" xsi:schemaLocation="
           http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans.xsd
           http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee.xsd">

    <!-- bean definitions here -->
</beans>
```

#### <jee:jndi-lookup/> (simple)

下面的示例显示如何使用 JNDI 查找没有 `jee` 模式的数据源:

```
<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MyDataSource"/>
</bean>
<bean id="userDao" class="com.foo.JdbcUserDao">
    <!-- Spring will do the cast automatically (as usual) -->
    <property name="dataSource" ref="dataSource"/>
</bean>
```

下面的示例演示如何使用 JNDI 来通过 `jee` 模式查找数据源：

```
<jee:jndi-lookup id="dataSource" jndi-name="jdbc/MyDataSource"/>

<bean id="userDao" class="com.foo.JdbcUserDao">
    <!-- Spring will do the cast automatically (as usual) -->
    <property name="dataSource" ref="dataSource"/>
</bean>
```

### <jee:jndi-lookup/>(具有单个 JNDI 环境设置)

下面的示例演示如何使用 JNDI 查找不带 `jee` 的环境变量：

```
<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MyDataSource"/>
    <property name="jndiEnvironment">
        <props>
            <prop key="ping">pong</prop>
        </props>
    </property>
</bean>
```

下面的示例演示如何使用 JNDI 通过 `jee` 查找环境变量：

```
<jee:jndi-lookup id="simple" jndi-name="jdbc/MyDataSource">
    <jee:environment>ping=pong</jee:environment>
</jee:jndi-lookup>
```

### <jee:jndi-lookup/>(具有多个 JNDI 环境设置)

下面的示例演示如何使用 JNDI 查找不带 `jee` 的多个环境变量：

```
<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MyDataSource"/>
    <property name="jndiEnvironment">
        <props>
            <prop key="sing">song</prop>
            <prop key="ping">pong</prop>
        </props>
    </property>
</bean>
```

以下示例显示如何使用 JNDI 通过 `jee` 查找多个环境变量：

```
<jee:jndi-lookup id="simple" jndi-name="jdbc/MyDataSource">
    <!-- newline-separated, key-value pairs for the environment (standard Properties for JNDI lookups) -->
    <jee:environment>
        sing=song
        ping=pong
    </jee:environment>
</jee:jndi-lookup>
```

## <jee:jndi-lookup/> (Complex)

下面的示例演示如何使用 JNDI 查找数据源以及许多没有 **jee** 的不同属性：

```
<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MyDataSource"/>
    <property name="cache" value="true"/>
    <property name="resourceRef" value="true"/>
    <property name="lookupOnStartup" value="false"/>
    <property name="expectedType" value="com.myapp.DefaultThing"/>
    <property name="proxyInterface" value="com.myapp.Thing"/>
</bean>
```

下面的示例演示如何使用 JNDI 使用 **jee** 查找数据源和许多不同的属性：

```
<jee:jndi-lookup id="simple"
    jndi-name="jdbc/MyDataSource"
    cache="true"
    resource-ref="true"
    lookup-on-startup="false"
    expected-type="com.myapp.DefaultThing"
    proxy-interface="com.myapp.Thing"/>
```

## <jee:local-slsb/> (Simple)

<**jee:local-slsb**> 元素配置对本地 EJB Stateless SessionBean 的引用。

以下示例显示如何配置对不带 **jee** 的本地 EJB Stateless SessionBean 的引用：

```
<bean id="simple"
    class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">
    <property name="jndiName" value="ejb/RentalServiceBean"/>
    <property name="businessInterface" value="com.foo.service.RentalService"/>
</bean>
```

以下示例显示如何使用 **jee** 配置对本地 EJB Stateless SessionBean 的引用：

```
<jee:local-slsb id="simpleSlsb" jndi-name="ejb/RentalServiceBean"  
business-interface="com.foo.service.RentalService"/>
```

## <jee:local-slsb/> (Complex)

<jee:local-slsb/> 元素配置对本地 EJB Stateless SessionBean 的引用。

以下示例说明如何配置对本地 EJB Stateless SessionBean 的引用以及许多不带有 jee 的属性：

```
<bean id="complexLocalEjb"  
      class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">  
<property name="jndiName" value="ejb/RentalServiceBean"/>  
<property name="businessInterface" value="com.example.service.RentalService"/>  
<property name="cacheHome" value="true"/>  
<property name="lookupHomeOnStartup" value="true"/>  
<property name="resourceRef" value="true"/>  
</bean>
```

以下示例显示如何使用 jee 配置对本地 EJB Stateless SessionBean 的引用以及许多属性：

```
<jee:local-slsb id="complexLocalEjb"  
      jndi-name="ejb/RentalServiceBean"  
      business-interface="com.foo.service.RentalService"  
      cache-home="true"  
      lookup-home-on-startup="true"  
      resource-ref="true">
```

## <jee:remote-slsb/>

<jee:remote-slsb/> 元素配置对 remote EJB Stateless SessionBean 的引用。

以下示例显示如何配置对不带 jee 的远程 EJB Stateless SessionBean 的引用：

```
<bean id="complexRemoteEjb"  
      class="org.springframework.ejb.access.SimpleRemoteStatelessSessionProxyFactoryBean">  
<property name="jndiName" value="ejb/MyRemoteBean"/>  
<property name="businessInterface" value="com.foo.service.RentalService"/>  
<property name="cacheHome" value="true"/>  
<property name="lookupHomeOnStartup" value="true"/>  
<property name="resourceRef" value="true"/>  
<property name="homeInterface" value="com.foo.service.RentalService"/>  
<property name="refreshHomeOnConnectFailure" value="true"/>  
</bean>
```

以下示例显示如何使用 `jee` 配置对远程 EJB Stateless SessionBean 的引用：

```
<jee:remote-slsb id="complexRemoteEjb"
    jndi-name="ejb/MyRemoteBean"
    business-interface="com.foo.service.RentalService"
    cache-home="true"
    lookup-home-on-startup="true"
    resource-ref="true"
    home-interface="com.foo.service.RentalService"
    refresh-home-on-connect-failure="true">
```

## 9.1.2. jms 模式

`jms` 元素用于配置与 JMS 相关的 bean，例如 Spring 的[消息侦听器容器](#)。这些元素在[JMS chapter](#) 标题为[JMS 命名空间支持](#)的部分中进行了详细说明。有关此支持和 `jms` 元素本身的完整详细信息，请参见该章。

为了完整起见，要使用 `jms` 模式中的元素，您需要在 Spring XML 配置文件的顶部具有以下序言。

以下代码段中的文本引用了正确的架构，以便您可以使用 `jms` 名称空间中的元素：

```
<?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:jms="http://www.springframework.org/schema/jms" xsi:schemaLocation="
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans.xsd
        http://www.springframework.org/schema/jms http://www.springframework.org/schema/jms.xsd">

    <!-- bean definitions here -->
</beans>
```

## 9.1.3. 使用\<>

[配置基于 Comments 的 MBean 导出](#) 中对此元素进行了详细说明。

## 9.1.4. 缓存架构

您可以使用 `cache` 元素启用对 Spring 的 `@CacheEvict`，`@CachePut` 和 `@Caching` `Comments` 的支持。它还支持基于声明式 XML 的缓存。有关详情，请参见[启用缓存 Comments](#)和[基于声明式 XML 的缓存](#)。

要使用 `cache` 模式中的元素，您需要在 Spring XML 配置文件的顶部具有以下序言。以下代码段中的文本引用了正确的架构，以便您可以使用 `cache` 名称空间中的元素：

