# Spring概述

Spring让创造企业级的Java应用变得轻松。它提供了在企业环境中使用Java语言的一切，支持Groovy和Kotlin作为JVM上的替代语言，并可根据应用程序的需要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提供了开箱即用的支持。

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，并且不得不运行在JDK环境中，同时应用服务器的升级周期超出开发人员控制范围。另外有一些其中嵌入了服务器可,可能在云环境中作为单个jar运行。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，提供持续不断的，基于各种各样的实际用例的反馈。这有助于Spring在很长一段时间内成功发展。

## Spring意味着什么

术语“spring”在不同的上下文中的所指代的东西有所不同。一开始的时候，它可能用来指代spring框架本身。而时过境迁，有一些基于spring框架的其他spring项目逐渐被创造了出来。目前大部分情况下，人们口中所说的“spring”，指的是整个spring家族的所有项目。本片文档主要用于阐述一切的基石：spring框架本身

整个spring框架被分为多个模块。应用可以选择其所需要的模块。最核心的模块是一个核心容器，包括一个配置模式和一个依赖注入机制。除去这些，spring框架提供了针对不同应用架构的基础支持，包括消息、事务数据和持久化，还有网络。同时它还包括基于servlet的spring MVC框架，以及与其同级别的，响应式web框架Spring WebFlux。

有关各个模块的说明：spring的框架jar包允许部署在JDK9的模块路径下（“Jigsaw”）。为了支持启用了Jigsaw的应用，spring 5框架jar包带有“Automatic-Module-Name”的MANIFEST（清单条目），它定义了以固定命名法命名的模块名称（例如“spring.core”，“spring.context”等等）独立于jar包架构名（那些jar包使用同样的名称，以分隔符“-”分隔，而不是“”，例如“spring-core”和“spring-context”）。当然spring框架的所有jar包同样能在JDK8和JDK9环境的classpath中正常使用。

## Spring以及Spring框架的历史

Spring成立于2003年，是早期复杂的J2EE规范的响应。有很多人将J2EE与spring视为竞争关系，然而实际上是，spring是J2EE的补充。Spring编程模式并没有接受Java EE平台的所有规范，相反，它结合了JavaEE规范中的精华：

* Servlet API ([JSR 340](https://jcp.org/en/jsr/detail?id=340))
* WebSocket API ([JSR 356](https://www.jcp.org/en/jsr/detail?id=356))
* Concurrency Utilities ([JSR 236](https://www.jcp.org/en/jsr/detail?id=236))
* JSON Binding API ([JSR 367](https://jcp.org/en/jsr/detail?id=367))
* Bean Validation ([JSR 303](https://jcp.org/en/jsr/detail?id=303))
* JPA ([JSR 338](https://jcp.org/en/jsr/detail?id=338))
* JMS ([JSR 914](https://jcp.org/en/jsr/detail?id=914))
* 以及必要情况下用于事务协调的JTA/JCA设置

Spring框架同样支持依赖注入（JSR330）和通用注解（JSR250）规范可供开发者使用，而并非选用spring框架提供的spring内部定义机制。

对于spring 5.0，需要至少Java EE 7（例如Servlet 3.1+，JPA2.1+），然而与此同时还提供了在运行时与更新的Java EE 8 API（例如Servlet4.0，JSON Binding绑定API）达到了开箱即用的整合效果。这保证了spring完全兼容tomcat 8 和 9，WebSphere 9，和JBoss EAP 7。

随着时间的推移，Java EE在应用的开发上已经发展进步。在过去的Java EE和spring中，应用在开发完成后需要部署在一个应用服务器上。而现在，在spring boot的帮助下，应用可以以开发友好和云友好的方式创建，带有内置servlet容器且易更改。到了spring 5框架，一个WebFlux应用甚至不需要直接使用servlet API也能在非servlet容器（比如Netty）上运行。

Spring一直在不断地创新发展。在spring框架以外，还有其他的一些项目，例如spring boot，spring security，spring data，spring cloud，spring batch等等，重要的是要记住每个项目都有自己的源代码存储库，问题跟踪器和发布节奏。有关Spring项目的完整列表，请参见[spring.io/projects](https://spring.io/projects)。

## 设计理念

当你在学习一个框架的时候，重要的是不仅仅需要知道它能做什么，还要知道它遵循的是什么理念。以下就是spring框架的指导思想：

* 每一个等级都提供选择。Spring让你尽可能地推迟设计决策。例如，你可以通过配置来更换持久化软件而不是改变你的代码。许多其他基础架构问题以及与第三方API的集成也是如此。
* 允许实现的多样性。Spring坚持它的灵活性并不固执地认为功能应该具体怎么样实现。它广泛的支持不同各种不同需求的应用。
* 保持强大的向后兼容能力。Spring的版本更替会经过精心的设计保证版本之间的出现尽量少的重大变动。Spring精心选择过支持的JDK版本范围和一些第三方库，以方便一些依赖spring的应用和第三方库便于维护
* 在API的设计上事无巨细。Spring团队在创建API的时候付出了大量的思考和时间，使其直观以及能够支撑多年多版本的使用。
* 高标准的代码质量。Spring框架强调有意义的，最新的且准确的Javadoc。对外宣称具有干净的代码架构且包与包之间没有循环依赖，这是极少数项目能够做到的。

## 反馈与贡献

针对如何操作的问题以及诊断或调试问题，我们建议使用StackOverflow，同时我们有一个列出了建议使用的标签的问题页。如果你十分确定在spring框架内部有一个问题或者希望添加一个特性，请使用 [JIRA issue tracker](https://jira.spring.io/browse/spr).

如果你的心中已有解决方案或者建议的解决办法，你可以在GitHub中提交一个pull请求。但是请记住，除了最微不足道的问题，我们期望能够在问题追踪器中保存有一个票据，记录下讨论点以供将来参考。

详情请查看[CONTRIBUTING](https://github.com/spring-projects/spring-framework/blob/master/CONTRIBUTING.md)项目首页的导航栏。

## 入门

如果你是spring的初学者，你可能需要通过创建一个基于spring boot的应用来开始使用spring框架。Spring boot提供了一个快速的（但spring式的）方式去创建一个基于spring的生产就绪的应用。其基于spring框架，支持约定高于配置，旨在帮助你进行敏捷开发。

你可以使用[start.spring.io](https://start.spring.io/)生成一个基础项目或者跟随其中一个[入门指引](https://spring.io/guides)，例如 [入门：创建一个RESTful的web服务](https://spring.io/guides/gs/rest-service/)。除了更容易理解以外，这些指引也十分注重实战，而且他们大部分都是基于spring boot。当解决一个特定的问题时，他们可能还涵盖了你可能会考虑使用的spring家族的其他项目。

# 核心技术

**这部分的参考文档涵盖了全部完全融入spring框架的不可或缺的技术。**

其中最重要的是spring框架的控制反转（IOC）容器。紧随其后的是spring框架内随处可见的面向切面编程（AOP）技术。Spring框架有其自己的AOP框架，它简明易懂，在Java企业编程中完美地解决了80%的AOP需求。

提供涵盖spring与aspectJ（当前在Java企业领域中，功能特性方面最丰富，也是最成熟的AOP实现框架）的整合

## IOC容器

* + 1. Spring IOC容器与beans的介绍

本章将涵盖spring框架对IOC理念的实现。IOC也被称为依赖注入（DI）。这是一道对象定义其自身依赖的工序，所谓依赖，也就是说那些对象内部所使用的对象是一个通过构造器或者工厂方法返回的实例。将此实例通过作为前述对象构造方法入参、工厂方法参数或者设置为内部属性。容器在创建bean的时候将为其注入那些依赖。这个过程就是基本的反转，因此称为*控制反转*（IoC），bean本身通过使用类的直接构造或诸如*服务定位器*模式之类的机制来控制其依赖关系的实例化或部署。

包org.springframework.beans和包org.springframework.context是spring框架IOC容器的基础。BeanFactory接口提供了使用任意类型的更丰富配置机制。[ApplicationContext](https://docs.spring.io/spring-framework/docs/5.0.12.RELEASE/javadoc-api/org/springframework/context/ApplicationContext.html)是BeanFactory接口的一个子接口。它使得整合spring AOP特性更容易；消息资源处理（用于国际化），事件发布；还有应用层专用容器例如用于web应用的WebApplicationContext。

简而言之，BeanFactory提供了配置式的架构和基础功能，然后ApplicationContext增加了更多企业级的功能。ApplicationContext可以说完全是BeanFactory的超集，而且在本章中专门用于描述spring的IOC容器。更多关于BeanFactory而不是ApplicationContext的相关信息，请参考 [The BeanFactory](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-beanfactory)。

在spring中，那些成为你的应用的骨架和被spring IOC容器管理的对象被称为beans。一个bean，是被一个spring IOC容器实例化、组装和管理的对象，要不然它只是在你的应用众多的对象中普通的一个。Beans跟他们之间的依赖关系反映在容器的配置元数据中。

* + 1. 容器概述

org.springframework.context.ApplicationContext代表了spring IOC容器，负责实例化、配置、组装前述的那些bean。容器通过读取配置元数据来获取有关对象实例化、配置和组装的指令。这些配置元数据可由XML配置文件、Java注解或者Java代码确定。你可以在其中定义组成你的应用的对象以及其间的相互依赖关系。

Spring提供了一些开箱即用的ApplicationContext的实现。在一个单独的应用中，创建一个[ClassPathXmlApplicationContext](https://docs.spring.io/spring-framework/docs/5.0.12.RELEASE/javadoc-api/org/springframework/context/support/ClassPathXmlApplicationContext.html) 或者 [FileSystemXmlApplicationContext](https://docs.spring.io/spring-framework/docs/5.0.12.RELEASE/javadoc-api/org/springframework/context/support/FileSystemXmlApplicationContext.html)很常见。然而XML文件的方式已经传统且老旧，你可以指定容器去使用支持，通过少量的XML文件声明式地启用有关Java注解式配置或者代码式配置。

在大部分的应用使用场景中，实例化一个或者多个IOC容器显式的配置代码不是必须的。比如，在一个web应用场景中，应用的web.xml文件中一个简单到8行（或多或少）的web模板描述就足够了（参照 [用于web应用的轻便ApplicationContext实例化](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#context-create)）。如果你使用的是[Spring Tool Suite](https://spring.io/tools/sts) Eclipse版增强开发环境，那么这个模板文件的配置就是点点鼠标按几下键盘的事。

下图是spring IOC工作方式的上帝视角。你的应用中的类结合了配置元数据，因此在ApplicationContext创建和初始化过后，你就能得到一个已完全配置的、可执行的系统或者应用。



*Figure 1. The Spring IoC container*

* + - 1. 配置元数据

如上图所示，spring IOC容器使用固定格式的配置元数据；这个配置元数据代表了你，作为一个应用开发者，告诉spring容器应该如何去实例化、配置、组装你应用中的对象。

配置元数据通过传统的简单直观的XML格式文件提供，也是在这一章中在表达spring IOC容器主要概念和特性时所用的方式。

基于XML格式的元数据不是配置元数据仅有的格式。Spring IOC容器本身已经与实际写入配置元数据的格式完全解耦。现在部分开发者针对他们的spring应用选择 [基于Java配置](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-java)的方式

有关在spring容器中其他形式的元数据配置请查看：

* [基于注解的配置](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-annotation-config)：spring 2.5介绍了基于注解的配置元数据的支持。
* [基于Java的配置](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-java)：从spring 3.0开始，提供许多特性的spring JavaConfig项目成为spring核心框架的一部分。因此你可以通过使用Java代码而不是XML文件在你的应用类的外部定义bean。若需要使用这些新的特性，请见@Configuration, @Bean, @Import 和@DependsOn这些注解。

Spring配置包含了，必须由容器管理的，至少一个一般来说是多个bean的定义。顶层<beans/>标签内部的<bean/>标签所展示的信息就是所有bean基于XML的配置元数据。对于基于的Java配置一般来说在一个标记了@Configuration注解的类中使用一个以注解@Bean标记的方法即可。

这些bean的定义与组成你的应用的对象直接关联。一般来说你会定义业务层对象、数据访文对象（DAOs）、表现层对象例如Struts的Action实例、基础架构对象例如Hibernate的SessionFactories、JMS的Queues等等。通常不会在容器内创建细粒度的域对象，因为这些是由DAOs的职责和业务逻辑去决定创建与加载的。然而你可以使用由spring整合的AspectJ去配置那些在IOC容器控制外创建的对象。

如下展示的就是基于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="...">

*<!-- 协作者及此bean的配置在此 -->*

</bean>

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

*<!-- 协作者及此bean的配置在此 -->*

</bean>

*<!-- 更多bean定义在此 -->*

</beans>

属性id是一个用于指定一个特定bean定义的字符串。属性class则是使用类的全限定名来定义对象的类型。属性id的值指向其对应的对象。对象之间的协作配置信息没有在这个XML示例中体现出来；详情请参考[Dependencies](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-dependencies)。

* + - 1. 实例化一个容器

实例化一个spring IOC容器可谓简单粗暴。提供给ApplicationContext构造器的路径参数实际上是一串字符串资源，允许容器从外部资源加载配置元信息，如本地文件系统、Java的CLASSPATH等等。

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

在了解spring IOC容器之后，你可能想了解更多有关于被称为[Resources](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#resources)的，抽象为spring内部资源resource的信息。它为使用URI语法描述的资源路径提供了轻便的读取流机制。特别是，在[应用上下文环境与资源路径](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#resources-app-ctx)中，resource资源路径用来构建应用的上下文信息。

以下示例是服务层对象的配置文件services.xml：

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

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

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

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

*<!-- services -->*

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

<property name="accountDao" ref="accountDao"/>

<property name="itemDao" ref="itemDao"/>

*<!-- 更多协作者与配置在此省略 -->*

</bean>

*<!-- 更多该服务的bean定义在此省略 -->*

</beans>

以下示例是数据访问对象的配置文件daos.xml：

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

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

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

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

<bean id="accountDao" class="org.springframework.samples.jpetstore.dao.jpa.JpaAccountDao">

*<!-- 更多协作者以及bean配置在此省略 -->*

</bean>

<bean id="itemDao" class="org.springframework.samples.jpetstore.dao.jpa.JpaItemDao">

*<!-- 更多协作者以及bean配置在此省略 -->*

</bean>

*<!-- 更多数据访问对象的bean定义在此省略 -->*

</beans>

在前述示例中，服务层包含类PetStoreServiceImpl，和两个类型分别为JpaAccountDao和JpaItemDao数据访问对象（基于JPA对象关系映射ORM标准）。元素property name指代JavaBean属性的名字，然后元素ref指向另一个bean定义的名字。元素id和ref之间的联系表明了各个协作对象之间的相互依赖关系。关于配置对象间的依赖的更多细节，见[Dependencies](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-dependencies)。

组合XML配置元数据

将多个bean定义分布在多个不同的XML文件中是有用的，通常每个独立的XML配置文件都代表着一个逻辑层或者一个在你的架构中的模块。

你可以使用应用上下文构造器通过这些XML片段去加载所有的bean定义。如前一小节中所说，这个构造器可接收多个资源resource路径。或者，使用一个或者多个<import/>元素去加载别的文件中的bean。如：

<beans>

<import resource="services.xml"/>

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

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

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

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

</beans>

在上述例子中，外部bean定义通过services.xml, messageSource.xml, 和themeSource.xml这三个文件去加载。在执行导入操作时，所有的路径都是相对于定义文件的，因此services.xml必须与执行导入操作的文件在同一个目录或者类路径地址（classpath location），而messageSource.xml和themeSource.xml必须在导入文件所在位置下的resources目录中。如你所见，以斜杠开头会被忽略，但鉴于路径是相对路径，最好不要使用斜杠。依据Spring Schema，导入文件的内容，包括顶层标签<beans/>，必须是有效的bean定义。

有可能但是不推荐，使用相对路径“../”去指向一个父级目录。这么做会在当前应用之外的一个文件中创建依赖。特别是，不推荐使用在“classpath:”的URL中（比如："classpath:../services.xml"），因为运行时解析进程会选择“最近的”classpath根目录然后查找它的父目录。Classpath配置的改变会导致选择一个不同的、不正确的目录。

你可以一直使用资源路径的绝对路径而不是相对路径：例如"file:C:/config/services.xml" 或者"classpath:/config/services.xml"。但是请注意，你将你的应用配置与明确的绝对路径耦合了。通常来说保持与绝对路径有一个间接关系，例如，通过由JVM系统在运行时解析占位符 "${…​}" 得到的属性。

直接导入配置文件的特性是由beans这个命名空间提供的。Spring提供了许多比普通的bean定义更多的配置特性在XML文件的命名空间里可供选择，例如"context"命名空间和“util”命名空间。

Groovy的bean定义DSL

作为外部定义配置元信息的另一个示例，bean定义同样可以使用spring的Groovy bean定义语言DSL，也就是来自于Grails框架。一般来说，这些配置需要保存在一个带有".groovy"后缀的文件中，带有如下样式的结构：

beans {

dataSource(BasicDataSource) {

driverClassName = "org.hsqldb.jdbcDriver"

url = "jdbc:hsqldb:mem:grailsDB"

username = "sa"

password = ""

settings = [mynew:"setting"]

}

sessionFactory(SessionFactory) {

dataSource = dataSource

}

myService(MyService) {

nestedBean = { AnotherBean bean ->

dataSource = dataSource

}

}

}

这种配置风格与XML格式的bean定义很大程度上是类似的，甚至支持spring的XML配置命名空间。它同样允许通过“importBeans”来直接导入XML bean定义文件。

* + - 1. 使用容器

ApplicationContext是一个增强型工厂的接口，用来维护不同bean跟它们的依赖的注册表。通过使用方法T getBean(String name, Class<T> requiredType)你能够获取到你的bean的实例。

ApplicationContext能够让你读取bean的定义并通过如下代码访文它们：

*// 创建和配置bean*

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

*// 获取已配置的实例*

PetStoreService service = context.getBean("petStore", PetStoreService.class);

*// 使用已经配置的实例*

List<String> userList = service.getUsernameList();

在Groovy配置的情况下，启动也是类似的，只不过是不同的context（上下文）实现（能够适配Groovy但也同时可以识别XML的bean定义）

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

更灵活的用法是使用GenericApplicationContext类的变量结合对应的读取者（reader），例如使用XmlBeanDefinitionReader读取XML文件：

GenericApplicationContext context = **new** GenericApplicationContext();

**new** XmlBeanDefinitionReader(context).loadBeanDefinitions("services.xml", "daos.xml");

context.refresh();

或者使用GroovyBeanDefinitionReader读取Groovy文件：

GenericApplicationContext context = **new** GenericApplicationContext();

**new** GroovyBeanDefinitionReader(context).loadBeanDefinitions("services.groovy", "daos.groovy");

context.refresh();

这些读取者可以混合并且匹配在同一个ApplicationContext中，从不同的配置源中读取bean定义，如果需要的话。

你可以使用getBean去获取你的bean的实例。ApplicationContext接口还有一些其他的方法能够获取bean，但理想情况下你的应用代码绝不能使用它们。实际上，你的应用代码甚至都不应该调用getBean()，因而完全不用依赖Spring的API。例如，spring整合web框架给各种各样的web框架组件如controller和JSF控件提供了依赖注入，即允许你通过元数据（如一个autowire自动连线注解）在一个明确的bean中声明一个依赖。

* + 1. Bean总览

一个spring IOC容器管理一个或者多个bean。这些bean根据你提供给容器的配置元数据来创建，例如，在XML <bean/>定义的格式中。

在容器内，这些bean定义都被表示为BeanDefinition对象，包含了如下的元数据（以及其他信息）：

* 类的全限定名：一般来说就是bean定义的实际实现类
* Bean的行为配置元素，也就是声明bean在容器内部的行为（范围或者说域，生命周期回调等等）
* 对被本bean需要用于执行自身工作的其他bean的引用；这些引用也被称为协作者（collaborators）或者依赖（dependencies）。
* 其他一些需要设置进新创建的对象的配置信息，例如一个管理连接池的bean中要使用的连接数，或者池的大小。

这些元数据转换为构成每一个bean定义的一组属性。

| *Table 1. bean定义 The bean definition* | |
| --- | --- |
| **Property 属性** | **Explained in…​ 解释在……** |
| class类 | [Instantiating beans](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-class) 实例化bean |
| name 名字 | [Naming beans](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-beanname) 命名bean |
| scope 域 | [Bean scopes](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-scopes) bean的域 |
| constructor arguments 构造器参数 | [Dependency Injection](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-collaborators) 依赖注入 |
| properties 属性 | [Dependency Injection](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-collaborators) 依赖注入 |
| autowiring mode 自动连线模式 | [Autowiring collaborators](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-autowire) 自动连线协作者 |
| lazy-initialization mode 懒加载模式 | [Lazy-initialized beans](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-lazy-init) 懒加载bean |
| initialization method 初始化方法 | [Initialization callbacks](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-initializingbean) 初始化回调 |
| destruction method 销毁方法 | [Destruction callbacks](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-disposablebean) 销毁回调 |

除了包含有关于创建一个bean的明确信息以外，ApplicationContext的实现同时允许注册在容器外由用户创建的对象。这个可以通过方法getBeanFactory()访问ApplicationContext的bean工厂BeanFactory来实现，该方法返回BeanFactory 的实现DefaultListableBeanFactory。DefaultListableBeanFactory支持通过方法registerSingleton(..) 和registerBeanDefinition(..)。然而，大部分的应用通过定义bean的元数据就可以稳定运行。

Bean的元数据跟手动提供的单例实例应该尽可能早地注册，以便容器能够在自动装配和其他阶段正确的推导出bean之间的关联。而覆写已存在的元数据和已存在的单例在某种程度上是支持的，但是运行时注册新的bean（同时并发访问工厂factory）并未得到官方的支持，因为如此将会导致并发访问异常和/或在bean容器的不稳定状态。

* + - 1. Bean的命名

一个bean都有一个或者多个标识符。这些标识符必须在bean托管的容器中唯一。一个bean经常只有一个标识符，但如果它需要多个，其余的将被视为它的别名。

在基于XML的配置元数据中，你使用id和或name属性去指定bean的标识符。属性id允许你指定一个id。通常来说这些名字都是字符和数字组合的（'myBean', 'fooService',等等），但同样也可以包含一些特殊字符。如果你想给一个bean取别名，你同样也可以在属性name中指定，用逗号（，）、分号（；）或者空格分隔。至于历史版本信息，在spring 3.1之前的版本，属性id本定义为xsd:ID类型，限制了一些字符的使用。到了3.1，它被定义成了xsd:string类型。注意，bean的id唯一性仍然由容器强制实施，虽然已经不再是XML解析器。你不需要去给一个bean提供一个名字或者id。如果没有显示指定名字或者id，容器将会自动给bean生成一个唯一的名字。但是，如果你想通过名字引用bean，可以使用ref元素或者[Service Locator](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-servicelocator)风格去查找，但你必须提供一个名字。在使用内嵌bean和自动装配autowire协作者时可不需要提供名字。

Bean的命名约定

这个约定就是使用标准Java命名规范来命名bean，就跟命名实例的成员变量一样。也就是说，bean名以小写字母开头，然后后面是驼峰法。示例如下（没有单引号）'accountManager', 'accountService', 'userDao', 'loginController', 等等。

坚持命名bean能让你的配置更简单易懂，而且尤其是当你使用spring AOP，通过bean名来配置与一个bean集相关的切面时将大有裨益

使用在类路径classpath中的组建扫描component scan时，spring会给未命名的组件生成bean名，遵循上述的规则：基本上是使用简单类名然后首字母小写。然而在一些特殊情况下，当有多个字符且其首字母与第二个字母都是大写时，将不会出现小写转换。这个跟定义在java.beans.Introspector.decapitalize中的规则一样（也就是spring使用的）

在bean定义外给bean定义别名

在一个bean定义内，你可以给bean提供超过一个name，结合一个id属性指定的一个名字和任意数量由name属性指定的名字。这些名称可以是一个bean的等效别名，也可以在某些情形下很有用，比如允许一个应用中的每一个组件通过指定的bean名去引用一个公共依赖。

在bean定义时定义出所有需要的别名不总是够的，有时需要在bean定义以外的其他地方引入别名。这个在大系统中很常见，因为配置被分为多个子系统，每一个子系统有自己的对象定义集合。在基于XML的配置元数据中，你可以使用<alias/>元素来达到此目的。

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

在这个情况下，一个bean（在同一个容器中）被命名为fromName的同时，使用了这个别名顶以后，也可以被toName引用。

例如，一个子系统A的配置元数据中需要引用一个名叫subsystemA-dataSource的数据源dataSource。另一个子系统B的配置中需要引用一个名叫subsystemB-dataSource的数据源。组合成一个完整的应用需要这些所有的子系统，而主应用需要引用一个数据源叫myApp-dataSource。为了将这三个名字都指向同一个对象，你可以加上如下的别名配置：

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

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

现在每一个组件跟主程序通过一个唯一名字指向一个数据源，并且保证不与其他的bean定义冲突（在创建命名空间上很有效）。

基于Java的配置

如果你使用基于Java的配置，那么@Bean注解可以用来提供别名，详见[Using the @Bean annotation](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-java-bean-annotation)。

* + - 1. 初始化bean

Bean定义是创建一个或者多个对象的基本配方。容器在被询问时会查看命名bean的配方，并使用由该bean定义封装的配置元数据来创建（或获取）实际对象。

如果你使用基于XML的配置，你需要在bean元素内class属性指定初始化对象的类型。class属性其实就是BeanDefinition实例的class属性，且通常是必须的。（有关异常，请参阅 [使用实例工厂方法](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-class-instance-factory-method)和[Bean定义继承](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-child-bean-definitions)[实例化](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-class-instance-factory-method)。）您可以通过以下两种方式之一使用该class属性：

* 通常来说，指定一个bean的类允许容器通过反射调用其构造器来构造bean，这样与Java代码中使用new关键字类似
* 少数情况下，可以指定一个类中的static factory静态工厂方法，使得容器在创建对象时调用其静态工厂方法去创建对象。对象的类型可以是同一个类，也可以是完全不同的类。

*内部类名*

如果你想给一个静态内部类配置bean定义，你必须使用内部类的二进制名称。

例如，如果你有一个在com.example包下的类Foo，在这个类中有一个内部类Bar，那么class属性的值应该是……

com.example.Foo$Bar

注意，$符用于分隔内部类名与外部类名

使用构造器实例化

当你通过构造方法创建bean时，所有的普通类都可用并于spring兼容。也就是说开发的类不需要实现任何接口或者是按照一定的风格去编码。简单地指定bean的类型就足够了。然而，依赖于对于bean所使用的IOC模式，你可能需要一个默认的（空的）构造方法。

Spring IOC容器可以管理任意你需要其管理的类，不仅仅局限于真正的JavaBean。大部分的spring用户更倾向于普通的JavaBean，仅有一个默认的无参构造函数和恰当的setter和getter方法。你还可以在容器中拥有其他奇异的的非bean样式类。例如，如果你需要使用完全不符合JavaBean规范的旧连接池，spring也能正常管理。

基于XML的配置，你可以按如下方式指定一个bean的类：

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

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

有关于在构造函数中提供参数（如果需要）和在对象被构建后设置对象实例的属性的机制，详见 [Injecting Dependencies依赖注入](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-collaborators)。

以静态工厂方法实例化

在定义一个你需要用静态工厂方法来创建的bean时，使用属性class来指定有静态工厂方法的类，然后属性factory-method用来指定静态工厂方法本身的名字。你应该能够去调用这个方法（带上可选参数，稍后解释）然后返回一个活动对象，这个对象随后被视为通过构造函数创建的对象。这种ben定义的用法其中一个目的就是能够在旧版代码中调用静态static工厂。

如下bean定义就是指定了bean由调用工厂方法创建。该定义没有指定bean的返回类型，只有包含了工厂方法的类。在这个示例中，createInstance()方法必须为静态方法。

<bean id="clientService"

class="examples.ClientService"

factory-method="createInstance"/>

**public** **class** **ClientService** {

**private** **static** ClientService clientService = **new** ClientService();

**private** ClientService() {}

**public** **static** ClientService createInstance() {

**return** clientService;

}

}

有关于给工厂方法提供（可选）参数并在对象实例返回后设置其内部属性的机制，详见[依赖与配置的细节](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-properties-detailed).

使用实例工厂方法实例化

与使用[静态工厂方法](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-class-static-factory-method)实例化类似，使用实例工厂方法实例化会从容器调用现有bean的非静态方法来创建新bean。要使用这个机制，将属性class保持为空，然后在属性factory-bean中指定在当前（父辈或者先祖）容器中包含该实例化方法的对象的名字。设置factory-method属性为需要调用的工厂方法名。

*<!-- the factory bean, which contains a method called createInstance() -->*

<bean id="serviceLocator" class="examples.DefaultServiceLocator">

*<!-- inject any dependencies required by this locator bean -->*

</bean>

*<!-- the bean to be created via the factory bean -->*

<bean id="clientService"

factory-bean="serviceLocator"

factory-method="createClientServiceInstance"/>

**public** **class** **DefaultServiceLocator** {

**private** **static** ClientService clientService = **new** ClientServiceImpl();

**public** ClientService createClientServiceInstance() {

**return** clientService;

}

}

一个工厂类也可以保有多个工厂方法，如下所示：

<bean id="serviceLocator" class="examples.DefaultServiceLocator">

*<!-- inject any dependencies required by this locator bean -->*

</bean>

<bean id="clientService"

factory-bean="serviceLocator"

factory-method="createClientServiceInstance"/>

<bean id="accountService"

factory-bean="serviceLocator"

factory-method="createAccountServiceInstance"/>

**public** **class** **DefaultServiceLocator** {

**private** **static** ClientService clientService = **new** ClientServiceImpl();

**private** **static** AccountService accountService = **new** AccountServiceImpl();

**public** ClientService createClientServiceInstance() {

**return** clientService;

}

**public** AccountService createAccountServiceInstance() {

**return** accountService;

}

}

这个方法显示了工厂bean本身能够通过依赖注入（DI）来管理和配置，详见[Dependencies依赖 和configuration配置细节](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-properties-detailed)。

在spring文档中，工厂bean指代了一个通过spring容器配置的bean，能够通过实例或静态方法去创建对象。相反的，FactoryBean（注意大小写）指的是spring指定的FactoryBean。

* + 1. 依赖

一个传统的企业级应用不可能仅由单一一个对象组成（或者在spring中所说的bean）。即使是最简单的应用也会由一些对象共同协作，最终在用户端展现出连贯的程序。下一节将介绍从如何定义多个独立的bean定义，到对象协作实现同一个目标的应用程序。

* + - 1. 依赖注入

本章将涵盖spring框架对IOC理念的实现。IOC也被称为依赖注入（DI）。这是一道对象定义其自身依赖的工序，所谓依赖，也就是说那些对象内部所使用的对象是一个通过构造器或者工厂方法返回的实例。将此实例通过作为前述对象构造方法入参、工厂方法参数或者设置为内部属性。容器在创建bean的时候将为其注入那些依赖。这个过程就是基本的反转，因此称为*控制反转*（IoC），bean本身通过使用类的直接构造或诸如*服务定位器service locator*模式之类的机制来控制其依赖关系的实例化或部署。

在DI的帮助下，代码能够变得更整洁，而且当对象在具有其依赖的同时，解耦变得更有效。该对象既不需要查找它的依赖，也不需要知道依赖的类型或者是位置。如此，你的类变得更容易测试，尤其当依赖类型是接口或者抽象基类时，因为它们允许在单元测试中使用stub桩实现或者mock模拟实现。

DI主要存在有两种变体，[Constructor-based基于构造函数的依赖注入](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-constructor-injection)和[Setter-based基于setter方法的依赖注入](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-setter-injection)。

基于构造函数的依赖注入

*Constructor-based基于构造函数*的依赖注入由容器来调用具有几个参数的构造函数来完成，每一个参数都代表着一个依赖。调用一个带有参数的静态工厂方法去构造一个bean也是几乎等效的，本文档将构造函数的入参与静态工厂方法的入参视为类似。如下例子展示了一个只能通过构造函数来完成依赖注入的类。注意这个类没什么特别的，也就是一个POJO，不依赖于容器特定的接口，基类或注释。

**public** **class** **SimpleMovieLister** {

*// SimpleMovieLister有一个依赖MovieFinder*

**private** MovieFinder movieFinder;

*// 一个spring容器能够注入MovieFinder的构造函数*

**public** SimpleMovieLister(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

*// 实际使用MovieFinder的业务逻辑已被省略*

}

构造器参数解析方案

构造器参数使用参数类型来解析匹配。如果在一个bean定义的构造函数参数中没有潜在的歧义，那么构造函数参数的定义顺序也就是bean在实例化时提供给合适的构造函数的参数顺序。参考如下类：

**package** x.y;

**public** **class** **Foo** {

**public** Foo(Bar bar, Baz baz) {

*// ...*

}

}

没有潜在的歧义存在，假设Bar跟Baz没有继承上的关联关系。因此如下配置是正确的，并且你不需要在<constructor-arg/>元素中显示指定构造函数的参数索引和或类型。

<beans>

<bean id="foo" class="x.y.Foo">

<constructor-arg ref="bar"/>

<constructor-arg ref="baz"/>

</bean>

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

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

</beans>

当另一个bean需要被引用时，类型为已知的，那么匹配便可正常执行（就如上述例子一般）。当一个简单类型被使用比如<value>true</value>，spring不能决定这个value值的类型，因此不能在没有帮助的情况下匹配。如下：

**package** examples;

**public** **class** **ExampleBean** {

*// 用于计算Ultimate Answer的年数*

**private** **int** years;

*//Life生命，宇宙和万物的结果*

**private** String ultimateAnswer;

**public** ExampleBean(**int** years, String ultimateAnswer) {

this.years = years;

this.ultimateAnswer = ultimateAnswer;

}

}

构造器参数类型匹配

在上述情景中，容器可以针对简单类型使用类型匹配，如果你显式地使用type属性去指定构造函数参数的类型。例如：

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

<constructor-arg type="int" value="7500000"/>

<constructor-arg type="java.lang.String" value="42"/>

</bean>

构造函数参数下标

使用属性index去显式地指定构造函数参数的下标index。例如：

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

<constructor-arg index="0" value="7500000"/>

<constructor-arg index="1" value="42"/>

</bean>

另外，在解析多个简单类型值的冲突时，需要在构造函数有两个相同类型的入参冲突时需要指定下标。注意下标从0开始。

构造函数参数名

你同样可以指定构造函数的参数名作为值的去歧义化：

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

<constructor-arg name="years" value="7500000"/>

<constructor-arg name="ultimateAnswer" value="42"/>

</bean>

需要谨记的是，若想此功能生效，你的代码必须要在debug模式下编译，这样spring才能查询从构造函数中查询参数名。如果你不能（或不想）在debug模式下编译，你可以使用[@ConstructorProperties](https://download.oracle.com/javase/6/docs/api/java/beans/ConstructorProperties.html) JDK的注解去显式地命名你的构造函数参数。示例类如下：

**package** examples;

**public** **class** **ExampleBean** {

*// 忽略成员变量*

@ConstructorProperties({"years", "ultimateAnswer"})

**public** ExampleBean(**int** years, String ultimateAnswer) {

this.years = years;

this.ultimateAnswer = ultimateAnswer;

}

}

基于setter方法的依赖注入

基于setter方法的依赖注入由容器在创建bean时调用无参构造方法或者无参静态静态工厂方法后，再调用setter方法去实现。

如下例子展示了依赖注入只能使用纯净的setter方法。这个类是传统的Java。它是一个POJO，在容器内不依赖任何接口，基类或者注解。

**public** **class** **SimpleMovieLister** {

*// SimpleMovieLister依赖于MovieFinder*

**private** MovieFinder movieFinder;

*// 一个setter方法让spring注入a MovieFinder*

**public** **void** setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

*// 省略使用MovieFinder的业务逻辑代码*

}

ApplicationContext对其管理的bean支持基于构造函数注入和基于setter方法注入。同时它也支持在使用构造函数注入后的bean使用基于setter方法注入。按照BeanDefinition的格式配置依赖，同时结合PropertyEditor实例去将属性从一种格式转化为另一种。然而大部分的spring用户不直接使用这些类（编程上），使用XML类的bean定义，已作注解的组件（例如已使用@Component作注解），或者在基于Java配置@Configuration注解的类中作@Bean注解的方法。这些资源之后便在内部被转换成BeanDefinition实例且用于加载整个的spring IOC容器实例。

基于构造函数注入还是setter方法注入?

由于你可能会混淆构造方法注入和setter方法注入，因此将构造函数用于强制依赖项和setter方法或可选依赖项的配置方法是一个很好的经验法则。请注意，在setter方法上使用 [@Required](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-required-annotation)注解能够使得该属性成为一个必须依赖

Spring团队倡导使用构造函数注入因为它允许将应用程序组件实现为不可变对象 ，保证必须的依赖为非空。此外，构造函数注入的组件始终以完全初始化的状态返回到客户端（调用端）代码。旁注：大量参数的构造函数是坏代码风格，意味着这个类可能有太多的责任，且应该恰当考虑分离的情况下重构。.

Setter方法注入基本上只能用在可选依赖上，这些可选依赖能够在类这种被分配合理的默认值.。否则必须在使用依赖的每一处执行非空检查。Setter方法注入的一个好处是setter方法使得这些类的对象更易控制，可以稍后再执行重配置或重注入。因此，通过[JMX MBeans](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/integration.html#jmx)管理是使用setter注入的一个优秀实现。

使用依赖注入使得一个特殊的类有了更多的含义。有时候，在处理你没有源代码的第三方类的时候，依赖注入就是为你量身打造的。例如，一个第三方类没有暴露出任何一个setter方法时，构造函数注入就是唯一可用的注入形式。

依赖解析过程

容器执行bean依赖解析为如下过程：

* ApplicationContext通过描述所有bean的配置元数据创建和初始化。配置元数据可以通过XML、Java代码或注解去指定。
* 对于每一个bean，它的依赖将以属性，构造函数参数，或者静态工厂方法参数（如果你使用的不是普通的构造函数）的形式表示。这些依赖会被提供给这个bean，*当bean真实地被创建后。*
* 每一个属性或者构造函数参数都是需要设置的值真实存在的定义，或者是对容器中另一个bean的引用。
* 每一个属性或者是构造函数参数作为值会从它被指定的格式被转换成属性或者构造函数参数的实际类型。默认情况下，spring可以将提供为字符串格式的值转换为所有的内置类型，比如int、long、String、boolean等等。

Spring容器会在其被创建时校验每一个bean的配置。然而，bean属性不会被设置除非bean已经被真正的创建出来。单例且被设置为预实例化（默认）的bean会在容器创建后被创建。Bean的域（scopes）定义在[Bean scopes](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-scopes)中。否则，bean只有在它被需要的时候创建。一个bean的创建会引发潜在一系列的bean的创建，也就是这个bean的依赖还有它依赖的依赖（等等）会被创建和分配。请注意，在这些依赖中的解析依赖出错会稍后出现，比如在被影响ben的第一次创建的时候。

循环依赖

如果你主要使用构造函数注入，那么有可能产生出无法解析的循环依赖的场景。

例如：类A需要通过构造函数注入类型B的实例，类B需要通过构造函数注入类A的实例。如果你配置的bean需要互相注入，Spring IoC容器在运行时检测到循环引用，会抛出BeanCurrentlyInCreationException异常。

一个可能的解决办法是更改源代码使得配置通过setter方法而不是构造函数。或者避免使用构造函数注入而只使用setter方法注入。也就是说，虽然不推荐，但你依然可以通过构造函数注入配置出循环依赖。

不像*正常情况*（无循环依赖），一个bean A跟bean B之间的循环依赖强制其中一个bean必须在另一个bean完全初始化之前注入（一个典型的鸡跟蛋问题）

通常情况下你可以相信spring是正确的。它能够检测出配置的问题，例如在容器加载时引用一个不存在的bean和循环依赖。在bean实实在在地被创建后, spring会尽量晚地设置属性和解析依赖。这也就意味着，在你请求一个有问题的bean或者其内部某个依赖有问题的bean的时候，已正确加载的spring容器能够在稍后生成一个异常。例如，一个bean抛出一个丢失或无效属性的异常。这潜在地延迟了一些配置问题的可见性，也就是为什么ApplicationContext的实现默认预实例化bean为单例。以实际需要之前创建这些bean的一些前期时间和内存为代价，你会在ApplicationContext创建时发现配置问题，而不是更晚。你可以通过覆写这个默认行为使得单例bean改为懒加载，而不是预实例化。

如果不存在循环依赖，当一个或多个协作bean被注入进依赖bean时，在注入前，每一个协作bean会先完全组装好。这也就意味着如果bean A依赖于bean B，那么spring IOC容器会在调用bean A的setter方法之前，先完全组装好bean B。换句话说，如果该bean不是预实例化的单例bean，那么在它被实例化后，它的依赖会被设定，而且相关联的生命周期方法（比如[configured init method配置初始化方法](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-initializingbean)或者[InitializingBean callback method初始化bean回调方法](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-initializingbean)）会被调用。

依赖注入示例

如下例子便是使用基于XML的配置信息使用基于setter方法的注入。一小部分的spring XML配置文件便可指定一些bean定义：

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

*<!-- setter注入使用内置ref元素 -->*

<property name="beanOne">

<ref bean="anotherExampleBean"/>

</property>

*<!-- setter注入使用ref属性 -->*

<property name="beanTwo" ref="yetAnotherBean"/>

<property name="integerProperty" value="1"/>

</bean>

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

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

**public** **class** **ExampleBean** {

**private** AnotherBean beanOne;

**private** YetAnotherBean beanTwo;

**private** **int** i;

**public** **void** setBeanOne(AnotherBean beanOne) {

this.beanOne = beanOne;

}

**public** **void** setBeanTwo(YetAnotherBean beanTwo) {

this.beanTwo = beanTwo;

}

**public** **void** setIntegerProperty(**int** i) {

this.i = i;

}

}

在前述例子中，setter声明用于匹配XML文件中指定的属性。如下例子使用基于构造函数的依赖注入：

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

*<!-- 构造函数注入使用内置ref元素 -->*

<constructor-arg>

<ref bean="anotherExampleBean"/>

</constructor-arg>

*<!-- 构造函数注入使用ref属性 -->*

<constructor-arg ref="yetAnotherBean"/>

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

</bean>

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

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

**public** **class** **ExampleBean** {

**private** AnotherBean beanOne;

**private** YetAnotherBean beanTwo;

**private** **int** i;

**public** ExampleBean(AnotherBean anotherBean, YetAnotherBean yetAnotherBean, **int** i) {

this.beanOne = anotherBean;

this.beanTwo = yetAnotherBean;

this.i = i;

}

}

在bean定义中指定的构造函数参数将作为ExampleBean构造器的入参。

现在考虑一下该示例的一个变种，相较于使用一个构造函数，spring被指定调用一个static静态工厂方法来返回一个对象实例。

<bean id="exampleBean" class="examples.ExampleBean" factory-method="createInstance">

<constructor-arg ref="anotherExampleBean"/>

<constructor-arg ref="yetAnotherBean"/>

<constructor-arg value="1"/>

</bean>

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

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

**public** **class** **ExampleBean** {

*// 一个私有的构造函数*

**private** ExampleBean(...) {

...

