Core Technologies

Version 5.1.8.RELEASE

目录

[Core Technologies 1](#_Toc15628307)

[1. The IoC Container 3](#_Toc15628308)

[1.1. Introduction to the Spring IoC Container and Beans 3](#_Toc15628309)

[1.2. Container Overview 5](#_Toc15628310)

[1.3. Bean Overview 13](#_Toc15628311)

[1.4. Dependencies 23](#_Toc15628312)

[1.5. Bean Scopes 58](#_Toc15628313)

[1.6. Customizing the Nature of a Bean 74](#_Toc15628314)

[1.7. Bean Definition Inheritance 88](#_Toc15628315)

[1.8. Container Extension Points 90](#_Toc15628316)

[1.9. Annotation-based Container Configuration 102](#_Toc15628317)

[1.10. Classpath Scanning and Managed Components 126](#_Toc15628318)

[1.11. Using JSR 330 Standard Annotations 142](#_Toc15628319)

[1.12. Java-based Container Configuration 147](#_Toc15628320)

[1.13. Environment Abstraction 176](#_Toc15628321)

[1.14. Registering a LoadTimeWeaver 189](#_Toc15628322)

[1.15. Additional Capabilities of the ApplicationContext 190](#_Toc15628323)

[1.16. The BeanFactory 207](#_Toc15628324)

[2. Resources 210](#_Toc15628325)

[2.1. Introduction 210](#_Toc15628326)

[2.2. The Resource Interface 211](#_Toc15628327)

[2.3. Built-in Resource Implementations 212](#_Toc15628328)

[2.4. The ResourceLoader 214](#_Toc15628329)

[2.5. The ResourceLoaderAware interface 215](#_Toc15628330)

[2.6. Resources as Dependencies 215](#_Toc15628331)

[2.7. Application Contexts and Resource Paths 216](#_Toc15628332)

[3. Validation, Data Binding, and Type Conversion 221](#_Toc15628333)

[3.1. Validation by Using Spring’s Validator Interface 222](#_Toc15628334)

[3.2. Resolving Codes to Error Messages 224](#_Toc15628335)

[3.3. Bean Manipulation and the BeanWrapper 224](#_Toc15628336)

[3.4. Spring Type Conversion 232](#_Toc15628337)

[3.5. Spring Field Formatting 238](#_Toc15628338)

[3.6. Configuring a Global Date and Time Format 243](#_Toc15628339)

[3.7. Spring Validation 245](#_Toc15628340)

[4. Spring Expression Language (SpEL) 249](#_Toc15628341)

[4.1. Evaluation 250](#_Toc15628342)

[4.2. Expressions in Bean Definitions 255](#_Toc15628343)

[4.3. Language Reference 258](#_Toc15628344)

[4.4. Classes Used in the Examples 272](#_Toc15628345)

[5. Aspect Oriented Programming with Spring 275](#_Toc15628346)

[5.1. AOP Concepts 276](#_Toc15628347)

[5.2. Spring AOP Capabilities and Goals 277](#_Toc15628348)

[5.3. AOP Proxies 278](#_Toc15628349)

[5.4. @AspectJ support 278](#_Toc15628350)

[5.5. Schema-based AOP Support 299](#_Toc15628351)

[5.6. Choosing which AOP Declaration Style to Use 312](#_Toc15628352)

[5.7. Mixing Aspect Types 314](#_Toc15628353)

[5.8. Proxying Mechanisms 314](#_Toc15628354)

[5.9. Programmatic Creation of @AspectJ Proxies 317](#_Toc15628355)

[5.10. Using AspectJ with Spring Applications 318](#_Toc15628356)

[5.11. Further Resources 332](#_Toc15628357)

[6. Spring AOP APIs 332](#_Toc15628358)

[6.1. Pointcut API in Spring 332](#_Toc15628359)

[6.2. Advice API in Spring 336](#_Toc15628360)

[6.3. The Advisor API in Spring 343](#_Toc15628361)

[6.4. Using the ProxyFactoryBean to Create AOP Proxies 343](#_Toc15628362)

[6.5. Concise Proxy Definitions 348](#_Toc15628363)

[6.6. Creating AOP Proxies Programmatically with the ProxyFactory 350](#_Toc15628364)

[6.7. Manipulating Advised Objects 350](#_Toc15628365)

[6.8. Using the "auto-proxy" facility 352](#_Toc15628366)

[6.9. Using TargetSource Implementations 354](#_Toc15628367)

[6.10. Defining New Advice Types 358](#_Toc15628368)

[7. Null-safety 358](#_Toc15628369)

[7.1. Use cases 358](#_Toc15628370)

[7.2. JSR-305 meta-annotations 359](#_Toc15628371)

[8. Data Buffers and Codecs 359](#_Toc15628372)

[8.1. DataBufferFactory 359](#_Toc15628373)

[8.2. DataBuffer 360](#_Toc15628374)

[8.3. PooledDataBuffer 360](#_Toc15628375)

[8.4. DataBufferUtils 360](#_Toc15628376)

[8.5. Codecs 360](#_Toc15628377)

[8.6. Using DataBuffer 361](#_Toc15628378)

[9. Appendix 362](#_Toc15628379)

[9.1. XML Schemas 362](#_Toc15628380)

[9.2. XML Schema Authoring 373](#_Toc15628381)

**错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。错误!超链接引用无效。**

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

这部分参考文档涵盖了Spring Framework绝对不可或缺的所有技术。

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

其中最重要的是Spring Framework的控制反转（IoC）容器。Spring框架的IoC容器的全面处理紧随其后，全面覆盖了Spring的面向方面编程（Aspect-Oriented Programming ：AOP）技术。Spring Framework有自己的AOP框架，它在概念上易于理解，并且成功地解决了Java企业编程中AOP要求的80％最佳点。

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

还提供了覆盖Spring与AspectJ的集成（目前是Java企业领域中，在功能方面最丰富、最成熟的AOP实现）。

1. The IoC Container

This chapter covers Spring’s Inversion of Control (IoC) container.

本章介绍Spring的控制反转（IoC）容器。

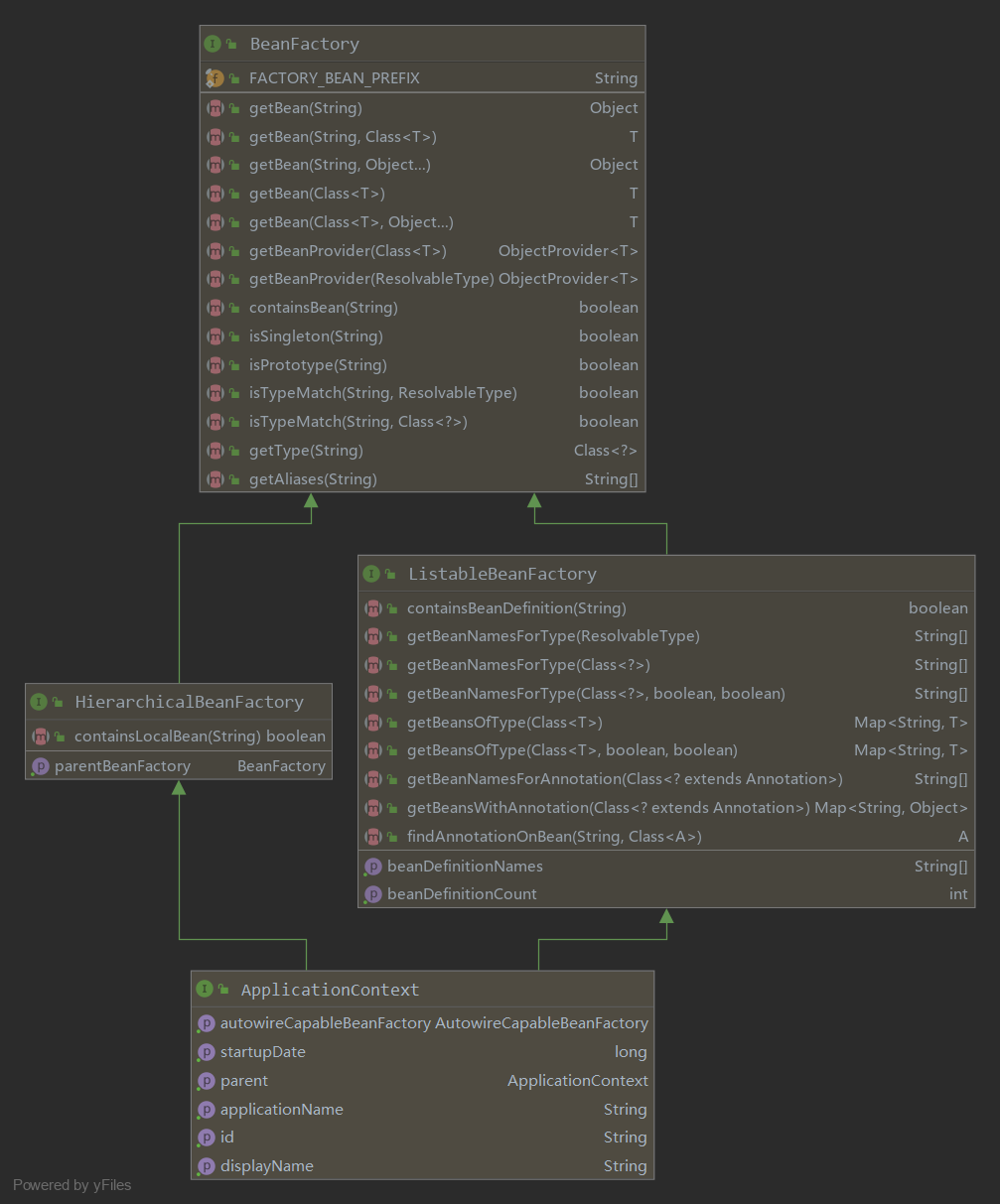
1.1. Introduction to the Spring IoC Container and Beans

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

本章介绍了控制反转（Inversion of Control：IoC）原理的Spring Framework实现。IoC也称为依赖注入（Dependency Injection：DI），这是一个过程，通过这个过程，对象只能通过构造函数参数、工厂方法的参数来定义他们的依赖关系，或在从构造函数、工厂方法返回后的对象实例上设置的属性来定义它们的依赖关系（即，它们使用的其他对象）。然后容器在创建bean时注入这些依赖项。此过程基本上是bean自身通过使用类的直接构造函数或诸如服务定位器模式的机制来控制其依赖关系的实例化或位置的逆过程（因此命名控制反转）。

The org.springframework.beans and org.springframework.context packages are the basis for Spring Framework’s IoC container. The [BeanFactory](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/beans/factory/BeanFactory.html) interface provides an advanced configuration mechanism capable of managing any type of object.[ApplicationContext](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/context/ApplicationContext.html) is a sub-interface of BeanFactory. It adds:

org.springframework.beans和org.springframework.context包是Spring框架的IoC容器的基础。[BeanFactory](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/beans/factory/BeanFactory.html) 接口提供了一种能够管理任何类型对象的高级配置机制。 [ApplicationContext](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/context/ApplicationContext.html) 是BeanFactory的一个子接口。它包括：



* Easier integration with Spring’s AOP features
* Message resource handling (for use in internationalization)
* Event publication
* Application-layer specific contexts such as the WebApplicationContext for use in web applications.
* 更容易与Spring的AOP功能集成
* 消息资源处理（用于国际化）
* 事件发布
* 应用层特定的上下文，例如用于Web应用程序中的WebApplicationContext 。

In short, the BeanFactory provides the configuration framework and basic functionality, and the ApplicationContext adds more enterprise-specific functionality. The ApplicationContext is a complete superset of the BeanFactory and is used exclusively in this chapter in descriptions of Spring’s IoC container. For more information on using the BeanFactory instead of the ApplicationContext, see [The BeanFactory](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-beanfactory).

简而言之，BeanFactory提供了配置框架和基本功能，ApplicationContext添加了更多特定于企业的功能。ApplicationContext是BeanFactory完整的超集，BeanFactory在本章中仅用于Spring的IoC容器的描述。有关使用BeanFactory而不是ApplicationContext,参考 [BeanFactory](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-beanfactory)更多信息。

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

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

1.2. Container Overview

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

org.springframework.context.ApplicationContext接口代表Spring IoC容器，负责实例化，配置和组装bean。容器通过读取配置元数据获取有关要实例化，配置和组装的对象的指令。配置元数据以XML，Java注释或Java代码表示。它允许您表达组成应用程序的对象以及这些对象之间丰富的相互依赖性。

Several implementations of the ApplicationContext interface are supplied with Spring. In stand-alone applications, it is common to create an instance of [ClassPathXmlApplicationContext](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/context/support/ClassPathXmlApplicationContext.html) or [FileSystemXmlApplicationContext](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/context/support/FileSystemXmlApplicationContext.html). While XML has been the traditional format for defining configuration metadata, you can instruct the container to use Java annotations or code as the metadata format by providing a small amount of XML configuration to declaratively enable support for these additional metadata formats.

Spring提供了ApplicationContext接口的几种实现。在独立应用程序中，通常会创建一个[ClassPathXmlApplicationContext](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/context/support/ClassPathXmlApplicationContext.html) 或 [FileSystemXmlApplicationContext](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/context/support/FileSystemXmlApplicationContext.html)的实例。虽然XML是定义配置元数据的传统格式，但您可以通过提供少量XML配置来声明容器使用Java注释（Java annotations）或代码作为元数据格式，以声明方式启用对这些元数据格式的支持。

In most application scenarios, explicit user code is not required to instantiate one or more instances of a Spring IoC container. For example, in a web application scenario, a simple eight (or so) lines of boilerplate web descriptor XML in the web.xml file of the application typically suffices (see [Convenient ApplicationContext Instantiation for Web Applications](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#context-create)). If you use the [Spring Tool Suite](https://spring.io/tools/sts) (an Eclipse-powered development environment), you can easily create this boilerplate configuration with a few mouse clicks or keystrokes.

在大多数应用程序方案中，不需要显式用户代码来实例化Spring IoC容器的一个或多个实例。例如，在Web应用程序场景中，在 web.xml中简单八行（左右）的样板XML Web描述符通常就足够了（请参阅[Web应用程序的便捷ApplicationContext实例](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html" \l "context-create)）。如果您使用 [Spring Tool Suite](https://spring.io/tools/sts)（基于Eclipse的开发环境），只需点击几下鼠标或按键即可轻松创建此样板配置。

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

下图显示了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ApplicationContext创建和初始化之后，您拥有一个已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

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

如上图所示，Spring IoC容器使用一种配置元数据。此配置元数据表示您作为应用程序开发人员告诉Spring容器在应用程序中如何实例化，配置和组装对象。

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

传统上，配置元数据以简单直观的XML格式提供，本章的大部分内容用于传达Spring IoC容器的关键概念和功能。

|  |
| --- |
| XML-based metadata is not the only allowed form of configuration metadata. The Spring IoC container itself is totally decoupled from the format in which this configuration metadata is actually written. These days, many developers choose [Java-based configuration](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-java) for their Spring applications.  基于XML的元数据不是唯一允许的配置元数据形式。在实际编写配置元数据时，Spring IoC容器本身与编写配置的形式完全分离。目前，许多开发人员为其Spring应用程序选择 [基于Java的配置](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-java)。 |

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

* [Annotation-based configuration](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-annotation-config): Spring 2.5 introduced support for annotation-based configuration metadata.
* [Java-based configuration](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-java): Starting with Spring 3.0, many features provided by the Spring JavaConfig project became part of the core Spring Framework. Thus, you can define beans external to your application classes by using Java rather than XML files. To use these new features, see the [@Configuration](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Configuration.html), [@Bean](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Bean.html), [@Import](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Import.html), and [@DependsOn](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/DependsOn.html) annotations.

有关在Spring容器中使用其他形式的元数据的信息，请参阅：

* [基于注释的配置](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-annotation-config)：Spring 2.5引入了对基于注释的配置元数据的支持。
* [基于Java的配置](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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 Framework的一部分。因此，您可以使用Java而不是XML文件在应用程序类外部定义bean。要使用这些新功能，请参阅 [@Configuration](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Configuration.html)， [@Bean](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Bean.html)， [@Import](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Import.html)，和[@DependsOn](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/DependsOn.html)注释。

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

Spring配置包含容器必须管理的至少一个且通常不止一个bean定义。基于XML的配置元数据将这些bean配置为顶级元素<beans/>内的<bean/>元素。Java配置通常在@Configuration类中使用@Bean-annotated的注释方法。

These bean definitions correspond to the actual objects that make up your application. Typically, you define service layer objects, data access objects (DAOs), presentation objects such as Struts Action instances, infrastructure objects such as Hibernate SessionFactories, JMS Queues, and so forth. Typically, one does not configure fine-grained domain objects in the container, because it is usually the responsibility of DAOs and business logic to create and load domain objects. However, you can use Spring’s integration with AspectJ to configure objects that have been created outside the control of an IoC container. See [Using AspectJ to dependency-inject domain objects with Spring](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-atconfigurable).

这些bean定义对应于构成应用程序的实际对象。通常，可以定义服务层对象、数据访问对象（DAO）、表示对象（如Struts Action实例）、基础结构对象（如Hibernate SessionFactories，JMS Queues）等等。通常，不会在容器中配置细粒度域对象，因为DAO和业务逻辑通常负责创建和加载域对象。但是，您可以使用Spring与AspectJ的集成来配置在IoC容器控制之外创建的对象。请参阅[使用AspectJ与Spring进行域对象的依赖注入](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html" \l "aop-atconfigurable)。

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

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

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

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

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

</bean>

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

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

</bean>

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

</beans>

|  |
| --- |
| The id attribute is a string that identifies the individual bean definition. |
| The class attribute defines the type of the bean and uses the fully qualified classname.  id属性是一个标识单个bean定义的字符串。  class属性定义bean的类型并使用完全限定的类名。 |

The value of the id attribute refers to collaborating objects. The XML for referring to collaborating objects is not shown in this example. See [Dependencies](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-dependencies) for more information.

id属性的值指的是协作对象。在此示例中未显示用于引用协作对象的XML。有关更多信息，请参阅 [依赖项](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-dependencies)。

1.2.2. Instantiating a Container

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

提供给ApplicationContext构造函数的路径位置是资源字符串，它允许容器从各种外部资源（如本地文件系统，Java CLASSPATH等）加载配置元数据。

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

|  |  |
| --- | --- |
|  | After you learn about Spring’s IoC container, you may want to know more about Spring’s Resource abstraction (as described in [Resources](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#resources)), which provides a convenient mechanism for reading an InputStream from locations defined in a URI syntax. In particular, Resource paths are used to construct applications contexts, as described in [Application Contexts and Resource Paths](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#resources-app-ctx).  在了解了Spring的IoC容器之后，您可能想要了解有关Spring Resource抽象的更多信息（如[参考资料中所述](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#resources)），它提供了一种从URI语法中定义的位置读取InputStream的便捷机制。特别是， Resource路径用于构建应用程序上下文，如[应用程序上下文和资源路径中所述](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#resources-app-ctx)。 |

The following example shows the service layer objects (services.xml) configuration file:

以下示例显示了服务层对象(services.xml)配置文件：

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

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

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

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

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

*<!-- services -->*

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

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

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

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

</bean>

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

</beans>

The following example shows the data access objects daos.xml file:

以下示例显示了数据访问对象daos.xml文件：

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

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

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

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

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

<bean id="accountDao"

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

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

</bean>

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

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

</bean>

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

</beans>

In the preceding example, the service layer consists of the PetStoreServiceImpl class and two data access objects of the types JpaAccountDao and JpaItemDao (based on the JPA Object-Relational Mapping standard). The property name element refers to the name of the JavaBean property, and the ref element refers to the name of another bean definition. This linkage between id and ref elements expresses the dependency between collaborating objects. For details of configuring an object’s dependencies, see [Dependencies](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-dependencies).

在前面的示例中，服务层包含PetStoreServiceImpl类和两个数据访问JpaAccountDao和JpaItemDao类（基于JPA对象关系映射标准ORM）。property name元素是指JavaBean属性的名称，以及ref元素指的是另一个bean定义的名称。元素id和ref元素之间的这种联系表达了协作对象之间的依赖关系。有关配置对象的依赖关系的详细信息，请参阅 [依赖关系](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-dependencies)。

Composing XML-based Configuration Metadata

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

让bean定义跨越多个XML文件会很有用。通常，每个单独的XML配置文件都代表架构中的逻辑层或模块。

You can use the application context constructor to load bean definitions from all these XML fragments. This constructor takes multiple Resource locations, as was shown in the [previous section](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-instantiation). Alternatively, use one or more occurrences of the <import/>element to load bean definitions from another file or files. The following example shows how to do so:

您可以使用应用程序上下文构造函数从所有这些XML片段加载bean定义。此构造函数采用多个Resource位置，如上[一节中](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-instantiation)所示 。或者，使用一个或多个<import/>元素来从另一个或多个文件加载bean定义。以下示例显示了如何执行此操作：

<beans>

<import resource="services.xml"/>

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

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

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

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

</beans>

In the preceding example, external bean definitions are loaded from three files: services.xml, messageSource.xml, and themeSource.xml. All location paths are relative to the definition file doing the importing, so services.xml must be in the same directory or classpath location as the file doing the importing, while messageSource.xml and themeSource.xml must be in a resources location below the location of the importing file. As you can see, a leading slash is ignored. However, given that these paths are relative, it is better form not to use the slash at all. The contents of the files being imported, including the top level <beans/> element, must be valid XML bean definitions, according to the Spring Schema.

在前面的例子中，外部bean定义是从三个文件加载： services.xml，messageSource.xml，和themeSource.xml。所有位置路径都与执行导入的定义文件相关，因此services.xml必须与执行导入的文件位于相同的目录或类路径位置，而且 messageSource.xml 和themeSource.xml必须位于resources导入文件位置下方的位置。如您所见，忽略前导斜杠。但是，鉴于这些路径是相对的，最好不要使用斜杠。根据Spring Schema，正在导入的文件的内容<beans/>（包括顶级元素）必须是有效的XML bean定义。

|  |
| --- |
| It is possible, but not recommended, to reference files in parent directories using a relative "../" path. Doing so creates a dependency on a file that is outside the current application. In particular, this reference is not recommended for classpath: URLs (for example, classpath:../services.xml), where the runtime resolution process chooses the “nearest” classpath root and then looks into its parent directory. Classpath configuration changes may lead to the choice of a different, incorrect directory.  可以（但不建议）使用相对“../”路径引用父目录中的文件。这样做会对当前应用程序之外的文件创建依赖关系。特别是，不建议使用此引用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导致选择不同的，不正确的目录。  You can always use fully qualified resource locations instead of relative paths: for example, file:C:/config/services.xml or classpath:/config/services.xml. However, be aware that you are coupling your application’s configuration to specific absolute locations. It is generally preferable to keep an indirection for such absolute locations — for example, through "${…​}" placeholders that are resolved against JVM system properties at runtime.  您始终可以使用完全限定的资源位置而不是相对路径：例如，file:C:/config/services.xml或classpath:/config/services.xml。但是，请注意您将应用程序的配置与特定的绝对位置耦合。通常最好为这些绝对位置保持间接 - 例如，通过在运行时针对JVM系统属性解析的“$ {...}”占位符。 |

The namespace itself provices the import directive feature. Further configuration features beyond plain bean definitions are available in a selection of XML namespaces provided by Spring — for example, the context and util namespaces.

命名空间本身提供了导入指令功能。Spring提供的一系列XML命名空间中提供了除普通bean定义之外的其他配置功能 - 例如，context和util名称空间。

The Groovy Bean Definition DSL

As a further example for externalized configuration metadata, bean definitions can also be expressed in Spring’s Groovy Bean Definition DSL, as known from the Grails framework. Typically, such configuration live in a ".groovy" file with the structure shown in the following example:

作为外化配置元数据的另一个示例，bean定义也可以在Spring的Groovy Bean定义DSL中表示，如Grails框架中所知。通常，此类配置位于“.groovy”文件中，其结构如下例所示：

beans {

dataSource(BasicDataSource) {

driverClassName = "org.hsqldb.jdbcDriver"

url = "jdbc:hsqldb:mem:grailsDB"

username = "sa"

password = ""

settings = [mynew:"setting"]

}

sessionFactory(SessionFactory) {

dataSource = dataSource

}

myService(MyService) {

nestedBean = { AnotherBean bean ->

dataSource = dataSource

}

}

}

This configuration style is largely equivalent to XML bean definitions and even supports Spring’s XML configuration namespaces. It also allows for importing XML bean definition files through an importBeans directive.

此配置样式在很大程度上等同于XML bean定义，甚至支持Spring的XML配置命名空间。它还允许通过importBeans指令导入XML bean定义文件。

1.2.3. Using the Container

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

ApplicationContext是高级工厂的接口，能够对不同bean的注册及其依赖项进行维护。通过使用该方法T getBean(String name, Class<T> requiredType)，您可以取回Bean的实例。

The ApplicationContext lets you read bean definitions and access them, as the following example shows:

ApplicationContext让你可以读取bean的定义和访问它们，如下例所示：

*// create and configure beans*

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

*// retrieve configured instance*

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

*// use configured instance*

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

With Groovy configuration, bootstrapping looks very similar. It has a different context implementation class which is Groovy-aware (but also understands XML bean definitions). The following example shows Groovy configuration:

使用Groovy配置，bootstrapping看起来非常相似。它有一个不同的上下文实现类，它是Groovy-aware（但也理解XML bean定义）。以下示例显示了Groovy配置：

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

The most flexible variant is GenericApplicationContext in combination with reader delegates — for example, with XmlBeanDefinitionReader for XML files, as the following example shows:

最灵活的变体是GenericApplicationContext与reader delegates相结合 - 例如，XmlBeanDefinitionReader，如以下示例所示：

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

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

context.refresh();

You can also use the GroovyBeanDefinitionReader for Groovy files, as the following example shows:

您还可以使用GroovyBeanDefinitionReaderfor Groovy文件，如以下示例所示：

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

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

context.refresh();

You can mix and match such reader delegates on the same ApplicationContext, reading bean definitions from diverse configuration sources.

您可以在同一ApplicationContext混合和匹配这样的reader委托，从不同的配置源读取bean定义。

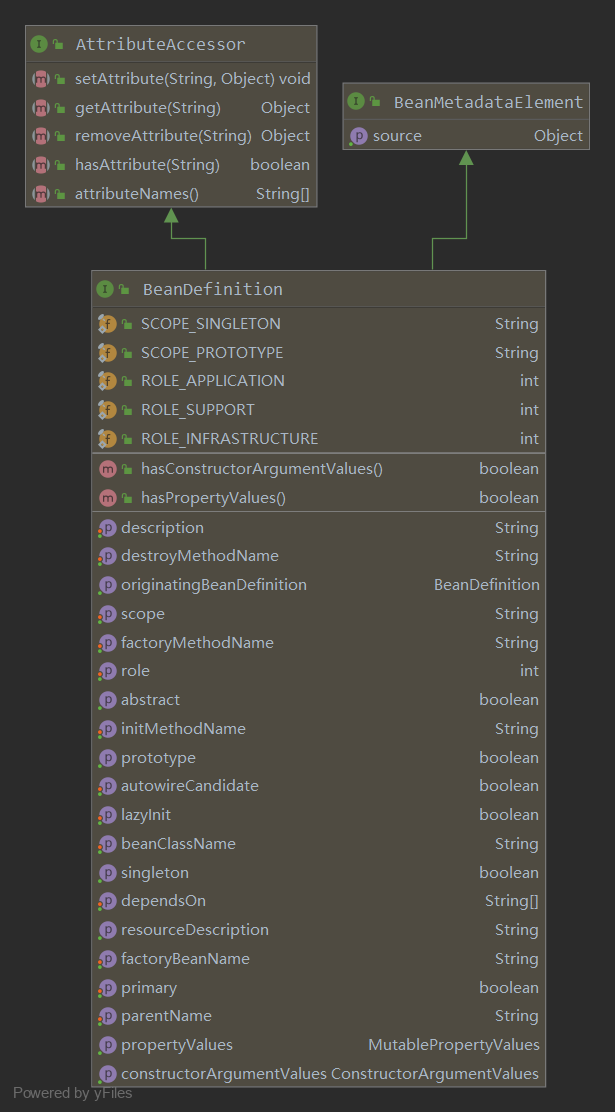
You can then use getBean to retrieve instances of your beans. The ApplicationContext interface has a few other methods for retrieving beans, but, ideally, your application code should never use them. Indeed, your application code should have no calls to the getBean() method at all and thus have no dependency on Spring APIs at all. For example, Spring’s integration with web frameworks provides dependency injection for various web framework components such as controllers and JSF-managed beans, letting you declare a dependency on a specific bean through metadata (such as an autowiring annotation).

然后，您可以使用getBean来获取Bean的实例。ApplicationContext 接口还有一些其他方法可用于检索bean，但理想情况下，应用程序代码永远不应使用它们。实际上，您的应用程序代码根本不应该调用 getBean()方法，因此根本不依赖于Spring API。例如，Spring与Web框架的集成为各种Web框架组件（如控制器和JSF托管bean）提供依赖注入，允许您通过元数据（例如自动装配注释）声明对特定bean的依赖性。

1.3. Bean Overview

A Spring IoC container manages one or more beans. These beans are created with the configuration metadata that you supply to the container (for example, in the form of XML <bean/> definitions).

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 ）。



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

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

* A package-qualified class name: typically, the actual implementation class of the bean being defined.
* 包限定的类名：通常是定义的bean的实际实现类。
* Bean behavioral configuration elements, which state how the bean should behave in the container (scope, lifecycle callbacks, and so forth).
* Bean行为配置元素，说明bean在容器中的行为方式（范围，生命周期回调等）。
* References to other beans that are needed for the bean to do its work. These references are also called collaborators or dependencies.
* Other configuration settings to set in the newly created object — for example, the size limit of the pool or the number of connections to use in a bean that manages a connection pool.
* 要在新创建的对象中设置的其他配置设置 - 例如，池的大小限制或在管理连接池的Bean中使用的连接数。

This metadata translates to a set of properties that make up each bean definition. The following table describes these properties:

此元数据转换为构成每个bean定义的一组属性。下表描述了这些属性：

|  |  |
| --- | --- |
| *Table 1. The bean definition* | |
| **Property** | **Explained in…​** |
| Class | [Instantiating Beans](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html" \l "beans-factory-class) |
| Name | [Naming Beans](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html" \l "beans-beanname) |
| Scope | [Bean Scopes](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html" \l "beans-factory-scopes) |
| Constructor arguments | [Dependency Injection](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html" \l "beans-factory-collaborators) |
| Properties | [Dependency Injection](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html" \l "beans-factory-collaborators) |
| Autowiring mode | [Autowiring Collaborators](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html" \l "beans-factory-autowire) |
| Lazy initialization mode | [Lazy-initialized Beans](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html" \l "beans-factory-lazy-init) |
| Initialization method | [Initialization Callbacks](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html" \l "beans-factory-lifecycle-initializingbean) |
| Destruction method | [Destruction Callbacks](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html" \l "beans-factory-lifecycle-disposablebean) |

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

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

|  |
| --- |
| Bean metadata and manually supplied singleton instances need to be registered as early as possible, in order for the container to properly reason about them during autowiring and other introspection steps. While overriding existing metadata and existing singleton instances is supported to some degree, the registration of new beans at runtime (concurrently with live access to the factory) is not officially supported and may lead to concurrent access exceptions, inconsistent state in the bean container, or both.  需要尽早注册Bean元数据和手动提供的单例实例，以便容器在自动装配和其他内省步骤期间正确推理它们。虽然在某种程度上支持覆盖现有元数据和现有单例实例，但是在运行时注册新bean（与对工厂的实时访问同时）并未得到官方支持，并且可能导致并发访问异常，bean容器中的状态不一致，或两者皆有可能发生。 |

1.3.1. Naming Beans

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

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。bean通常只有一个标识符。但是，如果需要多个，则额外的标识符可以被视为别名。

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

在基于XML的配置元数据中，您可以使用id属性，name属性或两者来指定bean标识符。id属性允许您指定一个id。通常，这些名称是字母数字（'myBean'，'someService'等），但它们也可以包含特殊字符。如果要为bean引入其他别名，还可以在name 属性中指定它们，用逗号（,），分号（;）或空格分隔。作为历史记录，在Spring 3.1之前的版本中， id属性被定义为一种xsd:ID类型，它约束了可能的字符。从3.1开始，它被定义为一种xsd:string类型。请注意，bean id仍然被容器强制唯一性，但不再是XML解析器。

You are not required to supply a name or an id for a bean. If you do not supply a name or id explicitly, the container generates a unique name for that bean. However, if you want to refer to that bean by name, through the use of the ref element or a[Service Locator](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-servicelocator) style lookup, you must provide a name. Motivations for not supplying a name are related to using [inner beans](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-inner-beans)and [autowiring collaborators](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-autowire).

您不需要为bean 提供 name或id 。如果您不显示提供 name或id，则容器会为该bean生成唯一的名称。但是，如果要按名称引用该bean，通过使用ref元素或 [Service Locator](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-servicelocator)样式查找，则必须提供名称。不提供名称的动机与使用[内部bean](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-inner-beans)和[自动装配协作者有关](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-autowire)。

Bean Naming Conventions

The convention is to use the standard Java convention for instance field names when naming beans. That is, bean names start with a lowercase letter and are camel-cased from there. Examples of such names include accountManager,accountService, userDao, loginController, and so forth.

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

Bean命名约定

惯例是在命名bean时使用标准Java约定作为实例字段名称。也就是说，bean名称以小写字母开头，并从那里开始驼峰。这样的名字的例子包括accountManager， accountService，userDao，loginController等等。

命名bean始终使您的配置更易于阅读和理解。此外，如果您使用Spring AOP，那么在将建议应用于与名称相关的一组bean时，它会有很大帮助。

|  |
| --- |
| With component scanning in the classpath, Spring generates bean names for unnamed components, following the rules described earlier: essentially, taking the simple class name and turning its initial character to lower-case. However, in the (unusual) special case when there is more than one character and both the first and second characters are upper case, the original casing gets preserved. These are the same rules as defined by java.beans.Introspector.decapitalize (which Spring uses here).  通过类路径中的组件扫描，Spring按照前面描述的规则为未命名的组件生成bean名称：实质上，采用简单的类名并将其初始字符转换为小写。但是，在（不常见的）特殊情况下，当有多个字符且第一个和第二个字符都是大写字母时，原始外壳将被保留。这些规则与java.beans.Introspector.decapitalize（Spring在此处使用）定义的规则相同。 |

Aliasing a Bean outside the Bean Definition

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

在bean定义本身中，您可以为bean提供多个名称，方法是使用id属性指定最多一个名称和使用name属性指定任意数量的其他名称的组合。这些名称可以是同一个bean的等效别名，对某些情况很有用，例如让应用程序中的每个组件通过使用特定于该组件本身的bean名称来引用公共依赖项。

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

但是，指定实际定义bean的所有别名并不总是足够的。有时需要为其他地方定义的bean引入别名。在大型系统中通常就是这种情况，其中配置在每个子系统之间分配，每个子系统具有其自己的一组对象定义。在基于XML的配置元数据中，您可以使用该<alias/>元素来完成此任务。以下示例显示了如何执行此操作：

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

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

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

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

例如，子系统A的配置元数据可以通过名称引用DataSource subsystemA-dataSource。子系统B的配置元数据可以通过名称引用DataSource subsystemB-dataSource。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引用DataSource myApp-dataSource。要使所有三个名称引用同一对象，可以将以下别名定义添加到配置元数据中：

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

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

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

现在，每个组件和主应用程序都可以通过一个唯一的名称引用dataSource，并保证不与任何其他定义冲突（有效地创建命名空间），但它们引用相同的bean。

Java-configuration

If you use Javaconfiguration, the @Bean annotation can be used to provide aliases. See [Using the @Bean Annotation](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-java-bean-annotation) for details.

Java的配置

如果使用Javaconfiguration，@Bean可以使用注释来提供别名。有关详细信息，请参阅[使用@Bean注释](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-java-bean-annotation)。

1.3.2. Instantiating Beans

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

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

If you use XML-based configuration metadata, you specify the type (or class) of object that is to be instantiated in the classattribute of the <bean/> element. This class attribute (which, internally, is a Class property on a BeanDefinition instance) is usually mandatory. (For exceptions, see [Instantiation by Using an Instance Factory Method](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-class-instance-factory-method) and [Bean Definition Inheritance](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-child-bean-definitions).) You can use the Class property in one of two ways:

如果使用基于XML的配置元数据，则要在<bean/>元素的class属性中指定实例化的对象的类型（或类）。 class属性（在内部，是BeanDefinition实例的Class属性）通常是必需的。（有关例外，请参阅 [使用实例工厂方法](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-class-instance-factory-method)和[Bean定义继承](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-child-bean-definitions)[进行实例化](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-class-instance-factory-method)。）您可以通过以下两种方式之一使用Class属性：

* Typically, to specify the bean class to be constructed in the case where the container itself directly creates the bean by calling its constructor reflectively, somewhat equivalent to Java code with the new operator.
* 通常，在容器本身通过反向调用其构造函数直接创建bean的情况下指定要构造的bean类，稍微等同于使用new运算符的Java代码。
* To specify the actual class containing the static factory method that is invoked to create the object, in the less common case where the container invokes a static factory method on a class to create the bean. The object type returned from the invocation of the static factory method may be the same class or another class entirely.
* 要指定为创建对象而调用包含static工厂方法的实际类，在不太常见的情况下，容器在类上调用static工厂方法来创建bean。调用static工厂方法返回的对象类型可以是同一个类或完全是另一个类。

*Inner class names*

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

双

For example, if you have a class called SomeThing in the com.example package, and this SomeThing class has a static nested class called OtherThing, the value of the class attribute on a bean definition would be com.example.SomeThing$OtherThing.

有个名为SomeThing的类名为的static嵌套类

Notice the use of the $ character in the name to separate the nested class name from the outer class name.

请注意，使用$名称中的字符将嵌套类名与外部类名分开。

Instantiation with a Constructor

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

使用构造函数实例化

当您通过构造函数方式创建bean时，所有普通类都可以使用并与Spring兼容。也就是说，正在开发的类不需要实现任何特定接口或以特定方式编码。简单地指定bean类就足够了。但是，根据您特定bean使用的IoC类型，您可能需要一个默认（空）构造函数。

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

它根据建模了适当的奇异

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

使用基于XML的配置元数据，您可以按如下方式指定bean类：

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

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

For details about the mechanism for supplying arguments to the constructor (if required) and setting object instance properties after the object is constructed, see [Injecting Dependencies](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-collaborators).

有关为构造函数提供参数的机制（如果需要）以及在构造对象后设置对象实例属性的详细信息，请参阅 [注入依赖项](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-collaborators)。

Instantiation with a Static Factory Method

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

使用静态工厂方法实例化

定义使用静态工厂方法创建bean时，请使用class 属性指定包含static工厂方法的类，并使用factory-method名称的属性指定工厂方法本身的名称。您应该能够调用此方法（使用可选参数，如稍后所述）并返回一个活动对象，随后将其视为通过构造函数创建的对象。这种bean定义的一个用途是在遗留代码中调用static工厂。

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

以下bean定义指定通过调用工厂方法来创建bean。该定义未指定返回对象的类型（类），仅指定包含工厂方法的类。在此示例中，createInstance() 方法必须是静态方法。以下示例显示如何指定工厂方法：

<bean id="clientService"

class="examples.ClientService"

factory-method="createInstance"/>

The following example shows a class that would work with the preceding bean definition:

以下示例显示了一个可以使用前面的bean定义的类：

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

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

**private** ClientService() {}

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

**return** clientService;

}

}

For details about the mechanism for supplying (optional) arguments to the factory method and setting object instance properties after the object is returned from the factory, see [Dependencies and Configuration in Detail](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-properties-detailed).

有关在从工厂返回对象后为工厂方法提供（可选）参数和设置对象实例属性的机制的详细信息，请参阅[依赖关系和详细配置](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-properties-detailed)。

Instantiation by Using an Instance Factory Method

Similar to instantiation through a [static factory method](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-class-static-factory-method), instantiation with an instance factory method invokes a non-static method of an existing bean from the container to create a new bean. To use this mechanism, leave the class attribute empty and, in the factory-bean attribute, specify the name of a bean in the current (or parent or ancestor) container that contains the instance method that is to be invoked to create the object. Set the name of the factory method itself with the factory-method attribute. The following example shows how to configure such a bean:

使用实例工厂方法实例化

与通过[静态工厂方法](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-class-static-factory-method)实例化类似，使用实例工厂方法进行实例化会从容器调用现有bean的非静态方法来创建新bean。要使用此机制，请将该class属性保留为空，并在factory-bean属性中指定当前（或父或祖先）容器中bean的名称，该容器包含要调用以创建对象的实例方法。使用factory-method属性设置工厂方法本身的名称。以下示例显示如何配置此类bean：

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

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

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

</bean>

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

<bean id="clientService"

factory-bean="serviceLocator"

factory-method="createClientServiceInstance"/>

The following example shows the corresponding Java class:

以下示例显示了相应的Java类：

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

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

**public** ClientService createClientServiceInstance() {

**return** clientService;

}

}

One factory class can also hold more than one factory method, as the following example shows:

一个工厂类也可以包含多个工厂方法，如以下示例所示：

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

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

</bean>

<bean id="clientService"

factory-bean="serviceLocator"

factory-method="createClientServiceInstance"/>

<bean id="accountService"

factory-bean="serviceLocator"

factory-method="createAccountServiceInstance"/>

The following example shows the corresponding Java class:

以下示例显示了相应的Java类：

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

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

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

**public** ClientService createClientServiceInstance() {

**return** clientService;

}

**public** AccountService createAccountServiceInstance() {

**return** accountService;

}

}

This approach shows that the factory bean itself can be managed and configured through dependency injection (DI). See [Dependencies and Configuration in Detail](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-properties-detailed).

这种方式表明工厂bean本身可以通过依赖注入（DI）进行管理和配置。请参阅[详细信息中的依赖关系和配置](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-properties-detailed)。

|  |
| --- |
| In Spring documentation, “factory bean” refers to a bean that is configured in the Spring container and that creates objects through an [instance](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-class-instance-factory-method) or [static](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-class-static-factory-method) factory method. By contrast, FactoryBean (notice the capitalization) refers to a Spring-specific [FactoryBean](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-extension-factorybean).  在Spring文档中，“factory bean”是指在Spring容器中配置并通过[实例](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-class-instance-factory-method)或 [静态](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-class-static-factory-method)工厂方法创建对象的bean 。相比之下，FactoryBean（注意大写）指的是特定于Spring的 [FactoryBean](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-extension-factorybean)。 |

1.4. Dependencies

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

典型的企业应用程序不包含单个对象（或Spring用法中的bean）。即使是最简单的应用程序也有一些对象可以协同工作，以呈现最终用户所看到连贯的应用程序。下一节将介绍如何定义多个独立的bean定义，以及对象协作实现目标的应用程序。

1.4.1. Dependency Injection

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

依赖注入（DI）是一个过程，通过这个过程，对象只能通过构造函数参数、工厂方法的参数来定义他们的依赖关系，或在从构造函数、工厂方法返回后的对象实例上设置的属性来定义它们的依赖关系（即，它们使用的其他对象）。从工厂方法返回。然后容器在创建bean时注入这些依赖项。此过程基本上是bean自身通过使用类的直接构造函数或诸如服务定位器模式的机制来控制其依赖关系的实例化或位置的逆过程（因此命名控制反转）。

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

使用DI原则的代码更清晰，当需要对象提供其依赖项时，解耦更有效。对象不查找其依赖项，也不知道依赖项的位置或依赖项的类。因此，您的类变得更容易测试，特别是当依赖关系在接口或抽象基类上时，这允许在单元测试中使用存根或模拟实现。

DI exists in two major variants: [Constructor-based dependency injection](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-constructor-injection) and [Setter-based dependency injection](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-setter-injection).

DI主要有两个变体：[基于构造函数的依赖注入](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-constructor-injection)和[基于Setter的依赖注入](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-setter-injection)。

Constructor-based Dependency Injection

Constructor-based DI is accomplished by the container invoking a constructor with a number of arguments, each representing a dependency. Calling a static factory method with specific arguments to construct the bean is nearly equivalent, and this discussion treats arguments to a constructor and to a static factory method similarly. The following example shows a class that can only be dependency-injected with constructor injection:

基于构造函数的DI由容器调用具有多个参数的构造函数来完成，每个参数表示一个依赖项。调用具有特定参数的static工厂方法来构造bean几乎是等效的，本讨论同样处理构造函数和static工厂方法的参数。以下示例显示了一个只能通过构造函数进行依赖注入的类：

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

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

**private** MovieFinder movieFinder;

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

**public** SimpleMovieLister(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

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

}

Notice that there is nothing special about this class. It is a POJO that has no dependencies on container specific interfaces, base classes or annotations.

请注意，这个类没有什么特别之处。它是一个POJO，它不依赖于容器特定的接口，基类或注释。

*Constructor Argument Resolution*

Constructor argument resolution matching occurs by using the argument’s type. If no potential ambiguity exists in the constructor arguments of a bean definition, the order in which the constructor arguments are defined in a bean definition is the order in which those arguments are supplied to the appropriate constructor when the bean is being instantiated. Consider the following class:

构造函数参数解析

通过使用参数的类型进行构造函数参数解析匹配。如果bean定义的构造函数参数中不存在潜在的歧义，那么在bean定义中定义构造函数参数的顺序是在实例化bean时将这些参数提供给适当的构造函数的顺序。考虑以下课程：

**package** x.y;

**public** **class** **ThingOne** {

**public** ThingOne(ThingTwo thingTwo, ThingThree thingThree) {

*// ...*

}

}

Assuming that ThingTwo and ThingThree classes are not related by inheritance, no potential ambiguity exists. Thus, the following configuration works fine, and you do not need to specify the constructor argument indexes or types explicitly in the <constructor-arg/> element.

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

<beans>

<bean id="beanOne" class="x.y.ThingOne">

<constructor-arg ref="beanTwo"/>

<constructor-arg ref="beanThree"/>

</bean>

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

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

</beans>

When another bean is referenced, the type is known, and matching can occur (as was the case with the preceding example). When a simple type is used, such as <value>true</value>, Spring cannot determine the type of the value, and so cannot match by type without help. Consider the following class:

当引用另一个bean时，类型是已知的，并且可以发生匹配（与前面的示例一样）。当使用简单类型时，例如 <value>true</value>，Spring无法确定值的类型，因此无法在没有帮助的情况下按类型进行匹配。考虑以下课程：

**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;

}

}

*Constructor argument type matching*

In the preceding scenario, the container can use type matching with simple types if you explicitly specify the type of the constructor argument by using the type attribute. as the following example shows:

*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，则容器可以使用与简单类型的类型匹配。如下例所示：

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

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

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

</bean>

*Constructor argument index*

You can use the index attribute to specify explicitly the index of constructor arguments, as the following example shows:

*构造函数参数索引*

您可以使用该index属性显式指定构造函数参数的索引，如以下示例所示：

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

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

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

</bean>

In addition to resolving the ambiguity of multiple simple values, specifying an index resolves ambiguity where a constructor has two arguments of the same type.

除了解决多个简单值的歧义之外，指定索引还可以解决构造函数具有相同类型的两个参数的歧义。

|  |  |
| --- | --- |
|  | The index is 0-based.  index从0开始。 |

*Constructor argument name*

You can also use the constructor parameter name for value disambiguation, as the following example shows:

*构造函数参数名称*

您还可以使用构造函数参数名称进行值消歧，如以下示例所示：

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

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

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

</bean>

Keep in mind that, to make this work out of the box, your code must be compiled with the debug flag enabled so that Spring can look up the parameter name from the constructor. If you cannot or do not want to compile your code with the debug flag, you can use the [@ConstructorProperties](https://download.oracle.com/javase/8/docs/api/java/beans/ConstructorProperties.html) JDK annotation to explicitly name your constructor arguments. The sample class would then have to look as follows:

请记住，为了使这项工作开箱即用，必须在启用调试标志的情况下编译代码，以便Spring可以从构造函数中查找参数名称。如果您不能或不想使用debug标志编译代码，则可以使用 [@ConstructorProperties](https://download.oracle.com/javase/8/docs/api/java/beans/ConstructorProperties.html) JDK批注显式命名构造函数参数。然后，示例类必须如下所示：

**package** examples;

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

*// Fields omitted*

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

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

this.years = years;

this.ultimateAnswer = ultimateAnswer;

}

}

Setter-based Dependency Injection

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

基于Setter的依赖注入

在调用无参数构造函数或无参数static工厂方法来实例化bean之后，基于setter的DI由bean上的容器调用setter方法完成。

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

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

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

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

**private** MovieFinder movieFinder;

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

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

this.movieFinder = movieFinder;

}

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

}

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

ApplicationContext管理的bean支持基于构造函数和基于setter的DI。在通过构造函数方法注入了一些依赖项之后，它还支持基于setter的DI。您可以以一个BeanDefinition 的形式配置依赖项，并将其与PropertyEditor实例结合使用，以将属性从一种格式转换为另一种格式。然而，大多数Spring用户不直接与这些类（即，编程），而是用XML bean 定义注释的组件（也就是带注释类@Component，@Controller等），或者@Bean方法在基于Java的@Configuration类。然后，这些源在内部转换为BeanDefinition实例，并用于加载整个Spring IoC容器实例。

Constructor-based or setter-based DI?

还是

Since you can mix constructor-based and setter-based DI, it is a good rule of thumb to use constructors for mandatory dependencies and setter methods or configuration methods for optional dependencies. Note that use of the [@Required](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-required-annotation)annotation on a setter method can be used to make the property be a required dependency; however, constructor injection with programmatic validation of arguments is preferable.

使用对于强制依赖项使用构造函数，对于可选依赖项使用setter方法或配置方法是一个很好的经验法则。可以带有参数

The Spring team generally advocates constructor injection, as it lets you implement application components as immutable objects and ensures that required dependencies are not null. Furthermore, constructor-injected components are always returned to the client (calling) code in a fully initialized state. As a side note, a large number of constructor arguments is a bad code smell, implying that the class likely has too many responsibilities and should be refactored to better address proper separation of concerns.

另一点需要注意正确性

Setter injection should primarily only be used for optional dependencies that can be assigned reasonable default values within the class. Otherwise, not-null checks must be performed everywhere the code uses the dependency. One benefit of setter injection is that setter methods make objects of that class amenable to reconfiguration or re-injection later. Management through [JMX MBeans](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/integration.html#jmx) is therefore a compelling use case for setter injection.

例子

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

使用对特定类最有意义的DI样式。有时，在处理您没有源的第三方类时，会为你做出选择。例如，如果第三方类没有公开任何setter方法，那么构造函数注入可能是唯一可用的DI形式。

Dependency Resolution Process

依赖解析过程

The container performs bean dependency resolution as follows:

容器执行bean依赖解析，如下所示：

* The ApplicationContext is created and initialized with configuration metadata that describes all the beans. Configuration metadata can be specified by XML, Java code, or annotations.
* 使用所有的描述bean（annotations）
* For each bean, its dependencies are expressed in the form of properties, constructor arguments, or arguments to the static-factory method (if you use that instead of a normal constructor). These dependencies are provided to the bean, when the bean is actually created.
* 都该方法当
* Each property or constructor argument is an actual definition of the value to set, or a reference to another bean in the container.
* 每个属性或构造函数参数都是要设置值的实际定义，或者是对容器中另一个bean的引用。
* Each property or constructor argument that is a value is converted from its specified format to the actual type of that property or constructor argument. By default, Spring can convert a value supplied in string format to all built-in types, such as int, long, String, boolean, and so forth.
* 作为值的每个属性或构造函数参数都从其指定的格式转换为该属性或构造函数参数的实际类型。默认情况下，Spring能够将提供的字符串格式的值转换成所有内置类型的值，例如int、 long、String、boolean等等。

The Spring container validates the configuration of each bean as the container is created. However, the bean properties themselves are not set until the bean is actually created. Beans that are singleton-scoped and set to be pre-instantiated (the default) are created when the container is created. Scopes are defined in [Bean Scopes](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-scopes). Otherwise, the bean is created only when it is requested. Creation of a bean potentially causes a graph of beans to be created, as the bean’s dependencies and its dependencies' dependencies (and so on) are created and assigned. Note that resolution mismatches among those dependencies may show up late — that is, on first creation of the affected bean.

Spring容器在创建容器时验证每个bean的配置。但是，在实际创建bean之前，不会设置bean属性本身。创建容器时会创建单例作用域并设置为预先实例化（默认值）的Bean。范围在[Bean范围](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-scopes)中定义。否则，仅在请求时才创建bean。创建bean可能会导致创建bean的图形，因为bean的依赖项及其依赖项的依赖项（依此类推）被创建和分配。请注意，这些依赖项之间的不匹配解析可能会出现较晚 - 也就是说，在首次创建受影响的bean时才会出现。

Circular dependencies

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

有可能会

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

的的的

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

一些类的源代码，这些类

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

与典型情况（没有循环依赖）不同，bean A和bean B之间的循环依赖强制其中一个bean在完全初始化之前被注入另一个bean（一个经典的鸡与蛋的场景）。

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

你通常可以相信Spring会做正确的事。它在容器加载时检测配置问题，例如对不存在的bean和循环依赖关系的引用。当实际创建bean时，Spring会尽可能晚地设置属性并解析依赖关系。这意味着，如果在创建该对象或其中一个依赖项时出现问题，则在请求对象时，正确加载的Spring容器可以在以后生成异常 - 例如，bean因缺失或无效而抛出异常属性。这可能会延迟一些配置问题的可见性，这是ApplicationContext默认情况下实现预实例化单例bean的原因。以实际需要之前创建这些bean的一些前期时间和内存为代价，您会在创建ApplicationContext时发现配置问题，而不是更晚。您仍然可以覆盖此默认行为，以便单例bean可以懒惰地初始化，而不是预先实例化。

If no circular dependencies exist, when one or more collaborating beans are being injected into a dependent bean, each collaborating bean is totally configured prior to being injected into the dependent bean. This means that, if bean A has a dependency on bean B, the Spring IoC container completely configures bean B prior to invoking the setter method on bean A. In other words, the bean is instantiated (if it is not a pre-instantiated singleton), its dependencies are set, and the relevant lifecycle methods (such as a [configured init method](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-initializingbean) or the [InitializingBean callback method](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-initializingbean)) are invoked.

如果不存在循环依赖关系，当一个或多个协作bean被注入依赖bean时，每个协作bean在被注入依赖bean之前完全配置。这意味着，如果bean A依赖于bean B，则Spring IoC容器在调用bean A上的setter方法之前完全配置bean B.换句话说，bean被实例化（如果它不是预先实例化的单例），设置其依赖项，并调用相关的生命周期方法（如[配置的init方法](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-initializingbean) 或[InitializingBean回调方法](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-initializingbean)）。

Examples of Dependency Injection

The following example uses XML-based configuration metadata for setter-based DI. A small part of a Spring XML configuration file specifies some bean definitions as follows:

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

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

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

<property name="beanOne">

<ref bean="anotherExampleBean"/>

</property>

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

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

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

</bean>

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

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

The following example shows the corresponding ExampleBean class:

以下示例显示了相应的ExampleBean类：

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

**private** AnotherBean beanOne;

**private** YetAnotherBean beanTwo;

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

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

this.beanOne = beanOne;

}

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

this.beanTwo = beanTwo;

}

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

this.i = i;

}

}

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

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

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

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

<constructor-arg>

<ref bean="anotherExampleBean"/>

</constructor-arg>

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

<constructor-arg ref="yetAnotherBean"/>

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

</bean>

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

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

The following example shows the corresponding ExampleBean class:

以下示例显示了相应的ExampleBean类：

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

**private** AnotherBean beanOne;

**private** YetAnotherBean beanTwo;

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

**public** ExampleBean(

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

this.beanOne = anotherBean;

this.beanTwo = yetAnotherBean;

this.i = i;

}

}

The constructor arguments specified in the bean definition are used as arguments to the constructor of the ExampleBean.

bean定义中指定的构造函数参数用作构造函数ExampleBean的参数。

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

现在考虑这个示例的变体，其中，不使用构造函数，而是告诉Spring调用static工厂方法来返回对象的实例：

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

<constructor-arg ref="anotherExampleBean"/>

<constructor-arg ref="yetAnotherBean"/>

<constructor-arg value="1"/>

</bean>

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

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

The following example shows the corresponding ExampleBean class:

以下示例显示了相应的ExampleBean类：

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

*// a private constructor*

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

...

