Core Technologies

Version 5.0.3.RELEASE

Translator：杨德军

参考文档的这一部分涵盖了Spring Framework绝对不可或缺的所有技术。

其中最重要的是Spring框架的控制反转（IoC）容器。 Spring框架的IoC容器的全面处理紧随其后，全面涵盖了Spring的面向切面编程（AOP）技术。 Spring框架拥有自己的AOP框架，这个框架在概念上很容易理解，并且成功地解决了Java企业编程中80％的AOP需求。

还提供了Spring与AspectJ的集成（目前在功能方面最为丰富 - 当然也是Java企业领域最成熟的AOP实现）。

1. The IoC container

1.1. Spring IoC container 和 beans介绍

本章涵盖了控制反转（IoC）[1]原理的Spring框架实现。 IoC也被称为依赖注入（DI）。 这是一个过程，对象通过构造函数参数，工厂方法的参数或在工厂方法构造或返回后在对象实例上设置的属性来定义它们的依赖关系，即它们所处理的其他对象。 然后容器在创建bean时注入这些依赖关系。 这个过程从根本上来说是相反的，因此名为控制反转（IoC），bean本身通过使用类的直接构造或诸如Service Locator模式的机制来控制其依赖性的实例化或位置。

org.springframework.beans和org.springframework.context包是Spring Framework的IoC容器的基础。 [BeanFactory](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/beans/factory/BeanFactory.html) 接口提供了一种高级配置机制，能够管理任何object.[ApplicationContext](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/context/ApplicationContext.html) 是BeanFactory的一个子接口。 它增加了与Spring的AOP功能更容易的集成; 消息资源处理（用于国际化），事件发布; 以及Web应用程序上下文（WebApplicationContext）等应用程序层特定的上下文，以用于Web应用程序。

总之，BeanFactory提供了配置框架和基本功能，而ApplicationContext增加了更多的企业特定功能。 ApplicationContext是BeanFactory的一个完整的超集，在本章中专门用于描述Spring的IoC容器。 有关使用BeanFactory而不是ApplicationContext的更多信息，请参阅 [The BeanFactory](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-beanfactory)。

在Spring中，构成应用程序的骨干并由Spring IoC容器管理的对象称为bean。 bean是一个实例化，组装并由Spring IoC容器管理的对象。 否则，bean只是应用程序中众多对象中的一个。 Bean和它们之间的依赖关系反映在容器使用的配置元数据中。

1.2. Container 概述

接口org.springframework.context.ApplicationContext代表Spring IoC容器，负责实例化，配置和组装上述bean。 容器通过读取配置元数据获取对象的实例化，配置和组装。 配置元数据以XML，Java注解或Java代码。 它允许你表达组成你的应用程序的对象以及这些对象之间丰富的相互依赖关系。

Spring提供了几个ApplicationContext接口的实现。 在独立应用程序中，通常会创建[ClassPathXmlApplicationContext](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/context/support/ClassPathXmlApplicationContext.html) or [FileSystemXmlApplicationContext](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/context/support/FileSystemXmlApplicationContext.html)的实例。 虽然XML是用于定义配置元数据的传统格式，但您可以通过提供少量的XML配置来指示容器使用Java注解或代码作为元数据格式，以声明方式支持这些额外的元数据格式。

在大多数应用场景中，显式用户代码不需要实例化Spring IoC容器的一个或多个实例。 例如，在Web应用程序场景中，应用程序的web.xml文件中的简单八行（或多行）样板Web描述符XML通常就足够了（请参阅[Convenient ApplicationContext instantiation for web applications](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#context-create)）。 如果您使用[Spring Tool Suite](https://spring.io/tools/sts) Eclipse-powered的开发环境，则只需点击几下鼠标或击键即可轻松创建此样板配置。

下图是Spring如何工作的高级视图。 您的应用程序类与配置元数据相结合，以便在创建并初始化ApplicationContext之后，您将拥有一个完全配置且可执行的系统或应用程序。



*Figure 1. The Spring IoC container*

1.2.1. Configuration metadata

如上图所示，Spring IoC容器使用一种形式的配置元数据; 这个配置元数据表示作为应用程序开发人员如何告诉Spring容器在应用程序中实例化，配置和组装对象。

传统上，配置元数据是以简单而直观的XML格式提供的，这是本章的大部分内容用来传达Spring IoC容器的关键概念和功能。

|  |  |
| --- | --- |
|  | 基于XML的元数据不是唯一允许的配置元数据形式。 Spring IoC容器本身与配置元数据实际写入的格式完全分离。 现在许多开发人员为他们的Spring应用程序选择 [Java-based configuration](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-java)。 |

有关在Spring容器中使用其他形式的元数据的信息，请参阅：

* [Annotation-based configuration](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-annotation-config): Spring 2.5 引入了对基于注解的配置元数据的支持
* [Java-based configuration](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-java): 从Spring 3.0开始,Spring JavaConfig项目提供的许多功能成为核心Spring框架的一部分。因此，可以使用Java而不是XML文件来定义应用程序外部的Bean。要使用这些新功能，请参阅 @Configuration, @Bean, @Import and @DependsOn注解

Spring配置由容器必须管理的至少一个，通常是多个bean定义组成。 基于XML的配置元数据将这些bean配置为顶级<beans />元素内的<bean />元素。 Java配置通常在@Configuration类中使用@Bean注释的方法。

这些bean定义对应于组成应用程序的实际对象。 通常，您可以定义服务层对象，数据访问对象（DAO），Struts Action实例等表示对象，Hibernate SessionFactory，JMS队列等基础结构对象。 通常，不会在容器中配置细粒度的域对象，因为创建和加载域对象通常是DAO和业务逻辑的责任。 但是，您可以使用Spring与AspectJ的集成来配置在IoC容器控制之外创建的对象。 请参阅 [Using AspectJ to dependency-inject domain objects with Spring](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-atconfigurable)。

以下示例显示了基于XML的配置元数据的基本结构：

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

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

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

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

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

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

*<!-- collaborators and configuration for this bean go here -->*

</bean>

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

*<!-- collaborators and configuration for this bean go here -->*

</bean>

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

</beans>

id属性是用来标识单个bean定义的字符串。 class属性定义了bean的类型并使用完全限定的类名。 id属性的值是指协作对象。 本示例中未显示用于引用协作对象的XML; 请参阅 [Dependencies](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-dependencies)以获取更多信息。

1.2.2. 实例化一个 container

实例化一个Spring IoC容器很简单。 提供给ApplicationContextconstructor的位置路径实际上是资源字符串，它允许容器从各种外部资源（例如本地文件系统，Java CLASSPATH等等）加载配置元数据。

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

|  |  |
| --- | --- |
|  | 在了解了Spring的IoC容器之后，您可能想了解更多关于Spring的资源抽象的知识，如参考资料中所述，它提供了一个从URI语法中定义的位置读取InputStream的方便机制。 具体来说，资源路径用于构建应用程序上下文，如[Application contexts and Resource paths](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#resources-app-ctx)中所述。 |

以下示例显示服务层对象（services.xml）配置文件：

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

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

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

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

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

*<!-- services -->*

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

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

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

*<!-- additional collaborators and configuration for this bean go here -->*

</bean>

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

</beans>

以下示例显示数据访问对象daos.xml文件：

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

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

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

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

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

<bean id="accountDao"

class="org.springframework.samples.jpetstore.dao.jpa.JpaAccountDao">

*<!-- additional collaborators and configuration for this bean go here -->*

</bean>

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

*<!-- additional collaborators and configuration for this bean go here -->*

</bean>

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

</beans>

在前面的例子中，服务层由PetStoreServiceImpl类和JpaAccountDao和JpaItemDao类型的两个数据访问对象（基于JPA Object/Relational映射标准）组成。 属性名称元素是指JavaBean属性的名称，ref元素是指另一个bean定义的名称。 id和ref元素之间的这种联系表示协作对象之间的依赖关系。 有关配置对象依赖关系的详细信息，请参阅[Dependencies](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-dependencies)。

构建 XML-based configuration metadata

让bean定义跨越多个XML文件是有用的。 通常，每个单独的XML配置文件都代表了架构中的逻辑层或模块。

您可以使用应用程序上下文构造函数从所有这些XML片段中加载bean定义。 这个构造函数有多个资源位置，如前一节所示。 或者，使用一个或多个<import />元素从另一个或多个文件加载bean定义。 例如：

<beans>

<import resource="services.xml"/>

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

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

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

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

</beans>

在前面的示例中，外部bean定义从三个文件中加载：services.xml，messageSource.xml和themeSource.xml。 所有的位置路径都是相对于导入的定义文件而言的，所以services.xml必须和导入的文件位于相同的目录或类路径位置，而messageSource.xml和themeSource.xml必须位于位置之下的资源位置 的导入文件。 正如你所看到的，一个前导的斜线(leading slash)被忽略，但是鉴于这些路径是相对的，最好不要使用斜线。 正在导入的文件的内容（包括顶级<beans />元素）必须是根据Spring架构的有效XML bean定义。

|  |  |
| --- | --- |
| | | 有可能，但不建议使用相对的“../”路径引用父目录中的文件。 这样做会创建对当前应用程序之外的文件的依赖关系。 特别是，不建议将此引用用于“classpath：”URL（例如“classpath：../ services.xml”），其中运行时解析过程选择“最近”的类路径根，然后查看其父目录。 类路径配置更改可能会导致选择不同的，不正确的目录。  您始终可以使用完全限定的资源位置而不是相对路径：例如"file:C:/config/services.xml"或 "classpath:/config/services.xml"。 但是，请注意，您将应用程序的配置耦合到特定的绝对位置。 通常最好保持这种绝对位置的间接性，例如通过在运行时针对JVM系统属性解析的"${…​}"占位符。 |

import指令是bean名称空间本身提供的一个功能。 除了普通的bean定义之外的其他配置特征可以在由Spring提供的XML命名空间的选择中获得，例如， "context" and the "util" namespace。

Groovy Bean 定义 DSL

作为外部化配置元数据的另一个例子，bean定义也可以在Spring的Groovy Bean Definition DSL中表示，如从Grails框架中已知的。 通常情况下，这样的配置将存在于“.groovy”文件中，结构如下：

beans {

dataSource(BasicDataSource) {

driverClassName = "org.hsqldb.jdbcDriver"

url = "jdbc:hsqldb:mem:grailsDB"

username = "sa"

password = ""

settings = [mynew:"setting"]

}

sessionFactory(SessionFactory) {

dataSource = dataSource

}

myService(MyService) {

nestedBean = { AnotherBean bean ->

dataSource = dataSource

}

}

}

这种配置风格在很大程度上等价于XML bean定义，甚至支持Spring的XML配置名称空间。 它还允许通过“importBeans”指令导入XML bean定义文件。

1.2.3. 使用 container

ApplicationContext是高级工厂的接口，能够维护不同bean及其依赖项的注册表。 使用方法T getBean(String name, Class<T> requiredType)，可以检索bean的实例。

ApplicationContext使您可以读取bean定义并按如下方式访问它们：

*// create and configure beans*

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

*// retrieve configured instance*

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

*// use configured instance*

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

使用Groovy配置，bootstrapping看起来非常相似，只是一个不同的上下文实现类，它可以感知Groovy（但也理解XML bean定义）：

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

最灵活的变体是GenericApplicationContext与reader delegates组合， 使用XML文件的XmlBeanDefinitionReader：

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

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

context.refresh();

或者使用Groovy文件的GroovyBeanDefinitionReader：

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

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

context.refresh();

如果需要，这样的阅读器代理可以在同一个ApplicationContext上混合和匹配，从不同的配置源读取bean定义。

然后可以使用getBean来检索bean的实例。 ApplicationContext接口还有其他一些检索bean的方法，但理想情况下应用程序代码不应该使用它们。 事实上，你的应用程序代码根本不应该调用getBean()方法，因此完全不依赖于Spring API。 例如，Spring与Web框架的集成为各种Web框架组件（如controllers and JSF-managed beans）提供了依赖注入，允许您通过元数据（例如autowiring annotation）声明对特定Bean的依赖关系。

1.3. Bean 概述

Spring IoC容器管理一个或多个bean。 这些bean是使用您提供给容器的配置元数据创建的，例如以XML <bean />定义的形式。

在容器本身中，这些bean定义表示为BeanDefinition对象，其中包含以下元数据（以及其他信息）：

* *package-qualified class name:通常是被定义的bean的实际实现类*.
* Bean 行为配置元素，它说明bean在容器中的行为(scope, lifecycle callbacks, and so forth).
* 引用bean所需的其他bean以完成其工作;这些引用也被称为*collaborators* or *dependencies*.
* 在新创建的对象中设置的其他配置，例如用于管理连接池的Bean的连接数量或池的大小限制.

这个元数据转化为一组构成每个bean定义的属性。

| *Table 1. The bean definition* | |
| --- | --- |
| **Property** | **Explained in…​** |
| class | [Instantiating beans](#_1.3.2._Instantiating_beans) |
| name | [Naming beans](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-beanname) |
| scope | [Bean scopes](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-scopes) |
| constructor arguments | [Dependency Injection](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-collaborators) |
| properties | [Dependency Injection](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-collaborators) |
| autowiring mode | [Autowiring collaborators](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-autowire) |
| lazy-initialization mode | [Lazy-initialized beans](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-lazy-init) |
| initialization method | [Initialization callbacks](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-lifecycle-initializingbean) |
| destruction method | [Destruction callbacks](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-lifecycle-disposablebean) |

除了包含有关如何创建特定bean的信息的bean定义之外，ApplicationContextimplementations还允许用户注册在容器外部创建的现有对象。 这是通过getBeanFactory()方法访问ApplicationContext的BeanFactory来完成的，该方法返回BeanFactory实现的DefaultListableBeanFactory。 DefaultListableBeanFactory通过方法registerSingleton（..）和 registerBeanDefinition（..）来支持这种注册。 但是，典型的应用程序只能通过元数据bean定义来定义bean。

|  |  |
| --- | --- |
|  | Bean元数据和手动提供的单例实例需要尽可能早地注册，以便容器在自动装配和其他内省步骤中正确推理它们。 虽然重写现有的元数据和现有的单例实例在某种程度上受到支持，但在运行时注册新的Bean（与实时访问工厂同时）并未得到正式支持，并且可能导致并发访问异常和/或bean容器中的状态不一致。 |

1.3.1. Naming beans

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

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

您不需要为bean提供名称或标识。 如果没有显式提供名称或标识，则容器为该bean生成一个唯一的名称。 但是，如果要通过名称引用该bean，则通过使用ref元素或 [Service Locator](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-servicelocator)查找，必须提供一个名称。 不提供名称的动机与使用 [inner beans](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-inner-beans) and [autowiring collaborators](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-autowire)有关。

Bean Naming Conventions(约定)

约定是在命名bean时使用标准Java约定作为实例字段名称。 也就是说，bean的名字以一个小写字母开头，并且从那开始就是骆驼式的。 这样的名字的例子是（不带引号）“accountManager”，“accountService”，“userDao”，“loginController”等等。

命名bean始终使您的配置更易于阅读和理解，如果您使用的是Spring AOP，则将建议应用于与名称相关的一组bean时会有很大的帮助。

|  |  |
| --- | --- |
|  | 通过类路径中的组件扫描，Spring为未命名的组件生成bean名称，遵循以上规则：本质上，采用简单的类名称并将其初始字符转换为小写。 但是，在不常见的特殊情况下，如果有多个字符，并且第一个字符和第二个字符都是大写字母，则会保留原始的外壳。 这些和java.beans.Introspector.decapitalize（Spring在这里使用的）定义的规则相同。 |

Aliasing a bean outside the bean definition

在bean定义本身中，可以通过使用由id属性指定的最多一个名称和name属性中的任意数量的其他名称的组合来为bean提供多个名称。 这些名称可以等同于同一个bean的别名，并且对于某些情况很有用，例如允许应用程序中的每个组件通过使用特定于该组件本身的bean名称引用公共依赖项。

但是，指定bean实际定义的所有别名并不总是足够的。 有时候需要为其他地方定义的bean引入一个别名。 在大型系统中，在每个子系统之间分配配置的情况通常是这种情况，每个子系统都有自己的一组对象定义。 在基于XML的配置元数据中，您可以使用<alias />元素来完成此操作。

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

在这种情况下，同名容器中名为fromName的bean也可以在使用这个别名定义之后被称为toName。

例如，子系统A的配置元数据可以通过名称subsystemA-dataSource来引用数据源。 子系统B的配置元数据可以通过名称subsystemB-dataSource引用数据源。 在编写使用这两个子系统的主应用程序时，主应用程序通过名称myApp-dataSource引用数据源。 要使所有三个名称都与您在MyApp配置元数据中添加的相同对象相关联，请使用以下别名定义：

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

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

现在，每个组件和主应用程序都可以通过唯一的名称引用dataSource，并保证不与其他任何定义冲突（有效地创建名称空间），但它们引用同一个bean。

Java-configuration

如果您正在使用Java配置，则可以使用@Bean注释来提供别名，请参阅 [Using the @Bean annotation](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-java-bean-annotation)以获取详细信息。

### 1.3.2. 实例化 beans

一个bean定义本质上是一个创建一个或多个对象的配方。 当容器被询问时，该容器查看命名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）实际的对象。

如果使用基于XML的配置元数据，则可以指定要在<bean />元素的classattribute中实例化的对象的类型（或类）。 这个类属性在内部是一个BeanDefinition实例的Class属性，通常是强制的。 （有关例外情况，请参阅 [Instantiation using an instance factory method](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-class-instance-factory-method) and [Bean definition inheritance](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-child-bean-definitions)）您可以通过以下两种方式之一使用Class属性：

* 通常，在容器本身通过反射调用其构造函数直接创建bean的情况下，指定要构建的bean类，这与使用new关键字的Java代码有些相同。
* 指定包含将被调用来创建多想的静态工厂方法的实际类，在容器调用类的静态工厂方法以创建该bean的情况较少。从调用静态工厂方法返回的对象类型可以是完全相同的类或另一个类。

*Inner class names*

如果要为静态嵌套类配置一个bean定义，则必须使用嵌套类的二进制名称。

例如，如果在com.example包中有一个名为Foo的类，并且此Foo类有一个名为Bar的静态嵌套类，那么bean定义上的“class”属性的值将是...

com.example.Foo$Bar

注意在名称中使用$字符来将嵌套类名与外部类名分开。

用 constructor 实例化

当你通过构造函数的方法创建一个bean时，所有的普通类都可以被Spring使用和兼容。 也就是说，正在开发的类不需要实现任何特定的接口或以特定的方式编码。 只需指定bean类就足够了。 但是，根据您用于特定bean的IoC类型，您可能需要一个默认（空）构造函数。

Spring IoC容器几乎可以管理任何你想要管理的类， 它不限于管理真正的JavaBeans。 大多数Spring用户更喜欢实际的JavaBeans，它只有一个默认的（无参数）构造函数，以及在容器中的属性之后建模的合适的setter和getter。 你也可以在你的容器中有更多异国情调的非bean风格的类。 例如，如果您需要使用绝对不符合JavaBean规范的传统连接池，Spring也可以管理它。

使用基于XML的配置元数据，您可以按如下方式指定您的bean类：

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

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

有关为构造函数提供参数（如果需要）和在构造对象后设置对象实例属性的机制的详细信息，请参阅 [Injecting Dependencies](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-collaborators)。

使用 static factory method 实例化

在定义使用静态工厂方法创建的bean时，可以使用class属性来指定包含静态工厂方法的类和名为factory-method的属性，以指定工厂方法本身的名称。

下面的bean定义指定了通过调用工厂方法创建的bean。 该定义没有指定返回对象的类型（类），而只指定了包含工厂方法的类。 在这个例子中， createInstance()方法必须是一个静态方法。

<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;

}

}

有关从工厂返回对象之后向工厂方法提供（可选）参数和设置对象实例属性的机制的详细信息，请参阅 [Dependencies and configuration in detail](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-properties-detailed)。

使用 instance factory method 实例化

与通过 [static factory method](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-class-static-factory-method)实例化类似，使用实例工厂方法的实例化从容器调用现有bean的非静态方法以创建新的bean。 要使用此机制，请将类属性保留为空，并在factory-bean属性中指定当前（或parent/ancestor）容器中包含要调用以创建对象的实例方法的bean的名称。 使用factory-method属性设置工厂方法本身的名称。

*<!-- 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;

}

}

一个工厂类也可以拥有多个工厂方法，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;

}

}

这种方法表明，工厂bean本身可以通过依赖注入（DI）进行管理和配置。 详见 [Dependencies and configuration in detail](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-properties-detailed)。

|  |  |
| --- | --- |
|  | 在Spring文档中，工厂bean指的是在Spring容器中配置的bean，它将通过 [instance](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-class-instance-factory-method) or [static](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-class-static-factory-method) 工厂方法创建对象。 相比之下，FactoryBean（注意大写字母）是指特定于Spring的 [FactoryBean](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-extension-factorybean)。 |

1.4. Dependencies

一个典型的企业应用程序不是由一个单一的对象组成（或Spring的说法中的bean）。 即使是最简单的应用程序也只有几个对象一起工作来呈现最终用户将其视为一个连贯的应用程序。 下一节将介绍如何从定义许多独立的bean定义到完全实现的应用程序，在这些应用程序中对象协作实现目标。

1.4.1. Dependency Injection(依赖注入)

依赖注入（DI）是一个过程，通过这个过程，对象可以通过构造函数参数，工厂方法的参数或者在构造或返回对象实例后设置的属性来定义它们的依赖关系 从工厂方法。 然后容器在创建bean时注入这些依赖关系。 这个过程从根本上说是相反的，因此名为控制反转（IoC），它本身通过使用类的直接构造或服务定位符模式来控制它自己的依赖关系的实例化或位置。

代码与DI原理相比更加清晰，并且在对象提供依赖关系时解耦更有效。 该对象不查找其依赖项，并且不知道依赖项的位置或类。 因此，您的类变得更容易测试，特别是当依赖关系在接口或抽象基类上时，这允许在单元测试中使用存根(stub)或模拟实现。

DI存在两种主要的变体，[Constructor-based dependency injection](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-constructor-injection) and [Setter-based dependency injection](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-setter-injection)。

Constructor-based dependency injection(基于构造函数的依赖注入)

基于构造器的DI通过容器调用具有多个参数的构造器来完成，每个参数表示一个依赖关系。 调用一个具有特定参数的静态工厂方法来构造这个bean几乎是等价的，而且这个讨论同样将参数作为构造函数和静态工厂方法来处理。 以下示例显示了只能通过构造函数注入进行依赖注入的类。 请注意，这个类没有什么特别之处，它是一个POJO，它不依赖容器特定的接口，基类或注解。

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

*// the SimpleMovieLister has a dependency on a MovieFinder*

**private** MovieFinder movieFinder;

*// a constructor so that the Spring container can inject a MovieFinder*

**public** SimpleMovieLister(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

*// business logic that actually uses the injected MovieFinder is omitted...*

}

Constructor argument resolution(构造函数参数解析)

构造函数参数解析匹配使用参数的类型进行匹配。 如果bean定义的构造函数参数中没有可能存在的歧义，那么在bean定义中定义构造函数参数的顺序就是在实例化bean时将这些参数提供给相应构造函数的顺序。 考虑以下类：

**package** x.y;

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

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

*// ...*

}

}

假设Bar和Baz类没有继承关系，就不存在潜在的歧义。 因此，以下配置可以正常工作，并且不需要在 <constructor-arg/>元素中显式指定构造函数参数索引和/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>

当引用另一个bean时，类型是已知的，并且可以发生匹配（就像前面的例子那样）。 当使用一个简单类型时，如<value> true </ value>，Spring不能确定值的类型，所以如果没有帮助，就无法按类型匹配。 考虑以下类：

**package** examples;

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

*// Number of years to calculate the Ultimate Answer*

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

*// The Answer to Life, the Universe, and Everything*

**private** String ultimateAnswer;

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

this.years = years;

this.ultimateAnswer = ultimateAnswer;

}

}

*构造函数参数类型匹配*

*在前面的场景中，如果使用type属性显式指定构造函数参数的类型，则容器可以使用简单类型的类型匹配。 例如：*

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

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

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

</bean>

*构造函数参数索引*

*使用index属性来明确指定构造函数参数的索引。 例如：*

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

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

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

</bean>

除了解决多个简单值的歧义之外，指定索引还解决了构造函数具有相同类型的两个参数的不明确性。 请注意，该索引是基于0的。

*构造函数的参数名称*

*您还可以使用构造函数参数名称来消除值：*

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

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

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

</bean>

请记住，为了使这项工作脱离您的代码，必须使用调试标志进行编译，以便Spring可以从构造函数中查找参数名称。 如果你不能用调试标志编译你的代码（或者不想），你可以使用[@ConstructorProperties](http://download.oracle.com/javase/6/docs/api/java/beans/ConstructorProperties.html)  JDK注解来显式地指定你的构造函数参数。 示例类将不得不如下所示：

**package** examples;

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

*// Fields omitted*

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

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

this.years = years;

this.ultimateAnswer = ultimateAnswer;

}

}

Setter-based dependency injection(基于Setter的依赖注入)

*在调用无参数构造函数或无参数静态工厂方法来实例化您的bean之后，基于Setter的DI通过调用bean上的容器调用setter方法来完成。*

以下示例显示了一个只能使用纯setter注入进行依赖注入的类。 这个类是传统的Java。 这是一个POJO，它不依赖于容器特定的接口，基类或注解。

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

*// the SimpleMovieLister has a dependency on the MovieFinder*

**private** MovieFinder movieFinder;

*// a setter method so that the Spring container can inject a MovieFinder*

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

this.movieFinder = movieFinder;

}

*// business logic that actually uses the injected MovieFinder is omitted...*

}

ApplicationContext 支持它所管理的bean的基于构造函数和基于setter的DI。 它也支持基于setter的DI之后，通过构造方法已经注入了一些依赖关系。 您可以以BeanDefinition的形式配置依赖项，您可以使用它与PropertyEditor实例一起使用，将属性从一种格式转换为另一种格式。 然而，大多数Spring用户并不直接使用这些类（即以编程方式），而是使用XML bean定义，带注解的组件（即，使用@Component，@ Controller等注释的类）或基于Java的@Bean方法 @Configuration类。 然后将这些源内部转换为BeanDefinition的实例，并用于加载整个Spring IoC容器实例。

Constructor-based or setter-based DI?

既然可以混合使用基于构造函数和基于setter的DI，那么使用强制依赖和构造方法的构造函数或可选依赖的配置方法是一个很好的经验法则。 请注意，可以使用setter方法上的 [@Required](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-required-annotation) annotation来使属性成为必需的依赖项。

Spring团队通常主张构造器注入，因为它使得可以将应用程序组件实现为不可变对象，并确保所需的依赖项不为空。 而且，构造器注入的组件总是返回到处于完全初始化状态的客户端（调用）代码。 作为一个侧面说明，大量的构造函数参数是一种糟糕的代码异味，这意味着类可能有太多的责任，应该重构，以更好地解决问题的分离。

Setter注入主要只应用于可选的依赖关系，可以在类中指定合理的默认值。 否则，在代码使用依赖关系的任何地方都必须执行非空的检查。 setter注入的一个好处是setter方法使得这个类的对象可以重新配置或稍后重新注入。 通过 [JMX MBeans](https://docs.spring.io/spring/docs/current/spring-framework-reference/integration.html#jmx)进行管理因此是一个引人注目的使用情况。

使用最适合特定类的DI风格。 有时，在处理没有源代码的第三方类的时候，选择适合你。 例如，如果第三方类不公开任何setter方法，则构造函数注入可能是DI唯一可用的形式。

Dependency resolution process(依赖性解决过程)

容器执行bean的依赖关系解析如下：

* ApplicationContext 是使用描述所有Bean的配置元数据创建和初始化的。配置元数据可以通过XML，Java代码或注解来指定。
* 对于每个bean，如果您正在使用它而不是普通构造函数，则它的依赖项将以属性，构造函数参数或静态工厂方法的参数的形式代表。当bean被实际创建时，这些依赖项被提供给bean。
* 每个属性或构造函数参数都是要设置的值的实际定义，或者是对容器中另一个bean的引用。
* 作为值的每个属性或构造函数参数将从其指定的格式转换为该属性或构造函数的实际类型。默认情况下，Spring可以将以字符串格式提供的值转换为所有内置类型，例如：int,long,String,boolean等。

Spring容器在容器创建时验证每个bean的配置。 但是，在实际创建bean之前，bean属性本身不会被设置。 Beans是singleton-scoped的，并且被设置为预先实例化的（默认的）是在创建容器时创建的。 [Bean scopes](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-scopes)范围中定义。 否则，只有在请求时才创建bean。 创建一个bean可能会导致创建一个bean的图形(graph)，因为bean的依赖关系及其依赖关系的依赖性（等等）被创建和分配。 请注意，这些依赖项之间的解决方案不匹配可能会出现较晚，即首次创建受影响的bean。

Circular(循环) dependencies

如果您主要使用构造函数注入，则可能会创建一个无法解析的循环依赖方案。

例如：类A需要通过构造函数注入的类B的实例，而类B需要通过构造函数注入的类A的实例。 如果将类A和B的Bean配置为相互注入，则Spring IoC容器将在运行时检测到此循环引用，并引发BeanCurrentlyInCreationException。

一个可能的解决方案是编辑一些类的源代码，由setter而不是构造函数来配置。 或者，避免构造函数注入，只使用setter注入。 换句话说，虽然不推荐，但是可以使用setter注入配置循环依赖。

与典型情况（没有循环依赖）不同，bean A和bean B之间的循环依赖关系迫使其中一个bean在被完全初始化之前被注入到另一个bean中（一个经典的鸡/鸡蛋场景）。

一般来说，你可以相信Spring做正确的事情。它在容器加载时检测配置问题，比如引用不存在的bean和循环依赖关系。当bean实际创建时，Spring会尽可能晚地设置属性并解决依赖关系。这意味着，如果创建该对象或其某个依赖关系时遇到问题，则在请求对象时，正确加载的Spring容器可能会稍后生成异常。例如，由于缺少或无效的属性，bean抛出异常。某些配置问题的可能延迟的可见性是为什么ApplicationContext实现默认预先实例化单例bean。在实际需要这些Bean之前，需要花费一些前期的时间和内存来创建这些Bean，但是在创建ApplicationContext时发现配置问题，而不是在以后。你仍然可以重写这个默认行为，这样singleton bean将会延迟初始化，而不是预先实例化。

如果不存在循环依赖关系，那么当一个或多个协作bean被注入到一个依赖bean中时，每个协作bean在被注入到依赖bean之前被完全配置。 这意味着如果bean A对bean B有依赖关系，Spring IoC容器在调用bean A上的setter方法之前完全配置bean B.换句话说，bean被实例化（如果不是预先实例化的单例），它的 依赖关系被设置，并调用相关的生命周期方法（例如 [configured init method](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-lifecycle-initializingbean) or the [InitializingBean callback method](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-lifecycle-initializingbean)）。

Examples of dependency injection(依赖注入的例子)

以下示例使用XML-based配置for setter-based DI。 Spring XML配置文件的一小部分指定了一些bean定义：

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

*<!-- setter injection using the nested ref element -->*

<property name="beanOne">

<ref bean="anotherExampleBean"/>

</property>

*<!-- setter injection using the neater ref attribute -->*

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

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

</bean>

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

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

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

**private** AnotherBean beanOne;

**private** YetAnotherBean beanTwo;

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

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

this.beanOne = beanOne;

}

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

this.beanTwo = beanTwo;

}

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

this.i = i;

}

}

在前面的例子中，setters被声明为匹配在XML文件中指定的属性。 以下示例使用基于构造函数的DI：

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

*<!-- constructor injection using the nested ref element -->*

<constructor-arg>

<ref bean="anotherExampleBean"/>

</constructor-arg>

*<!-- constructor injection using the neater ref attribute -->*

<constructor-arg ref="yetAnotherBean"/>

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

</bean>

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

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

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

**private** AnotherBean beanOne;

**private** YetAnotherBean beanTwo;

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

**public** ExampleBean(

AnotherBean anotherBean, YetAnotherBean yetAnotherBean, **int** i) {

this.beanOne = anotherBean;

this.beanTwo = yetAnotherBean;

this.i = i;

}

}

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

现在考虑一下这个例子的一个变种，在这里不是使用构造函数，而是告诉Spring调用静态工厂方法来返回对象的一个实例：

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

<constructor-arg ref="anotherExampleBean"/>

<constructor-arg ref="yetAnotherBean"/>

<constructor-arg value="1"/>

</bean>

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

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

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

*// a private constructor*

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

...

}

*// a static factory method; the arguments to this method can be*

*// considered the dependencies of the bean that is returned,*

*// regardless of how those arguments are actually used.*

**public** **static** ExampleBean createInstance (

AnotherBean anotherBean, YetAnotherBean yetAnotherBean, **int** i) {

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

*// some other operations...*

**return** eb;

}

}

静态工厂方法的参数通过<constructor-arg />元素提供，就像构造函数实际使用一样。 工厂方法返回的类的类型不必与包含静态工厂方法的类的类型相同，尽管在本例中它是。 一个实例（非静态）工厂方法将以基本相同的方式使用（除了使用factory-bean属性而不是class属性），所以在这里不讨论细节。

1.4.2. Dependencies and configuration in detail

如前一节所述，您可以将bean属性和构造函数参数定义为对其他受管Bean（协作者）的引用，或者将其定义为内联定义的值。 Spring的基于XML的配置元数据为此支持其<property />和<constructor-arg />元素中的子元素类型。

Straight values (primitives, Strings, and so on)

<property />元素的value属性将属性或构造函数参数指定为可读的字符串表示形式。 Spring的 [conversion service](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#core-convert-ConversionService-API)用于将这些值从String转换为属性或参数的实际类型。

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

*<!-- results in a setDriverClassName(String) call -->*

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

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

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

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

</bean>

以下示例使用 [p-namespace](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-p-namespace)进行更简洁的XML配置。

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

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

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

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

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

<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"

destroy-method="close"

p:driverClassName="com.mysql.jdbc.Driver"

p:url="jdbc:mysql://localhost:3306/mydb"

p:username="root"

p:password="masterkaoli"/>

</beans>

前面的XML更简洁; 然而，错误是在运行时而不是设计时发现的，除非您在创建bean定义时使用支持自动属性完成的IDE(如[IntelliJ IDEA](http://www.jetbrains.com/idea/) or the [Spring Tool Suite](https://spring.io/tools/sts) (STS) )。 强烈建议这种IDE帮助。

您也可以将java.util.Properties实例配置为：

<bean id="mappings"

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

*<!-- typed as a java.util.Properties -->*

<property name="properties">

<value>

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

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

</value>

</property>

</bean>

Spring容器通过使用JavaBeans PropertyEditor机制将<value />元素中的文本转换为java.util.Properties实例。 这是一个很好的捷径，它是Spring团队倾向于使用嵌套的<value />元素超过value属性样式的几个地方之一。

The idref element

idref元素只是一个防错的方式，将容器中另一个bean的id（字符串值 - 不是引用）传递给<constructor-arg />或<property />元素。

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

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

<property name="targetName">

<idref bean="theTargetBean"/>

</property>

</bean>

上面的bean定义片段与下面的片段完全等价（在运行时）：

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

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

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

</bean>

第一种形式比第二种形式要好，因为使用idref标签允许容器在部署时验证被引用的命名bean实际存在。 在第二个变体中，不会对传递给客户机bean的targetName属性的值执行验证。 当客户端bean实际被实例化时，才会发现错误（最有可能是致命的结果）。 如果客户端bean是 [prototype](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-scopes) bean，则此类型错误和生成的异常可能仅在部署容器后很长时间才能发现。

|  |  |
| --- | --- |
|  | 4.0 bean xsd中不再支持idref元素的本地属性，因为它不再提供超过常规bean引用的值。 升级到4.0架构时，只需将现有的idref本地引用更改为idref bean。 |

在<idref />元素带来价值的地方（至少在Spring 2.0之前的版本中），在ProxyFactoryBean bean定义中的 [AOP interceptors](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-pfb-1)的配置中。 指定拦截器名称时使用<idref />元素可以防止拼写错误。

References to other beans (collaborators)

ref元素是<constructor-arg />或<property />定义元素中的最后一个元素。 在这里，您将bean的指定属性的值设置为对容器管理的另一个bean（协作者）的引用。 被引用的bean是其属性将被设置的bean的依赖项，并且在属性设置之前根据需要初始化它。 （如果协作者是singleton bean，它可能已经被容器初始化了。）所有引用最终都是对另一个对象的引用。 Scoping 和 validation取决于您是通过bean，local或parent属性指定另一个对象的id / name。

通过<ref />标签的bean属性指定目标bean是最通用的形式，并允许创建对同一个容器或父容器中的任何bean的引用，而不管它是否在同一个XML文件中。 bean属性的值可以与目标bean的id属性相同，或者作为目标bean的name属性中的一个值。

<ref bean="someBean"/>

通过父属性指定目标bean将创建对当前容器的父容器中的bean的引用。 父属性的值可以与目标bean的id属性相同，也可以与目标bean的name属性中的一个值相同，并且目标bean必须位于当前bean的父容器中。 您主要在具有容器层次结构时使用此bean参考变体，并且想要使用与父bean名称相同的代理将父容器中的现有bean包装在父容器中。

*<!-- in the parent context -->*

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

*<!-- insert dependencies as required as here -->*

</bean>

*<!-- in the child (descendant) context -->*

<bean id="accountService" <!-- bean name is the same as the parent bean -->

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

<property name="target">

<ref parent="accountService"/> *<!-- notice how we refer to the parent bean -->*

</property>

*<!-- insert other configuration and dependencies as required here -->*

</bean>

|  |  |
| --- | --- |
|  | 4.0 bean xsd中不再支持ref元素的local属性，因为它不再提供超过普通bean引用的值。 在升级到4.0模式时，只需将现有的ref本地引用更改为ref bean。 |

Inner beans

<property />或<constructor-arg />元素中的<bean />元素定义了一个所谓的内部bean。

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

*<!-- instead of using a reference to a target bean, simply define the target bean inline -->*

<property name="target">

<bean class="com.example.Person"> *<!-- this is the inner bean -->*

<property name="name" value="Fiona Apple"/>

<property name="age" value="25"/>

</bean>

</property>

</bean>

内部bean定义不需要定义的id或名称; 如果指定，容器不使用这样的值作为标识符。 容器在创建时也会忽略范围标志：内部bean始终是匿名的，并且它们始终使用外部bean创建。 不可能将内部bean注入到除了封装bean之外的协作bean中，或者独立地访问它们。

作为一个例子，可以从自定义范围接收销毁回调，例如， 对于包含在单例bean中的请求范围的内部bean：创建内部bean实例将被绑定到它的包含bean，但是销毁回调允许它参与请求范围的生命周期。 这不是一个常见的情况; 内部bean通常简单地分享它们包含的bean的范围。

Collections

在 <list/>, <set/>, <map/>, and <props/> 元素中, 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">

*<!-- results in a setAdminEmails(java.util.Properties) call -->*

<property name="adminEmails">

<props>

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

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

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

</props>

</property>

*<!-- results in a setSomeList(java.util.List) call -->*

<property name="someList">

<list>

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

<ref bean="myDataSource" />

</list>

</property>

*<!-- results in a setSomeMap(java.util.Map) call -->*

<property name="someMap">

<map>

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

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

</map>

</property>

*<!-- results in a setSomeSet(java.util.Set) call -->*

<property name="someSet">

<set>

<value>just some string</value>

<ref bean="myDataSource" />

</set>

</property>

</bean>

*映射键或值或设置值的值也可以是以下任何元素：*

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

Collection merging

Spring容器也支持集合的合并。 应用程序开发人员可以定义父级样式的<list />，<map />，<set />或<props />元素，并具有子样式的<list />，<map />，<set />或 <props />元素继承并覆盖父集合中的值。 也就是说，子集合的值是合并父集合和子集合元素的结果，子集合元素覆盖父集合中指定的值。

*这部分关于合并讨论了父子bean机制。 不熟悉父代和子代bean定义的读者可能希望在继续之前阅读*[*relevant section*](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-child-bean-definitions)*。*

以下示例演示了集合合并：

<beans>

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

<property name="adminEmails">

<props>

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

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

</props>

</property>

</bean>

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

<property name="adminEmails">

*<!-- the merge is specified on the child collection definition -->*

<props merge="true">

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

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

</props>

</property>

</bean>

<beans>

注意在子bean定义的adminEmails属性的<props />元素上使用merge = true属性。 当子bean由容器解析并实例化时，生成的实例具有一个adminEmails Propertiescollection，其中包含合并子级的adminEmails集合与父级的adminEmails集合的结果。

administrator=administrator@example.com

sales=sales@example.com

support=support@example.co.uk

子集“Properties ”集合的值集继承父级<props />中的所有属性元素，并且子级的支持值值覆盖父级集合中的值。

这种合并行为同样适用于<list />，<map />和<set />集合类型。 在<list />元素的特定情况下，与List集合类型相关联的语义（即有序值集合的概念）被维护; 父项的值在所有子项列表的值之前。 对于Map，Set和Properties集合类型，不存在排序。 因此，对于容器在内部使用的关联的Map，Set和Properties实现类型的集合类型，没有任何排序语义生效。

Limitations of collection merging

您不能合并不同的集合类型（如Map和List），如果您尝试这样做，则会引发适当的Exception。 合并属性必须在较低的继承的子定义上指定; 在父集合定义上指定合并属性是多余的，并且不会导致所需的合并。

Strongly-typed collection

通过在Java 5中引入泛型类型，您可以使用强类型集合。 也就是说，可以声明一个Collection类型，使其只能包含String元素（例如）。 如果您使用Spring将强类型集合依赖注入到bean中，则可以利用Spring的类型转换支持，以便强类型Collection实例的元素在添加之前转换为适当的类型 集合。

**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属性准备注入时，强类型Map <String，Float>的元素类型的泛型信息可以通过反射获得。 因此，Spring的类型转换基础结构将不同的值元素识别为类型为Float，并将字符串值9.99,2.75和3.99转换为实际的Floattype。

Null and empty string values

Spring把空的参数当作空字符串来处理。 以下基于XML的配置元数据片段将email属性设置为空字符串值（“”）。

<bean class="ExampleBean">

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

</bean>

前面的示例等同于以下Java代码：

exampleBean.setEmail("");

<null />元素处理空值。 例如：

<bean class="ExampleBean">

<property name="email">

<null/>

</property>

</bean>

以上配置相当于以下Java代码：

exampleBean.setEmail(null);

XML shortcut with the p-namespace

p-namespace使您可以使用bean元素的属性来代替嵌套的<property />元素来描述属性值和/或合作bean。

Spring支持带有命名空间( [with namespaces](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#xsd-schemas))的可扩展配置格式，这些命名空间基于XML模式定义。 本章中讨论的beans配置格式是在XML Schema文档中定义的。 但是，p-namespace没有在XSD文件中定义，只存在于Spring的核心中。

以下示例显示了解析为相同结果的两个XML片段：第一个使用标准XML格式，第二个使用p-namespace。

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

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

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

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

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

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

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

</bean>

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

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

</beans>

该示例在bean定义中显示了名为email的p-namespace中的一个属性。 这告诉Spring包含一个属性声明。 如前所述，p-namespace没有模式定义，因此您可以将该属性的名称设置为属性名称。

这个下一个例子包含了两个bean定义，它们都引用了另一个bean：

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

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

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

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

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

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

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

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

</bean>

<bean name="john-modern"

class="com.example.Person"

p:name="John Doe"

p:spouse-ref="jane"/>

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

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

</bean>

</beans>

正如你所看到的，这个例子不仅包含使用p-命名空间的属性值，而且还使用特殊的格式来声明属性引用。 第一个bean定义使用<property name =“spouse”ref =“jane”/>来创建bean bean的引用，第二个bean定义使用p：spouse-ref =“jane”作为属性 完全一样的东西。 在这种情况下，配偶是属性名称，而-ref部分则表明这不是一个正值，而是对另一个bean的引用。

|  |  |
| --- | --- |
|  | p-namespace不像标准的XML格式那么灵活。 例如，声明属性引用的格式与以Ref结尾的属性冲突，而标准的XML格式则不会。 我们建议您谨慎选择您的方法，并将其传达给您的团队成员，以避免生成同时使用这三种方法的XML文档。 |

XML shortcut with the c-namespace

与带有 [XML shortcut with the p-namespace](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-p-namespace)类似，在Spring 3.1中新引入的*c-namespace*允许使用内联属性来配置构造函数参数，而不是嵌套constructor-arg元素。

让我们来看一下基于构造函数的依赖注入的例子：c：namespace：

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

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

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

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

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

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

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

*<!-- traditional declaration -->*

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

<constructor-arg ref="bar"/>

<constructor-arg ref="baz"/>

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

</bean>

*<!-- c-namespace declaration -->*

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

</beans>

c: namespace使用与p：1（用于bean引用的尾部-ref）相同的约定来通过名称来设置构造函数参数。 而且，即使它没有在XSD架构中定义（但它存在于Spring内核中），也需要声明它。

对于构造函数参数名称不可用的罕见情况（通常如果字节码是在没有调试信息的情况下编译的话），可以使用回退参数索引：

*<!-- c-namespace index declaration -->*

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

|  |  |
| --- | --- |
|  | 由于XML语法，索引表示法要求存在前导\_，因为XML属性名称不能以数字开头（即使某些IDE允许）。 |

实际上，构造函数解析机制( [mechanism](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-ctor-arguments-resolution))在匹配参数方面非常有效，除非真正需要，否则我们建议在整个配置中使用名称符号。

Compound property names

当您设置bean属性时，只要最终属性名称以外的路径的所有组件都不为null，就可以使用复合或嵌套的属性名称。 考虑下面的bean定义。

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

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

</bean>

foo bean有一个fred属性，它有一个bob属性，它有一个sammy属性，最终sammy属性被设置为123.为了这个工作，foo的fred属性和bob属性 fred一定不能是bean构造后的空，否则抛出NullPointerException。

1.4.3. Using depends-on(使用依赖)

如果一个bean是另一个bean的依赖，通常意味着一个bean被设置为另一个的属性。 通常，您可以使用基于XML的配置元数据中的 [<ref/> element](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-ref-element)来完成此操作。 但是，有时豆类之间的依赖关系并不那么直接; 例如，类中的静态初始化器需要被触发，比如数据库驱动程序注册。 依赖的属性可以明确地强制一个或多个bean在使用这个元素初始化之前被初始化。 以下示例使用depends-on属性来表示对单个bean的依赖关系：

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

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

要表示对多个bean的依赖关系，请提供一个bean名称列表作为depends-on属性的值，用逗号，空格和分号作为有效的分隔符：

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

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

</bean>

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

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

|  |  |
| --- | --- |
|  | bean定义中的depends-on属性可以指定一个初始化时间依赖项，并且在 [singleton](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-scopes-singleton) beans的情况下，可以指定相应的销毁时间依赖项。 定义与给定bean的独立关系的依赖bean在给定bean本身被销毁之前首先被销毁。 因此，依赖也可以控制关闭顺序。 |

1.4.4. Lazy-initialized beans(懒加载beans)

默认情况下，ApplicationContext实现作为初始化过程的一部分，急切地创建和配置所有的 [singleton](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-scopes-singleton) beans。 通常，这种预先实例化是可取的，因为配置或周围环境中的错误被立即发现，而不是数小时甚至数天后。 当这种行为不可取时，可以通过将bean定义标记为lazy-initialized来防止单例bean的预先实例化。 一个惰性初始化的bean告诉IoC容器在第一次请求时创建一个bean实例，而不是在启动时。

在XML中，这个行为是由<bean />元素的lazy-init属性控制的; 例如：

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

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

当一个ApplicationContext使用前面的配置时，名为lazy的bean在ApplicationContext启动时并不急于预先实例化，而not.lazy bean则急切地预先实例化。

然而，当一个懒惰初始化的bean是一个未经过延迟初始化的单例bean的依赖时，ApplicationContext在启动时会创建懒惰初始化的Bean，因为它必须满足单例的依赖关系。 懒惰初始化的bean被注入一个单独的bean中，并且没有被初始化。

您还可以通过在<beans />元素上使用default-lazy-init属性来控制容器级别的懒惰初始化; 例如：

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

*<!-- no beans will be pre-instantiated... -->*

</beans>

1.4.5. Autowiring collaborators(自动装配配合者)

Spring容器可以自动连接合作bean之间的关系。 您可以允许Spring通过检查ApplicationContext的内容来自动为您的bean解析协作者（其他bean）。 自动装配具有以下优点：

* 自动装配可以显着减少指定属性或构造函数参数的需要。(其他几只，例如本章其他地方 [discussed elsewhere in this chapter](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-child-bean-definitions) ，在这方面也很有价值)
* 自动装配可以随着对象的发展更新配置。例如，如果你需要向类中添加依赖项，则可以自动满足该依赖项，而无需修改配置。因此，在开发过程中，自动装配可能特别有用，而且在代码库变得更加稳定的情况下，不会否定切换到显示布线的选项。

当使用基于XML的配置元数据[2]时，您可以使用<bean />元素的autowireattribute指定一个bean定义的autowire模式。 自动装配功能有四种模式。 您可以指定每个bean的自动装配，因此可以选择哪些自动装配。

| *Table 2. Autowiring modes* | |
| --- | --- |
| **Mode** | **Explanation** |
| no | (Default) No autowiring. Bean references must be defined via a ref element. Changing the default setting is not recommended for larger deployments, because specifying collaborators explicitly gives greater control and clarity. To some extent, it documents the structure of a system. |
| byName | Autowiring by property name. Spring looks for a bean with the same name as the property that needs to be autowired. For example, if a bean definition is set to autowire by name, and it contains a *master* property (that is, it has a*setMaster(..)* method), Spring looks for a bean definition named master, and uses it to set the property. |
| byType | Allows a property to be autowired if exactly one bean of the property type exists in the container. If more than one exists, a fatal exception is thrown, which indicates that you may not use *byType* autowiring for that bean. If there are no matching beans, nothing happens; the property is not set. |
| constructor | Analogous to *byType*, but applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised. |

使用byType或构造函数自动装配模式，您可以连线数组和类型集合。 在这种情况下，容器中所有符合预期类型的自动装配候选都被提供来满足依赖关系。 如果预期的键类型是字符串，则可以自动装载强类型的映射。 自动装配的Maps值将由所有与预期类型匹配的bean实例组成，Maps键将包含相应的bean名称。

您可以将自动装配行为与自动装配完成后执行的依赖关系检查相结合。

Limitations and disadvantages of autowiring

自动装配在项目中一致使用时效果最佳。 如果通常不使用自动装配，开发人员可能会使用它来连接一个或两个bean定义。

考虑自动装配的局限性和缺点：

* 属性和构造函数参数中的显式依赖关系总是覆盖自动装配。 您不能自动调用所谓的简单属性，例如primitives, Strings, and Classes（以及这种简单属性的数组）。 这个限制是通过设计。
* 自动装配不如准确布线(explicit wiring)。 虽然，如上表所述，Spring在小心避免猜测可能会有意想不到的结果的情况下，Spring管理的对象之间的关系不再被明确记录。
* Wiring information可能无法用于可能从Spring容器生成文档的工具。
* 容器中的多个bean定义可以匹配由setter方法或构造函数参数指定的类型，以便自动装配。 对于arrays, collections, or Maps，这不一定是个问题。 然而对于期望单一值的依赖关系，这种模糊不是任意解决的。 如果没有唯一的bean定义可用，则抛出异常。

在后一种情况下，你有几个选择：

* 放弃autowiring in favor，以支持explicit wiring。
* 通过将autowire-candidate属性设置为false，避免为bean定义进行自动装配，如下一节所述。
* 通过将其<bean />元素的主属性设置为true，将单个bean定义指定为主要候选者。
* 如 [Annotation-based container configuration](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-annotation-config)中所述，通过基于注解的配置实现更加精细的控制。

Excluding a bean from autowiring(自动装配中排除一个bean)

在每个bean的基础上，您可以从自动装配中排除一个bean。 在Spring的XML格式中，将<bean />元素的autowire-candidate属性设置为false; 该容器使该特定的bean定义对自动装配基础结构不可用（包括诸如 [@Autowired](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-autowired-annotation)之类的注释样式配置）。

|  |  |
| --- | --- |
|  | autowire-candidate属性设计为仅影响基于类型的自动装配。 它不会影响名称的显式引用，即使指定的bean没有标记为自动连线候选，它也会得到解决。 因此，如果名称匹配，通过名称的自动装配将注入一个bean。 |

您还可以根据对bean名称进行模式匹配来限制自动导向候选项。 顶层的<beans />元素在其default-autowire-candidates属性中接受一个或多个模式。 例如，要将autowire候选者状态限制为名称以*Repository*结尾的任何bean，请提供值\* Repository。 要提供多种模式，请在逗号分隔的列表中定义它们。 对于bean定义autowire-candidate属性，显式值true或false总是优先的，对于这样的bean，模式匹配规则不适用。

这些技术对于不想通过自动装配注入其他bean的bean非常有用。 这并不意味着排除的bean本身不能使用自动装配进行配置。 相反，该bean本身不是自动装配其他bean的候选者。

1.4.6. Method injection(方法注入)

在大多数应用场景中，容器中的大部分bean都是 [singletons](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-scopes-singleton)。 当单例bean需要与另一个单例bean协作，或者非单例bean需要与另一个非单例bean协作时，通常通过将一个bean定义为另一个bean的属性来处理依赖。 当bean生命周期不同时会出现问题。 假设单例bean A需要使用非单例（原型）bean B，也许在A的每个方法调用上。容器只创建一个单例bean A，因此只有一次机会来设置属性。 每次需要时，容器都不能向bean A提供bean B的新实例

一个解决方案是放弃一些控制反转。 您可以通过实现ApplicationContextAware接口 [make bean A aware of the container](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-aware) ，并且每当bean A需要时， [making a getBean("B") call to the container](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-client)调用请求（通常是新的）bean B实例。 以下是这种方法的一个例子：

*// a class that uses a stateful Command-style class to perform some processing*

**package** fiona.apple;

*// Spring-API imports*

**import** org.springframework.beans.BeansException;

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

**import** org.springframework.context.ApplicationContextAware;

**public** **class** **CommandManager** **implements** ApplicationContextAware {

**private** ApplicationContext applicationContext;

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

*// grab a new instance of the appropriate Command*

Command command = createCommand();

*// set the state on the (hopefully brand new) Command instance*

command.setState(commandState);

**return** command.execute();

}

**protected** Command createCommand() {

*// notice the Spring API dependency!*

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

}

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

ApplicationContext applicationContext) **throws** BeansException {

this.applicationContext = applicationContext;

}

}

前面的内容是不可取的，因为业务代码知道并耦合到Spring框架。 方法注入是Spring IoC容器的一个高级特性，它允许以干净的方式处理这个用例。

您可以在[this blog entry](https://spring.io/blog/2004/08/06/method-injection/)中阅读更多关于方法注入的动机。

Lookup method injection(查找方法注入)

查找方法注入是容器覆盖容器管理的bean上方法的能力，以返回容器中另一个命名bean的查找结果。 查找通常包含一个原型bean，如前一节所述。 Spring Framework通过使用CGLIB库中的字节码生成来动态生成覆盖该方法的子类，从而实现了此方法注入。

|  |  |
| --- | --- |
|  | * 对于这个动态子类化来说，Spring bean容器将继承的类不能是final的，并且被覆盖的方法也不是final的。 * 对具有抽象方法的类进行单元测试需要您自己对该类进行子类化并提供抽象方法的存根实现。 * 组件扫描也需要具体的方法，这需要具体的类来提取。 * 另外一个关键的限制是查找方法不能用于工厂方法，特别是不能在配置类中使用@Bean方法，因为在这种情况下容器不负责创建实例，因此不能创建运行时生成的 飞行中的子类。 |

查看前面代码片断中的CommandManager类，可以看到Spring容器将动态地覆盖createCommand()方法的实现。 你的CommandManager类不会有任何的Spring依赖关系，在重做的例子中可以看到：

**package** fiona.apple;

*// no more Spring imports!*

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

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

*// grab a new instance of the appropriate Command interface*

Command command = createCommand();

*// set the state on the (hopefully brand new) Command instance*

command.setState(commandState);

**return** command.execute();

}

*// okay... but where is the implementation of this method?*

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

}

在包含要注入的方法的客户机类（本例中为CommandManager）中，要注入的方法需要以下形式的签名：

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

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

*<!-- a stateful bean deployed as a prototype (non-singleton) -->*

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

*<!-- inject dependencies here as required -->*

</bean>

*<!-- commandProcessor uses statefulCommandHelper -->*

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

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

</bean>

标识为commandManager的bean在需要myCommand bean的新实例时调用其自己的方法 createCommand()。 您必须小心地将myCommand bean作为原型部署，如果这实际上是需要的话。 如果它是 [singleton](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-scopes-singleton)，则每次返回myCommand bean的相同实例。

或者，在基于注释的组件模型中，您可以通过@Lookupannotation声明查找方法：

**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，更习惯地说，您可能依赖于针对查找方法的声明返回类型解析的目标bean：

**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，您通常会使用具体的存根实现来声明这样的带注解的查找方法，以使它们与Spring的组件扫描规则兼容，其中抽象类在默认情况下被忽略。 这个限制不适用于显式注册或显式导入的bean类。

|  |  |
| --- | --- |
|  | 访问不同范围的目标bean的另一种方式是ObjectFactory / Provider注入点。 检查 [Scoped beans as dependencies](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-scopes-other-injection)。  感兴趣的读者也可以找到ServiceLocatorFactoryBean （在org.springframework.beans.factory.config 包中）以供使用。 |

Arbitrary method replacement(任意方法替换)

与查找方法注入相比，不太有用的方法注入形式是能够用另一个方法实现来替换托管bean中的任意方法。 用户可以安全地跳过本节的其余部分，直到实际需要功能为止。

使用基于XML的配置元数据，您可以使用replaced-method元素将现有的方法实现替换为部署的bean。 考虑下面的类，我们想要覆盖一个方法computeValue：

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

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

*// some real code...*

}

*// some other methods...*

}

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

*/\*\**

*\* meant to be used to override the existing computeValue(String)*

*\* implementation in MyValueCalculator*

*\*/*

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

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

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

String input = (String) args[0];

...

return ...;

}

}

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">

*<!-- arbitrary method replacement -->*

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

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

</replaced-method>

</bean>

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

您可以使用<replaced-method />元素中的一个或多个包含的<arg-type />元素来指示被覆盖的方法的方法签名。 只有当方法被重载并且类中存在多个变体时，参数的签名才是必需的。 为方便起见，参数的类型字符串可能是完全限定类型名称的子字符串。 例如，以下全部匹配java.lang.String：

java.lang.String

String

Str

因为参数的数量往往足以区分每个可能的选择，所以这个快捷方式可以节省大量的输入，只允许输入与参数类型匹配的最短字符串。

1.5. Bean scopes(作用范围)

当你创建一个bean定义时，你创建一个配方来创建由该bean定义定义的类的实际实例。 bean定义是一个配方的想法很重要，因为这意味着，就像一个类一样，您可以从一个配方创建许多对象实例。

您不仅可以控制要插入到从特定的bean定义创建的对象中的各种依赖项和配置值，还可以控制从特定的bean定义创建的对象的范围。 这种方法功能强大且灵活，因为您可以选择通过配置创建的对象的范围，而不必在Java类级别上烘焙对象的范围。 Bean可以被定义为部署在多个作用域中的一个：开箱即用，Spring框架支持六个作用域，其中四个作用域只有在使用Web感知的ApplicationContext时才可用。

开箱即用支持以下范围。 您也可以创建 [a custom scope.](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-scopes-custom)。

| *Table 3. Bean scopes* | |
| --- | --- |
| **Scope** | **Description** |
| [singleton](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-scopes-singleton) | (Default) Scopes a single bean definition to a single object instance per Spring IoC container. |
| [prototype](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-scopes-prototype) | Scopes a single bean definition to any number of object instances. |
| [request](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-scopes-request) | Scopes a single bean definition to the lifecycle of a single HTTP request; that is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext. |
| [session](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-scopes-session) | Scopes a single bean definition to the lifecycle of an HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext. |
| [application](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-scopes-application) | Scopes a single bean definition to the lifecycle of a ServletContext. Only valid in the context of a web-aware Spring ApplicationContext. |
| [websocket](https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#websocket-stomp-websocket-scope) | Scopes a single bean definition to the lifecycle of a WebSocket. Only valid in the context of a web-aware Spring ApplicationContext. |

|  |  |
| --- | --- |
|  | 从Spring 3.0开始，*thread scope*可用，但默认情况下未注册。 有关更多信息，请参阅 [SimpleThreadScope](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/context/support/SimpleThreadScope.html)的文档。 有关如何注册此或任何其他自定义作用域的说明，请参阅[Using a custom scope](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-scopes-custom-using)。 |

1.5.1. The singleton scope

只管理单个bean的一个共享实例，并且具有与该bean定义匹配的id或id的bean的所有请求都会导致Spring容器返回一个特定的bean实例。

换句话说，当你定义一个bean定义并将其作为一个singleton作用域时，Spring IoC容器恰好创建了该bean定义定义的对象的一个实例。 这个单个实例被存储在这样的singleton bean的缓存中，并且该命名bean的所有后续请求和引用都会返回缓存的对象。



Spring的singleton bean概念与Gang of Four (GoF) 模式书中定义的Singleton模式不同。 GoF Singleton对对象的范围进行hard-codes，以便每个ClassLoader创建一个特定类的唯一实例。 Spring单例的范围最好按容器和每个bean来描述。 这意味着如果您为单个Spring容器中的特定类定义一个bean，那么Spring容器将创建该bean定义所定义的类的一个且仅有的一个实例。 单例作用域是Spring中的默认作用域。 要在XML中将bean定义为单例，您可以编写如下的代码：

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

*<!-- the following is equivalent, though redundant (singleton scope is the default) -->*

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

1.5.2. The prototype scope

bean的部署的non-singleton，prototype scope导致每次创建一个新的bean实例。 也就是说，该bean被注入到另一个bean中，或者通过容器上的getBean()方法调用来请求它。 通常，使用所有有状态bean的prototype scope和无状态bean的单例作用域。

下图说明了Spring的prototype scope。 数据访问对象（DAO）通常不配置为*prototype*，因为典型的DAO不具有任何对话状态; 这位作者更容易重用*the core of the singleton diagram*。



以下示例将bean定义为XML中的prototype：

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

与其他范围相比，Spring不管理prototype bean的完整生命周期：容器实例化，配置并以其他方式组装prototype对象，并将其交给客户端，而不再记录该prototype实例。 因此，虽然在所有对象上调用初始化生命周期回调方法，但是在prototype的情况下，不调用配置的销毁生命周期回调。 客户端代码必须清理prototype-scoped的对象，并释放prototype bean持有的昂贵资源。 为了让Spring容器释放原型范围bean所拥有的资源，可以尝试使用一个自定义 [bean post-processor](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-extension-bpp)，该后处理器保存对需要清理的bean的引用。

在某些方面，Spring容器在prototype-scoped的bean方面的角色是Java新运算符的替代。 所有生命周期管理过去都必须由客户来处理。 （有关Spring容器中bean的生命周期的详细信息，请参阅 [Lifecycle callbacks](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-lifecycle)。

1.5.3. Singleton beans with prototype-bean dependencies

当您使用具有依赖于singleton-scoped的单一范围的bean时，请注意在实例化时解析依赖关系。 因此，如果你依赖注入一个prototype-scoped的bean到一个单一范围的bean中，一个新的prototype-scoped被实例化，然后依赖注入到单例bean中。 prototype instance是唯一提供给单例范围bean的实例。

但是，假设您希望singleton-scoped的bean在运行时重复获取prototype-scoped的bean的新实例。 你不能依赖注入一个prototype-scoped的bean到你的单例bean中，因为这个注入只发生在Spring容器实例化单例bean并解析和注入它的依赖时。 如果您不止一次在运行时需要一个原型bean的新实例，请参阅 [Method injection](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-method-injection) 。

1.5.4. Request, session, application, and WebSocket scopes

request, session, application, and websocket scopes仅在使用web-aware Spring ApplicationContext implementation（如XmlWebApplicationContext）时才可用。 如果将这些范围与常规的Spring IoC容器（如ClassPathXmlApplicationContext）一起使用，则会抛出IllegalStateException 异常，抱怨未知的bean范围。

Initial web configuration(初始化web配置)

要在request, session, application, and websocket levels (web-scoped beans)中支持bean的范围界定，在定义bean之前需要一些次要的初始配置。 （标准范围，singleton 和 prototype不需要此初始设置。)

您如何完成此初始设置取决于您特定的Servlet环境。

如果你在Spring Web MVC中访问scoped beans，实际上，在由Spring DispatcherServlet处理的请求中，不需要特殊的设置：DispatcherServlet 已经公开了所有相关的状态。

如果您使用Servlet 2.5 Web容器，并且在Spring的DispatcherServlet之外处理请求（例如，使用JSF或Struts时），则需要注册org.springframework.web.context.request.RequestContextListenerServletRequestListener。 对于Servlet 3.0+，这可以通过WebApplicationInitializer接口以编程方式完成。 或者，对于较老的容器，将以下声明添加到Web应用程序的web.xml文件中：

<web-app>

...