}

*// 一个静态工厂方法; 参数可以是被返回bean的依赖*

*// 忽略这些参数是怎么使用的。*

**public** **static** ExampleBean createInstance (AnotherBean anotherBean, YetAnotherBean yetAnotherBean, **int** i) {

ExampleBean eb = **new** ExampleBean (...);

*// 一些其他的操作...*

**return** eb;

}

}

静态工厂方法参数通过<constructor-arg/>元素传递，和构造函数的使用方法一模一样。工厂方法的返回值类型不需要与包含该静态工厂方法的类保持一致，尽管在上述例子中是一致的。一个实例工厂方法（非静态）的使用与上述基本一致（排除使用factory-bean属性的情况），因此详细细节就不在此讨论。

* + - 1. 依赖与配置细节

上一节提到，你可以定义bean的属性和构造函数参数引用其他的受管理bean（协作者），或在内联中定义的值。因此spring基于XML的配置在<property/>和<constructor-arg/>支持子元素类型。

直接值（基础类型，String等等）

<property/>元素中的value属性以易读字符串的形式指定一个属性或者构造函数参数。Spring的[conversion service](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#core-convert-ConversionService-API)转换服务用于将这些值从String转化成属性或者参数的真实类型。

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

*<!-- 最终调用setDriverClassName(String)方法 -->*

<property name="driverClassName" value="com.mysql.jdbc.Driver"/>

<property name="url" value="jdbc:mysql://localhost:3306/mydb"/>

<property name="username" value="root"/>

<property name="password" value="masterkaoli"/>

</bean>

下述例子则是使用命名空间[p-namespace](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-p-namespace)，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>

前述XML更加简洁，但是拼写错误将会在运行时发现而不是设计时，除非你使用的IDE是[IntelliJ IDEA](http://www.jetbrains.com/idea/) 或[Spring Tool Suite](https://spring.io/tools/sts) (STS)，在创建bean定义时支持属性的自动装填。强烈推荐使用这类IDE助手。

你可以像这样配置一个java.util.Properties的实例：

<bean id="mappings"

class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

*<!-- 像java.util.Properties一样编写内容 -->*

<property name="properties">

<value>

jdbc.driver.className=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/mydb

</value>

</property>

</bean>

Spring容器通过使用JavaBeans的PropertyEditor机制将<value/>元素内的内容转换成java.util.Properties实例。这是一个很棒的捷径，也是Spring团队在value属性样式上支持<value/>使用嵌套元素的少数几个地方之一。

idref元素

元素idref只是单纯的一个防错方法，用于传递在容器中另一个bean的id（字符串值，不是引用）给<constructor-arg/>或<property/>元素。

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

<bean id="theClientBean" class="...">

<property name="targetName">

<idref bean="theTargetBean"/>

</property>

</bean>

如上bean定义片段与如下片段等效（运行时）：

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

<bean id="client" class="...">

<property name="targetName" value="theTargetBean"/>

</bean>

第一种格式较优于第二种，因为使用idref标签允许容器在*部署时*校验该引用，即对应名字的bean是否真实存在。在第二种情况中，将不会校验传递给client bean内targetName属性的值。拼写错误只会在client bean实例化时被发现（可能会是重大错误）。如果client bean是一个原型[prototype](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-scopes) bean，那么这个拼写错误及其导致的异常将只可能在容器部署后很长时间被发现。

在idref元素中local属性将不在4.0 beans 的xsd中受支持，因为它不再提供一个普通bean的引用值。升级至4.0 schema时只需要简单更换已存在的idref local引用到idref bean。

一个共同点（至少早于spring 2.0版本）是在ProxyFactoryBeand的bean定义中[AOP interceptors拦截器](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#aop-pfb-1)的配置信息，也是idref元素携带值的位置。当你指定拦截器名称时使用<idref/>元素能够防止拼写错误的拦截器ID。

引用其他bean（协作者）

元素ref是<constructor-arg/>或<property/>定义元素中的最后一个元素。这里你可以设置bean内指定属性引用另一个由容器管理的bean（协作者）。被引用的bean是被设置属性bean的依赖，它在属性被设置前按照需求初始化。如果协作者是一个单例bean，它将由容器初始化。所有的引用最终都是对另一个对象的引用。划定范围和有效性取决于是否通过bean，local,或parent属性指定其他对象的ID /名称。

通过bean属性的<ref/>标签来指定目标bean是最常见的方式，同时允许创建在同一个容器或者父容器中任意一个bean的引用，忽略是否在同一个XML文件中。bean属性的值可以是目标bean的id也可以是目标bean多个name属性中的一个。

<ref bean="someBean"/>

通过parent属性创建引用当前容器的父容器中的bean。parent属性的值可以是目标bean的id也可以是目标bean多个name属性中的一个，而且目标bean必须存在在当前容器的父容器中。使用这种形式的引用主要在当你有层次结构的容器的情况下，你想在父容器中已存在的bean上包裹一层代理且使用同一名称。

*<!-- 在父容器内 -->*

<bean id="accountService" class="com.foo.SimpleAccountService">

*<!-- 在此处插入需要的依赖 -->*

</bean>

*<!-- 在子（后代）容器中 -->*

<bean id="accountService" <!-- bean名与父bean一致 -->

class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="target">

<ref parent="accountService"/> *<!-- 注意我们如何指向一个父bean -->*

</property>

*<!-- 在此处插入其余需要的依赖配置 -->*

</bean>

元素ref中的local属性在spring 4.0 xsd后便不再支持，因为它不再提供一个普通bean的引用值。只需将现有ref local引用更改为ref bean升级到4.0架构。

内部bean

一个<bean/>元素在<property/>或<constructor-arg/>元素内部即所谓的内部bean

<bean id="outer" class="...">

*<!-- 相较于使用目标bean的引用，直接在内联中定义bean -->*

<property name="target">

<bean class="com.example.Person"> *<!-- 这个就是内部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id或者名字，即使指定，容器也不会将该值作为标识符。容器同样会在创建时忽略scope标记：因为内部bean总是匿名且与外部bean一起创建。除了将内部bean注入封闭bean或独立地访问它们之外，不可能将内部bean注入协作bean。

作为一个极端情况，可以从自定义域中接受销毁回调，例如，对于在一个单例bean内部的request-scoped域的内部bean：内部bean实例的创建会绑定在其包含bean上，但销毁回调允许它参与进request域的生命周期中。这不是一个常见的情形，内部bean一般情况下都与它们的包含bean共享同一个域。

集合

在<list/>，<set/>，<map/>和<props/>这些元素中，你可以分别设置Java集合类型List，Set，Map和Properties的属性和参数。

<bean id="moreComplexObject" class="example.ComplexObject">

*<!-- 最终调用setAdminEmails(java.util.Properties) -->*

<property name="adminEmails">

<props>

<prop key="administrator">administrator@example.org</prop>

<prop key="support">support@example.org</prop>

<prop key="development">development@example.org</prop>

</props>

</property>

*<!-- 最终调用setSomeList(java.util.List) -->*

<property name="someList">

<list>

<value>a list element followed by a reference</value>

<ref bean="myDataSource" />

</list>

</property>

*<!-- 最终调用setSomeMap(java.util.Map) -->*

<property name="someMap">

<map>

<entry key="an entry" value="just some string"/>

<entry key ="a ref" value-ref="myDataSource"/>

</map>

</property>

*<!-- 最终调用setSomeSet(java.util.Set) -->*

<property name="someSet">

<set>

<value>just some string</value>

<ref bean="myDataSource" />

</set>

</property>

</bean>

那些map的key或者value的值，或者set的值，可以同样使用如下元素

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

集合合并

Spring容器同样支持集合的合并。应用开发者可以定义一个父集合<list/>，<map/>，<set/>或者<props/>元素，然后可以以子集合<list/>，<map/>，<set/>或者<props/>元素继承和覆盖父集合内的值。也就是说，通过子集元素覆盖父集中指定的值，子集合内的值就是合并子集和父集的结果。

本节讨论的合并是父-子bean机制，不了解父与子bean定义的可以详读[*相关章节*](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-child-bean-definitions)后继续。

如下例子表示集合合并：

<beans>

<bean id="parent" abstract="true" class="example.ComplexObject">

<property name="adminEmails">

<props>

<prop key="administrator">administrator@example.com</prop>

<prop key="support">support@example.com</prop>

</props>

</property>

</bean>

<bean id="child" parent="parent">

<property name="adminEmails">

*<!-- 在子集定义中合并 -->*

<props merge="true">

<prop key="sales">sales@example.com</prop>

<prop key="support">support@example.co.uk</prop>

</props>

</property>

</bean>

<beans>

注意使用child bean定义adminEmails属性<props/>元素内的merge=true属性。当child bean被容器解析和实例化后，实例内会有adminEmails Properties集合包含子bean adminEmails集合和父bean adminEmails集合合并的结果。

administrator=administrator@example.com

sales=sales@example.com

support=support@example.co.uk

子Properties集合中的值继承了父集合<props/>元素内的所有值，子集合中support的值覆盖了父集合中的相应值。

<list/>，<map/>，和<set/>集合的合并动作与上类似。尤其在<list/>的情况下，保持与list类型相同的语义，即有序。父集的值出现在子集之前。在Map，Set，和Properties集合类型的情况下，不存在顺序。因此容器内部使用的Map，Set，和Properties实现类也都包含无序语义。

集合合并的局限性

你不能合并不同类型的集合（比如List和Map），而且如果你尝试如此做时便会抛出一个准确的异常。属性merge必须在低层级的继承子定义上指定，在父集定义上指定merge是多余的，不会导致任何的合并。

强类型集合

随着Java 5引入的泛型，你可以使用强类型集合。也就是说，有可能声明一个集合类型只容纳String类型的元素（例如）。如果你是用spring依赖注入一个强类型的集合给bean，spring的类型转换支持将你的强类型集合内的元素在加入集合前转换成合适的类型。

**public** **class** **Foo** {

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

**public** **void** setAccounts(Map<String, Float> accounts) {

this.accounts = accounts;

}

}

<beans>

<bean id="foo" class="x.y.Foo">

<property name="accounts">

<map>

<entry key="one" value="9.99"/>

<entry key="two" value="2.75"/>

<entry key="six" value="3.99"/>

</map>

</property>

</bean>

</beans>

当foo bean的accounts属性准备好注入时，强类型Map<String, Float>的元素类型泛型信息能够被反射。因此spring的类型转换架构将各种元素值识别为Float类型，字符值9.99，2.75，和3.99被转化为真实的Float类型。

Null或空字符值

Spring视属性的空参数为空字符Strings。下述XML配置片段设置email属性为空String值(“”)。

<bean class="ExampleBean">

<property name="email" value=""/>

</bean>

前述例子与如下Java代码等效：

exampleBean.setEmail("");

<null/>标签持有null值。例如：

<bean class="ExampleBean">

<property name="email">

<null/>

</property>

</bean>

上述配置与如下Java代码等效：

exampleBean.setEmail(null);

XML捷径：命名空间p-namespace

p-namespace允许你使用bean元素的属性去描述你的属性值和/或协作bean，而不是使用内嵌<property/>元素。

Spring支持使用[命名空间namespaces](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#xsd-schemas)进行基于XML schema文件定义的配置扩展。在本章中讨论的beans配置格式已在XML schema文件中定义。然而，p-namespace命名空间没有定义在XSD文件中，仅仅存在于spring核心内。

以下示例显示两个不同的XML能解析出同一个结果：第一个使用标准的XML格式而第二个使用了p-namespace命名空间。

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

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans

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

<bean name="classic" class="com.example.ExampleBean">

<property name="email" value="foo@bar.com"/>

</bean>

<bean name="p-namespace" class="com.example.ExampleBean"

p:email="foo@bar.com"/>

</beans>

示例中显示有一个p-namespace的属性叫email存在于bean定义中。这告诉了spring区包含一个属性声明。如前面提到的，p-namespace命名空间没有规范定义，所以你可以设置标签名称为bean属性名称。

如下示例是两个bean定义同时对另一个bean有引用：

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

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans

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

<bean name="john-classic" class="com.example.Person">

<property name="name" value="John Doe"/>

<property name="spouse" ref="jane"/>

</bean>

<bean name="john-modern"

class="com.example.Person"

p:name="John Doe"

p:spouse-ref="jane"/>

<bean name="jane" class="com.example.Person">

<property name="name" value="Jane Doe"/>

</bean>

</beans>

如你所见，这个例子包含了不仅仅是使用p-namespace命名空间的一个属性值，同样使用了一个特殊格式去声明属性引用。鉴于第一个bean定义使用<property name="spouse" ref="jane"/>去创建bean john去引用bean jane，第二个bean定义则是使用p:spouse-ref="jane"作为标签内属性去完成同一件事。在这个示例中，spouse是属性名，-ref指代了这不是一个普通值而是一个对另一个bean的引用。

命名空间p-namespace不像标准XML格式那样灵活。例如，声明属性引用会会以ref结尾的属性冲突，而标准XML格式则不会出现。我们建议使用时谨慎选择，与团队成员沟通，避免在编写XML格式文件时同时使用这三种方式。

XML捷径：命名空间c-namespace

与 [XML捷径：命名空间p-namespace](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-p-namespace)类似，c-namespace，spring 3.1新引入，允许在配置构造函数参数时使用内联属性，而不是使用内嵌constructor-arg元素。

让我们看看使用c: 命名空间来[基于构造函数的依赖注入](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-constructor-injection)：

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

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:c="http://www.springframework.org/schema/c"

xsi:schemaLocation="http://www.springframework.org/schema/beans

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

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

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

*<!-- 传统声明 -->*

<bean id="foo" class="x.y.Foo">

<constructor-arg ref="bar"/>

<constructor-arg ref="baz"/>

<constructor-arg value="foo@bar.com"/>

</bean>

*<!-- c-namespace 命名空间声明 -->*

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

</beans>

命名空间c: 使用与p: 相同的格式（以-ref结尾为bean引用）通过参数名来设置构造函数参数。同样，它需要声明，即使它没有在XSD规范中定义（但它存在于Spring核心内）。

少数情况下当构造函数参数名不可用（一般来说时字节码编译时没有调试信息），一种办法是可以使用参数索引

*<!-- c-namespace索引声明 -->*

<bean id="foo" class="x.y.Foo" c:\_0-ref="bar" c:\_1-ref="baz"/>

基于XML的语法，索引标记必须以\_（下划线）作为前导，因为XML属性名不能以数字为开头（即使部分IDE允许）。

事实上，构造器解析[机制](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-ctor-arguments-resolution)在匹配参数上十分高效，因此除非十分需要，否则我们建议在整个配置中使用名称表示法。

复合属性名称

当你设置bean属性时你可以使用复合或内嵌属性名，只要路径中除了最后一个属性名以外其余组件均不为null。思考如下bean定义。

<bean id="foo" class="foo.Bar">

<property name="fred.bob.sammy" value="123" />

</bean>

foo bean有fred属性，而其有一个bob属性，而bob属性又有一个sammy属性，然后最后的sammy属性被设为值123。若要使此合法，foo的属性fred，fred的属性bob必须要在bean构建后不可为空，否则NullPointerException异常将会抛出。

* + - 1. 使用depends-on属性

如果一个bean是另一个bean的依赖，通常意味着这个bean是另一个的属性。正常来说你可以通过[<ref/> 元素](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-ref-element)在基于XML的配置文件中达到此目的。然而，有些时候这些bean之间的依赖不是这么直接的。例如，一个类中的静态初始化工作需要触发，例如数据库驱动的注册。那么属性depends-on能够显示地强制一个或多个bean在使用了该元素的bean初始化之前初始化。如下示例便是使用了depends-on属性在一个bean中声明了一个依赖：

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

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

需要声明对多个bean的依赖，在depends-on属性中提供bean名的列表，以逗号，空格和分号分隔：

<bean id="beanOne" class="ExampleBean" depends-on="manager,accountDao">

<property name="manager" ref="manager" />

</bean>

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

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

Bean定义中的属性depends-on可以同时指定初始化依赖关系，和相关的销毁时依赖（仅在[单例](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-singleton)bean的情况下）。定义了depends-on关系的依赖bean先销毁，而后才是被依赖bean的销毁。因此depends-on能同样控制关闭顺序。

* + - 1. 懒加载bean

默认情况下，作为初始化过程之一，ApplicationContext的实现会主动创建和配置所有的[单例](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-singleton)bean。通常来说，预实例化是需要的，因为配置或者环境错误能够被马上发现，而不是几小时甚至几天后。当这个行为不被需要，你可以通过标记bean定义为lazy-initialized懒加载去防止单例bean的预实例化。一个懒加载的bean告诉IOC容器当需要时再创建bean实例，而不是在启动时。

在XML中，这个行为由<bean/>元素中的lazy-init属性控制；例如：

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

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

当上面的配置在被ApplicationContext消费时，叫lazy的bean不会在ApplicationContext启动时马上预实例化，而not.lazy的bean则会急切地预实例化。

但是，当一个懒加载bean是另一个单例非懒加载bean的依赖时，ApplicationContext会在启动时创建这个懒加载bean，因为它必须满足单例bean的依赖。懒加载bean需要被注入非懒加载bean中。

同样你可以通过<beans/>元素的属性default-lazy-init来在容器级别上控制懒加载；例如：

<beans default-lazy-init="true">

*<!-- 将没有bean预实例化... -->*

</beans>

* + - 1. 自动装配协作者

Spring容器可以在协助bean之间自动装配它们的关系。你可以允许spring通过检查ApplicationContext的内容来给你的bean自动解析协作者（其他bean）。自动装配有如下优势：

* 自动装配能够有效的降低指定属性或构造函数参数的需要（其余机制如[在本章其他位置讨论](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-child-bean-definitions)的bean模板也是出于此考虑）。
* 自动装配能够随着你的对象更新。例如，如果你想在一个类中新增依赖，那么这个依赖不需你去更改配置便能自动满足。因此自动装配在开发过程中变得十分有用，在不需要显示关联装配后代码变得更加稳定。

当使用基于XML配置时，你可以在bean定义中使用<bean/>元素内的autowire属性来开启自动装配模式。自动装配模式有四种模式。你可以为每一个bean指定一种模式。

| **Mode** | **Explanation** |
| --- | --- |
| no | (默认)不自动装配。Bean引用一定要指定ref元素。不推荐在较大应用中更改默认配置，因为显式指定协作者更清晰易控。在某种程度上，它将系统架构文档化了。 |
| byName | 通过属性名自动装配。Spring寻找与属性名同名的bean来装配。例如，如果一个bean定义设置为通过名字装配，而且它包含一个master属性(也就是说它有一个*setMaster(..)方法*)，spring寻找名字叫 master的bean，然后将其设置入该属性。 |
| byType | 允许属性自动装配当容器中有且仅有一个该属性类型的bean时。如果超过一个，一个错误异常将会抛出，指明或许你不应该使用byType来自动装配。如果没有适配的bean，则无事发生，属性将不会被设置。 |
| constructor | *与byType*类似，但是是给构造函数提供参数。如果容器中没有一个对应构造函数类型的bean，一个错误会被抛出。 |

在 *byType* 或者*constructor* 自动装配模式下，你可以装配数组和类型集合。在这种情况下，将提供容器中与预期类型匹配的所有自动装配候选，以满足依赖关系。你可以自动装配强类型Maps如果它的预期类型是String。自动装配的maps值将由匹配预期类型的所有bean实例组成，maps键将包含相应的bean名称。

你可以结合依赖检查和自动装配，这个会在自动装配结束后执行。

自动装配的劣势和局限性

* 在属性和构造函数参数的显示指定总是会覆盖自动装配。你不能自动装配所谓简单的属性如基本类型，String，Classes（还有这些简单属性的数组）。这个限制是设计决定。
* 自动装配没有显式指定准确。即使，在上述表格中所说，spring小心避免在有歧义的情况下猜测以导致不可预料的结果，但你的spring管理对象之间的关系将不再清晰明了。
* 装配信息可能对生成spring容器文档的工具不适用。
* 容器中的多个bean定义可以匹配setter方法或构造函数参数指定的类型以进行自动装配。对于数组，集合或者Maps，这不一定是个问题。但是，对于期望单个值的依赖关系，这种模糊性不是任意解决的。如果没有可用的唯一bean定义，则抛出异常。

在后一种情况下，你有这些选择：

* 放弃自动装配，使用显式指定
* 如下一节所说，将bean定义中的autowire-candidate设置为false来避免自动装配bean。
* 将<bean/>元素内的primary属性设置为true，指明一个bean定义为首要候选者。
* 使用基于注解的配置来实现更细粒度的控制，如[基于注解的容器配置](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-annotation-config)中描述。

从自动装配中排除bean

在每个bean中你可以将其从自动装配中排除。在spring XML格式中，设置<bean/>元素中的autowire-candidate属性为false；容器即可使得指定的bean在自动装配内失效（包括注解风格配置[@Autowired](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-autowired-annotation)）。

属性autowire-candidate仅设计用于影响基于类型的自动装配。它不会影响名称的显式引用，因为即使bean没有被标记为自动装配候选它还是能被解析。因此，如果名称匹配，按名称自动装配将注入bean。

你还可以根据与bean名称的模式匹配来限制自动装配候选者。顶层<beans/>元素允许在它的default-autowire-candidates属性中接受一个或多个匹配样式。例如，为了限制自动装配候选者bean的名字以“Repository”结尾，那么就设置该属性值为“\*Repository”。若要提供更多样式，定义一个以逗号分隔的列表。bean定义属性的显式值true或false为bean定义autowire-candidate属性的显式值 始终优先，对于此类bean，模式匹配规则不适用。

这些技术在你对某些bean不想使用自动装配来注入时是十分有用的。这不是说一个排除在外的bean本身不能使用自动装配进行配置。而是它不能作为其他自动装配bean的候选者。

* + - 1. 方法注入

在大部分的应用场景下，大部分在容器内的bean是[单例](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-singleton)的。当一个单例bean需要与另一个协作时，或一个非单例bean需要与另一个非单例bean协作时，你一般会将依赖处理为定义其中一个bean是另外一个的属性。此时就会有一个问题，这些bean的声明周期不同。假设单例bean A需要使用非单例（原型）bean B，或许在A的每一个方法调用时。容器只会创建单例bean A一次，因此也就只有一次机会去设置属性。容器不可能在每次bean A需要的时候都提供一个新的实例bean B。

一个解决方法是放弃一部分的控制反转。你可以通过实现ApplicationContextAware接口来 [让bean A接入容器](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-aware)，然后[创建一个getBean("B")向容器调用](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-client)来在每次bean A需要的时候请求bean B（一般是新的）。如下是这种方法的示例：

*// 使用有状态命令风格去执行某些处理的类*

**package** fiona.apple;

*// 导入Spring-API*

**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) {

*// 获取某个任务的示例*

Command command = createCommand();

*// 设置任务状态 (希望是全新的)*

command.setState(commandState);

**return** command.execute();

}

**protected** Command createCommand() {

*// 通知Spring API依赖!*

**return** this.applicationContext.getBean("command", Command.class);

}

**public** **void** setApplicationContext(

ApplicationContext applicationContext) **throws** BeansException {

this.applicationContext = applicationContext;

}

}

上述例子并不可取，因为业务代码与spring框架耦合。方法注入，某种更高级的spring IOC容器特性，允许在这种情况下保持更干净的风格。

你可以从这个[博客入口](https://spring.io/blog/2004/08/06/method-injection/)获取更多关于方法注入的动机。

查找方法注入

查找方法注入是容器覆盖容器管理bean的方法的能力，以返回容器中另一个命名bean的查找结果。查找一般设计原型bean，如上一节所描述的场景。Spring框架通过CGLIB库字节码生成功能来动态生成一个覆盖该方法的的子类去达到方法注入的目的。

* 若使动态子类正常工作，spring bean容器要子类化的目标不能为final，且需要被覆盖的方法也不能是final。
* 对具有abstract方法的类进行单元测试需要您自己对类进行子类化并提供该abstract方法的存根实现
* 需要明确类的组件扫描同样需要明确的方法。
* 一个更关键的限制是查找方法不适用于工厂方法，尤其不适合在配置类中的@Bean方法，因为容器不负责在这种情况下创建实例因而不能在运行时创建运行时生成子类的bean。

看上述CommandManager类的代码片段，你会看到spring容器将动态覆盖createCommand()方法的实现。你的CommandManager类将不会有任何的spring依赖，如如下重写例子所示：

**package** fiona.apple;

*// 没有Spring导入!*

**public** **abstract** **class** **CommandManager** {

**public** Object process(Object commandState) {

*// 获取Command接口的新实例*

Command command = createCommand();

*// 设置Command实例状态（希望为全新）*

command.setState(commandState);

**return** command.execute();

}

*// okay... 但是这个方法的实现在哪里?*

**protected** **abstract** Command createCommand();

}

包含需要注入方法的客户类（在这里是CommandManager），需要注入的方法需要按照以下格式标识：

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

如果方法是abstract抽象的，动态生成子类会实现这个方法。否则，动态生成子类会覆盖定义在原类中的方法。例如：

*<!-- 部署为prototype 原型(非单例)有状态的bean -->*

<bean id="myCommand" class="fiona.apple.AsyncCommand" scope="prototype">

*<!-- 此处注入所需依赖 -->*

</bean>

*<!-- commandProcessor使用statefulCommandHelper -->*

<bean id="commandManager" class="fiona.apple.CommandManager">

<lookup-method name="createCommand" bean="myCommand"/>

</bean>

标识为*commandManager*的bean在当其需要一个新的*myCommand*的bean实例时，调用其方法createCommand()。你必须注意将myCommand bean部署为原型prototype，如果确实是需要如此的话。如果它是一个单例[singleton](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-singleton)，那么每次调用都会返回同一个myCommand 实例bean。

相对应的，在基于注解的组件模型里，你可以通过@Lookup注解声明一个查找方法：

**public** **abstract** **class** **CommandManager** {

**public** Object process(Object commandState) {

Command command = createCommand();

command.setState(commandState);

**return** command.execute();

}

@Lookup("myCommand")

**protected** **abstract** Command createCommand();

}

或者，更符合语言习惯地，你可以依赖目标bean根据lookup方法的声明返回类型进行解析。

**public** **abstract** **class** **CommandManager** {

**public** Object process(Object commandState) {

MyCommand command = createCommand();

command.setState(commandState);

**return** command.execute();

}

@Lookup

**protected** **abstract** MyCommand createCommand();

}

请注意，你通常会声明这种带注释的查找方法带有具体实现，为了保证spring组件扫描能够兼容，因为抽象类会被组件扫描默认忽略。这个不会发生在显式注册或显式导入bean的类中。

另一个访问不同域的目标bean是ObjectFactory / Provider注入点。查看[Scoped不同域beans作为依赖](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-other-injection).

感兴趣的读者可以查看在在org.springframework.beans.factory.config包中的ServiceLocatorFactoryBean并使用。

任意方法替换

与查找方法注入相比，一种不太有用的方法注入形式是能够使用另一个方法实现替换托管bean中的任意方法。用户可以跳过如下章节除非这个功能真的需要使用。

基于XML的配置中，对于已部署的bean，你可以使用replaced-method元素去将已存在的方法实现替换成另一个。思考如下类，有一个计算值的方法，也就是我们想覆盖的：

**public** **class** **MyValueCalculator** {

**public** String computeValue(String input) {

*// 业务代码...*

}

*// 其他方法...*

}

一个实现了org.springframework.beans.factory.support.MethodReplacer接口的类提供了新方法定义。

*/\*\**

*\* 用于覆盖已在MyValueCalculator中存在的computeValue(String)实现*

*\**

*\*/*

**public** **class** **ReplacementComputeValue** **implements** MethodReplacer {

**public** Object reimplement(Object o, Method m, Object**[]** args) **throws** Throwable {

*// get the input value, work with it, and return a computed result*

String input = (String) args[0];

...

return ...;

}

}

部署原始类并指定方法覆盖的bean定义如下所示：

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

*<!-- 任意方法替代 -->*

<replaced-method name="computeValue" replacer="replacementComputeValue">

<arg-type>String</arg-type>

</replaced-method>

</bean>

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

你可以使用一个或多个包含在<replaced-method/>元素内的<arg-type/>元素去指定需要覆盖方法的参数类型。只有方法是重载且在类中有多个变量时参数类型才是必须的。方便起见，参数类型是字符型的可以是类全限定名的一部分，也就是说如下类型都可以匹配java.lang.String：

java.lang.String

String

Str

因为参数数量足够区分每个可能的选择，允许你键入最少的字符串就可以匹配一个参数类型，这样可以节省很多键入时间。

* + 1. Bean域

当你创建一个bean定义时，你也就创建了一个可以创建出类的真实实例的配方。Bean定义是菜单这个概念很重要，因为这意味着，像一个类一样，你可以从一个配方创建出多个对象实例。

你不仅仅可以控制即将插入对象（由bean定义创建的）的多样依赖和配置值，同样可以控制对象的所在域。这个方法强大且灵活，你可以选择通过配置创建的对象的域，而不是在Java类级别上控制对象域。Bean能够被定义为以多个域其中的一种去部署：即开箱即用（out of the box），spring框架支持六种域，只有当你使用接入web（web-aware）的ApplicationContext，其中四种才可用。

如下域支持开箱即用。你还可以创建自定义域[custom scope](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-custom)。

Bean域

| **域** | **描述** |
| --- | --- |
| [singleton](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-singleton)  单例 | (默认) 每一个Spring IoC container中一个bean定义对应一个bean实例 |
| [prototype](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-prototype)  原型 | 一个bean定义对应任意数量的实例 |
| [request](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-request)  请求 | 一个bean定义对应一个HTTP请求的生命周期。也就是说，每一个HTTP请求has有它自己的bean实例由一个bean定义创建。只有在一个web接入的上下文Spring ApplicationContext中有效。 |
| [session](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-session)  会话 | 一个bean定义对应一个HTTP会话生命周期。只有在一个web接入的上下文Spring ApplicationContext中有效。 |
| [application](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-application)  应用 | 一个bean定义对应一个ServletContext的生命周期。只有在一个web接入的上下文Spring ApplicationContext中有效。 |
| [websocket](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/web.html#websocket-stomp-websocket-scope) | 一个bean定义对应一个WebSocket的声明周期。只有在一个web接入的上下文Spring ApplicationContext中有效。 |

从spring 3.0开始，支持线程域，但默认情况下不会注册。详细信息，请见[SimpleThreadScope](https://docs.spring.io/spring-framework/docs/5.0.12.RELEASE/javadoc-api/org/springframework/context/support/SimpleThreadScope.html)文档。对于如何注册这个或其他自定义域，见[Using a custom scope](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-custom-using)。

* + - 1. 单例域

单例bean只有一个共享实例被管理，并且对与该bean定义匹配的id的所有请求Spring容器都会返回同一个特定的bean实例。

换句话说，当你定义一个bean为单例，spring IOC容器仅通过该定义创建一个对象实例。这个实例会保存在所有这些单例bean的缓存中，而后续发生的对这个名字的bean的请求和引用都会返回这个缓存中的对象。



Spring概念中的单例bean与Gang of Four (GoF)书中的单例模式不同。GoF的单例通过硬编码的方式使得每一个类加载器中有且仅有一个对象会被创建。Spring中的单例更准确的来说是一个容器一个bean。这也就意味着，如果你在一个spring容器中定义一个类的bean，那么spring容器通过bean定义会创建一个且仅有一个该类型的实例。单例域是spring中的默认域。XML格式的单例bean定义如下：

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

*<!—与下例等效，虽然多余(单例域为默认) -->*

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

* + - 1. 原型域

非单例，原型域的bean部署的结果就是在每一次请求该bean时都会创建出一个全新的bean实例。然后该bean会被注入另一个bean或者通过调用getBean()来向容器请求bean。通常对所有有状态的bean使用单例域，无状态的bean使用单例域。

下图说明了spring的原型域。一个数据访问对象（DAO）通常不会配置成原型域，因为一个传统的DAO不会保持任何的会话状态。此图仅仅用于说明情况。



如下例子是以XML的形式定义一个原型域的bean：

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

相比其他域，spring不会管理一个原型bean的完整生命周期：容器实例化、配置和组装原型对象，交由客户端后就不再对此原型实例记录。因此，即使任何域的对象都会调用初始化生命周期回调方法，在原型域中，配置的销毁生命周期回调方法不会被调用。客户端代码必须清理原型域的对象且释放被原型bean持有的珍贵资源。要是spring容器释放由原型bean持有的资源，尝试使用自定义的[bean后处理器](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-extension-bpp)，它包含需要清理的bean的引用。

在某些方面，spring容器对原型bean的角色也就相当于Java的new关键字。所有超过这点的生命周期管理应交由客户端处理。（有关spring容器中bean的声明周期，详见[生命周期回调](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle)）。

* + - 1. 持有原型bean依赖的单例bean

当你使用带有原型bean依赖的单例bean时，注意依赖是在实例化期间解析的。因此当你需要对单例bean注入一个原型bean时，一个新的原型bean将会被实例化然后注入到单例bean中。这个原型实例就是提供给单例bean的唯一实例。

然而，假如你想让单例bean在运行时重复不断地获取新的原型bean的实例。你不能往单例bean中通过依赖注入原型bean，因为注入只发生一次，仅当spring容器在实例化单例bean且解析注入它的依赖时。如果你需要在运行时超过一次的新原型bean实例，见[方法注入](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-method-injection)。

* + - 1. Request，session，application和webSocket域

Request（请求），session（会话），application（应用）和webSocket域仅在使用web接入的spring的ApplicationContext实现（例如XmlWebApplicationContext）时有效。如果你在常规的spring IOC容器如ClassPathXmlApplicationContext中使用这些域，一个非法状态异常IllegalStateException将会被抛出，以表示未知的bean域。

初始化web配置

为支持在request，session，application和webSocket等级的bean域（web相关域），在你定义bean前需要有一个小的初始化配置。（标准域，即单例和原型是不需要执行这个初始化。）

如何完成初始化设置取决于你的servlet环境

如果你在spring web MVC 中访问归入域的bean，实际上也就是由spring DispatcherServlet来处理的一个请求，则不需要特殊的设置：DispatcherServlet已经暴露所有相关的状态。

如果你使用servlet 2.5 web容器，请求交由除spring DispatcherServlet以外的框架处理（如JSF或者Struts），那么你需要注册servlet请求监听器ServletRequestListener： org.springframework.web.context.request.RequestContextListener。对于servlet 3.0+，这些可以通过WebApplicationInitializer接口来编程式完成。然则，或者对于旧容器，在你的web应用的web.xml文件中增加如下声明：

<web-app>

...

<listener>

<listener-class>

org.springframework.web.context.request.RequestContextListener

</listener-class>

</listener>

...

</web-app>

或者，如果你的监听器设置有问题，考虑使用spring的请求容器过滤器RequestContextFilter。过滤器的映射取决于它的web应用配置，因此你可以恰当地改变它为：

<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>

Servlet分发器DispatcherServlet，请求容器监听器RequestContextListener和请求容器过滤器RequestContextFilter做的全部是一件事情，即将HTTP请求对象绑定至处理请求的线程Thread。这使得请求域和会话域的bean可以在调用链中进一步使用。

Request域（请求域）

思考如下XML的bean定义：

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

Spring容器通过使用loginAction的bean定义去给每一个HTTP请求创建一个loginAction的bean新实例。也就是说，loginAction bean被归入HTTP请求等级的作用域内。你可以根据需要更改实例的内部状态，因为其他由此bean定义创建的实例不能看到更改，他们在请求间是独立的。当请求完成处理后，请求域bean会被丢弃。

当使用注解驱动组件或Java配置时，注解@RequestScope可以用于分配一个组件到请求域。

**@RequestScope**

@Component

**public** **class** **LoginAction** { *// ...* }

Session域（会话域）

思考如下XML格式的bean定义

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

Spring容器通过使用userPreferences的bean定义去创建UserPreferences bean的新实例，生命周期为单个HTTP session。换句话说，userPreferences bean有效地归入HTTP session级别的作用域中。就如request-scoped请求域bean一样，你根据需要可以改变实例内状态，因为它们相对于HTTP session是独立的，其他的由此bean定义创建的实例不能看到此变化。当HTTP session最终丢弃时，HTTP session域内的bean也会被丢弃。

使用注解驱动组件或Java配置时，注解@SessionScope可以用于分配session域的组件

**@SessionScope**

@Component

**public** **class** **UserPreferences** {

*// ...*

}

Application scope应用域

思考如下XML格式的bean定义

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

Spring容器通过使用一次appPreferences bean定义给整个web应用创建AppPreferences bean的新实例。也就是说appPreferences bean为ServletContext级别的作用域，存储为常规的ServletContext 属性。在某种程度上与spring的单例相似但又有两处不同：对于每一个ServletContext是单例，而不是每一个spring的应用上下文“ApplicationContext”（在任意给定的web应用中可能由几个上下文），而且它实际上是暴露在外的，因此可以作为一个ServletContext的属性可见。

使用注解驱动组件或Java配置时，注解@ApplicationScope可以分配application域的组件

**@ApplicationScope**

@Component

**public** **class** **AppPreferences** {

*// ...*

}

以作用域内bean为依赖

Spring IOC容器不仅管理对象（bean）的实例化，同样包括组装协作者（或依赖）。如果你需要将一个HTTP请求的bean注入另一个更长生命周期作用域内的bean，你可以选择注入AOP代理来代替作用域bean。也就是说，您需要注入一个代理对象，该对象公开与作用域对象有相同的公共接口，但也可以从作用域（例如HTTP请求）中检索真实目标对象，并将方法调用委托给真实对象。

您还可以<aop:scoped-proxy/>在作用域的bean之间使用singleton，通过引用经过可序列化的中间代理，从而能够在反序列化时重新获取目标单例bean。

当对一个原型作用域的beans声明<aop:scoped-proxy/>时，每一个共享代理的方法调用都会导致创建一个新的目标实例，并把调用传递给这个实例。

作用域代理不是以生命周期安全的方式从短期作用域中访问bean唯一方法。你同样可以简单地声明你的注入点（例如构造函数/setter方法参数或者自动装配成员）作为ObjectFactory<MyTargetBean>，允许在每次需要的时候调用getObject()去获取当前实例，而不需要单独存储它

作为上述的变种，你可以声明ObjectProvider<MyTargetBean>来传递更多的访问变量，包括getIfAvailable和getIfUnique。

JSR-330中对此的变体叫做Provider，每一次的检索都尝试通过使用结合Provider<MyTargetBean>声明和相关的get()调用来查找目标。有关JSR-330的详情见[here](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-standard-annotations)。

虽然如下配置示例只有一行，但了解“为什么”以及它背后的“如何”非常重要。

<?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">

*<!-- 一个HTTP Session作用域bean作为代理暴露出去 -->*

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

*<!-- 指示容器去代理围绕此标签的bean -->*

<aop:scoped-proxy/>

</bean>

*<!-- 注入上述bean的代理的单例bean -->*

<bean id="userService" class="com.foo.SimpleUserService">

*<!-- 被代理userPreferences bean的引用 -->*

<property name="userPreferences" ref="userPreferences"/>

</bean>

</beans>

要创建这样的一个代理，你需要在作用域bean定义内部插入子标签<aop:scoped-proxy/>（见[选择需要创建代理的类型](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-other-injection-proxies) 和 [基于XML规范文件的配置](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#xsd-schemas)）。为什么在request，session和自定义等级作用域的bean需要<aop:scoped-proxy/>标签？让我们测试如下的单例bean定义，并与你需要对前述作用域所作配置做对比（注意下例中userPreferences bean定义是不完整的）。

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

<bean id="userManager" class="com.foo.UserManager">

<property name="userPreferences" ref="userPreferences"/>

</bean>

在前述例子中，单例bean userManager被注入一个HTTP session作用域的userPreferences bean引用。显然在这里userManager bean是一个单例：每个容器它只会被实例化一次，它的依赖（这里只有一个依赖，userPreferences bean）也只会被注入一次。这也就意味着userManager bean只会操作同一个userPreferences对象，也就是一开始注入的对象。

这不是预期的行为在将较短生命作用域bean注入较长生命作用域bean时，例如将HTTP session作用域的协作者作为依赖注入一个单例bean。然而，你需要一个userManager对象，然后对于HTTP session生命周期，你需要一个userPreferences对象特定于描述HTTP session。因此容器创建一个对象，该对象暴露出与UserPreferences该类完全相同的公共接口（理想情况下*是一个* UserPreferences实例的对象），且此对象可以从作用域机制（HTTP请求，Session等）中获取真实对象。容器将代理对象注入userManager bean，但其并不知道UserPreferences引用是代理。在本例子中，当UserManager实例调用注入的UserPreferences对象的方法，实际上是调用代理的方法。之后代理便从（本例中）HTTP session中获取真实的UserPreferences对象，并将方法调用委托给检索到的真实对象。

选择创建代理的方式

默认情况下，当容器在为标记<aop:scoped-proxy/>元素的bean创建代理是，默认使用基于CGLIB的类代理。

CGLIB代理只能拦截public的方法调用！不要再代理上调用非public的方法；它们不会委托给作用域的实际对象。

或者，你可以通过指定<aop:scoped-proxy/>元素中的proxy-target-class属性值为false，来配置spring容器给作用域的bean创建使用标准JDK基于接口的代理。使用JDK基于接口的代理意味着你不需要在classpath类路径中导入任何库来使得代理生效。然而，它同样意味着作用域bean的类至少实现一个接口，因此注入作用域bean的所有合作者必须通过其接口之一引用bean。

*<!-- DefaultUserPreferences实现UserPreferences接口 -->*

<bean id="userPreferences" class="com.foo.DefaultUserPreferences" scope="session">

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

</bean>

<bean id="userManager" class="com.foo.UserManager">

<property name="userPreferences" ref="userPreferences"/>

</bean>

有关选择基于类或者基于接口代理的详细信息，见[Proxying mechanisms代理机制](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#aop-proxying)

* + - 1. 自定义作用域

Bean的作用域机制是可以扩展的；你可以定义你自己的作用域，或甚至重定义已存在的作用域尽管这被认为是坏实践，但你不能覆盖内置的单例和原型作用域。

创建自定义作用域

若需整合自定义作用域进spring容器，你需要实现即将在本节中介绍的接口org.springframework.beans.factory.config.Scope。有关如何实现自己的作用域的想法，详情请参阅在spring框架自身提供的Scope的实现和解释了哪些方法你应该去实现的[Scope javadocs](https://docs.spring.io/spring-framework/docs/5.0.12.RELEASE/javadoc-api/org/springframework/beans/factory/config/Scope.html)。

Scope接口有四种方法可以从作用域中获取对象，从作用域中删除它们，并允许它们被销毁。

如下方法从下面的作用域中返回对象。例如，session作用域的实现返回session作用域中的bean（且当其不存在时，该方法会返回bean的新实例，并将其绑定至session以供后面的引用）。

Object get(String name, ObjectFactory objectFactory)

如下方法从其作用域中删除对象。例如，session作用域的实现从其所在的session作用域中删除指定bean。被删除的bean应该返回，当指定名字的bean如果没找到时可以返回null。

Object remove(String name)

如下方法注册当作用域被销毁或指定作用域内对象被销毁时的回调方法。详情参阅javadocs或spring的作用域实现有关销毁回调的内容。

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

如下方法获取对应作用域的会话标识符。每一个作用域的标识符都不同。对于session作用域的实现，该标识符可以是session标识符

String getConversationId()

使用自定义作用域

在你编写和测试一个或多个自定义Scope实现后，你需要让spring容器知道你的新作用域。如下方法就是向spring容器注册新作用域的核心方法：

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

这个方法声明在ConfigurableBeanFactory接口中，该接口在Spring通过BeanFactory属性提供的大多数具体ApplicationContext实现中都可用。

registerScope(..)方法的第一个参数是关联作用域的唯一名，例如在spring容器内的作用域名singleton和prototype。第二个参数是你希望注册和使用的自定义Scope实现的实例。

假设你编写你的自定义Scope实现，然后按如下方式注册。

如下示例使用SimpleThreadScope，其包含在spring内但默认不注册。以下指引与你自己的自定义Scope实现通用

Scope threadScope = **new** SimpleThreadScope();

beanFactory.registerScope("thread", threadScope);

然后你可以依据自定义Scope的作用域规则创建bean定义：

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

对于一个自定义的Scope实现，你不仅仅局限于编程式注册作用域。你同样可以通过使用CustomScopeConfigurer声明式地注册Scope：

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

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

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:aop="http://www.springframework.org/schema/aop"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/aop

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

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

<property name="scopes">

<map>

<entry key="thread">

<bean class="org.springframework.context.support.SimpleThreadScope"/>

</entry>

</map>

</property>

</bean>

<bean id="bar" class="x.y.Bar" scope="thread">

<property name="name" value="Rick"/>

<aop:scoped-proxy/>

</bean>

<bean id="foo" class="x.y.Foo">

<property name="bar" ref="bar"/>

</bean>

</beans>

当你在FactoryBean实现中放置<aop:scoped-proxy/>，事实上是工厂bean本身归入作用域，而不是getObject()方法返回的对象。

* + 1. 自定义bean特性
       1. 生命周期回调

若需要与容器管理的bean生命周期作交互，你可以实现spring的InitializingBean和DisposableBean接口。容器对前者调用afterPropertiesSet()，对后者调用destroy()，来允许bena在初始化和销毁时执行特定的动作。

JSR-250规范中的@PostConstruct和@PreDestroy注解通常被视为在现代spring应用中获取获取生命周期回调的最佳实践。使用这些注解意味着你的bean没有与spring的任何接口耦合。详见[@PostConstruct和@PreDestroy](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-postconstruct-and-predestroy-annotations)。

如果你不想使用JSR-250规范的注解但你仍想解除耦合，考虑在对象定义元数据中使用初始化方法和销毁方法。

在内部，spring框架使用BeanPostProcessor实现去处理任何它能找到的回调接口并调用合适的方法。如果你需要自定义特性或其他spring没有提供开箱即用的生命周期行为，你可以自实现BeanPostProcessor。更多信息见[容器扩展点](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-extension)。

除了初始化和销毁​​回调之外，spring管理的对象还可以实现Lifecycle接口，这样这些对象就可以参与进由容器自身生命周期驱动的启动和关闭进程。

生命周期回调接口将会在本节描述。

初始化回调

org.springframework.beans.factory.InitializingBean接口允许允许bean在所有必须的属性由容器设置后执行初始化。InitializingBean接口指定了一个方法：

**void** afterPropertiesSet() **throws** Exception;

推荐不要使用InitializingBean接口因为没有必要将代码与spring耦合。相对于此，使用[@PostConstruct](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-postconstruct-and-predestroy-annotations)注解或指定一个POJO初始化方法。在基于XML的配置中，使用init-method属性指定一个无返回值的无参方法名。在Java代码配置中，使用@Bean中的initMethod属性，见[获取生命周期回调](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-java-lifecycle-callbacks)。如下例所示：

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

**public** **class** **ExampleBean** {

**public** **void** init() {

*// 执行初始化工作*

}

}

与下例等效：

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

**public** **class** **AnotherExampleBean** **implements** InitializingBean {

**public** **void** afterPropertiesSet() {

*// 执行初始化工作*

}

}

但没有代码与spring耦合。

销毁方法回调

实现org.springframework.beans.factory.DisposableBean接口允许bean在当持有它的容器销毁时获得回调。DisposableBean接口指定了一个方法

**void** destroy() **throws** Exception;

推荐不要使用DisposableBean接口因为没有必要将代码与spring耦合。相对于此，使用[@PreDestroy](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-postconstruct-and-predestroy-annotations)注解或指定一个bean定义支持的通用方法。在基于XML的配置中，使用在<bean/>中destroy-method属性。在Java代码配置中，使用@Bean中的destroyMethod属性，见[获取生命周期回调](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-java-lifecycle-callbacks)。如下例所示：

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

**public** **class** **ExampleBean** {

**public** **void** cleanup() {

*// 执行一些销毁工作(如释放池链接)*

}

}

与下例等效：

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

**public** **class** **AnotherExampleBean** **implements** DisposableBean {

**public** **void** destroy() {

*// 执行一些销毁工作(如释放池链接)*

}

}

但没有代码与spring耦合。

<bean>标签内的destroy-method属性可以分配一个特殊（推断）值，指示spring自动在特定类中检测一个public的close或shutdown方法（任意实现java.lang.AutoCloseable或java.io.Closeable的类将会因此匹配）。这个特殊（推断）值同样可以在<beans>标签的default-destroy-method属性中设置使得这个行为应用于整个bean集（见 [默认初始化和销毁方法](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-default-init-destroy-methods)）。注意这是Java代码配置的默认行为。

默认初始化和销毁方法

当你编写不使用spring指定InitializingBean和DisposableBean回调接口的初始化和销毁方法回调时，一般来说方法名为init()，initialize()，dispose()等等。理想情况下，这些生命周期回调方法在整个项目中应该标准化以使得所有开发者使用相同的方法名并保持一致性。

你可以设置spring容器在每一个bean中去查找初始化和销毁方法名。这意味着你，作为一名开发者，编写你的应用类且使用叫init()的初始化回调，不用为每一个bean定义设置init-method="init"。Spring IOC容器会在bean创建（并且符合前面描述的标准生命周期回调约定）时调用该方法。这个特性同样强制一个统一的初始化和销毁回调方法命名约定。

假设你的初始化回调方法命名为init()且销毁方法回调命名为destroy()。你的类将于下例类相似：

**public** **class** **DefaultBlogService** **implements** BlogService {

**private** BlogDao blogDao;

**public** **void** setBlogDao(BlogDao blogDao) {

this.blogDao = blogDao;

}

*// 不出意外这就是初始化回调方法*

**public** **void** init() {

**if** (this.blogDao == null) {

**throw** **new** IllegalStateException("The [blogDao] property must be set.");

}

}

}

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

<bean id="blogService" class="com.foo.DefaultBlogService">

<property name="blogDao" ref="blogDao" />

</bean>

</beans>

上例中顶级<beans/>元素的default-init-method属性值使得spring IOC容器在bean中识别叫init的方法作为初始化方法回调。当bean被创建和组装时，如果一个bean的类中有这个方法，它会在恰当的实际被调用。

配置销毁方法回调（在XML中）与初始化回调类似，使用顶级<beans/>元素的default-destroy-method属性。

当已存在的bean类已经有回调方法但与约定不一致，你可以通过指定（这里指XML）<bean/>的init-method属性和destroy-method属性为对应方法名来覆盖默认值。

Spring容器保证在为bean提供所有依赖项后立即调用已配置的初始化回调。因此，在原始bean引用上调用初始化回调，也就意味着AOP拦截器等尚未应用与bean。一个目标bean是先被完整创建，然后再应用具有其拦截器链的AOP代理（例如）。如果目标bean和代理是分开定义的，你的代码甚至能与原始目标bean交互而绕过代理。因此将拦截器应用在初始化方法是不合逻辑的，因为如此做会将目标bean的生命周期与它的代理/拦截器耦合，使得在你的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，有三个可供控制bean生命周期行为的选项：[InitializingBean](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-initializingbean)和[DisposableBean](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-disposablebean)接口；自定义init()和estroy()方法；还有[@PostConstruct和@PreDestroy注解](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-postconstruct-and-predestroy-annotations)。你可以将这些机制组合起来控制一个既定的bean。

如果一个bean被配置多个生命周期机制，且每一个机制都有不同方法名，这样每一个方法都会按照下列顺序执行。然而，若出现相同的方法名，例如，初始化方法init()在多个生命周期机制中，那么这个方法只会如前一节所说般执行一次。

同一个bean中配置多个生命周期机制，带有不同的初始化方法会按如下顺序调用：

* 注解@PostConstruct的方法
* 通过InitializingBean回调接口定义的afterPropertiesSet()
* 自定义配置的init()方法

销毁方法也会以相同顺序调用：

* 标注@PreDestroy的方法
* 通过DisposableBean回调接口定义的destroy()
* 自定义配置的destroy()方法

启动与停机回调

Lifecycle接口给任意有自身生命周期需求的对象定义了定义了基本方法（例如启停后台进程）：

**public** **interface** **Lifecycle** {

**void** start();

**void** stop();

**boolean** isRunning();

}

任意由spring管理的对象可以实现该接口。然后，当ApplicationContext接收到启动或停止信号时，例如一个运行时停止/重启场景，它会将所有这些调用关联到所有在上下文中定义的Lifecycle实现。通过委托给LifecycleProcessor去执行：

**public** **interface** **LifecycleProcessor** **extends** Lifecycle {

**void** onRefresh();

**void** onClose();

}

注意LifecycleProcessor本身继承Lifecycle接口。它还添加了另外两种方法来响应刷新和关闭的上下文。

注意常规org.springframework.context.Lifecycle接口只是显式启动/停止的普通通知合约，并不意味着在上下文刷新时自动启动。对特定bean实现更细粒度的自启动控制可以实现org.springframework.context.SmartLifecycle接口。同样，请注意停止通知不保证在销毁前：在常规停止中，所有的Lifecycle bean会现在销毁回调前接收到停止通知，然而，在上下文生存期内的热刷新或尝试退出刷新时，只有销毁方法被调用。

启动和停止的调用顺序很重要，如果一个“depends-on”关系存在于任意两个对象间，依赖方会后于其依赖项启动，先于其依赖项停止。但是有些时候依赖关系是未知的。你可能仅知道某个类型的对象会先于另一个类型的对象启动。在这种情况下，SmartLifecycle接口给出了另一个选项，在它的父接口，Phased中定义的名为getPhase()的方法。

**public** **interface** **Phased** {

**int** getPhase();

}

**public** **interface** **SmartLifecycle** **extends** Lifecycle, Phased {

**boolean** isAutoStartup();

**void** stop(Runnable callback);

}

启动时，在最低相位的对象先启动，停止时则相反。因此，一个实现SmartLifecycle和getPhase()方法返回值为Integer.MIN\_VALUE应是最先启动最后停止。在启动序列的另一头，相位值为Integer.MAX\_VALUE指明该对象应最后启动最先停止（可能它依赖于其他的运行进程）。在考虑相位值时，注意到任何普通的，没有实现SmartLifecycle的，Lifecycle对象的默认相位值为0同样重要。因此，任意负相位值将意味着一个对象将在那些标准组件前启动（且后于它们停止），反之亦然。

如你所见定义在SmartLifecycle的停止方法接受回调。任意实现必须在其停止进程完成后调用回调的run()方法。这样就可以在必要时启用异步关闭，因为LifecycleProcessor接口的默认实现，DefaultLifecycleProcessor，会等待每一个相位的对象都调用回调方法，直到超时。默认每一个相位的超时时间是30秒。你可以通过在上下文中定义名为“lifecycleProcessor”的bean来覆盖默认的生命周期处理器实例。若只想更改超时时间，则下例定义便足够：

<bean id="lifecycleProcessor" class="org.springframework.context.support.DefaultLifecycleProcessor">

*<!-- 超时单位为毫秒 -->*

<property name="timeoutPerShutdownPhase" value="10000"/>

</bean>

如前所述，该LifecycleProcessor接口还定义了用于刷新和关闭上下文的回调方法。后者简单地驱动停止进程就如stop()显式调用般，但它发生在上下文关闭时。另一方面，“刷新”回调启用了SmartLifecycle bean的另一特性。当上下文被刷新（在所有对象都被实例化与初始化），回调会被调用，且在这个时候默认生命周期处理器会检查每一个SmartLifecycle对象的isAutoStartup()方法的布尔返回值。若为“true”，此对象将在此时启动而不是等待上下文的或其自身的start()方法显式调用（不像上下文刷新，上下文启动不会在标准的上下文实现中自动发生）。“相位”值与任意的“depends-on（依赖于）”关系会与上述所说决定启动顺序。

在非web应用中优雅地关闭spring IOC容器

本节仅应用于非web应用。Spring基于web的ApplicationContext实现已经有在相关web应用关闭时优雅地关闭spring IOC容器的代码。

若你在非web应用环境中使用spring IOC容器，例如，在一个客户端桌面环境，你注册一个JVM的停机回调钩子（hook）。如此做保证优雅关闭且在你的单例bean上调用相关的销毁方法使得所有资源得到释放。当然，你仍需要正确配置和实现这些销毁回调。