```
<?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:cache="http://www.springframework.org/schema/cache" xsi:schemaLocation="
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans.xsd
        http://www.springframework.org/schema/cache http://www.springframework.org/schema/cache.xsd">

    <!-- bean definitions here -->
</beans>
```

# Language Support

## 1. Kotlin

[Kotlin](#) 是针对 JVM(和其他平台) 的静态类型语言，它允许编写简洁明了的代码，同时为使用 Java 编写的现有库提供很好的 [interoperability](#)。

Spring 框架为 Kotlin 提供了一流的支持，使开发人员几乎可以将 Spring 框架当作原生 Kotlin 框架来编写 Kotlin 应用程序。

了解 Spring 和 Kotlin 的最简单方法是遵循 [本综合教程](#)。如果需要支持，可以随时加入 [Kotlin Slack](#) 的 #springChannels，或在 [Stackoverflow](#) 上提问带有 `spring` 和 `kotlin` 作为标签的问题。

### 1.1. Requirements

Spring 框架支持 Kotlin 1.1，并要求 [kotlin-stdlib](#)(或其其中一个变体，例如 Kotlin 1.1 的 [kotlin-stdlib-jre8](#) 或 Kotlin 1.2 的 [kotlin-stdlib-jdk8](#))和 [kotlin-reflect](#) 出现在 Classpath 上。如果您在 [start.spring.io](#) 上引导 Kotlin 项目，则默认情况下会提供它们。

### 1.2. Extensions

Kotlin [extensions](#) 提供了使用其他功能扩展现有类的功能。Spring Framework Kotlin API 使用这

些扩展为现有的 Spring API 添加了新的 Kotlin 特定的便利。

[Spring Framework KDoc API](#) 列出并记录了所有可用的 Kotlin 扩展和 DSL。

### iNote

请记住，必须导入 Kotlin 扩展才能使用。例如，这意味着仅当导入

`org.springframework.context.support.registerBean` 时

`GenericApplicationContext.registerBean` `KotlinExtensions` 才可用。就是说，类似于静态导入，在大多数情况下，IDE 应该自动建议导入。

例如，[Kotlin 修饰类型参数](#)为 JVM [泛型类型擦除](#)提供了一种解决方法，而 Spring Framework 提供了一些扩展以利用此功能。这允许使用更好的 Kotlin API `RestTemplate`，Spring WebFlux 的新 `WebClient` 以及各种其他 API。

### iNote

其他库，例如 Reactor 和 Spring Data，也为其 API 提供了 Kotlin 扩展，因此总体上提供了更好的 Kotlin 开发经验。

要检索 Java 中的 `User` 对象的列表，通常需要编写以下内容：

```
Flux<User> users = client.get().retrieve().bodyToFlux(User.class)
```

使用 Kotlin 和 Spring Framework 扩展，您可以编写以下代码：

```
val users = client.get().retrieve().bodyToFlux<User>()
// or (both are equivalent)
val users : Flux<User> = client.get().retrieve().bodyToFlux()
```

与 Java 中一样，Kotlin 中的 `users` 是强类型的，但是 Kotlin 的聪明类型推断允许使用较短的语法

。

## 1.3. Null-safety

Kotlin 的主要功能之一是 [null-safety](#)，它在编译时干净地处理 `null` 值，而不是在运行时撞到著名的 `NullPointerException`。这通过可空性声明和表示“值或无值”的语义使应用程序更安全，而无需支付诸如 `Optional` 之类的包装器的费用。（Kotlin 允许使用具有可为空值的函数构造。请参阅此[Kotlin 空安全综合指南](#)。）

尽管 Java 不允许您在其类型系统中表示空安全性，但 Spring 框架通过 `org.springframework.lang` 包中声明的对工具友好的 `Comments` 提供了整个[Spring Framework API 的 null 安全性](#)。默认情况下，Kotlin 中使用的 Java API 中的类型被识别为 [platform types](#)，对此它们的空检查得到了放宽。[Kotlin 对 JSR-305 注解的支持](#)和 Spring 可空性 `Comments` 为 Kotlin 开发人员提供了整个 Spring Framework API 的空安全性，具有在编译时处理 `null` 相关问题的优势。

### iNote

诸如 Reactor 或 Spring Data 之类的库提供了空安全 API，以利用此功能。

您可以通过添加带有以下选项的 `-xjsr305` 编译器标志来配置 JSR-305 检查：

`xjsr305={strict|warn|ignore}`。

对于 kotlin 1.1 版，默认行为与 `-xjsr305=warn` 相同。必须使用 `strict` 值，才能从 Spring API 推断出的 Kotlin 类型中考虑到 Spring Framework API 的空安全性，但应在知道 Spring API 的空性声明即使在次要发行版之间可能会演变的情况下使用，并且应该在其中添加更多检查的前提下使用 `strict` 值。Future)。

### iNote

尚不支持泛型类型参数、`varargs` 和数组元素的可空性，但应在即将发布的版本中。有关最新

信息, 请参见[this discussion](#)。

## 1.4. 类和接口

Spring 框架支持各种 Kotlin 构造, 例如通过主构造函数实例化 Kotlin 类, 不可变的类数据绑定以及使用默认值函数可选参数。

Kotlin 参数名称是通过专用的 `KotlinReflectionParameterNameDiscoverer` 识别的, 该 `KotlinReflectionParameterNameDiscoverer` 允许查找接口方法参数名称, 而无需在编译过程中启用 Java 8 `-parameters` 编译器标志。

序列化或反序列化 JSON 数据所需的[Jackson-Kotlin](#) 模块在 Classpath 中找到时会自动注册, 如果在没有 Jackson 和 Kotlin 模块存在的情况下检测到 Jackson 和 Kotlin, 则会记录一条警告消息。

您可以将配置类声明为[顶层或嵌套但不内部](#), 因为后者需要引用外部类。

## 1.5. Annotations

Spring 框架还利用[Kotlin null-safety](#) 来确定是否需要 HTTP 参数, 而不必显式定义 `required` 属性。

这意味着 `@RequestParam name: String?` 被视为不是必需的, 相反, `@RequestParam name: String` 被视为是必需的。Spring Messaging `@Header` Comments 也支持此功能。

以类似的方式, 使用 `@Autowired`, `@Bean` 或 `@Inject` 注入 Spring bean 会使用此信息来确定是否需要 bean。

例如, `@Autowired lateinit var thing: Thing` 表示必须在应用程序上下文中注册类型 `Thing` 的 bean, 而如果 `@Autowired lateinit var thing: Thing?` 不存在, 则 `@Autowired lateinit var thing: Thing?` 不会引发错误。

按照相同的原理, `@Bean fun play(toy: Toy, car: Car?) = Baz(toy, Car)` 表示类型 `Toy`

的 bean 必须在应用程序上下文中注册，而类型 `Car` 的 bean 可能存在或可能不存在。相同的行为适用于自动装配的构造函数参数。

### iNote

如果对具有属性或主要构造函数参数的类使用 bean 验证，则可能需要使用 [Comments 使用场所目标](#)（例如 `@field:NotNull` 或 `@get:Size(min=5, max=15)`），如[此堆栈溢出响应](#)中所述。

## 1.6. Bean 定义 DSL

Spring Framework 5 通过使用 lambda 作为 XML 或 Java 配置（`@Configuration` 和 `@Bean`）的替代方法，引入了一种以功能性方式注册 Bean 的新方法。简而言之，它使您可以使用充当 `FactoryBean` 的 lambda 注册 bean。该机制非常有效，因为它不需要任何反射或 CGLIB 代理。

在 Java 中，您可以例如编写以下内容：