<listener>

<listener-class>

org.springframework.web.context.request.RequestContextListener

</listener-class>

</listener>

...

</web-app>

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

<web-app>

...

<filter>

<filter-name>requestContextFilter</filter-name>

<filter-class>org.springframework.web.filter.RequestContextFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>requestContextFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

...

</web-app>

DispatcherServlet，RequestContextListener和RequestContextFilter都完全相同，即将HTTP请求对象绑定到服务该请求的线程。 这使得请求和session-scoped的bean可以在调用链的下面。

Request scope

考虑以下用于bean定义的XML配置：

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

Spring容器通过对每个HTTP请求使用loginAction bean定义来创建LoginAction bean的新实例。 也就是说，loginAction bean的作用域是HTTP请求级别。 您可以根据需要更改创建的实例的内部状态，因为从同一个loginAction bean定义创建的其他实例不会看到这些状态变化; 他们对个人的要求很特别。 当请求完成处理时，作用于该请求的bean将被丢弃。

使用注解驱动的组件或Java Config时，可以使用@RequestScope注释将组件分配到request scope。

**@RequestScope**

@Component

**public** **class** **LoginAction** {

*// ...*

}

Session scope

考虑以下用于bean定义的XML配置：

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

Spring容器通过在单个HTTP会话的生命周期中使用userPreferences bean定义来创建UserPreferences bean的新实例。 换句话说，userPreferences bean在HTTP会话级别有效。 与请求范围bean一样，您可以根据需要更改所创建实例的内部状态，因为知道使用同一userPreferences bean定义创建的实例的其他HTTP Session实例在状态中看不到这些更改 ，因为它们对于单独的HTTP会话是特定的。 当HTTP会话最终被丢弃时，作用于该特定HTTP会话的bean也被丢弃。

在使用annotation-driven组件或Java Config时，可以使用@SessionScope注解将组件分配到session scope。

**@SessionScope**

@Component

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

*// ...*

}

Application scope

考虑以下用于bean定义的XML配置：

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

Spring容器通过对整个Web应用程序使用appPreferences bean定义一次来创建AppPreferences bean的新实例。 也就是说，appPreferences bean的作用域是ServletContext级别，作为一个regularServletContext属性存储。 这有点类似于Spring单例bean，但在两个重要方面有所不同：它是每个ServletContext的单例，而不是每个Spring的“ApplicationContext”（在任何给定的Web应用程序中可能有几个），它实际上是暴露的，因此 作为ServletContext属性可见。

使用annotation-driven的组件或Java Config时，可以使用@ApplicationScope注解将组件分配给应用application scope。

**@ApplicationScope**

@Component

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

*// ...*

}

Scoped beans as dependencies

Spring IoC容器不仅管理对象（bean）的实例化，还管理协作者(collaborators)（或依赖关系）的连接。 如果你想把一个HTTP请求范围的bean注入到另一个更长的作用域的bean中，你可以选择注入一个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相同的公共接口，但也可以从相关作用域（例如HTTP请求）中检索真实的目标对象，并将方法调用委托给实际对象。

|  |  |
| --- | --- |
|  | 您也可以在范围为singleton的bean之间使用<aop：scoped-proxy />，然后引用通过一个可序列化的中间代理，因此可以在反序列化中重新获得目标单例bean。  当针对范围原型的bean声明<aop：scoped-proxy />时，共享代理上的每个方法调用都将导致创建一个新的目标实例，然后该呼叫将被转发到该实例。  此外，范围代理不是以生命周期安全的方式从较短范围访问Bean的唯一方法。 您也可以简单地将注入点（即constructor/setter参数或autowired field）声明为ObjectFactory <MyTargetBean>，从而允许getObject（）调用在每次需要时按需检索当前needed - without保留 实例或单独存储。  作为一个扩展变体，您可以声明ObjectProvider <MyTargetBean>，它提供了几个额外的访问变体，包括getIfAvailable和getIfUnique。  这个JSR-330变种被称为Provider，与Provider <MyTargetBean>声明和相应的get（）调用一起用于每个检索尝试。 有关JSR-330的更多详细信息，请参阅 [here](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-standard-annotations)。 |

以下示例中的配置只有一行，但了解“why”以及其后面的“how”很重要。

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

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

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

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

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

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

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

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

*<!-- an HTTP Session-scoped bean exposed as a proxy -->*

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

*<!-- instructs the container to proxy the surrounding bean -->*

<aop:scoped-proxy/>

</bean>

*<!-- a singleton-scoped bean injected with a proxy to the above bean -->*

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

*<!-- a reference to the proxied userPreferences bean -->*

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

</bean>

</beans>

要创建这样一个代理(proxy)，可以将一个子<aop：scoped-proxy />元素插入到一个有作用域的bean定义中（请参阅 [Choosing the type of proxy to create](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-scopes-other-injection-proxies) and [XML Schema-based configuration](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#xsd-schemas)）。 为什么在request, session and custom-scope levels require级别定义的bean需要<aop：scoped-proxy />元素？ 让我们来看看下面的singleton bean的定义，并将其与您需要为上述范围定义的内容进行对比（请注意，以下userPreferences bean定义不完整）。

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

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

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

</bean>

在前面的例子中，singleton bean userManager注入了对HTTP Session范围的bean userPreferences的引用。 这里的要点是userManager bean是一个singleton：它将被实例化，每个容器只有一次，它的依赖关系（在这种情况下，只有一个，userPreferences bean）也只被注入一次。 这意味着userManager bean只能在完全相同的userPreferences对象上运行，也就是最初注入的对象。

这不是将寿命较短的作用域bean注入到更长寿命的作用域bean中时所需的行为，例如将HTTP会话作用域的合作bean作为依赖注入到单例bean中。相反，您需要一个userManager对象，并且在HTTP Session的生命周期中，您需要一个特定于所述HTTP Session的userPreferences对象。因此，容器创建一个对象，该对象公开与UserPreferences类（理想情况下为UserPreferences实例的对象）完全相同的公共接口，该对象可以从作用域机制（HTTP请求，Session等）中获取真实的UserPreferences对象。容器将这个代理对象注入到userManager bean中，而不知道这个UserPreferences引用是一个代理。在这个例子中，当一个UserManager实例在依赖注入的UserPreferences对象上调用一个方法时，它实际上是在代理上调用一个方法。代理然后从（在本例中）HTTP会话中获取realUserPreferences对象，并将方法调用委托给所检索的实际UserPreferences对象。

因此，在将 request- and session-scoped的bean注入协作对象时，您需要以下正确且完整的配置：

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

<aop:scoped-proxy/>

</bean>

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

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

</bean>

Choosing the type of proxy to create

默认情况下，当Spring容器为使用<aop：scoped-proxy />元素标记的bean创建代理时，将创建一个基于CGLIB的类代理。

|  |  |
| --- | --- |
|  | CGLIB代理只拦截公共方法调用！ 不要在这样的代理上调用非公共方法; 它们不会被委托给实际的作用域对象。 |

或者，可以通过为<aop：scoped-proxy />元素的proxy-target-class属性的值指定false来配置Spring容器，以便为此类范围的bean创建标准的基于JDK接口的代理。 使用基于JDK接口的代理意味着您不需要应用程序类路径中的其他库来实现这种代理。 但是，这也意味着作用域bean的类必须至少实现一个接口，并且注入了作用域bean的所有协作者都必须通过它的一个接口引用bean。

*<!-- DefaultUserPreferences implements the UserPreferences interface -->*

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

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

</bean>

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

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

</bean>

有关选择基于类或基于接口的代理的更多详细信息，请参阅 [Proxying mechanisms](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-proxying)。

1.5.5. Custom scopes

bean的范围机制是可扩展的; 你可以定义自己的范围，甚至重新定义现有的范围，尽管后者被认为是不好的做法，你不能重写built-in singleton and prototype scopes。

Creating a custom scope

要将自定义作用域集成到Spring容器中，需要实现本节中描述的org.springframework.beans.factory.config.Scope接口。 有关如何实现自己的作用域的想法，请参阅随Spring Framework本身提供的Scope实现和 [Scope javadocs](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/beans/factory/config/Scope.html)，它解释了您需要更详细地实现的方法。

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

以下方法从基础范围返回对象。 例如，会话范围实现返回会话范围的bean（如果它不存在，该方法在将其绑定到会话以供将来参考之后返回该bean的新实例）。

Object get(String name, ObjectFactory objectFactory)

以下方法将该对象从基础范围中移除。 例如，会话范围实现从基础会话中删除会话范围的bean。 应该返回该对象，但如果找不到具有指定名称的对象，则可以返回null。

Object remove(String name)

以下方法注册范围在销毁时或者范围中的指定对象被销毁时应执行的回调。 有关销毁回调的更多信息，请参阅javadocs或Spring范围实现。

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

以下方法获取基础范围的对话标识符。 这个标识符对于每个范围是不同的。 对于会话范围的实现，这个标识符可以是session标识符。

String getConversationId()

Using a custom scope

在编写和测试一个或多个自定义Scope实现之后，您需要使Spring容器知道新的作用域。 以下方法是使用Spring容器注册新Scope的核心方法：

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

此方法在ConfigurableBeanFactory接口上声明，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与范围关联的唯一名称; Spring容器本身的这些名字的例子是singleton和prototype。 registerScope（..）方法的第二个参数是您希望注册和使用的自定义Scope实现的实际实例。

假设你编写你自定义的Scope实现，然后如下注册它。

|  |  |
| --- | --- |
|  | 下面的例子使用Spring包含的SimpleThreadScope，但默认情况下未注册。 这些指令对于您自己的自定义Scope实现将是相同的。 |

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

beanFactory.registerScope("thread", threadScope);

然后创建符合自定义作用域范围规则的bean定义：

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

通过自定义的范围实现，您不限于范围的程序注册。 您还可以使用CustomScopeConfigurer类以声明方式执行Scoperegistration：

<?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>

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

1.6. Customizing the nature of a bean

1.6.1. Lifecycle callbacks

要与bean生命周期的容器管理进行交互，可以实现Spring InitializingBean 和DisposableBean 接口。 容器为前者调用afterPropertiesSet()，为后者调用destroy()以允许bean在初始化和销毁bean时执行某些操作。

|  |  |
| --- | --- |
|  | JSR-250 @PostConstruct and @PreDestroy注解通常被认为是在现代Spring应用程序中接收生命周期回调的最佳实践。 使用这些注解意味着你的bean没有耦合到Spring特定的接口。 有关详细信息，请参阅 [@PostConstruct and @PreDestroy](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-postconstruct-and-predestroy-annotations)。  如果您不想使用JSR-250注解，但仍想要移除耦合，请考虑使用init-method和destroy-method对象定义元数据。 |

在内部，Spring框架使用BeanPostProcessor 实现来处理它可以找到的任何回调接口并调用适当的方法。 如果您需要自定义功能或其他生命周期行为，Spring不提供开箱即用的功能，您可以自己实现BeanPostProcessor 。 有关更多信息，请参阅 [Container Extension Points](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-extension)。

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

本节描述生命周期回调接口。

Initialization callbacks

org.springframework.beans.factory.InitializingBean接口允许bean在bean的所有必要属性已由容器设置后执行初始化工作。 InitializingBean 接口指定一个方法：

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

建议您不要使用InitializingBean 接口，因为它不必要地将代码耦合到Spring。 或者，使用 [@PostConstruct](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-postconstruct-and-predestroy-annotations)注解或指定一个POJO初始化方法。 对于基于XML的配置元数据，可以使用init-method属性来指定具有void无参数签名的方法的名称。 使用Java配置，您可以使用@Bean的initMethod 属性，请参阅 [Receiving lifecycle callbacks](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-java-lifecycle-callbacks)。 例如，以下内容：

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

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

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

*// do some initialization work*

}

}

…​是完全一样的…​

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

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

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

*// do some initialization work*

}

}

但是不会将代码耦合到Spring

Destruction callbacks

实现 org.springframework.beans.factory.DisposableBean接口允许bean在包含它的容器被销毁时获得回调。 DisposableBean 接口指定一个方法：

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

建议您不要使用DisposableBean 回调接口，因为它不必要地将代码耦合到Spring。 或者，使用 [@PreDestroy](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-postconstruct-and-predestroy-annotations)注解或指定bean定义支持的通用方法。 使用XML-based的配置元数据时，可以使用<bean/>上的 destroy-method属性。 使用Java配置，您可以使用@Bean的destroyMethod 属性，请参阅 [Receiving lifecycle callbacks](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-java-lifecycle-callbacks)。 例如，下面的定义：

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

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

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

*// do some destruction work (like releasing pooled connections)*

}

}

是完全一样的:

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

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

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

*// do some destruction work (like releasing pooled connections)*

}

}

但不会将代码耦合到Spring

|  |  |
| --- | --- |
|  | 可以为<bean>元素的destroy-method属性赋予一个特殊的（推断的）值，它指示Spring自动检测特定bean类（实现java.lang.AutoCloseable or java.io.Closeable的任何类）上的public  close or shutdown方法。 因此io.Closeable会匹配）。 这个特殊的（推断的）值也可以在<beans>元素的default-destroy-method属性上设置，以将此行为应用于整个bean集合（请参阅 [Default initialization and destroy methods](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-lifecycle-default-init-destroy-methods)）。 请注意，这是Java配置的默认行为。 |

Default initialization and destroy methods

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

您可以配置Spring容器以查找命名初始化并销毁每个bean上的回调方法名称。 这意味着，作为应用程序开发人员，您可以编写应用程序类并使用称为 init()的初始化回调，而无需为每个bean定义配置init-method="init"属性。 Spring IoC容器在创建bean时（并根据前面描述的标准生命周期回调协议）调用该方法。 此功能还为初始化和销毁方法回调强制执行一致的命名约定。

假设你的初始化回调方法被命名为init()，并且销毁回调方法被命名为destroy()。 在下面的例子中，你的class将类似于class。

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

**private** BlogDao blogDao;

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

this.blogDao = blogDao;

}

*// this is (unsurprisingly) the initialization callback method*

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

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

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

}

}

}

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

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

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

</bean>

</beans>

顶层 <beans/> 元素属性中default-init-method属性的存在会导致Spring IoC容器识别出一个名为init 的方法作为初始化方法回调。 当一个bean被创建和组装时，如果bean类有这样一个方法，它会在适当的时候被调用。

通过在顶级 <beans/>元素上使用default-destroy-method属性，可以类似地配置destroy方法回调（即在XML中）。

在现有bean类已经具有与惯例不同的回调方法的情况下，可以通过使用<bean/>的init-method and destroy-method属性指定方法名称（即XML）来覆盖缺省值 本身。

Spring容器保证了一个配置好的初始化回调函数在bean被提供了所有的依赖关系后立即被调用。 因此初始化回调在原始bean引用上被调用，这意味着AOP拦截器等等还没有被应用到bean。 目标bean首先被完全创建，然后应用带有其拦截器链的AOP代理（例如）。 如果目标bean和代理是分别定义的，那么代码甚至可以绕过代理与原始目标bean进行交互。 因此，将拦截器应用于init方法会不一致，因为这样会将目标bean的生命周期与代理/拦截器耦合在一起，并在代码直接与原始目标bean交互时留下奇怪的语义。

Combining lifecycle mechanisms

从Spring 2.5开始，您有三个控制bean生命周期行为的选项：[InitializingBean](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-lifecycle-initializingbean) and [DisposableBean](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-lifecycle-disposablebean)接口; 自定义init() and destroy()方法; 和 [@PostConstruct and @PreDestroy annotations](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-postconstruct-and-predestroy-annotations)。 你可以结合这些机制来控制给定的bean。

|  |  |
| --- | --- |
|  | 如果为bean配置了多个生命周期机制，并且每个机制都配置了不同的方法名称，那么每个配置的方法都按照下面列出的顺序执行。 但是，如果为这些生命周期机制中的多个生命周期机制配置了相同的方法名称(例如初始化方法的init())，则该方法将执行一次，如前一部分所述。 |

为相同的bean配置多种生命周期机制，使用不同的初始化方法，如下所示：

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

Destroy 方法以相同的顺序被调用:

* 使用 @PreDestroy注解
* destroy() as 由DisposableBean  会掉借口定义的destroy()
* 自定义配置的 destroy() 方法

Startup and shutdown callbacks

Lifecycle 接口为任何具有自己生命周期要求的对象（例如启动和停止一些后台进程）定义基本方法：

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

**void** start();

**void** stop();

**boolean** isRunning();

}

任何Spring管理的对象都可以实现该接口。 然后，当ApplicationContext 本身接收到启动和停止信号时，例如 对于运行时的停止/重新启动场景，它会将这些调用级联到在该上下文中定义的所有Lifecycle 实现。 它通过委派给LifecycleProcessor来完成此操作：

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

**void** onRefresh();

**void** onClose();

}

请注意，LifecycleProcessor 本身就是Lifecycle 接口的扩展。 它还添加了两种其他方法来对正在刷新和关闭的上下文作出反应。

|  |  |
| --- | --- |
|  | 请注意，常规的 org.springframework.context.Lifecycle接口只是显式start/stop通知的普通协定，并不意味着在上下文刷新时自动启动。 考虑实施org.springframework.context.SmartLifecycle，而不是对特定bean的自动启动（包括启动阶段）进行细粒度控制。 此外，请注意，停止通知不保证在销毁之前发生：在正常关闭时，所有Lifecycle  bean将在传播通用销毁回调之前首先收到停止通知; 然而，在上下文的生命周期中的热刷新或中止刷新尝试时，只会调用销毁方法。 |

启动和关闭调用的顺序可能很重要。 如果任何两个对象之间存在"depends-on"，则依赖方将在其依赖关系之后启动，并且在依赖关系之前停止。 但是，有时直接依赖关系是未知的。 您可能只知道某种类型的对象应该在另一种类型的对象之前启动。 在这些情况下，SmartLifecycle 接口定义了另一个选项，即在其超级接口Phased上定义的getPhase()方法。

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

**int** getPhase();

}

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

**boolean** isAutoStartup();

**void** stop(Runnable callback);

}

启动时，相位最低的物体首先启动，停止时跟随相反的顺序。 因此，实现SmartLifecycle并且其 getPhase() 方法返回Integer.MIN\_VALUE的对象将成为第一个开始和最后一个停止的对象。 在频谱的另一端，Integer.MAX\_VALUE的相位值将指示该对象应该最后开始并且首先停止（可能是因为它取决于要运行的其他进程）。 在考虑相位值时，了解任何未实现SmartLifecycle 的"normal" Lifecycle 对象的默认阶段为0也很重要。因此，任何负相位值都表示对象应在这些标准组件之前启动（并且 在它们之后停止），反之亦然，对于任何正相位值。

正如您所看到的，SmartLifecycle 定义的stop方法接受回调。 任何实现必须在该实现的关闭过程完成后调用该回调的 run()方法。 这可以在需要时进行异步关闭，因为LifecycleProcessor 接口的默认实现DefaultLifecycleProcessor将等待每个阶段中的对象组的超时值以调用该回调。 每个阶段的默认超时时间是30秒。 您可以通过在上下文中定义一个名为"lifecycleProcessor"的bean来覆盖默认的生命周期处理器实例。 如果您只想修改超时值，那么定义以下就足够了：

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

*<!-- timeout value in milliseconds -->*

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

</bean>

如前所述，LifecycleProcessor 接口还定义了用于刷新和关闭上下文的回调方法。后者将简单地驱动关闭过程，就好像stop()已被显式调用一样，但是当上下文关闭时会发生。另一方面，'刷新'回调启用了SmartLifecycle beans的另一个功能。当上下文刷新时（在所有对象被实例化和初始化之后），该回调将被调用，并且此时默认生命周期处理器将检查每个SmartLifecycle object’s isAutoStartup()方法返回的布尔值。如果为“true”，那么该对象将在那个时候启动，而不是等待显式调用上下文或自己的start()方法（与上下文刷新不同，上下文启动不会自动执行标准上下文） 。 "phase"值以及任何"depends-on"关系将以与上述相同的方式确定启动顺序。

Shutting down the Spring IoC container gracefully in non-web applications

|  |  |
| --- | --- |
|  | 本节仅适用于非Web应用程序。 Spring的基于Web的ApplicationContext 实现已经有适当的代码来在关闭相关Web应用程序时正常关闭Spring IoC容器。 |

如果您在非Web应用程序环境中使用Spring的IoC容器， 例如，在富客户端桌面环境中; 您使用JVM注册了一个关闭钩子。 这样做可以确保正常关闭并在单例bean上调用相关的销毁方法，从而释放所有资源。 当然，您仍然必须正确配置和实施这些销毁回调。

要注册一个关闭挂钩，可以调用ConfigurableApplicationContext 接口上声明的registerShutdownHook()方法：

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

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

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

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

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

*// add a shutdown hook for the above context...*

ctx.registerShutdownHook();

*// app runs here...*

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

}

}

1.6.2. ApplicationContextAware and BeanNameAware

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

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

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

}

因此，bean可以通过ApplicationContext 接口或通过将引用强制转换为此接口的已知子类（如ConfigurableApplicationContext）来创建它们，从而以编程方式操作ApplicationContext ，该类可公开其他功能。 一个用途是对其他bean的程序化检索。 有时候这种能力是有用的; 但是，通常你应该避免它，因为它将代码耦合到Spring，并且不遵循Inversion of Control风格，其中协作者被提供给bean作为属性。 ApplicationContext的其他方法提供对文件资源的访问，发布应用程序事件以及访问MessageSource。 这些 [Additional capabilities of the ApplicationContext](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#context-introduction)中进行了描述。

从Spring 2.5开始，自动装配是获得对ApplicationContext的引用的另一种方法。 类型自动装配模式（如 [Autowiring collaborators](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-autowire)中所述）的"traditional"构造和分别可为typeApplicationContext 提供构造函数参数或setter方法参数的依赖关系。 为了获得更大的灵活性，包括自动装配字段和多个参数方法的能力，请使用新的基于注释的自动装配功能。 如果这样做，则ApplicationContext 会自动装入字段，构造函数参数或方法参数中，如果所涉及的字段，构造函数或方法携带 @Autowired 注释，则该参数将期望ApplicationContext 类型。 有关更多信息，请参阅 [@Autowired](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-autowired-annotation)。

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

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

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

}

这个回调函数是在正常的bean属性填充之后，但在初始化回调之前调用的，例如InitializingBean  *afterPropertiesSet* 或者一个自定义的init-method。

1.6.3. Other Aware interfaces

除了上面讨论的ApplicationContextAware and BeanNameAware 之外，Spring提供了一系列Aware接口，允许bean向容器指示它们需要某种基础设施依赖性。 最重要的Aware接口总结如下 - 作为一般规则，名称是依赖类型的一个很好的指示：

| *Table 4. Aware interfaces* | | |
| --- | --- | --- |
| **Name** | **Injected Dependency** | **Explained in…​** |
| ApplicationContextAware | Declaring ApplicationContext | [ApplicationContextAware and BeanNameAware](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-aware) |
| ApplicationEventPublisherAware | Event publisher of the enclosing ApplicationContext | [Additional capabilities of the ApplicationContext](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#context-introduction) |
| BeanClassLoaderAware | Class loader used to load the bean classes. | [Instantiating beans](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-class) |
| BeanFactoryAware | Declaring BeanFactory | [ApplicationContextAware and BeanNameAware](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-aware) |
| BeanNameAware | Name of the declaring bean | [ApplicationContextAware and BeanNameAware](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-aware) |
| BootstrapContextAware | Resource adapter BootstrapContextthe container runs in. Typically available only in JCA aware ApplicationContexts | [JCA CCI](https://docs.spring.io/spring/docs/current/spring-framework-reference/integration.html#cci) |
| LoadTimeWeaverAware | Defined *weaver* for processing class definition at load time | [Load-time weaving with AspectJ in the Spring Framework](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-aj-ltw) |
| MessageSourceAware | Configured strategy for resolving messages (with support for parametrization and internationalization) | [Additional capabilities of the ApplicationContext](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#context-introduction) |
| NotificationPublisherAware | Spring JMX notification publisher | [Notifications](https://docs.spring.io/spring/docs/current/spring-framework-reference/integration.html#jmx-notifications) |
| ResourceLoaderAware | Configured loader for low-level access to resources | [Resources](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#resources) |
| ServletConfigAware | Current ServletConfig the container runs in. Valid only in a web-aware Spring ApplicationContext | [Spring MVC](https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc) |
| ServletContextAware | Current ServletContext the container runs in. Valid only in a web-aware Spring ApplicationContext | [Spring MVC](https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc) |

再次注意，这些接口的使用将您的代码绑定到Spring API，并且不遵循控制反转样式。 因此，它们被推荐用于需要对容器进行编程访问的基础架构bean。

1.7. Bean definition inheritance

一个bean定义可以包含很多配置信息，包括构造函数参数，属性值和特定于容器的信息，例如初始化方法，静态工厂方法名称等等。 子bean定义从父定义继承配置数据。 根据需要，子定义可以覆盖一些值或添加其他值。 使用父和子bean定义可以节省大量的输入。 实际上，这是一种模板形式。

如果以编程方式使用ApplicationContext 接口，则子Bean定义由ChildBeanDefinition 类表示。 大多数用户在这个级别上不使用它们，而是用类似ClassPathXmlApplicationContext的方式声明性地配置bean定义。 在使用XML-based的配置元数据时，可以使用父属性指定子bean定义，并指定父bean作为此属性的值。

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

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

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

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

</bean>

<bean id="inheritsWithDifferentClass"

class="org.springframework.beans.DerivedTestBean"

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

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

*<!-- the age property value of 1 will be inherited from parent -->*

</bean>

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

子bean定义继承了父级的scope，构造函数参数值，属性值和方法重写，并且可以添加新值。 您指定的任何范围，初始化方法，销毁方法和/或静态工厂方法设置都将覆盖相应的父设置。

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

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

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

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

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

</bean>

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

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

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

*<!-- age will inherit the value of 1 from the parent bean definition-->*

</bean>

父bean不能自行实例化，因为它是不完整的，并且它也明确标记为抽象。 当定义像这样抽象时，它只能用作纯模板bean定义，作为子定义的父定义。 尝试单独使用这样的抽象父bean，通过将其作为另一个bean的ref属性或使用父bean id执行显式getBean()调用返回错误。 同样，容器的internalpreInstantiateSingletons()方法也会忽略定义为抽象的bean定义。

|  |  |
| --- | --- |
|  | ApplicationContext默认预先实例化所有单例。 因此，重要的是（至少对于singleton beans），如果你有一个你打算只用作模板的（父）bean定义，并且这个定义指定了一个类，那么你必须确保将abstract属性设置为true ，否则应用程序上下文将实际（尝试）预先实例化抽象bean。 |

1.8. Container Extension Points

通常，应用程序开发人员不需要继承ApplicationContext 实现类。 相反，Spring IoC容器可以通过插入特殊集成接口的实现来扩展。 接下来的几节将介绍这些集成接口。

1.8.1. Customizing beans using a BeanPostProcessor

BeanPostProcessor 接口定义了您可以实现的回调方法，以提供您自己的（或覆盖容器的默认）实例化逻辑，依赖关系解析逻辑等等。 如果你想在Spring容器完成实例化，配置和初始化bean之后实现一些定制逻辑，你可以插入一个或多个BeanPostProcessor 实现。

您可以配置多个BeanPostProcessor 实例，并且可以通过设置订单属性来控制这些BeanPostProcessors 执行的顺序。 只有BeanPostProcessor 实现Orderedinterface时，才能设置此属性; 如果你编写自己的BeanPostProcessor ，你应该考虑实现Ordered 接口。 有关更多详细信息，请参阅BeanPostProcessor 和Ordered 接口的javadocs。 另请参阅以下有关 [programmatic registration of BeanPostProcessors](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-programmatically-registering-beanpostprocessors)的注释。

|  |  |
| --- | --- |
|  | BeanPostProcessors对bean（或对象）实例进行操作; 也就是说，Spring IoC容器实例化一个bean实例，然后BeanPostProcessors完成它们的工作。  BeanPostProcessors是每个容器的作用域。 这只有在使用容器层次结构时才有意义。 如果你在一个容器中定义一个BeanPostProcessor，它只会在该容器中后处理这些bean。 换句话说，在一个容器中定义的bean不会被另一个容器中定义的BeanPostProcessor进行后处理，即使两个容器都是同一层次结构的一部分。  要更改实际的bean定义（即定义bean的*blueprint* ），您需要使用BeanFactoryPostProcessor ，如使用BeanFactoryPostProcessor [Customizing configuration metadata with a BeanFactoryPostProcessor](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-extension-factory-postprocessors)中所述。 |

org.springframework.beans.factory.config.BeanPostProcessor接口恰好包含两个回调方法。 当这样的类被注册为容器的后处理器(post-processor)时，对于由容器创建的每个bean实例，后处理器(post-processor)都会在容器初始化方法（如InitializingBean的afterPropertiesSet（）之前）和容器 声明的init方法）以及任何bean初始化回调之后被调用。 后处理器可以对bean实例执行任何操作，包括完全忽略回调。 一个bean后处理器通常检查回调接口，或者可能用一个代理包装一个bean。 一些Spring AOP基础设施类被实现为bean后处理器，以提供代理包装逻辑。

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

请注意，在配置类中使用@Bean工厂方法声明BeanPostProcessor时，工厂方法的返回类型应该是实现类本身，或者至少是org.springframework.beans.factory.config.BeanPostProcessor接口，清楚地指示 该bean的后处理器特性。 否则，在完全创建它之前，ApplicationContext将不能按类型自动检测它。 由于BeanPostProcessor需要尽早实例化以适用于上下文中其他bean的初始化，因此这种早期类型检测非常重要。

|  |  |
| --- | --- |
|  | *以编程方式注册BeanPostProcessors*  *虽然BeanPostProcessor注册的推荐方法是通过ApplicationContext自动检测（如上所述），但也可以使用addBeanPostProcessor方法以编程方式将其注册到ConfigurableBeanFactory。 当需要在注册之前评估条件逻辑，或者甚至跨层次结构中的上下文复制Bean后处理器时，这会非常有用。 但请注意，以编程方式添加的BeanPostProcessors不尊重Orderedinterface。 这是注册顺序决定执行顺序。 还要注意，以编程方式注册的BeanPostProcessors总是在通过自动检测注册的BeanPostProcessors之前进行处理，而不管任何明确的排序。* |
|  | *BeanPostProcessors和AOP自动代理*  *实现BeanPostProcessor接口的类是特殊的，并且容器对其进行不同的处理。 作为ApplicationContext特殊启动阶段的一部分，它们直接引用的所有BeanPostProcessors和Bean都将在启动时实例化。 接下来，所有BeanPostProcessors都以已排序的方式注册并应用于容器中的所有其他bean。 因为AOP自动代理被实现为BeanPostProcessor本身，所以BeanPostProcessors和它们直接引用的bean都不适用于自动代理，因此没有编入它们的方面。*  对于任何这样的bean，您应该看到一条信息性日志消息："*Bean foo is not eligible for getting processed by all BeanPostProcessor interfaces (for example: not eligible for auto-proxying)*"。  请注意，如果使用自动装配或@Resource（可能会回退到自动装配）将Bean连接到BeanPostProcessor中，Spring可能会在搜索类型匹配的依赖关系候选时访问意外的Bean，因此使它们不适用于自动代理或其他类型 bean后期处理。 例如，如果你有@Resource标注的依赖项，其中的field/setter名称不直接对应于bean的声明名称，并且没有使用名称属性，那么Spring将访问其他bean以便按类型匹配它们。 |

以下示例显示如何在ApplicationContext中编写，注册和使用BeanPostProcessors。

Example: Hello World, BeanPostProcessor-style

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

在自定义BeanPostProcessor实现类定义下面查找：

**package** scripting;

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

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

*// simply return the instantiated bean as-is*

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

**return** bean; *// we could potentially return any object reference here...*

}

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

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

**return** bean;

}

}

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

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

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

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

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

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

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

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

<lang:groovy id="messenger"

script-source="classpath:org/springframework/scripting/groovy/Messenger.groovy">

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

</lang:groovy>

*<!--*

*when the above bean (messenger) is instantiated, this custom*

*BeanPostProcessor implementation will output the fact to the system console*

*-->*

<bean class="scripting.InstantiationTracingBeanPostProcessor"/>

</beans>

注意InstantiationTracingBeanPostProcessor 是如何定义的。 它甚至没有名称，因为它是一个bean，它可以像其他任何bean一样依赖注入。 （前面的配置也定义了一个由Groovy脚本支持的bean。Spring动态语言支持在标题为 [Dynamic language support](https://docs.spring.io/spring/docs/current/spring-framework-reference/languages.html#dynamic-language)的章节中有详细介绍。）

以下简单的Java应用程序执行前面的代码和配置：

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

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

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

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

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

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

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

System.out.println(messenger);

}

}

前面的应用程序的输出类似于以下内容：

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

org.springframework.scripting.groovy.GroovyMessenger@272961

Example: The RequiredAnnotationBeanPostProcessor

将回调接口或注释与自定义BeanPostProcessor实现结合使用是扩展Spring IoC容器的常用方法。 Spring的RequiredAnnotationBeanPostProcessor就是一个例子，它是Spring发行版的一个BeanPostProcessor实现，它确保bean标记为（任意）注释的JavaBean属性实际上（配置为）依赖注入一个值。

1.8.2. Customizing configuration metadata with a BeanFactoryPostProcessor

下一个我们将看到的扩展点是org.springframework.beans.factory.config.BeanFactoryPostProcessor。 这个接口的语义与BeanPostProcessor的相似，主要区别在于：BeanFactoryPostProcessor在Bean配置元数据上运行; 也就是说，Spring IoC容器允许BeanFactoryPostProcessor读取配置元数据并在容器实例化除BeanFactoryPostProcessor之外的任何Bean之前对其进行更改。

您可以配置多个BeanFactoryPostProcessors，并且您可以通过设置order 属性来控制这些BeanFactoryPostProcessors执行的顺序。 但是，如果BeanFactoryPostProcessor实现Ordered接口，则只能设置此属性。 如果你编写你自己的BeanFactoryPostProcessor，你应该考虑实现Ordered接口。 有关更多详细信息，请参阅BeanFactoryPostProcessor和Orderedinterfaces的javadocs。

|  |  |
| --- | --- |
|  | 如果您想更改实际的bean实例（即从配置元数据创建的对象），则需要使用BeanPostProcessor（如上所述，使用 [Customizing beans using a BeanPostProcessor](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-extension-bpp)）。 虽然技术上可以在BeanFactoryPostProcessor中使用bean实例（例如，使用BeanFactory.getBean（）），但这样做会导致bean过早实例化，从而违反标准容器生命周期。 这可能会导致负面影响，如绕过bean后处理。  另外，BeanFactoryPostProcessors是每个容器的作用域。 这只有在使用容器层次结构时才有意义。 如果您在一个容器中定义了一个BeanFactoryPostProcessor，它将只应用于该容器中的bean定义。 一个容器中的Bean定义将不会由另一个容器中的BeanFactoryPostProcessors进行后处理，即使这两个容器都是同一层次结构的一部分。 |

一个bean工厂后处理器在ApplicationContext中声明时会自动执行，以便将更改应用于定义容器的配置元数据。 Spring包含许多预定义的bean工厂后处理器，例如PropertyOverrideConfigurer和PropertyPlaceholderConfigurer。 例如，自定义BeanFactoryPostProcessor也可用于注册自定义属性编辑器。

ApplicationContext自动检测部署到其中的实现BeanFactoryPostProcessor接口的任何Bean。 它在适当的时候使用这些bean作为bean工厂后处理器。 您可以像任何其他bean一样部署这些后处理器bean。

|  |  |
| --- | --- |
|  | 和BeanPostProcessors一样，您通常不希望将BeanFactoryPostProcessors配置为延迟初始化。 如果没有其他Bean引用Bean（Factory）PostProcessor，则该后处理器根本不会被实例化。 因此，将它标记为延迟初始化将被忽略，即使您在<beans />元素的声明中将default-lazy-init属性设置为true，Bean（Factory）PostProcessor也将被急切实例化。 |

Example: the Class name substitution PropertyPlaceholderConfigurer

您可以使用PropertyPlaceholderConfigurer，通过使用标准的Java属性格式在一个单独的文件中将Bean定义的属性值进行外部化。 通过这样做，部署应用程序的人员可以自定义特定于环境的属性，如数据库URL和密码，而无需修改容器主XML定义文件或文件的复杂性或风险。

考虑以下XML-based的配置元数据片段，其中定义了包含占位符值的DataSource。 该示例显示了从外部属性文件配置的属性。 在运行时，一个PropertyPlaceholderConfigurer被应用于将取代DataSource的一些属性的元数据。 要替换的值被指定为遵循Ant / log4j / JSP EL样式的$ {property-name}形式的占位符。

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

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

</bean>

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

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

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

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

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

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

</bean>

实际值来自标准Java属性格式中的另一个文件：

jdbc.driverClassName=org.hsqldb.jdbcDriver

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

jdbc.username=sa

jdbc.password=root

因此，字符串$ {jdbc.username}在运行时被替换为值'sa'，同样适用于与属性文件中的键匹配的其他占位符值。 PropertyPlaceholderConfigurer检查大多数属性中的占位符和bean定义的属性。 此外，占位符prefix and suffix可以自定义。

通过Spring 2.5中引入的上下文命名空间，可以使用专用的配置元素来配置属性占位符。 一个或多个位置可以作为位置属性中的逗号分隔列表提供。

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

PropertyPlaceholderConfigurer不仅在您指定的属性文件中查找属性。 默认情况下，它也检查Java系统属性，如果它无法在指定的属性文件中找到属性。 您可以通过使用以下三个支持的整数值之一来设置configurer的systemPropertiesMode属性来自定义此行为：

* *never* (0): 从不检查系统属性
* *fallback* (1): 如果在指定的属性文件中不可解析请检查系统属性。这是默认设置
* *override* (2): 在尝试指定的属性文件之前，先检查系统属性。这允许系统属性覆盖任何其他属性源

有关更多信息，请查阅PropertyPlaceholderConfigurer javadocs.

|  |  |
| --- | --- |
|  | 您可以使用PropertyPlaceholderConfigurer替换类名  当您必须在运行时选择特定的实现类时，这有时很有用。 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}"/>  如果在运行时无法将类解析为有效的类，那么当它即将创建时，bean的解析失败，这是在非惰性初始化Bean的ApplicationContext的preInstantiateSingletons（）阶段期间。 |

Example: the PropertyOverrideConfigurer

PropertyOverrideConfigurer是另一个bean工厂后处理器，类似于PropertyPlaceholderConfigurer，但与后者不同，原始定义对于bean属性可以具有默认值或根本没有值。 如果重写的属性文件没有某个bean属性的条目，则使用默认的上下文定义。

请注意，bean定义并不知道被重写，所以从XML定义文件中不会立即明显地看到正在使用覆盖配置器。 在为同一个bean属性定义不同值的多个PropertyOverrideConfigurer实例的情况下，由于重载机制，最后一个获胜。

属性文件配置行采用以下格式：

beanName.property=value

例如:

dataSource.driverClassName=com.mysql.jdbc.Driver

dataSource.url=jdbc:mysql:mydb

这个示例文件可以与容器定义一起使用，该容器定义包含一个名为dataSource的bean，该bean具有driver和urlproperties。

只要路径中除最终属性被重写的每个组件都已经非空（可能由构造函数初始化），也支持复合属性名称。 在这个例子中...

foo.fred.bob.sammy=123

1. the sammy property of the bob property of the fred property of the foo bean is set to the scalar value 123.

|  |  |
| --- | --- |
|  | 指定的覆盖值总是文字值; 它们不会被翻译成bean引用。 当XML bean定义中的原始值指定一个bean引用时，这个约定也适用。 |

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

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

1.8.3. Customizing instantiation logic with a FactoryBean

为自己工厂的对象实现org.springframework.beans.factory.FactoryBean接口。

FactoryBean接口是Spring IoC容器实例化逻辑的可插入点。 如果你有复杂的初始化代码，用Java可以更好地表达，而不是（可能）冗长的XML，你可以创建自己的FactoryFactory，在该类中编写复杂的初始化代码，然后将自定义FactoryBean插入到容器中。

FactoryBean接口提供三种方法：

* Object getObject(): 返回这个工厂创建的对象的实例。这个实例可能是共享的，这取决于这个工厂是否返回singletons or prototypes
* boolean isSingleton(): 如果此 FactoryBean 返回 singletons 则返回true，否则返回,false
* Class getObjectType(): 返回由 getObject() 方法返回的对象类型，如果事先位置类型，则返回null

FactoryBean的概念和接口用于Spring框架的许多地方; FactoryBean接口的50多个实现与Spring本身一起提供。

当你需要向一个实际的FactoryBean实例本身而不是它产生的bean请求一个容器时，在调用ApplicationContext的getBean（）方法时，用＆符号（＆）作为序言。 因此，对于给定的FactoryBean，其id为myBean，在容器上调用getBean（“myBean”）将返回FactoryBean的产品; 而调用getBean（“＆myBean”）则返回FactoryBean实例本身。

1.9. Annotation-based container configuration

Are annotations better than XML for configuring Spring?

引入基于注解的配置引发了这种方法是否比XML更好的问题。 简短的答案取决于。 长久的答案是每种方法都有其优缺点，通常由开发人员决定哪种策略更适合他们。 由于它们被定义的方式，注释在其声明中提供了大量的上下文，从而导致更短，更简洁的配置。 但是，XML在接触组件时不需要触及其源代码或重新编译它们就能胜任。 一些开发人员更喜欢布线(wiring)接近源(source)，而另一些开发人员则认为注解类不再是POJO，而且配置变得分散，难以控制。

无论选择什么，Spring都可以适应两种风格，甚至可以将它们混合在一起。 值得指出的是，通过 [JavaConfig](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-java)选项，Spring允许以非侵入方式使用注释，而无需触及目标组件源代码，并且在工具方面， [Spring Tool Suite](https://spring.io/tools/sts)支持所有配置样式。

基于注释的配置提供了XML设置的替代方法，该配置依赖字节码元数据来连接组件而不是角括号声明。开发人员不是使用XML来描述bean布线，而是通过在相关的类，方法或字段声明中使用注释将配置移入组件类本身。 [Example: The RequiredAnnotationBeanPostProcessor](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-extension-bpp-examples-rabpp)将BeanPostProcessor与注释一起使用是扩展Spring IoC容器的常用方法。例如，Spring 2.0引入了使用 [@Required](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-required-annotation)注解强制执行所需属性的可能性。 Spring 2.5使得遵循相同的通用方法来驱动Spring的依赖注入成为可能。实质上，@ Autowired注释提供了与自动装配( [Autowiring collaborators](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-autowire) )合作者中描述的相同的功能，但具有更细致的控制和更广泛的适用性。 Spring 2.5还增加了对JSR-250注释的支持，如@PostConstruct和@PreDestroy。 Spring 3.0增加了对包含在javax.inject包（例如@Inject和@Named）中的JSR-330（Dependency Injection for Java）注释的支持。有关这些注释的详细信息可以在相关部分找到( [relevant section](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-standard-annotations))。

|  |  |
| --- | --- |
|  | 注解注入是在XML注入之前执行的，因此后者配置将覆盖通过两种方法连接的属性的前者。 |

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

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

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

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

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

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

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

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

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

<context:annotation-config/>

</beans>

（隐式注册的后处理器包括[AutowiredAnnotationBeanPostProcessor](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/beans/factory/annotation/AutowiredAnnotationBeanPostProcessor.html),[CommonAnnotationBeanPostProcessor](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/context/annotation/CommonAnnotationBeanPostProcessor.html), [PersistenceAnnotationBeanPostProcessor](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/orm/jpa/support/PersistenceAnnotationBeanPostProcessor.html)，以及上述[RequiredAnnotationBeanPostProcessor](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/beans/factory/annotation/RequiredAnnotationBeanPostProcessor.html)。）

|  |  |
| --- | --- |
|  | <context：annotation-config />仅在与其定义的应用程序上下文中查找bean上的注释。 这意味着，如果将DispatcherServlet的<context：annotation-config />放在WebApplicationContext中，它只会检查控制器中的@Autowired bean，而不是您的服务。 有关更多信息，请参阅 [The DispatcherServlet](https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc-servlet)。 |

1.9.1. @Required

@Required注解适用于bean属性设置器方法，如下例所示：

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

**private** MovieFinder movieFinder;

@Required

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

this.movieFinder = movieFinder;

}

*// ...*

}

这个注解简单地表明受影响的bean属性必须在配置时通过bean定义中的显式属性值或通过自动装配来填充。 如果受影响的bean属性尚未填充，容器将引发异常; 这允许急切和明确的失败，以后避免NullPointerExceptions等。 仍然建议您将断言放入bean类本身，例如，放入init方法中。 这样做即使在容器外部使用该类时也会强制执行那些必需的引用和值。

1.9.2. @Autowired

|  |  |
| --- | --- |
|  | 在下面的例子中，JSR 330的@Inject注解可以用来代替Spring的@Autowired注解。 详情请看 [here](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-standard-annotations)。 |

您可以将@Autowired注释应用于构造函数：

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

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

@Autowired

**public** MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {

this.customerPreferenceDao = customerPreferenceDao;

}

*// ...*

}

|  |  |
| --- | --- |
|  | 从Spring Framework 4.3开始，如果目标bean只定义了一个构造函数，那么这种构造函数上的@Autowired注释就不再需要了。 但是，如果有几个构造函数可用，则必须至少注明一个构造函数来教授容器使用哪一个。 |

正如所料，您还可以将@Autowired注释应用于"traditional"设置器方法：

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

**private** MovieFinder movieFinder;

@Autowired

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

this.movieFinder = movieFinder;

}

*// ...*

}

您还可以将注释应用于具有任意名称和/或多个参数的方法：

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

**private** MovieCatalog movieCatalog;

**private** CustomerPreferenceDao customerPreferenceDao;

@Autowired

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

CustomerPreferenceDao customerPreferenceDao) {

this.movieCatalog = movieCatalog;

this.customerPreferenceDao = customerPreferenceDao;

}

*// ...*

}

您也可以将@Autowired应用于字段，甚至可以将其与构造函数混合使用：

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

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

@Autowired

**private** MovieCatalog movieCatalog;

@Autowired

**public** MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {

this.customerPreferenceDao = customerPreferenceDao;

}

*// ...*

}

|  |  |
| --- | --- |
|  | 确保您的目标组件（例如MovieCatalog，CustomerPreferenceDao）由用于@ Autowired注释的注入点的类型一致地声明。 否则由于在运行时找不到类型匹配，注入可能会失败。  对于通过类路径扫描找到的XML定义的bean或组件类，容器通常会预先知道具体类型。 但是，对于@Bean工厂方法，您需要确保声明的返回类型具有足够的表达力。 对于实现多个接口的组件或可能由其实现类型引用的组件，请考虑在工厂方法中声明最具体的返回类型（至少按照注入点对bean引用的要求）。 |

通过将注释添加到需要该类型数组的字段或方法，可以从ApplicationContext提供特定类型的所有Bean：

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

@Autowired

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

*// ...*

}

The same applies for typed collections:

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

**private** Set<MovieCatalog> movieCatalogs;

@Autowired

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

this.movieCatalogs = movieCatalogs;

}

*// ...*

}

|  |  |
| --- | --- |
|  | 您的目标bean可以实现org.springframework.core.Ordered接口，或者如果希望数组或列表中的项目按特定顺序排序，则使用@Order或标准@Priority注释。 否则，他们的订单将遵循容器中相应目标bean定义的注册顺序。  @Order注释可以在目标类层次上声明，也可以在@Bean方法中声明，可能每个bean定义都是非常单独的（在具有相同bean类的多个定义的情况下）。 @订单值可能会影响注入点的优先级，但请注意它们不影响单身启动顺序，这是由依赖关系和@DependsOn声明确定的正交关系。  请注意，标准的javax.annotation.Priority注解在@Bean级别不可用，因为它不能在方法上声明。 它的语义可以通过@Order值和@Primaryon来为每个类型的单个bean建模。 |

只要预期的键类型是字符串，即使键入的Map也可以自动装配。 Map值将包含期望类型的所有bean，并且键将包含相应的bean名称：

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

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

@Autowired

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

this.movieCatalogs = movieCatalogs;

}

*// ...*

}

默认情况下，只要零候选bean可用，自动装配失败; 默认行为是将注释的方法，构造函数和字段视为指示所需的依赖项。 这种行为可以改变，如下所示。

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

**private** MovieFinder movieFinder;

@Autowired(required = false)

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

this.movieFinder = movieFinder;

}

*// ...*

}

|  |  |
| --- | --- |
|  | 每个类只有一个带注释的构造函数可以标记为必需，但可以注释多个不需要的构造函数。 在这种情况下，每个人都被视为候选人，而Spring使用贪婪的构造函数(*greediest*constructor)，它的依赖关系可以得到满足，也就是具有最多参数的构造函数。  建议使用@Required注解来标记@Autowired的必需属性。 必需的属性表示该属性不是自动装配所需的，如果该属性不能自动装配，则该属性将被忽略。 另一方面，@Required更强大，因为它强制执行由集装箱支持的任何方式设置的财产。 如果没有值被注入，则会引发相应的异常。 |

或者，您可以通过Java 8的java.util.Optional表达特定依赖项的非必需性质：

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

@Autowired

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

...

}

}

从Spring Framework 5.0开始，您还可以使用@Nullable注释（任何包中的任何类型，例如来自JSR-305的javax.annotation.Nullable）：

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

@Autowired

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

...

}

}

您还可以使用@Autowired作为众所周知的可解析依赖项的接口：BeanFactory，ApplicationContext，Environment，ResourceLoader，ApplicationEventPublisher和MessageSource。 这些接口及其扩展接口（如ConfigurableApplicationContext或ResourcePatternResolver）会自动解析，无需进行特殊设置。

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

@Autowired

**private** ApplicationContext context;

**public** MovieRecommender() {

}

*// ...*

}

|  |  |
| --- | --- |
|  | @Autowired，@Inject，@Resource和@Value注解由Spring BeanPostProcessor实现来处理，这意味着您不能在自己的BeanPostProcessor或BeanFactoryPostProcessor类型（如果有）中应用这些注释。 这些类型必须通过XML或使用Spring @Bean方法明确“wired up”。 |

1.9.3. Fine-tuning annotation-based autowiring with @Primary

由于按类型自动装配可能会导致多个候选人，因此通常需要对选择过程有更多的控制权。 一种方法是使用Spring的@Primary注解。 @Primary表示当多个bean被自动装配到单值依赖项时，应该给予一个特定的bean优先。 如果候选人中只有一个'primary' bean，它将是自动装配的值。

假设我们有以下配置，将firstMovieCatalog定义为主MovieCatalog。

@Configuration

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

@Bean

**@Primary**

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

@Bean

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

*// ...*

}

通过这种配置，下面的MovieRecomder将通过firstMovieCatalog自动装配。

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

@Autowired

**private** MovieCatalog movieCatalog;

*// ...*

}

相应的bean定义如下所示。

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

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

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

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

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

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

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

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

<context:annotation-config/>

<bean class="example.SimpleMovieCatalog" primary="true">

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

</bean>

<bean class="example.SimpleMovieCatalog">

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

</bean>

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

</beans>

1.9.4. Fine-tuning annotation-based autowiring with qualifiers

@Primary是一种有效的方法，可以在确定一个主要候选人时使用多个实例的类型自动装配。 当需要对选择过程进行更多控制时，可以使用Spring的@Qualifier注释。 您可以将限定符值与特定参数相关联，缩小匹配类型的集合，以便为每个参数选择特定的bean。 在最简单的情况下，这可以是一个简单的描述性值：

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

@Autowired

**@Qualifier("main")**

**private** MovieCatalog movieCatalog;

*// ...*

}

@Qualifier注解也可以在单独的构造函数参数或方法参数中指定：

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

**private** MovieCatalog movieCatalog;

**private** CustomerPreferenceDao customerPreferenceDao;

@Autowired

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

CustomerPreferenceDao customerPreferenceDao) {

this.movieCatalog = movieCatalog;

this.customerPreferenceDao = customerPreferenceDao;

}

*// ...*

}

相应的bean定义如下所示。 具有限定符值“main”的bean与用相同值限定的构造函数参数连线。

<?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>

对于回退匹配，bean名称被视为默认限定符值。 因此，您可以使用id“main”而不是嵌套的限定符元素来定义bean，从而得到相同的匹配结果。 但是，虽然可以使用此惯例通过名称引用特定的bean，但@Autowired基本上是可选的语义限定符的类型驱动注入。 这意味着限定符值（即使使用bean名称后备）也会在匹配类型集合内缩小语义; 它们不会在语义上表达对唯一bean id的引用。 良好的限定符值是“main”或“EMEA”或“persistent”，表示与bean id无关的特定组件的特征，如果使用匿名bean定义（如上例中的匿名bean定义）。

限定符也适用于键入的集合，如上所述，例如，Set <MovieCatalog>。 在这种情况下，根据声明的限定符将所有匹配的bean作为集合注入。 这意味着限定词不必是唯一的; 它们只是构成过滤标准。 例如，您可以使用相同的限定符值“action”定义多个MovieCatalog bean，所有这些都将注入到由@Qualifier（“action”）注释的Set <MovieCatalog>中。

|  |  |
| --- | --- |
|  | 在类型匹配的候选者中，针对目标bean名称选择限定符值，在注入点甚至不需要@Qualifier注释。 如果没有其他解析指示符（例如限定符或主标记），那么对于非唯一的依赖性情况，Spring将匹配注入点名称（即字段名称或参数名称）与目标bean名称，并选择相同的名称， 命名候选人，如果有的话。  也就是说，如果您打算按名称表示注解驱动的注入，则不要主要使用@Autowired，即使能够在类型匹配的候选中按bean名称进行选择。 相反，请使用JSR-250 @Resourceannotation，它的语义定义是通过其唯一名称来标识特定的目标组件，而声明的类型与匹配过程无关。 @Autowired具有相当不同的语义：在按类型选择候选bean之后，指定的字符串限定符值将仅在这些类型选择的候选者中被考虑，例如， 将“account”限定符与标有相同限定符标签的bean相匹配。  对于本身被定义为集合/映射或数组类型的bean，@Resource是一个很好的解决方案，通过唯一名称引用特定集合或数组bean。 也就是说，只要元素类型信息保存在@Bean返回类型签名或集合继承层次结构中，就可以通过Spring的@Autowired类型匹配算法来匹配集合/映射和数组类型。 在这种情况下，可以使用限定符值在相同类型的集合中进行选择，如前一段所述。  从4.3开始，@Autowired还考虑自引用注入，即引用回当前注入的bean。 请注意，自我注入是后备; 对其他组件的正常依赖关系始终具有优先权。 从这个意义上说，自我引用不参与正规的候选人选择，因此尤其不是主要的; 相反，它们总是以最低优先权结束。 在实践中，仅使用自引用作为最后的手段，例如， 用于通过bean的事务代理调用同一实例上的其他方法：考虑在这种情况下将受影响的方法分解为单独的委托bean。 或者，使用@Resource可以通过其唯一名称获取代理返回到当前bean。  @Autowired适用于字段，构造函数和多参数方法，允许通过参数级别的限定符注释进行缩小。 相比之下，@Resource仅支持具有单个参数的字段和bean属性设置方法。 因此，如果注入目标是构造函数或多参数方法，则坚持使用限定符。 |

您可以创建自己的自定义限定符注释。 只需定义一个注释并在您的定义中提供@Qualifier注释：

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

@Retention(RetentionPolicy.RUNTIME)

**@Qualifier**

**public** @interface Genre {

String value();

}

然后，您可以在自动布线字段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;

}

*// ...*

}

接下来，提供候选bean定义的信息。 您可以将<qualifier />标记添加为<bean />标记的子元素，然后指定类型和值以匹配您的自定义限定符注释。 该类型与注释的完全限定类名相匹配。 或者，如果没有相互冲突名称存在的风险，为了方便起见，您可以使用短名称。 以下示例演示了这两种方法。

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

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

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

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

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

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

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

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

<context:annotation-config/>

<bean class="example.SimpleMovieCatalog">

**<qualifier type="Genre" value="Action"/>**

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

</bean>

<bean class="example.SimpleMovieCatalog">

**<qualifier type="example.Genre" value="Comedy"/>**

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

</bean>

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

</beans>

在 [Classpath scanning and managed components](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-classpath-scanning)，您将看到一种基于注释的替代方法来提供XML中的限定符元数据。 具体来说，请参阅 [Providing qualifier metadata with annotations](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-scanning-qualifiers)。

在某些情况下，使用没有值的注释可能就足够了。 当注释提供更通用的用途并且可以应用于多种不同类型的依赖关系时，这可能很有用。 例如，您可以提供一个offlinecatalog，当没有互联网连接时可以搜索。 首先定义简单的注释：

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

@Retention(RetentionPolicy.RUNTIME)

@Qualifier

**public** @interface Offline {

}

然后将注释添加到要自动装配的字段或属性中：

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

@Autowired

**@Offline**

**private** MovieCatalog offlineCatalog;

*// ...*

}

现在，bean定义只需要一个限定符类型：

<bean class="example.SimpleMovieCatalog">

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

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

</bean>

您还可以定义接受命名属性的自定义限定符注释，而不是简单的value属性。 如果在要自动装配的字段或参数上指定了多个属性值，则bean定义必须匹配allsuch属性值才能被视为自动导向候选项。 作为示例，请考虑以下注释定义：

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

@Retention(RetentionPolicy.RUNTIME)

@Qualifier

**public** @interface MovieQualifier {

String genre();

Format format();

}

在这种情况下，Format是一个枚举：

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

VHS, DVD, BLURAY

}

要自动装配的字段使用自定义限定符进行注释，并包含两个属性的值：genre and format。

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

@Autowired

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

**private** MovieCatalog actionVhsCatalog;

@Autowired

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

**private** MovieCatalog comedyVhsCatalog;

@Autowired

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

**private** MovieCatalog actionDvdCatalog;

@Autowired

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

**private** MovieCatalog comedyBluRayCatalog;

*// ...*

}

最后，bean定义应该包含匹配的限定符值。 这个例子还演示了bean meta属性可以用来代替<qualifier />子元素。 如果可用，则<qualifier />及其属性优先，但如果不存在此类限定符，则自动装配机制将回退到<meta />标记中提供的值，如以下示例中的最后两个bean定义

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

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

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

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

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

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

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

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

<context:annotation-config/>

<bean class="example.SimpleMovieCatalog">

<qualifier type="MovieQualifier">

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

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

</qualifier>

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

</bean>

<bean class="example.SimpleMovieCatalog">

<qualifier type="MovieQualifier">

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

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

</qualifier>

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

</bean>

<bean class="example.SimpleMovieCatalog">

<meta key="format" value="DVD"/>

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

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

</bean>

<bean class="example.SimpleMovieCatalog">

<meta key="format" value="BLURAY"/>

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

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

</bean>

</beans>

1.9.5. Using generics(泛型) as autowiring qualifiers

除了@Qualifier注释之外，还可以使用Java通用类型作为隐式形式的限定条件。 例如，假设您有以下配置：

@Configuration

**public** **class** **MyConfiguration** {

@Bean

**public** StringStore stringStore() {

**return** **new** StringStore();

}

@Bean

**public** IntegerStore integerStore() {

**return** **new** IntegerStore();

}

}

假设上面的Bean实现了一个通用接口，即Store <String>和Store <Integer>，您可以@Autowire Store接口和泛型将用作限定符：

@Autowired

**private** Store<String> s1; *// <String> qualifier, injects the stringStore bean*

@Autowired

**private** Store<Integer> s2; *// <Integer> qualifier, injects the integerStore bean*

自动装配列表，地Maps and Arrays时也适用通用限定符：

*// Inject all Store beans as long as they have an <Integer> generic*

*// Store<String> beans will not appear in this list*

@Autowired

**private** List<Store<Integer>> s;

1.9.6. CustomAutowireConfigurer

[CustomAutowireConfigurer](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/beans/factory/annotation/CustomAutowireConfigurer.html) 是一个BeanFactoryPostProcessor，它使您可以注册自己的自定义限定符注释类型，即使它们没有使用Spring的@Qualifier注释进行注释。

<bean id="customAutowireConfigurer"

class="org.springframework.beans.factory.annotation.CustomAutowireConfigurer">

<property name="customQualifierTypes">

<set>

<value>example.CustomQualifier</value>

</set>

</property>

</bean>

AutowireCandidateResolver通过以下方式确定自动装配候选者：

* 每个bean定义的 autowire-candidate值
* <beans/>元素上的人户默认自动装在候选模式
* 存在@Qualifier批注和通过CustomAutowireConfigurer 注册的任何自定义批注

当多个bean有资格作为autowire候选者时，"primary"的确定如下所示：如果候选者中恰好有一个bean定义的主属性设置为true，则它将被选中。

1.9.7. @Resource

Spring还支持使用字段上的JSR-250 @Resource批注或bean属性设置器方法进行注入。 这是Java EE 5和6中的常见模式，例如在JSF 1.2托管bean或JAX-WS 2.0端点中。 Spring也支持Spring管理对象的这种模式。

@Resource接受一个name属性，默认情况下，Spring将该值解释为要注入的bean名称。 换句话说，它遵循名义语义，如本例所示：

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

**private** MovieFinder movieFinder;

**@Resource(name="myMovieFinder")**

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

this.movieFinder = movieFinder;

}

}

如果没有明确指定名称，则默认名称是从字段名称或设置方法派生的。 如果是字段，则需要字段名称; 在setter方法的情况下，它采用bean属性名称。 所以下面的例子将把名为“movieFinder”的bean注入到它的setter方法中：

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

**private** MovieFinder movieFinder;

**@Resource**

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

this.movieFinder = movieFinder;

}

}

|  |  |
| --- | --- |
|  | 与注释一起提供的名称通过CommonContentAnnotationBeanPostProcessor所了解的ApplicationContext被解析为bean名称。 如果您明确配置Spring的[SimpleJndiBeanFactory](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/jndi/support/SimpleJndiBeanFactory.html) ，可以通过JNDI解析名称。 但是，建议您依赖默认行为并简单地使用Spring的JNDI查找功能来保留间接级别。 |

在@Resource用法中，没有明确指定名称，类似于@Autowired，@Resource查找主类型匹配而不是特定的命名bean，并且解析了众所周知的可解析依赖关系：BeanFactory，ApplicationContext，ResourceLoader，ApplicationEventPublisher， 和MessageSource接口。

因此，在以下示例中，customerPreferenceDao字段首先查找名为customerPreferenceDao的bean，然后返回到CustomerPreferenceDao类型的主类型匹配。 “context”字段是基于已知可解析依赖类型ApplicationContext注入的。

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

@Resource

**private** CustomerPreferenceDao customerPreferenceDao;

@Resource

**private** ApplicationContext context;

**public** MovieRecommender() {

}

*// ...*

}

1.9.8. @PostConstruct and @PreDestroy

CommonAnnotationBeanPostProcessor不仅识别@Resource注释，还识别JSR-250生命周期注释。 在Spring 2.5中引入的对这些注释的支持为[initialization callbacks](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-lifecycle-initializingbean) and [destruction callbacks](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-lifecycle-disposablebean)中描述的那些提供了另一种替代方案。 如果CommonAnnotationBeanPostProcessor在Spring ApplicationContext中注册，则在生命周期的相同时间点调用携带其中一个注释的方法，该方法与生命周期中相应的Spring生命周期接口方法或显式声明的回调方法相同。 在下面的示例中，缓存将在初始化时预填充，并在销毁时清除。

**public** **class** **CachingMovieLister** {

@PostConstruct

**public** **void** populateMovieCache() {

*// populates the movie cache upon initialization...*

}

@PreDestroy

**public** **void** clearMovieCache() {

*// clears the movie cache upon destruction...*

}

}

|  |  |
| --- | --- |
|  | 有关组合各种生命周期机制的效果的详细信息，请参阅 [Combining lifecycle mechanisms](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-lifecycle-combined-effects)。 |

1.10. Classpath scanning and managed components

本章中的大多数示例都使用XML来指定在Spring容器中生成每个BeanDefinition的配置元数据。 上一节([Annotation-based container configuration](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-annotation-config))演示了如何通过源代码级注释提供大量配置元数据。 但是，即使在这些示例中，"base"bean定义在XML文件中显式定义，而注释仅驱动依赖注入。 本节介绍通过扫描类路径隐式检测候选组件的选项。 候选组件是与过滤条件相匹配的类，并具有在容器中注册的相应的bean定义。 这消除了使用XML来执行bean注册的需要; 相反，您可以使用注释（例如@Component），AspectJ类型表达式或您自己的自定义过滤条件来选择哪些类将具有注册到容器的bean定义。

|  |  |
| --- | --- |
|  | 从Spring 3.0开始，Spring JavaConfig项目提供的许多功能都是核心Spring框架的一部分。 这使您可以使用Java定义bean，而不是使用传统的XML文件。 查看@Configuration，@Bean，@Import和@DependsOn注释以获取如何使用这些新功能的示例。 |

1.10.1. @Component and further stereotype annotations

@Repository注释是任何实现存储库（也称为数据访问对象或DAO）角色或构造型的类的标记。 这种标记的用法是 [Exception translation](https://docs.spring.io/spring/docs/current/spring-framework-reference/data-access.html#orm-exception-translation)中所述的自动翻译异常。

Spring提供了更多的构造型注释：@Component，@Service和@Controller。 @Component是任何Spring管理组件的通用构造型。对于更具体的用例，@Repository，@Service和@Controller是@Component的特化，例如，分别在持久层，服务和表示层中。因此，您可以使用@Component注释您的组件类，但通过使用@Repository，@Service或@Controller注释它们，您的类更适合通过工具进行处理或与方面相关联。例如，这些刻板印象注解为切入点提供了理想的目标。 @Repository，@Service和@Controller也可能在Spring Framework的未来版本中携带额外的语义。因此，如果您在为服务层使用@Component或@Service之间进行选择，@Service显然是更好的选择。同样，如上所述，已经支持@Repository作为持久层自动异常转换的标记。

1.10.2. Meta-annotations

Spring提供的许多注释可以在您自己的代码中用作元注释。 元注释只是一个可以应用于其他注释的注释。 例如，上面提到的@Service注释使用@Component进行元注释：

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

**@Component** *// Spring will see this and treat @Service in the same way as @Component*

**public** @interface Service {

*// ....*

}

Meta-annotations也可以组合起来创建组合注释。 例如，Spring MVC的@RestController注释由@Controller和@ResponseBody组成。

另外，组合的注释可以可选地重新声明来自元注释的属性以允许用户定制。 当您只想暴露元注释属性的子集时，这可能特别有用。 例如，Spring的@SessionScope注解将范围名称hardcodes为会话，但仍允许定制proxyMode。

@Target({ElementType.TYPE, ElementType.METHOD})

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Scope(WebApplicationContext.SCOPE\_SESSION)

**public** @interface SessionScope {

*/\*\**

*\* Alias for {@link Scope#proxyMode}.*

*\* <p>Defaults to {@link ScopedProxyMode#TARGET\_CLASS}.*

*\*/*

@AliasFor(annotation = Scope.class)

ScopedProxyMode proxyMode() **default** ScopedProxyMode.TARGET\_CLASS;

}

然后可以在不声明proxyMode的情况下使用@SessionScope，如下所示：

@Service

**@SessionScope**

**public** **class** **SessionScopedService** {

*// ...*

}

或者为proxyMode重写一个值，如下所示：

@Service

**@SessionScope(proxyMode = ScopedProxyMode.INTERFACES)**

**public** **class** **SessionScopedUserService** **implements** UserService {

*// ...*

}

有关更多详细信息，请参阅 [Spring Annotation Programming Model](https://github.com/spring-projects/spring-framework/wiki/Spring-Annotation-Programming-Model)wiki页面。

1.10.3. Automatically detecting classes and registering bean definitions

Spring可以自动检测定型类，并使用ApplicationContext注册相应的BeanDefinitions。 例如，以下两个类适用于这种自动检测：

@Service

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

**private** MovieFinder movieFinder;

@Autowired

**public** SimpleMovieLister(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

}

@Repository

**public** **class** **JpaMovieFinder** **implements** MovieFinder {

*// implementation elided for clarity*

}

要自动检测这些类并注册相应的bean，需要将@ComponentScan添加到您的@Configuration类，其中basePackages属性是这两个类的常见父级包。 （或者，您可以指定包含每个类的父包的以comma/semicolon/space-separated分隔的列表。）

@Configuration

@ComponentScan(basePackages = "org.example")

**public** **class** **AppConfig** {

...