要注册一个停机回调钩子（hook），调用声明在ConfigurableApplicationContext接口的registerShutdownHook()方法：

**import** org.springframework.context.ConfigurableApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **final** **class** **Boot** {

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

ConfigurableApplicationContext ctx = **new** ClassPathXmlApplicationContext("beans.xml");

*// 给上述上下文增加停机回调...*

ctx.registerShutdownHook();

*// app在此运行...*

*// main方法存在, hook先于app停止被调用...*

}

}

* + - 1. ApplicationContextAware和BeanNameAware

当ApplicationContext创建实现org.springframework.context.ApplicationContextAware接口的实例时，将为该实例提供ApplicationContext的引用。

**public** **interface** **ApplicationContextAware** {

**void** setApplicationContext(ApplicationContext applicationContext) **throws** BeansException;

}

因此bean可以编程式操作创建它们的ApplicationContext，通过ApplicationContext接口，或通过转换引用类型为该接口的已知子类，例如ConfigurableApplicationContext，它暴露出更多功能。一种用法是编程式检索其余bean。有些时候这个功能很有用，然而通常情况下应避免使用它，因为它使代码与spring耦合且不遵循协作者以属性提供给bean的控制反转风格。ApplicationContext的其他方法可供访问文件资源，发布应用事件，和访问MessageSource。这些额外的特性在[ApplicationContext的其他特性](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#context-introduction)中描述。

从spring 2.5开始，自动装配是从ApplicationContext中获取引用的另一方法。“传统的”constructor和byType自动装配模式（如[自动装配协作者](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-autowire)所描述）可以分别给构造函数参数或setter方法参数提供类型为ApplicationContext的依赖。为了灵活性，包括能自动装配成员变量和多个方法参数，使用新的基于注解的自动装配特性。如此做，将会在带有@Autowired注解的期望类型为ApplicationContext的成员变量、构造函数参数或方法参数中，自动装配ApplicationContext。详见[@Autowired](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-autowired-annotation)。

当ApplicationContext创建实现org.springframework.beans.factory.BeanNameAware接口的类时，将为此类提供其关联对象定义的名字的引用。

**public** **interface** **BeanNameAware** {

**void** setBeanName(String name) **throws** BeansException;

}

这个回调会在普通bean属性被填充后，但在初始化回调如InitializingBean的afterPropertiesSet方法或自定义初始化方法调用前，被调用。

* + - 1. 其他Aware接口

除了上述讨论到的ApplicationContextAware和BeanNameAware接口，spring提供了一些Aware接口，允许bean在需要明确的底层架构依赖时指示容器。最重要的Aware接口总结如下，通常来说名字就能指明依赖类型：

| *Table 4. Aware interfaces* | | |
| --- | --- | --- |
| **Name名字** | **Injected Dependency所注入的依赖** | **Explained in…​解释在…** |
| ApplicationContextAware | 声明的ApplicationContext | [ApplicationContextAware和BeanNameAware](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-aware) |
| ApplicationEventPublisherAware | 封装ApplicationContext的事件发布者 | [ApplicationContext的额外特性](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#context-introduction) |
| BeanClassLoaderAware | 用于加载bean类的类加载器 | [实例化beans](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-class) |
| BeanFactoryAware | 声明的BeanFactory | [ApplicationContextAware和BeanNameAware](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-aware) |
| BeanNameAware | 声明的bean名 | [ApplicationContextAware和BeanNameAware](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-aware) |
| BootstrapContextAware | 容器运行的资源适配器BootstrapContext。一般仅在接入JCA的ApplicationContexts中 | [JCA CCI](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/integration.html#cci) |
| LoadTimeWeaverAware | 加载时处理类定义的weaver | [加载时在spring框架内使用AspectJ织入](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#aop-aj-ltw) |
| MessageSourceAware | 解析消息的配置策略 （支持参数化和国际化） | [ApplicationContext的额外特性](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#context-introduction) |
| NotificationPublisherAware | Spring JMX通知发布者 | [通知](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/integration.html#jmx-notifications) |
| ResourceLoaderAware | 低级访问资源的配置加载器 | [资源](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#resources) |
| ServletConfigAware | 当前容器的 ServletConfig。仅在接入web的Spring ApplicationContext中有效 | [Spring MVC](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/web.html#mvc) |
| ServletContextAware | 当前容器的 ServletContext。仅在接入web的Spring ApplicationContext中有效 | [Spring MVC](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/web.html#mvc) |

再次注意使用这些接口将使你的代码与spring API绑定，不能遵循控制反转风格。因此，推荐基础架构bean需要编程式访问容器使用。

* + 1. Bean定义继承

一个bean定义可以包含许多配置信息，包括构造函数参数，属性值和容器特性信息如初始化方法、静态工厂方法名等等。一个子bean定义继承配置信息自一个父bean定义。子定义可以覆盖根据需要增加或覆盖值。使用父子bean定义可以省去许多键入时间，这也是一种有效的模板格式。

如果你编程式使用ApplicationContext接口，子bean定义由ChildBeanDefinition类表示。大部分用户不会在这个级别使用它们，而是声明式地配置bean定义在如ClassPathXmlApplicationContext中。当你使用基于XML的配置元数据时，使用parent属性指明一个子bean定义，指定该属性值为对应的父bean名。

<bean id="inheritedTestBean" abstract="true"

class="org.springframework.beans.TestBean">

<property name="name" value="parent"/>

<property name="age" value="1"/>

</bean>

<bean id="inheritsWithDifferentClass"

class="org.springframework.beans.DerivedTestBean"

parent="inheritedTestBean" init-method="initialize">

<property name="name" value="override"/>

*<!-- age属性值1将继承自父bean -->*

</bean>

如果子bean定义中为指定bean类，那么将使用父bean定义的类，但也同样可以覆盖。后者中子bean类必须兼容父bean类，即必须接受父类的属性值。

子bean定义继承作用域，构造函数参数值，属性值和可以增加新值来覆盖方法。任意指定的作用域，初始化方法，和/或静态工厂方法设定都会覆盖父bean的对应设定。

其余设置始终从子bean定义中获取：依赖于*，*自动装配模式，依赖检测，单例，懒初始化。

前述例子中使用abstract属性显式地标记父bean定义为抽象的。若父定义没有指定类，显式指定为abstract则是必须的，如下所示：

<bean id="inheritedTestBeanWithoutClass" abstract="true">

<property name="name" value="parent"/>

<property name="age" value="1"/>

</bean>

<bean id="inheritsWithClass" class="org.springframework.beans.DerivedTestBean"

parent="inheritedTestBeanWithoutClass" init-method="initialize">

<property name="name" value="override"/>

*<!-- age属性值1将继承自父bean -->*

</bean>

父bean不能被实例化因为它不完整，且被标记为abstract。当一个定义像abstract是抽象的，其仅在作为父定义提供给子定义作为模板是有用。尝试在另一个bean的属性中引用或通过父bean的id显式调用getBean()方法，去获取一个抽象的父bean，将会返回错误。相似地，容器内部的preInstantiateSingletons()方法忽略定义为抽象的bean定义。

ApplicationContext默认预实例化所有的单例。因此，如果你有一个只想用作模板的（父）bean定义，且该定义指定了一个类，你必须保证设置abstract属性为true，否则应用上下文会真的（尝试）预实例化这个abstract bean，这很重要（至少对于单例bean来说）。

* + 1. 容器扩展点

一般来说，应用开发者不需要继承ApplicationContext的实现类。相反，spring IOC容器可以通过插入特殊的整合接口实现来扩展。以下几节介绍这些整合接口。

* + - 1. 使用BeanPostProcessor自定义bean

BeanPostProcessor接口定义了回调方法，你可以实现它以提供你自己（或覆盖容器默认的）的实例化逻辑，依赖解析逻辑等等。若你想在spring容器完成实例化，配置和初始化bean后实现一些自定义逻辑，你可以插入一个或多个自定义BeanPostProcessor的实现。

你可以配置多个BeanPostProcessor实例，通过设置order属性来控制这些BeanPostProcessor的执行顺序。只有在BeanPostProcessor实现Ordered接口时才可设置此属性；若你编写你自己的BeanPostProcessor你也应该考虑实现Ordered接口。更多细节，咨询BeanPostProcessor与Ordered接口的javadocs。查看一下注意事项有关[编程式注册BeanPostProcessors](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-programmatically-registering-beanpostprocessors)。

BeanPostProcessor操作bean（或对象）实例；也就是说，spring IOC容器实例化bean实例然后BeanPostProcessor执行它们的工作。

BeanPostProcessor作用域为单容器（即对容器为单例）。这仅在你使用容器架构时才有意义。若在容器中定义一个BeanPostProcessor，它仅对容器中的bean进行后处理。换句话说，定义在一个容器中的bean不会被定义在另一个容器的BeanPostProcessor进行后处理，即使两个容器在同一个架构内。

若需改变真实的bean定义（也就是定义bean的蓝图），如 [使用BeanFactoryPostProcessor自定义配置元数据](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-extension-factory-postprocessors)中所描述，你需要使用BeanFactoryPostProcessor来实现。

org.springframework.beans.factory.config.BeanPostProcessor接口有两个回调方法。当这样的一个类作为后处理器注册进容器，对于每一个由容器创建的bean实例，在调用容器初始化方法（如初始化bean的*afterPropertiesSet()*或者任何声明的初始化方法）之前或任何bean初始化回调之后，后处理器都会从容器中获取到回调。后处理器可以对bean实例采取任何措施，包括完全忽略回调。Bean后处理器通常检查回调接口或用代理包装bean。一些spring的AOP基础类就是作为bean后处理器来实现，以便提供包装代理逻辑。

ApplicationContext自动检测定义在配置元数据中，实现了BeanPostProcessor接口的任何bean。ApplicationContext将这些bean注册为后处理器，这样它们能在稍后的bean创建时调用。Bean后处理器可以像其他bean一样部署在容器内。

注意当在配置类中使用@Bean工厂方法声明BeanPostProcessor时，该工厂方法的返回类型是实现类或至少是org.springframework.beans.factory.config.BeanPostProcessor接口，清楚地指明这个bean的后处理特性。否则，ApplicationContext将不能在在完全创建其之前通过类型自动检测出来。由于BeanPostProcessor需要尽早地实例化以便应用于容器内其他bean的实例化，这个早期的类型检测至关重要。

**编程式注册BeanPostProcessors**

对于上述推荐的BeanPostProcessor注册方法是通过ApplicationContext自动检测，同样通过使用ConfigurableBeanFactory中的addBeanPostProcessor方法来编程式地注册也是可行的。当需要在注册之前评估条件逻辑，或者甚至需要在架构内跨上下文复制bean后处理器，这可能非常有用。然而注意编程式添加BeanPostProcessors不会遵循Ordered接口。在这里注册的顺序决定了执行的顺序。另外注意，忽略任何显式的排序，编程式注册BeanPostProcessor总是在通过自动检测注册前先执行处理工作。

**BeanPostProcessors与AOP自动代理**

实现BeanPostProcessor接口的类会被容器特别对待。作为ApplicationContext特殊启动阶段的一部分，所有的BeanPostProcessor和和它们直接引用的bean都将在启动阶段实例化。然后，所有的BeanPostProcessor有序地被注册且应用于容器内后来的bean。因为AOP自动代理本身实现为BeanPostProcessor，不管BeanPostProcessor还是它们直接引用的bean都不适合自动代理，也因此没有对它们织入切面。

对于任意这样的bean，你应该看到日志信息如：“Bean foo不适用于被所有的bean处理器接口处理（例如：不适用于自动代理）”。

注意如果你有bean经过自动装配或者@Resource（可能会回退到自动装配）装配入你的BeanPostProcessor，spring可能会在类型匹配的依赖候选者时访问到意外的bean，也会因此使得它们不适用于自动代理或其他类型的bean后处理。例如，如果有依赖注解为@Resource，但其成员变量或setter方法名称没有关联到bean，且注解内属性name没有使用，那么spring将会以类型匹配来访问其他的bean。

以下例子展示了如何编写，注册和在ApplicationContext中使用BeanPostProcessor。

示例：Hello World，bean后处理风格

第一个例子说明基本使用，展示了一个自定义BeanPostProcessor实现在每一个bean被容器创建时调用toString()方法，然后打印结果字符串到控制台。

在下面可以看到对应的自定义BeanPostProcessor实现的类定义：

**package** scripting;

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

**public** **class** **InstantiationTracingBeanPostProcessor** **implements** BeanPostProcessor {

*// 简单返回实例化的bean*

**public** Object postProcessBeforeInitialization(Object bean, String beanName) {

**return** bean; *// 我们可以在此返回任意对象引用...*

}

**public** Object postProcessAfterInitialization(Object bean, String beanName) {

System.out.println("Bean '" + beanName + "' created : " + bean.toString());

**return** bean;

}

}

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

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

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:lang="http://www.springframework.org/schema/lang"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/lang

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

<lang:groovy id="messenger" script-source="classpath:org/springframework/scripting/groovy/Messenger.groovy">

<lang:property name="message" value="Fiona Apple Is Just So Dreamy."/>

</lang:groovy>

*<!--*

*当上述bean(messenger)被实例化, 这个自定义的bean后处理器实现*

*将输出至系统控制台*

*-->*

<bean class="scripting.InstantiationTracingBeanPostProcessor"/>

</beans>

注意如何简单定义InstantiationTracingBeanPostProcessor。它甚至不需要名字，且因为它是一个bean所以它可以像其他bean一样被依赖注入。（上例配置中同样定义了一个由Groovy脚本支持的bean。Spring动态语言支持详情在章节[动态语言支持](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/languages.html#dynamic-language)。）

如下简单Java应用执行上述代码和配置：

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**import** org.springframework.scripting.Messenger;

**public** **final** **class** **Boot** {

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

ApplicationContext ctx = **new** ClassPathXmlApplicationContext("scripting/beans.xml");

Messenger messenger = (Messenger) ctx.getBean("messenger");

System.out.println(messenger);

}

}

上述应用输出类似如下：

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

org.springframework.scripting.groovy.GroovyMessenger@272961

示例：必填注解bean后处理器

使用回调接口或注解与自定义BeanPostProcessor实现结合是扩展spring IOC容器的常见方法。一个例子就是spring的RequiredAnnotationBeanPostProcessor——一个随着spring发行的BeanPostProcessor实现，它保证了bean中被（专横的）注解标记的JavaBean属性确实会（被配置为）依赖注入一个值。

* + - 1. 使用bean工厂后处理器自定义配置元数据

我们来看org.springframework.beans.factory.config.BeanFactoryPostProcessor下一个扩展点。这个接口的语义类似于BeanPostProcessor，主要的不同点在于：BeanFactoryPostProcessor操作bean的配置元数据；也就是说spring IOC容器允许它去读取配置元数据并在容器实例化除BeanFactoryPostProcessor外的所有bean之前去改变配置元数据。

你可以配置多个BeanFactoryPostProcessor，也可以通过设置order属性控制其执行顺序，但仅限于当BeanFactoryPostProcessor实现了Ordered接口时。如果你编写你自己的BeanFactoryPostProcessor，你也应该考虑实现Ordered接口。更多细节，咨询BeanFactoryPostProcessor与Ordered接口的javadocs。

如果你想改变真正的bean实例（也就是从配置元数据创建的对象），那么你需要使用BeanPostProcessor（如上述[使用BeanPostProcessor自定义bean](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-extension-bpp)）。虽然技术上可以在BeanFactoryPostProcessor中使用bean实例，如使用BeanFactory.getBean()，但如此做会导致bean的提前实例化，违反标准的容器生命周期。这会导致一些负面效果如跳过bean后处理。

同样，BeanFactoryPostProcessor对于容器是单例作用域。这仅在你使用容器架构时才有意义。若在容器内定义一个BeanFactoryPostProcessor，它将仅应用于在该容器中的bean定义。一个容器内的bean定义不会被处于另一个容器的BeanFactoryPostProcessor后处理，即使两个容器都是同一个架构的一部分。

当bean工厂后处理器在ApplicationContext中声明时它会被自动执行，以便将更改应用于定义容器的配置元数据。Spring包含一些预定义的bean工厂后处理器，例如PropertyOverrideConfigurer和PropertyPlaceholderConfigurer。自定义如注册自定义属性编辑器的BeanFactoryPostProcessor也可以使用。

ApplicationContext自动检测部署在其内部实现BeanFactoryPostProcessor接口的任何bean。它会在恰当的时候以这些bean作为bean工厂后处理器使用。你可以像处理任何其他bean一样部署这些后处理器bean。

与BeanPostProcessor一样，通常不需要配置BeanFactoryPostProcessor为懒加载。若没有其他的bean引用Bean(Factory)PostProcessor，那么这些后处理器将根本不会实例化。因此将其标记为懒加载会被忽略，且Bean(Factory)PostProcessor会被饿汉式实例化即使你设置<beans />元素的default-lazy-init属性为true。

示例：使用PropertyPlaceholderConfigurer更换类名

使用PropertyPlaceholderConfigurer从bean定义中分离出值到标准Java Properties格式文件中。如此做使得部署应用程序人员可以自定义特定于环境的属性如数据库URL和密码，不需要承担修改容器主XML定义文件的复杂型和风险性。

思考下例基于XML的配置元数据片段，有一个DataSource定义了一个占位符值。此例展示了由外部Properties文件配置的属性。在运行时，PropertyPlaceholderConfigurer应用于元数据，将替换DataSource内的部分属性，需要替换的值被指定为占位符格式${property-name}，遵循了Ant / log4j / JSP EL格式。

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

<property name="locations" value="classpath:com/foo/jdbc.properties"/>

</bean>

<bean id="dataSource" destroy-method="close"

class="org.apache.commons.dbcp.BasicDataSource">

<property name="driverClassName" value="${jdbc.driverClassName}"/>

<property name="url" value="${jdbc.url}"/>

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

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

</bean>

真实值来自于另一个标准Java Properties格式文件：

jdbc.driverClassName=org.hsqldb.jdbcDriver

jdbc.url=jdbc:hsqldb:hsql://production:9002

jdbc.username=sa

jdbc.password=root

因此，字符串${jdbc.username}会在运行时被替换为值‘sa‘，这同样适用于其他匹配properties文件内关键字的占位符值。PropertyPlaceholderConfigurer检查bean定义中的大多数属性的占位符。此外，占位符的前缀和后缀都可以自定义。

通过自spring 2.5起的context命名空间，可以使用一个专用的配置元素去配置属性占位符。多个路径可以以逗号分隔的方式给location属性提供一个列表。

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

PropertyPlaceholderConfigurer（属性占位符配置器）不仅仅查找你指定的Properties文件内的属性。默认情况下它还检查Java System属性，在它不能在指定属性文件中查找到属性的情况下。你可以通过设置配置器的systemPropertiesMode属性去自定义这个行为，使用如下支持的三个整型值的其中一个：

* 从不（0）：从不检查系统属性值
* 回退（1）：若无法在指定的属性文件中解析到属性，则检查系统属性值。默认。
* 覆盖（2）：在尝试指定的属性文件前先检查系统属性值。这允许系统属性值覆盖任意其他的属性源。

更多信息查看PropertyPlaceholderConfigurer的javadoc。

你可以使用PropertyPlaceholderConfigurer去更换类名，这个在你不得不在运行时选择一个实现类时十分有用。例如：

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

<property name="locations">

<value>classpath:com/foo/strategy.properties</value>

</property>

<property name="properties">

<value>custom.strategy.class=com.foo.DefaultStrategy</value>

</property>

</bean>

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

如果类不能在运行时解析为一个有效类，bean的解析在它即将被创建的时候会失败，即一个非懒加载的bean在ApplicationContext的preInstantiateSingletons()阶段。

示例：属性覆盖配置器PropertyOverrideConfigurer

PropertyOverrideConfigurer， 另一个bean工厂后处理器，类似但并不十分像PropertyPlaceholderConfigurer，原始的定义在bean属性中可以有默认值或什么值都没有。如果覆盖Properties文件中没有对应bean属性的内容，默认上下文定义会被使用。

注意bean定义对于被覆盖是无法感知的，所以对于被使用覆盖配置器的XML定义并不直接明了。在多个PropertyOverrideConfigurer实例的情况下，给同一个bean属性定义不同值，根据覆盖机制，那么最终值为最后一个属性覆盖配置器的值。

属性配置文件每一行都依照如下格式：

beanName.property=value

例如：

dataSource.driverClassName=com.mysql.jdbc.Driver

dataSource.url=jdbc:mysql:mydb

这个示例文件应使用于容器定义包含一个名叫dataSource的bean，且其有driver和url属性。

混合属性名也是支持的，只要除最后一个被覆盖的属性外路径上的每一个组件都非空（可能由构造函数初始化）。如下所示：

foo.fred.bob.sammy=123

foo bean的fred属性中的bob属性中的sammy属性设置为标量值123。

指定覆盖值总是字面值；它们不会被转化为bean引用。当XML bean定义中指定的是bean引用时，此约定同样适用。

通过自spring 2.5起的context命名空间， 可以使用一个专用的配置元素去配置属性覆盖：

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

* + - 1. 通过FactoryBean（工厂bean）自定义实例化逻辑

实现org.springframework.beans.factory.FactoryBean接口使得对象成为工厂对象。

FactoryBean接口是spring IOC容器实例化逻辑的一个可插拔点。如果有复杂的初始化代码，相比于（可能）冗长的XML，使用Java来表示会更好，那么可以创建自己的FactoryBean，在类内编写复杂的初始化代码，然后将自定义FactoryBean插入容器。

FactoryBean接口提供三个方法：

* Object getObject()：返回本工厂创建的该对象实例。此实例可能是共享的，取决于工厂返回的是单例还是原型。
* boolean isSingleton()：若本FactoryBean返回单例，则该方法返回true，否则返回false。
* Class getObjectType()：返回getObject()方法所返回对象的类型，或未知时返回null。

FactoryBean概念和接口在spring框架内部多处使用；spring自带超过50个FactoryBean接口的实现。

当你需要向容器请求一个真实的FactoryBean实例而不是它产出的bean时，可以在调用ApplicationContext的getBean()方法时在bean的id前加“&”符号。例如，有一个FactoryBean生成id为myBean的bean，在容器内调用getBean("myBean")返回的FactoryBean产物；然而，调用getBean("&myBean")即可返回FactoryBean本身。

* + 1. 基于注解的容器配置

对于配置spring注解比XML好吗?

引入基于注解的配置引发了一个问题：这种方式是否优于XML。较短的答案是，看情况。较长的答案是，每种方式都有其优缺点，通常都由开发者决定哪种方式更适合他们。鉴于它们定义的方式，注解在其声明中提供了大量的上下文，使得配置配置更加精简准确。然而，XML胜在组装组件而不需要接触源代码或编译它们。一些开发者喜欢将装配靠近源码而其他人觉得觉得标注的类不再是POJO且配置变得更加分散不易控制。

不管选择哪一个，Spring可以兼容两种风格甚至可以混用。值得值出的是，通过它的[JavaConfig](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-java)功能，Spring允许注解以非侵入式使用，不需要接触目标组件源码且就工具而言，所有配置风格都在[Spring Tool Suite](https://spring.io/tools/sts)中支持。

基于注解的配置提供了XML配置的替代方案，该配置依赖于字节码元数据来装配组件而不是角括号声明。相较于使用XML来描述bean装配，开发者通过在相关类、方法、成员变量声明上使用注解，来把配置信息移动至组件类内部。如[示例：The RequiredAnnotationBeanPostProcessor](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-extension-bpp-examples-rabpp)（必填注解bean后处理器）所描述的，使用BeanPostProcessor结合注解是常见的扩展spring IOC容器的方式。例如，spring 2.0引入了[@Required](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-required-annotation)注解以强制注入所需属性。spring 2.5使其可以通过通用方法来驱动spring的依赖注入。基本上，@Autowired注解提供了与[自动装配协作者](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-autowire)中描述的同样功能，但有更细粒度的控制和更广的适用性。Spring 2.5也添加了对于JSR-250注解如@PostConstruct和@PreDestroy。Spring 3.0添加支持包含在在javax.inject包下的JSR-330（Java的依赖注入）注解如@Inject和@Named。有关这些注解详见[相关章节](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-standard-annotations)。

注解注入会在XML注入前执行，因此通过两种方式装配属性的，后者将会覆盖前者的值。

与往常一样，你可以将它们作为独立的bean定义注册，但它们也可以通过包含如下基于XML配置的标签来显式注册（注意包含context命名空间）：

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

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

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context

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

<context:annotation-config/>

</beans>

（被显式注册的后处理器包括[AutowiredAnnotationBeanPostProcessor](https://docs.spring.io/spring-framework/docs/5.0.12.RELEASE/javadoc-api/org/springframework/beans/factory/annotation/AutowiredAnnotationBeanPostProcessor.html)，[CommonAnnotationBeanPostProcessor](https://docs.spring.io/spring-framework/docs/5.0.12.RELEASE/javadoc-api/org/springframework/context/annotation/CommonAnnotationBeanPostProcessor.html)，[PersistenceAnnotationBeanPostProcessor](https://docs.spring.io/spring-framework/docs/5.0.12.RELEASE/javadoc-api/org/springframework/orm/jpa/support/PersistenceAnnotationBeanPostProcessor.html)，还有前面提到的[RequiredAnnotationBeanPostProcessor](https://docs.spring.io/spring-framework/docs/5.0.12.RELEASE/javadoc-api/org/springframework/beans/factory/annotation/RequiredAnnotationBeanPostProcessor.html)。）

<context:annotation-config/>仅寻找与其定义在同一个应用上下文中的注解bean，这也就意味着，如果你在WebApplicationContext中给DispatcherServlet放置<context:annotation-config/>，那么它仅检查在你的控制器（controller）中@Autowired的bean，而不包括你的服务层（services）。详见[DispatcherServlet](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/web.html#mvc-servlet)。

* + - 1. @Required

@Required注解应用于bean属性的setter方法，如下所示：

**public** **class** **SimpleMovieLister** {

**private** MovieFinder movieFinder;

@Required

**public** **void** setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

*// ...*

}

这个注解简单指明了，受影响的bean属性应该在配置时被填充，通过在bean定义中显式的属性值或通过自动装配。如果受影响的bean属性没有被填充则容器会抛出一个异常；这样则允许了急切和显式的失败，避免随后产生像NullPointerException一样的异常。仍然建议你将断言放入bean类内部，例如，放入init方法。这样即使你在容器外部使用类，也能够强制填充那些必需的引用和值。

* + - 1. @Autowired

JSR-330的@Inject注解在此可以替代spring的@Autowired注解。详见于[此](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-standard-annotations)。

你可以在构造函数上应用@Autowired注解：

**public** **class** **MovieRecommender** {

**private** **final** CustomerPreferenceDao customerPreferenceDao;

@Autowired

**public** MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {

this.customerPreferenceDao = customerPreferenceDao;

}

*// ...*

}

自spring框架4.3开始，如果目标bean仅定义一个构造函数，在构造函数上的@Autowired注解已不再必要。然而，如果多个构造函数可用，那么至少应有一个被标注以告诉容器使用哪一个。

正如所期待的，你可以在传统的setter方法上应用@Autowired注解：

**public** **class** **SimpleMovieLister** {

**private** MovieFinder movieFinder;

@Autowired

**public** **void** setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

*// ...*

}

你同样可以在任意名称和/或多个参数的方法上应用该注解：

**public** **class** **MovieRecommender** {

**private** MovieCatalog movieCatalog;

**private** CustomerPreferenceDao customerPreferenceDao;

@Autowired

**public** **void** prepare(MovieCatalog movieCatalog,

CustomerPreferenceDao customerPreferenceDao) {

this.movieCatalog = movieCatalog;

this.customerPreferenceDao = customerPreferenceDao;

}

*// ...*

}

你可以在成员变量上应用@Autowired甚至与构造函数混用：

**public** **class** **MovieRecommender** {

**private** **final** CustomerPreferenceDao customerPreferenceDao;

@Autowired

**private** MovieCatalog movieCatalog;

@Autowired

**public** MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {

this.customerPreferenceDao = customerPreferenceDao;

}

*// ...*

}

确保你的目标组件（如MovieCatalog，CustomerPreferenceDao）声明类型与使用@Autowired注解注入点类型保持一致。否则由于运行时没有找到类注入将失败。

对于XML定义的bean或通过类路径扫描的组件类，容器通常能预先知道具体类型。然而，对于@Bean工厂方法，你需要确保其声明的返回类型是明显有效的。对于实现多个接口的组件或潜在引用其实现类的组件，考虑在你的工厂方法上声明最准确的返回类型（至少精确到注入点引用bean所需要的类型）。

通过将注释添加到需要该类型数组的字段或方法中，可以从applicationContext中提取特定类型的所有bean：

**public** **class** **MovieRecommender** {

@Autowired

**private** MovieCatalog**[]** movieCatalogs;

*// ...*

}

同样适用于该类型的集合：

**public** **class** **MovieRecommender** {

**private** Set<MovieCatalog> movieCatalogs;

@Autowired

**public** **void** setMovieCatalogs(Set<MovieCatalog> movieCatalogs) {

this.movieCatalogs = movieCatalogs;

}

*// ...*

}

若需要数组或列表中的元素以一定的顺序排序，那么你可以使用@Order或@Priority注解，或者你的目标bean实现org.springframework.core.Ordered接口。否则它们的顺序将会遵循容器中相关目标bean定义的注册顺序。

@Order注解可以声明在目标类级别，也可以在@Bean方法上，潜台词就是每一个bean定义都是一个独立的个体（在多个bean定义都用同一个类的情况下）。@Order值会影响注入点的优先级，但请注意它们不会影响单例的启动顺序，这是由依赖关系和@DependsOn声明决定的

注意标准的javax.annotation.Priority注解在@Bean级别不适用因为它不能在方法上声明。它的语义通过@Order的值结合每种类型的一个bean上的@Primary来构建。

甚至固定类型的Map可以实现自动装配只要预期的key是String。Map的值会包含所有预期类型的bean，且key会包含相对应的bean名：

**public** **class** **MovieRecommender** {

**private** Map<String, MovieCatalog> movieCatalogs;

@Autowired

**public** **void** setMovieCatalogs(Map<String, MovieCatalog> movieCatalogs) {

this.movieCatalogs = movieCatalogs;

}

*// ...*

}

默认情况下，在没有候选者bean时自动装配将会失败；默认行为是把标记的方法，构造函数和成员变量都视为必须依赖。这个行为可以取消，如下所示：

**public** **class** **SimpleMovieLister** {

**private** MovieFinder movieFinder;

@Autowired(required = false)

**public** **void** setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

*// ...*

}

每一个类只有一个构造函数被标注为必须，但多个非必需的构造函数都可以被标注。在这种情况下，在候选对象中每一个构造函数都会被考虑且Spring使用最贪婪的构造函数，使得它的依赖性可以得到最大满足，也就是说，拥有最多参数的构造函数。

建议使用@Autowired的required属性而不是@Required注解。required属性指明了类属性不是自动装配所必须的，若不能自动装配则该属性可被忽略。然而@Required更强大，它使得不管如何只要容器支持那么这个属性必须要被设置。若没有值被注入，一个相关的异常将会弹出。

或者，你可以通过Java 8的java.util.Optional来表示一个特殊依赖的非必需特性：

**public** **class** **SimpleMovieLister** {

@Autowired

**public** **void** setMovieFinder(Optional<MovieFinder> movieFinder) {

...

}

}

至于spring框架5.0，你还可以使用@Nullable注解（任何包中的任何类如在JSR-305中的javax.annotation.Nullable）：

**public** **class** **SimpleMovieLister** {

@Autowired

**public** **void** setMovieFinder(@Nullable MovieFinder movieFinder) {

...

}

}

你同样还可以对知名的可解析依赖：BeanFactory，ApplicationContext，Environment，ResourceLoader，ApplicationEventPublisher，MessageSource接口使用@Autowired。这些接口以及它们的子接口，不需要任何特殊必要的设置，都可以自动解析依赖如ConfigurableApplicationContext或ResourcePatternResolver。

**public** **class** **MovieRecommender** {

@Autowired

**private** ApplicationContext context;

**public** MovieRecommender() {

}

*// ...*

}

@Autowired，@Inject，@Resource，@Value注解由spring的BeanPostProcessor实现持有，反过来也就是你不能将这些注解应用在你自己的BeanPostProcessor或BeanFactoryPostProcessor中类型中（如果有）。这些类型必须显式通过XML或使用spring的@Bean方法“装配”。

* + - 1. 使用@Primary实现基于注解的自动装配微调

因为通过类型自动装配会导致多个候选者，经常有必要对选择机制做出更多控制。达到此目的的一种方法是使用spring的@Primary注解。@Primary指示了一个特定的bean在多个bean作为候选者被自动装配入一个单值的依赖时，应该优先选择该bean。如果恰好在候选者中存在一个“主要”bean，那么它会作为自动装配值。

假设我们有如下设置，定义firstMovieCatalog为主要MovieCatalog。

@Configuration

**public** **class** **MovieConfiguration** {

@Bean

**@Primary**

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

@Bean

**public** MovieCatalog secondMovieCatalog() { ... }

*// ...*

}

在这种情况下，如下MovieRecommender将会自动装配入firstMovieCatalog。

**public** **class** **MovieRecommender** {

@Autowired

**private** MovieCatalog movieCatalog;

*// ...*

}

相关bean定义展示如下：

<?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. 使用限定符实现基于注解的自动装配微调

当可以确定一个主要候选者时，在众多实例中通过类型自动装配时@Primary是一种有效方式。若需要在选择过程中有更多的控制，可以使用spring的@Qualifier注解。你可以给限定符关联特定参数值，以缩小类型匹配的范围使得每一个参数都对应一个明确的bean。在最简单的情况下，可以是普通的描述性值：

**public** **class** **MovieRecommender** {

@Autowired

**@Qualifier("main")**

**private** MovieCatalog movieCatalog;

*// ...*

}

@Qualifier注解可以在一个单独的构造函数参数或者方法参数上指定：

**public** **class** **MovieRecommender** {

**private** MovieCatalog movieCatalog;

**private** CustomerPreferenceDao customerPreferenceDao;

@Autowired

**public** **void** prepare(**@Qualifier("main")**MovieCatalog movieCatalog,

CustomerPreferenceDao customerPreferenceDao) {

this.movieCatalog = movieCatalog;

this.customerPreferenceDao = customerPreferenceDao;

}

*// ...*

}

相关bean定义如下。带有合格值“main”的bean会被自动装配入带有同样值的构造函数参数中。

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

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

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context

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

<context:annotation-config/>

<bean class="example.SimpleMovieCatalog">

**<qualifier value="main"/>**

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

</bean>

<bean class="example.SimpleMovieCatalog">

**<qualifier value="action"/>**

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

</bean>

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

</beans>

对于回滚匹配，bean名被视为默认限定符值。因此你可以指定bean id为“main”而不需要在内嵌限定符元素中指定，也有同样的效果。然而，即使你可以使用这个约定去通过名字引用指定的bean，但@Autowired是基于类型驱动的注入，带有可选限定符的语义。这也就意味着即使使用bean名回滚，限定符在类型匹配集中总是有缩小范围的语义；它们在语义上不会表达对一个唯一bean id的引用。好的限定符值如“main”或“EMEA”或“persisten”，表示独立于bean id的特定组件的特征，像前述例子一样会在匿名bean定义的情况下会自动生成。

如前面所说，限定符同样适用于指定类型集合，例如Set<MovieCatalog>。在这种情况下，所有根据声明的限定符匹配的bean将被作为结合注入进去。这意味着限定符不必是唯一的；它们只是简单的组成了过滤标准。例如，你可以定义多个MovieCatalog bean带有同样的限定符值“action”，所有的这些都会被注入标记了@Qualifier("action")的Set<MovieCatalog>。

让限定符值来选择对应的bean，在类型匹配的候选者范围内，根本不需要在注入点增加@Qualifier注解。在非唯一依赖的情况下，如果没有其他的解析指示（如限定符或首选标记），spring将匹配与注入点名称（成员变量名或参数名）相同名称的目标bean，如果有的话。

也就是说，如果你需要表示通过名称的注解驱动注入，不要都使用@Autowired，即使其能够在类型匹配的候选者内通过bean名去选择。相较于此，使用JSR-250d的@Resource注解，其在语义上定义为通过其唯一bean名来识别一个明确的目标组件，声明类型与匹配过程无关。@Autowired有相当不同的语义：在通过类型选择候选者bean后，指定的字符型限定符值仅在这些类型匹配的候选者中被考虑，例如将“account”限定符与标记有相同限定符标签的bean匹配。

对于那些被定义为集合/map或数组类型的bean，@Resource是一种好的选择，通过唯一名称引用明确的集合或数组bean。也就是说，从4.3开始，集合/map和数组类型可以通过spring的@Autowired类型匹配算法去匹配，只要元素类型信息保存在@Bean的返回类型信息或集合继承结构内。在这种情况下，限定符值可用在相同类型中进行选择，如上一段所述。

从4.3开始，@Autowired同样考虑了自引用注入，也就是说引用回正在注入的bean。注意，自注入是一种备用操作；对其他组件的常规依赖始终具有优先权。也就是说，自引用没有参与进常规候选者选择且因此从来都不是首选的；相反的，它们总是最低的优先级。在实践中，使用自引用仅作为最终策略，例如通过bean的事务代理同一实例的其他方法：在这种场景下，考虑将受影响的方法分解到另一个代理bean中。或者，使用@Resource可以通过其唯一名来获得代理回到当前bean。

@Autowired应用在成员变量，构造函数和多参数方法中，允许通过限定符注解在参数级别上缩小范围。相反地，@Resource仅支持在成员变量和bean属性单参数的setter方法上。因此，若注入目标是构造函数或多参数方法，请坚定使用限定符。

你可以创建自定义限定符注解。简单定义一个注解然后在定义内提供@Qualifier注解：

@Target({ElementType.FIELD, ElementType.PARAMETER})

@Retention(RetentionPolicy.RUNTIME)

**@Qualifier**

**public** @interface Genre {

String value();

}

然后你可以在自动装配成员变量和参数上提供自定义限定符：

**public** **class** **MovieRecommender** {

@Autowired

**@Genre("Action")**

**private** MovieCatalog actionCatalog;

**private** MovieCatalog comedyCatalog;

@Autowired

**public** **void** setComedyCatalog(**@Genre("Comedy")** MovieCatalog comedyCatalog) {

this.comedyCatalog = comedyCatalog;

}

*// ...*

}

接着，提供候选的bean定义信息。你可以在<bean/>标签下增加<qualifier/>标签作为子元素，然后指定type和value以匹配你的自定义限定符注解。类型type要匹配注解的类全限定名。或者，如果不存在命名冲突的风险，使用短类名是很方便的选择。两种方法都在下例中给出：

<?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[类路径扫描和组件管理](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-classpath-scanning)中，你会看到相对于XML方式，使用基于注解的方式来提供限定符元数据。具体来说，见[通过注解提供限定符元数据](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-scanning-qualifiers)。

在某种情况下，使用没有值的注解就已足够。当注解以一个更通用的目的跨越多个不同类型的依赖应用时，这将很有用。比如你提供一个在没有有效internet连接时会被搜索的offline目录。首先定义一个简单的注解：

@Target({ElementType.FIELD, ElementType.PARAMETER})

@Retention(RetentionPolicy.RUNTIME)

@Qualifier

**public** @interface Offline {

}

然后在需要自动装配的成员变量或属性上添加该注解：

**public** **class** **MovieRecommender** {

@Autowired

**@Offline**

**private** MovieCatalog offlineCatalog;

*// ...*

}

现在bean定义只需要一个限定符类型type：

<bean class="example.SimpleMovieCatalog">

**<qualifier type="Offline"/>**

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

</bean>

同样你可以定义自定义的限定符注解以增加接受命名属性或替代简单的value值属性。如果指定在自动装配成员变量或属性上有多个属性值，bean定义必须匹配所有这些值才可被视为自动装配候选者。如下例注解定义所示：

@Target({ElementType.FIELD, ElementType.PARAMETER})

@Retention(RetentionPolicy.RUNTIME)

@Qualifier

**public** @interface MovieQualifier {

String genre();

Format format();

}

此时Format是个枚举：

**public** **enum** Format {

VHS, DVD, BLURAY

}

需自动装配的成员变量被标记了自定义限定符且包含两个属性的的值：genre和format。

**public** **class** **MovieRecommender** {

@Autowired

@MovieQualifier(format=Format.VHS, genre="Action")

**private** MovieCatalog actionVhsCatalog;

@Autowired

@MovieQualifier(format=Format.VHS, genre="Comedy")

**private** MovieCatalog comedyVhsCatalog;

@Autowired

@MovieQualifier(format=Format.DVD, genre="Action")

**private** MovieCatalog actionDvdCatalog;

@Autowired

@MovieQualifier(format=Format.BLURAY, genre="Comedy")

**private** MovieCatalog comedyBluRayCatalog;

*// ...*

}

最后，bean定义应包含匹配限定符的值。此示例同样表示了可以使用bean的元属性而不是子元素<qualifier/>。如果可用，<qualifier/>与其属性优先，但若不存在这样的限定符，自动装配机制将回退为在<meta/>标签内提供的值，如后两个bean定义所示：

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

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

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context

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

<context:annotation-config/>

<bean class="example.SimpleMovieCatalog">

<qualifier type="MovieQualifier">

<attribute key="format" value="VHS"/>

<attribute key="genre" value="Action"/>

</qualifier>

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

</bean>

<bean class="example.SimpleMovieCatalog">

<qualifier type="MovieQualifier">

<attribute key="format" value="VHS"/>

<attribute key="genre" value="Comedy"/>

</qualifier>

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

</bean>

<bean class="example.SimpleMovieCatalog">

<meta key="format" value="DVD"/>

<meta key="genre" value="Action"/>

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

</bean>

<bean class="example.SimpleMovieCatalog">

<meta key="format" value="BLURAY"/>

<meta key="genre" value="Comedy"/>

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

</bean>

</beans>

* + - 1. 使用泛型作为自动装配限定符

除注解@Qualifier以外，同样可以使用Java泛型作为隐式的限定格式。例如假设你有如下配置：

@Configuration

**public** **class** **MyConfiguration** {

@Bean

**public** StringStore stringStore() {

**return** **new** StringStore();

}

@Bean

**public** IntegerStore integerStore() {

**return** **new** IntegerStore();

}

}

假设如上bean实现了一个泛型接口，也就是Store<String>和Store<Integer>，你可以@Autowire接口Store且泛型将作为限定符：

@Autowired

**private** Store<String> s1; *// <String>限定符, 注入stringStore bean*

@Autowired

**private** Store<Integer> s2; *// <Integer>限定符, 注入integerStore bean*

泛型限定符同样适用于装配Lists，Maps和数组：

*// 注入所有Store beans只要它们有<Integer>泛型*

*// Store<String> beans不会在此list中出现*

@Autowired

**private** List<Store<Integer>> s;

* + - 1. 自定义自动装配配置器CustomAutowireConfigurer

[CustomAutowireConfigurer](https://docs.spring.io/spring-framework/docs/5.0.12.RELEASE/javadoc-api/org/springframework/beans/factory/annotation/CustomAutowireConfigurer.html)是一个BeanFactoryPostProcessor，允许你注册自定义限定符注解类型即使它们不被spring的@Qualifier注解标记。

<bean id="customAutowireConfigurer"

class="org.springframework.beans.factory.annotation.CustomAutowireConfigurer">

<property name="customQualifierTypes">

<set>

<value>example.CustomQualifier</value>

</set>

</property>

</bean>

AutowireCandidateResolver通过如下决定自动装配候选者：

* 每个bean定义的autowire-candidate值
* 在<beans/>元素上任意有效的default-autowire-candidates表达式
* @Qualifier注解的存在或是任意注册于CustomAutowireConfigurer自定义注解

当多个bean符合自动装配候选者，“primary”的决策如下：若恰好在候选者中有一个bean定义的primary属性设置为true，那么它将被选择。

* + - 1. @Resource

Spring也支持使用JSR-250@Resource注解来注入成员变量或bean属性的setter方法。这是在Java EE 5和6的通用规范，例如在JSF1.2中管理的bean或JAX-WS 2.0的端点endpoints。Spring也在spring管理的对象内支持此规范。

@Resource有一个name属性，默认情况下spring将其解释为被注入bena的名字。也就是说，它遵循名称语义，如下所示：

**public** **class** **SimpleMovieLister** {

**private** MovieFinder movieFinder;

**@Resource(name="myMovieFinder")**

**public** **void** setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

}

如果没有显式指定名称，默认从成员变量名或setter方法中获取名称。成员变量的情况下，成员变量情况下，它获取成员变量名；setter方法情况下，它获取属性名。因此下例将把名字为“movieFinder”的bean注入其setter方法：

**public** **class** **SimpleMovieLister** {

**private** MovieFinder movieFinder;

**@Resource**

**public** **void** setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

}

通过注解提供的名字会被接入CommonAnnotationBeanPostProcessor的ApplicationContext解析为bean名。名字可以通过JNDI来解析如果你显式配置了spring的[SimpleJndiBeanFactory](https://docs.spring.io/spring-framework/docs/5.0.12.RELEASE/javadoc-api/org/springframework/jndi/support/SimpleJndiBeanFactory.html)。然而建议你依赖于默认行为，简单地使用spring的JNDI查询功能以保持间接性。

在没有显式指定名字的情况下，@Resource跟@Autowired类似，会寻找一个匹配的主类型而不是明确明名的bean，并解析已知的可解析依赖如：BeanFactory，ApplicationContext，ResourceLoader，ApplicationEventPublisher，MessageSource接口。

因此在下例中，成员变量customerPreferenceDao先查询命名为customerPreferenceDao的bean，然后回退为匹配类型的CustomerPreferenceDao主类。“context”成员变量则被注入基于已知的可解析依赖类型ApplicationContext。

**public** **class** **MovieRecommender** {

@Resource

**private** CustomerPreferenceDao customerPreferenceDao;

@Resource

**private** ApplicationContext context;

**public** MovieRecommender() {

}

*// ...*

}

* + - 1. @PostConstruct和@PreDestroy

CommonAnnotationBeanPostProcessor不仅仅能够识别@Resource注解，也同样能够识别JSR-250的生命周期注解。自spring 2.5，对这些注解的支持提供了相较于[initialization callbacks](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-initializingbean)和[destruction callbacks](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-disposablebean)中描述的另一种方法。若CommonAnnotationBeanPostProcessor被注册进spring的ApplicationContext，带有这些注解的方法会在相关spring生命周期接口方法或显式回调函数调用时被调用。下例中，缓存将先于初始化时填充，并在销毁前清除。

**public** **class** **CachingMovieLister** {

@PostConstruct

**public** **void** populateMovieCache() {

*// populates the movie cache upon initialization...*

}

@PreDestroy

**public** **void** clearMovieCache() {

*// clears the movie cache upon destruction...*

}

}

关于组合多种生命周期机制的效果，参考 [Combining lifecycle mechanisms](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-combined-effects)。

* + 1. 类路径扫描和组件管理

在本章中的大部分示例都是用XML去指定配置元数据，并对应每一个bean定义都在spring容器内生成BeanDefinition。上一节（[基于注解的容器配置](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-annotation-config)）中表示了如何通过源码级别注解提供大量的配置元数据。然而那些例子中，“基础”bean定义在XML文件中显式定义，而注解只是驱动了依赖注入。本节将介绍一个功能，扫描类路径来隐式检测候选者组件。候选组件是匹配一个过滤器条件的类，且有相对应的注册进容器的bean定义。这样就去掉了用来表示bean定义的XML；但你要使用注解（如@Component），AspectJ类型表达式，或者你自定义的过滤规则去选择那些类能够将其bean定义注册进容器。

从spring 3.0开始，由spring JavaConfig项目提供的一些特性成为spring框架的一部分。这允许你使用Java而不是传统XML文件去定义bean。看看@Configuration，@Bean，@Import，@DependsOn注解示例以了解如何使用这些新特性。

* + - 1. @Component及其扩展注解

@Repository注解是一个标记，标记任意完成一个库（也被称为数据访问对象或DAO）角色的类。这个标记的用途是异常的自动转换，如[Exception translation异常转换](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/data-access.html#orm-exception-translation)中所描述。

Spring提供了扩展注解：@Component，@Service和@Controller。@Component是一个对spring管理组件的泛指模板。@Repository，@Service和@Controller是@Component的特殊化，用于更明确的场景，如与之相对应的持久层，服务层和显示层。因此，你可以在你的组件类上标记@Component，但标记@Repository，@Service或者@Controller，你的类将更适用于工具处理或相关联的切面。例如，这些模板注解是切入点的理想目标。同样@Repository，@Service，@Controller可能会在未来spring框架版本中携带更多的语义。因此如果你在你的服务层选择使用@Component或者@Service时，@Service明显是更好的选择。类似地，如上所描述，@Repository已经在你的持久层作为自动异常转换的标记。

* + - 1. 元注释

一些由spring提供的注解能够作为在代码里面的元注解使用。元注解即使即使可以简单应用在其他注解上的注解。例如，上述提到的@Service注解的元注解就是@Component：

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

**@Component** *// Spring 会将本注解视为@Component*

**public** @interface Service {

*// ....*

}

元注解可以结合其他注解以创建组合注解。例如，在spring MVC中的@RestController注解就是由@Controller和@ResponseBody组合。

此外，组合注解可以选择性地重新声明在元注解中的属性，以允许用户自定义。当你仅需要暴露出元注解属性的子集时这将很有用。例如，spring的@SessionScope注解将作用域硬编码于session但仍允许proxyMode的自定义。

@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;

}

@SessionScope可以用于不声明proxyMode，如下：

@Service

**@SessionScope**

**public** **class** **SessionScopedService** {

*// ...*

}

或者覆写proxyMode的值，如下：

@Service

**@SessionScope(proxyMode = ScopedProxyMode.INTERFACES)**

**public** **class** **SessionScopedUserService** **implements** UserService {

*// ...*

}

详情咨询[Spring注解编程模型](https://github.com/spring-projects/spring-framework/wiki/Spring-Annotation-Programming-Model)wiki页。

* + - 1. 自动检测类与注册bean定义

Spring可以自动原型类并注册相关bean定义进ApplicationContext。例如如下两个类符合自动检测机制：

@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*

}

为了自动检测这些类且注册相关bean，你需要在你的@Configuration类上增加@ComponentScan，其basePackages属性是两个类的共同父包。（或者你可以指定以逗号/分号/空格分隔的列表，以包括每一个类的父包）

@Configuration

@ComponentScan(basePackages = "org.example")

**public** **class** **AppConfig** {

...