```
GenericApplicationContext context = new GenericApplicationContext();
context.registerBean(Foo.class);
context.registerBean(Bar.class, () -> new Bar(context.getBean(Foo.class)));
);
```

在 Kotlin 中，使用类型化参数和 `GenericApplicationContext` [Kotlin extensions](#)，您可以 Rewrite 以下内容：

```
val context = GenericApplicationContext().apply {
    registerBean<Foo>()
    registerBean { Bar(it.getBean<Foo>()) }
}
```

为了允许使用更具声明性的方法和更简洁的语法，Spring Framework 提供了 [Kotlin bean 定义 DSL](#)，它通过干净的声明性 API 声明了 `ApplicationContextInitializer`，该 API 使您可以处理概要文件和 `Environment` 来自定义如何注册 Bean。以下示例创建一个 `play` 配置文件：

```

fun beans() = beans {
    bean<UserHandler>()
    bean<Routes>()
    bean<WebHandler>("webHandler") {
        RouterFunctions.toWebHandler(
            ref<Routes>().router(),
            HandlerStrategies.builder().viewResolver(ref()).build()
        )
    }
    bean("messageSource") {
        ReloadableResourceBundleMessageSource().apply {
            setBasename("messages")
            setDefaultEncoding("UTF-8")
        }
    }
    bean {
        val prefix = "classpath:/templates/"
        val suffix = ".mustache"
        val loader = MustacheResourceTemplateLoader(prefix, suffix)
        MustacheViewResolver(Mustache.compiler().withLoader(loader)).apply {
            setPrefix(prefix)
            setSuffix(suffix)
        }
    }
    profile("play") {
        bean<Play>()
    }
}

```

在前面的示例中，`bean<Routes>()` 使用构造函数自动装配，而 `ref<Routes>()` 是 `applicationContext.getBean(Routes::class.java)` 的快捷方式。

然后，您可以使用此 `beans()` 函数在应用程序上下文中注册 bean，如以下示例所示：

```

val context = GenericApplicationContext().apply {
    beans().initialize(this)
    refresh()
}

```

### 1 Note

该 DSL 是编程的，这意味着它允许通过 `if` 表达式，`for` 循环或任何其他 Kotlin 构造对 bean 进行自定义注册逻辑。

有关具体示例，请参见[Spring kotlin 功能 bean 声明](#)。

## iNote

Spring Boot 基于 Java 配置和尚未为功能 Bean 定义提供特定支持，但是您可以通过 Spring Boot 的 `ApplicationContextInitializer` 支持实验性地使用功能性 bean 定义。有关更多详细信息和最新信息，请参见[这个堆栈溢出答案](#)。

## 1.7. Web

### 1.7.1. WebFlux 功能性 DSL

Spring Framework 现在带有[Kotlin 路由 DSL](#)，可让您使用[WebFlux 功能 API](#)编写干净且惯用的 Kotlin 代码，如以下示例所示：

```
router {
    accept(TEXT_HTML).nest {
        GET("/") { ok().render("index") }
        GET("/sse") { ok().render("sse") }
        GET("/users", userHandler::findAllView)
    }
    "/api".nest {
        accept(APPLICATION_JSON).nest {
            GET("/users", userHandler::findAll)
        }
        accept(TEXT_EVENT_STREAM).nest {
            GET("/users", userHandler::stream)
        }
    }
    resources("//**", ClassPathResource("static/"))
}
```

## iNote

该 DSL 是编程的，这意味着它允许通过 `if` 表达式、`for` 循环或任何其他 Kotlin 构造对 bean 进行自定义注册逻辑。当您需要根据动态数据(例如，来自数据库)注册路由时，这很有用。

有关具体示例，请参见[MiXiT 项目 Route](#)。

### 1.7.2. Kotlin 脚本模板

从 4.3 版本开始，Spring 框架提供了一个[ScriptTemplateView](#)来通过使用脚本引擎来呈现模板。它支持[JSR-223](#)。Spring Framework 5 进一步扩展了此功能到 WebFlux 并支持[i18n 和嵌套模板](#)。

Kotlin 提供了类似的支持，并允许渲染基于 Kotlin 的模板。有关详情，请参见[this commit](#)。

这启用了一些有趣的用例-例如通过使用[kotlinx.html](#) DSL 或使用带有插值的 Kotlin 多行 `String` 来编写类型安全的模板。

这可以让您在受支持的 IDE 中编写具有完全自动完成和重构支持的 Kotlin 模板，如以下示例所示：

```
import io.spring.demo.*

"""
${include("header")}
<h1>${i18n("title")}</h1>
<ul>
${users.joinToLine{ "<li>${i18n("user")} ${it.firstname} ${it.lastname}</li>" }}
</ul>
${include("footer")}
"""
```

有关更多详细信息，请参见[kotlin-script-templating](#)示例项目。

## 1.8. Kotlin 的 Spring 项目

本节提供了一些值得在 Kotlin 中开发 Spring 项目的特定提示和建议。

### 1.8.1. 默认为最终

默认情况下，[Kotlin 的所有类均为期末考试](#)。类上的 `open` 修饰符与 Java 的 `final` 相反：它允许其他人从此类继承。这也适用于成员函数，因为它们需要标记为 `open` 才能被覆盖。

尽管 Kotlin 的 JVM 友好设计通常与 Spring 毫无冲突，但如果考虑这一事实，此 Kotlin 特定功能可能会阻止应用程序启动。这是因为 Spring Bean(例如出于技术原因需要在运行时继承的 `@Configuration` 类)通常由 CGLIB 代理。解决方法是在由 CGLIB 代理的 Spring bean 的每个类和成员函数上添加 `open` 关键字(例如 `@Configuration` 类)，这会很快变得很痛苦，并且违反了保持代码简洁和可预测的 Kotlin 原则。

幸运的是，Kotlin 现在提供了一个[kotlin-spring](#)插件([kotlin-allopen](#) 插件的预配置版本)，该插件会自动打开使用以下注解之一进行注解或元注解的类型的类及其成员函数：

- `@Component`
- `@Async`
- `@Transactional`
- `@Cacheable`

支持元 Comments，这意味着将自动打开用 `@Configuration`，`@Controller`，`@RestController`，`@Service` 或 `@Repository` Comments 的类型，因为这些 Comments 用 `@Component` 进行了元 Comments。

[start.spring.io](#)默认启用它，因此，实际上，您可以编写 Java Bean，而无需使用其他 `open` 关键字，就像 Java 中一样。

## 1.8.2. 使用不可变的类实例进行持久化

在 Kotlin 中，在主构造函数中声明只读属性非常方便，并且被视为最佳实践，如以下示例所示：

```
class Person(val name: String, val age: Int)
```

您可以选择添加[数据关键字](#)，以使编译器自动从主要构造函数中声明的所有属性派生以下成员：

- `equals()` 和 `hashCode()`
- `toString()`，格式为 `"User(name=John, age=42)"`
- `componentN()` 按声明 Sequences 与属性相对应的函数
- `copy()` 功能

如下例所示，即使 `Person` 属性是只读的，也可以轻松更改各个属性：

```
data class Person(val name: String, val age: Int)

val jack = Person(name = "Jack", age = 1)
val olderJack = jack.copy(age = 2)
```

常见的持久性技术(例如 JPA)要求使用默认的构造函数，以防止此类设计。幸运的是，此“[默认构造函数地狱](#)”现在有一种解决方法，因为 Kotlin 提供了[kotlin-jpa](#)插件，该插件可以为使用 JPA 注解的类生成合成的无参数构造函数。

如果您需要将这种机制用于其他持久性技术，则可以配置[kotlin-noarg](#)插件。

#### ❶ Note

从 Kay 发行版开始，Spring Data 支持 Kotlin 不可变类实例，并且如果该模块使用 Spring Data 对象 Map(例如 MongoDB, Redis, Cassandra 等)，则不需要 [kotlin-noarg](#) 插件。

### 1.8.3. 注入依赖

我们的建议是尝试使用 [val](#) 只读(并且在可能时不可为空)[properties](#)来支持构造函数注入，如以下示例所示：