}

|  |  |
| --- | --- |
|  | 为简洁起见，上面可能使用了注释的值属性，即@ComponentScan（“org.example”） |

以下是使用XML的替代方法

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

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

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

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

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

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

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

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

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

</beans>

|  |  |
| --- | --- |
|  | 使用<context：component-scan>隐式启用<context：annotation-config>的功能。 当使用<context：component-scan>时，通常不需要包含<context：annotation-config>元素。 |
|  | 类路径包的扫描要求类路径中存在相应的目录条目。 在使用Ant构建JAR时，请确保不要激活JAR任务的仅文件开关。 此外，在某些环境中，类路径目录可能不会基于安全策略暴露，例如， JDK 1.7.0\_45及更高版本上的独立应用程序（需要在您的清单中设置“Trusted-Library”;请参阅[http://stackoverflow.com/questions/19394570/java-jre-7u45-breaks-classloader-getresources](https://stackoverflow.com/questions/19394570/java-jre-7u45-breaks-classloader-getresources)）。  在JDK 9的模块路径（Jigsaw）中，Spring的类路径扫描一般按预期工作。 但是，请确保您的组件类在模块信息描述符中导出; 如果您希望Spring调用类的非公共成员，请确保它们是“打开”的（即在模块信息描述符中使用打开声明而不是导出声明）。 |

此外，当您使用组件扫描元素时，AutowiredAnnotationBeanPostProcessor和CommonAnnotationBeanPostProcessor都会隐式包含。 这意味着这两个组件都是自动检测和连接在一起的 - 所有这些都没有以XML提供的任何bean配置元数据。

|  |  |
| --- | --- |
|  | 您可以通过将annotation-config属性的值设置为false来禁用AutowiredAnnotationBeanPostProcessor和CommonAnnotationBeanPostProcessor的注册。 |

1.10.4. Using filters to customize scanning

默认情况下，使用@Component，@Repository，@Service，@Controller注释的类或者本身使用@Component注释的自定义注释是唯一检测到的候选组件。 但是，只需应用自定义过滤器即可修改和扩展此行为。 将它们添加为@ComponentScan注释的includeFilters或excludeFilters参数（或者作为component-scan元素的include-filter或exclude-filter子元素）。 每个过滤器元素都需要类型和表达式属性。 下表介绍了过滤选项。

| *Table 5. Filter Types* | | |
| --- | --- | --- |
| **Filter Type** | **Example Expression** | **Description** |
| annotation (default) | org.example.SomeAnnotation | An annotation to be present at the type level in target components. |
| assignable | org.example.SomeClass | A class (or interface) that the target components are assignable to (extend/implement). |
| aspectj | org.example..\*Service+ | An AspectJ type expression to be matched by the target components. |
| regex | org\.example\.Default.\* | A regex expression to be matched by the target components class names. |
| custom | org.example.MyTypeFilter | A custom implementation of the org.springframework.core.type .TypeFilter interface. |

以下示例显示了忽略所有@Repository注释并使用"stub"存储库的配置。

@Configuration

@ComponentScan(basePackages = "org.example",

includeFilters = @Filter(type = FilterType.REGEX, pattern = ".\*Stub.\*Repository"),

excludeFilters = @Filter(Repository.class))

**public** **class** **AppConfig** {

...

}

和使用XML的等价物

<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>

|  |  |
| --- | --- |
|  | 您还可以通过在注释中设置useDefaultFilters = false或提供use-default-filters =“false”作为<component-scan />元素的属性来禁用默认过滤器。 这将实际上禁用自动检测用@Component，@Repository，@Service，@Controller或@Configuration注解的类。 |

1.10.5. Defining bean metadata within components

Spring组件也可以将bean定义元数据提供给容器。 您可以使用相同的@Bean注释来定义@Configuration注释类中的bean元数据。 这是一个简单的例子：

@Component

**public** **class** **FactoryMethodComponent** {

@Bean

@Qualifier("public")

**public** TestBean publicInstance() {

**return** **new** TestBean("publicInstance");

}

**public** **void** doWork() {

*// Component method implementation omitted*

}

}

该类是一个Spring组件，它的doWork()方法中包含了特定于应用程序的代码。 但是，它也提供了一个bean定义，该定义具有引用publicInstance()方法的工厂方法。 @Bean注释通过@Qualifier注释来标识工厂方法和其他bean定义属性，例如限定符值。 其他可以指定的方法级别注释是@Scope，@Lazy和自定义限定符注释。

|  |  |
| --- | --- |
|  | 除了用于组件初始化的角色外，@Lazy注释还可以放置在标有@Autowired或@Inject的注入点上。 在这种情况下，它导致注入一个懒惰的解析代理。 |

如前所述，支持自动装配的字段和方法，并支持自动装配@Bean方法：

@Component

**public** **class** **FactoryMethodComponent** {

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

@Bean

@Qualifier("public")

**public** TestBean publicInstance() {

**return** **new** TestBean("publicInstance");

}

*// use of a custom qualifier and autowiring of method parameters*

@Bean

**protected** TestBean protectedInstance(

@Qualifier("public") TestBean spouse,

@Value("#{privateInstance.age}") String country) {

TestBean tb = **new** TestBean("protectedInstance", 1);

tb.setSpouse(spouse);

tb.setCountry(country);

**return** tb;

}

@Bean

**private** TestBean privateInstance() {

**return** **new** TestBean("privateInstance", i++);

}

@Bean

@RequestScope

**public** TestBean requestScopedInstance() {

**return** **new** TestBean("requestScopedInstance", 3);

}

}

该示例将String方法参数country自动装入另一个名为privateInstance的bean的Age属性的值。 Spring表达式语言元素通过符号#{ <expression> }定义属性的值。 对于@Value注释，表达式解析器预先配置为在解析表达式文本时查找bean名称。

从Spring Framework 4.3开始，您还可以声明InjectionPoint类型的工厂方法参数（或其更具体的子类DependencyDescriptor），以访问触发创建当前bean的请求注入点。 请注意，这只适用于实际创建的bean实例，而不适用于注入现有实例。 因此，对于原型范围的bean来说，这个特性最有意义。 对于其他作用域，factory方法只会看到触发在给定范围内创建新bean实例的注入点：例如，触发创建惰性单例bean的依赖关系。 在这种情况下使用提供的注入点元数据和语义保护。

@Component

**public** **class** **FactoryMethodComponent** {

@Bean @Scope("prototype")

**public** TestBean prototypeInstance(InjectionPoint injectionPoint) {

**return** **new** TestBean("prototypeInstance for " + injectionPoint.getMember());

}

}

常规Spring组件中的@Bean方法的处理方式与Spring @Configuration类中的对应方法不同。 不同之处在于，@Component类不会使用CGLIB来拦截方法和字段的调用。 CGLIB代理是通过@Configurationclasses中的@Bean方法中的调用方法或字段为合作对象创建bean元数据引用的手段; 这种方法不是用普通的Java语义来调用，而是通过容器来提供Spring bean的通常的生命周期管理和代理，即使通过对@Bean方法的编程调用来引用其他bean。 相比之下，在普通的@Component类中调用@Beanmethod中的方法或字段具有标准的Java语义，不需要应用特殊的CGLIB处理或其他约束。

|  |  |
| --- | --- |
|  | 您可以将@Bean方法声明为静态方法，允许在不创建其包含的配置类作为实例的情况下调用它们。 当定义后处理器豆时，这是特别有意义的，例如 BeanFactoryPostProcessor或BeanPostProcessor类型，因为这些bean将在容器生命周期的早期初始化，并且应避免在此时触发配置的其他部分。  请注意，对静态@Bean方法的调用永远不会被容器拦截，即使在@Configuration类中也是如此（参见上文）。 这是由于技术限制：CGLIB子类只能覆盖非静态方法。 因此，直接调用另一个@Bean方法将具有标准Java语义，从而导致独立实例从工厂方法本身直接返回。  @Bean方法的Java语言可见性不会立即影响Spring容器中的结果bean定义。 你可以自由地声明你的工厂方法，就像你在非@配置类中看到的那样，也可以在任何地方用静态方法。 然而，@Configuration类中的常规@Bean方法需要被覆盖，即不能将它们声明为private或final。  @Bean方法也将在给定组件或配置类的基类上以及在由组件或配置类实现的接口中声明的Java 8默认方法上发现。 这为构建复杂的配置安排提供了很大的灵活性，从Spring 4.2起，通过Java 8默认方法甚至可以实现多重继承。  最后，请注意，单个类可能会为同一个bean保存多个@Bean方法，因为要根据运行时可用的依赖关系来使用多个工厂方法。 这与在其他配置方案中选择"greediest"构造函数或工厂方法的算法相同：在构建时将选择具有最大可满足依赖项数的变体，类似于容器在多个@Autowired构造函数之间进行选择的方式。 |

1.10.6. Naming autodetected components

当一个组件作为扫描进程的一部分被自动检测时，它的bean名称由该扫描器已知的BeanNameGeneratorstrategy生成。 默认情况下，包含名称值的任何Spring构造型注释（@Component，@ Repository，@Service和@ Controller）都会将该名称提供给相应的bean定义。

如果这样的注释不包含任何名称值或其他任何检测到的组件（例如自定义过滤器发现的那些），那么默认的bean名称生成器将返回未注册的非限定类名称。 例如，如果检测到以下两个组件，则名称将为myMovieLister和movieFinderImpl：

@Service("myMovieLister")

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

*// ...*

}

@Repository

**public** **class** **MovieFinderImpl** **implements** MovieFinder {

*// ...*

}

|  |  |
| --- | --- |
|  | 如果你不想依赖默认的bean命名策略，你可以提供一个自定义的bean命名策略。 首先，实现[BeanNameGenerator](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/beans/factory/support/BeanNameGenerator.html) 接口，并确保包含默认的无参数构造函数。 然后，在配置扫描仪时提供完全合格的类名称： |

@Configuration

@ComponentScan(basePackages = "org.example", nameGenerator = MyNameGenerator.class)

**public** **class** **AppConfig** {

...

}

<beans>

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

name-generator="org.example.MyNameGenerator" />

</beans>

作为一般规则，当其他组件可能正在对其进行明确引用时，请考虑在注释中指定名称。 另一方面，只要容器负责布线，自动生成的名称就足够了。

1.10.7. Providing a scope for autodetected components

与一般的Spring管理组件一样，自动检测组件的默认和最常见的作用域是单例。 但是，有时您需要一个可以通过@Scope注释指定的不同范围。 只需在注释中提供范围的名称即可：

@Scope("prototype")

@Repository

**public** **class** **MovieFinderImpl** **implements** MovieFinder {

*// ...*

}

有关特定于Web的范围的详细信息，请参阅 [Request, session, application, and WebSocket scopes](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-scopes-other)

|  |  |
| --- | --- |
|  | 腰围范围解析提供自定义策略而不是依赖基于注解的方法，请实现[ScopeMetadataResolver](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/context/annotation/ScopeMetadataResolver.html) 接口，并确保包含默认的无参构造函数。然后在配置扫描是提供完整合格的类名称： |

@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>

在使用某些non-singleton作用域时，可能需要为作用域对象生成代理。原因在 [Scoped beans as dependencies](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-scopes-other-injection) 。为此，组件扫描元素上提供了*scoped-proxy* 属性。三个可能的值是:no,interfaces和targetClass。例如，以下配置将导致标准的JDK动态代理:

@Configuration

@ComponentScan(basePackages = "org.example", scopedProxy = ScopedProxyMode.INTERFACES)

**public** **class** **AppConfig** {

...

}

<beans>

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

scoped-proxy="interfaces" />

</beans>

1.10.8. Providing qualifier metadata with annotations

@Qualifier 注解在 [Fine-tuning annotation-based autowiring with qualifiers](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-autowired-annotation-qualifiers) 中进行讨论。该部分中的示例演示了在解析自动导向候选时，使用@Qualifier 注解和自定义限定符注解来提供细粒度控制。由于这些示例基于XML bean定义，因此限定符元数据是使用XML中bean元素的限定符或元子元素在候选bean定义上提供的。当依靠类路径扫描来自动检测组件时，你可以在候选类上提供限定符元数据和类型级别注解。以下三个示例演示了这种技术：

@Component

**@Qualifier("Action")**

**public** **class** **ActionMovieCatalog** **implements** MovieCatalog {

*// ...*

}

@Component

**@Genre("Action")**

**public** **class** **ActionMovieCatalog** **implements** MovieCatalog {

*// ...*

}

@Component

**@Offline**

**public** **class** **CachingMovieCatalog** **implements** MovieCatalog {

*// ...*

}

|  |  |
| --- | --- |
|  | 与大多数基于注解的替代方案一样，请记住，注释元数据绑定到类定义本身，而XML的使用允许相同类型的多个bean提供其限定符元数据的变体，因为元数据是per-instance而不是per-class。 |

1.10.9. Generating an index of candidate components

虽然类路径扫描速度非常快，但通过在编译时创建候选静态列表，可以提高大型应用程序的启动性能。 在这种模式下，应用程序的所有模块都必须使用这种机制，因为当ApplicationContext检测到这样的索引时，它将自动使用它而不是扫描类路径。

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附加依赖项即可：

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context-indexer</artifactId>

<version>5.0.3.RELEASE</version>

<optional>true</optional>

</dependency>

</dependencies>

Or, using Gradle:

dependencies {

compileOnly("org.springframework:spring-context-indexer:5.0.3.RELEASE")

}

该进程将生成一个将包含在jar中的META-INF/spring.components文件。

|  |  |
| --- | --- |
|  | 在IDE中使用此模式时，必须将spring-context-indexer注册为注释处理器，以确保在更新候选组件时索引是最新的。 |
|  | 当在类路径上找到META-INF/spring.components时，索引会自动启用。 如果索引对于某些库（或用例）部分可用，但无法为整个应用程序构建，则可以通过将s spring.index.ignoreto设置为常规类路径安排（即根本没有索引）。 true，可以是系统属性，也可以是类路径根目录下的s spring.properties文件。 |

1.11. Using JSR 330 Standard Annotations

从Spring 3.0开始，Spring提供对JSR-330标准注释（依赖注入）的支持。 这些注释以与Spring注释相同的方式进行扫描。 你只需要在你的类路径中有相关的jar。

|  |  |
| --- | --- |
|  | 如果您使用的是Maven，则标准Maven存储库中可以使用javax.inject工件（[http://repo1.maven.org/maven2/javax/inject/javax.inject/1/](https://repo1.maven.org/maven2/javax/inject/javax.inject/1/)）。 您可以将以下依赖项添加到您的文件pom.xml中：  <dependency>  <groupId>javax.inject</groupId>  <artifactId>javax.inject</artifactId>  <version>1</version>  </dependency> |

1.11.1. Dependency Injection with @Inject and @Named

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

**import** javax.inject.Inject;

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

**private** MovieFinder movieFinder;

@Inject

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

this.movieFinder = movieFinder;

}

**public** **void** listMovies() {

this.movieFinder.findMovies(...);

...

}

}

与@Autowired一样，可以在字段级别，方法级别和构造函数参数级别使用@Inject。 此外，您可以将注入点声明为Provider，允许按需访问较短范围的bean或通过Provider.get()调用对其他Bean的惰性访问。 作为上述示例的变体：

**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(...);

...

}

}

如果您想为应该注入的依赖项使用限定名称，则应该按如下方式使用@Named注释：

**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;

}

*// ...*

}

像@Autowired一样，@Inject也可以和java.util.Optional或@Nullable一起使用。 这更适用于此，因为@Inject没有必需的属性。

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

@Inject

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

...

}

}

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

@Inject

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

...

}

}

1.11.2. @Named and @ManagedBean: standard equivalents to the @Component annotation

可以使用@ javax.inject.Named或javax.annotation.ManagedBean来代替@Component，如下所示：

**import** javax.inject.Inject;

**import** javax.inject.Named;

@Named("movieListener") *// @ManagedBean("movieListener") could be used as well*

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

**private** MovieFinder movieFinder;

@Inject

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

this.movieFinder = movieFinder;

}

*// ...*

}

使用@Component而不指定组件的名称是很常见的。 @Named可以以类似的方式使用：

**import** javax.inject.Inject;

**import** javax.inject.Named;

@Named

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

**private** MovieFinder movieFinder;

@Inject

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

this.movieFinder = movieFinder;

}

*// ...*

}

使用@Named或@ManagedBean时，可以像使用Spring注释一样使用组件扫描：

@Configuration

@ComponentScan(basePackages = "org.example")

**public** **class** **AppConfig** {

...

}

|  |  |
| --- | --- |
|  | 与@Component相反，JSR-330 @Named和JSR-250 ManagedBean注释不可组合。 请使用Spring的构造型模型来构建自定义组件注释。 |

1.11.3. Limitations of JSR-330 standard annotations

在使用标准注释时，重要的是要知道一些重要的功能不可用，如下表所示：

| *Table 6. Spring component model elements vs. JSR-330 variants* | | |
| --- | --- | --- |
| **Spring** | **javax.inject.\*** | **javax.inject restrictions / comments** |
| @Autowired | @Inject | @Inject has no 'required' attribute; can be used with Java 8’s Optional instead. |
| @Component | @Named / @ManagedBean | JSR-330 does not provide a composable model, just a way to identify named components. |
| @Scope("singleton") | @Singleton | The JSR-330 default scope is like Spring’s prototype. However, in order to keep it consistent with Spring’s general defaults, a JSR-330 bean declared in the Spring container is a singleton by default. In order to use a scope other than singleton, you should use Spring’s @Scope annotation. javax.inject also provides a [@Scope](http://download.oracle.com/javaee/6/api/javax/inject/Scope.html)annotation. Nevertheless, this one is only intended to be used for creating your own annotations. |
| @Qualifier | @Qualifier / @Named | javax.inject.Qualifier is just a meta-annotation for building custom qualifiers. Concrete String qualifiers (like Spring’s @Qualifier with a value) can be associated through javax.inject.Named. |
| @Value | - | no equivalent |
| @Required | - | no equivalent |
| @Lazy | - | no equivalent |
| ObjectFactory | Provider | javax.inject.Provider is a direct alternative to Spring’s ObjectFactory, just with a shorter get() method name. It can also be used in combination with Spring’s @Autowiredor with non-annotated constructors and setter methods. |

1.12. Java-based container configuration

1.12.1. Basic concepts: @Bean and @Configuration

Spring新的Java配置支持中的中心构件是@Configuration-annotated的类和@Bean-annotated的方法。

@Bean注释用于表示一个方法实例化，配置并初始化一个新的对象，以便由Spring IoC容器管理。 对于那些熟悉Spring的<beans /> XML配置的人来说，@Bean注释和<bean />元素具有相同的作用。 你可以对任何Spring @Component使用@Bean注解方法，但是，它们通常与@Configuration bean一起使用。

用@Configuration注解一个类表明它的主要目的是作为bean定义的来源。 此外，@Configuration类允许通过简单地调用同一个类中的其他@Bean方法来定义bean间依赖关系。 最简单可能的@Configuration类将如下所示：

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** MyService myService() {

**return** **new** MyServiceImpl();

}

}

上面的AppConfig类将等同于以下Spring <beans /> XML：

<beans>

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

</beans>

Full @Configuration vs 'lite' @Bean mode?

当@Bean方法在没有使用@Configuration注释的类中声明时，它们被称为在'lite'模式下处理。 在@Component中甚至在普通的旧类中声明的Bean方法将被视为'lite'，其中包含的类的主要用途不同，并且@Bean方法仅仅是一种奖励。 例如，服务组件可能会在每个适用的组件类上通过额外的@Bean方法向容器公开管理视图。 在这种情况下，@Bean方法是一种简单的通用工厂方法机制。

与完整的@Configuration不同，lite @Bean方法不能声明bean间依赖关系。 相反，他们对包含组件的内部状态和可选的参数进行操作，它们可能会声明。 这样的@Bean方法因此不应该调用其他的@Bean方法; 每个这样的方法实际上只是一个特定的bean引用的工厂方法，没有任何特殊的运行时语义。 这里的积极副作用是，在运行时不需要应用CGLIB子类，所以在类设计方面没有限制（即，包含的类可能是最终的）。

在常见情况下，@Bean方法将在@Configuration类中声明，确保始终使用'full'模式，并且跨方法引用将因此重定向到容器的生命周期管理。 这将防止相同的@Bean方法被意外地通过常规的Java调用调用，这有助于减少在' lite '模式下操作时难以追踪的细微错误。

@Bean和@Configuration注解将在下面的章节中深入讨论。 首先，我们将介绍使用Java-based的配置创建Spring容器的各种方法。

1.12.2. Instantiating the Spring container using AnnotationConfigApplicationContext

下面的章节介绍Spring的AnnotationConfigApplicationContext，这是Spring 3.0中的新功能。 这种多功能的ApplicationContext实现不仅可以接受@Configuration类作为输入，还可以接受用JSR-330元数据注释的普通@Component类和类。

当提供@Configuration类作为输入时，@Configuration类本身被注册为一个bean定义，并且该类中所有声明的@Bean方法也被注册为bean定义。

当提供@Component和JSR-330类时，它们被注册为bean定义，并且假定在必要时在这些类中使用DI元数据（例如@Autowired或@Inject）。

Simple construction

与实例化ClassPathXmlApplicationContext时使用Spring XML文件作为输入的方式大致相同，在实例化AnnotationConfigApplicationContext时，@Configuration类可用作输入。 这允许完全无XML地使用Spring容器：

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

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(AppConfig.class);

MyService myService = ctx.getBean(MyService.class);

myService.doStuff();

}

如上所述，AnnotationConfigApplicationContext不仅限于使用@Configuration类。 任何@Component或JSR-330注释类可作为输入提供给构造函数。 例如：

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

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(MyServiceImpl.class, Dependency1.class, Dependency2.class);

MyService myService = ctx.getBean(MyService.class);

myService.doStuff();

}

上面假设MyServiceImpl，Dependency1和Dependency2使用Spring依赖注入注释，例如@Autowired。

Building the container programmatically using register(Class<?>…​)

AnnotationConfigApplicationContext可以使用无参数构造函数实例化，然后使用register()方法进行配置。 以编程方式构建AnnotationConfigApplicationContext时，此方法特别有用。

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

AnnotationConfigApplicationContext ctx = **new** AnnotationConfigApplicationContext();

ctx.register(AppConfig.class, OtherConfig.class);

ctx.register(AdditionalConfig.class);

ctx.refresh();

MyService myService = ctx.getBean(MyService.class);

myService.doStuff();

}

Enabling component scanning with scan(String…​)

要启用组件扫描，只需注释您的@Configuration类如下：

@Configuration

@ComponentScan(basePackages = "com.acme")

**public** **class** **AppConfig** {

...

}

|  |  |
| --- | --- |
|  | 有经验的Spring用户将熟悉Spring声明中的XML context:namespace  <beans>  <context:component-scan base-package="com.acme"/>  </beans> |

在上面的示例中，将扫描com.acme包，查找任何@ Component注释的类，并且这些类将在容器中注册为Spring bean定义。 AnnotationConfigApplicationContext公开thescan（String ...）方法以允许使用相同的component-scanning功能：

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

AnnotationConfigApplicationContext ctx = **new** AnnotationConfigApplicationContext();

ctx.scan("com.acme");

ctx.refresh();

MyService myService = ctx.getBean(MyService.class);

}

|  |  |
| --- | --- |
|  | 请记住@Configuration类是用@Component进行 [meta-annotated](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-meta-annotations)的，所以它们是组件扫描的候选对象！ 在上面的例子中，假设AppConfig在com.acme包（或下面的任何包）中声明，它将在调用scan()期间被拾取，并且在refresh()时，将处理和注册它的所有@Beanmethods 作为容器内的bean定义。 |

Support for web applications with AnnotationConfigWebApplicationContext

AnnotationConfigApplicationContext的WebApplicationContext变体可与AnnotationConfigWebApplicationContext一起使用。 当配置Spring ContextLoaderListener servlet侦听器，Spring MVC DispatcherServlet等时，可以使用此实现。接下来是配置典型Spring MVC Web应用程序的web.xml片段。 请注意contextClass context-param和init-param的使用：

<web-app>

*<!-- Configure ContextLoaderListener to use AnnotationConfigWebApplicationContext*

*instead of the default XmlWebApplicationContext -->*

<context-param>

<param-name>contextClass</param-name>

<param-value>

org.springframework.web.context.support.AnnotationConfigWebApplicationContext

</param-value>

</context-param>

*<!-- Configuration locations must consist of one or more comma- or space-delimited*

*fully-qualified @Configuration classes. Fully-qualified packages may also be*

*specified for component-scanning -->*

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>com.acme.AppConfig</param-value>

</context-param>

*<!-- Bootstrap the root application context as usual using ContextLoaderListener -->*

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

*<!-- Declare a Spring MVC DispatcherServlet as usual -->*

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

*<!-- Configure DispatcherServlet to use AnnotationConfigWebApplicationContext*

*instead of the default XmlWebApplicationContext -->*

<init-param>

<param-name>contextClass</param-name>

<param-value>

org.springframework.web.context.support.AnnotationConfigWebApplicationContext

</param-value>

</init-param>

*<!-- Again, config locations must consist of one or more comma- or space-delimited*

*and fully-qualified @Configuration classes -->*

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>com.acme.web.MvcConfig</param-value>

</init-param>

</servlet>

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

<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

<url-pattern>/app/\*</url-pattern>

</servlet-mapping>

</web-app>

1.12.3. Using the @Bean annotation

@Bean是一个方法级别的注释，并且是XML <bean />元素的直接模拟。 注解支持<bean />提供的一些属性，例如：i [init-method](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-lifecycle-initializingbean), [destroy-method](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-lifecycle-disposablebean), [autowiring](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-autowire) 和name。

您可以在带有@ Configuration注释或带有@ Component注释的类中使用@Bean注释。

Declaring a bean

要声明一个bean，只需使用@Bean注释来注释一个方法即可。 您可以使用此方法在指定为方法返回值的类型的ApplicationContext中注册bean定义。 默认情况下，bean名称将与方法名称相同。 以下是一个@Bean方法声明的简单示例：

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** TransferServiceImpl transferService() {

**return** **new** TransferServiceImpl();

}

}

上述配置完全等同于以下Spring XML：

<beans>

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

</beans>

这两个声明都在ApplicationContext中创建一个名为transferService的bean，绑定到TransferServiceImpl类型的对象实例：

transferService -> com.acme.TransferServiceImpl

你也可以用接口（或base class）返回类型声明你的@Bean方法：

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** TransferService transferService() {

**return** **new** TransferServiceImpl();

}

}

但是，这会将预测类型预测的可见性限制为指定的接口类型（TransferService），然后，一旦实例化受影响的单例bean，该类型（TransferServiceImpl）只能由容器知道。 非懒惰的singleton bean根据它们的声明顺序被实例化，因此您可能会看到不同的类型匹配结果，具体取决于另一个组件尝试通过非声明类型进行匹配的时间（例如@Autowired TransferServiceImpl，它只会解析一次“transferService” bean已经被实例化）。

|  |  |
| --- | --- |
|  | 如果您始终通过声明的服务接口来引用您的类型，那么您的@Bean返回类型可以安全地加入该设计决策。 但是，对于实现多个接口的组件或其实现类型可能引用的组件，声明最具体的返回类型是可能的（至少与注入点引用您的bean所要求的相同）是比较安全的。 |

Bean dependencies

@Bean注释的方法可以有任意数量的参数来描述构建bean所需的依赖关系。 例如，如果我们的TransferService需要一个AccountRepository，我们可以通过一个方法参数实现这个依赖：

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** TransferService transferService(AccountRepository accountRepository) {

**return** **new** TransferServiceImpl(accountRepository);

}

}

解析机制与基于构造函数的依赖注入非常相似，请参阅[the relevant section](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-constructor-injection) 以获取更多详细信息。

Receiving lifecycle callbacks

任何使用@Bean注释定义的类都支持常规生命周期回调，并且可以使用JSR-250中的@PostConstruct和@PreDestroy注释，请参阅[JSR-250 annotations](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-postconstruct-and-predestroy-annotations) 以获取更多详细信息。

常规的Spring [lifecycle](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-nature) 回调也被完全支持。 如果一个bean实现了InitializingBean，DisposableBean或Lifecycle，那么它们各自的方法由容器调用。

标准的\* Aware接口，如[BeanFactoryAware](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-beanfactory), [BeanNameAware](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-aware), [MessageSourceAware](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#context-functionality-messagesource),[ApplicationContextAware](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-aware)等也完全支持。

@Bean注解支持指定任意的初始化和销毁回调方法，就像Spring XML的bean元素的init-method和destroy-method属性一样：

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

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

*// initialization logic*

}

}

**public** **class** **Bar** {

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

*// destruction logic*

}

}

@Configuration

**public** **class** **AppConfig** {

@Bean(initMethod = "init")

**public** Foo foo() {

**return** **new** Foo();

}

@Bean(destroyMethod = "cleanup")

**public** Bar bar() {

**return** **new** Bar();

}

}

|  |  |
| --- | --- |
|  | 默认情况下，使用具有公共close or shutdown方法的Java配置定义的bean将自动列入销毁回调。 如果你有一个public close或shutdown方法，并且你不希望在容器关闭时调用它，只需在你的bean定义中添加@Bean（destroyMethod =“”）来禁用默认（inferred）模式。  您可能希望为通过JNDI获取的资源默认执行此操作，因为其生命周期在应用程序外部进行管理。 特别是，确保始终为DataSource执行此操作，因为它已知在Java EE应用程序服务器上存在问题。  @Bean(destroyMethod="")  **public** DataSource dataSource() **throws** NamingException {  **return** (DataSource) jndiTemplate.lookup("MyDS");  }  此外，使用@Bean方法，通常会选择使用编程式JNDI查找：使用Spring的JndiTemplate / JndiLocatorDelegate帮助器(helpers)或直接使用JNDI InitialContext，但不使用JndiObjectFactoryBean变体，这会强制您将返回类型声明为FactoryBean类型，而不是 实际的目标类型，这使得在其他@Bean方法中用于引用所提供资源的交叉引用调用更加困难。 |

当然，就上面的Foo而言，在构造过程中直接调用init()方法同样有效：

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** Foo foo() {

Foo foo = **new** Foo();

foo.init();

**return** foo;

}

*// ...*

}

|  |  |
| --- | --- |
|  | 当您直接使用Java进行工作时，您可以对您的对象执行任何您喜欢的操作，并不总是需要依赖容器生命周期！ |

Specifying bean scope

Using the @Scope annotation

您可以指定使用@Bean注释定义的bean应具有特定范围。 您可以使用 [Bean Scopes](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-scopes) 部分中指定的任何标准范围。

默认范围是singleton，但您可以使用@Scope注释覆盖它：

@Configuration

**public** **class** **MyConfiguration** {

@Bean

**@Scope("prototype")**

**public** Encryptor encryptor() {

*// ...*

}

}

@Scope and scoped-proxy

Spring提供了一种通过 [scoped proxies](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-scopes-other-injection)来处理作用域依赖关系的便捷方式。 使用XML配置时创建此类代理的最简单方法是<aop：scoped-proxy />元素。 使用@Scope注释在Java中配置bean提供了与proxyMode属性等效的支持。 默认值是无代理（ScopedProxyMode.NO），但您可以指定ScopedProxyMode.TARGET\_CLASS或ScopedProxyMode.INTERFACES。

如果您使用Java将scoped proxy示例从XML参考文档（请参阅前面的链接）移植到我们的@Bean，它将如下所示：

*// 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;

}

Customizing bean naming

默认情况下，配置类使用@Bean方法的名称作为结果bean的名称。 但是，可以使用name属性覆盖此功能。

@Configuration

**public** **class** **AppConfig** {

@Bean(name = "myFoo")

**public** Foo foo() {

**return** **new** Foo();

}

}

Bean aliasing

正如在 [Naming beans](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-beanname)中所讨论的，有时需要为单个bean提供多个名称，称为bean别名。 @Bean注释的name属性为此接受一个String数组。

@Configuration

**public** **class** **AppConfig** {

@Bean(name = { "dataSource", "subsystemA-dataSource", "subsystemB-dataSource" })

**public** DataSource dataSource() {

*// instantiate, configure and return DataSource bean...*

}

}

Bean description

有时候提供一个更详细的bean的文本描述是有帮助的。 当bean暴露（可能通过JMX）用于监视目的时，这可能特别有用。

要将描述添加到@Bean，可以使用 [@Description](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/context/annotation/Description.html)注释：

@Configuration

**public** **class** **AppConfig** {

@Bean

**@Description("Provides a basic example of a bean")**

**public** Foo foo() {

**return** **new** Foo();

}

}

1.12.4. Using the @Configuration annotation

@Configuration是一个类级注释，指示一个对象是一个bean定义的来源。 @Configuration类通过public @Bean注释方法声明bean。 调用@Configuration类上的@Bean方法也可以用来定义bean间的依赖关系。 有关一般介绍，请参阅基 [Basic concepts: @Bean and @Configuration](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-java-basic-concepts)。

Injecting inter-bean dependencies

当@Beans彼此依赖时，表达这种依赖就如同一个bean方法调用另一个一样简单：

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** Foo foo() {

**return** **new** Foo(bar());

}

@Bean

**public** Bar bar() {

**return** **new** Bar();

}

}

在上面的示例中，foo bean通过构造函数注入接收对bar的引用。

|  |  |
| --- | --- |
|  | 这种声明bean间依赖关系的方法只有在@Configuration类中声明@Bean方法时才有效。 你不能用简单的@Component类来声明bean间的依赖关系。 |

Lookup method injection

如前所述， [lookup method injection](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-method-injection)是一种您很少使用的高级功能。 在singleton-scoped的bean对prototype-scoped的bean具有依赖关系的情况下，它很有用。 对这种类型的配置使用Java提供了实现这种模式的自然方法。

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

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

*// grab a new instance of the appropriate Command interface*

Command command = createCommand();

*// set the state on the (hopefully brand new) Command instance*

command.setState(commandState);

**return** command.execute();

}

*// okay... but where is the implementation of this method?*

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

}

使用Java配置支持，您可以创建CommandManager的子类，其中抽象的createCommand() 方法被重写，以便查找新的（prototype）命令对象：

@Bean

@Scope("prototype")

**public** AsyncCommand asyncCommand() {

AsyncCommand command = **new** AsyncCommand();

*// inject dependencies here as required*

**return** command;

}

@Bean

**public** CommandManager commandManager() {

*// return new anonymous implementation of CommandManager with command() overridden*

*// to return a new prototype Command object*

**return** **new** CommandManager() {

**protected** Command createCommand() {

**return** asyncCommand();

}

}

}

Further information about how Java-based configuration works internally

以下示例显示了一个被调用两次的@Bean注释方法：

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** ClientService clientService1() {

ClientServiceImpl clientService = **new** ClientServiceImpl();

clientService.setClientDao(clientDao());

**return** clientService;

}

@Bean

**public** ClientService clientService2() {

ClientServiceImpl clientService = **new** ClientServiceImpl();

clientService.setClientDao(clientDao());

**return** clientService;

}

@Bean

**public** ClientDao clientDao() {

**return** **new** ClientDaoImpl();

}

}

clientDao（）在clientService1（）中被调用一次，在clientService2（）中被调用一次。 由于此方法创建ClientDaoImpl的新实例并将其返回，因此通常期望拥有2个实例（每个服务一个实例）。 这肯定会有问题：在Spring中，实例化的bean默认具有单例作用域。 这就是神奇的地方：所有@Configuration类在启动时都使用CGLIB进行子类化。 在子类中，child方法在调用父方法并创建新实例之前首先检查容器是否有缓存的（范围）bean。 请注意，从Spring 3.2开始，不再需要将CGLIB添加到类路径中，因为CGLIB类已在org.springframework.cglib下重新打包，并直接包含在Spring-Core JAR中。

|  |  |
| --- | --- |
|  | 根据您的bean的范围，行为可能会有所不同。 我们在这里讨论singletons。 |
|  | 由于CGLIB在启动时动态添加功能，因此存在一些限制，特别是配置类不能是最终的。 但是，从4.3开始，任何构造函数都可以在配置类上使用，包括对默认注入使用@Autowired或单个非默认构造函数声明。  如果您希望避免任何CGLIB限制，请考虑在非non-@Configuration类上声明您的@Bean方法，例如使用简单的@Component类。 @Bean方法之间的跨方法调用不会被拦截，因此您必须在构造方法或方法级别专门依赖依赖注入。 |

1.12.5. Composing Java-based configurations

Using the @Import annotation

就像在Spring XML文件中使用<import />元素以帮助模块化配置一样，@Import注释允许从另一个配置类加载@Bean定义：

@Configuration

**public** **class** **ConfigA** {

@Bean

**public** A a() {

**return** **new** A();

}

}

@Configuration

@Import(ConfigA.class)

**public** **class** **ConfigB** {

@Bean

**public** B b() {

**return** **new** B();

}

}

现在，不需要在实例化上下文时指定ConfigA.class和ConfigB.class，只需要显式提供ConfigBneeds：

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

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(ConfigB.class);

*// now both beans A and B will be available...*

A a = ctx.getBean(A.class);

B b = ctx.getBean(B.class);

}

这种方法简化了容器实例化，因为只有一个类需要处理，而不需要开发人员在构建过程中记住大量的@Configuration类。

|  |  |
| --- | --- |
|  | 从Spring Framework 4.2开始，@Import还支持对常规组件类的引用，类似于AnnotationConfigApplicationContext.register方法。 如果您想要避免组件扫描，使用几个配置类作为明确定义所有组件的入口点，这特别有用。 |

Injecting dependencies on imported @Bean definitions

上面的例子工作，但是很简单。 在大多数实际场景中，bean将跨配置类彼此依赖。 当使用XML时，这本身并不是一个问题，因为不涉及编译器，并且可以简单地声明ref =“someBean”并相信Spring将在容器初始化期间解决它。 当然，在使用@Configuration类时，Java编译器会对配置模型施加约束，因为对其他bean的引用必须是有效的Java语法。

幸运的是，解决这个问题很简单。 正如 [we already discussed](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-java-dependencies)，@Bean方法可以有任意数量的描述bean依赖关系的参数。 让我们考虑一个更现实的场景，其中有几个@Configuration类，每个类都依赖于其他类中声明的bean：

@Configuration

**public** **class** **ServiceConfig** {

@Bean

**public** TransferService transferService(AccountRepository accountRepository) {

**return** **new** TransferServiceImpl(accountRepository);

}

}

@Configuration

**public** **class** **RepositoryConfig** {

@Bean

**public** AccountRepository accountRepository(DataSource dataSource) {

**return** **new** JdbcAccountRepository(dataSource);

}

}

@Configuration

@Import({ServiceConfig.class, RepositoryConfig.class})

**public** **class** **SystemTestConfig** {

@Bean

**public** DataSource dataSource() {

*// return new DataSource*

}

}

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

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(SystemTestConfig.class);

*// everything wires up across configuration classes...*

TransferService transferService = ctx.getBean(TransferService.class);

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

}

还有另一种方法可以达到相同的结果。 请记住@Configuration类最终只是容器中的另一个bean：这意味着它们可以像任何其他bean一样利用@Autowired和@Value注入等等！

|  |  |
| --- | --- |
|  | 确保以这种方式注入的依赖关系只有最简单的一种。 @Configuration类在上下文初始化期间处理得相当早，并强制依赖性以这种方式注入可能会导致意外的早期初始化。 在可能的情况下，采用基于参数的注入，如上例所示。  此外，通过@Bean特别小心BeanPostProcessor和BeanFactoryPostProcessor定义。 这些通常应该声明为静态的@Bean方法，而不是触发其包含的配置类的实例化。 否则，@Autowired和@Value将不会在配置类本身上工作，因为它太早创建为bean实例。 |

@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类中的构造函数注入仅在Spring Framework 4.3中受支持。 另请注意，如果目标bean只定义一个构造函数，则不需要指定@Autowired; 在上面的例子中，RepositoryConfig构造函数不需要@Autowired。 |

*Fully-qualifying导入的beans以方便导航*

*在上面的场景中，使用@Autowired可以很好地工作并提供所需的模块化，但是确切地确定自动布线bean定义的声明位置仍然有些模糊。 例如，作为开发人员查看ServiceConfig，您如何确切知道@Autowired AccountRepository bean的声明位置？ 它在代码中并不明确，这可能会很好。 请记住，* [Spring Tool Suite](https://spring.io/tools/sts)*提供的工具可以呈现图表，显示如何连接所有东西 - 这可能就是您所需要的。 此外，您的Java IDE可以轻松找到AccountRepository类型的所有声明和用法，并且可以快速向您显示返回该类型的@Bean方法的位置。*

在这种不明确性不可接受的情况下，如果您希望从IDE内的一个@Configuration类直接导航到另一个类，请考虑自动装配配置类本身：

@Configuration

**public** **class** **ServiceConfig** {

@Autowired

**private** RepositoryConfig repositoryConfig;

@Bean

**public** TransferService transferService() {

*// navigate 'through' the config class to the @Bean method!*

**return** **new** TransferServiceImpl(repositoryConfig.accountRepository());

}

}

在上面的情况中，它完全明确了AccountRepository的定义。 但是，ServiceConfig现在与RepositoryConfig紧密耦合; 这是权衡。 通过使用基于接口的或基于抽象的基于类的@Configuration类，可以稍微缓解这种紧密耦合。 考虑以下：

@Configuration

**public** **class** **ServiceConfig** {

@Autowired

**private** RepositoryConfig repositoryConfig;

@Bean

**public** TransferService transferService() {

**return** **new** TransferServiceImpl(repositoryConfig.accountRepository());

}

}

@Configuration

**public** **interface** **RepositoryConfig** {

@Bean

AccountRepository accountRepository();

}

@Configuration

**public** **class** **DefaultRepositoryConfig** **implements** RepositoryConfig {

@Bean

**public** AccountRepository accountRepository() {

**return** **new** JdbcAccountRepository(...);

}

}

@Configuration

@Import({ServiceConfig.class, DefaultRepositoryConfig.class}) *// import the concrete config!*

**public** **class** **SystemTestConfig** {

@Bean

**public** DataSource dataSource() {

*// return DataSource*

}

}

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

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(SystemTestConfig.class);

TransferService transferService = ctx.getBean(TransferService.class);

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

}

现在ServiceConfig与具体的DefaultRepositoryConfig松散耦合，并且内置IDE工具仍然有用：开发人员可以轻松获得RepositoryConfig实现的类型层次结构。 通过这种方式，浏览@Configuration类及其依赖关系与导航基于接口的代码的常见过程无异。

|  |  |
| --- | --- |
|  | 如果您想影响某些bean的启动创建顺序，请考虑将其中的一些声明为@Lazy（用于在第一次访问时创建，而不是在启动时创建）或在某些其他bean上声明为@DependsOn（确保特定的其他Bean将 在当前bean之前创建，超出后者的直接依赖意味）。 |

Conditionally include @Configuration classes or @Bean methods

根据某些任意系统状态，有条件地启用或禁用完整的@Configuration类或者甚至单个@Bean方法通常很有用。 一个常见的例子是，只有在Spring环境中启用了特定的配置文件时，才使用@Profile注释来激活bean（有关详细信息，请参阅 [Bean definition profiles](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-definition-profiles)）。

@Profile注释实际上是使用一种更为灵活的名为 [@Conditional](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/context/annotation/Conditional.html)的注释来实现的。 @Conditional注释表示在@Bean注册之前应该查阅的特定org.springframework.context.annotation.Condition实现。

Condition接口的实现只是提供一个返回true或false的matches（...）方法。 例如，以下是用于@Profile的实际Condition实现：

@Override

**public** **boolean** matches(ConditionContext context, AnnotatedTypeMetadata metadata) {

**if** (context.getEnvironment() != null) {

*// Read the @Profile annotation attributes*

MultiValueMap<String, Object> attrs = metadata.getAllAnnotationAttributes(Profile.class.getName());

**if** (attrs != null) {

**for** (Object value : attrs.get("value")) {

**if** (context.getEnvironment().acceptsProfiles(((String**[]**) value))) {

**return** true;

}

}

**return** false;

}

}

**return** true;

}

有关更多详细信息，请参阅 [@Conditional javadocs](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/context/annotation/Conditional.html)。

Combining Java and XML configuration

Spring的@Configuration类支持并不旨在成为Spring XML的100％完全替代品。 Spring XML命名空间等一些工具仍然是配置容器的理想方式。 在XML方便或必要的情况下，您可以选择：使用例如ClassPathXmlApplicationContext以“XML-centric”的方式实例化容器，或者使用AnnotationConfigApplicationContext和@ImportResource注解以“Java-centric”的方式实例化容器 根据需要导入XML。

XML-centric use of @Configuration classes

最好从XML引导Spring容器，并以临时方式包含@Configuration类。 例如，在使用Spring XML的大型现有代码库中，根据需要创建@Configuration类并从现有XML文件中包含它们会更容易。 下面你会发现在这种“XML-centric”的情况下使用@Configuration类的选项。

*将@Configuration类声明为普通的Spring <bean />元素*

*请记住，@Configuration类最终只是容器中的bean定义。 在这个例子中，我们创建一个名为AppConfig的@Configuration类，并将它作为<bean />定义包含在system-test-config.xml中。 由于<context：annotation-config />已打开，容器将识别@Configuration注释并正确处理在AppConfig中声明的@Bean方法。*

@Configuration

**public** **class** **AppConfig** {

@Autowired

**private** DataSource dataSource;

@Bean

**public** AccountRepository accountRepository() {

**return** **new** JdbcAccountRepository(dataSource);

}

@Bean

**public** TransferService transferService() {

**return** **new** TransferService(accountRepository());

}

}

**system-test-config.xml**:

<beans>

*<!-- enable processing of annotations such as @Autowired and @Configuration -->*

<context:annotation-config/>

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

<bean class="com.acme.AppConfig"/>

<bean class="org.springframework.jdbc.datasource.DriverManagerDataSource">

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

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

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

</bean>

</beans>

**jdbc.properties**:

jdbc.url=jdbc:hsqldb: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);

*// ...*

}

|  |  |
| --- | --- |
|  | 在上面的system-test-config.xml中，AppConfig <bean />没有声明一个id元素。 虽然这样做是可以接受的，但没有必要考虑到其他bean将不会引用它，并且它不太可能通过名称明确从容器中提取。 与DataSource bean类似 - 它只能通过类型自动装配，因此不需要显式的bean id。 |

*使用<context：component-scan />来获取@Configuration类*

*由于@Configuration是使用@Component进行元注释的，所以@ Configuration-annotated类自动成为组件扫描的候选对象。 使用与上面相同的场景，我们可以重新定义system-test-config.xml以利用组件扫描。 请注意，在这种情况下，我们不需要显式声明<context：annotation-config />，因为<context：component-scan />启用了相同的功能。*

**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 class-centric use of XML with @ImportResource

在@Configuration类是配置容器的主要机制的应用程序中，仍然可能有必要使用至少一些XML。 在这些场景中，只需使用@ImportResource并根据需要定义尽可能多的XML。 这样做可以实现“ava-centric”的方式来配置容器并将XML保持最低限度。

@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);

*// ...*

}

1.13. Environment abstraction

[Environment](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/core/env/Environment.html)是集成在容器中的抽象，它模拟应用程序环境的两个关键方面： [*profiles*](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-definition-profiles) and [*properties*](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-property-source-abstraction)。

A *profile* is a named, logical group of bean definitions to be registered with the container only if the given profile is active. Beans may be assigned to a profile whether defined in XML or via annotations. The role of the Environment object with relation to profiles is in determining which profiles (if any) are currently active, and which profiles (if any) should be active by default.

Properties play an important role in almost all applications, and may originate from a variety of sources: properties files, JVM system properties, system environment variables, JNDI, servlet context parameters, ad-hoc Properties objects, Maps, and so on. The role of the Environment object with relation to properties is to provide the user with a convenient service interface for configuring property sources and resolving properties from them.

1.13.1. Bean definition profiles

Bean definition profiles is a mechanism in the core container that allows for registration of different beans in different environments. The word *environment* can mean different things to different users and this feature can help with many use cases, including:

* working against an in-memory datasource in development vs looking up that same datasource from JNDI when in QA or production
* registering monitoring infrastructure only when deploying an application into a performance environment
* registering customized implementations of beans for customer A vs. customer B deployments

Let’s consider the first use case in a practical application that requires a DataSource. In a test environment, the configuration may look like this:

@Bean

**public** DataSource dataSource() {

**return** **new** EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.HSQL)

.addScript("my-schema.sql")

.addScript("my-test-data.sql")

.build();

}

Let’s now consider how this application will be deployed into a QA or production environment, assuming that the datasource for the application will be registered with the production application server’s JNDI directory. Our dataSource bean now looks like this:

@Bean(destroyMethod="")

**public** DataSource dataSource() **throws** Exception {

Context ctx = **new** InitialContext();

**return** (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");

}

The problem is how to switch between using these two variations based on the current environment. Over time, Spring users have devised a number of ways to get this done, usually relying on a combination of system environment variables and XML <import/> statements containing ${placeholder} tokens that resolve to the correct configuration file path depending on the value of an environment variable. Bean definition profiles is a core container feature that provides a solution to this problem.

If we generalize the example use case above of environment-specific bean definitions, we end up with the need to register certain bean definitions in certain contexts, while not in others. You could say that you want to register a certain profile of bean definitions in situation A, and a different profile in situation B. Let’s first see how we can update our configuration to reflect this need.

@Profile

The [@Profile](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/context/annotation/Profile.html) annotation allows you to indicate that a component is eligible for registration when one or more specified profiles are active. Using our example above, we can rewrite the dataSource configuration as follows:

@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");

}

}

|  |  |
| --- | --- |
|  | As mentioned before, with @Bean methods, you will typically choose to use programmatic JNDI lookups: either using Spring’s JndiTemplate/JndiLocatorDelegate helpers or the straight JNDI InitialContext usage shown above, but not the JndiObjectFactoryBean variant which would force you to declare the return type as the FactoryBean type. |

@Profile can be used as a [meta-annotation](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-meta-annotations) for the purpose of creating a custom *composed annotation*. The following example defines a custom @Production annotation that can be used as a drop-in replacement for @Profile("production"):

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

**@Profile("production")**

**public** @interface Production {

}

|  |  |
| --- | --- |
|  | If a @Configuration class is marked with @Profile, all of the @Bean methods and @Import annotations associated with that class will be bypassed unless one or more of the specified profiles are active. If a @Component or @Configuration class is marked with @Profile({"p1", "p2"}), that class will not be registered/processed unless profiles 'p1' and/or 'p2' have been activated. If a given profile is prefixed with the NOT operator (!), the annotated element will be registered if the profile is **not** active. For example, given @Profile({"p1", "!p2"}), registration will occur if profile 'p1' is active or if profile 'p2' is not active. |

@Profile can also be declared at the method level to include only one particular bean of a configuration class, e.g. for alternative variants of a particular bean:

@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");

}

}

|  |  |
| --- | --- |
|  | With @Profile on @Bean methods, a special scenario may apply: In the case of overloaded @Bean methods of the same Java method name (analogous to constructor overloading), an @Profile condition needs to be consistently declared on all overloaded methods. If the conditions are inconsistent, only the condition on the first declaration among the overloaded methods will matter. @Profile can therefore not be used to select an overloaded method with a particular argument signature over another; resolution between all factory methods for the same bean follows Spring’s constructor resolution algorithm at creation time.  If you would like to define alternative beans with different profile conditions, use distinct Java method names pointing to the same bean name via the @Bean name attribute, as indicated in the example above. If the argument signatures are all the same (e.g. all of the variants have no-arg factory methods), this is the only way to represent such an arrangement in a valid Java class in the first place (since there can only be one method of a particular name and argument signature). |

XML bean definition profiles

The XML counterpart is the profile attribute of the <beans> element. Our sample configuration above can be rewritten in two XML files as follows:

<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>

It is also possible to avoid that split and nest <beans/> elements within the same file:

<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>

The spring-bean.xsd has been constrained to allow such elements only as the last ones in the file. This should help provide flexibility without incurring clutter in the XML files.

Activating a profile

Now that we have updated our configuration, we still need to instruct Spring which profile is active. If we started our sample application right now, we would see a NoSuchBeanDefinitionException thrown, because the container could not find the Spring bean named dataSource.

Activating a profile can be done in several ways, but the most straightforward is to do it programmatically against the Environment API which is available via an ApplicationContext:

AnnotationConfigApplicationContext ctx = **new** AnnotationConfigApplicationContext();

ctx.getEnvironment().setActiveProfiles("development");

ctx.register(SomeConfig.class, StandaloneDataConfig.class, JndiDataConfig.class);

ctx.refresh();

In addition, profiles may also be activated declaratively through the spring.profiles.active property which may be specified through system environment variables, JVM system properties, servlet context parameters in web.xml, or even as an entry in JNDI (see [PropertySource abstraction](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-property-source-abstraction)). In integration tests, active profiles can be declared via the @ActiveProfiles annotation in the spring-test module (see [Context configuration with environment profiles](https://docs.spring.io/spring/docs/current/spring-framework-reference/testing.html#testcontext-ctx-management-env-profiles)).

Note that profiles are not an "either-or" proposition; it is possible to activate multiple profiles at once. Programmatically, simply provide multiple profile names to the setActiveProfiles() method, which accepts String…​ varargs:

ctx.getEnvironment().setActiveProfiles("profile1", "profile2");

Declaratively, spring.profiles.active may accept a comma-separated list of profile names:

-Dspring.profiles.active="profile1,profile2"

Default profile

The *default* profile represents the profile that is enabled by default. Consider the following:

@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();

}

}

If no profile is active, the dataSource above will be created; this can be seen as a way to provide a *default* definition for one or more beans. If any profile is enabled, the *default* profile will not apply.

The name of the default profile can be changed using setDefaultProfiles() on the Environment or declaratively using the spring.profiles.default property.

1.13.2. PropertySource abstraction

Spring’s Environment abstraction provides search operations over a configurable hierarchy of property sources. To explain fully, consider the following:

ApplicationContext ctx = **new** GenericApplicationContext();

Environment env = ctx.getEnvironment();

**boolean** containsFoo = env.containsProperty("foo");

System.out.println("Does my environment contain the 'foo' property? " + containsFoo);

In the snippet above, we see a high-level way of asking Spring whether the foo property is defined for the current environment. To answer this question, the Environment object performs a search over a set of [PropertySource](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/core/env/PropertySource.html) objects. A PropertySource is a simple abstraction over any source of key-value pairs, and Spring’s [StandardEnvironment](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/core/env/StandardEnvironment.html) is configured with two PropertySource objects — one representing the set of JVM system properties (*a la* System.getProperties()) and one representing the set of system environment variables (*a la* System.getenv()).

|  |  |
| --- | --- |
|  | These default property sources are present for StandardEnvironment, for use in standalone applications. [StandardServletEnvironment](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/web/context/support/StandardServletEnvironment.html) is populated with additional default property sources including servlet config and servlet context parameters. It can optionally enable a [JndiPropertySource](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/jndi/JndiPropertySource.html). See the javadocs for details. |

Concretely, when using the StandardEnvironment, the call to env.containsProperty("foo") will return true if a foo system property or foo environment variable is present at runtime.

|  |  |
| --- | --- |
|  | The search performed is hierarchical. By default, system properties have precedence over environment variables, so if the foo property happens to be set in both places during a call to env.getProperty("foo"), the system property value will 'win' and be returned preferentially over the environment variable. Note that property values will not get merged but rather completely overridden by a preceding entry.  For a common StandardServletEnvironment, the full hierarchy looks as follows, with the highest-precedence entries at the top:   * ServletConfig parameters (if applicable, e.g. in case of a DispatcherServlet context) * ServletContext parameters (web.xml context-param entries) * JNDI environment variables ("java:comp/env/" entries) * JVM system properties ("-D" command-line arguments) * JVM system environment (operating system environment variables) |

Most importantly, the entire mechanism is configurable. Perhaps you have a custom source of properties that you’d like to integrate into this search. No problem — simply implement and instantiate your own PropertySource and add it to the set of PropertySources for the current Environment:

ConfigurableApplicationContext ctx = **new** GenericApplicationContext();

MutablePropertySources sources = ctx.getEnvironment().getPropertySources();

sources.addFirst(**new** MyPropertySource());

In the code above, MyPropertySource has been added with highest precedence in the search. If it contains a foo property, it will be detected and returned ahead of any foo property in any other PropertySource. The [MutablePropertySources](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/core/env/MutablePropertySources.html) API exposes a number of methods that allow for precise manipulation of the set of property sources.

1.13.3. @PropertySource

The [@PropertySource](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/context/annotation/PropertySource.html) annotation provides a convenient and declarative mechanism for adding a PropertySource to Spring’s Environment.

Given a file "app.properties" containing the key/value pair testbean.name=myTestBean, the following @Configuration class uses @PropertySource in such a way that a call to testBean.getName() will return "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;

}

}

Any ${…​} placeholders present in a @PropertySource resource location will be resolved against the set of property sources already registered against the environment. For example:

@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;

}

}

Assuming that "my.placeholder" is present in one of the property sources already registered, e.g. system properties or environment variables, the placeholder will be resolved to the corresponding value. If not, then "default/path" will be used as a default. If no default is specified and a property cannot be resolved, an IllegalArgumentException will be thrown.

1.13.4. Placeholder resolution in statements

Historically, the value of placeholders in elements could be resolved only against JVM system properties or environment variables. No longer is this the case. Because the Environment abstraction is integrated throughout the container, it’s easy to route resolution of placeholders through it. This means that you may configure the resolution process in any way you like: change the precedence of searching through system properties and environment variables, or remove them entirely; add your own property sources to the mix as appropriate.