}

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

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

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

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

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

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

*// some other operations...*

**return** eb;

}

}

Arguments to the static factory method are supplied by <constructor-arg/> elements, exactly the same as if a constructor had actually been used. The type of the class being returned by the factory method does not have to be of the same type as the class that contains the static factory method (although, in this example, it is). An instance (non-static) factory method can be used in an essentially identical fashion (aside from the use of the factory-bean attribute instead of the class attribute), so we do not discuss those details here.

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

1.4.2. Dependencies and Configuration in Detail

As mentioned in the [previous section](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-collaborators), you can define bean properties and constructor arguments as references to other managed beans (collaborators) or as values defined inline. Spring’s XML-based configuration metadata supports sub-element types within its <property/> and <constructor-arg/> elements for this purpose.

如上[一节所述](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-collaborators)，您可以将bean属性和构造函数参数定义为对其他托管bean（协作者）的引用，或者将其定义为内联的值。为此目的，Spring的基于XML的配置元数据支持子元素类型<property/>和<constructor-arg/>。

Straight Values (Primitives, Strings, and so on)

The value attribute of the <property/> element specifies a property or constructor argument as a human-readable string representation. Spring’s [conversion service](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#core-convert-ConversionService-API) is used to convert these values from a String to the actual type of the property or argument. The following example shows various values being set:

直接值（内建值，字符串Strings等）

<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 [转换服务](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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。以下示例显示了要设置的各种值：

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

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

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

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

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

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

</bean>

The following example uses the [p-namespace](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-p-namespace) for even more succinct XML configuration:

以下示例使用[p命名空间](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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

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

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

destroy-method="close"

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

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

p:username="root"

p:password="masterkaoli"/>

</beans>

The preceding XML is more succinct. However, typos are discovered at runtime rather than design time, unless you use an IDE (such as [IntelliJ IDEA](https://www.jetbrains.com/idea/) or the [Spring Tool Suite](https://spring.io/tools/sts)) that supports automatic property completion when you create bean definitions. Such IDE assistance is highly recommended.

前面的XML更简洁。但是，除非您在创建bean定义时使用支持自动属性完成的IDE（例如[IntelliJ IDEA](https://www.jetbrains.com/idea/)或[Spring Tool Suite](https://spring.io/tools/sts)），否则会在运行时而不是设计时发现拼写错误。强烈建议使用此类IDE帮助。

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

您还可以配置java.util.Properties实例，如下所示：

<bean id="mappings"

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

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

<property name="properties">

<value>

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

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

</value>

</property>

</bean>

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

Spring容器通过使用JavaBeans  PropertyEditor机制将<value/>元素内的文本转换为 java.util.Properties实例。这是一个很好的快捷方式， 采用<value/>嵌套而不是value属性样式，这也是Spring团队支持这样使用的少数几个地方之一。

**The idref element**

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

**idref元素**

idref元素只是一种简单防错方法，可以将容器中另一个bean 的id（一个字符串值 - 而不是引用）传递给<constructor-arg/>或<property/> 元素。以下示例显示了如何使用它：

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

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

<property name="targetName">

<idref bean="theTargetBean"/>

</property>

</bean>

The preceding bean definition snippet is exactly equivalent (at runtime) to the following snippet:

前面的bean定义代码段与以下代码段完全等效（在运行时）：

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

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

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

</bean>

The first form is preferable to the second, because using the idref tag lets the container validate at deployment time that the referenced, named bean actually exists. In the second variation, no validation is performed on the value that is passed to the targetName property of the client bean. Typos are only discovered (with most likely fatal results) when the client bean is actually instantiated. If the client bean is a [prototype](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-scopes) bean, this typo and the resulting exception may only be discovered long after the container is deployed.

第一种形式优于第二种形式，因为使用idref标记允许容器在部署时验证引用的命名bean实际存在。在第二个形式中，不对传递给client bean 的targetName属性值执行验证。只有在client实际实例化bean 时才会发现错误（很可能导致致命的结果）。如果client bean是[原型](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-scopes) bean，则只能在部署容器后很长时间才能发现此错误和产生异常。

The local attribute on the idref element is no longer supported in the 4.0 beans XSD, since it does not provide value over a regular bean reference any more. Change your existing idref local references to idref bean when upgrading to the 4.0 schema.

4.0 beans XSD不再支持idref元素的local属性，因为它不再提供常规bean引用的值。升级到4.0架构时，将存在的idref local引用更改为idref bean。



A common place (at least in versions earlier than Spring 2.0) where the <idref/> element brings value is in the configuration of [AOP interceptors](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-pfb-1) in a ProxyFactoryBean bean definition. Using <idref/> elements when you specify the interceptor names prevents you from misspelling an interceptor ID.

其中一个共同的地方（至少在Spring 2.0之前的版本中）<idref/>元素带来的值在配置[AOP拦截](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-pfb-1)在 ProxyFactoryBeanbean定义。指定拦截器名称时使用<idref/>元素可防止拦截器ID拼写错误。

References to Other Beans (Collaborators)

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

**引用其他beans（协作者）**

ref元素是定义<constructor-arg/>或<property/> 元素内部的最终元素。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。引用的bean是要设置其属性的bean的依赖项，并且在设置该属性之前根据需要对其进行初始化。（如果协作者是单例bean，它可能已经被容器初始化。）所有引用最终都是对另一个对象的引用。范围和有效性取决于是否通过bean，local,或parent属性指定其他对象的ID或名称。

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

通过 <ref/>的bean属性指定目标bean是最常用的形式，并允许创建对同一容器或父容器中的任何bean的引用，而不管它是否在同一XML文件中。bean属性的值可以和目标bean 的id属性的值相同，或者与目标bean的name属性的值相同。以下示例显示如何使用ref元素：

<ref bean="someBean"/>

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

通过parent属性指定目标bean 会创建对当前容器的父容器中的bean的引用。parent 属性的值可以与目标bean 的id属性或目标bean的name属性中的值之一相同。目标bean必须位于当前bean的父容器中。您应该使用此bean引用变体，主要是当您有容器层次结构并且希望将现有bean包装在父容器中时，该容器具有与父bean同名的代理。以下一对列表显示了如何使用该parent属性：

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

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

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

</bean>

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

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

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

<property name="target">

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

</property>

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

</bean>



Inner Beans

A <bean/> element inside the <property/> or <constructor-arg/> elements defines an inner bean, as the following example shows:

在<property/>或<constructor-arg/>元素里的一个<bean/>定义了一个内部bean，如下面的示例所示：

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

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

<property name="target">

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

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

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

</bean>

</property>

</bean>

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

内部bean定义不需要定义的ID或名称。如果指定，容器不会使用此类值作为标识符。容器还会在创建时忽略scope标志，因为内部bean始终是匿名的，并且始终由外部bean创建。不可能独立访问内部bean或将它们注入除封闭bean之外的协作bean。

As a corner case, it is possible to receive destruction callbacks from a custom scope — for example, for a request-scoped inner bean contained within a singleton bean. The creation of the inner bean instance is tied to its containing bean, but destruction callbacks let it participate in the request scope’s lifecycle. This is not a common scenario. Inner beans typically simply share their containing bean’s scope.

作为一个极端情况，可以从自定义范围接收销毁回调 - 例如，对于包含在单例bean中的请求范围内部bean。内部bean实例的创建与其包含bean相关联，但是销毁回调允许它参与请求范围的生命周期。这不是常见的情况。内部bean通常只是共享其包含bean的范围。

Collections

The <list/>, <set/>, <map/>, and <props/> elements set the properties and arguments of the Java Collection types List, Set, Map, and Properties, respectively. The following example shows how to use them:

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

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

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

<property name="adminEmails">

<props>

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

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

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

</props>

</property>

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

<property name="someList">

<list>

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

<ref bean="myDataSource" />

</list>

</property>

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

<property name="someMap">

<map>

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

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

</map>

</property>

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

<property name="someSet">

<set>

<value>just some string</value>

<ref bean="myDataSource" />

</set>

</property>

</bean>

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

一个map的键或值，或者一个set值也可以是以下任何元素：

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

Collection Merging

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

Collection合并

Spring容器还支持合并集合。应用程序开发人员可以定义父<list/>，<map/>，<set/>或<props/>元素，并有孩子<list/>，<map/>，<set/>或<props/>元素从父集合继承和覆盖值。也就是说，子集合的值是合并父集合和子集合的元素的结果，子集合的元素覆盖父集合中指定的值。

This section on merging discusses the parent-child bean mechanism. Readers unfamiliar with parent and child bean definitions may wish to read the [relevant section](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-child-bean-definitions) before continuing.

The following example demonstrates collection merging:

关于合并的这一部分讨论了父子bean机制。不熟悉父母和子bean定义的读者可能希望在继续之前阅读 [相关部分](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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>

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

注意在childbean定义的adminEmails属性上的<props/>元素使用merge=true  。当child容器解析并实例化bean时，生成的实例有一个adminEmails Properties集合，其中包含将子集合adminEmails与父adminEmails集合合并的结果 。以下清单显示了结果：

administrator=administrator@example.com

sales=sales@example.com

support=support@example.co.uk

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

子Properties集合的值继承父所有属性元素<props/>，子support的值将覆盖父集合的值。

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

这一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，保持与List集合类型（即，ordered值集合的概念）相关联的语义。父级的值位于所有子级列表的值之前。在Map，Set和Properties集合类型的情况下，没有顺序存在。因此，没有排序的语义在背后的关联的集合类型的效果Map，Set以及Properties使用容器内部实现类型。

Limitations of Collection Merging

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

集合合并的局限性

您无法合并不同的集合类型（例如a Map和a List）。如果您尝试这样做，则会引发相应的Exception异常。merge属性必须在较低的子定义继承上。merge在父集合定义上指定属性是无效的，并且不会导致所需的合并。

Strongly-typed collection

With the introduction of generic types in Java 5, you can use strongly typed collections. That is, it is possible to declare a Collection type such that it can only contain (for example) String elements. If you use Spring to dependency-inject a strongly-typed Collection into a bean, you can take advantage of Spring’s type-conversion support such that the elements of your strongly-typed Collection instances are converted to the appropriate type prior to being added to the Collection. The following Java class and bean definition show how to do so:

强类型集合

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，可以声明一种Collection类型，使得它只能包含（例如）String元素。如果使用Spring将强类型依赖注入Collection到bean中，则可以利用Spring的类型转换支持，以便强类型Collection 实例的元素在添加到Collection之前转换为适当的类型。以下Java类和bean定义显示了如何执行此操作：

**public** **class** **SomeClass** {

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

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

this.accounts = accounts;

}

}

<beans>

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

<property name="accounts">

<map>

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

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

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

</map>

</property>

</bean>

</beans>

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

当something bean 为注入accounts属性准备时，通过反射可获得something 的Map<String, Float>关于强类型的元素类型的泛型信息。因此，Spring的类型转换基础结构将各种值元素识别为类型Float，并将字符串值（9.99, 2.75，和 3.99）转换为实际Float类型。

Null and Empty String Values

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

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

<bean class="ExampleBean">

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

</bean>

The preceding example is equivalent to the following Java code:

上面的示例等效于以下Java代码：

exampleBean.setEmail("");

The <null/> element handles null values. The following listing shows an example:

该<null/>元素处理null值。以下清单显示了一个示例：

<bean class="ExampleBean">

<property name="email">

<null/>

</property>

</bean>

The preceding configuration is equivalent to the following Java code:

上述配置等同于以下Java代码：

exampleBean.setEmail(null);

XML Shortcut with the p-namespace

The p-namespace lets you use the bean element’s attributes (instead of nested <property/> elements) to describe your property values collaborating beans, or both.

带有p命名空间的XML快捷方式

p-namespace允许您使用bean元素的属性（而不是嵌套 <property/>元素）来描述属性值、协作bean，或两者都支持。

Spring supports extensible configuration formats [with namespaces](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#xsd-schemas), which are based on an XML Schema definition. The beansconfiguration format discussed in this chapter is defined in an XML Schema document. However, the p-namespace is not defined in an XSD file and exists only in the core of Spring.

Spring支持[具有命名空间的](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#xsd-schemas)可扩展配置格式，这些[命名空间](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#xsd-schemas)基于XML Schema定义。beans本章中讨论的配置格式在XML Schema文档中定义。但是，p-namespace未在XSD文件中定义，仅存在于Spring的核心中。

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

以下示例显示了两个XML片段（第一个使用标准XML格式，第二个使用p命名空间）解析为相同的结果：

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

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

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

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

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

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

<property name="email" value="someone@somewhere.com"/>

</bean>

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

p:email="someone@somewhere.com"/>

</beans>

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

该示例显示了在bean定义中调用email的p命名空间属性。这告诉Spring包含一个属性声明。如前所述，p命名空间没有架构定义，因此您可以将属性的名称设置为属性名称。

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

下一个示例包括另外两个bean定义，它们都引用了另一个bean：

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

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

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

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

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

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

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

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

</bean>

<bean name="john-modern"

class="com.example.Person"

p:name="John Doe"

p:spouse-ref="jane"/>

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

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

</bean>

</beans>

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

此示例不仅包含使用p命名空间的属性值，还使用特殊格式来声明属性引用。第一个bean定义用于<property name="spouse" ref="jane"/>创建从bean john到bean 的引用 jane，而第二个bean定义p:spouse-ref="jane"用作属性来执行完全相同的操作。在这种情况下，spouse是属性名称，而该-ref部分表示这不是直接值，而是对另一个bean的引用。

The p-namespace is not as flexible as the standard XML format. For example, the format for declaring property references clashes with properties that end in Ref, whereas the standard XML format does not. We recommend that you choose your approach carefully and communicate this to your team members to avoid producing XML documents that use all three approaches at the same time.

p命名空间不如标准XML格式灵活。例如，声明属性引用的格式与最终的属性冲突Ref，而标准XML格式则不然。我们建议您仔细选择您的方法并将其传达给您的团队成员，以避免生成同时使用所有三种方法的XML文档。



XML Shortcut with the c-namespace

Similar to the [XML Shortcut with the p-namespace](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-p-namespace), the c-namespace, introduced in Spring 3.1, allows inlined attributes for configuring the constructor arguments rather then nested constructor-arg elements.

与[带有p-namespace](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-p-namespace)的[XML Shortcut](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-p-namespace)类似，Spring 3.1中引入的c-namespace允许使用内联属性来配置构造函数参数，而不是嵌套constructor-arg元素。

The following example uses the c: namespace to do the same thing as the from [Constructor-based Dependency Injection](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-constructor-injection):

以下示例基于构造函数的依赖注入使用c:命名空间执行同样的操作：

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

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

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

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

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

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

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

*<!-- traditional declaration with optional argument names -->*

<bean id="beanOne" class="x.y.ThingOne">

<constructor-arg name="thingTwo" ref="beanTwo"/>

<constructor-arg name="thingThree" ref="beanThree"/>

<constructor-arg name="email" value="something@somewhere.com"/>

</bean>

*<!-- c-namespace declaration with argument names -->*

<bean id="beanOne" class="x.y.ThingOne" c:thingTwo-ref="beanTwo"

c:thingThree-ref="beanThree" c:email="something@somewhere.com"/>

</beans>

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

c:命名空间使用与p:相同的约定（尾部-ref作为bean的引用），供他们的名字设置构造函数的参数。类似地，它需要在XML文件中声明，甚至它不需在XSD中定义（它存在于Spring核心内部）。

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

对于构造函数参数名称不可见的罕见情况（通常在没有调试信息的情况下编译字节码），您可以回退到使用参数索引，如下所示：

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

<bean id="beanOne" class="x.y.ThingOne" c:\_0-ref="beanTwo" c:\_1-ref="beanThree"

c:\_2="something@somewhere.com"/>



Due to the XML grammar, the index notation requires the presence of the leading \_, as XML attribute names cannot start with a number (even though some IDEs allow it). A corresponding index notation is also available for <constructor-arg> elements but not commonly used since the plain order of declaration is usually sufficient there.

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）。对于<constructor-arg>元素也可以使用相应的索引符号，但不常用，因为通常的声明顺序就足够了。

In practice, the constructor resolution [mechanism](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-ctor-arguments-resolution) is quite efficient in matching arguments, so unless you really need to, we recommend using the name notation through-out your configuration.

实际上，构造函数解析[机制](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-ctor-arguments-resolution)在匹配参数方面非常有效，因此除非您确实需要，否则我们建议在整个配置中使用名称表示法。

Compound Property Names

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

设置bean属性时，可以使用复合或嵌套属性名称，只要除最终属性名称之外的路径的所有组件都不是null。考虑以下bean定义：

<bean id="something" class="things.ThingOne">

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

</bean>

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

something bean具有一个fred属性，该属性具有bob属性，bob属性具有sammy 属性，并且最终sammy属性的值设置为123。为了使其工作，在构造bean之后，something的fred属性和bob属性不得为null。否则，将抛出一个NullPointerException异常。

1.4.3. Using depends-on

If a bean is a dependency of another bean, that usually means that one bean is set as a property of another. Typically you accomplish this with the [<ref/> element](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-ref-element) in XML-based configuration metadata. However, sometimes dependencies between beans are less direct. An example is when a static initializer in a class needs to be triggered, such as for database driver registration. The depends-on attribute can explicitly force one or more beans to be initialized before the bean using this element is initialized. The following example uses the depends-on attribute to express a dependency on a single bean:

如果bean是另一个bean的依赖项，那通常意味着将一个bean设置为另一个bean的属性。通常，您可以使用基于XML的配置元数据中的[<ref/> 元素](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-ref-element)来完成此操作。但是，有时bean之间的依赖关系不那么直接。例如，需要触发类中的静态初始化程序，例如数据库驱动程序注册。在使用此元素（<ref/> ）初始化bean之前，depends-on属性可以显式强制初始化一个或多个bean。以下示例使用该depends-on属性表示对单个bean的依赖关系：

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

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

To express a dependency on multiple beans, supply a list of bean names as the value of the depends-on attribute (commas, whitespace, and semicolons are valid delimiters):

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

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

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

</bean>

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

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



The depends-on attribute can specify both an initialization-time dependency and, in the case of [singleton](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html" \l "beans-factory-scopes-singleton) beans only, a corresponding destruction-time dependency. Dependent beans that define a depends-on relationship with a given bean are destroyed first, prior to the given bean itself being destroyed. Thus, depends-on can also control shutdown order.

depends-on属性既可以指定初始化时间依赖项，也可以指定相应的销毁时间依赖项（仅在[单例](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html" \l "beans-factory-scopes-singleton)bean的情况下）。在给定bean本身销毁之前，用depends-on定义依赖关系的bean首先被销毁。这样，depends-on也可以控制关闭顺序。

1.4.4. Lazy-initialized Beans

By default, ApplicationContext implementations eagerly create and configure all [singleton](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-singleton) beans as part of the initialization process. Generally, this pre-instantiation is desirable, because errors in the configuration or surrounding environment are discovered immediately, as opposed to hours or even days later. When this behavior is not desirable, you can prevent pre-instantiation of a singleton bean by marking the bean definition as being lazy-initialized. A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup.

默认情况下，ApplicationContext会预先创建和配置所有 [单例](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-singleton) bean，作为初始化过程的一部分。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可立即发现，而不是几小时甚至几天后才发现。当不希望出现这种情况时，可以通过将bean定义标记为延迟初始化来阻止单例bean的预实例化。延迟初始化的bean告诉IoC容器在第一次请求时创建bean实例，而不是在启动时。

In XML, this behavior is controlled by the lazy-init attribute on the <bean/> element, as the following example shows:

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

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

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

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

当前面的配置被一个ApplicationContext使用时，lazybean在ApplicationContext启动时不会急切地预先实例化，而not.lazybean被预先实例化。

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

但是，当延迟初始化的bean是未进行延迟初始化的单例bean的依赖项时，ApplicationContext会在启动时创建延迟初始化的bean，因为它必须满足单例的依赖关系。惰性初始化的bean被注入到其他地方的单独的bean中，而这个bean并不是惰性初始化的。

You can also control lazy-initialization at the container level by using the default-lazy-init attribute on the <beans/> element, a the following example shows:

您还可以通过使用<beans/>元素的default-lazy-init属性来控制容器级别的延迟初始化， 以下示例显示：

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

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

</beans>

1.4.5. Autowiring Collaborators

自动装配协作者

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

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

* Autowiring can significantly reduce the need to specify properties or constructor arguments. (Other mechanisms such as a bean template [discussed elsewhere in this chapter](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-child-bean-definitions) are also valuable in this regard.)
* 自动装配可以显著减少指定属性或构造函数参数的需要。（本章其他地方讨论的bean模板等其他机制在这方面也很有价值。）
* Autowiring can update a configuration as your objects evolve. For example, if you need to add a dependency to a class, that dependency can be satisfied automatically without you needing to modify the configuration. Thus autowiring can be especially useful during development, without negating the option of switching to explicit wiring when the code base becomes more stable.
* 自动装配可以随着对象的发展更新配置。例如，如果需要向类添加依赖项，则可以自动满足该依赖项，而无需修改配置。因此，自动动装配在开发期间尤其有用，而不需要在代码库变得更稳定时取消切换到显式动装配的选项。

When using XML-based configuration metadata (see [Dependency Injection](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-collaborators)), you can specify the autowire mode for a bean definition with the autowire attribute of the <bean/> element. The autowiring functionality has four modes. You specify autowiring per bean and can thus choose which ones to autowire. The following table describes the four autowiring modes:

使用基于XML的配置元数据（请参阅[依赖注入](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-collaborators)）时，可以使用<bean/>元素的autowire属性为 bean定义指定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，因此可以选择要自动装配的那些。下表描述了四种自动装配模式：

|  |  |
| --- | --- |
| *Table 2. Autowiring modes* | |
| **Mode** | **Explanation** |
| no | (Default) No autowiring. Bean references must be defined by ref elements. 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.  （默认）无自动装配。Bean引用必须由ref元素定义。不建议对较大的部署更改默认设置，因为明确指定协作者可以提供更好的控制和清晰度。在某种程度上，它记录了系统的结构。 |
| 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.  按属性名称自动装配。Spring查找与需要自动装配的属性同名的bean。例如，如果bean定义按名称设置为autowire并且它包含一个master属性（即，它有一个 setMaster(..)方法），则Spring会查找名为bean的定义master并使用它来设置属性。 |
| byType | Lets a property 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).  如果容器中只存在一个属性类型的bean，则允许属性自动装配。如果存在多个，则抛出致命异常，这表示您可能不会byType对该bean 使用自动装配。如果没有匹配的bean，则不会发生任何事情（该属性未设置）。 |
| 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.  类似byType但适用于构造函数参数。如果容器中没有构造函数参数类型的一个bean，则会引发致命错误。 |



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

使用bytype或构造函数自动连接模式，可以连接数组和集合类型。在这种情况下，将提供容器中与预期类型匹配的所有自动连接候选，以满足依赖关系。如果所需的键类型为字符串，则可以自动连接强类型映射实例。自动连接的映射实例的值由匹配预期类型的所有bean实例组成，并且映射实例的键包含相应的bean名称。

Limitations and Disadvantages of Autowiring

Autowiring works best when it is used consistently across a project. If autowiring is not used in general, it might be confusing to developers to use it to wire only one or two bean definitions.

Consider the limitations and disadvantages of autowiring:

考虑自动装配的局限和缺点：

* Explicit dependencies in property and constructor-arg settings always override autowiring. You cannot autowire simple properties such as primitives, Strings, and Classes (and arrays of such simple properties). This limitation is by-design.
* 属性和构造函数参数设置中的显式依赖项始终覆盖自动连接。不能自动连接简单属性，如内建类型、字符串和类（以及简单属性的数组）。这种限制是由设计造成的。
* Autowiring is less exact than explicit wiring. Although, as noted in the earlier table, Spring is careful to avoid guessing in case of ambiguity that might have unexpected results. The relationships between your Spring-managed objects are no longer documented explicitly.
* 自动装配不如显式装配精确。尽管如前表所述，Spring谨慎地避免猜测，以防出现可能产生意外结果的歧义。您的Spring管理对象之间的关系不再明确记录。
* Wiring information may not be available to tools that may generate documentation from a Spring container.
* 装配信息可能无法用于可能从spring容器生成文档的工具。
* Multiple bean definitions within the container may match the type specified by the setter method or constructor argument to be autowired. For arrays, collections, or Map instances, this is not necessarily a problem. However, for dependencies that expect a single value, this ambiguity is not arbitrarily resolved. If no unique bean definition is available, an exception is thrown.
* 容器中的多个bean定义可以匹配setter方法或要自动连接的构造函数参数指定的类型。对于数组、集合或Map 实例，这不一定是问题。但是，对于期望单个值的依赖项，这种模糊性不会被随意解决。如果没有唯一的bean定义可用，则引发异常。

In the latter scenario, you have several options:

在后一种情况下，您有几种选择：

* Abandon autowiring in favor of explicit wiring.
* 放弃自动装配，支持显式装配。
* Avoid autowiring for a bean definition by setting its autowire-candidate attributes to false, as described in the [next section](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-autowire-candidate).
* bean 的false
* Designate a single bean definition as the primary candidate by setting the primary attribute of its <bean/> element to true.
* <bean/>元素的true
* Implement the more fine-grained control available with annotation-based configuration, as described in [Annotation-based Container Configuration](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-annotation-config).

Excluding a Bean from Autowiring

从自动装配中排除Bean

On a per-bean basis, you can exclude a bean from autowiring. In Spring’s XML format, set the autowire-candidate attribute of the <bean/> element to false. The container makes that specific bean definition unavailable to the autowiring infrastructure (including annotation style configurations such as [@Autowired](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-autowired-annotation)).

在每个bean的基础上，您可以从自动装配中排除bean。在Spring的XML格式中，将元素<bean/>的autowire-candidate属性设置为false。容器使特定的bean定义对自动装配基础结构不可用（包括注释样式配置等[@Autowired](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-autowired-annotation)）。

The autowire-candidate attribute is designed to only affect type-based autowiring. It does not affect explicit references by name, which get resolved even if the specified bean is not marked as an autowire candidate. As a consequence, autowiring by name nevertheless injects a bean if the name matches.

autowire-candidate属性旨在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未标记为autowire候选，也会解析它。因此，如果按名称匹配，则会按名称自动装配注入bean。



You can also limit autowire candidates based on pattern-matching against bean names. The top-level <beans/> element accepts one or more patterns within its default-autowire-candidates attribute. For example, to limit autowire candidate status to any bean whose name ends with Repository, provide a value of \*Repository. To provide multiple patterns, define them in a comma-separated list. An explicit value of true or false for a bean definition’s autowire-candidate attribute always takes precedence. For such beans, the pattern matching rules do not apply.

您还可以根据与bean名称的模式匹配来限制autowire候选者。顶级<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候选状态限制为名称以其结尾的任何bean Repository，请提供值\*Repository。要提供多个模式，请在逗号分隔的列表中定义它们。bean定义的属性的显式值 true或优先级始终优先。对于此类bean，模式匹配规则不适用。falseautowire-candidate

These techniques are useful for beans that you never want to be injected into other beans by autowiring. It does not mean that an excluded bean cannot itself be configured by using autowiring. Rather, the bean itself is not a candidate for autowiring other beans.

这些技术对于您永远不希望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，bean本身不是自动装配其他bean的候选者。

1.4.6. Method Injection

In most application scenarios, most beans in the container are [singletons](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-singleton). When a singleton bean needs to collaborate with another singleton bean or a non-singleton bean needs to collaborate with another non-singleton bean, you typically handle the dependency by defining one bean as a property of the other. A problem arises when the bean lifecycles are different. Suppose singleton bean A needs to use non-singleton (prototype) bean B, perhaps on each method invocation on A. The container creates the singleton bean A only once, and thus only gets one opportunity to set the properties. The container cannot provide bean A with a new instance of bean B every time one is needed.

在大多数应用程序场景中，容器中的大多数bean都是 [单例](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-singleton)。当单例bean需要与另一个单例bean协作或非单例bean需要与另一个非单例bean协作时，通常通过将一个bean定义为另一个bean的属性来处理依赖关系。当bean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，可能是在A上的每个方法调用上。容器只创建一次单例bean A，因此只有一次机会来设置属性。每次需要时，容器都不能为bean A提供bean B的新实例。

A solution is to forego some inversion of control. You can [make bean A aware of the container](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-aware) by implementing the ApplicationContextAware interface, and by [making a getBean("B") call to the container](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-client) ask for (a typically new) bean B instance every time bean A needs it. The following example shows this approach:

一个解决方案是放弃一些控制反转。当bean A需要bean B时，你可以通过实现ApplicationContextAware接口使bean了解容器，并通过[getBean("B")](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html" \l "beans-factory-client)调用容器请求bean B（新实例）的实例。以下示例显示了此方法：

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

**package** fiona.apple;

*// Spring-API imports*

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

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

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

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

**private** ApplicationContext applicationContext;

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

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

Command command = createCommand();

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

command.setState(commandState);

**return** command.execute();

}

**protected** Command createCommand() {

*// notice the Spring API dependency!*

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

}

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

ApplicationContext applicationContext) **throws** BeansException {

this.applicationContext = applicationContext;

}

}

The preceding is not desirable, because the business code is aware of and coupled to the Spring Framework. Method Injection, a somewhat advanced feature of the Spring IoC container, lets you handle this use case cleanly.

前面的内容是不可取的，因为业务代码耦合到Spring Framework。方法注入是Spring IoC容器的一个高级功能，可以让您干净地处理这个情况。

You can read more about the motivation for Method Injection in [this blog entry](https://spring.io/blog/2004/08/06/method-injection/).

您可以在[此博客条目中](https://spring.io/blog/2004/08/06/method-injection/)阅读有关Method Injection的动机的更多信息 。

Lookup Method Injection

Lookup method injection is the ability of the container to override methods on container-managed beans and return the lookup result for another named bean in the container. The lookup typically involves a prototype bean, as in the scenario described in [the preceding section](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-method-injection). The Spring Framework implements this method injection by using bytecode generation from the CGLIB library to dynamically generate a subclass that overrides the method.

查找方法注入是容器重写其管理的bean上的方法并返回容器中另一个命名bean的查找结果的能力。查找通常涉及一个原型bean，如上[一节](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-method-injection)中描述的场景。Spring Framework通过使用CGLIB库中的字节码来动态生成覆盖该方法的子类来实现此方法注入。

* For this dynamic subclassing to work, the class that the Spring bean container subclasses cannot be final, and the method to be overridden cannot be final, either.
* 要使这个动态子类工作，Spring bean容器子类不能是final类，以及要重写的方法也不能是final。
* Unit-testing a class that has an abstract method requires you to subclass the class yourself and to supply a stub implementation of the abstract method.
* 单元测试具有抽象方法的类需要您自己对类进行子类化，并提供抽象方法的存根实现。
* Concrete methods are also necessary for component scanning, which requires concrete classes to pick up.
* 组件扫描也需要具体的方法，这需要具体的类来获取。
* A further key limitation is that lookup methods do not work with factory methods and in particular not with @Bean methods in configuration classes, since, in that case, the container is not in charge of creating the instance and therefore cannot create a runtime-generated subclass on the fly.
* 另一个关键限制是查找方法不适用于工厂方法，特别是@Bean配置类中的方法，因为在这种情况下，容器不负责创建实例，因此无法创建运行时生成的子类。



In the case of the CommandManager class in the previous code snippet, the Spring container dynamically overrides the implementation of the createCommand() method. The CommandManager class does not have any Spring dependencies, as the reworked example shows:

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

**package** fiona.apple;

*// no more Spring imports!*

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

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

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

Command command = createCommand();

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

command.setState(commandState);

**return** command.execute();

}

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

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

}

In the client class that contains the method to be injected (the CommandManager in this case), the method to be injected requires a signature of the following form:

在包含要注入的方法的客户端类中（在本例中是CommandManager），要注入的方法需要以下形式的签名：

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

If the method is abstract, the dynamically-generated subclass implements the method. Otherwise, the dynamically-generated subclass overrides the concrete method defined in the original class. Consider the following example:

如果方法是abstract，则动态生成的子类实现该方法。否则，动态生成的子类将覆盖原始类中定义的具体方法。请考虑以下示例：

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

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

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

</bean>

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

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

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

</bean>

The bean identified as commandManager calls its own createCommand() method whenever it needs a new instance of the myCommand bean. You must be careful to deploy the myCommand bean as a prototype if that is actually what is needed. If it is a [singleton](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-singleton), the same instance of the myCommand bean is returned each time.

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

Alternatively, within the annotation-based component model, you can declare a lookup method through the @Lookupannotation, as the following example shows:

或者，在基于注释的组件模型中，您可以通过@Lookup注释声明查找方法，如以下示例所示：

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

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

Command command = createCommand();

command.setState(commandState);

**return** command.execute();

}

@Lookup("myCommand")

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

}

Or, more idiomatically, you can rely on the target bean getting resolved against the declared return type of the lookup method:

或者，更惯用地，您可以依赖于针对查找方法的声明返回类型解析目标bean：

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

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

MyCommand command = createCommand();

command.setState(commandState);

**return** command.execute();

}

@Lookup

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

}

Note that you should typically declare such annotated lookup methods with a concrete stub implementation, in order for them to be compatible with Spring’s component scanning rules where abstract classes get ignored by default. This limitation does not apply to explicitly registered or explicitly imported bean classes.

请注意，您通常应该使用具体的存根实现来声明这种带注释的查找方法，以使它们与Spring的组件扫描规则兼容，其中默认情况下抽象类被忽略。此限制不适用于显式注册或显式导入的bean类。

Another way of accessing differently scoped target beans is an ObjectFactory/ Provider injection point. See [Scoped Beans as Dependencies](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html" \l "beans-factory-scopes-other-injection).

You may also find the ServiceLocatorFactoryBean (in the org.springframework.beans.factory.config package) to be useful.

访问不同范围的目标bean的另一种方法是ObjectFactory/ Provider注入点。请参阅[Scoped Beans作为依赖关系](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html" \l "beans-factory-scopes-other-injection)。

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



Arbitrary Method Replacement

A less useful form of method injection than lookup method injection is the ability to replace arbitrary methods in a managed bean with another method implementation. You can safely skip the rest of this section until you actually need this functionality.

任意方法替换

与查找方法注入相比，一种不太有用的方法注入形式是能够使用另一个方法实现替换托管bean中的任意方法。您可以安全地跳过本节的其余部分，直到您确实需要此功能。

With XML-based configuration metadata, you can use the replaced-method element to replace an existing method implementation with another, for a deployed bean. Consider the following class, which has a method called computeValue that we want to override:

使用基于XML的配置元数据，您可以使用该replaced-method元素将已存在的方法实现替换为已部署的bean。考虑以下类，它有一个computeValue我们想要覆盖的方法：

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

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

*// some real code...*

}

*// some other methods...*

}

A class that implements the org.springframework.beans.factory.support.MethodReplacer interface provides the new method definition, as the following example shows:

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

*/\*\**

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

*\* implementation in MyValueCalculator*

*\*/*

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

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

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

String input = (String) args[0];

...

return ...;

}

}

The bean definition to deploy the original class and specify the method override would resemble the following example:

部署原始类并指定方法覆盖的bean定义类似于以下示例：

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

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

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

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

</replaced-method>

</bean>

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

You can use one or more <arg-type/> elements within the <replaced-method/> element to indicate the method signature of the method being overridden. The signature for the arguments is necessary only if the method is overloaded and multiple variants exist within the class. For convenience, the type string for an argument may be a substring of the fully qualified type name. For example, the following all match java.lang.String:

您可以在元素<replaced-method/> 中使用<arg-type/>元素个或多个来指示被覆盖的方法签名。仅当方法重载且类中存在多个变体时，才需要参数的签名。为方便起见，参数的类型字符串可以是完全限定类型名称的子字符串。例如，以下所有匹配java.lang.String：

java.lang.String

String

Str

Because the number of arguments is often enough to distinguish between each possible choice, this shortcut can save a lot of typing, by letting you type only the shortest string that matches an argument type.

因为参数的数量通常足以区分每个可能的选择，所以通过让您只键入与参数类型匹配的最短字符串，此快捷方式可以节省大量的输入。

1.5. Bean Scopes

When you create a bean definition, you create a recipe for creating actual instances of the class defined by that bean definition. The idea that a bean definition is a recipe is important, because it means that, as with a class, you can create many object instances from a single recipe.

创建bean定义时，就创建了一个配方，该配方用于通过该bean的定义进行创建的实际实例。bean定义是一个配方的想法很重要，因为它意味着，与类的方式相同，您可以从一个配方创建许多对象实例。

You can control not only the various dependencies and configuration values that are to be plugged into an object that is created from a particular bean definition but also control the scope of the objects created from a particular bean definition. This approach is powerful and flexible, because you can choose the scope of the objects you create through configuration instead of having to bake in the scope of an object at the Java class level. Beans can be defined to be deployed in one of a number of scopes. The Spring Framework supports six scopes, four of which are available only if you use a web-aware ApplicationContext. You can also create [a custom scope.](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-custom)

您不仅可以控制要插入到从特定bean定义创建的对象中的各种依赖项和配置值，还可以控制从特定bean定义创建的对象的范围。这种方法功能强大且灵活，因为您可以选择通过配置创建的对象的范围，而不必在Java类级别配置对象的范围。bean可以定义为部署在多个作用域中的一个作用域中。Spring Framework支持六个作用域，其中四个作用域仅在您使用Web时才可用ApplicationContext。您还可以创建 [自定义作用域。](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html" \l "beans-factory-scopes-custom)

The following table describes the supported scopes:

下表描述了支持的范围：

|  |  |
| --- | --- |
| *Table 3. Bean scopes* | |
| **Scope** | **Description** |
| [singleton](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-singleton) | (Default) Scopes a single bean definition to a single object instance for each Spring IoC container.  （默认）将单个bean定义范围限定为每个Spring IoC容器的单个对象实例。 |
| [prototype](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-prototype) | Scopes a single bean definition to any number of object instances.  将单个bean定义范围限定为任意数量的对象实例。 |
| [request](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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.  将单个bean定义范围限定为单个HTTP请求的生命周期。也就是说，每个HTTP请求都有自己的bean实例，它是在单个bean定义的后面创建的。仅在具有Web功能的Spring环境中有效ApplicationContext。 |
| [session](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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.  将单个bean定义范围限定为HTTP的生命周期Session。仅在具有Web感知功能的Spring环境中有效ApplicationContext。 |
| [application](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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.  将单个bean定义范围限定为一个ServletContext生命周期。仅在具有Web感知功能的Spring环境中有效ApplicationContext |
| [websocket](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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.  将单个bean定义范围限定为一个WebSocket的生命周期。仅在具有Web感知功能的Spring环境中有效ApplicationContext。 |



As of Spring 3.0, a thread scope is available but is not registered by default: see [SimpleThreadScope](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/context/support/SimpleThreadScope.html). As of Spring 4.2, a transaction scope is available as well: [SimpleTransactionScope](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/transaction/support/SimpleTransactionScope.html). For instructions on how to register these or any other custom scopes, see [Using a Custom Scope](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html" \l "beans-factory-scopes-custom-using).

从Spring 3.0开始，线程作用域范围可用，但默认情况下未注册：请参阅[SimpleThreadScope](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/context/support/SimpleThreadScope.html)。从Spring 4.2开始，还有一个事务作用域范围： [SimpleTransactionScope](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/transaction/support/SimpleTransactionScope.html)。有关如何注册这些或任何其他自定义作用域的说明，请参阅 [使用自定义作用域](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html" \l "beans-factory-scopes-custom-using)。



1.5.1. The Singleton Scope

Only one shared instance of a singleton bean is managed, and all requests for beans with an ID or IDs that match that bean definition result in that one specific bean instance being returned by the Spring container.

只管理单个bean的一个共享实例，并且对具有与该bean定义匹配的ID或多个ID的bean的所有请求都会导致Spring容器返回特定的一个bean实例。

To put it another way, when you define a bean definition and it is scoped as a singleton, the Spring IoC container creates exactly one instance of the object defined by that bean definition. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object. The following image shows how the singleton scope works:

换句话说，当您定义一个bean定义并将其作为单例作用域时，Spring IoC容器只创建该bean定义一个​​实例对象。此单个实例存储在此类单例bean的缓存中，并且该命名Bean的所有后续请求和引用都将返回缓存对象。下图显示了单例范围的工作原理：



Spring’s concept of a singleton bean differs from the singleton pattern as defined in the Gang of Four (GoF) patterns book. The GoF singleton hard-codes the scope of an object such that one and only one instance of a particular class is created per ClassLoader. The scope of the Spring singleton is best described as being per-container and per-bean. This means that, if you define one bean for a particular class in a single Spring container, the Spring container creates one and only one instance of the class defined by that bean definition. The singleton scope is the default scope in Spring. To define a bean as a singleton in XML, you can define a bean as shown in the following example:

Spring的单例bean概念不同于Gang of Four（GoF）模式书中定义的单例模式。GoF单例对一个对象的范围进行硬编码，使得每个ClassLoader创建一个且只有一个特定类的实例。Spring单例的作用域范围最好描述为每个容器和每个bean。这意味着，如果在单个Spring容器中为特定类定义一个bean，则Spring容器将创建该bean定义所定义的类的一个且仅一个实例。单例范围是Spring中的默认作用域范围。要将bean定义为XML中的单例，您可以如以下示例所示定义一个bean：

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

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

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

1.5.2. The Prototype Scope（2019/8/5）

The non-singleton prototype scope of bean deployment results in the creation of a new bean instance every time a request for that specific bean is made. That is, the bean is injected into another bean or you request it through a getBean() method call on the container. As a rule, you should use the prototype scope for all stateful beans and the singleton scope for stateless beans.

bean部署的非单例原型范围导致每次发出对该特定bean的请求时都创建新的bean实例。也就是说，bean被注入另一个bean，或者通过getBean()对容器的方法调用来请求它。通常，您应该对所有有状态bean使用原型范围，对无状态bean使用单例范围。

The following diagram illustrates the Spring prototype scope:

下图说明了Spring原型范围：



(A data access object (DAO) is not typically configured as a prototype, because a typical DAO does not hold any conversational state. It was easier for us to reuse the core of the singleton diagram.)

（数据访问对象（DAO）通常不配置为原型，因为典型的DAO不会保持任何会话状态。我们更容易重用单例图的核心。）

The following example defines a bean as a prototype in XML:

以下示例将bean定义为XML中的原型：

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

In contrast to the other scopes, Spring does not manage the complete lifecycle of a prototype bean. The container instantiates, configures, and otherwise assembles a prototype object and hands it to the client, with no further record of that prototype instance. Thus, although initialization lifecycle callback methods are called on all objects regardless of scope, in the case of prototypes, configured destruction lifecycle callbacks are not called. The client code must clean up prototype-scoped objects and release expensive resources that the prototype beans hold. To get the Spring container to release resources held by prototype-scoped beans, try using a custom [bean post-processor](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-extension-bpp), which holds a reference to beans that need to be cleaned up.

与其他范围相比，Spring不管理原型bean的完整生命周期。容器实例化，配置和组装原型对象并将其交给客户端，而没有该原型实例的进一步记录。因此，尽管无论范围如何都在所有对象上调用初始化生命周期回调方法，但在原型的情况下，不会调用已配置的销毁生命周期回调。客户端代码必须清理原型范围的对象并释放原型bean所拥有的昂贵资源。要使Spring容器释放原型范围的bean所拥有的资源，请尝试使用自定义[bean后处理器](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-extension-bpp)，它包含对需要清理的bean的引用。

In some respects, the Spring container’s role in regard to a prototype-scoped bean is a replacement for the Java new operator. All lifecycle management past that point must be handled by the client. (For details on the lifecycle of a bean in the Spring container, see [Lifecycle Callbacks](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle).)

在某些方面，Spring容器关于原型范围bean的角色是Java new运算符的替代品。超过该点的所有生命周期管理必须由客户端处理。（有关Spring容器中bean的生命周期的详细信息，请参阅[Lifecycle Callbacks](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle)。）

1.5.3. Singleton Beans with Prototype-bean Dependencies

When you use singleton-scoped beans with dependencies on prototype beans, be aware that dependencies are resolved at instantiation time. Thus, if you dependency-inject a prototype-scoped bean into a singleton-scoped bean, a new prototype bean is instantiated and then dependency-injected into the singleton bean. The prototype instance is the sole instance that is ever supplied to the singleton-scoped bean.

当您使用具有依赖于原型bean的单例作用域bean时，请注意在实例化时解析依赖项。因此，如果依赖项将原型范围的bean注入到单例范围的bean中，则会实例化一个新的原型bean，然后将依赖注入到单例bean中。原型实例是唯一提供给单例范围bean的实例。

However, suppose you want the singleton-scoped bean to acquire a new instance of the prototype-scoped bean repeatedly at runtime. You cannot dependency-inject a prototype-scoped bean into your singleton bean, because that injection occurs only once, when the Spring container instantiates the singleton bean and resolves and injects its dependencies. If you need a new instance of a prototype bean at runtime more than once, see [Method Injection](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-method-injection)

但是，假设您希望单例范围的bean在运行时重复获取原型范围的bean的新实例。您不能将原型范围的bean依赖注入到您的单例bean中，因为当Spring容器实例化单例bean并解析并注入其依赖项时，该注入只发生一次。如果您需要在运行时多次使用原型bean的新实例，请参阅[方法注入](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-method-injection)

1.5.4. Request, Session, Application, and WebSocket Scopes

The request, session, application, and websocket scopes are available only if you use a web-aware Spring ApplicationContext implementation (such as XmlWebApplicationContext). If you use these scopes with regular Spring IoC containers, such as the ClassPathXmlApplicationContext, an IllegalStateException that complains about an unknown bean scope is thrown.

在request，session，application，和websocket范围只有当你使用一个基于web的Spring可ApplicationContext实现（例如XmlWebApplicationContext）。如果将这些范围与常规的Spring IoC容器一起使用，例如ClassPathXmlApplicationContext，IllegalStateException则会引发抱怨未知Bean范围的问题。

Initial Web Configuration

To support the scoping of beans at the request, session, application, and websocket levels (web-scoped beans), some minor initial configuration is required before you define your beans. (This initial setup is not required for the standard scopes: singleton and prototype.)

为了支持豆的范围界定在request，session，application，和 websocket（即具有web作用域bean），需要做少量的初始配置定义你的豆之前。（标准范围不需要此初始设置：singleton和prototype。）

How you accomplish this initial setup depends on your particular Servlet environment.

如何完成此初始设置取决于您的特定Servlet环境。

If you access scoped beans within Spring Web MVC, in effect, within a request that is processed by the Spring DispatcherServlet, no special setup is necessary. DispatcherServlet already exposes all relevant state.

如果您在Spring Web MVC中访问scoped bean，实际上是在Spring处理的请求中，则DispatcherServlet无需进行特殊设置。DispatcherServlet已暴露所有相关国家。

If you use a Servlet 2.5 web container, with requests processed outside of Spring’s DispatcherServlet (for example, when using JSF or Struts), you need to register the org.springframework.web.context.request.RequestContextListenerServletRequestListener. For Servlet 3.0+, this can be done programmatically by using the WebApplicationInitializerinterface. Alternatively, or for older containers, add the following declaration to your web application’s web.xml file:

如果您使用Servlet 2.5 Web容器，并且在Spring之外处理请求 DispatcherServlet（例如，使用JSF或Struts时），则需要注册org.springframework.web.context.request.RequestContextListener ServletRequestListener。对于Servlet 3.0+，可以使用该WebApplicationInitializer 接口以编程方式完成。或者，或者对于旧容器，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>

Alternatively, if there are issues with your listener setup, consider using Spring’s RequestContextFilter. The filter mapping depends on the surrounding web application configuration, so you have to change it as appropriate. The following listing shows the filter part of a web application:

或者，如果您的侦听器设置存在问题，请考虑使用Spring 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。以下清单显示了Web应用程序的过滤器部分：

<web-app>

...

<filter>

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

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

</filter>

<filter-mapping>

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

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

</filter-mapping>

...

</web-app>

DispatcherServlet, RequestContextListener, and RequestContextFilter all do exactly the same thing, namely bind the HTTP request object to the Thread that is servicing that request. This makes beans that are request- and session-scoped available further down the call chain.

DispatcherServlet，RequestContextListener和RequestContextFilter所有做同样的事情，即将HTTP请求对象绑定到Thread为该请求提供服务的对象。这使得请求和会话范围的bean可以在调用链的下游进一步使用。

Request scope

Consider the following XML configuration for a bean definition:

考虑bean定义的以下XML配置：

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

The Spring container creates a new instance of the LoginAction bean by using the loginAction bean definition for each and every HTTP request. That is, the loginAction bean is scoped at the HTTP request level. You can change the internal state of the instance that is created as much as you want, because other instances created from the same loginAction bean definition do not see these changes in state. They are particular to an individual request. When the request completes processing, the bean that is scoped to the request is discarded.

Spring容器LoginAction通过loginAction对每个HTTP请求使用bean定义来创建bean 的新实例。也就是说， loginActionbean的范围是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loginActionbean定义创建的其他实例在状态中看不到这些更改。它们特别针对个人要求。当请求完成处理时，将放弃作用于请求的bean。

When using annotation-driven components or Java configuration, the @RequestScope annotation can be used to assign a component to the request scope. The following example shows how to do so:

使用注释驱动的组件或Java配置时，@RequestScope注释可用于将组件分配给request范围。以下示例显示了如何执行此操作：

@RequestScope

@Component

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

*// ...*

}

Session Scope

Consider the following XML configuration for a bean definition:

考虑bean定义的以下XML配置：

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

The Spring container creates a new instance of the UserPreferences bean by using the userPreferences bean definition for the lifetime of a single HTTP Session. In other words, the userPreferences bean is effectively scoped at the HTTP Session level. As with request-scoped beans, you can change the internal state of the instance that is created as much as you want, knowing that other HTTP Session instances that are also using instances created from the same userPreferences bean definition do not see these changes in state, because they are particular to an individual HTTP Session. When the HTTP Session is eventually discarded, the bean that is scoped to that particular HTTP Session is also discarded.

Spring容器UserPreferences通过在userPreferences单个HTTP的生存期内使用bean定义来创建bean 的新实例Session。换句话说，userPreferencesbean在HTTP Session级别上有效地作用域。与请求范围的bean一样，您可以根据需要更改创建的实例的内部状态，因为知道Session同样使用从同一userPreferencesbean定义创建的实例的其他HTTP 实例在状态中看不到这些更改，因为它们特定于单个HTTP Session。当Session最终丢弃HTTP时Session，也将丢弃作用于该特定HTTP的bean 。

When using annotation-driven components or Java configuration, you can use the @SessionScope annotation to assign a component to the session scope.

使用注释驱动的组件或Java配置时，可以使用 @SessionScope注释将组件分配给session范围。

@SessionScope

@Component

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

*// ...*

}

Application Scope

Consider the following XML configuration for a bean definition:

考虑bean定义的以下XML配置：

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

The Spring container creates a new instance of the AppPreferences bean by using the appPreferences bean definition once for the entire web application. That is, the appPreferences bean is scoped at the ServletContext level and stored as a regularServletContext attribute. This is somewhat similar to a Spring singleton bean but differs in two important ways: It is a singleton per ServletContext, not per Spring 'ApplicationContext' (for which there may be several in any given web application), and it is actually exposed and therefore visible as a ServletContext attribute.

Spring容器AppPreferences通过appPreferences对整个Web应用程序使用一次bean定义来创建bean 的新实例。也就是说，appPreferencesbean在该ServletContext级别作用域并存储为常规 ServletContext属性。这有点类似于Spring单例bean，但在两个重要方面有所不同：它是一个单独的ServletContext，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，因此是可见的作为一个ServletContext属性。

When using annotation-driven components or Java configuration, you can use the @ApplicationScope annotation to assign a component to the application scope. The following example shows how to do so:

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

@ApplicationScope

@Component

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

*// ...*

}

Scoped Beans as Dependencies

The Spring IoC container manages not only the instantiation of your objects (beans), but also the wiring up of collaborators (or dependencies). If you want to inject (for example) an HTTP request-scoped bean into another bean of a longer-lived scope, you may choose to inject an AOP proxy in place of the scoped bean. That is, you need to inject a proxy object that exposes the same public interface as the scoped object but that can also retrieve the real target object from the relevant scope (such as an HTTP request) and delegate method calls onto the real object.

Spring IoC容器不仅管理对象（bean）的实例化，还管理协作者（或依赖关系）的连接。如果要将（例如）HTTP请求范围的bean注入到寿命较长范围的另一个bean中，您可以选择注入AOP代理来代替范围内的bean。也就是说，您需要注入一个代理对象，该对象公开与范围对象相同的公共接口，但也可以从相关范围（例如HTTP请求）中检索真实目标对象，并将方法调用委托给真实对象。

|  |  |
| --- | --- |
|  | You may also use <aop:scoped-proxy/> between beans that are scoped as singleton, with the reference then going through an intermediate proxy that is serializable and therefore able to re-obtain the target singleton bean on deserialization.  您还可以<aop:scoped-proxy/>在作用域的bean之间使用singleton，然后通过引用然后通过可序列化的中间代理，从而能够在反序列化时重新获取目标单例bean。  When declaring <aop:scoped-proxy/> against a bean of scope prototype, every method call on the shared proxy leads to the creation of a new target instance to which the call is then being forwarded.  当声明<aop:scoped-proxy/>范围的bean时prototype，共享代理上的每个方法调用都会导致创建一个新的目标实例，然后转发该调用。  Also, scoped proxies are not the only way to access beans from shorter scopes in a lifecycle-safe fashion. You may also declare your injection point (that is, the constructor or setter argument or autowired field) as ObjectFactory<MyTargetBean>, allowing for a getObject() call to retrieve the current instance on demand every time it is needed — without holding on to the instance or storing it separately.  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。您还可以将注入点（即构造函数或setter参数或autowired字段）声明为ObjectFactory<MyTargetBean>允许getObject()调用，以便在每次需要时按需检索当前实例 - 无需保留实例或单独存储它。  As an extended variant, you may declare ObjectProvider<MyTargetBean>, which delivers several additional access variants, including getIfAvailable and getIfUnique.  作为扩展变体，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。  The JSR-330 variant of this is called Provider and is used with a Provider<MyTargetBean> declaration and a corresponding get() call for every retrieval attempt. See [here](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-standard-annotations) for more details on JSR-330 overall.  调用它的JSR-330变体，Provider并与每次检索尝试的Provider<MyTargetBean> 声明和相应get()调用一起使用。有关JSR-330整体的更多详细信息，请参见[此处](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-standard-annotations)。 |

The configuration in the following example is only one line, but it is important to understand the “why” as well as the “how” behind it:

以下示例中的配置只有一行，但了解“为什么”以及它背后的“如何”非常重要：

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

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

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

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

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

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

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

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

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

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

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

<aop:scoped-proxy/>

</bean>

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

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

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

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

</bean>

</beans>

|  |  |
| --- | --- |
|  | The line that defines the proxy. |
|  | 定义代理的行。 |

To create such a proxy, you insert a child <aop:scoped-proxy/> element into a scoped bean definition (see [Choosing the Type of Proxy to Create](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-other-injection-proxies) and [XML Schema-based configuration](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#xsd-schemas)). Why do definitions of beans scoped at the request, session and custom-scope levels require the <aop:scoped-proxy/> element? Consider the following singleton bean definition and contrast it with what you need to define for the aforementioned scopes (note that the following userPreferences bean definition as it stands is incomplete):

要创建此类代理，请将子<aop:scoped-proxy/>元素插入到作用域bean定义中（请参阅[选择要创建的代理类型](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-other-injection-proxies)和 [基于XML架构的配置](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#xsd-schemas)）。豆类的定义为何作用域的request，session和自定义范围水平要求<aop:scoped-proxy/>元素？考虑以下单例bean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 userPreferencesbean定义不完整）：

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

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

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

</bean>

In the preceding example, the singleton bean (userManager) is injected with a reference to the HTTP Session-scoped bean (userPreferences). The salient point here is that the userManager bean is a singleton: it is instantiated exactly once per container, and its dependencies (in this case only one, the userPreferences bean) are also injected only once. This means that the userManager bean operates only on the exact same userPreferences object (that is, the one with which it was originally injected.