```
@Component
class YourBean(
    private val mongoTemplate: MongoTemplate,
    private val solrClient: SolrClient
)
```

#### ❶ Note

从 Spring Framework 4.3 开始，具有单个构造函数的类的参数将自动自动关联，这就是为什么在上面显示的示例中不需要显式的 [@Autowired constructor](#) 的原因。

如果确实需要使用字段注入，则可以使用 [lateinit var](#) 构造，如以下示例所示：

```
@Component
class YourBean {
```

```
@Autowired  
lateinit var mongoTemplate: MongoTemplate  
  
@Autowired  
lateinit var solrClient: SolrClient  
}
```

## 1.8.4. 注入配置属性

在 Java 中，您可以使用注解(例如 `@Value("${property}")`)注入配置属性。但是，在 Kotlin 中，

`$` 是用于[string interpolation](#)的保留字符。

因此，如果您想在 Kotlin 中使用 `@Value` Comments，则需要通过编写

`@Value("\$${property}")` 来转义 `$` 字符。

另外，您可以pass 语句以下配置 Bean 来定制属性占位符前缀：

```
@Bean  
fun propertyConfigurer() = PropertySourcesPlaceholderConfigurer().apply {  
    setPlaceholderPrefix("%{")  
}
```

您可以使用配置 Bean 自定义使用 `\${...}` 语法的现有代码(例如 Spring BootActuator 或

`@LocalServerPort` )，如以下示例所示：

```
@Bean  
fun kotlinPropertyConfigurer() = PropertySourcesPlaceholderConfigurer().apply {  
    setPlaceholderPrefix("%{")  
    setIgnoreUnresolvablePlaceholders(true)  
}  
  
@Bean  
fun defaultPropertyConfigurer() = PropertySourcesPlaceholderConfigurer()
```

### iNote

如果您使用 Spring Boot，则可以使用[@ConfigurationProperties](#)而不是 `@Value`

Comments。但是，目前，这仅适用于 `lateinit` 或可为空的 `var` 属性(建议使用前者)，因为尚不支持由构造函数初始化的不可变类。有关更多详细信息，请参见有关[不可变 POJO 的@ConfigurationProperties 绑定](#)和[接口上的@ConfigurationProperties 绑定](#)的这些问题。

## 1.8.5. Comments 数组属性

KotlinComments 与 JavaComments 大部分相似，但是数组属性(在 Spring 中广泛使用)的行为有所不同。如[Kotlin documentation](#)中所述，您可以省略 `value` 属性名称(与其他属性不同)，并将其指定为 `vararg` 参数。

要理解这意味着什么，请以 `@RequestMapping` (这是最广泛使用的 SpringComments 之一)为例。此 JavaComments 声明如下：

```
public @interface RequestMapping {  
  
    @AliasFor("path")  
    String[] value() default {};  
  
    @AliasFor("value")  
    String[] path() default {};  
  
    RequestMethod[] method() default {};  
  
    // ...  
}
```

`@RequestMapping` 的典型用例是将处理程序方法 Map 到特定的路径和方法。在 Java 中，可以为 Comments 数组属性指定一个值，该值将自动转换为数组。

这就是为什么可以写 `@RequestMapping(value = "/toys", method = RequestMethod.GET)` 或 `@RequestMapping(path = "/toys", method = RequestMethod.GET)` 的原因。

但是，在 Kotlin 1.2 中，您必须写 `@RequestMapping("/toys", method =`

`[RequestMethod.GET])` 或 `@RequestMapping(path = ["/toys"], method =`

`[RequestMethod.GET])` (必须使用命名数组属性指定方括号)。

此特定 `method` 属性(最常见的一种)的替代方法是使用快捷方式 `Comments`, 例如 `@GetMapping`, `@PostMapping` 等。

### iNote

提醒: 如果未指定 `@RequestMapping` `method` 属性, 则将匹配所有 HTTP 方法, 而不仅是 `GET` 一个。

## 1.8.6. Testing

本节介绍结合 Kotlin 和 Spring 框架进行的测试。

### PER\_CLASS Lifecycle

Kotlin 允许您在反引号(`)注解, 这是一种非常适合 Kotlin。

现在, 由于具有 `junit.jupiter.testinstance.lifecycle.default = per_class` 属性的 `junit-platform.properties` 文件, 您可以将默认行为更改为 `PER_CLASS`。

以下是非静态方法的示例 `@BeforeAll` 和 `@AfterAll` 注解:

```
class IntegrationTests {

    val application = Application(8181)
    val client = WebClient.create("http://localhost:8181")

    @BeforeAll
    fun beforeAll() {
        application.start()
    }

    @Test
    fun `Find all users on HTML page`() {
        client.get().uri("/users")
            .accept(TEXT_HTML)
            .retrieve()
            .bodyToMono<String>()
            .test()
            .expectNextMatches { it.contains("Foo") }
            .verifyComplete()
    }
}
```

```
@AfterAll  
fun afterAll() {  
    application.stop()  
}  
}
```

## Specification-like Tests

您可以使用 JUnit 5 和 Kotlin 创建类似规范的测试。以下示例显示了如何执行此操作：

```
class SpecificationLikeTests {  
  
    @Nested  
    @DisplayName("a calculator")  
    inner class Calculator {  
        val calculator = SampleCalculator()  
  
        @Test  
        fun `should return the result of adding the first number to the second number`() {  
            val sum = calculator.sum(2, 4)  
            assertEquals(6, sum)  
        }  
  