Concretely, the following statement works regardless of where the customer property is defined, as long as it is available in the Environment:

<beans>

<import resource="com/bank/service/${customer}-config.xml"/>

</beans>

1.14. Registering a LoadTimeWeaver

The LoadTimeWeaver is used by Spring to dynamically transform classes as they are loaded into the Java virtual machine (JVM).

To enable load-time weaving add the @EnableLoadTimeWeaving to one of your @Configuration classes:

@Configuration

@EnableLoadTimeWeaving

**public** **class** **AppConfig** {

}

Alternatively for XML configuration use the context:load-time-weaver element:

<beans>

<context:load-time-weaver/>

</beans>

Once configured for the ApplicationContext. Any bean within that ApplicationContext may implement LoadTimeWeaverAware, thereby receiving a reference to the load-time weaver instance. This is particularly useful in combination with [Spring’s JPA support](https://docs.spring.io/spring/docs/current/spring-framework-reference/data-access.html#orm-jpa) where load-time weaving may be necessary for JPA class transformation. Consult the LocalContainerEntityManagerFactoryBean javadocs for more detail. For more on AspectJ load-time weaving, see [Load-time weaving with AspectJ in the Spring Framework](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-aj-ltw).

1.15. Additional capabilities of the ApplicationContext

As was discussed in the chapter introduction, the org.springframework.beans.factory package provides basic functionality for managing and manipulating beans, including in a programmatic way. The org.springframework.context package adds the[ApplicationContext](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/context/ApplicationContext.html) interface, which extends the BeanFactory interface, in addition to extending other interfaces to provide additional functionality in a more *application framework-oriented style*. Many people use the ApplicationContext in a completely declarative fashion, not even creating it programmatically, but instead relying on support classes such as ContextLoader to automatically instantiate an ApplicationContext as part of the normal startup process of a Java EE web application.

To enhance BeanFactory functionality in a more framework-oriented style the context package also provides the following functionality:

* *Access to messages in i18n-style*, through the MessageSource interface.
* *Access to resources*, such as URLs and files, through the ResourceLoader interface.
* *Event publication* to namely beans implementing the ApplicationListener interface, through the use of the ApplicationEventPublisher interface.
* *Loading of multiple (hierarchical) contexts*, allowing each to be focused on one particular layer, such as the web layer of an application, through the HierarchicalBeanFactory interface.

1.15.1. Internationalization using MessageSource

The ApplicationContext interface extends an interface called MessageSource, and therefore provides internationalization (i18n) functionality. Spring also provides the interface HierarchicalMessageSource, which can resolve messages hierarchically. Together these interfaces provide the foundation upon which Spring effects message resolution. The methods defined on these interfaces include:

* String getMessage(String code, Object[] args, String default, Locale loc): The basic method used to retrieve a message from the MessageSource. When no message is found for the specified locale, the default message is used. Any arguments passed in become replacement values, using the MessageFormat functionality provided by the standard library.
* String getMessage(String code, Object[] args, Locale loc): Essentially the same as the previous method, but with one difference: no default message can be specified; if the message cannot be found, a NoSuchMessageException is thrown.
* String getMessage(MessageSourceResolvable resolvable, Locale locale): All properties used in the preceding methods are also wrapped in a class named MessageSourceResolvable, which you can use with this method.

When an ApplicationContext is loaded, it automatically searches for a MessageSource bean defined in the context. The bean must have the name messageSource. If such a bean is found, all calls to the preceding methods are delegated to the message source. If no message source is found, the ApplicationContext attempts to find a parent containing a bean with the same name. If it does, it uses that bean as the MessageSource. If the ApplicationContext cannot find any source for messages, an empty DelegatingMessageSource is instantiated in order to be able to accept calls to the methods defined above.

Spring provides two MessageSource implementations, ResourceBundleMessageSource and StaticMessageSource. Both implement HierarchicalMessageSource in order to do nested messaging. The StaticMessageSource is rarely used but provides programmatic ways to add messages to the source. The ResourceBundleMessageSource is shown in the following example:

<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>

In the example it is assumed you have three resource bundles defined in your classpath called format, exceptions and windows. Any request to resolve a message will be handled in the JDK standard way of resolving messages through ResourceBundles. For the purposes of the example, assume the contents of two of the above resource bundle files are…​

# in format.properties

message=Alligators rock!

# in exceptions.properties

argument.required=The {0} argument is required.

A program to execute the MessageSource functionality is shown in the next example. Remember that all ApplicationContextimplementations are also MessageSource implementations and so can be cast to the MessageSource interface.

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

MessageSource resources = **new** ClassPathXmlApplicationContext("beans.xml");

String message = resources.getMessage("message", null, "Default", null);

System.out.println(message);

}

The resulting output from the above program will be…​

Alligators rock!

So to summarize, the MessageSource is defined in a file called beans.xml, which exists at the root of your classpath. The messageSource bean definition refers to a number of resource bundles through its basenames property. The three files that are passed in the list to the basenames property exist as files at the root of your classpath and are called format.properties, exceptions.properties, and windows.properties respectively.

The next example shows arguments passed to the message lookup; these arguments will be converted into Strings and inserted into placeholders in the lookup message.

<beans>

*<!-- this MessageSource is being used in a web application -->*

<bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">

<property name="basename" value="exceptions"/>

</bean>

*<!-- lets inject the above MessageSource into this POJO -->*

<bean id="example" class="com.foo.Example">

<property name="messages" ref="messageSource"/>

</bean>

</beans>

**public** **class** **Example** {

**private** MessageSource messages;

**public** **void** setMessages(MessageSource messages) {

this.messages = messages;

}

**public** **void** execute() {

String message = this.messages.getMessage("argument.required",

**new** Object **[]** {"userDao"}, "Required", null);

System.out.println(message);

}

}

The resulting output from the invocation of the execute() method will be…​

The userDao argument is required.

With regard to internationalization (i18n), Spring’s various MessageSource implementations follow the same locale resolution and fallback rules as the standard JDK ResourceBundle. In short, and continuing with the example messageSource defined previously, if you want to resolve messages against the British (en-GB) locale, you would create files called format\_en\_GB.properties, exceptions\_en\_GB.properties, and windows\_en\_GB.properties respectively.

Typically, locale resolution is managed by the surrounding environment of the application. In this example, the locale against which (British) messages will be resolved is specified manually.

# 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);

}

The resulting output from the running of the above program will be…​

Ebagum lad, the 'userDao' argument is required, I say, required.

You can also use the MessageSourceAware interface to acquire a reference to any MessageSource that has been defined. Any bean that is defined in an ApplicationContext that implements the MessageSourceAware interface is injected with the application context’s MessageSource when the bean is created and configured.

|  |  |
| --- | --- |
|  | *As an alternative to ResourceBundleMessageSource, Spring provides a ReloadableResourceBundleMessageSourceclass. This variant supports the same bundle file format but is more flexible than the standard JDK basedResourceBundleMessageSource implementation.* In particular, it allows for reading files from any Spring resource location (not just from the classpath) and supports hot reloading of bundle property files (while efficiently caching them in between). Check out the ReloadableResourceBundleMessageSource javadocs for details. |

1.15.2. Standard and custom events

Event handling in the ApplicationContext is provided through the ApplicationEvent class and ApplicationListener interface. If a bean that implements the ApplicationListener interface is deployed into the context, every time an ApplicationEvent gets published to the ApplicationContext, that bean is notified. Essentially, this is the standard *Observer* design pattern.

|  |  |
| --- | --- |
|  | As of Spring 4.2, the event infrastructure has been significantly improved and offer an [annotation-based model](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#context-functionality-events-annotation)as well as the ability to publish any arbitrary event, that is an object that does not necessarily extend from ApplicationEvent. When such an object is published we wrap it in an event for you. |

Spring provides the following standard events:

| *Table 7. Built-in Events* | |
| --- | --- |
| **Event** | **Explanation** |
| ContextRefreshedEvent | Published when the ApplicationContext is initialized or refreshed, for example, using the refresh() method on the ConfigurableApplicationContext interface. "Initialized" here means that all beans are loaded, post-processor beans are detected and activated, singletons are pre-instantiated, and the ApplicationContext object is ready for use. As long as the context has not been closed, a refresh can be triggered multiple times, provided that the chosen ApplicationContext actually supports such "hot" refreshes. For example, XmlWebApplicationContext supports hot refreshes, but GenericApplicationContext does not. |
| ContextStartedEvent | Published when the ApplicationContext is started, using the start() method on the ConfigurableApplicationContextinterface. "Started" here means that all Lifecycle beans receive an explicit start signal. Typically this signal is used to restart beans after an explicit stop, but it may also be used to start components that have not been configured for autostart , for example, components that have not already started on initialization. |
| ContextStoppedEvent | Published when the ApplicationContext is stopped, using the stop() method on the ConfigurableApplicationContextinterface. "Stopped" here means that all Lifecycle beans receive an explicit stop signal. A stopped context may be restarted through a start() call. |
| ContextClosedEvent | Published when the ApplicationContext is closed, using the close() method on the ConfigurableApplicationContextinterface. "Closed" here means that all singleton beans are destroyed. A closed context reaches its end of life; it cannot be refreshed or restarted. |
| RequestHandledEvent | A web-specific event telling all beans that an HTTP request has been serviced. This event is published *after* the request is complete. This event is only applicable to web applications using Spring’s DispatcherServlet. |

You can also create and publish your own custom events. This example demonstrates a simple class that extends Spring’s ApplicationEvent base class:

**public** **class** **BlackListEvent** **extends** ApplicationEvent {

**private** **final** String address;

**private** **final** String test;

**public** BlackListEvent(Object source, String address, String test) {

super(source);

this.address = address;

this.test = test;

}

*// accessor and other methods...*

}

To publish a custom ApplicationEvent, call the publishEvent() method on an ApplicationEventPublisher. Typically this is done by creating a class that implements ApplicationEventPublisherAware and registering it as a Spring bean. The following example demonstrates such a class:

**public** **class** **EmailService** **implements** ApplicationEventPublisherAware {

**private** List<String> blackList;

**private** ApplicationEventPublisher publisher;

**public** **void** setBlackList(List<String> blackList) {

this.blackList = blackList;

}

**public** **void** setApplicationEventPublisher(ApplicationEventPublisher publisher) {

this.publisher = publisher;

}

**public** **void** sendEmail(String address, String text) {

**if** (blackList.contains(address)) {

BlackListEvent event = **new** BlackListEvent(this, address, text);

publisher.publishEvent(event);

**return**;

}

*// send email...*

}

}

At configuration time, the Spring container will detect that EmailService implements ApplicationEventPublisherAware and will automatically call setApplicationEventPublisher(). In reality, the parameter passed in will be the Spring container itself; you’re simply interacting with the application context via its ApplicationEventPublisher interface.

To receive the custom ApplicationEvent, create a class that implements ApplicationListener and register it as a Spring bean. The following example demonstrates such a class:

**public** **class** **BlackListNotifier** **implements** ApplicationListener<BlackListEvent> {

**private** String notificationAddress;

**public** **void** setNotificationAddress(String notificationAddress) {

this.notificationAddress = notificationAddress;

}

**public** **void** onApplicationEvent(BlackListEvent event) {

*// notify appropriate parties via notificationAddress...*

}

}

Notice that ApplicationListener is generically parameterized with the type of your custom event, BlackListEvent. This means that the onApplicationEvent() method can remain type-safe, avoiding any need for downcasting. You may register as many event listeners as you wish, but note that by default event listeners receive events synchronously. This means the publishEvent() method blocks until all listeners have finished processing the event. One advantage of this synchronous and single-threaded approach is that when a listener receives an event, it operates inside the transaction context of the publisher if a transaction context is available. If another strategy for event publication becomes necessary, refer to the javadoc for Spring’sApplicationEventMulticaster interface.

The following example shows the bean definitions used to register and configure each of the classes above:

<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>

Putting it all together, when the sendEmail() method of the emailService bean is called, if there are any emails that should be blacklisted, a custom event of type BlackListEvent is published. The blackListNotifier bean is registered as anApplicationListener and thus receives the BlackListEvent, at which point it can notify appropriate parties.

|  |  |
| --- | --- |
|  | Spring’s eventing mechanism is designed for simple communication between Spring beans within the same application context. However, for more sophisticated enterprise integration needs, the separately-maintained[Spring Integration](https://projects.spring.io/spring-integration/) project provides complete support for building lightweight, [pattern-oriented](http://www.enterpriseintegrationpatterns.com/), event-driven architectures that build upon the well-known Spring programming model. |

Annotation-based event listeners

As of Spring 4.2, an event listener can be registered on any public method of a managed bean via the EventListenerannotation. The BlackListNotifier can be rewritten as follows:

**public** **class** **BlackListNotifier** {

**private** String notificationAddress;

**public** **void** setNotificationAddress(String notificationAddress) {

this.notificationAddress = notificationAddress;

}

@EventListener

**public** **void** processBlackListEvent(BlackListEvent event) {

*// notify appropriate parties via notificationAddress...*

}

}

As you can see above, the method signature once again declares the event type it listens to, but this time with a flexible name and without implementing a specific listener interface. The event type can also be narrowed through generics as long as the actual event type resolves your generic parameter in its implementation hierarchy.

If your method should listen to several events or if you want to define it with no parameter at all, the event type(s) can also be specified on the annotation itself:

@EventListener({ContextStartedEvent.class, ContextRefreshedEvent.class})

**public** **void** handleContextStart() {

...

}

It is also possible to add additional runtime filtering via the condition attribute of the annotation that defines a [SpELexpression](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#expressions) that should match to actually invoke the method for a particular event.

For instance, our notifier can be rewritten to be only invoked if the test attribute of the event is equal to foo:

@EventListener(condition = "#blEvent.test == 'foo'")

**public** **void** processBlackListEvent(BlackListEvent blEvent) {

*// notify appropriate parties via notificationAddress...*

}

Each SpEL expression evaluates again a dedicated context. The next table lists the items made available to the context so one can use them for conditional event processing:

| *Table 8. Event SpEL available metadata* | | | |
| --- | --- | --- | --- |
| **Name** | **Location** | **Description** | **Example** |
| Event | root object | The actual ApplicationEvent | #root.event |
| Arguments array | root object | The arguments (as array) used for invoking the target | #root.args[0] |
| *Argument name* | evaluation context | Name of any of the method arguments. If for some reason the names are not available (e.g. no debug information), the argument names are also available under the #a<#arg> where *#arg* stands for the argument index (starting from 0). | #blEvent or #a0 (one can also use #p0 or #p<#arg>notation as an alias). |

Note that #root.event allows you to access to the underlying event, even if your method signature actually refers to an arbitrary object that was published.

If you need to publish an event as the result of processing another, just change the method signature to return the event that should be published, something like:

@EventListener

**public** ListUpdateEvent handleBlackListEvent(BlackListEvent event) {

*// notify appropriate parties via notificationAddress and*

*// then publish a ListUpdateEvent...*

}

|  |  |
| --- | --- |
|  | This feature is not supported for [asynchronous listeners](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#context-functionality-events-async). |

This new method will publish a new ListUpdateEvent for every BlackListEvent handled by the method above. If you need to publish several events, just return a Collection of events instead.

Asynchronous Listeners

If you want a particular listener to process events asynchronously, simply reuse the [regular @Async support](https://docs.spring.io/spring/docs/current/spring-framework-reference/integration.html#scheduling-annotation-support-async):

@EventListener

@Async

**public** **void** processBlackListEvent(BlackListEvent event) {

*// BlackListEvent is processed in a separate thread*

}

Be aware of the following limitations when using asynchronous events:

1. If the event listener throws an Exception it will not be propagated to the caller, check AsyncUncaughtExceptionHandler for more details.
2. Such event listener cannot send replies. If you need to send another event as the result of the processing, inject ApplicationEventPublisher to send the event manually.

Ordering listeners

If you need the listener to be invoked before another one, just add the @Order annotation to the method declaration:

@EventListener

@Order(42)

**public** **void** processBlackListEvent(BlackListEvent event) {

*// notify appropriate parties via notificationAddress...*

}

Generic events

You may also use generics to further define the structure of your event. Consider an EntityCreatedEvent<T> where T is the type of the actual entity that got created. You can create the following listener definition to only receive EntityCreatedEvent for a Person:

@EventListener

**public** **void** onPersonCreated(EntityCreatedEvent<Person> event) {

...

}

Due to type erasure, this will only work if the event that is fired resolves the generic parameter(s) on which the event listener filters on (that is something like class PersonCreatedEvent extends EntityCreatedEvent<Person> { …​ }).

In certain circumstances, this may become quite tedious if all events follow the same structure (as it should be the case for the event above). In such a case, you can implement ResolvableTypeProvider to *guide* the framework beyond what the runtime environment provides:

**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()));

}

}

|  |  |
| --- | --- |
|  | This works not only for ApplicationEvent but any arbitrary object that you’d send as an event. |

1.15.3. Convenient access to low-level resources

For optimal usage and understanding of application contexts, users should generally familiarize themselves with Spring’s Resource abstraction, as described in the chapter [Resources](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#resources).

An application context is a ResourceLoader, which can be used to load Resources. A Resource is essentially a more feature rich version of the JDK class java.net.URL, in fact, the implementations of the Resource wrap an instance of java.net.URL where appropriate. A Resource can obtain low-level resources from almost any location in a transparent fashion, including from the classpath, a filesystem location, anywhere describable with a standard URL, and some other variations. If the resource location string is a simple path without any special prefixes, where those resources come from is specific and appropriate to the actual application context type.

You can configure a bean deployed into the application context to implement the special callback interface, ResourceLoaderAware, to be automatically called back at initialization time with the application context itself passed in as theResourceLoader. You can also expose properties of type Resource, to be used to access static resources; they will be injected into it like any other properties. You can specify those Resource properties as simple String paths, and rely on a special JavaBean PropertyEditor that is automatically registered by the context, to convert those text strings to actual Resourceobjects when the bean is deployed.

The location path or paths supplied to an ApplicationContext constructor are actually resource strings, and in simple form are treated appropriately to the specific context implementation. ClassPathXmlApplicationContext treats a simple location path as a classpath location. You can also use location paths (resource strings) with special prefixes to force loading of definitions from the classpath or a URL, regardless of the actual context type.

1.15.4. Convenient ApplicationContext instantiation for web applications

You can create ApplicationContext instances declaratively by using, for example, a ContextLoader. Of course you can also create ApplicationContext instances programmatically by using one of the ApplicationContext implementations.

You can register an ApplicationContext using the ContextLoaderListener as follows:

<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>

The listener inspects the contextConfigLocation parameter. If the parameter does not exist, the listener uses /WEB-INF/applicationContext.xml as a default. When the parameter *does* exist, the listener separates the String by using predefined delimiters (comma, semicolon and whitespace) and uses the values as locations where application contexts will be searched. Ant-style path patterns are supported as well. Examples are /WEB-INF/\*Context.xml for all files with names ending with "Context.xml", residing in the "WEB-INF" directory, and /WEB-INF/\*\*/\*Context.xml, for all such files in any subdirectory of "WEB-INF".

1.15.5. Deploying a Spring ApplicationContext as a Java EE RAR file

It is possible to deploy a Spring ApplicationContext as a RAR file, encapsulating the context and all of its required bean classes and library JARs in a Java EE RAR deployment unit. This is the equivalent of bootstrapping a standalone ApplicationContext, just hosted in Java EE environment, being able to access the Java EE servers facilities. RAR deployment is more natural alternative to scenario of deploying a headless WAR file, in effect, a WAR file without any HTTP entry points that is used only for bootstrapping a Spring ApplicationContext in a Java EE environment.

RAR deployment is ideal for application contexts that do not need HTTP entry points but rather consist only of message endpoints and scheduled jobs. Beans in such a context can use application server resources such as the JTA transaction manager and JNDI-bound JDBC DataSources and JMS ConnectionFactory instances, and may also register with the platform’s JMX server - all through Spring’s standard transaction management and JNDI and JMX support facilities. Application components can also interact with the application server’s JCA WorkManager through Spring’s TaskExecutor abstraction.

Check out the javadoc of the [SpringContextResourceAdapter](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/jca/context/SpringContextResourceAdapter.html) class for the configuration details involved in RAR deployment.

*For a simple deployment of a Spring ApplicationContext as a Java EE RAR file:* package all application classes into a RAR file, which is a standard JAR file with a different file extension. Add all required library JARs into the root of the RAR archive. Add a "META-INF/ra.xml" deployment descriptor (as shown in SpringContextResourceAdapters javadoc) and the corresponding Spring XML bean definition file(s) (typically "META-INF/applicationContext.xml"), and drop the resulting RAR file into your application server’s deployment directory.

|  |  |
| --- | --- |
|  | Such RAR deployment units are usually self-contained; they do not expose components to the outside world, not even to other modules of the same application. Interaction with a RAR-based ApplicationContext usually occurs through JMS destinations that it shares with other modules. A RAR-based ApplicationContext may also, for example, schedule some jobs, reacting to new files in the file system (or the like). If it needs to allow synchronous access from the outside, it could for example export RMI endpoints, which of course may be used by other application modules on the same machine. |

1.16. The BeanFactory

The BeanFactory provides the underlying basis for Spring’s IoC functionality but it is only used directly in integration with other third-party frameworks and is now largely historical in nature for most users of Spring. The BeanFactory and related interfaces, such as BeanFactoryAware, InitializingBean, DisposableBean, are still present in Spring for the purposes of backward compatibility with the large number of third-party frameworks that integrate with Spring. Often third-party components that can not use more modern equivalents such as @PostConstruct or @PreDestroy in order to avoid a dependency on JSR-250.

This section provides additional background into the differences between the BeanFactory and ApplicationContext and how one might access the IoC container directly through a classic singleton lookup.

1.16.1. BeanFactory or ApplicationContext?

Use an ApplicationContext unless you have a good reason for not doing so.

Because the ApplicationContext includes all functionality of the BeanFactory, it is generally recommended over the BeanFactory, except for a few situations such as in embedded applications running on resource-constrained devices where memory consumption might be critical and a few extra kilobytes might make a difference. However, for most typical enterprise applications and systems, the ApplicationContext is what you will want to use. Spring makes *heavy* use of the [BeanPostProcessor extension point](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-extension-bpp) (to effect proxying and so on). If you use only a plain BeanFactory, a fair amount of support such as transactions and AOP will not take effect, at least not without some extra steps on your part. This situation could be confusing because nothing is actually wrong with the configuration.

The following table lists features provided by the BeanFactory and ApplicationContext interfaces and implementations.

| *Table 9. Feature Matrix* | | |
| --- | --- | --- |
| **Feature** | BeanFactory | ApplicationContext |
| Bean instantiation/wiring | Yes | Yes |
| Automatic BeanPostProcessorregistration | No | Yes |
| Automatic BeanFactoryPostProcessorregistration | No | Yes |
| Convenient MessageSource access (for i18n) | No | Yes |
| ApplicationEvent publication | No | Yes |

To explicitly register a bean post-processor with a BeanFactory implementation, you need to write code like this:

DefaultListableBeanFactory factory = **new** DefaultListableBeanFactory();

*// populate the factory with bean definitions*

*// now register any needed BeanPostProcessor instances*

MyBeanPostProcessor postProcessor = **new** MyBeanPostProcessor();

factory.addBeanPostProcessor(postProcessor);

*// now start using the factory*

To explicitly register a BeanFactoryPostProcessor when using a BeanFactory implementation, you must write code like this:

DefaultListableBeanFactory factory = **new** DefaultListableBeanFactory();

XmlBeanDefinitionReader reader = **new** XmlBeanDefinitionReader(factory);

reader.loadBeanDefinitions(**new** FileSystemResource("beans.xml"));

*// bring in some property values from a Properties file*

PropertyPlaceholderConfigurer cfg = **new** PropertyPlaceholderConfigurer();

cfg.setLocation(**new** FileSystemResource("jdbc.properties"));

*// now actually do the replacement*

cfg.postProcessBeanFactory(factory);

In both cases, the explicit registration step is inconvenient, which is one reason why the various ApplicationContextimplementations are preferred above plain BeanFactory implementations in the vast majority of Spring-backed applications, especially when using BeanFactoryPostProcessors and BeanPostProcessors. These mechanisms implement important functionality such as property placeholder replacement and AOP.

2. Resources

2.1. Introduction

Java’s standard java.net.URL class and standard handlers for various URL prefixes unfortunately are not quite adequate enough for all access to low-level resources. For example, there is no standardized URL implementation that may be used to access a resource that needs to be obtained from the classpath, or relative to a ServletContext. While it is possible to register new handlers for specialized URL prefixes (similar to existing handlers for prefixes such as http:), this is generally quite complicated, and the URL interface still lacks some desirable functionality, such as a method to check for the existence of the resource being pointed to.

2.2. The Resource interface

Spring’s Resource interface is meant to be a more capable interface for abstracting access to low-level resources.

**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;

}

Some of the most important methods from the Resource interface are:

* getInputStream(): locates and opens the resource, returning an InputStream for reading from the resource. It is expected that each invocation returns a fresh InputStream. It is the responsibility of the caller to close the stream.
* exists(): returns a boolean indicating whether this resource actually exists in physical form.
* isOpen(): returns a boolean indicating whether this resource represents a handle with an open stream. If true, the InputStream cannot be read multiple times, and must be read once only and then closed to avoid resource leaks. Will be false for all usual resource implementations, with the exception of InputStreamResource.
* getDescription(): returns a description for this resource, to be used for error output when working with the resource. This is often the fully qualified file name or the actual URL of the resource.

Other methods allow you to obtain an actual URL or File object representing the resource (if the underlying implementation is compatible, and supports that functionality).

The Resource abstraction is used extensively in Spring itself, as an argument type in many method signatures when a resource is needed. Other methods in some Spring APIs (such as the constructors to various ApplicationContext implementations), take a String which in unadorned or simple form is used to create a Resource appropriate to that context implementation, or via special prefixes on the String path, allow the caller to specify that a specific Resource implementation must be created and used.

While the Resource interface is used a lot with Spring and by Spring, it’s actually very useful to use as a general utility class by itself in your own code, for access to resources, even when your code doesn’t know or care about any other parts of Spring. While this couples your code to Spring, it really only couples it to this small set of utility classes, which are serving as a more capable replacement for URL, and can be considered equivalent to any other library you would use for this purpose.

It is important to note that the Resource abstraction does not replace functionality: it wraps it where possible. For example, a UrlResource wraps a URL, and uses the wrapped URL to do its work.

2.3. Built-in Resource implementations

There are a number of Resource implementations that come supplied straight out of the box in Spring:

2.3.1. UrlResource

The UrlResource wraps a java.net.URL, and may be used to access any object that is normally accessible via a URL, such as files, an HTTP target, an FTP target, etc. All URLs have a standardized String representation, such that appropriate standardized prefixes are used to indicate one URL type from another. This includes file: for accessing filesystem paths, http: for accessing resources via the HTTP protocol, ftp: for accessing resources via FTP, etc.

A UrlResource is created by Java code explicitly using the UrlResource constructor, but will often be created implicitly when you call an API method which takes a String argument which is meant to represent a path. For the latter case, a JavaBeansPropertyEditor will ultimately decide which type of Resource to create. If the path string contains a few well-known (to it, that is) prefixes such as classpath:, it will create an appropriate specialized Resource for that prefix. However, if it doesn’t recognize the prefix, it will assume the this is just a standard URL string, and will create a UrlResource.

2.3.2. ClassPathResource

This class represents a resource which should be obtained from the classpath. This uses either the thread context class loader, a given class loader, or a given class for loading resources.

This Resource implementation supports resolution as java.io.File if the class path resource resides in the file system, but not for classpath resources which reside in a jar and have not been expanded (by the servlet engine, or whatever the environment is) to the filesystem. To address this the various Resource implementations always support resolution as a java.net.URL.

A ClassPathResource is created by Java code explicitly using the ClassPathResource constructor, but will often be created implicitly when you call an API method which takes a String argument which is meant to represent a path. For the latter case, a JavaBeans PropertyEditor will recognize the special prefix classpath: on the string path, and create a ClassPathResource in that case.

2.3.3. FileSystemResource

This is a Resource implementation for java.io.File handles. It obviously supports resolution as a File, and as a URL.

2.3.4. ServletContextResource

This is a Resource implementation for ServletContext resources, interpreting relative paths within the relevant web application’s root directory.

This always supports stream access and URL access, but only allows java.io.File access when the web application archive is expanded and the resource is physically on the filesystem. Whether or not it’s expanded and on the filesystem like this, or accessed directly from the JAR or somewhere else like a DB (it’s conceivable) is actually dependent on the Servlet container.

2.3.5. InputStreamResource

A Resource implementation for a given InputStream. This should only be used if no specific Resource implementation is applicable. In particular, prefer ByteArrayResource or any of the file-based Resource implementations where possible.

In contrast to other Resource implementations, this is a descriptor for an *already* opened resource - therefore returning truefrom isOpen(). Do not use it if you need to keep the resource descriptor somewhere, or if you need to read a stream multiple times.

2.3.6. ByteArrayResource

This is a Resource implementation for a given byte array. It creates a ByteArrayInputStream for the given byte array.

It’s useful for loading content from any given byte array, without having to resort to a single-use InputStreamResource.

2.4. The ResourceLoader

The ResourceLoader interface is meant to be implemented by objects that can return (i.e. load) Resource instances.

**public** **interface** **ResourceLoader** {

Resource getResource(String location);

}

All application contexts implement the ResourceLoader interface, and therefore all application contexts may be used to obtain Resource instances.

When you call getResource() on a specific application context, and the location path specified doesn’t have a specific prefix, you will get back a Resource type that is appropriate to that particular application context. For example, assume the following snippet of code was executed against a ClassPathXmlApplicationContext instance:

Resource template = ctx.getResource("some/resource/path/myTemplate.txt");

What would be returned would be a ClassPathResource; if the same method was executed against a FileSystemXmlApplicationContext instance, you’d get back a FileSystemResource. For a WebApplicationContext, you’d get back a ServletContextResource, and so on.

As such, you can load resources in a fashion appropriate to the particular application context.

On the other hand, you may also force ClassPathResource to be used, regardless of the application context type, by specifying the special classpath: prefix:

Resource template = ctx.getResource("classpath:some/resource/path/myTemplate.txt");

Similarly, one can force a UrlResource to be used by specifying any of the standard java.net.URL prefixes:

Resource template = ctx.getResource("file:///some/resource/path/myTemplate.txt");

Resource template = ctx.getResource("http://myhost.com/resource/path/myTemplate.txt");

The following table summarizes the strategy for converting Strings to Resources:

| *Table 10. Resource strings* | | |
| --- | --- | --- |
| **Prefix** | **Example** | **Explanation** |
| classpath: | classpath:com/myapp/config.xml | Loaded from the classpath. |
| file: | [file:///data/config.xml](file:///\\data\config.xml) | Loaded as a URL, from the filesystem. [[3](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#_footnote_3)] |
| http: | <http://myserver/logo.png> | Loaded as a URL. |
| (none) | /data/config.xml | Depends on the underlying ApplicationContext. |

2.5. The ResourceLoaderAware interface

The ResourceLoaderAware interface is a special marker interface, identifying objects that expect to be provided with a ResourceLoader reference.

**public** **interface** **ResourceLoaderAware** {

**void** setResourceLoader(ResourceLoader resourceLoader);

}

When a class implements ResourceLoaderAware and is deployed into an application context (as a Spring-managed bean), it is recognized as ResourceLoaderAware by the application context. The application context will then invoke thesetResourceLoader(ResourceLoader), supplying itself as the argument (remember, all application contexts in Spring implement the ResourceLoader interface).

Of course, since an ApplicationContext is a ResourceLoader, the bean could also implement the ApplicationContextAwareinterface and use the supplied application context directly to load resources, but in general, it’s better to use the specializedResourceLoader interface if that’s all that’s needed. The code would just be coupled to the resource loading interface, which can be considered a utility interface, and not the whole Spring ApplicationContext interface.

As of Spring 2.5, you can rely upon autowiring of the ResourceLoader as an alternative to implementing the ResourceLoaderAware interface. The "traditional" constructor and byType autowiring modes (as described in [Autowiring collaborators](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-autowire)) are now capable of providing a dependency of type ResourceLoader for either a constructor argument or setter method parameter respectively. For more flexibility (including the ability to autowire fields and multiple parameter methods), consider using the new annotation-based autowiring features. In that case, the ResourceLoader will be autowired into a field, constructor argument, or method parameter that is expecting the ResourceLoader type as long as the field, constructor, or method in question carries the @Autowired annotation. For more information, see [@Autowired](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-autowired-annotation).

2.6. Resources as dependencies

If the bean itself is going to determine and supply the resource path through some sort of dynamic process, it probably makes sense for the bean to use the ResourceLoader interface to load resources. Consider as an example the loading of a template of some sort, where the specific resource that is needed depends on the role of the user. If the resources are static, it makes sense to eliminate the use of the ResourceLoader interface completely, and just have the bean expose the Resource properties it needs, and expect that they will be injected into it.

What makes it trivial to then inject these properties, is that all application contexts register and use a special JavaBeans PropertyEditor which can convert String paths to Resource objects. So if myBean has a template property of type Resource, it can be configured with a simple string for that resource, as follows:

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

<property name="template" value="some/resource/path/myTemplate.txt"/>

</bean>

Note that the resource path has no prefix, so because the application context itself is going to be used as the ResourceLoader, the resource itself will be loaded via a ClassPathResource, FileSystemResource, or ServletContextResource (as appropriate) depending on the exact type of the context.

If there is a need to force a specific Resource type to be used, then a prefix may be used. The following two examples show how to force a ClassPathResource and a UrlResource (the latter being used to access a filesystem file).

<property name="template" value="classpath:some/resource/path/myTemplate.txt">

<property name="template" value="file:///some/resource/path/myTemplate.txt"/>

2.7. Application contexts and Resource paths

2.7.1. Constructing application contexts

An application context constructor (for a specific application context type) generally takes a string or array of strings as the location path(s) of the resource(s) such as XML files that make up the definition of the context.

When such a location path doesn’t have a prefix, the specific Resource type built from that path and used to load the bean definitions, depends on and is appropriate to the specific application context. For example, if you create aClassPathXmlApplicationContext as follows:

ApplicationContext ctx = **new** ClassPathXmlApplicationContext("conf/appContext.xml");

The bean definitions will be loaded from the classpath, as a ClassPathResource will be used. But if you create a FileSystemXmlApplicationContext as follows:

ApplicationContext ctx =

**new** FileSystemXmlApplicationContext("conf/appContext.xml");

The bean definition will be loaded from a filesystem location, in this case relative to the current working directory.

Note that the use of the special classpath prefix or a standard URL prefix on the location path will override the default type of Resource created to load the definition. So this FileSystemXmlApplicationContext…​

ApplicationContext ctx =

**new** FileSystemXmlApplicationContext("classpath:conf/appContext.xml");

1. will actually load its bean definitions from the classpath. However, it is still a FileSystemXmlApplicationContext. If it is subsequently used as a ResourceLoader, any unprefixed paths will still be treated as filesystem paths.

Constructing ClassPathXmlApplicationContext instances - shortcuts

The ClassPathXmlApplicationContext exposes a number of constructors to enable convenient instantiation. The basic idea is that one supplies merely a string array containing just the filenames of the XML files themselves (without the leading path information), and one *also* supplies a Class; the ClassPathXmlApplicationContext will derive the path information from the supplied class.

An example will hopefully make this clear. Consider a directory layout that looks like this:

com/

foo/

services.xml

daos.xml

MessengerService.class

A ClassPathXmlApplicationContext instance composed of the beans defined in the 'services.xml' and 'daos.xml' could be instantiated like so…​

ApplicationContext ctx = **new** ClassPathXmlApplicationContext(

**new** String**[]** {"services.xml", "daos.xml"}, MessengerService.class);

Please do consult the ClassPathXmlApplicationContext javadocs for details on the various constructors.

2.7.2. Wildcards in application context constructor resource paths

The resource paths in application context constructor values may be a simple path (as shown above) which has a one-to-one mapping to a target Resource, or alternately may contain the special "classpath\*:" prefix and/or internal Ant-style regular expressions (matched using Spring’s PathMatcher utility). Both of the latter are effectively wildcards

One use for this mechanism is when doing component-style application assembly. All components can 'publish' context definition fragments to a well-known location path, and when the final application context is created using the same path prefixed via classpath\*:, all component fragments will be picked up automatically.

Note that this wildcarding is specific to use of resource paths in application context constructors (or when using the PathMatcher utility class hierarchy directly), and is resolved at construction time. It has nothing to do with the Resource type itself. It’s not possible to use the classpath\*: prefix to construct an actual Resource, as a resource points to just one resource at a time.

Ant-style Patterns

When the path location contains an Ant-style pattern, for example:

/WEB-INF/\*-context.xml

com/mycompany/\*\*/applicationContext.xml

file:C:/some/path/\*-context.xml

classpath:com/mycompany/\*\*/applicationContext.xml

The resolver follows a more complex but defined procedure to try to resolve the wildcard. It produces a Resource for the path up to the last non-wildcard segment and obtains a URL from it. If this URL is not a jar: URL or container-specific variant (e.g. zip: in WebLogic, wsjar in WebSphere, etc.), then a java.io.File is obtained from it and used to resolve the wildcard by traversing the filesystem. In the case of a jar URL, the resolver either gets a java.net.JarURLConnection from it or manually parses the jar URL and then traverses the contents of the jar file to resolve the wildcards.

Implications on portability

If the specified path is already a file URL (either explicitly, or implicitly because the base ResourceLoader is a filesystem one, then wildcarding is guaranteed to work in a completely portable fashion.

If the specified path is a classpath location, then the resolver must obtain the last non-wildcard path segment URL via a Classloader.getResource() call. Since this is just a node of the path (not the file at the end) it is actually undefined (in theClassLoader javadocs) exactly what sort of a URL is returned in this case. In practice, it is always a java.io.File representing the directory, where the classpath resource resolves to a filesystem location, or a jar URL of some sort, where the classpath resource resolves to a jar location. Still, there is a portability concern on this operation.

If a jar URL is obtained for the last non-wildcard segment, the resolver must be able to get a java.net.JarURLConnection from it, or manually parse the jar URL, to be able to walk the contents of the jar, and resolve the wildcard. This will work in most environments, but will fail in others, and it is strongly recommended that the wildcard resolution of resources coming from jars be thoroughly tested in your specific environment before you rely on it.

The classpath\*: prefix

When constructing an XML-based application context, a location string may use the special classpath\*: prefix:

ApplicationContext ctx =

**new** ClassPathXmlApplicationContext("classpath\*:conf/appContext.xml");

This special prefix specifies that all classpath resources that match the given name must be obtained (internally, this essentially happens via a ClassLoader.getResources(…​) call), and then merged to form the final application context definition.

|  |  |
| --- | --- |
|  | The wildcard classpath relies on the getResources() method of the underlying classloader. As most application servers nowadays supply their own classloader implementation, the behavior might differ especially when dealing with jar files. A simple test to check if classpath\* works is to use the classloader to load a file from within a jar on the classpath: getClass().getClassLoader().getResources("<someFileInsideTheJar>"). Try this test with files that have the same name but are placed inside two different locations. In case an inappropriate result is returned, check the application server documentation for settings that might affect the classloader behavior. |

The classpath\*: prefix can also be combined with a PathMatcher pattern in the rest of the location path, for example classpath\*:META-INF/\*-beans.xml. In this case, the resolution strategy is fairly simple: a ClassLoader.getResources() call is used on the last non-wildcard path segment to get all the matching resources in the class loader hierarchy, and then off each resource the same PathMatcher resolution strategy described above is used for the wildcard subpath.

Other notes relating to wildcards

Please note that classpath\*: when combined with Ant-style patterns will only work reliably with at least one root directory before the pattern starts, unless the actual target files reside in the file system. This means that a pattern like classpath\*:\*.xmlmight not retrieve files from the root of jar files but rather only from the root of expanded directories.

Spring’s ability to retrieve classpath entries originates from the JDK’s ClassLoader.getResources() method which only returns file system locations for a passed-in empty string (indicating potential roots to search). Spring evaluates URLClassLoaderruntime configuration and the "java.class.path" manifest in jar files as well but this is not guaranteed to lead to portable behavior.

|  |  |
| --- | --- |
|  | The scanning of classpath packages requires the presence of corresponding directory entries in the classpath. When you build JARs with Ant, make sure that you do *not* activate the files-only switch of the JAR task. Also, classpath directories may not get exposed based on security policies in some environments, e.g. standalone apps on JDK 1.7.0\_45 and higher (which requires 'Trusted-Library' setup in your manifests; see[http://stackoverflow.com/questions/19394570/java-jre-7u45-breaks-classloader-getresources](https://stackoverflow.com/questions/19394570/java-jre-7u45-breaks-classloader-getresources)).  On JDK 9’s module path (Jigsaw), Spring’s classpath scanning generally works as expected. Putting resources into a dedicated directory is highly recommendable here as well, avoiding the aforementioned portability problems with searching the jar file root level. |

Ant-style patterns with classpath: resources are not guaranteed to find matching resources if the root package to search is available in multiple class path locations. This is because a resource such as

com/mycompany/package1/service-context.xml

may be in only one location, but when a path such as

classpath:com/mycompany/\*\*/service-context.xml

is used to try to resolve it, the resolver will work off the (first) URL returned by getResource("com/mycompany");. If this base package node exists in multiple classloader locations, the actual end resource may not be underneath. Therefore, preferably, use " `classpath\*:`" with the same Ant-style pattern in such a case, which will search all class path locations that contain the root package.

2.7.3. FileSystemResource caveats

A FileSystemResource that is not attached to a FileSystemApplicationContext (that is, a FileSystemApplicationContext is not the actual ResourceLoader) will treat absolute vs. relative paths as you would expect. Relative paths are relative to the current working directory, while absolute paths are relative to the root of the filesystem.

For backwards compatibility (historical) reasons however, this changes when the FileSystemApplicationContext is the ResourceLoader. The FileSystemApplicationContext simply forces all attached FileSystemResource instances to treat all location paths as relative, whether they start with a leading slash or not. In practice, this means the following are equivalent:

ApplicationContext ctx =

**new** FileSystemXmlApplicationContext("conf/context.xml");

ApplicationContext ctx =

**new** FileSystemXmlApplicationContext("/conf/context.xml");

As are the following: (Even though it would make sense for them to be different, as one case is relative and the other absolute.)

FileSystemXmlApplicationContext ctx = ...;

ctx.getResource("some/resource/path/myTemplate.txt");

FileSystemXmlApplicationContext ctx = ...;

ctx.getResource("/some/resource/path/myTemplate.txt");

In practice, if true absolute filesystem paths are needed, it is better to forgo the use of absolute paths with FileSystemResource/ FileSystemXmlApplicationContext, and just force the use of a UrlResource, by using the file: URL prefix.

*// actual context type doesn't matter, the Resource will always be UrlResource*

ctx.getResource("file:///some/resource/path/myTemplate.txt");

*// force this FileSystemXmlApplicationContext to load its definition via a UrlResource*

ApplicationContext ctx =

**new** FileSystemXmlApplicationContext("file:///conf/context.xml");

3. Validation, Data Binding, and Type Conversion

3.1. Introduction

JSR-303/JSR-349 Bean Validation

Spring Framework 4.0 supports Bean Validation 1.0 (JSR-303) and Bean Validation 1.1 (JSR-349) in terms of setup support, also adapting it to Spring’s Validator interface.

An application can choose to enable Bean Validation once globally, as described in [Spring Validation](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#validation-beanvalidation), and use it exclusively for all validation needs.

An application can also register additional Spring Validator instances per DataBinder instance, as described in [Configuring a DataBinder](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#validation-binder). This may be useful for plugging in validation logic without the use of annotations.

There are pros and cons for considering validation as business logic, and Spring offers a design for validation (and data binding) that does not exclude either one of them. Specifically validation should not be tied to the web tier, should be easy to localize and it should be possible to plug in any validator available. Considering the above, Spring has come up with a Validatorinterface that is both basic and eminently usable in every layer of an application.

Data binding is useful for allowing user input to be dynamically bound to the domain model of an application (or whatever objects you use to process user input). Spring provides the so-called DataBinder to do exactly that. The Validator and theDataBinder make up the validation package, which is primarily used in but not limited to the MVC framework.

The BeanWrapper is a fundamental concept in the Spring Framework and is used in a lot of places. However, you probably will not have the need to use the BeanWrapper directly. Because this is reference documentation however, we felt that some explanation might be in order. We will explain the BeanWrapper in this chapter since, if you were going to use it at all, you would most likely do so when trying to bind data to objects.

Spring’s DataBinder and the lower-level BeanWrapper both use PropertyEditors to parse and format property values. The PropertyEditor concept is part of the JavaBeans specification, and is also explained in this chapter. Spring 3 introduces a "core.convert" package that provides a general type conversion facility, as well as a higher-level "format" package for formatting UI field values. These new packages may be used as simpler alternatives to PropertyEditors, and will also be discussed in this chapter.

3.2. Validation using Spring’s Validator interface

Spring features a Validator interface that you can use to validate objects. The Validator interface works using an Errorsobject so that while validating, validators can report validation failures to the Errors object.

Let’s consider a small data object:

**public** **class** **Person** {

**private** String name;

**private** **int** age;

*// the usual getters and setters...*

}

We’re going to provide validation behavior for the Person class by implementing the following two methods of the org.springframework.validation.Validator interface:

* supports(Class) - Can this Validator validate instances of the supplied Class?
* validate(Object, org.springframework.validation.Errors) - validates the given object and in case of validation errors, registers those with the given Errors object

Implementing a Validator is fairly straightforward, especially when you know of the ValidationUtils helper class that the Spring Framework also provides.

**public** **class** **PersonValidator** **implements** Validator {

*/\*\**

*\* This Validator validates \*just\* Person instances*

*\*/*

**public** **boolean** supports(Class clazz) {

**return** Person.class.equals(clazz);

}

**public** **void** validate(Object obj, Errors e) {

ValidationUtils.rejectIfEmpty(e, "name", "name.empty");

Person p = (Person) obj;

**if** (p.getAge() < 0) {

e.rejectValue("age", "negativevalue");

} **else** **if** (p.getAge() > 110) {

e.rejectValue("age", "too.darn.old");

}

}

}

As you can see, the static rejectIfEmpty(..) method on the ValidationUtils class is used to reject the 'name' property if it is null or the empty string. Have a look at the ValidationUtils javadocs to see what functionality it provides besides the example shown previously.

While it is certainly possible to implement a single Validator class to validate each of the nested objects in a rich object, it may be better to encapsulate the validation logic for each nested class of object in its own Validator implementation. A simple example of a *'rich'* object would be a Customer that is composed of two String properties (a first and second name) and a complex Address object. Address objects may be used independently of Customer objects, and so a distinct AddressValidatorhas been implemented. If you want your CustomerValidator to reuse the logic contained within the AddressValidator class without resorting to copy-and-paste, you can dependency-inject or instantiate an AddressValidator within your CustomerValidator, and use it like so:

**public** **class** **CustomerValidator** **implements** Validator {

**private** **final** Validator addressValidator;

**public** CustomerValidator(Validator addressValidator) {

**if** (addressValidator == null) {

**throw** **new** IllegalArgumentException("The supplied [Validator] is " +

"required and must not be null.");

}

**if** (!addressValidator.supports(Address.class)) {

**throw** **new** IllegalArgumentException("The supplied [Validator] must " +

"support the validation of [Address] instances.");

}

this.addressValidator = addressValidator;

}

*/\*\**

*\* This Validator validates Customer instances, and any subclasses of Customer too*

*\*/*

**public** **boolean** supports(Class clazz) {

**return** Customer.class.isAssignableFrom(clazz);

}

**public** **void** validate(Object target, Errors errors) {

ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName", "field.required");

ValidationUtils.rejectIfEmptyOrWhitespace(errors, "surname", "field.required");

Customer customer = (Customer) target;

**try** {

errors.pushNestedPath("address");

ValidationUtils.invokeValidator(this.addressValidator, customer.getAddress(), errors);

} **finally** {

errors.popNestedPath();

}

}

}

Validation errors are reported to the Errors object passed to the validator. In case of Spring Web MVC you can use <spring:bind/> tag to inspect the error messages, but of course you can also inspect the errors object yourself. More information about the methods it offers can be found in the javadocs.

3.3. Resolving codes to error messages

We’ve talked about databinding and validation. Outputting messages corresponding to validation errors is the last thing we need to discuss. In the example we’ve shown above, we rejected the name and the age field. If we’re going to output the error messages by using a MessageSource, we will do so using the error code we’ve given when rejecting the field ('name' and 'age' in this case). When you call (either directly, or indirectly, using for example the ValidationUtils class) rejectValue or one of the other reject methods from the Errors interface, the underlying implementation will not only register the code you’ve passed in, but also a number of additional error codes. What error codes it registers is determined by the MessageCodesResolver that is used. By default, the DefaultMessageCodesResolver is used, which for example not only registers a message with the code you gave, but also messages that include the field name you passed to the reject method. So in case you reject a field usingrejectValue("age", "too.darn.old"), apart from the too.darn.old code, Spring will also register too.darn.old.age and too.darn.old.age.int (so the first will include the field name and the second will include the type of the field); this is done as a convenience to aid developers in targeting error messages and suchlike.

More information on the MessageCodesResolver and the default strategy can be found online in the javadocs of[MessageCodesResolver](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/validation/MessageCodesResolver.html) and [DefaultMessageCodesResolver](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/validation/DefaultMessageCodesResolver.html), respectively.

3.4. Bean manipulation and the BeanWrapper

The org.springframework.beans package adheres to the JavaBeans standard provided by Oracle. A JavaBean is simply a class with a default no-argument constructor, which follows a naming convention where (by way of an example) a property named bingoMadness would have a setter method setBingoMadness(..) and a getter method getBingoMadness(). For more information about JavaBeans and the specification, please refer to Oracle’s website ( [javabeans](http://docs.oracle.com/javase/6/docs/api/java/beans/package-summary.html)).

One quite important class in the beans package is the BeanWrapper interface and its corresponding implementation ( BeanWrapperImpl). As quoted from the javadocs, the BeanWrapper offers functionality to set and get property values (individually or in bulk), get property descriptors, and to query properties to determine if they are readable or writable. Also, the BeanWrapper offers support for nested properties, enabling the setting of properties on sub-properties to an unlimited depth. Then, the BeanWrapper supports the ability to add standard JavaBeans PropertyChangeListeners and VetoableChangeListeners, without the need for supporting code in the target class. Last but not least, the BeanWrapperprovides support for the setting of indexed properties. The BeanWrapper usually isn’t used by application code directly, but by the DataBinder and the BeanFactory.

The way the BeanWrapper works is partly indicated by its name: *it wraps a bean* to perform actions on that bean, like setting and retrieving properties.

3.4.1. Setting and getting basic and nested properties

Setting and getting properties is done using the setPropertyValue(s) and getPropertyValue(s) methods that both come with a couple of overloaded variants. They’re all described in more detail in the javadocs Spring comes with. What’s important to know is that there are a couple of conventions for indicating properties of an object. A couple of examples:

| *Table 11. Examples of properties* | |
| --- | --- |
| **Expression** | **Explanation** |
| name | Indicates the property name corresponding to the methods getName() or isName() and setName(..) |
| account.name | Indicates the nested property name of the property accountcorresponding e.g. to the methods getAccount().setName()or getAccount().getName() |
| account[2] | Indicates the *third* element of the indexed property account. Indexed properties can be of type array, list or other *naturally ordered* collection |
| account[COMPANYNAME] | Indicates the value of the map entry indexed by the key *COMPANYNAME* of the Map property account |

Below you’ll find some examples of working with the BeanWrapper to get and set properties.

*(This next section is not vitally important to you if you’re not planning to work with the BeanWrapper directly. If you’re just using the DataBinder and the BeanFactory and their out-of-the-box implementation, you should skip ahead to the section aboutPropertyEditors.)*

Consider the following two classes:

**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;

}

}

The following code snippets show some examples of how to retrieve and manipulate some of the properties of instantiated Companies and Employees:

BeanWrapper company = **new** BeanWrapperImpl(**new** Company());

*// setting the company name..*

company.setPropertyValue("name", "Some Company Inc.");

*// ... can also be done like this:*

PropertyValue value = **new** PropertyValue("name", "Some Company Inc.");

company.setPropertyValue(value);

*// ok, let's create the director and tie it to the company:*

BeanWrapper jim = **new** BeanWrapperImpl(**new** Employee());

jim.setPropertyValue("name", "Jim Stravinsky");

company.setPropertyValue("managingDirector", jim.getWrappedInstance());

*// retrieving the salary of the managingDirector through the company*

Float salary = (Float) company.getPropertyValue("managingDirector.salary");

3.4.2. Built-in PropertyEditor implementations

Spring uses the concept of PropertyEditors to effect the conversion between an Object and a String. If you think about it, it sometimes might be handy to be able to represent properties in a different way than the object itself. For example, a Date can be represented in a human readable way (as the String '2007-14-09'), while we’re still able to convert the human readable form back to the original date (or even better: convert any date entered in a human readable form, back to Date objects). This behavior can be achieved by *registering custom editors*, of type java.beans.PropertyEditor. Registering custom editors on a BeanWrapper or alternately in a specific IoC container as mentioned in the previous chapter, gives it the knowledge of how to convert properties to the desired type. Read more about PropertyEditors in the javadocs of the java.beans package provided by Oracle.

A couple of examples where property editing is used in Spring:

* *setting properties on beans* is done using PropertyEditors. When mentioning java.lang.String as the value of a property of some bean you’re declaring in XML file, Spring will (if the setter of the corresponding property has a Class-parameter) use the ClassEditor to try to resolve the parameter to a Class object.
* *parsing HTTP request parameters* in Spring’s MVC framework is done using all kinds of PropertyEditors that you can manually bind in all subclasses of the CommandController.

Spring has a number of built-in PropertyEditors to make life easy. Each of those is listed below and they are all located in the org.springframework.beans.propertyeditors package. Most, but not all (as indicated below), are registered by default byBeanWrapperImpl. Where the property editor is configurable in some fashion, you can of course still register your own variant to override the default one:

| *Table 12. Built-in PropertyEditors* | |
| --- | --- |
| **Class** | **Explanation** |
| ByteArrayPropertyEditor | Editor for byte arrays. Strings will simply be converted to their corresponding byte representations. Registered by default by BeanWrapperImpl. |
| ClassEditor | Parses Strings representing classes to actual classes and the other way around. When a class is not found, an IllegalArgumentException is thrown. Registered by default by BeanWrapperImpl. |
| CustomBooleanEditor | Customizable property editor for Boolean properties. Registered by default by BeanWrapperImpl, but, can be overridden by registering custom instance of it as custom editor. |
| CustomCollectionEditor | Property editor for Collections, converting any source Collection to a given target Collection type. |
| CustomDateEditor | Customizable property editor for java.util.Date, supporting a custom DateFormat. NOT registered by default. Must be user registered as needed with appropriate format. |
| CustomNumberEditor | Customizable property editor for any Number subclass like Integer, Long, Float, Double. Registered by default by BeanWrapperImpl, but can be overridden by registering custom instance of it as a custom editor. |
| FileEditor | Capable of resolving Strings to java.io.File objects. Registered by default by BeanWrapperImpl. |
| InputStreamEditor | One-way property editor, capable of taking a text string and producing (via an intermediate ResourceEditor and Resource) an InputStream, so InputStream properties may be directly set as Strings. Note that the default usage will not close the InputStream for you! Registered by default by BeanWrapperImpl. |
| LocaleEditor | Capable of resolving Strings to Locale objects and vice versa (the String format is *[country]*[variant], which is the same thing the toString() method of Locale provides). Registered by default by BeanWrapperImpl. |
| PatternEditor | Capable of resolving Strings to java.util.regex.Patternobjects and vice versa. |
| PropertiesEditor | Capable of converting Strings (formatted using the format as defined in the javadocs of the java.util.Properties class) to Properties objects. Registered by default by BeanWrapperImpl. |
| StringTrimmerEditor | Property editor that trims Strings. Optionally allows transforming an empty string into a null value. NOT registered by default; must be user registered as needed. |
| URLEditor | Capable of resolving a String representation of a URL to an actual URL object. Registered by default by BeanWrapperImpl. |

Spring uses the java.beans.PropertyEditorManager to set the search path for property editors that might be needed. The search path also includes sun.bean.editors, which includes PropertyEditor implementations for types such as Font, Color, and most of the primitive types. Note also that the standard JavaBeans infrastructure will automatically discover PropertyEditor classes (without you having to register them explicitly) if they are in the same package as the class they handle, and have the same name as that class, with 'Editor' appended; for example, one could have the following class and package structure, which would be sufficient for the FooEditor class to be recognized and used as the PropertyEditor for Foo-typed properties.

com

chank

pop

Foo

FooEditor // the PropertyEditor for the Foo class

Note that you can also use the standard BeanInfo JavaBeans mechanism here as well (described [in not-amazing-detail here](http://docs.oracle.com/javase/tutorial/javabeans/advanced/customization.html)). Find below an example of using the BeanInfo mechanism for explicitly registering one or more PropertyEditor instances with the properties of an associated class.

com

chank

pop

Foo

FooBeanInfo // the BeanInfo for the Foo class

Here is the Java source code for the referenced FooBeanInfo class. This would associate a CustomNumberEditor with the ageproperty of the Foo class.

**public** **class** **FooBeanInfo** **extends** SimpleBeanInfo {

**public** PropertyDescriptor**[]** getPropertyDescriptors() {

**try** {

**final** PropertyEditor numberPE = **new** CustomNumberEditor(Integer.class, true);

PropertyDescriptor ageDescriptor = **new** PropertyDescriptor("age", Foo.class) {

**public** PropertyEditor createPropertyEditor(Object bean) {

**return** numberPE;

};

};

**return** **new** PropertyDescriptor**[]** { ageDescriptor };

}

**catch** (IntrospectionException ex) {

**throw** **new** Error(ex.toString());

}

}

}

Registering additional custom PropertyEditors

When setting bean properties as a string value, a Spring IoC container ultimately uses standard JavaBeans PropertyEditors to convert these Strings to the complex type of the property. Spring pre-registers a number of custom PropertyEditors (for example, to convert a classname expressed as a string into a real Class object). Additionally, Java’s standard JavaBeans PropertyEditor lookup mechanism allows a PropertyEditor for a class simply to be named appropriately and placed in the same package as the class it provides support for, to be found automatically.

If there is a need to register other custom PropertyEditors, there are several mechanisms available. The most manual approach, which is not normally convenient or recommended, is to simply use the registerCustomEditor() method of theConfigurableBeanFactory interface, assuming you have a BeanFactory reference. Another, slightly more convenient, mechanism is to use a special bean factory post-processor called CustomEditorConfigurer. Although bean factory post-processors can be used with BeanFactory implementations, the CustomEditorConfigurer has a nested property setup, so it is strongly recommended that it is used with the ApplicationContext, where it may be deployed in similar fashion to any other bean, and automatically detected and applied.

Note that all bean factories and application contexts automatically use a number of built-in property editors, through their use of something called a BeanWrapper to handle property conversions. The standard property editors that the BeanWrapperregisters are listed in [the previous section](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-beans-conversion). Additionally, ApplicationContexts also override or add an additional number of editors to handle resource lookups in a manner appropriate to the specific application context type.

Standard JavaBeans PropertyEditor instances are used to convert property values expressed as strings to the actual complex type of the property. CustomEditorConfigurer, a bean factory post-processor, may be used to conveniently add support for additional PropertyEditor instances to an ApplicationContext.

Consider a user class ExoticType, and another class DependsOnExoticType which needs ExoticType set as a property:

**package** example;

**public** **class** **ExoticType** {

**private** String name;

**public** ExoticType(String name) {

this.name = name;

}

}

**public** **class** **DependsOnExoticType** {

**private** ExoticType type;

**public** **void** setType(ExoticType type) {

this.type = type;

}

}

When things are properly set up, we want to be able to assign the type property as a string, which a PropertyEditor will behind the scenes convert into an actual ExoticType instance:

<bean id="sample" class="example.DependsOnExoticType">

<property name="type" value="aNameForExoticType"/>

</bean>

The PropertyEditor implementation could look similar to this:

*// converts string representation to ExoticType object*

**package** example;

**public** **class** **ExoticTypeEditor** **extends** PropertyEditorSupport {

**public** **void** setAsText(String text) {

setValue(**new** ExoticType(text.toUpperCase()));

}

}

Finally, we use CustomEditorConfigurer to register the new PropertyEditor with the ApplicationContext, which will then be able to use it as needed:

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

<property name="customEditors">

<map>

<entry key="example.ExoticType" value="example.ExoticTypeEditor"/>

</map>

</property>

</bean>

Using PropertyEditorRegistrars

Another mechanism for registering property editors with the Spring container is to create and use a PropertyEditorRegistrar. This interface is particularly useful when you need to use the same set of property editors in several different situations: write a corresponding registrar and reuse that in each case. PropertyEditorRegistrars work in conjunction with an interface called PropertyEditorRegistry, an interface that is implemented by the Spring BeanWrapper (and DataBinder). PropertyEditorRegistrars are particularly convenient when used in conjunction with the CustomEditorConfigurer (introduced [here](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-beans-conversion-customeditor-registration)), which exposes a property called setPropertyEditorRegistrars(..): PropertyEditorRegistrars added to aCustomEditorConfigurer in this fashion can easily be shared with DataBinder and Spring MVC Controllers. Furthermore, it avoids the need for synchronization on custom editors: a PropertyEditorRegistrar is expected to create fresh PropertyEditorinstances for each bean creation attempt.

Using a PropertyEditorRegistrar is perhaps best illustrated with an example. First off, you need to create your own PropertyEditorRegistrar implementation:

**package** com.foo.editors.spring;

**public** **final** **class** **CustomPropertyEditorRegistrar** **implements** PropertyEditorRegistrar {

**public** **void** registerCustomEditors(PropertyEditorRegistry registry) {

*// it is expected that new PropertyEditor instances are created*

registry.registerCustomEditor(ExoticType.class, **new** ExoticTypeEditor());

*// you could register as many custom property editors as are required here...*

}

}

See also the org.springframework.beans.support.ResourceEditorRegistrar for an example PropertyEditorRegistrarimplementation. Notice how in its implementation of the registerCustomEditors(..) method it creates new instances of each property editor.

Next we configure a CustomEditorConfigurer and inject an instance of our CustomPropertyEditorRegistrar into it:

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

<property name="propertyEditorRegistrars">

<list>

<ref bean="customPropertyEditorRegistrar"/>

</list>

</property>

</bean>

<bean id="customPropertyEditorRegistrar"

class="com.foo.editors.spring.CustomPropertyEditorRegistrar"/>

Finally, and in a bit of a departure from the focus of this chapter, for those of you using [Spring’s MVC web framework](https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc), using PropertyEditorRegistrars in conjunction with data-binding Controllers (such as SimpleFormController) can be very convenient. Find below an example of using a PropertyEditorRegistrar in the implementation of an initBinder(..) method:

**public** **final** **class** **RegisterUserController** **extends** SimpleFormController {

**private** **final** PropertyEditorRegistrar customPropertyEditorRegistrar;

**public** RegisterUserController(PropertyEditorRegistrar propertyEditorRegistrar) {

this.customPropertyEditorRegistrar = propertyEditorRegistrar;

}

**protected** **void** initBinder(HttpServletRequest request,

ServletRequestDataBinder binder) **throws** Exception {

**this.customPropertyEditorRegistrar.registerCustomEditors(binder);**

}

*// other methods to do with registering a User*

}

This style of PropertyEditor registration can lead to concise code (the implementation of initBinder(..) is just one line long!), and allows common PropertyEditor registration code to be encapsulated in a class and then shared amongst as manyControllers as needed.