在前面的示例中，singleton bean（userManager）注入了对HTTP Session-scoped bean（userPreferences）的引用。这里的重点是 userManagerbean是一个单例：它每个容器只实例化一次，它的依赖关系（在这种情况下只有一个，userPreferencesbean）也只注入一次。这意味着userManagerbean只在完全相同的userPreferences对象（即最初注入它的对象）上运行。

This is not the behavior you want when injecting a shorter-lived scoped bean into a longer-lived scoped bean (for example, injecting an HTTP Session-scoped collaborating bean as a dependency into singleton bean). Rather, you need a single userManager object, and, for the lifetime of an HTTP Session, you need a userPreferences object that is specific to the HTTP Session. Thus, the container creates an object that exposes the exact same public interface as the UserPreferences class (ideally an object that is a UserPreferences instance), which can fetch the real UserPreferences object from the scoping mechanism (HTTP request, Session, and so forth). The container injects this proxy object into the userManager bean, which is unaware that this UserPreferences reference is a proxy. In this example, when a UserManager instance invokes a method on the dependency-injected UserPreferences object, it is actually invoking a method on the proxy. The proxy then fetches the realUserPreferences object from (in this case) the HTTP Session and delegates the method invocation onto the retrieved real UserPreferences object.

当将一个寿命较短的scoped bean注入一个寿命较长的scoped bean时，这不是你想要的行为（例如，将一个HTTP Session-scoped协作bean作为依赖注入到singleton bean中）。相反，您需要一个userManager 对象，并且，在HTTP的生命周期中Session，您需要一个userPreferences特定于HTTP 的对象Session。因此，容器创建一个对象，该对象公开与UserPreferences该类完全相同的公共接口（理想情况下是一个UserPreferences实例的对象），该UserPreferences对象可以从作用域机制（HTTP请求Session等）中获取真实对象。容器将此代理对象注入到userManagerbean中，该bean不知道此UserPreferences引用是代理。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，它实际上是在代理上调用一个方法。然后，代理UserPreferences从（在这种情况下）HTTP中Session获取真实UserPreferences对象，并将方法调用委托给检索到的真实对象。

Thus, you need the following (correct and complete) configuration when injecting request- and session-scoped beans into collaborating objects, as the following example shows:

因此，在将bean request-和session-scopedbean注入协作对象时，您需要以下（正确和完整）配置 ，如以下示例所示：

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

<aop:scoped-proxy/>

</bean>

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

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

</bean>

Choosing the Type of Proxy to Create

By default, when the Spring container creates a proxy for a bean that is marked up with the <aop:scoped-proxy/> element, a CGLIB-based class proxy is created.

选择要创建的代理类型

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

|  |  |
| --- | --- |
|  | CGLIB proxies intercept only public method calls! Do not call non-public methods on such a proxy. They are not delegated to the actual scoped target object.  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。它们不会委托给实际的作用域目标对象。 |

Alternatively, you can configure the Spring container to create standard JDK interface-based proxies for such scoped beans, by specifying false for the value of the proxy-target-class attribute of the <aop:scoped-proxy/> element. Using JDK interface-based proxies means that you do not need additional libraries in your application classpath to affect such proxying. However, it also means that the class of the scoped bean must implement at least one interface and that all collaborators into which the scoped bean is injected must reference the bean through one of its interfaces. The following example shows a proxy based on an interface:

或者，您可以通过指定元素属性false的值，将Spring容器配置为为此类作用域bean创建基于JDK接口的标准代理。使用基于JDK接口的代理意味着您不需要在应用程序类路径中使用其他库来影响此类代理。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的所有协作者必须通过其中一个接口引用bean。以下示例显示了基于接口的代理：proxy-target-class<aop:scoped-proxy/>

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

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

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

</bean>

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

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

</bean>

For more detailed information about choosing class-based or interface-based proxying, see [Proxying Mechanisms](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-proxying).

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[代理机制](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-proxying)。

1.5.5. Custom Scopes

The bean scoping mechanism is extensible. You can define your own scopes or even redefine existing scopes, although the latter is considered bad practice and you cannot override the built-in singleton and prototype scopes.

bean范围机制是可扩展的。您可以定义自己的范围，甚至可以重新定义现有范围，尽管后者被认为是不好的做法，您无法覆盖内置singleton和prototype范围。

Creating a Custom Scope

To integrate your custom scopes into the Spring container, you need to implement theorg.springframework.beans.factory.config.Scope interface, which is described in this section. For an idea of how to implement your own scopes, see the Scope implementations that are supplied with the Spring Framework itself and the [Scope](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/beans/factory/config/Scope.html) javadoc, which explains the methods you need to implement in more detail.

要将自定义作用域集成到Spring容器中，需要实现org.springframework.beans.factory.config.Scope本节中描述的 接口。有关如何实现自己的作用域的想法，请参阅Scope Spring Framework本身和[Scope](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/beans/factory/config/Scope.html)javadoc 提供的实现 ，它们解释了您需要更详细地实现的方法。

The Scope interface has four methods to get objects from the scope, remove them from the scope, and let them be destroyed.

该Scope接口有四种方法可以从作用域中获取对象，将其从作用域中删除，然后将其销毁。

The session scope implementation, for example, returns the session-scoped bean (if it does not exist, the method returns a new instance of the bean, after having bound it to the session for future reference). The following method returns the object from the underlying scope:

例如，会话范围实现返回会话范围的bean（如果它不存在，则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）。以下方法从基础范围返回对象：

Object get(String name, ObjectFactory objectFactory)

The session scope implementation, for example, removes the session-scoped bean from the underlying session. The object should be returned, but you can return null if the object with the specified name is not found. The following method removes the object from the underlying scope:

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

Object remove(String name)

The following method registers the callbacks the scope should execute when it is destroyed or when the specified object in the scope is destroyed:

以下方法记录范围在销毁时或范围中指定对象被销毁时应执行的回调：

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

See the [javadoc](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/beans/factory/config/Scope.html#registerDestructionCallback) or a Spring scope implementation for more information on destruction callbacks.

有关 销毁回调的更多信息，请参阅[javadoc](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/beans/factory/config/Scope.html#registerDestructionCallback)或Spring作用域实现。

The following method obtains the conversation identifier for the underlying scope:

以下方法获取基础范围的对话标识符：

String getConversationId()

This identifier is different for each scope. For a session scoped implementation, this identifier can be the session identifier.

每个范围的标识符都不同。对于会话范围的实现，该标识符可以是会话标识符。

Using a Custom Scope

After you write and test one or more custom Scope implementations, you need to make the Spring container aware of your new scopes. The following method is the central method to register a new Scope with the Spring container:

在编写并测试一个或多个自定义Scope实现之后，您需要让Spring容器知道您的新范围。以下方法是Scope使用Spring容器注册new的核心方法：

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

This method is declared on the ConfigurableBeanFactory interface, which is available through the BeanFactory property on most of the concrete ApplicationContext implementations that ship with Spring.

此方法在ConfigurableBeanFactory接口上声明，该接口可通过 Spring随附的BeanFactory大多数具体ApplicationContext实现的属性获得。

The first argument to the registerScope(..) method is the unique name associated with a scope. Examples of such names in the Spring container itself are singleton and prototype. The second argument to the registerScope(..) method is an actual instance of the custom Scope implementation that you wish to register and use.

该registerScope(..)方法的第一个参数是与范围关联的唯一名称。Spring容器本身中的这些名称的示例是singleton和 prototype。该registerScope(..)方法的第二个参数是Scope您希望注册和使用的自定义实现的实际实例。

Suppose that you write your custom Scope implementation, and then register it as shown in the next example.

假设您编写自定义Scope实现，然后注册它，如下一个示例所示。

|  |  |
| --- | --- |
|  | The next example uses SimpleThreadScope, which is included with Spring but is not registered by default. The instructions would be the same for your own custom Scope implementations.  下一个示例使用SimpleThreadScope，它包含在Spring中，但默认情况下未注册。您自己的自定义Scope 实现的说明是相同的。 |

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

beanFactory.registerScope("thread", threadScope);

You can then create bean definitions that adhere to the scoping rules of your custom Scope, as follows:

然后，您可以创建符合自定义的作用域规则的bean定义， Scope如下所示：

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

With a custom Scope implementation, you are not limited to programmatic registration of the scope. You can also do the Scoperegistration declaratively, by using the CustomScopeConfigurer class, as the following example shows:

使用自定义Scope实现，您不仅限于范围的编程注册。您还可以Scope使用CustomScopeConfigurer该类以声明方式进行注册 ，如以下示例所示：

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

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

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

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

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

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

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

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

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

<property name="scopes">

<map>

<entry key="thread">

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

</entry>

</map>

</property>

</bean>

<bean id="thing2" class="x.y.Thing2" scope="thread">

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

<aop:scoped-proxy/>

</bean>

<bean id="thing1" class="x.y.Thing1">

<property name="thing2" ref="thing2"/>

</bean>

</beans>

|  |  |
| --- | --- |
|  | When you place <aop:scoped-proxy/> in a FactoryBean implementation, it is the factory bean itself that is scoped, not the object returned from getObject().  放置<aop:scoped-proxy/>在FactoryBean实现中时，工厂bean本身是作用域的，而不是从中返回的对象getObject()。 |

1.6. Customizing the Nature of a Bean

The Spring Framework provides a number of interfaces you can use to customize the nature of a bean. This section groups them as follows:

Spring Framework提供了许多可用于自定义bean特性的接口。本节将它们分组如下：

* [Lifecycle Callbacks](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle)
* [ApplicationContextAware and BeanNameAware](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-aware)
* [Other Aware Interfaces](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aware-list)
* [生命周期回调](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle)
* [ApplicationContextAware 和 BeanNameAware](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-aware)
* [其他Aware接口](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aware-list)

1.6.1. Lifecycle Callbacks

To interact with the container’s management of the bean lifecycle, you can implement the Spring InitializingBean and DisposableBean interfaces. The container calls afterPropertiesSet() for the former and destroy() for the latter to let the bean perform certain actions upon initialization and destruction of your beans.

要与容器的bean生命周期管理进行交互，可以实现Spring InitializingBean和DisposableBean接口。容器调用afterPropertiesSet()前者，destroy()后者让bean在初始化和销毁​​bean时执行某些操作。

|  |  |
| --- | --- |
|  | The JSR-250 @PostConstruct and @PreDestroy annotations are generally considered best practice for receiving lifecycle callbacks in a modern Spring application. Using these annotations means that your beans are not coupled to Spring-specific interfaces. For details, see [Using @PostConstruct and @PreDestroy](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-postconstruct-and-predestroy-annotations).  JSR-250 @PostConstruct和@PreDestroy注释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。使用这些注释意味着您的bean不会耦合到特定于Spring的接口。有关详细信息，请参阅[使用@PostConstruct和@PreDestroy](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-postconstruct-and-predestroy-annotations)。  If you do not want to use the JSR-250 annotations but you still want to remove coupling, consider init-methodand destroy-method bean definition metadata.  如果您不想使用JSR-250注释但仍想删除耦合，请考虑init-method和destroy-methodbean定义元数据。 |

Internally, the Spring Framework uses BeanPostProcessor implementations to process any callback interfaces it can find and call the appropriate methods. If you need custom features or other lifecycle behavior Spring does not by default offer, you can implement a BeanPostProcessor yourself. For more information, see [Container Extension Points](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-extension).

在内部，Spring Framework使用BeanPostProcessor实现来处理它可以找到的任何回调接口并调用适当的方法。如果您需要自定义功能或其他生命周期行为Spring默认不提供，您可以BeanPostProcessor自己实现。有关更多信息，请参阅 [容器扩展点](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-extension)。

In addition to the initialization and destruction callbacks, Spring-managed objects may also implement the Lifecycle interface so that those objects can participate in the startup and shutdown process, as driven by the container’s own lifecycle.

除了初始化和销毁​​回调之外，Spring管理的对象还可以实现Lifecycle接口，以便这些对象可以参与启动和关闭过程，这是由容器自身的生命周期驱动的。

The lifecycle callback interfaces are described in this section.

本节描述了生命周期回调接口。

Initialization Callbacks

The org.springframework.beans.factory.InitializingBean interface lets a bean perform initialization work after the container has set all necessary properties on the bean. The InitializingBean interface specifies a single method:

该org.springframework.beans.factory.InitializingBean接口允许在容器上设置bean的所有必要属性后，一个bean进行初始化工作。的InitializingBean接口规定了一个方法：

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

We recommend that you do not use the InitializingBean interface, because it unnecessarily couples the code to Spring. Alternatively, we suggest using the [@PostConstruct](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-postconstruct-and-predestroy-annotations) annotation or specifying a POJO initialization method. In the case of XML-based configuration metadata, you can use the init-method attribute to specify the name of the method that has a void no-argument signature. With Java configuration, you can use the initMethod attribute of @Bean. See [Receiving Lifecycle Callbacks](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-java-lifecycle-callbacks). Consider the following example:

我们建议您不要使用该InitializingBean接口，因为它会不必要地将代码耦合到Spring。或者，我们建议使用[@PostConstruct](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-postconstruct-and-predestroy-annotations)注释或指定POJO初始化方法。对于基于XML的配置元数据，您可以使用该init-method属性指定具有void无参数签名的方法的名称。使用Java配置，您可以使用。的initMethod属性 @Bean。请参阅[接收生命周期回调](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-java-lifecycle-callbacks)。请考虑以下示例：

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

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

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

*// do some initialization work*

}

}

The preceding example has almost exactly the same effect as the following example (which consists of two listings):

前面的示例与以下示例几乎完全相同（包含两个列表）：

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

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

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

*// do some initialization work*

}

}

However, the first of the two preceding examples does not couple the code to Spring.

但是，前面两个示例中的第一个没有将代码耦合到Spring。

Destruction Callbacks

Implementing the org.springframework.beans.factory.DisposableBean interface lets a bean get a callback when the container that contains it is destroyed. The DisposableBean interface specifies a single method:

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。的 DisposableBean接口规定了一个方法：

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

We recommend that you do not use the DisposableBean callback interface, because it unnecessarily couples the code to Spring. Alternatively, we suggest using the [@PreDestroy](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-postconstruct-and-predestroy-annotations) annotation or specifying a generic method that is supported by bean definitions. With XML-based configuration metadata, you can use the destroy-method attribute on the <bean/>. With Java configuration, you can use the destroyMethod attribute of @Bean. See [Receiving Lifecycle Callbacks](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-java-lifecycle-callbacks). Consider the following definition:

我们建议您不要使用DisposableBean回调接口，因为它会不必要地将代码耦合到Spring。或者，我们建议使用[@PreDestroy](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-postconstruct-and-predestroy-annotations)注释或指定bean定义支持的泛型方法。使用基于XML的配置元数据，您可以使用该destroy-method属性<bean/>。使用Java配置，您可以使用。的destroyMethod属性@Bean。请参阅 [接收生命周期回调](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-java-lifecycle-callbacks)。考虑以下定义：

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

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

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

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

}

}

The preceding definition has almost exactly the same effect as the following definition:

前面的定义与以下定义几乎完全相同：

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

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

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

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

}

}

However, the first of the two preceding definitions does not couple the code to Spring.

但是，前面两个定义中的第一个没有将代码耦合到Spring。

|  |  |
| --- | --- |
|  | You can assign the destroy-method attribute of a <bean> element a special (inferred) value, which instructs Spring to automatically detect a public close or shutdown method on the specific bean class. (Any class that implements java.lang.AutoCloseable or java.io.Closeable would therefore match.) You can also set this special (inferred) value on the default-destroy-method attribute of a <beans> element to apply this behavior to an entire set of beans (see [Default Initialization and Destroy Methods](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-default-init-destroy-methods)). Note that this is the default behavior with Java configuration.  您可以destroy-method为<bean>元素的属性指定一个特殊 (inferred)值，该值指示Spring自动检测特定bean类的公共close或 shutdown方法。（任何实现 java.lang.AutoCloseable或java.io.Closeable因此匹配的类。）您还可以(inferred)在元素的default-destroy-method属性 上设置此特殊值，<beans>以将此行为应用于整组bean（请参阅[默认初始化和销毁​​方法](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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

When you write initialization and destroy method callbacks that do not use the Spring-specific InitializingBean and DisposableBean callback interfaces, you typically write methods with names such as init(), initialize(), dispose(), and so on. Ideally, the names of such lifecycle callback methods are standardized across a project so that all developers use the same method names and ensure consistency.

当你写的初始化和销毁不使用Spring的具体方法回调InitializingBean和DisposableBean回调接口，你通常写有名字，如方法init()，initialize()，dispose()，等等。理想情况下，此类生命周期回调方法的名称在项目中是标准化的，以便所有开发人员使用相同的方法名称并确保一致性。

You can configure the Spring container to “look” for named initialization and destroy callback method names on every bean. This means that you, as an application developer, can write your application classes and use an initialization callback calledinit(), without having to configure an init-method="init" attribute with each bean definition. The Spring IoC container calls that method when the bean is created (and in accordance with the standard lifecycle callback contract [described previously](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle)). This feature also enforces a consistent naming convention for initialization and destroy method callbacks.

您可以将Spring容器配置为“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调用的初始化回调 init()，而无需为init-method="init"每个bean定义配置属性。Spring IoC容器在创建bean时调用该方法（并且符合[前面描述](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle)的标准生命周期回调协定）。此功能还强制执行初始化和销毁​​方法回调的一致命名约定。

Suppose that your initialization callback methods are named init() and your destroy callback methods are named destroy(). Your class then resembles the class in the following example:

假设您的初始化回调方法已命名，init()并且您的destroy回调方法已命名destroy()。然后，您的类类似于以下示例中的类：

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

**private** BlogDao blogDao;

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

this.blogDao = blogDao;

}

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

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

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

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

}

}

}

You could then use that class in a bean resembling the following:

然后，您可以在类似于以下内容的bean中使用该类：

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

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

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

</bean>

</beans>

The presence of the default-init-method attribute on the top-level <beans/> element attribute causes the Spring IoC container to recognize a method called init on the bean class as the initialization method callback. When a bean is created and assembled, if the bean class has such a method, it is invoked at the appropriate time.

default-init-method顶级<beans/>元素属性上存在属性会导致Spring IoC容器init将bean类上调用的方法识别为初始化方法回调。当bean被创建和组装时，如果bean类具有这样的方法，则在适当的时候调用它。

You can configure destroy method callbacks similarly (in XML, that is) by using the default-destroy-method attribute on the top-level <beans/> element.

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

Where existing bean classes already have callback methods that are named at variance with the convention, you can override the default by specifying (in XML, that is) the method name by using the init-method and destroy-method attributes of the <bean/> itself.

如果现有的bean类已经具有与约定一致的回调方法，则可以通过使用 自身的init-method和destroy-method属性指定（在XML中，即方法名称）来覆盖默认值<bean/>。

The Spring container guarantees that a configured initialization callback is called immediately after a bean is supplied with all dependencies. Thus, the initialization callback is called on the raw bean reference, which means that AOP interceptors and so forth are not yet applied to the bean. A target bean is fully created first and then an AOP proxy (for example) with its interceptor chain is applied. If the target bean and the proxy are defined separately, your code can even interact with the raw target bean, bypassing the proxy. Hence, it would be inconsistent to apply the interceptors to the init method, because doing so would couple the lifecycle of the target bean to its proxy or interceptors and leave strange semantics when your code interacts directly with the raw target bean.

Spring容器保证在为bean提供所有依赖项后立即调用已配置的初始化回调。因此，在原始bean引用上调用初始化回调，这意味着AOP拦截器等尚未应用于bean。首先完全创建目标bean，然后应用带有拦截器链的AOP代理（例如）。如果目标bean和代理是分开定义的，那么您的代码甚至可以绕过代理与原始目标bean交互。因此，将拦截器应用于init方法是不一致的，因为这样做会将目标bean的生命周期耦合到其代理或拦截器，并在代码直接与原始目标bean交互时留下奇怪的语义。

Combining Lifecycle Mechanisms

As of Spring 2.5, you have three options for controlling bean lifecycle behavior:

从Spring 2.5开始，您有三个控制bean生命周期行为的选项：

* The [InitializingBean](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-initializingbean) and [DisposableBean](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-disposablebean) callback interfaces
* Custom init() and destroy() methods
* The [@PostConstruct and @PreDestroy annotations](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-postconstruct-and-predestroy-annotations). You can combine these mechanisms to control a given bean.
* 在[InitializingBean](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-initializingbean)和 [DisposableBean](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-disposablebean)回调接口
* 定制init()和destroy()方法
* 在[@PostConstruct和@PreDestroy 注释](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-postconstruct-and-predestroy-annotations)。您可以组合这些机制来控制给定的bean。

|  |  |
| --- | --- |
|  | If multiple lifecycle mechanisms are configured for a bean and each mechanism is configured with a different method name, then each configured method is executed in the order listed after this note. However, if the same method name is configured — for example, init() for an initialization method — for more than one of these lifecycle mechanisms, that method is executed once, as explained in the [preceding section](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-default-init-destroy-methods).  如果为bean配置了多个生命周期机制，并且每个机制都配置了不同的方法名称，则每个配置的方法都按照此注释后列出的顺序执行。但是，如果init()为多个这些生命周期机制配置了相同的方法名称（例如， 对于初始化方法），则该方法将执行一次，如上 [一节中所述](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-default-init-destroy-methods)。 |

Multiple lifecycle mechanisms configured for the same bean, with different initialization methods, are called as follows:

为同一个bean配置的多个生命周期机制具有不同的初始化方法，如下所示：

1. Methods annotated with @PostConstruct
2. afterPropertiesSet() as defined by the InitializingBean callback interface
3. A custom configured init() method
4. 用注释方法注释 @PostConstruct
5. afterPropertiesSet()由InitializingBean回调接口定义
6. 自定义配置的init()方法

Destroy methods are called in the same order:

1. Methods annotated with @PreDestroy
2. destroy() as defined by the DisposableBean callback interface
3. A custom configured destroy() method

Destroy方法以相同的顺序调用：

1. 用注释方法注释 @PreDestroy
2. destroy()由DisposableBean回调接口定义
3. 自定义配置的destroy()方法

Startup and Shutdown Callbacks

The Lifecycle interface defines the essential methods for any object that has its own lifecycle requirements (such as starting and stopping some background process):

该Lifecycle接口为任何具有自己的生命周期要求的对象（例如启动和停止某些后台进程）定义了基本方法：

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

**void** start();

**void** stop();

**boolean** isRunning();

}

Any Spring-managed object may implement the Lifecycle interface. Then, when the ApplicationContext itself receives start and stop signals (for example, for a stop/restart scenario at runtime), it cascades those calls to all Lifecycle implementations defined within that context. It does this by delegating to a LifecycleProcessor, shown in the following listing:

任何Spring管理的对象都可以实现该Lifecycle接口。然后，当它 ApplicationContext自己接收到启动和停止信号时（例如，对于运行时的停止/重启场景），它将这些调用级联到Lifecycle该上下文中定义的所有实现。它通过委托给a来实现LifecycleProcessor，如下面的清单所示：

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

**void** onRefresh();

**void** onClose();

}

Notice that the LifecycleProcessor is itself an extension of the Lifecycle interface. It also adds two other methods for reacting to the context being refreshed and closed.

请注意，LifecycleProcessor它本身是Lifecycle 接口的扩展。它还添加了另外两种方法来响应刷新和关闭的上下文。

|  |  |
| --- | --- |
|  | Note that the regular org.springframework.context.Lifecycle interface is a plain contract for explicit start and stop notifications and does not imply auto-startup at context refresh time. For fine-grained control over auto-startup of a specific bean (including startup phases), consider implementing org.springframework.context.SmartLifecycle instead.  请注意，常规org.springframework.context.Lifecycle接口是显式启动和停止通知的简单合约，并不意味着在上下文刷新时自动启动。要对特定bean的自动启动（包括启动阶段）进行细粒度控制，请考虑实现org.springframework.context.SmartLifecycle。  Also, please note that stop notifications are not guaranteed to come before destruction. On regular shutdown, all Lifecycle beans first receive a stop notification before the general destruction callbacks are being propagated. However, on hot refresh during a context’s lifetime or on aborted refresh attempts, only destroy methods are called.  此外，请注意，在销毁之前不保证停止通知。在常规关闭时，所有Lifecyclebean在传播一般销毁回调之前首先收到停止通知。但是，在上下文生命周期中的热刷新或中止刷新尝试时，仅调用destroy方法。 |

The order of startup and shutdown invocations can be important. If a “depends-on” relationship exists between any two objects, the dependent side starts after its dependency, and it stops before its dependency. However, at times, the direct dependencies are unknown. You may only know that objects of a certain type should start prior to objects of another type. In those cases, the SmartLifecycle interface defines another option, namely the getPhase() method as defined on its super-interface, Phased. The following listing shows the definition of the Phased interface:

启动和关闭调用的顺序非常重要。如果任何两个对象之间存在“依赖”关系，则依赖方在其依赖之后开始，并且在其依赖之前停止。但是，有时，直接依赖性是未知的。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开始。在这些情况下，SmartLifecycle接口定义了另一个选项，即getPhase()在其超级接口上定义的方法 Phased。以下清单显示了Phased界面的定义：

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

**int** getPhase();

}

The following listing shows the definition of the SmartLifecycle interface:

以下清单显示了SmartLifecycle界面的定义：

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

**boolean** isAutoStartup();

**void** stop(Runnable callback);

}

When starting, the objects with the lowest phase start first. When stopping, the reverse order is followed. Therefore, an object that implements SmartLifecycle and whose getPhase() method returns Integer.MIN\_VALUE would be among the first to start and the last to stop. At the other end of the spectrum, a phase value of Integer.MAX\_VALUE would indicate that the object should be started last and stopped first (likely because it depends on other processes to be running). When considering the phase value, it is also important to know that the default phase for any “normal” Lifecycle object that does not implement SmartLifecycle is 0. Therefore, any negative phase value indicates that an object should start before those standard components (and stop after them). The reverse is true for any positive phase value.

启动时，具有最低相位的对象首先开始。停止时，遵循相反的顺序。因此，实现SmartLifecycle和getPhase()返回其方法的对象Integer.MIN\_VALUE将是第一个开始和最后一个停止的对象。在频谱的另一端，相位值 Integer.MAX\_VALUE将指示对象应该最后启动并首先停止（可能因为它依赖于正在运行的其他进程）。当考虑相位值，同样重要的是要知道，对于任何“正常”的默认阶段 Lifecycle目标没有实现SmartLifecycle的0。因此，任何负相位值都表示对象应该在这些标准组件之前启动（并在它们之后停止）。任何正相值都是相反的。

The stop method defined by SmartLifecycle accepts a callback. Any implementation must invoke that callback’s run()method after that implementation’s shutdown process is complete. That enables asynchronous shutdown where necessary, since the default implementation of the LifecycleProcessor interface, DefaultLifecycleProcessor, waits up to its timeout value for the group of objects within each phase to invoke that callback. The default per-phase timeout is 30 seconds. You can override the default lifecycle processor instance by defining a bean named lifecycleProcessor within the context. If you want only to modify the timeout, defining the following would suffice:

定义的stop方法SmartLifecycle接受回调。任何实现必须run()在该实现的关闭过程完成后调用该回调的方法。这样就可以在必要时启用异步关闭，因为LifecycleProcessor接口 的默认实现DefaultLifecycleProcessor等待每个阶段内的对象组的超时值来调用该回调。默认的每阶段超时为30秒。您可以通过定义lifecycleProcessor在上下文中命名的bean来覆盖缺省生命周期处理器实例 。如果您只想修改超时，则定义以下内容就足够了：

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

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

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

</bean>

As mentioned earlier, the LifecycleProcessor interface defines callback methods for the refreshing and closing of the context as well. The latter drives the shutdown process as if stop() had been called explicitly, but it happens when the context is closing. The 'refresh' callback, on the other hand, enables another feature of SmartLifecycle beans. When the context is refreshed (after all objects have been instantiated and initialized), that callback is invoked. At that point, the default lifecycle processor checks the boolean value returned by each SmartLifecycle object’s isAutoStartup() method. If true, that object is started at that point rather than waiting for an explicit invocation of the context’s or its own start() method (unlike the context refresh, the context start does not happen automatically for a standard context implementation). The phase value and any “depends-on” relationships determine the startup order as described earlier.

如前所述，该LifecycleProcessor接口还定义了用于刷新和关闭上下文的回调方法。后者驱动关闭过程就好像stop()已经显式调用一样，但它在上下文关闭时发生。另一方面，'refresh'回调启用了SmartLifecyclebean的另一个功能 。刷新上下文时（在实例化并初始化所有对象之后），将调用该回调。此时，默认生命周期处理器检查每个SmartLifecycle对象的isAutoStartup()方法返回的布尔值 。如果true，那个对象是在那个点开始的，而不是等待显式调用上下文或它自己的对象start()方法（与上下文刷新不同，上下文启动不会自动发生在标准上下文实现中）。该phase值与任何“依赖式”的关系确定为前面所述的启动顺序。

Shutting Down the Spring IoC Container Gracefully in Non-Web Applications

##### 在非Web应用程序中优雅地关闭Spring IoC容器

|  |  |
| --- | --- |
|  | This section applies only to non-web applications. Spring’s web-based ApplicationContext implementations already have code in place to gracefully shut down the Spring IoC container when the relevant web application is shut down.  本节仅适用于非Web应用程序。Spring的基于Web的 ApplicationContext实现已经具有代码，可以在相关Web应用程序关闭时正常关闭Spring IoC容器。 |

If you use Spring’s IoC container in a non-web application environment (for example, in a rich client desktop environment), register a shutdown hook with the JVM. Doing so ensures a graceful shutdown and calls the relevant destroy methods on your singleton beans so that all resources are released. You must still configure and implement these destroy callbacks correctly.

如果在非Web应用程序环境中使用Spring的IoC容器（例如，在富客户机桌面环境中），请使用JVM注册关闭挂钩。这样做可确保正常关闭并在单例bean上调用相关的destroy方法，以便释放所有资源。您仍然必须正确配置和实现这些destroy回调。

To register a shutdown hook, call the registerShutdownHook() method that is declared on the ConfigurableApplicationContextinterface, as the following example shows:

要注册关闭挂钩，请调用接口registerShutdownHook()上声明的方法ConfigurableApplicationContext，如以下示例所示：

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

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

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

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

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

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

ctx.registerShutdownHook();

*// app runs here...*

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

}

}

1.6.2. ApplicationContextAware and BeanNameAware

When an ApplicationContext creates an object instance that implements theorg.springframework.context.ApplicationContextAware interface, the instance is provided with a reference to that ApplicationContext. The following listing shows the definition of the ApplicationContextAware interface:

当ApplicationContext创建实现org.springframework.context.ApplicationContextAware接口的对象实例时，将 为该实例提供对该实例的引用ApplicationContext。以下清单显示了ApplicationContextAware界面的定义：

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

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

}

Thus, beans can programmatically manipulate the ApplicationContext that created them, through the ApplicationContextinterface or by casting the reference to a known subclass of this interface (such as ConfigurableApplicationContext, which exposes additional functionality). One use would be the programmatic retrieval of other beans. Sometimes this capability is useful. However, in general, you should avoid it, because it couples the code to Spring and does not follow the Inversion of Control style, where collaborators are provided to beans as properties. Other methods of the ApplicationContext provide access to file resources, publishing application events, and accessing a MessageSource. These additional features are described in [Additional Capabilities of the ApplicationContext](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#context-introduction).

因此，bean可以ApplicationContext通过ApplicationContext接口或通过将引用转换为此接口的已知子类（例如ConfigurableApplicationContext，公开其他功能）以编程方式操纵创建它们的方法。一种用途是对其他bean进行编程检索。有时这种能力很有用。但是，一般情况下，您应该避免使用它，因为它将代码耦合到Spring并且不遵循Inversion of Control样式，其中协作者作为属性提供给bean。其他方法 ApplicationContext提供对文件资源的访问，发布应用程序事件和访问MessageSource。这些附加功能在[附加ApplicationContext](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#context-introduction)功能中描述 。

As of Spring 2.5, autowiring is another alternative to obtain a reference to the ApplicationContext. The “traditional” constructor and byType autowiring modes (as described in [Autowiring Collaborators](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-autowire)) can provide a dependency of typeApplicationContext for a constructor argument or a setter method parameter, respectively. For more flexibility, including the ability to autowire fields and multiple parameter methods, use the new annotation-based autowiring features. If you do, the ApplicationContext is autowired into a field, constructor argument, or method parameter that expects the ApplicationContexttype if the field, constructor, or method in question carries the @Autowired annotation. For more information, see [Using @Autowired](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-autowired-annotation).

从Spring 2.5开始，自动装配是另一种获取参考的方法 ApplicationContext。“传统” constructor和byType自动[装配](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-autowire)模式（如[自动装配协作者中所述](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-autowire)）可以分别为ApplicationContext构造函数参数或setter方法参数提供类型的依赖性 。为了获得更大的灵活性，包括自动装配字段和多参数方法的能力，请使用基于注释的新自动装配功能。如果这样做，ApplicationContext则自动装入一个字段，构造函数参数或方法参数，ApplicationContext如果相关的字段，构造函数或方法带有@Autowired注释，则该参数需要该类型。有关更多信息，请参阅 [使用@Autowired](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-autowired-annotation)。

When an ApplicationContext creates a class that implements the org.springframework.beans.factory.BeanNameAwareinterface, the class is provided with a reference to the name defined in its associated object definition. The following listing shows the definition of the BeanNameAware interface:

当ApplicationContext创建实现org.springframework.beans.factory.BeanNameAware接口的类时，将为 该类提供对其关联对象定义中定义的名称的引用。以下清单显示了BeanNameAware接口的定义：

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

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

}

The callback is invoked after population of normal bean properties but before an initialization callback such as InitializingBean, afterPropertiesSet, or a custom init-method.

回调正常bean属性的人口之后，但在一个初始化回调诸如调用InitializingBean，afterPropertiesSet或自定义的初始化方法。

1.6.3. Other Aware Interfaces

Besides ApplicationContextAware and BeanNameAware (discussed [earlier](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-aware)), Spring offers a wide range of Aware callback interfaces that let beans indicate to the container that they require a certain infrastructure dependency. As a general rule, the name indicates the dependency type. The following table summarizes the most important Aware interfaces:

除了ApplicationContextAware和BeanNameAware讨论（[早期](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-aware)），Spring提供了广泛的Aware，让豆子指示，他们需要一定的基础设施的依赖容器回调接口。作为一般规则，名称表示依赖关系类型。下表总结了最重要的Aware接口：

|  |  |  |
| --- | --- | --- |
| *Table 4. Aware interfaces* | | |
| **Name** | **Injected Dependency** | **Explained in…​** |
| ApplicationContextAware | Declaring ApplicationContext. | [ApplicationContextAware and BeanNameAware](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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/5.1.8.RELEASE/spring-framework-reference/core.html#context-introduction) |
| BeanClassLoaderAware | Class loader used to load the bean classes. | [Instantiating Beans](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-class) |
| BeanFactoryAware | Declaring BeanFactory. | [ApplicationContextAware and BeanNameAware](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-aware) |
| BeanNameAware | Name of the declaring bean. | [ApplicationContextAware and BeanNameAware](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-aware) |
| BootstrapContextAware | Resource adapter BootstrapContextthe container runs in. Typically available only in JCA-aware ApplicationContextinstances. | [JCA CCI](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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/5.1.8.RELEASE/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/5.1.8.RELEASE/spring-framework-reference/core.html#context-introduction) |
| NotificationPublisherAware | Spring JMX notification publisher. | [Notifications](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/integration.html#jmx-notifications) |
| ResourceLoaderAware | Configured loader for low-level access to resources. | [Resources](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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/5.1.8.RELEASE/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/5.1.8.RELEASE/spring-framework-reference/web.html#mvc) |

|  |  |  |
| --- | --- | --- |
| *表4.感知接口* | | |
| **名称** | **注入依赖** | **解释在......** |
| ApplicationContextAware | 宣布ApplicationContext。 | [ApplicationContextAware 和 BeanNameAware](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-aware) |
| ApplicationEventPublisherAware | 封闭的事件发布者ApplicationContext。 | [附加功能 ApplicationContext](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#context-introduction) |
| BeanClassLoaderAware | 用于加载bean类的类加载器。 | [实例化豆​​类](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-class) |
| BeanFactoryAware | 宣布BeanFactory。 | [ApplicationContextAware 和 BeanNameAware](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-aware) |
| BeanNameAware | 声明bean的名称。 | [ApplicationContextAware 和 BeanNameAware](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-aware) |
| BootstrapContextAware | BootstrapContext容器运行的资源适配器。通常仅在JCA感知ApplicationContext实例中可用。 | [JCA CCI](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/integration.html#cci) |
| LoadTimeWeaverAware | 定义的weaver用于在加载时处理类定义。 | [在Spring框架中使用AspectJ进行加载时编织](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-aj-ltw) |
| MessageSourceAware | 用于解析消息的已配置策略（支持参数化和国际化）。 | [附加功能 ApplicationContext](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#context-introduction) |
| NotificationPublisherAware | Spring JMX通知发布者。 | [通知](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/integration.html#jmx-notifications) |
| ResourceLoaderAware | 配置的加载程序，用于对资源进行低级访问。 | [资源](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#resources) |
| ServletConfigAware | 当前ServletConfig容器运行。仅在Web感知弹簧中有效 ApplicationContext。 | [Spring MVC](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/web.html#mvc) |
| ServletContextAware | 当前ServletContext容器运行。仅在Web感知弹簧中有效 ApplicationContext。 | [Spring MVC](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/web.html#mvc) |

Note again that using these interfaces ties your code to the Spring API and does not follow the Inversion of Control style. As a result, we recommend them for infrastructure beans that require programmatic access to the container.

请再次注意，使用这些接口会将您的代码绑定到Spring API，而不会遵循Inversion of Control样式。因此，我们建议将它们用于需要以编程方式访问容器的基础架构bean。

1.7. Bean Definition Inheritance

A bean definition can contain a lot of configuration information, including constructor arguments, property values, and container-specific information, such as the initialization method, a static factory method name, and so on. A child bean definition inherits configuration data from a parent definition. The child definition can override some values or add others as needed. Using parent and child bean definitions can save a lot of typing. Effectively, this is a form of templating.

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

If you work with an ApplicationContext interface programmatically, child bean definitions are represented by the ChildBeanDefinition class. Most users do not work with them on this level. Instead, they configure bean definitions declaratively in a class such as the ClassPathXmlApplicationContext. When you use XML-based configuration metadata, you can indicate a child bean definition by using the parent attribute, specifying the parent bean as the value of this attribute. The following example shows how to do so:

如果以ApplicationContext编程方式使用接口，则子bean定义由ChildBeanDefinition类表示。大多数用户不在此级别上使用它们。相反，它们在类中以声明方式配置bean定义ClassPathXmlApplicationContext。使用基于XML的配置元数据时，可以使用该parent属性指定子bean定义，并将父bean指定为此属性的值。以下示例显示了如何执行此操作：

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

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

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

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

</bean>

<bean id="inheritsWithDifferentClass"

class="org.springframework.beans.DerivedTestBean"

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

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

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

</bean>

|  |  |
| --- | --- |
|  | Note the parent attribute. |
|  | 注意parent属性。 |

A child bean definition uses the bean class from the parent definition if none is specified but can also override it. In the latter case, the child bean class must be compatible with the parent (that is, it must accept the parent’s property values).

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

A child bean definition inherits scope, constructor argument values, property values, and method overrides from the parent, with the option to add new values. Any scope, initialization method, destroy method, or static factory method settings that you specify override the corresponding parent settings.

子bean定义从父级继承范围，构造函数参数值，属性值和方法覆盖，并带有添加新值的选项。static您指定的任何范围，初始化方法，销毁方法或工厂方法设置都会覆盖相应的父设置。

The remaining settings are always taken from the child definition: depends on, autowire mode, dependency check, singleton, and lazy init.

其余设置始终取自子定义：取决于，autowire模式，依赖性检查，单例和惰性初始化。

The preceding example explicitly marks the parent bean definition as abstract by using the abstract attribute. If the parent definition does not specify a class, explicitly marking the parent bean definition as abstract is required, as the following example shows:

前面的示例通过使用该abstract属性将父bean定义显式标记为abstract 。如果父定义未指定类，abstract则根据需要显式标记父bean定义，如以下示例所示：

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

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

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

</bean>

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

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

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

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

</bean>

The parent bean cannot be instantiated on its own because it is incomplete, and it is also explicitly marked as abstract. When a definition is abstract, it is usable only as a pure template bean definition that serves as a parent definition for child definitions. Trying to use such an abstract parent bean on its own, by referring to it as a ref property of another bean or doing an explicit getBean() call with the parent bean ID returns an error. Similarly, the container’s internal preInstantiateSingletons() method ignores bean definitions that are defined as abstract.

父bean不能单独实例化，因为它不完整，并且也明确标记为abstract。定义时abstract，它仅可用作纯模板bean定义，用作子定义的父定义。尝试使用这样的abstract父bean，通过将其称为另一个bean的ref属性或getBean()使用父bean ID 进行显式调用，将返回错误。类似地，容器的内部 preInstantiateSingletons()方法忽略定义为abstract的bean定义。

|  |  |
| --- | --- |
|  | ApplicationContext pre-instantiates all singletons by default. Therefore, it is 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 (attempt to) pre-instantiate the abstract bean.  ApplicationContext默认情况下预先实例化所有单例。因此，重要的是（至少对于单例bean），如果你有一个（父）bean定义，你只打算用作模板，并且这个定义指定了一个类，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abstractbean。 |

1.8. Container Extension Points

Typically, an application developer does not need to subclass ApplicationContext implementation classes. Instead, the Spring IoC container can be extended by plugging in implementations of special integration interfaces. The next few sections describe these integration interfaces.

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

1.8.1. Customizing Beans by Using a BeanPostProcessor

The BeanPostProcessor interface defines callback methods that you can implement to provide your own (or override the container’s default) instantiation logic, dependency resolution logic, and so forth. If you want to implement some custom logic after the Spring container finishes instantiating, configuring, and initializing a bean, you can plug in one or more custom BeanPostProcessor implementations.

该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。如果要在Spring容器完成实例化，配置和初始化bean之后实现某些自定义逻辑，则可以插入一个或多个自定义BeanPostProcessor实现。

You can configure multiple BeanPostProcessor instances, and you can control the order in which these BeanPostProcessorinstances execute by setting the order property. You can set this property only if the BeanPostProcessor implements the Ordered interface. If you write your own BeanPostProcessor, you should consider implementing the Ordered interface, too. For further details, see the javadoc of the [BeanPostProcessor](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/beans/factory/config/BeanPostProcessor.html) and [Ordered](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/core/Ordered.html) interfaces. See also the note on [programmatic registration of BeanPostProcessor instances](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-programmatically-registering-beanpostprocessors).

您可以配置多个BeanPostProcessor实例，并且可以BeanPostProcessor通过设置order属性来控制这些实例的执行顺序。只有在BeanPostProcessor实现Ordered 接口时才能设置此属性。如果你自己编写BeanPostProcessor，你也应该考虑实现这个Ordered接口。有关更多详细信息，请参阅[BeanPostProcessor](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/beans/factory/config/BeanPostProcessor.html) 和[Ordered](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/core/Ordered.html)接口的javadoc 。另见关于[实例的程序化登记BeanPostProcessor](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-programmatically-registering-beanpostprocessors)的说明。

|  |
| --- |
| BeanPostProcessor instances operate on bean (or object) instances. That is, the Spring IoC container instantiates a bean instance and then BeanPostProcessor instances do their work.  BeanPostProcessor实例在bean（或对象）实例上运行。也就是说，Spring IoC容器实例化一个bean实例，然后BeanPostProcessor 实例执行它们的工作。  BeanPostProcessor instances are scoped per-container. This is relevant only if you use container hierarchies. If you define a BeanPostProcessor in one container, it post-processes only the beans in that container. In other words, beans that are defined in one container are not post-processed by a BeanPostProcessor defined in another container, even if both containers are part of the same hierarchy.  BeanPostProcessor实例的范围是每个容器。仅当您使用容器层次结构时，这才是相关的。如果BeanPostProcessor在一个容器中定义一个容器，它只会对该容器中的bean进行后处理。换句话说，BeanPostProcessor即使两个容器都是同一层次结构的一部分，在一个容器中定义的bean也不会被另一个容器中定义的bean进行后处理。  To change the actual bean definition (that is, the blueprint that defines the bean), you instead need to use a BeanFactoryPostProcessor, as described in [Customizing Configuration Metadata with a BeanFactoryPostProcessor](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-extension-factory-postprocessors).  要更改实际的bean定义（即定义bean的蓝图），您需要使用a BeanFactoryPostProcessor，如 使用a [定制配置元数据中所述BeanFactoryPostProcessor](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-extension-factory-postprocessors)。 |

The org.springframework.beans.factory.config.BeanPostProcessor interface consists of exactly two callback methods. When such a class is registered as a post-processor with the container, for each bean instance that is created by the container, the post-processor gets a callback from the container both before container initialization methods (such as InitializingBean.afterPropertiesSet() or any declared init method) are called, and after any bean initialization callbacks. The post-processor can take any action with the bean instance, including ignoring the callback completely. A bean post-processor typically checks for callback interfaces, or it may wrap a bean with a proxy. Some Spring AOP infrastructure classes are implemented as bean post-processors in order to provide proxy-wrapping logic.

该org.springframework.beans.factory.config.BeanPostProcessor接口由两个回调方法组成。当这样的类被注册为具有容器的后处理器时，对于由容器创建的每个bean实例，后处理器在容器初始化方法（例如InitializingBean.afterPropertiesSet()或任何声明的init方法）之前都从容器获得回调。调用，并在任何bean初始化后回调。后处理器可以对bean实例执行任何操作，包括完全忽略回调。bean后处理器通常检查回调接口，或者它可以用代理包装bean。一些Spring AOP基础结构类实现为bean后处理器，以便提供代理包装逻辑。

An ApplicationContext automatically detects any beans that are defined in the configuration metadata that implements the BeanPostProcessor interface. The ApplicationContext registers these beans as post-processors so that they can be called later, upon bean creation. Bean post-processors can be deployed in the container in the same fashion as any other beans.

的ApplicationContext自动检测中的实现的配置的元数据中定义的任何豆BeanPostProcessor接口。将 ApplicationContext这些bean注册为后处理器，以便在创建bean时可以稍后调用它们。Bean后处理器可以以与任何其他bean相同的方式部署在容器中。

Note that, when declaring a BeanPostProcessor by using an @Bean factory method on a configuration class, the return type of the factory method should be the implementation class itself or at least the org.springframework.beans.factory.config.BeanPostProcessor interface, clearly indicating the post-processor nature of that bean. Otherwise, the ApplicationContext cannot autodetect it by type before fully creating it. Since a BeanPostProcessorneeds to be instantiated early in order to apply to the initialization of other beans in the context, this early type detection is critical.

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

|  |  |
| --- | --- |
|  | *Programmatically registering BeanPostProcessor instances*  While the recommended approach for BeanPostProcessor registration is through ApplicationContext auto-detection (as described earlier), you can register them programmatically against a ConfigurableBeanFactory by using the addBeanPostProcessor method. This can be useful when you need to evaluate conditional logic before registration or even for copying bean post processors across contexts in a hierarchy. Note, however, that BeanPostProcessor instances added programmatically do not respect the Ordered interface. Here, it is the order of registration that dictates the order of execution. Note also that BeanPostProcessor instances registered programmatically are always processed before those registered through auto-detection, regardless of any explicit ordering.  *以编程方式注册BeanPostProcessor实例*  虽然推荐的BeanPostProcessor注册方法是通过 ApplicationContext自动检测（如前所述），但您可以ConfigurableBeanFactory使用该addBeanPostProcessor 方法以编程方式对其进行注册。当您需要在注册前评估条件逻辑或甚至跨层次结构中的上下文复制Bean post处理器时，这非常有用。但请注意，以BeanPostProcessor编程方式添加的实例不尊重Ordered接口。这里，注册的顺序决定了执行的顺序。另请注意，以BeanPostProcessor编程方式注册的实例始终在通过自动检测注册的实例之前处理，而不管任何显式排序。 |
|  | *BeanPostProcessor instances and AOP auto-proxying*  Classes that implement the BeanPostProcessor interface are special and are treated differently by the container. All BeanPostProcessor instances and beans that they directly reference are instantiated on startup, as part of the special startup phase of the ApplicationContext. Next, all BeanPostProcessor instances are registered in a sorted fashion and applied to all further beans in the container. Because AOP auto-proxying is implemented as a BeanPostProcessor itself, neither BeanPostProcessor instances nor the beans they directly reference are eligible for auto-proxying and, thus, do not have aspects woven into them.  *BeanPostProcessor 实例和AOP自动代理*  实现BeanPostProcessor接口的类是特殊的，容器会对它们进行不同的处理。BeanPostProcessor他们直接引用的所有实例和bean都会在启动时实例化，作为特殊启动阶段的一部分ApplicationContext。接下来，所有BeanPostProcessor实例都以排序方式注册，并应用于容器中的所有其他bean。因为AOP自动代理是作为一个BeanPostProcessor自身实现的，所以BeanPostProcessor 实例和它们直接引用的bean都不符合自动代理的条件，因此没有编织方面。  For any such bean, you should see an informational log message: Bean someBean is not eligible for getting processed by all BeanPostProcessor interfaces (for example: not eligible for auto-proxying).  对于任何此类bean，您应该看到一条信息性日志消息：Bean someBean is not eligible for getting processed by all BeanPostProcessor interfaces (for example: not eligible for auto-proxying)。  If you have beans wired into your BeanPostProcessor by using autowiring or @Resource (which may fall back to autowiring), Spring might access unexpected beans when searching for type-matching dependency candidates and, therefore, make them ineligible for auto-proxying or other kinds of bean post-processing. For example, if you have a dependency annotated with @Resource where the field or setter name does not directly correspond to the declared name of a bean and no name attribute is used, Spring accesses other beans for matching them by type.  如果您BeanPostProcessor通过使用自动装配或@Resource（可能回退到自动装配）将bean连接到您的 ，则Spring可能会在搜索类型匹配依赖项候选项时访问意外的bean，因此，使它们不符合自动代理或其他类型的bean post -处理。例如，如果您有一个依赖项，@Resource其中字段或setter名称与bean的声明名称没有直接对应，并且没有使用name属性，则Spring会访问其他bean以按类型匹配它们。 |

The following examples show how to write, register, and use BeanPostProcessor instances in an ApplicationContext.

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

Example: Hello World, BeanPostProcessor-style

This first example illustrates basic usage. The example shows a custom BeanPostProcessor implementation that invokes the toString() method of each bean as it is created by the container and prints the resulting string to the system console.

第一个例子说明了基本用法。该示例显示了一个自定义 BeanPostProcessor实现，该实现调用toString()容器创建的每个bean 的方法，并将生成的字符串输出到系统控制台。

The following listing shows the custom BeanPostProcessor implementation class definition:

以下清单显示了自定义BeanPostProcessor实现类定义：

**package** scripting;

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

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

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

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

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

}

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

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

**return** bean;

}

}

The following beans element uses the InstantiationTracingBeanPostProcessor:

以下beans元素使用InstantiationTracingBeanPostProcessor：

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

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

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

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

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

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

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

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

<lang:groovy id="messenger"

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

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

</lang:groovy>

*<!--*

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

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

*-->*

<bean class="scripting.InstantiationTracingBeanPostProcessor"/>

</beans>

Notice how the InstantiationTracingBeanPostProcessor is merely defined. It does not even have a name, and, because it is a bean, it can be dependency-injected as you would any other bean. (The preceding configuration also defines a bean that is backed by a Groovy script. The Spring dynamic language support is detailed in the chapter entitled [Dynamic Language Support](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/languages.html#dynamic-language).)

请注意它InstantiationTracingBeanPostProcessor是如何定义的。它甚至没有名称，并且，因为它是一个bean，它可以像任何其他bean一样依赖注入。（前面的配置还定义了一个由Groovy脚本支持的bean。在动态语言支持一章中详细介绍了Spring [动态语言支持](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/languages.html#dynamic-language)。）

The following Java application runs the preceding code and configuration:

以下Java应用程序运行上述代码和配置：

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

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

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

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

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

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

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

System.out.println(messenger);

}

}

The output of the preceding application resembles the following:

上述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

Using callback interfaces or annotations in conjunction with a custom BeanPostProcessor implementation is a common means of extending the Spring IoC container. An example is Spring’s RequiredAnnotationBeanPostProcessor — a BeanPostProcessorimplementation that ships with the Spring distribution and that ensures that JavaBean properties on beans that are marked with an (arbitrary) annotation are actually (configured to be) dependency-injected with a value.

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。一个例子是Spring RequiredAnnotationBeanPostProcessor - 一个 BeanPostProcessor随Spring发行版一起提供的实现，它确保标记有（任意）注释的bean上的JavaBean属性实际上（配置为）依赖注入值。

1.8.2. Customizing Configuration Metadata with a BeanFactoryPostProcessor

The next extension point that we look at is the org.springframework.beans.factory.config.BeanFactoryPostProcessor. The semantics of this interface are similar to those of the BeanPostProcessor, with one major difference: BeanFactoryPostProcessoroperates on the bean configuration metadata. That is, the Spring IoC container lets a BeanFactoryPostProcessor read the configuration metadata and potentially change it *before* the container instantiates any beans other than BeanFactoryPostProcessor instances.

我们看到的下一个扩展点是 org.springframework.beans.factory.config.BeanFactoryPostProcessor。这个接口的语义类似于BeanPostProcessor它的一个主要区别：BeanFactoryPostProcessor操作bean配置元数据。也就是说，Spring IoC容器允许BeanFactoryPostProcessor读取配置元数据，并可能在容器实例化除实例之外的任何bean 之前更改它BeanFactoryPostProcessor。

You can configure multiple BeanFactoryPostProcessor instances, and you can control the order in which these BeanFactoryPostProcessor instances run by setting the order property. However, you can only set this property if the BeanFactoryPostProcessor implements the Ordered interface. If you write your own BeanFactoryPostProcessor, you should consider implementing the Ordered interface, too. See the javadoc of the [BeanFactoryPostProcessor](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/beans/factory/config/BeanFactoryPostProcessor.html) and [Ordered](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/core/Ordered.html) interfaces for more details.