        @Test  
        fun `should return the result of subtracting the second number from the first number`() {  
            val subtract = calculator.subtract(4, 2)  
            assertEquals(2, subtract)  
        }  
    }  
}
```

## Kotlin 中的 WebTestClient 类型推断问题

由于[类型推断问题](#)，您必须使用 Kotlin `expectBody` extensions(例如

`.expectBody<String>().isEqualTo("toys")`)，因为它为 Java API 的 Kotlin 问题提供了一种解决方法。

另请参阅相关的[SPR-16057](#)问题。

## 1.9. 入门

本节描述了结合 Kotlin 和 Spring 框架的项目的最快入门方法。

### 1.9.1. 使用 start.spring.io

在 Kotlin 中启动新的 Spring Framework 5 项目的最简单方法是在[start.spring.io](https://start.spring.io)上创建一个新的 Spring Boot 2 项目。

您还可以按照[此博客文章](#)所述创建独立的 WebFlux 项目。

## 1.9.2. 选择网络风味

Spring Framework 现在带有两个不同的 Web 堆栈：[Spring MVC](#)和[Spring WebFlux](#)。

如果您要创建处理延迟，长期连接，流方案的应用程序，或者要使用网络功能的 Kotlin DSL，建议使用 Spring WebFlux。

对于其他用例，尤其是在使用阻塞技术(例如 JPA)，Spring MVC 及其基于 Comments 的编程模型的情况下，它是完全有效且完全受支持的选择。

## 1.10. Resources

我们建议以下资源供人们学习如何使用 Kotlin 和 Spring 框架构建应用程序：

- [Kotlin 语言参考](#)
- [Kotlin Slack](#)(带有专用的#springChannels)
- [Stackoverflow](#), 带有 spring 和 kotlin 标签
- [在浏览器中尝试 Kotlin](#)
- [Kotlin blog](#)
- [Awesome Kotlin](#)

### 1.10.1. Tutorials

我们建议以下教程：

- [使用 Spring Boot 和 Kotlin 构建 Web 应用程序](#)
- [使用 Spring Boot 创建 RESTful Web 服务](#)

## 1.10.2. 博客文章

以下博客文章提供了更多详细信息：

- [使用 Kotlin 开发 Spring Boot 应用程序](#)
- [带有 Kotlin, Spring Boot 和 PostgreSQL 的地理空间 Messenger](#)
- [在 Spring Framework 5.0 中引入 Kotlin 支持](#)
- [Spring Framework 5 Kotlin API 的功能方式](#)

## 1.10.3. Examples

以下 Github 项目提供了示例，您可以从中学习甚至扩展：

- [spring-boot-kotlin-demo](#): 常规 Spring Boot 和 Spring Data JPA 项目
- [mixit](#): Spring Boot 2, WebFlux 和反应式 Spring Data MongoDB
- [spring-kotlin-functional](#): 独立的 WebFlux 和功能 Bean 定义 DSL
- [spring-kotlin-fullstack](#): WebFlux Kotlin 全栈示例，其中 Kotlin2js 用于前端，而不是 JavaScript 或 TypeScript
- [spring-petclinic-kotlin](#): Spring PetClinic 示例应用程序的 Kotlin 版本
- [spring-kotlin-deepdive](#): 从 Boot 1.0 和 Java 到 Boot 2.0 和 Kotlin 的逐步迁移指南

## 1.10.4. Issues

以下列表对与 Spring 和 Kotlin 支持有关的未决问题进行了分类：

- Spring Framework
- [无法在 Kotlin 中将 WebTestClient 与模拟服务器一起使用](#)
  - [在泛型, 可变参数和数组元素级别支持空安全性](#)
  - [添加对 Kotlin 协程的支持](#)

- Spring Boot
- 允许`@ConfigurationProperties` 绑定用于不可变的 POJO
  - 允许接口上的`@ConfigurationProperties` 绑定
  - 通过`SpringApplication` 公开功能性的 bean 注册 API
  - 在 Spring Boot API 上添加 null 安全 Comments
  - 使用 Kotlin 的 Bom 为 Kotlin 提供依赖项 Management
- Kotlin
- Spring 框架支持的父问题
  - Kotlin 需要 Java 不需要的类型推断
  - 更好的泛型 null 安全支持
  - 开放类的智能演员表回归
  - 不可能将所有 SAM 参数都传递为函数
  - 将 JSR 305 元 Comments 应用于通用类型参数
  - 为库提供一种避免混合 Kotlin 1.0 和 1.1 依赖项的方法
  - 通过脚本变量直接支持 JSR 223 绑定
  - 在 Kotlin Eclipse 插件中支持全开放和无参数编译器插件

## 2. Apache Groovy

---

Groovy 是一种功能强大的，可选类型的动态语言，具有静态键入和静态编译功能。它提供了简洁的语法，并且可以与任何现有的 Java 应用程序顺利集成。

Spring 框架提供了专用的 `ApplicationContext`，该 `ApplicationContext` 支持基于 Groovy 的 Bean 定义 DSL。有关更多详细信息，请参见[Groovy Bean 定义 DSL](#)。

[动态语言支持](#)提供了对 Groovy 的进一步支持，包括用 Groovy 编写的 bean，可刷新的脚本 bean 等。

## 3. 动态语言支持

---

Spring 2.0 引入了对使用类和对象的全面支持，这些类和对象已通过在 Spring 中使用动态语言(例如 JRuby)进行了定义。这种支持使您可以用受支持的动态语言编写任意数量的类，并使 Spring 容器透明地实例化，配置和依赖性注入结果对象。

Spring 当前支持以下动态语言：

- JRuby 1.5+
- Groovy 1.8+
- BeanShell 2.0

为什么只有这些语言？

我们选择支持这些语言是因为：

- 这些语言在 Java 企业社区中具有很大的吸引力。
- 添加此支持时，未请求其他语言的请求
- Spring 开发人员最熟悉它们。

您可以在[Scenarios](#)中找到可以立即使用这种动态语言支持的完整示例。

### 3.1. 第一个例子

本章的大部分内容与详细描述动态语言支持有关。在深入探讨动态语言支持的所有内容之前，我们来看一个使用动态语言定义的 bean 的简单示例。第一个 bean 的动态语言是 Groovy。（该示例的基础取自 Spring 测试套件。如果要查看其他任何受支持语言的等效示例，请查看源代码）。

下一个示例显示了 Groovy bean 将要实现的 `Messenger` 接口。请注意，此接口是用纯 Java 定义的。注入对 `Messenger` 的引用的相关对象不知道基础实现是 Groovy 脚本。以下 Lists 显示了

**Messenger** 接口：

```
package org.springframework.scripting;

public interface Messenger {

    String getMessage();

}
```

下面的示例定义一个对 **Messenger** 接口具有依赖性的类：

```
package org.springframework.scripting;

public class DefaultBookingService implements BookingService {

    private Messenger messenger;

    public void setMessenger(Messenger messenger) {
        this.messenger = messenger;
    }

    public void processBooking() {
        // use the injected Messenger object...
    }

}
```

下面的示例在 Groovy 中实现 **Messenger** 接口：

```
// from the file 'Messenger.groovy'
package org.springframework.scripting.groovy;

// import the Messenger interface (written in Java) that is to be implemented
import org.springframework.scripting.Messenger

// define the implementation in Groovy
class GroovyMessenger implements Messenger {

    String message

}
```

### iNote

要使用定制的动态语言标签来定义支持动态语言的 Bean，您需要在 Spring XML 配置文件的顶部具有 XML Schema 前导。您还需要使用 Spring **ApplicationContext** 实现作为 IoC 容器。

器。支持将动态语言支持的 Bean 与纯 `BeanFactory` 实现一起使用，但是您必须 ManagementSpring 内部的管道。

有关基于架构的配置的更多信息，请参见[基于 XML 模式的配置](#)。

最后，以下示例显示了将 Groovy 定义的 `Messenger` 实现注入到 `DefaultBookingService` 类的实例中的 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:lang="http://www.springframework.org/schema/lang"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans-2.0.xsd
           http://www.springframework.org/schema/lang http://www.springframework.org/schema/lang-2.0.xsd">

    <!-- this is the bean definition for the Groovy-backed Messenger implementation -->
    <lang:groovy id="messenger" script-source="classpath:Messenger.groovy">
        <lang:property name="message" value="I Can Do The Frug" />
    </lang:groovy>

    <!-- an otherwise normal bean that will be injected by the Groovy-backed Messenger -->
    <bean id="bookingService" class="x.y.DefaultBookingService">
        <property name="messenger" ref="messenger" />
    </bean>

</beans>
```

`bookingService` bean(一个 `DefaultBookingService`)现在可以正常使用其私有 `messenger` 成员变量，因为注入到其中的 `Messenger` 实例是一个 `Messenger` 实例。这里没有什么特别的事情-只是普通的 Java 和普通的 Groovy。

希望前面的 XML 代码段是不言自明的，但如果不是，请不要过分担心。`continue` 阅读有关前面配置的原因和原因的详细信息。

## 3.2. 定义由动态语言支持的 Bean

本节准确地描述了如何使用任何受支持的动态语言来定义 Spring 托管的 bean。

注意，本章并不试图解释受支持的动态语言的语法和惯用语。例如，如果您想使用 Groovy 在您的应用程序中编写某些类，我们假设您已经了解 Groovy。如果您需要有关动态语言本身的更多详细