}

更精简些，如上可以使用注解的value属性替换，如@ComponentScan("org.example")

如下是相对应的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">

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

</beans>

使用<context:component-scan>显式地启用了<context:annotation-config>的功能。在使用<context:component-scan>时一般不再需要<context:annotation-config>元素。

类路径包扫描需要在类路径下存在相应的相关目录。当你使用Ant来搭建你的jar包时，确定不要打开JAR任务的仅文件开关。同样，类路径目录基于某些环境的安全策略将不会暴露出来，比如在JDK 1.7.0\_45或更高的独立应用（需要在清单中设置“受信任的库”；详见[http://stackoverflow.com/questions/19394570/java-jre-7u45-breaks-classloader-getresources](https://stackoverflow.com/questions/19394570/java-jre-7u45-breaks-classloader-getresources)）

在JDK9的模块路径（Jigsaw），spring类路径扫描一般会正常执行。然而，请保证组件类存在于module-info描述文件中；若你期望spring能够调用类的非public成员，确保它们是“open”的（也就是在module-info描述文件中使用opens声明而不是exports声明）。

AutowiredAnnotationBeanPostProcessor和CommonAnnotationBeanPostProcessor都被隐式地启用，当你使用组件扫描注解时。这就意味着两个组件会被自动检测且装配-所有这些都不需要在XML中提供任何的bean配置元数据。

可以通过给注解配置属性false值禁用AutowiredAnnotationBeanPostProcessor和CommonAnnotationBeanPostProcessor的注册。

* + - 1. 使用过滤器来自定义扫描

默认情况下，标记了@Component，@Repository，@Service，@Controller，或者本身具有@Component标记的自定义注解的类是被检测的候选者组件。然而，你可以简单地通过应用自定义过滤器来更改和扩展这个行为。在注解@ComponentScan中添加*includeFilters*或*excludeFilters*参数（或者在component-scan元素中添加*include-filter*或*exclude-filter*子元素）。每一个过滤器元素需要type和expression属性。如下表格描述了过滤选项

| *Table 5. 过滤类型* | | |
| --- | --- | --- |
| **过滤类型** | **表达式示例** | **描述** |
| 注解  (默认) | org.example.SomeAnnotation | 在目标组件中需要出现在类级别的注释 |
| 从属 | org.example.SomeClass | 目标组件通过继承或实现从属于某个类或接口 |
| aspectj | org.example..\*Service+ | 匹配AspectJ风格表达式的目标组件 |
| 正则 | org\.example\.Default.\* | 匹配正则表达式类名的目标组件 |
| 自定义 | org.example.MyTypeFilter | org.springframework.core.type.TypeFilter接口的自定义实现 |

如下示例展示了忽略@Repository注解，使用“stub” repositories。

@Configuration

@ComponentScan(basePackages = "org.example",includeFilters = @Filter(type = FilterType.REGEX, pattern = ".\*Stub.\*Repository"),

excludeFilters = @Filter(Repository.class))

**public** **class** **AppConfig** {

...

}

等效XML如下

<beans>

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

<context:include-filter type="regex"

expression=".\*Stub.\*Repository"/>

<context:exclude-filter type="annotation"

expression="org.springframework.stereotype.Repository"/>

</context:component-scan>

</beans>

你可以通过在注解上设置useDefaultFilters=false来禁用默认过滤器或者在<component-scan/>元素上提供use-default-filters="false"属性。这些将关闭标注了@Component，@Repository，@Service，@Controller，或者@Configuration的类的自动检测。

* + - 1. 在组件内定义bean元数据

Spring组件同样可以为容器提供bean定义元数据。和@Configuration标记类一样，使用@Bean注解来定义bean元数据。例子如下：

@Component

**public** **class** **FactoryMethodComponent** {

@Bean

@Qualifier("public")

**public** TestBean publicInstance() {

**return** **new** TestBean("publicInstance");

}

**public** **void** doWork() {

*// Component method implementation omitted*

}

}

此类是spring组件，在其doWork()方法中包含应用代码。然而，它同样提供了一个bean定义，以其publicInstance()方法作为工厂方法。@Bean注解标记了工厂方法和其他bean定义属性，例如通过@Qualifier注解的限定符值。另一些可指定的方法级别的注解为@Scope，@Lazy和自定义限定注解。

@Lazy注解除了对组件初始化产生影响以外，它还能放置在标记为@Autowired或@Inject的注入点上。在这个上下文环境中，它会引出惰性解析代理的注入。

自动装配成员变量和方法已论述过，同样也支持@Bean方法的自动装配：

@Component

**public** **class** **FactoryMethodComponent** {

**private** **static** **int** i;

@Bean

@Qualifier("public")

**public** TestBean publicInstance() {

**return** **new** TestBean("publicInstance");

}

*// 使用自定义限定符且自动装配入方法参数*

@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);

}

}

上例中将另一个privateInstanc bean的属性age的值装入类型为String的方法参数country中。Spring表达式语言通过标记#{ <expression> }来传值。对于@Value注解，表达式解析器预设定表达式中的内容作为bean名去解析。

自spring框架4.3开始，你可以声明带有InjectionPoint类型入参的工厂方法（或更明确的子类DependencyDescriptor），以访问触发当前bean创建的注入点。注意这仅适用于bean实例的创建过程，而不适用于已存在bean的注入。因此，这个特性在原型作用域的bean中有很大的意义。对于其他作用域，在既定的作用域内，工厂方法仅在触发新bean创建时能够访问注入点：例如，触发懒单例bean创建的依赖。在这种场景下以语义相关性使用给定的注入点元数据。

@Component

**public** **class** **FactoryMethodComponent** {

@Bean @Scope("prototype")

**public** TestBean prototypeInstance(InjectionPoint injectionPoint) {

**return** **new** TestBean("prototypeInstance for " + injectionPoint.getMember());

}

}

@Bean方法在普通spring组件中的处理与在spring @Configuration类中不同。不同点在于@Component类没有经过CGLIB的增强去拦截方法和成员变量的调用。CGLIB代理，是通过调用@Configuration类中@Bean方法内的方法或变量，来创建对协作对象应用的bean元数据；这些方法不会通过正常的Java语义去调用，而是经过容器以提供常规生命周期管理和spring的bean代理，即使通过编程式调用@Bean方法去引用其他bean也是如此。相反，在普通的@Component类内调用方法或成员变量具有标准的Java语义，没有特殊的CGLIB处理或其他别的限制。

你可以声明@Bean方法为static，允许在不生成它们的包含配置类对象的情况下被调用。这在定义bean后处理器时会有特殊意义，例如，对于类型BeanFactoryPostProcessor或BeanPostProcessor，因为这些bean会在容器声明周期早期的时候初始化，且此时应避免触发配置的其他部分。

注意对静态@Bean方法的调用永远不会被容器所拦截，即使是在@Configuration类内（如上所述）。这是由于技术上的限制：CGLIB子类仅能够覆写非静态方法。因此，直接调用另一个@Bean方法将具有标准Java语义，直接由工厂方法返回一个独立的实例。

@Bean方法的Java语言可见性不会对spring容器内的bean定义产生直接影响。你可以随意声明你的工厂方法在非@Configuration类中你认为合适的地方或任意位置的静态方法。然而，在@Configuration类中的常规@Bean方法需为可覆盖的，也就是不能声明为private或final。

@Bean方法同样可以存在在既定的组件或配置类的基类中，在Java8平台中还可以声明在接口内的缺省方法，而接口由组件或配置类实现。这使得复杂的配置管理具有很大的灵活性，自spring 4.2开始设置可以通过Java8的缺省方法来实现多重继承。

最后，注意单个类可能具有多个返回同一个bean的@Bean方法，对多个工厂方法的分配使用取决于运行时的可用依赖。这个算法在其他配置场景下，与选择“最贪婪的”构造函数或工厂方法是一样的：具有最大数量满足的依赖的方法会在构造时运行，与容器在选择多个的@Autowired构造函数时的策略一样。

* + - 1. 命名自动检测组件

当一个组件被自动检测为扫描过程的一部分时，它的bean名会由BeanNameGenerator策略经过扫描器生成。默认情况下，任何spring的原型注解（@Component，@Repository，@Service，和@Controller）都会包含一个value属性来提供名字给相关bean定义。

如果注解中没有包含命名value为的属性或其他被检测组件（如那些由自定义过滤器发现的组件）也没有，默认bean名生成器返回类的非全限定小写名称。例如，若如下组件类被检测到，则其名字分别是myMovieLister和movieFinderImpl。

@Service("myMovieLister")

**public** **class** **SimpleMovieLister** {

*// ...*

}

@Repository

**public** **class** **MovieFinderImpl** **implements** MovieFinder {

*// ...*

}

若你不想依赖默认的bean命名生成策略，你可以提供自定义的bean命名生成策略。首先，实现[BeanNameGenerator](https://docs.spring.io/spring-framework/docs/5.0.12.RELEASE/javadoc-api/org/springframework/beans/factory/support/BeanNameGenerator.html)接口，保证有默认的无参构造方法。然后在配置扫描器时提供类的全限定名：

@Configuration

@ComponentScan(basePackages = "org.example", nameGenerator = MyNameGenerator.class)

**public** **class** **AppConfig** {

...

}

<beans>

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

name-generator="org.example.MyNameGenerator" />

</beans>

一般来说，考虑在其他组件对其使用显式引用时在注解中指定bean名。同时，在容器负责装配时自动生成名字是足够的。

* + - 1. 给自动检测组件提供作用域

通常来说对于spring管理的组件，默认情况下自动检测组件是singleton。然而，有时你可以通过@Scope注解指定不同的作用域。仅需要在注解内提供作用域名称即可：

@Scope("prototype")

@Repository

**public** **class** **MovieFinderImpl** **implements** MovieFinder {

*// ...*

}

@Scope注解仅适用于固定的bean类上（已标注组件）或者工厂方法（@Bean方法）。相比XML bean定义，没有bean定义继承的概念，在类级别的继承关系也与bean定义元数据无关。

对于在spring上下文中web指定的作用域如“request”/“session”的细节，查看[Request, session, application, and WebSocket scopes](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-other)。像这些作用域的预构建注解一样，你同样可以使用spring的元注解方式来组成你自己的作用域注解：例如，一个自定义的注解带有@Scope("prototype")元注解，可能声明了一个自定义的作用域代理模式

选择提供作用域解析的自定义策略而不是依赖于基于注解的方式，实现[ScopeMetadataResolver](https://docs.spring.io/spring-framework/docs/5.0.12.RELEASE/javadoc-api/org/springframework/context/annotation/ScopeMetadataResolver.html)接口，然后保证包含默认的无参构造方法。然后在配置扫描器时提供类的全限定名：

@Configuration

@ComponentScan(basePackages = "org.example", scopeResolver = MyScopeResolver.class)

**public** **class** **AppConfig** {

...

}

<beans>

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

</beans>

在使用明确的非单例作用域时，有必要对作用域内对象生成代理。原因在[Scoped beans as dependencies](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-other-injection)中描述了。出于此目的，在componen-scan元素中作用域代理属性可用。三个可能的值是：no, interface, targetClass。例如，如下配置会使用标准JDK动态代理：

@Configuration

@ComponentScan(basePackages = "org.example", scopedProxy = ScopedProxyMode.INTERFACES)

**public** **class** **AppConfig** {

...

}

<beans>

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

</beans>

* + - 1. 给注解提供限定符元数据

@Qualifier注解已在[Fine-tuning annotation-based autowiring with qualifiers](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-autowired-annotation-qualifiers)中描述。在此节中的示例描述了@Qualifier注解的使用和自定义限定符注解来在你解析自动装配候选者时提供更细粒度的控制。因为这些示例都是基于XML的bean定义，在XML中限定符元数据由候选者bean定义通过qualifier或bean元素的meta子元素来提供。当依赖于自动检测组件的类路径扫描时，可以在候选者类的类型级别注解上提供限定符元数据。如下三个示例展示了这个技巧：

@Component

**@Qualifier("Action")**

**public** **class** **ActionMovieCatalog** **implements** MovieCatalog {

*// ...*

}

@Component

**@Genre("Action")**

**public** **class** **ActionMovieCatalog** **implements** MovieCatalog {

*// ...*

}

@Component

**@Offline**

**public** **class** **CachingMovieCatalog** **implements** MovieCatalog {

*// ...*

}

对于大多数基于注解的配置，注意注解元数据与类定义绑定，然而使用XML时允许同一个类型的多个bean，以它们的限定符元数据来区分，因为此时元数据提供维度是每实例而不是每类。

* + - 1. 生成候选者组件索引

类路径扫描很快，可以通过在编译时创建静态候选者列表来提高大型应用的启动性能。在这个模式下，应用的所有模块必须使用这个机制，也就是当ApplicationContext检测到索引时，它将会自动启用而不是扫描类路径。

要生成索引，只需向包含组件扫描指令目标组件的每个模块添加一个附加依赖项：

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context-indexer</artifactId>

<version>5.0.12.RELEASE</version>

<optional>true</optional>

</dependency>

</dependencies>

或者，使用Gradle：

dependencies {

compileOnly("org.springframework:spring-context-indexer:5.0.12.RELEASE")

}

这样会在jar包中生成META-INF/spring.components文件。

在你的IDE中使用这个模式的时候，spring-context-indexer必须以注解处理器的形式注册以保证索引在候选者更新时得到更新

索引会在类路径中存在META-INF/spring.components文件时自动启用。如果索引仅在部分库（或使用场景）下适用但并不是为了整个应用而构建的，你可以回退到常规的类路径管理模式（也就是没有索引的模式），通过设置spring.index.ignore为true，将其设为系统属性或者在类路径根目录的spring.properties文件中。

* + 1. 使用JSR 330标准注解

自spring 3.0始，spring提供了JSR-330标准注解的支持（依赖注入）。这些注解与spring注解一样被扫描。你只需要在你的类路径中添加对应的jar包即可。

若你是用Maven，javax.inject在标准maven库中可用（[http://repo1.maven.org/maven2/javax/inject/javax.inject/1/](https://repo1.maven.org/maven2/javax/inject/javax.inject/1/)）。你可以在你的pom.xml文件中添加如下依赖：

<dependency>

<groupId>javax.inject</groupId>

<artifactId>javax.inject</artifactId>

<version>1</version>

</dependency>

* + - 1. 通过@Inject与@Named依赖注入

相较于@Autowired，@javax.inject.Inject可使用如下：

**import** javax.inject.Inject;

**public** **class** **SimpleMovieLister** {

**private** MovieFinder movieFinder;

@Inject

**public** **void** setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

**public** **void** listMovies() {

this.movieFinder.findMovies(...);

...

}

}

与@Autowired一样，可以在成员变量级别，方法级别和构造函数参数级别使用@Inject。此外，你可以声明你的注入点为Provider，允许按需访问较短作用域的bean或通过Provider.get()的调用来实现对其他bean的懒访问。如下示例为上述示例的变体：

**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(...);

...

}

}

若你需要使用限定名作为依赖，你应该使用@Named注解，如下所示：

**import** javax.inject.Inject;

**import** javax.inject.Named;

**public** **class** **SimpleMovieLister** {

**private** MovieFinder movieFinder;

@Inject

**public** **void** setMovieFinder(@Named("main") MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

*// ...*

}

如@Autowired一样，@Inject可以与java.util.Optional或者@Nullable一起使用。这在这里更适用因为@Inject没有required属性。

**public** **class** **SimpleMovieLister** {

@Inject

**public** **void** setMovieFinder(Optional<MovieFinder> movieFinder) {

...

}

}

**public** **class** **SimpleMovieLister** {

@Inject

**public** **void** setMovieFinder(@Nullable MovieFinder movieFinder) {

...

}

}

* + - 1. @Named和@ManagedBean：等效于@Component注解

相较于@Component，@javax.inject.Named或javax.annotation.ManagedBean可按如下适用：

**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;

}

*// ...*

}

使用@Component时不指定名字很常见。@Named也可以以类似风格使用：

**import** javax.inject.Inject;

**import** javax.inject.Named;

@Named

**public** **class** **SimpleMovieLister** {

**private** MovieFinder movieFinder;

@Inject

**public** **void** setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

*// ...*

}

当使用@Named或@ManagedBean时，应该与使用spring注解时一样使用组件扫描：

@Configuration

@ComponentScan(basePackages = "org.example")

**public** **class** **AppConfig** {

...

}

相比@Component，JSR-330的@Named和JSR-250的ManagedBean注解是不可组合的。请在创建自定义组件注解时使用spring原型。

* + - 1. JSR-330标准注解的局限性

在使用标准注解时，知道一些有意义的特性不会适用时重要的，如下表所示：

| *Table 6. Spring组件模型 vs. JSR-330变体* | | |
| --- | --- | --- |
| **Spring** | **javax.inject.\*** | **javax.inject restrictions / comments** |
| @Autowired | @Inject | @Inject 没有'required'属性;可以与Java8的Optional 一同使用. |
| @Component | @Named / @ManagedBean | JSR-330没有提供可组合模型,仅是标识命名组件的一种方式 |
| @Scope("singleton") | @Singleton | The JSR-330的默认作用域像Spring的prototype。然而，为了保持其与spring的常规默认一致,，JSR-330 bean在Spring容器中声明默认为singleton。若需使用除singleton以外的其他作用域你应该使用Spring的@Scope注解。javax.inject 也提供一个[@Scope](https://download.oracle.com/javaee/6/api/javax/inject/Scope.html)注解。不过，这个仅用于创建自己的注解 |
| @Qualifier | @Qualifier / @Named | javax.inject.Qualifier仅是用于创建自定义限定符。固定的String限定符(像Spring的@Qualifier带有一个值)可以通过javax.inject.Named连接起来 |
| @Value |  | 无对应等效 |
| @Required |  | 无对应等效 |
| @Lazy |  | 无对应等效 |
| ObjectFactory | Provider | javax.inject.Provider可直接替换Spring的ObjectFactory，仅需要一个较短的get()方法名。它同样可以与Spring的@Autowired结合或在无标记的构造函数或setter方法中使用。 |

* + 1. 基于Java的容器配置
       1. 基本概念：@Bean和@Configuration

在spring的Java配置中，主要核心就是@Configuration标记类和@Bean标记方法。

@Bean注解用于指示一个方法用来实例化、配置并初始化一个交由spring IoC容器管理的新对象。对于已熟悉spring的<beans/>XML配置而言，@Bean注解与<bean/>元素扮演着同样的角色。你可以使用@Bean标记的方法在任意的spring @Component内，但是大多数情况下它们都在@Configuration bean内。

标记一个类为@Configuration意味着它的主要目的就是bean的定义源。此外，@Configuration类允许通过简单调用在同一个类内的其他@Bean方法来定义内部bean依赖关系。最简单的@Configuration类如下：

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** MyService myService() {

**return** **new** MyServiceImpl();

}

}

AppConfig类与如下spring <beans/>XML等效：

<beans>

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

</beans>

全@Configuration vs 轻量@Bean模式?

当 @Bean 方法声明在没有标注为 @Configuration 的类内时，它们被视为轻量模式。Bean方法声明在@Component 甚至是一个普通的类中时，会被视为“轻量”，与包含类的主要目的不同，@Bean 方法只是额外的特点。例如，service组件会在每个适用的组建类内通过额外@Bean方法暴露管理试图给容器。在这个场景下，@Bean 只是工厂方法机制下的一个普通方法。

不像全@Configuration，轻量@Bean 不能声明内部bean依赖。相反，他们会在他们的包含组件的内部状态上和可能选择声明的参数上产生作用。@Bean 方法因此不能调用其他 @Bean 方法；每一个这样的方法只是某个特殊bean引用的工厂方法，不带有任何特殊运行时语义。正面的影响是不会有CGLIB在运行时应用生成子类，因此在类设计上不会有任何限制（也就是包含类不应该为final等等）。

通常情况下，@Bean 方法应声明在@Configuration 类内，以使用“全”模式并保证跨方法引用会重定向到容器生命周期管理中。这样可以防止同一个@Bean 方法被通过常规的Java调用方式意外调用，以减少难以追踪的bug的产生，在像操作轻量模式那样。

@Bean和@Configuration注解会在下面的章节中详细介绍，但我们要知道适用基于Java配置来创建spring容器的多种方式

* + - 1. 使用AnnotationConfigApplicationContext来实例化spring容器

如下便是spring的AnnotationConfigApplicationContext使用文档，始于spring 3.0。这个多功能的ApplicationContext实现不仅仅支持@Configuration类作为输入，也支持普通的@Component类和以JSR-330元注解标记的类。

当@Configuration类作为输入，@Configuration类本身会以bean定义的方式注册进容器，所有在类内声明的@Bean方法也会作为bean定义注册进容器。

当@Component和JSR-330作为输入时，它们会注册为bean定义，且DI元注解如@Autowired或@Inject会在必要时使用

简单构造

在大多数情况下，在实例化ClassPathXmlApplicationContext时使用spring XML文件作为输入，在实例化AnnotationConfigApplicationContext时则使用@Configuration类作为输入。这使得spring容器完全不需要XML文件：

**public** **static** **void** main(String**[]** args) {

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(AppConfig.class);

MyService myService = ctx.getBean(MyService.class);

myService.doStuff();

}

如上所述，AnnotationConfigApplicationContext不仅限于@Configuration类，任意@Component或JSR-330标记的类也可以作为构造器的输入。例如：

**public** **static** **void** main(String**[]** args) {

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(MyServiceImpl.class, Dependency1.class, Dependency2.class);

MyService myService = ctx.getBean(MyService.class);

myService.doStuff();

}

上例假设MyServiceImpl，Dependency1，Dependency2使用spring的依赖注入注解如@Autowired。

使用register(Class<?>…)编程式创建容器

AnnotationConfigApplicationContext可能由无参构造函数实例化，然后使用register()方法来配置。这种方式在编程式创建AnnotationConfigApplicationContext时十分有用。

**public** **static** **void** main(String**[]** args) {

AnnotationConfigApplicationContext ctx = **new** AnnotationConfigApplicationContext();

ctx.register(AppConfig.class, OtherConfig.class);

ctx.register(AdditionalConfig.class);

ctx.refresh();

MyService myService = ctx.getBean(MyService.class);

myService.doStuff();

}

启用组建扫描scan(String…)

若需启用组建扫描，只需如下标记你的@Configuration类：

@Configuration

@ComponentScan(basePackages = "com.acme")

**public** **class** **AppConfig** {

...

}

丰富经验的spring使用者会对如下在spring的context:命名空间下XML的等效声明感到熟悉

<beans>

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

</beans>

在上述例子中，com.acme包会被扫描，查找所有的@Component标记类，然后这些类会作为spring的bean定义注册进容器。AnnotationConfigApplicationContext暴露了方法以实现同样的组建扫描功能scan(String…​)：

**public** **static** **void** main(String**[]** args) {

AnnotationConfigApplicationContext ctx = **new** AnnotationConfigApplicationContext();

ctx.scan("com.acme");

ctx.refresh();

MyService myService = ctx.getBean(MyService.class);

}

记住@Configuration类的[元注解](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-meta-annotations)是@Component，因此它们也是被组建扫描所包含的！在上述例子中，假设AppConfig声明在com.acme包内（或任意更下一层级的包），它会在调用scan()时被识别，并在refresh()后它的所有@Bean方法会执行且在容器内注册为bean定义

* + - 1. 以AnnotationConfigWebApplicationContext支持web应用

AnnotationConfigWebApplicationContext是一种WebApplicationContext，也是AnnotationConfigApplicationContext的变体。这个实现类可在配置spring servlet监听器ContextLoaderListener，Spring MVC DispatcherServlet等等时使用。如下是web.xml的片段，配置了传统Spring MVC的web应用。注意contextClass的context-param和init-param的使用：

<web-app>

*<!-- 配置ContextLoaderListener使用AnnotationConfigWebApplicationContext而不是默认的XmlWebApplicationContext -->*

<context-param>

<param-name>contextClass</param-name>

<param-value> org.springframework.web.context.support.AnnotationConfigWebApplicationContext

</param-value>

</context-param>

*<!-- 配置资源位置必须包含一个或多个以逗号或空格分割的@Configuration类全限定名。包全限定名也可用于配置组件扫描 -->*

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>com.acme.AppConfig</param-value>

</context-param>

*<!—使用ContextLoaderListener启动根应用上下文 -->*

<listener>

<listener-class>

org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>

*<!-- 声明Spring MVC DispatcherServlet -->*

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

*<!-- 配置DispatcherServlet以使用 AnnotationConfigWebApplicationContext非默认的XmlWebApplicationContext -->*

<init-param>

<param-name>contextClass</param-name>

<param-value>

org.springframework.web.context.support.AnnotationConfigWebApplicationContext

</param-value>

</init-param>

*<!-- 配置资源位置必须包含一个或多个以逗号或空格分割的@Configuration类全限定名。 -->*

<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. 使用@Bean注解

@Bean是方法级别的注解，等价于XML <bean/>元素。该注解支持该元素中的部分属性 ，如：[init-method](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-initializingbean)，[destroy-method](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-disposablebean)，[autowiring](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-autowire)和name。

你可以在@Configuration标记类或@Component标记类内使用@Bean注解。

声明一个bean

只需在方法上标记@Bean注解即可声明一个bean。使用这个方法在ApplicationContext中注册一个bean定义，类型是方法的返回类型。默认情况下，bean名与方法名保持一致。如下便是@Bean方法声明的简单例子：

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** TransferServiceImpl transferService() {

**return** **new** TransferServiceImpl();

}

}

上例与下面的Spring XML等效：

<beans>

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

</beans>

两种方式都在ApplicationContext中创建了一个名为transferService的bean，并将其绑定至一个类型为TransferServiceImpl的对象实例：

transferService -> com.acme.TransferServiceImpl

同样你也可以声明你的@Bean方法返回一个接口（或基类）类型：

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** TransferService transferService() {

**return** **new** TransferServiceImpl();

}

}

然而，这样把类型预测限制在了指定的接口类（TransferService），一旦受影响的单例bean完成实例化，就只有容器能够知道它的真实类型（TransferServiceImpl）。非懒加载的单例bean依据声明顺序来实例化，所以你会看到不同的类型匹配结果，取决于当另一个组件尝试以非声明类型匹配时（例如@Autowired TransferServiceImpl就只会在“transferService”完成实例化后哦解析）。

如果你始终通过声明的服务接口来引用你的类型，你的@Bean返回类型可以安全适用。然而，对于实现多个接口的组件，或由它们的实现类型引用的组件，声明准确的返回类型更安全（至少对于bean内引用的注入点足够准确）。

Bean依赖

@Bean标记的方法可以有任意数量的参数，作为构造该bean的依赖。如若TransferService需要AccountRepository我们可以通过方法参数实现依赖：

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** TransferService transferService(AccountRepository accountRepository) {

**return** **new** TransferServiceImpl(accountRepository);

}

}

解析机制与基于构造方法的依赖注入几乎完全相同，详见[相关章节](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-constructor-injection)。

接收生命周期回调

任意带有@Bean注解的类都支持常规生命周期回调且可以使用JSR-250的@PostConstruct和@PreDestroy注解，详见[JSR-250注解](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-postconstruct-and-predestroy-annotations)。

常规Spring[生命周期](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-nature)回调也完全支持。如果一个bean实现了InitializingBean，DisposableBean，或Lifecycle，对应的方法会由容器调用。

标准\*Aware集内的接口如[BeanFactoryAware](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-beanfactory)，[BeanNameAware](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-aware)，[MessageSourceAware](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#context-functionality-messagesource)，[ApplicationContextAware](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-aware)等等都完全支持。

@Bean注解支持指定任意的初始化方法和销毁回调方法，很像Spring XML的init-method和destroy-method属性在bean元素上。

**public** **class** **Foo** {

**public** **void** init() {

*// initialization logic*

}

}

**public** **class** **Bar** {

**public** **void** cleanup() {

*// destruction logic*

}

}

@Configuration

**public** **class** **AppConfig** {

@Bean(initMethod = "init")

**public** Foo foo() {

**return** **new** Foo();

}

@Bean(destroyMethod = "cleanup")

**public** Bar bar() {

**return** **new** Bar();

}

}

默认情况下，适用Java配置的bean定义，在有public的close或shutdown方法存在时会自动纳入销毁回调中。如果你有一个public的close或shutdown方法且你不期望在容器关闭时调用，简单添加@Bean(destroyMethod="")在你的bean定义中以禁用默认的(inferred)模式。

对于通过JNDI获取的资源，你可能希望默认执行此操作，因为它的生命周期由应用外部管理。特别是，确保对DataSource永远执行因为已知其会在Java EE应用服务器上存在问题

@Bean(destroyMethod="")

**public** DataSource dataSource() **throws** NamingException {

**return** (DataSource) jndiTemplate.lookup("MyDS");

}

还有，在@Bean方法中，一般你会选择使用编程式查找：要么使用Spring的JndiTemplate或者JndiLocatorDelegate帮助类，要么直接使用JNDI InitialContext，但不建议使用JndiObjectFactoryBean变体，因为它会强制要求声明返回类型为FactoryBean而不是真实的目标类型，使得需要引用此处的资源的其他@Bean方法在交叉引用调用时更难以使用。

当然，在上面的Foo中，与直接在构造时调用init()方法等效：

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** Foo foo() {

Foo foo = **new** Foo();

foo.init();

**return** foo;

}

*// ...*

}

当您直接使用Java工作时，您可以使用对象执行任何您喜欢的操作，并且不必总是依赖于容器生命周期！

指定bean的作用域

使用@Scope注解

你可以给带有@Bean注解的bean指定一个明确的作用域。。你可以使用在[Bean作用域](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-scopes)一节中指出的任何标准作用域。

默认作用域是singleton，但你可以使用@Scope注解来覆盖：

@Configuration

**public** **class** **MyConfiguration** {

@Bean

**@Scope("prototype")**

**public** Encryptor encryptor() {

*// ...*

}

}

@Scope和作用域代理scoped-proxy

Spring通过[scoped proxies作用域代理](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-other-injection)提供了处理作用于内依赖的简便方法。最简单的就是在使用XML配置的时候，通过<aop:scoped-proxy/>元素来创建代理。以Java的方法来配置bean，@Scope注解有proxyMode属性支持等效的配置。默认是没有代理（ScopedProxyMode.NO），但你可以指定ScopedProxyMode.TARGET\_CLASS或ScopedProxyMode.INTERFACES。

如果你将作用域代理配置从XML转化至使用@Bean的Java配置，示例如下（对比前述章节的XML配置）：

*// 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方法名作为bean名。可以使用name属性覆写：

@Configuration

**public** **class** **AppConfig** {

@Bean(name = "myFoo")

**public** Foo foo() {

**return** **new** Foo();

}

}

Bean别名

如[Naming beans](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-beanname)中所描述，有时候需要给一个bean提供多个名字，其余的作为bean的别名。@Bean注解的name属性接收一个String数组实现此功能。

@Configuration

**public** **class** **AppConfig** {

@Bean({"dataSource", "subsystemA-dataSource", "subsystemB-dataSource"})

**public** DataSource dataSource() {

*// instantiate, configure and return DataSource bean...*

}

}

Bean描述

有时提供bean更详细的文字描述会很有用。尤其是当bean以监测目的暴露时（可能通过JMX）。

使用[@Description](https://docs.spring.io/spring-framework/docs/5.0.12.RELEASE/javadoc-api/org/springframework/context/annotation/Description.html)注解在@Bean上添加描述：

@Configuration

**public** **class** **AppConfig** {

@Bean

**@Description("Provides a basic example of a bean")**

**public** Foo foo() {

**return** **new** Foo();

}

}

* + - 1. 使用@Configuration注解

@Configuration是类级别注解，指明一个对象是bean定义的源。@Configuration类通过public的@Bean标记方法来声明bean。在@Configuration类调用@Bean方法同样可以用于定义内部bean依赖。基本概念介绍见：[基本概念: @Bean and @Configuration](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-java-basic-concepts)。

注入内部bean依赖

当@Bean有对于其他bean的依赖，表示依赖关系只需要在一个bean方法中调用另一个方法：

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** Foo foo() {

**return** **new** Foo(bar());

}

@Bean

**public** Bar bar() {

**return** **new** Bar();

}

}

上述例子中，foo bean通过构造器注入接收bar引用。

这种方式声明内部bean依赖仅在@Bean方法声明于@Configuration类内时有效。在使用普通@Component类时不能声明内部bean依赖。

Lookup方法注入

如前述所示，[lookup方法注入](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-factory-method-injection)是一项很少使用的高级功能。在单例作用域bean有原型作用域bean的依赖项的情况下有用。对于这种类型的配置，使用Java提供了实现这种模式的自然方式。

**public** **abstract** **class** **CommandManager** {

**public** Object process(Object commandState) {

*// 获取Command接口的合适的新实例*

Command command = createCommand();

*// 设置Command实例的状态（期望是全新的）*

command.setState(commandState);

**return** command.execute();

}

*// okay... but where is the implementation of this method?*

**protected** **abstract** Command createCommand();

}

使用基于Java的配置，可以创建一个覆写createCommand()方法的CommandManager的子类，用于查找一个新的（原型）command对象：

@Bean

@Scope("prototype")

**public** AsyncCommand asyncCommand() {

AsyncCommand command = **new** AsyncCommand();

*// 注入所需依赖*

**return** command;

}

@Bean

**public** CommandManager commandManager() {

*// 通过覆写的command()返回匿名的command实现*

**return** **new** CommandManager() {

**protected** Command createCommand() {

**return** asyncCommand();

}

}

}

基于Java配置的内部原理

如下实例显示@Bean方法被动调用两次：

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** ClientService clientService1() {

ClientServiceImpl clientService = **new** ClientServiceImpl();

clientService.setClientDao(clientDao());

**return** clientService;

}

@Bean

**public** ClientService clientService2() {

ClientServiceImpl clientService = **new** ClientServiceImpl();

clientService.setClientDao(clientDao());

**return** clientService;

}

@Bean

**public** ClientDao clientDao() {

**return** **new** ClientDaoImpl();

}

}

clientDao()在clientService1()和clientService2()中各被调用了一次。由于这个方法创建并返回一个新的ClientDaoImpl，正常来说你会期待有两个实例（每个service一个）。但是这样肯定会出问题：在Spring中，实例bean默认情况下是单例。这也就是为什么，所有的配置类@Configuration会在启动时通过CGLIB去被继承。在这些子类中，子方法会先检查容器查看是否有任意缓存的bean，在它调用父方法生成一个新实例之前。注意自Spring3.2起，不会再存在org.springframework.cglib因其已包含进spring的核心jar包内。

根据不同的作用域会有方法不同的行为，这里指的是单例作用域

CGLIB在启动阶段动态增加特性会有一些限制，尤其是配置类不可以是final的。然而，自4.3开始，任意构造函数都可以存在于配置类中，包括@Autowired的使用，或者一个非默认构造函数作为默认注入的声明。

若你想避免由CGLIB带来的限制，考虑声明你的@Bean方法在非@Configuration类中，例如在普通的@Component类中。在@Bean方法间的跨方法调用将不会被拦截，这样你不得不完全依赖于构造函数或方法级别的依赖注入。

* + - 1. 编写基于Java的配置

使用@Import注解

与Spring XML文件中的<import/>元素类似，@Import注解允许从其他配置类中加载@Bean定义，用于支持模块化的配置：

@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();

}

}

这样，相较于需要指定两个ConfigA.class和ConfigB.class，只需要显示提供ConfigB即可实例化上下文：

**public** **static** **void** main(String**[]** args) {

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(ConfigB.class);

*// 现在bean A和bean B都可用...*

A a = ctx.getBean(A.class);

B b = ctx.getBean(B.class);

}

这个方法简化了容器的实例化，只需要处理一个类即可，而不是在构建时需要开发者记住可能是大量的@Configuration类。

自Spring框架4.2起，@Import支持引用常规组件类，类似于注解配置上下文的register方法AnnotationConfigApplicationContext.register。在你想避免组件扫描时尤其有用，使用一些配置类作为进入点来显示定义你所有的组件。

在导入的@Bean定义中注入依赖

上例有效但简单。在大多数实际场景中，beans会有依赖跨越配置类。当使用XML时，这不是问题，因为此时没有编译介入，其中一个可以简单声明ref="someBean"且相信spring在容器初始化时自动解决。当然，在使用@Configuration类时，Java编译器会对配置模型施加约束，对其他bean的引用必须是有效的Java语法。

幸运的是，解决问题是很简单的。如[我们说过](https://docs.spring.io/spring/docs/5.0.12.RELEASE/spring-framework-reference/core.html#beans-java-dependencies)，@Bean方法可以有任意数量的入参来描述bean依赖。让我们来考虑一个更现实的场景，多个@Configuration类每个都依赖着其他类中声明的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同样的效果。记住@Configuration类最终也是容器内的一个bean：这就意味着它们可以像其他bean一样受益于@Autowired和@Value注入等。

确保以这种方式注入的依赖是最简单的那种。@Configuration类会在容器初始化时会很早被处理，这种方式强制性依赖注入会导致意外的早期初始化。如果可以，使用如上所属基于参数的注入。

还有，特殊注意BeanPostProcessor和BeanFactoryPostProcessor定义（由@Bean定义的）。这些应声明为static @Bean方法，不会触发它们的包含配置类的初始化。否则，@Autowired和@Value将不会在配置类上生效因为它作为实例太早被创建了。

@Configuration

**public** **class** **ServiceConfig** {

@Autowired

**private** AccountRepository accountRepository;

@Bean

**public** TransferService transferService() {

**return** **new** TransferServiceImpl(accountRepository);

}

}

@Configuration

**public** **class** **RepositoryConfig** {

**private** **final** DataSource dataSource;

@Autowired

**public** RepositoryConfig(DataSource dataSource) {

this.dataSource = dataSource;

}

@Bean

**public** AccountRepository accountRepository() {

**return** **new** JdbcAccountRepository(dataSource);

}

}

@Configuration

@Import({ServiceConfig.class, RepositoryConfig.class})

**public** **class** **SystemTestConfig** {

@Bean

**public** DataSource dataSource() {

*// return new DataSource*

}

}

**public** **static** **void** main(String**[]** args) {

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(SystemTestConfig.class);

*// everything wires up across configuration classes...*

TransferService transferService = ctx.getBean(TransferService.class);

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

}

@Configuration类的构造函数注入仅在Spring框架4.3后支持。注意若目标bean仅定义一个构造函数则不需要指定@Autowired；在上述例子中，在RepositoryConfig构造函数上的@Autowired是没必要的。

全限定名导入bean，便于导航

在上述情景中，使用@Autowired有效并提供了需要的模块，但决定自动装配bean确切的声明位置依旧模棱两可。例如，作为开发者查找ServiceConfig，你如何知道@Autowired AccountRepositorybean声明的确切位置？它在代码中并不明显，但[Spring Tool Suite](https://spring.io/tools/sts)提供了工具，以图表的形式展示了bean之间的依赖关系。同样，你的Java IDE可以轻易地找到所有AccountRepository的声明与使用，并快速的展示返回该类型@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中从一个@Configuration类直接导航到另一个类，请考虑自行装配配置类：

@Configuration

**public** **class** **ServiceConfig** {

@Autowired

**private** RepositoryConfig repositoryConfig;

@Bean

**public** TransferService transferService() {

*// 通过@Bean方法来导航到配置类!*

**return** **new** TransferServiceImpl(repositoryConfig.accountRepository());

}

}

在上述情景下，AccountRepository定义的位置很明显。然而，出现了ServiceConfig与RepositoryConfig的紧耦合；这是互相之间的权衡。使用基于接口或抽象@Configuration类来。查看如下示例：

@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通过对DefaultRepositoryConfig的引用实现了松耦合，同时内置IDE工具仍然有用：开发者可以很轻易的获取RepositoryConfig实现的类调用架构。这种情况下，导航@Configuration类与它们的依赖与常规的基于接口的导航无异。

若你想改变某些bean的启动实例化顺序，可在它们的上面声明@Lazy（在第一次访问时创建而不是启动就创建）或者声明@DependsOn在另一些bean上（确保在当前bean之前实例化，）

有条件地包含@Configuration类或@Bean方法

有条件地启用或禁用@Configuration类通常是有用的，甚至单独的@Bean方法，基于不同的系统状态。一个常规的例子是使用@Profile注解，来在Spring Environment内启用特定的文档（profile）时，激活bean（详见[Bean定义文档](https://docs.spring.io/spring/docs/5.0.14.RELEASE/spring-framework-reference/core.html#beans-definition-profiles)）。

注解@Profile实际上由一个更灵活的注解[@Conditional](https://docs.spring.io/spring-framework/docs/5.0.14.RELEASE/javadoc-api/org/springframework/context/annotation/Conditional.html)实现。@Conditional注解指定在@Bean注册前需经过明确的org.springframework.context.annotation.Condition实现类。

Condition接口提供了一个matches(…​)方法返回true或false。例如，这是用于@Profile的实际Condition实现类：

@Override

**public** **boolean** matches(ConditionContext context, AnnotatedTypeMetadata metadata) {

**if** (context.getEnvironment() != null) {

*// 读取 @Profile注解属性*

MultiValueMap<String, Object> attrs = metadata.getAllAnnotationAttributes(Profile.class.getName());

**if** (attrs != null) {

**for** (Object value : attrs.get("value")) {

**if** (context.getEnvironment().acceptsProfiles(((String**[]**) value))) {

**return** true;

}

}

**return** false;

}

}

**return** true;

}

详见[@Conditional javadocs](https://docs.spring.io/spring-framework/docs/5.0.14.RELEASE/javadoc-api/org/springframework/context/annotation/Conditional.html) 。

结合Java和XML配置

Spring的@Configuration类不是要100%取代Spring XML。如Spring命名空间等机制是配置Spring容器的理想方式。不管XML是方便还是必要，你都有个选择：要么实例化一个以“XML中心”的容器，如ClassPathXmlApplicationContext；要么以“Java中心”的风格使用AnnotationConfigApplicationContext和@ImportResource注解来导入所需的XML。

XML中心配置使用@Configuration类

较好的是以XML配置启动Spring容器，以@Configuration类作为附加配置条件。如，在一个既存的使用Spring XML的代码库中。根据需要创建@Configuration类并将其包含在现有XML文件中会更容易。下面你将找到@Configuration在这种“以XML为中心”的情况下使用类的选项。

如普通Spring<bean/>元素一样声明@Configuration类

记住@Configuration类最终在容器中也是bean定义。在本示例中，我们创建了名为AppConfig的@Configuration类，并将其作为<bean/>定义包含进system-test-config.xml。因为<context:annotation-config/>已经打开，容器将声明在AppConfig正确识别@Configuration注解并处理@Bean方法。

@Configuration

**public** **class** **AppConfig** {

@Autowired

**private** DataSource dataSource;

@Bean

**public** AccountRepository accountRepository() {

**return** **new** JdbcAccountRepository(dataSource);

}

@Bean

**public** TransferService transferService() {

**return** **new** TransferService(accountRepository());

}

}

**system-test-config.xml**:

<beans>

*<!—启用注解处理如 @Autowired 和@Configuration -->*

<context:annotation-config/>

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

<bean class="com.acme.AppConfig"/>

<bean class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="url" value="${jdbc.url}"/>

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

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

</bean>

</beans>

**jdbc.properties**:

jdbc.url=jdbc:hsqldb:hsql://localhost/xdb

jdbc.username=sa

jdbc.password=

**public** **static** **void** main(String**[]** args) {

ApplicationContext ctx = **new** ClassPathXmlApplicationContext("classpath:/com/acme/system-test-config.xml");

TransferService transferService = ctx.getBean(TransferService.class);

*// ...*

}

在上述system-test-config.xml中，AppConfig <bean/>没有声明id元素。这是允许的，在没有bean引用时没有必要，并且不太可能从容器中通过名字来显式获取。就像DataSource这个bean，它仅会通过类型来自动装配，因此显式的id不是严格要求。

使用<context:component-scan/>去获取@Configuration类

由于@Component是@Configuration的元注解，@Configuration标记的类会被组件扫描。在上述情况下，我们可以重定义system-test-config.xml以利用组件扫描。注意在这种情况下，不需要显式声明<context:annotation-config/>，因为<context:component-scan/>启动了相同的功能。

**system-test-config.xml**:

<beans>

*<!-- picks up and registers AppConfig as a bean definition -->*

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

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

<bean class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="url" value="${jdbc.url}"/>

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

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

</bean>

</beans>

@Configuration类中心配置使用@ImportResource导入XML

在应用中@Configuration类是配置容器的主要配置项，但还是有必要存在一些XML。在这种情况下，只需要使用@ImportResource且定义需要的XML。这样就是“Java中心”的方式来配置容器并保持XML尽量小。

@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:hsql://localhost/xdb

jdbc.username=sa

jdbc.password=

**public** **static** **void** main(String**[]** args) {

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(AppConfig.class);

TransferService transferService = ctx.getBean(TransferService.class);

*// ...*

}

* + 1. 环境抽象

[Environment](https://docs.spring.io/spring-framework/docs/5.0.14.RELEASE/javadoc-api/org/springframework/core/env/Environment.html)是整合在容器内的抽象，描述了应用环境的两个要点：[*profiles*](https://docs.spring.io/spring/docs/5.0.14.RELEASE/spring-framework-reference/core.html#beans-definition-profiles) 和[*properties*](https://docs.spring.io/spring/docs/5.0.14.RELEASE/spring-framework-reference/core.html#beans-property-source-abstraction)。

*profile*也就是，当给定的profile激活时，一个逻辑组的bean定义将注册进容器。Bean会被分配到一个profile中，不管是以XML还是注解定义的。与profile有关的Environment对象的环境取决于当前激活的profile（若存在），和默认激活的profile。

Properties几乎在所有的应用中都有着很重要的角色，并可能有多种源：properties文件，JVM系统properties，系统环境变量，JNDI，servlet上下文参数，附加properties对象，Maps等等。与properties相关的Environment对象旨在提供给用户一个方便的服务接口，用于配置属性源并从其中解析。

* + - 1. Bean定义profiles

Bean定义profiles是在核心容器中的一种机制，允许在不同的环境下注册不同的bean。*Environment*意味着对于不同用户的不同东西，且该特性可在多种情况下适用，包括：

* 在开发中使用内存中的数据源，在QA或生产环境中通过JNDI查找相同数据源
* 仅当应用部署在性能检测环境时注册监测框架
* 针对不同的用户A和用户B注册自定义的bean实现

我们假设第一种情况下在应用中需要一个DataSource。在一个测试环境中，配置如下：

@Bean

**public** DataSource dataSource() {

**return** **new** EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.HSQL)

.addScript("my-schema.sql")

.addScript("my-test-data.sql")

.build();

}

现在考虑这个应用如何部署到QA或者生产环境，假设这个应用的数据源通过生产应用服务的JNDI目录注册。我们的dataSource bean现在看起来是这样：

@Bean(destroyMethod="")

**public** DataSource dataSource() **throws** Exception {

Context ctx = **new** InitialContext();

**return** (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");

}

问题是如何基于当前环境去切换这两个bean。随着时间的推移，Spring用户有许多方式可以解决，通常依赖于系统环境变量与XML配置的结合，通过<import/>语句来导入包含${placeholder}字符的文件，根据环境变量去解析获取到正确的配置文件路径。Bean定义profiles是核心容器的特性，提供了此类问题的解决方案。

若我们概括上述情况为环境特定bean定义，我们最终需要的是注册特定的bean定义到特定的上下文中，而不是别的。可以说你想在情景A下注册固定profile下的bean定义，情景B下是另一个不同的profile。查看如下示例以响应这个需求。

@Profile

注解[@Profile](https://docs.spring.io/spring-framework/docs/5.0.14.RELEASE/javadoc-api/org/springframework/context/annotation/Profile.html)允许指定在一个或多个明确的profile激活的情况下注册某些特定组件。使用上述例子，如下可以重写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");

}

}

如上所述，通过@Bean方法，你可以选择使用编程式的JDNI查找：使用Spring的JndiTemplate/JndiLocatorDelegate帮助类或如上例使用JNDI InitialContext，但不能使用JndiObjectFactoryBean变体因为它属于FactoryBean类，会强迫声明返回类型

@Profile可以作为自定义组合注解的元注解。如下例子便是定义一个自定义的@Production注解用于代替@Profile("production")：

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

**@Profile("production")**

**public** @interface Production {

}

若@Configuration类有@Profile标记，那么这个类的所有@Bean方法和@Import注解会被忽略，除非特定的profile被激活。如果@Component或@Configuration类被标记为@Profile({"p1", "p2"})，那么这个类不会被注册或处理，除非“p1”和/或“p2”被激活。如果一个给定的profile前缀是非操作符（！），那么被标记的的元素则会在profile不被激活时注册。例如，给定@Profile({"p1", "!p2"})，那么注册会发生在profile“p1”被激活或pofile“p2”不被激活时。