您可以配置多个BeanFactoryPostProcessor实例，并且可以BeanFactoryPostProcessor通过设置order属性来控制这些实例的运行顺序。但是，如果BeanFactoryPostProcessor实现 Ordered接口，则只能设置此属性。如果你自己编写BeanFactoryPostProcessor，你也应该考虑实现这个Ordered接口。有关更多详细信息，请参阅[BeanFactoryPostProcessor](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/beans/factory/config/BeanFactoryPostProcessor.html)和[Ordered](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/core/Ordered.html)接口的javadoc 。

|  |  |
| --- | --- |
|  | If you want to change the actual bean instances (that is, the objects that are created from the configuration metadata), then you instead need to use a BeanPostProcessor (described earlier in [Customizing Beans by Using a BeanPostProcessor](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-extension-bpp)). While it is technically possible to work with bean instances within a BeanFactoryPostProcessor (for example, by using BeanFactory.getBean()), doing so causes premature bean instantiation, violating the standard container lifecycle. This may cause negative side effects, such as bypassing bean post processing.  如果要更改实际的bean实例（即，从配置元数据创建的对象），则需要使用a BeanPostProcessor （前面在[使用a定制Bean中进行了描述BeanPostProcessor](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-extension-bpp)）。虽然技术上可以在a中使用bean实例BeanFactoryPostProcessor（例如，通过使用 BeanFactory.getBean()），但这样做会导致过早的bean实例化，从而违反标准的容器生命周期。这可能会导致负面影响，例如绕过bean后期处理。  Also, BeanFactoryPostProcessor instances are scoped per-container. This is only relevant if you use container hierarchies. If you define a BeanFactoryPostProcessor in one container, it is applied only to the bean definitions in that container. Bean definitions in one container are not post-processed by BeanFactoryPostProcessorinstances in another container, even if both containers are part of the same hierarchy.  此外，BeanFactoryPostProcessor实例的范围是每个容器的范围。仅当您使用容器层次结构时，这才有意义。如果BeanFactoryPostProcessor在一个容器中定义一个容器，则它仅应用于该容器中的bean定义。BeanFactoryPostProcessor即使两个容器都是同一层次结构的一部分，一个容器中的Bean定义也不会被另一个容器中的实例进行后处理。 |

A bean factory post-processor is automatically executed when it is declared inside an ApplicationContext, in order to apply changes to the configuration metadata that define the container. Spring includes a number of predefined bean factory post-processors, such as PropertyOverrideConfigurer and PropertyPlaceholderConfigurer. You can also use a custom BeanFactoryPostProcessor — for example, to register custom property editors.

Bean工厂后处理器在其内部声明时会自动执行 ApplicationContext，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。您还可以使用自定义BeanFactoryPostProcessor - 例如，注册自定义属性编辑器。

An ApplicationContext automatically detects any beans that are deployed into it that implement the BeanFactoryPostProcessor interface. It uses these beans as bean factory post-processors, at the appropriate time. You can deploy these post-processor beans as you would any other bean.

一个ApplicationContext自动检测部署在它实现了任何豆BeanFactoryPostProcessor接口。它在适当的时候使用这些bean作为bean工厂后处理器。您可以像处理任何其他bean一样部署这些后处理器bean。

|  |  |
| --- | --- |
|  | As with BeanPostProcessors , you typically do not want to configure BeanFactoryPostProcessors for lazy initialization. If no other bean references a Bean(Factory)PostProcessor, that post-processor will not get instantiated at all. Thus, marking it for lazy initialization will be ignored, and the Bean(Factory)PostProcessorwill be instantiated eagerly even if you set the default-lazy-init attribute to true on the declaration of your <beans /> element.  与BeanPostProcessors一样，您通常不希望BeanFactoryPostProcessor为延迟初始化配置 s。如果没有其他bean引用a Bean(Factory)PostProcessor，则该后处理器根本不会被实例化。因此，将其标记为延迟初始化将被忽略，Bean(Factory)PostProcessor会急切地实例化，即使你设定的 default-lazy-init属性true对你的声明<beans />元素。 |

Example: The Class Name Substitution PropertyPlaceholderConfigurer

You can use the PropertyPlaceholderConfigurer to externalize property values from a bean definition in a separate file by using the standard Java Properties format. Doing so enables the person deploying an application to customize environment-specific properties, such as database URLs and passwords, without the complexity or risk of modifying the main XML definition file or files for the container.

您可以使用PropertyPlaceholderConfigurer标准Java Properties格式在单独的文件中使用bean定义中的外部化属性值。这样做可以使部署应用程序的人员自定义特定于环境的属性，例如数据库URL和密码，而不会出现修改主XML定义文件或容器文件的复杂性或风险。

Consider the following XML-based configuration metadata fragment, where a DataSource with placeholder values is defined:

请考虑以下基于XML的配置元数据片段，其中DataSource 定义了占位符值：

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

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

</bean>

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

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

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

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

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

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

</bean>

The example shows properties configured from an external Properties file. At runtime, a PropertyPlaceholderConfigurer is applied to the metadata that replaces some properties of the DataSource. The values to replace are specified as placeholders of the form ${property-name}, which follows the Ant and log4j and JSP EL style.

该示例显示了从外部Properties文件配置的属性。在运行时，a PropertyPlaceholderConfigurer将应用于替换DataSource的某些属性的元数据。要替换的值被指定为表单的占位符${property-name}，它遵循Ant和log4j以及JSP EL样式。

The actual values come from another file in the standard Java Properties format:

实际值来自标准Java Properties格式的另一个文件：

jdbc.driverClassName=org.hsqldb.jdbcDriver

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

jdbc.username=sa

jdbc.password=root

Therefore, the ${jdbc.username} string is replaced at runtime with the value, 'sa', and the same applies for other placeholder values that match keys in the properties file. The PropertyPlaceholderConfigurer checks for placeholders in most properties and attributes of a bean definition. Furthermore, you can customize the placeholder prefix and suffix.

因此，${jdbc.username}在运行时使用值“sa”替换字符串，这同样适用于与属性文件中的键匹配的其他占位符值。在PropertyPlaceholderConfigurer为大多数属性和bean定义的属性占位符检查。此外，您可以自定义占位符前缀和后缀。

With the context namespace introduced in Spring 2.5, you can configure property placeholders with a dedicated configuration element. You can provide one or more locations as a comma-separated list in the location attribute, as the following example shows:

使用contextSpring 2.5中引入的命名空间，您可以使用专用配置元素配置属性占位符。您可以在location属性中提供一个或多个位置作为逗号分隔列表，如以下示例所示：

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

The PropertyPlaceholderConfigurer not only looks for properties in the Properties file you specify. By default, if it cannot find a property in the specified properties files, it also checks against the Java System properties. You can customize this behavior by setting the systemPropertiesMode property of the configurer with one of the following three supported integer values:

在PropertyPlaceholderConfigurer不仅将查找在属性Properties 指定的文件。默认情况下，如果它在指定的属性文件中找不到属性，它还会检查Java System属性。您可以通过systemPropertiesMode使用以下三个受支持的整数值之一设置configurer 的属性来自定义此行为：

* never (0): Never check system properties.
* fallback (1): Check system properties if not resolvable in the specified properties files. This is the default.
* override (2): Check system properties first, before trying the specified properties files. This lets system properties override any other property source.
* never （0）：从不检查系统属性。
* fallback（1）：如果在指定的属性文件中无法解析，则检查系统属性。这是默认值。
* override（2）：在尝试指定的属性文件之前，首先检查系统属性。这使系统属性可以覆盖任何其他属性源。

See the [PropertyPlaceholderConfigurer](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/beans/factory/config/PropertyPlaceholderConfigurer.html) javadoc for more information.

有关[PropertyPlaceholderConfigurer](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/beans/factory/config/PropertyPlaceholderConfigurer.html)更多信息，请参阅javadoc。

|  |  |
| --- | --- |
|  | You can use the PropertyPlaceholderConfigurer to substitute class names, which is sometimes useful when you have to pick a particular implementation class at runtime. The following example shows how to do so:  您可以使用PropertyPlaceholderConfigurer替换类名称，这在您必须在运行时选择特定实现类时有时很有用。以下示例显示了如何执行此操作：  <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">  <property name="locations">  <value>classpath:com/something/strategy.properties</value>  </property>  <property name="properties">  <value>custom.strategy.class=com.something.DefaultStrategy</value>  </property>  </bean>  <bean id="serviceStrategy" class="${custom.strategy.class}"/>  If the class cannot be resolved at runtime to a valid class, resolution of the bean fails when it is about to be created, which is during the preInstantiateSingletons() phase of an ApplicationContext for a non-lazy-init bean.  如果类不能在运行时被解析为一个有效的类，bean的分辨率，当它即将被创造，这是在失败preInstantiateSingletons() 的阶段ApplicationContext对非延迟实例化的bean。 |

Example: The PropertyOverrideConfigurer

The PropertyOverrideConfigurer, another bean factory post-processor, resembles the PropertyPlaceholderConfigurer, but unlike the latter, the original definitions can have default values or no values at all for bean properties. If an overridingProperties file does not have an entry for a certain bean property, the default context definition is used.

在PropertyOverrideConfigurer另一个bean工厂后置处理器，类似 PropertyPlaceholderConfigurer，但不同的是后者，原来的定义可以有缺省值或者根本没有值的bean属性。如果覆盖 Properties文件没有某个bean属性的条目，则使用默认上下文定义。

Note that the bean definition is not aware of being overridden, so it is not immediately obvious from the XML definition file that the override configurer is being used. In case of multiple PropertyOverrideConfigurer instances that define different values for the same bean property, the last one wins, due to the overriding mechanism.

请注意，bean定义不知道被覆盖，因此从XML定义文件中可以立即看出正在使用覆盖配置器。如果多个PropertyOverrideConfigurer实例为同一个bean属性定义了不同的值，则由于覆盖机制，最后一个实例会获胜。

Properties file configuration lines take the following format:

属性文件配置行采用以下格式：

beanName.property=value

The following listing shows an example of the format:

以下清单显示了格式的示例：

dataSource.driverClassName=com.mysql.jdbc.Driver

dataSource.url=jdbc:mysql:mydb

This example file can be used with a container definition that contains a bean called dataSource that has driver and urlproperties.

此示例文件可以与包含名为dataSourcehas has driver和urlproperties 的bean的容器定义一起使用 。

Compound property names are also supported, as long as every component of the path except the final property being overridden is already non-null (presumably initialized by the constructors). In the following example, the sammy property of the bob property of the fred property of the tom bean is set to the scalar value 123:

也支持复合属性名称，只要路径的每个组件（重写的最终属性除外）都已经非空（可能由构造函数初始化）。在下面的例子中，sammy所述的属性bob的财产fred的财产tom豆被设置为标量值123：

tom.fred.bob.sammy=123

|  |  |
| --- | --- |
|  | Specified override values are always literal values. They are not translated into bean references. This convention also applies when the original value in the XML bean definition specifies a bean reference.  指定的覆盖值始终是文字值。它们不会被翻译成bean引用。当XML bean定义中的原始值指定bean引用时，此约定也适用。 |

With the context namespace introduced in Spring 2.5, it is possible to configure property overriding with a dedicated configuration element, as the following example shows:

使用contextSpring 2.5中引入的命名空间，可以使用专用配置元素配置属性覆盖，如以下示例所示：

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

1.8.3. Customizing Instantiation Logic with a FactoryBean

You can implement the org.springframework.beans.factory.FactoryBean interface for objects that are themselves factories.

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

The FactoryBean interface is a point of pluggability into the Spring IoC container’s instantiation logic. If you have complex initialization code that is better expressed in Java as opposed to a (potentially) verbose amount of XML, you can create your ownFactoryBean, write the complex initialization inside that class, and then plug your custom FactoryBean into the container.

该FactoryBean接口是Spring IoC容器实例化逻辑的可插拔点。如果你有一个复杂的初始化代码，用Java表示，而不是（可能）冗长的XML，你可以创建自己的 FactoryBean，在该类中编写复杂的初始化，然后将自定义FactoryBean插入容器。

The FactoryBean interface provides three methods:

该FactoryBean接口提供了三种方法：

* Object getObject(): Returns an instance of the object this factory creates. The instance can possibly be shared, depending on whether this factory returns singletons or prototypes.
* boolean isSingleton(): Returns true if this FactoryBean returns singletons or false otherwise.
* Class getObjectType(): Returns the object type returned by the getObject() method or null if the type is not known in advance.
* Object getObject()：返回此工厂创建的对象的实例。可以共享实例，具体取决于此工厂是返回单例还是原型。
* boolean isSingleton()：true如果FactoryBean返回单例或false其他方式返回 。
* Class getObjectType()：返回getObject()方法返回的对象类型，或者null如果事先不知道类型。

The FactoryBean concept and interface is used in a number of places within the Spring Framework. More than 50 implementations of the FactoryBean interface ship with Spring itself.

该FactoryBean概念和接口被一些Spring框架内的场所。超过50个FactoryBean接口的实现随Spring一起提供。

When you need to ask a container for an actual FactoryBean instance itself instead of the bean it produces, preface the bean’s id with the ampersand symbol (&) when calling the getBean() method of the ApplicationContext. So, for a given FactoryBean with an id of myBean, invoking getBean("myBean") on the container returns the product of the FactoryBean, whereas invoking getBean("&myBean") returns the FactoryBean instance itself.

当你需要向一个容器询问一个实际的FactoryBean实例本身而不是它生成的bean 时，在调用the的方法时id，用strersand符号（&）作为前缀。因此，对于给定 与的，调用在容器上返回的产品，而调用返回的 实例本身。getBean()ApplicationContextFactoryBeanidmyBeangetBean("myBean")FactoryBeangetBean("&myBean")FactoryBean

1.9. Annotation-based Container Configuration

基于注释的容器配置

Are annotations better than XML for configuring Spring?

注释是否比配置Spring的XML更好？

The introduction of annotation-based configuration raised the question of whether this approach is “better” than XML. The short answer is “it depends.” The long answer is that each approach has its pros and cons, and, usually, it is up to the developer to decide which strategy suits them better. Due to the way they are defined, annotations provide a lot of context in their declaration, leading to shorter and more concise configuration. However, XML excels at wiring up components without touching their source code or recompiling them. Some developers prefer having the wiring close to the source while others argue that annotated classes are no longer POJOs and, furthermore, that the configuration becomes decentralized and harder to control.

基于注释的配置的引入引发了这种方法是否比XML“更好”的问题。简短的回答是“它取决于。”长期的答案是每种方法都有其优点和缺点，通常，由开发人员决定哪种策略更适合他们。由于它们的定义方式，注释在其声明中提供了大量上下文，从而导致更短更简洁的配置。但是，XML擅长在不触及源代码或重新编译它们的情况下连接组件。一些开发人员更喜欢将布线靠近源，而另一些开发人员则认为注释类不再是POJO，而且配置变得分散且难以控制。

No matter the choice, Spring can accommodate both styles and even mix them together. It is worth pointing out that through its [JavaConfig](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-java) option, Spring lets annotations be used in a non-invasive way, without touching the target components source code and that, in terms of tooling, all configuration styles are supported by the [Spring Tool Suite](https://spring.io/tools/sts).

无论选择如何，Spring都可以兼顾两种风格，甚至可以将它们混合在一起。值得指出的是，通过其[JavaConfig](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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 。

An alternative to XML setup is provided by annotation-based configuration, which relies on the bytecode metadata for wiring up components instead of angle-bracket declarations. Instead of using XML to describe a bean wiring, the developer moves the configuration into the component class itself by using annotations on the relevant class, method, or field declaration. As mentioned in [Example: The RequiredAnnotationBeanPostProcessor](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-extension-bpp-examples-rabpp), using a BeanPostProcessor in conjunction with annotations is a common means of extending the Spring IoC container. For example, Spring 2.0 introduced the possibility of enforcing required properties with the [@Required](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-required-annotation) annotation. Spring 2.5 made it possible to follow that same general approach to drive Spring’s dependency injection. Essentially, the @Autowired annotation provides the same capabilities as described in [Autowiring Collaborators](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-autowire) but with more fine-grained control and wider applicability. Spring 2.5 also added support for JSR-250 annotations, such as @PostConstruct and @PreDestroy. Spring 3.0 added support for JSR-330 (Dependency Injection for Java) annotations contained in the javax.inject package such as @Inject and @Named. Details about those annotations can be found in the [relevant section](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-standard-annotations).

基于注释的配置提供了XML设置的替代方案，该配置依赖于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连接，而是通过在相关的类，方法或字段声明上使用注释将配置移动到组件类本身。如[示例中所述：RequiredAnnotationBeanPostProcessor](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-extension-bpp-examples-rabpp)使用BeanPostProcessor与注释结合使用是扩展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/5.1.8.RELEASE/spring-framework-reference/core.html#beans-required-annotation)注释强制执行所需属性的可能性。Spring 2.5使得有可能采用相同的通用方法来驱动Spring的依赖注入。基本上，@Autowired注释提供与[自动装配协作者中](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-autowire)描述的相同的功能，但具有更细粒度的控制和更广泛的适用性。Spring 2.5还增加了对JSR-250注释的支持，例如 @PostConstruct和@PreDestroy。Spring 3.0增加了对javax.inject包中包含的JSR-330（Java的依赖注入）注释的支持，例如@Inject 和@Named。有关这些注释的详细信息，请参阅 [相关章节](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-standard-annotations)。

|  |  |
| --- | --- |
|  | Annotation injection is performed before XML injection. Thus, the XML configuration overrides the annotations for properties wired through both approaches.  注释注入在XML注入之前执行。因此，XML配置会覆盖通过这两种方法连接的属性的注释。 |

As always, you can register them as individual bean definitions, but they can also be implicitly registered by including the following tag in an XML-based Spring configuration (notice the inclusion of the context namespace):

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（请注意包含context命名空间）：

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

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

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

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

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

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

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

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

<context:annotation-config/>

</beans>

(The implicitly registered post-processors include [AutowiredAnnotationBeanPostProcessor](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/beans/factory/annotation/AutowiredAnnotationBeanPostProcessor.html),[CommonAnnotationBeanPostProcessor](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/context/annotation/CommonAnnotationBeanPostProcessor.html), [PersistenceAnnotationBeanPostProcessor](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/orm/jpa/support/PersistenceAnnotationBeanPostProcessor.html), and the aforementioned[RequiredAnnotationBeanPostProcessor](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/beans/factory/annotation/RequiredAnnotationBeanPostProcessor.html).)

（在隐式注册后处理器包括 [AutowiredAnnotationBeanPostProcessor](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/beans/factory/annotation/AutowiredAnnotationBeanPostProcessor.html)， [CommonAnnotationBeanPostProcessor](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/context/annotation/CommonAnnotationBeanPostProcessor.html)，[PersistenceAnnotationBeanPostProcessor](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/orm/jpa/support/PersistenceAnnotationBeanPostProcessor.html)，和前面提到的 [RequiredAnnotationBeanPostProcessor](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/beans/factory/annotation/RequiredAnnotationBeanPostProcessor.html)。）

|  |  |
| --- | --- |
|  | <context:annotation-config/> only looks for annotations on beans in the same application context in which it is defined. This means that, if you put <context:annotation-config/> in a WebApplicationContext for a DispatcherServlet, it only checks for @Autowired beans in your controllers, and not your services. See [The DispatcherServlet](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/web.html#mvc-servlet) for more information.  <context:annotation-config/>仅查找在定义它的同一应用程序上下文中的bean上的注释。这意味着，如果你<context:annotation-config/>输入一个WebApplicationContextfor DispatcherServlet，它只会检查@Autowired你的控制器中的bean，而不是你的服务。有关更多信息，请参阅 [DispatcherServlet](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/web.html#mvc-servlet)。 |

1.9.1. @Required

The @Required annotation applies to bean property setter methods, as in the following example:

该@Required注释适用于bean属性setter方法，如下面的例子：

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

**private** MovieFinder movieFinder;

@Required

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

this.movieFinder = movieFinder;

}

*// ...*

}

This annotation indicates that the affected bean property must be populated at configuration time, through an explicit property value in a bean definition or through autowiring. The container throws an exception if the affected bean property has not been populated. This allows for eager and explicit failure, avoiding NullPointerException instances or the like later on. We still recommend that you put assertions into the bean class itself (for example, into an init method). Doing so enforces those required references and values even when you use the class outside of a container.

此批注指示必须在配置时通过bean定义中的显式属性值或通过自动装配填充受影响的bean属性。如果尚未填充受影响的bean属性，则容器将引发异常。这允许急切和明确的失败，以后避免NullPointerException 实例等。我们仍然建议您将断言放入bean类本身（例如，转换为init方法）。即使您在容器外部使用类，这样做也会强制执行那些必需的引用和值。

|  |  |
| --- | --- |
|  | The @Required annotation is formally deprecated as of Spring Framework 5.1, in favor of using constructor injection for required settings (or a custom implementation of InitializingBean.afterPropertiesSet() along with bean property setter methods).  从@RequiredSpring Framework 5.1开始，注释正式被弃用，支持使用构造函数注入所需的设置（或者InitializingBean.afterPropertiesSet()bean属性setter方法的自定义实现 ）。 |

1.9.2. Using @Autowired

|  |  |
| --- | --- |
|  | JSR 330’s @Inject annotation can be used in place of Spring’s @Autowired annotation in the examples included in this section. See [here](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-standard-annotations) for more details.  在本节中包含的示例中，@Inject可以使用JSR 330的注释代替Spring的@Autowired注释。有关详细信息，请参见[此处](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-standard-annotations) |

You can apply the @Autowired annotation to constructors, as the following example shows:

您可以将@Autowired注释应用于构造函数，如以下示例所示：

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

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

@Autowired

**public** MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {

this.customerPreferenceDao = customerPreferenceDao;

}

*// ...*

}

|  |  |
| --- | --- |
|  | As of Spring Framework 4.3, an @Autowired annotation on such a constructor is no longer necessary if the target bean defines only one constructor to begin with. However, if several constructors are available, at least one must be annotated to teach the container which one to use.  从Spring Framework 4.3开始，@Autowired如果目标bean只定义了一个开头的构造函数，则不再需要对这样的构造函数进行注释。但是，如果有几个构造器可用，则必须注释至少一个构造器以教导容器使用哪一个。 |

You can also apply the @Autowired annotation to “traditional” setter methods, as the following example shows:

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

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

**private** MovieFinder movieFinder;

@Autowired

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

this.movieFinder = movieFinder;

}

*// ...*

}

You can also apply the annotation to methods with arbitrary names and multiple arguments, as the following example shows:

您还可以将注释应用于具有任意名称和多个参数的方法，如以下示例所示：

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

**private** MovieCatalog movieCatalog;

**private** CustomerPreferenceDao customerPreferenceDao;

@Autowired

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

CustomerPreferenceDao customerPreferenceDao) {

this.movieCatalog = movieCatalog;

this.customerPreferenceDao = customerPreferenceDao;

}

*// ...*

}

You can apply @Autowired to fields as well and even mix it with constructors, as the following example shows:

您也可以应用于@Autowired字段，甚至可以将它与构造函数混合使用，如下例所示：

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

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

@Autowired

**private** MovieCatalog movieCatalog;

@Autowired

**public** MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {

this.customerPreferenceDao = customerPreferenceDao;

}

*// ...*

}

|  |  |
| --- | --- |
|  | Make sure that your target components (for example, MovieCatalog or CustomerPreferenceDao) are consistently declared by the type that you use for your @Autowired-annotated injection points. Otherwise, injection may fail due to no type match found at runtime.  确保目标组件（例如，MovieCatalog或CustomerPreferenceDao）始终按照用于@Autowired注释注入点的类型声明。否则，由于在运行时未找到类型匹配，注入可能会失败。  For XML-defined beans or component classes found through a classpath scan, the container usually knows the concrete type up front. However, for @Bean factory methods, you need to make sure that the declared return type is sufficiently expressive. For components that implement several interfaces or for components potentially referred to by their implementation type, consider declaring the most specific return type on your factory method (at least as specific as required by the injection points referring to your bean).  对于通过类路径扫描找到的XML定义的bean或组件类，容器通常预先知道具体类型。但是，对于@Bean工厂方法，您需要确保声明的返回类型具有足够的表现力。对于实现多个接口的组件或可能由其实现类型引用的组件，请考虑在工厂方法上声明最具体的返回类型（至少与引用bean的注入点所需的特定类型一致）。 |

You can also provide all beans of a particular type from the ApplicationContext by adding the annotation to a field or method that expects an array of that type, as the following example shows:

您还可以ApplicationContext 通过将注释添加到需要该类型数组的字段或方法来提供特定类型的所有bean ，如以下示例所示：

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

@Autowired

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

*// ...*

}

The same applies for typed collections, as the following example shows:

这同样适用于类型化集合，如以下示例所示：

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

**private** Set<MovieCatalog> movieCatalogs;

@Autowired

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

this.movieCatalogs = movieCatalogs;

}

*// ...*

}

|  |  |
| --- | --- |
|  | Your target beans can implement the org.springframework.core.Ordered interface or use the @Order or standard @Priority annotation if you want items in the array or list to be sorted in a specific order. Otherwise, their order follows the registration order of the corresponding target bean definitions in the container.  如果希望按特定顺序对数组或列表中的项进行排序，则目标bean可以实现org.springframework.core.Ordered接口或使用@Order或标准@Priority注释。否则，它们的顺序遵循容器中相应目标bean定义的注册顺序。  You can declare the @Order annotation at the target class level and on @Bean methods, potentially by individual bean definition (in case of multiple definitions that use the same bean class). @Order values may influence priorities at injection points, but be aware that they do not influence singleton startup order, which is an orthogonal concern determined by dependency relationships and @DependsOn declarations.  您可以@Order在目标类级别和@Bean方法上声明注释，可能是通过单个bean定义（在多个定义使用相同bean类的情况下）。@Order值可能会影响注入点的优先级，但要注意它们不会影响单例启动顺序，这是由依赖关系和@DependsOn声明确定的正交关注点。  Note that the standard javax.annotation.Priority annotation is not available at the @Bean level, since it cannot be declared on methods. Its semantics can be modeled through @Order values in combination with @Primary on a single bean for each type.  请注意，标准javax.annotation.Priority注释在该@Bean级别不可用 ，因为它无法在方法上声明。它的语义可以通过@Order值与@Primary每种类型的单个bean 相结合来建模。 |

Even typed Map instances can be autowired as long as the expected key type is String. The Map values contain all beans of the expected type, and the keys contain the corresponding bean names, as the following example shows:

Map只要预期的密钥类型是，即使是类型化的实例也可以自动装配String。Map值包含所有期望类型的bean，并且键包含相应的bean名称，如以下示例所示：

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

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

@Autowired

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

this.movieCatalogs = movieCatalogs;

}

*// ...*

}

By default, autowiring fails when no matching candidate beans are available for a given injection point. In the case of a declared array, collection or map, at least one matching element is expected.

默认情况下，当给定注入点没有匹配的候选bean时，自动装配失败。对于声明的数组，集合或映射，至少需要一个匹配元素。

The default behavior is to treat annotated methods and fields as indicating required dependencies. You can change this behavior as demonstrated in the following example, enabling the framework to skip a non-satisfiable injection point through marking it as non-required:

默认行为是将带注释的方法和字段视为指示所需的依赖项。您可以更改此行为，如以下示例所示，使框架能够通过将其标记为非必需来跳过不可满足的注入点：

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

**private** MovieFinder movieFinder;

@Autowired(required = false)

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

this.movieFinder = movieFinder;

}

*// ...*

}

A non-required method will not be called at all if its dependency (or one of its dependencies in case of multiple arguments) is not available. A non-required field will not get populated at all in such case, leaving its default value in place.

如果不可用的依赖项（或多个参数情况下的依赖项之一），则根本不会调用非必需的方法。在这种情况下，根本不会填充非必填字段，保留其默认值。

Injected constructor and factory method arguments are a special case since the 'required' flag on @Autowired has a somewhat different meaning due to Spring’s constructor resolution algorithm potentially dealing with multiple constructors. Constructor and factory method arguments are effectively required by default but with a few special rules in a single-constructor scenario, such as multi-element injection points (arrays, collections, maps) resolving to empty instances if no matching beans are available. This allows for a common implementation pattern where all dependencies can be declared in a unique multi-argument constructor, e.g. declared as a single public constructor without an @Autowired annotation.

注入的构造函数和工厂方法参数是一种特殊情况，因为@Autowired由于Spring的构造函数解析算法可能涉及多个构造函数，因此'required'标志的含义有些不同。默认情况下有效地需要构造函数和工厂方法参数，但在单构造函数场景中有一些特殊规则，例如，如果没有匹配的bean可用，则解析为空实例的多元素注入点（数组，集合，映射）。这允许一个通用的实现模式，其中所有依赖项都可以在一个唯一的多参数构造函数中声明，例如声明为没有@Autowired注释的单个公共构造函数。

|  |  |
| --- | --- |
|  | Only one annotated constructor per class can be marked as required, but multiple non-required constructors can be annotated. In that case, each is considered among the candidates and Spring uses the greediest constructor whose dependencies can be satisfied — that is, the constructor that has the largest number of arguments. The constructor resolution algorithm is the same as for non-annotated classes with overloaded constructors, just narrowing the candidates to annotated constructors.  每个类只能标记一个带注释的构造函数，但可以注释多个非必需的构造函数。在这种情况下，每个都被认为是候选者之一，Spring使用最贪婪的构造函数，其依赖性可以得到满足 - 也就是说，具有最多参数的构造函数。构造函数解析算法与具有重载构造函数的非注释类相同，只是将候选者缩小到带注释的构造函数。  The 'required' attribute of @Autowired is recommended over the @Required annotation on setter methods. The 'required' attribute indicates that the property is not required for autowiring purposes. The property is ignored if it cannot be autowired. @Required, on the other hand, is stronger in that it enforces the property to be set by any means supported by the container. If no value is defined, a corresponding exception is raised.  建议使用“必需”属性而@Autowired不是@Requiredsetter方法的注释。“required”属性表示该属性不是自动装配所必需的。如果无法自动装配，则会忽略该属性。@Required另一方面，它更强大，因为它强制通过容器支持的任何方式设置属性。如果未定义任何值，则会引发相应的异常。 |

Alternatively, you can express the non-required nature of a particular dependency through Java 8’s java.util.Optional, as the following example shows:

或者，您可以通过Java 8表达特定依赖关系的非必需特性java.util.Optional，如以下示例所示：

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

@Autowired

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

...

}

}

As of Spring Framework 5.0, you can also use a @Nullable annotation (of any kind in any package — for example, javax.annotation.Nullable from JSR-305):

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的注释 - 例如，javax.annotation.Nullable来自JSR-305）：

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

@Autowired

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

...

}

}

You can also use @Autowired for interfaces that are well-known resolvable dependencies: BeanFactory, ApplicationContext, Environment, ResourceLoader, ApplicationEventPublisher, and MessageSource. These interfaces and their extended interfaces, such as ConfigurableApplicationContext or ResourcePatternResolver, are automatically resolved, with no special setup necessary. The following example autowires an ApplicationContext object:

您还可以使用@Autowired对于那些众所周知的解析依赖接口：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，无需特殊设置。以下示例自动装配一个ApplicationContext对象：

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

@Autowired

**private** ApplicationContext context;

**public** MovieRecommender() {

}

*// ...*

}

|  |  |
| --- | --- |
|  | The @Autowired, @Inject, @Value, and @Resource annotations are handled by Spring BeanPostProcessorimplementations. This means that you cannot apply these annotations within your own BeanPostProcessor or BeanFactoryPostProcessor types (if any). These types must be 'wired up' explicitly by using XML or a Spring @Bean method.  在@Autowired，@Inject，@Value，和@Resource注释由Spring处理 BeanPostProcessor实现。这意味着您无法在自己的类型BeanPostProcessor或BeanFactoryPostProcessor类型（如果有）中应用这些注释。必须使用XML或Spring @Bean方法显式地“连接”这些类型。 |

1.9.3. Fine-tuning Annotation-based Autowiring with @Primary

Because autowiring by type may lead to multiple candidates, it is often necessary to have more control over the selection process. One way to accomplish this is with Spring’s @Primary annotation. @Primary indicates that a particular bean should be given preference when multiple beans are candidates to be autowired to a single-valued dependency. If exactly one primary bean exists among the candidates, it becomes the autowired value.

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进行更多控制。实现这一目标的一种方法是使用Spring的@Primary注释。@Primary表示当多个bean可以自动装配到单值依赖项时，应该优先选择特定的bean。如果候选者中只存在一个主bean，则它将成为自动装配的值。

Consider the following configuration that defines firstMovieCatalog as the primary MovieCatalog:

请考虑以下定义firstMovieCatalog为主要的配置MovieCatalog：

@Configuration

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

@Bean

@Primary

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

@Bean

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

*// ...*

}

With the preceding configuration, the following MovieRecommender is autowired with the firstMovieCatalog:

使用上述配置，以下MovieRecommender内容自动装配 firstMovieCatalog：

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

@Autowired

**private** MovieCatalog movieCatalog;

*// ...*

}

The corresponding bean definitions follow:

相应的bean定义如下：

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

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

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

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

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

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

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

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

<context:annotation-config/>

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

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

</bean>

<bean class="example.SimpleMovieCatalog">

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

</bean>

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

</beans>

1.9.4. Fine-tuning Annotation-based Autowiring with Qualifiers

使用限定符微调基于注释的自动装配

@Primary is an effective way to use autowiring by type with several instances when one primary candidate can be determined. When you need more control over the selection process, you can use Spring’s @Qualifier annotation. You can associate qualifier values with specific arguments, narrowing the set of type matches so that a specific bean is chosen for each argument. In the simplest case, this can be a plain descriptive value, as shown in the following example:

@Primary当可以确定一个主要候选者时，是通过具有多个实例的类型使用自动装配的有效方式。当您需要更多控制选择过程时，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，缩小类型匹配集，以便为每个参数选择特定的bean。在最简单的情况下，这可以是一个简单的描述性值，如以下示例所示：

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

@Autowired

@Qualifier("main")

**private** MovieCatalog movieCatalog;

*// ...*

}

You can also specify the @Qualifier annotation on individual constructor arguments or method parameters, as shown in the following example:

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

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

**private** MovieCatalog movieCatalog;

**private** CustomerPreferenceDao customerPreferenceDao;

@Autowired

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

CustomerPreferenceDao customerPreferenceDao) {

this.movieCatalog = movieCatalog;

this.customerPreferenceDao = customerPreferenceDao;

}

*// ...*

}

The following example shows corresponding bean definitions.

以下示例显示了相应的bean定义。

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

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

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

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

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

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

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

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

<context:annotation-config/>

<bean class="example.SimpleMovieCatalog">

<qualifier value="main"/>

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

</bean>

<bean class="example.SimpleMovieCatalog">

<qualifier value="action"/>

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

</bean>

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

</beans>

|  |  |  |  |
| --- | --- | --- | --- |
|  | The bean with the main qualifier value is wired with the constructor argument that is qualified with the same value.  具有main限定符值的bean与使用相同值限定的构造函数参数连接。   |  |  | | --- | --- | |  |  | |
|  | The bean with the action qualifier value is wired with the constructor argument that is qualified with the same value.   |  | | --- | | 具有action限定符值的bean与使用相同值限定的构造函数参数连接。 | |

For a fallback match, the bean name is considered a default qualifier value. Thus, you can define the bean with an id of maininstead of the nested qualifier element, leading to the same matching result. However, although you can use this convention to refer to specific beans by name, @Autowired is fundamentally about type-driven injection with optional semantic qualifiers. This means that qualifier values, even with the bean name fallback, always have narrowing semantics within the set of type matches. They do not semantically express a reference to a unique bean id. Good qualifier values are main or EMEA or persistent, expressing characteristics of a specific component that are independent from the bean id, which may be auto-generated in case of an anonymous bean definition such as the one in the preceding example.

对于回退匹配，bean名称被视为默认限定符值。因此，可以用一个定义bean id的main代替嵌套限定符元素，导致相同的匹配结果。但是，尽管您可以使用此约定来按名称引用特定bean，但@Autowired基本上是关于具有可选语义限定符的类型驱动注入。这意味着即使使用bean名称回退，限定符值在类型匹配集中也总是具有缩小的语义。它们在语义上不表示对唯一bean的引用id。良好限定的值是main或EMEA或persistent，表达独立于从所述豆的特定部件的特性id，在匿名bean定义的情况下可以自动生成，例如前面例子中的定义。

Qualifiers also apply to typed collections, as discussed earlier — for example, to Set<MovieCatalog>. In this case, all matching beans, according to the declared qualifiers, are injected as a collection. This implies that qualifiers do not have to be unique. Rather, they constitute filtering criteria. For example, you can define multiple MovieCatalog beans with the same qualifier value “action”, all of which are injected into a Set<MovieCatalog> annotated with @Qualifier("action").

限定符也适用于类型集合，如前所述 - 例如，to Set<MovieCatalog>。在这种情况下，根据声明的限定符，所有匹配的bean都作为集合注入。这意味着限定符不必是唯一的。相反，它们构成了过滤标准。例如，您可以MovieCatalog使用相同的限定符值“action” 定义多个bean，所有这些bean都注入带Set<MovieCatalog>注释的注释中@Qualifier("action")。

|  |  |
| --- | --- |
|  | Letting qualifier values select against target bean names, within the type-matching candidates, does not require a @Qualifier annotation at the injection point. If there is no other resolution indicator (such as a qualifier or a primary marker), for a non-unique dependency situation, Spring matches the injection point name (that is, the field name or parameter name) against the target bean names and choose the same-named candidate, if any.  在类型匹配候选项中，根据目标bean名称选择限定符值，不需要@Qualifier注入点处的注释。如果没有其他解析指示符（例如限定符或主要标记），则对于非唯一依赖性情况，Spring会将注入点名称（即字段名称或参数名称）与目标bean名称进行匹配，然后选择同名的候选人，如果有的话。  That said, if you intend to express annotation-driven injection by name, do not primarily use @Autowired, even if it is capable of selecting by bean name among type-matching candidates. Instead, use the JSR-250 @Resourceannotation, which is semantically defined to identify a specific target component by its unique name, with the declared type being irrelevant for the matching process. @Autowired has rather different semantics: After selecting candidate beans by type, the specified String qualifier value is considered within those type-selected candidates only (for example, matching an account qualifier against beans marked with the same qualifier label).  也就是说，如果您打算按名称表达注释驱动的注入，请不要主要使用@Autowired，即使它能够在类型匹配候选项中通过bean名称进行选择。相反，使用JSR-250 @Resource注释，该注释在语义上定义为通过其唯一名称标识特定目标组件，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，String 仅在那些类型选择的候选中考虑指定的限定符值（例如，将account限定符与标记有相同限定符标签的bean 匹配）。  For beans that are themselves defined as a collection, Map, or array type, @Resource is a fine solution, referring to the specific collection or array bean by unique name. That said, as of 4.3, collection, you can match Map, and array types through Spring’s @Autowired type matching algorithm as well, as long as the element type information is preserved in @Bean return type signatures or collection inheritance hierarchies. In this case, you can use qualifier values to select among same-typed collections, as outlined in the previous paragraph.  对于自身定义为集合Map或数组类型的bean来说，这@Resource 是一个很好的解决方案，它通过唯一名称引用特定的集合或数组bean。也就是说，从4.3开始，只要在返回类型签名或集合继承层次结构中保留元素类型信息，就可以Map通过Spring的@Autowired类型匹配算法匹配和数组类型 @Bean。在这种情况下，您可以使用限定符值在相同类型的集合中进行选择，如上一段所述。  As of 4.3, @Autowired also considers self references for injection (that is, references back to the bean that is currently injected). Note that self injection is a fallback. Regular dependencies on other components always have precedence. In that sense, self references do not participate in regular candidate selection and are therefore in particular never primary. On the contrary, they always end up as lowest precedence. In practice, you should use self references as a last resort only (for example, for calling other methods on the same instance through the bean’s transactional proxy). Consider factoring out the effected methods to a separate delegate bean in such a scenario. Alternatively, you can use @Resource, which may obtain a proxy back to the current bean by its unique name.  从4.3开始，@Autowired还考虑了自引用注入（即，引用回到当前注入的bean）。请注意，自我注入是一种后备。对其他组件的常规依赖性始终具有优先权。从这个意义上说，自我引用并不参与常规的候选人选择，因此特别是不是主要的。相反，它们总是最低优先级。在实践中，您应该仅使用自引用作为最后的手段（例如，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，您可以使用@Resource，它可以通过其唯一名称获取代理回到当前bean。  @Autowired applies to fields, constructors, and multi-argument methods, allowing for narrowing through qualifier annotations at the parameter level. By contrast, @Resource is supported only for fields and bean property setter methods with a single argument. As a consequence, you should stick with qualifiers if your injection target is a constructor or a multi-argument method.  @Autowired适用于字段，构造函数和多参数方法，允许在参数级别通过限定符注释缩小范围。相比之下，@Resource 仅支持具有单个参数的字段和bean属性setter方法。因此，如果注射目标是构造函数或多参数方法，则应该使用限定符。 |

You can create your own custom qualifier annotations. To do so, define an annotation and provide the @Qualifier annotation within your definition, as the following example shows:

您可以创建自己的自定义限定符注释。为此，请定义注释并@Qualifier在定义中提供注释，如以下示例所示：

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

@Retention(RetentionPolicy.RUNTIME)

@Qualifier

**public** @interface Genre {

String value();

}

Then you can provide the custom qualifier on autowired fields and parameters, as the following example shows:

然后，您可以在自动装配的字段和参数上提供自定义限定符，如以下示例所示：

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

@Autowired

@Genre("Action")

**private** MovieCatalog actionCatalog;

**private** MovieCatalog comedyCatalog;

@Autowired

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

this.comedyCatalog = comedyCatalog;

}

*// ...*

}

Next, you can provide the information for the candidate bean definitions. You can add <qualifier/> tags as sub-elements of the <bean/> tag and then specify the type and value to match your custom qualifier annotations. The type is matched against the fully-qualified class name of the annotation. Alternately, as a convenience if no risk of conflicting names exists, you can use the short class name. The following example demonstrates both approaches:

接下来，您可以提供候选bean定义的信息。您可以将<qualifier/>标记添加为 标记的子元素，<bean/>然后指定type和 value匹配自定义限定符注释。类型与注释的完全限定类名匹配。或者，为方便起见，如果不存在冲突名称的风险，您可以使用短类名称。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

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

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

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

<context:annotation-config/>

<bean class="example.SimpleMovieCatalog">

<qualifier type="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>

In [Classpath Scanning and Managed Components](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-classpath-scanning), you can see an annotation-based alternative to providing the qualifier metadata in XML. Specifically, see [Providing Qualifier Metadata with Annotations](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-scanning-qualifiers).

在[类路径扫描和托管组件中](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-classpath-scanning)，您可以看到基于注释的替代方法，即在XML中提供限定符元数据。具体来说，请参阅[使用注释提供限定符元数据](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-scanning-qualifiers)。

In some cases, using an annotation without a value may suffice. This can be useful when the annotation serves a more generic purpose and can be applied across several different types of dependencies. For example, you may provide an offline catalog that can be searched when no Internet connection is available. First, define the simple annotation, as the following example shows:

在某些情况下，使用没有值的注释可能就足够了。当注释用于更通用的目的并且可以应用于多种不同类型的依赖项时，这可能很有用。例如，您可以提供可在没有Internet连接时搜索的脱机目录。首先，定义简单注释，如以下示例所示：

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

@Retention(RetentionPolicy.RUNTIME)

@Qualifier

**public** @interface Offline {

}

Then add the annotation to the field or property to be autowired, as shown in the following example:

然后将注释添加到要自动装配的字段或属性中，如以下示例所示：

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

@Autowired

@Offline

**private** MovieCatalog offlineCatalog;

*// ...*

}

|  |  |
| --- | --- |
|  | This line adds the @Offline annotation. |
|  | 此行添加@Offline注释。 |

Now the bean definition only needs a qualifier type, as shown in the following example:

现在bean定义只需要一个限定符type，如下例所示：

<bean class="example.SimpleMovieCatalog">

<qualifier type="Offline"/>

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

</bean>

|  |  |
| --- | --- |
|  | This element specifies the qualifier. |
|  | 此元素指定限定符。 |

You can also define custom qualifier annotations that accept named attributes in addition to or instead of the simple valueattribute. If multiple attribute values are then specified on a field or parameter to be autowired, a bean definition must match all such attribute values to be considered an autowire candidate. As an example, consider the following annotation definition:

您还可以定义除简单value属性之外或代替简单属性接受命名属性的自定义限定符注释。如果随后在要自动装配的字段或参数上指定了多个属性值，则bean定义必须匹配所有此类属性值才能被视为自动装配候选。例如，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();

}

In this case Format is an enum, defined as follows:

在这种情况下Format是一个枚举，定义如下：

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

VHS, DVD, BLURAY

}

The fields to be autowired are annotated with the custom qualifier and include values for both attributes: genre and format, as the following example shows:

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并且format，如以下示例所示：

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

@Autowired

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

**private** MovieCatalog actionVhsCatalog;

@Autowired

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

**private** MovieCatalog comedyVhsCatalog;

@Autowired

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

**private** MovieCatalog actionDvdCatalog;

@Autowired

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

**private** MovieCatalog comedyBluRayCatalog;

*// ...*

}

Finally, the bean definitions should contain matching qualifier values. This example also demonstrates that you can use bean meta attributes instead of the <qualifier/> elements. If available, the <qualifier/> element and its attributes take precedence, but the autowiring mechanism falls back on the values provided within the <meta/> tags if no such qualifier is present, as in the last two bean definitions in the following example:

最后，bean定义应包含匹配的限定符值。此示例还演示了您可以使用bean元属性而不是 <qualifier/>元素。如果可用，则<qualifier/>元素及其属性优先，但<meta/>如果不存在此类限定符，则自动装配机制将回退到标记内提供的值 ，如以下示例中的最后两个bean定义：

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

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

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

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

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

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

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

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

<context:annotation-config/>

<bean class="example.SimpleMovieCatalog">

<qualifier type="MovieQualifier">

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

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

</qualifier>

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

</bean>

<bean class="example.SimpleMovieCatalog">

<qualifier type="MovieQualifier">

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

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

</qualifier>

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

</bean>

<bean class="example.SimpleMovieCatalog">

<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 as Autowiring Qualifiers

使用泛型作为自动装配限定符

In addition to the @Qualifier annotation, you can use Java generic types as an implicit form of qualification. For example, suppose you have the following configuration:

除了@Qualifier注释之外，您还可以使用Java泛型类型作为隐式的限定形式。例如，假设您具有以下配置：

@Configuration

**public** **class** **MyConfiguration** {

@Bean

**public** StringStore stringStore() {

**return** **new** StringStore();

}

@Bean

**public** IntegerStore integerStore() {

**return** **new** IntegerStore();

}

}

Assuming that the preceding beans implement a generic interface, (that is, Store<String> and Store<Integer>), you can @Autowire the Store interface and the generic is used as a qualifier, as the following example shows:

假设前面的bean实现了一个通用接口（即Store<String>和， Store<Integer>），您可以@Autowire将Store接口和泛型用作限定符，如下例所示：

@Autowired

**private** Store<String> s1; *// <String> qualifier, injects the stringStore bean*

@Autowired

**private** Store<Integer> s2; *// <Integer> qualifier, injects the integerStore bean*

Generic qualifiers also apply when autowiring lists, Map instances and arrays. The following example autowires a generic List:

通用限定符也适用于自动装配列表，Map实例和数组。以下示例自动装配通用List：

*// Inject all Store beans as long as they have an <Integer> generic*

*// Store<String> beans will not appear in this list*

@Autowired

**private** List<Store<Integer>> s;

1.9.6. Using CustomAutowireConfigurer

[CustomAutowireConfigurer](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/beans/factory/annotation/CustomAutowireConfigurer.html) is a BeanFactoryPostProcessor that lets you register your own custom qualifier annotation types, even if they are not annotated with Spring’s @Qualifier annotation. The following example shows how to use CustomAutowireConfigurer:

[CustomAutowireConfigurer](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/beans/factory/annotation/CustomAutowireConfigurer.html) 是一个BeanFactoryPostProcessor允许您注册自己的自定义限定符注释类型的，即使它们没有使用Spring的@Qualifier注释进行注释。以下示例显示如何使用CustomAutowireConfigurer：

<bean id="customAutowireConfigurer"

class="org.springframework.beans.factory.annotation.CustomAutowireConfigurer">

<property name="customQualifierTypes">

<set>

<value>example.CustomQualifier</value>

</set>

</property>

</bean>

The AutowireCandidateResolver determines autowire candidates by:

通过以下方式AutowireCandidateResolver确定autowire候选人：

* The autowire-candidate value of each bean definition
* Any default-autowire-candidates patterns available on the <beans/> element
* The presence of @Qualifier annotations and any custom annotations registered with the CustomAutowireConfigurer
* autowire-candidate每个bean定义的值
* 元素default-autowire-candidates上可用的任何模式<beans/>
* @Qualifier注释的存在以及注册的任何自定义注释CustomAutowireConfigurer

When multiple beans qualify as autowire candidates, the determination of a “primary” is as follows: If exactly one bean definition among the candidates has a primary attribute set to true, it is selected.

当多个bean有资格作为autowire候选者时，“primary”的确定如下：如果候选者中只有一个bean定义具有primary 设置为的属性true，则选择它。

1.9.7. Injection with @Resource

Spring also supports injection by using the JSR-250 @Resource annotation (javax.annotation.Resource) on fields or bean property setter methods. This is a common pattern in Java EE: for example, in JSF-managed beans and JAX-WS endpoints. Spring supports this pattern for Spring-managed objects as well.

Spring还通过在字段或bean属性setter方法上使用JSR-250 @Resourceannotation（javax.annotation.Resource）来支持注入。这是Java EE中的常见模式：例如，在JSF管理的bean和JAX-WS端点中。Spring也支持Spring管理对象的这种模式。

@Resource takes a name attribute. By default, Spring interprets that value as the bean name to be injected. In other words, it follows by-name semantics, as demonstrated in the following example:

@Resource采用名称属性。默认情况下，Spring将该值解释为要注入的bean名称。换句话说，它遵循按名称语义，如以下示例所示：

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

**private** MovieFinder movieFinder;

@Resource(name="myMovieFinder")

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

this.movieFinder = movieFinder;

}

}

|  |  |
| --- | --- |
|  | This line injects a @Resource. |

If no name is explicitly specified, the default name is derived from the field name or setter method. In case of a field, it takes the field name. In case of a setter method, it takes the bean property name. The following example is going to have the bean named movieFinder injected into its setter method:

如果未明确指定名称，则默认名称是从字段名称或setter方法派生的。如果是字段，则采用字段名称。在setter方法的情况下，它采用bean属性名称。下面的例子将把bean movieFinder注入其setter方法：

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

**private** MovieFinder movieFinder;

@Resource

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

this.movieFinder = movieFinder;

}

}

|  |  |
| --- | --- |
|  | The name provided with the annotation is resolved as a bean name by the ApplicationContext of which the CommonAnnotationBeanPostProcessor is aware. The names can be resolved through JNDI if you configure Spring’s [SimpleJndiBeanFactory](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/jndi/support/SimpleJndiBeanFactory.html) explicitly. However, we recommend that you rely on the default behavior and use Spring’s JNDI lookup capabilities to preserve the level of indirection.  提供注解的名称解析由一个bean的名称 ApplicationContext，其中的CommonAnnotationBeanPostProcessor知道。如果您[SimpleJndiBeanFactory](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/jndi/support/SimpleJndiBeanFactory.html) 明确配置Spring，则可以通过JNDI解析名称 。但是，我们建议您依赖于默认行为并使用Spring的JNDI查找功能来保留间接级别。 |

In the exclusive case of @Resource usage with no explicit name specified, and similar to @Autowired, @Resource finds a primary type match instead of a specific named bean and resolves well known resolvable dependencies: the BeanFactory,ApplicationContext, ResourceLoader, ApplicationEventPublisher, and MessageSource interfaces.

在专属情况下，@Resource不指定明确的名称，以及类似的使用@Autowired，@Resource发现的主要类型的比赛，而不是一个具体的bean并解决众所周知的解析依存关系：BeanFactory， ApplicationContext，ResourceLoader，ApplicationEventPublisher，和MessageSource 接口。

Thus, in the following example, the customerPreferenceDao field first looks for a bean named "customerPreferenceDao" and then falls back to a primary type match for the type CustomerPreferenceDao:

因此，在以下示例中，customerPreferenceDao字段首先查找名为“customerPreferenceDao”的bean，然后返回到该类型的主要类型匹配 CustomerPreferenceDao：

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

@Resource

**private** CustomerPreferenceDao customerPreferenceDao;

@Resource

**private** ApplicationContext context;

**public** MovieRecommender() {

}

*// ...*

}

|  |  |
| --- | --- |
|  | The context field is injected based on the known resolvable dependency type: ApplicationContext.  context根据已知的可解析依赖类型注入该字段： ApplicationContext。 |

1.9.8. Using @PostConstruct and @PreDestroy

The CommonAnnotationBeanPostProcessor not only recognizes the @Resource annotation but also the JSR-250 lifecycle annotations: javax.annotation.PostConstruct and javax.annotation.PreDestroy. Introduced in Spring 2.5, the support for these annotations offers an alternative to the lifecycle callback mechanism described in [initialization callbacks](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-initializingbean) and [destruction callbacks](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-disposablebean). Provided that the CommonAnnotationBeanPostProcessor is registered within the Spring ApplicationContext, a method carrying one of these annotations is invoked at the same point in the lifecycle as the corresponding Spring lifecycle interface method or explicitly declared callback method. In the following example, the cache is pre-populated upon initialization and cleared upon destruction:

将CommonAnnotationBeanPostProcessor不仅承认了@Resource注解也是JSR-250的生命周期注解：javax.annotation.PostConstruct和 javax.annotation.PreDestroy。在Spring 2.5中引入，对这些注释的支持提供了[初始化回调](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-initializingbean)和 [销毁回调中](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-disposablebean)描述的生命周期回调机制的替代 [方法](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-disposablebean)。如果 CommonAnnotationBeanPostProcessor在Spring中注册ApplicationContext，则在生命周期的同一点调用带有这些注释之一的方法，作为相应的Spring生命周期接口方法或显式声明的回调方法。在以下示例中，缓存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...*

}

}

For details about the effects of combining various lifecycle mechanisms, see [Combining Lifecycle Mechanisms](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-combined-effects).

有关组合各种生命周期机制的效果的详细信息，请参阅 [组合生命周期机制](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-combined-effects)。

|  |  |
| --- | --- |
|  | Like @Resource, the @PostConstruct and @PreDestroy annotation types were a part of the standard Java libraries from JDK 6 to 8. However, the entire javax.annotation package got separated from the core Java modules in JDK 9 and eventually removed in JDK 11. If needed, the javax.annotation-api artifact needs to be obtained via Maven Central now, simply to be added to the application’s classpath like any other library.  例如@Resource，@PostConstruct和@PreDestroy注释类型是JDK 6到8的标准Java库的一部分。但是，整个javax.annotation 包与JDK 9中的核心Java模块分离，最终在JDK 11中删除。如果需要，javax.annotation-api工件需要是现在通过Maven Central获得，只需像任何其他库一样添加到应用程序的类路径中。 |

1.10. Classpath Scanning and Managed Components

Most examples in this chapter use XML to specify the configuration metadata that produces each BeanDefinition within the Spring container. The previous section ([Annotation-based Container Configuration](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-annotation-config)) demonstrates how to provide a lot of the configuration metadata through source-level annotations. Even in those examples, however, the “base” bean definitions are explicitly defined in the XML file, while the annotations drive only the dependency injection. This section describes an option for implicitly detecting the candidate components by scanning the classpath. Candidate components are classes that match against a filter criteria and have a corresponding bean definition registered with the container. This removes the need to use XML to perform bean registration. Instead, you can use annotations (for example, @Component), AspectJ type expressions, or your own custom filter criteria to select which classes have bean definitions registered with the container.

本章中的大多数示例都使用XML来指定BeanDefinition在Spring容器中生成每个元素的配置元数据。上一节（[基于注释的容器配置](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-annotation-config)）演示了如何通过源级注释提供大量配置元数据。但是，即使在这些示例中，“基本”bean定义也在XML文件中显式定义，而注释仅驱动依赖项注入。本节介绍通过扫描类路径隐式检测候选组件的选项。候选组件是与筛选条件匹配的类，并且具有向容器注册的相应bean定义。这消除了使用XML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筛选条件来选择哪些类具有向容器注册的bean定义。

|  |  |
| --- | --- |
|  | Starting with Spring 3.0, many features provided by the Spring JavaConfig project are part of the core Spring Framework. This allows you to define beans using Java rather than using the traditional XML files. Take a look at the @Configuration, @Bean, @Import, and @DependsOn annotations for examples of how to use these new features.  从Spring 3.0开始，Spring JavaConfig项目提供的许多功能都是核心Spring Framework的一部分。这允许您使用Java而不是使用传统的XML文件来定义bean。看看的@Configuration，@Bean， @Import，和@DependsOn注释有关如何使用这些新功能的例子。 |

1.10.1. @Component and Further Stereotype Annotations

@Component和进一步的刻板印象注释

The @Repository annotation is a marker for any class that fulfills the role or stereotype of a repository (also known as Data Access Object or DAO). Among the uses of this marker is the automatic translation of exceptions, as described in [Exception Translation](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/data-access.html#orm-exception-translation).