3.5. Spring Type Conversion

Spring 3 introduces a core.convert package that provides a general type conversion system. The system defines an SPI to implement type conversion logic, as well as an API to execute type conversions at runtime. Within a Spring container, this system can be used as an alternative to PropertyEditors to convert externalized bean property value strings to required property types. The public API may also be used anywhere in your application where type conversion is needed.

3.5.1. Converter SPI

The SPI to implement type conversion logic is simple and strongly typed:

**package** org.springframework.core.convert.converter;

**public** **interface** **Converter**<S, T> {

T convert(S source);

}

To create your own converter, simply implement the interface above. Parameterize S as the type you are converting from, and T as the type you are converting to. Such a converter can also be applied transparently if a collection or array of S needs to be converted to an array or collection of T, provided that a delegating array/collection converter has been registered as well (which DefaultConversionService does by default).

For each call to convert(S), the source argument is guaranteed to be NOT null. Your Converter may throw any unchecked exception if conversion fails; specifically, an IllegalArgumentException should be thrown to report an invalid source value. Take care to ensure that your Converter implementation is thread-safe.

Several converter implementations are provided in the core.convert.support package as a convenience. These include converters from Strings to Numbers and other common types. Consider StringToInteger as an example for a typical Converterimplementation:

**package** org.springframework.core.convert.support;

**final** **class** **StringToInteger** **implements** Converter<String, Integer> {

**public** Integer convert(String source) {

**return** Integer.valueOf(source);

}

}

3.5.2. ConverterFactory

When you need to centralize the conversion logic for an entire class hierarchy, for example, when converting from String to java.lang.Enum objects, implement ConverterFactory:

**package** org.springframework.core.convert.converter;

**public** **interface** **ConverterFactory**<S, R> {

<T **extends** R> Converter<S, T> getConverter(Class<T> targetType);

}

Parameterize S to be the type you are converting from and R to be the base type defining the *range* of classes you can convert to. Then implement getConverter(Class<T>), where T is a subclass of R.

Consider the StringToEnum ConverterFactory as an example:

**package** org.springframework.core.convert.support;

**final** **class** **StringToEnumConverterFactory** **implements** ConverterFactory<String, Enum> {

**public** <T **extends** Enum> Converter<String, T> getConverter(Class<T> targetType) {

**return** **new** StringToEnumConverter(targetType);

}

**private** **final** **class** **StringToEnumConverter**<T **extends** Enum> **implements** Converter<String, T> {

**private** Class<T> enumType;

**public** StringToEnumConverter(Class<T> enumType) {

this.enumType = enumType;

}

**public** T convert(String source) {

**return** (T) Enum.valueOf(this.enumType, source.trim());

}

}

}

3.5.3. GenericConverter

When you require a sophisticated Converter implementation, consider the GenericConverter interface. With a more flexible but less strongly typed signature, a GenericConverter supports converting between multiple source and target types. In addition, a GenericConverter makes available source and target field context you can use when implementing your conversion logic. Such context allows a type conversion to be driven by a field annotation, or generic information declared on a field signature.

**package** org.springframework.core.convert.converter;

**public** **interface** **GenericConverter** {

**public** Set<ConvertiblePair> getConvertibleTypes();

Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType);

}

To implement a GenericConverter, have getConvertibleTypes() return the supported source→target type pairs. Then implement convert(Object, TypeDescriptor, TypeDescriptor) to implement your conversion logic. The source TypeDescriptor provides access to the source field holding the value being converted. The target TypeDescriptor provides access to the target field where the converted value will be set.

A good example of a GenericConverter is a converter that converts between a Java Array and a Collection. Such an ArrayToCollectionConverter introspects the field that declares the target Collection type to resolve the Collection’s element type. This allows each element in the source array to be converted to the Collection element type before the Collection is set on the target field.

|  |  |
| --- | --- |
|  | Because GenericConverter is a more complex SPI interface, only use it when you need it. Favor Converter or ConverterFactory for basic type conversion needs. |

ConditionalGenericConverter

Sometimes you only want a Converter to execute if a specific condition holds true. For example, you might only want to execute a Converter if a specific annotation is present on the target field. Or you might only want to execute a Converter if a specific method, such as a static valueOf method, is defined on the target class. ConditionalGenericConverter is the union of the GenericConverter and ConditionalConverter interfaces that allows you to define such custom matching criteria:

**public** **interface** **ConditionalConverter** {

**boolean** matches(TypeDescriptor sourceType, TypeDescriptor targetType);

}

**public** **interface** **ConditionalGenericConverter**

**extends** GenericConverter, ConditionalConverter {

}

A good example of a ConditionalGenericConverter is an EntityConverter that converts between an persistent entity identifier and an entity reference. Such a EntityConverter might only match if the target entity type declares a static finder method e.g.findAccount(Long). You would perform such a finder method check in the implementation of matches(TypeDescriptor, TypeDescriptor).

3.5.4. ConversionService API

The ConversionService defines a unified API for executing type conversion logic at runtime. Converters are often executed behind this facade interface:

**package** org.springframework.core.convert;

**public** **interface** **ConversionService** {

**boolean** canConvert(Class<?> sourceType, Class<?> targetType);

<T> T convert(Object source, Class<T> targetType);

**boolean** canConvert(TypeDescriptor sourceType, TypeDescriptor targetType);

Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType);

}

Most ConversionService implementations also implement ConverterRegistry, which provides an SPI for registering converters. Internally, a ConversionService implementation delegates to its registered converters to carry out type conversion logic.

A robust ConversionService implementation is provided in the core.convert.support package. GenericConversionService is the general-purpose implementation suitable for use in most environments. ConversionServiceFactory provides a convenient factory for creating common ConversionService configurations.

3.5.5. Configuring a ConversionService

A ConversionService is a stateless object designed to be instantiated at application startup, then shared between multiple threads. In a Spring application, you typically configure a ConversionService instance per Spring container (or ApplicationContext). That ConversionService will be picked up by Spring and then used whenever a type conversion needs to be performed by the framework. You may also inject this ConversionService into any of your beans and invoke it directly.

|  |  |
| --- | --- |
|  | If no ConversionService is registered with Spring, the original PropertyEditor-based system is used. |

To register a default ConversionService with Spring, add the following bean definition with id conversionService:

<bean id="conversionService"

class="org.springframework.context.support.ConversionServiceFactoryBean"/>

A default ConversionService can convert between strings, numbers, enums, collections, maps, and other common types. To supplement or override the default converters with your own custom converter(s), set the converters property. Property values may implement either of the Converter, ConverterFactory, or GenericConverter interfaces.

<bean id="conversionService"

class="org.springframework.context.support.ConversionServiceFactoryBean">

<property name="converters">

<set>

<bean class="example.MyCustomConverter"/>

</set>

</property>

</bean>

It is also common to use a ConversionService within a Spring MVC application. See [Conversion and Formatting](https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc-config-conversion) in the Spring MVC chapter.

In certain situations you may wish to apply formatting during conversion. See [FormatterRegistry SPI](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#format-FormatterRegistry-SPI) for details on usingFormattingConversionServiceFactoryBean.

3.5.6. Using a ConversionService programmatically

To work with a ConversionService instance programmatically, simply inject a reference to it like you would for any other bean:

@Service

**public** **class** **MyService** {

@Autowired

**public** MyService(ConversionService conversionService) {

this.conversionService = conversionService;

}

**public** **void** doIt() {

this.conversionService.convert(...)

}

}

For most use cases, the convert method specifying the *targetType* can be used but it will not work with more complex types such as a collection of a parameterized element. If you want to convert a List of Integer to a List of Stringprogrammatically, for instance, you need to provide a formal definition of the source and target types.

Fortunately, TypeDescriptor provides various options to make that straightforward:

DefaultConversionService cs = **new** DefaultConversionService();

List<Integer> input = ....

cs.convert(input,

TypeDescriptor.forObject(input), *// List<Integer> type descriptor*

TypeDescriptor.collection(List.class, TypeDescriptor.valueOf(String.class)));

Note that DefaultConversionService registers converters automatically which are appropriate for most environments. This includes collection converters, scalar converters, and also basic Object to String converters. The same converters can be registered with any ConverterRegistry using the *static* addDefaultConverters method on the DefaultConversionService class.

Converters for value types will be reused for arrays and collections, so there is no need to create a specific converter to convert from a Collection of S to a Collection of T, assuming that standard collection handling is appropriate.

3.6. Spring Field Formatting

As discussed in the previous section, [core.convert](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#core-convert) is a general-purpose type conversion system. It provides a unified ConversionService API as well as a strongly-typed Converter SPI for implementing conversion logic from one type to another. A Spring Container uses this system to bind bean property values. In addition, both the Spring Expression Language (SpEL) and DataBinder use this system to bind field values. For example, when SpEL needs to coerce a Short to a Long to complete an expression.setValue(Object bean, Object value) attempt, the core.convert system performs the coercion.

Now consider the type conversion requirements of a typical client environment such as a web or desktop application. In such environments, you typically convert *from String* to support the client postback process, as well as back *to String* to support the view rendering process. In addition, you often need to localize String values. The more general *core.convert* Converter SPI does not address such *formatting* requirements directly. To directly address them, Spring 3 introduces a convenient Formatter SPI that provides a simple and robust alternative to PropertyEditors for client environments.

In general, use the Converter SPI when you need to implement general-purpose type conversion logic; for example, for converting between a java.util.Date and and java.lang.Long. Use the Formatter SPI when you’re working in a client environment, such as a web application, and need to parse and print localized field values. The ConversionService provides a unified type conversion API for both SPIs.

3.6.1. Formatter SPI

The Formatter SPI to implement field formatting logic is simple and strongly typed:

**package** org.springframework.format;

**public** **interface** **Formatter**<T> **extends** Printer<T>, Parser<T> {

}

Where Formatter extends from the Printer and Parser building-block interfaces:

**public** **interface** **Printer**<T> {

String print(T fieldValue, Locale locale);

}

**import** java.text.ParseException;

**public** **interface** **Parser**<T> {

T parse(String clientValue, Locale locale) **throws** ParseException;

}

To create your own Formatter, simply implement the Formatter interface above. Parameterize T to be the type of object you wish to format, for example, java.util.Date. Implement the print() operation to print an instance of T for display in the client locale. Implement the parse() operation to parse an instance of T from the formatted representation returned from the client locale. Your Formatter should throw a ParseException or IllegalArgumentException if a parse attempt fails. Take care to ensure your Formatter implementation is thread-safe.

Several Formatter implementations are provided in format subpackages as a convenience. The number package provides a NumberFormatter, CurrencyFormatter, and PercentFormatter to format java.lang.Number objects using a java.text.NumberFormat. The datetime package provides a DateFormatter to format java.util.Date objects with a java.text.DateFormat. The datetime.joda package provides comprehensive datetime formatting support based on the [Joda Time library](http://joda-time.sourceforge.net/).

Consider DateFormatter as an example Formatter implementation:

**package** org.springframework.format.datetime;

**public** **final** **class** **DateFormatter** **implements** Formatter<Date> {

**private** String pattern;

**public** DateFormatter(String pattern) {

this.pattern = pattern;

}

**public** String print(Date date, Locale locale) {

**if** (date == null) {

**return** "";

}

**return** getDateFormat(locale).format(date);

}

**public** Date parse(String formatted, Locale locale) **throws** ParseException {

**if** (formatted.length() == 0) {

**return** null;

}

**return** getDateFormat(locale).parse(formatted);

}

**protected** DateFormat getDateFormat(Locale locale) {

DateFormat dateFormat = **new** SimpleDateFormat(this.pattern, locale);

dateFormat.setLenient(false);

**return** dateFormat;

}

}

The Spring team welcomes community-driven Formatter contributions; see [jira.spring.io](https://jira.spring.io/browse/SPR) to contribute.

3.6.2. Annotation-driven Formatting

As you will see, field formatting can be configured by field type or annotation. To bind an Annotation to a formatter, implement AnnotationFormatterFactory:

**package** org.springframework.format;

**public** **interface** **AnnotationFormatterFactory**<A **extends** Annotation> {

Set<Class<?>> getFieldTypes();

Printer<?> getPrinter(A annotation, Class<?> fieldType);

Parser<?> getParser(A annotation, Class<?> fieldType);

}

Parameterize A to be the field annotationType you wish to associate formatting logic with, for example org.springframework.format.annotation.DateTimeFormat. Have getFieldTypes() return the types of fields the annotation may be used on. Have getPrinter() return a Printer to print the value of an annotated field. Have getParser() return a Parser to parse a clientValue for an annotated field.

The example AnnotationFormatterFactory implementation below binds the @NumberFormat Annotation to a formatter. This annotation allows either a number style or pattern to be specified:

**public** **final** **class** **NumberFormatAnnotationFormatterFactory**

**implements** AnnotationFormatterFactory<NumberFormat> {

**public** Set<Class<?>> getFieldTypes() {

**return** **new** HashSet<Class<?>>(asList(**new** Class<?>**[]** {

Short.class, Integer.class, Long.class, Float.class,

Double.class, BigDecimal.class, BigInteger.class }));

}

**public** Printer<Number> getPrinter(NumberFormat annotation, Class<?> fieldType) {

**return** configureFormatterFrom(annotation, fieldType);

}

**public** Parser<Number> getParser(NumberFormat annotation, Class<?> fieldType) {

**return** configureFormatterFrom(annotation, fieldType);

}

**private** Formatter<Number> configureFormatterFrom(NumberFormat annotation,

Class<?> fieldType) {

**if** (!annotation.pattern().isEmpty()) {

**return** **new** NumberFormatter(annotation.pattern());

} **else** {

Style style = annotation.style();

**if** (style == Style.PERCENT) {

**return** **new** PercentFormatter();

} **else** **if** (style == Style.CURRENCY) {

**return** **new** CurrencyFormatter();

} **else** {

**return** **new** NumberFormatter();

}

}

}

}

To trigger formatting, simply annotate fields with @NumberFormat:

**public** **class** **MyModel** {

@NumberFormat(style=Style.CURRENCY)

**private** BigDecimal decimal;

}

Format Annotation API

A portable format annotation API exists in the org.springframework.format.annotation package. Use @NumberFormat to format java.lang.Number fields. Use @DateTimeFormat to format java.util.Date, java.util.Calendar, java.util.Long, or Joda Time fields.

The example below uses @DateTimeFormat to format a java.util.Date as a ISO Date (yyyy-MM-dd):

**public** **class** **MyModel** {

@DateTimeFormat(iso=ISO.DATE)

**private** Date date;

}

3.6.3. FormatterRegistry SPI

The FormatterRegistry is an SPI for registering formatters and converters. FormattingConversionService is an implementation of FormatterRegistry suitable for most environments. This implementation may be configured programmatically or declaratively as a Spring bean using FormattingConversionServiceFactoryBean. Because this implementation also implements ConversionService, it can be directly configured for use with Spring’s DataBinder and the Spring Expression Language (SpEL).

Review the FormatterRegistry SPI below:

**package** org.springframework.format;

**public** **interface** **FormatterRegistry** **extends** ConverterRegistry {

**void** addFormatterForFieldType(Class<?> fieldType, Printer<?> printer, Parser<?> parser);

**void** addFormatterForFieldType(Class<?> fieldType, Formatter<?> formatter);

**void** addFormatterForFieldType(Formatter<?> formatter);

**void** addFormatterForAnnotation(AnnotationFormatterFactory<?, ?> factory);

}

As shown above, Formatters can be registered by fieldType or annotation.

The FormatterRegistry SPI allows you to configure Formatting rules centrally, instead of duplicating such configuration across your Controllers. For example, you might want to enforce that all Date fields are formatted a certain way, or fields with a specific annotation are formatted in a certain way. With a shared FormatterRegistry, you define these rules once and they are applied whenever formatting is needed.

3.6.4. FormatterRegistrar SPI

The FormatterRegistrar is an SPI for registering formatters and converters through the FormatterRegistry:

**package** org.springframework.format;

**public** **interface** **FormatterRegistrar** {

**void** registerFormatters(FormatterRegistry registry);

}

A FormatterRegistrar is useful when registering multiple related converters and formatters for a given formatting category, such as Date formatting. It can also be useful where declarative registration is insufficient. For example when a formatter needs to be indexed under a specific field type different from its own <T> or when registering a Printer/Parser pair. The next section provides more information on converter and formatter registration.

3.6.5. Configuring Formatting in Spring MVC

See [Conversion and Formatting](https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc-config-conversion) in the Spring MVC chapter.

3.7. Configuring a global date & time format

By default, date and time fields that are not annotated with @DateTimeFormat are converted from strings using the DateFormat.SHORT style. If you prefer, you can change this by defining your own global format.

You will need to ensure that Spring does not register default formatters, and instead you should register all formatters manually. Use the org.springframework.format.datetime.joda.JodaTimeFormatterRegistrar ororg.springframework.format.datetime.DateFormatterRegistrar class depending on whether you use the Joda Time library.

For example, the following Java configuration will register a global ' `yyyyMMdd’ format. This example does not depend on the Joda Time library:

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** FormattingConversionService conversionService() {

*// Use the DefaultFormattingConversionService but do not register defaults*

DefaultFormattingConversionService conversionService = **new** DefaultFormattingConversionService(false);

*// Ensure @NumberFormat is still supported*

conversionService.addFormatterForFieldAnnotation(**new** NumberFormatAnnotationFormatterFactory());

*// Register date conversion with a specific global format*

DateFormatterRegistrar registrar = **new** DateFormatterRegistrar();

registrar.setFormatter(**new** DateFormatter("yyyyMMdd"));

registrar.registerFormatters(conversionService);

**return** conversionService;

}

}

If you prefer XML based configuration you can use a FormattingConversionServiceFactoryBean. Here is the same example, this time using Joda Time:

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

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

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

xsi:schemaLocation="

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

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

<bean id="conversionService" class="org.springframework.format.support.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 provides separate distinct types to represent date, time and date-time values. The dateFormatter, timeFormatter and dateTimeFormatter properties of the JodaTimeFormatterRegistrar should be used to configure the different formats for each type. The DateTimeFormatterFactoryBean provides a convenient way to create formatters. |

If you are using Spring MVC remember to explicitly configure the conversion service that is used. For Java based @Configuration this means extending the WebMvcConfigurationSupport class and overriding the mvcConversionService()method. For XML you should use the 'conversion-service' attribute of the mvc:annotation-driven element. See [Conversion and Formatting](https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc-config-conversion) for details.

3.8. Spring Validation

Spring 3 introduces several enhancements to its validation support. First, the JSR-303 Bean Validation API is now fully supported. Second, when used programmatically, Spring’s DataBinder can now validate objects as well as bind to them. Third, Spring MVC now has support for declaratively validating @Controller inputs.

3.8.1. Overview of the JSR-303 Bean Validation API

JSR-303 standardizes validation constraint declaration and metadata for the Java platform. Using this API, you annotate domain model properties with declarative validation constraints and the runtime enforces them. There are a number of built-in constraints you can take advantage of. You may also define your own custom constraints.

To illustrate, consider a simple PersonForm model with two properties:

**public** **class** **PersonForm** {

**private** String name;

**private** **int** age;

}

JSR-303 allows you to define declarative validation constraints against such properties:

**public** **class** **PersonForm** {

@NotNull

@Size(max=64)

**private** String name;

@Min(0)

**private** **int** age;

}

When an instance of this class is validated by a JSR-303 Validator, these constraints will be enforced.

For general information on JSR-303/JSR-349, see the [Bean Validation website](http://beanvalidation.org/). For information on the specific capabilities of the default reference implementation, see the [Hibernate Validator](https://www.hibernate.org/412.html) documentation. To learn how to setup a Bean Validation provider as a Spring bean, keep reading.

3.8.2. Configuring a Bean Validation Provider

Spring provides full support for the Bean Validation API. This includes convenient support for bootstrapping a JSR-303/JSR-349 Bean Validation provider as a Spring bean. This allows for a javax.validation.ValidatorFactory or javax.validation.Validator to be injected wherever validation is needed in your application.

Use the LocalValidatorFactoryBean to configure a default Validator as a Spring bean:

<bean id="validator"

class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"/>

The basic configuration above will trigger Bean Validation to initialize using its default bootstrap mechanism. A JSR-303/JSR-349 provider, such as Hibernate Validator, is expected to be present in the classpath and will be detected automatically.

Injecting a Validator

LocalValidatorFactoryBean implements both javax.validation.ValidatorFactory and javax.validation.Validator, as well as Spring’s org.springframework.validation.Validator. You may inject a reference to either of these interfaces into beans that need to invoke validation logic.

Inject a reference to javax.validation.Validator if you prefer to work with the Bean Validation API directly:

**import** javax.validation.Validator;

@Service

**public** **class** **MyService** {

@Autowired

**private** Validator validator;

Inject a reference to org.springframework.validation.Validator if your bean requires the Spring Validation API:

**import** org.springframework.validation.Validator;

@Service

**public** **class** **MyService** {

@Autowired

**private** Validator validator;

}

Configuring Custom Constraints

Each Bean Validation constraint consists of two parts. First, a @Constraint annotation that declares the constraint and its configurable properties. Second, an implementation of the javax.validation.ConstraintValidator interface that implements the constraint’s behavior. To associate a declaration with an implementation, each @Constraint annotation references a corresponding ValidationConstraint implementation class. At runtime, a ConstraintValidatorFactory instantiates the referenced implementation when the constraint annotation is encountered in your domain model.

By default, the LocalValidatorFactoryBean configures a SpringConstraintValidatorFactory that uses Spring to create ConstraintValidator instances. This allows your custom ConstraintValidators to benefit from dependency injection like any other Spring bean.

Shown below is an example of a custom @Constraint declaration, followed by an associated ConstraintValidatorimplementation that uses Spring for dependency injection:

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

@Retention(RetentionPolicy.RUNTIME)

@Constraint(validatedBy=MyConstraintValidator.class)

**public** @interface MyConstraint {

}

**import** javax.validation.ConstraintValidator;

**public** **class** **MyConstraintValidator** **implements** ConstraintValidator {

@Autowired;

**private** Foo aDependency;

...

}

As you can see, a ConstraintValidator implementation may have its dependencies @Autowired like any other Spring bean.

Spring-driven Method Validation

The method validation feature supported by Bean Validation 1.1, and as a custom extension also by Hibernate Validator 4.3, can be integrated into a Spring context through a MethodValidationPostProcessor bean definition:

<bean class="org.springframework.validation.beanvalidation.MethodValidationPostProcessor"/>

In order to be eligible for Spring-driven method validation, all target classes need to be annotated with Spring’s @Validatedannotation, optionally declaring the validation groups to use. Check out the MethodValidationPostProcessor javadocs for setup details with Hibernate Validator and Bean Validation 1.1 providers.

Additional Configuration Options

The default LocalValidatorFactoryBean configuration should prove sufficient for most cases. There are a number of configuration options for various Bean Validation constructs, from message interpolation to traversal resolution. See theLocalValidatorFactoryBean javadocs for more information on these options.

3.8.3. Configuring a DataBinder

Since Spring 3, a DataBinder instance can be configured with a Validator. Once configured, the Validator may be invoked by calling binder.validate(). Any validation Errors are automatically added to the binder’s BindingResult.

When working with the DataBinder programmatically, this can be used to invoke validation logic after binding to a target object:

Foo target = **new** Foo();

DataBinder binder = **new** DataBinder(target);

binder.setValidator(**new** FooValidator());

*// bind to the target object*

binder.bind(propertyValues);

*// validate the target object*

binder.validate();

*// get BindingResult that includes any validation errors*

BindingResult results = binder.getBindingResult();

A DataBinder can also be configured with multiple Validator instances via dataBinder.addValidators and dataBinder.replaceValidators. This is useful when combining globally configured Bean Validation with a Spring Validatorconfigured locally on a DataBinder instance. See [[validation-mvc-configuring]](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#validation-mvc-configuring).

3.8.4. Spring MVC 3 Validation

See [Validation](https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc-config-validation) in the Spring MVC chapter.

4. Spring Expression Language (SpEL)

4.1. Introduction

The Spring Expression Language (SpEL for short) is a powerful expression language that supports querying and manipulating an object graph at runtime. The language syntax is similar to Unified EL but offers additional features, most notably method invocation and basic string templating functionality.

While there are several other Java expression languages available, OGNL, MVEL, and JBoss EL, to name a few, the Spring Expression Language was created to provide the Spring community with a single well supported expression language that can be used across all the products in the Spring portfolio. Its language features are driven by the requirements of the projects in the Spring portfolio, including tooling requirements for code completion support within the eclipse based Spring Tool Suite. That said, SpEL is based on a technology agnostic API allowing other expression language implementations to be integrated should the need arise.

While SpEL serves as the foundation for expression evaluation within the Spring portfolio, it is not directly tied to Spring and can be used independently. In order to be self contained, many of the examples in this chapter use SpEL as if it were an independent expression language. This requires creating a few bootstrapping infrastructure classes such as the parser. Most Spring users will not need to deal with this infrastructure and will instead only author expression strings for evaluation. An example of this typical use is the integration of SpEL into creating XML or annotated based bean definitions as shown in the section [Expression support for defining bean definitions.](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#expressions-beandef)

This chapter covers the features of the expression language, its API, and its language syntax. In several places an Inventor and Inventor’s Society class are used as the target objects for expression evaluation. These class declarations and the data used to populate them are listed at the end of the chapter.

4.2. Feature overview

The expression language supports the following functionality

* Literal expressions
* Boolean and relational operators
* Regular expressions
* Class expressions
* Accessing properties, arrays, lists, maps
* Method invocation
* Relational operators
* Assignment
* Calling constructors
* Bean references
* Array construction
* Inline lists
* Inline maps
* Ternary operator
* Variables
* User defined functions
* Collection projection
* Collection selection
* Templated expressions

4.3. Expression evaluation using Spring’s Expression interface

This section introduces the simple use of SpEL interfaces and its expression language. The complete language reference can be found in the section [Language Reference](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#expressions-language-ref).

The following code introduces the SpEL API to evaluate the literal string expression 'Hello World'.

ExpressionParser parser = **new** SpelExpressionParser();

Expression exp = parser.parseExpression("**'Hello World'**");

String message = (String) exp.getValue();

The value of the message variable is simply 'Hello World'.

The SpEL classes and interfaces you are most likely to use are located in the packages org.springframework.expression and its sub packages and spel.support.

The interface ExpressionParser is responsible for parsing an expression string. In this example the expression string is a string literal denoted by the surrounding single quotes. The interface Expression is responsible for evaluating the previously defined expression string. There are two exceptions that can be thrown, ParseException and EvaluationException when calling parser.parseExpression and exp.getValue respectively.

SpEL supports a wide range of features, such as calling methods, accessing properties, and calling constructors.

As an example of method invocation, we call the concat method on the string literal.

ExpressionParser parser = **new** SpelExpressionParser();

Expression exp = parser.parseExpression("**'Hello World'.concat('!')**");

String message = (String) exp.getValue();

The value of message is now 'Hello World!'.

As an example of calling a JavaBean property, the String property Bytes can be called as shown below.

ExpressionParser parser = **new** SpelExpressionParser();

*// invokes 'getBytes()'*

Expression exp = parser.parseExpression("**'Hello World'.bytes**");

**byte[]** bytes = (**byte[]**) exp.getValue();

SpEL also supports nested properties using standard *dot* notation, i.e. prop1.prop2.prop3 and the setting of property values

Public fields may also be accessed.

ExpressionParser parser = **new** SpelExpressionParser();

*// invokes 'getBytes().length'*

Expression exp = parser.parseExpression("**'Hello World'.bytes.length**");

**int** length = (Integer) exp.getValue();

The String’s constructor can be called instead of using a string literal.

ExpressionParser parser = **new** SpelExpressionParser();

Expression exp = parser.parseExpression("**new String('hello world').toUpperCase()**");

String message = exp.getValue(String.class);

Note the use of the generic method public <T> T getValue(Class<T> desiredResultType). Using this method removes the need to cast the value of the expression to the desired result type. An EvaluationException will be thrown if the value cannot be cast to the type T or converted using the registered type converter.

The more common usage of SpEL is to provide an expression string that is evaluated against a specific object instance (called the root object). There are two options here and which to choose depends on whether the object against which the expression is being evaluated will be changing with each call to evaluate the expression. In the following example we retrieve the nameproperty from an instance of the Inventor class.

*// Create and set a calendar*

GregorianCalendar c = **new** GregorianCalendar();

c.set(1856, 7, 9);

*// The constructor arguments are name, birthday, and nationality.*

Inventor tesla = **new** Inventor("Nikola Tesla", c.getTime(), "Serbian");

ExpressionParser parser = **new** SpelExpressionParser();

Expression exp = parser.parseExpression("**name**");

EvaluationContext context = **new** StandardEvaluationContext(tesla);

String name = (String) exp.getValue(context);

In the last line, the value of the string variable name will be set to "Nikola Tesla". The class StandardEvaluationContext is where you can specify which object the "name" property will be evaluated against. This is the mechanism to use if the root object is unlikely to change, it can simply be set once in the evaluation context. If the root object is likely to change repeatedly, it can be supplied on each call to getValue, as this next example shows:

/ Create and set a calendar

GregorianCalendar c = **new** GregorianCalendar();

c.set(1856, 7, 9);

*// The constructor arguments are name, birthday, and nationality.*

Inventor tesla = **new** Inventor("Nikola Tesla", c.getTime(), "Serbian");

ExpressionParser parser = **new** SpelExpressionParser();

Expression exp = parser.parseExpression("**name**");

String name = (String) exp.getValue(tesla);

In this case the inventor tesla has been supplied directly to getValue and the expression evaluation infrastructure creates and manages a default evaluation context internally - it did not require one to be supplied.

The StandardEvaluationContext is relatively expensive to construct and during repeated usage it builds up cached state that enables subsequent expression evaluations to be performed more quickly. For this reason it is better to cache and reuse them where possible, rather than construct a new one for each expression evaluation.

In some cases it can be desirable to use a configured evaluation context and yet still supply a different root object on each call to getValue. getValue allows both to be specified on the same call. In these situations the root object passed on the call is considered to override any (which maybe null) specified on the evaluation context.

|  |  |
| --- | --- |
|  | In standalone usage of SpEL there is a need to create the parser, parse expressions and perhaps provide evaluation contexts and a root context object. However, more common usage is to provide only the SpEL expression string as part of a configuration file, for example for Spring bean or Spring Web Flow definitions. In this case, the parser, evaluation context, root object and any predefined variables are all set up implicitly, requiring the user to specify nothing other than the expressions. |

As a final introductory example, the use of a boolean operator is shown using the Inventor object in the previous example.

Expression exp = parser.parseExpression("name == 'Nikola Tesla'");

**boolean** result = exp.getValue(context, Boolean.class); *// evaluates to true*

4.3.1. The EvaluationContext interface

The interface EvaluationContext is used when evaluating an expression to resolve properties, methods, fields, and to help perform type conversion. The out-of-the-box implementation, StandardEvaluationContext, uses reflection to manipulate the object, caching java.lang.reflect.Method, java.lang.reflect.Field, and java.lang.reflect.Constructor instances for increased performance.

The StandardEvaluationContext is where you may specify the root object to evaluate against via the method setRootObject()or passing the root object into the constructor. You can also specify variables and functions that will be used in the expression using the methods setVariable() and registerFunction(). The use of variables and functions are described in the language reference sections [Variables](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#expressions-ref-variables) and [Functions](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#expressions-ref-functions). The StandardEvaluationContext is also where you can register customConstructorResolvers, MethodResolvers, and PropertyAccessors to extend how SpEL evaluates expressions. Please refer to the javadoc of these classes for more details.

Type conversion

By default SpEL uses the conversion service available in Spring core ( org.springframework.core.convert.ConversionService). This conversion service comes with many converters built in for common conversions but is also fully extensible so custom conversions between types can be added. Additionally it has the key capability that it is generics aware. This means that when working with generic types in expressions, SpEL will attempt conversions to maintain type correctness for any objects it encounters.

What does this mean in practice? Suppose assignment, using setValue(), is being used to set a List property. The type of the property is actually List<Boolean>. SpEL will recognize that the elements of the list need to be converted to Boolean before being placed in it. A simple example:

**class** **Simple** {

**public** List<Boolean> booleanList = **new** ArrayList<Boolean>();

}

Simple simple = **new** Simple();

simple.booleanList.add(true);

StandardEvaluationContext simpleContext = **new** StandardEvaluationContext(simple);

*// false is passed in here as a string. SpEL and the conversion service will*

*// correctly recognize that it needs to be a Boolean and convert it*

parser.parseExpression("booleanList[0]").setValue(simpleContext, "false");

*// b will be false*

Boolean b = simple.booleanList.get(0);

4.3.2. Parser configuration

It is possible to configure the SpEL expression parser using a parser configuration object (org.springframework.expression.spel.SpelParserConfiguration). The configuration object controls the behavior of some of the expression components. For example, if indexing into an array or collection and the element at the specified index is null it is possible to automatically create the element. This is useful when using expressions made up of a chain of property references. If indexing into an array or list and specifying an index that is beyond the end of the current size of the array or list it is possible to automatically grow the array or list to accommodate that index.

**class** **Demo** {

**public** List<String> list;

}

*// Turn on:*

*// - auto null reference initialization*

*// - auto collection growing*

SpelParserConfiguration config = **new** SpelParserConfiguration(true,true);

ExpressionParser parser = **new** SpelExpressionParser(config);

Expression expression = parser.parseExpression("list[3]");

Demo demo = **new** Demo();

Object o = expression.getValue(demo);

*// demo.list will now be a real collection of 4 entries*

*// Each entry is a new empty String*

It is also possible to configure the behaviour of the SpEL expression compiler.

4.3.3. SpEL compilation

Spring Framework 4.1 includes a basic expression compiler. Expressions are usually interpreted which provides a lot of dynamic flexibility during evaluation but does not provide the optimum performance. For occasional expression usage this is fine, but when used by other components like Spring Integration, performance can be very important and there is no real need for the dynamism.

The new SpEL compiler is intended to address this need. The compiler will generate a real Java class on the fly during evaluation that embodies the expression behavior and use that to achieve much faster expression evaluation. Due to the lack of typing around expressions the compiler uses information gathered during the interpreted evaluations of an expression when performing compilation. For example, it does not know the type of a property reference purely from the expression but during the first interpreted evaluation it will find out what it is. Of course, basing the compilation on this information could cause trouble later if the types of the various expression elements change over time. For this reason compilation is best suited to expressions whose type information is not going to change on repeated evaluations.

For a basic expression like this:

someArray[0].someProperty.someOtherProperty < 0.1

which involves array access, some property derefencing and numeric operations, the performance gain can be very noticeable. In an example micro benchmark run of 50000 iterations, it was taking 75ms to evaluate using only the interpreter and just 3ms using the compiled version of the expression.

Compiler configuration

The compiler is not turned on by default, but there are two ways to turn it on. It can be turned on using the parser configuration process discussed earlier or via a system property when SpEL usage is embedded inside another component. This section discusses both of these options.

It is important to understand that there are a few modes the compiler can operate in, captured in an enum (org.springframework.expression.spel.SpelCompilerMode). The modes are as follows:

* OFF - The compiler is switched off; this is the default.
* IMMEDIATE - In immediate mode the expressions are compiled as soon as possible. This is typically after the first interpreted evaluation. If the compiled expression fails (typically due to a type changing, as described above) then the caller of the expression evaluation will receive an exception.
* MIXED - In mixed mode the expressions silently switch between interpreted and compiled mode over time. After some number of interpreted runs they will switch to compiled form and if something goes wrong with the compiled form (like a type changing, as described above) then the expression will automatically switch back to interpreted form again. Sometime later it may generate another compiled form and switch to it. Basically the exception that the user gets in IMMEDIATE mode is instead handled internally.

IMMEDIATE mode exists because MIXED mode could cause issues for expressions that have side effects. If a compiled expression blows up after partially succeeding it may have already done something that has affected the state of the system. If this has happened the caller may not want it to silently re-run in interpreted mode since part of the expression may be running twice.

After selecting a mode, use the SpelParserConfiguration to configure the parser:

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);

When specifying the compiler mode it is also possible to specify a classloader (passing null is allowed). Compiled expressions will be defined in a child classloader created under any that is supplied. It is important to ensure if a classloader is specified it can see all the types involved in the expression evaluation process. If none is specified then a default classloader will be used (typically the context classloader for the thread that is running during expression evaluation).

The second way to configure the compiler is for use when SpEL is embedded inside some other component and it may not be possible to configure via a configuration object. In these cases it is possible to use a system property. The propertyspring.expression.compiler.mode can be set to one of the SpelCompilerMode enum values (off, immediate, or mixed).

Compiler limitations

With Spring Framework 4.1 the basic compilation framework is in place. However, the framework does not yet support compiling every kind of expression. The initial focus has been on the common expressions that are likely to be used in performance critical contexts. These kinds of expression cannot be compiled at the moment:

* expressions involving assignment
* expressions relying on the conversion service
* expressions using custom resolvers or accessors
* expressions using selection or projection

More and more types of expression will be compilable in the future.

4.4. Expression support for defining bean definitions

SpEL expressions can be used with XML or annotation-based configuration metadata for defining BeanDefinitions. In both cases the syntax to define the expression is of the form #{ <expression string> }.

4.4.1. XML based configuration

A property or constructor-arg value can be set using expressions as shown below.

<bean id="numberGuess" class="org.spring.samples.NumberGuess">

<property name="randomNumber" value="#{ T(java.lang.Math).random() \* 100.0 }"/>

*<!-- other properties -->*

</bean>

The variable systemProperties is predefined, so you can use it in your expressions as shown below. Note that you do not have to prefix the predefined variable with the # symbol in this context.

<bean id="taxCalculator" class="org.spring.samples.TaxCalculator">

<property name="defaultLocale" value="#{ systemProperties['user.region'] }"/>

*<!-- other properties -->*

</bean>

You can also refer to other bean properties by name, for example.

<bean id="numberGuess" class="org.spring.samples.NumberGuess">

<property name="randomNumber" value="#{ T(java.lang.Math).random() \* 100.0 }"/>

*<!-- other properties -->*

</bean>

<bean id="shapeGuess" class="org.spring.samples.ShapeGuess">

<property name="initialShapeSeed" value="#{ numberGuess.randomNumber }"/>

*<!-- other properties -->*

</bean>

4.4.2. Annotation-based configuration

The @Value annotation can be placed on fields, methods and method/constructor parameters to specify a default value.

Here is an example to set the default value of a field variable.

**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;

}

}

The equivalent but on a property setter method is shown below.

**public** **static** **class** **PropertyValueTestBean**

**private** String defaultLocale;

@Value("#{ systemProperties['user.region'] }")

**public** **void** setDefaultLocale(String defaultLocale) {

this.defaultLocale = defaultLocale;

}

**public** String getDefaultLocale() {

**return** this.defaultLocale;

}

}

Autowired methods and constructors can also use the @Value annotation.

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

**private** MovieFinder movieFinder;

**private** String defaultLocale;

@Autowired

**public** **void** configure(MovieFinder movieFinder,

@Value("#{ systemProperties['user.region'] }") String defaultLocale) {

this.movieFinder = movieFinder;

this.defaultLocale = defaultLocale;

}

*// ...*

}

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

**private** String defaultLocale;

**private** CustomerPreferenceDao customerPreferenceDao;

@Autowired

**public** MovieRecommender(CustomerPreferenceDao customerPreferenceDao,

@Value("#{systemProperties['user.country']}") String defaultLocale) {

this.customerPreferenceDao = customerPreferenceDao;

this.defaultLocale = defaultLocale;

}

*// ...*

}

4.5. Language Reference

4.5.1. Literal expressions

The types of literal expressions supported are strings, numeric values (int, real, hex), boolean and null. Strings are delimited by single quotes. To put a single quote itself in a string, use two single quote characters.

The following listing shows simple usage of literals. Typically they would not be used in isolation like this but rather as part of a more complex expression, for example using a literal on one side of a logical comparison operator.

ExpressionParser parser = **new** SpelExpressionParser();

*// evals to "Hello World"*

String helloWorld = (String) parser.parseExpression("'Hello World'").getValue();

**double** avogadrosNumber = (Double) parser.parseExpression("6.0221415E+23").getValue();

*// evals to 2147483647*

**int** maxValue = (Integer) parser.parseExpression("0x7FFFFFFF").getValue();

**boolean** trueValue = (Boolean) parser.parseExpression("true").getValue();

Object nullValue = parser.parseExpression("null").getValue();

Numbers support the use of the negative sign, exponential notation, and decimal points. By default real numbers are parsed using Double.parseDouble().

4.5.2. Properties, Arrays, Lists, Maps, Indexers

Navigating with property references is easy: just use a period to indicate a nested property value. The instances of the Inventorclass, pupin, and tesla, were populated with data listed in the section [Classes used in the examples](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#expressions-example-classes). To navigate "down" and get Tesla’s year of birth and Pupin’s city of birth the following expressions are used.

*// evals to 1856*

**int** year = (Integer) parser.parseExpression("Birthdate.Year + 1900").getValue(context);

String city = (String) parser.parseExpression("placeOfBirth.City").getValue(context);

Case insensitivity is allowed for the first letter of property names. The contents of arrays and lists are obtained using square bracket notation.

ExpressionParser parser = **new** SpelExpressionParser();

*// Inventions Array*

StandardEvaluationContext teslaContext = **new** StandardEvaluationContext(tesla);

*// evaluates to "Induction motor"*

String invention = parser.parseExpression("inventions[3]").getValue(

teslaContext, String.class);

*// Members List*

StandardEvaluationContext societyContext = **new** StandardEvaluationContext(ieee);

*// evaluates to "Nikola Tesla"*

String name = parser.parseExpression("Members[0].Name").getValue(

societyContext, String.class);

*// List and Array navigation*

*// evaluates to "Wireless communication"*

String invention = parser.parseExpression("Members[0].Inventions[6]").getValue(

societyContext, String.class);

The contents of maps are obtained by specifying the literal key value within the brackets. In this case, because keys for the Officers map are strings, we can specify string literals.

*// Officer's Dictionary*

Inventor pupin = parser.parseExpression("Officers['president']").getValue(

societyContext, Inventor.class);

*// evaluates to "Idvor"*

String city = parser.parseExpression("Officers['president'].PlaceOfBirth.City").getValue(

societyContext, String.class);

*// setting values*

parser.parseExpression("Officers['advisors'][0].PlaceOfBirth.Country").setValue(

societyContext, "Croatia");

4.5.3. Inline lists

Lists can be expressed directly in an expression using {} notation.

*// evaluates to a Java list containing the four numbers*

List numbers = (List) parser.parseExpression("{1,2,3,4}").getValue(context);

List listOfLists = (List) parser.parseExpression("{{'a','b'},{'x','y'}}").getValue(context);

{} by itself means an empty list. For performance reasons, if the list is itself entirely composed of fixed literals then a constant list is created to represent the expression, rather than building a new list on each evaluation.

4.5.4. Inline Maps

Maps can also be expressed directly in an expression using {key:value} notation.

*// evaluates to a Java map containing the two entries*

Map inventorInfo = (Map) parser.parseExpression("{name:'Nikola',dob:'10-July-1856'}").getValue(context);

Map mapOfMaps = (Map) parser.parseExpression("{name:{first:'Nikola',last:'Tesla'},dob:{day:10,month:'July',year:1856}}").getValue(context);

{:} by itself means an empty map. For performance reasons, if the map is itself composed of fixed literals or other nested constant structures (lists or maps) then a constant map is created to represent the expression, rather than building a new map on each evaluation. Quoting of the map keys is optional, the examples above are not using quoted keys.

4.5.5. Array construction

Arrays can be built using the familiar Java syntax, optionally supplying an initializer to have the array populated at construction time.

**int[]** numbers1 = (**int[]**) parser.parseExpression("new int[4]").getValue(context);

*// Array with initializer*

**int[]** numbers2 = (**int[]**) parser.parseExpression("new int[]{1,2,3}").getValue(context);

*// Multi dimensional array*

**int[][]** numbers3 = (**int[][]**) parser.parseExpression("new int[4][5]").getValue(context);

It is not currently allowed to supply an initializer when constructing a multi-dimensional array.

4.5.6. Methods

Methods are invoked using typical Java programming syntax. You may also invoke methods on literals. Varargs are also supported.

*// string literal, evaluates to "bc"*

String bc = parser.parseExpression("'abc'.substring(1, 3)").getValue(String.class);

*// evaluates to true*

**boolean** isMember = parser.parseExpression("isMember('Mihajlo Pupin')").getValue(

societyContext, Boolean.class);

4.5.7. Operators

Relational operators

The relational operators; equal, not equal, less than, less than or equal, greater than, and greater than or equal are supported using standard operator notation.

*// evaluates to true*

**boolean** trueValue = parser.parseExpression("2 == 2").getValue(Boolean.class);

*// evaluates to false*

**boolean** falseValue = parser.parseExpression("2 < -5.0").getValue(Boolean.class);

*// evaluates to true*

**boolean** trueValue = parser.parseExpression("'black' < 'block'").getValue(Boolean.class);

|  |  |
| --- | --- |
|  | Greater/less-than comparisons against null follow a simple rule: null is treated as nothing here (i.e. NOT as zero). As a consequence, any other value is always greater than null (X > null is always true) and no other value is ever less than nothing (X < null is always false).  If you prefer numeric comparisons instead, please avoid number-based null comparisons in favor of comparisons against zero (e.g. X > 0 or X < 0). |

In addition to standard relational operators SpEL supports the instanceof and regular expression based matches operator.

*// evaluates to false*

**boolean** falseValue = parser.parseExpression(

"'xyz' instanceof T(Integer)").getValue(Boolean.class);

*// evaluates to true*

**boolean** trueValue = parser.parseExpression(

"'5.00' matches '\^-?\\d+(\\.\\d{2})?$'").getValue(Boolean.class);

*//evaluates to false*

**boolean** falseValue = parser.parseExpression(

"'5.0067' matches '\^-?\\d+(\\.\\d{2})?$'").getValue(Boolean.class);

|  |  |
| --- | --- |
|  | Be careful with primitive types as they are immediately boxed up to the wrapper type, so 1 instanceof T(int)evaluates to false while 1 instanceof T(Integer) evaluates to true, as expected. |

Each symbolic operator can also be specified as a purely alphabetic equivalent. This avoids problems where the symbols used have special meaning for the document type in which the expression is embedded (eg. an XML document). The textual equivalents are shown here: lt (<), gt (>), le (⇐), ge (>=), eq (==), ne (!=), div (/), mod (%), not (!). These are case insensitive.

Logical operators

The logical operators that are supported are and, or, and not. Their use is demonstrated below.

*// -- AND --*

*// evaluates to false*

**boolean** falseValue = parser.parseExpression("true and false").getValue(Boolean.class);

*// evaluates to true*

String expression = "isMember('Nikola Tesla') and isMember('Mihajlo Pupin')";

**boolean** trueValue = parser.parseExpression(expression).getValue(societyContext, Boolean.class);

*// -- OR --*

*// evaluates to true*

**boolean** trueValue = parser.parseExpression("true or false").getValue(Boolean.class);

*// evaluates to true*

String expression = "isMember('Nikola Tesla') or isMember('Albert Einstein')";

**boolean** trueValue = parser.parseExpression(expression).getValue(societyContext, Boolean.class);

*// -- NOT --*

*// evaluates to false*

**boolean** falseValue = parser.parseExpression("!true").getValue(Boolean.class);

*// -- AND and NOT --*

String expression = "isMember('Nikola Tesla') and !isMember('Mihajlo Pupin')";

**boolean** falseValue = parser.parseExpression(expression).getValue(societyContext, Boolean.class);

Mathematical operators

The addition operator can be used on both numbers and strings. Subtraction, multiplication and division can be used only on numbers. Other mathematical operators supported are modulus (%) and exponential power (^). Standard operator precedence is enforced. These operators are demonstrated below.

*// Addition*

**int** two = parser.parseExpression("1 + 1").getValue(Integer.class); *// 2*

String testString = parser.parseExpression(

"'test' + ' ' + 'string'").getValue(String.class); *// 'test string'*

*// Subtraction*

**int** four = parser.parseExpression("1 - -3").getValue(Integer.class); *// 4*

**double** d = parser.parseExpression("1000.00 - 1e4").getValue(Double.class); *// -9000*

*// Multiplication*

**int** six = parser.parseExpression("-2 \* -3").getValue(Integer.class); *// 6*

**double** twentyFour = parser.parseExpression("2.0 \* 3e0 \* 4").getValue(Double.class); *// 24.0*

*// Division*

**int** minusTwo = parser.parseExpression("6 / -3").getValue(Integer.class); *// -2*

**double** one = parser.parseExpression("8.0 / 4e0 / 2").getValue(Double.class); *// 1.0*

*// Modulus*

**int** three = parser.parseExpression("7 % 4").getValue(Integer.class); *// 3*

**int** one = parser.parseExpression("8 / 5 % 2").getValue(Integer.class); *// 1*

*// Operator precedence*

**int** minusTwentyOne = parser.parseExpression("1+2-3\*8").getValue(Integer.class); *// -21*

4.5.8. Assignment

Setting of a property is done by using the assignment operator. This would typically be done within a call to setValue but can also be done inside a call to getValue.

Inventor inventor = **new** Inventor();

StandardEvaluationContext inventorContext = **new** StandardEvaluationContext(inventor);

parser.parseExpression("Name").setValue(inventorContext, "Alexander Seovic2");

*// alternatively*

String aleks = parser.parseExpression(

"Name = 'Alexandar Seovic'").getValue(inventorContext, String.class);

4.5.9. Types

The special T operator can be used to specify an instance of java.lang.Class (the *type*). Static methods are invoked using this operator as well. The StandardEvaluationContext uses a TypeLocator to find types and the StandardTypeLocator (which can be replaced) is built with an understanding of the java.lang package. This means T() references to types within java.lang do not need to be fully qualified, but all other type references must be.

Class dateClass = parser.parseExpression("T(java.util.Date)").getValue(Class.class);

Class stringClass = parser.parseExpression("T(String)").getValue(Class.class);

**boolean** trueValue = parser.parseExpression(

"T(java.math.RoundingMode).CEILING < T(java.math.RoundingMode).FLOOR")

.getValue(Boolean.class);

4.5.10. Constructors

Constructors can be invoked using the new operator. The fully qualified class name should be used for all but the primitive type and String (where int, float, etc, can be used).

Inventor einstein = p.parseExpression(

"new org.spring.samples.spel.inventor.Inventor('Albert Einstein', 'German')")

.getValue(Inventor.class);

*//create new inventor instance within add method of List*

p.parseExpression(

"Members.add(new org.spring.samples.spel.inventor.Inventor(

'Albert Einstein', 'German'))").getValue(societyContext);

4.5.11. Variables

Variables can be referenced in the expression using the syntax #variableName. Variables are set using the method setVariable on the StandardEvaluationContext.

Inventor tesla = **new** Inventor("Nikola Tesla", "Serbian");

StandardEvaluationContext context = **new** StandardEvaluationContext(tesla);

context.setVariable("newName", "Mike Tesla");

parser.parseExpression("Name = #newName").getValue(context);

System.out.println(tesla.getName()) *// "Mike Tesla"*

The #this and #root variables

The variable #this is always defined and refers to the current evaluation object (against which unqualified references are resolved). The variable #root is always defined and refers to the root context object. Although #this may vary as components of an expression are evaluated, #root always refers to the root.

*// create an array of integers*

List<Integer> primes = **new** ArrayList<Integer>();

primes.addAll(Arrays.asList(2,3,5,7,11,13,17));

*// create parser and set variable 'primes' as the array of integers*

ExpressionParser parser = **new** SpelExpressionParser();

StandardEvaluationContext context = **new** StandardEvaluationContext();

context.setVariable("primes",primes);

*// all prime numbers > 10 from the list (using selection ?{...})*

*// evaluates to [11, 13, 17]*

List<Integer> primesGreaterThanTen = (List<Integer>) parser.parseExpression(

"#primes.?[#this>10]").getValue(context);

4.5.12. Functions

You can extend SpEL by registering user defined functions that can be called within the expression string. The function is registered with the StandardEvaluationContext using the method.

**public** **void** registerFunction(String name, Method m)

A reference to a Java Method provides the implementation of the function. For example, a utility method to reverse a string is shown below.

**public** **abstract** **class** **StringUtils** {

**public** **static** String reverseString(String input) {

StringBuilder backwards = **new** StringBuilder();

**for** (**int** i = 0; i < input.length(); i++)

backwards.append(input.charAt(input.length() - 1 - i));

}

**return** backwards.toString();

}

}

This method is then registered with the evaluation context and can be used within an expression string.

ExpressionParser parser = **new** SpelExpressionParser();

StandardEvaluationContext context = **new** StandardEvaluationContext();

context.registerFunction("reverseString",

StringUtils.class.getDeclaredMethod("reverseString", **new** Class**[]** { String.class }));

String helloWorldReversed = parser.parseExpression(

"#reverseString('hello')").getValue(context, String.class);

4.5.13. Bean references

If the evaluation context has been configured with a bean resolver it is possible to lookup beans from an expression using the (@) symbol.

ExpressionParser parser = **new** SpelExpressionParser();

StandardEvaluationContext context = **new** StandardEvaluationContext();

context.setBeanResolver(**new** MyBeanResolver());

*// This will end up calling resolve(context,"foo") on MyBeanResolver during evaluation*

Object bean = parser.parseExpression("@foo").getValue(context);

To access a factory bean itself, the bean name should instead be prefixed with a (&) symbol.

ExpressionParser parser = **new** SpelExpressionParser();

StandardEvaluationContext context = **new** StandardEvaluationContext();

context.setBeanResolver(**new** MyBeanResolver());

*// This will end up calling resolve(context,"&foo") on MyBeanResolver during evaluation*

Object bean = parser.parseExpression("&foo").getValue(context);

4.5.14. Ternary Operator (If-Then-Else)

You can use the ternary operator for performing if-then-else conditional logic inside the expression. A minimal example is:

String falseString = parser.parseExpression(

"false ? 'trueExp' : 'falseExp'").getValue(String.class);

In this case, the boolean false results in returning the string value 'falseExp'. A more realistic example is shown below.

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"*

Also see the next section on the Elvis operator for an even shorter syntax for the ternary operator.

4.5.15. The Elvis Operator

The Elvis operator is a shortening of the ternary operator syntax and is used in the [Groovy](http://www.groovy-lang.org/operators.html#_elvis_operator) language. With the ternary operator syntax you usually have to repeat a variable twice, for example:

String name = "Elvis Presley";

String displayName = name != null ? name : "Unknown";

Instead you can use the Elvis operator, named for the resemblance to Elvis' hair style.

ExpressionParser parser = **new** SpelExpressionParser();

String name = parser.parseExpression("name?:'Unknown'").getValue(String.class);

System.out.println(name); *// 'Unknown'*

Here is a more complex example.

ExpressionParser parser = **new** SpelExpressionParser();

Inventor tesla = **new** Inventor("Nikola Tesla", "Serbian");

StandardEvaluationContext context = **new** StandardEvaluationContext(tesla);

String name = parser.parseExpression("Name?:'Elvis Presley'").getValue(context, String.class);

System.out.println(name); *// Nikola Tesla*

tesla.setName(null);

name = parser.parseExpression("Name?:'Elvis Presley'").getValue(context, String.class);

System.out.println(name); *// Elvis Presley*

4.5.16. Safe Navigation operator

The Safe Navigation operator is used to avoid a NullPointerException and comes from the [Groovy](http://www.groovy-lang.org/operators.html#_safe_navigation_operator) language. Typically when you have a reference to an object you might need to verify that it is not null before accessing methods or properties of the object. To avoid this, the safe navigation operator will simply return null instead of throwing an exception.

ExpressionParser parser = **new** SpelExpressionParser();

Inventor tesla = **new** Inventor("Nikola Tesla", "Serbian");

tesla.setPlaceOfBirth(**new** PlaceOfBirth("Smiljan"));

StandardEvaluationContext context = **new** StandardEvaluationContext(tesla);

String city = parser.parseExpression("PlaceOfBirth?.City").getValue(context, String.class);

System.out.println(city); *// Smiljan*

tesla.setPlaceOfBirth(null);

city = parser.parseExpression("PlaceOfBirth?.City").getValue(context, String.class);

System.out.println(city); *// null - does not throw NullPointerException!!!*

|  |  |
| --- | --- |
|  | The Elvis operator can be used to apply default values in expressions, e.g. in an @Value expression:  @Value("#{systemProperties['pop3.port'] ?: 25}")  This will inject a system property pop3.port if it is defined or 25 if not. |

4.5.17. Collection Selection

Selection is a powerful expression language feature that allows you to transform some source collection into another by selecting from its entries.

Selection uses the syntax .?[selectionExpression]. This will filter the collection and return a new collection containing a subset of the original elements. For example, selection would allow us to easily get a list of Serbian inventors:

List<Inventor> list = (List<Inventor>) parser.parseExpression(

"Members.?[Nationality == 'Serbian']").getValue(societyContext);

Selection is possible upon both lists and maps. In the former case the selection criteria is evaluated against each individual list element whilst against a map the selection criteria is evaluated against each map entry (objects of the Java type Map.Entry). Map entries have their key and value accessible as properties for use in the selection.

This expression will return a new map consisting of those elements of the original map where the entry value is less than 27.

Map newMap = parser.parseExpression("map.?[value<27]").getValue();

In addition to returning all the selected elements, it is possible to retrieve just the first or the last value. To obtain the first entry matching the selection the syntax is ^[…​] whilst to obtain the last matching selection the syntax is $[…​].

4.5.18. Collection Projection

Projection allows a collection to drive the evaluation of a sub-expression and the result is a new collection. The syntax for projection is ![projectionExpression]. Most easily understood by example, suppose we have a list of inventors but want the list of cities where they were born. Effectively we want to evaluate 'placeOfBirth.city' for every entry in the inventor list. Using projection:

*// returns ['Smiljan', 'Idvor' ]*

List placesOfBirth = (List)parser.parseExpression("Members.![placeOfBirth.city]");

A map can also be used to drive projection and in this case the projection expression is evaluated against each entry in the map (represented as a Java Map.Entry). The result of a projection across a map is a list consisting of the evaluation of the projection expression against each map entry.

4.5.19. Expression templating

Expression templates allow a mixing of literal text with one or more evaluation blocks. Each evaluation block is delimited with prefix and suffix characters that you can define, a common choice is to use #{ } as the delimiters. For example,

String randomPhrase = parser.parseExpression(

"random number is #{T(java.lang.Math).random()}",

**new** TemplateParserContext()).getValue(String.class);

*// evaluates to "random number is 0.7038186818312008"*

The string is evaluated by concatenating the literal text 'random number is ' with the result of evaluating the expression inside the #{ } delimiter, in this case the result of calling that random() method. The second argument to the method parseExpression() is of the type ParserContext. The ParserContext interface is used to influence how the expression is parsed in order to support the expression templating functionality. The definition of TemplateParserContext is shown below.

**public** **class** **TemplateParserContext** **implements** ParserContext {

**public** String getExpressionPrefix() {

**return** "#{";

}

**public** String getExpressionSuffix() {

**return** "}";

}

**public** **boolean** isTemplate() {

**return** true;

}

}

4.6. Classes used in the examples

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;

}

}

5. Aspect Oriented Programming with Spring

5.1. Introduction

*Aspect-Oriented Programming* (AOP) complements Object-Oriented Programming (OOP) by providing another way of thinking about program structure. The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the *aspect*. Aspects enable the modularization of concerns such as transaction management that cut across multiple types and objects. (Such concerns are often termed *crosscutting* concerns in AOP literature.)

One of the key components of Spring is the *AOP framework*. While the Spring IoC container does not depend on AOP, meaning you do not need to use AOP if you don’t want to, AOP complements Spring IoC to provide a very capable middleware solution.

Spring 2.0+ AOP

Spring 2.0 introduced a simpler and more powerful way of writing custom aspects using either a [schema-based approach](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-schema)or the [@AspectJ annotation style](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-ataspectj). Both of these styles offer fully typed advice and use of the AspectJ pointcut language, while still using Spring AOP for weaving.

The Spring 2.0+ schema- and @AspectJ-based AOP support is discussed in this chapter. The lower-level AOP support, as commonly exposed in Spring 1.2 applications, is discussed in [the following chapter](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-api).

AOP is used in the Spring Framework to…​

* …​ provide declarative enterprise services, especially as a replacement for EJB declarative services. The most important such service is [*declarative transaction management*](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#transaction-declarative).
* …​ allow users to implement custom aspects, complementing their use of OOP with AOP.

|  |  |
| --- | --- |
|  | If you are interested only in generic declarative services or other pre-packaged declarative middleware services such as pooling, you do not need to work directly with Spring AOP, and can skip most of this chapter. |

5.1.1. AOP concepts

Let us begin by defining some central AOP concepts and terminology. These terms are not Spring-specific…​ unfortunately, AOP terminology is not particularly intuitive; however, it would be even more confusing if Spring used its own terminology.

* *Aspect*: a modularization of a concern that cuts across multiple classes. Transaction management is a good example of a crosscutting concern in enterprise Java applications. In Spring AOP, aspects are implemented using regular classes (the [schema-based approach](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-schema)) or regular classes annotated with the @Aspect annotation (the [@AspectJ style](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-ataspectj)).
* *Join point*: a point during the execution of a program, such as the execution of a method or the handling of an exception. In Spring AOP, a join point *always* represents a method execution.
* *Advice*: action taken by an aspect at a particular join point. Different types of advice include "around," "before" and "after" advice. (Advice types are discussed below.) Many AOP frameworks, including Spring, model an advice as an *interceptor*, maintaining a chain of interceptors *around* the join point.
* *Pointcut*: a predicate that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (for example, the execution of a method with a certain name). The concept of join points as matched by pointcut expressions is central to AOP, and Spring uses the AspectJ pointcut expression language by default.
* *Introduction*: declaring additional methods or fields on behalf of a type. Spring AOP allows you to introduce new interfaces (and a corresponding implementation) to any advised object. For example, you could use an introduction to make a bean implement an IsModified interface, to simplify caching. (An introduction is known as an inter-type declaration in the AspectJ community.)
* *Target object*: object being advised by one or more aspects. Also referred to as the *advised* object. Since Spring AOP is implemented using runtime proxies, this object will always be a *proxied* object.
* *AOP proxy*: an object created by the AOP framework in order to implement the aspect contracts (advise method executions and so on). In the Spring Framework, an AOP proxy will be a JDK dynamic proxy or a CGLIB proxy.
* *Weaving*: linking aspects with other application types or objects to create an advised object. This can be done at compile time (using the AspectJ compiler, for example), load time, or at runtime. Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime.

Types of advice:

* *Before advice*: Advice that executes before a join point, but which does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).
* *After returning advice*: Advice to be executed after a join point completes normally: for example, if a method returns without throwing an exception.
* *After throwing advice*: Advice to be executed if a method exits by throwing an exception.
* *After (finally) advice*: Advice to be executed regardless of the means by which a join point exits (normal or exceptional return).
* *Around advice*: Advice that surrounds a join point such as a method invocation. This is the most powerful kind of advice. Around advice can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception.

Around advice is the most general kind of advice. Since Spring AOP, like AspectJ, provides a full range of advice types, we recommend that you use the least powerful advice type that can implement the required behavior. For example, if you need only to update a cache with the return value of a method, you are better off implementing an after returning advice than an around advice, although an around advice can accomplish the same thing. Using the most specific advice type provides a simpler programming model with less potential for errors. For example, you do not need to invoke the proceed() method on the JoinPoint used for around advice, and hence cannot fail to invoke it.