@Profile也可以声明在方法级别，以在配置类内包含一个特定的bean。例如以某个bean作为变量：

@Configuration

**public** **class** **AppConfig** {

@Bean("dataSource")

**@Profile("development")**

**public** DataSource standaloneDataSource() {

**return** **new** EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.HSQL)

.addScript("classpath:com/bank/config/sql/schema.sql")

.addScript("classpath:com/bank/config/sql/test-data.sql")

.build();

}

@Bean("dataSource")

**@Profile("production")**

**public** DataSource jndiDataSource() **throws** Exception {

Context ctx = **new** InitialContext();

**return** (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");

}

}

通过在@Bean方法上的@Profile，会出现一个特殊的场景：在@Bean方法重载的情况下具有相同的方法名（与构造函数重载类似），@Profile需要在所有的重载方法上声明一致。若声明不一致，那么只有在重载方法上声明的第一个条件生效。因此@Profile并非用于在某个特殊参数下选择其中一个重载方法；所有工厂方法的解析都遵循在创建时Spring的构造方法解析算法。

若你想在不同profile条件下定义不同的bean，使用不同的Java方法名，通过@Bean的name属性，指向同一个bean名，就如上述例子一样。若方法参数签名都一样（例如都是无参工厂方法），因为在一个类中同样参数列表的方法必须不同名（即方法唯一性），这是在有效的Java类中实现上述情景的唯一途径。

XML的bean定义profile

XML中对应的是<beans>元素的profile属性。上述配置例子可以重写为XML如下：

<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>

也可以避免分开两个文件，在同一个文件内使用内嵌<beans/>元素：

<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文件约束了beans标签必须是文件的最后一个。这在保证灵活性的同时避免了XML文件的混乱。

激活profile

现在我们已经更新了我们的配置，我们仍然需要告诉Spring哪个profile应该激活。如果我们马上启动例程，会看到NoSuchBeanDefinitionException异常抛出，因为容器不能找到命名为dataSource的bean。

激活profile有多种方式，最直接的是通过ApplicationContext调用Environment的API编程式激活：

AnnotationConfigApplicationContext ctx = **new** AnnotationConfigApplicationContext();

ctx.getEnvironment().setActiveProfiles("development");

ctx.register(SomeConfig.class,StandaloneDataConfig.class,JndiDataConfig.class);

ctx.refresh();

同样，profile也可以通过spring.profiles.active属性来声明式激活，这个属性可以声明在系统环境变量，JVM系统属性，web.xml内servlet上下文参数，甚至是JNDI中的属性值（详见[PropertySource abstraction](https://docs.spring.io/spring/docs/5.0.14.RELEASE/spring-framework-reference/core.html#beans-property-source-abstraction)）。在集成测试中，可通过spring-test模块中的@ActiveProfiles注解来激活profile（详见[Context configuration with environment profiles](https://docs.spring.io/spring/docs/5.0.14.RELEASE/spring-framework-reference/testing.html#testcontext-ctx-management-env-profiles)）。

注意profile不是一个“或”命题；同时激活多个profile是有可能的。编程式提供多个profile名给setActiveProfiles()方法，它是可以接受String…​参数的：

ctx.getEnvironment().setActiveProfiles("profile1", "profile2");

声明式的，spring.profiles.active可接受多个profile名，以逗号分隔：

-Dspring.profiles.active="profile1,profile2"

默认profile

默认profile即是默认启用的profile。如下所示：

@Configuration

**@Profile("default")**

**public** **class** **DefaultDataConfig** {

@Bean

**public** DataSource dataSource() {

**return** **new** EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.HSQL)

.addScript("classpath:com/bank/config/sql/schema.sql")

.build();

}

}

若无profile激活，则上述dataSource会被创建；这个可以作为一个对bean的默认定义。如果任意一个profile被激活，那么默认profile就不会被应用。

默认profile的名字可以通过Environment的setDefaultProfiles()方法来更改，或声明式使用spring.profiles.default属性。

* + - 1. 属性源抽象

Spring的Environment抽象提供了对可配置的属性源层次结构的搜索操作。详见如下示例：

ApplicationContext ctx = **new** GenericApplicationContext();

Environment env = ctx.getEnvironment();

**boolean** containsFoo = env.containsProperty("foo");

System.out.println("Does my environment contain the 'foo' property? " + containsFoo);

在上述片段中，我们看到了一种询问Spring foo属性是否在当前环境中定义的高级方式。对于该询问的响应，Environment执行了对[PropertySource](https://docs.spring.io/spring-framework/docs/5.0.14.RELEASE/javadoc-api/org/springframework/core/env/PropertySource.html)对象集合的查询。PropertySource是任意键值对源的抽象，且Spring的[StandardEnvironment](https://docs.spring.io/spring-framework/docs/5.0.14.RELEASE/javadoc-api/org/springframework/core/env/StandardEnvironment.html)被设置为两个PropertySource对象——一个代表JVM系统属性集合（System.getProperties()），另一个代表系统环境变量（System.getenv()）。

默认属性源由StandardEnvironment表示，用于独立应用。

[StandardServletEnvironment](https://docs.spring.io/spring-framework/docs/5.0.14.RELEASE/javadoc-api/org/springframework/web/context/support/StandardServletEnvironment.html)还补充了其余默认属性源，包括servlet配置和servlet上下文参数。其可选择性启用[JndiPropertySource](https://docs.spring.io/spring-framework/docs/5.0.14.RELEASE/javadoc-api/org/springframework/jndi/JndiPropertySource.html)。详见javadocs。

当使用StandardEnvironment时，调用env.containsProperty("foo")会返回true，若系统属性foo或环境变量foo在运行时存在。

搜索是分层次的。默认情况下，系统属性优先于环境变量，因此如果碰巧foo属性在两处都有设置，在调用env.getProperty("foo")时，系统属性将优先于环境变量值而返回。注意属性值不会被合并，而是被前面的值覆盖。

对于常规的StandardServletEnvironment，全层次的查询顺序如下，最高优先级的在上面：

1. Servlet配置参数（若适用，在DispatcherServlet上下文的情况下）
2. Servlet上下文参数（web.xml上下文参数对）
3. JNDI环境变量（“java:comp/env/”键值对）
4. JVM系统属性（“-D”命令行参数）
5. JVM系统环境（操作系统环境变量）

更重要的是，这整个机制是可配置的。或许你有一些自定义的属性源想整合进这个搜索机制。没问题，只需要简单实现并实例化你自己的PropertySource且将其添加进当前Environment的PropertySources集合中：

ConfigurableApplicationContext ctx = **new** GenericApplicationContext();

MutablePropertySources sources = ctx.getEnvironment().getPropertySources();

sources.addFirst(**new** MyPropertySource());

在上述代码中，MyPropertySource已经添加到搜索列表的最高优先级 。若其包含一个foo属性，它会被检测到并在其他PropertySource的foo属性之前返回，[MutablePropertySources](https://docs.spring.io/spring-framework/docs/5.0.14.RELEASE/javadoc-api/org/springframework/core/env/MutablePropertySources.html) API暴露了一些方法，精确操纵这些属性源。

* + - 1. [@PropertySource](mailto:@PropertySource)

[@PropertySource](https://docs.spring.io/spring-framework/docs/5.0.14.RELEASE/javadoc-api/org/springframework/context/annotation/PropertySource.html)注解提供了一个更方便、声明式机制用于添加PropertySource进Spring的Environment。

给定文件“app.properties”包含键值对testbean.name=myTestBean，如下配置类使用了注解@PropertySource，因此调用testBean.getName()会返回“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;

}

}

出现在@PropertySource资源位置内的任意${…​}占位符都将被解析，以已被注册进环境的属性值替换，例如：

@Configuration

@PropertySource("classpath:/com/${my.placeholder:default/path}/app.properties")

**public** **class** **AppConfig** {

@Autowired

Environment env;

@Bean

**public** TestBean testBean() {

TestBean testBean = **new** TestBean();

testBean.setName(env.getProperty("testbean.name"));

**return** testBean;

}

}

假设“my.placeholder”已在当前注册的属性源中，例如系统属性或者环境变量，占位符将解析为相应的值。若不存在，那么“default/path”将会用作默认值。若没有指定默认值那么该属性则无法解析，将抛出IllegalArgumentException异常。

@PropertySource注解根据Java8的规定是可以重复的。但是，所有的该注解必须声明在同一层级：要么在配置类上要么是同一个自定义注解内的元注解。直接注解与元注解混合是不推荐的，因为直接注解将会覆盖元注解的效果。

* + - 1. 声明语句中的占位符解析

过去在元素内的占位符解析只能解析到JVM系统属性或者环境变量。现在已经不是这样了，因为抽象环境已经整合进容器，可经其轻易解析出占位符的值。这意味着你可以轻易的按需配置解析过程：改变系统属性和环境变量的搜索优先级，或完全移除它们；添加自定义的属性源。

如下声明忽略customer属性的定义位置，只要其在内

<beans>

<import resource="com/bank/service/${customer}-config.xml"/>

</beans>

* + 1. 注册加载时织入器

LoadTimeWeaver是类被加载入Java虚拟机（JVM）时，被Spring用于动态转换类。在你配置类中的某一个上面添加@EnableLoadTimeWeaving来启用加载时织入：

@Configuration

@EnableLoadTimeWeaving

**public** **class** **AppConfig** {

}

对应的在XML文件内使用context:load-time-weaver元素：

<beans>

<context:load-time-weaver/>

</beans>

一旦在ApplicationContext中配置好。任意在ApplicationContext内的bean都会实现LoadTimeWeaverAware，因此会收到一个加载时织入器的引用。在整合[Spring’s JPA support](https://docs.spring.io/spring/docs/5.0.14.RELEASE/spring-framework-reference/data-access.html#orm-jpa)尤其有用JPA类转换需要加载时织入。详见LocalContainerEntityManagerFactoryBeanJava文档。对于AspectJ加载时织入，查看 [在Spring框架内使用AspectJ加载时织入](https://docs.spring.io/spring/docs/5.0.14.RELEASE/spring-framework-reference/core.html#aop-aj-ltw)。

* + 1. ApplicationContext的更多能力

org.springframework.beans.factory包提供了用于管理和操作bean的基础功能，包括编程式的方式。org.springframework.context包添加了[ApplicationContext](https://docs.spring.io/spring-framework/docs/5.0.14.RELEASE/javadoc-api/org/springframework/context/ApplicationContext.html)接口，它继承了BeanFactory接口，扩展其他接口旨在以更面向应用框架的方式来提供额外的功能。许多人以完全声明式的风格使用ApplicationContext，甚至不编程式创建它，但却依赖于支持类ContextLoader来自动实例化ApplicationContext，作为Java EE web应用正常启动的一环。

为了给BeanFactory加强更面向框架的功能，context包提供了如下功能：

* 通过MessageSource接口访问i18n风格的消息
* 通过ResourceLoader接口访问资源，例如URL和文件
* 通过使用ApplicationEventPublisher接口，向实现ApplicationListener接口的bean发布事件
* 通过HierarchicalBeanFactory 接口，加载多层级上下文，允许每一层专注于该层级的任务，如应用的web层
  + - 1. 使用MessageSource国际化

ApplicationContext接口继承了MessageSource接口，因此提供了国际化功能（i18n）。Spring提供了接口HierarchicalMessageSource，可以多层级解析消息。

这些接口组成了Spring的消息解析机制。定义的方法包括：

* String getMessage(String code, Object[] args, String default, Locale loc)：基础方法，用于从MessageSource中获取消息。当在消息中没有找到指定的本地化配置，则使用默认配置。使用标准库提供的MessageFormat功能，任意传入的参数将成为替换值
* String getMessage(String code, Object[] args, Locale loc)：基本与上述方法一样，但有一处不同：没有指定默认消息；若没有则抛出NoSuchMessageException
* String getMessage(MessageSourceResolvable resolvable, Locale locale)：上述方法的所有属性封装在一个MessageSourceResolvable内时，使用此方法

当ApplicationContext被加载，其将自动搜索定义在容器中的MessageSource bean。此bean必须命名为MessageSource 。若存在，上述所有方法的调用将代理至这个消息源。若不存在消息源，ApplicationContext尝试查找一个包含该名称bean的父bean。若存在，其将被作为MessageSource。若ApplicationContext无法找到任何有关消息的源，为了能够接收上述定义的所有方法的调用，一个空消息源DelegatingMessageSource将被调用。

Spring提供了两种MessageSource实现，资源包ResourceBundleMessageSource与静态StaticMessageSource。两个实现都实现了HierarchicalMessageSource接口以处理内嵌消息。StaticMessageSource很少用但提供了编程式添加消息入消息源的方法。如下例子为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>

上述例子中定义在类路径中有三个资源包，分别是format，exceptions 和windows。任意解析消息的请求都会通过资源包交由JDK的标准解析方式处理。作为例子，假设上述资源包文件内容为：

# in format.properties

message=Alligators rock!

# in exceptions.properties

argument.required=The {0} argument is required.

下例展示了一个执行MessageSource功能的程序。记住所有的ApplicationContext实现也同样是MessageSource实现，因此可以转换至MessageSource类。

**public** **static** **void** main(String**[]** args) {

MessageSource resources = **new** ClassPathXmlApplicationContext("beans.xml");

String message = resources.getMessage("message", null, "Default", null);

System.out.println(message);

}

上述程序输出结果为：

Alligators rock!

简单来说，MessageSource 定义在beans.xml文件内，存在于类路径的根目录下。messageSource 的bean定义通过其basenames属性引用了一些资源包。在basenames属性列表内的三个文件存在于类路径根目录下，名字分别为format.properties，exceptions.properties和windows.properties。

下例展示参数传递到消息查询内；这些参数会被转换为字符串，并插入至查找消息的占位符内。

<beans>

*<!-- 这个消息源用于web应用 -->*

<bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">

<property name="basename" value="exceptions"/>

</bean>

*<!-- 将MessageSource注入此POJO -->*

<bean id="example" class="com.foo.Example">

<property name="messages" ref="messageSource"/>

</bean>

</beans>

**public** **class** **Example** {

**private** MessageSource messages;

**public** **void** setMessages(MessageSource messages) {

this.messages = messages;

}

**public** **void** execute() {

String message = this.messages.getMessage("argument.required",

**new** Object **[]** {"userDao"}, "Required", null);

System.out.println(message);

}

}

调用execute()方法输出结果如下

The userDao argument is required.

参照国际化（i18n），Spring多个MessageSource实现遵循同样的本地化解析方式，且能回退到标准JDK的ResourceBundle。简而言之，接着上述的定义的messageSource例子，若你想以British英国本地化解析消息，你该创建名三个文件如下，分别为format\_en\_GB.properties，exceptions\_en\_GB.properties，windows\_en\_GB.properties。

# 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);

}

上述程序运行结果为：

Ebagum lad, the 'userDao' argument is required, I say, required.

你同样可以使用MessageSourceAware接口类获取对已定义MessageSource的引用。定义在ApplicationContext内实现了MessageSourceAware接口的任意bean都会在其被创建和配置时，注入应用上下文的MessageSource。

作为ResourceBundleMessageSource的替换，Spring提供了可重载型资源包消息源类ReloadableResourceBundleMessageSource。这个变体提供了同样的包文件格式但相较于基于标准JDK ResourceBundleMessageSource实现更灵活。尤其是它允许从任意Spring资源位置读取文件（不只是类路径）且支持包内属性文件的热重载（同时有效地缓存它们）。详见ReloadableResourceBundleMessageSource javadocs。

* + - 1. 标准与自定义事件

ApplicationContext内的事件是通过ApplicationEvent类和ApplicationListener接口处理的。若一个实现了ApplicationListener 的bean部署在上下文内，每发布一次事件ApplicationEvent 到ApplicationContext，该bean就会被唤醒。基本上这是标准的观察者设计模式。

自Spring4.2起，事件机制被改进并提供了[基于注解模型](https://docs.spring.io/spring/docs/5.0.14.RELEASE/spring-framework-reference/core.html#context-functionality-events-annotation)来发布任意事件，这是一个不需要继承ApplicationEvent的对象。当这个对象被发布时我们将其包裹成一个事件。

Spring提供如下标准事件：

| *table 7. Built-in Events* | |
| --- | --- |
| **Event** | **Explanation** |
| ContextRefreshedEvent  上下文刷新事件 | 当ApplicationContext被初始化或被刷新时发布，例如，使用ConfigurableApplicationContext 接口内的refresh() 方法。“初始化”在这里以为着所有的bean都被加载，前处理器bean被检测并激活，单例被预实例化且ApplicationContext 对象可用。只要上下文没有被关闭，刷新可以被触发多次，只要提供的ApplicationContext 实际支持“热”刷新。例如，XmlWebApplicationContext 支持热刷新，但GenericApplicationContext不支持。 |
| ContextStartedEvent  上下文启动事件 | 在ApplicationContext 启动时发布，使用 ConfigurableApplicationContext接口的start() 方法。“启动”在这里意味着所有Lifecycle 生命周期内bean都会收到一个显式的启动信号。正常来说这个信号用于在显式停止信号后重启bean，但它同样也可用于启动违背配置为自动启动的组件，例如初始化时尚未启动的组件 |
| ContextStoppedEvent  上下文停止事件 | 在 ApplicationContext 停止时发布，使用ConfigurableApplicationContext接口的stop()方法. 在这里“停止”意味着所有生命周期Lifecycle 内的bean都会收到一个显式的停止信号。一个停止的上下文可通过start() 调用重启。 |
| ContextClosedEvent  上下文关闭事件 | 在ApplicationContext 关闭时发布， 使用 ConfigurableApplicationContext接口的close()方法. 此处“关闭”意味着所有的单例bean被销毁。一个关闭的上下文到达了生命周期末端；它不能被刷新或重启。 |
| RequestHandledEvent  请求处理事件 | 一个web专用事件，告诉所有的bean一个HTTP请求已被处理。这个事件在这个请求完成后发布。此事件仅适用于使用Spring DispatcherServlet web应用。 |

你也可以创建和发布自定义事件。下例展示了一个继承Spring的ApplicationEvent的简单类：

**public** **class** **BlackListEvent** **extends** ApplicationEvent {

**private** **final** String address;

**private** **final** String content;

**public** BlackListEvent(Object source, String address, String content) {

super(source);

this.address = address;

this.content = content;

}

*// accessor and other methods...*

}

若需发布自定义ApplicationEvent，在ApplicationEventPublisher上调用发布事件方法publishEvent()。一般来说，只要创建一个类实现ApplicationEventPublisherAware并将其注册为bean即可。如下例所示：

**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容器会检测实现ApplicationEventPublisherAware的EmailService ，并自动调用setApplicationEventPublisher()。实际上传入的参数是Spring容器本身；通过其ApplicationEventPublisher接口与应用上下文简单交互。

为了接收自定义ApplicationEvent，实现ApplicationListener并注册为Spring bean。如下所示：

**public** **class** **BlackListNotifier** **implements** ApplicationListener<BlackListEvent>

{

**private** String notificationAddress;

**public** **void** setNotificationAddress(String notificationAddress) {

this.notificationAddress = notificationAddress;

}

**public** **void** onApplicationEvent(BlackListEvent event) {

*// 通过notificationAddress唤醒合适的处理环节...*

}

}

注意ApplicationListener使用自定义事件的类型BlackListEvent。这意味着方法onApplicationEvent()可以保持类型安全，避免向下转型。你可以注册任意数量的事件监听器，但注意默认事件监听器是同步接收事件的。也就是说publishEvent()方法会阻塞直到所有的监听器处理完事件。这个同步机制和单线程的优势就是当一个监听器接收到一个事件时，如果上下文事务可用，那么它会在发布者的上下文事务内处理。若有必要对事件发布启用别的策略，参考Spring的ApplicationEventMulticaster接口javadoc。

下例展示了注册和配置上述类的bean定义：

<bean id="emailService" class="example.EmailService">

<property name="blackList">

<list>

<value>known.spammer@example.org</value>

<value>known.hacker@example.org</value>

<value>john.doe@example.org</value>

</list>

</property>

</bean>

<bean id="blackListNotifier" class="example.BlackListNotifier">

<property name="notificationAddress" value="blacklist@example.org"/>

</bean>

放在一起看，当emailService的sendEmail()方法被调用，若有邮件应该被列入黑名单，一个类型为BlackListEvent的自定义事件会被发布。blackListNotifier  bean被注册为监听器ApplicationListener ，因此会收到黑名单事件BlackListEvent，且会唤醒恰当的处理环节。

Spring的事件机制设计于在同一个应用上下文内Spring bean之间的简单交互。然而，针对复杂的企业应用整合需求，独立维护的[Spring Integration](https://projects.spring.io/spring-integration/)项目对构建轻量的，[面向范式的](https://www.enterpriseintegrationpatterns.com/)事件驱动架构提供了完全的支持，构建于已知的Spring变成模型之上。

基于注解的事件监听器

自Spring 4.2起，事件监听器可以通过EventListener注解注册在任意受管理bean的public方法上。上述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) {

*// 通过notificationAddress唤醒恰当的处理环节...*

}

}

如上所述，方法签名再一次声明了它监听的事件类型，但这次是更灵活的名字且不用实现指定的监听器接口。事件类型可以通过泛型来缩小范围，只要可以在真实的事件类型的实现结构层次中解析出你所需要的泛型参数。

若你的方法应该监听多个事件，甚至你不想定义方法入参，事件类型可以在注解上指定：

@EventListener({ContextStartedEvent.class, ContextRefreshedEvent.class})

**public** **void** handleContextStart() {

...

}

同样可以通过注解的condition 属性来添加额外的运行时过滤，通过定义一个[SpELexpression](https://docs.spring.io/spring/docs/5.0.14.RELEASE/spring-framework-reference/core.html#expressions)表达式来对匹配的事件调用此方法。

例如，唤醒器可以重写为，仅在事件的content属性等于foo时被调用：

@EventListener(condition = "#blEvent.content == 'foo'")

**public** **void** processBlackListEvent(BlackListEvent blEvent) {

*// 通过notificationAddress唤醒恰当的处理环节...*

}

每一个SpEL表达式都有专用的上下文来解析。下表罗列了在上下文内可用的项目，因此可用于条件性的事件处理：

| *Table 8. 事件SpEL可用的元数据* | | | |
| --- | --- | --- | --- |
| **Name** | **Location** | **Description** | **Example** |
| Event  事件 | root object  根对象 | 真正的应用事件 | #root.event |
| Arguments array  参数数组 | root object  根对象 | 参数（作为数组）用于目标调用 | #root.args[0] |
| *Argument name*  *参数名* | evaluation context  解析上下文 | 任意方法参数名。若某种情况下不可用（例如没有debug信息），参数名同样在#a<#arg> 下可用，此处 *#arg* 表示参数下表（从0开始）。 | #blEvent 或 #a0 （可以使用 #p0或#p<#arg>作为别名）。 |

注意#root.event允许你访问潜在的事件，即使你的方法签名实际上引用了发布的另外任意对象。

若你需要发布另一个事件的处理结果作为事件，只需要更改方法的返回值为需要发布的另一个事件即可，如：

@EventListener

**public** ListUpdateEvent handleBlackListEvent(BlackListEvent event) {

*// 通过notificationAddress唤醒恰当的处理环节...*

*// 然后发布一个ListUpdateEvent事件...*

}

此特性在[异步监听器](https://docs.spring.io/spring/docs/5.0.14.RELEASE/spring-framework-reference/core.html#context-functionality-events-async)不支持

这个新方法将会发布一个新的ListUpdateEvent，在每一个BlackListEvent被上述方法处理之后。如果你需要发布多个事件，只需要返回事件集合Collection即可。

异步监听器

若需要监听器异步处理事件，仅需使用[常规@Async 支持](https://docs.spring.io/spring/docs/5.0.14.RELEASE/spring-framework-reference/integration.html#scheduling-annotation-support-async)：

@EventListener

@Async

**public** **void** processBlackListEvent(BlackListEvent event) {

*// BlackListEvent会被另一个线程处理*

}

使用异步事件时注意如下限制：

* 若事件监听器抛出异常Exception，该异常将不会传播至其调用线程，详细信息可查看AsyncUncaughtExceptionHandler。
* 这些事件监听器不能发送回复。若你需要将处理结果作为事件发送出去，手动发送该事件可通过注入ApplicationEventPublisher实现。

监听器排序

若你需要有序性调用监听器，仅需在方法声明出添加@Order注解：

@EventListener

@Order(42)

**public** **void** processBlackListEvent(BlackListEvent event) {

*// 通过notificationAddress唤醒恰当的处理环节...*

}

泛型事件

你也可以使用泛型事件以待后续扩展。使用EntityCreatedEvent<T>，此处T是即将要创建的实际类型。你可以创建如下监听器以仅接收Person的EntityCreatedEvent事件：

@EventListener

**public** **void** onPersonCreated(EntityCreatedEvent<Person> event) {

...

}

由于类型擦除覆盖，仅当解析出事件类型泛型匹配事件监听器的过滤时有效（也就是类似于class PersonCreatedEvent extends EntityCreatedEvent<Person> { …​ }）。

某些特定情况下，所有的事件必须要遵循同样的结构将会十分冗余（因其必须遵循上述事件类型结构）。在这种情况下，你可以实现ResolvableTypeProvider 以引导框架越过运行时环境提供的解析方法：

**public** **class** **EntityCreatedEvent**<T> **extends** ApplicationEvent **implements** ResolvableTypeProvider

{

**public** EntityCreatedEvent(T entity) {

super(entity);

}

@Override

**public** ResolvableType getResolvableType() {

**return** ResolvableType.forClassWithGenerics(getClass(),

ResolvableType.forInstance(getSource()));

}

}

这不仅仅在ApplicationEvent 中有效，也适用于任意你想作为事件的对象。

* + - 1. 对底层资源的轻易访问

为深刻理解与使用应用上下文，用户应该通过Spring的Resource抽象以通晓它们，如章节[Resources](https://docs.spring.io/spring/docs/5.0.14.RELEASE/spring-framework-reference/core.html#resources)所述。

一个应用上下文就是一个资源加载器ResourceLoader，资源Resource基本上是JDK内类java.net.URL的更丰富版本，实际上在某些恰当的情境下Resource 的实现就是java.net.URL的封装。Resource 可以持有可以透明地持有底层几乎所有位置的资源，包括类路径、文件系统路径、任意以标准URL描述的位置，以及其他变体。如果资源路径字符串是没有任何前缀简单路径，那么这些资源是明确且适配真实应用上下文类型的。

你可以配置部署在应用上下文的bean实现特定的回调接口ResourceLoaderAware，以自动在初始化时以上下文自身作为ResourceLoader参数回调。你可以暴露出类型Resource的属性，以用于访问静态资源；它们将向其余任意属性一样会被注入。你可以指定那些Resource 属性为简单的字符串路径，并依赖于自动注册进上下文的特殊JavaBean PropertyEditor ，以在bean被部署时转化这些文本字符串成真实的Resource对象。

提供给ApplicationContext 构造函数的资源路径实际上是字符串，具有简单格式并交由特定的上下文实现去处理。ClassPathXmlApplicationContext 将简单路径视为类路径。你也可以使用带有特殊前缀的资源路径以强制从类路径或URL中去加载，忽略实际的上下文类型。

* + - 1. 针对web应用的轻便ApplicationContext实例化

例如你可以使用ContextLoader声明式的创建ApplicationContext 实例。当然你也可以使用ApplicationContext 的某个实现编程式创建ApplicationContext 。

你可以使用ContextLoaderListener 注册一个ApplicationContext ，如下所示：

<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>

监听器检查contextConfigLocation 参数。若该参数不存在，监听器会默认使用/WEB-INF/applicationContext.xml。当该参数存在，监听器将使用预定义的分隔符（逗号，分号和空格）来分割参数字符串，并以此为路径值查找应用上下文配置。同样支持Ant风格的路径模板。如：/WEB-INF/\*Context.xml适用于位于“WEB-INF”内结尾为“Context.xml”所有文件，而/WEB-INF/\*\*/\*Context.xml，则适用于“WEB-INF”下的所有子目录。

* + - 1. 以Java EE RAR文件部署Spring ApplicationContext。

以RAR文件部署Spring ApplicationContext，在一个Java EE RAR的部署单元内封装上下文及其所需要的bean类和JAR库。这相当于引导启动一个独立的ApplicationContext，托管在Java EE环境内，并能够访问Java EE服务器。部署RAR是针对部署无头WAR文件场景的自然替换，实际上，没有任何HTTP进入点的WAR文件仅用于在Java EE环境内引导启动Spring ApplicationContext.

RAR部署是应用上下文的理想模式，不需要HTTP进入点但仅包含消息端和周期任务。在这样一个上下文内的bean可以使用应用服务器资源如JTA事务管理、JNDI绑定的JDBC数据源、JMS连接工厂实例，也可以所有都通过Spring的标准事务管理和JNDI和JMX的支持机制向平台的JMX服务器注册。应用组件也可以通过Spring的TaskExecutor 抽象与应用服务器的JCA工作管理器交互。

涉及RAR部署的配置详情见[SpringContextResourceAdapter](https://docs.spring.io/spring-framework/docs/5.0.14.RELEASE/javadoc-api/org/springframework/jca/context/SpringContextResourceAdapter.html) Javadoc。

作为部署Spring ApplicationContext为Java EE RAR文件的简单示例：将所有的应用类打包为RAR文件，其实是一个jar文件带有不同的扩展名。在RAR文件的根目录添加所有的依赖jar包。添加一个“META-INF/ra.xml”文件为部署说明（如SpringContextResourceAdapter javadoc内所述），及相关的Spring bean定义文件（一般是“META-INF/applicationContext.xml”），并将最终的RAR文件丢尽服务器的部署目录。

如此的RAR部署单元一般是独立的；它们没有向外暴露组件，甚至同一个应用内的其他模块。与基于RAR的ApplicationContext的交互通过与其他模块共享的JMS终端。例如基于RAR的ApplicationContext也会安排一些任务，对文件系统的新文件做出反应（若需要）。若其需要来自外部的同步访问，它可以比如暴露RMI终端，也同样会被同机器上的其他应用模块使用

* + 1. BeanFactory

BeanFactory API提供了Spring IOC机制提供了潜在的基础。它指明了Spring其余部分与相关第三方框架整合的规范，并且其实现DefaultListableBeanFactory 是更高层级容器GenericApplicationContext 内的一个关键代理。

BeanFactory 及其接口如BeanFactoryAware，InitializingBean，DisposableBean ，是其他框架组件的重要整合关键点：不需要任何注解或反射，它们使容器与组件间的交互非常高效。应用层的bean可以使用该回调接口但一般来说用声明式依赖注入会更好，通过注解或编程式设置。

注意核心API BeanFactory 和它的实现DefaultListableBeanFactory 不会假定配置格式或任意组件注解。这些特性都是通过扩展来实现的如：XmlBeanDefinitionReader 和AutowiredAnnotationBeanPostProcessor，基于共享对象BeanDefinition 所表示的元信息上执行对应操作。这就是使得Spring容器如此灵活和易扩展的本质。

如下小节解释了BeanFactory 和ApplicationContext 容器间的区别，以及在引导启动时可能的影响。

* + - 1. BeanFactory还是ApplicationContext？

没有理由不用ApplicationContext ，使用GenericApplicationContext 及其子类实现AnnotationConfigApplicationContext 作为自定义引导启动的常见实现。如下是Spring核心容器的主要入口点，都具有相同的用意：加载配置文件，触发类路径扫描，编程式注册bean定义和注解类，且自5.0起还注册一些功能型bean定义。

因为ApplicationContext 包含了BeanFactory的所有功能，相比普通BeanFactory一般推荐ApplicationContext ，除非在需要完全控制bean处理的场景下。在一个应用上下文ApplicationContext 内如GenericApplicationContext实现，多种bean将会根据规范被检测到（也就是通过bean名或bean类型），尤其是后处理器，而DefaultListableBeanFactory对任意特殊类型的bean都不可知。

对于一些扩展的容器特性例如注解处理和AOP代理，有必要存在[BeanPostProcessor 扩展点](https://docs.spring.io/spring/docs/5.0.14.RELEASE/spring-framework-reference/core.html#beans-factory-extension-bpp)。若你使用普通DefaultListableBeanFactory，那么将不会默认检测和激活后处理器。这样会导致困惑因为你的bean配置无误；在这种场景下就需要容器有更进一步的设置才可完全引导启动。

如下表列出BeanFactory 和ApplicationContext 接口实现所提供的特性。

| *Table 9. Feature Matrix特性表* | | |
| --- | --- | --- |
| **Feature特性** | BeanFactory | ApplicationContext |
| 实例化bean与连接 | Yes | Yes |
| 整合生命周期管理 | No | Yes |
| Bean后处理器自动注册 | No | Yes |
| Bean工厂后处理器自动注册 | No | Yes |
| 消息源MessageSource的轻便访问（用于国际化） | No | Yes |
| 内置ApplicationEvent事件发布机制 | No | Yes |

在DefaultListableBeanFactory内显示注册bean后处理器，你需要编程式调用addBeanPostProcessor：

DefaultListableBeanFactory factory = **new** DefaultListableBeanFactory();

*// 透过bean定义创建工厂*

*// 注册需要的BeanPostProcessor后处理器实例*

factory.addBeanPostProcessor(**new** AutowiredAnnotationBeanPostProcessor());

factory.addBeanPostProcessor(**new** MyBeanPostProcessor());

*// 开始使用工厂*

在普通DefaultListableBeanFactory上应用BeanFactoryPostProcessor，你需要调用其postProcessBeanFactory方法：

DefaultListableBeanFactory factory = **new** DefaultListableBeanFactory();

XmlBeanDefinitionReader reader = **new** XmlBeanDefinitionReader(factory);

reader.loadBeanDefinitions(**new** FileSystemResource("beans.xml"));

*// 从Properties文件中带入属性值*

PropertyPlaceholderConfigurer cfg = **new** PropertyPlaceholderConfigurer();

cfg.setLocation(**new** FileSystemResource("jdbc.properties"));

*// 现在开始执行替换*

cfg.postProcessBeanFactory(factory);

在上述两种情况下，显式注册很不方便，这也就是为什么许多ApplicationContext 变体在基于Spring的应用中要优于普通DefaultListableBeanFactory 的原因，尤其是在传统企业应用的设置内，容器的扩展功能依赖于BeanFactoryPostProcessor和BeanPostProcessors时。

AnnotationConfigApplicationContext具有所有常见的开箱即用注解后处理器，并可以通过配置注解@EnableTransactionManagement在底层带入额外的后处理器。在Spring基于注解配置模型的抽象层，bean后处理器仅仅是容器内部细节的一个概念。

## 资源

* + 1. 简介

Java标准java.net.URL类和多种URL前缀标准处理程序并不能很好的适配底层资源的访问。例如，没有标准化的URL 实现用于访问从类路径或相对应ServletContext的资源。虽然可以给特定的URL 前缀注册新的处理程序（类似http:前缀的已存在处理程序），但这是比较复杂的，且URL 接口仍然缺少一些必不可少的功能，如检查指向的资源是否存在。

* + 1. Resource接口

Spring的Resource 接口即是用于对底层资源抽象访问功能更强大的接口。

**public** **interface** **Resource** **extends** InputStreamSource {

**boolean** exists();

**boolean** isOpen();

URL getURL() **throws** IOException;

File getFile() **throws** IOException;

Resource createRelative(String relativePath) **throws** IOException;

String getFilename();

String getDescription();

}

**public** **interface** **InputStreamSource** {

InputStream getInputStream() **throws** IOException;

}

Resource 接口最重要的几个方法是：

* getInputStream()：定位并打开资源，返回读取资源的InputStream。每一次调用都应返回一个新的输入流InputStream。关闭流操作交给调用方。
* exists()：返回一个boolean 值指明此资源是否真实存在。
* isOpen()：返回一个boolean 值指明该资源是否由一个已打开的流持有。若是true，则该InputStream不能被多次读，仅可读一次然后需要关闭流以防止内存泄漏。对于所有常见的资源实现，都将为false ，但InputStreamResource除外。
* getDescription()：返回对该资源的描述，用于在使用该资源时的错误输出。通常是资源的全限定文件名或者真实URL。

其他方法允许你持有表示某资源的真实URL 或File 对象（如果相应的实现兼容且支持该功能）。

Resource 抽象在Spring内部广泛运用，在许多方法需要资源时作为方法入参。在一些Spring API内的其他方法（如多种ApplicationContext 实现的构造函数），通过一个简单的String 来创建适用于该context实现的Resource 实现，或者通过有特殊前缀的String 路径。允许调用者指定特定的Resource 实现创建和使用。

鉴于Resource 接口在Spring内的大量使用，将其作为你代码中的通用工具类用于访问资源是有一定意义的，即使在当你的代码不需要知道或关心Spring的其余部分时。但这将使的代码与Spring耦合，仅与一小部分能替代URL的具备更丰富功能的工具类耦合，且可以等效为因此引入的其他库。

需要注意Resource 抽象不会替换已有的功能：它尽可能的进行封装。例如UrlResource封装了一个URL，并使用被封装的URL 来完成其功能。

* + 1. 内置Resource实现

Spring内置一些可开箱即用的Resource 实现。

* + - 1. UrlResource

UrlResource 封装了java.net.URL，并可用于访问任意能够通过URL访问的资源，例如文件，HTTP目标，FTP目标等等。所有的URL都有标准的String 表示，标准的前缀能指明URL类型。包括file:用于访问文件系统路径，http:用于访问通过HTTP协议的资源，ftp:用于通过FTP访问资源等等。

UrlResource 通过Java代码使用UrlResource 构造函数来显示创建，但一般会在你调用以一个表示资源路径的String 入参API方法时隐式创建。对于后者，JavaBean PropertyEditor 会最终决定哪个类型的Resource 应该被创建。如果路径字符串包含一些已知的前缀（对该JavaBean已知）如classpath:，其将对该前缀创建合适的Resource。然而，若无法识别前缀，它将假设这是一个标准的URL字符串并创建UrlResource。

* + - 1. ClassPathResource

这个类表示了从类路径中获取的资源。它使用当前线程上下文类加载器，或给定的类加载器，或用于加载资源的给定类。

这个Resource 实现支持在类路径资源在文件系统内时解析为java.io.File，但当资源文件在jar内且没有被解压到文件系统（通过servlet引擎或者其他环境）时不适用。为了解决这个问题，各个Resource 实现始终支持解析为java.net.URL。

ClassPathResource 由Java代码使用ClassPathResource 构造函数显式创建，但通常在调用以一个表示资源路径的String 入参的API方法时隐式创建。后者JavaBean PropertyEditor将识别在字符串路径中的特殊前缀classpath:，创建ClassPathResource 。

* + - 1. FileSystemResource

这是Resource 实现用于处理java.io.File，显然其支持解析为File和URL。

* + - 1. ServletContextResource

这是对ServletContext 资源的Resource 实现，在对应的web应用根目录内解析相对路径。

这个总是支持支持流访问和URL访问，当且仅当web应用文档被展开且资源物理存在于文件系统时其允许java.io.File访问。不管是如上述在文件系统中展开，还是直接从JAR中访问，亦或是从其他地方如DB（可能存在的）具体都要依赖于Servlet容器。

* + - 1. InputStreamResource

给定InputStream的Resource 实现。仅在没有其他Resource 实现可用的情况下使用。尤其是如果可以，更倾向于使用ByteArrayResource 或任意基于文件的Resource 实现。

相比其他Resource 实现，这是对一个已经打开的资源描述-因此isOpen()的返回值是true。若你需要保存此资源描述在某处或你需要多次读流，不要使用它

* + - 1. ByteArrayResource

这是对给定字节数组的Resource 实现。对给定字节数组创建一个ByteArrayInputStream 。在从给定字节数组加载内容时有用，不需要使用只能使用一次的InputStreamResource。

* + 1. ResourceLoader资源加载器

ResourceLoader接口旨在被实现为通过对象可返回（或加载）Resource 实例。

**public** **interface** **ResourceLoader** {

Resource getResource(String location);

}

所有的应用上下文容器都实现了ResourceLoader 接口，因此所有的应用上下文都可用于获得Resource 实例。

当在指定的应用上下文上调用getResource()时，资源位置路径不需要特定的前缀，你将获得一个适用于该应用上下文的Resource 类型对象。例如，ClassPathXmlApplicationContext 假设下述代码片段在该实例上执行：

Resource template = ctx.getResource("some/resource/path/myTemplate.txt");

该方法将返回ClassPathResource；若由FileSystemXmlApplicationContext 实例执行同一个方法，你将获得一个FileSystemResource。对于WebApplicationContext，你将获得ServletContextResource，以此类推。

如上，你可以特定风格加载资源对应恰当的应用上下文。

换句话说，你也可以强制使用ClassPathResource，忽略应用上下文类型，通过指定特定的前缀classpath:：

Resource template = ctx.getResource("classpath:some/resource/path/myTemplate.txt");

相似的，通过指定标准java.net.URL前缀可以强制使用UrlResource

Resource template = ctx.getResource("file:///some/resource/path/myTemplate.txt");

Resource template = ctx.getResource("https://myhost.com/resource/path/myTemplate.txt");

如下表格总结了String向Resource转化的策略：

| *Table 10. Resource strings* | | |
| --- | --- | --- |
| **Prefix** | **Example** | **Explanation** |
| classpath: | classpath:com/myapp/config.xml | 从类路径加载 |
| file: | [file:///data/config.xml](file:///\\data\config.xml) | 从文件系统作为URL加载 [[3](https://docs.spring.io/spring/docs/5.0.14.RELEASE/spring-framework-reference/core.html#_footnote_3)] |
| http: | <https://myserver/logo.png> | 作为URL加载. |
| (none) | /data/config.xml | 取决于ApplicationContext. |

* + 1. ResourceLoaderAware接口

ResourceLoaderAware 接口用于标记需要提供ResourceLoader 引用的组件，是特殊回调接口：

**public** **interface** **ResourceLoaderAware** {

**void** setResourceLoader(ResourceLoader resourceLoader);

}

当类实现了ResourceLoaderAware 且其被部署在应用上下文内（作为被Spring管理的bean），其将被容器识别为ResourceLoaderAware。方法setResourceLoader(ResourceLoader)将被容器调用，将其作为入参提供给该方法（记住，所有的Spring应用上下文都实现了ResourceLoader 接口）。

当然，由于ApplicationContext 是一个ResourceLoader，ApplicationContextAware同样可以被bean实现并用提供的应用上下文直接加载资源，但通常来说，专用的ResourceLoader 接口会更好如果这是所需要的。代码将仅与资源加载接口，可以视为一个工具接口，而不是一整个Spring的ApplicationContext 接口。

自Spring2.5起，你可以对ResourceLoader 依赖注入而不是实现ResourceLoaderAware 接口。“传统”constructor和byType 自动装配模式（如[Autowiring collaborators](https://docs.spring.io/spring/docs/5.0.14.RELEASE/spring-framework-reference/core.html#beans-factory-autowire)中所描述）现在能够提供ResourceLoader 类型依赖，给构造函数参数或setter方法参数。为了更灵活（包括自动装配成员变量和多参数方法），考虑使用基于注解的新自动装配特性。在这种情况下，ResourceLoader将会被自动装配入成员变量，构造函数参数，或方法为ResourceLoader 类型的参数，只要对应的成员变量、构造函数和方法带有@Autowired注解。更多详见[@Autowired](https://docs.spring.io/spring/docs/5.0.14.RELEASE/spring-framework-reference/core.html#beans-autowired-annotation)。

* + 1. 资源作为依赖

若bean需要通过某些动态途径决定并提供资源路径，可以使用ResourceLoader 接口来加载资源。例如，加载某种类型的模板，其中所需的特定资源取决于用户的角色。若为静态资源，有必要完全排除ResourceLoader 接口的使用，仅需要bean暴露出其所需的Resource 属性，并等待后续注入。

使得注入变得微不足道是所有的应用上下文都注册并使用特殊JavaBean PropertyEditor，用于转化String 路径为Resource 对象。因此若myBean 有模板为Resource类型的属性，其可配置为资源的简单字符路径，如下：

<bean id="myBean" class="...">

<property name="template" value="some/resource/path/myTemplate.txt"/>

</bean>

注意资源路径没有前缀，因此应用上下文会作为ResourceLoader被使用，资源会通过ClassPathResource，FileSystemResource，或ServletContextResource（适用的话）被加载，具体类型取决于上下文类型。

若需要强制指定特定Resource类型，则可使用对应前缀。如下两例展示了强制使用ClassPathResource 和UrlResource（后者用于访问文件系统文件）

<property name="template" value="classpath:some/resource/path/myTemplate.txt">

<property name="template" value="file:///some/resource/path/myTemplate.txt"/>

* + 1. 应用上下文和资源路径
       1. 构建应用上下文

一个应用上下文构造函数（对应应用上下文类型）一般接收资源路径字符串或字符串数组如XML用于组成上下文的定义。

当一个资源路径没有前缀，从路径中构建出来并被用于加载bean定义的Resource 类型取决于对应的应用上下文。例如，若你创建一个ClassPathXmlApplicationContext ：

ApplicationContext ctx = **new** ClassPathXmlApplicationContext("conf/appContext.xml");

bean定义将从类路径中加载，将使用ClassPathResource。但如果你创建了一个文件系统上下文FileSystemXmlApplicationContext：

ApplicationContext ctx = **new** FileSystemXmlApplicationContext("conf/appContext.xml");

Bean定义将加载至文件系统位置，在这个条件下是当前工作目录。

注意在路径内特殊类路径前缀或标准URL前缀的使用将会覆盖默认创建来加载定义Resource的类型。因此这个FileSystemXmlApplicationContext

ApplicationContext ctx =

**new** FileSystemXmlApplicationContext("classpath:conf/appContext.xml");

将会实际上从类路径加载其bean定义。然而其仍为FileSystemXmlApplicationContext。若随后其作为ResourceLoader使用，任意无前缀路径将被视为文件系统路径。

构建ClassPathXmlApplicationContext实例-捷径

ClassPathXmlApplicationContext 暴露了一些构造函数用于启用简便的实例化。基本思想是只需要提供包含XML文件名的字符串数组（不需要前缀的路径信息），然后再提供一个Class；ClassPathXmlApplicationContext 将从提供的class解析。如下例，假设目录关系为：

com/

foo/

services.xml

daos.xml

MessengerService.class

ClassPathXmlApplicationContext 实例由'services.xml'和'daos.xml'内定义的bean组成，实例化如下：

ApplicationContext ctx = **new** ClassPathXmlApplicationContext(

**new** String**[]** {"services.xml", "daos.xml"}, MessengerService.class);

查看ClassPathXmlApplicationContext javadoc获取有关构造函数更多信息。

* + - 1. 应用上下文资源路径通配

在应用上下文构造函数中资源路径的值可以是简单（如上）路径，一对一匹配对应的资源，或者可以包含特殊前缀“classpath:”和内含Ant风格的正则表达式（使用Spring的PathMatcher 工具去匹配）。后两者都是有效的通配符手段。

此机制的用法之一是用于执行组件风格的应用组装时。所有的组件可以“发表”上下文定义片段到对应已知的资源路径，然后当最终的应用上下文被由同一路径带有前缀classpath\*:的资源创建，所有的组件片段将自动被识别合并。

注意通配符仅针对于应用上下文构造函数的资源路径（或者在调用时直接使用PathMatcher 工具类），且其在构造时解析。与Resource 类型没有联系，不可能通过使用classpath\*:前缀即可构建出一个真实的Resource，一个资源接入点一次只能对应一个资源。

Ant风格范式

当路径包含Ant风格范式，如：

/WEB-INF/\*-context.xml

com/mycompany/\*\*/applicationContext.xml

file:C:/some/path/\*-context.xml

classpath:com/mycompany/\*\*/applicationContext.xml

解析器遵循更复杂但已定义好的程序去解析通配符。其所提供对应路径的资源取决于最后一个非通配符段并从中获取一个URL。如果这个URL不是jar: URL或者容器指定变体（例如WebLogic中的zip:，WebSphere中的wsjar等等），将由此获得一个java.io.File且通过遍历文件系统用于解析通配符。在jar URL的情况下，解析器会由此取得java.net.JarURLConnection或者手动解析匹配jar URL然后遍历jar文件内的内容来解析通配符。

可移植性的含义

若指定的路径已经是个文件URL（显式或隐式由于基ResourceLoader 是文件系统资源加载器），那么通配符保证能够在一个完全可移植的条件下使用。

若指定路径是类路径，那么解析器必须通过调用Classloader.getResource()获取最后一个非通配路径URL段。由于这只是路径的一个节点（非最终文件），实际上未定义哪种类型URL应该被返回（见ClassLoader javadocs）。实际上，在类路径资源在文件系统目录内解析时，永远是以java.io.File来表示目录，或者在类路径资源在jar位置，则是一个jar URL。然而可移植性问题仍存在。

若一个jar URL持有最后一个非通配路径段，解析器由此获取java.net.JarURLConnection，或手动解析jar URL以能够遍历jar的内容并解析通配符。这在大多数环境下适用，但会在另一些环境中失败。强烈建议jar内资源的通配符解析应在你依赖其开发前在对应的环境上进行过全面的测试。

classpath\*:前缀

当构建一个基于XML的应用上下文时，路径字符串使用特殊classpath\*:前缀：

ApplicationContext ctx =

**new** ClassPathXmlApplicationContext("classpath\*:conf/appContext.xml");

这个前缀指明所有匹配该名字的类路径资源都应该被获取（由内部执行，基本上通过调用方法ClassLoader.getResources(…​)实现），然后合并以形成最终的应用上下文定义。