的@Repository注释是针对满足的存储库（也被称为数据访问对象或DAO）的作用或者固定型的任何类的标记。此标记的用法之一是异常的自动转换，如 [异常转换中所述](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/data-access.html#orm-exception-translation)。

Spring provides further stereotype annotations: @Component, @Service, and @Controller. @Component is a generic stereotype for any Spring-managed component. @Repository, @Service, and @Controller are specializations of @Component for more specific use cases (in the persistence, service, and presentation layers, respectively). Therefore, you can annotate your component classes with @Component, but, by annotating them with @Repository, @Service, or @Controller instead, your classes are more properly suited for processing by tools or associating with aspects. For example, these stereotype annotations make ideal targets for pointcuts. @Repository, @Service, and @Controller can also carry additional semantics in future releases of the Spring Framework. Thus, if you are choosing between using @Component or @Service for your service layer, @Service is clearly the better choice. Similarly, as stated earlier, @Repository is already supported as a marker for automatic exception translation in your persistence layer.

Spring提供进一步典型化注解：@Component，@Service，和 @Controller。@Component是任何Spring管理组件的通用构造型。@Repository，@Service和，@Controller是@Component更具体的用例的专业化（分别在持久性，服务和表示层）。因此，您可以来注解你的组件类有 @Component，但是，通过与注解它们@Repository，@Service或者@Controller ，你的类能更好地被工具处理，或与切面进行关联。例如，这些刻板印象注释成为切入点的理想目标。@Repository，@Service并且@Controller还可以在Spring Framework的未来版本中携带其他语义。因此，如果您在使用之间进行选择@Component或者@Service对于您的服务层，@Service显然是更好的选择。同样，如前所述，@Repository已经支持将其作为持久层中自动异常转换的标记。

1.10.2. Using Meta-annotations and Composed Annotations

使用元注释和组合注释

Many of the annotations provided by Spring can be used as meta-annotations in your own code. A meta-annotation is an annotation that can be applied to another annotation. For example, the @Service annotation mentioned [earlier](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-stereotype-annotations) is meta-annotated with @Component, as the following example shows:

Spring提供的许多注释都可以在您自己的代码中用作元注释。元注释是可以应用于另一个注释的注释。例如，@Service提及的注释[前面](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-stereotype-annotations)是间注释有@Component，如下面的示例所示：

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Component

**public** @interface Service {

*// ....*

}

|  |  |
| --- | --- |
|  | The Component causes @Service to be treated in the same way as @Component. |
|  | 要以同样的方式对待的Component原因。@Service@Component |

You can also combine meta-annotations to create “composed annotations”. For example, the @RestController annotation from Spring MVC is composed of @Controller and @ResponseBody.

您还可以组合元注释来创建“组合注释”。例如，@RestControllerSpring MVC 的注释由@Controller和 组成@ResponseBody。

In addition, composed annotations can optionally redeclare attributes from meta-annotations to allow customization. This can be particularly useful when you want to only expose a subset of the meta-annotation’s attributes. For example, Spring’s@SessionScope annotation hardcodes the scope name to session but still allows customization of the proxyMode. The following listing shows the definition of the SessionScope annotation:

此外，组合注释可以选择从元注释重新声明属性以允许自定义。当您只想公开元注释属性的子集时，这可能特别有用。例如，Spring的@SessionScope注释将范围名称硬编码为session但仍允许自定义proxyMode。以下清单显示了SessionScope注释的定义 ：

@Target({ElementType.TYPE, ElementType.METHOD})

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Scope(WebApplicationContext.SCOPE\_SESSION)

**public** @interface SessionScope {

*/\*\**

*\* Alias for {@link Scope#proxyMode}.*

*\* <p>Defaults to {@link ScopedProxyMode#TARGET\_CLASS}.*

*\*/*

@AliasFor(annotation = Scope.class)

ScopedProxyMode proxyMode() **default** ScopedProxyMode.TARGET\_CLASS;

}

You can then use @SessionScope without declaring the proxyMode as follows:

然后您可以使用@SessionScope而不声明proxyMode如下：

@Service

@SessionScope

**public** **class** **SessionScopedService** {

*// ...*

}

You can also override the value for the proxyMode, as the following example shows:

您还可以覆盖该值proxyMode，如以下示例所示：

@Service

@SessionScope(proxyMode = ScopedProxyMode.INTERFACES)

**public** **class** **SessionScopedUserService** **implements** UserService {

*// ...*

}

For further details, see the [Spring Annotation Programming Model](https://github.com/spring-projects/spring-framework/wiki/Spring-Annotation-Programming-Model) wiki page.

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 can automatically detect stereotyped classes and register corresponding BeanDefinition instances with the ApplicationContext. For example, the following two classes are eligible for such autodetection:

Spring可以自动检测构造型类并注册相应的 BeanDefinition实例ApplicationContext。例如，以下两个类符合此类自动检测的条件：

@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*

}

To autodetect these classes and register the corresponding beans, you need to add @ComponentScan to your @Configurationclass, where the basePackages attribute is a common parent package for the two classes. (Alternatively, you can specify a comma- or semicolon- or space-separated list that includes the parent package of each class.)

要自动检测这些类并注册相应的bean，您需要添加 @ComponentScan到您的@Configuration类，其中该basePackages属性是两个类的公共父包。（或者，您可以指定包含每个类的父包的逗号或分号或空格分隔列表。）

@Configuration

@ComponentScan(basePackages = "org.example")

**public** **class** **AppConfig** {

...

}

|  |  |
| --- | --- |
|  | For brevity, the preceding example could have used the value attribute of the annotation (that is, @ComponentScan("org.example")).  为简洁起见，前面的示例可能使用value了注释的属性（即@ComponentScan("org.example")）。 |

The following alternative uses XML:

以下替代方法使用XML：

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

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

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

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

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

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

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

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

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

</beans>

|  |  |
| --- | --- |
|  | The use of <context:component-scan> implicitly enables the functionality of <context:annotation-config>. There is usually no need to include the <context:annotation-config> element when using <context:component-scan>.  使用<context:component-scan>隐式启用功能 <context:annotation-config>。<context:annotation-config>使用时通常不需要包含 元素<context:component-scan>。 |
|  | 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 be exposed based on security policies in some environments — for example, standalone apps on JDK 1.7.0\_45 and higher (which requires 'Trusted-Library' setup in your manifests — see<https://stackoverflow.com/questions/19394570/java-jre-7u45-breaks-classloader-getresources>).  扫描类路径包需要在类路径中存在相应的目录条目。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，可能不会基于安全策略公开类路径目录 - 例如，JDK 1.7.0\_45及更高版本上的独立应用程序（需要在清单中设置“Trusted-Library” - 请参阅 [https://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. However, make sure that your component classes are exported in your module-info descriptors. If you expect Spring to invoke non-public members of your classes, make sure that they are 'opened' (that is, that they use an opens declaration instead of an exports declaration in your module-info descriptor).  在JDK 9的模块路径（Jigsaw）上，Spring的类路径扫描通常按预期工作。但是，请确保在module-info 描述符中导出组件类。如果您希望Spring调用类的非公共成员，请确保它们已“打开”（即，它们在描述符中使用opens声明而不是exports声明module-info）。 |

Furthermore, the AutowiredAnnotationBeanPostProcessor and CommonAnnotationBeanPostProcessor are both implicitly included when you use the component-scan element. That means that the two components are autodetected and wired together — all without any bean configuration metadata provided in XML.

此外，当您使用component-scan元素时，隐式包含AutowiredAnnotationBeanPostProcessor和CommonAnnotationBeanPostProcessor。这意味着这两个组件是自动检测并连接在一起的 - 所有这些都没有在XML中提供任何bean配置元数据。

|  |  |
| --- | --- |
|  | You can disable the registration of AutowiredAnnotationBeanPostProcessor andCommonAnnotationBeanPostProcessor by including the annotation-config attribute with a value of false.  您可以禁用注册，AutowiredAnnotationBeanPostProcessor并将 CommonAnnotationBeanPostProcessorannotation-config属性包含在值中false。 |

1.10.4. Using Filters to Customize Scanning

使用过滤器自定义扫描

By default, classes annotated with @Component, @Repository, @Service, @Controller, or a custom annotation that itself is annotated with @Component are the only detected candidate components. However, you can modify and extend this behavior by applying custom filters. Add them as includeFilters or excludeFilters parameters of the @ComponentScan annotation (or as include-filter or exclude-filter child elements of the component-scan element). Each filter element requires the type and expression attributes. The following table describes the filtering options:

默认情况下，类注有@Component，@Repository，@Service， @Controller，或者本身都标注有一个自定义的注释@Component是唯一检测到的候选组件。但是，您可以通过应用自定义筛选器来修改和扩展此行为。他们加为includeFilters或excludeFilters 的参数@ComponentScan注释（或include-filter或exclude-filter 在的子元素component-scan元素）。每个过滤器元素都需要type 和expression属性。下表介绍了筛选选项：

|  |  |  |
| --- | --- | --- |
| *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 or 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. |

|  |  |  |
| --- | --- | --- |
| *表5.过滤器类型* | | |
| **过滤器类型** | **示例表达** | **描述** |
| 注释（默认） | org.example.SomeAnnotation | 要在目标组件中的类型级别出现的注释。 |
| 分配 | org.example.SomeClass | 目标组件可分配给（扩展或实现）的类（或接口）。 |
| AspectJ的 | org.example..\*Service+ | 要由目标组件匹配的AspectJ类型表达式。 |
| 正则表达式 | org\.example\.Default.\* | 要由目标组件类名匹配的正则表达式。 |
| 习惯 | org.example.MyTypeFilter | org.springframework.core.type .TypeFilter接口的自定义实现。 |

The following example shows the configuration ignoring all @Repository annotations and using “stub” repositories instead:

以下示例显示忽略所有@Repository注释并使用“存根”存储库的配置：

@Configuration

@ComponentScan(basePackages = "org.example",

includeFilters = @Filter(type = FilterType.REGEX, pattern = ".\*Stub.\*Repository"),

excludeFilters = @Filter(Repository.class))

**public** **class** **AppConfig** {

...

}

The following listing shows the equivalent XML:

以下清单显示了等效的XML：

<beans>

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

<context:include-filter type="regex"

expression=".\*Stub.\*Repository"/>

<context:exclude-filter type="annotation"

expression="org.springframework.stereotype.Repository"/>

</context:component-scan>

</beans>

|  |  |
| --- | --- |
|  | You can also disable the default filters by setting useDefaultFilters=false on the annotation or by providing use-default-filters="false" as an attribute of the <component-scan/> element. This, in effect, disables automatic detection of classes annotated with @Component, @Repository, @Service, @Controller, or @Configuration.  您还可以通过设置useDefaultFilters=false注释或提供元素use-default-filters="false"属性来禁用默认过滤器<component-scan/>。此，实际上，禁用与注解的类自动检测@Component，@Repository， @Service，@Controller，或@Configuration。 |

1.10.5. Defining Bean Metadata within Components

Spring components can also contribute bean definition metadata to the container. You can do this with the same @Beanannotation used to define bean metadata within @Configuration annotated classes. The following example shows how to do so:

Spring组件还可以向容器提供bean定义元数据。您可以@Bean使用用于在带@Configuration 注释的类中定义bean元数据的相同注释来执行此操作。以下示例显示了如何执行此操作：

@Component

**public** **class** **FactoryMethodComponent** {

@Bean

@Qualifier("public")

**public** TestBean publicInstance() {

**return** **new** TestBean("publicInstance");

}

**public** **void** doWork() {

*// Component method implementation omitted*

}

}

The preceding class is a Spring component that has application-specific code in its doWork() method. However, it also contributes a bean definition that has a factory method referring to the method publicInstance(). The @Bean annotation identifies the factory method and other bean definition properties, such as a qualifier value through the @Qualifier annotation. Other method-level annotations that can be specified are @Scope, @Lazy, and custom qualifier annotations.

前面的类是一个Spring组件，在其doWork()方法中具有特定于应用程序的代码 。但是，它还提供了一个bean定义，该定义具有引用该方法的工厂方法publicInstance()。该@Bean注释标识工厂方法和其它bean定义特性，如通过一个限定值@Qualifier注释。可以指定其他方法级别的注解是 @Scope，@Lazy和自定义限定器注解。

|  |  |
| --- | --- |
|  | In addition to its role for component initialization, you can also place the @Lazy annotation on injection points marked with @Autowired or @Inject. In this context, it leads to the injection of a lazy-resolution proxy.  除了组件初始化的作用外，您还可以将@Lazy注释放在标有@Autowired或的注入点上@Inject。在这种情况下，它会导致注入惰性解析代理。 |

Autowired fields and methods are supported, as previously discussed, with additional support for autowiring of @Bean methods. The following example shows how to do so:

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

@Component

**public** **class** **FactoryMethodComponent** {

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

@Bean

@Qualifier("public")

**public** TestBean publicInstance() {

**return** **new** TestBean("publicInstance");

}

*// use of a custom qualifier and autowiring of method parameters*

@Bean

**protected** TestBean protectedInstance(

@Qualifier("public") TestBean spouse,

@Value("#{privateInstance.age}") String country) {

TestBean tb = **new** TestBean("protectedInstance", 1);

tb.setSpouse(spouse);

tb.setCountry(country);

**return** tb;

}

@Bean

**private** TestBean privateInstance() {

**return** **new** TestBean("privateInstance", i++);

}

@Bean

@RequestScope

**public** TestBean requestScopedInstance() {

**return** **new** TestBean("requestScopedInstance", 3);

}

}

The example autowires the String method parameter country to the value of the age property on another bean named privateInstance. A Spring Expression Language element defines the value of the property through the notation #{ <expression> }. For @Value annotations, an expression resolver is preconfigured to look for bean names when resolving expression text.

该示例将Stringmethod参数自动装配country到age 另一个名为的bean 的属性值privateInstance。Spring Expression Language元素通过符号定义属性的值#{ <expression> }。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。

As of Spring Framework 4.3, you may also declare a factory method parameter of type InjectionPoint (or its more specific subclass: DependencyDescriptor) to access the requesting injection point that triggers the creation of the current bean. Note that this applies only to the actual creation of bean instances, not to the injection of existing instances. As a consequence, this feature makes most sense for beans of prototype scope. For other scopes, the factory method only ever sees the injection point that triggered the creation of a new bean instance in the given scope (for example, the dependency that triggered the creation of a lazy singleton bean). You can use the provided injection point metadata with semantic care in such scenarios. The following example shows how to do use InjectionPoint:

从Spring Framework 4.3开始，您还可以声明类型的工厂方法参数 InjectionPoint（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，这仅适用于实例创建bean实例，而不适用于注入现有实例。因此，此功能对原型范围的bean最有意义。对于其他作用域，工厂方法只能看到触发在给定作用域中创建新bean实例的注入点（例如，触发创建惰性单例bean的依赖项）。在这种情况下，您可以使用提供的注入点元数据和语义关注。以下示例显示了如何使用InjectionPoint：

@Component

**public** **class** **FactoryMethodComponent** {

@Bean @Scope("prototype")

**public** TestBean prototypeInstance(InjectionPoint injectionPoint) {

**return** **new** TestBean("prototypeInstance for " + injectionPoint.getMember());

}

}

The @Bean methods in a regular Spring component are processed differently than their counterparts inside a Spring @Configuration class. The difference is that @Component classes are not enhanced with CGLIB to intercept the invocation of methods and fields. CGLIB proxying is the means by which invoking methods or fields within @Bean methods in @Configurationclasses creates bean metadata references to collaborating objects. Such methods are not invoked with normal Java semantics but rather go through the container in order to provide the usual lifecycle management and proxying of Spring beans, even when referring to other beans through programmatic calls to @Bean methods. In contrast, invoking a method or field in a @Beanmethod within a plain @Component class has standard Java semantics, with no special CGLIB processing or other constraints applying.

将@Bean在普通的Spring组件方法比春天里的同行处理方式不同@Configuration类。不同之处在于@Component ，CGLIB不会增强类来拦截方法和字段的调用。CGLIB代理是调用类中@Bean方法中的方法或字段@Configuration创建对协作对象的bean元数据引用的方法。这些方法不是用普通的Java语义调用的，而是通过容器来提供通常的生命周期管理和Spring bean的代理，即使在通过对@Bean方法的编程调用引用其他bean时也是如此。相反，@Bean在plain 中的方法中调用方法或字段@Component class具有标准的Java语义，没有特殊的CGLIB处理或其他约束应用。

|  |  |
| --- | --- |
|  | You may declare @Bean methods as static, allowing for them to be called without creating their containing configuration class as an instance. This makes particular sense when defining post-processor beans (for example, of type BeanFactoryPostProcessor or BeanPostProcessor), since such beans get initialized early in the container lifecycle and should avoid triggering other parts of the configuration at that point.  您可以将@Bean方法声明为static，允许在不创建包含配置类作为实例的情况下调用它们。这在定义后处理器bean（例如，类型BeanFactoryPostProcessor或 BeanPostProcessor）时特别有意义，因为这样的bean在容器生命周期的早期就会初始化，并且应该避免在那时触发配置的其他部分。  Calls to static @Bean methods never get intercepted by the container, not even within @Configuration classes (as described earlier in this section), due to technical limitations: CGLIB subclassing can override only non-static methods. As a consequence, a direct call to another @Bean method has standard Java semantics, resulting in an independent instance being returned straight from the factory method itself.  由于技术限制，对静态@Bean方法的调用永远不会被容器拦截，甚至在@Configuration类中也不会被拦截（如本节前面所述）：CGLIB子类化只能覆盖非静态方法。因此，直接调用另一个@Bean方法具有标准的Java语义，从而导致直接从工厂方法本身返回一个独立的实例。  The Java language visibility of @Bean methods does not have an immediate impact on the resulting bean definition in Spring’s container. You can freely declare your factory methods as you see fit in non-@Configuration classes and also for static methods anywhere. However, regular @Bean methods in @Configuration classes need to be overridable — that is, they must not be declared as private or final.  方法的Java语言可见性@Bean不会立即影响Spring容器中的结果bean定义。您可以根据自己的需要在非@Configuration类中自由声明工厂方法，也可以在任何地方自由声明静态方法。但是，类中的常规@Bean方法@Configuration需要可以覆盖 - 也就是说，它们不能声明为private或final。  @Bean methods are also discovered on base classes of a given component or configuration class, as well as on Java 8 default methods declared in interfaces implemented by the component or configuration class. This allows for a lot of flexibility in composing complex configuration arrangements, with even multiple inheritance being possible through Java 8 default methods as of Spring 4.2.  @Bean还可以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缺省方法上发现方法。这使得在编写复杂的配置安排时具有很大的灵活性，从Spring 4.2开始，甚至可以通过Java 8默认方法实现多重继承。  Finally, a single class may hold multiple @Bean methods for the same bean, as an arrangement of multiple factory methods to use depending on available dependencies at runtime. This is the same algorithm as for choosing the “greediest” constructor or factory method in other configuration scenarios: The variant with the largest number of satisfiable dependencies is picked at construction time, analogous to how the container selects between multiple @Autowired constructors.  最后，单个类可以@Bean为同一个bean 保存多个方法，作为根据运行时可用依赖性使用多个工厂方法的安排。这与在其他配置方案中选择“最贪婪”构造函数或工厂方法的算法相同：在构造时选择具有最大数量的可满足依赖项的变体，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

When a component is autodetected as part of the scanning process, its bean name is generated by the BeanNameGeneratorstrategy known to that scanner. By default, any Spring stereotype annotation (@Component, @Repository, @Service, and@Controller) that contains a name value thereby provides that name to the corresponding bean definition.

当组件作为扫描过程的一部分自动检测时，其bean名称由该扫描程序BeanNameGenerator已知的策略生成。默认情况下，任何Spring刻板印象注释（@Component，@Repository，@Service，并 @Controller包含一个名字）value，从而提供了名字相应的bean定义。

If such an annotation contains no name value or for any other detected component (such as those discovered by custom filters), the default bean name generator returns the uncapitalized non-qualified class name. For example, if the following component classes were detected, the names would be myMovieLister and movieFinderImpl:

如果此类注释不包含任何名称value或任何其他检测到的组件（例如自定义过滤器发现的那些组件），则默认的bean名称生成器将返回未大写的非限定类名称。例如，如果检测到以下组件类，则名称将为：myMovieLister和movieFinderImpl：

@Service("myMovieLister")

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

*// ...*

}

@Repository

**public** **class** **MovieFinderImpl** **implements** MovieFinder {

*// ...*

}

|  |  |
| --- | --- |
|  | If you do not want to rely on the default bean-naming strategy, you can provide a custom bean-naming strategy. First, implement the [BeanNameGenerator](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/beans/factory/support/BeanNameGenerator.html) interface, and be sure to include a default no-arg constructor. Then, provide the fully qualified class name when configuring the scanner, as the following example annotation and bean definition show:  如果您不想依赖默认的bean命名策略，则可以提供自定义bean命名策略。首先，实现 [BeanNameGenerator](https://docs.spring.io/spring-framework/docs/5.1.8.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，在配置扫描程序时提供完全限定的类名，如以下示例注释和bean定义所示： |

@Configuration

@ComponentScan(basePackages = "org.example", nameGenerator = MyNameGenerator.class)

**public** **class** **AppConfig** {

...

}

<beans>

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

name-generator="org.example.MyNameGenerator" />

</beans>

As a general rule, consider specifying the name with the annotation whenever other components may be making explicit references to it. On the other hand, the auto-generated names are adequate whenever the container is responsible for wiring.

作为一般规则，考虑在其他组件可能对其进行显式引用时使用注释指定名称。另一方面，只要容器负责接线，自动生成的名称就足够了。

1.10.7. Providing a Scope for Autodetected Components

As with Spring-managed components in general, the default and most common scope for autodetected components is singleton. However, sometimes you need a different scope that can be specified by the @Scope annotation. You can provide the name of the scope within the annotation, as the following example shows:

与Spring管理的组件一样，自动检测组件的默认和最常见的范围是singleton。但是，有时您需要一个可以由@Scope注释指定的不同范围。您可以在注释中提供范围的名称，如以下示例所示：

@Scope("prototype")

@Repository

**public** **class** **MovieFinderImpl** **implements** MovieFinder {

*// ...*

}

|  |  |
| --- | --- |
|  | @Scope annotations are only introspected on the concrete bean class (for annotated components) or the factory method (for @Bean methods). In contrast to XML bean definitions, there is no notion of bean definition inheritance, and inheritance hierarchies at the class level are irrelevant for metadata purposes.  @Scope注释仅在具体bean类（对于带注释的组件）或工厂方法（对于@Bean方法）上进行了内省。与XML bean定义相比，没有bean定义继承的概念，类级别的继承层次结构与元数据目的无关。 |

For details on web-specific scopes such as “request” or “session” in a Spring context, see [Request, Session, Application, and WebSocket Scopes](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-other). As with the pre-built annotations for those scopes, you may also compose your own scoping annotations by using Spring’s meta-annotation approach: for example, a custom annotation meta-annotated with @Scope("prototype"), possibly also declaring a custom scoped-proxy mode.

有关特定于Web的范围（如Spring上下文中的“request”或“session”）的详细信息，请参阅[请求，会话，应用程序和WebSocket范围](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-other)。与这些范围的预构建注释一样，您也可以使用Spring的元注释方法编写自己的范围注释：例如，使用元注释的自定义注释@Scope("prototype")，可能还会声明自定义范围代理模式。

|  |  |
| --- | --- |
|  | To provide a custom strategy for scope resolution rather than relying on the annotation-based approach, you can implement the [ScopeMetadataResolver](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/context/annotation/ScopeMetadataResolver.html) interface. Be sure to include a default no-arg constructor. Then you can provide the fully qualified class name when configuring the scanner, as the following example of both an annotation and a bean definition shows:  要为范围解析提供自定义策略而不是依赖基于注释的方法，您可以实现该 [ScopeMetadataResolver](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/context/annotation/ScopeMetadataResolver.html) 接口。请确保包含默认的无参数构造函数。然后，您可以在配置扫描程序时提供完全限定的类名，因为以下注释和bean定义示例显示： |

@Configuration

@ComponentScan(basePackages = "org.example", scopeResolver = MyScopeResolver.class)

**public** **class** **AppConfig** {

...

}

<beans>

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

</beans>

When using certain non-singleton scopes, it may be necessary to generate proxies for the scoped objects. The reasoning is described in [Scoped Beans as Dependencies](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-other-injection). For this purpose, a scoped-proxy attribute is available on the component-scan element. The three possible values are: no, interfaces, and targetClass. For example, the following configuration results in standard JDK dynamic proxies:

使用某些非单例作用域时，可能需要为作用域对象生成代理。这种推理在[Scoped Beans中](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-other-injection)描述[为Dependencies](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-other-injection)。为此，component-scan元素上提供了scoped-proxy属性。三个可能的值是：no，interfaces，和targetClass。例如，以下配置导致标准JDK动态代理：

@Configuration

@ComponentScan(basePackages = "org.example", scopedProxy = ScopedProxyMode.INTERFACES)

**public** **class** **AppConfig** {

...

}

<beans>

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

</beans>

1.10.8. Providing Qualifier Metadata with Annotations

使用注释提供限定符元数据

The @Qualifier annotation is discussed in [Fine-tuning Annotation-based Autowiring with Qualifiers](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-autowired-annotation-qualifiers). The examples in that section demonstrate the use of the @Qualifier annotation and custom qualifier annotations to provide fine-grained control when you resolve autowire candidates. Because those examples were based on XML bean definitions, the qualifier metadata was provided on the candidate bean definitions by using the qualifier or meta child elements of the bean element in the XML. When relying upon classpath scanning for auto-detection of components, you can provide the qualifier metadata with type-level annotations on the candidate class. The following three examples demonstrate this technique:

在@Qualifier注释中讨论[与预选赛微调基于注解的自动连接](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-autowired-annotation-qualifiers)。该部分中的示例演示了@Qualifier在解析自动线候选时使用注释和自定义限定符注释来提供细粒度控制。因为这些示例基于XML bean定义，所以通过使用XML中 元素的qualifier或meta元素bean元素在候选bean定义上提供限定符元数据。当依靠类路径扫描来自动检测组件时，可以在候选类上为类型级注释提供限定符元数据。以下三个示例演示了此技术：

@Component

@Qualifier("Action")

**public** **class** **ActionMovieCatalog** **implements** MovieCatalog {

*// ...*

}

@Component

@Genre("Action")

**public** **class** **ActionMovieCatalog** **implements** MovieCatalog {

*// ...*

}

@Component

@Offline

**public** **class** **CachingMovieCatalog** **implements** MovieCatalog {

*// ...*

}

|  |  |
| --- | --- |
|  | As with most annotation-based alternatives, keep in mind that the annotation metadata is bound to the class definition itself, while the use of XML allows for multiple beans of the same type to provide variations in their qualifier metadata, because that metadata is provided per-instance rather than per-class.  与大多数基于注释的备选方案一样，请记住注释元数据绑定到类定义本身，而XML的使用允许多个相同类型的bean在其限定符元数据中提供变体，因为每个元数据都是按照 - 实例而不是每班。 |

1.10.9. Generating an Index of Candidate Components

While classpath scanning is very fast, it is possible to improve the startup performance of large applications by creating a static list of candidates at compilation time. In this mode, all modules that are target of component scan must use this mechanism.

虽然类路径扫描速度非常快，但可以通过在编译时创建候选的静态列表来提高大型应用程序的启动性能。在此模式下，所有作为组件扫描目标的模块都必须使用此机制。

|  |  |
| --- | --- |
|  | Your existing @ComponentScan or <context:component-scan directives must stay as is to request the context to scan candidates in certain packages. When the ApplicationContext detects such an index, it automatically uses it rather than scanning the classpath.  您的现有@ComponentScan或<context:component-scan指令必须保持原样，以请求上下文扫描某些包中的候选项。当ApplicationContext检测到这样的索引时，它会自动使用它而不是扫描类路径。 |

To generate the index, add an additional dependency to each module that contains components that are targets for component scan directives. The following example shows how to do so with Maven:

要生成索引，请为包含组件扫描指令目标的组件的每个模块添加其他依赖项。以下示例显示了如何使用Maven执行此操作：

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context-indexer</artifactId>

<version>5.1.8.RELEASE</version>

<optional>true</optional>

</dependency>

</dependencies>

With Gradle 4.5 and earlier, the dependency should be declared in the compileOnly configuration, as shown in the following example:

对于Gradle 4.5及更早版本，应在compileOnly 配置中声明依赖项，如以下示例所示：

dependencies {

compileOnly "org.springframework:spring-context-indexer:5.1.8.RELEASE"

}

With Gradle 4.6 and later, the dependency should be declared in the annotationProcessor configuration, as shown in the following example:

使用Gradle 4.6及更高版本时，应在annotationProcessor 配置中声明依赖项，如以下示例所示：

dependencies {

annotationProcessor "org.springframework:spring-context-indexer:5.1.8.RELEASE"

}

That process generates a META-INF/spring.components file that is included in the jar file.

该进程生成一个META-INF/spring.components包含在jar文件中的文件。

|  |  |
| --- | --- |
|  | When working with this mode in your IDE, the spring-context-indexer must be registered as an annotation processor to make sure the index is up-to-date when candidate components are updated.  在IDE中使用此模式时，spring-context-indexer必须将其注册为注释处理器，以确保在更新候选组件时索引是最新的。 |
|  | The index is enabled automatically when a META-INF/spring.components is found on the classpath. If an index is partially available for some libraries (or use cases) but could not be built for the whole application, you can fallback to a regular classpath arrangement (as though no index was present at all) by setting spring.index.ignore to true, either as a system property or in a spring.properties file at the root of the classpath.  META-INF/spring.components在类路径中找到 a时，将自动启用索引。如果索引部分可用一些库（或用例），但整个应用程序无法建立，可以通过设置回退到普通类路径安排（好像没有索引存在的话）spring.index.ignore来 true，无论是作为一个系统属性或spring.properties类路径根目录下的文件。 |

1.11. Using JSR 330 Standard Annotations

Starting with Spring 3.0, Spring offers support for JSR-330 standard annotations (Dependency Injection). Those annotations are scanned in the same way as the Spring annotations. To use them, you need to have the relevant jars in your classpath.

从Spring 3.0开始，Spring提供对JSR-330标准注释（依赖注入）的支持。这些注释的扫描方式与Spring注释相同。要使用它们，您需要在类路径中包含相关的jar。

|  |  |
| --- | --- |
|  | If you use Maven, the javax.inject artifact is available in the standard Maven repository (<https://repo1.maven.org/maven2/javax/inject/javax.inject/1/>). You can add the following dependency to your file pom.xml:  如果您使用Maven，则javax.inject工件可在标准Maven存储库中找到（<https://repo1.maven.org/maven2/javax/inject/javax.inject/1/>）。您可以将以下依赖项添加到文件pom.xml：  <dependency>  <groupId>javax.inject</groupId>  <artifactId>javax.inject</artifactId>  <version>1</version>  </dependency> |

1.11.1. Dependency Injection with @Inject and @Named

Instead of @Autowired, you can use @javax.inject.Inject as follows:

而不是@Autowired，您可以使用@javax.inject.Inject如下：

**import** javax.inject.Inject;

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

**private** MovieFinder movieFinder;

@Inject

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

this.movieFinder = movieFinder;

}

**public** **void** listMovies() {

this.movieFinder.findMovies(...);

...

}

}

As with @Autowired, you can use @Inject at the field level, method level and constructor-argument level. Furthermore, you may declare your injection point as a Provider, allowing for on-demand access to beans of shorter scopes or lazy access to other beans through a Provider.get() call. The following example offers a variant of the preceding example:

与此同时@Autowired，您可以@Inject在字段级别，方法级别和构造函数 - 参数级别使用。此外，您可以将注入点声明为a Provider，允许按需访问较短范围的bean或通过Provider.get()调用对其他bean进行延迟访问。以下示例提供了上述示例的变体：

**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(...);

...

}

}

If you would like to use a qualified name for the dependency that should be injected, you should use the @Named annotation, as the following example shows:

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

**import** javax.inject.Inject;

**import** javax.inject.Named;

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

**private** MovieFinder movieFinder;

@Inject

**public** **void** setMovieFinder(@Named("main") MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

*// ...*

}

As with @Autowired, @Inject can also be used with java.util.Optional or @Nullable. This is even more applicable here, since @Inject does not have a required attribute. The following pair of examples show how to use @Inject and @Nullable:

与之一样@Autowired，@Inject也可以与java.util.Optional或 一起使用@Nullable。这更适用于此，因为@Inject没有required属性。以下一对示例显示了如何使用@Inject和 @Nullable：

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

@Inject

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

...

}

}

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

@Inject

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

...

}

}

1.11.2. @Named and @ManagedBean: Standard Equivalents to the @Component Annotation

@Named和@ManagedBean：@Component注释的标准等价物

Instead of @Component, you can use @javax.inject.Named or javax.annotation.ManagedBean, as the following example shows:

@Component您可以使用@javax.inject.Named或代替，javax.annotation.ManagedBean如下例所示：

**import** javax.inject.Inject;

**import** javax.inject.Named;

@Named("movieListener") *// @ManagedBean("movieListener") could be used as well*

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

**private** MovieFinder movieFinder;

@Inject

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

this.movieFinder = movieFinder;

}

*// ...*

}

It is very common to use @Component without specifying a name for the component. @Named can be used in a similar fashion, as the following example shows:

在@Component不指定组件名称的情况下使用是很常见的。 @Named可以以类似的方式使用，如以下示例所示：

**import** javax.inject.Inject;

**import** javax.inject.Named;

@Named

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

**private** MovieFinder movieFinder;

@Inject

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

this.movieFinder = movieFinder;

}

*// ...*

}

When you use @Named or @ManagedBean, you can use component scanning in the exact same way as when you use Spring annotations, as the following example shows:

使用@Named或时@ManagedBean，可以使用与使用Spring注释时完全相同的方式使用组件扫描，如以下示例所示：

@Configuration

@ComponentScan(basePackages = "org.example")

**public** **class** **AppConfig** {

...

}

|  |  |
| --- | --- |
|  | In contrast to @Component, the JSR-330 @Named and the JSR-250 ManagedBean annotations are not composable. You should use Spring’s stereotype model for building custom component annotations.  与此相反@Component，JSR-330 @Named和JSR-250 ManagedBean 注释不可组合。您应该使用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

When you work with standard annotations, you should know that some significant features are not available, as the following table shows:

使用标准注释时，您应该知道某些重要功能不可用，如下表所示：

|  |  |  |
| --- | --- | --- |
| *Table 6. Spring component model elements versus JSR-330 variants* | | |
| **Spring** | **javax.inject.\*** | **javax.inject restrictions / comments** |
| @Autowired | @Inject | @Inject has no 'required' attribute. Can be used with Java 8’s Optionalinstead. |
| @Component | @Named / @ManagedBean | JSR-330 does not provide a composable model, only 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](https://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, only with a shorter get() method name. It can also be used in combination with Spring’s @Autowiredor with non-annotated constructors and setter methods. |

|  |  |  |
| --- | --- | --- |
| *表6. Spring组件模型元素与JSR-330变体* | | |
| **弹簧** | **javax.inject。\*** | **javax.inject限制/评论** |
| @Autowired | @注入 | @Inject没有“必需”属性。可以与Java 8一起使用Optional。 |
| @零件 | @Named / @ManagedBean | JSR-330不提供可组合模型，只是一种识别命名组件的方法。 |
| @Scope（ “单”） | @辛格尔顿 | JSR-330的默认范围就像Spring一样prototype。但是，为了使其与Spring的一般默认值保持一致，singleton默认情况下在Spring容器中声明的JSR-330 bean是一个默认值。为了使用除以外的范围singleton，您应该使用Spring的@Scope注释。javax.inject还提供了 [@Scope](https://download.oracle.com/javaee/6/api/javax/inject/Scope.html)注释。然而，这个仅用于创建自己的注释。 |
| @Qualifier | @Qualifier / @Named | javax.inject.Qualifier只是构建自定义限定符的元注释。具体String限定符（如@Qualifier带有值的Spring ）可以通过关联javax.inject.Named。 |
| @值 | - | 没有等价物 |
| @需要 | - | 没有等价物 |
| @懒 | - | 没有等价物 |
| 的ObjectFactory | 提供商 | javax.inject.Provider是Spring的直接替代品ObjectFactory，只有较短的get()方法名称。它也可以与Spring @Autowired或非注释构造函数和setter方法结合使用。 |

1.12. Java-based Container Configuration

This section covers how to use annotations in your Java code to configure the Spring container. It includes the following topics:

* [Basic Concepts: @Bean and @Configuration](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-java-basic-concepts)
* [Instantiating the Spring Container by Using AnnotationConfigApplicationContext](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-java-instantiating-container)
* [Using the @Bean Annotation](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-java-bean-annotation)
* [Using the @Configuration annotation](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-java-configuration-annotation)
* [Composing Java-based Configurations](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-java-composing-configuration-classes)
* [Bean Definition Profiles](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-definition-profiles)
* [PropertySource Abstraction](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-property-source-abstraction)
* [Using @PropertySource](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-using-propertysource)
* [Placeholder Resolution in Statements](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-placeholder-resolution-in-statements)

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

* [基本概念：@Bean和@Configuration](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-java-basic-concepts)
* [使用。实例化Spring容器 AnnotationConfigApplicationContext](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-java-instantiating-container)
* [使用@Bean注释](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-java-bean-annotation)
* [使用@Configuration注释](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-java-configuration-annotation)
* [编写基于Java的配置](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-java-composing-configuration-classes)
* [Bean定义配置文件](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-definition-profiles)
* [PropertySource 抽象化](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-property-source-abstraction)
* [运用 @PropertySource](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-using-propertysource)
* [占位符决议在声明中](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-placeholder-resolution-in-statements)

1.12.1. Basic Concepts: @Bean and @Configuration

The central artifacts in Spring’s new Java-configuration support are @Configuration-annotated classes and @Bean-annotated methods.

Spring的新Java配置支持中的中心工件是 @Configuration注释类和@Bean注释方法。

The @Bean annotation is used to indicate that a method instantiates, configures, and initializes a new object to be managed by the Spring IoC container. For those familiar with Spring’s <beans/> XML configuration, the @Bean annotation plays the same role as the <bean/> element. You can use @Bean-annotated methods with any Spring @Component. However, they are most often used with @Configuration beans.

该@Bean注释被用于指示一个方法实例，配置和初始化为通过Spring IoC容器进行管理的新对象。对于那些熟悉Spring的<beans/>XML配置的人来说，@Bean注释与<bean/>元素扮演的角色相同。你可以@Bean在任何Spring中使用-annotated方法 @Component。但是，它们最常用于@Configuration豆类。

Annotating a class with @Configuration indicates that its primary purpose is as a source of bean definitions. Furthermore, @Configuration classes let inter-bean dependencies be defined by calling other @Bean methods in the same class. The simplest possible @Configuration class reads as follows:

对类进行注释@Configuration表明其主要目的是作为bean定义的来源。此外，@Configuration类允许通过调用@Bean同一类中的其他方法来定义bean间依赖关系。最简单的@Configuration类如下：

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** MyService myService() {

**return** **new** MyServiceImpl();

}

}

The preceding AppConfig class is equivalent to the following Spring <beans/> XML:

上面的AppConfig类等效于以下Spring <beans/>XML：

<beans>

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

</beans>

Full @Configuration vs “lite” @Bean mode?

完整@Configuration vs“lite”@Bean模式？

When @Bean methods are declared within classes that are not annotated with @Configuration, they are referred to as being processed in a “lite” mode. Bean methods declared in a @Component or even in a plain old class are considered to be “lite”, with a different primary purpose of the containing class and a @Bean method being a sort of bonus there. For example, service components may expose management views to the container through an additional @Bean method on each applicable component class. In such scenarios, @Bean methods are a general-purpose factory method mechanism.

当@Bean在未注释的类中声明方法时 @Configuration，它们被称为以“精简”模式处理。在一个@Component或甚至在一个普通的旧类中声明的Bean方法被认为是“精简”，包含类的主要目的不同，并且@Bean方法在那里是一种奖励。例如，服务组件可以通过@Bean每个适用组件类的附加方法将管理视图公开给容器。在这种情况下，@Bean方法是通用的工厂方法机制。

Unlike full @Configuration, lite @Bean methods cannot declare inter-bean dependencies. Instead, they operate on their containing component’s internal state and, optionally, on arguments that they may declare. Such a @Bean method should therefore not invoke other @Bean methods. Each such method is literally only a factory method for a particular bean reference, without any special runtime semantics. The positive side-effect here is that no CGLIB subclassing has to be applied at runtime, so there are no limitations in terms of class design (that is, the containing class may be final and so forth).

与full不同@Configuration，lite @Bean方法不能声明bean间依赖关系。相反，它们对其包含组件的内部状态进行操作，并且可选地，对它们可以声明的参数进行操作。@Bean因此，这种方法不应该引用其他 @Bean方法。每个这样的方法实际上只是特定bean引用的工厂方法，没有任何特殊的运行时语义。这里的积极副作用是不必在运行时应用CGLIB子类，因此在类设计方面没有限制（也就是说，包含类可能是final等等）。

In common scenarios, @Bean methods are to be declared within @Configuration classes, ensuring that “full” mode is always used and that cross-method references therefore get redirected to the container’s lifecycle management. This prevents the same @Bean method from accidentally being invoked through a regular Java call, which helps to reduce subtle bugs that can be hard to track down when operating in “lite” mode.

在常见的场景中，@Bean方法将在@Configuration类中声明，确保始终使用“完整”模式，并因此将交叉方法引用重定向到容器的生命周期管理。这可以防止@Bean通过常规Java调用意外地调用相同的 方法，这有助于减少在“精简”模式下操作时难以跟踪的细微错误。

The @Bean and @Configuration annotations are discussed in depth in the following sections. First, however, we cover the various ways of creating a spring container using by Java-based configuration.

@Bean和@Configuration注解的深度在以下章节中讨论。首先，我们将介绍使用基于Java的配置创建弹簧容器的各种方法。

1.12.2. Instantiating the Spring Container by Using AnnotationConfigApplicationContext

The following sections document Spring’s AnnotationConfigApplicationContext, introduced in Spring 3.0. This versatile ApplicationContext implementation is capable of accepting not only @Configuration classes as input but also plain @Component classes and classes annotated with JSR-330 metadata.

以下部分AnnotationConfigApplicationContext介绍了在Spring 3.0中引入的Spring。这种通用ApplicationContext实现不仅能够接受@Configuration类作为输入，还能接受 @Component使用JSR-330元数据注释的普通类和类。

When @Configuration classes are provided as input, the @Configuration class itself is registered as a bean definition and all declared @Bean methods within the class are also registered as bean definitions.

当@Configuration提供类作为输入时，@Configuration类本身被注册为bean定义，并且@Bean类中的所有声明的方法也被注册为bean定义。

When @Component and JSR-330 classes are provided, they are registered as bean definitions, and it is assumed that DI metadata such as @Autowired or @Inject are used within those classes where necessary.

当@Component提供JSR-330类时，它们被注册为bean定义，并且假定DI元数据例如@Autowired或@Inject在必要时在这些类中使用。

Simple Construction

In much the same way that Spring XML files are used as input when instantiating a ClassPathXmlApplicationContext, you can use @Configuration classes as input when instantiating an AnnotationConfigApplicationContext. This allows for completely XML-free usage of the Spring container, as the following example shows:

与实例化a时Spring XML文件用作输入的方式大致相同 ClassPathXmlApplicationContext，可以@Configuration在实例化时使用类作为输入AnnotationConfigApplicationContext。这允许完全无XML使用Spring容器，如以下示例所示：

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

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(AppConfig.class);

MyService myService = ctx.getBean(MyService.class);

myService.doStuff();

}

As mentioned earlier, AnnotationConfigApplicationContext is not limited to working only with @Configuration classes. Any @Component or JSR-330 annotated class may be supplied as input to the constructor, as the following example shows:

如前所述，AnnotationConfigApplicationContext并不仅限于使用@Configuration类。@Component可以将任何或JSR-330带注释的类作为输入提供给构造函数，如以下示例所示：

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

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(MyServiceImpl.class, Dependency1.class, Dependency2.class);

MyService myService = ctx.getBean(MyService.class);

myService.doStuff();

}

The preceding example assumes that MyServiceImpl, Dependency1, and Dependency2 use Spring dependency injection annotations such as @Autowired.

前面的例子中假定MyServiceImpl，Dependency1以及Dependency2使用Spring依赖注入注解，例如@Autowired。

Building the Container Programmatically by Using register(Class<?>…​)

You can instantiate an AnnotationConfigApplicationContext by using a no-arg constructor and then configure it by using the register() method. This approach is particularly useful when programmatically building an AnnotationConfigApplicationContext. The following example shows how to do so:

您可以AnnotationConfigApplicationContext使用no-arg构造函数实例化一个，然后使用该register()方法对其进行配置。这种方法在以编程方式构建时特别有用AnnotationConfigApplicationContext。以下示例显示了如何执行此操作：

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

AnnotationConfigApplicationContext ctx = **new** AnnotationConfigApplicationContext();

ctx.register(AppConfig.class, OtherConfig.class);

ctx.register(AdditionalConfig.class);

ctx.refresh();

MyService myService = ctx.getBean(MyService.class);

myService.doStuff();

}

Enabling Component Scanning with scan(String…​)

To enable component scanning, you can annotate your @Configuration class as follows:

要启用组件扫描，您可以@Configuration按如下方式注释您的类：

@Configuration

@ComponentScan(basePackages = "com.acme")

**public** **class** **AppConfig** {

...

}

|  |  |
| --- | --- |
|  | This annotation enables component scanning.  此注释可启用组件扫描。 |
|  | Experienced Spring users may be familiar with the XML declaration equivalent from Spring’s context:namespace, shown in the following example:  有经验的Spring用户可能熟悉与Spring context:命名空间等效的XML声明，如下例所示：  <beans>  <context:component-scan base-package="com.acme"/>  </beans> |

In the preceding example, the com.acme package is scanned to look for any @Component-annotated classes, and those classes are registered as Spring bean definitions within the container. AnnotationConfigApplicationContext exposes the scan(String…​) method to allow for the same component-scanning functionality, as the following example shows:

在前面的示例中，com.acme扫描包以查找任何已 @Component注释的类，并将这些类注册为容器中的Spring bean定义。AnnotationConfigApplicationContext公开 scan(String…​)方法以允许相同的组件扫描功能，如以下示例所示：

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

AnnotationConfigApplicationContext ctx = **new** AnnotationConfigApplicationContext();

ctx.scan("com.acme");

ctx.refresh();

MyService myService = ctx.getBean(MyService.class);

}

|  |  |
| --- | --- |
|  | Remember that @Configuration classes are [meta-annotated](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-meta-annotations) with @Component, so they are candidates for component-scanning. In the preceding example, assuming that AppConfig is declared within the com.acmepackage (or any package underneath), it is picked up during the call to scan(). Upon refresh(), all its @Beanmethods are processed and registered as bean definitions within the container.  请记住，@Configuration类是[元注释](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-meta-annotations) 的@Component，因此它们是组件扫描的候选者。在前面的示例中，假设AppConfig在com.acme包（或下面的任何包）中声明了它，它在调用期间被拾取scan()。之后refresh()，它的所有@Bean 方法都被处理并在容器中注册为bean定义。 |

Support for Web Applications with AnnotationConfigWebApplicationContext

A WebApplicationContext variant of AnnotationConfigApplicationContext is available with AnnotationConfigWebApplicationContext. You can use this implementation when configuring the Spring ContextLoaderListener servlet listener, Spring MVC DispatcherServlet, and so forth. The following web.xml snippet configures a typical Spring MVC web application (note the use of the contextClass context-param and init-param):

可用的WebApplicationContext变体。在配置Spring servlet侦听器，Spring MVC等时 ，可以使用此实现。以下代码段配置典型的Spring MVC Web应用程序（请注意context-param和init-param的使用）：AnnotationConfigApplicationContextAnnotationConfigWebApplicationContextContextLoaderListenerDispatcherServletweb.xmlcontextClass

<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 is a method-level annotation and a direct analog of the XML <bean/> element. The annotation supports some of the attributes offered by <bean/>, such as: \* [init-method](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-initializingbean) \* [destroy-method](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-disposablebean) \* [autowiring](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-autowire) \* name.

@Bean是方法级注释和XML <bean/>元素的直接模拟。注释支持一些提供的属性<bean/>，例如：\* [init-method](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-initializingbean) \* [destroy-method](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-lifecycle-disposablebean) \* [autowiring](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-autowire) \* name。

You can use the @Bean annotation in a @Configuration-annotated or in a @Component-annotated class.

您可以在带@Bean注释的类@Configuration或带 注释的类中使用注释@Component。

Declaring a Bean

To declare a bean, you can annotate a method with the @Bean annotation. You use this method to register a bean definition within an ApplicationContext of the type specified as the method’s return value. By default, the bean name is the same as the method name. The following example shows a @Bean method declaration:

要声明bean，可以使用注释注释方法@Bean。您可以使用此方法在ApplicationContext指定为方法的返回值的类型中注册bean定义。默认情况下，bean名称与方法名称相同。以下示例显示了@Bean方法声明：

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** TransferServiceImpl transferService() {

**return** **new** TransferServiceImpl();

}

}

The preceding configuration is exactly equivalent to the following Spring XML:

上述配置与以下Spring XML完全等效：

<beans>

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

</beans>

Both declarations make a bean named transferService available in the ApplicationContext, bound to an object instance of type TransferServiceImpl, as the following text image shows:

这两个声明都将一个名为transferServiceavailable 的bean命名为ApplicationContext绑定到类型的对象实例，TransferServiceImpl如下图所示：

transferService -> com.acme.TransferServiceImpl

You can also declare your @Bean method with an interface (or base class) return type, as the following example shows:

您还可以@Bean使用接口（或基类）返回类型声明您的方法，如以下示例所示：

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** TransferService transferService() {

**return** **new** TransferServiceImpl();

}

}

However, this limits the visibility for advance type prediction to the specified interface type (TransferService). Then, with the full type (TransferServiceImpl) known to the container only once, the affected singleton bean has been instantiated. Non-lazy singleton beans get instantiated according to their declaration order, so you may see different type matching results depending on when another component tries to match by a non-declared type (such as @Autowired TransferServiceImpl, which resolves only once the transferService bean has been instantiated).

但是，这会将高级类型预测的可见性限制为指定的接口类型（TransferService）。然后，TransferServiceImpl只有容器已知的完整类型（）一次，已经实例化了受影响的单例bean。非延迟单例bean根据其声明顺序进行实例化，因此您可能会看到不同的类型匹配结果，具体取决于另一个组件何时尝试通过非声明类型进行匹配（例如@Autowired TransferServiceImpl，只有transferService在实例化bean之后才会解析）。

|  |  |
| --- | --- |
|  | If you consistently refer to your types by a declared service interface, your @Bean return types may safely join that design decision. However, for components that implement several interfaces or for components potentially referred to by their implementation type, it is safer to declare the most specific return type possible (at least as specific as required by the injection points that refer to your bean).  如果您始终通过声明的服务接口引用您的类型，则您的 @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可能由其实现类型引用的组件，更可能声明可能的最具体的返回类型（至少与引用您的bean的注入点所需的具体相同）。 |

Bean Dependencies

A @Bean-annotated method can have an arbitrary number of parameters that describe the dependencies required to build that bean. For instance, if our TransferService requires an AccountRepository, we can materialize that dependency with a method parameter, as the following example shows:

带@Bean注释的方法可以有任意数量的参数来描述构建该bean所需的依赖关系。例如，如果我们TransferService 需要a AccountRepository，我们可以使用方法参数来实现该依赖关系，如下例所示：

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** TransferService transferService(AccountRepository accountRepository) {

**return** **new** TransferServiceImpl(accountRepository);

}

}

The resolution mechanism is pretty much identical to constructor-based dependency injection. See [the relevant section](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-constructor-injection) for more details.

解析机制与基于构造函数的依赖注入非常相似。[有关](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-constructor-injection)详细信息，请参阅[相关部分](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-constructor-injection)。

Receiving Lifecycle Callbacks

Any classes defined with the @Bean annotation support the regular lifecycle callbacks and can use the @PostConstruct and @PreDestroy annotations from JSR-250. See [JSR-250 annotations](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-postconstruct-and-predestroy-annotations) for further details.

使用@Bean注释定义的任何类都支持常规生命周期回调，并且可以使用JSR-250中的注释@PostConstruct和@PreDestroy注释。有关更多详细信息，请参阅 [JSR-250注释](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-postconstruct-and-predestroy-annotations)。

The regular Spring [lifecycle](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-nature) callbacks are fully supported as well. If a bean implements InitializingBean, DisposableBean, or Lifecycle, their respective methods are called by the container.

完全支持常规的Spring [生命周期](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-nature)回调。如果bean实现InitializingBean，DisposableBean或者Lifecycle它们各自的方法由容器调用。

The standard set of \*Aware interfaces (such as [BeanFactoryAware](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-beanfactory), [BeanNameAware](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-aware), [MessageSourceAware](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#context-functionality-messagesource),[ApplicationContextAware](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-aware), and so on) are also fully supported.

还完全支持标准\*Aware接口集（例如[BeanFactoryAware](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-beanfactory)， [BeanNameAware](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-aware)， [MessageSourceAware](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#context-functionality-messagesource)， [ApplicationContextAware](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-aware)等）。

The @Bean annotation supports specifying arbitrary initialization and destruction callback methods, much like Spring XML’s init-method and destroy-method attributes on the bean element, as the following example shows:

该@Bean注释支持指定任意初始化和销毁回调方法，就像春天XML的init-method，并destroy-method在属性上的bean元素，如下例所示：

**public** **class** **BeanOne** {

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

*// initialization logic*

}

}

**public** **class** **BeanTwo** {

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

*// destruction logic*

}

}

@Configuration

**public** **class** **AppConfig** {

@Bean(initMethod = "init")

**public** BeanOne beanOne() {

**return** **new** BeanOne();

}

@Bean(destroyMethod = "cleanup")

**public** BeanTwo beanTwo() {

**return** **new** BeanTwo();

}

}

|  |  |
| --- | --- |
|  | By default, beans defined with Java configuration that have a public close or shutdown method are automatically enlisted with a destruction callback. If you have a public close or shutdown method and you do not wish for it to be called when the container shuts down, you can add @Bean(destroyMethod="") to your bean definition to disable the default (inferred) mode.  默认情况下，使用Java配置定义的具有public close或shutdown method的bean 会自动使用销毁回调登记。如果您有公共 close或shutdown方法，并且您不希望在容器关闭时调用它，则可以添加@Bean(destroyMethod="")到bean定义以禁用默认(inferred)模式。  You may want to do that by default for a resource that you acquire with JNDI, as its lifecycle is managed outside the application. In particular, make sure to always do it for a DataSource, as it is known to be problematic on Java EE application servers.  对于使用JNDI获取的资源，您可能希望默认执行此操作，因为其生命周期在应用程序之外进行管理。特别是，确保始终为a执行此操作DataSource，因为已知它在Java EE应用程序服务器上存在问题。  The following example shows how to prevent an automatic destruction callback for a DataSource:  以下示例显示如何防止自动销毁回调 DataSource：  @Bean(destroyMethod="")  **public** DataSource dataSource() **throws** NamingException {  **return** (DataSource) jndiTemplate.lookup("MyDS");  }  Also, with @Bean methods, you typically use programmatic JNDI lookups, either by using Spring’s JndiTemplateor JndiLocatorDelegate helpers or straight JNDI InitialContext usage but not the JndiObjectFactoryBeanvariant (which would force you to declare the return type as the FactoryBean type instead of the actual target type, making it harder to use for cross-reference calls in other @Bean methods that intend to refer to the provided resource here).  此外，使用@Bean方法，您通常使用编程JNDI查找，通过使用Spring JndiTemplate或JndiLocatorDelegate帮助程序或直接JNDI InitialContext用法但不使用JndiObjectFactoryBean变量（这将强制您将返回类型声明为FactoryBean类型而不是实际目标类型，使得更难以用于其他@Bean打算在此处引用所提供资源的方法中的交叉引用调用。 |

In the case of BeanOne from the example above the preceding note, it would be equally valid to call the init() method directly during construction, as the following example shows:

在BeanOne前面注释中的示例的情况下，init() 在构造期间直接调用该方法同样有效，如以下示例所示：

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** BeanOne beanOne() {

BeanOne beanOne = **new** BeanOne();

beanOne.init();

**return** beanOne;

}

*// ...*

}

|  |  |
| --- | --- |
|  | When you work directly in Java, you can do anything you like with your objects and do not always need to rely on the container lifecycle.  当您直接使用Java工作时，您可以使用对象执行任何您喜欢的操作，并且不必总是依赖于容器生命周期。 |

Specifying Bean Scope

Spring includes the @Scope annotation so that you can specify the scope of a bean.