In Spring 2.0, all advice parameters are statically typed, so that you work with advice parameters of the appropriate type (the type of the return value from a method execution for example) rather than Object arrays.

The concept of join points, matched by pointcuts, is the key to AOP which distinguishes it from older technologies offering only interception. Pointcuts enable advice to be targeted independently of the Object-Oriented hierarchy. For example, an around advice providing declarative transaction management can be applied to a set of methods spanning multiple objects (such as all business operations in the service layer).

5.1.2. Spring AOP capabilities and goals

Spring AOP is implemented in pure Java. There is no need for a special compilation process. Spring AOP does not need to control the class loader hierarchy, and is thus suitable for use in a Servlet container or application server.

Spring AOP currently supports only method execution join points (advising the execution of methods on Spring beans). Field interception is not implemented, although support for field interception could be added without breaking the core Spring AOP APIs. If you need to advise field access and update join points, consider a language such as AspectJ.

Spring AOP’s approach to AOP differs from that of most other AOP frameworks. The aim is not to provide the most complete AOP implementation (although Spring AOP is quite capable); it is rather to provide a close integration between AOP implementation and Spring IoC to help solve common problems in enterprise applications.

Thus, for example, the Spring Framework’s AOP functionality is normally used in conjunction with the Spring IoC container. Aspects are configured using normal bean definition syntax (although this allows powerful "autoproxying" capabilities): this is a crucial difference from other AOP implementations. There are some things you cannot do easily or efficiently with Spring AOP, such as advise very fine-grained objects (such as domain objects typically): AspectJ is the best choice in such cases. However, our experience is that Spring AOP provides an excellent solution to most problems in enterprise Java applications that are amenable to AOP.

Spring AOP will never strive to compete with AspectJ to provide a comprehensive AOP solution. We believe that both proxy-based frameworks like Spring AOP and full-blown frameworks such as AspectJ are valuable, and that they are complementary, rather than in competition. Spring seamlessly integrates Spring AOP and IoC with AspectJ, to enable all uses of AOP to be catered for within a consistent Spring-based application architecture. This integration does not affect the Spring AOP API or the AOP Alliance API: Spring AOP remains backward-compatible. See [the following chapter](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-api) for a discussion of the Spring AOP APIs.

|  |  |
| --- | --- |
|  | One of the central tenets of the Spring Framework is that of *non-invasiveness*; this is the idea that you should not be forced to introduce framework-specific classes and interfaces into your business/domain model. However, in some places the Spring Framework does give you the option to introduce Spring Framework-specific dependencies into your codebase: the rationale in giving you such options is because in certain scenarios it might be just plain easier to read or code some specific piece of functionality in such a way. The Spring Framework (almost) always offers you the choice though: you have the freedom to make an informed decision as to which option best suits your particular use case or scenario.  One such choice that is relevant to this chapter is that of which AOP framework (and which AOP style) to choose. You have the choice of AspectJ and/or Spring AOP, and you also have the choice of either the @AspectJ annotation-style approach or the Spring XML configuration-style approach. The fact that this chapter chooses to introduce the @AspectJ-style approach first should not be taken as an indication that the Spring team favors the @AspectJ annotation-style approach over the Spring XML configuration-style.  See [Choosing which AOP declaration style to use](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-choosing) for a more complete discussion of the whys and wherefores of each style. |

5.1.3. AOP Proxies

Spring AOP defaults to using standard JDK *dynamic proxies* for AOP proxies. This enables any interface (or set of interfaces) to be proxied.

Spring AOP can also use CGLIB proxies. This is necessary to proxy classes rather than interfaces. CGLIB is used by default if a business object does not implement an interface. As it is good practice to program to interfaces rather than classes; business classes normally will implement one or more business interfaces. It is possible to [force the use of CGLIB](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-proxying), in those (hopefully rare) cases where you need to advise a method that is not declared on an interface, or where you need to pass a proxied object to a method as a concrete type.

It is important to grasp the fact that Spring AOP is *proxy-based*. See [Understanding AOP proxies](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-understanding-aop-proxies) for a thorough examination of exactly what this implementation detail actually means.

5.2. @AspectJ support

@AspectJ refers to a style of declaring aspects as regular Java classes annotated with annotations. The @AspectJ style was introduced by the [AspectJ project](https://www.eclipse.org/aspectj) as part of the AspectJ 5 release. Spring interprets the same annotations as AspectJ 5, using a library supplied by AspectJ for pointcut parsing and matching. The AOP runtime is still pure Spring AOP though, and there is no dependency on the AspectJ compiler or weaver.

|  |  |
| --- | --- |
|  | Using the AspectJ compiler and weaver enables use of the full AspectJ language, and is discussed in [Using AspectJ with Spring applications](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-using-aspectj). |

5.2.1. Enabling @AspectJ Support

To use @AspectJ aspects in a Spring configuration you need to enable Spring support for configuring Spring AOP based on @AspectJ aspects, and *autoproxying* beans based on whether or not they are advised by those aspects. By autoproxying we mean that if Spring determines that a bean is advised by one or more aspects, it will automatically generate a proxy for that bean to intercept method invocations and ensure that advice is executed as needed.

The @AspectJ support can be enabled with XML or Java style configuration. In either case you will also need to ensure that AspectJ’s aspectjweaver.jar library is on the classpath of your application (version 1.6.8 or later). This library is available in the'lib' directory of an AspectJ distribution or via the Maven Central repository.

Enabling @AspectJ Support with Java configuration

To enable @AspectJ support with Java @Configuration add the @EnableAspectJAutoProxy annotation:

@Configuration

@EnableAspectJAutoProxy

**public** **class** **AppConfig** {

}

Enabling @AspectJ Support with XML configuration

To enable @AspectJ support with XML based configuration use the aop:aspectj-autoproxy element:

<aop:aspectj-autoproxy/>

This assumes that you are using schema support as described in [XML Schema-based configuration](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#xsd-schemas). See [the AOP schema](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#xsd-schemas-aop) for how to import the tags in the aop namespace.

5.2.2. Declaring an aspect

With the @AspectJ support enabled, any bean defined in your application context with a class that is an @AspectJ aspect (has the @Aspect annotation) will be automatically detected by Spring and used to configure Spring AOP. The following example shows the minimal definition required for a not-very-useful aspect:

A regular bean definition in the application context, pointing to a bean class that has the @Aspect annotation:

<bean id="myAspect" class="org.xyz.NotVeryUsefulAspect">

*<!-- configure properties of aspect here as normal -->*

</bean>

And the NotVeryUsefulAspect class definition, annotated with org.aspectj.lang.annotation.Aspect annotation;

**package** org.xyz;

**import** org.aspectj.lang.annotation.Aspect;

@Aspect

**public** **class** **NotVeryUsefulAspect** {

}

Aspects (classes annotated with @Aspect) may have methods and fields just like any other class. They may also contain pointcut, advice, and introduction (inter-type) declarations.

|  |  |
| --- | --- |
|  | *Autodetecting aspects through component scanning*  You may register aspect classes as regular beans in your Spring XML configuration, or autodetect them through classpath scanning - just like any other Spring-managed bean. However, note that the *@Aspect* annotation is *not* sufficient for autodetection in the classpath: For that purpose, you need to add a separate *@Component*annotation (or alternatively a custom stereotype annotation that qualifies, as per the rules of Spring’s component scanner). |
|  | *Advising aspects with other aspects?*  In Spring AOP, it is *not* possible to have aspects themselves be the target of advice from other aspects. The *@Aspect* annotation on a class marks it as an aspect, and hence excludes it from auto-proxying. |

5.2.3. Declaring a pointcut

Recall that pointcuts determine join points of interest, and thus enable us to control when advice executes. *Spring AOP only supports method execution join points for Spring beans*, so you can think of a pointcut as matching the execution of methods on Spring beans. A pointcut declaration has two parts: a signature comprising a name and any parameters, and a pointcut expression that determines *exactly* which method executions we are interested in. In the @AspectJ annotation-style of AOP, a pointcut signature is provided by a regular method definition, and the pointcut expression is indicated using the @Pointcutannotation (the method serving as the pointcut signature *must* have a void return type).

An example will help make this distinction between a pointcut signature and a pointcut expression clear. The following example defines a pointcut named 'anyOldTransfer' that will match the execution of any method named 'transfer':

@Pointcut("execution(\* transfer(..))")*// the pointcut expression*

**private** **void** anyOldTransfer() {}*// the pointcut signature*

The pointcut expression that forms the value of the @Pointcut annotation is a regular AspectJ 5 pointcut expression. For a full discussion of AspectJ’s pointcut language, see the [AspectJ Programming Guide](https://www.eclipse.org/aspectj/doc/released/progguide/index.html) (and for extensions, the [AspectJ 5 Developers Notebook](https://www.eclipse.org/aspectj/doc/released/adk15notebook/index.html)) or one of the books on AspectJ such as "Eclipse AspectJ" by Colyer et. al. or "AspectJ in Action" by Ramnivas Laddad.

Supported Pointcut Designators

Spring AOP supports the following AspectJ pointcut designators (PCD) for use in pointcut expressions:

Other pointcut types

The full AspectJ pointcut language supports additional pointcut designators that are not supported in Spring. These are: call, get, set, preinitialization, staticinitialization, initialization, handler, adviceexecution, withincode, cflow, cflowbelow, if, @this, and @withincode. Use of these pointcut designators in pointcut expressions interpreted by Spring AOP will result in an IllegalArgumentException being thrown.

The set of pointcut designators supported by Spring AOP may be extended in future releases to support more of the AspectJ pointcut designators.

* *execution* - for matching method execution join points, this is the primary pointcut designator you will use when working with Spring AOP
* *within* - limits matching to join points within certain types (simply the execution of a method declared within a matching type when using Spring AOP)
* *this* - limits matching to join points (the execution of methods when using Spring AOP) where the bean reference (Spring AOP proxy) is an instance of the given type
* *target* - limits matching to join points (the execution of methods when using Spring AOP) where the target object (application object being proxied) is an instance of the given type
* *args* - limits matching to join points (the execution of methods when using Spring AOP) where the arguments are instances of the given types
* *@target* - limits matching to join points (the execution of methods when using Spring AOP) where the class of the executing object has an annotation of the given type
* *@args* - limits matching to join points (the execution of methods when using Spring AOP) where the runtime type of the actual arguments passed have annotations of the given type(s)
* *@within* - limits matching to join points within types that have the given annotation (the execution of methods declared in types with the given annotation when using Spring AOP)
* *@annotation* - limits matching to join points where the subject of the join point (method being executed in Spring AOP) has the given annotation

Because Spring AOP limits matching to only method execution join points, the discussion of the pointcut designators above gives a narrower definition than you will find in the AspectJ programming guide. In addition, AspectJ itself has type-based semantics and at an execution join point both this and target refer to the same object - the object executing the method. Spring AOP is a proxy-based system and differentiates between the proxy object itself (bound to this) and the target object behind the proxy (bound to target).

|  |  |
| --- | --- |
|  | Due to the proxy-based nature of Spring’s AOP framework, calls within the target object are by definition *not*intercepted. For JDK proxies, only public interface method calls on the proxy can be intercepted. With CGLIB, public and protected method calls on the proxy will be intercepted, and even package-visible methods if necessary. However, common interactions through proxies should always be designed through public signatures.  Note that pointcut definitions are generally matched against any intercepted method. If a pointcut is strictly meant to be public-only, even in a CGLIB proxy scenario with potential non-public interactions through proxies, it needs to be defined accordingly.  If your interception needs include method calls or even constructors within the target class, consider the use of Spring-driven [native AspectJ weaving](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-aj-ltw) instead of Spring’s proxy-based AOP framework. This constitutes a different mode of AOP usage with different characteristics, so be sure to make yourself familiar with weaving first before making a decision. |

Spring AOP also supports an additional PCD named bean. This PCD allows you to limit the matching of join points to a particular named Spring bean, or to a set of named Spring beans (when using wildcards). The bean PCD has the following form:

bean(idOrNameOfBean)

The idOrNameOfBean token can be the name of any Spring bean: limited wildcard support using the \* character is provided, so if you establish some naming conventions for your Spring beans you can quite easily write a bean PCD expression to pick them out. As is the case with other pointcut designators, the bean PCD can be &&'ed, ||'ed, and ! (negated) too.

|  |  |
| --- | --- |
|  | Please note that the bean PCD is *only* supported in Spring AOP - and *not* in native AspectJ weaving. It is a Spring-specific extension to the standard PCDs that AspectJ defines and therefore not available for aspects declared in the @Aspect model.  The bean PCD operates at the *instance* level (building on the Spring bean name concept) rather than at the type level only (which is what weaving-based AOP is limited to). Instance-based pointcut designators are a special capability of Spring’s proxy-based AOP framework and its close integration with the Spring bean factory, where it is natural and straightforward to identify specific beans by name. |

Combining pointcut expressions

Pointcut expressions can be combined using '&&', '||' and '!'. It is also possible to refer to pointcut expressions by name. The following example shows three pointcut expressions: anyPublicOperation (which matches if a method execution join point represents the execution of any public method); inTrading (which matches if a method execution is in the trading module), and tradingOperation (which matches if a method execution represents any public method in the trading module).

@Pointcut("execution(public \* \*(..))")

**private** **void** anyPublicOperation() {}

@Pointcut("within(com.xyz.someapp.trading..\*)")

**private** **void** inTrading() {}

@Pointcut("anyPublicOperation() && inTrading()")

**private** **void** tradingOperation() {}

It is a best practice to build more complex pointcut expressions out of smaller named components as shown above. When referring to pointcuts by name, normal Java visibility rules apply (you can see private pointcuts in the same type, protected pointcuts in the hierarchy, public pointcuts anywhere and so on). Visibility does not affect pointcut *matching*.

Sharing common pointcut definitions

When working with enterprise applications, you often want to refer to modules of the application and particular sets of operations from within several aspects. We recommend defining a "SystemArchitecture" aspect that captures common pointcut expressions for this purpose. A typical such aspect would look as follows:

**package** com.xyz.someapp;

**import** org.aspectj.lang.annotation.Aspect;

**import** org.aspectj.lang.annotation.Pointcut;

@Aspect

**public** **class** **SystemArchitecture** {

*/\*\**

*\* A join point is in the web layer if the method is defined*

*\* in a type in the com.xyz.someapp.web package or any sub-package*

*\* under that.*

*\*/*

@Pointcut("within(com.xyz.someapp.web..\*)")

**public** **void** inWebLayer() {}

*/\*\**

*\* A join point is in the service layer if the method is defined*

*\* in a type in the com.xyz.someapp.service package or any sub-package*

*\* under that.*

*\*/*

@Pointcut("within(com.xyz.someapp.service..\*)")

**public** **void** inServiceLayer() {}

*/\*\**

*\* A join point is in the data access layer if the method is defined*

*\* in a type in the com.xyz.someapp.dao package or any sub-package*

*\* under that.*

*\*/*

@Pointcut("within(com.xyz.someapp.dao..\*)")

**public** **void** inDataAccessLayer() {}

*/\*\**

*\* A business service is the execution of any method defined on a service*

*\* interface. This definition assumes that interfaces are placed in the*

*\* "service" package, and that implementation types are in sub-packages.*

*\**

*\* If you group service interfaces by functional area (for example,*

*\* in packages com.xyz.someapp.abc.service and com.xyz.someapp.def.service) then*

*\* the pointcut expression "execution(\* com.xyz.someapp..service.\*.\*(..))"*

*\* could be used instead.*

*\**

*\* Alternatively, you can write the expression using the 'bean'*

*\* PCD, like so "bean(\*Service)". (This assumes that you have*

*\* named your Spring service beans in a consistent fashion.)*

*\*/*

@Pointcut("execution(\* com.xyz.someapp..service.\*.\*(..))")

**public** **void** businessService() {}

*/\*\**

*\* A data access operation is the execution of any method defined on a*

*\* dao interface. This definition assumes that interfaces are placed in the*

*\* "dao" package, and that implementation types are in sub-packages.*

*\*/*

@Pointcut("execution(\* com.xyz.someapp.dao.\*.\*(..))")

**public** **void** dataAccessOperation() {}

}

The pointcuts defined in such an aspect can be referred to anywhere that you need a pointcut expression. For example, to make the service layer transactional, you could write:

<aop:config>

<aop:advisor

pointcut="com.xyz.someapp.SystemArchitecture.businessService()"

advice-ref="tx-advice"/>

</aop:config>

<tx:advice id="tx-advice">

<tx:attributes>

<tx:method name="\*" propagation="REQUIRED"/>

</tx:attributes>

</tx:advice>

The <aop:config> and <aop:advisor> elements are discussed in [Schema-based AOP support](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-schema). The transaction elements are discussed in [Transaction Management](https://docs.spring.io/spring/docs/current/spring-framework-reference/data-access.html#transaction).

Examples

Spring AOP users are likely to use the execution pointcut designator the most often. The format of an execution expression is:

execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?name-pattern(param-pattern)

**throws**-pattern?)

All parts except the returning type pattern (ret-type-pattern in the snippet above), name pattern, and parameters pattern are optional. The returning type pattern determines what the return type of the method must be in order for a join point to be matched. Most frequently you will use \* as the returning type pattern, which matches any return type. A fully-qualified type name will match only when the method returns the given type. The name pattern matches the method name. You can use the \*wildcard as all or part of a name pattern. If specifying a declaring type pattern then include a trailing . to join it to the name pattern component. The parameters pattern is slightly more complex: () matches a method that takes no parameters, whereas (..) matches any number of parameters (zero or more). The pattern (\*) matches a method taking one parameter of any type,(\*,String) matches a method taking two parameters, the first can be of any type, the second must be a String. Consult the[Language Semantics](https://www.eclipse.org/aspectj/doc/released/progguide/semantics-pointcuts.html) section of the AspectJ Programming Guide for more information.

Some examples of common pointcut expressions are given below.

* the execution of any public method:

execution(**public** \* \*(..))

* the execution of any method with a name beginning with "set":

execution(\* set\*(..))

* the execution of any method defined by the AccountService interface:

execution(\* com.xyz.service.AccountService.\*(..))

* the execution of any method defined in the service package:

execution(\* com.xyz.service.\*.\*(..))

* the execution of any method defined in the service package or a sub-package:

execution(\* com.xyz.service..\*.\*(..))

* any join point (method execution only in Spring AOP) within the service package:

within(com.xyz.service.\*)

* any join point (method execution only in Spring AOP) within the service package or a sub-package:

within(com.xyz.service..\*)

* any join point (method execution only in Spring AOP) where the proxy implements the AccountService interface:

this(com.xyz.service.AccountService)

|  |  |
| --- | --- |
|  | 'this' is more commonly used in a binding form :- see the following section on advice for how to make the proxy object available in the advice body. |

* any join point (method execution only in Spring AOP) where the target object implements the AccountService interface:

target(com.xyz.service.AccountService)

|  |  |
| --- | --- |
|  | 'target' is more commonly used in a binding form :- see the following section on advice for how to make the target object available in the advice body. |

* any join point (method execution only in Spring AOP) which takes a single parameter, and where the argument passed at runtime is Serializable:

args(java.io.Serializable)

|  |  |
| --- | --- |
|  | 'args' is more commonly used in a binding form :- see the following section on advice for how to make the method arguments available in the advice body. |

Note that the pointcut given in this example is different to execution(\* \*(java.io.Serializable)): the args version matches if the argument passed at runtime is Serializable, the execution version matches if the method signature declares a single parameter of type Serializable.

* any join point (method execution only in Spring AOP) where the target object has an @Transactional annotation:

@target(org.springframework.transaction.annotation.Transactional)

|  |  |
| --- | --- |
|  | '@target' can also be used in a binding form :- see the following section on advice for how to make the annotation object available in the advice body. |

* any join point (method execution only in Spring AOP) where the declared type of the target object has an @Transactionalannotation:

@within(org.springframework.transaction.annotation.Transactional)

|  |  |
| --- | --- |
|  | '@within' can also be used in a binding form :- see the following section on advice for how to make the annotation object available in the advice body. |

* any join point (method execution only in Spring AOP) where the executing method has an @Transactional annotation:

@annotation(org.springframework.transaction.annotation.Transactional)

|  |  |
| --- | --- |
|  | '@annotation' can also be used in a binding form :- see the following section on advice for how to make the annotation object available in the advice body. |

* any join point (method execution only in Spring AOP) which takes a single parameter, and where the runtime type of the argument passed has the @Classified annotation:

@args(com.xyz.security.Classified)

|  |  |
| --- | --- |
|  | '@args' can also be used in a binding form :- see the following section on advice for how to make the annotation object(s) available in the advice body. |

* any join point (method execution only in Spring AOP) on a Spring bean named tradeService:

bean(tradeService)

* any join point (method execution only in Spring AOP) on Spring beans having names that match the wildcard expression \*Service:

bean(\*Service)

Writing good pointcuts

During compilation, AspectJ processes pointcuts in order to try and optimize matching performance. Examining code and determining if each join point matches (statically or dynamically) a given pointcut is a costly process. (A dynamic match means the match cannot be fully determined from static analysis and a test will be placed in the code to determine if there is an actual match when the code is running). On first encountering a pointcut declaration, AspectJ will rewrite it into an optimal form for the matching process. What does this mean? Basically pointcuts are rewritten in DNF (Disjunctive Normal Form) and the components of the pointcut are sorted such that those components that are cheaper to evaluate are checked first. This means you do not have to worry about understanding the performance of various pointcut designators and may supply them in any order in a pointcut declaration.

However, AspectJ can only work with what it is told, and for optimal performance of matching you should think about what they are trying to achieve and narrow the search space for matches as much as possible in the definition. The existing designators naturally fall into one of three groups: kinded, scoping and context:

* Kinded designators are those which select a particular kind of join point. For example: execution, get, set, call, handler
* Scoping designators are those which select a group of join points of interest (of probably many kinds). For example: within, withincode
* Contextual designators are those that match (and optionally bind) based on context. For example: this, target, @annotation

A well written pointcut should try and include at least the first two types (kinded and scoping), whilst the contextual designators may be included if wishing to match based on join point context, or bind that context for use in the advice. Supplying either just a kinded designator or just a contextual designator will work but could affect weaving performance (time and memory used) due to all the extra processing and analysis. Scoping designators are very fast to match and their usage means AspectJ can very quickly dismiss groups of join points that should not be further processed - that is why a good pointcut should always include one if possible.

5.2.4. Declaring advice

Advice is associated with a pointcut expression, and runs before, after, or around method executions matched by the pointcut. The pointcut expression may be either a simple reference to a named pointcut, or a pointcut expression declared in place.

Before advice

Before advice is declared in an aspect using the @Before annotation:

**import** org.aspectj.lang.annotation.Aspect;

**import** org.aspectj.lang.annotation.Before;

@Aspect

**public** **class** **BeforeExample** {

@Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")

**public** **void** doAccessCheck() {

*// ...*

}

}

If using an in-place pointcut expression we could rewrite the above example as:

**import** org.aspectj.lang.annotation.Aspect;

**import** org.aspectj.lang.annotation.Before;

@Aspect

**public** **class** **BeforeExample** {

@Before("execution(\* com.xyz.myapp.dao.\*.\*(..))")

**public** **void** doAccessCheck() {

*// ...*

}

}

After returning advice

After returning advice runs when a matched method execution returns normally. It is declared using the @AfterReturningannotation:

**import** org.aspectj.lang.annotation.Aspect;

**import** org.aspectj.lang.annotation.AfterReturning;

@Aspect

**public** **class** **AfterReturningExample** {

@AfterReturning("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")

**public** **void** doAccessCheck() {

*// ...*

}

}

|  |  |
| --- | --- |
|  | Note: it is of course possible to have multiple advice declarations, and other members as well, all inside the same aspect. We’re just showing a single advice declaration in these examples to focus on the issue under discussion at the time. |

Sometimes you need access in the advice body to the actual value that was returned. You can use the form of @AfterReturningthat binds the return value for this:

**import** org.aspectj.lang.annotation.Aspect;

**import** org.aspectj.lang.annotation.AfterReturning;

@Aspect

**public** **class** **AfterReturningExample** {

@AfterReturning(

pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()",

returning="retVal")

**public** **void** doAccessCheck(Object retVal) {

*// ...*

}

}

The name used in the returning attribute must correspond to the name of a parameter in the advice method. When a method execution returns, the return value will be passed to the advice method as the corresponding argument value. A returningclause also restricts matching to only those method executions that return a value of the specified type ( Object in this case, which will match any return value).

Please note that it is *not* possible to return a totally different reference when using after-returning advice.

After throwing advice

After throwing advice runs when a matched method execution exits by throwing an exception. It is declared using the @AfterThrowing annotation:

**import** org.aspectj.lang.annotation.Aspect;

**import** org.aspectj.lang.annotation.AfterThrowing;

@Aspect

**public** **class** **AfterThrowingExample** {

@AfterThrowing("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")

**public** **void** doRecoveryActions() {

*// ...*

}

}

Often you want the advice to run only when exceptions of a given type are thrown, and you also often need access to the thrown exception in the advice body. Use the throwing attribute to both restrict matching (if desired, use Throwable as the exception type otherwise) and bind the thrown exception to an advice parameter.

**import** org.aspectj.lang.annotation.Aspect;

**import** org.aspectj.lang.annotation.AfterThrowing;

@Aspect

**public** **class** **AfterThrowingExample** {

@AfterThrowing(

pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()",

throwing="ex")

**public** **void** doRecoveryActions(DataAccessException ex) {

*// ...*

}

}

The name used in the throwing attribute must correspond to the name of a parameter in the advice method. When a method execution exits by throwing an exception, the exception will be passed to the advice method as the corresponding argument value. A throwing clause also restricts matching to only those method executions that throw an exception of the specified type ( DataAccessException in this case).

After (finally) advice

After (finally) advice runs however a matched method execution exits. It is declared using the @After annotation. After advice must be prepared to handle both normal and exception return conditions. It is typically used for releasing resources, etc.

**import** org.aspectj.lang.annotation.Aspect;

**import** org.aspectj.lang.annotation.After;

@Aspect

**public** **class** **AfterFinallyExample** {

@After("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")

**public** **void** doReleaseLock() {

*// ...*

}

}

Around advice

The final kind of advice is around advice. Around advice runs "around" a matched method execution. It has the opportunity to do work both before and after the method executes, and to determine when, how, and even if, the method actually gets to execute at all. Around advice is often used if you need to share state before and after a method execution in a thread-safe manner (starting and stopping a timer for example). Always use the least powerful form of advice that meets your requirements (i.e. don’t use around advice if simple before advice would do).

Around advice is declared using the @Around annotation. The first parameter of the advice method must be of type ProceedingJoinPoint. Within the body of the advice, calling proceed() on the ProceedingJoinPoint causes the underlying method to execute. The proceed method may also be called passing in an Object[] - the values in the array will be used as the arguments to the method execution when it proceeds.

|  |  |
| --- | --- |
|  | The behavior of proceed when called with an Object[] is a little different than the behavior of proceed for around advice compiled by the AspectJ compiler. For around advice written using the traditional AspectJ language, the number of arguments passed to proceed must match the number of arguments passed to the around advice (not the number of arguments taken by the underlying join point), and the value passed to proceed in a given argument position supplants the original value at the join point for the entity the value was bound to (Don’t worry if this doesn’t make sense right now!). The approach taken by Spring is simpler and a better match to its proxy-based, execution only semantics. You only need to be aware of this difference if you are compiling @AspectJ aspects written for Spring and using proceed with arguments with the AspectJ compiler and weaver. There is a way to write such aspects that is 100% compatible across both Spring AOP and AspectJ, and this is discussed in the following section on advice parameters. |

**import** org.aspectj.lang.annotation.Aspect;

**import** org.aspectj.lang.annotation.Around;

**import** org.aspectj.lang.ProceedingJoinPoint;

@Aspect

**public** **class** **AroundExample** {

@Around("com.xyz.myapp.SystemArchitecture.businessService()")

**public** Object doBasicProfiling(ProceedingJoinPoint pjp) **throws** Throwable {

*// start stopwatch*

Object retVal = pjp.proceed();

*// stop stopwatch*

**return** retVal;

}

}

The value returned by the around advice will be the return value seen by the caller of the method. A simple caching aspect for example could return a value from a cache if it has one, and invoke proceed() if it does not. Note that proceed may be invoked once, many times, or not at all within the body of the around advice, all of these are quite legal.

Advice parameters

Spring offers fully typed advice - meaning that you declare the parameters you need in the advice signature (as we saw for the returning and throwing examples above) rather than work with Object[] arrays all the time. We’ll see how to make argument and other contextual values available to the advice body in a moment. First let’s take a look at how to write generic advice that can find out about the method the advice is currently advising.

Access to the current JoinPoint

Any advice method may declare as its first parameter, a parameter of type org.aspectj.lang.JoinPoint (please note that around advice is *required* to declare a first parameter of type ProceedingJoinPoint, which is a subclass of JoinPoint. TheJoinPoint interface provides a number of useful methods such as getArgs() (returns the method arguments), getThis()(returns the proxy object), getTarget() (returns the target object), getSignature() (returns a description of the method that is being advised) and toString() (prints a useful description of the method being advised). Please do consult the javadocs for full details.

Passing parameters to advice

We’ve already seen how to bind the returned value or exception value (using after returning and after throwing advice). To make argument values available to the advice body, you can use the binding form of args. If a parameter name is used in place of a type name in an args expression, then the value of the corresponding argument will be passed as the parameter value when the advice is invoked. An example should make this clearer. Suppose you want to advise the execution of dao operations that take an Account object as the first parameter, and you need access to the account in the advice body. You could write the following:

@Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation() && args(account,..)")

**public** **void** validateAccount(Account account) {

*// ...*

}

The args(account,..) part of the pointcut expression serves two purposes: firstly, it restricts matching to only those method executions where the method takes at least one parameter, and the argument passed to that parameter is an instance of Account; secondly, it makes the actual Account object available to the advice via the account parameter.

Another way of writing this is to declare a pointcut that "provides" the Account object value when it matches a join point, and then just refer to the named pointcut from the advice. This would look as follows:

@Pointcut("com.xyz.myapp.SystemArchitecture.dataAccessOperation() && args(account,..)")

**private** **void** accountDataAccessOperation(Account account) {}

@Before("accountDataAccessOperation(account)")

**public** **void** validateAccount(Account account) {

*// ...*

}

The interested reader is once more referred to the AspectJ programming guide for more details.

The proxy object ( this), target object ( target), and annotations ( @within, @target, @annotation, @args) can all be bound in a similar fashion. The following example shows how you could match the execution of methods annotated with an @Auditableannotation, and extract the audit code.

First the definition of the @Auditable annotation:

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.METHOD)

**public** @interface Auditable {

AuditCode value();

}

And then the advice that matches the execution of @Auditable methods:

@Before("com.xyz.lib.Pointcuts.anyPublicMethod() && @annotation(auditable)")

**public** **void** audit(Auditable auditable) {

AuditCode code = auditable.value();

*// ...*

}

Advice parameters and generics

Spring AOP can handle generics used in class declarations and method parameters. Suppose you have a generic type like this:

**public** **interface** **Sample**<T> {

**void** sampleGenericMethod(T param);

**void** sampleGenericCollectionMethod(Collection<T> param);

}

You can restrict interception of method types to certain parameter types by simply typing the advice parameter to the parameter type you want to intercept the method for:

@Before("execution(\* ..Sample+.sampleGenericMethod(\*)) && args(param)")

**public** **void** beforeSampleMethod(MyType param) {

*// Advice implementation*

}

That this works is pretty obvious as we already discussed above. However, it’s worth pointing out that this won’t work for generic collections. So you cannot define a pointcut like this:

@Before("execution(\* ..Sample+.sampleGenericCollectionMethod(\*)) && args(param)")

**public** **void** beforeSampleMethod(Collection<MyType> param) {

*// Advice implementation*

}

To make this work we would have to inspect every element of the collection, which is not reasonable as we also cannot decide how to treat null values in general. To achieve something similar to this you have to type the parameter to Collection<?> and manually check the type of the elements.

Determining argument names

The parameter binding in advice invocations relies on matching names used in pointcut expressions to declared parameter names in (advice and pointcut) method signatures. Parameter names are *not* available through Java reflection, so Spring AOP uses the following strategies to determine parameter names:

* If the parameter names have been specified by the user explicitly, then the specified parameter names are used: both the advice and the pointcut annotations have an optional "argNames" attribute which can be used to specify the argument names of the annotated method - these argument names *are* available at runtime. For example:

@Before(value="com.xyz.lib.Pointcuts.anyPublicMethod() && target(bean) && @annotation(auditable)",

argNames="bean,auditable")

**public** **void** audit(Object bean, Auditable auditable) {

AuditCode code = auditable.value();

*// ... use code and bean*

}

If the first parameter is of the JoinPoint, ProceedingJoinPoint, or JoinPoint.StaticPart type, you may leave out the name of the parameter from the value of the "argNames" attribute. For example, if you modify the preceding advice to receive the join point object, the "argNames" attribute need not include it:

@Before(value="com.xyz.lib.Pointcuts.anyPublicMethod() && target(bean) && @annotation(auditable)",

argNames="bean,auditable")

**public** **void** audit(JoinPoint jp, Object bean, Auditable auditable) {

AuditCode code = auditable.value();

*// ... use code, bean, and jp*

}

The special treatment given to the first parameter of the JoinPoint, ProceedingJoinPoint, and JoinPoint.StaticPart types is particularly convenient for advice that do not collect any other join point context. In such situations, you may simply omit the "argNames" attribute. For example, the following advice need not declare the "argNames" attribute:

@Before("com.xyz.lib.Pointcuts.anyPublicMethod()")

**public** **void** audit(JoinPoint jp) {

*// ... use jp*

}

* Using the 'argNames' attribute is a little clumsy, so if the 'argNames' attribute has not been specified, then Spring AOP will look at the debug information for the class and try to determine the parameter names from the local variable table. This information will be present as long as the classes have been compiled with debug information ( '-g:vars' at a minimum). The consequences of compiling with this flag on are: (1) your code will be slightly easier to understand (reverse engineer), (2) the class file sizes will be very slightly bigger (typically inconsequential), (3) the optimization to remove unused local variables will not be applied by your compiler. In other words, you should encounter no difficulties building with this flag on.

|  |  |
| --- | --- |
|  | If an @AspectJ aspect has been compiled by the AspectJ compiler (ajc) even without the debug information then there is no need to add the argNames attribute as the compiler will retain the needed information. |

* If the code has been compiled without the necessary debug information, then Spring AOP will attempt to deduce the pairing of binding variables to parameters (for example, if only one variable is bound in the pointcut expression, and the advice method only takes one parameter, the pairing is obvious!). If the binding of variables is ambiguous given the available information, then an AmbiguousBindingException will be thrown.
* If all of the above strategies fail then an IllegalArgumentException will be thrown.

Proceeding with arguments

We remarked earlier that we would describe how to write a proceed call *with arguments* that works consistently across Spring AOP and AspectJ. The solution is simply to ensure that the advice signature binds each of the method parameters in order. For example:

@Around("execution(List<Account> find\*(..)) && " +

"com.xyz.myapp.SystemArchitecture.inDataAccessLayer() && " +

"args(accountHolderNamePattern)")

**public** Object preProcessQueryPattern(ProceedingJoinPoint pjp,

String accountHolderNamePattern) **throws** Throwable {

String newPattern = preProcess(accountHolderNamePattern);

**return** pjp.proceed(**new** Object**[]** {newPattern});

}

In many cases you will be doing this binding anyway (as in the example above).

Advice ordering

What happens when multiple pieces of advice all want to run at the same join point? Spring AOP follows the same precedence rules as AspectJ to determine the order of advice execution. The highest precedence advice runs first "on the way in" (so given two pieces of before advice, the one with highest precedence runs first). "On the way out" from a join point, the highest precedence advice runs last (so given two pieces of after advice, the one with the highest precedence will run second).

When two pieces of advice defined in *different* aspects both need to run at the same join point, unless you specify otherwise the order of execution is undefined. You can control the order of execution by specifying precedence. This is done in the normal Spring way by either implementing the org.springframework.core.Ordered interface in the aspect class or annotating it with the Order annotation. Given two aspects, the aspect returning the lower value from Ordered.getValue() (or the annotation value) has the higher precedence.

When two pieces of advice defined in *the same* aspect both need to run at the same join point, the ordering is undefined (since there is no way to retrieve the declaration order via reflection for javac-compiled classes). Consider collapsing such advice methods into one advice method per join point in each aspect class, or refactor the pieces of advice into separate aspect classes - which can be ordered at the aspect level.

5.2.5. Introductions

Introductions (known as inter-type declarations in AspectJ) enable an aspect to declare that advised objects implement a given interface, and to provide an implementation of that interface on behalf of those objects.

An introduction is made using the @DeclareParents annotation. This annotation is used to declare that matching types have a new parent (hence the name). For example, given an interface UsageTracked, and an implementation of that interface DefaultUsageTracked, the following aspect declares that all implementors of service interfaces also implement the UsageTracked interface. (In order to expose statistics via JMX for example.)

@Aspect

**public** **class** **UsageTracking** {

@DeclareParents(value="com.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();

}

}

The interface to be implemented is determined by the type of the annotated field. The value attribute of the @DeclareParentsannotation is an AspectJ type pattern :- any bean of a matching type will implement the UsageTracked interface. Note that in the before advice of the above example, service beans can be directly used as implementations of the UsageTracked interface. If accessing a bean programmatically you would write the following:

UsageTracked usageTracked = (UsageTracked) context.getBean("myService");

5.2.6. Aspect instantiation models

|  |  |
| --- | --- |
|  | (This is an advanced topic, so if you are just starting out with AOP you can safely skip it until later.) |

By default there will be a single instance of each aspect within the application context. AspectJ calls this the singleton instantiation model. It is possible to define aspects with alternate lifecycles :- Spring supports AspectJ’s perthis and pertargetinstantiation models ( percflow, percflowbelow, and pertypewithin are not currently supported).

A "perthis" aspect is declared by specifying a perthis clause in the @Aspect annotation. Let’s look at an example, and then we’ll explain how it works.

@Aspect("perthis(com.xyz.myapp.SystemArchitecture.businessService())")

**public** **class** **MyAspect** {

**private** **int** someState;

@Before(com.xyz.myapp.SystemArchitecture.businessService())

**public** **void** recordServiceUsage() {

*// ...*

}

}

The effect of the 'perthis' clause is that one aspect instance will be created for each unique service object executing a business service (each unique object bound to 'this' at join points matched by the pointcut expression). The aspect instance is created the first time that a method is invoked on the service object. The aspect goes out of scope when the service object goes out of scope. Before the aspect instance is created, none of the advice within it executes. As soon as the aspect instance has been created, the advice declared within it will execute at matched join points, but only when the service object is the one this aspect is associated with. See the AspectJ programming guide for more information on per-clauses.

The 'pertarget' instantiation model works in exactly the same way as perthis, but creates one aspect instance for each unique target object at matched join points.

5.2.7. Example

Now that you have seen how all the constituent parts work, let’s put them together to do something useful!

The execution of business services can sometimes fail due to concurrency issues (for example, deadlock loser). If the operation is retried, it is quite likely to succeed next time round. For business services where it is appropriate to retry in such conditions (idempotent operations that don’t need to go back to the user for conflict resolution), we’d like to transparently retry the operation to avoid the client seeing a PessimisticLockingFailureException. This is a requirement that clearly cuts across multiple services in the service layer, and hence is ideal for implementing via an aspect.

Because we want to retry the operation, we will need to use around advice so that we can call proceed multiple times. Here’s how the basic aspect implementation looks:

@Aspect

**public** **class** **ConcurrentOperationExecutor** **implements** Ordered {

**private** **static** **final** **int** DEFAULT\_MAX\_RETRIES = 2;

**private** **int** maxRetries = DEFAULT\_MAX\_RETRIES;

**private** **int** order = 1;

**public** **void** setMaxRetries(**int** maxRetries) {

this.maxRetries = maxRetries;

}

**public** **int** getOrder() {

**return** this.order;

}

**public** **void** setOrder(**int** order) {

this.order = order;

}

@Around("com.xyz.myapp.SystemArchitecture.businessService()")

**public** Object doConcurrentOperation(ProceedingJoinPoint pjp) **throws** Throwable {

**int** numAttempts = 0;

PessimisticLockingFailureException lockFailureException;

**do** {

numAttempts++;

**try** {

**return** pjp.proceed();

}

**catch**(PessimisticLockingFailureException ex) {

lockFailureException = ex;

}

} **while**(numAttempts <= this.maxRetries);

**throw** lockFailureException;

}

}

Note that the aspect implements the Ordered interface so we can set the precedence of the aspect higher than the transaction advice (we want a fresh transaction each time we retry). The maxRetries and order properties will both be configured by Spring. The main action happens in the doConcurrentOperation around advice. Notice that for the moment we’re applying the retry logic to all businessService()s. We try to proceed, and if we fail with an PessimisticLockingFailureException we simply try again unless we have exhausted all of our retry attempts.

The corresponding Spring configuration is:

<aop:aspectj-autoproxy/>

<bean id="concurrentOperationExecutor" class="com.xyz.myapp.service.impl.ConcurrentOperationExecutor">

<property name="maxRetries" value="3"/>

<property name="order" value="100"/>

</bean>

To refine the aspect so that it only retries idempotent operations, we might define an Idempotent annotation:

@Retention(RetentionPolicy.RUNTIME)

**public** @interface Idempotent {

*// marker annotation*

}

and use the annotation to annotate the implementation of service operations. The change to the aspect to only retry idempotent operations simply involves refining the pointcut expression so that only @Idempotent operations match:

@Around("com.xyz.myapp.SystemArchitecture.businessService() && " +

"@annotation(com.xyz.myapp.service.Idempotent)")

**public** Object doConcurrentOperation(ProceedingJoinPoint pjp) **throws** Throwable {

...

}

5.3. Schema-based AOP support

If you prefer an XML-based format, then Spring also offers support for defining aspects using the new "aop" namespace tags. The exact same pointcut expressions and advice kinds are supported as when using the @AspectJ style, hence in this section we will focus on the new *syntax* and refer the reader to the discussion in the previous section ([@AspectJ support](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-ataspectj)) for an understanding of writing pointcut expressions and the binding of advice parameters.

To use the aop namespace tags described in this section, you need to import the spring-aop schema as described in [XML Schema-based configuration](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#xsd-schemas). See [the AOP schema](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#xsd-schemas-aop) for how to import the tags in the aop namespace.

Within your Spring configurations, all aspect and advisor elements must be placed within an <aop:config> element (you can have more than one <aop:config> element in an application context configuration). An <aop:config> element can contain pointcut, advisor, and aspect elements (note these must be declared in that order).

|  |  |
| --- | --- |
|  | The <aop:config> style of configuration makes heavy use of Spring’s [auto-proxying](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-autoproxy) mechanism. This can cause issues (such as advice not being woven) if you are already using explicit auto-proxying via the use ofBeanNameAutoProxyCreator or suchlike. The recommended usage pattern is to use either just the <aop:config>style, or just the AutoProxyCreator style. |

5.3.1. Declaring an aspect

Using the schema support, an aspect is simply a regular Java object defined as a bean in your Spring application context. The state and behavior is captured in the fields and methods of the object, and the pointcut and advice information is captured in the XML.

An aspect is declared using the <aop:aspect> element, and the backing bean is referenced using the ref attribute:

<aop:config>

<aop:aspect id="myAspect" ref="aBean">

...

</aop:aspect>

</aop:config>

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

...

</bean>

The bean backing the aspect (" `aBean`" in this case) can of course be configured and dependency injected just like any other Spring bean.

5.3.2. Declaring a pointcut

A named pointcut can be declared inside an <aop:config> element, enabling the pointcut definition to be shared across several aspects and advisors.

A pointcut representing the execution of any business service in the service layer could be defined as follows:

<aop:config>

<aop:pointcut id="businessService"

expression="execution(\* com.xyz.myapp.service.\*.\*(..))"/>

</aop:config>

Note that the pointcut expression itself is using the same AspectJ pointcut expression language as described in [@AspectJ support](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-ataspectj). If you are using the schema based declaration style, you can refer to named pointcuts defined in types (@Aspects) within the pointcut expression. Another way of defining the above pointcut would be:

<aop:config>

<aop:pointcut id="businessService"

expression="com.xyz.myapp.SystemArchitecture.businessService()"/>

</aop:config>

Assuming you have a SystemArchitecture aspect as described in [Sharing common pointcut definitions](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-common-pointcuts).

Declaring a pointcut inside an aspect is very similar to declaring a top-level pointcut:

<aop:config>

<aop:aspect id="myAspect" ref="aBean">

<aop:pointcut id="businessService"

expression="execution(\* com.xyz.myapp.service.\*.\*(..))"/>

...

</aop:aspect>

</aop:config>

Much the same way in an @AspectJ aspect, pointcuts declared using the schema based definition style may collect join point context. For example, the following pointcut collects the 'this' object as the join point context and passes it to advice:

<aop:config>

<aop:aspect id="myAspect" ref="aBean">

<aop:pointcut id="businessService"

expression="execution(\* com.xyz.myapp.service.\*.\*(..)) &amp;&amp; this(service)"/>

<aop:before pointcut-ref="businessService" method="monitor"/>

...

</aop:aspect>

</aop:config>

The advice must be declared to receive the collected join point context by including parameters of the matching names:

**public** **void** monitor(Object service) {

...

}

When combining pointcut sub-expressions, '&&' is awkward within an XML document, and so the keywords 'and', 'or' and 'not' can be used in place of '&&', '||' and '!' respectively. For example, the previous pointcut may be better written as:

<aop:config>

<aop:aspect id="myAspect" ref="aBean">

<aop:pointcut id="businessService"

expression="execution(\* com.xyz.myapp.service.\*.\*(..)) \*\*and\*\* this(service)"/>

<aop:before pointcut-ref="businessService" method="monitor"/>

...

</aop:aspect>

</aop:config>

Note that pointcuts defined in this way are referred to by their XML id and cannot be used as named pointcuts to form composite pointcuts. The named pointcut support in the schema based definition style is thus more limited than that offered by the @AspectJ style.

5.3.3. Declaring advice

The same five advice kinds are supported as for the @AspectJ style, and they have exactly the same semantics.

Before advice

Before advice runs before a matched method execution. It is declared inside an <aop:aspect> using the <aop:before> element.

<aop:aspect id="beforeExample" ref="aBean">

<aop:before

pointcut-ref="dataAccessOperation"

method="doAccessCheck"/>

...

</aop:aspect>

Here dataAccessOperation is the id of a pointcut defined at the top ( <aop:config>) level. To define the pointcut inline instead, replace the pointcut-ref attribute with a pointcut attribute:

<aop:aspect id="beforeExample" ref="aBean">

<aop:before

pointcut="execution(\* com.xyz.myapp.dao.\*.\*(..))"

method="doAccessCheck"/>

...

</aop:aspect>

As we noted in the discussion of the @AspectJ style, using named pointcuts can significantly improve the readability of your code.

The method attribute identifies a method ( doAccessCheck) that provides the body of the advice. This method must be defined for the bean referenced by the aspect element containing the advice. Before a data access operation is executed (a method execution join point matched by the pointcut expression), the "doAccessCheck" method on the aspect bean will be invoked.

After returning advice

After returning advice runs when a matched method execution completes normally. It is declared inside an <aop:aspect> in the same way as before advice. For example:

<aop:aspect id="afterReturningExample" ref="aBean">

<aop:after-returning

pointcut-ref="dataAccessOperation"

method="doAccessCheck"/>

...

</aop:aspect>

Just as in the @AspectJ style, it is possible to get hold of the return value within the advice body. Use the returning attribute to specify the name of the parameter to which the return value should be passed:

<aop:aspect id="afterReturningExample" ref="aBean">

<aop:after-returning

pointcut-ref="dataAccessOperation"

returning="retVal"

method="doAccessCheck"/>

...

</aop:aspect>

The doAccessCheck method must declare a parameter named retVal. The type of this parameter constrains matching in the same way as described for @AfterReturning. For example, the method signature may be declared as:

**public** **void** doAccessCheck(Object retVal) {...

After throwing advice

After throwing advice executes when a matched method execution exits by throwing an exception. It is declared inside an <aop:aspect> using the after-throwing element:

<aop:aspect id="afterThrowingExample" ref="aBean">

<aop:after-throwing

pointcut-ref="dataAccessOperation"

method="doRecoveryActions"/>

...

</aop:aspect>

Just as in the @AspectJ style, it is possible to get hold of the thrown exception within the advice body. Use the throwing attribute to specify the name of the parameter to which the exception should be passed:

<aop:aspect id="afterThrowingExample" ref="aBean">

<aop:after-throwing

pointcut-ref="dataAccessOperation"

throwing="dataAccessEx"

method="doRecoveryActions"/>

...

</aop:aspect>

The doRecoveryActions method must declare a parameter named dataAccessEx. The type of this parameter constrains matching in the same way as described for @AfterThrowing. For example, the method signature may be declared as:

**public** **void** doRecoveryActions(DataAccessException dataAccessEx) {...

After (finally) advice

After (finally) advice runs however a matched method execution exits. It is declared using the after element:

<aop:aspect id="afterFinallyExample" ref="aBean">

<aop:after

pointcut-ref="dataAccessOperation"

method="doReleaseLock"/>

...

</aop:aspect>

Around advice

The final kind of advice is around advice. Around advice runs "around" a matched method execution. It has the opportunity to do work both before and after the method executes, and to determine when, how, and even if, the method actually gets to execute at all. Around advice is often used if you need to share state before and after a method execution in a thread-safe manner (starting and stopping a timer for example). Always use the least powerful form of advice that meets your requirements; don’t use around advice if simple before advice would do.

Around advice is declared using the aop:around element. The first parameter of the advice method must be of type ProceedingJoinPoint. Within the body of the advice, calling proceed() on the ProceedingJoinPoint causes the underlying method to execute. The proceed method may also be calling passing in an Object[] - the values in the array will be used as the arguments to the method execution when it proceeds. See [Around advice](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-ataspectj-around-advice) for notes on calling proceed with an Object[].

<aop:aspect id="aroundExample" ref="aBean">

<aop:around

pointcut-ref="businessService"

method="doBasicProfiling"/>

...

</aop:aspect>

The implementation of the doBasicProfiling advice would be exactly the same as in the @AspectJ example (minus the annotation of course):

**public** Object doBasicProfiling(ProceedingJoinPoint pjp) **throws** Throwable {

*// start stopwatch*

Object retVal = pjp.proceed();

*// stop stopwatch*

**return** retVal;

}

Advice parameters

The schema based declaration style supports fully typed advice in the same way as described for the @AspectJ support - by matching pointcut parameters by name against advice method parameters. See [Advice parameters](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-ataspectj-advice-params) for details. If you wish to explicitly specify argument names for the advice methods (not relying on the detection strategies previously described) then this is done using the arg-names attribute of the advice element, which is treated in the same manner to the "argNames" attribute in an advice annotation as described in [Determining argument names](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-ataspectj-advice-params-names). For example:

<aop:before

pointcut="com.xyz.lib.Pointcuts.anyPublicMethod() and @annotation(auditable)"

method="audit"

arg-names="auditable"/>

The arg-names attribute accepts a comma-delimited list of parameter names.

Find below a slightly more involved example of the XSD-based approach that illustrates some around advice used in conjunction with a number of strongly typed parameters.

**package** x.y.service;

**public** **interface** **FooService** {

Foo getFoo(String fooName, **int** age);

}

**public** **class** **DefaultFooService** **implements** FooService {

**public** Foo getFoo(String name, **int** age) {

**return** **new** Foo(name, age);

}

}

Next up is the aspect. Notice the fact that the profile(..) method accepts a number of strongly-typed parameters, the first of which happens to be the join point used to proceed with the method call: the presence of this parameter is an indication that the profile(..) is to be used as around advice:

**package** x.y;

**import** org.aspectj.lang.ProceedingJoinPoint;

**import** org.springframework.util.StopWatch;

**public** **class** **SimpleProfiler** {

**public** Object profile(ProceedingJoinPoint call, String name, **int** age) **throws** Throwable {

StopWatch clock = **new** StopWatch("Profiling for '" + name + "' and '" + age + "'");

**try** {

clock.start(call.toShortString());

**return** call.proceed();

} **finally** {

clock.stop();

System.out.println(clock.prettyPrint());

}

}

}

Finally, here is the XML configuration that is required to effect the execution of the above advice for a particular join point:

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

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

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

xsi:schemaLocation="

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

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

*<!-- this is the object that will be proxied by Spring's AOP infrastructure -->*

<bean id="fooService" class="x.y.service.DefaultFooService"/>

*<!-- this is the actual advice itself -->*

<bean id="profiler" class="x.y.SimpleProfiler"/>

<aop:config>

<aop:aspect ref="profiler">

<aop:pointcut id="theExecutionOfSomeFooServiceMethod"

expression="execution(\* x.y.service.FooService.getFoo(String,int))

and args(name, age)"/>

<aop:around pointcut-ref="theExecutionOfSomeFooServiceMethod"

method="profile"/>

</aop:aspect>

</aop:config>

</beans>

If we had the following driver script, we would get output something like this on standard output:

**import** org.springframework.beans.factory.BeanFactory;

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

**import** x.y.service.FooService;

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

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

BeanFactory ctx = **new** ClassPathXmlApplicationContext("x/y/plain.xml");

FooService foo = (FooService) ctx.getBean("fooService");

foo.getFoo("Pengo", 12);

}

}

StopWatch 'Profiling for 'Pengo' and '12'': running time (millis) = 0

-----------------------------------------

ms % Task name

-----------------------------------------

00000 ? execution(getFoo)

Advice ordering

When multiple advice needs to execute at the same join point (executing method) the ordering rules are as described in [Advice ordering](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-ataspectj-advice-ordering). The precedence between aspects is determined by either adding the Order annotation to the bean backing the aspect or by having the bean implement the Ordered interface.

5.3.4. Introductions

Introductions (known as inter-type declarations in AspectJ) enable an aspect to declare that advised objects implement a given interface, and to provide an implementation of that interface on behalf of those objects.

An introduction is made using the aop:declare-parents element inside an aop:aspect This element is used to declare that matching types have a new parent (hence the name). For example, given an interface UsageTracked, and an implementation of that interface DefaultUsageTracked, the following aspect declares that all implementors of service interfaces also implement the UsageTracked interface. (In order to expose statistics via JMX for example.)

<aop:aspect id="usageTrackerAspect" ref="usageTracking">

<aop:declare-parents

types-matching="com.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>

The class backing the usageTracking bean would contain the method:

**public** **void** recordUsage(UsageTracked usageTracked) {

usageTracked.incrementUseCount();

}

The interface to be implemented is determined by implement-interface attribute. The value of the types-matching attribute is an AspectJ type pattern :- any bean of a matching type will implement the UsageTracked interface. Note that in the before advice of the above example, service beans can be directly used as implementations of the UsageTracked interface. If accessing a bean programmatically you would write the following:

UsageTracked usageTracked = (UsageTracked) context.getBean("myService");

5.3.5. Aspect instantiation models

The only supported instantiation model for schema-defined aspects is the singleton model. Other instantiation models may be supported in future releases.

5.3.6. Advisors

The concept of "advisors" is brought forward from the AOP support defined in Spring and does not have a direct equivalent in AspectJ. An advisor is like a small self-contained aspect that has a single piece of advice. The advice itself is represented by a bean, and must implement one of the advice interfaces described in [Advice types in Spring](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-api-advice-types). Advisors can take advantage of AspectJ pointcut expressions though.

Spring supports the advisor concept with the <aop:advisor> element. You will most commonly see it used in conjunction with transactional advice, which also has its own namespace support in Spring. Here’s how it looks:

<aop:config>

<aop:pointcut id="businessService"

expression="execution(\* com.xyz.myapp.service.\*.\*(..))"/>

<aop:advisor

pointcut-ref="businessService"

advice-ref="tx-advice"/>

</aop:config>

<tx:advice id="tx-advice">

<tx:attributes>

<tx:method name="\*" propagation="REQUIRED"/>

</tx:attributes>

</tx:advice>

As well as the pointcut-ref attribute used in the above example, you can also use the pointcut attribute to define a pointcut expression inline.

To define the precedence of an advisor so that the advice can participate in ordering, use the order attribute to define the Ordered value of the advisor.

5.3.7. Example

Let’s see how the concurrent locking failure retry example from [Example](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-ataspectj-example) looks when rewritten using the schema support.

The execution of business services can sometimes fail due to concurrency issues (for example, deadlock loser). If the operation is retried, it is quite likely it will succeed next time round. For business services where it is appropriate to retry in such conditions (idempotent operations that don’t need to go back to the user for conflict resolution), we’d like to transparently retry the operation to avoid the client seeing a PessimisticLockingFailureException. This is a requirement that clearly cuts across multiple services in the service layer, and hence is ideal for implementing via an aspect.

Because we want to retry the operation, we’ll need to use around advice so that we can call proceed multiple times. Here’s how the basic aspect implementation looks (it’s just a regular Java class using the schema support):

**public** **class** **ConcurrentOperationExecutor** **implements** Ordered {

**private** **static** **final** **int** DEFAULT\_MAX\_RETRIES = 2;

**private** **int** maxRetries = DEFAULT\_MAX\_RETRIES;

**private** **int** order = 1;

**public** **void** setMaxRetries(**int** maxRetries) {

this.maxRetries = maxRetries;

}

**public** **int** getOrder() {

**return** this.order;

}

**public** **void** setOrder(**int** order) {

this.order = order;

}

**public** Object doConcurrentOperation(ProceedingJoinPoint pjp) **throws** Throwable {

**int** numAttempts = 0;

PessimisticLockingFailureException lockFailureException;

**do** {

numAttempts++;

**try** {

**return** pjp.proceed();

}

**catch**(PessimisticLockingFailureException ex) {

lockFailureException = ex;

}

} **while**(numAttempts <= this.maxRetries);

**throw** lockFailureException;

}

}

Note that the aspect implements the Ordered interface so we can set the precedence of the aspect higher than the transaction advice (we want a fresh transaction each time we retry). The maxRetries and order properties will both be configured by Spring. The main action happens in the doConcurrentOperation around advice method. We try to proceed, and if we fail with a PessimisticLockingFailureException we simply try again unless we have exhausted all of our retry attempts.

|  |  |
| --- | --- |
|  | This class is identical to the one used in the @AspectJ example, but with the annotations removed. |

The corresponding Spring configuration is:

<aop:config>

<aop:aspect id="concurrentOperationRetry" ref="concurrentOperationExecutor">

<aop:pointcut id="idempotentOperation"

expression="execution(\* com.xyz.myapp.service.\*.\*(..))"/>

<aop:around

pointcut-ref="idempotentOperation"

method="doConcurrentOperation"/>

</aop:aspect>

</aop:config>

<bean id="concurrentOperationExecutor"

class="com.xyz.myapp.service.impl.ConcurrentOperationExecutor">

<property name="maxRetries" value="3"/>

<property name="order" value="100"/>

</bean>

Notice that for the time being we assume that all business services are idempotent. If this is not the case we can refine the aspect so that it only retries genuinely idempotent operations, by introducing an Idempotent annotation:

@Retention(RetentionPolicy.RUNTIME)

**public** @interface Idempotent {

*// marker annotation*

}

and using the annotation to annotate the implementation of service operations. The change to the aspect to retry only idempotent operations simply involves refining the pointcut expression so that only @Idempotent operations match:

<aop:pointcut id="idempotentOperation"

expression="execution(\* com.xyz.myapp.service.\*.\*(..)) and

@annotation(com.xyz.myapp.service.Idempotent)"/>

5.4. Choosing which AOP declaration style to use

Once you have decided that an aspect is the best approach for implementing a given requirement, how do you decide between using Spring AOP or AspectJ, and between the Aspect language (code) style, @AspectJ annotation style, or the Spring XML style? These decisions are influenced by a number of factors including application requirements, development tools, and team familiarity with AOP.

5.4.1. Spring AOP or full AspectJ?

Use the simplest thing that can work. Spring AOP is simpler than using full AspectJ as there is no requirement to introduce the AspectJ compiler / weaver into your development and build processes. If you only need to advise the execution of operations on Spring beans, then Spring AOP is the right choice. If you need to advise objects not managed by the Spring container (such as domain objects typically), then you will need to use AspectJ. You will also need to use AspectJ if you wish to advise join points other than simple method executions (for example, field get or set join points, and so on).

When using AspectJ, you have the choice of the AspectJ language syntax (also known as the "code style") or the @AspectJ annotation style. Clearly, if you are not using Java 5+ then the choice has been made for you…​ use the code style. If aspects play a large role in your design, and you are able to use the [AspectJ Development Tools (AJDT)](https://www.eclipse.org/ajdt/) plugin for Eclipse, then the AspectJ language syntax is the preferred option: it is cleaner and simpler because the language was purposefully designed for writing aspects. If you are not using Eclipse, or have only a few aspects that do not play a major role in your application, then you may want to consider using the @AspectJ style and sticking with a regular Java compilation in your IDE, and adding an aspect weaving phase to your build script.

5.4.2. @AspectJ or XML for Spring AOP?

If you have chosen to use Spring AOP, then you have a choice of @AspectJ or XML style. There are various tradeoffs to consider.

The XML style will be most familiar to existing Spring users and it is backed by genuine POJOs. When using AOP as a tool to configure enterprise services then XML can be a good choice (a good test is whether you consider the pointcut expression to be a part of your configuration you might want to change independently). With the XML style arguably it is clearer from your configuration what aspects are present in the system.

The XML style has two disadvantages. Firstly it does not fully encapsulate the implementation of the requirement it addresses in a single place. The DRY principle says that there should be a single, unambiguous, authoritative representation of any piece of knowledge within a system. When using the XML style, the knowledge of *how* a requirement is implemented is split across the declaration of the backing bean class, and the XML in the configuration file. When using the @AspectJ style there is a single module - the aspect - in which this information is encapsulated. Secondly, the XML style is slightly more limited in what it can express than the @AspectJ style: only the "singleton" aspect instantiation model is supported, and it is not possible to combine named pointcuts declared in XML. For example, in the @AspectJ style you can write something like:

@Pointcut(execution(\* get\*()))

**public** **void** propertyAccess() {}

@Pointcut(execution(org.xyz.Account+ \*(..))

**public** **void** operationReturningAnAccount() {}

@Pointcut(propertyAccess() && operationReturningAnAccount())

**public** **void** accountPropertyAccess() {}

In the XML style I can declare the first two pointcuts:

<aop:pointcut id="propertyAccess"

expression="execution(\* get\*())"/>

<aop:pointcut id="operationReturningAnAccount"

expression="execution(org.xyz.Account+ \*(..))"/>

The downside of the XML approach is that you cannot define the accountPropertyAccess pointcut by combining these definitions.

The @AspectJ style supports additional instantiation models, and richer pointcut composition. It has the advantage of keeping the aspect as a modular unit. It also has the advantage the @AspectJ aspects can be understood (and thus consumed) both by Spring AOP and by AspectJ - so if you later decide you need the capabilities of AspectJ to implement additional requirements then it is very easy to migrate to an AspectJ-based approach. On balance the Spring team prefer the @AspectJ style whenever you have aspects that do more than simple "configuration" of enterprise services.

5.5. Mixing aspect types

It is perfectly possible to mix @AspectJ style aspects using the autoproxying support, schema-defined <aop:aspect> aspects, <aop:advisor> declared advisors and even proxies and interceptors defined using the Spring 1.2 style in the same configuration. All of these are implemented using the same underlying support mechanism and will co-exist without any difficulty.

5.6. Proxying mechanisms

Spring AOP uses either JDK dynamic proxies or CGLIB to create the proxy for a given target object. (JDK dynamic proxies are preferred whenever you have a choice).

If the target object to be proxied implements at least one interface then a JDK dynamic proxy will be used. All of the interfaces implemented by the target type will be proxied. If the target object does not implement any interfaces then a CGLIB proxy will be created.