类路径通配符依赖于对应类加载器的getResources()方法。当前大多数应用服务器都提供自实现的类加载器，加载器行为会有所不同尤其是在处理jar文件时。一个简单地测试classpath\*是否有效的例子是，使用类加载器加载类路径jar包内的文件：getClass().getClassLoader().getResources("<someFileInsideTheJar>")。使用同名但不同位置的文件来测试上述测试用例。若返回了不正确的结果，检查应用服务器文档查找可能影响类加载器行为的相关设置。

classpath\*:前缀可以在剩下的路径中结合PathMatcher范式，例如classpath\*:META-INF/\*-beans.xml。在这种情况下，解析策略相当简单：最后一个非通配符路径段上调用ClassLoader.getResources()来获取所有匹配在类加载器层次结构中的资源，然后对于每一个资源都使用上述PathMatcher解析策略来解析通配子路径。

有关通配符的其他注意事项

注意在classpath\*:与Ant风格范式的生效依赖于在范式开始前至少有一个根目录，除非实际目标文件在文件系统内。也就是说如classpath\*:\*.xml的范式不能从jar文件根目录获取文件但只能从扩展开的根目录获取。

Spring检索类路径条目的能力来自于JDK的ClassLoader.getResources()方法，其以一个空字符串入参（意指检索根目录）返回文件系统位置。Spring也会查看URLClassLoader运行时配置和jar文件内的“java.class.path”清单但不能保证具有可移植行为

扫描类路径包需要类路径内存在相关目录条目。当你通过Ant构建jar包时，确保不要激活jar任务的files-only开关。同样。类路径目录在某些环境下基于安全策略不能向外暴露，例如JDK1.7.0\_45和更高版本的独立应用（需要在清单manifests内设置信任库‘Trusted-Library’；见<https://stackoverflow.com/questions/19394570/java-jre-7u45-breaks-classloader-getresources>）。

在JDK9的模块路径内（Jigsaw），Spring的类路径扫描一般能正常执行。在此强烈推荐将资源放入专用目录，避免前述的在检索jar文件根目录时的可移植性问题。

若检索的根包在多个类路径目录中有效，带有classpath:的Ant风格范式不保证能否找到匹配的资源。因为假设有资源如：

com/mycompany/package1/service-context.xml

文件位置是唯一的，但如果路径如：

classpath:com/mycompany/\*\*/service-context.xml

是被解析对象，解析器会在getResource("com/mycompany")返回的（第一个）URL下工作；若此基础包节点存在于多个类加载器位置，实际的最终资源可能不在该位置下。因此，在这种情况下更倾向于使用“classpath\*:”搭配相同的Ant风格范式，由此会搜索所有包含根包的类路径位置。

* + - 1. 文件系统资源警告

FileSystemResource没有附带在一个FileSystemApplicationContext（也就是说，一个FileSystemApplicationContext不是实际的ResourceLoader）将以你的配置来识别路径为绝对或相对。相对路径即相对于当前工作目录，而绝对目录是相对文件系统的根目录。

由于向后兼容（历史）的原因，当FileSystemApplicationContext是ResourceLoader时与上不同。FileSystemApplicationContext 强制所有附带的FileSystemResource 实例将所有位置路径视为相对的，不管路径是否以斜杠开始。实际上，以下等效：

ApplicationContext ctx =

**new** FileSystemXmlApplicationContext("conf/context.xml");

ApplicationContext ctx =

**new** FileSystemXmlApplicationContext("/conf/context.xml");

如下所示：（即使它们是不同的，一个是相对路径另一个是绝对路径）

FileSystemXmlApplicationContext ctx = ...;

ctx.getResource("some/resource/path/myTemplate.txt");

FileSystemXmlApplicationContext ctx = ...;

ctx.getResource("/some/resource/path/myTemplate.txt");

实际上，如果真需要文件系统绝对路径，最好放弃使用FileSystemResource /FileSystemXmlApplicationContext，只需要通过使用file:URL前缀强制UrlResource。

*// 真实上下文类型不重要, Resource总会是UrlResource*

ctx.getResource("file:///some/resource/path/myTemplate.txt");

*// 强制此FileSystemXmlApplicationContext通过UrlResource加载其定义*

ApplicationContext ctx =

**new** FileSystemXmlApplicationContext("file:///conf/context.xml");

## 校验，数值绑定，和类型转换

* + 1. 介绍

JSR-303/JSR-349 Bean校验

Spring框架4.0从设置支持上来说支持Bean校验1.0 (JSR-303)和Bean校验1.1 (JSR-349)，并将其与Spring的 Validator 接口适配。

应用可选择全局启用一次Bean校验，如[Spring Validation](https://docs.spring.io/spring/docs/5.0.14.RELEASE/spring-framework-reference/core.html#validation-beanvalidation)中所描述，并将其专用于所有校验需求。

应用也可给每一个 DataBinder 实例注册额外的Spring Validator 实例，如[Configuring a DataBinder](https://docs.spring.io/spring/docs/5.0.14.RELEASE/spring-framework-reference/core.html#validation-binder)中所描述。不需使用注解即可接入校验逻辑。

有一些属性和构造函数将校验视为业务逻辑，Spring提供用于不排除其中任何一种校验（和数值绑定）的设计。尤其是校验不应该与web层绑定，且应轻易本地化并可插入任意可用的校验器。考虑上述情况，Spring设计出Validator接口可以既底层又可在应用各层使用。

数值绑定用于允许用户输入与应用域模型动态绑定（或任何用于处理用户输入的对象）。Spring提供DataBinder 来专门处理。Validator 和DataBinder 组成validation 包，其基本用于但不仅限于MVC框架。

BeanWrapper 是Spring框架用于多处的一个基础概念。然而不大可能直接用到BeanWrapper。因为这是参考文档，我们觉得有些解释有必要。我们会在本章解释BeanWrapper，你会很大可能在尝试绑定数值到对象是使用它。

Spring的数值绑定和底层的BeanWrapper都是用属性编辑器PropertyEditors来解析匹配格式化属性值。PropertyEditor 概念是JavaBean规范的一部分，同样也将在本章解释。Spring 3介绍了“core.convert”包，提供通用的类转换工具，就如用于格式化UI域值得高层级的“format”包。这些新包可以简单替代PropertyEditor，且将会在本章介绍。

* + 1. 使用Spring的Validator接口校验

Spring的Validator 接口可用于校验对象。该接口与Errors对象协同工作，当校验器校验失败时可将结果反映至Errors对象。

假设有如下数据对象：

**public** **class** **Person** {

**private** String name;

**private** **int** age;

*// 常规getters和setters...*

}

我们将通过实现org.springframework.validation.Validator接口的两个方法来给类Person 提供校验行为：

* supports(Class)- 此Validator 可否校验此类Class的实例？
* validate(Object, org.springframework.validation.Errors) – 校验给定对象并根据校验错误将其注册进给定的Errors 对象

实现一个Validator 相当简单，尤其是当你知道Spring框架提供的ValidationUtils帮助类。

**public** **class** **PersonValidator** **implements** Validator {

*/\*\**

*\* 此校验器仅校验Person实例*

*\*/*

**public** **boolean** supports(Class clazz) {

**return** Person.class.equals(clazz);

}

**public** **void** validate(Object obj, Errors e) {

ValidationUtils.rejectIfEmpty(e, "name", "name.empty");

Person p = (Person) obj;

**if** (p.getAge() < 0) {

e.rejectValue("age", "negativevalue");

} **else** **if** (p.getAge() > 110) {

e.rejectValue("age", "too.darn.old");

}

}

}

如你所见，在ValidationUtils 类的static rejectIfEmpty(..) 方法用于在'name'属性为null 或空字符串时提示拒绝。查看ValidationUtils javadocs以了解该类提供的功能。

可以实现一个单一Validator 类来校验富对象内的每一个内嵌对象，最好封装每一个内嵌对象类的校验逻辑在其自己的Validator 实现内。一个“富”对象的例子是一个Customer 由两个String 属性（姓与名）和一个复杂的Address 对象组成。Address 对象可独立于Customer 对象使用，所以需要实现一个独立的AddressValidator。若你希望CustomerValidator 重用包含在AddressValidator 类内的逻辑而不需要复制粘贴，可以在CustomerValidator内通过依赖注入或实例化AddressValidator 并使用如下：

**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;

}

*/\*\**

*\* 此校验器校验Customer实例,和任意其他Customer的子类*

*\*/*

**public** **boolean** supports(Class clazz) {

**return** Customer.class.isAssignableFrom(clazz);

}

**public** **void** validate(Object target, Errors errors) {

ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName", "field.required");

ValidationUtils.rejectIfEmptyOrWhitespace(errors, "surname", "field.required");

Customer customer = (Customer) target;

**try** {

errors.pushNestedPath("address");

ValidationUtils.invokeValidator(this.addressValidator, customer.getAddress(), errors);

} **finally** {

errors.popNestedPath();

}

}

}

校验错误在Errors 对象内保存传递给校验器。在Spring Web MVC你可以使用<spring:bind/>标签来检查错误信息，当然你可以自己检查错误对象。更多有关其提供的方法信息可在javadocs内查看

* + 1. 解析代码为错误信息

我们已经讲述了数值绑定和校验。有关校验错误的信息输出是最后需要讨论的。上例中，拒绝了name 和age 成员变量。若我们需要通过MessageSource输出错误信息，可在拒绝成员变量时（在这里是“name”和“age”）使用已得的错误码。当调用（直接或间接使用上例的ValidationUtils 类）rejectValue 或从Errors 接口的其他reject 方法时，潜在实现不会仅注册你已传入的代码，还有一系列额外的错误代码。注册何种错误代码取决于使用的MessageCodesResolver。默认情况下，使用DefaultMessageCodesResolver，其不仅注册你给定的错误码，还有传递给reject方法的成员变量名等相关信息。所以就是当使用rejectValue("age", "too.darn.old")拒绝一个成员变量，除了too.darn.old代码，Spring将还注册too.darn.old.age和too.darn.old.age.int（第一个包含成员变量名，第二个包含成员变量类型）；方便开发者定位错误信息。

更多有关MessageCodesResolver 和默认策略可以查看在线[MessageCodesResolver](https://docs.spring.io/spring-framework/docs/5.0.14.RELEASE/javadoc-api/org/springframework/validation/MessageCodesResolver.html)和[DefaultMessageCodesResolver](https://docs.spring.io/spring-framework/docs/5.0.14.RELEASE/javadoc-api/org/springframework/validation/DefaultMessageCodesResolver.html) javadocs。

* + 1. Bean控制和Bean包裹BeanWrapper

org.springframework.beans遵循Oracle提供的JavaBean标准。JavaBean就是一个有默认无参构造函数的简单类，且遵循传统命名规范即有属性名如bingoMadness 即有对应的setter方法setBingoMadness(..)和getter方法getBingoMadness()。更多有关JavaBean的信息和规范，参考Oracle官网（[javabeans](https://docs.oracle.com/javase/6/docs/api/java/beans/package-summary.html)）。

在beans包内一个重要的类是BeanWrapper 接口及其相关实现（BeanWrapperImpl）。如javadocs中所提到，BeanWrapper 提供设置和获取属性值（单一或多个）的功能，获取属性描述和查看属性是否可读或可写。BeanWrapper 也支持内嵌属性，可以设置子属性的属性，深度不限制。然后BeanWrapper 支持给标准JavaBean添加属性变化监听器PropertyChangeListeners 和VetoableChangeListeners，不需要在目标类中添加对应的支持代码。最后，BeanWrapper还支持索引属性的设置。BeanWrapper 一般不会直接被应用代码使用，而是由DataBinder 和bean工厂BeanFactory使用。

BeanWrapper 的工作方式如其名字所示：其包裹一个bean以对该bean实行对应的行为，如设置或者获取属性。

* + - 1. 设置和获取基础和内嵌属性

可以通过setPropertyValue(s)和getPropertyValue(s)方法以及一组重载变量来设置和获取属性值。详见Spring的javadocs。重要的是要知道指明对象属性的规范，如

| *Table 11. 属性示例* | |
| --- | --- |
| **表达式** | **解释** |
| name | 指属性 name 相关的 getName() 或 isName() 和 setName(..)方法 |
| account.name | 指属性 account的内嵌属性name 相关联的方法如 getAccount().setName()或 getAccount().getName() |
| account[2] | 指索引属性account的第三个元素. 索引属性可以是 array, list or 或其他自排序集合 |
| account[COMPANYNAME] | 指Map属性 account 以key *COMPANYNAME* 索引得出的值 |

如下可以看到使用BeanWrapper 来获取和设置属性的示例。

（若你不打算直接使用*BeanWrapper* 则如下小节并不十分重要。若你只是使用*DataBinder*和*BeanFactory*及其开箱即用实现，你可以直接跳到*PropertyEditors*相关小节）

如下类所示：

**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;

}

}

如下代码片段显示了如何获取和控制已实例化的Companies和Employees的属性：

BeanWrapper company = **new** BeanWrapperImpl(**new** Company());

*// 设置company name..*

company.setPropertyValue("name", "Some Company Inc.");

*// ... 也可以这么做:*

PropertyValue value = **new** PropertyValue("name", "Some Company Inc.");

company.setPropertyValue(value);

*// 然后让我们创建领导人并将其绑定到company:*

BeanWrapper jim = **new** BeanWrapperImpl(**new** Employee());

jim.setPropertyValue("name", "Jim Stravinsky");

company.setPropertyValue("managingDirector", jim.getWrappedInstance());

*// 通过company获取managingDirector的薪水*

Float salary = (Float) company.getPropertyValue("managingDirector.salary");

* + - 1. 内建PropertyEditor实现

Spring以PropertyEditors 的概念来描述Object 和String之间的转换。仔细想想，它有时候会比一个对象更方便地表示一个属性。例如，Date可以以更易读的方式显示（如String '2007-14-09'），然而我们还可以将易读格式转换回原始日期对象（或者：转换任意输入的易读格式为Date 对象）。这个行为可以通过注册自定义java.beans.PropertyEditor类型的的编辑器来达成。注册自定义编辑器在BeanWrapper或在上述章节中描述的对应IoC容器内，以此可将属性转化为所需的类型。详见Oracle提供的java.beans包内PropertyEditors 的javadocs。

如下是在Spring内使用属性编辑的例子：

* 使用PropertyEditors设置bean内属性。当声明在XML文件中某个bean的属性值被设为String时，Spring会（若该属性相关setter有Class 入参）使用ClassEditor尝试将参数解析为Class对象。
* Spring MVC框架内解析HTTP请求参数是通过各种PropertyEditors来完成的，你可以在CommandController的所有子类内手动绑定。

Spring自带部分PropertyEditors ，使得Spring更简单易用。如下表所示，它们都位于org.springframework.beans.propertyeditors包内。大部分但非全部（下表已指出）都通过BeanWrapperImpl默认注册。属性编辑器可以通过多种方式配置，你可以通过注册自定义编辑器以覆盖默认值：

| *Table 12. 内置属性编辑器PropertyEditors* | |
| --- | --- |
| **Class** | **Explanation** |
| ByteArrayPropertyEditor | 字节数组编辑器。字符串会简单转化为相对应的字节数组表示。默认通过BeanWrapperImpl注册. |
| ClassEditor | 解析字符串表示的类为对应真实的类，反之亦然。当找不到类时，抛出异常 IllegalArgumentException。默认通过BeanWrapperImpl注册。 |
| CustomBooleanEditor | 对Boolean属性自定义属性编辑器。默认通过BeanWrapperImpl注册但可通过注册自定义实例覆盖。 |
| CustomCollectionEditor | 集合属性编辑器，转化任意源集合为给定的集合类型。 |
| CustomDateEditor | java.util.Date的可自定义属性编辑器，支持自定义日期格式 DateFormat。非默认注册。用户按需注册并指定格式。 |
| CustomNumberEditor | 可自定义Number子类编辑器如 Integer, Long, Float, Double. 通过BeanWrapperImpl默认注册，但可通过注册自定义实例覆盖。 |
| FileEditor | 可将字符串解析为 java.io.File 对象。默认通过 BeanWrapperImpl注册。 |
| InputStreamEditor | 单向属性编辑器，可以将文本字符串输出为(直接通过ResourceEditor和Resource)一个 InputStream，因此 InputStream 属性可以直接设置为字符串。注意使用时默认不关闭流InputStream ！默认通过BeanWrapperImpl注册。 |
| LocaleEditor | 可将字符串解析为 Locale 对象，反之亦然(字符串格式是 *[国家]*[变量], 这也是Locale的toString()方法输出格式). 默认通过BeanWrapperImpl注册。 |
| PatternEditor | 可将字符串解析为 java.util.regex.Pattern对象，反之亦可 |
| PropertiesEditor | 可将字符串解析为(格式化使用定义在 java.util.Properties 类内的格式) Properties 对象。默认通过BeanWrapperImpl注册。 |
| StringTrimmerEditor | 缩减字符串的属性编辑器。选择性允许转换空字符串为 null 值。非默认注册；用户自行据需注册。 |
| URLEditor | 可将字符串表示的URL 转化为真实 URL 对象。默认通过BeanWrapperImpl注册。 |

Spring使用java.beans.PropertyEditorManager来给所需的属性编辑器设置检索路径。检索路径包括sun.bean.editors，其包括了类型如Font，Color和其余大部分基础类型的实现类。注意，标准的javaBean结构会自动发现PropertyEditor 类（无需显式注册），前提是他们的类在同一个包内，且类名相同带有'Editor'后缀；例如，有如下类和包结构，满足FooEditor 类被识别并作为Foo类型属性的PropertyEditor 。

com

chank

pop

Foo

FooEditor // the PropertyEditor for the Foo class

注意，在此处你也可以使用标准BeanInfo javaBeans机制（如[in not-amazing-detail here](https://docs.oracle.com/javase/tutorial/javabeans/advanced/customization.html)所述）。如下例所示，使用BeanInfo 机制显式注册一个或多个相关类属性的PropertyEditor 实例。

com

chank

pop

Foo

FooBeanInfo // the BeanInfo for the Foo class

如下Java源码引用FooBeanInfo 类。此举将Foo的age属性与CustomNumberEditor 关联。

**public** **class** **FooBeanInfo** **extends** SimpleBeanInfo {

**public** PropertyDescriptor**[]** getPropertyDescriptors() {

**try** {

**final** PropertyEditor numberPE = **new** CustomNumberEditor(Integer.class, true);

PropertyDescriptor ageDescriptor = **new** PropertyDescriptor("age", Foo.class) {

**public** PropertyEditor createPropertyEditor(Object bean) {

**return** numberPE;

};

};

**return** **new** PropertyDescriptor**[]** { ageDescriptor };

}

**catch** (IntrospectionException ex) {

**throw** **new** Error(ex.toString());

}

}

}

注册额外的自定义属性编辑器

当bean属性设为字符串值，Spring IoC容器最终使用标准javaBeans PropertyEditors 来将这些字符串转换为属性的复杂类型。Spring预注册了一些自定义PropertyEditors（例如，转化以字符串表示的类名为真实Class对象）。另外，Java的标准JavaBeans PropertyEditor 查询机制允许一个类的PropertyEditor (被命名为合适的名字（如上述）并与类位于同一个包内)被自动发现。

若需要注册其他自定义PropertyEditors，有一些机制适用。最手动化的方法，通常不方便或不受推荐，即使用ConfigurableBeanFactory 接口的registerCustomEditor()方法，在你有一个BeanFactory 引用的情况下。另外，稍微方便些，使用特殊的bean工厂后处理器CustomEditorConfigurer。虽然bean工厂后处理器作用域BeanFactory 实现，自定义编辑器配置器CustomEditorConfigurer有内嵌属性可以设置，强烈建议其与ApplicationContext一起使用，因此其可以与其他bean一样部署、被检测与应用。

注意，所有的bean工厂和应用上下文都自动使用一些内建的属性编辑器，通过BeanWrapper 来处理属性转换。BeanWrapper注册的标准属性编辑器在[上一节](https://docs.spring.io/spring/docs/5.0.15.RELEASE/spring-framework-reference/core.html#beans-beans-conversion)中已列出。另外，ApplicationContexts 也覆盖或者添加了一些额外的编辑器来处理资源检索，适配于特定的应用上下文类型。

标准JavaBeans PropertyEditor 实例用于转换字符串表示的属性值为真实复杂类型的属性。CustomEditorConfigurer，bean工厂后处理器，可用于方便地为ApplicationContext提供额外的PropertyEditor实例。

如下有ExoticType类，另一个DependsOnExoticType 类需要ExoticType 作为属性：

**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;

}

}

当属性设置正确时，我们期待可以给type属性分配为字符串，这样PropertyEditor 会在将其转换为真实的ExoticType 实例：

<bean id="sample" class="example.DependsOnExoticType">

<property name="type" value="aNameForExoticType"/>

</bean>

PropertyEditor 实现类似如：

*// 转换字符串表示为ExoticType对象*

**package** example;

**public** **class** **ExoticTypeEditor** **extends** PropertyEditorSupport {

**public** **void** setAsText(String text) {

setValue(**new** ExoticType(text.toUpperCase()));

}

}

然后用CustomEditorConfigurer 注册新PropertyEditor到ApplicationContext，接着它将如预期般被使用：

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

<property name="customEditors">

<map>

<entry key="example.ExoticType" value="example.ExoticTypeEditor"/>

</map>

</property>

</bean>

使用属性编辑器注册器PropertyEditorRegistrars

另一注册属性编辑器的机制是创建和使用PropertyEditorRegistrar。这个接口在你需要在不同条件下使用同一套属性编辑器时尤其有用：编写一个相关的注册器并在每种情况下重用。属性编辑器注册器PropertyEditorRegistrars 与PropertyEditorRegistry接口协同发挥作用，其由Spring的BeanWrapper（和DataBinder）实现。

PropertyEditorRegistrars在结合CustomEditorConfigurer（如上所述）使用时十分方便，其暴露了一个属性setPropertyEditorRegistrars(..)：将一个属性编辑器注册器PropertyEditorRegistrars添加以此添加到PropertyEditorRegistrars可以轻松地与DataBinder 和Spring MVC Controllers共享。并且，其避免了需要对自定义编辑器的同步操作：PropertyEditorRegistrar 对每一个bean的创建都创建出一个新的PropertyEditor 实例。

使用PropertyEditorRegistrar 或许最好用一个例子来说明。首先你需要创建一个自实现的PropertyEditorRegistrar：

**package** com.foo.editors.spring;

**public** **final** **class** **CustomPropertyEditorRegistrar** **implements** PropertyEditorRegistrar {

**public** **void** registerCustomEditors(PropertyEditorRegistry registry) {

*// 期望创建新PropertyEditor实例*

registry.registerCustomEditor(ExoticType.class, **new** ExoticTypeEditor());

*// 你可据需注册自定义属性编辑器*

}

}

也可参考org.springframework.beans.support.ResourceEditorRegistrar对PropertyEditorRegistrar 的实现。注意其在registerCustomEditors(..)方法的实现中对每一个属性编辑器如何创建新实例。

然后我们配置CustomEditorConfigurer并注入CustomPropertyEditorRegistrar 实例：

<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，说句题外话，对于那些使用[Spring’s MVC web framework](https://docs.spring.io/spring/docs/5.0.15.RELEASE/spring-framework-reference/web.html#mvc)的用户，使用属性编辑器注册器PropertyEditorRegistrars与数据绑定的Controllers（如SimpleFormController）配合将会十分方便。如下例，使用PropertyEditorRegistrar实现的initBinder(..)方法：

**public** **final** **class** **RegisterUserController** **extends** SimpleFormController {

**private** **final** PropertyEditorRegistrar customPropertyEditorRegistrar;

**public** RegisterUserController(PropertyEditorRegistrar propertyEditorRegistrar) {

this.customPropertyEditorRegistrar = propertyEditorRegistrar;

}

**protected** **void** initBinder(HttpServletRequest request,

ServletRequestDataBinder binder) **throws** Exception {

**this.customPropertyEditorRegistrar.registerCustomEditors(binder);**

}

*// other methods to do with registering a User*

}

PropertyEditor 由具体的代码注册（仅一行的initBinder(..)实现），且允许普通的PropertyEditor 注册代码封装在一个类中与其他Controllers 共享。

* + 1. Spring类型转换

Spring 3开始便有core.convert包提供通用类型转换系统。改系统定义了SPI去实现类型转换逻辑，如API一样在运行时执行类型转换。在Spring容器内，该系统可用于替代属性编辑器PropertyEditors来转换外部bean字符串属性值为所需的属性类型。公用API可用于你的应用内任意需要类型转换的地方。

* + - 1. 转换器SPI

实现类型转换逻辑的SPI简单并且是强类型：

**package** org.springframework.core.convert.converter;

**public** **interface** **Converter**<S, T> {

T convert(S source);

}

实现上述接口即可创建自己的转换器。泛型S表示被转换的类型，T表示转换的结果类型。假如一个S类型的集合或数组需要转换为元素类型为T的集合或数组，这样的转换器就可以透明地应用，前提是已经注册了一个代理数组/集合转换器（DefaultConversionService会默认执行）。

每一次调用convert(S)，源参数保证不为null。你的转换器可以在转换失败时抛出任意未检查异常；尤其应抛出IllegalArgumentException来报告无效的源值。注意保证你的Converter实现是线程安全的。

一些转换器实现已经提供在core.convert.support包内。包含了从字符串转换为数字及其他常见类型的转换器。下例为典型的Converter 实现StringToInteger：

**package** org.springframework.core.convert.support;

**final** **class** **StringToInteger** **implements** Converter<String, Integer> {

**public** Integer convert(String source) {

**return** Integer.valueOf(source);

}

}

* + - 1. 转换器工厂ConverterFactory

当你需要将转换逻辑集中化到一个类结构内（如转换String 为Enum 对象），你可以实现如下所示ConverterFactory：

**package** org.springframework.core.convert.converter;

**public** **interface** **ConverterFactory**<S, R> {

<T **extends** R> Converter<S, T> getConverter(Class<T> targetType);

}

泛型S是需要转换的类型，R是转换的结果类型基类。实现getConverter(Class<T>)，此处T是R的子类。

如下例StringToEnumConverterFactory所示：

**package** org.springframework.core.convert.support;

**final** **class** **StringToEnumConverterFactory** **implements** ConverterFactory<String, Enum> {

**public** <T **extends** Enum> Converter<String, T> getConverter(Class<T> targetType) {

**return** **new** StringToEnumConverter(targetType);

}

**private** **final** **class** **StringToEnumConverter**<T **extends** Enum> **implements** Converter<String, T> {

**private** Class<T> enumType;

**public** StringToEnumConverter(Class<T> enumType) {

this.enumType = enumType;

}

**public** T convert(String source) {

**return** (T) Enum.valueOf(this.enumType, source.trim());

}

}

}

* + - 1. 通用转换器GenericConverter

当你需要一个功能丰富的Converter实现时，可以使用GenericConverter接口。更加灵活且比Converter偏弱类型，GenericConverter支持多源与目标类型之间的转换。另外，通用转换器GenericConverter 使得源及目标域上下文可用，你可以利用来实现自己的转换逻辑。该上下文允许类型转换由域注解或声明在域签名上的通用信息驱动。如下是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);

}

实现GenericConverter，需要getConvertibleTypes()返回支持的源-目标类型对。然后实现convert(Object, TypeDescriptor, TypeDescriptor)来容纳你的转换逻辑。源TypeDescriptor 可以访问持有需转换值的源域。目标TypeDescriptor可以访问需设置转换后值的目标域。

GenericConverter的一个良好实践是能互相转换Java数组和集合的转换器。例如一个ArrayToCollectionConverter 会查看声明目标集合类型的域，以解析集合内的元素类型。这让源数组内的每一个元素在集合被设置到目标域前，转化为集合元素类型。

因为GenericConverter 是一个较复杂的SPI接口，注意据需使用。对于基础类型转换的需求，建议使用Converter 或ConverterFactory 。

条件型通用转换器

有时你只需要在某种情况下才执行Converter。例如，你只想在某个注解存在于目标域上时才执行Converter。或者你可能想在某个方法，如static valueOf方法定义在目标类内时才执行Converter。ConditionalGenericConverter 是通用转化器GenericConverter 和条件转换器ConditionalConverter接口的结合，允许你自定义匹配规则：

**public** **interface** **ConditionalConverter** {

**boolean** matches(TypeDescriptor sourceType, TypeDescriptor targetType);

}

**public** **interface** **ConditionalGenericConverter** **extends** GenericConverter, ConditionalConverter {

}

一个ConditionalGenericConverter的良好实现是实体转换器EntityConverter，其使得实体在持久实体标识符和实体引用之间转换。这样的一个实体转换器仅在目标实体类型声明了一个静态的查找方法如findAccount(Long)时匹配。在matches(TypeDescriptor, TypeDescriptor)的实现内可执行这样的查找方法检查。

* + - 1. 转换服务ConversionService API

转换服务ConversionService定义了统一的API用于运行时执行类型转换。转换器通常在此接口后执行：

**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);

}

大部分的转换服务实现也实现了ConverterRegistry，其提供了注册转换器的SPI。框架内部由转换服务代理其已注册的转换器执行类型转换逻辑。

core.convert.support包提供了个健壮的转换服务实现。GenericConversionService是通用目的实现适用于大多数场景。ConversionServiceFactory则提供了创建通用转换服务配置的一个方便的工厂。

* + - 1. 配置转换服务

转换服务是个无状态的对象，设计于应用启动时实例化，并在多个线程间共享。在Spring应用内，一般每一个Spring容器（或应用上下文）配置一个转换服务实例。Spring将自动选择转换服务并在类型转换需要由框架执行时使用。你也可以将其注入任意bean以直接调用。

如果没有转换服务注册在Spring，则原始基于属性编辑的系统将被使用

添加如下带有id为conversionService的bean定义，以注册默认转换服务到Spring：

<bean id="conversionService"

class="org.springframework.context.support.ConversionServiceFactoryBean"/>

默认转换服务可以转换字符串、数字、枚举、集合、map和其他常见类型。若以自定义转换器来要增强或覆盖默认转换器，设置converters属性。属性值可以实现Converter、ConverterFactory或GenericConverter接口。

<bean id="conversionService"

class="org.springframework.context.support.ConversionServiceFactoryBean">

<property name="converters">

<set>

<bean class="example.MyCustomConverter"/>

</set>

</property>

</bean>

在Spring MVC应用内使用转换服务也很常见。详见Spring MVC章[转换和格式化](https://docs.spring.io/spring/docs/5.0.15.RELEASE/spring-framework-reference/web.html#mvc-config-conversion)。

在特定情况下你会期待在转换时实现格式化。详见[FormatterRegistry SPI](https://docs.spring.io/spring/docs/5.0.15.RELEASE/spring-framework-reference/core.html#format-FormatterRegistry-SPI)以使用格式转换服务FormattingConversionServiceFactoryBean。

* + - 1. 编程式使用转换服务

编程式使用转换服务实例，只需要简单在所需bean内注入其引用即可：

@Service

**public** **class** **MyService** {

@Autowired

**public** MyService(ConversionService conversionService) {

this.conversionService = conversionService;

}

**public** **void** doIt() {

this.conversionService.convert(...)

}

}

在大多数应用场景下，convert方法可指定复杂的目标类型但不会起作用，例如泛型元素的集合。若你想编程式转换一个Integer的List或String的List，你需要提供源类型及目标类型的正式定义。

幸运的是，TypeDescriptor提供了更直接的多种选项：

DefaultConversionService cs = **new** DefaultConversionService();

List<Integer> input = ....

cs.convert(input,

TypeDescriptor.forObject(input), *// List<Integer> 类型描述符* TypeDescriptor.collection(List.class, TypeDescriptor.valueOf(String.class)));

注意DefaultConversionService自动注册适用于大多数环境的转换器。这包括了集合转换器，标量转换器和基础Object到String转换器。同一转换器可通过任意ConverterRegistry使用DefaultConversionService类的静态方法addDefaultConverters来注册。

值类型的转换器将被数组和集合重用，因此不需要创建特定转换器用于转换S的Collection到T的Collection，假设标准集合处理是合适的。

* + 1. Spring成员变量格式化

如前一节所述，[core.convert](https://docs.spring.io/spring/docs/5.0.15.RELEASE/spring-framework-reference/core.html#core-convert)是通用类型转换系统。其提供了统一的类型转换API与强类型转换器SPI用于实现从一个类型到另一类型的转换逻辑。Spring容器使用该系统绑定bean属性值。另外Spring表达式语言（SpEL）和数值绑定器DataBinder使用该系统绑定成员变量值。例如，当SpEL需要强制转换Short为Long以完成expression.setValue(Object bean, Object value)，那么core.convert系统将执行该强制行为。

那么假设一个典型客户端环境如web或桌面应用有类型转换需求。在这样的环境下，一般从字符串转换以支持客户端网络提交处理，或转换为字符串以支持视图渲染。另外，你经常需要本地化字符串值。通用core.convert转换器SPI没有直接处理这种格式化需求。Spring 3提供了方便的格式化器SPI，相较属性编辑器简单且强大，用于客户端环境以直接处理格式化需求。

通常，你可以使用Converter SPI在需要实现常用的类转换逻辑时，比如转换java.util.Date与Long。你可以在客户端环境（如web应用）内需要解析并打印本地化成员变量值时使用Formatter SPI。ConversionService对这些SPI提供了统一类型转换API。

* + - 1. 格式化器SPI

格式化器SPI需要实现的成员变量格式化逻辑简单并强类型：

**package** org.springframework.format;

**public** **interface** **Formatter**<T> **extends** Printer<T>, Parser<T> {

}

格式化器所继承的Printer和Parser接口描述如下：

**public** **interface** **Printer**<T> {

String print(T fieldValue, Locale locale);

}

**import** java.text.ParseException;

**public** **interface** **Parser**<T> {

T parse(String clientValue, Locale locale) **throws** ParseException;

}

创建自定义的格式化器，只需实现前述的Formatter接口。参数化类型T应是你期望格式化的对象类型—例如java.util.Date。实现print()操作以打印T的实例用于客户端展示。实现parse()操作以从客户端返回的格式化的描述解析为T的实例。你的格式化器应在尝试解析失败时抛出ParseException或IllegalArgumentException。注意保证你的解析器是线程安全的。

format子包提供了一些Formatter实现以便使用。number包提供了多个使用了java.text.NumberFormat用于格式化Number 对象的格式化器：NumberStyleFormatter，CurrencyStyleFormatter，PercentStyleFormatter。datetime包提供DateFormatter用于以java.text.DateFormat格式来格式化java.util.Date对象。datetime.joda包则提供基于[Joda-Time library](http://joda-time.sourceforge.net/)更便于理解的日期时间格式支持。

如下DateFormatter 则是Formatter 实现示例：

**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;

}

}

Spring团队欢迎社区Formatter 贡献；见[jira.spring.io](https://jira.spring.io/browse/SPR)贡献代码。

* + - 1. 注解驱动格式化

如你所见，成员变量格式化可以由其类型或注解来配置。实现AnnotationFormatterFactory以绑定一个注解到格式化器上：

**package** org.springframework.format;

**public** **interface** **AnnotationFormatterFactory**<A **extends** Annotation> {

Set<Class<?>> getFieldTypes();

Printer<?> getPrinter(A annotation, Class<?> fieldType);

Parser<?> getParser(A annotation, Class<?> fieldType);

}

参数A是与格式化逻辑关联的成员变量注解类型，例如

org.springframework.format.annotation.DateTimeFormat。getFieldTypes()方法返回注解标注的成员变量类型。getPrinter()方法返回一个Printer用于打印被标注成员变量的值。getParser()方法返回Parser用于解析客户端值到被标注成员变量。

如下AnnotationFormatterFactory实现示例将@NumberFormat注解与格式化器绑定。此注解允许指定数字格式或范式：

**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<?> fieldType) {

**if** (!annotation.pattern().isEmpty()) {

**return** **new** NumberStyleFormatter(annotation.pattern());

} **else** {

Style style = annotation.style();

**if** (style == Style.PERCENT) {

**return** **new** PercentStyleFormatter();

} **else** **if** (style == Style.CURRENCY) {

**return** **new** CurrencyStyleFormatter();

} **else** {

**return** **new** NumberStyleFormatter();

}

}

}

}

触发格式化只需使用@NumberFormat标记成员变量即可：

**public** **class** **MyModel** {

@NumberFormat(style=Style.CURRENCY)

**private** BigDecimal decimal;

}

格式注解API

org.springframework.format.annotation包内存在格式化注解API。你可以使用@NumberFormat去格式化Number成员变量如Double和Long，@DateTimeFormat格式化java.util.Date，java.util.Calendar，Long （毫秒时间戳）以及JSR-310java.time和Joda-Time值类型。

如下示例使用@DateTimeFormat格式化java.util.Date为ISO日期（yyyy-MM-dd）：

**public** **class** **MyModel** {

@DateTimeFormat(iso=ISO.DATE)

**private** Date date;

}

* + - 1. 格式化器注册器SPI

FormatterRegistry是用于注册格式化器跟转换器的SPI。FormattingConversionService则是该接口适用于大多数情况下的实现。该实现可以编程式配置或者可以作为Spring bean的形式使用FormattingConversionServiceFactoryBean来声明式配置。由于该实现同样实现了ConversionService，所以它也可以直接配置用于Spring的DataBinder和Spring表达式语言（SpEL）。

FormatterRegistry接口如下：

**package** org.springframework.format;

**public** **interface** **FormatterRegistry** **extends** ConverterRegistry {

**void** addFormatterForFieldType(Class<?> fieldType, Printer<?> printer, Parser<?> parser);

**void** addFormatterForFieldType(Class<?> fieldType, Formatter<?> formatter);

**void** addFormatterForFieldType(Formatter<?> formatter);

**void** addFormatterForAnnotation(AnnotationFormatterFactory<?, ?> factory);

}

如上可见，格式化器可以通过成员变量类型或注解注册。

FormatterRegistry SPI允许你中心化设置格式化规则，而不是在你的控制器之间不断复制。例如你会需要对所有的Date类型成员变量都以一定格式格式化，或带有特定注解的成员变量格式化。使用共享的FormatterRegistry，你可以一次定义规则，就可多次在需格式化的地方应用。

* + - 1. FormatterRegistrar SPI

FormatterRegistrar则是通过FormatterRegistry注册格式化器和转换器的SPI：

**package** org.springframework.format;

**public** **interface** **FormatterRegistrar** {

**void** registerFormatters(FormatterRegistry registry);

}

FormatterRegistrar在对某个既定格式化组类注册多个相关的转换器和格式化器时很有用，比如Date类型格式化。同时在声明式注册不能满足时也十分有用。例如当格式化器在与其自身<T>类型不同的指定成员变量类型下编制索引或当注册一对Printer/Parser时。下一节将提供更多有关转换器和注册器的注册信息。

* + - 1. 在Spring MVC内配置格式化

详见Spring MVC章[Conversion and Formatting](https://docs.spring.io/spring/docs/5.0.15.RELEASE/spring-framework-reference/web.html#mvc-config-conversion)。

* + 1. 配置全局日期和时间格式

默认情况下，没有被@DateTimeFormat的标记日期和时间成员变量会以DateFormat.SHORT格式字符串表示转换为对象。若需要，你可以通过更改来定义自己的全局格式。

你需要保证Spring不会注册默认注册器，而是由你来手动注册所有格式化器。考虑使用Spring提供的org.springframework.format.datetime.joda.JodaTimeFormatterRegistrar或rg.springframework.format.datetime.DateFormatterRegistrar取决于你是否使用Joda-Time第三方库。

例如，如下Java设置将注册一个全局“yyyyMMdd”格式。此示例不依赖于Joda-Time库：

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** FormattingConversionService conversionService() {

*// 使用DefaultFormattingConversionService但不注册缺省值*

DefaultFormattingConversionService conversionService = **new** DefaultFormattingConversionService(false);

*// 保证@NumberFormat仍支持*

conversionService.addFormatterForFieldAnnotation(**new** NumberFormatAnnotationFormatterFactory());

*// 以一个指定的全局格式注册日期转换*

DateFormatterRegistrar registrar = **new** DateFormatterRegistrar();

registrar.setFormatter(**new** DateFormatter("yyyyMMdd"));

registrar.registerFormatters(conversionService);

**return** conversionService;

}

}

如下则是基于XML配置，使用FormattingConversionServiceFactoryBean，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

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

<bean id="conversionService" class="org.springframework.format.support.FormattingConversionServiceFactoryBean">

<property name="registerDefaultFormatters" value="false" />

<property name="formatters">

<set>

<bean class="org.springframework.format.number.NumberFormatAnnotationFormatterFactory" />

</set>

</property>

<property name="formatterRegistrars">

<set>

<bean class="org.springframework.format.datetime.joda.JodaTimeFormatterRegistrar">

<property name="dateFormatter">

<bean class="org.springframework.format.datetime.joda.DateTimeFormatterFactoryBean">

<property name="pattern" value="yyyyMMdd"/>

</bean>

</property>

</bean>

</set>

</property>

</bean>

</beans>

Joda-Time提供了不同的类型去表示date，time和date-time值。dateFormatter，timeFormatter和dateTimeFormatter是JodaTimeFormatterRegistrar的内部属性，用于为每一个类型配置不同格式。DateTimeFormatterFactoryBean则提供用于方便创建格式化器的方式。

如果你使用Spring MVC记得显式配置使用的转换服务。对于基于Java的@Configuration意味着需要继承WebMvcConfigurationSupport类并覆写mvcConversionService()方法。对于XML，则应使用mvc:annotation-driven元素的'conversion-service'属性。详见[Conversion and Formatting](https://docs.spring.io/spring/docs/5.0.15.RELEASE/spring-framework-reference/web.html#mvc-config-conversion)。

* + 1. Spring校验

Spring 3加强了对校验的支持。首先，完全支持JSR-303 Bean校验API；第二，在编程式使用时，Spring的DataBinder可以在绑定时校验对象；第三，Spring MVC支持声明式校验标记为的@Controller输入。

* + - 1. JSR-303 Bean校验API简介

JSR-303标准化Java平台的校验约束声明和元数据。通过该API，你可以使用声明式校验约束标记域模型属性，以此在运行时执行该约束。如下有部分内建约束可供使用。你同样可以自定义自己的约束。

为了方便说明，假设有一个简单的PersonForm模型，带有两个属性：

**public** **class** **PersonForm** {

**private** String name;

**private** **int** age;

}

JSR-303允许你对属性定义声明式校验约束：

**public** **class** **PersonForm** {

@NotNull

@Size(max=64)

**private** String name;

@Min(0)

**private** **int** age;

}

当一个该类的实例受到JSR-303校验器的校验时，该约束将被执行。

有关JSR-303/JSR-349的详细信息，见[Bean Validation website](https://beanvalidation.org/)。关于默认参考实现的特定能力，见[Hibernate Validator](https://www.hibernate.org/412.html)文档。学习如何设置Bean校验提供器为Spring bean，如下。

* + - 1. 配置Bean校验提供器

Spring全面支持Bean校验API。即包括了支持启动JSR-303/JSR-349 Bean校验提供器为Spring bean。这就允许无论何时你的应用需要校验时javax.validation.ValidatorFactory或javax.validation.Validator被注入。

使用LocalValidatorFactoryBean来设置默认校验器为Spring bean：

<bean id="validator" class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"/>

上述基础配置将触发bean校验使用其默认启动机制来初始化。一个JSR-303/JSR-349提供器，例如Hibernate校验器，放置在类路径内才会被自动检测到。

注入一个校验器

LocalValidatorFactoryBean实现了接口javax.validation.ValidatorFactory和接口javax.validation.Validator，和org.springframework.validation.Validator。你会需要将这些接口的实现引用注入到bean内以实现校验逻辑。

注入javax.validation.Validator的引用，在需要直接使用Bean校验API时：

**import** javax.validation.Validator;

@Service

**public** **class** **MyService** {

@Autowired

**private** Validator validator;

注入org.springframework.validation.Validator的引用，如果需要Spring校验API：

**import** org.springframework.validation.Validator;

@Service

**public** **class** **MyService** {

@Autowired

**private** Validator validator;

}

配置自定义约束

每一个bean校验约束包含两个部分。首先，@Constraint声明约束及其配置属性。第二，实现javax.validation.ConstraintValidator接口的约束行为。关联一个声明与一个实现，则每一个@Constraint注解需引用相关的ConstraintValidator 实现类。在运行时，约束校验器工厂ConstraintValidatorFactory 则会实例化引用的实现，当在你的域模型中遇到约束注解时。

默认时，LocalValidatorFactoryBean设置SpringConstraintValidatorFactory 来使用Spring去创建约束校验器实例。此则允许你的自定义约束校验器像其他Spring bean一样可用于依赖注入。

如下是自定义@Constraint声明，带有对应的ConstraintValidator实现，使用Spring以方便依赖注入：

@Target({ElementType.METHOD, ElementType.FIELD})

@Retention(RetentionPolicy.RUNTIME)

@Constraint(validatedBy=MyConstraintValidator.class)

**public** @interface MyConstraint {

}

**import** javax.validation.ConstraintValidator;

**public** **class** **MyConstraintValidator** **implements** ConstraintValidator {

@Autowired;

**private** Foo aDependency;

...