Spring包含@Scope注释，以便您可以指定bean的范围。

Using the @Scope Annotation

You can specify that your beans defined with the @Bean annotation should have a specific scope. You can use any of the standard scopes specified in the [Bean Scopes](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-scopes) section.

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/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-scopes)部分中指定的任何标准作用域 。

The default scope is singleton, but you can override this with the @Scope annotation, as the following example shows:

默认范围是singleton，但您可以使用@Scope注释覆盖它，如以下示例所示：

@Configuration

**public** **class** **MyConfiguration** {

@Bean

@Scope("prototype")

**public** Encryptor encryptor() {

*// ...*

}

}

@Scope and scoped-proxy

Spring offers a convenient way of working with scoped dependencies through [scoped proxies](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-other-injection). The easiest way to create such a proxy when using the XML configuration is the <aop:scoped-proxy/> element. Configuring your beans in Java with a @Scopeannotation offers equivalent support with the proxyMode attribute. The default is no proxy (ScopedProxyMode.NO), but you can specify ScopedProxyMode.TARGET\_CLASS or ScopedProxyMode.INTERFACES.

Spring提供了一种通过[作用域代理](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-other-injection)处理作用域依赖项的便捷方法 。使用XML配置时创建此类代理的最简单方法是<aop:scoped-proxy/>元素。使用@Scope注释在Java中配置bean 提供了对该proxyMode属性的等效支持。默认值为no proxy（ScopedProxyMode.NO），但您可以指定ScopedProxyMode.TARGET\_CLASS或ScopedProxyMode.INTERFACES。

If you port the scoped proxy example from the XML reference documentation (see [scoped proxies](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-other-injection)) to our @Bean using Java, it resembles the following:

如果将scoped代理示例从XML参考文档（请参阅[范围代理](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-other-injection)）移植 到@Bean使用Java，它类似于以下内容：

*// an HTTP Session-scoped bean exposed as a proxy*

@Bean

@SessionScope

**public** UserPreferences userPreferences() {

**return** **new** UserPreferences();

}

@Bean

**public** Service userService() {

UserService service = **new** SimpleUserService();

*// a reference to the proxied userPreferences bean*

service.setUserPreferences(userPreferences());

**return** service;

}

Customizing Bean Naming

By default, configuration classes use a @Bean method’s name as the name of the resulting bean. This functionality can be overridden, however, with the name attribute, as the following example shows:

默认情况下，配置类使用@Bean方法的名称作为结果bean的名称。但是，可以使用name属性覆盖此功能，如以下示例所示：

@Configuration

**public** **class** **AppConfig** {

@Bean(name = "myThing")

**public** Thing thing() {

**return** **new** Thing();

}

}

Bean Aliasing

As discussed in [Naming Beans](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-beanname), it is sometimes desirable to give a single bean multiple names, otherwise known as bean aliasing. The name attribute of the @Bean annotation accepts a String array for this purpose. The following example shows how to set a number of aliases for a bean:

正如[Naming Beans中](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-beanname)所讨论的，有时需要为单个bean提供多个名称，也称为bean别名。 为此目的name，@Bean注释的属性接受String数组。以下示例显示如何为bean设置多个别名：

@Configuration

**public** **class** **AppConfig** {

@Bean({"dataSource", "subsystemA-dataSource", "subsystemB-dataSource"})

**public** DataSource dataSource() {

*// instantiate, configure and return DataSource bean...*

}

}

Bean Description

Sometimes, it is helpful to provide a more detailed textual description of a bean. This can be particularly useful when beans are exposed (perhaps through JMX) for monitoring purposes.

有时，提供更详细的bean文本描述会很有帮助。当bean（可能通过JMX）进行监视时，这可能特别有用。

To add a description to a @Bean, you can use the [@Description](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/context/annotation/Description.html) annotation, as the following example shows:

要向a添加描述@Bean，可以使用 [@Description](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/context/annotation/Description.html) 注释，如以下示例所示：

@Configuration

**public** **class** **AppConfig** {

@Bean

@Description("Provides a basic example of a bean")

**public** Thing thing() {

**return** **new** Thing();

}

}

1.12.4. Using the @Configuration annotation

@Configuration is a class-level annotation indicating that an object is a source of bean definitions. @Configuration classes declare beans through public @Bean annotated methods. Calls to @Bean methods on @Configuration classes can also be used to define inter-bean dependencies. See [Basic Concepts: @Bean and @Configuration](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-java-basic-concepts) for a general introduction.

@Configuration是一个类级别的注释，指示对象是bean定义的来源。@Configurationclasses通过公共@Bean注释方法声明bean 。@Bean对@Configuration类上的方法的调用也可用于定义bean间依赖项。请参阅[基本概念：@Bean和@Configuration](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-java-basic-concepts)一般性介绍。

Injecting Inter-bean Dependencies

When beans have dependencies on one another, expressing that dependency is as simple as having one bean method call another, as the following example shows:

当bean彼此依赖时，表达该依赖关系就像让一个bean方法调用另一个bean一样简单，如下例所示：

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** BeanOne beanOne() {

**return** **new** BeanOne(beanTwo());

}

@Bean

**public** BeanTwo beanTwo() {

**return** **new** BeanTwo();

}

}

In the preceding example, beanOne receives a reference to beanTwo through constructor injection.

在前面的示例中，beanOne接收对beanTwo构造函数注入的引用。

|  |  |
| --- | --- |
|  | This method of declaring inter-bean dependencies works only when the @Bean method is declared within a @Configuration class. You cannot declare inter-bean dependencies by using plain @Component classes.  这种声明bean间依赖关系的@Bean方法只有在@Configuration类中声明方法时才有效。您不能使用普通@Component类声明bean间依赖项。 |

Lookup Method Injection

As noted earlier, [lookup method injection](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-method-injection) is an advanced feature that you should use rarely. It is useful in cases where a singleton-scoped bean has a dependency on a prototype-scoped bean. Using Java for this type of configuration provides a natural means for implementing this pattern. The following example shows how to use lookup method injection:

如前所述，[查找方法注入](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-method-injection)是一项很少使用的高级功能。在单例范围的bean依赖于原型范围的bean的情况下，它很有用。将Java用于此类配置提供了实现此模式的自然方法。以下示例显示如何使用查找方法注入：

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

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

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

Command command = createCommand();

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

command.setState(commandState);

**return** command.execute();

}

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

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

}

By using Java configuration, you can create a subclass of CommandManager where the abstract createCommand() method is overridden in such a way that it looks up a new (prototype) command object. The following example shows how to do so:

通过使用Java配置，您可以创建一个子类，CommandManager其中抽象createCommand()方法被覆盖，以便查找新的（原型）命令对象。以下示例显示了如何执行此操作：

@Bean

@Scope("prototype")

**public** AsyncCommand asyncCommand() {

AsyncCommand command = **new** AsyncCommand();

*// inject dependencies here as required*

**return** command;

}

@Bean

**public** CommandManager commandManager() {

*// return new anonymous implementation of CommandManager with createCommand()*

*// 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

Consider the following example, which shows a @Bean annotated method being called twice:

请考虑以下示例，该示例显示了@Bean两次调用的带注释的方法：

@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() has been called once in clientService1() and once in clientService2(). Since this method creates a new instance of ClientDaoImpl and returns it, you would normally expect to have two instances (one for each service). That definitely would be problematic: In Spring, instantiated beans have a singleton scope by default. This is where the magic comes in: All @Configuration classes are subclassed at startup-time with CGLIB. In the subclass, the child method checks the container first for any cached (scoped) beans before it calls the parent method and creates a new instance.

clientDao()被称为一次进入clientService1()和进入一次clientService2()。由于此方法创建了一个新实例ClientDaoImpl并将其返回，因此通常需要两个实例（每个服务一个）。这肯定会有问题：在Spring中，实例化的bean singleton默认具有范围。这就是魔术的用武之地：所有@Configuration类都在启动时被子类化CGLIB。在子类中，子方法在调用父方法并创建新实例之前，首先检查容器是否有任何缓存（作用域）bean。

|  |  |
| --- | --- |
|  | The behavior could be different according to the scope of your bean. We are talking about singletons here.  根据bean的范围，行为可能会有所不同。我们在这里谈论单身人士。 |
|  | As of Spring 3.2, it is no longer necessary to add CGLIB to your classpath because CGLIB classes have been repackaged under org.springframework.cglib and included directly within the spring-core JAR.  从Spring 3.2开始，不再需要将CGLIB添加到类路径中，因为CGLIB类已经重新打包org.springframework.cglib并直接包含在spring-core JAR中。 |

|  |  |
| --- | --- |
|  | There are a few restrictions due to the fact that CGLIB dynamically adds features at startup-time. In particular, configuration classes must not be final. However, as of 4.3, any constructors are allowed on configuration classes, including the use of @Autowired or a single non-default constructor declaration for default injection.  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，配置类允许使用任何构造函数，包括使用 @Autowired默认注入的单个非默认构造函数声明。  If you prefer to avoid any CGLIB-imposed limitations, consider declaring your @Bean methods on non-@Configuration classes (for example, on plain @Component classes instead). Cross-method calls between @Beanmethods are not then intercepted, so you have to exclusively rely on dependency injection at the constructor or method level there.  如果您希望避免任何CGLIB强加的限制，请考虑@Bean 在非@Configuration类上声明您的方法（例如，在普通@Component类上）。@Bean然后拦截方法之间的跨方法调用，因此您必须完全依赖于构造函数或方法级别的依赖注入。 |

1.12.5. Composing Java-based Configurations

Spring’s Java-based configuration feature lets you compose annotations, which can reduce the complexity of your configuration.

Spring的基于Java的配置功能允许您撰写注释，这可以降低配置的复杂性。

Using the @Import Annotation

Much as the <import/> element is used within Spring XML files to aid in modularizing configurations, the @Import annotation allows for loading @Bean definitions from another configuration class, as the following example shows:

就像<import/>在Spring XML文件中使用该元素来帮助模块化配置一样，@Import注释允许@Bean从另一个配置类加载定义，如以下示例所示：

@Configuration

**public** **class** **ConfigA** {

@Bean

**public** A a() {

**return** **new** A();

}

}

@Configuration

@Import(ConfigA.class)

**public** **class** **ConfigB** {

@Bean

**public** B b() {

**return** **new** B();

}

}

Now, rather than needing to specify both ConfigA.class and ConfigB.class when instantiating the context, only ConfigBneeds to be supplied explicitly, as the following example shows:

现在，不需要同时指定ConfigA.class和ConfigB.class实例化上下文，只ConfigB需要显式提供，如下例所示：

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

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(ConfigB.class);

*// now both beans A and B will be available...*

A a = ctx.getBean(A.class);

B b = ctx.getBean(B.class);

}

This approach simplifies container instantiation, as only one class needs to be dealt with, rather than requiring you to remember a potentially large number of @Configuration classes during construction.

这种方法简化了容器实例化，因为只需要处理一个类，而不是要求您@Configuration在构造期间记住可能大量的 类。

|  |  |
| --- | --- |
|  | As of Spring Framework 4.2, @Import also supports references regular component classes, analogous to the AnnotationConfigApplicationContext.register method. This is particularly useful if you want to avoid component scanning, by using a few configuration classes as entry points to explicitly define all your components.  从Spring Framework 4.2开始，@Import还支持引用常规组件类，类似于AnnotationConfigApplicationContext.register方法。如果要避免组件扫描，这一点特别有用，可以使用一些配置类作为明确定义所有组件的入口点。 |

Injecting Dependencies on Imported @Bean Definitions

The preceding example works but is simplistic. In most practical scenarios, beans have dependencies on one another across configuration classes. When using XML, this is not an issue, because no compiler is involved, and you can declareref="someBean" and trust Spring to work it out during container initialization. When using @Configuration classes, the Java compiler places constraints on the configuration model, in that references to other beans must be valid Java syntax.

前面的例子有效，但很简单。在大多数实际情况中，bean跨配置类彼此依赖。使用XML时，这不是问题，因为不涉及编译器，并且您可以声明 ref="someBean"并信任Spring在容器初始化期间解决它。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。

Fortunately, solving this problem is simple. As [we already discussed](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-java-dependencies), a @Bean method can have an arbitrary number of parameters that describe the bean dependencies. Consider the following more real-world scenario with several @Configurationclasses, each depending on beans declared in the others:

幸运的是，解决这个问题很简单。正如[我们已经讨论过的](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-java-dependencies)，一个@Bean方法可以有任意数量的参数来描述bean的依赖关系。考虑以下更多真实场景​​，其中包含几个@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");

}

There is another way to achieve the same result. Remember that @Configuration classes are ultimately only another bean in the container: This means that they can take advantage of @Autowired and @Value injection and other features the same as any other bean.

还有另一种方法可以达到相同的效果。请记住，@Configuration类最终只是容器中的另一个bean：这意味着它们可以利用 @Autowired和@Value注入以及与任何其他bean相同的其他功能。

|  |  |
| --- | --- |
|  | Make sure that the dependencies you inject that way are of the simplest kind only. @Configuration classes are processed quite early during the initialization of the context, and forcing a dependency to be injected this way may lead to unexpected early initialization. Whenever possible, resort to parameter-based injection, as in the preceding example.  确保以这种方式注入的依赖项只是最简单的类型。@Configuration 在上下文初始化期间很早就处理了类，并且强制以这种方式注入依赖项可能会导致意外的早期初始化。尽可能采用基于参数的注入，如前面的示例所示。  Also, be particularly careful with BeanPostProcessor and BeanFactoryPostProcessor definitions through @Bean. Those should usually be declared as static @Bean methods, not triggering the instantiation of their containing configuration class. Otherwise, @Autowired and @Value do not work on the configuration class itself, since it is being created as a bean instance too early.  另外，要特别注意BeanPostProcessor和BeanFactoryPostProcessor定义@Bean。这些通常应该声明为static @Bean方法，而不是触发其包含配置类的实例化。否则，@Autowired而@Value不要在配置类本身的工作，因为它是被作为一个bean实例创建为时尚早。 |

The following example shows how one bean can be autowired to another bean:

以下示例显示了如何将一个bean自动连接到另一个bean：

@Configuration

**public** **class** **ServiceConfig** {

@Autowired

**private** AccountRepository accountRepository;

@Bean

**public** TransferService transferService() {

**return** **new** TransferServiceImpl(accountRepository);

}

}

@Configuration

**public** **class** **RepositoryConfig** {

**private** **final** DataSource dataSource;

@Autowired

**public** RepositoryConfig(DataSource dataSource) {

this.dataSource = dataSource;

}

@Bean

**public** AccountRepository accountRepository() {

**return** **new** JdbcAccountRepository(dataSource);

}

}

@Configuration

@Import({ServiceConfig.class, RepositoryConfig.class})

**public** **class** **SystemTestConfig** {

@Bean

**public** DataSource dataSource() {

*// return new DataSource*

}

}

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

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(SystemTestConfig.class);

*// everything wires up across configuration classes...*

TransferService transferService = ctx.getBean(TransferService.class);

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

}

|  |  |
| --- | --- |
|  | Constructor injection in @Configuration classes is only supported as of Spring Framework 4.3. Note also that there is no need to specify @Autowired if the target bean defines only one constructor. In the preceding example, @Autowired is not necessary on the RepositoryConfig constructor.  @Configuration仅在Spring Framework 4.3中支持类中的 构造函数注入。另请注意，无需指定@Autowired目标bean是否仅定义了一个构造函数。在前面的示例中，构造函数@Autowired上没有必要RepositoryConfig。 |

*Fully-qualifying imported beans for ease of navigation*

In the preceding scenario, using @Autowired works well and provides the desired modularity, but determining exactly where the autowired bean definitions are declared is still somewhat ambiguous. For example, as a developer looking at ServiceConfig, how do you know exactly where the @Autowired AccountRepository bean is declared? It is not explicit in the code, and this may be just fine. Remember that the [Spring Tool Suite](https://spring.io/tools/sts) provides tooling that can render graphs showing how everything is wired, which may be all you need. Also, your Java IDE can easily find all declarations and uses of the AccountRepository type and quickly show you the location of @Bean methods that return that type.

在前面的场景中，使用@Autowired效果很好并提供了所需的模块性，但确定声明自动装配的bean定义的确切位置仍然有些模棱两可。例如，作为开发人员ServiceConfig，您如何确切地知道@Autowired AccountRepositorybean的声明位置？它在代码中并不明确，这可能就好了。请记住， [Spring Tool Suite](https://spring.io/tools/sts)提供的工具可以呈现图形，显示所有内容的连线方式，这可能就是您所需要的。此外，您的Java IDE可以轻松找到该AccountRepository类型的所有声明和用法，并快速显示@Bean返回该类型的方法的位置。

In cases where this ambiguity is not acceptable and you wish to have direct navigation from within your IDE from one @Configuration class to another, consider autowiring the configuration classes themselves. The following example shows how to do so:

如果这种歧义是不可接受的，并且您希望从IDE中直接从一个@Configuration类导航到另一个类，请考虑自行装配配置类本身。以下示例显示了如何执行此操作：

@Configuration

**public** **class** **ServiceConfig** {

@Autowired

**private** RepositoryConfig repositoryConfig;

@Bean

**public** TransferService transferService() {

*// navigate 'through' the config class to the @Bean method!*

**return** **new** TransferServiceImpl(repositoryConfig.accountRepository());

}

}

In the preceding situation, where AccountRepository is defined is completely explicit. However, ServiceConfig is now tightly coupled to RepositoryConfig. That is the tradeoff. This tight coupling can be somewhat mitigated by using interface-based or abstract class-based @Configuration classes. Consider the following example:

在前面的情况中，AccountRepository定义的位置是完全明确的。但是，ServiceConfig现在紧紧联系在一起RepositoryConfig。这是权衡。通过使用基于接口的或基于@Configuration类的抽象类，可以在某种程度上减轻这种紧密耦合。请考虑以下示例：

@Configuration

**public** **class** **ServiceConfig** {

@Autowired

**private** RepositoryConfig repositoryConfig;

@Bean

**public** TransferService transferService() {

**return** **new** TransferServiceImpl(repositoryConfig.accountRepository());

}

}

@Configuration

**public** **interface** **RepositoryConfig** {

@Bean

AccountRepository accountRepository();

}

@Configuration

**public** **class** **DefaultRepositoryConfig** **implements** RepositoryConfig {

@Bean

**public** AccountRepository accountRepository() {

**return** **new** JdbcAccountRepository(...);

}

}

@Configuration

@Import({ServiceConfig.class, DefaultRepositoryConfig.class}) *// import the concrete config!*

**public** **class** **SystemTestConfig** {

@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");

}

Now ServiceConfig is loosely coupled with respect to the concrete DefaultRepositoryConfig, and built-in IDE tooling is still useful: You can easily get a type hierarchy of RepositoryConfig implementations. In this way, navigating @Configurationclasses and their dependencies becomes no different than the usual process of navigating interface-based code.

现在ServiceConfig与具体的松散耦合 DefaultRepositoryConfig，内置的IDE工具仍然有用：您可以轻松获得实现的类型层次结构RepositoryConfig。通过这种方式，导航@Configuration类及其依赖关系与导航基于接口的代码的常规过程没有什么不同。

|  |  |
| --- | --- |
|  | If you want to influence the startup creation order of certain beans, consider declaring some of them as @Lazy(for creation on first access instead of on startup) or as @DependsOn certain other beans (making sure that specific other beans are created before the current bean, beyond what the latter’s direct dependencies imply).  如果要影响某些bean的启动创建顺序，可以考虑将它们中的一些声明为@Lazy（用于在第一次访问时创建而不是在启动时）或@DependsOn某些其他bean（确保在当前bean之前创建特定的其他bean，超出后者的直接依赖意味着什么）。 |

Conditionally Include @Configuration Classes or @Bean Methods

It is often useful to conditionally enable or disable a complete @Configuration class or even individual @Bean methods, based on some arbitrary system state. One common example of this is to use the @Profile annotation to activate beans only when a specific profile has been enabled in the Spring Environment (see [Bean Definition Profiles](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-definition-profiles) for details).

基于某些任意系统状态，有条件地启用或禁用完整@Configuration类或甚至单个@Bean方法通常很有用。一个常见的例子是@Profile只有在Spring中启用了特定的配置文件时才使用注释激活bean Environment（ 有关详细[信息](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-definition-profiles)，请参阅[Bean定义配置文件](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-definition-profiles)）。

The @Profile annotation is actually implemented by using a much more flexible annotation called [@Conditional](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/context/annotation/Conditional.html). The @Conditional annotation indicates specific org.springframework.context.annotation.Condition implementations that should be consulted before a @Bean is registered.

该@Profile注释是通过使用一种称为更灵活的注释实际执行[@Conditional](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/context/annotation/Conditional.html)。该@Conditional注释指示特定org.springframework.context.annotation.Condition前应谘询的实施@Bean是注册。

Implementations of the Condition interface provide a matches(…​) method that returns true or false. For example, the following listing shows the actual Condition implementation used for @Profile:

Condition接口的实现提供了一个matches(…​) 返回true或的方法false。例如，以下清单显示了Condition用于的实际 实现@Profile：

@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;

}

See the [@Conditional](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/context/annotation/Conditional.html) javadoc for more detail.

有关[@Conditional](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/context/annotation/Conditional.html) 更多详细信息，请参阅javadoc。

Combining Java and XML Configuration

Spring’s @Configuration class support does not aim to be a 100% complete replacement for Spring XML. Some facilities, such as Spring XML namespaces, remain an ideal way to configure the container. In cases where XML is convenient or necessary, you have a choice: either instantiate the container in an “XML-centric” way by using, for example, ClassPathXmlApplicationContext, or instantiate it in a “Java-centric” way by using AnnotationConfigApplicationContext and the @ImportResource annotation to import XML as needed.

Spring的@Configuration类支持并非旨在成为Spring XML的100％完全替代品。某些工具（如Spring XML命名空间）仍然是配置容器的理想方法。在XML方便或必要的情况下，您可以选择：例如，通过使用“以XML为中心”的方式实例化容器ClassPathXmlApplicationContext，或者通过使用AnnotationConfigApplicationContext和@ImportResource注释以“以Java为中心”的方式实例化它。 根据需要导入XML。

XML-centric Use of @Configuration Classes

It may be preferable to bootstrap the Spring container from XML and include @Configuration classes in an ad-hoc fashion. For example, in a large existing codebase that uses Spring XML, it is easier to create @Configuration classes on an as-needed basis and include them from the existing XML files. Later in this section, we cover the options for using @Configuration classes in this kind of “XML-centric” situation.

最好从XML引导Spring容器并@Configuration以ad-hoc方式包含 类。例如，在使用Spring XML的大型现有代码库中，可以@Configuration根据需要更轻松地创建类，并将其包含在现有XML文件中。在本节的后面部分，我们将介绍@Configuration在这种“以XML为中心”的情况下使用类的选项。

*Declaring @Configuration classes as plain Spring <bean/> elements*

Remember that @Configuration classes are ultimately bean definitions in the container. In this series examples, we create a @Configuration class named AppConfig and include it within system-test-config.xml as a <bean/> definition. Because<context:annotation-config/> is switched on, the container recognizes the @Configuration annotation and processes the @Bean methods declared in AppConfig properly.

请记住，@Configuration类最终是容器中的bean定义。在本系列示例中，我们创建了一个@Configuration名为的类，AppConfig并将其system-test-config.xml作为<bean/>定义包含在其中。因为 <context:annotation-config/>已打开，容器会识别@Configuration注释并 正确处理@Bean声明的方法AppConfig。

The following example shows an ordinary configuration class in Java:

以下示例显示了Java中的普通配置类：

@Configuration

**public** **class** **AppConfig** {

@Autowired

**private** DataSource dataSource;

@Bean

**public** AccountRepository accountRepository() {

**return** **new** JdbcAccountRepository(dataSource);

}

@Bean

**public** TransferService transferService() {

**return** **new** TransferService(accountRepository());

}

}

The following example shows part of a sample system-test-config.xml file:

以下示例显示了示例system-test-config.xml文件的一部分：

<beans>

*<!-- enable processing of annotations such as @Autowired and @Configuration -->*

<context:annotation-config/>

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

<bean class="com.acme.AppConfig"/>

<bean class="org.springframework.jdbc.datasource.DriverManagerDataSource">

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

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

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

</bean>

</beans>

The following example shows a possible jdbc.properties file:

以下示例显示了一个可能的jdbc.properties文件：

jdbc.url=jdbc:hsqldb:hsql://localhost/xdb

jdbc.username=sa

jdbc.password=

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

ApplicationContext ctx = **new** ClassPathXmlApplicationContext("classpath:/com/acme/system-test-config.xml");

TransferService transferService = ctx.getBean(TransferService.class);

*// ...*

}

|  |  |
| --- | --- |
|  | In system-test-config.xml file, the AppConfig <bean/> does not declare an id element. While it would be acceptable to do so, it is unnecessary, given that no other bean ever refers to it, and it is unlikely to be explicitly fetched from the container by name. Similarly, the DataSource bean is only ever autowired by type, so an explicit bean id is not strictly required.  在system-test-config.xml文件中，AppConfig <bean/>不声明id 元素。虽然这样做是可以接受的，但是没有必要，因为没有其他bean引用它，并且不太可能通过名称从容器中明确地获取它。类似地，DataSourcebean只是按类型自动装配，因此id 不严格要求显式bean 。 |

*Using <context:component-scan/> to pick up @Configuration classes*

Because @Configuration is meta-annotated with @Component, @Configuration-annotated classes are automatically candidates for component scanning. Using the same scenario as describe in the previous example, we can redefine system-test-config.xml to take advantage of component-scanning. Note that, in this case, we need not explicitly declare<context:annotation-config/>, because <context:component-scan/> enables the same functionality.

因为@Configuration带有元注释@Component，注释@Configuration类自动成为组件扫描的候选者。使用与前一个示例中描述的相同的方案，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/>启用相同的功能。

The following example shows the modified system-test-config.xml file:

以下示例显示了已修改的system-test-config.xml文件：

<beans>

*<!-- picks up and registers AppConfig as a bean definition -->*

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

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

<bean class="org.springframework.jdbc.datasource.DriverManagerDataSource">

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

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

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

</bean>

</beans>

@Configuration Class-centric Use of XML with @ImportResource

In applications where @Configuration classes are the primary mechanism for configuring the container, it is still likely necessary to use at least some XML. In these scenarios, you can use @ImportResource and define only as much XML as you need. Doing so achieves a “Java-centric” approach to configuring the container and keeps XML to a bare minimum. The following example (which includes a configuration class, an XML file that defines a bean, a properties file, and the main class) shows how to use the @ImportResource annotation to achieve “Java-centric” configuration that uses XML as needed:

在@Configuration类是配置容器的主要机制的应用程序中，仍然可能需要使用至少一些XML。在这些场景中，您可以@ImportResource根据需要使用和定义尽可能多的XML。这样做可以实现“以Java为中心”的方法来配置容器并将XML保持在最低限度。以下示例（包括配置类，定义bean的XML文件，属性文件和main类）显示了如何使用@ImportResource注释来实现根据需要使用XML的“以Java为中心”的配置：

@Configuration

@ImportResource("classpath:/com/acme/properties-config.xml")

**public** **class** **AppConfig** {

@Value("${jdbc.url}")

**private** String url;

@Value("${jdbc.username}")

**private** String username;

@Value("${jdbc.password}")

**private** String password;

@Bean

**public** DataSource dataSource() {

**return** **new** DriverManagerDataSource(url, username, password);

}

}

properties-config.xml

<beans>

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

</beans>

jdbc.properties

jdbc.url=jdbc:hsqldb: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

The [Environment](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/core/env/Environment.html) interface is an abstraction integrated in the container that models two key aspects of the application environment: [profiles](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-definition-profiles) and [properties](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-property-source-abstraction).

该[Environment](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/core/env/Environment.html)接口是集成在容器模型应用环境的两个关键方面的抽象：[型材](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-definition-profiles) 和[性能](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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 with 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.

配置文件是仅在给定配置文件处于活动状态时才向容器注册的bean定义的命名逻辑组。可以将Bean分配给配置文件，无论是以XML还是使用注释定义。Environment与配置文件相关的对象的作用是确定哪些配置文件（如果有）当前处于活动状态，以及默认情况下哪些配置文件（如果有）应处于活动状态。

Properties 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, Map objects, 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.

属性在几乎所有应用程序中都发挥着重要作用，可能源自各种来源：属性文件，JVM系统属性，系统环境变量，JNDI，servlet上下文参数，ad-hoc Properties对象，Map对象等。Environment与属性相关的对象的作用是为用户提供方便的服务接口，用于配置属性源并从中解析属性。

1.13.1. Bean Definition Profiles

Bean definition profiles provide 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:

Bean定义配置文件在核心容器中提供了一种机制，允许在不同环境中注册不同的bean。“环境”这个词对不同的用户来说意味着不同的东西，这个功能可以帮助许多用例，包括：

* Working against an in-memory datasource in development versus 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 versus customer B deployments.
* 在QA或生产环境中，针对开发中的内存数据源而不是从JNDI查找相同的数据源。
* 仅在将应用程序部署到性能环境时注册监视基础结构。
* 为客户A和客户B部署注册bean的自定义实施。

Consider the first use case in a practical application that requires a DataSource. In a test environment, the configuration might resemble the following:

考虑实际应用中需要的第一个用例 DataSource。在测试环境中，配置可能类似于以下内容：

@Bean

**public** DataSource dataSource() {

**return** **new** EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.HSQL)

.addScript("my-schema.sql")

.addScript("my-test-data.sql")

.build();

}

Now consider how this application can be deployed into a QA or production environment, assuming that the datasource for the application is registered with the production application server’s JNDI directory. Our dataSource bean now looks like the following listing:

现在考虑如何将此应用程序部署到QA或生产环境中，假设应用程序的数据源已在生产应用程序服务器的JNDI目录中注册。我们的dataSourcebean现在看起来如下：

@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.

问题是如何根据当前环境在使用这两种变体之间切换。随着时间的推移，Spring用户已经设计了许多方法来完成这项工作，通常依赖于系统环境变量和<import/>包含${placeholder}令牌的XML 语句的组合，这些令牌根据环境变量的值解析为正确的配置文件路径。Bean定义配置文件是核心容器功能，可为此问题提供解决方案。

If we generalize the use case shown in the preceding example of environment-specific bean definitions, we end up with the need to register certain bean definitions in certain contexts but 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. We start by updating our configuration to reflect this need.

如果我们概括了前面的特定于环境的bean定义示例中显示的用例，我们最终需要在某些上下文中注册某些bean定义，而在其他上下文中则不需要。您可以说您希望在情境A中注册特定的bean定义配置文件，在情况B中注册不同的配置文件。我们首先更新配置以反映此需求。

Using @Profile

The [@Profile](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/context/annotation/Profile.html) annotation lets you indicate that a component is eligible for registration when one or more specified profiles are active. Using our preceding example, we can rewrite the dataSource configuration as follows:

通过[@Profile](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/context/annotation/Profile.html) 注释，您可以指示当一个或多个指定的配置文件处于活动状态时，组件符合注册条件。使用前面的示例，我们可以dataSource按如下方式重写配置：

@Configuration

@Profile("development")

**public** **class** **StandaloneDataConfig** {

@Bean

**public** DataSource dataSource() {

**return** **new** EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.HSQL)

.addScript("classpath:com/bank/config/sql/schema.sql")

.addScript("classpath:com/bank/config/sql/test-data.sql")

.build();

}

}

@Configuration

@Profile("production")

**public** **class** **JndiDataConfig** {

@Bean(destroyMethod="")

**public** DataSource dataSource() **throws** Exception {

Context ctx = **new** InitialContext();

**return** (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");

}

}

|  |  |
| --- | --- |
|  | As mentioned earlier, with @Bean methods, you typically choose to use programmatic JNDI lookups, by using either Spring’s JndiTemplate/JndiLocatorDelegate helpers or the straight JNDI InitialContext usage shown earlier but not the JndiObjectFactoryBean variant, which would force you to declare the return type as the FactoryBean type.  如前所述，对于@Bean方法，您通常选择使用编程JNDI查找，使用Spring的JndiTemplate/JndiLocatorDelegatehelper或InitialContext前面显示的直接JNDI 用法，但不使用JndiObjectFactoryBean 变量，这会强制您将返回类型声明为FactoryBean类型。 |

The profile string may contain a simple profile name (for example, production) or a profile expression. A profile expression allows for more complicated profile logic to be expressed (for example, production & us-east). The following operators are supported in profile expressions:

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

* !: A logical “not” of the profile
* &: A logical “and” of the profiles
* |: A logical “or” of the profiles
* !：配置文件的逻辑“不”
* &：配置文件的逻辑“和”
* |：配置文件的逻辑“或”

|  |  |
| --- | --- |
|  | You cannot mix the & and | operators without using parentheses. For example, production & us-east | eu-central is not a valid expression. It must be expressed as production & (us-east | eu-central).  不使用括号， 不能混合使用&和|运算符。例如， production & us-east | eu-central不是有效的表达式。它必须表达为 production & (us-east | eu-central)。 |

You can use @Profile as a [meta-annotation](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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 you can use as a drop-in replacement for @Profile("production"):

您可以将其@Profile用作[元注释](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-meta-annotations)，以创建自定义组合注释。以下示例定义了一个自定义 @Production注释，您可以将其用作以下内容的替代品 @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 are bypassed unless one or more of the specified profiles are active. If a @Componentor @Configuration class is marked with @Profile({"p1", "p2"}), that class is not registered or processed unless profiles 'p1' or 'p2' have been activated. If a given profile is prefixed with the NOT operator (!), the annotated element is registered only 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.  如果@Configuration标记了类，则除非一个或多个指定的配置文件处于活动状态，否则将绕过与该类关联的@Profile所有@Bean方法和 @Import注释。如果a @Component或@Configurationclass被标记@Profile({"p1", "p2"})，则除非已激活配置文件'p1'或'p2'，否则不会注册或处理该类。如果给定的配置文件以NOT运算符（!）作为前缀，则仅在配置文件未激活时才注册带注释的元素。例如，@Profile({"p1", "!p2"})如果配置文件“p1”处于活动状态或配置文件“p2”未激活，则会发生注册。 |

@Profile can also be declared at the method level to include only one particular bean of a configuration class (for example, for alternative variants of a particular bean), as the following example shows:

@Profile 也可以在方法级别声明只包含配置类的一个特定bean（例如，对于特定bean的替代变体），如以下示例所示：

@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");

}

}

|  |  |
| --- | --- |
|  | The standaloneDataSource method is available only in the development profile.  该standaloneDataSource方法仅在development配置文件中可用。 |
|  | The jndiDataSource method is available only in the production profile.  该jndiDataSource方法仅在production配置文件中可用。 |
|  | 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), a @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 matters. Therefore, @Profile can 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.  使用@Profileon @Bean方法，可能会应用特殊方案：对于@Bean相同Java方法名称的重载方法（类似于构造函数重载），@Profile需要在所有重载方法上一致地声明条件。如果条件不一致，则只有重载方法中第一个声明的条件才重要。因此，@Profile不能用于选择具有特定参数签名的重载方法。在创建时，Spring的构造函数解析算法遵循同一bean的所有工厂方法之间的分辨率。  If you want to define alternative beans with different profile conditions, use distinct Java method names that point to the same bean name by using the @Bean name attribute, as shown in the preceding example. If the argument signatures are all the same (for example, 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).  如果要定义具有不同配置文件条件的备用Bean，请使用通过使用@Beanname属性指向相同bean名称的不同Java方法名称，如上例所示。如果参数签名都是相同的（例如，所有变体都具有no-arg工厂方法），这是首先在有效的Java类中表示这种排列的唯一方法（因为只有一个特定名称和参数签名的方法）。 |

XML Bean Definition Profiles

The XML counterpart is the profile attribute of the <beans> element. Our preceding sample configuration can be rewritten in two XML files, as follows:

XML对应物是元素的profile属性<beans>。我们之前的示例配置可以在两个XML文件中重写，如下所示：

<beans profile="development"

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

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

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

xsi:schemaLocation="...">

<jdbc:embedded-database id="dataSource">

<jdbc:script location="classpath:com/bank/config/sql/schema.sql"/>

<jdbc:script location="classpath:com/bank/config/sql/test-data.sql"/>

</jdbc:embedded-database>

</beans>

<beans profile="production"

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

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

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

xsi:schemaLocation="...">

<jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>

</beans>

It is also possible to avoid that split and nest <beans/> elements within the same file, as the following example shows:

也可以避免<beans/>在同一文件中使用split和nest 元素，如下例所示：

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

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

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

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

xsi:schemaLocation="...">

*<!-- other bean definitions -->*

<beans profile="development">

<jdbc:embedded-database id="dataSource">

<jdbc:script location="classpath:com/bank/config/sql/schema.sql"/>

<jdbc:script location="classpath:com/bank/config/sql/test-data.sql"/>

</jdbc:embedded-database>

</beans>

<beans profile="production">

<jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>

</beans>

</beans>

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.

在spring-bean.xsd受到了制约，使这些元素只能作为文件中的最后一个人。这应该有助于提供灵活性，而不会在XML文件中引起混乱。

|  |  |
| --- | --- |
|  | The XML counterpart does not support the profile expressions described earlier. It is possible, however, to negate a profile by using the ! operator. It is also possible to apply a logical “and” by nesting the profiles, as the following example shows:  XML副本不支持前面描述的配置文件表达式。但是，可以通过使用!运算符来否定轮廓。也可以通过嵌套配置文件来应用逻辑“和”，如以下示例所示：  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:jdbc="http://www.springframework.org/schema/jdbc"  xmlns:jee="http://www.springframework.org/schema/jee"  xsi:schemaLocation="...">  *<!-- other bean definitions -->*  <beans profile="production">  <beans profile="us-east">  <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>  </beans>  </beans>  </beans>  In the preceding example, the dataSource bean is exposed if both the production and us-east profiles are active.  在前面的示例中，dataSource如果两个production和 us-east配置文件都处于活动状态，则会公开Bean 。 |

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.

现在我们已经更新了配置，我们仍然需要指示Spring哪个配置文件处于活动状态。如果我们现在开始我们的示例应用程序，我们会看到NoSuchBeanDefinitionException抛出，因为容器找不到名为的Spring bean 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 through an ApplicationContext. The following example shows how to do so:

激活配置文件可以通过多种方式完成，但最直接的方法是以编程方式对Environment可通过API提供的API进行操作ApplicationContext。以下示例显示了如何执行此操作：

AnnotationConfigApplicationContext ctx = **new** AnnotationConfigApplicationContext();

ctx.getEnvironment().setActiveProfiles("development");

ctx.register(SomeConfig.class, StandaloneDataConfig.class, JndiDataConfig.class);

ctx.refresh();

In addition, you can also declaratively activate profiles 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/5.1.8.RELEASE/spring-framework-reference/core.html#beans-property-source-abstraction)). In integration tests, active profiles can be declared by using the @ActiveProfilesannotation in the spring-test module (see [context configuration with environment profiles](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/testing.html#testcontext-ctx-management-env-profiles)).

此外，您还可以通过spring.profiles.active属性声明性地激活配置文件，该 属性可以通过系统环境变量，JVM系统属性，servlet上下文参数web.xml或甚至作为JNDI中的条目来指定（请参阅[PropertySource抽象](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-property-source-abstraction)）。在集成测试中，可以使用 模块中的@ActiveProfiles注释声明活动配置文件spring-test（请参阅[具有环境配置文件的上下文配置](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/testing.html#testcontext-ctx-management-env-profiles)）。

Note that profiles are not an “either-or” proposition. You can activate multiple profiles at once. Programmatically, you can provide multiple profile names to the setActiveProfiles() method, which accepts String…​ varargs. The following example activates multiple profiles:

请注意，配置文件不是“任何 - 或”命题。您可以一次激活多个配置文件。以编程方式，您可以为setActiveProfiles()方法提供多个配置文件名称，该 方法接受String…​varargs。以下示例激活多个配置文件：

ctx.getEnvironment().setActiveProfiles("profile1", "profile2");

Declaratively, spring.profiles.active may accept a comma-separated list of profile names, as the following example shows:

声明性地，spring.profiles.active可以接受以逗号分隔的配置文件名称列表，如以下示例所示：

-Dspring.profiles.active="profile1,profile2"

Default Profile

The default profile represents the profile that is enabled by default. Consider the following example:

默认配置文件表示默认启用的配置文件。请考虑以下示例：

@Configuration

@Profile("default")

**public** **class** **DefaultDataConfig** {

@Bean

**public** DataSource dataSource() {

**return** **new** EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.HSQL)

.addScript("classpath:com/bank/config/sql/schema.sql")

.build();

}

}

If no profile is active, the dataSource is created. You can see this as a way to provide a default definition for one or more beans. If any profile is enabled, the default profile does not apply.

如果没有激活配置文件，dataSource则创建该配置文件。您可以将此视为一种为一个或多个bean提供默认定义的方法。如果启用了任何配置文件，则默认配置文件不适用。

You can change the name of the default profile by using setDefaultProfiles() on the Environment or ,declaratively, by using the spring.profiles.default property.

您可以通过更改默认的配置文件的名称setDefaultProfiles()上Environment，或者声明，通过使用spring.profiles.default属性。

1.13.2. PropertySource Abstraction

Spring’s Environment abstraction provides search operations over a configurable hierarchy of property sources. Consider the following listing:

Spring的Environment抽象提供了对可配置的属性源层次结构的搜索操作。请考虑以下列表：

ApplicationContext ctx = **new** GenericApplicationContext();

Environment env = ctx.getEnvironment();

**boolean** containsMyProperty = env.containsProperty("my-property");

System.out.println("Does my environment contain the 'my-property' property? " + containsMyProperty);

In the preceding snippet, we see a high-level way of asking Spring whether the my-property 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/5.1.8.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/5.1.8.RELEASE/javadoc-api/org/springframework/core/env/StandardEnvironment.html) is configured with two PropertySource objects — one representing the set of JVM system properties (System.getProperties()) and one representing the set of system environment variables (System.getenv()).

在前面的代码片段中，我们看到了一种向Spring询问是否my-property为当前环境定义属性的高级方法。要回答此问题，Environment对象将对一组对象执行搜索[PropertySource](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/core/env/PropertySource.html) 。A PropertySource是对任何键值对源的简单抽象，Spring [StandardEnvironment](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/core/env/StandardEnvironment.html) 配置有两个PropertySource对象 - 一个表示JVM系统属性集（System.getProperties()），另一个表示系统环境变量集（System.getenv()）。

|  |  |
| --- | --- |
|  | These default property sources are present for StandardEnvironment, for use in standalone applications. [StandardServletEnvironment](https://docs.spring.io/spring-framework/docs/5.1.8.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/5.1.8.RELEASE/javadoc-api/org/springframework/jndi/JndiPropertySource.html). See the javadoc for details.  这些默认属性源StandardEnvironment适用于独立应用程序。[StandardServletEnvironment](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/web/context/support/StandardServletEnvironment.html) 填充了其他默认属性源，包括servlet配置和servlet上下文参数。它可以选择启用a [JndiPropertySource](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/jndi/JndiPropertySource.html)。有关详细信息，请参阅javadoc。 |

Concretely, when you use the StandardEnvironment, the call to env.containsProperty("my-property") returns true if a my-property system property or my-property environment variable is present at runtime.

具体来说，当您使用时StandardEnvironment，env.containsProperty("my-property") 如果运行时存在my-property系统属性或my-property环境变量，则调用将返回true 。

|  |  |
| --- | --- |
|  | The search performed is hierarchical. By default, system properties have precedence over environment variables. So, if the my-property property happens to be set in both places during a call to env.getProperty("my-property"), the system property value “wins” and is returned. Note that property values are not merged but rather completely overridden by a preceding entry.  执行的搜索是分层的。默认情况下，系统属性优先于环境变量。因此，如果my-property在调用期间恰好在两个位置都设置了属性env.getProperty("my-property")，则系统属性值“wins”并返回。请注意，属性值不会合并，而是由前面的条目完全覆盖。  For a common StandardServletEnvironment, the full hierarchy is as follows, with the highest-precedence entries at the top:  对于公共StandardServletEnvironment层次结构，完整层次结构如下，最高优先级条目位于顶部：   1. ServletConfig parameters (if applicable — for example, in case of a DispatcherServlet context) 2. ServletContext parameters (web.xml context-param entries) 3. JNDI environment variables (java:comp/env/ entries) 4. JVM system properties (-D command-line arguments) 5. JVM system environment (operating system environment variables) 6. ServletConfig参数（如果适用 - 例如，在DispatcherServlet上下文的情况下） 7. ServletContext参数（web.xml context-param条目） 8. JNDI环境变量（java:comp/env/条目） 9. JVM系统属性（-D命令行参数） 10. JVM系统环境（操作系统环境变量） |

Most importantly, the entire mechanism is configurable. Perhaps you have a custom source of properties that you want to integrate into this search. To do so, implement and instantiate your own PropertySource and add it to the set of PropertySources for the current Environment. The following example shows how to do so:

最重要的是，整个机制是可配置的。您可能希望将自定义的属性源集成到此搜索中。为此，请实现并实例化您自己的PropertySource并将其添加到PropertySources当前的集合中Environment。以下示例显示了如何执行此操作：

ConfigurableApplicationContext ctx = **new** GenericApplicationContext();

MutablePropertySources sources = ctx.getEnvironment().getPropertySources();

sources.addFirst(**new** MyPropertySource());

In the preceding code, MyPropertySource has been added with highest precedence in the search. If it contains a my-propertyproperty, the property is detected and returned, in favor of any my-property property in any other PropertySource. The[MutablePropertySources](https://docs.spring.io/spring-framework/docs/5.1.8.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.

在上面的代码中，MyPropertySource在搜索中添加了最高优先级。如果它包含my-property属性，则检测并返回该属性，以支持my-property任何其他属性PropertySource。所述 [MutablePropertySources](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/core/env/MutablePropertySources.html) API公开了大量的，其允许该组的属性源的精确操作方法。

1.13.3. Using @PropertySource

The [@PropertySource](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/context/annotation/PropertySource.html) annotation provides a convenient and declarative mechanism for adding a PropertySource to Spring’s Environment.

该[@PropertySource](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/context/annotation/PropertySource.html) 注解提供便利和声明的机制添加PropertySource 到Spring的Environment。

Given a file called app.properties that contains the key-value pair testbean.name=myTestBean, the following @Configurationclass uses @PropertySource in such a way that a call to testBean.getName() returns myTestBean:

给定一个名为app.properties包含键值对的文件testbean.name=myTestBean，以下@Configuration类使用以下@PropertySource方式调用testBean.getName()返回myTestBean：

@Configuration

@PropertySource("classpath:/com/myco/app.properties")

**public** **class** **AppConfig** {

@Autowired

Environment env;

@Bean

**public** TestBean testBean() {

TestBean testBean = **new** TestBean();

testBean.setName(env.getProperty("testbean.name"));

**return** testBean;

}

}

Any ${…​} placeholders present in a @PropertySource resource location are resolved against the set of property sources already registered against the environment, as the following example shows:

资源位置中${…​}存在的任何占位符@PropertySource都是针对已针对环境注册的属性源集合进行解析的，如以下示例所示：

@Configuration

@PropertySource("classpath:/com/${my.placeholder:default/path}/app.properties")

**public** **class** **AppConfig** {

@Autowired

Environment env;

@Bean

**public** TestBean testBean() {

TestBean testBean = **new** TestBean();

testBean.setName(env.getProperty("testbean.name"));

**return** testBean;

}

}

Assuming that my.placeholder is present in one of the property sources already registered (for example, system properties or environment variables), the placeholder is resolved to the corresponding value. If not, then default/path is used as a default. If no default is specified and a property cannot be resolved, an IllegalArgumentException is thrown.

假设它my.placeholder已存在于已注册的其中一个属性源中（例如，系统属性或环境变量），则占位符将解析为相应的值。如果没有，则default/path用作默认值。如果未指定默认值且无法解析属性， IllegalArgumentException则抛出a。

|  |  |
| --- | --- |
|  | The @PropertySource annotation is repeatable, according to Java 8 conventions. However, all such @PropertySource annotations need to be declared at the same level, either directly on the configuration class or as meta-annotations within the same custom annotation. Mixing direct annotations and meta-annotations is not recommended, since direct annotations effectively override meta-annotations.  该@PropertySource注释是可重复的，根据Java的8约定。但是，所有这些@PropertySource注释都需要在同一级别声明，可以直接在配置类上声明，也可以在同一自定义注释中作为元注释声明。不建议混合直接注释和元注释，因为直接注释有效地覆盖了元注释。 |

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. This is no longer the case. Because the Environment abstraction is integrated throughout the container, it is easy to route resolution of placeholders through it. This means that you may configure the resolution process in any way you like. You can change the precedence of searching through system properties and environment variables or remove them entirely. You can also add your own property sources to the mix, as appropriate.

从历史上看，元素中占位符的值只能针对JVM系统属性或环境变量进行解析。这已不再是这种情况。因为Environment抽象集成在整个容器中，所以很容易通过它来解决占位符的分辨率。这意味着您可以以任何您喜欢的方式配置解析过程。您可以更改搜索系统属性和环境变量的优先级，或完全删除它们。您也可以根据需要将自己的属性源添加到混合中。

Concretely, the following statement works regardless of where the customer property is defined, as long as it is available in the Environment:

具体而言，以下语句无论customer 属性的定义位置如何都可以，只要它在以下位置可用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).

在LoadTimeWeaver用于由Spring动态变换的类，因为它们被装载到Java虚拟机（JVM）。

To enable load-time weaving, you can add the @EnableLoadTimeWeaving to one of your @Configuration classes, as the following example shows:

要启用加载时编织，可以将其添加@EnableLoadTimeWeaving到其中一个 @Configuration类中，如以下示例所示：

@Configuration

@EnableLoadTimeWeaving

**public** **class** **AppConfig** {

}

Alternatively, for XML configuration, you can use the context:load-time-weaver element:

或者，对于XML配置，您可以使用以下context:load-time-weaver元素：

<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/5.1.8.RELEASE/spring-framework-reference/data-access.html#orm-jpa) where load-time weaving may be necessary for JPA class transformation. Consult the[LocalContainerEntityManagerFactoryBean](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/orm/jpa/LocalContainerEntityManagerFactoryBean.html) javadoc 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/5.1.8.RELEASE/spring-framework-reference/core.html#aop-aj-ltw).

一旦为其中的ApplicationContext任何bean 配置，就ApplicationContext 可以实现LoadTimeWeaverAware，从而接收对加载时weaver实例的引用。这与[Spring的JPA支持](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/data-access.html#orm-jpa)结合使用特别有用， 其中JPA类转换可能需要加载时编织。有关[LocalContainerEntityManagerFactoryBean](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/orm/jpa/LocalContainerEntityManagerFactoryBean.html) 更多详细信息，请参阅 javadoc。有关AspectJ加载时编织的更多信息，请参阅[Spring Framework中使用AspectJ的加载时编织](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-aj-ltw)。

1.15. Additional Capabilities of the ApplicationContext

As discussed in the [chapter introduction](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans), 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/5.1.8.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.

正如[章节介绍中](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans)所讨论的，该org.springframework.beans.factory 包提供了管理和操作bean的基本功能，包括以编程方式。除了扩展其他接口以提供更多面向应用程序框架的样式的附加功能外 ，该org.springframework.context软件包还添加了 [ApplicationContext](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/context/ApplicationContext.html)扩展BeanFactory接口的接口。许多人ApplicationContext以完全声明的方式使用它，甚至不以编程方式创建它，而是依赖于支持类，例如ContextLoader自动实例化 ApplicationContext作为Java EE Web应用程序的正常启动过程的一部分。

To enhance BeanFactory functionality in a more framework-oriented style, the context package also provides the following functionality:

为了BeanFactory以更加面向框架的样式增强功能，上下文包还提供以下功能：

* Access to messages in i18n-style, through the MessageSource interface.
* Access to resources, such as URLs and files, through the ResourceLoader interface.
* Event publication, namely to beans that implement the ApplicationListener interface, through the use of the ApplicationEventPublisher interface.
* Loading of multiple (hierarchical) contexts, letting each be focused on one particular layer, such as the web layer of an application, through the HierarchicalBeanFactory interface.
* 通过MessageSource界面访问i18n风格的消息。
* 通过ResourceLoader界面访问URL和文件等资源。
* 事件发布，即ApplicationListener通过使用接口实现接口的bean ApplicationEventPublisher。
* 加载多个（分层）上下文，让每个上下文通过HierarchicalBeanFactory界面聚焦在一个特定层上，例如应用程序的Web层 。

1.15.1. Internationalization using MessageSource

The ApplicationContext interface extends an interface called MessageSource and, therefore, provides internationalization (“i18n”) functionality. Spring also provides the HierarchicalMessageSource interface, which can resolve messages hierarchically. Together, these interfaces provide the foundation upon which Spring effects message resolution. The methods defined on these interfaces include:

该ApplicationContext接口扩展了一个名为的接口MessageSource，因此提供了国际化（“i18n”）功能。Spring还提供了HierarchicalMessageSource接口，可以分层次地解析消息。这些接口共同提供了Spring影响消息解析的基础。这些接口上定义的方法包括：

* String getMessage(String code, Object[] args, String default, Locale loc): 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, String default, Locale loc)：用于从中检索消息的基本方法MessageSource。如果未找到指定区域设置的消息，则使用默认消息。传入的任何参数都使用MessageFormat标准库提供的功能成为替换值。
* 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(String code, Object[] args, Locale loc)：基本上与前一个方法相同，但有一点不同：无法指定默认消息。如果找不到该消息，NoSuchMessageException则抛出a。
* 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.
* String getMessage(MessageSourceResolvable resolvable, Locale locale)：前面方法中使用的所有属性也包装在一个名为的类中MessageSourceResolvable，您可以使用此方法。

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.

当ApplicationContext被加载时，它自动搜索MessageSource 在上下文中定义的bean。bean必须具有名称messageSource。如果找到这样的bean，则对前面方法的所有调用都被委托给消息源。如果未找到任何消息源，则ApplicationContext尝试查找包含具有相同名称的bean的父级。如果是，它使用该bean作为MessageSource。如果 ApplicationContext找不到任何消息源，DelegatingMessageSource则实例化为空 以便能够接受对上面定义的方法的调用。

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 following example shows ResourceBundleMessageSource:

Spring提供了两种MessageSource实现方式，ResourceBundleMessageSource和 StaticMessageSource。两者都是HierarchicalMessageSource为了进行嵌套消息传递而实现的。在StaticMessageSource很少使用，但提供了编程的方式向消息源添加消息。以下示例显示ResourceBundleMessageSource：

<beans>

<bean id="messageSource"

class="org.springframework.context.support.ResourceBundleMessageSource">

<property name="basenames">

<list>

<value>format</value>

<value>exceptions</value>

<value>windows</value>

</list>

</property>

</bean>

</beans>

The example assumes that you have three resource bundles called format, exceptions and windows defined in your classpath. Any request to resolve a message is handled in the JDK-standard way of resolving messages through ResourceBundle objects. For the purposes of the example, assume the contents of two of the above resource bundle files are as follows:

该示例假定您有三个资源包被调用format，exceptions并windows 在类路径中定义。解决消息的任何请求都以JDK标准的方式处理，通过ResourceBundle对象解析消息。出于示例的目的，假设上述两个资源包文件的内容如下：

# in format.properties

message=Alligators rock!