If you want to force the use of CGLIB proxying (for example, to proxy every method defined for the target object, not just those implemented by its interfaces) you can do so. However, there are some issues to consider:

* final methods cannot be advised, as they cannot be overridden.
* As of Spring 3.2, it is no longer necessary to add CGLIB to your project classpath, as CGLIB classes are repackaged under org.springframework and included directly in the spring-core JAR. This means that CGLIB-based proxy support 'just works' in the same way that JDK dynamic proxies always have.
* As of Spring 4.0, the constructor of your proxied object will NOT be called twice anymore since the CGLIB proxy instance will be created via Objenesis. Only if your JVM does not allow for constructor bypassing, you might see double invocations and corresponding debug log entries from Spring’s AOP support.

To force the use of CGLIB proxies set the value of the proxy-target-class attribute of the <aop:config> element to true:

<aop:config proxy-target-class="true">

*<!-- other beans defined here... -->*

</aop:config>

To force CGLIB proxying when using the @AspectJ autoproxy support, set the 'proxy-target-class' attribute of the <aop:aspectj-autoproxy> element to true:

<aop:aspectj-autoproxy proxy-target-class="true"/>

|  |  |
| --- | --- |
|  | Multiple <aop:config/> sections are collapsed into a single unified auto-proxy creator at runtime, which applies the *strongest* proxy settings that any of the <aop:config/> sections (typically from different XML bean definition files) specified. This also applies to the <tx:annotation-driven/> and <aop:aspectj-autoproxy/>elements.  To be clear: using proxy-target-class="true" on <tx:annotation-driven/>, <aop:aspectj-autoproxy/> or <aop:config/> elements will force the use of CGLIB proxies *for all three of them*. |

5.6.1. Understanding AOP proxies

Spring AOP is *proxy-based*. It is vitally important that you grasp the semantics of what that last statement actually means before you write your own aspects or use any of the Spring AOP-based aspects supplied with the Spring Framework.

Consider first the scenario where you have a plain-vanilla, un-proxied, nothing-special-about-it, straight object reference, as illustrated by the following code snippet.

**public** **class** **SimplePojo** **implements** Pojo {

**public** **void** foo() {

*// this next method invocation is a direct call on the 'this' reference*

this.bar();

}

**public** **void** bar() {

*// some logic...*

}

}

If you invoke a method on an object reference, the method is invoked *directly* on that object reference, as can be seen below.



**public** **class** **Main** {

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

Pojo pojo = **new** SimplePojo();

*// this is a direct method call on the 'pojo' reference*

pojo.foo();

}

}

Things change slightly when the reference that client code has is a proxy. Consider the following diagram and code snippet.



**public** **class** **Main** {

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

ProxyFactory factory = **new** ProxyFactory(**new** SimplePojo());

factory.addInterface(Pojo.class);

factory.addAdvice(**new** RetryAdvice());

Pojo pojo = (Pojo) factory.getProxy();

*// this is a method call on the proxy!*

pojo.foo();

}

}

The key thing to understand here is that the client code inside the main(..) of the Main class *has a reference to the proxy*. This means that method calls on that object reference will be calls on the proxy, and as such the proxy will be able to delegate to all of the interceptors (advice) that are relevant to that particular method call. However, once the call has finally reached the target object, the SimplePojo reference in this case, any method calls that it may make on itself, such as this.bar() or this.foo(), are going to be invoked against the *this* reference, and *not* the proxy. This has important implications. It means that self-invocation is *not* going to result in the advice associated with a method invocation getting a chance to execute.

Okay, so what is to be done about this? The best approach (the term best is used loosely here) is to refactor your code such that the self-invocation does not happen. For sure, this does entail some work on your part, but it is the best, least-invasive approach. The next approach is absolutely horrendous, and I am almost reticent to point it out precisely because it is so horrendous. You can (choke!) totally tie the logic within your class to Spring AOP by doing this:

**public** **class** **SimplePojo** **implements** Pojo {

**public** **void** foo() {

*// this works, but... gah!*

((Pojo) AopContext.currentProxy()).bar();

}

**public** **void** bar() {

*// some logic...*

}

}

This totally couples your code to Spring AOP, *and* it makes the class itself aware of the fact that it is being used in an AOP context, which flies in the face of AOP. It also requires some additional configuration when the proxy is being created:

**public** **class** **Main** {

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

ProxyFactory factory = **new** ProxyFactory(**new** SimplePojo());

factory.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();

}

}

Finally, it must be noted that AspectJ does not have this self-invocation issue because it is not a proxy-based AOP framework.

5.7. Programmatic creation of @AspectJ Proxies

In addition to declaring aspects in your configuration using either <aop:config> or <aop:aspectj-autoproxy>, it is also possible programmatically to create proxies that advise target objects. For the full details of Spring’s AOP API, see the next chapter. Here we want to focus on the ability to automatically create proxies using @AspectJ aspects.

The class org.springframework.aop.aspectj.annotation.AspectJProxyFactory can be used to create a proxy for a target object that is advised by one or more @AspectJ aspects. Basic usage for this class is very simple, as illustrated below. See the javadocs for full information.

*// create a factory that can generate a proxy for the given target object*

AspectJProxyFactory factory = **new** AspectJProxyFactory(targetObject);

*// add an aspect, the class must be an @AspectJ aspect*

*// you can call this as many times as you need with different aspects*

factory.addAspect(SecurityManager.class);

*// you can also add existing aspect instances, the type of the object supplied must be an @AspectJ aspect*

factory.addAspect(usageTracker);

*// now get the proxy object...*

MyInterfaceType proxy = factory.getProxy();

5.8. Using AspectJ with Spring applications

Everything we’ve covered so far in this chapter is pure Spring AOP. In this section, we’re going to look at how you can use the AspectJ compiler/weaver instead of, or in addition to, Spring AOP if your needs go beyond the facilities offered by Spring AOP alone.

Spring ships with a small AspectJ aspect library, which is available standalone in your distribution as spring-aspects.jar; you’ll need to add this to your classpath in order to use the aspects in it. [Using AspectJ to dependency inject domain objects with Spring](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-atconfigurable) and [Other Spring aspects for AspectJ](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-ajlib-other) discuss the content of this library and how you can use it. [Configuring AspectJ aspects using Spring IoC](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-aj-configure) discusses how to dependency inject AspectJ aspects that are woven using the AspectJ compiler. Finally, [Load-time weaving with AspectJ in the Spring Framework](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-aj-ltw) provides an introduction to load-time weaving for Spring applications using AspectJ.

5.8.1. Using AspectJ to dependency inject domain objects with Spring

The Spring container instantiates and configures beans defined in your application context. It is also possible to ask a bean factory to configure a *pre-existing* object given the name of a bean definition containing the configuration to be applied. The spring-aspects.jar contains an annotation-driven aspect that exploits this capability to allow dependency injection of *any object*. The support is intended to be used for objects created *outside of the control of any container*. Domain objects often fall into this category because they are often created programmatically using the new operator, or by an ORM tool as a result of a database query.

The @Configurable annotation marks a class as eligible for Spring-driven configuration. In the simplest case it can be used just as a marker annotation:

**package** com.xyz.myapp.domain;

**import** org.springframework.beans.factory.annotation.Configurable;

@Configurable

**public** **class** **Account** {

*// ...*

}

When used as a marker interface in this way, Spring will configure new instances of the annotated type ( Account in this case) using a bean definition (typically prototype-scoped) with the same name as the fully-qualified type name (com.xyz.myapp.domain.Account). Since the default name for a bean is the fully-qualified name of its type, a convenient way to declare the prototype definition is simply to omit the id attribute:

<bean class="com.xyz.myapp.domain.Account" scope="prototype">

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

</bean>

If you want to explicitly specify the name of the prototype bean definition to use, you can do so directly in the annotation:

**package** com.xyz.myapp.domain;

**import** org.springframework.beans.factory.annotation.Configurable;

@Configurable("account")

**public** **class** **Account** {

*// ...*

}

Spring will now look for a bean definition named "account" and use that as the definition to configure new Account instances.

You can also use autowiring to avoid having to specify a dedicated bean definition at all. To have Spring apply autowiring use the autowire property of the @Configurable annotation: specify either @Configurable(autowire=Autowire.BY\_TYPE) or@Configurable(autowire=Autowire.BY\_NAME for autowiring by type or by name respectively. As an alternative, as of Spring 2.5 it is preferable to specify explicit, annotation-driven dependency injection for your @Configurable beans by using @Autowired or @Inject at the field or method level (see [Annotation-based container configuration](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-annotation-config) for further details).

Finally you can enable Spring dependency checking for the object references in the newly created and configured object by using the dependencyCheck attribute (for example: @Configurable(autowire=Autowire.BY\_NAME,dependencyCheck=true)). If this attribute is set to true, then Spring will validate after configuration that all properties (*which are not primitives or collections*) have been set.

Using the annotation on its own does nothing of course. It is the AnnotationBeanConfigurerAspect in spring-aspects.jar that acts on the presence of the annotation. In essence the aspect says "after returning from the initialization of a new object of a type annotated with @Configurable, configure the newly created object using Spring in accordance with the properties of the annotation". In this context, *initialization* refers to newly instantiated objects (e.g., objects instantiated with the new operator) as well as to Serializable objects that are undergoing deserialization (e.g., via [readResolve()](http://docs.oracle.com/javase/6/docs/api/java/io/Serializable.html)).

|  |  |
| --- | --- |
|  | One of the key phrases in the above paragraph is '*in essence*'. For most cases, the exact semantics of '*after returning from the initialization of a new object*' will be fine…​ in this context, '*after initialization*' means that the dependencies will be injected *after* the object has been constructed - this means that the dependencies will not be available for use in the constructor bodies of the class. If you want the dependencies to be injected *before*the constructor bodies execute, and thus be available for use in the body of the constructors, then you need to define this on the @Configurable declaration like so:  @Configurable(preConstruction=true)  You can find out more information about the language semantics of the various pointcut types in AspectJ [in this appendix](https://www.eclipse.org/aspectj/doc/next/progguide/semantics-joinPoints.html) of the [AspectJ Programming Guide](https://www.eclipse.org/aspectj/doc/next/progguide/index.html). |

For this to work the annotated types must be woven with the AspectJ weaver - you can either use a build-time Ant or Maven task to do this (see for example the [AspectJ Development Environment Guide](https://www.eclipse.org/aspectj/doc/released/devguide/antTasks.html)) or load-time weaving (see [Load-time weaving with AspectJ in the Spring Framework](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-aj-ltw)). The AnnotationBeanConfigurerAspect itself needs configuring by Spring (in order to obtain a reference to the bean factory that is to be used to configure new objects). If you are using Java based configuration simply add @EnableSpringConfigured to any @Configuration class.

@Configuration

@EnableSpringConfigured

**public** **class** **AppConfig** {

}

If you prefer XML based configuration, the Spring [context namespace](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#xsd-schemas-context) defines a convenient context:spring-configuredelement:

<context:spring-configured/>

Instances of @Configurable objects created *before* the aspect has been configured will result in a message being issued to the debug log and no configuration of the object taking place. An example might be a bean in the Spring configuration that creates domain objects when it is initialized by Spring. In this case you can use the "depends-on" bean attribute to manually specify that the bean depends on the configuration aspect.

<bean id="myService"

class="com.xzy.myapp.service.MyService"

depends-on="org.springframework.beans.factory.aspectj.AnnotationBeanConfigurerAspect">

*<!-- ... -->*

</bean>

|  |  |
| --- | --- |
|  | Do not activate @Configurable processing through the bean configurer aspect unless you really mean to rely on its semantics at runtime. In particular, make sure that you do not use @Configurable on bean classes which are registered as regular Spring beans with the container: You would get double initialization otherwise, once through the container and once through the aspect. |

Unit testing @Configurable objects

One of the goals of the @Configurable support is to enable independent unit testing of domain objects without the difficulties associated with hard-coded lookups. If @Configurable types have not been woven by AspectJ then the annotation has no affect during unit testing, and you can simply set mock or stub property references in the object under test and proceed as normal. If @Configurable types *have* been woven by AspectJ then you can still unit test outside of the container as normal, but you will see a warning message each time that you construct an @Configurable object indicating that it has not been configured by Spring.

Working with multiple application contexts

The AnnotationBeanConfigurerAspect used to implement the @Configurable support is an AspectJ singleton aspect. The scope of a singleton aspect is the same as the scope of static members, that is to say there is one aspect instance per classloader that defines the type. This means that if you define multiple application contexts within the same classloader hierarchy you need to consider where to define the @EnableSpringConfigured bean and where to place spring-aspects.jar on the classpath.

Consider a typical Spring web-app configuration with a shared parent application context defining common business services and everything needed to support them, and one child application context per servlet containing definitions particular to that servlet. All of these contexts will co-exist within the same classloader hierarchy, and so the AnnotationBeanConfigurerAspectcan only hold a reference to one of them. In this case we recommend defining the @EnableSpringConfigured bean in the shared (parent) application context: this defines the services that you are likely to want to inject into domain objects. A consequence is that you cannot configure domain objects with references to beans defined in the child (servlet-specific) contexts using the @Configurable mechanism (probably not something you want to do anyway!).

When deploying multiple web-apps within the same container, ensure that each web-application loads the types in spring-aspects.jar using its own classloader (for example, by placing spring-aspects.jar in 'WEB-INF/lib'). If spring-aspects.jar is only added to the container wide classpath (and hence loaded by the shared parent classloader), all web applications will share the same aspect instance which is probably not what you want.

5.8.2. Other Spring aspects for AspectJ

In addition to the @Configurable aspect, spring-aspects.jar contains an AspectJ aspect that can be used to drive Spring’s transaction management for types and methods annotated with the @Transactional annotation. This is primarily intended for users who want to use the Spring Framework’s transaction support outside of the Spring container.

The aspect that interprets @Transactional annotations is the AnnotationTransactionAspect. When using this aspect, you must annotate the *implementation* class (and/or methods within that class), *not* the interface (if any) that the class implements. AspectJ follows Java’s rule that annotations on interfaces are *not inherited*.

A @Transactional annotation on a class specifies the default transaction semantics for the execution of any *public* operation in the class.

A @Transactional annotation on a method within the class overrides the default transaction semantics given by the class annotation (if present). Methods of any visibility may be annotated, including private methods. Annotating non-public methods directly is the only way to get transaction demarcation for the execution of such methods.

|  |  |
| --- | --- |
|  | Since Spring Framework 4.2, spring-aspects provides a similar aspect that offers the exact same features for the standard javax.transaction.Transactional annotation. Check JtaAnnotationTransactionAspect for more details. |

For AspectJ programmers that want to use the Spring configuration and transaction management support but don’t want to (or cannot) use annotations, spring-aspects.jar also contains abstract aspects you can extend to provide your own pointcut definitions. See the sources for the AbstractBeanConfigurerAspect and AbstractTransactionAspect aspects for more information. As an example, the following excerpt shows how you could write an aspect to configure all instances of objects defined in the domain model using prototype bean definitions that match the fully-qualified class names:

**public** aspect DomainObjectConfiguration **extends** AbstractBeanConfigurerAspect {

**public** DomainObjectConfiguration() {

setBeanWiringInfoResolver(**new** ClassNameBeanWiringInfoResolver());

}

*// the creation of a new bean (any object in the domain model)*

**protected** pointcut beanCreation(Object beanInstance) :

initialization(**new**(..)) &&

SystemArchitecture.inDomainModel() &&

this(beanInstance);

}

5.8.3. Configuring AspectJ aspects using Spring IoC

When using AspectJ aspects with Spring applications, it is natural to both want and expect to be able to configure such aspects using Spring. The AspectJ runtime itself is responsible for aspect creation, and the means of configuring the AspectJ created aspects via Spring depends on the AspectJ instantiation model (the per-xxx clause) used by the aspect.

The majority of AspectJ aspects are *singleton* aspects. Configuration of these aspects is very easy: simply create a bean definition referencing the aspect type as normal, and include the bean attribute 'factory-method="aspectOf"'. This ensures that Spring obtains the aspect instance by asking AspectJ for it rather than trying to create an instance itself. For example:

<bean id="profiler" class="com.xyz.profiler.Profiler"

factory-method="aspectOf">

<property name="profilingStrategy" ref="jamonProfilingStrategy"/>

</bean>

Non-singleton aspects are harder to configure: however it is possible to do so by creating prototype bean definitions and using the @Configurable support from spring-aspects.jar to configure the aspect instances once they have bean created by the AspectJ runtime.

If you have some @AspectJ aspects that you want to weave with AspectJ (for example, using load-time weaving for domain model types) and other @AspectJ aspects that you want to use with Spring AOP, and these aspects are all configured using Spring, then you will need to tell the Spring AOP @AspectJ autoproxying support which exact subset of the @AspectJ aspects defined in the configuration should be used for autoproxying. You can do this by using one or more <include/> elements inside the <aop:aspectj-autoproxy/> declaration. Each <include/> element specifies a name pattern, and only beans with names matched by at least one of the patterns will be used for Spring AOP autoproxy configuration:

<aop:aspectj-autoproxy>

<aop:include name="thisBean"/>

<aop:include name="thatBean"/>

</aop:aspectj-autoproxy>

|  |  |
| --- | --- |
|  | Do not be misled by the name of the <aop:aspectj-autoproxy/> element: using it will result in the creation of *Spring AOP proxies*. The @AspectJ style of aspect declaration is just being used here, but the AspectJ runtime is *not* involved. |

5.8.4. Load-time weaving with AspectJ in the Spring Framework

Load-time weaving (LTW) refers to the process of weaving AspectJ aspects into an application’s class files as they are being loaded into the Java virtual machine (JVM). The focus of this section is on configuring and using LTW in the specific context of the Spring Framework: this section is not an introduction to LTW though. For full details on the specifics of LTW and configuring LTW with just AspectJ (with Spring not being involved at all), see the [LTW section of the AspectJ Development Environment Guide](https://www.eclipse.org/aspectj/doc/released/devguide/ltw.html).

The value-add that the Spring Framework brings to AspectJ LTW is in enabling much finer-grained control over the weaving process. 'Vanilla' AspectJ LTW is effected using a Java (5+) agent, which is switched on by specifying a VM argument when starting up a JVM. It is thus a JVM-wide setting, which may be fine in some situations, but often is a little too coarse. Spring-enabled LTW enables you to switch on LTW on a *per-ClassLoader* basis, which obviously is more fine-grained and which can make more sense in a 'single-JVM-multiple-application' environment (such as is found in a typical application server environment).

Further, [in certain environments](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-aj-ltw-environments), this support enables load-time weaving *without making any modifications to the application server’s launch script* that will be needed to add -javaagent:path/to/aspectjweaver.jar or (as we describe later in this section)-javaagent:path/to/org.springframework.instrument-{version}.jar (previously named spring-agent.jar). Developers simply modify one or more files that form the application context to enable load-time weaving instead of relying on administrators who typically are in charge of the deployment configuration such as the launch script.

Now that the sales pitch is over, let us first walk through a quick example of AspectJ LTW using Spring, followed by detailed specifics about elements introduced in the following example. For a complete example, please see the [Petclinic sample application](https://github.com/spring-projects/spring-petclinic).

A first example

Let us assume that you are an application developer who has been tasked with diagnosing the cause of some performance problems in a system. Rather than break out a profiling tool, what we are going to do is switch on a simple profiling aspect that will enable us to very quickly get some performance metrics, so that we can then apply a finer-grained profiling tool to that specific area immediately afterwards.

|  |  |
| --- | --- |
|  | The example presented here uses XML style configuration, it is also possible to configure and use @AspectJ with [Java Configuration](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-java). Specifically the @EnableLoadTimeWeaving annotation can be used as an alternative to<context:load-time-weaver/> (see [below](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-aj-ltw-spring) for details). |

Here is the profiling aspect. Nothing too fancy, just a quick-and-dirty time-based profiler, using the @AspectJ-style of aspect declaration.

**package** foo;

**import** org.aspectj.lang.ProceedingJoinPoint;

**import** org.aspectj.lang.annotation.Aspect;

**import** org.aspectj.lang.annotation.Around;

**import** org.aspectj.lang.annotation.Pointcut;

**import** org.springframework.util.StopWatch;

**import** org.springframework.core.annotation.Order;

@Aspect

**public** **class** **ProfilingAspect** {

@Around("methodsToBeProfiled()")

**public** Object profile(ProceedingJoinPoint pjp) **throws** Throwable {

StopWatch sw = **new** StopWatch(getClass().getSimpleName());

**try** {

sw.start(pjp.getSignature().getName());

**return** pjp.proceed();

} **finally** {

sw.stop();

System.out.println(sw.prettyPrint());

}

}

@Pointcut("execution(public \* foo..\*.\*(..))")

**public** **void** methodsToBeProfiled(){}

}

We will also need to create an META-INF/aop.xml file, to inform the AspectJ weaver that we want to weave our ProfilingAspectinto our classes. This file convention, namely the presence of a file (or files) on the Java classpath called META-INF/aop.xml is standard AspectJ.

<!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN" "http://www.eclipse.org/aspectj/dtd/aspectj.dtd">

<aspectj>

<weaver>

*<!-- only weave classes in our application-specific packages -->*

<include within="foo.\*"/>

</weaver>

<aspects>

*<!-- weave in just this aspect -->*

<aspect name="foo.ProfilingAspect"/>

</aspects>

</aspectj>

Now to the Spring-specific portion of the configuration. We need to configure a LoadTimeWeaver (all explained later, just take it on trust for now). This load-time weaver is the essential component responsible for weaving the aspect configuration in one or more META-INF/aop.xml files into the classes in your application. The good thing is that it does not require a lot of configuration, as can be seen below (there are some more options that you can specify, but these are detailed later).

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

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

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

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

xsi:schemaLocation="

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

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

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

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

*<!-- a service object; we will be profiling its methods -->*

<bean id="entitlementCalculationService"

class="foo.StubEntitlementCalculationService"/>

*<!-- this switches on the load-time weaving -->*

**<context:load-time-weaver/>**

</beans>

Now that all the required artifacts are in place - the aspect, the META-INF/aop.xml file, and the Spring configuration -, let us create a simple driver class with a main(..) method to demonstrate the LTW in action.

**package** foo;

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

**public** **final** **class** **Main** {

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

ApplicationContext ctx = **new** ClassPathXmlApplicationContext("beans.xml", Main.class);

EntitlementCalculationService entitlementCalculationService

= (EntitlementCalculationService) ctx.getBean("entitlementCalculationService");

*// the profiling aspect is 'woven' around this method execution*

entitlementCalculationService.calculateEntitlement();

}

}

There is one last thing to do. The introduction to this section did say that one could switch on LTW selectively on a per- ClassLoader basis with Spring, and this is true. However, just for this example, we are going to use a Java agent (supplied with Spring) to switch on the LTW. This is the command line we will use to run the above Main class:

java -javaagent:C:/projects/foo/lib/global/spring-instrument.jar foo.Main

The -javaagent is a flag for specifying and enabling [agents to instrument programs running on the JVM](http://docs.oracle.com/javase/6/docs/api/java/lang/instrument/package-summary.html). The Spring Framework ships with such an agent, the InstrumentationSavingAgent, which is packaged in the spring-instrument.jar that was supplied as the value of the -javaagent argument in the above example.

The output from the execution of the Main program will look something like that below. (I have introduced a Thread.sleep(..)statement into the calculateEntitlement() implementation so that the profiler actually captures something other than 0 milliseconds - the 01234 milliseconds is *not* an overhead introduced by the AOP :) )

Calculating entitlement

StopWatch 'ProfilingAspect': running time (millis) = 1234

------ ----- ----------------------------

ms % Task name

------ ----- ----------------------------

01234 100% calculateEntitlement

Since this LTW is effected using full-blown AspectJ, we are not just limited to advising Spring beans; the following slight variation on the Main program will yield the same result.

**package** foo;

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

**public** **final** **class** **Main** {

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

**new** ClassPathXmlApplicationContext("beans.xml", Main.class);

EntitlementCalculationService entitlementCalculationService =

**new** StubEntitlementCalculationService();

*// the profiling aspect will be 'woven' around this method execution*

entitlementCalculationService.calculateEntitlement();

}

}

Notice how in the above program we are simply bootstrapping the Spring container, and then creating a new instance of the StubEntitlementCalculationService totally outside the context of Spring…​ the profiling advice still gets woven in.

The example admittedly is simplistic…​ however the basics of the LTW support in Spring have all been introduced in the above example, and the rest of this section will explain the 'why' behind each bit of configuration and usage in detail.

|  |  |
| --- | --- |
|  | The ProfilingAspect used in this example may be basic, but it is quite useful. It is a nice example of a development-time aspect that developers can use during development (of course), and then quite easily exclude from builds of the application being deployed into UAT or production. |

Aspects

The aspects that you use in LTW have to be AspectJ aspects. They can be written in either the AspectJ language itself or you can write your aspects in the @AspectJ-style. It means that your aspects are then both valid AspectJ *and* Spring AOP aspects. Furthermore, the compiled aspect classes need to be available on the classpath.

'META-INF/aop.xml'

The AspectJ LTW infrastructure is configured using one or more META-INF/aop.xml files, that are on the Java classpath (either directly, or more typically in jar files).

The structure and contents of this file is detailed in the main AspectJ reference documentation, and the interested reader is[referred to that resource](https://www.eclipse.org/aspectj/doc/released/devguide/ltw-configuration.html). (I appreciate that this section is brief, but the aop.xml file is 100% AspectJ - there is no Spring-specific information or semantics that apply to it, and so there is no extra value that I can contribute either as a result), so rather than rehash the quite satisfactory section that the AspectJ developers wrote, I am just directing you there.)

Required libraries (JARS)

At a minimum you will need the following libraries to use the Spring Framework’s support for AspectJ LTW:

* spring-aop.jar (version 2.5 or later, plus all mandatory dependencies)
* aspectjweaver.jar (version 1.6.8 or later)

If you are using the [Spring-provided agent to enable instrumentation](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-aj-ltw-environment-generic), you will also need:

* spring-instrument.jar

Spring configuration

The key component in Spring’s LTW support is the LoadTimeWeaver interface (in theorg.springframework.instrument.classloading package), and the numerous implementations of it that ship with the Spring distribution. A LoadTimeWeaver is responsible for adding one or more java.lang.instrument.ClassFileTransformers to a ClassLoader at runtime, which opens the door to all manner of interesting applications, one of which happens to be the LTW of aspects.

|  |  |
| --- | --- |
|  | If you are unfamiliar with the idea of runtime class file transformation, you are encouraged to read the javadoc API documentation for the java.lang.instrument package before continuing. This is not a huge chore because there is - rather annoyingly - precious little documentation there…​ the key interfaces and classes will at least be laid out in front of you for reference as you read through this section. |

Configuring a LoadTimeWeaver for a particular ApplicationContext can be as easy as adding one line. (Please note that you almost certainly will need to be using an ApplicationContext as your Spring container - typically a BeanFactory will not be enough because the LTW support makes use of BeanFactoryPostProcessors.)

To enable the Spring Framework’s LTW support, you need to configure a LoadTimeWeaver, which typically is done using the @EnableLoadTimeWeaving annotation.

@Configuration

@EnableLoadTimeWeaving

**public** **class** **AppConfig** {

}

Alternatively, if you prefer XML based configuration, use the <context:load-time-weaver/> element. Note that the element is defined in the context namespace.

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

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

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

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

xsi:schemaLocation="

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

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

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

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

<context:load-time-weaver/>

</beans>

The above configuration will define and register a number of LTW-specific infrastructure beans for you automatically, such as a LoadTimeWeaver and an AspectJWeavingEnabler. The default LoadTimeWeaver is the DefaultContextLoadTimeWeaver class, which attempts to decorate an automatically detected LoadTimeWeaver: the exact type of LoadTimeWeaver that will be 'automatically detected' is dependent upon your runtime environment (summarized in the following table).

| *Table 13. DefaultContextLoadTimeWeaver LoadTimeWeavers* | |
| --- | --- |
| **Runtime Environment** | LoadTimeWeaver**implementation** |
| Running in Oracle’s [WebLogic](http://www.oracle.com/technetwork/middleware/weblogic/overview/index-085209.html) | WebLogicLoadTimeWeaver |
| Running in Oracle’s [GlassFish](http://glassfish.dev.java.net/) | GlassFishLoadTimeWeaver |
| Running in [Apache Tomcat](https://tomcat.apache.org/) | TomcatLoadTimeWeaver |
| Running in Red Hat’s [JBoss AS](http://www.jboss.org/jbossas/) or [WildFly](http://www.wildfly.org/) | JBossLoadTimeWeaver |
| Running in IBM’s [WebSphere](https://www-01.ibm.com/software/webservers/appserv/was/) | WebSphereLoadTimeWeaver |
| JVM started with Spring InstrumentationSavingAgent *(java -javaagent:path/to/spring-instrument.jar)* | InstrumentationLoadTimeWeaver |
| Fallback, expecting the underlying ClassLoader to follow common conventions (e.g. applicable to TomcatInstrumentableClassLoader and [Resin](http://www.caucho.com/)) | ReflectiveLoadTimeWeaver |

Note that these are just the LoadTimeWeavers that are autodetected when using the DefaultContextLoadTimeWeaver: it is of course possible to specify exactly which LoadTimeWeaver implementation that you wish to use.

To specify a specific LoadTimeWeaver with Java configuration implement the LoadTimeWeavingConfigurer interface and override the getLoadTimeWeaver() method:

@Configuration

@EnableLoadTimeWeaving

**public** **class** **AppConfig** **implements** LoadTimeWeavingConfigurer {

@Override

**public** LoadTimeWeaver getLoadTimeWeaver() {

**return** **new** ReflectiveLoadTimeWeaver();

}

}

If you are using XML based configuration you can specify the fully-qualified classname as the value of the weaver-classattribute on the <context:load-time-weaver/> element:

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

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

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

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

xsi:schemaLocation="

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

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

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

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

<context:load-time-weaver

weaver-class="org.springframework.instrument.classloading.ReflectiveLoadTimeWeaver"/>

</beans>

The LoadTimeWeaver that is defined and registered by the configuration can be later retrieved from the Spring container using the well-known name loadTimeWeaver. Remember that the LoadTimeWeaver exists just as a mechanism for Spring’s LTW infrastructure to add one or more ClassFileTransformers. The actual ClassFileTransformer that does the LTW is the ClassPreProcessorAgentAdapter (from the org.aspectj.weaver.loadtime package) class. See the class-level javadocs of theClassPreProcessorAgentAdapter class for further details, because the specifics of how the weaving is actually effected is beyond the scope of this section.

There is one final attribute of the configuration left to discuss: the aspectjWeaving attribute (or aspectj-weaving if you are using XML). This is a simple attribute that controls whether LTW is enabled or not; it is as simple as that. It accepts one of three possible values, summarized below, with the default value being autodetect if the attribute is not present.

| *Table 14. AspectJ weaving attribute values* | | |
| --- | --- | --- |
| **Annotation Value** | **XML Value** | **Explanation** |
| ENABLED | on | AspectJ weaving is on, and aspects will be woven at load-time as appropriate. |
| DISABLED | off | LTW is off…​ no aspect will be woven at load-time. |
| AUTODETECT | autodetect | If the Spring LTW infrastructure can find at least one META-INF/aop.xml file, then AspectJ weaving is on, else it is off. This is the default value. |

Environment-specific configuration

This last section contains any additional settings and configuration that you will need when using Spring’s LTW support in environments such as application servers and web containers.

Tomcat

Historically, [Apache Tomcat](https://tomcat.apache.org/)'s default class loader did not support class transformation which is why Spring provides an enhanced implementation that addresses this need. Named TomcatInstrumentableClassLoader, the loader works on Tomcat 6.0 and above.

|  |  |
| --- | --- |
|  | Do not define TomcatInstrumentableClassLoader anymore on Tomcat 8.0 and higher. Instead, let Spring automatically use Tomcat’s new native InstrumentableClassLoader facility through the TomcatLoadTimeWeaverstrategy. |

If you still need to use TomcatInstrumentableClassLoader, it can be registered individually for *each* web application as follows:

* Copy org.springframework.instrument.tomcat.jar into *$CATALINA\_HOME*/lib, where *$CATALINA\_HOME* represents the root of the Tomcat installation)
* Instruct Tomcat to use the custom class loader (instead of the default) by editing the web application context file:

<Context path="/myWebApp" docBase="/my/webApp/location">

<Loader

loaderClass="org.springframework.instrument.classloading.tomcat.TomcatInstrumentableClassLoader"/>

</Context>

Apache Tomcat (6.0+) supports several context locations:

* server configuration file - *$CATALINA\_HOME/conf/server.xml*
* default context configuration - *$CATALINA\_HOME/conf/context.xml* - that affects all deployed web applications
* per-web application configuration which can be deployed either on the server-side at*$CATALINA\_HOME/conf/[enginename]/[hostname]/[webapp]-context.xml* or embedded inside the web-app archive at *META-INF/context.xml*

For efficiency, the embedded per-web-app configuration style is recommended because it will impact only applications that use the custom class loader and does not require any changes to the server configuration. See the Tomcat 6.0.x [documentation](https://tomcat.apache.org/tomcat-6.0-doc/config/context.html) for more details about available context locations.

Alternatively, consider the use of the Spring-provided generic VM agent, to be specified in Tomcat’s launch script (see above). This will make instrumentation available to all deployed web applications, no matter what ClassLoader they happen to run on.

WebLogic, WebSphere, Resin, GlassFish, JBoss

Recent versions of WebLogic Server (version 10 and above), IBM WebSphere Application Server (version 7 and above), Resin (3.1 and above) and JBoss (6.x or above) provide a ClassLoader that is capable of local instrumentation. Spring’s native LTW leverages such ClassLoaders to enable AspectJ weaving. You can enable LTW by simply activating load-time weaving as described earlier. Specifically, you do *not* need to modify the launch script to add -javaagent:path/to/spring-instrument.jar.

Note that GlassFish instrumentation-capable ClassLoader is available only in its EAR environment. For GlassFish web applications, follow the Tomcat setup instructions as outlined above.

Note that on JBoss 6.x, the app server scanning needs to be disabled to prevent it from loading the classes before the application actually starts. A quick workaround is to add to your artifact a file named WEB-INF/jboss-scanning.xml with the following content:

<scanning xmlns="urn:jboss:scanning:1.0"/>

Generic Java applications

When class instrumentation is required in environments that do not support or are not supported by the existing LoadTimeWeaver implementations, a JDK agent can be the only solution. For such cases, Spring provides InstrumentationLoadTimeWeaver, which requires a Spring-specific (but very general) VM agent,org.springframework.instrument-{version}.jar (previously named spring-agent.jar).

To use it, you must start the virtual machine with the Spring agent, by supplying the following JVM options:

-javaagent:/path/to/org.springframework.instrument-{version}.jar

Note that this requires modification of the VM launch script which may prevent you from using this in application server environments (depending on your operation policies). Additionally, the JDK agent will instrument the *entire* VM which can prove expensive.

For performance reasons, it is recommended to use this configuration only if your target environment (such as [Jetty](https://www.eclipse.org/jetty/)) does not have (or does not support) a dedicated LTW.

5.9. Further Resources

More information on AspectJ can be found on the [AspectJ website](https://www.eclipse.org/aspectj).

The book *Eclipse AspectJ* by Adrian Colyer et. al. (Addison-Wesley, 2005) provides a comprehensive introduction and reference for the AspectJ language.

The book *AspectJ in Action, Second Edition* by Ramnivas Laddad (Manning, 2009) comes highly recommended; the focus of the book is on AspectJ, but a lot of general AOP themes are explored (in some depth).

6. Spring AOP APIs

6.1. Introduction

The previous chapter described the Spring’s support for AOP using @AspectJ and schema-based aspect definitions. In this chapter we discuss the lower-level Spring AOP APIs and the AOP support typically used in Spring 1.2 applications. For new applications, we recommend the use of the Spring 2.0 and later AOP support described in the previous chapter, but when working with existing applications, or when reading books and articles, you may come across Spring 1.2 style examples. Spring 5 remains backwards compatible with Spring 1.2 and everything described in this chapter is fully supported in Spring 5.

6.2. Pointcut API in Spring

Let’s look at how Spring handles the crucial pointcut concept.

6.2.1. Concepts

Spring’s pointcut model enables pointcut reuse independent of advice types. It’s possible to target different advice using the same pointcut.

The org.springframework.aop.Pointcut interface is the central interface, used to target advices to particular classes and methods. The complete interface is shown below:

**public** **interface** **Pointcut** {

ClassFilter getClassFilter();

MethodMatcher getMethodMatcher();

}

Splitting the Pointcut interface into two parts allows reuse of class and method matching parts, and fine-grained composition operations (such as performing a "union" with another method matcher).

The ClassFilter interface is used to restrict the pointcut to a given set of target classes. If the matches() method always returns true, all target classes will be matched:

**public** **interface** **ClassFilter** {

**boolean** matches(Class clazz);

}

The MethodMatcher interface is normally more important. The complete interface is shown below:

**public** **interface** **MethodMatcher** {

**boolean** matches(Method m, Class targetClass);

**boolean** isRuntime();

**boolean** matches(Method m, Class targetClass, Object**[]** args);

}

The matches(Method, Class) method is used to test whether this pointcut will ever match a given method on a target class. This evaluation can be performed when an AOP proxy is created, to avoid the need for a test on every method invocation. If the 2-argument matches method returns true for a given method, and the isRuntime() method for the MethodMatcher returns true, the 3-argument matches method will be invoked on every method invocation. This enables a pointcut to look at the arguments passed to the method invocation immediately before the target advice is to execute.

Most MethodMatchers are static, meaning that their isRuntime() method returns false. In this case, the 3-argument matches method will never be invoked.

|  |  |
| --- | --- |
|  | If possible, try to make pointcuts static, allowing the AOP framework to cache the results of pointcut evaluation when an AOP proxy is created. |

6.2.2. Operations on pointcuts

Spring supports operations on pointcuts: notably, *union* and *intersection*.

* Union means the methods that either pointcut matches.
* Intersection means the methods that both pointcuts match.
* Union is usually more useful.
* Pointcuts can be composed using the static methods in the *org.springframework.aop.support.Pointcuts* class, or using the*ComposablePointcut* class in the same package. However, using AspectJ pointcut expressions is usually a simpler approach.

6.2.3. AspectJ expression pointcuts

Since 2.0, the most important type of pointcut used by Spring is org.springframework.aop.aspectj.AspectJExpressionPointcut. This is a pointcut that uses an AspectJ supplied library to parse an AspectJ pointcut expression string.

See the previous chapter for a discussion of supported AspectJ pointcut primitives.

6.2.4. Convenience pointcut implementations

Spring provides several convenient pointcut implementations. Some can be used out of the box; others are intended to be subclassed in application-specific pointcuts.

Static pointcuts

Static pointcuts are based on method and target class, and cannot take into account the method’s arguments. Static pointcuts are sufficient - *and best* - for most usages. It’s possible for Spring to evaluate a static pointcut only once, when a method is first invoked: after that, there is no need to evaluate the pointcut again with each method invocation.

Let’s consider some static pointcut implementations included with Spring.

Regular expression pointcuts

One obvious way to specify static pointcuts is regular expressions. Several AOP frameworks besides Spring make this possible.org.springframework.aop.support.JdkRegexpMethodPointcut is a generic regular expression pointcut, using the regular expression support in the JDK.

Using the JdkRegexpMethodPointcut class, you can provide a list of pattern Strings. If any of these is a match, the pointcut will evaluate to true. (So the result is effectively the union of these pointcuts.)

The usage is shown below:

<bean id="settersAndAbsquatulatePointcut"

class="org.springframework.aop.support.JdkRegexpMethodPointcut">

<property name="patterns">

<list>

<value>.\*set.\*</value>

<value>.\*absquatulate</value>

</list>

</property>

</bean>

Spring provides a convenience class, RegexpMethodPointcutAdvisor, that allows us to also reference an Advice (remember that an Advice can be an interceptor, before advice, throws advice etc.). Behind the scenes, Spring will use a JdkRegexpMethodPointcut. Using RegexpMethodPointcutAdvisor simplifies wiring, as the one bean encapsulates both pointcut and advice, as shown below:

<bean id="settersAndAbsquatulateAdvisor"

class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">

<property name="advice">

<ref bean="beanNameOfAopAllianceInterceptor"/>

</property>

<property name="patterns">

<list>

<value>.\*set.\*</value>

<value>.\*absquatulate</value>

</list>

</property>

</bean>

*RegexpMethodPointcutAdvisor* can be used with any Advice type.

Attribute-driven pointcuts

An important type of static pointcut is a *metadata-driven* pointcut. This uses the values of metadata attributes: typically, source-level metadata.

Dynamic pointcuts

Dynamic pointcuts are costlier to evaluate than static pointcuts. They take into account method *arguments*, as well as static information. This means that they must be evaluated with every method invocation; the result cannot be cached, as arguments will vary.

The main example is the control flow pointcut.

Control flow pointcuts

Spring control flow pointcuts are conceptually similar to AspectJ *cflow* pointcuts, although less powerful. (There is currently no way to specify that a pointcut executes below a join point matched by another pointcut.) A control flow pointcut matches the current call stack. For example, it might fire if the join point was invoked by a method in the com.mycompany.web package, or by the SomeCaller class. Control flow pointcuts are specified using the org.springframework.aop.support.ControlFlowPointcutclass.

|  |  |
| --- | --- |
|  | Control flow pointcuts are significantly more expensive to evaluate at runtime than even other dynamic pointcuts. In Java 1.4, the cost is about 5 times that of other dynamic pointcuts. |

6.2.5. Pointcut superclasses

Spring provides useful pointcut superclasses to help you to implement your own pointcuts.

Because static pointcuts are most useful, you’ll probably subclass StaticMethodMatcherPointcut, as shown below. This requires implementing just one abstract method (although it’s possible to override other methods to customize behavior):

**class** **TestStaticPointcut** **extends** StaticMethodMatcherPointcut {

**public** **boolean** matches(Method m, Class targetClass) {

*// return true if custom criteria match*

}

}

There are also superclasses for dynamic pointcuts.

You can use custom pointcuts with any advice type in Spring 1.0 RC2 and above.

6.2.6. Custom pointcuts

Because pointcuts in Spring AOP are Java classes, rather than language features (as in AspectJ) it’s possible to declare custom pointcuts, whether static or dynamic. Custom pointcuts in Spring can be arbitrarily complex. However, using the AspectJ pointcut expression language is recommended if possible.

|  |  |
| --- | --- |
|  | Later versions of Spring may offer support for "semantic pointcuts" as offered by JAC: for example, "all methods that change instance variables in the target object." |

6.3. Advice API in Spring

Let’s now look at how Spring AOP handles advice.

6.3.1. Advice lifecycles

Each advice is a Spring bean. An advice instance can be shared across all advised objects, or unique to each advised object. This corresponds to *per-class* or *per-instance* advice.

Per-class advice is used most often. It is appropriate for generic advice such as transaction advisors. These do not depend on the state of the proxied object or add new state; they merely act on the method and arguments.

Per-instance advice is appropriate for introductions, to support mixins. In this case, the advice adds state to the proxied object.

It’s possible to use a mix of shared and per-instance advice in the same AOP proxy.

6.3.2. Advice types in Spring

Spring provides several advice types out of the box, and is extensible to support arbitrary advice types. Let us look at the basic concepts and standard advice types.

Interception around advice

The most fundamental advice type in Spring is *interception around advice*.

Spring is compliant with the AOP Alliance interface for around advice using method interception. MethodInterceptors implementing around advice should implement the following interface:

**public** **interface** **MethodInterceptor** **extends** Interceptor {

Object invoke(MethodInvocation invocation) **throws** Throwable;

}

The MethodInvocation argument to the invoke() method exposes the method being invoked; the target join point; the AOP proxy; and the arguments to the method. The invoke() method should return the invocation’s result: the return value of the join point.

A simple MethodInterceptor implementation looks as follows:

**public** **class** **DebugInterceptor** **implements** MethodInterceptor {

**public** Object invoke(MethodInvocation invocation) **throws** Throwable {

System.out.println("Before: invocation=[" + invocation + "]");

Object rval = invocation.proceed();

System.out.println("Invocation returned");

**return** rval;

}

}

Note the call to the MethodInvocation’s proceed() method. This proceeds down the interceptor chain towards the join point. Most interceptors will invoke this method, and return its return value. However, a MethodInterceptor, like any around advice, can return a different value or throw an exception rather than invoke the proceed method. However, you don’t want to do this without good reason!

|  |  |
| --- | --- |
|  | MethodInterceptors offer interoperability with other AOP Alliance-compliant AOP implementations. The other advice types discussed in the remainder of this section implement common AOP concepts, but in a Spring-specific way. While there is an advantage in using the most specific advice type, stick with MethodInterceptor around advice if you are likely to want to run the aspect in another AOP framework. Note that pointcuts are not currently interoperable between frameworks, and the AOP Alliance does not currently define pointcut interfaces. |

Before advice

A simpler advice type is a *before advice*. This does not need a MethodInvocation object, since it will only be called before entering the method.

The main advantage of a before advice is that there is no need to invoke the proceed() method, and therefore no possibility of inadvertently failing to proceed down the interceptor chain.

The MethodBeforeAdvice interface is shown below. (Spring’s API design would allow for field before advice, although the usual objects apply to field interception and it’s unlikely that Spring will ever implement it).

**public** **interface** **MethodBeforeAdvice** **extends** BeforeAdvice {

**void** before(Method m, Object**[]** args, Object target) **throws** Throwable;

}

Note the return type is void. Before advice can insert custom behavior before the join point executes, but cannot change the return value. If a before advice throws an exception, this will abort further execution of the interceptor chain. The exception will propagate back up the interceptor chain. If it is unchecked, or on the signature of the invoked method, it will be passed directly to the client; otherwise it will be wrapped in an unchecked exception by the AOP proxy.

An example of a before advice in Spring, which counts all method invocations:

**public** **class** **CountingBeforeAdvice** **implements** MethodBeforeAdvice {

**private** **int** count;

**public** **void** before(Method m, Object**[]** args, Object target) **throws** Throwable {

++count;

}

**public** **int** getCount() {

**return** count;

}

}

|  |  |
| --- | --- |
|  | Before advice can be used with any pointcut. |

Throws advice

*Throws advice* is invoked after the return of the join point if the join point threw an exception. Spring offers typed throws advice. Note that this means that the org.springframework.aop.ThrowsAdvice interface does not contain any methods: It is a tag interface identifying that the given object implements one or more typed throws advice methods. These should be in the form of:

afterThrowing([Method, args, target], subclassOfThrowable)

Only the last argument is required. The method signatures may have either one or four arguments, depending on whether the advice method is interested in the method and arguments. The following classes are examples of throws advice.

The advice below is invoked if a RemoteException is thrown (including subclasses):

**public** **class** **RemoteThrowsAdvice** **implements** ThrowsAdvice {

**public** **void** afterThrowing(RemoteException ex) **throws** Throwable {

*// Do something with remote exception*

}

}

The following advice is invoked if a ServletException is thrown. Unlike the above advice, it declares 4 arguments, so that it has access to the invoked method, method arguments and target object:

**public** **class** **ServletThrowsAdviceWithArguments** **implements** ThrowsAdvice {

**public** **void** afterThrowing(Method m, Object**[]** args, Object target, ServletException ex) {

*// Do something with all arguments*

}

}

The final example illustrates how these two methods could be used in a single class, which handles both RemoteException and ServletException. Any number of throws advice methods can be combined in a single class.

**public** **static** **class** **CombinedThrowsAdvice** **implements** ThrowsAdvice {

**public** **void** afterThrowing(RemoteException ex) **throws** Throwable {

*// Do something with remote exception*

}

**public** **void** afterThrowing(Method m, Object**[]** args, Object target, ServletException ex) {

*// Do something with all arguments*

}

}

|  |  |
| --- | --- |
|  | If a throws-advice method throws an exception itself, it will override the original exception (i.e. change the exception thrown to the user). The overriding exception will typically be a RuntimeException; this is compatible with any method signature. However, if a throws-advice method throws a checked exception, it will have to match the declared exceptions of the target method and is hence to some degree coupled to specific target method signatures. *Do not throw an undeclared checked exception that is incompatible with the target method’s signature!* |
|  | Throws advice can be used with any pointcut. |

After Returning advice

An after returning advice in Spring must implement the *org.springframework.aop.AfterReturningAdvice* interface, shown below:

**public** **interface** **AfterReturningAdvice** **extends** Advice {

**void** afterReturning(Object returnValue, Method m, Object**[]** args, Object target)

**throws** Throwable;

}

An after returning advice has access to the return value (which it cannot modify), invoked method, methods arguments and target.

The following after returning advice counts all successful method invocations that have not thrown exceptions:

**public** **class** **CountingAfterReturningAdvice** **implements** AfterReturningAdvice {

**private** **int** count;

**public** **void** afterReturning(Object returnValue, Method m, Object**[]** args, Object target)

**throws** Throwable {

++count;

}

**public** **int** getCount() {

**return** count;

}

}

This advice doesn’t change the execution path. If it throws an exception, this will be thrown up the interceptor chain instead of the return value.

|  |  |
| --- | --- |
|  | After returning advice can be used with any pointcut. |

Introduction advice

Spring treats introduction advice as a special kind of interception advice.

Introduction requires an IntroductionAdvisor, and an IntroductionInterceptor, implementing the following interface:

**public** **interface** **IntroductionInterceptor** **extends** MethodInterceptor {

**boolean** implementsInterface(Class intf);

}

The invoke() method inherited from the AOP Alliance MethodInterceptor interface must implement the introduction: that is, if the invoked method is on an introduced interface, the introduction interceptor is responsible for handling the method call - it cannot invoke proceed().

Introduction advice cannot be used with any pointcut, as it applies only at class, rather than method, level. You can only use introduction advice with the IntroductionAdvisor, which has the following methods:

**public** **interface** **IntroductionAdvisor** **extends** Advisor, IntroductionInfo {

ClassFilter getClassFilter();

**void** validateInterfaces() **throws** IllegalArgumentException;

}

**public** **interface** **IntroductionInfo** {

Class**[]** getInterfaces();

}

There is no MethodMatcher, and hence no Pointcut, associated with introduction advice. Only class filtering is logical.

The getInterfaces() method returns the interfaces introduced by this advisor.

The validateInterfaces() method is used internally to see whether or not the introduced interfaces can be implemented by the configured IntroductionInterceptor.

Let’s look at a simple example from the Spring test suite. Let’s suppose we want to introduce the following interface to one or more objects:

**public** **interface** **Lockable** {

**void** lock();

**void** unlock();

**boolean** locked();

}

This illustrates a *mixin*. We want to be able to cast advised objects to Lockable, whatever their type, and call lock and unlock methods. If we call the lock() method, we want all setter methods to throw a LockedException. Thus we can add an aspect that provides the ability to make objects immutable, without them having any knowledge of it: a good example of AOP.

Firstly, we’ll need an IntroductionInterceptor that does the heavy lifting. In this case, we extend the org.springframework.aop.support.DelegatingIntroductionInterceptor convenience class. We could implement IntroductionInterceptor directly, but using DelegatingIntroductionInterceptor is best for most cases.

The DelegatingIntroductionInterceptor is designed to delegate an introduction to an actual implementation of the introduced interface(s), concealing the use of interception to do so. The delegate can be set to any object using a constructor argument; the default delegate (when the no-arg constructor is used) is this. Thus in the example below, the delegate is the LockMixinsubclass of DelegatingIntroductionInterceptor. Given a delegate (by default itself), a DelegatingIntroductionInterceptorinstance looks for all interfaces implemented by the delegate (other than IntroductionInterceptor), and will support introductions against any of them. It’s possible for subclasses such as LockMixin to call the suppressInterface(Class intf) method to suppress interfaces that should not be exposed. However, no matter how many interfaces an IntroductionInterceptor is prepared to support, the IntroductionAdvisor used will control which interfaces are actually exposed. An introduced interface will conceal any implementation of the same interface by the target.

Thus LockMixin extends DelegatingIntroductionInterceptor and implements Lockable itself. The superclass automatically picks up that Lockable can be supported for introduction, so we don’t need to specify that. We could introduce any number of interfaces in this way.

Note the use of the locked instance variable. This effectively adds additional state to that held in the target object.

**public** **class** **LockMixin** **extends** DelegatingIntroductionInterceptor **implements** Lockable {

**private** **boolean** locked;

**public** **void** lock() {

this.locked = true;

}

**public** **void** unlock() {

this.locked = false;

}

**public** **boolean** locked() {

**return** this.locked;

}

**public** Object invoke(MethodInvocation invocation) **throws** Throwable {

**if** (locked() && invocation.getMethod().getName().indexOf("set") == 0) {

**throw** **new** LockedException();

}

**return** super.invoke(invocation);

}

}

Often it isn’t necessary to override the invoke() method: the DelegatingIntroductionInterceptor implementation - which calls the delegate method if the method is introduced, otherwise proceeds towards the join point - is usually sufficient. In the present case, we need to add a check: no setter method can be invoked if in locked mode.

The introduction advisor required is simple. All it needs to do is hold a distinct LockMixin instance, and specify the introduced interfaces - in this case, just Lockable. A more complex example might take a reference to the introduction interceptor (which would be defined as a prototype): in this case, there’s no configuration relevant for a LockMixin, so we simply create it using new.

**public** **class** **LockMixinAdvisor** **extends** DefaultIntroductionAdvisor {

**public** LockMixinAdvisor() {

super(**new** LockMixin(), Lockable.class);

}

}

We can apply this advisor very simply: it requires no configuration. (However, it *is* necessary: It’s impossible to use an IntroductionInterceptor without an *IntroductionAdvisor*.) As usual with introductions, the advisor must be per-instance, as it is stateful. We need a different instance of LockMixinAdvisor, and hence LockMixin, for each advised object. The advisor comprises part of the advised object’s state.

We can apply this advisor programmatically, using the Advised.addAdvisor() method, or (the recommended way) in XML configuration, like any other advisor. All proxy creation choices discussed below, including "auto proxy creators," correctly handle introductions and stateful mixins.

6.4. Advisor API in Spring

In Spring, an Advisor is an aspect that contains just a single advice object associated with a pointcut expression.

Apart from the special case of introductions, any advisor can be used with any advice.org.springframework.aop.support.DefaultPointcutAdvisor is the most commonly used advisor class. For example, it can be used with a MethodInterceptor, BeforeAdvice or ThrowsAdvice.

It is possible to mix advisor and advice types in Spring in the same AOP proxy. For example, you could use a interception around advice, throws advice and before advice in one proxy configuration: Spring will automatically create the necessary interceptor chain.

6.5. Using the ProxyFactoryBean to create AOP proxies

If you’re using the Spring IoC container (an ApplicationContext or BeanFactory) for your business objects - and you should be! - you will want to use one of Spring’s AOP FactoryBeans. (Remember that a factory bean introduces a layer of indirection, enabling it to create objects of a different type.)

|  |  |
| --- | --- |
|  | The Spring AOP support also uses factory beans under the covers. |

The basic way to create an AOP proxy in Spring is to use the *org.springframework.aop.framework.ProxyFactoryBean*. This gives complete control over the pointcuts and advice that will apply, and their ordering. However, there are simpler options that are preferable if you don’t need such control.

6.5.1. Basics

The ProxyFactoryBean, like other Spring FactoryBean implementations, introduces a level of indirection. If you define a ProxyFactoryBean with name foo, what objects referencing foo see is not the ProxyFactoryBean instance itself, but an object created by the ProxyFactoryBean's implementation of the getObject() method. This method will create an AOP proxy wrapping a target object.

One of the most important benefits of using a ProxyFactoryBean or another IoC-aware class to create AOP proxies, is that it means that advices and pointcuts can also be managed by IoC. This is a powerful feature, enabling certain approaches that are hard to achieve with other AOP frameworks. For example, an advice may itself reference application objects (besides the target, which should be available in any AOP framework), benefiting from all the pluggability provided by Dependency Injection.

6.5.2. JavaBean properties

In common with most FactoryBean implementations provided with Spring, the ProxyFactoryBean class is itself a JavaBean. Its properties are used to:

* Specify the target you want to proxy.
* Specify whether to use CGLIB (see below and also [JDK- and CGLIB-based proxies](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-pfb-proxy-types)).

Some key properties are inherited from org.springframework.aop.framework.ProxyConfig (the superclass for all AOP proxy factories in Spring). These key properties include:

* proxyTargetClass: true if the target class is to be proxied, rather than the target class' interfaces. If this property value is set to true, then CGLIB proxies will be created (but see also [JDK- and CGLIB-based proxies](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-pfb-proxy-types)).
* optimize: controls whether or not aggressive optimizations are applied to proxies *created via CGLIB*. One should not blithely use this setting unless one fully understands how the relevant AOP proxy handles optimization. This is currently used only for CGLIB proxies; it has no effect with JDK dynamic proxies.
* frozen: if a proxy configuration is frozen, then changes to the configuration are no longer allowed. This is useful both as a slight optimization and for those cases when you don’t want callers to be able to manipulate the proxy (via the Advisedinterface) after the proxy has been created. The default value of this property is false, so changes such as adding additional advice are allowed.
* exposeProxy: determines whether or not the current proxy should be exposed in a ThreadLocal so that it can be accessed by the target. If a target needs to obtain the proxy and the exposeProxy property is set to true, the target can use theAopContext.currentProxy() method.

Other properties specific to ProxyFactoryBean include:

* proxyInterfaces: array of String interface names. If this isn’t supplied, a CGLIB proxy for the target class will be used (but see also [JDK- and CGLIB-based proxies](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-pfb-proxy-types)).
* interceptorNames: String array of Advisor, interceptor or other advice names to apply. Ordering is significant, on a first come-first served basis. That is to say that the first interceptor in the list will be the first to be able to intercept the invocation.

The names are bean names in the current factory, including bean names from ancestor factories. You can’t mention bean references here since doing so would result in the ProxyFactoryBean ignoring the singleton setting of the advice.

You can append an interceptor name with an asterisk ( \*). This will result in the application of all advisor beans with names starting with the part before the asterisk to be applied. An example of using this feature can be found in [Using 'global' advisors](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-global-advisors).

* singleton: whether or not the factory should return a single object, no matter how often the getObject() method is called. Several FactoryBean implementations offer such a method. The default value is true. If you want to use stateful advice - for example, for stateful mixins - use prototype advices along with a singleton value of false.

6.5.3. JDK- and CGLIB-based proxies

This section serves as the definitive documentation on how the ProxyFactoryBean chooses to create one of either a JDK- and CGLIB-based proxy for a particular target object (that is to be proxied).

|  |  |
| --- | --- |
|  | The behavior of the ProxyFactoryBean with regard to creating JDK- or CGLIB-based proxies changed between versions 1.2.x and 2.0 of Spring. The ProxyFactoryBean now exhibits similar semantics with regard to auto-detecting interfaces as those of the TransactionProxyFactoryBean class. |

If the class of a target object that is to be proxied (hereafter simply referred to as the target class) doesn’t implement any interfaces, then a CGLIB-based proxy will be created. This is the easiest scenario, because JDK proxies are interface based, and no interfaces means JDK proxying isn’t even possible. One simply plugs in the target bean, and specifies the list of interceptors via the interceptorNames property. Note that a CGLIB-based proxy will be created even if the proxyTargetClass property of theProxyFactoryBean has been set to false. (Obviously this makes no sense, and is best removed from the bean definition because it is at best redundant, and at worst confusing.)

If the target class implements one (or more) interfaces, then the type of proxy that is created depends on the configuration of the ProxyFactoryBean.

If the proxyTargetClass property of the ProxyFactoryBean has been set to true, then a CGLIB-based proxy will be created. This makes sense, and is in keeping with the principle of least surprise. Even if the proxyInterfaces property of theProxyFactoryBean has been set to one or more fully qualified interface names, the fact that the proxyTargetClass property is set to true *will* cause CGLIB-based proxying to be in effect.

If the proxyInterfaces property of the ProxyFactoryBean has been set to one or more fully qualified interface names, then a JDK-based proxy will be created. The created proxy will implement all of the interfaces that were specified in the proxyInterfaces property; if the target class happens to implement a whole lot more interfaces than those specified in the proxyInterfaces property, that is all well and good but those additional interfaces will not be implemented by the returned proxy.

If the proxyInterfaces property of the ProxyFactoryBean has *not* been set, but the target class *does implement one (or more)*interfaces, then the ProxyFactoryBean will auto-detect the fact that the target class does actually implement at least one interface, and a JDK-based proxy will be created. The interfaces that are actually proxied will be *all* of the interfaces that the target class implements; in effect, this is the same as simply supplying a list of each and every interface that the target class implements to the proxyInterfaces property. However, it is significantly less work, and less prone to typos.

6.5.4. Proxying interfaces

Let’s look at a simple example of ProxyFactoryBean in action. This example involves:

* A *target bean* that will be proxied. This is the "personTarget" bean definition in the example below.
* An Advisor and an Interceptor used to provide advice.
* An AOP proxy bean definition specifying the target object (the personTarget bean) and the interfaces to proxy, along with the advices to apply.

<bean id="personTarget" class="com.mycompany.PersonImpl">

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

<property name="age" value="51"/>

</bean>

<bean id="myAdvisor" class="com.mycompany.MyAdvisor">

<property name="someProperty" value="Custom string property value"/>

</bean>

<bean id="debugInterceptor" class="org.springframework.aop.interceptor.DebugInterceptor">

</bean>

<bean id="person"

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

<property name="proxyInterfaces" value="com.mycompany.Person"/>

<property name="target" ref="personTarget"/>

<property name="interceptorNames">

<list>

<value>myAdvisor</value>

<value>debugInterceptor</value>

</list>

</property>

</bean>

Note that the interceptorNames property takes a list of String: the bean names of the interceptor or advisors in the current factory. Advisors, interceptors, before, after returning and throws advice objects can be used. The ordering of advisors is significant.

|  |  |
| --- | --- |
|  | You might be wondering why the list doesn’t hold bean references. The reason for this is that if the ProxyFactoryBean’s singleton property is set to false, it must be able to return independent proxy instances. If any of the advisors is itself a prototype, an independent instance would need to be returned, so it’s necessary to be able to obtain an instance of the prototype from the factory; holding a reference isn’t sufficient. |

The "person" bean definition above can be used in place of a Person implementation, as follows:

Person person = (Person) factory.getBean("person");

Other beans in the same IoC context can express a strongly typed dependency on it, as with an ordinary Java object:

<bean id="personUser" class="com.mycompany.PersonUser">

<property name="person"><ref bean="person"/></property>

</bean>

The PersonUser class in this example would expose a property of type Person. As far as it’s concerned, the AOP proxy can be used transparently in place of a "real" person implementation. However, its class would be a dynamic proxy class. It would be possible to cast it to the Advised interface (discussed below).

It’s possible to conceal the distinction between target and proxy using an anonymous *inner bean*, as follows. Only the ProxyFactoryBean definition is different; the advice is included only for completeness:

<bean id="myAdvisor" class="com.mycompany.MyAdvisor">

<property name="someProperty" value="Custom string property value"/>

</bean>

<bean id="debugInterceptor" class="org.springframework.aop.interceptor.DebugInterceptor"/>

<bean id="person" class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="proxyInterfaces" value="com.mycompany.Person"/>

*<!-- Use inner bean, not local reference to target -->*

<property name="target">

<bean class="com.mycompany.PersonImpl">

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

<property name="age" value="51"/>

</bean>

</property>

<property name="interceptorNames">

<list>

<value>myAdvisor</value>

<value>debugInterceptor</value>

</list>

</property>

</bean>

This has the advantage that there’s only one object of type Person: useful if we want to prevent users of the application context from obtaining a reference to the un-advised object, or need to avoid any ambiguity with Spring IoC *autowiring*. There’s also arguably an advantage in that the ProxyFactoryBean definition is self-contained. However, there are times when being able to obtain the un-advised target from the factory might actually be an *advantage*: for example, in certain test scenarios.