}

如你所见，一个约束校验器ConstraintValidator实现可以像其他Spring bean一样使用依赖注入。

Spring驱动方法校验

Bean Validation 1.1支持方法校验特性，并在Hibernate Validator 4.3内作为自定义插件，可以通过MethodValidationPostProcessorbean定义整合入Spring上下文内：

<bean class="org.springframework.validation.beanvalidation.MethodValidationPostProcessor"/>

为了适配Spring驱动方法校验，所有目标类都需要标记以Spring的@Validated注解，可以声明使用的校验组。查看MethodValidationPostProcessor javadocs，以了解Hibernate Validator和Bean Validation 1.1 providers的配置细节。

额外配置选项

默认LocalValidatorFactoryBean配置满足大多数场景的需求。有一系列用于Bean Validation构造器的配置选项，从消息插值到遍历解析。详见LocalValidatorFactoryBean javadocs。

* + - 1. 配置数值绑定器DataBinder

自Spring 3起，DataBinder实例可以设置一个校验器Validator。一旦设置完成，Validator则会通过调用binder.validate()唤起。任意校验错误都会自动添加到binder的绑定结果BindingResult中。

编程式使用DataBinder时，如下可以在绑定目标对象后唤起校验逻辑：

Foo target = **new** Foo();

DataBinder binder = **new** DataBinder(target);

binder.setValidator(**new** FooValidator());

*// 绑定目标对象*

binder.bind(propertyValues);

*// 校验目标对象*

binder.validate();

*// 获取包含所有绑定错误的绑定结果*

BindingResult results = binder.getBindingResult();

数值绑定器可以以多个Validator实例，通过数值绑定器dataBinder.addValidators和dataBinder.replaceValidators方法来配置。这通常在组合全局配置的bean校验和通过数值绑定器DataBinder局部配置的Spring Validator 。见[[validation-mvc-configuring]](https://docs.spring.io/spring/docs/5.0.15.RELEASE/spring-framework-reference/core.html#validation-mvc-configuring)。

* + - 1. Spring MVC 3校验

见Spring MVC章[Validation](https://docs.spring.io/spring/docs/5.0.15.RELEASE/spring-framework-reference/web.html#mvc-config-validation)。

## Spring表达式语言（SpEL）

* + 1. 简介

Spring表达式语言（缩写SpEL）是非常丰富的表达式语言，支持运行时查询和改变对象的内容。该语言的语法与通用的表达式语言相似，但提供了更多的特性，特别是方法调用和基础字符串模板功能。

当前已存在一些其他的Java表达式语言——OGNL，MVEL，和JBoss EL等等。Spring表达式语言的创建是为了给Spring社区提供统一支持的表达式语言，并通用于在Spring全家桶，包括基于Eclipse的Spring Tool Suite。也就是说，SpEL基于与技术无关的API，允许在需要时集成其他表达式语言实现。

然而SpEL在Spring全家桶内是作为一个基础模块存在的，不直接与Spring绑定，可独立使用。为了显示其独立性，本章的许多例子都将其作为独立的表达式语言来使用。这就需要创建一些引导结构类如解析器。大多数的Spring用户都不需要处理这些基础结构，仅需要编写表达式字符串即可。一个经典例子就是整合SpEL进XML或注解bean定义，如小节[Expression support for defining bean definitions](https://docs.spring.io/spring/docs/5.0.15.RELEASE/spring-framework-reference/core.html#expressions-beandef)所示。

本章涵盖了表达式语言的特性、API及其语法语义。在某些地方，Inventor和Inventor的Society类被作为表达式的目标对象使用。这些类的声明及其数据之间的转移在本章末列出。

表达式语言支持如下功能：

* 字面表达式
* 布尔和相关操作符
* 正则表达式
* 类表达式
* 访问属性、数组、列表、map
* 方法调用
* 关系操作符
* 分配
* 构造器调用
* Bean引用
* 数组构造
* 内联列表
* 内联map
* 三元运算符
* 标量
* 用户定义函数
* 集合映射
* 集合选择
* 模板表达式
  + 1. 预测

本节介绍了SpEL接口及其表达式语言的简单使用。完整的语言参考可以在小节[Language Reference](https://docs.spring.io/spring/docs/5.0.15.RELEASE/spring-framework-reference/core.html#expressions-language-ref)中找到。

如下代码表示以SpEL API来解析字符串表达式“Hello World”。

ExpressionParser parser = **new** SpelExpressionParser();

Expression exp = parser.parseExpression("**'Hello World'**");

String message = (String) exp.getValue();

该条消息变量的值就是简单的“Hello World”。

你可能使用的SpEL类和接口位于包org.springframework.expression极其子包spel.support内。

接口ExpressionParser 负责匹配表达式字符串。在此例中表达式字符串是由单引号包括的简单字面意义字符串。接口Expression负责解析前面定义的表达式字符串。此时会有两种异常抛出，分别在调用parser.parseExpression和exp.getValue时抛出ParseException 和EvaluationException 。

SpEL支持广泛的特性，如方法调用，属性访问，构造器调用。

如下方法调用的示例，我们调用String的concat 方法。

ExpressionParser parser = **new** SpelExpressionParser();

Expression exp = parser.parseExpression("**'Hello World'.concat('!')**");

String message = (String) exp.getValue();

此时这条消息的值为‘Hello World!’。

调用JavaBean属性的示例，如下String属性的Bytes 被获取。

ExpressionParser parser = **new** SpelExpressionParser();

*// 调用 'getBytes()'*

Expression exp = parser.parseExpression("**'Hello World'.bytes**");

**byte[]** bytes = (**byte[]**) exp.getValue();

SpEL同样支持通过点表达式获取内嵌属性（如prop1.prop2.prop3），也可以设置相关的属性值。Public成员变量也可以被访问。

如下示例演示了点表达式获取一段文字的长度：

ExpressionParser parser = **new** SpelExpressionParser();

*// 调用 'getBytes().length'*

Expression exp = parser.parseExpression("**'Hello World'.bytes.length**");

**int** length = (Integer) exp.getValue();

字符串构造器可以替换直接字面语言。

ExpressionParser parser = **new** SpelExpressionParser();

Expression exp = parser.parseExpression("**new String('hello world').toUpperCase()**");

String message = exp.getValue(String.class);

注意泛型方法public <T> T getValue(Class<T> desiredResultType)的使用。使用该方法可以不需要将表达式的值转换为所需类型。EvaluationException 在值不能转换为类型T或通过注册的类型转换器转换时抛出。

SpEL最常用的用法是提供一个表达式针对一个特定的对象实例（称为根对象）来解析。如下演示了如何通过Inventor 类的实例来获取其中的name 属性或创建一个布尔值：

*// 创建并设置一个calendar*

GregorianCalendar c = **new** GregorianCalendar();

c.set(1856, 7, 9);

*// 构造器入参分别为name, birthday, and nationality.*

Inventor tesla = **new** Inventor("Nikola Tesla", c.getTime(), "Serbian");

ExpressionParser parser = **new** SpelExpressionParser();

Expression exp = parser.parseExpression("**name**");

String name = (String) exp.getValue(tesla);

*// name == "Nikola Tesla"*

exp = parser.parseExpression("name == 'Nikola Tesla'");

**boolean** result = exp.getValue(tesla, Boolean.class);

*// result == true*

* + - 1. **EvaluationContext**

接口EvaluationContext 用于在解析表达式时提取属性、方法或者成员变量并帮助执行类型转换。这里有两个开箱即用的实现。

* SimpleEvaluationContext ——暴露了SpEL语言的子集，包含必要特性跟配置选项，对于那些不需要SpEL语言语义完全扩展内容的表达式应有一定的限制。示例包括但不限于数值绑定表达式，基于属性的过滤器等等。
* StandardEvaluationContext ——暴露了SpEL语言的所有特性和配置选项。你可通过其指定一个默认根对象并配置每一个可用的解析相关策略。

SimpleEvaluationContext 设计用于仅支持SpEL语言的部分语义。其剔除了Java类型引用，构造函数和bean引用。它同样需要在表达式中显示指定受支持的属性和方法的等级。默认情况下，create()静态工厂方法允许对属性的读访问。你可以通过一个builder来配置受支持的具体等级，指定如下一个或多个组合：

* 仅使用自定义的PropertyAccessor（无反射）
* 只读访问的数值绑定属性
* 可读写的数值绑定属性

类型转换

Spring core服务（org.springframework.core.convert.ConversionService），默认情况下会被SpEL使用其转换服务。该转换服务带有许多内建转换器用于通用的转换，但其完全支持扩展，因此自定义的类型转换也可以被添加。另外它有个重点功能是泛型可知，即当在表达式中使用的泛型时，SpEL将会尝试在转换时对遇到的对象保持类型正确。

在实践中怎么操作？假设在分配时，使用setValue()来设置一个List 属性。但属性的类型实际上是List<Boolean>。SpEL将在元素放入列表内前识别并将其转换为Boolean 。

**class** **Simple** {

**public** List<Boolean> booleanList = **new** ArrayList<Boolean>();

}

Simple simple = **new** Simple();

simple.booleanList.add(true);

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();

*// "false" 在此处以字符串传入. SpEL及转换服务将识别并将其转换为Boolean*

parser.parseExpression("booleanList[0]").setValue(context, simple, "false");

*// b 将是false*

Boolean b = simple.booleanList.get(0);

* + - 1. 解析器配置

可以使用org.springframework.expression.spel.SpelParserConfiguration解析器配置对象来配置SpEL表达式解析器。配置对象控制着一些表达式组件的行为。例如，如果根据下标查找一个数组或集合元素内容为null 时，可以自动创建该元素。这在使用表达式构成一个属性引用链时很有用。如果根据下标查找数组或列表，在下标越界时可以自动扩容以容纳该下标。

**class** **Demo** {

**public** List<String> list;

}

*// 打开:*

*// - 自动初始化空引用*

*// - 集合自动扩容*

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 此时是一个拥有四个元素的真实集合*

*// 每个元素都是一个空字符串*

也可以设置SpEL表达式编译器的行为。

* + - 1. SpEL编译

Spring框架4.1包含了基础的表达式编译器。表达式通常由解释执行，可以在运行时提供高的灵活性但效率并不高。对于临时的表达式使用，这没有关系，但是若是由其他组件如Spring整合项目使用，那么性能就显得尤为重要，且动态就不是必需。

SpEL编译器用于响应此需求。在解析运行时，编译器将生成一个Java类，在运行时封装表达式行为，并使用该类来达到更快的表达式运行。由于缺少表达式的上下文环境，编译器使用该表达式在解释执行时收集到的相关信息来执行编译。例如，它并不能单纯从表达式中就知道属性引用的类型，但通过第一次的解释运行就可以获取这些信息。当然，基于这些信息的编译会在表达式经常更改时出现问题。因此，编译行为最适合在类型信息不常变化的表达式内进行。

普通表达式如下：

someArray[0].someProperty.someOtherProperty < 0.1

该表达式涉及数组访问、属性引用和数值计算，性能差距将十分明显。此示例经历过50000次迭代运行后，使用解释执行的方式需要75ms而使用表达式的编译执行模式只需要3ms。

编译器设置

编译器默认情况下不会打开，有两种方式可以打开。可以通过前述的解析器配置来打开或者在SpEL内嵌在某些组件内时通过系统属性来打开。本节讨论这两种方式。

编译器可在多种模式（org.springframework.expression.spel.SpelCompilerMode）下运行，所有模式均包含下述枚举中：

* OFF ——编译器关闭，默认值
* IMMEDIATE ——直接模式下表达式直接被编译。这一般在发生在第一次解释执行后。如果表达式编译失败（一般由于类型变化，如上所述）那么调用者将收到异常。
* MIXED ——在混合模式下表达式会在解释执行和编译执行之间切换。在经过一定次数的解释执行后切换至编译模式且在编译模式出错时（出现类型变化，如上所述）则表达式则自动切换回解释执行模式。一段时间后其又将切换至编译模式。基本上用户在IMMEDIATE 模式获取到的异常都已在内部处理。

IMMEDIATE 模式存在是因为MIXED 模式会因产生具有副作用的表达式。若一个已编译的表达式在部分成功后崩溃，那么其已经影响了系统的状态。假如这已经发生那么调用者则并不希望其净末地重新执行，因为表达式的一部分将执行两次。

选择模式后，使用SpelParserConfiguration 设置解析器：

SpelParserConfiguration config = **new** SpelParserConfiguration(SpelCompilerMode.IMMEDIATE,

this.getClass().getClassLoader());

SpelExpressionParser parser = **new** SpelExpressionParser(config);

Expression expr = parser.parseExpression("payload");

MyMessage message = **new** MyMessage();

Object payload = expr.getValue(message);

在指定编译器模式时，你还可以指定类加载器（允许null值）。已编译表达式会由入参类加载器创建的子类加载器来定义。需要确认，在类加载器指定时，其对表达式运行进程内的所有类型可见。若不指定类加载器，将使用默认类加载器（一般来说使用当前表达式运行的线程上下文类加载器）。

第二种设置方式则用于，在SpEL内嵌于其他模块且不能通过配置对象来设置时，可以使用系统属性值。可以设置spring.expression.compiler.mode属性值为SpelCompilerMode 枚举内的任意一个值（off，immediate，mixed）。

编译器限制

自Spring框架4.1始，内置了基础编译框架。然而框架不能支持所有类型表达式。其专注点在性能敏感类型的上下文内使用的通用表达式。当前仍不能编译的表达式：

* 涉及赋值的表达式
* 依赖转换服务的表达式
* 使用自定义解析器或访问器的表达式
* 使用选择器或反射的表达式

未来将有越来越多的表达式可编译。

* + 1. Bean定义内的表达式

SpEL表达式可用于XML或基于注解配置元数据的bean定义BeanDefinition。在这两种情况下都可以使用表达式，格式为#{ <expression string> }。

* + - 1. XML 配置

属性或构造器参数值可以用表达式作如下定义。

<bean id="numberGuess" class="org.spring.samples.NumberGuess">

<property name="randomNumber" value="#{ T(java.lang.Math).random() \* 100.0 }"/>

*<!-- 其他属性 -->*

</bean>

变量systemProperties 已经预定义，因此你可以在你的表达式内使用。注意在该上下文内，你不需要以#作为预定义变量的前缀。

<bean id="taxCalculator" class="org.spring.samples.TaxCalculator">

<property name="defaultLocale" value="#{ systemProperties['user.region'] }"/>

*<!-- 其他属性 -->*

</bean>

也可通过bean名引用bean属性，如

<bean id="numberGuess" class="org.spring.samples.NumberGuess">

<property name="randomNumber" value="#{ T(java.lang.Math).random() \* 100.0 }"/>

*<!-- 其他属性 -->*

</bean>

<bean id="shapeGuess" class="org.spring.samples.ShapeGuess">

<property name="initialShapeSeed" value="#{ numberGuess.randomNumber }"/>

*<!-- 其他属性 -->*

</bean>

* + - 1. 注解配置

@Value注解可添加在成员变量、方法和方法/构造方法参数上以指定默认值。

如下示例是给成员变量设置默认值。

**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;

}

}

与上述等价，放置在属性setter方法上

**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;

}

}

在autowired方法和构造函数上使用@Value注解

**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;

}

*// ...*

}

* + 1. 语言参考书
       1. 普通表达式

普通表达式支持字符串、数值（整型，真实值，十六进制）、布尔值和null。字符串则由单引号标记。若需以单引号为字符串，使用两个单引号。

如下列表展示了普通表达式的简单使用。一般情况下它们不会像这样被单独使用，而是作为更复杂表达式内的一部分，如在逻辑比较运算符的一边使用简单表达式。

ExpressionParser parser = **new** SpelExpressionParser();

*// 等价于 "Hello World"*

String helloWorld = (String) parser.parseExpression("'Hello World'").getValue();

**double** avogadrosNumber = (Double) parser.parseExpression("6.0221415E+23").getValue();

*// 等价于 2147483647*

**int** maxValue = (Integer) parser.parseExpression("0x7FFFFFFF").getValue();

**boolean** trueValue = (Boolean) parser.parseExpression("true").getValue();

Object nullValue = parser.parseExpression("null").getValue();

数字支持负数、指数和小数点。默认情况下真实数值使用Double.parseDouble()解析。

* + - 1. 属性，数组，列表，Map，索引

使用属性引用索引数据很简单：只需要使用句号来表示内嵌属性。Inventor 类的实例pupin和tesla，由小节[Classes used in the examples](https://docs.spring.io/spring/docs/5.0.15.RELEASE/spring-framework-reference/core.html#expressions-example-classes)内所列数据填充。若需向下索引获取tesla的生日年份和pupin的出生城市只需如下：

*// 等于 1856*

**int** year = (Integer) parser.parseExpression("Birthdate.Year + 1900").getValue(context);

String city = (String) parser.parseExpression("placeOfBirth.City").getValue(context);

属性名的首字母大小写均可。数组和列表内容获取通过方括号表示。

ExpressionParser parser = **new** SpelExpressionParser();

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();

*// Inventions 数组*

*// 解析为 "Induction motor"*

String invention = parser.parseExpression("inventions[3]").getValue(

context, tesla, String.class);

*// 成员列表*

*// 解析为 "Nikola Tesla"*

String name = parser.parseExpression("Members[0].Name").getValue(

context, ieee, String.class);

*// 数组和列表导航解析为 "Wireless communication"*

String invention = parser.parseExpression("Members[0].Inventions[6]").getValue(

context, ieee, String.class);

map内的内容可以通过指定在方括号内的key值来获取。在这里，由于Officers的key是String值，所以要用String表示法。

*// Officer's 字典*

Inventor pupin = parser.parseExpression("Officers['president']").getValue(

societyContext, Inventor.class);

*// 解析为 "Idvor"*

String city = parser.parseExpression("Officers['president'].PlaceOfBirth.City").getValue(

societyContext, String.class);

*// 设置值*

parser.parseExpression("Officers['advisors'][0].PlaceOfBirth.Country").setValue(

societyContext, "Croatia");

* + - 1. 内联列表

列表可以直接通过{}来表示。

*// 解析为包含四个数字的Java list*

List numbers = (List) parser.parseExpression("{1,2,3,4}").getValue(context);

List listOfLists = (List) parser.parseExpression("{{'a','b'},{'x','y'}}").getValue(context);

{}自身表示空列表。处于性能考虑，如果该列表本身是由固定文字组成，那么将会创建一个常量列表，而不是在每一次的解析时都创建一个新的列表。

* + - 1. 内敛map

Map也可以通过{key:value}直接表示。

*// 解析为一个具有两个键值对的Java map*

Map inventorInfo = (Map) parser.parseExpression("{name:'Nikola',dob:'10-July-1856'}").getValue(context);

Map mapOfMaps = (Map) parser.parseExpression("{name:{first:'Nikola',last:'Tesla'},dob:{day:10,month:'July',year:1856}}").getValue(context);

{:}本身表示一个空map。处于性能考虑，如果map由固定的文字或其他内嵌固定的结构（列表或map），那么该表达式将指向一个常量map，而不是在每一次的解析时都创建出新的。Map的key单引号是可选的，上述例子没有使用单引key。

* + - 1. 数组构建

数组可以使用Java一样的语法来构建，可以提供初始值来在构建时填充初始值。

**int[]** numbers1 = (**int[]**) parser.parseExpression("new int[4]").getValue(context);

*// 数组带有初始值*

**int[]** numbers2 = (**int[]**) parser.parseExpression("new int[]{1,2,3}").getValue(context);

*// 多维数组*

**int[][]** numbers3 = (**int[][]**) parser.parseExpression("new int[4][5]").getValue(context);

当前不支持构建多维数组时提供初始值。

* + - 1. 方法

方法调用使用传统Java编程式语法。可以直接调用字符串方法。支持多参数。

*// 字符串文字, 解析为 "bc"*

String bc = parser.parseExpression("'abc'.substring(1, 3)").getValue(String.class);

*// 解析为 true*

**boolean** isMember = parser.parseExpression("isMember('Mihajlo Pupin')").getValue(

societyContext, Boolean.class);

* + - 1. 操作符

关系操作符

关系操作符包括：等、不等、小于、小等于、大于、大等于，使用标准操作符表示。

*// 解析为 true*

**boolean** trueValue = parser.parseExpression("2 == 2").getValue(Boolean.class);

*// 解析为 false*

**boolean** falseValue = parser.parseExpression("2 < -5.0").getValue(Boolean.class);

*// 解析为 true*

**boolean** trueValue = parser.parseExpression("'black' < 'block'").getValue(Boolean.class);

与null 的大小比较遵循以下简单规则：此处null视为无（并非0）.因此，任意值均大于null （X > null恒为true），没有任何值小于无（X < null恒为false）。

若你仅希望数值比较，请注意避免出现null 在期望比较0时（如X > 0或X < 0）

另外除去标准关系操作符SpEL还支持instanceof 和正则表达式matches 操作符。

*// 解析为 false*

**boolean** falseValue = parser.parseExpression( "'xyz' instanceof T(Integer)").getValue(Boolean.class);

*// 解析为 to true*

**boolean** trueValue = parser.parseExpression("'5.00' matches '^-?\\d+(\\.\\d{2})?$'").getValue(Boolean.class);

*//解析为 to false*

**boolean** falseValue = parser.parseExpression("'5.0067' matches '^-?\\d+(\\.\\d{2})?$'").getValue(Boolean.class);

注意基础类型因为它们直接自动装箱为其包裹类型，即1 instanceof T(int)解析为false 而1 instanceof T(Integer)解析为true。

每一个符号操作符都可以由纯字母表达替代。这避免了这些符号使用在某些内嵌表达式文件类型中具有其他特殊含义（如XML文件）。字符表示如下：lt (<)、gt (>)、 le (<=)、 ge (>=)、 eq (==)、ne (!=)、div (/)、mod (%)、not (!)。所有字符表达式大小写敏感。

逻辑操作符

逻辑操作符支持与、或、非。使用示例如下。

*// -- 与 --*

*// 解析为 false*

**boolean** falseValue = parser.parseExpression("true and false").getValue(Boolean.class);

*// 解析为 true*

String expression = "isMember('Nikola Tesla') and isMember('Mihajlo Pupin')";

**boolean** trueValue = parser.parseExpression(expression).getValue(societyContext, Boolean.class);

*// -- 或 --*

*// 解析为 true*

**boolean** trueValue = parser.parseExpression("true or false").getValue(Boolean.class);

*// 解析为 true*

String expression = "isMember('Nikola Tesla') or isMember('Albert Einstein')";

**boolean** trueValue = parser.parseExpression(expression).getValue(societyContext, Boolean.class);

*// -- 非 --*

*// 解析为 false*

**boolean** falseValue = parser.parseExpression("!true").getValue(Boolean.class);

*// -- 与和非 --*

String expression = "isMember('Nikola Tesla') and !isMember('Mihajlo Pupin')";

**boolean** falseValue = parser.parseExpression(expression).getValue(societyContext, Boolean.class);

数值操作符

加符号可用于数字与字符串。减、乘和除仅可用于数字。其他数值运算符是模（%）和指数（^）。运算符优先级与标准一致。运算符用法如下。

*// 加*

**int** two = parser.parseExpression("1 + 1").getValue(Integer.class); *// 2*

String testString = parser.parseExpression("'test' + ' ' + 'string'").getValue(String.class); *// 'test string'*

*// 减*

**int** four = parser.parseExpression("1 - -3").getValue(Integer.class); *// 4*

**double** d = parser.parseExpression("1000.00 - 1e4").getValue(Double.class); *// -9000*

*// 乘*

**int** six = parser.parseExpression("-2 \* -3").getValue(Integer.class); *// 6*

**double** twentyFour = parser.parseExpression("2.0 \* 3e0 \* 4").getValue(Double.class); *// 24.0*

*// 除*

**int** minusTwo = parser.parseExpression("6 / -3").getValue(Integer.class); *// -2*

**double** one = parser.parseExpression("8.0 / 4e0 / 2").getValue(Double.class); *// 1.0*

*// 模*

**int** three = parser.parseExpression("7 % 4").getValue(Integer.class); *// 3*

**int** one = parser.parseExpression("8 / 5 % 2").getValue(Integer.class); *// 1*

*// 运算优先级*

**int** minusTwentyOne = parser.parseExpression("1+2-3\*8").getValue(Integer.class); *// -21*

* + - 1. 赋值

通过赋值设置属性。一般在调用setValue 时完成，也可以在调用getValue时执行。

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);

* + - 1. 类型

特殊运算符T用于指定java.lang.Class（类型）的实例。其也可以用于调用静态方法。StandardEvaluationContext 使用TypeLocator 来查找类型且StandardTypeLocator （可替换）创建于可访问java.lang.package。这意味着T()引用java.lang内的类型不需要全限定名，但其他类型引用需要。

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);

* + - 1. 构造函数

构造函数可以通过new运算符调用。除去初始类型和String(int, float等等)，其他都应使用类全限定名。

Inventor einstein = p.parseExpression("new org.spring.samples.spel.inventor.Inventor('Albert Einstein', 'German')").getValue(Inventor.class);

*//在List的add方法内创建inventor对象*

p.parseExpression("Members.add(new org.spring.samples.spel.inventor.Inventor('Albert Einstein', 'German'))").getValue(societyContext);

* + - 1. 变量

可使用语法#variableName在表达式内引用变量。变量由EvaluationContext 实现内的方法setVariable 设置。

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

变量#this 总是预定义并指向当前解析对象（对比非法的引用）。变量#root 则指向根上下文对象。即使#this会随着表达式解析对象的更改而改变，#root总是指向根对象。

*// 创建整型数组*

List<Integer> primes = **new** ArrayList<Integer>();

primes.addAll(Arrays.asList(2,3,5,7,11,13,17));

*// 创造解析器并设置变量'primes'为该整型数组*

ExpressionParser parser = **new** SpelExpressionParser();

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataAccess();

context.setVariable("primes", primes);

*// 所有列表内 > 10 的质数 (使用selection ?{...})*

*// 得出[11, 13, 17]*

List<Integer> primesGreaterThanTen = (List<Integer>) parser.parseExpression("#primes.?[#this>10]").getValue(context);

* + - 1. 函数

你可以扩展SpEL通过注册可在表达式内调用的自定义函数。函数通过EvaluationContext注册。

Method method = ...;

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();

context.setVariable("myFunction", method);

例如，既定的反转字符串工具方法如下：

**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();

}

}

上述方法可通过如下方式注册并使用：

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);

* + - 1. Bean引用

如果解析上下文配置有bean解析器，那么就可以通过在表达式内使用@符号来查找bean。

ExpressionParser parser = **new** SpelExpressionParser();

StandardEvaluationContext context = **new** StandardEvaluationContext();

context.setBeanResolver(**new** MyBeanResolver());

*// 解析时最终会在MyBeanResolver调用resolve(context,"foo")*

Object bean = parser.parseExpression("@foo").getValue(context);

若需访问工厂bean，那么bean名就需要以&作为前缀。

ExpressionParser parser = **new** SpelExpressionParser();

StandardEvaluationContext context = **new** StandardEvaluationContext();

context.setBeanResolver(**new** MyBeanResolver());

*// 解析时最终会在MyBeanResolver调用resolve(context,"&foo")*

Object bean = parser.parseExpression("&foo").getValue(context);

* + - 1. 三目运算符（if-then-else）

你可以在表达式内使用三目运算符实现if-then-else条件逻辑判断。简单示例如下：

String falseString = parser.parseExpression("false ? 'trueExp' : 'falseExp'").getValue(String.class);

在这种情况下，false布尔值会返回字符值'falseExp'。更实际的例子如下：

parser.parseExpression("Name").setValue(societyContext, "IEEE");

societyContext.setVariable("queryName", "Nikola Tesla");

expression = "isMember(#queryName)? #queryName + ' is a member of the ' + Name + ' Society' : #queryName + ' is not a member of the ' + Name + ' Society'";

String queryResultString = parser.parseExpression(expression).getValue(societyContext, String.class);

*// queryResultString = "Nikola Tesla is a member of the IEEE Society"*

* + - 1. Elvis运算符

Elvis运算符是三目运算符的简化版并用于[Groovy](http://www.groovy-lang.org/operators.html#_elvis_operator)语言。在三目运算符内，一般需要对一个变量写两遍，如下所示：

String name = "Elvis Presley";

String displayName = (name != null ? name : "Unknown");

然而，使用Elvis运算符（命名来源于猫王的发型）。如下示例为Elvis运算符用法：

ExpressionParser parser = **new** SpelExpressionParser();

String name = parser.parseExpression("name?:'Unknown'").getValue(String.class);

System.out.println(name); *// 'Unknown'*

如下列出了更复杂的用法：

ExpressionParser parser = **new** SpelExpressionParser();

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();

Inventor tesla = **new** Inventor("Nikola Tesla", "Serbian");

String name = parser.parseExpression("Name?:'Elvis Presley'").getValue(context, tesla, String.class);

System.out.println(name); *// Nikola Tesla*

tesla.setName(null);

name = parser.parseExpression("Name?:'Elvis Presley'").getValue(context, tesla, String.class);

System.out.println(name); *// Elvis Presley*

* + - 1. 安全导航操作符

安全导航操作符来自于[Groovy](http://www.groovy-lang.org/operators.html#_safe_navigation_operator)语言用于避免NullPointerException 。一般情况下在当你引用一个对象访问方法或或对象属性时需校验其不为空。为避免此操作，安全导航操作符则直接返回null而非抛出异常。

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–不会抛出 NullPointerException!!!*

Elvis运算符可应用于添加表达式默认值，如@Value内的表达式：

@Value("#{systemProperties['pop3.port'] ?: 25}")

这样将注入系统属性pop3.port值或没有的情况下为25

* + - 1. 集合选择器

选择器是表达式语言很强大的特性，允许通过选择其元素来将源集合转换为另一个。

选择器使用语法.?[selectionExpression]。这会过滤集合并返回包含源集合内元素子集的新集合。例如，选择器允许我们轻易获得Serbian inventors列表：

List<Inventor> list = (List<Inventor>) parser.parseExpression("Members.?[Nationality == 'Serbian']").getValue(societyContext);

选择器可用于list和map。在前述例子中选择条件针对list内的每一个元素都做筛选，而对于map结构选择条件则对每一个map内键值对做筛选（Java类型Map.Entry的对象）。Map键值对的键跟值需要在选择器内可访问。

如下表达式会返回一个新map，包含原始map内值小于27的元素。

Map newMap = parser.parseExpression("map.?[value<27]").getValue();

另外相对于返回所有选择的元素，也可以只获取第一个或最后一个。获取第一个符合条件的语法是.^[selectionExpression]，获取最后一个的语法是.$[selectionExpression]。

* + - 1. 集合投影

投影允许集合驱动子表达式求值，以生成新的集合，语法为.![projectionExpression]。假设我们有inventors列表，但需要这些inventors的出生城市列表。我们想直接在inventor列表内对每个元素解析'placeOfBirth.city'。使用投影：

*// returns ['Smiljan', 'Idvor' ]*

List placesOfBirth = (List)parser.parseExpression("Members.![placeOfBirth.city]");

map也可以用于驱动投影，此时投影表达式将针对map内的键值对（由Java类型Map.Entry表示）做解析。针对map的投影结果是对map键值对进行投影表达式分析而生成的列表。

* + - 1. 表达式模板

表达式模板允许一串语句带有多个解析块。每一个解析块都可以用自定义前缀和后缀符分隔开，一般选择#{ }作为分隔符。例如，

String randomPhrase = parser.parseExpression("random number is #{T(java.lang.Math).random()}",**new** TemplateParserContext()).getValue(String.class);

*// 解析为 "random number is 0.7038186818312008"*

表达式解析结果为拼接'random number is '和#{ }分隔符内表达式的结果，在本例中为调用random()方法。parseExpression()方法的第二个参数是ParserContext类型。接口ParserContext用于干预表达式的解析以支持模板表达式功能。TemplateParserContext 定义如下：

**public** **class** **TemplateParserContext** **implements** ParserContext {

**public** String getExpressionPrefix() {

**return** "#{";

}

**public** String getExpressionSuffix() {

**return** "}";

}

**public** **boolean** isTemplate() {

**return** true;

}

}

* + 1. 本节所用示例类

Inventor.java

**package** org.spring.samples.spel.inventor;

**import** java.util.Date;

**import** java.util.GregorianCalendar;

**public** **class** **Inventor** {

**private** String name;

**private** String nationality;

**private** String**[]** inventions;

**private** Date birthdate;

**private** PlaceOfBirth placeOfBirth;

**public** Inventor(String name, String nationality) {

GregorianCalendar c= **new** GregorianCalendar();

this.name = name;

this.nationality = nationality;

this.birthdate = c.getTime();

}

**public** Inventor(String name, Date birthdate, String nationality) {

this.name = name;

this.nationality = nationality;

this.birthdate = birthdate;

}

**public** Inventor() {

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

this.name = name;

}

**public** String getNationality() {

**return** nationality;

}

**public** **void** setNationality(String nationality) {

this.nationality = nationality;

}

**public** Date getBirthdate() {

**return** birthdate;

}

**public** **void** setBirthdate(Date birthdate) {

this.birthdate = birthdate;

}

**public** PlaceOfBirth getPlaceOfBirth() {

**return** placeOfBirth;

}

**public** **void** setPlaceOfBirth(PlaceOfBirth placeOfBirth) {

this.placeOfBirth = placeOfBirth;

}

**public** **void** setInventions(String**[]** inventions) {

this.inventions = inventions;

}

**public** String**[]** getInventions() {

**return** inventions;

}

}

PlaceOfBirth.java

**package** org.spring.samples.spel.inventor;

**public** **class** **PlaceOfBirth** {

**private** String city;

**private** String country;

**public** PlaceOfBirth(String city) {

this.city=city;

}

**public** PlaceOfBirth(String city, String country) {

this(city);

this.country = country;

}

**public** String getCity() {

**return** city;

}

**public** **void** setCity(String s) {

this.city = s;

}

**public** String getCountry() {

**return** country;

}

**public** **void** setCountry(String country) {

this.country = country;

}

}

Society.java

**package** org.spring.samples.spel.inventor;

**import** java.util.\*;

**public** **class** **Society** {

**private** String name;

**public** **static** String Advisors = "advisors";

**public** **static** String President = "president";

**private** List<Inventor> members = **new** ArrayList<Inventor>();

**private** Map officers = **new** HashMap();

**public** List getMembers() {

**return** members;

}

**public** Map getOfficers() {

**return** officers;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

this.name = name;

}

**public** **boolean** isMember(String name) {

**for** (Inventor inventor : members) {

**if** (inventor.getName().equals(name)) {

**return** true;

}

}

**return** false;

}

}

## Spring面向切面编程

* + 1. 介绍

面向切面编程（AOP）通过另一种对程序结构的思考补充了面向对象编程（OOP）。OOP的关键单元是类，而AOP的关键单元是切面。切面使得一些操作如事务管理，在横跨多个对象与类中切割，形成模块化（在AOP文献中这个点叫切点）

Spring的重要组件之一就是AOP框架。但Spring IoC容器不依赖AOP，意味着若不需要则可以不使用AOP，AOP给Spring IoC补充了许多中间件解决方案。

Spring 2.0+ AOP

Spring 2.0 给出了更简单且强大的，通过使用[基于schema方式](https://docs.spring.io/spring/docs/5.0.15.RELEASE/spring-framework-reference/core.html#aop-schema)或[@AspectJ注解风格](https://docs.spring.io/spring/docs/5.0.15.RELEASE/spring-framework-reference/core.html#aop-ataspectj)，编写自定义切面的方法。两种风格都提供了完全类型化的advice和使用AspectJ切点语言，而仍旧使用Spring AOP来编织。

Spring 2.0以上的基于schema和@AspectJ的AOP支持将在本章介绍。底层的AOP支持，在Spring 1.2应用内常见，在[下面章节](https://docs.spring.io/spring/docs/5.0.15.RELEASE/spring-framework-reference/core.html#aop-api)中讨论。

AOP在Spring框架内用于……

* …提供声明式企业服务，尤其是作为EJB声明式服务的替代。其中最重要的服务是[*声明式事务管理*](https://docs.spring.io/spring/docs/5.0.15.RELEASE/spring-framework-reference/core.html#transaction-declarative)。
* …允许用户实现自定义切面，以AOP补充他们的OOP实现。

如果你对通用声明式服务或其他预打包的声明式中间件服务如池，感兴趣，你可以不用直接与Spring AOP交互，可跳过本章大部分内容。

* + - 1. AOP概念

现在，我们先定义一些主要APO概念和术语。这些术语并非Spring指定，不幸的是，AOP术语并不够直观；然而若Spring使用其自己的术语那将更混乱。

* 切面：横切多个类的模块化点。事务管理就是在企业级Java应用中对横切点最好的示例。在Spring AOP中，切面由常规类实现（[基于schema方式](https://docs.spring.io/spring/docs/5.0.15.RELEASE/spring-framework-reference/core.html#aop-schema)）或常规类标记有@Aspect注解（[@AspectJ 方式](https://docs.spring.io/spring/docs/5.0.15.RELEASE/spring-framework-reference/core.html#aop-ataspectj)）。
* 连接点。程序执行时的点，如方法的执行或异常的处理。在Spring AOP中，连接点总是表示方法的执行。
* Advice增强：切面在特定连接点采取的动作。不同类型的advice包括”around”, “before”和”after” advice。（advice类型将在下述内容中讨论）一些AOP框架，包括Spring，将advice构建为拦截器，在连接点的前后保持着一条拦截器链。
* 切点。匹配连接点的的统一称谓。Advice与切点表达式协同运作，在由切点匹配的任意连接点上运行（例如，特定名称的方法执行）。切点表达式所匹配到的连接点是AOP的中心思想，Spring默认使用AspectJ切点表达式语言。
* Introduction引入：声明其他的方法和成员变量以丰富一个类型。Spring AOP允许对任意被切割的对象增加新的接口（及其相关实现类）。例如，你可以利用introduction来使得bean实现IsModified接口，以简化缓存。（introduction在AspectJ社区一般作为内部类声明）。
* 目标对象。被一个或多个切面切割的对象。也被称为被切割对象。由于Spring AOP通过运行时代理实现，因此此对象通常是被代理对象。
* AOP代理：由AOP框架创建用于实现切面协议（切割方法执行等等）的对象。在Spring框架，AOP代理是JDK动态代理或CGLIB代理。
* 织入：将切面与其他应用类型或对象连接以创建切割对象。可以在编译时执行（例如使用AspectJ编译器），加载时，或运行时。Spring AOP，像其他纯Java AOP框架一样，在运行时织入。

切割的类型：

* 前缀advice：在连接点前执行的行为，但不具备避免执行流走向连接点的能力（除非抛出异常）。
* 返回后advice：在连接点正常结束后执行的行为：例如，一个方法的完成且没有抛出异常。
* 抛出后advice：在方法抛出异常退出时执行的行为。
* 最终advice：无视连接点退出的方式（正常退出或异常抛出）退出时执行的行为。
* 围绕advice：包围连接点如方法调用的advice。这是最强大的advice类型。围绕型advice可以在方法调用前和调用后执行自定义行为。同样其负责选择是继续执行连接点内容还是通过返回该advice的返回值或抛出异常来切断方法的执行。

围绕型advice是最常用的advice类型。因为Spring AOP，如AspectJ，提供了全类型的advice，我们建议尽量按需使用。例如，如果你仅需要使用方法返回值来更新缓存，相比围绕型advice，你最好实现一个返回后advice，即便围绕型advice可以实现同样的功能。使用最准确的advice类型可以提供更简单的程序编写和更少的潜在错误。例如，你不需要在JoinPoint 上调用用于围绕型advice的proceed()方法，因此也不会出现调用失败。

在Spring 2.0，所有advice参数都已静态写好，因此可以使用合适的advice参数类型（例如方法执行的返回值类型）而不是用Object 数组。

连接点，由切点匹配，是AOP的关键概念，用于区分AOP与仅提供拦截功能的旧技术。切点允许advice不依赖于面向对象架构。例如，一个围绕型advice提供了声明式事务管理，可应用于跨越多个对象的多个方法（譬如在整个service层的业务操作）。

* + - 1. Spring AOP的能力与目标

Spring AOP是纯Java实现。没有其他特殊的编译要求。Spring AOP不需要控制类加载器结构，因此其适合用于servlet容器或应用服务器内。

Spring AOP当前仅支持方法执行连接点（在Spring bean的方法执行上切割）。成员变量拦截没有实现，即便成员变量拦截可以在不影响Spring AOP API的情况下添加。若你需要在成员变量访问和更新上切割，考虑ApsectJ语言。

Spring AOP实现AOP的方式与大部分AOP框架不同。其目的不是提供最全的AOP实现（即使Spring AOP也可做到）；而是提供AOP实现与Spring IoC的紧密整合以解决企业应用的大部分问题。

因此，例如，Spring框架的AOP功能经常与Spring IoC容器的联合使用。切面用常规的bean定义语法来配置（即使其允许强大的“自动代理”功能）：这就是与其他的AOP实现有很大的不同；Spring AOP有时并不简单或高效，如切割非常细粒度的对象（如域对象）：在这种情况下AspectJ是最好的选择。然而以我们的经验，Spring AOP提供了企业级Java应用的极佳解决方案，是经过考验的AOP框架。

Spring AOP永不会与AspectJ力争提供最好的AOP解决方案。我们相信基于代理的框架如Spring AOP和成熟框架如AspectJ都是有价值的，并且是互补而不是竞争关系。Spring通过AspectJ无缝整合Spring AOP和IoC，以在始终基于Spring的应用架构内满足所有AOP的需求。整合没有影响Spring AOP API或者AOP联盟API：Spring AOP保持向后兼容性。关于Spring AOP API详见[如下章节](https://docs.spring.io/spring/docs/5.0.16.RELEASE/spring-framework-reference/core.html#aop-api)。

Spring的其中一条宗旨是非侵入性；也就是说你不应该被迫在你的业务/域模型内引入框架指定的类和接口。然而在某种情况下Spring框架确实提供了引入Spring框架指定的依赖到你的代码库中：提供该选择的基本原理是因为在某种特定情况下为了可读性或完成某种特殊的功能。Spring框架几乎总是给你提供选择：你可以自由选择哪种选项最符合你的使用场景。

其中一个选择与本章相关，就是选择哪种AOP框架（亦即哪种AOP风格）。可以选AspectJ和/或Spring AOP，你也可以选@AspectJ注解风格或Spring XML配置风格。本章首先介绍@AspectJ风格并不是在暗示Spring团队喜欢@AspectJ注解风格过Spring XML配置风格。

有关为什么和在哪里选择哪种风格的讨论见[选择使用何种AOP声明风格](https://docs.spring.io/spring/docs/5.0.16.RELEASE/spring-framework-reference/core.html#aop-choosing)。

* + - 1. AOP代理

Spring AOP默认使用标准JDK动态代理作为AOP代理。这允许任意接口（或接口集）被代理。

Spring AOP也可以使用CGLIB代理。这就需要代理类而不是接口。CGLIB在业务对象没有实现接口时默认使用。最好的做法是针对接口编程而不是对类；业务类一般实现一个或多个业务接口。可以[强制使用CGLIB](https://docs.spring.io/spring/docs/5.0.16.RELEASE/spring-framework-reference/core.html#aop-proxying)，在那些（极少数）需要切割没有在接口内没有声明的方法，或者需要将代理对象作为具体类型传递给方法的情况下。

重点是要抓住Spring AOP是基于代理的概念。参阅 [了解AOP代理](https://docs.spring.io/spring/docs/5.0.16.RELEASE/spring-framework-reference/core.html#aop-understanding-aop-proxies)以全面了解此实现细节的实际含义。

* + 1. @ApsectJ支持

@AspectJ提供了一种风格，声明切面为普通带有注解的Java类。@AspectJ风格在AspectJ 5版本中作为[AspectJ项目](https://www.eclipse.org/aspectj)引入。Spring解释了AspectJ 5的相同注解，使用AspectJ提供的库用于切点解析和匹配。但AOP运行时还是纯Spring AOP，且没有依赖AspectJ编译器和织入器。

使用AspectJ编译器和织入器可启用完整的AspectJ语言，详见 [Spring应用内使用AspectJ](https://docs.spring.io/spring/docs/5.0.16.RELEASE/spring-framework-reference/core.html#aop-using-aspectj) 。

* + - 1. 启用@AspectJ支持

在Spring配置内使用@AspectJ切面需要启用Spring的有关支持以实现基于@AspectJ切面配置Spring AOP，并基于bean是否应该被切面切割而自动代理。对于自动代理我们指的是，如果Spring决定一个bean是否应该被一个或多个切面切割，其将据需自动对该bean生成一个代理，以拦截方法调用并保证advice正常执行。

@AspectJ支持可以通过XML或Java配置风格启用。另外你还要确认AspectJ的aspectjweaver.jar库在你的应用的类路径下（1.8或更高版本）。这个库在AspectJ发行版的'lib'目录内可以找到，或通过Maven中心仓库找到。

通过Java配置开启@AspectJ支持

通过Java配置启用@AspectJ支持需要添加@EnableAspectJAutoProxy注解：

@Configuration

@EnableAspectJAutoProxy

**public** **class** **AppConfig** {

}

通过XML配置启用@AspectJ支持

在XML配置内启用@AspectJ支持需要使用元素aop:aspectj-autoproxy：

<aop:aspectj-autoproxy/>

此时假定你已经在使用如[基于XML规范配置](https://docs.spring.io/spring/docs/5.0.16.RELEASE/spring-framework-reference/core.html#xsd-schemas)所述的规范支持。如何引入aop命名空间内的标签，参阅[AOP规范文件](https://docs.spring.io/spring/docs/5.0.16.RELEASE/spring-framework-reference/core.html#xsd-schemas-aop)。

* + - 1. 声明一个切面

在@AspectJ支持启动的情况下，定义在在应用上下文内的，类文件为一个@AspectJ切面（即类上有@Aspect注解）的任意bean，都会被Spring自动检测到并用于配置Spring AOP。如下示例展示了切面的最简配置，但没有用：

应用上下文内常规的bean定义，指向带有@Aspect注解的bean类：

<bean id="myAspect" class="org.xyz.NotVeryUsefulAspect">

*<!-- 正常配置切面属性 -->*

</bean>

NotVeryUsefulAspect 类定义标注有org.aspectj.lang.annotation.Aspect注解；

**package** org.xyz;

**import** org.aspectj.lang.annotation.Aspect;

@Aspect

**public** **class** **NotVeryUsefulAspect** {

}

切面（标注有@Aspect的类）可以跟其他类一样有成员变量和方法。也可以包含切点，advice增强，和introduction引入（内部类）声明。

**通过组件扫描自动检测切面**

你可以在你的Spring XML配置内正常注册切面类，或通过类路径扫描自动检测——就像其他受Spring管理的bean。然而，注意@Aspect注解不满足类路径的自动检测：你需要多加一个@Component注解已实现自动检测（或者自定义的模板注解，符合Spring组件扫描器的规则）。

**切面上的切面？**

在Spring AOP中，不会有切面成为其他切面需要切割的对象。@Aspect注解在类上即将其标记为切面，因此将其从自动代理中排除。

* + - 1. 声明切点

记住切点决定了对应的相关连接点，因此其允许我们控制何时执行advice增强。Spring AOP仅支持对Spring bean的方法执行连接点，因此你可以考虑切点匹配于Spring bean上的方法执行时。切点声明包含两点：带有签名的名称和任意数量参数，且一个切点表达式决定了具体哪个方法执行是需要切割的。在@AspectJ注解风格的AOP中，切点的签名由常规的方法定义提供，且切点表达式指定由@Pointcut注解完成（用于描述切点的方法签名必须返回值类型为void ）。

示例将有助于清晰区分切点签名和切点表达式。如下示例定义了一个名为'anyOldTransfer'的切点且其将匹配任意名为'transfer'方法的执行：

@Pointcut("execution(\* transfer(..))")*// 切点表达式*

**private** **void** anyOldTransfer() {}*// 切点签名*

切点表达式，即@Pointcut注解内的值的格式就是常规的AspectJ 5切点表达式。有关AspectJ的切点表达式语言的详细论述，见[AspectJ Programming Guide](https://www.eclipse.org/aspectj/doc/released/progguide/index.html)（有关扩展见[AspectJ 5 Developers Notebook](https://www.eclipse.org/aspectj/doc/released/adk15notebook/index.html)）或者有关AspectJ的书如《Eclipse AspectJ》Colyer et. al.著，《AspectJ in Action》Ramnivas Laddad.著

支持的切点标识符

Spring AOP支持如下AspectJ切点标识符（PCD）用于切点表达式内：

其他切点类型

完整的AspectJ切点表达式语言支持除在Spring以外的其他切点标识符。分别是：call, get, set, preinitialization, staticinitialization, initialization, handler, adviceexecution, withincode, cflow, cflowbelow, if, @this, 和 @withincode。在切点表达式内使用这些标识符，由Spring AOP解释执行将抛出IllegalArgumentException。

由Spring AOP支持的切点标识符将会在未来版本中扩展以支持更多AspectJ的切点标识符。

* execution – 用于匹配方法执行连接点，这是使用Spring AOP时主要使用的切点标识符。
* within – 限制匹配连接点为特定类型（使用Spring AOP时限定所执行的方法在某类型内）。
* this – 限制匹配连接点（使用Spring AOP时的方法执行）为当bean引用（Spring AOP代理）是给定类型的实例。
* target – 限制匹配连接点（使用Spring AOP时的方法执行）为当目标对象（被代理的应用对象）是给定类型的实例。
* @target – 限制匹配连接点（使用Spring AOP时的方法执行）为执行对象的类上有给定类型的注解
* @args – 限制匹配连接点（使用Spring AOP时的方法执行）为当运行时真实的入参类型上有给定的注解类型。
* @within – 限制匹配连接点在具有给定注解类型的类内（使用Spring AOP时方法执行声明在具有给定注解的类内）
* @annotation – 限制匹配连接点为连接点（使用Spring AOP时的方法执行）具有给定的注解。

由于Spring AOP限制仅匹配方法执行连接点，上述的切点标识符的定义比在AspectJ编程指引内范围更小更狭义。另外，AspectJ有基于类型的语义且在执行时连接点内this 和target 指向同一个对象 – 执行该方法的对象。Spring AOP是基于代理的系统，因此代理对象本身（绑定于this）与在代理后的目标对象（绑定于target）不同。

由于Spring AOP框架基于代理的自然特性，在目标对象内的调用自然不会被拦截。对于JDK代理，仅仅在调用代理上的公共接口方法会被拦截。对于CGLIB，public和protected方法在代理上调用时会被拦截，如果需要，包可见方法也可以被拦截。然而，通常通过代理的交互应该设计为通过public来交互。

注意切点定义通常匹配任意被拦截的方法。如果切点需要严格限制为仅public方法，甚至在CGLIB代理情景下代理可能出现的非public方法交互，那么就需要准确定义切点。

如果你的拦截需要包含在类内的方法调用，甚至构造函数调用，可以考虑使用Spring驱动的[本地AspectJ编织](https://docs.spring.io/spring/docs/5.0.16.RELEASE/spring-framework-reference/core.html#aop-aj-ltw)而不是Spring基于代理的AOP框架。这形成了不同特征的不同模式AOP运用，因此在做决定之前首先确保对编织有足够的熟悉。

Spring AOP支持额外的PCD（切点标识符）bean。这个PCD允许你限制匹配连接点为特定名称的Spring bean，或一系列已命名的Spring bean（当使用通配符时）。bean PCD有如下格式：

bean(idOrNameOfBean)

idOrNameOfBean 处可以是任意Spring bean的名字：限制通配符支持\* ，因此你若给你的Spring bean建立了某些命名规范则便可轻易编写bean PCD表达式来选出它们。在与其他切点标识符的共同使用时，bean PCD也可以“&&”，“||”，“!”（与或非）。

请注意beanPCD仅在Spring AOP内支持 – 不在本地AspectJ编织内。这是Spring对AspectJ定义的标准PCD的扩展，因此不可用于在@Aspect模型内声明的切面。

beanPCD在实例层操作（在Spring bean名上构建）而不是仅在类型层（也是基于编织的AOP的操作层）。基于实例的切点标识符是基于代理的Spring AOP框架的特殊能力，并与Spring bean工厂紧密结合，可以自然并直接通过名字识别对应的bean。

组合切点表达式

切点表达式可以通过“&&”，“||”，“!”组合。同样可以通过名称引用切点表达式。如下实例展示三个切点表达式：anyPublicOperation（匹配方法连接点为任意public方法的执行）；inTrading（匹配在trading模块里的方法执行），和tradingOperation（匹配在trading模块里的任意public方法执行）。

@Pointcut("execution(public \* \*(..))")

**private** **void** anyPublicOperation() {}

@Pointcut("within(com.xyz.someapp.trading..\*)")

**private** **void** inTrading() {}

@Pointcut("anyPublicOperation() && inTrading()")

**private** **void** tradingOperation() {}

最佳实践是在较小命名组件外构建复杂的切点表达式，如上所示。当通过名字引用切点时，应用常规Java可见性规则（你可以在同一个类内看到私有切点，继承结构内看到protected切点，任意位置看到public等等）。可见性不会影响切点匹配。

共享通用切点定义

企业级应用内，经常需要引用应用内的某些模块和在一些切面内执行一系列对应的操作。我们推荐定义“SystemArchitecture”切面，以捕捉通用的切点表达式。此类切面的示例如下：

**package** com.xyz.someapp;

**import** org.aspectj.lang.annotation.Aspect;

**import** org.aspectj.lang.annotation.Pointcut;

@Aspect

**public** **class** **SystemArchitecture** {

*/\*\**

*\* web层连接点，方法定义在com.xyz.someapp.web或其子包内*

*\*/*

@Pointcut("within(com.xyz.someapp.web..\*)")

**public** **void** inWebLayer() {}

*/\*\**

*\* service连接点，方法定义在com.xyz.someapp.service包或其子包内*

*\*/*

@Pointcut("within(com.xyz.someapp.service..\*)")

**public** **void** inServiceLayer() {}

*/\*\**

*\* dao连接点，方法定义在com.xyz.someapp.service包或其子包内*

*\*/*

@Pointcut("within(com.xyz.someapp.dao..\*)")

**public** **void** inDataAccessLayer() {}

*/\*\**

*\* 业务service是定义在service接口内的任意方法执行，该定义假设接口位于*

*\* "service"包内, 且其实现类在其子包内*

*\**

*\* 若你通过功能来分组service接口（例如，在包*

*\* com.xyz.someapp.abc.service和com.xyz.someapp.def.service)此时*

*\* 切点表达式为 "execution(\* com.xyz.someapp..service.\*.\*(..))"*

*\**

*\**

*\* 或者，你可通过使用'bean'PCD来编写表达式*

*\* 如"bean(\*Service)"。（此则假设你已以固定的风格命名spring service*

*\* beans）*

*\*/*

@Pointcut("execution(\* com.xyz.someapp..service.\*.\*(..))")