# in exceptions.properties

argument.required=The {0} argument is required.

The next example shows a program to execute the MessageSource functionality. Remember that all ApplicationContextimplementations are also MessageSource implementations and so can be cast to the MessageSource interface.

下一个示例显示了执行该MessageSource功能的程序。请记住，所有ApplicationContext实现都是MessageSource 实现，因此可以强制转换为MessageSource接口。

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

MessageSource resources = **new** ClassPathXmlApplicationContext("beans.xml");

String message = resources.getMessage("message", null, "Default", null);

System.out.println(message);

}

The resulting output from the above program is as follows:

上述程序产生的结果如下：

Alligators rock!

鳄鱼摇滚！

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.

总而言之，它MessageSource是在一个名为的文件中定义的，该文件beans.xml存在于类路径的根目录中。该messageSourcebean定义是指通过它的一些资源包的basenames属性。这是在列表中传递的三个文件basenames属性存在于你的classpath根目录的文件，被称为format.properties，exceptions.properties和 windows.properties分别。

The next example shows arguments passed to the message lookup. These arguments are converted into String objects and inserted into placeholders in the lookup message.

下一个示例显示传递给消息查找的参数。这些参数将转换为String对象并插入到查找消息中的占位符中。

<beans>

*<!-- this MessageSource is being used in a web application -->*

<bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">

<property name="basename" value="exceptions"/>

</bean>

*<!-- lets inject the above MessageSource into this POJO -->*

<bean id="example" class="com.something.Example">

<property name="messages" ref="messageSource"/>

</bean>

</beans>

**public** **class** **Example** {

**private** MessageSource messages;

**public** **void** setMessages(MessageSource messages) {

this.messages = messages;

}

**public** **void** execute() {

String message = this.messages.getMessage("argument.required",

**new** Object **[]** {"userDao"}, "Required", null);

System.out.println(message);

}

}

The resulting output from the invocation of the execute() method is as follows:

调用该execute()方法得到的结果如下：

The userDao argument is required.

userDao参数是必需的。

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.

关于国际化（“i18n”），Spring的各种MessageSource 实现遵循与标准JDK相同的区域设置解析和回退规则 ResourceBundle。总之，和继续该示例messageSource先前定义的，如果你想解析British（消息en-GB）语言环境中，您将创建文件名为format\_en\_GB.properties，exceptions\_en\_GB.properties和 windows\_en\_GB.properties分别。

Typically, locale resolution is managed by the surrounding environment of the application. In the following example, the locale against which (British) messages are resolved is specified manually:

通常，区域设置解析由应用程序的周围环境管理。在以下示例中，手动指定解析（英国）消息的区域设置：

# in exceptions\_en\_GB.properties

argument.required=Ebagum lad, the {0} argument is required, I say, required.

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

MessageSource resources = **new** ClassPathXmlApplicationContext("beans.xml");

String message = resources.getMessage("argument.required",

**new** Object **[]** {"userDao"}, "Required", Locale.UK);

System.out.println(message);

}

The resulting output from the running of the above program is as follows:

运行上述程序产生的结果如下：

Ebagum lad, the 'userDao' argument is required, I say, required.

Ebagum小伙子，我说，'userDao'论证是必需的。

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.

您还可以使用该MessageSourceAware界面获取对MessageSource已定义的任何内容的引用 。在创建和配置bean时，将使用应用程序上下文注入ApplicationContext实现MessageSourceAware接口的任何bean MessageSource。

|  |  |
| --- | --- |
|  | 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 only from the classpath) and supports hot reloading of bundle property files (while efficiently caching them in between). See the [ReloadableResourceBundleMessageSource](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/context/support/ReloadableResourceBundleMessageSource.html) javadoc for details.  作为替代ResourceBundleMessageSource，Spring提供了一个 ReloadableResourceBundleMessageSource类。此变体支持相同的捆绑文件格式，但比基于标准JDK的ResourceBundleMessageSource实现更灵活 。特别是，它允许从任何Spring资源位置（不仅从类路径）读取文件，并支持bundle属性文件的热重新加载（同时有效地在它们之间缓存它们）。有关[ReloadableResourceBundleMessageSource](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/context/support/ReloadableResourceBundleMessageSource.html) 详细信息，请参阅javadoc。 |

1.15.2. Standard and Custom Events

Event handling in the ApplicationContext is provided through the ApplicationEvent class and the ApplicationListenerinterface. If a bean that implements the ApplicationListener interface is deployed into the context, every time anApplicationEvent gets published to the ApplicationContext, that bean is notified. Essentially, this is the standard Observer design pattern.

ApplicationContext通过ApplicationEvent 类和ApplicationListener接口提供事件处理。如果将实现ApplicationListener接口的bean 部署到上下文中，则每次 ApplicationEvent将其发布到该ApplicationContextbean时，都会通知该bean。从本质上讲，这是标准的Observer设计模式。

|  |  |
| --- | --- |
|  | As of Spring 4.2, the event infrastructure has been significantly improved and offers an [annotation-based model](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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.  从Spring 4.2开始，事件基础结构得到了显着改进，并提供了[基于注释的模型](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#context-functionality-events-annotation)以及发布任意事件（即不一定从中扩展的对象ApplicationEvent）的能力。当发布这样的对象时，我们将它包装在一个事件中。 |

The following table describes the standard events that Spring provides:

下表描述了Spring提供的标准事件：

|  |  |
| --- | --- |
| *Table 7. Built-in Events* | |
| **Event** | **Explanation** |
| ContextRefreshedEvent | Published when the ApplicationContext is initialized or refreshed (for example, by using the refresh() method on the ConfigurableApplicationContext interface). Here, “initialized” 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, butGenericApplicationContext does not. |
| ContextStartedEvent | Published when the ApplicationContext is started by using the start() method on the ConfigurableApplicationContext interface. Here, “started” 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 by using the stop() method on the ConfigurableApplicationContext interface. Here, “stopped” 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 by using the close() method on the ConfigurableApplicationContext interface. Here, “closed” 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 that use Spring’s DispatcherServlet. |

|  |  |
| --- | --- |
| *表7.内置事件* | |
| **事件** | **说明** |
| ContextRefreshedEvent | ApplicationContext初始化或刷新时发布（例如，通过refresh()在ConfigurableApplicationContext接口上使用该方法）。这里，“初始化”意味着加载所有bean，检测并激活后处理器bean，预先实例化单例，并且ApplicationContext对象已准备好使用。只要上下文尚未关闭，只要所选择的ApplicationContext实际支持这种“热”刷新，就可以多次触发刷新。例如，XmlWebApplicationContext支持热刷新，但GenericApplicationContext不支持 。 |
| ContextStartedEvent | ApplicationContext通过start()在ConfigurableApplicationContext接口上使用该方法启动时发布。这里，“已启动”意味着所有Lifecycle bean都会收到明确的启动信号。通常，此信号用于在显式停止后重新启动Bean，但它也可用于启动尚未为自动启动配置的组件（例如，尚未在初始化时启动的组件）。 |
| ContextStoppedEvent | ApplicationContext通过stop()在ConfigurableApplicationContext接口上使用方法 停止时发布。这里，“停止”意味着所有Lifecycle bean都会收到明确的停止信号。可以通过start()呼叫重新启动已停止的上下文 。 |
| ContextClosedEvent | ApplicationContext通过close()在ConfigurableApplicationContext接口上使用方法 关闭时发布。这里，“关闭”意味着所有单例bean都被销毁。封闭的环境达到了生命的终点。它无法刷新或重新启动。 |
| RequestHandledEvent | 一个特定于Web的事件，告诉所有bean已经为HTTP请求提供服务。请求完成后发布此事件。此事件仅适用于使用Spring的Web应用程序DispatcherServlet。 |

You can also create and publish your own custom events. The following example shows a simple class that extends Spring’s ApplicationEvent base class:

您还可以创建和发布自己的自定义事件。以下示例显示了一个扩展Spring ApplicationEvent基类的简单类：

**public** **class** **BlackListEvent** **extends** ApplicationEvent {

**private** **final** String address;

**private** **final** String content;

**public** BlackListEvent(Object source, String address, String content) {

super(source);

this.address = address;

this.content = content;

}

*// accessor and other methods...*

}

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 shows such a class:

要发布自定义ApplicationEvent，请在publishEvent()方法上调用该方法 ApplicationEventPublisher。通常，这是通过创建一个实现ApplicationEventPublisherAware并将其注册为Spring bean 的类来完成的 。以下示例显示了这样一个类：

**public** **class** **EmailService** **implements** ApplicationEventPublisherAware {

**private** List<String> blackList;

**private** ApplicationEventPublisher publisher;

**public** **void** setBlackList(List<String> blackList) {

this.blackList = blackList;

}

**public** **void** setApplicationEventPublisher(ApplicationEventPublisher publisher) {

this.publisher = publisher;

}

**public** **void** sendEmail(String address, String content) {

**if** (blackList.contains(address)) {

publisher.publishEvent(**new** BlackListEvent(this, address, content));

**return**;

}

*// send email...*

}

}

At configuration time, the Spring container detects that EmailService implements ApplicationEventPublisherAware and automatically calls setApplicationEventPublisher(). In reality, the parameter passed in is the Spring container itself. You are interacting with the application context through its ApplicationEventPublisher interface.

在配置时，Spring容器检测到EmailService实现 ApplicationEventPublisherAware并自动调用 setApplicationEventPublisher()。实际上，传入的参数是Spring容器本身。您正通过其ApplicationEventPublisher界面与应用程序上下文进行 交互。

To receive the custom ApplicationEvent, you can create a class that implements ApplicationListener and register it as a Spring bean. The following example shows such a class:

要接收自定义ApplicationEvent，您可以创建一个实现 ApplicationListener并将其注册为Spring bean的类。以下示例显示了这样一个类：

**public** **class** **BlackListNotifier** **implements** ApplicationListener<BlackListEvent> {

**private** String notificationAddress;

**public** **void** setNotificationAddress(String notificationAddress) {

this.notificationAddress = notificationAddress;

}

**public** **void** onApplicationEvent(BlackListEvent event) {

*// notify appropriate parties via notificationAddress...*

}

}

Notice that ApplicationListener is generically parameterized with the type of your custom event (BlackListEvent in the preceding example). This means that the onApplicationEvent() method can remain type-safe, avoiding any need for downcasting. You can register as many event listeners as you wish, but note that, by default, event listeners receive events synchronously. This means that 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, See the javadoc for Spring’s [ApplicationEventMulticaster](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/context/event/ApplicationEventMulticaster.html) interface.

请注意，ApplicationListener通常使用自定义事件的类型进行参数化（BlackListEvent在前面的示例中）。这意味着该onApplicationEvent()方法可以保持类型安全，避免任何向下转换的需要。您可以根据需要注册任意数量的事件侦听器，但请注意，默认情况下，事件侦听器会同步接收事件。这意味着该publishEvent()方法将阻塞，直到所有侦听器都已完成对事件的处理。这种同步和单线程方法的一个优点是，当侦听器接收到事件时，如果事务上下文可用，它将在发布者的事务上下文内运行。如果需要另一个事件发布策略，请参阅Spring的[ApplicationEventMulticaster](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/context/event/ApplicationEventMulticaster.html)界面的javadoc 。

The following example shows the bean definitions used to register and configure each of the classes above:

以下示例显示了用于注册和配置上述每个类的bean定义：

<bean id="emailService" class="example.EmailService">

<property name="blackList">

<list>

<value>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 email messages that should be blacklisted, a custom event of type BlackListEvent is published. The blackListNotifier bean is registered as anApplicationListener and receives the BlackListEvent, at which point it can notify appropriate parties.

总而言之，当调用bean 的sendEmail()方法时emailService，如果有任何应该列入黑名单的电子邮件消息，BlackListEvent则会发布类型的自定义事件 。该blackListNotifierbean被注册为 ApplicationListener与接收BlackListEvent，此时它可以通知有关各方。

|  |  |
| --- | --- |
|  | 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](https://www.enterpriseintegrationpatterns.com/), event-driven architectures that build upon the well-known Spring programming model.  Spring的事件机制是为在同一应用程序上下文中的Spring bean之间的简单通信而设计的。但是，对于更复杂的企业集成需求，单独维护的 [Spring Integration](https://projects.spring.io/spring-integration/)项目为构建基于众所周知的Spring编程模型的轻量级，[面向模式](https://www.enterpriseintegrationpatterns.com/)，事件驱动的体系结构提供了完整的支持 。 |

Annotation-based Event Listeners

As of Spring 4.2, you can register an event listener on any public method of a managed bean by using the EventListenerannotation. The BlackListNotifier can be rewritten as follows:

从Spring 4.2开始，您可以使用EventListener注释在托管bean的任何公共方法上注册事件侦听器。该BlackListNotifier可改写如下：

**public** **class** **BlackListNotifier** {

**private** String notificationAddress;

**public** **void** setNotificationAddress(String notificationAddress) {

this.notificationAddress = notificationAddress;

}

@EventListener

**public** **void** processBlackListEvent(BlackListEvent event) {

*// notify appropriate parties via notificationAddress...*

}

}

The method signature once again declares the event type to which it listens, 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.

方法签名再次声明它侦听的事件类型，但这次使用灵活的名称并且没有实现特定的侦听器接口。只要实际事件类型在其实现层次结构中解析通用参数，也可以通过泛型缩小事件类型。

If your method should listen to several events or if you want to define it with no parameter at all, the event types can also be specified on the annotation itself. The following example shows how to do so:

如果您的方法应该监听多个事件，或者您想要根据任何参数进行定义，那么也可以在注释本身上指定事件类型。以下示例显示了如何执行此操作：

@EventListener({ContextStartedEvent.class, ContextRefreshedEvent.class})

**public** **void** handleContextStart() {

...

}

It is also possible to add additional runtime filtering by using the condition attribute of the annotation that defines a [SpELexpression](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#expressions) , which should match to actually invoke the method for a particular event.

还可以通过使用condition定义[SpEL表达式](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#expressions)的注释的属性来添加额外的运行时过滤，该属性应该匹配以实际调用特定事件的方法。

The following example shows how our notifier can be rewritten to be invoked only if the content attribute of the event is equal to my-event:

以下示例显示了仅当content事件的属性等于时，才能重写我们的通知程序以进行调用 my-event：

@EventListener(condition = "#blEvent.content == 'my-event'")

**public** **void** processBlackListEvent(BlackListEvent blEvent) {

*// notify appropriate parties via notificationAddress...*

}

Each SpEL expression evaluates against a dedicated context. The following table lists the items made available to the context so that you can use them for conditional event processing:

每个SpEL表达式都针对专用上下文进行评估。下表列出了可用于上下文的项目，以便您可以将它们用于条件事件处理：

|  |  |  |  |
| --- | --- | --- | --- |
| *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 | The name of any of the method arguments. If, for some reason, the names are not available (for example, because there is 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 (you can also use #p0 or #p<#arg>notation as an alias) |

|  |  |  |  |
| --- | --- | --- | --- |
| *表8.事件SpEL可用元数据* | | | |
| **名称** | **地点** | **描述** | **例** |
| 事件 | 根对象 | 实际的ApplicationEvent。 | #root.event |
| 参数数组 | 根对象 | 用于调用目标的参数（作为数组）。 | #root.args[0] |
| *参数名称* | 评估背景 | 任何方法参数的名称。如果由于某种原因，名称不可用（例如，因为没有调试信息），参数名称也可以在#a<#arg>where #arg参数索引（从0开始）下找到。 | #blEvent或#a0（您也可以使用#p0或#p<#arg>表示法作为别名） |

Note that #root.event gives you access to the underlying event, even if your method signature actually refers to an arbitrary object that was published.

请注意#root.event，即使您的方法签名实际引用已发布的任意对象，也可以访问基础事件。

If you need to publish an event as the result of processing another event, you can change the method signature to return the event that should be published, as the following example shows:

如果您需要作为处理其他事件的结果发布事件，则可以更改方法签名以返回应发布的事件，如以下示例所示：

@EventListener

**public** ListUpdateEvent handleBlackListEvent(BlackListEvent event) {

*// notify appropriate parties via notificationAddress and*

*// then publish a ListUpdateEvent...*

}

|  |  |
| --- | --- |
|  | This feature is not supported for [asynchronous listeners](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#context-functionality-events-async).  [异步侦听](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#context-functionality-events-async) 器不支持此功能 。 |

This new method publishes a new ListUpdateEvent for every BlackListEvent handled by the method above. If you need to publish several events, you can return a Collection of events instead.

这个新方法ListUpdateEvent为BlackListEvent上述方法处理的每个方法发布一个新的方法。如果您需要发布多个事件，则可以返回一个Collection事件。

Asynchronous Listeners

If you want a particular listener to process events asynchronously, you can reuse the [regular @Async support](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/integration.html#scheduling-annotation-support-async). The following example shows how to do so:

如果希望特定侦听器异步处理事件，则可以重用 [常规@Async支持](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/integration.html#scheduling-annotation-support-async)。以下示例显示了如何执行此操作：

@EventListener

@Async

**public** **void** processBlackListEvent(BlackListEvent event) {

*// BlackListEvent is processed in a separate thread*

}

Be aware of the following limitations when using asynchronous events:

使用异步事件时请注意以下限制：

* If the event listener throws an Exception, it is not propagated to the caller See AsyncUncaughtExceptionHandler for more details.
* 如果事件侦听器抛出一个Exception，则它不会传播给调用者。有关AsyncUncaughtExceptionHandler详细信息，请参阅。
* Such event listener cannot send replies. If you need to send another event as the result of the processing, inject[ApplicationEventPublisher](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/aop/interceptor/AsyncUncaughtExceptionHandler.html) to send the event manually.
* 此类事件监听器无法发送回复。如果您需要作为处理结果发送另一个事件，请注入 [ApplicationEventPublisher](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/aop/interceptor/AsyncUncaughtExceptionHandler.html) 以手动发送事件。

Ordering Listeners

If you need one listener to be invoked before another one, you can add the @Order annotation to the method declaration, as the following example shows:

如果需要在另一个侦听器之前调用一个侦听器，则可以将@Order 注释添加到方法声明中，如以下示例所示：

@EventListener

@Order(42)

**public** **void** processBlackListEvent(BlackListEvent event) {

*// notify appropriate parties via notificationAddress...*

}

Generic Events

You can also use generics to further define the structure of your event. Consider using an EntityCreatedEvent<T> where T is the type of the actual entity that got created. For example, you can create the following listener definition to receive only EntityCreatedEvent for a Person:

您还可以使用泛型来进一步定义事件的结构。考虑使用 EntityCreatedEvent<T>where T是创建的实际实体的类型。例如，您可以创建以下侦听器定义只接收EntityCreatedEvent了 Person：

@EventListener

**public** **void** onPersonCreated(EntityCreatedEvent<Person> event) {

...

}

Due to type erasure, this works only if the event that is fired resolves the generic parameters on which the event listener filters (that is, something like class PersonCreatedEvent extends EntityCreatedEvent<Person> { …​ }).

由于类型擦除，仅当被触发的事件解析事件侦听器过滤的泛型参数（即类似的东西 class PersonCreatedEvent extends EntityCreatedEvent<Person> { …​ }）时，此方法才有效。

In certain circumstances, this may become quite tedious if all events follow the same structure (as should be the case for the event in the preceding example). In such a case, you can implement ResolvableTypeProvider to guide the framework beyond what the runtime environment provides. The following event shows how to do so:

在某些情况下，如果所有事件都遵循相同的结构，这可能会变得相当繁琐（前面示例中的事件应该如此）。在这种情况下，您可以实现ResolvableTypeProvider指导框架超出运行时环境提供的范围。以下事件显示了如何执行此操作：

**public** **class** **EntityCreatedEvent**<T> **extends** ApplicationEvent **implements** ResolvableTypeProvider {

**public** EntityCreatedEvent(T entity) {

super(entity);

}

@Override

**public** ResolvableType getResolvableType() {

**return** ResolvableType.forClassWithGenerics(getClass(), ResolvableType.forInstance(getSource()));

}

}

|  |  |
| --- | --- |
|  | This works not only for ApplicationEvent but any arbitrary object that you send as an event.  这不仅适用于ApplicationEvent您作为事件发送的任何对象。 |

1.15.3. Convenient Access to Low-level Resources

For optimal usage and understanding of application contexts, you should familiarize yourself with Spring’s Resourceabstraction, as described in [Resources](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#resources).

为了最佳地使用和理解应用程序上下文，您应该熟悉Spring的Resource抽象，如[参考资料中所述](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#resources)。

An application context is a ResourceLoader, which can be used to load Resource objects. A Resource is essentially a more feature rich version of the JDK java.net.URL class. 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.

应用程序上下文是a ResourceLoader，可用于加载Resource对象。A Resource本质上是JDK java.net.URL类的功能更丰富的版本。实际上，在适当Resource的情况下包装一个实例的实现java.net.URL。A Resource可以透明的方式从几乎任何位置获取低级资源，包括从类路径，文件系统位置，任何可用标准URL描述的位置，以及一些其他变体。如果资源位置字符串是没有任何特殊前缀的简单路径，那么这些资源来自特定且适合于实际应用程序上下文类型。

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 the ResourceLoader. You can also expose properties of type Resource, to be used to access static resources. They are injected into it like any other properties. You can specify those Resource properties as simple String paths and rely on automatic conversion from those text strings to actual Resource objects when the bean is deployed.

您可以配置部署到应用程序上下文中的bean来实现特殊的回调接口，ResourceLoaderAware在初始化时自动回调，应用程序上下文本身作为传入ResourceLoader。您还可以公开Resource要用于访问静态资源的类型属性。它们像任何其他属性一样被注入其中。您可以将这些Resource 属性指定为简单String路径，并在Resource部署Bean时依赖从这些文本字符串到实际对象的自动转换。

The location path or paths supplied to an ApplicationContext constructor are actually resource strings and, in simple form, are treated appropriately according to the specific context implementation. For example 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.

提供给ApplicationContext构造函数的位置路径实际上是资源字符串，并且以简单的形式根据特定的上下文实现进行适当处理。例如，ClassPathXmlApplicationContext将简单的位置路径视为类路径位置。您还可以使用具有特殊前缀的位置路径（资源字符串）来强制从类路径或URL加载定义，而不管实际的上下文类型如何。

1.15.4. 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.

您可以ApplicationContext使用例如a以声明方式创建实例 ContextLoader。当然，您也可以ApplicationContext使用其中一个ApplicationContext实现以编程方式创建实例。

You can register an ApplicationContext by using the ContextLoaderListener, as the following example shows:

您可以ApplicationContext使用ContextLoaderListener，注册一个，如下例所示：

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/daoContext.xml /WEB-INF/applicationContext.xml</param-value>

</context-param>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

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 are searched. Ant-style path patterns are supported as well. Examples are /WEB-INF/\*Context.xml (for all files with names that end with Context.xml and that reside in the WEB-INF directory) and /WEB-INF/\*\*/\*Context.xml (for all such files in any subdirectory of WEB-INF).

监听器检查contextConfigLocation参数。如果参数不存在，则侦听器将/WEB-INF/applicationContext.xml默认使用。当参数确实存在时，侦听器String使用预定义的分隔符（逗号，分号和空格）分隔，并将值用作搜索应用程序上下文的位置。还支持Ant样式的路径模式。示例是/WEB-INF/\*Context.xml（对于名称Context.xml以WEB-INF目录结尾 并位于目录/WEB-INF/\*\*/\*Context.xml 中的所有文件）和（对于任何子目录中的所有此类文件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 stand-alone ApplicationContext(only hosted in Java EE environment) being able to access the Java EE servers facilities. RAR deployment is a more natural alternative to a 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.

可以将Spring部署ApplicationContext为RAR文件，将上下文及其所有必需的bean类和库JAR封装在Java EE RAR部署单元中。这相当于ApplicationContext能够访问Java EE服务器工具的独立引导（仅在Java EE环境中托管）。RAR部署是部署无头WAR文件的一种更自然的替代方案 - 实际上是一个没有任何HTTP入口点的WAR文件，仅用于ApplicationContext在Java EE环境中引导Spring 。

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 DataSource instances and JMS ConnectionFactory instances and can 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.

RAR部署非常适用于不需要HTTP入口点但仅包含消息端点和预定作业的应用程序上下文。在这样的上下文中的Bean可以使用应用程序服务器资源，例如JTA事务管理器和JNDI绑定的JDBC DataSource实例和JMS ConnectionFactory实例，并且还可以通过Spring的标准事务管理以及JNDI和JMX支持工具向平台的JMX服务器注册。应用程序组件还可以WorkManager通过Spring的TaskExecutor抽象与应用程序服务器的JCA交互。

See the javadoc of the [SpringContextResourceAdapter](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/jca/context/SpringContextResourceAdapter.html) class for the configuration details involved in RAR deployment.

有关[SpringContextResourceAdapter](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/jca/context/SpringContextResourceAdapter.html) RAR部署中涉及的配置详细信息，请参阅该类的javadoc 。

For a simple deployment of a Spring ApplicationContext as a Java EE RAR file:

对于将Spring ApplicationContext简单部署为Java EE RAR文件：

1. 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 the [javadoc for SpringContextResourceAdapter](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/jca/context/SpringContextResourceAdapter.html)) and the corresponding Spring XML bean definition file(s) (typically `META-INF/applicationContext.xml).

将所有应用程序类打包到一个RAR文件（这是一个具有不同文件扩展名的标准JAR文件）。。将所有必需的库JAR添加到RAR存档的根目录中。。添加 META-INF/ra.xml部署描述符（如[javadoc中SpringContextResourceAdapter](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/jca/context/SpringContextResourceAdapter.html)所示）和相应的Spring XML bean定义文件（通常为“META-INF / applicationContext.xml”）。

1. Drop the resulting RAR file into your application server’s deployment directory.

将生成的RAR文件放入应用程序服务器的部署目录中。

|  |  |
| --- | --- |
|  | 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 or react 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 may be used by other application modules on the same machine.  这种RAR部署单元通常是独立的。它们不会将组件暴露给外部世界，甚至不会暴露给同一应用程序的其他模块。与基于RAR的交互ApplicationContext通常通过与其他模块共享的JMS目标进行。ApplicationContext例如，基于RAR的还可以调度一些作业或对文件系统（或类似物）中的新文件作出反应。如果它需要允许来自外部的同步访问，它可以（例如）导出RMI端点，这可以由同一台机器上的其他应用程序模块使用。 |

1.16. The BeanFactory

The BeanFactory API provides the underlying basis for Spring’s IoC functionality. Its specific contracts are mostly used in integration with other parts of Spring and related third-party frameworks, and its DefaultListableBeanFactory implementation is a key delegate within the higher-level GenericApplicationContext container.

该BeanFactoryAPI提供了春天的IoC功能的基本依据。其特定合同主要用于与Spring的其他部分和相关的第三方框架集成，其DefaultListableBeanFactory实现是更高级别GenericApplicationContext容器中的关键委托。

BeanFactory and related interfaces (such as BeanFactoryAware, InitializingBean, DisposableBean) are important integration points for other framework components. By not requiring any annotations or even reflection, they allow for very efficient interaction between the container and its components. Application-level beans may use the same callback interfaces but typically prefer declarative dependency injection instead, either through annotations or through programmatic configuration.

BeanFactory和相关接口（例如BeanFactoryAware，InitializingBean， DisposableBean）对于其他框架组件的重要结合点。通过不需要任何注释或甚至反射，它们允许容器与其组件之间的非常有效的交互。应用程序级bean可以使用相同的回调接口，但通常更喜欢通过注释或通过编程配置进行声明性依赖注入。

Note that the core BeanFactory API level and its DefaultListableBeanFactory implementation do not make assumptions about the configuration format or any component annotations to be used. All of these flavors come in through extensions (such as XmlBeanDefinitionReader and AutowiredAnnotationBeanPostProcessor) and operate on shared BeanDefinition objects as a core metadata representation. This is the essence of what makes Spring’s container so flexible and extensible.

请注意，核心BeanFactoryAPI级别及其DefaultListableBeanFactory 实现不会对配置格式或要使用的任何组件注释做出假设。所有这些风格都通过扩展（例如XmlBeanDefinitionReader和AutowiredAnnotationBeanPostProcessor）进行，并BeanDefinition作为核心元数据表示在共享对象上运行。这是使Spring的容器如此灵活和可扩展的本质。

1.16.1. BeanFactory or ApplicationContext?

This section explains the differences between the BeanFactory and ApplicationContext container levels and the implications on bootstrapping.

本节介绍之间的差异BeanFactory和 ApplicationContext容器级别和引导的意义。

You should use an ApplicationContext unless you have a good reason for not doing so, with GenericApplicationContext and its subclass AnnotationConfigApplicationContext as the common implementations for custom bootstrapping. These are the primary entry points to Spring’s core container for all common purposes: loading of configuration files, triggering a classpath scan, programmatically registering bean definitions and annotated classes, and (as of 5.0) registering functional bean definitions.

您应该使用a，ApplicationContext除非您有充分的理由不这样做， GenericApplicationContext并将其子类AnnotationConfigApplicationContext 作为自定义引导的常见实现。这些是Spring用于所有常见目的的核心容器的主要入口点：加载配置文件，触发类路径扫描，以编程方式注册bean定义和带注释的类，以及（从5.0开始）注册功能bean定义。

Because an ApplicationContext includes all the functionality of a BeanFactory, it is generally recommended over a plain BeanFactory, except for scenarios where full control over bean processing is needed. Within an ApplicationContext (such as the GenericApplicationContext implementation), several kinds of beans are detected by convention (that is, by bean name or by bean type — in particular, post-processors), while a plain DefaultListableBeanFactory is agnostic about any special beans.

因为a ApplicationContext包含a的所有功能BeanFactory，所以BeanFactory除了需要完全控制bean处理的场景之外，通常建议使用它。在一个ApplicationContext（例如 GenericApplicationContext实现）中，按照约定（即通过bean名称或bean类型 - 特别是后处理器）检测到几种bean，而plain DefaultListableBeanFactory对任何特殊bean都是不可知的。

For many extended container features, such as annotation processing and AOP proxying, the [BeanPostProcessor extension point](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-extension-bpp) is essential. If you use only a plain DefaultListableBeanFactory, such post-processors do not get detected and activated by default. This situation could be confusing, because nothing is actually wrong with your bean configuration. Rather, in such a scenario, the container needs to be fully bootstrapped through additional setup.

对于许多扩展容器功能，例如注释处理和AOP代理，[BeanPostProcessor扩展点](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-extension-bpp)是必不可少的。如果仅使用普通DefaultListableBeanFactory处理器，则默认情况下不会检测到并激活此类后处理器。这种情况可能令人困惑，因为您的bean配置实际上没有任何问题。相反，在这种情况下，容器需要通过额外的设置完全自举。

The following table lists features provided by the BeanFactory and ApplicationContext interfaces and implementations.

下表列出了提供的功能BeanFactory和 ApplicationContext接口和实现。

|  |  |  |
| --- | --- | --- |
| *Table 9. Feature Matrix* | | |
| **Feature** | BeanFactory | ApplicationContext |
| Bean instantiation/wiring | Yes | Yes |
| Integrated lifecycle management | No | Yes |
| Automatic BeanPostProcessor registration | No | Yes |
| Automatic BeanFactoryPostProcessor registration | No | Yes |
| Convenient MessageSource access (for internalization) | No | Yes |
| Built-in ApplicationEvent publication mechanism | No | Yes |

|  |  |  |
| --- | --- | --- |
| *表9.特征矩阵* | | |
| **特征** | BeanFactory | ApplicationContext |
| Bean实例化/布线 | 是 | 是 |
| 集成的生命周期管理 | 没有 | 是 |
| 自动BeanPostProcessor注册 | 没有 | 是 |
| 自动BeanFactoryPostProcessor注册 | 没有 | 是 |
| 方便MessageSource访问（内化） | 没有 | 是 |
| 内置ApplicationEvent发布机制 | 没有 | 是 |

To explicitly register a bean post-processor with a DefaultListableBeanFactory, you need to programmatically call addBeanPostProcessor, as the following example shows:

要使用a显式注册bean后处理器DefaultListableBeanFactory，您需要以编程方式调用addBeanPostProcessor，如以下示例所示：

DefaultListableBeanFactory factory = **new** DefaultListableBeanFactory();

*// populate the factory with bean definitions*

*// now register any needed BeanPostProcessor instances*

factory.addBeanPostProcessor(**new** AutowiredAnnotationBeanPostProcessor());

factory.addBeanPostProcessor(**new** MyBeanPostProcessor());

*// now start using the factory*

To apply a BeanFactoryPostProcessor to a plain DefaultListableBeanFactory, you need to call its postProcessBeanFactorymethod, as the following example shows:

要应用于BeanFactoryPostProcessorplain DefaultListableBeanFactory，需要调用其postProcessBeanFactory方法，如以下示例所示：

DefaultListableBeanFactory factory = **new** DefaultListableBeanFactory();

XmlBeanDefinitionReader reader = **new** XmlBeanDefinitionReader(factory);

reader.loadBeanDefinitions(**new** FileSystemResource("beans.xml"));

*// bring in some property values from a Properties file*

PropertyPlaceholderConfigurer cfg = **new** PropertyPlaceholderConfigurer();

cfg.setLocation(**new** FileSystemResource("jdbc.properties"));

*// now actually do the replacement*

cfg.postProcessBeanFactory(factory);

In both cases, the explicit registration steps are inconvenient, which is why the various ApplicationContext variants are preferred over a plain DefaultListableBeanFactory in Spring-backed applications, especially when relying on BeanFactoryPostProcessor and BeanPostProcessor instances for extended container functionality in a typical enterprise setup.

在这两种情况下，显式注册步骤都不方便，这就是为什么各种ApplicationContext变体优于DefaultListableBeanFactorySpring支持的应用程序中的普通模式 ，尤其是在典型企业设置中依赖于BeanFactoryPostProcessor和BeanPostProcessor扩展容器功能的实例时。

|  |  |
| --- | --- |
|  | An AnnotationConfigApplicationContext has all common annotation post-processors registered and may bring in additional processors underneath the covers through configuration annotations, such as @EnableTransactionManagement. At the abstraction level of Spring’s annotation-based configuration model, the notion of bean post-processors becomes a mere internal container detail.  An AnnotationConfigApplicationContext已经注册了所有常见的注释后处理器，并且可以通过配置注释在封面下引入额外的处理器，例如@EnableTransactionManagement。在Spring的基于注释的配置模型的抽象级别，bean后处理器的概念变成仅仅是内部容器细节。 |

2. Resources

This chapter covers how Spring handles resources and how you can work with resources in Spring. It includes the following topics:

* [Introduction](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#resources-introduction)
* [The Resource Interface](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#resources-resource)
* [Built-in Resource Implementations](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#resources-implementations)
* [The ResourceLoader](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#resources-resourceloader)
* [The ResourceLoaderAware interface](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#resources-resourceloaderaware)
* [Resources as Dependencies](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#resources-as-dependencies)
* [Application Contexts and Resource Paths](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#resources-app-ctx)

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. The following listing shows the Resource interface definition:

**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();

}

As the definition of the Resource interface shows, it extends the InputStreamSource interface. The following listing shows the definition of the InputStreamSource interface:

**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. Returns 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 let you obtain an actual URL or File object representing the resource (if the underlying implementation is compatible and supports that functionality).

Spring itself uses the Resource abstraction extensively, 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 aString 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, let the caller 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 is actually very useful to use as a general utility class by itself in your own code, for access to resources, even when your code does not 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 serve as a more capable replacement for URL and can be considered equivalent to any other library you would use for this purpose.

|  |  |
| --- | --- |
|  | The Resource abstraction does not replace functionality. It wraps it where possible. For example, a UrlResourcewraps a URL and uses the wrapped URL to do its work. |

2.3. Built-in Resource Implementations

Spring includes the following Resource implementations:

* [UrlResource](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#resources-implementations-urlresource)
* [ClassPathResource](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#resources-implementations-classpathresource)
* [FileSystemResource](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#resources-implementations-filesystemresource)
* [ServletContextResource](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#resources-implementations-servletcontextresource)
* [InputStreamResource](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#resources-implementations-inputstreamresource)
* [ByteArrayResource](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#resources-implementations-bytearrayresource)

2.3.1. UrlResource

UrlResource wraps a java.net.URL and can be used to access any object that is normally accessible with a URL, such as files, an HTTP target, an FTP target, and others. 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 through the HTTP protocol, ftp: for accessing resources through FTP, and others.

A UrlResource is created by Java code by explicitly using the UrlResource constructor but is often created implicitly when you call an API method that takes a String argument meant to represent a path. For the latter case, a JavaBeans PropertyEditorultimately decides which type of Resource to create. If the path string contains well-known (to it, that is) prefix (such as classpath:), it creates an appropriate specialized Resource for that prefix. However, if it does not recognize the prefix, it assume the string is a standard URL string and creates a UrlResource.

2.3.2. ClassPathResource

This class represents a resource that should be obtained from the classpath. It 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 that 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 by explicitly using the ClassPathResource constructor but is often created implicitly when you call an API method that takes a String argument meant to represent a path. For the latter case, a JavaBeans PropertyEditor recognizes the special prefix, classpath:, on the string path and creates a ClassPathResource in that case.

2.3.3. FileSystemResource

This is a Resource implementation for java.io.File and java.nio.file.Path handles. It supports resolution as a File and as a URL.

2.3.4. ServletContextResource

This is a Resource implementation for ServletContext resources that interprets relative paths within the relevant web application’s root directory.

It always supports stream access and URL access but allows java.io.File access only when the web application archive is expanded and the resource is physically on the filesystem. Whether or not it is expanded and on the filesystem or accessed directly from the JAR or somewhere else like a database (which is conceivable) is actually dependent on the Servlet container.

2.3.5. InputStreamResource

An InputStreamResource is a Resource implementation for a given InputStream. It should be used only if no specific Resourceimplementation 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, it returns 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 is 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 (that is, load) Resource instances. The following listing shows the ResourceLoader interface definition:

**public** **interface** **ResourceLoader** {

Resource getResource(String location);

}

All application contexts implement the ResourceLoader interface. 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 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");

Against a ClassPathXmlApplicationContext, that code returns a ClassPathResource. If the same method were executed against a FileSystemXmlApplicationContext instance, it would return a FileSystemResource. For a WebApplicationContext, it would return a ServletContextResource. It would similarly return appropriate objects for each context.

As a result, 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, as the following example shows:

Resource template = ctx.getResource("classpath:some/resource/path/myTemplate.txt");

Similarly, you can force a UrlResource to be used by specifying any of the standard java.net.URL prefixes. The following pair of examples use the file and http prefixes:

Resource template = ctx.getResource("file:///some/resource/path/myTemplate.txt");

Resource template = ctx.getResource("https://myhost.com/resource/path/myTemplate.txt");

The following table summarizes the strategy for converting String objects to Resource objects:

| *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. See also [FileSystemResource Caveats](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#resources-filesystemresource-caveats). |
| http: | <https://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 callback interface which identifies components that expect to be provided with a ResourceLoader reference. The following listing shows the definition of the ResourceLoaderAware interface:

**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 then invokes setResourceLoader(ResourceLoader), supplying itself as the argument (remember, all application contexts in Spring implement the ResourceLoader interface).

Since an ApplicationContext is a ResourceLoader, the bean could also implement the ApplicationContextAware interface and use the supplied application context directly to load resources. However, in general, it is better to use the specialized ResourceLoader interface if that is all you need. The code would be coupled only to the resource loading interface (which can be considered a utility interface) and not to the whole Spring ApplicationContext interface.

In application components, you may also 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/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-autowire)) are capable of providing a ResourceLoader for either a constructor argument or a setter method parameter, respectively. For more flexibility (including the ability to autowire fields and multiple parameter methods), consider using the annotation-based autowiring features. In that case, the ResourceLoader is autowired into a field, constructor argument, or method parameter that expects the ResourceLoader type as long as the field, constructor, or method in question carries the @Autowired annotation. For more information, see [Using @Autowired](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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. For example, consider 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, have the bean expose the Resource properties it needs, and expect them to 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 the following example shows:

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

<property name="template" value="some/resource/path/myTemplate.txt"/>

</bean>

Note that the resource path has no prefix. Consequently, because the application context itself is going to be used as the ResourceLoader, the resource itself is loaded through a ClassPathResource, a FileSystemResource, or a ServletContextResource, depending on the exact type of the context.

If you need to force a specific Resource type to be used, you can use a prefix. 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

This section covers how to create application contexts with resources, including shortcuts that work with XML, how to use wildcards, and other details.

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 paths of the resources, such as XML files that make up the definition of the context.

When such a location path does not 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, consider the following example, which creates a ClassPathXmlApplicationContext:

ApplicationContext ctx = **new** ClassPathXmlApplicationContext("conf/appContext.xml");

The bean definitions are loaded from the classpath, because a ClassPathResource is used. However, consider the following example, which creates a FileSystemXmlApplicationContext:

ApplicationContext ctx =

**new** FileSystemXmlApplicationContext("conf/appContext.xml");

Now the bean definition is 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 overrides the default type of Resource created to load the definition. Consider the following example:

ApplicationContext ctx =

**new** FileSystemXmlApplicationContext("classpath:conf/appContext.xml");

Using FileSystemXmlApplicationContext loads the bean definitions from the classpath. However, it is still aFileSystemXmlApplicationContext. If it is subsequently used as a ResourceLoader, any unprefixed paths are still treated as filesystem paths.

Constructing ClassPathXmlApplicationContext Instances — Shortcuts

The ClassPathXmlApplicationContext exposes a number of constructors to enable convenient instantiation. The basic idea is that you can supply merely a string array that contains only the filenames of the XML files themselves (without the leading path information) and also supplies a Class. The ClassPathXmlApplicationContext then derives the path information from the supplied class.

Consider the following directory layout:

com/

foo/

services.xml

daos.xml

MessengerService.class

The following example shows how a ClassPathXmlApplicationContext instance composed of the beans defined in files named services.xml and daos.xml (which are on the classpath) can be instantiated:

ApplicationContext ctx = **new** ClassPathXmlApplicationContext(

**new** String**[]** {"services.xml", "daos.xml"}, MessengerService.class);

See the [ClassPathXmlApplicationContext](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/jca/context/SpringContextResourceAdapter.html) javadoc 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 simple paths (as shown earlier), each of which has a one-to-one mapping to a target Resource or, alternately, may contain the special "classpath\*:" prefix or internal Ant-style regular expressions (matched by using Spring’s PathMatcher utility). Both of the latter are effectively wildcards.

One use for this mechanism is when you need to do 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 with classpath\*:, all component fragments are automatically picked up.

Note that this wildcarding is specific to the use of resource paths in application context constructors (or when you use the PathMatcher utility class hierarchy directly) and is resolved at construction time. It has nothing to do with the Resource type itself. You cannot use the classpath\*: prefix to construct an actual Resource, as a resource points to just one resource at a time.

Ant-style Patterns

Path locations can contain Ant-style patterns, as the following example shows:

/WEB-INF/\*-context.xml

com/mycompany/\*\*/applicationContext.xml

file:C:/some/path/\*-context.xml

classpath:com/mycompany/\*\*/applicationContext.xml

When the path location contains an Ant-style pattern, the resolver follows a more complex 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 (such as zip: in WebLogic, wsjar in WebSphere, and so on), 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 implicitly because the base ResourceLoader is a filesystem one or explicitly), wildcarding is guaranteed to work in a completely portable fashion.

If the specified path is a classpath location, the resolver must obtain the last non-wildcard path segment URL by making 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 javadoc) 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 does work in most environments but fails in others, and we strongly recommend 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, as the following example shows:

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 through a call to ClassLoader.getResources(…​)) 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. |

You can also combine the classpath\*: prefix 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 earlier is used for the wildcard subpath.

Other Notes Relating to Wildcards

Note that classpath\*:, when combined with Ant-style patterns, only works 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 such as classpath\*:\*.xml might 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 an empty string (indicating potential roots to search). Spring evaluates URLClassLoader runtime 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, 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 — for example, stand-alone applications on JDK 1.7.0\_45 and higher (which requires 'Trusted-Library' to be set up in your manifests. See<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. Consider the following example of a resource location:

com/mycompany/package1/service-context.xml

Now consider an Ant-style path that someone might use to try to find that file:

classpath:com/mycompany/\*\*/service-context.xml

Such a resource may be in only one location, but when a path such as the preceding example is used to try to resolve it, the resolver works 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 there. Therefore, in such a case you should prefer using classpath\*:with the same Ant-style pattern, which searches 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, when a FileSystemApplicationContextis not the actual ResourceLoader) treats absolute and 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 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 examples are equivalent:

ApplicationContext ctx =

**new** FileSystemXmlApplicationContext("conf/context.xml");

ApplicationContext ctx =

**new** FileSystemXmlApplicationContext("/conf/context.xml");

The following examples are also equivalent (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 you need true absolute filesystem paths, you should avoid using absolute paths with FileSystemResource or FileSystemXmlApplicationContext and force the use of a UrlResource by using the file: URL prefix. The following examples show how to do so:

*// 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

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 and should be easy to localize, and it should be possible to plug in any available validator. Considering these concerns, Spring has come up with a Validator interface that is both basic and eminently usable in every layer of an application.

Data binding is useful for letting user input be dynamically bound to the domain model of an application (or whatever objects you use to process user input). Spring provides the aptly named 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 do not need to use the BeanWrapper directly. Because this is reference documentation, however, we felt that some explanation might be in order. We explain the BeanWrapper in this chapter, since, if you are going to use it at all, you are most likely do so when trying to bind data to objects.

Spring’s DataBinder and the lower-level BeanWrapper both use PropertyEditorSupport implementations to parse and format property values. The PropertyEditor and PropertyEditorSupport types are part of the JavaBeans specification and are also explained in this chapter. Spring 3 introduced a core.convert package that provides a general type conversion facility, as well as a higher-level “format” package for formatting UI field values. You can use these packages as simpler alternatives toPropertyEditorSupport implementations. They are also discussed in this chapter.

JSR-303/JSR-349 Bean Validation

As of version 4.0, Spring Framework supports Bean Validation 1.0 (JSR-303) and Bean Validation 1.1 (JSR-349) for setup support and adapting them 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/5.1.8.RELEASE/spring-framework-reference/core.html#validation-beanvalidation), and use it exclusively for all validation needs.

An application can also register additional Spring Validator instances for each DataBinder instance, as described in [Configuring a DataBinder](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#validation-binder). This may be useful for plugging in validation logic without the use of annotations.

3.1. Validation by Using Spring’s Validator Interface

Spring features a Validator interface that you can use to validate objects. The Validator interface works by using an Errorsobject so that, while validating, validators can report validation failures to the Errors object.

Consider the following example of a small data object:

**public** **class** **Person** {

**private** String name;

**private** **int** age;

*// the usual getters and setters...*

}

The next example provides 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. The following example implements Validator for Person instances:

**public** **class** **PersonValidator** **implements** Validator {

*/\*\**

*\* This Validator validates \*only\* Person instances*

*\*/*

**public** **boolean** supports(Class clazz) {

**return** Person.class.equals(clazz);

}

**public** **void** validate(Object obj, Errors e) {

ValidationUtils.rejectIfEmpty(e, "name", "name.empty");

Person p = (Person) obj;

**if** (p.getAge() < 0) {

e.rejectValue("age", "negativevalue");

} **else** **if** (p.getAge() > 110) {

e.rejectValue("age", "too.darn.old");

}

}

}

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](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/validation/ValidationUtils.html) javadoc 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 a second name) and a complex Address object. Address objects may be used independently of Customer objects, so a distinct AddressValidator has 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, as the following example shows:

**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 the case of Spring Web MVC, you can use the <spring:bind/> tag to inspect the error messages, but you can also inspect the Errors object yourself. More information about the methods it offers can be found in the [javadoc](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframeworkvalidation/Errors.html).

3.2. Resolving Codes to Error Messages

We covered databinding and validation. This section covers outputting messages that correspond to validation errors. In the example shown in the [preceding section](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#validator), we rejected the name and age fields. If we want to output the error messages by using a MessageSource, we can do so using the error code we provide when rejecting the field ('name' and 'age' in this case). When you call (either directly, or indirectly, by using, for example, the ValidationUtils class) rejectValue or one of the other rejectmethods from the Errors interface, the underlying implementation not only registers the code you passed in but also registers a number of additional error codes. The MessageCodesResolver determines which error codes the Errors interface registers. By default, the DefaultMessageCodesResolver is used, which (for example) not only registers a message with the code you gave but also registers messages that include the field name you passed to the reject method. So, if you reject a field by using rejectValue("age", "too.darn.old"), apart from the too.darn.old code, Spring also registers too.darn.old.age andtoo.darn.old.age.int (the first includes the field name and the second includes the type of the field). This is done as a convenience to aid developers when targeting error messages.

More information on the MessageCodesResolver and the default strategy can be found in the javadoc of [MessageCodesResolver](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/validation/MessageCodesResolver.html)and [DefaultMessageCodesResolver](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/validation/DefaultMessageCodesResolver.html), respectively.

3.3. Bean Manipulation and the BeanWrapper

The org.springframework.beans package adheres to the JavaBeans standard. A JavaBean is a class with a default no-argument constructor and that follows a naming convention where (for example) a property named bingoMadness would have a setter method setBingoMadness(..) and a getter method getBingoMadness(). For more information about JavaBeans and the specification, see [javabeans](https://docs.oracle.com/javase/8/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 javadoc, the BeanWrapper offers functionality to set and get property values (individually or in bulk), get property descriptors, and query properties to determine if they are readable or writable. Also, the BeanWrapperoffers support for nested properties, enabling the setting of properties on sub-properties to an unlimited depth. TheBeanWrapper also 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 BeanWrapper provides support for setting indexed properties. The BeanWrapper usually is not used by application code directly but is used 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, such as setting and retrieving properties.

3.3.1. Setting and Getting Basic and Nested Properties

Setting and getting properties is done by using the setPropertyValue, setPropertyValues, getPropertyValue, and getPropertyValues methods which come with a couple of overloaded variants. Springs javadoc describes them in more detail. The JavaBeans specification has conventions for indicating properties of an object. The following table shows some examples of these conventions:

| *Table 11. Examples of properties* | |
| --- | --- |
| **Expression** | **Explanation** |
| name | Indicates the property name that corresponds to the getName() or isName() and setName(..) methods. |
| account.name | Indicates the nested property name of the property accountthat corresponds to (for example) the getAccount().setName() or getAccount().getName()methods. |
| 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 COMPANYNAME key of the account Map property. |

(This next section is not vitally important to you if you do not plan to work with the BeanWrapper directly. If you use only the DataBinder and the BeanFactory and their default implementations, you should skip ahead to the [section on PropertyEditors](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-beans-conversion).)

The following two example classes use the BeanWrapper to get and set properties:

**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");

3.3.2. Built-in PropertyEditor Implementations

Spring uses the concept of a PropertyEditor to effect the conversion between an Object and a String. It can be handy 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 can still 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, alternatively, in a specific IoC container (as mentioned in the previous chapter), gives it the knowledge of how to convert properties to the desired type. For more about PropertyEditor, see [the javadoc of the java.beans package from Oracle](https://docs.oracle.com/javase/8/docs/api/java/beans/package-summary.html).

A couple of examples where property editing is used in Spring:

* Setting properties on beans is done by using PropertyEditor implementations. When you use String as the value of a property of some bean that you declare in an XML file, Spring (if the setter of the corresponding property has a Classparameter) uses ClassEditor to try to resolve the parameter to a Class object.
* Parsing HTTP request parameters in Spring’s MVC framework is done by using all kinds of PropertyEditor implementations that you can manually bind in all subclasses of the CommandController.

Spring has a number of built-in PropertyEditor implementations to make life easy. They are all located in the org.springframework.beans.propertyeditors package. Most, (but not all, as indicated in the following table) are, by default, registered by BeanWrapperImpl. Where the property editor is configurable in some fashion, you can still register your own variant to override the default one. The following table describes the various PropertyEditor implementations that Spring provides:

| *Table 12. Built-in PropertyEditor Implementations* | |
| --- | --- |
| **Class** | **Explanation** |
| ByteArrayPropertyEditor | Editor for byte arrays. Converts strings to their corresponding byte representations. Registered by default by BeanWrapperImpl. |
| ClassEditor | Parses Strings that represent classes to actual classes and vice-versa. When a class is not found, an IllegalArgumentException is thrown. By default, registered byBeanWrapperImpl. |
| CustomBooleanEditor | Customizable property editor for Boolean properties. By default, registered byBeanWrapperImpl but can be overridden by registering a custom instance of it as a custom editor. |
| CustomCollectionEditor | Property editor for collections, converting any source Collection to a given targetCollection type. |
| CustomDateEditor | Customizable property editor for java.util.Date, supporting a custom DateFormat. NOT registered by default. Must be user-registered with the appropriate format as needed. |
| CustomNumberEditor | Customizable property editor for any Number subclass, such as Integer, Long, Float, or Double. By default, registered by BeanWrapperImpl but can be overridden by registering a custom instance of it as a custom editor. |
| FileEditor | Resolves strings to java.io.File objects. By default, registered by BeanWrapperImpl. |
| InputStreamEditor | One-way property editor that can take a string and produce (through an intermediate ResourceEditor and Resource) an InputStream so that InputStream properties may be directly set as strings. Note that the default usage does not close the InputStreamfor you. By default, registered by BeanWrapperImpl. |
| LocaleEditor | Can resolve strings to Locale objects and vice-versa (the string format is *[country]*[variant], same as the toString() method of Locale). By default, registered by BeanWrapperImpl. |
| PatternEditor | Can resolve strings to java.util.regex.Pattern objects and vice-versa. |
| PropertiesEditor | Can convert strings (formatted with the format defined in the javadoc of thejava.util.Properties class) to Properties objects. By default, registered 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. |
| URLEditor | Can resolve a string representation of a URL to an actual URL object. By default, registered 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 automatically discovers PropertyEditorclasses (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 SomethingEditor class to be recognized and used as the PropertyEditor for Something-typed properties.

com

chank

pop

Something

SomethingEditor // the PropertyEditor for the Something class

Note that you can also use the standard BeanInfo JavaBeans mechanism here as well (described to some extent [here](https://docs.oracle.com/javase/tutorial/javabeans/advanced/customization.html)). The following example use the BeanInfo mechanism to explicitly register one or more PropertyEditor instances with the properties of an associated class:

com

chank

pop

Something

SomethingBeanInfo // the BeanInfo for the Something class

The following Java source code for the referenced SomethingBeanInfo class associates a CustomNumberEditor with the ageproperty of the Something class:

**public** **class** **SomethingBeanInfo** **extends** SimpleBeanInfo {

**public** PropertyDescriptor**[]** getPropertyDescriptors() {

**try** {

**final** PropertyEditor numberPE = **new** CustomNumberEditor(Integer.class, true);

PropertyDescriptor ageDescriptor = **new** PropertyDescriptor("age", Something.class) {

**public** PropertyEditor createPropertyEditor(Object bean) {

**return** numberPE;

};

};

**return** **new** PropertyDescriptor**[]** { ageDescriptor };

}

**catch** (IntrospectionException ex) {

**throw** **new** Error(ex.toString());

}

}

}

Registering Additional Custom PropertyEditor Implementations

When setting bean properties as string values, a Spring IoC container ultimately uses standard JavaBeans PropertyEditorimplementations to convert these strings to the complex type of the property. Spring pre-registers a number of custom PropertyEditor implementations (for example, to convert a class name expressed as a string into a Class object). Additionally, Java’s standard JavaBeans PropertyEditor lookup mechanism lets a PropertyEditor for a class be named appropriately and placed in the same package as the class for which it provides support, so that it can be found automatically.