6.5.5. Proxying classes

What if you need to proxy a class, rather than one or more interfaces?

Imagine that in our example above, there was no Person interface: we needed to advise a class called Person that didn’t implement any business interface. In this case, you can configure Spring to use CGLIB proxying, rather than dynamic proxies. Simply set the proxyTargetClass property on the ProxyFactoryBean above to true. While it’s best to program to interfaces, rather than classes, the ability to advise classes that don’t implement interfaces can be useful when working with legacy code. (In general, Spring isn’t prescriptive. While it makes it easy to apply good practices, it avoids forcing a particular approach.)

If you want to, you can force the use of CGLIB in any case, even if you do have interfaces.

CGLIB proxying works by generating a subclass of the target class at runtime. Spring configures this generated subclass to delegate method calls to the original target: the subclass is used to implement the *Decorator* pattern, weaving in the advice.

CGLIB proxying should generally be transparent to users. However, there are some issues to consider:

* Final methods can’t be advised, as they can’t be overridden.
* There is no need to add CGLIB to your classpath. As of Spring 3.2, CGLIB is repackaged and included in the spring-core JAR. In other words, CGLIB-based AOP will work "out of the box" just as do JDK dynamic proxies.

There’s little performance difference between CGLIB proxying and dynamic proxies. As of Spring 1.0, dynamic proxies are slightly faster. However, this may change in the future. Performance should not be a decisive consideration in this case.

6.5.6. Using 'global' advisors

By appending an asterisk to an interceptor name, all advisors with bean names matching the part before the asterisk, will be added to the advisor chain. This can come in handy if you need to add a standard set of 'global' advisors:

<bean id="proxy" class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="target" ref="service"/>

<property name="interceptorNames">

<list>

<value>global\*</value>

</list>

</property>

</bean>

<bean id="global\_debug" class="org.springframework.aop.interceptor.DebugInterceptor"/>

<bean id="global\_performance" class="org.springframework.aop.interceptor.PerformanceMonitorInterceptor"/>

6.6. Concise proxy definitions

Especially when defining transactional proxies, you may end up with many similar proxy definitions. The use of parent and child bean definitions, along with inner bean definitions, can result in much cleaner and more concise proxy definitions.

First a parent, *template*, bean definition is created for the proxy:

<bean id="txProxyTemplate" abstract="true"

class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">

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

<property name="transactionAttributes">

<props>

<prop key="\*">PROPAGATION\_REQUIRED</prop>

</props>

</property>

</bean>

This will never be instantiated itself, so may actually be incomplete. Then each proxy which needs to be created is just a child bean definition, which wraps the target of the proxy as an inner bean definition, since the target will never be used on its own anyway.

<bean id="myService" parent="txProxyTemplate">

<property name="target">

<bean class="org.springframework.samples.MyServiceImpl">

</bean>

</property>

</bean>

It is of course possible to override properties from the parent template, such as in this case, the transaction propagation settings:

<bean id="mySpecialService" parent="txProxyTemplate">

<property name="target">

<bean class="org.springframework.samples.MySpecialServiceImpl">

</bean>

</property>

<property name="transactionAttributes">

<props>

<prop key="get\*">PROPAGATION\_REQUIRED,readOnly</prop>

<prop key="find\*">PROPAGATION\_REQUIRED,readOnly</prop>

<prop key="load\*">PROPAGATION\_REQUIRED,readOnly</prop>

<prop key="store\*">PROPAGATION\_REQUIRED</prop>

</props>

</property>

</bean>

Note that in the example above, we have explicitly marked the parent bean definition as *abstract* by using the *abstract* attribute, as described [previously](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-child-bean-definitions), so that it may not actually ever be instantiated. Application contexts (but not simple bean factories) will by default pre-instantiate all singletons. It is therefore important (at least for singleton beans) that if you have a (parent) bean definition which you intend to use only as a template, and this definition specifies a class, you must make sure to set the *abstract* attribute to *true*, otherwise the application context will actually try to pre-instantiate it.

6.7. Creating AOP proxies programmatically with the ProxyFactory

It’s easy to create AOP proxies programmatically using Spring. This enables you to use Spring AOP without dependency on Spring IoC.

The following listing shows creation of a proxy for a target object, with one interceptor and one advisor. The interfaces implemented by the target object will automatically be proxied:

ProxyFactory factory = **new** ProxyFactory(myBusinessInterfaceImpl);

factory.addAdvice(myMethodInterceptor);

factory.addAdvisor(myAdvisor);

MyBusinessInterface tb = (MyBusinessInterface) factory.getProxy();

The first step is to construct an object of type org.springframework.aop.framework.ProxyFactory. You can create this with a target object, as in the above example, or specify the interfaces to be proxied in an alternate constructor.

You can add advices (with interceptors as a specialized kind of advice) and/or advisors, and manipulate them for the life of the ProxyFactory. If you add an IntroductionInterceptionAroundAdvisor, you can cause the proxy to implement additional interfaces.

There are also convenience methods on ProxyFactory (inherited from AdvisedSupport) which allow you to add other advice types such as before and throws advice. AdvisedSupport is the superclass of both ProxyFactory and ProxyFactoryBean.

|  |  |
| --- | --- |
|  | Integrating AOP proxy creation with the IoC framework is best practice in most applications. We recommend that you externalize configuration from Java code with AOP, as in general. |

6.8. Manipulating advised objects

However you create AOP proxies, you can manipulate them using the org.springframework.aop.framework.Advised interface. Any AOP proxy can be cast to this interface, whichever other interfaces it implements. This interface includes the following methods:

Advisor**[]** getAdvisors();

**void** addAdvice(Advice advice) **throws** AopConfigException;

**void** addAdvice(**int** pos, Advice advice) **throws** AopConfigException;

**void** addAdvisor(Advisor advisor) **throws** AopConfigException;

**void** addAdvisor(**int** pos, Advisor advisor) **throws** AopConfigException;

**int** indexOf(Advisor advisor);

**boolean** removeAdvisor(Advisor advisor) **throws** AopConfigException;

**void** removeAdvisor(**int** index) **throws** AopConfigException;

**boolean** replaceAdvisor(Advisor a, Advisor b) **throws** AopConfigException;

**boolean** isFrozen();

The getAdvisors() method will return an Advisor for every advisor, interceptor or other advice type that has been added to the factory. If you added an Advisor, the returned advisor at this index will be the object that you added. If you added an interceptor or other advice type, Spring will have wrapped this in an advisor with a pointcut that always returns true. Thus if you added a MethodInterceptor, the advisor returned for this index will be an DefaultPointcutAdvisor returning your MethodInterceptorand a pointcut that matches all classes and methods.

The addAdvisor() methods can be used to add any Advisor. Usually the advisor holding pointcut and advice will be the generic DefaultPointcutAdvisor, which can be used with any advice or pointcut (but not for introductions).

By default, it’s possible to add or remove advisors or interceptors even once a proxy has been created. The only restriction is that it’s impossible to add or remove an introduction advisor, as existing proxies from the factory will not show the interface change. (You can obtain a new proxy from the factory to avoid this problem.)

A simple example of casting an AOP proxy to the Advised interface and examining and manipulating its advice:

Advised advised = (Advised) myObject;

Advisor**[]** advisors = advised.getAdvisors();

**int** oldAdvisorCount = advisors.length;

System.out.println(oldAdvisorCount + " advisors");

*// Add an advice like an interceptor without a pointcut*

*// Will match all proxied methods*

*// Can use for interceptors, before, after returning or throws advice*

advised.addAdvice(**new** DebugInterceptor());

*// Add selective advice using a pointcut*

advised.addAdvisor(**new** DefaultPointcutAdvisor(mySpecialPointcut, myAdvice));

assertEquals("Added two advisors", oldAdvisorCount + 2, advised.getAdvisors().length);

|  |  |
| --- | --- |
|  | It’s questionable whether it’s advisable (no pun intended) to modify advice on a business object in production, although there are no doubt legitimate usage cases. However, it can be very useful in development: for example, in tests. I have sometimes found it very useful to be able to add test code in the form of an interceptor or other advice, getting inside a method invocation I want to test. (For example, the advice can get inside a transaction created for that method: for example, to run SQL to check that a database was correctly updated, before marking the transaction for roll back.) |

Depending on how you created the proxy, you can usually set a frozen flag, in which case the Advised isFrozen() method will return true, and any attempts to modify advice through addition or removal will result in an AopConfigException. The ability to freeze the state of an advised object is useful in some cases, for example, to prevent calling code removing a security interceptor. It may also be used in Spring 1.1 to allow aggressive optimization if runtime advice modification is known not to be required.

6.9. Using the "auto-proxy" facility

So far we’ve considered explicit creation of AOP proxies using a ProxyFactoryBean or similar factory bean.

Spring also allows us to use "auto-proxy" bean definitions, which can automatically proxy selected bean definitions. This is built on Spring "bean post processor" infrastructure, which enables modification of any bean definition as the container loads.

In this model, you set up some special bean definitions in your XML bean definition file to configure the auto proxy infrastructure. This allows you just to declare the targets eligible for auto-proxying: you don’t need to use ProxyFactoryBean.

There are two ways to do this:

* Using an auto-proxy creator that refers to specific beans in the current context.
* A special case of auto-proxy creation that deserves to be considered separately; auto-proxy creation driven by source-level metadata attributes.

6.9.1. Autoproxy bean definitions

The org.springframework.aop.framework.autoproxy package provides the following standard auto-proxy creators.

BeanNameAutoProxyCreator

The BeanNameAutoProxyCreator class is a BeanPostProcessor that automatically creates AOP proxies for beans with names matching literal values or wildcards.

<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">

<property name="beanNames" value="jdk\*,onlyJdk"/>

<property name="interceptorNames">

<list>

<value>myInterceptor</value>

</list>

</property>

</bean>

As with ProxyFactoryBean, there is an interceptorNames property rather than a list of interceptors, to allow correct behavior for prototype advisors. Named "interceptors" can be advisors or any advice type.

As with auto proxying in general, the main point of using BeanNameAutoProxyCreator is to apply the same configuration consistently to multiple objects, with minimal volume of configuration. It is a popular choice for applying declarative transactions to multiple objects.

Bean definitions whose names match, such as "jdkMyBean" and "onlyJdk" in the above example, are plain old bean definitions with the target class. An AOP proxy will be created automatically by the BeanNameAutoProxyCreator. The same advice will be applied to all matching beans. Note that if advisors are used (rather than the interceptor in the above example), the pointcuts may apply differently to different beans.

DefaultAdvisorAutoProxyCreator

A more general and extremely powerful auto proxy creator is DefaultAdvisorAutoProxyCreator. This will automagically apply eligible advisors in the current context, without the need to include specific bean names in the auto-proxy advisor’s bean definition. It offers the same merit of consistent configuration and avoidance of duplication as BeanNameAutoProxyCreator.

Using this mechanism involves:

* Specifying a DefaultAdvisorAutoProxyCreator bean definition.
* Specifying any number of Advisors in the same or related contexts. Note that these *must* be Advisors, not just interceptors or other advices. This is necessary because there must be a pointcut to evaluate, to check the eligibility of each advice to candidate bean definitions.

The DefaultAdvisorAutoProxyCreator will automatically evaluate the pointcut contained in each advisor, to see what (if any) advice it should apply to each business object (such as "businessObject1" and "businessObject2" in the example).

This means that any number of advisors can be applied automatically to each business object. If no pointcut in any of the advisors matches any method in a business object, the object will not be proxied. As bean definitions are added for new business objects, they will automatically be proxied if necessary.

Autoproxying in general has the advantage of making it impossible for callers or dependencies to obtain an un-advised object. Calling getBean("businessObject1") on this ApplicationContext will return an AOP proxy, not the target business object. (The "inner bean" idiom shown earlier also offers this benefit.)

<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>

<bean class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">

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

</bean>

<bean id="customAdvisor" class="com.mycompany.MyAdvisor"/>

<bean id="businessObject1" class="com.mycompany.BusinessObject1">

*<!-- Properties omitted -->*

</bean>

<bean id="businessObject2" class="com.mycompany.BusinessObject2"/>

The DefaultAdvisorAutoProxyCreator is very useful if you want to apply the same advice consistently to many business objects. Once the infrastructure definitions are in place, you can simply add new business objects without including specific proxy configuration. You can also drop in additional aspects very easily - for example, tracing or performance monitoring aspects - with minimal change to configuration.

The DefaultAdvisorAutoProxyCreator offers support for filtering (using a naming convention so that only certain advisors are evaluated, allowing use of multiple, differently configured, AdvisorAutoProxyCreators in the same factory) and ordering. Advisors can implement the org.springframework.core.Ordered interface to ensure correct ordering if this is an issue. The TransactionAttributeSourceAdvisor used in the above example has a configurable order value; the default setting is unordered.

6.10. Using TargetSources

Spring offers the concept of a *TargetSource*, expressed in the org.springframework.aop.TargetSource interface. This interface is responsible for returning the "target object" implementing the join point. The TargetSource implementation is asked for a target instance each time the AOP proxy handles a method invocation.

Developers using Spring AOP don’t normally need to work directly with TargetSources, but this provides a powerful means of supporting pooling, hot swappable and other sophisticated targets. For example, a pooling TargetSource can return a different target instance for each invocation, using a pool to manage instances.

If you do not specify a TargetSource, a default implementation is used that wraps a local object. The same target is returned for each invocation (as you would expect).

Let’s look at the standard target sources provided with Spring, and how you can use them.

|  |  |
| --- | --- |
|  | When using a custom target source, your target will usually need to be a prototype rather than a singleton bean definition. This allows Spring to create a new target instance when required. |

6.10.1. Hot swappable target sources

The org.springframework.aop.target.HotSwappableTargetSource exists to allow the target of an AOP proxy to be switched while allowing callers to keep their references to it.

Changing the target source’s target takes effect immediately. The HotSwappableTargetSource is threadsafe.

You can change the target via the swap() method on HotSwappableTargetSource as follows:

HotSwappableTargetSource swapper = (HotSwappableTargetSource) beanFactory.getBean("swapper");

Object oldTarget = swapper.swap(newTarget);

The XML definitions required look as follows:

<bean id="initialTarget" class="mycompany.OldTarget"/>

<bean id="swapper" class="org.springframework.aop.target.HotSwappableTargetSource">

<constructor-arg ref="initialTarget"/>

</bean>

<bean id="swappable" class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="targetSource" ref="swapper"/>

</bean>

The above swap() call changes the target of the swappable bean. Clients who hold a reference to that bean will be unaware of the change, but will immediately start hitting the new target.

Although this example doesn’t add any advice - and it’s not necessary to add advice to use a TargetSource - of course any TargetSource can be used in conjunction with arbitrary advice.

6.10.2. Pooling target sources

Using a pooling target source provides a similar programming model to stateless session EJBs, in which a pool of identical instances is maintained, with method invocations going to free objects in the pool.

A crucial difference between Spring pooling and SLSB pooling is that Spring pooling can be applied to any POJO. As with Spring in general, this service can be applied in a non-invasive way.

Spring provides out-of-the-box support for Commons Pool 2.2, which provides a fairly efficient pooling implementation. You’ll need the commons-pool Jar on your application’s classpath to use this feature. It’s also possible to subclassorg.springframework.aop.target.AbstractPoolingTargetSource to support any other pooling API.

|  |  |
| --- | --- |
|  | Commons Pool 1.5+ is also supported but deprecated as of Spring Framework 4.2. |

Sample configuration is shown below:

<bean id="businessObjectTarget" class="com.mycompany.MyBusinessObject"

scope="prototype">

... properties omitted

</bean>

<bean id="poolTargetSource" class="org.springframework.aop.target.CommonsPool2TargetSource">

<property name="targetBeanName" value="businessObjectTarget"/>

<property name="maxSize" value="25"/>

</bean>

<bean id="businessObject" class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="targetSource" ref="poolTargetSource"/>

<property name="interceptorNames" value="myInterceptor"/>

</bean>

Note that the target object - "businessObjectTarget" in the example - *must* be a prototype. This allows the PoolingTargetSourceimplementation to create new instances of the target to grow the pool as necessary. See the javadocs ofAbstractPoolingTargetSource and the concrete subclass you wish to use for information about its properties: "maxSize" is the most basic, and always guaranteed to be present.

In this case, "myInterceptor" is the name of an interceptor that would need to be defined in the same IoC context. However, it isn’t necessary to specify interceptors to use pooling. If you want only pooling, and no other advice, don’t set the interceptorNames property at all.

It’s possible to configure Spring so as to be able to cast any pooled object to theorg.springframework.aop.target.PoolingConfig interface, which exposes information about the configuration and current size of the pool through an introduction. You’ll need to define an advisor like this:

<bean id="poolConfigAdvisor" class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">

<property name="targetObject" ref="poolTargetSource"/>

<property name="targetMethod" value="getPoolingConfigMixin"/>

</bean>

This advisor is obtained by calling a convenience method on the AbstractPoolingTargetSource class, hence the use of MethodInvokingFactoryBean. This advisor’s name ("poolConfigAdvisor" here) must be in the list of interceptors names in the ProxyFactoryBean exposing the pooled object.

The cast will look as follows:

PoolingConfig conf = (PoolingConfig) beanFactory.getBean("businessObject");

System.out.println("Max pool size is " + conf.getMaxSize());

|  |  |
| --- | --- |
|  | Pooling stateless service objects is not usually necessary. We don’t believe it should be the default choice, as most stateless objects are naturally thread safe, and instance pooling is problematic if resources are cached. |

Simpler pooling is available using auto-proxying. It’s possible to set the TargetSources used by any auto-proxy creator.

6.10.3. Prototype target sources

Setting up a "prototype" target source is similar to a pooling TargetSource. In this case, a new instance of the target will be created on every method invocation. Although the cost of creating a new object isn’t high in a modern JVM, the cost of wiring up the new object (satisfying its IoC dependencies) may be more expensive. Thus you shouldn’t use this approach without very good reason.

To do this, you could modify the poolTargetSource definition shown above as follows. (I’ve also changed the name, for clarity.)

<bean id="prototypeTargetSource" class="org.springframework.aop.target.PrototypeTargetSource">

<property name="targetBeanName" ref="businessObjectTarget"/>

</bean>

There’s only one property: the name of the target bean. Inheritance is used in the TargetSource implementations to ensure consistent naming. As with the pooling target source, the target bean must be a prototype bean definition.

6.10.4. ThreadLocal target sources

ThreadLocal target sources are useful if you need an object to be created for each incoming request (per thread that is). The concept of a ThreadLocal provide a JDK-wide facility to transparently store resource alongside a thread. Setting up aThreadLocalTargetSource is pretty much the same as was explained for the other types of target source:

<bean id="threadlocalTargetSource" class="org.springframework.aop.target.ThreadLocalTargetSource">

<property name="targetBeanName" value="businessObjectTarget"/>

</bean>

|  |  |
| --- | --- |
|  | ThreadLocals come with serious issues (potentially resulting in memory leaks) when incorrectly using them in a multi-threaded and multi-classloader environments. One should always consider wrapping a threadlocal in some other class and never directly use the ThreadLocal itself (except of course in the wrapper class). Also, one should always remember to correctly set and unset (where the latter simply involved a call toThreadLocal.set(null)) the resource local to the thread. Unsetting should be done in any case since not unsetting it might result in problematic behavior. Spring’s ThreadLocal support does this for you and should always be considered in favor of using ThreadLocals without other proper handling code. |

6.11. Defining new Advice types

Spring AOP is designed to be extensible. While the interception implementation strategy is presently used internally, it is possible to support arbitrary advice types in addition to the out-of-the-box interception around advice, before, throws advice and after returning advice.

The org.springframework.aop.framework.adapter package is an SPI package allowing support for new custom advice types to be added without changing the core framework. The only constraint on a custom Advice type is that it must implement theorg.aopalliance.aop.Advice marker interface.

Please refer to the org.springframework.aop.framework.adapter javadocs for further information.

7. Null-safety

Although Java does not allow to express null-safety with its type system, Spring Framework now provides following annotations in the org.springframework.lang package to declare nullability of APIs and fields:

* [@NonNull](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/lang/NonNull.html) annotation where specific parameter, return value or field cannot be null (not needed on parameter and return value where @NonNullApi and @NonNullFields apply) .
* [@Nullable](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/lang/Nullable.html) annotation where specific parameter, return value or field can be null.
* [@NonNullApi](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/lang/NonNullApi.html) annotation at package level declares non-null as the default behavior for parameters and return values.
* [@NonNullFields](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/lang/NonNullFields.html) annotation at package level declares non-null as the default behavior for fields.

Spring Framework leverages itself these annotations, but they can also be used in any Spring based Java project to declare null-safe APIs and optionally null-safe fields. Generic type arguments, varargs and array elements nullability are not supported yet, but should be in an upcoming release, see [SPR-15942](https://jira.spring.io/browse/SPR-15942) for up-to-date information. Nullability declaration are expected to be fine-tuned between Spring Framework release, including minor ones. Nullability of types used inside method bodies is outside of the scope of this feature.

|  |  |
| --- | --- |
|  | Libraries like Reactor or Spring Data provide null-safe APIs leveraging this feature. |

7.1. Use cases

In addition to providing an explicit declaration for Spring Framework API nullability, these annotation can be used by IDE (such as IDEA or Eclipse) to provide useful warnings to Java developers related to null-safety in order to avoid NullPointerExceptionat runtime.

They are also used to make Spring API null-safe in Kotlin projects since Kotlin natively supports [null-safety](https://kotlinlang.org/docs/reference/null-safety.html). More details are available in [Kotlin support documentation](https://docs.spring.io/spring/docs/current/spring-framework-reference/languages.html#kotlin-null-safety).

7.2. JSR 305 meta-annotations

Spring annotations are meta-annotated with [JSR 305](https://jcp.org/en/jsr/detail?id=305) annotations (a dormant but widely spread JSR). JSR 305 meta-annotations allows tooling vendors like IDEA or Kotlin to provide null-safety support in a generic way, without having to hard-code support for Spring annotations.

It is not necessary nor recommended to add JSR 305 dependency in project classpath to take advantage of Spring null-safe API. Only projects like Spring-based libraries using null-safety annotations in their codebase should addcom.google.code.findbugs:jsr305:3.0.2 with compileOnly Gradle configuration or Maven provided scope to avoid compile warnings.

8. Data Buffers and Codecs

8.1. Introduction

The DataBuffer interface defines an abstraction over byte buffers. The main reason for introducing it, and not use the standard java.nio.ByteBuffer instead, is Netty. Netty does not use ByteBuffer, but instead offers ByteBuf as an alternative. Spring’s DataBuffer is a simple abstraction over ByteBuf that can also be used on non-Netty platforms (i.e. Servlet 3.1+).

8.2. DataBufferFactory

The DataBufferFactory offers functionality to allocate new data buffers, as well as to wrap existing data. The allocatemethods allocate a new data buffer, with a default or given capacity. Though DataBuffer implementation grow and shrink on demand, it is more efficient to give the capacity upfront, if known. The wrap methods decorate an existing ByteBuffer or byte array. Wrapping does not involve allocation: it simply decorates the given data with a DataBuffer implementation.

There are two implementation of DataBufferFactory: the NettyDataBufferFactory which is meant to be used on Netty platforms, such as Reactor Netty. The other implementation, the DefaultDataBufferFactory, is used on other platforms, such as Servlet 3.1+ servers.

8.3. The DataBuffer interface

The DataBuffer interface is similar to ByteBuffer, but offers a number of advantages. Similar to Netty’s ByteBuf, the DataBuffer abstraction offers independent read and write positions. This is different from the JDK’s ByteBuffer, which only exposes one position for both reading and writing, and a separate flip() operation to switch between the two I/O operations. In general, the following invariant holds for the read position, write position, and the capacity:

`0` <= \_read position\_ <= \_write position\_ <= \_capacity\_

When reading bytes from the DataBuffer, the read position is automatically updated in accordance with the amount of data read from the buffer. Similarly, when writing bytes to the DataBuffer, the write position is updated with the amount of data written to the buffer. Also, when writing data, the capacity of a DataBuffer is automatically expanded, just like StringBuilder,ArrayList, and similar types.

Besides the reading and writing functionality mentioned above, the DataBuffer also has methods to view a (slice of a) buffer as ByteBuffer, InputStream, or OutputStream. Additionally, it offers methods to determine the index of a given byte.

There are two implementation of DataBuffer: the NettyDataBuffer which is meant to be used on Netty platforms, such as Reactor Netty. The other implementation, the DefaultDataBuffer, is used on other platforms, such as Servlet 3.1+ servers.

8.3.1. PooledDataBuffer

The PooledDataBuffer is an extension to DataBuffer that adds methods for reference counting. The retain method increases the reference count by one. The release method decreases the count by one, and releases the buffer’s memory when the count reaches 0. Both of these methods are related to *reference counting*, a mechanism that is explained below.

Note that DataBufferUtils offers useful utility methods for releasing and retaining pooled data buffers. These methods take a plain DataBuffer as parameter, but only call retain or release if the passed data buffer is an instance of PooledDataBuffer.

Reference Counting

Reference counting is not a common technique in Java; it is much more common in other programming languages such as Object C and C++. In and of itself, reference counting is not complex: it basically involves tracking the number of references that apply to an object. The reference count of a PooledDataBuffer starts at 1, is incremented by calling retain, and decremented by calling release. As long as the buffer’s reference count is larger than 0 the buffer will not be released. When the number decreases to 0, the instance will be released. In practice, this means that the reserved memory captured by the buffer will be returned back to the memory pool, ready to be used for future allocations.

In general, *the last component to access a DataBuffer is responsible for releasing it*. Withing Spring, there are two sorts of components that release buffers: decoders and transports. Decoders are responsible for transforming a stream of buffers into other types (see [Codecs](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#codecs) below), and transports are responsible for sending buffers across a network boundary, typically as an HTTP message. This means that if you allocate data buffers for the purpose of putting them into an outbound HTTP message (i.e. client-side request or server-side response), they do not have to be released. The other consequence of this rule is that if you allocate data buffers that do not end up in the body, for instance because of a thrown exception, you will have to release them yourself. The following snippet shows a typical DataBuffer usage scenario when dealing with methods that throw exceptions:

DataBufferFactory factory = ...

DataBuffer buffer = factory.allocateBuffer();

**boolean** release = true;

**try** {

writeDataToBuffer(buffer);

putBufferInHttpBody(buffer);

release = false;

}

**finally** {

**if** (release) {

DataBufferUtils.release(buffer);

}

}

**private** **void** writeDataToBuffer(DataBuffer buffer) **throws** IOException {

...

}

|  |  |
| --- | --- |
|  | A new buffer is allocated. |
|  | A boolean flag indicates whether the allocated buffer should be released. |
|  | This example method loads data into the buffer. Note that the method can throw an IOException, and therefore a finally block to release the buffer is required. |
|  | If no exception occurred, we switch the release flag to false as the buffer will now be released as part of sending the HTTP body across the wire. |
|  | If an exception did occur, the flag is still set to true, and the buffer will be released here. |

8.3.2. DataBufferUtils

DataBufferUtils contains various utility methods that operate on data buffers. It contains methods for reading a Flux of DataBuffer objects from an InputStream or NIO Channel, and methods for writing a data buffer Flux to an OutputStream or Channel. DataBufferUtils also exposes retain and release methods that operate on plain DataBuffer instances (so that casting to a PooledDataBuffer is not required).

Additionally, DataBufferUtils exposes compose, which merges a stream of data buffers into one. For instance, this method can be used to convert the entire HTTP body into a single buffer (and from that, a String, or InputStream). This is particularly useful when dealing with older, blocking APIs. Note, however, that this puts the entire body in memory, and therefore uses more memory than a pure streaming solution would.

Codecs

The org.springframework.core.codec package contains the two main abstractions for converting a stream of bytes into a stream of objects, or vice-versa. The Encoder is a strategy interface that encodes a stream of objects into an output stream of data buffers. The Decoder does the reverse: it turns a stream of data buffers into a stream of objects. Note that a decoder instance needs to consider [reference counting](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#databuffer-reference-counting).

Spring comes with a wide array of default codecs, capable of converting from/to String, ByteBuffer, byte arrays, and also codecs that support marshalling libraries such as JAXB and Jackson (with [Jackson 2.9+ support for non-blocking parsing](https://github.com/FasterXML/jackson-core/issues/57)). Withing the context of Spring WebFlux, codecs are used to convert the request body into a @RequestMapping parameter, or to convert the return type into the response body that is sent back to the client. The default codecs are configured in the WebFluxConfigurationSupport class, and can easily be changed by overriding the configureHttpMessageCodecs when inheriting from that class. For more information about using codecs in WebFlux, see [this section](https://docs.spring.io/spring/docs/current/spring-framework-reference/web-reactive.html#webflux-codecs).

9. Appendix

9.1. XML Schemas

This part of the appendix lists XML schemas related to the core container.

9.1.1. The util schema

As the name implies, the util tags deal with common, *utility* configuration issues, such as configuring collections, referencing constants, and suchlike. To use the tags in the util schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the snippet below references the correct schema so that the tags in the util namespace are available to you.

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

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

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

*xmlns:util="http://www.springframework.org/schema/util"* xsi:schemaLocation="

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

*http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util.xsd"*> *<!-- bean definitions here -->*

</beans>

<util:constant/>

Before…​

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

<property name="isolation">

<bean id="java.sql.Connection.TRANSACTION\_SERIALIZABLE"

class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean" />

</property>

</bean>

The above configuration uses a Spring FactoryBean implementation, the FieldRetrievingFactoryBean, to set the value of the isolation property on a bean to the value of the java.sql.Connection.TRANSACTION\_SERIALIZABLE constant. This is all well and good, but it is a tad verbose and (unnecessarily) exposes Spring’s internal plumbing to the end user.

The following XML Schema-based version is more concise and clearly expresses the developer’s intent (*'inject this constant value'*), and it just reads better.

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

<property name="isolation">

<util:constant static-field="java.sql.Connection.TRANSACTION\_SERIALIZABLE"/>

</property>

</bean>

Setting a bean property or constructor arg from a field value

[FieldRetrievingFactoryBean](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/beans/factory/config/FieldRetrievingFactoryBean.html) is a FactoryBean which retrieves a static or non-static field value. It is typically used for retrieving public static final constants, which may then be used to set a property value or constructor arg for another bean.

Find below an example which shows how a static field is exposed, by using the [staticField](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/beans/factory/config/FieldRetrievingFactoryBean.html#setStaticField(java.lang.String)) property:

<bean id="myField"

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

<property name="staticField" value="java.sql.Connection.TRANSACTION\_SERIALIZABLE"/>

</bean>

There is also a convenience usage form where the static field is specified as the bean name:

<bean id="java.sql.Connection.TRANSACTION\_SERIALIZABLE"

class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean"/>

This does mean that there is no longer any choice in what the bean id is (so any other bean that refers to it will also have to use this longer name), but this form is very concise to define, and very convenient to use as an inner bean since the id doesn’t have to be specified for the bean reference:

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

<property name="isolation">

<bean id="java.sql.Connection.TRANSACTION\_SERIALIZABLE"

class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean" />

</property>

</bean>

It is also possible to access a non-static (instance) field of another bean, as described in the API documentation for the[FieldRetrievingFactoryBean](https://docs.spring.io/spring-framework/docs/5.0.3.RELEASE/javadoc-api/org/springframework/beans/factory/config/FieldRetrievingFactoryBean.html) class.

Injecting enum values into beans as either property or constructor arguments is very easy to do in Spring, in that you don’t actually have to *do* anything or know anything about the Spring internals (or even about classes such as theFieldRetrievingFactoryBean). Let’s look at an example to see how easy injecting an enum value is; consider this enum:

**package** javax.persistence;

**public** **enum** PersistenceContextType {

TRANSACTION,

EXTENDED

}

Now consider a setter of type PersistenceContextType:

**package** example;

**public** **class** **Client** {

**private** PersistenceContextType persistenceContextType;

**public** **void** setPersistenceContextType(PersistenceContextType type) {

this.persistenceContextType = type;

}

}

1. and the corresponding bean definition:

<bean class="example.Client">

<property name="persistenceContextType" value="TRANSACTION"/>

</bean>

<util:property-path/>

Before…​

*<!-- target bean to be referenced by name -->*

<bean id="testBean" class="org.springframework.beans.TestBean" scope="prototype">

<property name="age" value="10"/>

<property name="spouse">

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

<property name="age" value="11"/>

</bean>

</property>

</bean>

*<!-- will result in 10, which is the value of property 'age' of bean 'testBean' -->*

<bean id="testBean.age" class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>

The above configuration uses a Spring FactoryBean implementation, the PropertyPathFactoryBean, to create a bean (of type int) called testBean.age that has a value equal to the age property of the testBean bean.

After…​

*<!-- target bean to be referenced by name -->*

<bean id="testBean" class="org.springframework.beans.TestBean" scope="prototype">

<property name="age" value="10"/>

<property name="spouse">

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

<property name="age" value="11"/>

</bean>

</property>

</bean>

*<!-- will result in 10, which is the value of property 'age' of bean 'testBean' -->*

<util:property-path id="name" path="testBean.age"/>

The value of the path attribute of the <property-path/> tag follows the form beanName.beanProperty.

Using <util:property-path/> to set a bean property or constructor-argument

PropertyPathFactoryBean is a FactoryBean that evaluates a property path on a given target object. The target object can be specified directly or via a bean name. This value may then be used in another bean definition as a property value or constructor argument.

Here’s an example where a path is used against another bean, by name:

// target bean to be referenced by name

<bean id="person" class="org.springframework.beans.TestBean" scope="prototype">

<property name="age" value="10"/>

<property name="spouse">

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

<property name="age" value="11"/>

</bean>

</property>

</bean>

// will result in 11, which is the value of property 'spouse.age' of bean 'person'

<bean id="theAge"

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

<property name="targetBeanName" value="person"/>

<property name="propertyPath" value="spouse.age"/>

</bean>

In this example, a path is evaluated against an inner bean:

*<!-- will result in 12, which is the value of property 'age' of the inner bean -->*

<bean id="theAge"

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

<property name="targetObject">

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

<property name="age" value="12"/>

</bean>

</property>

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

</bean>

There is also a shortcut form, where the bean name is the property path.

*<!-- will result in 10, which is the value of property 'age' of bean 'person' -->*

<bean id="person.age"

class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>

This form does mean that there is no choice in the name of the bean. Any reference to it will also have to use the same id, which is the path. Of course, if used as an inner bean, there is no need to refer to it at all:

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

<property name="age">

<bean id="person.age"

class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>

</property>

</bean>

The result type may be specifically set in the actual definition. This is not necessary for most use cases, but can be of use for some. Please see the Javadocs for more info on this feature.

<util:properties/>

Before…​

*<!-- creates a java.util.Properties instance with values loaded from the supplied location -->*

<bean id="jdbcConfiguration" class="org.springframework.beans.factory.config.PropertiesFactoryBean">

<property name="location" value="classpath:com/foo/jdbc-production.properties"/>

</bean>

The above configuration uses a Spring FactoryBean implementation, the PropertiesFactoryBean, to instantiate a java.util.Properties instance with values loaded from the supplied [Resource](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#resources) location).

After…​

*<!-- creates a java.util.Properties instance with values loaded from the supplied location -->*

<util:properties id="jdbcConfiguration" location="classpath:com/foo/jdbc-production.properties"/>

<util:list/>

Before…​

*<!-- creates a java.util.List instance with values loaded from the supplied 'sourceList' -->*

<bean id="emails" class="org.springframework.beans.factory.config.ListFactoryBean">

<property name="sourceList">

<list>

<value>pechorin@hero.org</value>

<value>raskolnikov@slums.org</value>

<value>stavrogin@gov.org</value>

<value>porfiry@gov.org</value>

</list>

</property>

</bean>

The above configuration uses a Spring FactoryBean implementation, the ListFactoryBean, to create a java.util.List instance initialized with values taken from the supplied sourceList.

After…​

*<!-- creates a java.util.List instance with the supplied values -->*

<util:list id="emails">

<value>pechorin@hero.org</value>

<value>raskolnikov@slums.org</value>

<value>stavrogin@gov.org</value>

<value>porfiry@gov.org</value>

</util:list>

You can also explicitly control the exact type of List that will be instantiated and populated via the use of the list-classattribute on the <util:list/> element. For example, if we really need a java.util.LinkedList to be instantiated, we could use the following configuration:

<util:list id="emails" list-class="java.util.LinkedList">

<value>jackshaftoe@vagabond.org</value>

<value>eliza@thinkingmanscrumpet.org</value>

<value>vanhoek@pirate.org</value>

<value>d'Arcachon@nemesis.org</value>

</util:list>

If no list-class attribute is supplied, a List implementation will be chosen by the container.

<util:map/>

Before…​

*<!-- creates a java.util.Map instance with values loaded from the supplied 'sourceMap' -->*

<bean id="emails" class="org.springframework.beans.factory.config.MapFactoryBean">

<property name="sourceMap">

<map>

<entry key="pechorin" value="pechorin@hero.org"/>

<entry key="raskolnikov" value="raskolnikov@slums.org"/>

<entry key="stavrogin" value="stavrogin@gov.org"/>

<entry key="porfiry" value="porfiry@gov.org"/>

</map>

</property>

</bean>

The above configuration uses a Spring FactoryBean implementation, the MapFactoryBean, to create a java.util.Map instance initialized with key-value pairs taken from the supplied 'sourceMap'.

After…​

*<!-- creates a java.util.Map instance with the supplied key-value pairs -->*

<util:map id="emails">

<entry key="pechorin" value="pechorin@hero.org"/>

<entry key="raskolnikov" value="raskolnikov@slums.org"/>

<entry key="stavrogin" value="stavrogin@gov.org"/>

<entry key="porfiry" value="porfiry@gov.org"/>

</util:map>

You can also explicitly control the exact type of Map that will be instantiated and populated via the use of the 'map-class'attribute on the <util:map/> element. For example, if we really need a java.util.TreeMap to be instantiated, we could use the following configuration:

<util:map id="emails" map-class="java.util.TreeMap">

<entry key="pechorin" value="pechorin@hero.org"/>

<entry key="raskolnikov" value="raskolnikov@slums.org"/>

<entry key="stavrogin" value="stavrogin@gov.org"/>

<entry key="porfiry" value="porfiry@gov.org"/>

</util:map>

If no 'map-class' attribute is supplied, a Map implementation will be chosen by the container.

<util:set/>

Before…​

*<!-- creates a java.util.Set instance with values loaded from the supplied 'sourceSet' -->*

<bean id="emails" class="org.springframework.beans.factory.config.SetFactoryBean">

<property name="sourceSet">

<set>

<value>pechorin@hero.org</value>

<value>raskolnikov@slums.org</value>

<value>stavrogin@gov.org</value>

<value>porfiry@gov.org</value>

</set>

</property>

</bean>

The above configuration uses a Spring FactoryBean implementation, the SetFactoryBean, to create a java.util.Set instance initialized with values taken from the supplied 'sourceSet'.

After…​

*<!-- creates a java.util.Set instance with the supplied values -->*

<util:set id="emails">

<value>pechorin@hero.org</value>

<value>raskolnikov@slums.org</value>

<value>stavrogin@gov.org</value>

<value>porfiry@gov.org</value>

</util:set>

You can also explicitly control the exact type of Set that will be instantiated and populated via the use of the 'set-class'attribute on the <util:set/> element. For example, if we really need a java.util.TreeSet to be instantiated, we could use the following configuration:

<util:set id="emails" set-class="java.util.TreeSet">

<value>pechorin@hero.org</value>

<value>raskolnikov@slums.org</value>

<value>stavrogin@gov.org</value>

<value>porfiry@gov.org</value>

</util:set>

If no 'set-class' attribute is supplied, a Set implementation will be chosen by the container.

9.1.2. The aop schema

The aop tags deal with configuring all things AOP in Spring: this includes Spring’s own proxy-based AOP framework and Spring’s integration with the AspectJ AOP framework. These tags are comprehensively covered in the chapter entitled [Aspect Oriented Programming with Spring](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop).

In the interest of completeness, to use the tags in the aop schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the following snippet references the correct schema so that the tags in the aopnamespace are available to you.

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

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

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

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

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

*http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd"*> *<!-- bean definitions here -->*

</beans>

9.1.3. The context schema

The context tags deal with ApplicationContext configuration that relates to plumbing - that is, not usually beans that are important to an end-user but rather beans that do a lot of grunt work in Spring, such as BeanfactoryPostProcessors. The following snippet references the correct schema so that the tags in the context namespace are available to you.

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

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

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

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

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

*http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd"*> *<!-- bean definitions here -->*

</beans>

<property-placeholder/>

This element activates the replacement of ${…​} placeholders, resolved against the specified properties file (as a [Spring resource location](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#resources)). This element is a convenience mechanism that sets up a[PropertyPlaceholderConfigurer](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-placeholderconfigurer) for you; if you need more control over the PropertyPlaceholderConfigurer, just define one yourself explicitly.

<annotation-config/>

Activates the Spring infrastructure for various annotations to be detected in bean classes: Spring’s [@Required](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-required-annotation) and [@Autowired](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-annotation-config), as well as JSR 250’s @PostConstruct, @PreDestroy and @Resource (if available), and JPA’s @PersistenceContext and@PersistenceUnit (if available). Alternatively, you can choose to activate the individual BeanPostProcessors for those annotations explicitly.

|  |  |
| --- | --- |
|  | This element does *not* activate processing of Spring’s [@Transactional](https://docs.spring.io/spring/docs/current/spring-framework-reference/data-access.html#transaction-declarative-annotations) annotation. Use the [<tx:annotation-driven/>](https://docs.spring.io/spring/docs/current/spring-framework-reference/data-access.html#tx-decl-explained) element for that purpose. |

<component-scan/>

This element is detailed in [Annotation-based container configuration](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-annotation-config).

<load-time-weaver/>

This element is detailed in [Load-time weaving with AspectJ in the Spring Framework](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-aj-ltw).

<spring-configured/>

This element is detailed in [Using AspectJ to dependency inject domain objects with Spring](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-atconfigurable).

<mbean-export/>

This element is detailed in [Configuring annotation based MBean export](https://docs.spring.io/spring/docs/current/spring-framework-reference/integration.html#jmx-context-mbeanexport).

9.1.4. The beans schema

Last but not least we have the tags in the beans schema. These are the same tags that have been in Spring since the very dawn of the framework. Examples of the various tags in the beans schema are not shown here because they are quite comprehensively covered in [Dependencies and configuration in detail](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-properties-detailed) (and indeed in that entire [chapter](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans)).

Note that it is possible to add zero or more key / value pairs to <bean/> XML definitions. What, if anything, is done with this extra metadata is totally up to your own custom logic (and so is typically only of use if you are writing your own custom tags as described in the appendix entitled [XML Schema Authoring](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#xml-custom)).

Find below an example of the <meta/> tag in the context of a surrounding <bean/> (please note that without any logic to interpret it the metadata is effectively useless as-is).

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

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

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

xsi:schemaLocation="

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

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

*<meta key="cacheName" value="foo"/>*

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

</bean>

</beans>

In the case of the above example, you would assume that there is some logic that will consume the bean definition and set up some caching infrastructure using the supplied metadata.

9.2. XML Schema Authoring

9.2.1. Introduction

Since version 2.0, Spring has featured a mechanism for schema-based extensions to the basic Spring XML format for defining and configuring beans. This section is devoted to detailing how you would go about writing your own custom XML bean definition parsers and integrating such parsers into the Spring IoC container.

To facilitate the authoring of configuration files using a schema-aware XML editor, Spring’s extensible XML configuration mechanism is based on XML Schema. If you are not familiar with Spring’s current XML configuration extensions that come with the standard Spring distribution, please first read the appendix entitled[[xsd-config]](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#xsd-config).

Creating new XML configuration extensions can be done by following these (relatively) simple steps:

* [Authoring](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#xsd-custom-schema) an XML schema to describe your custom element(s).
* [Coding](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#xsd-custom-namespacehandler) a custom NamespaceHandler implementation (this is an easy step, don’t worry).
* [Coding](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#xsd-custom-parser) one or more BeanDefinitionParser implementations (this is where the real work is done).
* [Registering](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#xsd-custom-registration) the above artifacts with Spring (this too is an easy step).

What follows is a description of each of these steps. For the example, we will create an XML extension (a custom XML element) that allows us to configure objects of the type SimpleDateFormat (from the java.text package) in an easy manner. When we are done, we will be able to define bean definitions of type SimpleDateFormat like this:

<myns:dateformat id="dateFormat"

pattern="yyyy-MM-dd HH:mm"

lenient="true"/>

*(Don’t worry about the fact that this example is very simple; much more detailed examples follow afterwards. The intent in this first simple example is to walk you through the basic steps involved.)*

9.2.2. Authoring the schema

Creating an XML configuration extension for use with Spring’s IoC container starts with authoring an XML Schema to describe the extension. What follows is the schema we’ll use to configure SimpleDateFormat objects.

*<!-- myns.xsd (inside package org/springframework/samples/xml) -->*

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

<xsd:schema xmlns="http://www.mycompany.com/schema/myns"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

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

targetNamespace="http://www.mycompany.com/schema/myns"

elementFormDefault="qualified"

attributeFormDefault="unqualified">

<xsd:import namespace="http://www.springframework.org/schema/beans"/>

<xsd:element name="dateformat">

<xsd:complexType>

<xsd:complexContent>

<xsd:extension base="beans:identifiedType">

<xsd:attribute name="lenient" type="xsd:boolean"/>

<xsd:attribute name="pattern" type="xsd:string" use="required"/>

</xsd:extension>

</xsd:complexContent>

</xsd:complexType>

</xsd:element>

</xsd:schema>

(The emphasized line contains an extension base for all tags that will be identifiable (meaning they have an id attribute that will be used as the bean identifier in the container). We are able to use this attribute because we imported the Spring-provided'beans' namespace.)

The above schema will be used to configure SimpleDateFormat objects, directly in an XML application context file using the <myns:dateformat/> element.

<myns:dateformat id="dateFormat"

pattern="yyyy-MM-dd HH:mm"

lenient="true"/>

Note that after we’ve created the infrastructure classes, the above snippet of XML will essentially be exactly the same as the following XML snippet. In other words, we’re just creating a bean in the container, identified by the name 'dateFormat' of typeSimpleDateFormat, with a couple of properties set.

<bean id="dateFormat" class="java.text.SimpleDateFormat">

<constructor-arg value="yyyy-HH-dd HH:mm"/>

<property name="lenient" value="true"/>

</bean>

|  |  |
| --- | --- |
|  | The schema-based approach to creating configuration format allows for tight integration with an IDE that has a schema-aware XML editor. Using a properly authored schema, you can use autocompletion to have a user choose between several configuration options defined in the enumeration. |

9.2.3. Coding a NamespaceHandler

In addition to the schema, we need a NamespaceHandler that will parse all elements of this specific namespace Spring encounters while parsing configuration files. The NamespaceHandler should in our case take care of the parsing of the myns:dateformat element.

The NamespaceHandler interface is pretty simple in that it features just three methods:

* init() - allows for initialization of the NamespaceHandler and will be called by Spring before the handler is used
* BeanDefinition parse(Element, ParserContext) - called when Spring encounters a top-level element (not nested inside a bean definition or a different namespace). This method can register bean definitions itself and/or return a bean definition.
* BeanDefinitionHolder decorate(Node, BeanDefinitionHolder, ParserContext) - called when Spring encounters an attribute or nested element of a different namespace. The decoration of one or more bean definitions is used for example with the[out-of-the-box scopes Spring supports](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-scopes). We’ll start by highlighting a simple example, without using decoration, after which we will show decoration in a somewhat more advanced example.

Although it is perfectly possible to code your own NamespaceHandler for the entire namespace (and hence provide code that parses each and every element in the namespace), it is often the case that each top-level XML element in a Spring XML configuration file results in a single bean definition (as in our case, where a single <myns:dateformat/> element results in a single SimpleDateFormat bean definition). Spring features a number of convenience classes that support this scenario. In this example, we’ll make use the NamespaceHandlerSupport class:

**package** org.springframework.samples.xml;

**import** org.springframework.beans.factory.xml.NamespaceHandlerSupport;

**public** **class** **MyNamespaceHandler** **extends** NamespaceHandlerSupport {

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

**registerBeanDefinitionParser("dateformat", new SimpleDateFormatBeanDefinitionParser());**

}

}

The observant reader will notice that there isn’t actually a whole lot of parsing logic in this class. Indeed…​ the NamespaceHandlerSupport class has a built in notion of delegation. It supports the registration of any number of BeanDefinitionParser instances, to which it will delegate to when it needs to parse an element in its namespace. This clean separation of concerns allows a NamespaceHandler to handle the orchestration of the parsing of *all* of the custom elements in its namespace, while delegating to BeanDefinitionParsers to do the grunt work of the XML parsing; this means that each BeanDefinitionParser will contain just the logic for parsing a single custom element, as we can see in the next step

9.2.4. BeanDefinitionParser

A BeanDefinitionParser will be used if the NamespaceHandler encounters an XML element of the type that has been mapped to the specific bean definition parser (which is 'dateformat' in this case). In other words, the BeanDefinitionParser is responsible for parsing *one* distinct top-level XML element defined in the schema. In the parser, we’ll have access to the XML element (and thus its subelements too) so that we can parse our custom XML content, as can be seen in the following example:

**package** org.springframework.samples.xml;

**import** org.springframework.beans.factory.support.BeanDefinitionBuilder;

**import** org.springframework.beans.factory.xml.AbstractSingleBeanDefinitionParser;

**import** org.springframework.util.StringUtils;

**import** org.w3c.dom.Element;

**import** java.text.SimpleDateFormat;

**public** **class** **SimpleDateFormatBeanDefinitionParser** **extends** AbstractSingleBeanDefinitionParser {

**protected** Class getBeanClass(Element element) {

**return** SimpleDateFormat.class;

}

**protected** **void** doParse(Element element, BeanDefinitionBuilder bean) {

*// this will never be null since the schema explicitly requires that a value be supplied*

String pattern = element.getAttribute("pattern");

bean.addConstructorArg(pattern);

*// this however is an optional property*

String lenient = element.getAttribute("lenient");

**if** (StringUtils.hasText(lenient)) {

bean.addPropertyValue("lenient", Boolean.valueOf(lenient));

}

}

}

|  |  |
| --- | --- |
|  | We use the Spring-provided AbstractSingleBeanDefinitionParser to handle a lot of the basic grunt work of creating a *single* BeanDefinition. |
|  | We supply the AbstractSingleBeanDefinitionParser superclass with the type that our single BeanDefinition will represent. |

In this simple case, this is all that we need to do. The creation of our single BeanDefinition is handled by the AbstractSingleBeanDefinitionParser superclass, as is the extraction and setting of the bean definition’s unique identifier.

9.2.5. Registering the handler and the schema

The coding is finished! All that remains to be done is to somehow make the Spring XML parsing infrastructure aware of our custom element; we do this by registering our custom namespaceHandler and custom XSD file in two special purpose properties files. These properties files are both placed in a 'META-INF' directory in your application, and can, for example, be distributed alongside your binary classes in a JAR file. The Spring XML parsing infrastructure will automatically pick up your new extension by consuming these special properties files, the formats of which are detailed below.

'META-INF/spring.handlers'

The properties file called 'spring.handlers' contains a mapping of XML Schema URIs to namespace handler classes. So for our example, we need to write the following:

http\://www.mycompany.com/schema/myns=org.springframework.samples.xml.MyNamespaceHandler

*(The ':' character is a valid delimiter in the Java properties format, and so the ':' character in the URI needs to be escaped with a backslash.)*

The first part (the key) of the key-value pair is the URI associated with your custom namespace extension, and needs to *match exactly* the value of the 'targetNamespace' attribute as specified in your custom XSD schema.

'META-INF/spring.schemas'

The properties file called 'spring.schemas' contains a mapping of XML Schema locations (referred to along with the schema declaration in XML files that use the schema as part of the 'xsi:schemaLocation' attribute) to *classpath* resources. This file is needed to prevent Spring from absolutely having to use a default EntityResolver that requires Internet access to retrieve the schema file. If you specify the mapping in this properties file, Spring will search for the schema on the classpath (in this case'myns.xsd' in the 'org.springframework.samples.xml' package):

http\://www.mycompany.com/schema/myns/myns.xsd=org/springframework/samples/xml/myns.xsd

The upshot of this is that you are encouraged to deploy your XSD file(s) right alongside the NamespaceHandler and BeanDefinitionParser classes on the classpath.

9.2.6. Using a custom extension in your Spring XML configuration

Using a custom extension that you yourself have implemented is no different from using one of the 'custom' extensions that Spring provides straight out of the box. Find below an example of using the custom <dateformat/> element developed in the previous steps in a Spring XML configuration file.

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

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

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

xmlns:myns="http://www.mycompany.com/schema/myns"

xsi:schemaLocation="

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

http://www.mycompany.com/schema/myns http://www.mycompany.com/schema/myns/myns.xsd">

*<!-- as a top-level bean -->*

<myns:dateformat id="defaultDateFormat" pattern="yyyy-MM-dd HH:mm" lenient="true"/>

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

<property name="dateFormat">

*<!-- as an inner bean -->*

<myns:dateformat pattern="HH:mm MM-dd-yyyy"/>

</property>

</bean>

</beans>

9.2.7. Meatier examples

Find below some much meatier examples of custom XML extensions.

Nesting custom tags within custom tags

This example illustrates how you might go about writing the various artifacts required to satisfy a target of the following configuration:

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

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

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

xmlns:foo="http://www.foo.com/schema/component"

xsi:schemaLocation="

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

http://www.foo.com/schema/component http://www.foo.com/schema/component/component.xsd">

<foo:component id="bionic-family" name="Bionic-1">

<foo:component name="Mother-1">

<foo:component name="Karate-1"/>

<foo:component name="Sport-1"/>

</foo:component>

<foo:component name="Rock-1"/>

</foo:component>

</beans>

The above configuration actually nests custom extensions within each other. The class that is actually configured by the above <foo:component/> element is the Component class (shown directly below). Notice how the Component class does *not* expose a setter method for the 'components' property; this makes it hard (or rather impossible) to configure a bean definition for the Component class using setter injection.

**package** com.foo;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** **Component** {

**private** String name;

**private** List<Component> components = **new** ArrayList<Component> ();

*// mmm, there is no setter method for the 'components'*

**public** **void** addComponent(Component component) {

this.components.add(component);

}

**public** List<Component> getComponents() {

**return** components;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

this.name = name;

}

}

The typical solution to this issue is to create a custom FactoryBean that exposes a setter property for the 'components'property.

**package** com.foo;

**import** org.springframework.beans.factory.FactoryBean;

**import** java.util.List;

**public** **class** **ComponentFactoryBean** **implements** FactoryBean<Component> {

**private** Component parent;

**private** List<Component> children;

**public** **void** setParent(Component parent) {

this.parent = parent;

}

**public** **void** setChildren(List<Component> children) {

this.children = children;

}

**public** Component getObject() **throws** Exception {

**if** (this.children != null && this.children.size() > 0) {

**for** (Component child : children) {

this.parent.addComponent(child);

}

}

**return** this.parent;

}

**public** Class<Component> getObjectType() {

**return** Component.class;

}

**public** **boolean** isSingleton() {

**return** true;

}

}

This is all very well, and does work nicely, but exposes a lot of Spring plumbing to the end user. What we are going to do is write a custom extension that hides away all of this Spring plumbing. If we stick to [the steps described previously](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#xsd-custom-introduction), we’ll start off by creating the XSD schema to define the structure of our custom tag.

<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<xsd:schema xmlns="http://www.foo.com/schema/component"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

targetNamespace="http://www.foo.com/schema/component"

elementFormDefault="qualified"

attributeFormDefault="unqualified">

<xsd:element name="component">

<xsd:complexType>

<xsd:choice minOccurs="0" maxOccurs="unbounded">

<xsd:element ref="component"/>

</xsd:choice>

<xsd:attribute name="id" type="xsd:ID"/>

<xsd:attribute name="name" use="required" type="xsd:string"/>

</xsd:complexType>

</xsd:element>

</xsd:schema>

We’ll then create a custom NamespaceHandler.

**package** com.foo;

**import** org.springframework.beans.factory.xml.NamespaceHandlerSupport;

**public** **class** **ComponentNamespaceHandler** **extends** NamespaceHandlerSupport {

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

registerBeanDefinitionParser("component", **new** ComponentBeanDefinitionParser());

}

}

Next up is the custom BeanDefinitionParser. Remember that what we are creating is a BeanDefinition describing a ComponentFactoryBean.

**package** com.foo;

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

**import** org.springframework.beans.factory.support.AbstractBeanDefinition;

**import** org.springframework.beans.factory.support.BeanDefinitionBuilder;

**import** org.springframework.beans.factory.support.ManagedList;

**import** org.springframework.beans.factory.xml.AbstractBeanDefinitionParser;

**import** org.springframework.beans.factory.xml.ParserContext;

**import** org.springframework.util.xml.DomUtils;

**import** org.w3c.dom.Element;

**import** java.util.List;

**public** **class** **ComponentBeanDefinitionParser** **extends** AbstractBeanDefinitionParser {

**protected** AbstractBeanDefinition parseInternal(Element element, ParserContext parserContext) {

**return** parseComponentElement(element);

}

**private** **static** AbstractBeanDefinition parseComponentElement(Element element) {

BeanDefinitionBuilder factory = BeanDefinitionBuilder.rootBeanDefinition(ComponentFactoryBean.class);

factory.addPropertyValue("parent", parseComponent(element));

List<Element> childElements = DomUtils.getChildElementsByTagName(element, "component");

**if** (childElements != null && childElements.size() > 0) {

parseChildComponents(childElements, factory);

}

**return** factory.getBeanDefinition();

}

**private** **static** BeanDefinition parseComponent(Element element) {

BeanDefinitionBuilder component = BeanDefinitionBuilder.rootBeanDefinition(Component.class);

component.addPropertyValue("name", element.getAttribute("name"));

**return** component.getBeanDefinition();

}

**private** **static** **void** parseChildComponents(List<Element> childElements, BeanDefinitionBuilder factory) {

ManagedList<BeanDefinition> children = **new** ManagedList<BeanDefinition>(childElements.size());

**for** (Element element : childElements) {

children.add(parseComponentElement(element));

}

factory.addPropertyValue("children", children);

}

}

Lastly, the various artifacts need to be registered with the Spring XML infrastructure.

# in 'META-INF/spring.handlers'

http\://www.foo.com/schema/component=com.foo.ComponentNamespaceHandler

# in 'META-INF/spring.schemas'

http\://www.foo.com/schema/component/component.xsd=com/foo/component.xsd

Custom attributes on 'normal' elements

Writing your own custom parser and the associated artifacts isn’t hard, but sometimes it is not the right thing to do. Consider the scenario where you need to add metadata to already existing bean definitions. In this case you certainly don’t want to have to go off and write your own entire custom extension; rather you just want to add an additional attribute to the existing bean definition element.

By way of another example, let’s say that the service class that you are defining a bean definition for a service object that will (unknown to it) be accessing a clustered [JCache](https://jcp.org/en/jsr/detail?id=107), and you want to ensure that the named JCache instance is eagerly started within the surrounding cluster:

<bean id="checkingAccountService" class="com.foo.DefaultCheckingAccountService"

jcache:cache-name="checking.account">

*<!-- other dependencies here... -->*

</bean>

What we are going to do here is create another BeanDefinition when the 'jcache:cache-name' attribute is parsed; this BeanDefinition will then initialize the named JCache for us. We will also modify the existing BeanDefinition for the'checkingAccountService' so that it will have a dependency on this new JCache-initializing BeanDefinition.

**package** com.foo;

**public** **class** **JCacheInitializer** {

**private** String name;

**public** JCacheInitializer(String name) {

this.name = name;

}

**public** **void** initialize() {

*// lots of JCache API calls to initialize the named cache...*

}

}

Now onto the custom extension. Firstly, the authoring of the XSD schema describing the custom attribute (quite easy in this case).

<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<xsd:schema xmlns="http://www.foo.com/schema/jcache"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

targetNamespace="http://www.foo.com/schema/jcache"

elementFormDefault="qualified">

<xsd:attribute name="cache-name" type="xsd:string"/>

</xsd:schema>

Next, the associated NamespaceHandler.

**package** com.foo;

**import** org.springframework.beans.factory.xml.NamespaceHandlerSupport;

**public** **class** **JCacheNamespaceHandler** **extends** NamespaceHandlerSupport {

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

super.registerBeanDefinitionDecoratorForAttribute("cache-name",

**new** JCacheInitializingBeanDefinitionDecorator());

}

}

Next, the parser. Note that in this case, because we are going to be parsing an XML attribute, we write a BeanDefinitionDecorator rather than a BeanDefinitionParser.

**package** com.foo;

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

**import** org.springframework.beans.factory.support.AbstractBeanDefinition;

**import** org.springframework.beans.factory.support.BeanDefinitionBuilder;

**import** org.springframework.beans.factory.xml.BeanDefinitionDecorator;

**import** org.springframework.beans.factory.xml.ParserContext;

**import** org.w3c.dom.Attr;

**import** org.w3c.dom.Node;

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.List;

**public** **class** **JCacheInitializingBeanDefinitionDecorator** **implements** BeanDefinitionDecorator {

**private** **static** **final** String**[]** EMPTY\_STRING\_ARRAY = **new** String[0];

**public** BeanDefinitionHolder decorate(Node source, BeanDefinitionHolder holder,

ParserContext ctx) {

String initializerBeanName = registerJCacheInitializer(source, ctx);

createDependencyOnJCacheInitializer(holder, initializerBeanName);

**return** holder;

}

**private** **void** createDependencyOnJCacheInitializer(BeanDefinitionHolder holder,

String initializerBeanName) {

AbstractBeanDefinition definition = ((AbstractBeanDefinition) holder.getBeanDefinition());

String**[]** dependsOn = definition.getDependsOn();

**if** (dependsOn == null) {

dependsOn = **new** String**[]**{initializerBeanName};

} **else** {

List dependencies = **new** ArrayList(Arrays.asList(dependsOn));

dependencies.add(initializerBeanName);

dependsOn = (String**[]**) dependencies.toArray(EMPTY\_STRING\_ARRAY);

}

definition.setDependsOn(dependsOn);

}

**private** String registerJCacheInitializer(Node source, ParserContext ctx) {

String cacheName = ((Attr) source).getValue();

String beanName = cacheName + "-initializer";

**if** (!ctx.getRegistry().containsBeanDefinition(beanName)) {

BeanDefinitionBuilder initializer = BeanDefinitionBuilder.rootBeanDefinition(JCacheInitializer.class);

initializer.addConstructorArg(cacheName);

ctx.getRegistry().registerBeanDefinition(beanName, initializer.getBeanDefinition());

}

**return** beanName;

}

}

Lastly, the various artifacts need to be registered with the Spring XML infrastructure.

# in 'META-INF/spring.handlers'

http\://www.foo.com/schema/jcache=com.foo.JCacheNamespaceHandler

# in 'META-INF/spring.schemas'

http\://www.foo.com/schema/jcache/jcache.xsd=com/foo/jcache.xsd

[**1**](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#_footnoteref_1). See [Inversion of Control](https://docs.spring.io/spring/docs/current/spring-framework-reference/overview.html#background-ioc)

[**2**](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#_footnoteref_2). See [Dependency Injection](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-collaborators)

[**3**](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#_footnoteref_3). But see also [FileSystemResource caveats](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#resources-filesystemresource-caveats).

Version 5.0.3.RELEASE  
Last updated 2018-01-23 09:11:25 UTC