**public** **void** businessService() {}

*/\*\**

*\* 数据访问操作则是定义在dao接口内的任意方法*

*\* 此定义则假设接口都位于"dao"包及其实现类在子包内*

*\*/*

@Pointcut("execution(\* com.xyz.someapp.dao.\*.\*(..))")

**public** **void** dataAccessOperation() {}

}

在该切面内定义的切点可在任意需要切点表达式时引用。例如，为使service层支持事务，如下：

<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>

<aop:config>和<aop:advisor>标签在[基于schema的AOP支持](https://docs.spring.io/spring/docs/5.0.16.RELEASE/spring-framework-reference/core.html#aop-schema)论述。事务标签则在[事务管理](https://docs.spring.io/spring/docs/5.0.16.RELEASE/spring-framework-reference/data-access.html#transaction)。

示例

Spring AOP使用者可能最常用execution 切点标识符。格式如下：

execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?name-pattern(param-pattern) **throws**-pattern?)

除返回类型、名称、入参以外，其他参数可选。返回类型决定方法的返回值类型以便连接点匹配。此处常见\* ，表示任意返回值类型。全限定类型名将在方法为给定类型匹配。名称匹配方法名。可用\* 通配符表示所有或部分名称。若指定声明类型时需以. 结尾因而与名称拼接。入参稍微复杂点：()匹配无参方法，(..)匹配任意参数方法（0个或多个）。(\*)则表示方法有一个任意类型的参数，(\*,String)则匹配带有两个参数的方法并且其第一个参数可为任意类型，第二个必须为String。详见AspectJ编程指引的[语言语义](https://www.eclipse.org/aspectj/doc/released/progguide/semantics-pointcuts.html)。

部分常用切点表达式示例如下：

* 任意public方法的执行：

execution(**public** \* \*(..))

* 任意方法名带有“set”的方法执行：

execution(\* set\*(..))

* 任意由AccountService 接口定义的方法执行：

execution(\* com.xyz.service.AccountService.\*(..))

* 任意定义在service包内的方法执行：

execution(\* com.xyz.service.\*.\*(..))

* 任意定义在service包及其子包内的方法执行

execution(\* com.xyz.service..\*.\*(..))

* 任意在service包内的连接点（仅表示在Spring AOP内的方法执行）：

within(com.xyz.service.\*)

* 任意在service包及其子包内的连接点仅表示在Spring AOP内的方法执行）：

within(com.xyz.service..\*)

* 任意在实现AccountService接口的代理内的连接点（仅表示在Spring AOP内的方法执行）：

this(com.xyz.service.AccountService)

“this”常用于绑定格式：见下小节的advice增强内，如何使得代理对象可用于增强体内

* 任意在实现AccountService接口的目标对象（被代理对象）内的连接点（仅表示在Spring AOP内的方法执行）：

target(com.xyz.service.AccountService)

“target” 常用于绑定格式：见下小节的advice增强内，如何使得代理对象可用于增强体内

* 任意单入参且运行时传入类型为Serializable的连接点（仅表示在Spring AOP内的方法执行）：

args(java.io.Serializable)

“args” 常用于绑定格式：见下小节的advice增强内，如何使得代理对象可用于增强体内

注意在此例中给出的切点表达式不同于execution(\* \*(java.io.Serializable))：args表示运行时入参类型为Serializable，而execution则表示匹配到的方法签名声明是Serializable类型的单参数。

* 目标对象有@Transactional注解的任意连接点（仅表示在Spring AOP内的方法执行）：

@target(org.springframework.transaction.annotation.Transactional)

“@target”常用于绑定格式：见下小节的advice增强内，如何使得代理对象可用于增强体内

* 任意目标对象声明类上带有@Transactional注解的连接点（仅表示在Spring AOP内的方法执行）：

@within(org.springframework.transaction.annotation.Transactional)

“@within”常用于绑定格式：见下小节的advice增强内，如何使得代理对象可用于增强体内

* 任意单参数，且运行时入参类型带有@Classified注解的连接点（仅表示在Spring AOP内的方法执行）：

@args(com.xyz.security.Classified)

“@args”常用于绑定格式：见下小节的advice增强内，如何使得代理对象可用于增强体内

* 任意在Spring bean名为tradeService的连接点（仅表示在Spring AOP内的方法执行）：

bean(tradeService)

任意在Spring bean中命名符合通配符表达式\*Service的连接点仅表示在Spring AOP内的方法执行）：

bean(\*Service)

编写良好的切点表达式

编译时，AspectJ处理切点以使其匹配性能最大化。测试代码并确定每一个连接点与给定的切点匹配（静态或动态）是否性能不佳。（动态匹配意味着不能完全通过静态分析来确定且此时测试代码将会被放置于此处以确认在代码运行时是否真正匹配上）。在首次遇到切点声明时，AspectJ将其以合适格式重写以适配匹配进程。什么意思？基本上切点都会以DNF(析取范式)格式重写，切点的组件会被排序，因此那些易解析组件将先被检查。这也就意味着你不需要去理解多个切点表达式的性能并在声明切点时提供顺序。

然而，AspectJ只能根据给出的切面声明来生成切面，为使得匹配性能最优你需要在定义切面时仔细考虑需要达成的效果以尽可能缩小需要匹配的范围。已有的标识符存在于以下三个组内：kinded精准、scoping范围和context上下文：

* 精准标识符可以准确选择某类连接点。如：execution，get，set，call，handler
* 范围标识符选择某一组别连接点（可能为多种类型）。如：within，withincode
* 上下文型标识符则需要基于上下文来匹配（或且绑定）如：this，target，@annotation

编写良好的切点应尝试包含至少前两种类型（kinded和scoping），尽管上下文型标识符可能包含在内，用于基于连接点上下文匹配，或将上下文绑定用于增强内。仅提供一个精准标识符或一个上下文标识符也可正常执行但由于额外的处理和分析将影响织入性能（时间和内存消耗）。范围标识符匹配时非常快，且其使用则意味着AspectJ可以很快的忽略其他组连接点并后续不再对其处理——这也就是为什么一个好的切点应总是尽量包含之。

* + - 1. 声明advice增强

增强与切点表达式相关，并在匹配到切点的方法执行时，执行前、后或环绕类型的增强。切点表达式可以是简单的已命名切点的引用，或者是一个直接声明的切点表达式。

前增强

前增强可在切面内使用@Before注解来声明：

**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() {

*// ...*

}

}

使用本地切点表达式则上述示例可重写为：

**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() {

*// ...*

}

}

返回后增强

返回后增强运行在匹配方法执行正常返回时。使用@AfterReturning注解声明：

**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() {

*// ...*

}

}

注意：一个切面内当然可以出现多个增强声明，及其他成员。此处只是展示了一个简单的增强声明以关注其特点。

有时你需要在增强体内访问返回的真实值。你可以使用@AfterReturning的格式以绑定返回值：

**import** org.aspectj.lang.annotation.Aspect;

**import** org.aspectj.lang.annotation.AfterReturning;

@Aspect

**public** **class** **AfterReturningExample** {

@AfterReturning(pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()",returning="retVal")

**public** **void** doAccessCheck(Object retVal) {

*// ...*

}

}

returning属性的命名必须与增强方法参数名保持一致。当方法执行返回时，返回值将通过该参数传递给增强方法。returning同样限制匹配方法的返回值为指定类型（此处使用Object，表示任意返回值）。

请注意在使用返回后增强时不可能返回一个完全不同的引用。

抛出后增强

抛出后增强运行在匹配方法执行因抛出异常退出时。使用@AfterThrowing注解声明如下：

**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() {

*// ...*

}

}

一般你希望增强仅在给定类型异常抛出时运行，且需要在增强体内访问。使用throwing属性限制匹配和绑定抛出的异常为增强入参（若需要，使用Throwable作为异常类型）。

**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属性的值需要与增强方法的参数名保持一致。当方法执行通过抛出异常退出时，异常值将以方法相关入参的方式传递。throwing限制匹配方法抛出异常的类型为指定类型（此处异常类型为DataAccessException）。

后（最终）增强

最终增强不管方法如何退出均会运行。使用@After注解来声明。后增强必须考虑方法正常退出和异常退出两种情况。通常用来释放资源，等。

**import** org.aspectj.lang.annotation.Aspect;

**import** org.aspectj.lang.annotation.After;

@Aspect

**public** **class** **AfterFinallyExample** {

@After("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")

**public** **void** doReleaseLock() {

*// ...*

}

}

环绕增强

最后一种增强方式是环绕增强。环绕增强运行“环绕于”匹配方法的执行。其有机会在方法执行前后都有对应行为，来决定方法执行时间，方式甚至是否执行。环绕增强一般用于在你需要在方法执行前后以线程安全的方式共享状态（如开始和停止一个计时器）。注意尽量使用贴近需求的增强（即能使用前增强或后增强即可达成需求时就不用环绕增强）。

环绕增强使用@Around注解来声明。增强方法的首个参数类型必须为ProceedingJoinPoint。在增强体内，调用ProceedingJoinPoint 的proceed()来使得被代理方法得以执行。proceed 方法也可通过传递Object[]来调用——该数组内的值将用于方法执行时的入参。

通过Object[]来调用proceed的行为与由AspectJ编译器编译出的环绕增强的proceed有些许不同。对于使用传统AspectJ语言编写的环绕增强，传递给下一层方法执行的参数数量必须与传递给环绕增强（而不是基础连接点的参数数量）的保持一致，并且传递给给定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，并且更好地匹配基于代理的仅执行语义。你只需要意识到这种区别，在需要编译为Spring形式的@AspectJ切面，但又要使用AspectJ编译器和织入器中使用proceed来处理参数，。有一种方法可以在Spring AOP和AspectJ之间100％兼容，并且在下面有关增强参数的部分中对此进行了讨论。

**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;

}

}

环绕增强的返回值将是方法调用方看到的返回值。例如，一个简单的缓存切面将在缓存存在时返回缓存值，并在不存在缓存时调用proceed()。注意，proceed在环绕增强体内可被调用不止一次或不调用，这些都是合法的。

增强参数

Spring支持全限定类型增强—也就是说你可以在增强签名中声明你需要的参数类型（如上所述的返回和抛出示例）而不是一直使用Object[]数组编码。我们将看到参数和其他有关的上下文值可用于增强体内。首先我们先看如何编写通用增强，以了解该增强当前所增强的方法。

访问当前连接点

任意增强方法都需要声明其第一个参数类型为org.aspectj.lang.JoinPoint（注意环绕增强则需要声明其第一个参数类型为ProceedingJoinPoint，其为JoinPoint的子类）。JoinPoint 接口提供了一些有用的方法如getArgs()（返回方法参数），getThis()（返回代理对象），getTarget()（返回目标对象），getSignature()（返回被增强方法的描述签名）和toString()（打印被增强方法的描述）。详见javadocs。

向增强传参

我们已经看到如何绑定返回值或异常值（使用返回后和抛出后增强）。为使参数可用于增强内，你可以使用绑定语法args。如果在args表达式内使用参数名而非类型，则在增强被调用时相关联的参数值将会作为方法入参传递。示例将有助于理解。假设你需要增强一个以Account对象为第一个参数的dao操作方法，需要在增强体内访问account，则可以如此写：

@Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation() && args(account,..)")

**public** **void** validateAccount(Account account) {

*// ...*

}

切点表达式内args(account,..)的有两层意思：首先，其限定匹配的方法有至少一个参数，且传递的参数为Account的实例；其次，通过account 参数使得真实的Account 对象可用于增强内。

另一种方法则是，声明一个切点用于在匹配到连接点时“提供”Account 对象，然后只需要在增强内引用该命名切点，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) {

*// ...*

}

感兴趣的读者可以查询AspectJ编码指引以获取详情。

代理对象（this），目标对象（target），和注解（@within, @target, @annotation, @args）都可以以相同格式绑定。如下示例则是展示了匹配带有@Auditable注解的方法执行，及audit相关的代码

首先定义@Auditable注解：

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.METHOD)

**public** @interface Auditable {

AuditCode value();

}

然后增强匹配@Auditable方法的执行：

@Before("com.xyz.lib.Pointcuts.anyPublicMethod() && @annotation(auditable)")

**public** **void** audit(Auditable auditable) {

AuditCode code = auditable.value();

*// ...*

}

增强参数和泛型

Spring AOP可以处理在类声明和方法参数中的泛型。假设你有泛型如下：

**public** **interface** **Sample**<T> {

**void** sampleGenericMethod(T param);

**void** sampleGenericCollectionMethod(Collection<T> param);

}

你可以通过简单键入需要拦截方法的类型在增强参数内，以达到限定方法类型为特定类型的目的

@Before("execution(\* ..Sample+.sampleGenericMethod(\*)) && args(param)")

**public** **void** beforeSampleMethod(MyType param) {

*// Advice implementation*

}

如上所述，这样显然可行。然而，需要注意的是不可以在泛型集合内如此定义，即如下定义为无效：

@Before("execution(\* ..Sample+.sampleGenericCollectionMethod(\*)) && args(param)")

**public** **void** beforeSampleMethod(Collection<MyType> param) {

*// Advice implementation*

}

若需生效则我们需要查看集合内的每一个元素，这不合理，且我们也无法如何处理null 值。若要达到类似效果则需要将参数类型改为Collection<?>并手动检查元素类型。

确定参数名

绑定在增强调用的参数名依赖于切点表达式中使用的名称与（增强和切点）方法签名中声明的参数名称的匹配。参数名无法通过Java反射获取，因此Spring AOP使用如下策略确定参数名：

* 如果参数名由用户显式指定，则将使用指定的参数名：增强和切点注解都有一个“argNames”属性选项，可用于指定被注解方法的参数名—这些参数名在运行时可用。比如：

@Before(value="com.xyz.lib.Pointcuts.anyPublicMethod() && target(bean) && @annotation(auditable)",argNames="bean,auditable")

**public** **void** audit(Object bean, Auditable auditable) {

AuditCode code = auditable.value();

*// ... use code and bean*

}

若第一个参数是JoinPoint，ProceedingJoinPoint，或JoinPoint.StaticPart类型，则可以在“argNames”属性内省略该参数名。例如，若你更改前述增强以接纳连接点对象，“argNames”属性则不需要将其包含在内：

@Before(value="com.xyz.lib.Pointcuts.anyPublicMethod() && target(bean) && @annotation(auditable)", argNames="bean,auditable")

**public** **void** audit(JoinPoint jp, Object bean, Auditable auditable) {

AuditCode code = auditable.value();

*// ... use code, bean, and jp*

}

首参数为JoinPoint，ProceedingJoinPoint，或JoinPoint.StaticPart类型的特殊处理对于不需要收集任意连接点上下文的增强非常方便。在这种情况下你可以直接忽略“argNames”属性。例如，如下增强不需要声明“argNames”属性：

@Before("com.xyz.lib.Pointcuts.anyPublicMethod()")

**public** **void** audit(JoinPoint jp) {

*// ... use jp*

}

* 使用'argNames'属性有点臃肿，所以在该属性没有被指定的情况下，Spring AOP会查看class文件的debug信息以尝试从本地变量表内确定参数名。只要class在带有debug信息的情况下编译（'-g:vars'至少）生成，这些信息就会存在。带有此标记的编译结果将有：（1）你的代码将易于理解（逆向工程），（2）class文件将稍大一些（通常无关紧要），（3）你的编译器将不会使用删除无用本地变量的优化。也就是说，启用该标记构建没有困难。

若@AspectJ切面由AspectJ编译器（ajc）编译，即使没有debug信息也不需要添加argNames属性因为编译器会保留所需信息

* 若代码编译时没有带有必要的debug信息，则Spring AOP会尝试推测绑定变量与参数的匹配（若只有一个变量在切点表达式内绑定，且增强方法仅接收一个参数，这就显而易见）。若绑定值与给定信息有歧义，则抛出AmbiguousBindingException 异常。
* 若上述策略均失效则抛出IllegalArgumentException 异常。

进行参数处理

我们前面提到我们会描述如何编写在Spring AOP和AspectJ上运行结果一致的带有参数的进一步调用。方法是只需要保证增强的签名绑定的方法参数按照固定顺序即可。例如：

@Around("execution(List<Account> find\*(..)) && " +

"com.xyz.myapp.SystemArchitecture.inDataAccessLayer() && " +

"args(accountHolderNamePattern)")

**public** Object preProcessQueryPattern(ProceedingJoinPoint pjp,

String accountHolderNamePattern) **throws** Throwable {

String newPattern = preProcess(accountHolderNamePattern);

**return** pjp.proceed(**new** Object**[]** {newPattern});

}

大多数情况下你都要做此绑定（如上例所示）。

增强顺序

若多个增强均需要在同一个连接点上执行会发生什么？Spring AOP遵循与AspectJ同样的规则来决定增强的执行顺序。最高优先级的增强首先在“进入时”时执行（即给定两个前增强，高优先级的先运行）。从连接点“出来时”，最高优先级增强最后执行（即给定两个后增强，具有高优先级的第二执行）。

当两个定义在不同切面的增强但需在同一连接点上时，除非指定否则执行顺序是未定义的。你可以通过指定优先级控制执行顺序。在Spring可通过实现org.springframework.core.Ordered接口或标记Order 注解在切面类上来控制顺序。给定两个切面，Ordered.getValue()返回值（或注解值）较小者具有较高的优先级。

当定义在同一个切面的两个增强需要在同一连接点执行时，顺序是未定义的（由于不能对经javac编译的class通过反射获取其声明顺序）。考虑在每一个切面类的将同一连接点的增强合并为一个，或者将增强重构为不同的切面类—因此可通过切面顺序来确定增强顺序。

* + - 1. 引入

引入（即AspectJ的内部类声明）允许切面对被增强对象声明为实现一个给定的接口，并提供一个该接口的实现以代表被增强对象。

使用@DeclareParents注解实现引入。此注解用于声明匹配的类型有一个新的父类（根据名称）。例如，给定一个接口UsageTracked，以及该接口实现DefaultUsageTracked，如下切面声明了所有service接口的实现同样实现了UsageTracked 接口。（如为了通过JMX暴露数据）

@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();

}

}

需要实现的接口取决于标记的成员变量。@DeclareParents注解的value 属性是AspectJ类型范式：任意匹配类型的bean均将实现UsageTracked接口。注意在上例中的前增强内，service bean可以直接用于UsageTracked 接口的实现。若编程式访问该bean即可如此：

UsageTracked usageTracked=(UsageTracked)context.getBean("myService");

* + - 1. 切面实例化模型

本小节为拓展小节，若你刚开始使用AOP则可直接跳过

默认情况下在应用上下文内每个切面都是单实例的。AspectJ将此称为单实例模型。也可以以不同的生命周期定义切面：Spring支持AspectJ的perthis和pertarget 实例模型（当前不支持percflow, percflowbelow,和pertypewithin ）。

一个“perthis”切面通过在@Aspect注解上指定一个perthis 块。如下例所示，然后我们将解释其工作原理。

@Aspect("perthis(com.xyz.myapp.SystemArchitecture.businessService())")

**public** **class** **MyAspect** {

**private** **int** someState;

@Before(com.xyz.myapp.SystemArchitecture.businessService())

**public** **void** recordServiceUsage() {

*// ...*

}

}

'perthis'块的效果是一个切面实例将由每一个唯一的执行业务逻辑的service对象创建（每一个唯一对象绑定到由切点表达式匹配的连接点的“this”上）。切面实力在service对象方法第一次被调用时创建。在service对象在作用域之外时切面对象也在作用域外。在切面实例创建前，其内部没有一个增强执行。只要切面实例被创建，声明在其中的增强将在匹配的连接点上执行，但仅当service对象与当前切面关联时。详见AspectJ编码指引的per块部分。

'pertarget'实例模型与perthis一样，只是对每一个唯一的匹配连接点的目标对象创建一个切面。

* + - 1. 示例

现在你已经看到所有组件的工作方式，现在让我们把它们组合起来付诸实施！

业务服务的执行有时会由于并发问题而失败（例如死锁）。如果该操作重试，下一次极有可能成功。对于业务服务在该种情形下可以重试的（不用返回至用户端解决冲突的幂等操作），这样我们就可以透明地重试该操作而避免用户看到PessimisticLockingFailureException。这显然是个在service层跨越多个service切割的需求，因此理想实现是通过切面。

因为我们想重试操作，我们将需要使用围绕增强以可多次调用。如下代码所示：

@Aspect

**public** **class** **ConcurrentOperationExecutor** **implements** Ordered {

**private** **static** **final** **int** DEFAULT\_MAX\_RETRIES = 2;

**private** **int** maxRetries = DEFAULT\_MAX\_RETRIES;

**private** **int** order = 1;

**public** **void** setMaxRetries(**int** maxRetries) {

this.maxRetries = maxRetries;

}

**public** **int** getOrder() {

**return** this.order;

}

**public** **void** setOrder(**int** order) {

this.order = order;

}

@Around("com.xyz.myapp.SystemArchitecture.businessService()")

**public** Object doConcurrentOperation(ProceedingJoinPoint pjp) **throws** Throwable {

**int** numAttempts = 0;

PessimisticLockingFailureException lockFailureException;

**do** {

numAttempts++;

**try** {

**return** pjp.proceed();

}

**catch**(PessimisticLockingFailureException ex) {

lockFailureException = ex;

}

} **while**(numAttempts <= this.maxRetries);

**throw** lockFailureException;

}

}

注意该切面实现了Ordered 接口因此我们可以设置切面的优先级高于事务增强（我们期望每次重试都是一个全新的事务）。maxRetries 和order 属性可以通过Spring配置。主要行为在围绕增强doConcurrentOperation 内。注意我们已将该重试逻辑应用于所有的businessService()。我们尝试执行，并在因PessimisticLockingFailureException失败时直接重试除非达到最大重试次数。

相关Spring配置为：

<aop:aspectj-autoproxy/>

<bean id="concurrentOperationExecutor" class="com.xyz.myapp.service.impl.ConcurrentOperationExecutor">

<property name="maxRetries" value="3"/>

<property name="order" value="100"/>

</bean>

若需改进该切面以仅重试幂等性操作时，我们需要定义一个Idempotent 注解：

@Retention(RetentionPolicy.RUNTIME)

**public** @interface Idempotent {

*// 标记注解*

}

并用于标记对应的service操作实现。对于仅重试幂等操作，切面只需改进切点表达式以仅匹配@Idempotent操作：

@Around("com.xyz.myapp.SystemArchitecture.businessService() && @annotation(com.xyz.myapp.service.Idempotent)")

**public** Object doConcurrentOperation(ProceedingJoinPoint pjp) **throws** Throwable {

...

}

* + 1. 基于schema的AOP支持

若你倾向于使用基于XML格式的配置，Spring同样提供支持用于定义切面的“aop”命名空间标签。与@Aspect风格使用的切点表达式和增强都是一样的，因此本节我们只专注于新的语法，并对比讨论前一节（[@AspectJ support](https://docs.spring.io/spring/docs/5.0.16.RELEASE/spring-framework-reference/core.html#aop-ataspectj)），以便更好地理解切点表达式编写和增强参数绑定。

需使用本节所述的aop命名空间标签，你需要导入spring-aop schema（如[基于XML Schema的配置](https://docs.spring.io/spring/docs/5.0.16.RELEASE/spring-framework-reference/core.html#xsd-schemas)所述）。如何导入aop 命名空间的标签详见[the AOP schema](https://docs.spring.io/spring/docs/5.0.16.RELEASE/spring-framework-reference/core.html#xsd-schemas-aop)。

在你的Spring配置内，所有的切面和增强元素必须置于<aop:config>元素内（在一个应用上下文内你可以有多个<aop:config>元素）。一个<aop:config>元素可以包含切点，增强点和切面元素（注意这些元素必须依次声明）。

<aop:config>风格的配置重度使用Spring的[自动代理](https://docs.spring.io/spring/docs/5.0.16.RELEASE/spring-framework-reference/core.html#aop-autoproxy)机制。这可能导致某些问题（如增强不被正确织入）若你已通过使用BeanNameAutoProxyCreator 或其余类似来声明显式自动代理。推荐用法则是仅使用<aop:config>风格模式，或AutoProxyCreator 风格模式。

* + - 1. 声明切面

使用schema支持，切面就是在Spring应用上下文内定义为bean的一个常规Java对象。状态和行为由该对象内的成员变量来确定，切点和增强信息则有XML内容来确定。

使用<aop:aspect>元素声明的切面，通过ref 属性引用其所的依靠bean：

<aop:config>

<aop:aspect id="myAspect" ref="aBean">

...

</aop:aspect>

</aop:config>

<bean id="aBean" class="...">

...

</bean>

切面依靠的bean（此处是"aBean"）当然可以像其他Spring bean一样配置和依赖注入。

* + - 1. 声明切点

命名切点可以在<aop:config>元素内声明，允许切点定义在多个切面跟增强体内共用。

一个代表服务层任意业务服务的执行的切点可定义如下：

<aop:config>

<aop:pointcut id="businessService" expression="execution(\* com.xyz.myapp.service.\*.\*(..))"/>

</aop:config>

注意切点表达式本身使用AspectJ切点表达式语言，与[@AspectJ support](https://docs.spring.io/spring/docs/5.0.16.RELEASE/spring-framework-reference/core.html#aop-ataspectj)中描述保持一致。若你在使用基于schema的声明风格，你可以在切点表达式内引用定义在类（@Aspects）内的已命名切点。上述切点的另一种定义方式为：

<aop:config>

<aop:pointcut id="businessService" expression="com.xyz.myapp.SystemArchitecture.businessService()"/>

</aop:config>

假设此时已有一个SystemArchitecture 切面用于描述[共享通用切点定义](https://docs.spring.io/spring/docs/5.0.16.RELEASE/spring-framework-reference/core.html#aop-common-pointcuts)。

在切面内声明切点跟声明顶层切点一样：

<aop:config>

<aop:aspect id="myAspect" ref="aBean">

<aop:pointcut id="businessService"

expression="execution(\* com.xyz.myapp.service.\*.\*(..))"/>

...

</aop:aspect>

</aop:config>

与@AspectJ切面一样，使用基于schema定义风格的切点可以收集连接点上下文。例如，如下切点收集了“this”对象作为连接点上下文并传递给增强：

<aop:config>

<aop:aspect id="myAspect" ref="aBean">

<aop:pointcut id="businessService"

expression="execution(\* com.xyz.myapp.service.\*.\*(..)) &amp;&amp; this(service)"/>

<aop:before pointcut-ref="businessService" method="monitor"/>

...

</aop:aspect>

</aop:config>

增强则必须声明为，包含对应名称的参数，以接收连接点上下文：

**public** **void** monitor(Object service) {

...

}

当组合切点子表达式时，在XML文件内无法正确识别&&，因此关键字and，or和not 可分别用于替代&&，||和!。例如，前面的切点如此编写更好：

<aop:config>

<aop:aspect id="myAspect" ref="aBean">

<aop:pointcut id="businessService"

expression="execution(\* com.xyz.myapp.service.**.**(..)) **and** this(service)"/>

<aop:before pointcut-ref="businessService" method="monitor"/>

...

</aop:aspect>

</aop:config>

注意如此定义的切点可以由其XML id来引用但不能将已命名切点用于组成组合切点。因此基于schema定义风格的已命名切点的支持相较于@AspectJ风格有更多的限制。

* + - 1. 声明增强

在@AspectJ风格内的五种增强在此同样支持，并具备相同的语义。

前增强

前增强在匹配的方法执行前执行。使用<aop:before>元素在<aop:aspect>内声明。

<aop:aspect id="beforeExample" ref="aBean">

<aop:before

pointcut-ref="dataAccessOperation"

method="doAccessCheck"/>

...

</aop:aspect>

在此dataAccessOperation 是在上层（<aop:config>）定义的切点id。若需内联定义切点，将pointcut-ref属性替换为pointcut 属性：

<aop:aspect id="beforeExample" ref="aBean">

<aop:before

pointcut="execution(\* com.xyz.myapp.dao.\*.\*(..))"

method="doAccessCheck"/>

...

</aop:aspect>

就如我们在@AspectJ风格内所说的，使用命名切点可以有效地提高代码可读性。

方法属性标识了提供增强体的方法（doAccessCheck）。此方法必须定义在由包含该增强的该切面元素引用的bean内。在数据访问操作执行前（由切点表达式匹配到的方法执行连接点），在切面bean内的“doAccessCheck”方法会被调用。

返回后增强

返回后增强运行于在方法正常执行完毕时。与前增强一样，其声明在<aop:aspect>内。例如：

<aop:aspect id="afterReturningExample" ref="aBean">

<aop:after-returning

pointcut-ref="dataAccessOperation"

method="doAccessCheck"/>

...

</aop:aspect>

跟@AspectJ风格一样，也可以在增强体内获取其返回值。使用返回属性来指定需传递的返回值名称：

<aop:aspect id="afterReturningExample" ref="aBean">

<aop:after-returning

pointcut-ref="dataAccessOperation"

returning="retVal"

method="doAccessCheck"/>

...

</aop:aspect>

doAccessCheck方法必须声明入参命名为retVal。该参数类型约束与@AfterReturning中一样。例如，方法签名可以如此声明：

**public** **void** doAccessCheck(Object retVal) {...

抛出后增强

抛出后增强在匹配方法因抛出异常而退出时执行。使用after-throwing元素在<aop:aspect>内声明：

<aop:aspect id="afterThrowingExample" ref="aBean">

<aop:after-throwing

pointcut-ref="dataAccessOperation"

method="doRecoveryActions"/>

...

</aop:aspect>

如@AspectJ风格一样，可以在增强体内获取抛出的异常。使用throwing属性来指定需要传递的异常参数名称：

<aop:aspect id="afterThrowingExample" ref="aBean">

<aop:after-throwing

pointcut-ref="dataAccessOperation"

throwing="dataAccessEx"

method="doRecoveryActions"/>

...

</aop:aspect>

doRecoveryActions方法必须声明入参参数名为dataAccessEx。该参数类型约束与@AfterThrowing中一样。例如，方法签名声明为：

**public** **void** doRecoveryActions(DataAccessException dataAccessEx) {...

后（finally）增强

后增强（finally）在不管方法如何退出时都执行。使用after 元素声明如下：

<aop:aspect id="afterFinallyExample" ref="aBean">

<aop:after

pointcut-ref="dataAccessOperation"

method="doReleaseLock"/>

...

</aop:aspect>

环绕增强

最后是环绕增强。环绕增强包围匹配方法执行。可以达到前增强和后增强的相同效果，并可以决定方法执行的时间、方式甚至可以决定方法是否可以执行。环绕增强经常使用在你需要以线程安全的方式在方法执行前后共享状态（如开启停止一个计时器）。注意按需使用增强，在前增强可以满足需求的情况下不要使用环绕增强。

使用aop:around元素声明环绕增强。增强方法的首参数类型必须为ProceedingJoinPoint。在增强体内，调用ProceedingJoinPoint 的proceed()执行到下一个方法。proceed 方法也可以传参Object[]—数组内的值将作为在执行下一个方法时，方法的入参。有关调用proceed的传参Object[]的注意事项详见[Around advice](https://docs.spring.io/spring/docs/5.0.16.RELEASE/spring-framework-reference/core.html#aop-ataspectj-around-advice)。

<aop:aspect id="aroundExample" ref="aBean">

<aop:around

pointcut-ref="businessService"

method="doBasicProfiling"/>

...

</aop:aspect>

doBasicProfiling 增强的实现则与@AspectJ示例一模一样（当然，去掉注解）：

**public** Object doBasicProfiling(ProceedingJoinPoint pjp) **throws** Throwable {

*// start stopwatch*

Object retVal = pjp.proceed();

*// stop stopwatch*

**return** retVal;

}

增强参数

基于schema风格的声明支持全限定名类型的增强，如@AspectJ一样，通过增强方法的参数名与匹配切点参数名的互相匹配。详见[Advice parameters](https://docs.spring.io/spring/docs/5.0.16.RELEASE/spring-framework-reference/core.html#aop-ataspectj-advice-params)。若你希望为增强方法显式指定参数名（而不依赖前述的检测机制）则可以通过advice元素的arg-names属性，与[Determining argument names](https://docs.spring.io/spring/docs/5.0.16.RELEASE/spring-framework-reference/core.html#aop-ataspectj-advice-params-names)小节中所说的advice注解内的“argNames”属性保持一致。例如：

<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列表。

如下便是基于XSD方式，使用强类型参数的环绕增强示例。

**package** x.y.service;

**public** **interface** **FooService** {

Foo getFoo(String fooName, **int** age);

}

**public** **class** **DefaultFooService** **implements** FooService {

**public** Foo getFoo(String name, **int** age) {

**return** **new** Foo(name, age);

}

}

下一步是切面。注意profile(..)接收多个强类型的的参数，第一个参数刚好是用于执行下一个方法的连接点：该参数的出现表示profile(..)是around 增强：

**package** x.y;

**import** org.aspectj.lang.ProceedingJoinPoint;

**import** org.springframework.util.StopWatch;

**public** **class** **SimpleProfiler** {

**public** Object profile(ProceedingJoinPoint call, String name, **int** age) **throws** Throwable {

StopWatch clock = **new** StopWatch("Profiling for '" + name + "' and '" + age + "'");

**try** {

clock.start(call.toShortString());

**return** call.proceed();

} **finally** {

clock.stop();

System.out.println(clock.prettyPrint());

}

}

}

最后，以下则是对于特定的连接点生效的完整增强配置：

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

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:aop="http://www.springframework.org/schema/aop"

xsi:schemaLocation="

http://www.springframework.org/schema/beans https://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/aop https://www.springframework.org/schema/aop/spring-aop.xsd">

*<!-- 这是将被Spring's AOP框架代理的对象 -->*

<bean id="fooService" class="x.y.service.DefaultFooService"/>

*<!-- 增强本身 -->*

<bean id="profiler" class="x.y.SimpleProfiler"/>

<aop:config>

<aop:aspect ref="profiler">

<aop:pointcut id="theExecutionOfSomeFooServiceMethod"

expression="execution(\* x.y.service.FooService.getFoo(String,int))

and args(name, age)"/>

<aop:around pointcut-ref="theExecutionOfSomeFooServiceMethod"

method="profile"/>

</aop:aspect>

</aop:config>

</beans>

如下是驱动脚本，我们将在标准输出内获得如下输出内容：

**import** org.springframework.beans.factory.BeanFactory;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**import** x.y.service.FooService;

**public** **final** **class** **Boot** {

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

BeanFactory ctx = **new** ClassPathXmlApplicationContext("x/y/plain.xml");

FooService foo = (FooService) ctx.getBean("fooService");

foo.getFoo("Pengo", 12);

}

}

StopWatch 'Profiling for 'Pengo' and '12'': running time (millis) = 0

-----------------------------------------

ms % Task name

-----------------------------------------

00000 ? execution(getFoo)

增强顺序

当多个增强需要在同一个连接点（执行方法）执行，则顺序遵循规则与[Advice ordering](https://docs.spring.io/spring/docs/5.0.16.RELEASE/spring-framework-reference/core.html#aop-ataspectj-advice-ordering)保持一致。切面间的顺序取决于切面依赖的bean上的Order 注解或其实现的Ordered 接口。

* + - 1. 引入

引入（在AspectJ内被称为内部类声明）允许声明被增强对象去实现一个既定的接口，并代替这些对象提供该接口的实现。

引入可以通过在aop:aspect内使用aop:declare-parents元素。这个元素用于声明匹配的类型有新的父类。例如，给定接口UsageTracked，及其实现DefaultUsageTracked，如下切面声明了所有service接口的实现同样实现了UsageTracked 接口。（如为了通过JMX暴露数据）

<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>

usageTracking bean的class内应包含方法：

**public** **void** recordUsage(UsageTracked usageTracked) {

usageTracked.incrementUseCount();

}

需要实现的接口取决于implement-interface属性。types-matching属性值则是一个AspectJ类型范式：任意匹配类型的bean都会实现UsageTracked 接口。注意在前面的示例中的前增强，service bean可以直接用作于UsageTracked 接口的实现。若编程式访问该bean可以如此写：

UsageTracked usageTracked = (UsageTracked) context.getBean("myService");

* + - 1. 切面实例化模型

基于schema定义的切面支持的实例化模型仅有单例模型。其他实例化模型或将在未来版本中添加。

* + - 1. 增强者

“advisor”概念来自于Spring AOP，与AspectJ中的“advisor”并不直接等效。一个增强者像是只有一个增强的独立小切面。增强本身由bean表示，且必须实现在[Spring的增强类型](https://docs.spring.io/spring/docs/5.0.17.RELEASE/spring-framework-reference/core.html#aop-api-advice-types)所述的增强接口之一。增强者则可以使用AspectJ的切点表达式。

Spring通过<aop:advisor>元素支持增强者概念。一般你会看到其用于事务增强，这些也在Spring内由其命名空间支持。增强者示例如下：

<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>

跟上例使用的pointcut-ref属性一样，你可以使用pointcut 属性来定义一个切点表达式。

若需控制增强者的顺序，以便增强可依序执行，使用order 属性来定义增强者的Ordered 值。

* + - 1. 示例

下面我们来看看在5.2.7的示例中的并发锁失败重试如何使用schema风格重写。

业务服务有时可能因某些并发问题执行失败（例如死锁失败）。假如该操作被重试，极其有可能在下一次成功。对于这种适合重试的情形（幂等操作，不需要返回给用户做冲突处理），我们倾向于做一个透明地重试而不是给客户端返回PessimisticLockingFailureException。该需求显然跨过了多个服务层的服务，因此是使用切面实现的理想模型。

因为我们希望重试操作，则需要使用环绕增强以便可以多次执行到下一层。如下是基本切面实现（常规Java类并使用schema支持）：

**public** **class** **ConcurrentOperationExecutor** **implements** Ordered {

**private** **static** **final** **int** DEFAULT\_MAX\_RETRIES = 2;

**private** **int** maxRetries = DEFAULT\_MAX\_RETRIES;

**private** **int** order = 1;

**public** **void** setMaxRetries(**int** maxRetries) {

this.maxRetries = maxRetries;

}

**public** **int** getOrder() {

**return** this.order;

}

**public** **void** setOrder(**int** order) {

this.order = order;

}

**public** Object doConcurrentOperation(ProceedingJoinPoint pjp) **throws** Throwable {

**int** numAttempts = 0;

PessimisticLockingFailureException lockFailureException;

**do** {

numAttempts++;

**try** {

**return** pjp.proceed();

}

**catch**(PessimisticLockingFailureException ex) {

lockFailureException = ex;

}

} **while**(numAttempts <= this.maxRetries);

**throw** lockFailureException;

}

}

注意切面实现了Ordered 接口因此我们可以设置该切面的优先级高于事务增强（我们希望每次重试都是一个全新事务）。maxRetries 和order属性通过Spring来配置。主要动作在doConcurrentOperation 环绕增强方法内执行。我们尝试执行，并在因并发问题产生的异常失败时捕获PessimisticLockingFailureException 异常并重试，除非达到最大重试次数。

此类与@AspectJ示例中的类一样，只不过去掉了注解。

相关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>

注意我们假设所有的业务都是幂等的。如果不是这样，那么我们可以通过引入Idempotent 注解，来更改切面以只重试幂等操作：

@Retention(RetentionPolicy.RUNTIME)

**public** @interface Idempotent {

*// marker annotation*

}

使用该注解标记幂等的业务操作。切面的更改只需要更改切点表达式以匹配@Idempotent操作：

<aop:pointcut id="idempotentOperation"

expression="execution(\* com.xyz.myapp.service.\*.\*(..)) and

@annotation(com.xyz.myapp.service.Idempotent)"/>

* + 1. 如何选择AOP声明风格

当决定使用切面来实现需求时，是选择Spring AOP还是AspectJ，是选择AspectJ语言编码风格，@AspetJ注解风格，或者Spring XML风格？这些决定受许多因素影响，包括应用需求，开发工具和团队对AOP的熟悉程度。

* + - 1. Spring AOP或全AspectJ

使用最简单可用的。Spring AOP相较全AspectJ要简单，因为不需要在开发和编译时引入额外的AspectJ编译器/编织器。如果你仅需要增强Spring bean内的执行操作，则Spring AOP是正确的选择。如果你需要增强不由Spring容器管理的对象（一般如域对象），则你需要使用AspectJ。但你希望增强除了方法执行的其他连接点时（例如，成员变量的设置或获取连接点等等）。

在使用AspectJ时，你可以选择AspectJ语言（或者说编码风格）或@AspectJ注解风格。显然若非使用Java5以上，则需要使用AspectJ语言。若切面在你的设计中很大比重，你可以使用Eclipse插件[AspectJ Development Tools (AJDT)](https://www.eclipse.org/ajdt/)，则AspectJ语言则是优选：由于该语言专用于编写切面，因此清晰简洁。如果你没有在使用Eclipse，或只有一部分切面，并没有在应用中占很大比重，则你可以考虑使用@AspectJ风格，并在你的IDE内保持常规Java编译，并在你的构建脚本内添加切面织入。

* + - 1. @AspectJ还是Spring AOP XML

若你决定使用Spring AOP，则你需要选择使用@AspectJ还是XML风格。其中有多种优劣需要衡量。

现有的Spring用户对XML会比较熟悉，且其实际上是真正的POJO。当使用AOP作为工具去配置企业级应用时XML会是个不错的选择（也就是你将切点表达式视为配置的一部分，可以独立更改）。可以说使用XML可以很清晰地知道在你的系统配置中有哪些切面。

XML有两个不足。首先其不能完全将实现封装在一个地方。DRY原则是一个系统内任何设计都应有其单一、无歧义、权威准确的表示。当使用XML风格时，一个需求的实现被分割成bean类的实现和XML文件配置。当使用@AspectJ风格时则只有一个模块—切面，封装了所有需要的信息。另外，XML风格相较@AspectJ风格表达能力有限：仅支持“单例”切面实力化模型，且不可以拼装声明在XML的命名切点。例如，@AspectJ风格你可以这么写：

@Pointcut(execution(\* get\*()))

**public** **void** propertyAccess() {}

@Pointcut(execution(org.xyz.Account+ \*(..))

**public** **void** operationReturningAnAccount() {}

@Pointcut(propertyAccess() && operationReturningAnAccount())

**public** **void** accountPropertyAccess() {}

在XML风格我可以声明前两个切点：

<aop:pointcut id="propertyAccess" expression="execution(\* get\*())"/>

<aop:pointcut id="operationReturningAnAccount"

expression="execution(org.xyz.Account+ \*(..))"/>

XML方式的缺点就是你不能通过组合两个定义来定义accountPropertyAccess 切点。

@AspectJ风格支持额外的实例化模型，以及更丰富的切点组合。有把切面作为模组单元的优势。同样其还可被Spring AOP或AspectJ理解（并处理）——也就是说如果你需要使用AspectJ的其他功能来实现更多的需求，那么就更容易通过基于AspectJ的方式去合并更改。权衡之下Spring团队更倾向于@AspectJ风格只要你的切面在企业应用内不仅仅是做配置类用途时。

* + 1. 混合切面类型

通过使用自动代理支持可以混合@AspectJ风格、基于schema定义的<aop:aspect>切面，以及<aop:advisor>定义的增强者，甚至使用Spring1.2风格定义的代理和拦截器在同一个配置内。所有的这些使用相同潜在类型的实现即可轻易共存。

* + 1. 代理机制

Spring AOP使用JDK动态代理或CGLIB来为目标对象创建代理（只要有选择，推荐JDK动态代理）。

如果被代理的目标对象实现了至少一个接口那么JDK动态代理就会被使用。目标类型所实现的所有接口都会被代理。若目标对象没有实现任何一个接口那么CGLIB代理会被创建。

若你想强制使用CGLIB代理（例如，代理定义在目标对象内的所有方法，而不仅仅是接口的实现方法）也可以。但此处有几个问题需要考虑：

* final 方法不可被增强，因为它们不可被覆盖。
* 自Spring 3.2始，没有必要在项目类路径内添加CGLIB，因为CGLIB已经被重新打包在org.springframework下并直接包含在spring-core JAR包内。这也就是说基于CGLIB的代理也跟JDK动态代理一样都被支持。
* 自Spring 4.0始，代理对象的构造函数不会被调用两次因为CGLIB代理实例有Objenesis创建。只有你的JVM不允许跳过构造函数时，你将会通过Spring AOP支持看到两次调用及其相关debug日志信息。

可通过设置<aop:config>元素的proxy-target-class属性为true来强制开启CGLIB代理：

<aop:config proxy-target-class="true">

*<!-- 其他bean定义在此... -->*

</aop:config>

使用@AspectJ自动代理支持时，设置<aop:aspectj-autoproxy>元素的'proxy-target-class'属性为true来强制使用CGLIB代理：

<aop:aspectj-autoproxy proxy-target-class="true"/>

多个<aop:config/>片段会在运行时由autp-proxy创建器合并成一个，应用多个<aop:config/>片段（一般来自不同的XML bean定义文件）中最强力的代理设置。这些同样适用于<tx:annotation-driven/>和<aop:aspectj-autoproxy/>元素。

需要说明：在<tx:annotation-driven/>，<aop:aspectj-autoproxy/>或<aop:config/>元素上使用proxy-target-class="true"将使得三个元素都强制使用CGLIB代理。

* + - 1. 理解AOP代理

Spring AOP是基于代理的。在你编写自己的切面或者使用由Spring框架支持的基于Spring AOP的切面前，明白你写下的每一行语句的语义是极其重要的。

假设你有一个普通的、未被代理的、没有任何特别的、直接对象引用，如下代码片段所示。

**public** **class** **SimplePojo** **implements** Pojo {

**public** **void** foo() {

*// 此方法直接调用“this”引用的方法*

this.bar();

}

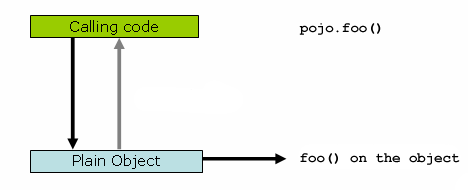
**public** **void** bar() {

*// some logic...*

}

}

若你在对象引用上调用方法，则方法是在该对象引用上直接被调用，如下图所示：



**public** **class** **Main** {

**public** **static** **void** main(String**[]** args) {

Pojo pojo = **new** SimplePojo();

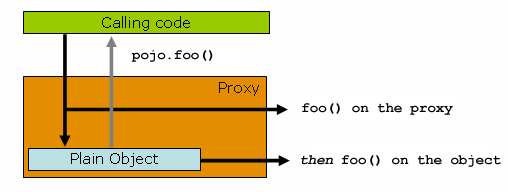
*// 在'pojo' 引用上直接调用foo方法*

pojo.foo();

}

}

在客户端代码的引用是一个代理时，就会有一点不同。如下图和代码段所示。



**public** **class** **Main** {

**public** **static** **void** main(String**[]** args) {

ProxyFactory factory = **new** ProxyFactory(**new** SimplePojo());

factory.addInterface(Pojo.class);

factory.addAdvice(**new** RetryAdvice());

Pojo pojo = (Pojo) factory.getProxy();

*// 在代理上调用方法!*

pojo.foo();

}

}

理解的关键点是Main 类的main(..)方法内部有对代理的引用。也就是说调用对象引用的方法将会调用该代理，且该代理将可以委托给与该方法相关的拦截器（增强）。然而，一旦调用达到目标对象，在本例中的引用，任何其内部方法的互相调用都会在this引用上调用，而不是代理。这是个重要特性，也就意味着自调用不会导致被调用方法相关的增强被执行。

好吧，那这该怎么办呢？最好的办法就是（最好就是松散耦合）重构代码使得自调用不会发生。当然，这将带来更多的工作，但这也是最好的，最无侵略性的方法。另外，还有一种可怕的方式，因此我在此处讳莫如深地提出。你可以（咳咳咳）将你的逻辑与Spring AOP绑定，如下：

**public** **class** **SimplePojo** **implements** Pojo {

**public** **void** foo() {

*// 这样可以执行, 但... 呃~!*

((Pojo) AopContext.currentProxy()).bar();

}

**public** **void** bar() {

*// some logic...*

}

}

这样你的代码就会完全与Spring AOP耦合，使得类本身耦合进AOP上下文中。这样还需要在代理创建时新增额外配置：

**public** **class** **Main** {

**public** **static** **void** main(String**[]** args) {

ProxyFactory factory = **new** ProxyFactory(**new** SimplePojo());

factory.adddInterface(Pojo.class);

factory.addAdvice(**new** RetryAdvice());

factory.setExposeProxy(true);

Pojo pojo = (Pojo) factory.getProxy();

*// this is a method call on the proxy!*

pojo.foo();

}

}

最后需要注意，AspectJ不会有自调用问题，因其不是基于代理的AOP框架。

* + 1. 编程式创建@AspectJ代理

## 资源