信息，请参阅本章末尾的[Further Resources](#)。

### 3.2.1. 共同概念

使用动态语言支持的 bean 涉及的步骤如下：

- 自然地为动态语言源代码编写测试。
- 然后编写动态语言源代码本身。
- 通过在 XML 配置中使用适当的 `<lang:language/>` 元素来定义支持动态语言的 Bean(您可以使用 Spring API 以编程方式定义此类 Bean，尽管您将必须查阅源代码以获取有关如何执行此操作的指导，因为这本章未涵盖此类高级配置)。请注意，这是一个迭代步骤。每个动态语言源文件至少需要一个 bean 定义(尽管多个 bean 定义可以引用同一个动态语言源文件)。

前两个步骤(测试和编写动态语言源文件)不在本章范围之内。有关选择的动态语言，请参见语言规范和参考手册，并 continue 开发动态语言源文件。不过，您首先要阅读本章的其余部分，因为 Spring 的动态语言支持确实对动态语言源文件的内容做了一些(小的)假设。

#### `<lang:language/>`元素

[preceding section](#) 中列表的最后一步涉及定义动态语言支持的 bean 定义，每个要配置的 bean 定义一个(这与常规 JavaBean 配置没有什么不同)。但是，可以使用 `<lang:language/>` 元素定义动态语言支持的 bean，而不是指定要由容器实例化和配置的类的完全限定的类名。

每种受支持的语言都有一个对应的 `<lang:language/>` 元素：

- `<lang:groovy/>` (时髦)
- `<lang:bsh/>` (BeanShell)
- `<lang:std/>` (JSR-223)

可用于配置的确切属性和子元素完全取决于定义该 bean 所使用的语言(本章稍后的特定于语言的部分对此进行了详细说明)。

## Refreshable Beans

Spring 对动态语言的支持(也许是唯一)中最引人注目的增值之一就是“可刷新 bean”功能。

可刷新的 bean 是动态语言支持的 bean。通过少量配置，支持动态语言的 Bean 可以监视其基础源文件资源中的更改，然后在动态语言源文件更改时重新加载自身(例如，当您在 Windows 上编辑并保存对文件的更改时)。文件系统)。

这使您可以将任何数量的动态语言源文件部署为应用程序的一部分，配置 Spring 容器以创建由动态语言源文件支持的 bean(使用本章中描述的机制)，以及(随后，随着需求的变化或其他一些变化)外部因素起作用)编辑动态语言源文件，并使它们所做的任何更改都反映在更改后的动态语言源文件支持的 Bean 中。无需关闭正在运行的应用程序(或在 Web 应用程序的情况下重新部署)。如此修改的支持动态语言的 Bean 从更改后的动态语言源文件中获取了新的状态和逻辑。

### iNote

默认情况下，此功能是关闭的。

现在我们来看一个例子，看看开始使用可刷新 bean 是多么容易。要启用可刷新 bean 功能，必须在 bean 定义的 `<lang:language/>` 元素上确切指定一个附加属性。因此，如果我们坚持使用本章前面的[the example](#)，则以下示例显示了我们将在 Spring XML 配置中进行哪些更改以实现可刷新的 bean：

```
<beans>

    <!-- this bean is now 'refreshable' due to the presence of the 'refresh-check-delay'
        attribute -->
    <lang:groovy id="messenger"
        refresh-check-delay="5000" <!-- switches refreshing on with 5 seconds between
        script-source="classpath:Messenger.groovy">
        <lang:property name="message" value="I Can Do The Frug" />
    </lang:groovy>

    <bean id="bookingService" class="x.y.DefaultBookingService">
        <property name="messenger" ref="messenger" />
    </bean>

</beans>
```

这确实是您要做的。在 `messenger` bean 定义上定义的 `refresh-check-delay` 属性是毫秒数

, 在此毫秒后, 将对基础动态语言源文件进行任何更改来刷新 bean。您可以通过为 `refresh-check-delay` 属性分配负值来关闭刷新行为。请记住, 默认情况下, 刷新行为是禁用的。如果您不希望刷新行为, 请不要定义属性。

如果然后运行以下应用程序, 则可以使用可刷新的功能(请在下一部分代码中使用“通过箍跳转到暂停执行”的假名)。`System.in.read()` 调用仅在此处当您(在此情况下为开发人员)关闭并编辑基础动态语言源文件时, 程序的执行将暂停, 以便在程序恢复执行时在由动态语言支持的 bean 上触发刷新。

以下 Lists 显示了此示例应用程序:

```
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("beans.xml");
        Messenger messenger = (Messenger) ctx.getBean("messenger");
        System.out.println(messenger.getMessage());
        // pause execution while I go off and make changes to the source file...
        System.in.read();
        System.out.println(messenger.getMessage());
    }
}
```

然后, 出于本示例的目的, 假设必须更改对 `Messenger` 实现的 `getMessage()` 方法的所有调用, 以使消息用引号起来。以下 Lists 显示了您(开发人员)应在程序执行暂停时对

`Messenger.groovy` 源文件进行的更改:

```
package org.springframework.scripting

class GroovyMessenger implements Messenger {

    private String message = "Bingo"

    public String getMessage() {
        // change the implementation to surround the message in quotes
        return '"' + this.message + '"'
    }

    public void setMessage(String message) {
        this.message = message
    }
}
```

```
    }  
}
```

程序运行时，Importing 暂停之前的输出为 `I Can Do The Frug`。更改并保存对源文件的更改，并且程序恢复执行后，在支持动态语言的 `Messenger` 实现上调用 `getMessage()` 方法的结果是 `'I Can Do The Frug'` (请注意包含附加引号)。

如果更改发生在 `refresh-check-delay` 值的窗口内，则对脚本所做的更改不会触发刷新。直到在支持动态语言的 Bean 上调用方法之前，实际上不会对脚本进行更改。只有在支持动态语言的 Bean 上调用方法时，它才会检查其基础脚本源是否已更改。与刷新脚本有关的任何异常(例如遇到编译错误或发现脚本文件已被删除)都将导致致命异常传播到调用代码。

前面描述的可刷新 bean 行为不适用于用 `<lang:inline-script/>` 元素表示法定义的动态语言源文件(请参见[内联动态语言源文件](#))。此外，它仅适用于实际上可以检测到基础源文件更改的 Bean(例如，通过检查文件系统上存在的动态语言源文件的最后修改日期的代码)。

## 内联动态语言源文件

动态语言支持还可以迎合直接嵌入在 Spring bean 定义中的动态语言源文件。更具体地说，您可以使用 `<lang:inline-script/>` 元素在 Spring 配置文件中立即定义动态语言源。一个示例可以阐明内联脚本功能的工作方式：

```
<lang:groovy id="messenger">  
  <lang:inline-script>  
  
    package org.springframework.scripting.groovy;  
  
    import org.springframework.scripting.Messenger  
  
    class GroovyMessenger implements Messenger {  
      String message  
    }  
  
  </lang:inline-script>  
  <lang:property name="message" value="I Can Do The Frug" />  
</lang:groovy>
```

如果将有关在 Spring 配置文件中定义动态语言源是否是一种好习惯的问题放在一边，则

`<lang:inline-script/>` 元素在某些情况下会很有用。例如，我们可能想快速将 Spring

`Validator` 实现添加到 Spring MVC `Controller`。使用内联源代码只是一时的工作。(有关此类示例，请参见[Scripted Validators](#)。)

## 在动态语言支持的 Bean 上下文中了解构造函数注入

关于 Spring 的动态语言支持，有一件非常重要的事情要注意。即，您不能(当前)向动态语言支持的 bean 提供构造函数参数(因此，构造函数注入不适用于动态语言支持的 bean)。为了使对构造函数和属性的特殊处理 100% 清晰，以下代码和配置的混合不起作用：

### 例子 1.一种行不通的方法

```
// from the file 'Messenger.groovy'
package org.springframework.scripting.groovy;

import org.springframework.scripting.Messenger

class GroovyMessenger implements Messenger {

    GroovyMessenger() {}

    // this constructor is not available for Constructor Injection
    GroovyMessenger(String message) {
        this.message = message;
    }

    String message

    String anotherMessage

}
```

```
<lang:groovy id="badMessenger"
script-source="classpath:Messenger.groovy">
<!-- this next constructor argument will not be injected into the GroovyMessenger --
<!-- in fact, this isn't even allowed according to the schema -->
<constructor-arg value="This will not work" />

<!-- only property values are injected into the dynamic-language-backed object -->
<lang:property name="anotherMessage" value="Passed straight through to the dynamic-->

</lang>
```

实际上，这种限制并不像它最初出现的那样重要，因为二传手注入是绝大多数开发人员所偏爱的注入方式(我们将讨论是否对另一天来说是一件好事)。

### 3.2.2. 杂色 bean

本节描述如何在 Spring 中使用 Groovy 中定义的 bean。

Groovy 库依赖

Spring 中的 Groovy 脚本支持需要以下库位于应用程序的 Classpath 中：

- groovy-1.8.jar
- asm-3.2.jar
- antlr-2.7.7.jar

Groovy 主页包括以下描述：

“Groovy 是 Java 2 平台的一种敏捷动态语言，具有许多人们喜欢的功能，例如 Python, Ruby 和 Smalltalk，使它们可以使用类似 Java 的语法供 Java 开发人员使用。”

如果您从头开始直接阅读了本章，则已经有了[看到一个例子](#)的 Groovy 动态语言支持的 bean。现在考虑另一个示例(再次使用 Spring 测试套件中的示例)：

```
package org.springframework.scripting;

public interface Calculator {
    int add(int x, int y);
}
```

下面的示例在 Groovy 中实现 `Calculator` 接口：

```
// from the file 'calculator.groovy'
package org.springframework.scripting.groovy

class GroovyCalculator implements Calculator {

    int add(int x, int y) {
        x + y
    }
}
```

以下 bean 定义使用 Groovy 中定义的计算器：

```
<-- from the file 'beans.xml' -->
<beans>
    <lang:groovy id="calculator" script-source="classpath:calculator.groovy"/>
</beans>
```

最后，以下小型应用程序将执行上述配置：

```
package org.springframework.scripting;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");
        Calculator calc = (Calculator) ctx.getBean("calculator");
        System.out.println(calc.add(2, 8));
    }
}
```

运行上述程序的结果输出是 [10](#) (毫不奇怪)。 (有关更多有趣的示例，请参见动态语言展示项目以获取更复杂的示例，或参见本章稍后的示例[Scenarios](#))。

每个 Groovy 源文件中定义的类不得超过一个。尽管这在 Groovy 中是完全合法的，但(可以说)这是一种不好的做法。为了采用一致的方法，您(在 Spring 小组看来)应该遵守每个源文件一个(公共)类的标准 Java 约定。

## 通过使用回调自定义 Groovy 对象

[GroovyObjectCustomizer](#) 接口是一个回调，使您可以将其他创建逻辑挂接到创建 Groovy 支持的 bean 的过程中。例如，此接口的实现可以调用任何必需的初始化方法，设置一些默认属性值或指定自定义 [MetaClass](#)。以下 Lists 显示了 [GroovyObjectCustomizer](#) 接口定义：

```
public interface GroovyObjectCustomizer {

    void customize(GroovyObject goo);
}
```

Spring 框架实例化您的 Groovy 支持的 bean 的实例，然后将创建的 [GroovyObject](#) 传递给指定的 [GroovyObjectCustomizer](#) (如果已定义)。您可以使用提供的 [GroovyObject](#) 参考进行任何操作

。我们希望大多数人都希望通过此回调设置自定义 `MetaClass`，以下示例显示了如何执行此操作：

```
public final class SimpleMethodTracingCustomizer implements GroovyObjectCustomizer {  
  
    public void customize(GroovyObject goo) {  
        DelegatingMetaClass metaClass = new DelegatingMetaClass(goo.getMetaClass()) {  
  
            public Object invokeMethod(Object object, String methodName, Object[] arguments) {  
                System.out.println("Invoking '" + methodName + "'.");  
                return super.invokeMethod(object, methodName, arguments);  
            }  
        };  
        metaClass.initialize();  
        goo.setMetaClass(metaClass);  
    }  
}
```

Groovy 中对元编程的完整讨论超出了 Spring 参考手册的范围。请参阅 Groovy 参考手册的相关部分，或在线进行搜索。很多文章讨论了这个主题。实际上，如果使用 Spring 名称空间支持，使用 `GroovyObjectCustomizer` 很容易，如以下示例所示：

```
<!-- define the GroovyObjectCustomizer just like any other bean -->  
<bean id="tracingCustomizer" class="example.SimpleMethodTracingCustomizer"/>  
  
<!-- ... and plug it into the desired Groovy bean via the 'customizer-ref' attribute  
<lang:groovy id="calculator"  
    script-source="classpath:org/springframework/scripting/groovy/Calculator.groovy"  
    customizer-ref="tracingCustomizer"/>
```

如果不使用 Spring 名称空间支持，则仍然可以使用 `GroovyObjectCustomizer` 功能，如以下示例所示：

```
<bean id="calculator" class="org.springframework.scripting.groovy.GroovyScriptFactory">  
    <constructor-arg value="classpath:org/springframework/scripting/groovy/Calculator.groovy"/>  
    <!-- define the GroovyObjectCustomizer (as an inner bean) -->  
    <constructor-arg>  
        <bean id="tracingCustomizer" class="example.SimpleMethodTracingCustomizer"/>  
    </constructor-arg>  
</bean>  
  
<bean class="org.springframework.scripting.support.ScriptFactoryPostProcessor" />
```

### iNote

从 Spring Framework 4.3.3 开始，您还可以在与 Spring 的 `GroovyObjectCustomizer` 相同的位置指定 Groovy `CompilationCustomizer` (例如 `ImportCustomizer`) 甚至是完整的 Groovy `CompilerConfiguration` 对象。

### 3.2.3. BeanShellbean

本节描述如何在 Spring 中使用 BeanShell bean。

BeanShell 库依赖项

Spring 中的 BeanShell 脚本支持需要以下库位于应用程序的 Classpath 中：

- `bsh-2.0b4.jar`

BeanShell 主页包含以下描述：\{

“ BeanShell 是一个小型的，免费的，可嵌入的 Java 源代码解释器，具有用 Java 编写的动态语言功能。 BeanShell 可动态执行标准 Java 语法，并通过通用的脚本编写便利进行扩展，例如松散类型，命令和方法闭包(如 Perl 和 JavaScript 中的那些)。”

与 Groovy 相比， BeanShell 支持的 bean 定义需要一些(小的)附加配置。在 Spring 中实现 BeanShell 动态语言支持很有趣，因为 Spring 创建了一个 JDK 动态代理，该代理实现了 `<lang:bsh>` 元素的 `script-interfaces` 属性值中指定的所有接口(这就是为什么必须在其中提供至少一个接口的原因)属性的值，并因此在使用 BeanShell 支持的 bean 时编程到接口)。这意味着对 BeanShell 支持的对象的每个方法调用都将通过 JDK 动态代理调用机制进行。

现在，我们可以显示一个使用基于 BeanShell 的 Bean 的完整示例，该 Bean 实现了本章前面定义的 `Messenger` 接口。我们再次显示 `Messenger` 接口的定义：

```
package org.springframework.scripting;
public interface Messenger {
    String getMessage();
```

```
}
```

下面的示例显示了 `Messenger` 接口的 BeanShell“实现”(在此我们宽松地使用术语):

```
String message;

String getMessage() {
    return message;
}

void setMessage(String aMessage) {
    message = aMessage;
}
```

以下示例显示了 Spring XML, 该 XML 定义了上述“类”的“实例”(再次, 我们在这里非常宽松地使用这些术语):

```
<lang:bsh id="messageService" script-source="classpath:BshMessenger.bsh"
script-interfaces="org.springframework.scripting.Messenger">

<lang:property name="message" value="Hello World!" />
</lang:bsh>
```

有关您可能要使用基于 BeanShell 的 bean 的某些方案, 请参见[Scenarios](#)。

## 3.3. Scenarios

用脚本语言定义 Spring 托管 Bean 会带来好处的可能方案有很多, 而且有很多。本节描述了 Spring 中对动态语言支持的两种可能的用例。

### 3.3.1. 脚本化 Spring MVC 控制器

可以从使用动态语言支持的 bean 中受益的一组类是 Spring MVC 控制器的类。在纯 Spring MVC 应用程序中, 通过 Web 应用程序的导航流程在很大程度上取决于封装在 Spring MVC 控制器中的代码。由于需要更新 Web 应用程序的导航流程和其他表示层逻辑以响应支持问题或不断变化的业务需求, 因此通过编辑一个或多个动态语言源文件并查看这些变化可能很容易实现任何此类必需的更改。更改立即反映在正在运行的应用程序的状态中。

请记住, 在像 Spring 这样的项目所拥护的轻量级架构模型中, 您通常旨在拥有一个 true 的瘦表示

层，而应用程序的所有繁琐的业务逻辑都包含在域和服务层类中。通过将 Spring MVC 控制器开发为支持动态语言的 Bean，可以通过编辑和保存文本文件来更改表示层逻辑。对此类动态语言源文件的任何更改(取决于配置)都会自动反映在由动态语言源文件支持的 Bean 中。

### iNote

要对动态语言支持的 bean 进行任何更改的这种自动“拾取”，必须启用“可刷新 bean”功能。有关此功能的完整说明，请参见[Refreshable Beans](#)。

以下示例显示了使用 Groovy 动态语言实现的

`org.springframework.web.servlet.mvc.Controller`：

```
// from the file '/WEB-INF/groovy/FortuneController.groovy'
package org.springframework.showcase.fortune.web

import org.springframework.showcase.fortune.service.FortuneService
import org.springframework.showcase.fortune.domain.Fortune
import org.springframework.web.servlet.ModelAndView
import org.springframework.web.servlet.mvc.Controller

import javax.servlet.http.HttpServletRequest
import javax.servlet.http.HttpServletResponse

class FortuneController implements Controller {

    @Property FortuneService fortuneService

    ModelAndView handleRequest(HttpServletRequest request,
        HttpServletResponse httpServletResponse) {
        return new ModelAndView("tell", "fortune", this.fortuneService.tellFortune())
    }
}
```

```
<lang:groovy id="fortune"
    refresh-check-delay="3000"
    script-source="/WEB-INF/groovy/FortuneController.groovy">
    <lang:property name="fortuneService" ref="fortuneService"/>
</lang:groovy>
```

### 3.3.2. 脚本验证器

使用 Spring 可以从动态语言支持的 bean 提供的灵 Active 中受益的应用程序开发的另一个领域是验证领域。与常规 Java 相比，使用松散类型的动态语言(也可能支持内联正则表达式)来表达复杂的

验证逻辑会更容易。

同样，将验证器开发为动态语言支持的 bean，使您可以通过编辑和保存简单的文本文件来更改验证逻辑。任何此类更改(取决于配置)都会自动反映在正在运行的应用程序的执行中，而无需重新启动应用程序。

### iNote

要对支持动态语言的 bean 进行任何更改的自动“提取”，必须启用“可刷新 bean”功能。

有关此功能的完整和详细说明，请参见[Refreshable Beans](#)。

以下示例显示了使用 Groovy 动态语言实现的 Spring

`org.springframework.validation.Validator` (有关 `Validator` 接口的讨论，请参见[使用 Spring 的 Validator 接口进行验证](#)):

```
import org.springframework.validation.Validator
import org.springframework.validation.Errors
import org.springframework.beans.TestBean

class TestBeanValidator implements Validator {

    boolean supports(Class clazz) {
        return TestBean.class.isAssignableFrom(clazz)
    }

    void validate(Object bean, Errors errors) {
        if(bean.name?.trim()?.size() > 0) {
            return
        }
        errors.reject("whitespace", "Cannot be composed wholly of whitespace.")
    }
}
```

## 3.4. 额外细节

最后一部分包含与动态语言支持有关的一些其他详细信息。

### 3.4.1. AOP — 建议脚本化 Bean

您可以使用 Spring AOP 框架来建议脚本化的 bean。实际上，Spring AOP 框架没有意识到建议使

用的 Bean 可能是脚本 Bean，因此您使用(或打算使用)的所有 AOP 用例和功能都可以与脚本 Bean 一起使用。当建议脚本 bean 时，不能使用基于类的代理。您必须使用[interface-based proxies](#)。

您不仅限于建议脚本化的 bean。您还可以使用受支持的动态语言自己编写方面，并使用此类 bean 来建议其他 Spring bean。不过，这确实是对动态语言支持的高级使用。

### 3.4.2. Scoping

万一这不是立即显而易见的话，可以以与任何其他 Bean 相同的方式确定脚本 Bean 的范围。各种 `<lang:language/>` 元素上的 `scope` 属性使您可以像使用常规 Bean 一样控制基础脚本化 Bean 的范围。(默认范围是[singleton](#)，与“常规” bean 一样。)

下面的示例使用 `scope` 属性定义范围为 [prototype](#) 的 Groovy bean：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:lang="http://www.springframework.org/schema/lang"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans-2.0.xsd
           http://www.springframework.org/schema/lang http://www.springframework.org/schema/lang-2.0.xsd">

    <lang:groovy id="messenger" script-source="classpath:Messenger.groovy" scope="prototype">
        <lang:property name="message" value="I Can Do The RoboCop" />
    </lang:groovy>

    <bean id="bookingService" class="x.y.DefaultBookingService">
        <property name="messenger" ref="messenger" />
    </bean>

</beans>
```

有关 Spring 框架中范围支持的完整讨论，请参见[IoC 容器](#)中的[Bean scopes](#)。

### 3.4.3. lang XML 模式

Spring XML 配置中的 `lang` 元素用于将以动态语言(例如 JRuby 或 Groovy)编写的对象暴露为 Spring 容器中的 bean。

这些元素(以及动态语言支持)在[动态语言支持](#)中全面介绍。有关此支持和 `lang` 元素的完整详细信息，请参见该章。

要使用 `lang` 模式中的元素，您需要在 Spring XML 配置文件的顶部具有以下序言。以下代码段中的文本引用了正确的架构，以便您可以使用 `lang` 名称空间中的标记：

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

<!-- bean definitions here -->

</beans>
```

## 3.5. 更多资源

以下链接提供了有关本章中描述的各种动态语言的更多资源：

- [JRuby](#)主页
- [Groovy](#)主页
- [BeanShell](#)主页