If there is a need to register other custom PropertyEditors, several mechanisms are available. The most manual approach, which is not normally convenient or recommended, is to 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 you can use bean factory post-processors with BeanFactory implementations, the CustomEditorConfigurer has a nested property setup, so we strongly recommend that you use it with the ApplicationContext, where you can deploy it in similar fashion to any other bean and where it can be automatically detected and applied.

Note that all bean factories and application contexts automatically use a number of built-in property editors, through their use a BeanWrapper to handle property conversions. The standard property editors that the BeanWrapper registers are listed in the [previous section](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-beans-conversion). Additionally, ApplicationContexts also override or add additional 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. You can use CustomEditorConfigurer, a bean factory post-processor, to conveniently add support for additional PropertyEditor instances to an ApplicationContext.

Consider the following example, which defines a user class called ExoticType and another class called 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 converts into an actual ExoticType instance. The following bean definition shows how to set up this relationship:

<bean id="sample" class="example.DependsOnExoticType">

<property name="type" value="aNameForExoticType"/>

</bean>

The PropertyEditor implementation could look similar to the following:

*// converts string representation to ExoticType object*

**package** example;

**public** **class** **ExoticTypeEditor** **extends** PropertyEditorSupport {

**public** **void** setAsText(String text) {

setValue(**new** ExoticType(text.toUpperCase()));

}

}

Finally, the following example shows how to use CustomEditorConfigurer to register the new PropertyEditor with theApplicationContext, 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 PropertyEditorRegistrar

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. You can write a corresponding registrar and reuse it in each case. PropertyEditorRegistrar instances work in conjunction with an interface called PropertyEditorRegistry, an interface that is implemented by the Spring BeanWrapper (and DataBinder). PropertyEditorRegistrar instances are particularly convenient when used in conjunction with CustomEditorConfigurer(described [here](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-beans-conversion-customeditor-registration)), which exposes a property called setPropertyEditorRegistrars(..). PropertyEditorRegistrar instances added to a CustomEditorConfigurer in this fashion can easily be shared with DataBinder and Spring MVC controllers. Furthermore, it avoids the need for synchronization on custom editors: A PropertyEditorRegistrar is expected to create fresh PropertyEditor instances for each bean creation attempt.

The following example shows how 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.

The next example shows how to configure a CustomEditorConfigurer and inject an instance of ourCustomPropertyEditorRegistrar 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/5.1.8.RELEASE/spring-framework-reference/web.html#mvc)), using PropertyEditorRegistrars in conjunction with data-binding Controllers (such as SimpleFormController) can be very convenient. The following example uses 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 only one line long) and lets common PropertyEditor registration code be encapsulated in a class and then shared amongst as many Controllersas needed.

3.4. Spring Type Conversion

Spring 3 introduced a core.convert package that provides a general type conversion system. The system defines an SPI to implement type conversion logic and an API to perform type conversions at runtime. Within a Spring container, you can use this system as an alternative to PropertyEditor implementations to convert externalized bean property value strings to the required property types. You can also use the public API anywhere in your application where type conversion is needed.

3.4.1. Converter SPI

The SPI to implement type conversion logic is simple and strongly typed, as the following interface definition shows:

**package** org.springframework.core.convert.converter;

**public** **interface** **Converter**<S, T> {

T convert(S source);

}

To create your own converter, implement the Converter interface and parameterize S as the type you are converting from and T as the type you are converting to. You can also transparently apply such a converter if a collection or array of S needs to be converted to an array or collection of T, provided that a delegating array or collection converter has been registered as well (which DefaultConversionService does by default).

For each call to convert(S), the source argument is guaranteed to not be null. Your Converter may throw any unchecked exception if conversion fails. Specifically, it should throw an IllegalArgumentException 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. The following listing shows the StringToInteger class, which is a typical Converter implementation:

**package** org.springframework.core.convert.support;

**final** **class** **StringToInteger** **implements** Converter<String, Integer> {

**public** Integer convert(String source) {

**return** Integer.valueOf(source);

}

}

3.4.2. Using ConverterFactory

When you need to centralize the conversion logic for an entire class hierarchy (for example, when converting from String to Enum objects), you can implement ConverterFactory, as the following example shows:

**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 StringToEnumConverterFactory 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());

}

}

}

3.4.3. Using GenericConverter

When you require a sophisticated Converter implementation, consider using the GenericConverter interface. With a more flexible but less strongly typed signature than Converter, a GenericConverter supports converting between multiple source and target types. In addition, a GenericConverter makes available source and target field context that you can use when you implement your conversion logic. Such context lets a type conversion be driven by a field annotation or by generic information declared on a field signature. The following listing shows the interface definition of GenericConverter:

**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 contain your conversion logic. The source TypeDescriptorprovides access to the source field that holds the value being converted. The target TypeDescriptor provides access to the target field where the converted value is to 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 lets each element in the source array be converted to the collection element type before the collection is set on the target field.

|  |  |
| --- | --- |
|  | Because GenericConverter is a more complex SPI interface, you should use it only when you need it. Favor Converter or ConverterFactory for basic type conversion needs. |

Using ConditionalGenericConverter

Sometimes, you want a Converter to run only if a specific condition holds true. For example, you might want to run a Converteronly if a specific annotation is present on the target field, or you might want to run a Converter only if a specific method (such as a static valueOf method) is defined on the target class. ConditionalGenericConverter is the union of the GenericConverterand ConditionalConverter interfaces that lets you 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 a persistent entity identifier and an entity reference. Such an EntityConverter might match only if the target entity type declares a static finder method (for example, findAccount(Long)). You might perform such a finder method check in the implementation ofmatches(TypeDescriptor, TypeDescriptor).

3.4.4. The ConversionService API

ConversionService defines a unified API for executing type conversion logic at runtime. Converters are often executed behind the following 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.

3.4.5. Configuring a ConversionService

A ConversionService is a stateless object designed to be instantiated at application startup and then shared between multiple threads. In a Spring application, you typically configure a ConversionService instance for each Spring container (or ApplicationContext). Spring picks up that ConversionService and uses it whenever a type conversion needs to be performed by the framework. You can 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 an id of 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 converters, set the converters property. Property values can implement any 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/5.1.8.RELEASE/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 [The FormatterRegistry SPI](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#format-FormatterRegistry-SPI) for details on using FormattingConversionServiceFactoryBean.

3.4.6. Using a ConversionService Programmatically

To work with a ConversionService instance programmatically, you can inject a reference to it like you would for any other bean. The following example shows how to do so:

@Service

**public** **class** **MyService** {

@Autowired

**public** MyService(ConversionService conversionService) {

this.conversionService = conversionService;

}

**public** **void** doIt() {

this.conversionService.convert(...)

}

}

For most use cases, you can use the convert method that specifies the targetType, but it does not work with more complex types, such as a collection of a parameterized element. For example, if you want to convert a List of Integer to a List of String programmatically, you need to provide a formal definition of the source and target types.

Fortunately, TypeDescriptor provides various options to make doing so straightforward, as the following example shows:

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 automatically registers converters that are appropriate for most environments. This includes collection converters, scalar converters, and basic Object-to-String converters. You can register the same converters with any ConverterRegistry by using the static addDefaultConverters method on the DefaultConversionService class.

Converters for value types are 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.

3.5. Spring Field Formatting

As discussed in the previous section, [core.convert](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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 ConverterSPI does not address such formatting requirements directly. To directly address them, Spring 3 introduced a convenient Formatter SPI that provides a simple and robust alternative to PropertyEditor implementations for client environments.

In general, you can use the Converter SPI when you need to implement general-purpose type conversion logic — for example, for converting between a java.util.Date and a Long. You can use the Formatter SPI when you work 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.

3.5.1. The Formatter SPI

The Formatter SPI to implement field formatting logic is simple and strongly typed. The following listing shows the Formatterinterface definition:

**package** org.springframework.format;

**public** **interface** **Formatter**<T> **extends** Printer<T>, Parser<T> {

}

Formatter extends from the Printer and Parser building-block interfaces. The following listing shows the definitions of those two 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, implement the Formatter interface shown earlier. 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 an IllegalArgumentException if a parse attempt fails. Take care to ensure that your Formatter implementation is thread-safe.

The format subpackages provide several Formatter implementations as a convenience. The number package provides NumberStyleFormatter, CurrencyStyleFormatter, and PercentStyleFormatter to format Number objects that use 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/).

The following DateFormatter is 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 [GitHub Issues](https://github.com/spring-projects/spring-framework/issues) to contribute.

3.5.2. Annotation-driven Formatting

Field formatting can be configured by field type or annotation. To bind an annotation to a Formatter, implement AnnotationFormatterFactory. The following listing shows the definition of the AnnotationFormatterFactory interface:

**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);

}

To create an implementation: . Parameterize A to be the field annotationType with which you wish to associate formatting logic — for example org.springframework.format.annotation.DateTimeFormat. . Have getFieldTypes() return the types of fields on which the annotation can be used. . 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 following example AnnotationFormatterFactory implementation binds the @NumberFormat annotation to a formatter to let a number style or pattern 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** NumberStyleFormatter(annotation.pattern());

} **else** {

Style style = annotation.style();

**if** (style == Style.PERCENT) {

**return** **new** PercentStyleFormatter();

} **else** **if** (style == Style.CURRENCY) {

**return** **new** CurrencyStyleFormatter();

} **else** {

**return** **new** NumberStyleFormatter();

}

}

}

}

To trigger formatting, you can annotate fields with @NumberFormat, as the following example shows:

**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. You can use @NumberFormatto format Number fields such as Double and Long, and @DateTimeFormat to format java.util.Date, java.util.Calendar, Long(for millisecond timestamps) as well as JSR-310 java.time and Joda-Time value types.

The following example uses @DateTimeFormat to format a java.util.Date as an ISO Date (yyyy-MM-dd):

**public** **class** **MyModel** {

@DateTimeFormat(iso=ISO.DATE)

**private** Date date;

}

3.5.3. The FormatterRegistry SPI

The FormatterRegistry is an SPI for registering formatters and converters. FormattingConversionService is an implementation of FormatterRegistry suitable for most environments. You can programmatically or declaratively configure this variant as a Spring bean, e.g. by using FormattingConversionServiceFactoryBean. Because this implementation also implements ConversionService, you can directly configure it for use with Spring’s DataBinder and the Spring Expression Language (SpEL).

The following listing shows the FormatterRegistry SPI:

**package** org.springframework.format;

**public** **interface** **FormatterRegistry** **extends** ConverterRegistry {

**void** addFormatterForFieldType(Class<?> fieldType, Printer<?> printer, Parser<?> parser);

**void** addFormatterForFieldType(Class<?> fieldType, Formatter<?> formatter);

**void** addFormatterForFieldType(Formatter<?> formatter);

**void** addFormatterForAnnotation(AnnotationFormatterFactory<?, ?> factory);

}

As shown in the preceding listing, you can register formatters by field type or by annotation.

The FormatterRegistry SPI lets you 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 that 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.

3.5.4. The FormatterRegistrar SPI

FormatterRegistrar is an SPI for registering formatters and converters through the FormatterRegistry. The following listing shows its interface definition:

**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.

3.5.5. Configuring Formatting in Spring MVC

See [Conversion and Formatting](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/web.html#mvc-config-conversion) in the Spring MVC chapter.

3.6. Configuring a Global Date and Time Format

By default, date and time fields that are not annotated with @DateTimeFormat are converted from strings by using the DateFormat.SHORT style. If you prefer, you can change this by defining your own global format.

To do so, you need to ensure that Spring does not register default formatters. 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 registers 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. The following example shows how to do so (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

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

<bean id="conversionService" class="org.springframework.format.support.FormattingConversionServiceFactoryBean">

<property name="registerDefaultFormatters" value="false" />

<property name="formatters">

<set>

<bean class="org.springframework.format.number.NumberFormatAnnotationFormatterFactory" />

</set>

</property>

<property name="formatterRegistrars">

<set>

<bean class="org.springframework.format.datetime.joda.JodaTimeFormatterRegistrar">

<property name="dateFormatter">

<bean class="org.springframework.format.datetime.joda.DateTimeFormatterFactoryBean">

<property name="pattern" value="yyyyMMdd"/>

</bean>

</property>

</bean>

</set>

</property>

</bean>

</beans>

|  |  |
| --- | --- |
|  | Joda-Time 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 use 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 themvc:annotation-driven element. See [Conversion and Formatting](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/web.html#mvc-config-conversion) for details. |

3.7. Spring Validation

Spring 3 introduced several enhancements to its validation support. First, the JSR-303 Bean Validation API is fully supported. Second, when used programmatically, Spring’s DataBinder can validate objects as well as bind to them. Third, Spring MVC has support for declaratively validating @Controller inputs.

3.7.1. Overview of the JSR-303 Bean Validation API

JSR-303 standardizes validation constraint declaration and metadata for the Java platform. By using this API, you annotate domain model properties with declarative validation constraints and the runtime enforces them. You can use a number of built-in constraints. You can also define your own custom constraints.

Consider the following example, which shows a simple PersonForm model with two properties:

**public** **class** **PersonForm** {

**private** String name;

**private** **int** age;

}

JSR-303 lets you define declarative validation constraints against such properties, as the following example shows:

**public** **class** **PersonForm** {

@NotNull

@Size(max=64)

**private** String name;

@Min(0)

**private** **int** age;

}

When a JSR-303 Validator validates an instance of this class, these constraints are enforced.

For general information on JSR-303 and JSR-349, see the [Bean Validation website](https://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 set up a bean validation provider as a Spring bean, keep reading.

3.7.2. Configuring a Bean Validation Provider

Spring provides full support for the Bean Validation API. This includes convenient support for bootstrapping a JSR-303 or JSR-349 Bean Validation provider as a Spring bean. This lets you inject a javax.validation.ValidatorFactory or javax.validation.Validator wherever validation is needed in your application.

You can use the LocalValidatorFactoryBean to configure a default Validator as a Spring bean, as the following example shows:

<bean id="validator"

class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"/>

The basic configuration in the preceding example triggers bean validation to initialize by using its default bootstrap mechanism. A JSR-303 or JSR-349 provider, such as the Hibernate Validator, is expected to be present in the classpath and is automatically detected.

Injecting a Validator

LocalValidatorFactoryBean implements both javax.validation.ValidatorFactory and javax.validation.Validator, as well as Spring’s org.springframework.validation.Validator. You can inject a reference to either of these interfaces into beans that need to invoke validation logic.

You can inject a reference to javax.validation.Validator if you prefer to work with the Bean Validation API directly, as the following example shows:

**import** javax.validation.Validator;

@Service

**public** **class** **MyService** {

@Autowired

**private** Validator validator;

You can inject a reference to org.springframework.validation.Validator if your bean requires the Spring Validation API, as the following example shows:

**import** org.springframework.validation.Validator;

@Service

**public** **class** **MyService** {

@Autowired

**private** Validator validator;

}

Configuring Custom Constraints

Each bean validation constraint consists of two parts: \* A @Constraint annotation that declares the constraint and its configurable properties. \* 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 ConstraintValidator 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 lets your custom ConstraintValidators benefit from dependency injection like any other Spring bean.

The following example shows 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 the preceding example shows, a ConstraintValidator implementation can have its dependencies @Autowired as any other Spring bean.

Spring-driven Method Validation

You can integrate the method validation feature supported by Bean Validation 1.1 (and, as a custom extension, also by Hibernate Validator 4.3) into a Spring context through a MethodValidationPostProcessor bean definition, as follows:

<bean class="org.springframework.validation.beanvalidation.MethodValidationPostProcessor"/>

To be eligible for Spring-driven method validation, all target classes need to be annotated with Spring’s @Validated annotation. (Optionally, you can also declare the validation groups to use.) See the [MethodValidationPostProcessor](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/validation/beanvalidation/MethodValidationPostProcessor.html) javadoc for setup details with the Hibernate Validator and Bean Validation 1.1 providers.

Additional Configuration Options

The default LocalValidatorFactoryBean configuration suffices for most cases. There are a number of configuration options for various Bean Validation constructs, from message interpolation to traversal resolution. See the [LocalValidatorFactoryBean](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/validation/beanvalidation/LocalValidatorFactoryBean.html)javadoc for more information on these options.

3.7.3. Configuring a DataBinder

Since Spring 3, you can configure a DataBinder instance with a Validator. Once configured, you can invoke the Validator by calling binder.validate(). Any validation Errors are automatically added to the binder’s BindingResult.

The following example shows how to use a DataBinder programmatically 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();

You can also configure a DataBinder with multiple Validator instances through 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/5.1.8.RELEASE/spring-framework-reference/core.html#validation-mvc-configuring).

3.7.4. Spring MVC 3 Validation

See [Validation](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/web.html#mvc-config-validation) in the Spring MVC chapter.

4. Spring Expression Language (SpEL)

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 that lets other expression language implementations 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. 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 need not deal with this infrastructure and can, instead, author only expression strings for evaluation. An example of this typical use is the integration of SpEL into creating XML or annotation-based bean definitions, as shown in [Expression support for defining bean definitions](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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, Inventor and Society classes 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.

The expression language supports the following functionality:

* Literal expressions
* Boolean and relational operators
* Regular expressions
* Class expressions
* Accessing properties, arrays, lists, and maps
* Method invocation
* Relational operators
* Assignment
* Calling constructors
* Bean references
* Array construction
* Inline lists
* Inline maps
* Ternary operator
* Variables
* User-defined functions
* Collection projection
* Collection selection
* Templated expressions

4.1. Evaluation

This section introduces the simple use of SpEL interfaces and its expression language. The complete language reference can be found in [Language Reference](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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 'Hello World'. |

The SpEL classes and interfaces you are most likely to use are located in the org.springframework.expression package and its sub-packages, such as spel.support.

The ExpressionParser interface is responsible for parsing an expression string. In the preceding example, the expression string is a string literal denoted by the surrounding single quotation marks. The Expression interface is responsible for evaluating the previously defined expression string. 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.

In the following 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!'. |

The following example of calling a JavaBean property calls the String property Bytes:

ExpressionParser parser = **new** SpelExpressionParser();

*// invokes 'getBytes()'*

Expression exp = parser.parseExpression("'Hello World'.bytes");

**byte[]** bytes = (**byte[]**) exp.getValue();

|  |  |
| --- | --- |
|  | This line converts the literal to a byte array. |

SpEL also supports nested properties by using the standard dot notation (such as prop1.prop2.prop3) and also the corresponding setting of property values. Public fields may also be accessed.

The following example shows how to use dot notation to get the length of a literal:

ExpressionParser parser = **new** SpelExpressionParser();

*// invokes 'getBytes().length'*

Expression exp = parser.parseExpression("'Hello World'.bytes.length");

**int** length = (Integer) exp.getValue();

|  |  |
| --- | --- |
|  | 'Hello World'.bytes.length gives the length of the literal. |

The String’s constructor can be called instead of using a string literal, as the following example shows:

ExpressionParser parser = **new** SpelExpressionParser();

Expression exp = parser.parseExpression("new String('hello world').toUpperCase()");

String message = exp.getValue(String.class);

|  |  |
| --- | --- |
|  | Construct a new String from the literal and make it be upper case. |

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 is thrown if the value cannot be cast to the type T or converted by 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). The following example shows how to retrieve the name property from an instance of the Inventor class or create a boolean condition:

*// Create and set a calendar*

GregorianCalendar c = **new** GregorianCalendar();

c.set(1856, 7, 9);

*// The constructor arguments are name, birthday, and nationality.*

Inventor tesla = **new** Inventor("Nikola Tesla", c.getTime(), "Serbian");

ExpressionParser parser = **new** SpelExpressionParser();

Expression exp = parser.parseExpression("name");

String name = (String) exp.getValue(tesla);

*// name == "Nikola Tesla"*

exp = parser.parseExpression("name == 'Nikola Tesla'");

**boolean** result = exp.getValue(tesla, Boolean.class);

*// result == true*

|  |  |
| --- | --- |
|  | Parse name as an expression. |

4.1.1. Understanding EvaluationContext

The EvaluationContext interface is used when evaluating an expression to resolve properties, methods, or fields and to help perform type conversion. Spring provides two implementations.

* SimpleEvaluationContext: Exposes a subset of essential SpEL language features and configuration options, for categories of expressions that do not require the full extent of the SpEL language syntax and should be meaningfully restricted. Examples include but are not limited to data binding expressions and property-based filters.
* StandardEvaluationContext: Exposes the full set of SpEL language features and configuration options. You can use it to specify a default root object and to configure every available evaluation-related strategy.

SimpleEvaluationContext is designed to support only a subset of the SpEL language syntax. It excludes Java type references, constructors, and bean references. It also requires you to explicitly choose the level of support for properties and methods in expressions. By default, the create() static factory method enables only read access to properties. You can also obtain a builder to configure the exact level of support needed, targeting one or some combination of the following:

* Custom PropertyAccessor only (no reflection)
* Data binding properties for read-only access
* Data binding properties for read and write

Type Conversion

By default, SpEL uses the conversion service available in Spring core (org.springframework.core.convert.ConversionService). This conversion service comes with many built-in converters for common conversions but is also fully extensible so that you can add custom conversions between types. Additionally, it is generics-aware. This means that, when you work with generic types in expressions, SpEL attempts 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 recognizes that the elements of the list need to be converted to Boolean before being placed in it. The following example shows how to do so:

**class** **Simple** {

**public** List<Boolean> booleanList = **new** ArrayList<Boolean>();

}

Simple simple = **new** Simple();

simple.booleanList.add(true);

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();

*// "false" is passed in here as a String. SpEL and the conversion service*

*// will recognize that it needs to be a Boolean and convert it accordingly.*

parser.parseExpression("booleanList[0]").setValue(context, simple, "false");

*// b is false*

Boolean b = simple.booleanList.get(0);

4.1.2. Parser Configuration

It is possible to configure the SpEL expression parser by 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 you index into an array or collection and the element at the specified index is null, you can automatically create the element. This is useful when using expressions made up of a chain of property references. If you index into an array or list and specifying an index that is beyond the end of the current size of the array or list, you can automatically grow the array or list to accommodate that index. The following example demonstrates how to automatically grow the list:

**class** **Demo** {

**public** List<String> list;

}

*// Turn on:*

*// - auto null reference initialization*

*// - auto collection growing*

SpelParserConfiguration config = **new** SpelParserConfiguration(true,true);

ExpressionParser parser = **new** SpelExpressionParser(config);

Expression expression = parser.parseExpression("list[3]");

Demo demo = **new** Demo();

Object o = expression.getValue(demo);

*// demo.list will now be a real collection of 4 entries*

*// Each entry is a new empty String*

4.1.3. SpEL 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 optimum performance. For occasional expression usage, this is fine, but, when used by other components such as Spring Integration, performance can be very important, and there is no real need for the dynamism.

The SpEL compiler is intended to address this need. During evaluation, the compiler generates a Java class that embodies the expression behavior at runtime and uses that class 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 finds out what it is. Of course, basing compilation on such derived information can 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.

Consider the following basic expression:

someArray[0].someProperty.someOtherProperty < 0.1

Because the preceding expression involves array access, some property de-referencing, and numeric operations, the performance gain can be very noticeable. In an example micro benchmark run of 50000 iterations, it took 75ms to evaluate by using the interpreter and only 3ms using the compiled version of the expression.

Compiler Configuration

The compiler is not turned on by default, but you can turn it on in either of two different ways. You can turn it on by using the parser configuration process ([discussed earlier](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#expressions-parser-configuration)) or by using a system property when SpEL usage is embedded inside another component. This section discusses both of these options.

The compiler can operate in one of three modes, which are captured in theorg.springframework.expression.spel.SpelCompilerMode enum. The modes are as follows:

* OFF (default): The compiler is switched off.
* 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 earlier), the caller of the expression evaluation receives an exception.
* MIXED: In mixed mode, the expressions silently switch between interpreted and compiled mode over time. After some number of interpreted runs, they switch to compiled form and, if something goes wrong with the compiled form (such as a type changing, as described earlier), the expression automatically switches 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. The following example shows how to do so:

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 you specify the compiler mode, you can also specify a classloader (passing null is allowed). Compiled expressions are defined in a child classloader created under any that is supplied. It is important to ensure that, if a classloader is specified, it can see all the types involved in the expression evaluation process. If you do not specify a classloader, a default classloader is 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 it through a configuration object. In these cases, it is possible to use a system property. You can set the spring.expression.compiler.mode property to one of the SpelCompilerMode enum values (off, immediate, or mixed).

Compiler Limitations

Since 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. The following 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 types of expression will be compilable in the future.

4.2. Expressions in Bean Definitions

You can use SpEL expressions with XML-based or annotation-based configuration metadata for defining BeanDefinitioninstances. In both cases, the syntax to define the expression is of the form #{ <expression string> }.

4.2.1. XML Configuration

A property or constructor argument value can be set by using expressions, as the following example shows:

<bean id="numberGuess" class="org.spring.samples.NumberGuess">

<property name="randomNumber" value="#{ T(java.lang.Math).random() \* 100.0 }"/>

*<!-- other properties -->*

</bean>

The systemProperties variable is predefined, so you can use it in your expressions, as the following example shows:

<bean id="taxCalculator" class="org.spring.samples.TaxCalculator">

<property name="defaultLocale" value="#{ systemProperties['user.region'] }"/>

*<!-- other properties -->*

</bean>

Note that you do not have to prefix the predefined variable with the # symbol in this context.

You can also refer to other bean properties by name, as the following example shows:

<bean id="numberGuess" class="org.spring.samples.NumberGuess">

<property name="randomNumber" value="#{ T(java.lang.Math).random() \* 100.0 }"/>

*<!-- other properties -->*

</bean>

<bean id="shapeGuess" class="org.spring.samples.ShapeGuess">

<property name="initialShapeSeed" value="#{ numberGuess.randomNumber }"/>

*<!-- other properties -->*

</bean>

4.2.2. Annotation Configuration

To specify a default value, you can place the @Value annotation on fields, methods, and method or constructor parameters.

The following example sets 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 following example shows the equivalent but on a property setter method:

**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, as the following examples show:

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

**private** MovieFinder movieFinder;

**private** String defaultLocale;

@Autowired

**public** **void** configure(MovieFinder movieFinder,

@Value("#{ systemProperties['user.region'] }") String defaultLocale) {

this.movieFinder = movieFinder;

this.defaultLocale = defaultLocale;

}

*// ...*

}

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

**private** String defaultLocale;

**private** CustomerPreferenceDao customerPreferenceDao;

@Autowired

**public** MovieRecommender(CustomerPreferenceDao customerPreferenceDao,

@Value("#{systemProperties['user.country']}") String defaultLocale) {

this.customerPreferenceDao = customerPreferenceDao;

this.defaultLocale = defaultLocale;

}

*// ...*

}

4.3. Language Reference

This section describes how the Spring Expression Language works. It covers the following topics:

* [Literal Expressions](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#expressions-ref-literal)
* [Properties, Arrays, Lists, Maps, and Indexers](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#expressions-properties-arrays)
* [Inline Lists](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#expressions-inline-lists)
* [Inline Maps](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#expressions-inline-maps)
* [Array Construction](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#expressions-array-construction)
* [Methods](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#expressions-methods)
* [Operators](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#expressions-operators)
* [Types](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#expressions-types)
* [Constructors](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#expressions-constructors)
* [Variables](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#expressions-ref-variables)
* [Functions](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#expressions-ref-functions)
* [Bean References](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#expressions-bean-references)
* [Ternary Operator (If-Then-Else)](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#expressions-operator-ternary)
* [The Elvis Operator](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#expressions-operator-elvis)
* [Safe Navigation Operator](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#expressions-operator-safe-navigation)

4.3.1. Literal Expressions

The types of literal expressions supported are strings, numeric values (int, real, hex), boolean, and null. Strings are delimited by single quotation marks. To put a single quotation mark itself in a string, use two single quotation mark characters.

The following listing shows simple usage of literals. Typically, they are not 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 by using Double.parseDouble().

4.3.2. Properties, Arrays, Lists, Maps, and Indexers

Navigating with property references is easy. To do so, use a period to indicate a nested property value. The instances of the Inventor class, pupin and tesla, were populated with data listed in the [Classes used in the examples](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#expressions-example-classes) section. To navigate “down” and get Tesla’s year of birth and Pupin’s city of birth, we use the following expressions:

*// 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 by using square bracket notation, as the following example shows:

ExpressionParser parser = **new** SpelExpressionParser();

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();

*// Inventions Array*

*// evaluates to "Induction motor"*

String invention = parser.parseExpression("inventions[3]").getValue(

context, tesla, String.class);

*// Members List*

*// evaluates to "Nikola Tesla"*

String name = parser.parseExpression("Members[0].Name").getValue(

context, ieee, String.class);

*// List and Array navigation*

*// evaluates to "Wireless communication"*

String invention = parser.parseExpression("Members[0].Inventions[6]").getValue(

context, ieee, String.class);

The contents of maps are obtained by specifying the literal key value within the brackets. In the following example, 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");

4.3.3. Inline Lists

You can directly express lists in an expression by 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, a constant list is created to represent the expression (rather than building a new list on each evaluation).

4.3.4. Inline Maps

You can also directly express maps in an expression by using {key:value} notation. The following example shows how to do so:

*// 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), 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 do not use quoted keys.

4.3.5. Array Construction

You can build arrays by using the familiar Java syntax, optionally supplying an initializer to have the array populated at construction time. The following example shows how to do so:

**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);

You cannot currently supply an initializer when you construct multi-dimensional array.

4.3.6. Methods

You can invoke methods by using typical Java programming syntax. You can also invoke methods on literals. Variable arguments are also supported. The following examples show how to invoke methods:

*// string literal, evaluates to "bc"*

String bc = parser.parseExpression("'abc'.substring(1, 3)").getValue(String.class);

*// evaluates to true*

**boolean** isMember = parser.parseExpression("isMember('Mihajlo Pupin')").getValue(

societyContext, Boolean.class);

4.3.7. Operators

The Spring Expression Language supports the following kinds of operators:

* [Relational Operators](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#expressions-operators-relational)
* [Logical Operators](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#expressions-operators-logical)
* [Mathematical Operators](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#expressions-operators-mathematical)
* [The Assignment Operator](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#expressions-assignment)

Relational Operators

The relational operators (equal, not equal, less than, less than or equal, greater than, and greater than or equal) are supported by using standard operator notation. The following listing shows a few examples of operators:

*// 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-than and less-than comparisons against null follow a simple rule: null is treated as nothing (that is 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, avoid number-based null comparisons in favor of comparisons against zero (for example, X > 0 or X < 0). |

In addition to the standard relational operators, SpEL supports the instanceof and regular expression-based matches operator. The following listing shows examples of both:

*// 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 (such as in an XML document). The textual equivalents are:

* lt (<)
* gt (>)
* le (<=)
* ge (>=)
* eq (==)
* ne (!=)
* div (/)
* mod (%)
* not (!).

All of the textual operators are case-insensitive.

Logical Operators

SpEL supports the following logical operators:

* and
* or
* not

The following example shows how to use the logical operators

*// -- 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

You can use the addition operator on both numbers and strings. You can use the subtraction, multiplication, and division operators only on numbers. You can also use the modulus (%) and exponential power (^) operators. Standard operator precedence is enforced. The following example shows the mathematical operators in use:

*// 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*

The Assignment Operator

To setting a property, use the assignment operator (=). This is typically done within a call to setValue but can also be done inside a call to getValue. The following listing shows both ways to use the assignment operator:

Inventor inventor = **new** Inventor();

EvaluationContext context = SimpleEvaluationContext.forReadWriteDataBinding().build();

parser.parseExpression("Name").setValue(context, inventor, "Aleksandar Seovic");

*// alternatively*

String aleks = parser.parseExpression(

"Name = 'Aleksandar Seovic'").getValue(context, inventor, String.class);

4.3.8. Types

You can use the special T operator to specify an instance of java.lang.Class (the type). Static methods are invoked by 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 that T() references to types within java.lang do not need to be fully qualified, but all other type references must be. The following example shows how to use the T operator:

Class dateClass = parser.parseExpression("T(java.util.Date)").getValue(Class.class);

Class stringClass = parser.parseExpression("T(String)").getValue(Class.class);

**boolean** trueValue = parser.parseExpression(

"T(java.math.RoundingMode).CEILING < T(java.math.RoundingMode).FLOOR")

.getValue(Boolean.class);

4.3.9. Constructors

You can invoke constructors by using the new operator. You should use the fully qualified class name for all but the primitive types (int, float, and so on) and String. The following example shows how to use the new operator to invoke constructors:

Inventor einstein = p.parseExpression(

"new org.spring.samples.spel.inventor.Inventor('Albert Einstein', 'German')")

.getValue(Inventor.class);

*//create new inventor instance within add method of List*

p.parseExpression(

"Members.add(new org.spring.samples.spel.inventor.Inventor(

'Albert Einstein', 'German'))").getValue(societyContext);

4.3.10. Variables

You can reference variables in the expression by using the #variableName syntax. Variables are set by using the setVariablemethod on EvaluationContext implementations. The following example shows how to use variables:

Inventor tesla = **new** Inventor("Nikola Tesla", "Serbian");

EvaluationContext context = SimpleEvaluationContext.forReadWriteDataBinding().build();

context.setVariable("newName", "Mike Tesla");

parser.parseExpression("Name = #newName").getValue(context, tesla);

System.out.println(tesla.getName()) *// "Mike Tesla"*

The #this and #root Variables

The #this variable is always defined and refers to the current evaluation object (against which unqualified references are resolved). The #root variable 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. The following examples show how to use the #this and #rootvariables:

*// create an array of integers*

List<Integer> primes = **new** ArrayList<Integer>();

primes.addAll(Arrays.asList(2,3,5,7,11,13,17));

*// create parser and set variable 'primes' as the array of integers*

ExpressionParser parser = **new** SpelExpressionParser();

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataAccess();

context.setVariable("primes", primes);

*// all prime numbers > 10 from the list (using selection ?{...})*

*// evaluates to [11, 13, 17]*

List<Integer> primesGreaterThanTen = (List<Integer>) parser.parseExpression(

"#primes.?[#this>10]").getValue(context);

4.3.11. Functions

You can extend SpEL by registering user-defined functions that can be called within the expression string. The function is registered through the EvaluationContext. The following example shows how to register a user-defined function:

Method method = ...;

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();

context.setVariable("myFunction", method);

For example, consider the following utility method that reverses a string:

**public** **abstract** **class** **StringUtils** {

**public** **static** String reverseString(String input) {

StringBuilder backwards = **new** StringBuilder(input.length());

**for** (**int** i = 0; i < input.length(); i++)

backwards.append(input.charAt(input.length() - 1 - i));

}

**return** backwards.toString();

}

}

You can then register and use the preceding method, as the following example shows:

ExpressionParser parser = **new** SpelExpressionParser();

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();

context.setVariable("reverseString",

StringUtils.class.getDeclaredMethod("reverseString", String.class));

String helloWorldReversed = parser.parseExpression(

"#reverseString('hello')").getValue(context, String.class);

4.3.12. Bean References

If the evaluation context has been configured with a bean resolver, you can look up beans from an expression by using the @symbol. The following example shows how to do so:

ExpressionParser parser = **new** SpelExpressionParser();

StandardEvaluationContext context = **new** StandardEvaluationContext();

context.setBeanResolver(**new** MyBeanResolver());

*// This will end up calling resolve(context,"something") on MyBeanResolver during evaluation*

Object bean = parser.parseExpression("@something").getValue(context);

To access a factory bean itself, you should instead prefix the bean name with an & symbol. The following example shows how to do so:

ExpressionParser parser = **new** SpelExpressionParser();

StandardEvaluationContext context = **new** StandardEvaluationContext();

context.setBeanResolver(**new** MyBeanResolver());

*// This will end up calling resolve(context,"&foo") on MyBeanResolver during evaluation*

Object bean = parser.parseExpression("&foo").getValue(context);

4.3.13. Ternary Operator (If-Then-Else)

You can use the ternary operator for performing if-then-else conditional logic inside the expression. The following listing shows a minimal example:

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 follows:

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"*

See the next section on the Elvis operator for an even shorter syntax for the ternary operator.

4.3.14. 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, as the following example shows:

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). The following example shows how to use the Elvis operator:

ExpressionParser parser = **new** SpelExpressionParser();

String name = parser.parseExpression("name?:'Unknown'").getValue(String.class);

System.out.println(name); *// 'Unknown'*

The following listing shows a more complex example:

ExpressionParser parser = **new** SpelExpressionParser();

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();

Inventor tesla = **new** Inventor("Nikola Tesla", "Serbian");

String name = parser.parseExpression("Name?:'Elvis Presley'").getValue(context, tesla, String.class);

System.out.println(name); *// Nikola Tesla*

tesla.setName(null);

name = parser.parseExpression("Name?:'Elvis Presley'").getValue(context, tesla, String.class);

System.out.println(name); *// Elvis Presley*

|  |  |
| --- | --- |
|  | You can use the Elvis operator to apply default values in expressions. The following example shows how to use the Elvis operator in a @Value expression:  @Value("#{systemProperties['pop3.port'] ?: 25}")  This will inject a system property pop3.port if it is defined or 25 if not. |

4.3.15. 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 returns null instead of throwing an exception. The following example shows how to use the safe navigation operator:

ExpressionParser parser = **new** SpelExpressionParser();

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();

Inventor tesla = **new** Inventor("Nikola Tesla", "Serbian");

tesla.setPlaceOfBirth(**new** PlaceOfBirth("Smiljan"));

String city = parser.parseExpression("PlaceOfBirth?.City").getValue(context, tesla, String.class);

System.out.println(city); *// Smiljan*

tesla.setPlaceOfBirth(null);

city = parser.parseExpression("PlaceOfBirth?.City").getValue(context, tesla, String.class);

System.out.println(city); *// null - does not throw NullPointerException!!!*

4.3.16. Collection Selection

Selection is a powerful expression language feature that lets you transform a source collection into another collection by selecting from its entries.

Selection uses a syntax of .?[selectionExpression]. It filters the collection and returns a new collection that contain a subset of the original elements. For example, selection lets us easily get a list of Serbian inventors, as the following example shows:

List<Inventor> list = (List<Inventor>) parser.parseExpression(

"Members.?[Nationality == 'Serbian']").getValue(societyContext);

Selection is possible upon both lists and maps. For a list, the selection criteria is evaluated against each individual list element. Against a map, the selection criteria is evaluated against each map entry (objects of the Java type Map.Entry). Each map entry has its key and value accessible as properties for use in the selection.

The following expression returns a new map that consists 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, you can retrieve only the first or the last value. To obtain the first entry matching the selection, the syntax is .^[selectionExpression]. To obtain the last matching selection, the syntax is.$[selectionExpression].

4.3.17. Collection Projection

Projection lets a collection drive the evaluation of a sub-expression, and the result is a new collection. The syntax for projection is .![projectionExpression]. For 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. The following example uses projection to do so:

*// returns ['Smiljan', 'Idvor' ]*

List placesOfBirth = (List)parser.parseExpression("Members.![placeOfBirth.city]");

You can also use a map 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 that consists of the evaluation of the projection expression against each map entry.

4.3.18. Expression templating

Expression templates allow mixing 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, as the following example shows:

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 parseExpression()method 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 follows:

**public** **class** **TemplateParserContext** **implements** ParserContext {

**public** String getExpressionPrefix() {

**return** "#{";

}

**public** String getExpressionSuffix() {

**return** "}";

}

**public** **boolean** isTemplate() {

**return** true;

}

}

4.4. Classes Used in the Examples

This section lists the classes used in the examples throughout this chapter.

*Example 1. 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;

}

}

*Example 2. 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;

}

}

*Example 3. 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

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 AOP with AspectJ pointcuts

Spring provides simple and powerful ways of writing custom aspects by using either a [schema-based approach](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-schema) or the [@AspectJ annotation style](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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.

This chapter discusses the schema- and @AspectJ-based AOP support. The lower-level AOP support is discussed in [the following chapter](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-api).

AOP is used in the Spring Framework to:

* Provide declarative enterprise services. The most important such service is [declarative transaction management](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/data-access.html#transaction-declarative).
* Let users 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. 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 by using regular classes (the [schema-based approach](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-schema)) or regular classes annotated with the @Aspect annotation (the [@AspectJ style](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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 later.) Many AOP frameworks, including Spring, model an advice as an interceptor and maintain 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 lets you 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: An object being advised by one or more aspects. Also referred to as the “advised object”. Since Spring AOP is implemented by using runtime proxies, this object is always 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 is 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.

Spring AOP includes the following types of advice:

* Before advice: Advice that runs before a join point but that 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 run 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, you cannot fail to invoke it.

All advice parameters are statically typed so that you work with advice parameters of the appropriate type (e.g. the type of the return value from a method execution) 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, you can apply an around advice providing declarative transaction management to a set of methods that span multiple objects (such as all business operations in the service layer).

5.2. 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). Rather, the aim is 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 by using normal bean definition syntax (although this allows powerful “auto-proxying” capabilities). This is a crucial difference from other AOP implementations. You cannot do some things easily or efficiently with Spring AOP, such as advise very fine-grained objects (typically, domain objects). 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 never strives to compete with AspectJ to provide a comprehensive AOP solution. We believe that both proxy-based frameworks such as 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 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/5.1.8.RELEASE/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 or 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. However, the Spring Framework (almost) always offers you the choice: 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, Spring AOP, or both. 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/5.1.8.RELEASE/spring-framework-reference/core.html#aop-choosing) for a more complete discussion of the “whys and wherefores” of each style. |

5.3. 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. By default, CGLIB is used if a business object does not implement an interface. As it is good practice to program to interfaces rather than classes, business classes normally implement one or more business interfaces. It is possible to [force the use of CGLIB](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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/5.1.8.RELEASE/spring-framework-reference/core.html#aop-understanding-aop-proxies) for a thorough examination of exactly what this implementation detail actually means.

5.4. @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/5.1.8.RELEASE/spring-framework-reference/core.html#aop-using-aspectj). |

5.4.1. 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 auto-proxying beans based on whether or not they are advised by those aspects. By auto-proxying, we mean that, if Spring determines that a bean is advised by one or more aspects, it automatically generates a proxy for that bean to intercept method invocations and ensures that advice is executed as needed.

The @AspectJ support can be enabled with XML- or Java-style configuration. In either case, you also need to ensure that AspectJ’s aspectjweaver.jar library is on the classpath of your application (version 1.8 or later). This library is available in thelib directory of an AspectJ distribution or from the Maven Central repository.

Enabling @AspectJ Support with Java Configuration

To enable @AspectJ support with Java @Configuration, add the @EnableAspectJAutoProxy annotation, as the following example shows:

@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, as the following example shows:

<aop:aspectj-autoproxy/>

This assumes that you use schema support as described in [XML Schema-based configuration](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#xsd-schemas). See [the AOP schema](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#xsd-schemas-aop) for how to import the tags in the aop namespace.

5.4.2. Declaring an Aspect

With @AspectJ support enabled, any bean defined in your application context with a class that is an @AspectJ aspect (has the @Aspect annotation) is automatically detected by Spring and used to configure Spring AOP. The next two examples show the minimal definition required for a not-very-useful aspect.

The first of the two example shows a regular bean definition in the application context that points to a bean class that has the @Aspect annotation:

<bean id="myAspect" class="org.xyz.NotVeryUsefulAspect">

*<!-- configure properties of the aspect here -->*

</bean>

The second of the two examples shows the NotVeryUsefulAspect class definition, which is annotated with the org.aspectj.lang.annotation.Aspect annotation;

**package** org.xyz;

**import** org.aspectj.lang.annotation.Aspect;

@Aspect

**public** **class** **NotVeryUsefulAspect** {

}

Aspects (classes annotated with @Aspect) can have methods and fields, the same as any other class. They can also contain pointcut, advice, and introduction (inter-type) declarations.

|  |  |
| --- | --- |
|  | *Autodetecting aspects through component scanning*  You can register aspect classes as regular beans in your Spring XML configuration or autodetect them through classpath scanning — the same as 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 @Componentannotation (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, aspects themselves cannot be the targets of advice from other aspects. The @Aspect annotation on a class marks it as an aspect and, hence, excludes it from auto-proxying. |

5.4.3. Declaring a Pointcut

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 by using the @Pointcutannotation (the method serving as the pointcut signature must have a void return type).

An example may help make this distinction between a pointcut signature and a pointcut expression clear. The following example defines a pointcut named anyOldTransfer that matches 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 Developer’s 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:

* execution: For matching method execution join points. This is the primary pointcut designator to use when working with Spring AOP.
* within: Limits matching to join points within certain types (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 types.
* @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 (the method being executed in Spring AOP) has the given annotation.

Other pointcut types

The full AspectJ pointcut language supports additional pointcut designators that are not supported in Spring: 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 results 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.

Because Spring AOP limits matching to only method execution join points, the preceding discussion of the pointcut designators gives a narrower definition than you can 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 (which is bound to this) and the target object behind the proxy (which is 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 are 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/5.1.8.RELEASE/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 before making a decision. |

Spring AOP also supports an additional PCD named bean. This PCD lets you 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 that uses the \* character is provided, so, if you establish some naming conventions for your Spring beans, you can write a bean PCD expression to select them. As is the case with other pointcut designators, the bean PCD can be used with the && (and), || (or), and ! (negation) operators, too.

|  |  |
| --- | --- |
|  | The bean PCD is supported only in Spring AOP and not in native AspectJ weaving. It is a Spring-specific extension to the standard PCDs that AspectJ defines and is, 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 (to which weaving-based AOP is limited). 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

You can combine pointcut expressions by using &&, || and !. You can also refer to pointcut expressions by name. The following example shows three pointcut expressions:

@Pointcut("execution(public \* \*(..))")

**private** **void** anyPublicOperation() {}

@Pointcut("within(com.xyz.someapp.trading..\*)")

**private** **void** inTrading() {}

@Pointcut("anyPublicOperation() && inTrading()")

**private** **void** tradingOperation() {}

|  |  |
| --- | --- |
|  | anyPublicOperation matches if a method execution join point represents the execution of any public method. |
|  | inTrading matches if a method execution is in the trading module. |
|  | tradingOperation matches if a method execution represents any public method in the trading module. |

It is a best practice to build more complex pointcut expressions out of smaller named components, as shown earlier. 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, developers 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. Such an aspect typically resembles the following example:

**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() {}

}

You can refer to the pointcuts defined in such an aspect anywhere you need a pointcut expression. For example, to make the service layer transactional, you could write the following:

<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/5.1.8.RELEASE/spring-framework-reference/core.html#aop-schema). The transaction elements are discussed in [Transaction Management](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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 follows:

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 preceding snippet), the name pattern, and the 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. \* is most frequently used as the returning type pattern. It matches any return type. A fully-qualified type name matches 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 you specify a declaring type pattern, 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 (zero or more) of parameters. The (\*) pattern matches a method that takes one parameter of any type. (\*,String) matches a method that takes two parameters. The first can be of any type, while 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.

The following examples show some common pointcut expressions:

* The execution of any public method:

execution(**public** \* \*(..))

* The execution of any method with a name that begins 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 one of its sub-packages:

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 one of its sub-packages:

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 section on [Declaring Advice](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-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 [Declaring Advice](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-advice) section for how to make the target object available in the advice body. |

* Any join point (method execution only in Spring AOP) that 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 [Declaring Advice](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-advice) section for how to make the method arguments available in the advice body. |

Note that the pointcut given in this example is different from execution(\* \*(java.io.Serializable)). The args version matches if the argument passed at runtime is Serializable, and 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 a @Transactional annotation:

@target(org.springframework.transaction.annotation.Transactional)

|  |  |
| --- | --- |
|  | You can also use '@target' in a binding form. See the [Declaring Advice](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-advice) section 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)

|  |  |
| --- | --- |
|  | You can also use '@within' in a binding form. See the [Declaring Advice](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-advice) section 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)

|  |  |
| --- | --- |
|  | You can also use '@annotation' in a binding form. See the [Declaring Advice](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-advice) section 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)

|  |  |
| --- | --- |
|  | You can also use '@args' in a binding form. See the [Declaring Advice](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-advice) section 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 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 that a test is placed in the code to determine if there is an actual match when the code is running). On first encountering a pointcut declaration, AspectJ rewrites 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 work only with what it is told. 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 contextual:

* Kinded designators select a particular kind of join point: execution, get, set, call, and handler.
* Scoping designators select a group of join points of interest (probably of many kinds): within and withincode
* Contextual designators match (and optionally bind) based on context: this, target, and @annotation

A well written pointcut should include at least the first two types (kinded and scoping). You can include the contextual designators to match based on join point context or bind that context for use in the advice. Supplying only a kinded designator or only a contextual designator works but could affect weaving performance (time and memory used), due to extra processing and analysis. Scoping designators are very fast to match, and using them usage means AspectJ can very quickly dismiss groups of join points that should not be further processed. A good pointcut should always include one if possible.

5.4.4. 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

You can declare before advice in an aspect by 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 we use an in-place pointcut expression, we could rewrite the preceding example as the following example:

**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. You can declare it by using the @AfterReturning annotation:

**import** org.aspectj.lang.annotation.Aspect;

**import** org.aspectj.lang.annotation.AfterReturning;

@Aspect

**public** **class** **AfterReturningExample** {

@AfterReturning("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")

**public** **void** doAccessCheck() {

*// ...*

}

}

|  |  |
| --- | --- |
|  | You can have multiple advice declarations (and other members as well), all inside the same aspect. We show only a single advice declaration in these examples to focus the effect of each one. |

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 to get that access, as the following example shows:

**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 is passed to the advice method as the corresponding argument value. A returning clause also restricts matching to only those method executions that return a value of the specified type (in this case, Object, which matches 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. You can declare it by using the @AfterThrowing annotation, as the following example shows:

**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. You can use the throwing attribute to both restrict matching (if desired — use Throwableas the exception type otherwise) and bind the thrown exception to an advice parameter. The following example shows how to do so:

**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 is 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 when a matched method execution exits. It is declared by using the @After annotation. After advice must be prepared to handle both normal and exception return conditions. It is typically used for releasing resources and similar purposes. The following example shows how to use after finally advice:

**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 last kind of advice is around advice. Around advice runs “around” a matched method’s 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 (that is, do not use around advice if before advice would do).

Around advice is declared by 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 can also pass in an Object[]. The values in the array are 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 (do not worry if this does not 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 compile @AspectJ aspects written for Spring and use 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](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-ataspectj-advice-params). |

The following example shows how to use around advice:

**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 is the return value seen by the caller of the method. For example, a simple caching aspect 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 legal.

Advice Parameters

Spring offers fully typed advice, meaning that you declare the parameters you need in the advice signature (as we saw earlier for the returning and throwing examples) rather than work with Object[] arrays all the time. We see how to make argument and other contextual values available to the advice body later in this section. First, we 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 (note that around advice is required to declare a first parameter of type ProceedingJoinPoint, which is a subclass of JoinPoint. The JoinPointinterface provides a number of useful methods:

* 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.
* toString(): Prints a useful description of the method being advised.

See the [javadoc](https://www.eclipse.org/aspectj/doc/released/runtime-api/org/aspectj/lang/JoinPoint.html) for more detail.

Passing Parameters to Advice

We have 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 you use a parameter name in place of a type name in an args expression, the value of the corresponding argument is 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. First, 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. Second, it makes the actual Account object available to the advice through 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 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) {

*// ...*

}

See the AspectJ programming guide for more details.

The proxy object ( this), target object ( target), and annotations ( @within, @target, @annotation, and @args) can all be bound in a similar fashion. The next two examples show how to match the execution of methods annotated with an @Auditableannotation and extract the audit code:

The first of the two examples shows the definition of the @Auditable annotation:

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.METHOD)

**public** @interface Auditable {

AuditCode value();

}

The second of the two examples shows 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 the following:

**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 typing the advice parameter to the parameter type for which you want to intercept the method:

@Before("execution(\* ..Sample+.sampleGenericMethod(\*)) && args(param)")

**public** **void** beforeSampleMethod(MyType param) {

*// Advice implementation*

}

This approach does not work for generic collections. So you cannot define a pointcut as follows:

@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 strategy to determine parameter names:

* If the parameter names have been explicitly specified by the user, the specified parameter names are used. Both the advice and the pointcut annotations have an optional argNames attribute that you can use to specify the argument names of the annotated method. These argument names are available at runtime. The following example shows how to use the argNamesattribute:
* @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 can 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 instances that do not collect any other join point context. In such situations, you may 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, Spring AOP looks at the debug information for the class and tries to determine the parameter names from the local variable table. This information is 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 is slightly easier to understand (reverse engineer), (2) the class file sizes are very slightly bigger (typically inconsequential), (3) the optimization to remove unused local variables is not applied by your compiler. In other words, you should encounter no difficulties by building with this flag on.

|  |  |
| --- | --- |
|  | If an @AspectJ aspect has been compiled by the AspectJ compiler (ajc) even without the debug information, you need not add the argNames attribute, as the compiler retain the needed information. |

* If the code has been compiled without the necessary debug information, Spring AOP tries 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 takes only one parameter, the pairing is obvious). If the binding of variables is ambiguous given the available information, an AmbiguousBindingException is thrown.
* If all of the above strategies fail, an IllegalArgumentException is 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 to ensure that the advice signature binds each of the method parameters in order. The following example shows how to do so:

@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 do this binding anyway (as in the preceding example).

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 through 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 that you can order at the aspect level.

5.4.5. 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.

You can make an introduction by 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 named UsageTracked and an implementation of that interface named DefaultUsageTracked, the following aspect declares that all implementors of service interfaces also implement the UsageTracked interface (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 implements the UsageTracked interface. Note that, in the before advice of the preceding 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");

5.4.6. Aspect Instantiation Models

|  |  |
| --- | --- |
|  | This is an advanced topic. If you are just starting out with AOP, you can safely skip it until later. |

By default, there is a single instance of each aspect within the application context. AspectJ calls this the singleton instantiation model. It is possible to define aspects with alternate lifecycles. Spring supports AspectJ’s perthis and pertarget instantiation models ( percflow, percflowbelow, and pertypewithin are not currently supported).

You can declare a perthis aspect by specifying a perthis clause in the @Aspect annotation. Consider the following example:

@Aspect("perthis(com.xyz.myapp.SystemArchitecture.businessService())")

**public** **class** **MyAspect** {

**private** **int** someState;

@Before(com.xyz.myapp.SystemArchitecture.businessService())

**public** **void** recordServiceUsage() {

*// ...*

}

}

In the preceding example, the effect of the 'perthis' clause is that one aspect instance is created for each unique service object that executes 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 executes at matched join points, but only when the service object is the one with which this aspect is associated. See the AspectJ Programming Guide for more information on perclauses.

The pertarget instantiation model works in exactly the same way as perthis, but it creates one aspect instance for each unique target object at matched join points.

5.4.7. An AOP Example

Now that you have seen how all the constituent parts work, we can put them together to do something useful.

The execution of business services can sometimes fail due to concurrency issues (for example, a deadlock loser). If the operation is retried, it is likely to succeed on the next try. For business services where it is appropriate to retry in such conditions (idempotent operations that do not need to go back to the user for conflict resolution), we want 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 through an aspect.

Because we want to retry the operation, we need to use around advice so that we can call proceed multiple times. The following listing shows the basic aspect implementation:

@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 that 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 are both configured by Spring. The main action happens in the doConcurrentOperation around advice. Notice that, for the moment, we apply the retry logic to each businessService(). We try to proceed, and if we fail with a PessimisticLockingFailureException, we try again, unless we have exhausted all of our retry attempts.

The corresponding Spring configuration follows:

<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 retries only idempotent operations, we might define the following Idempotent annotation:

@Retention(RetentionPolicy.RUNTIME)

**public** @interface Idempotent {

*// marker annotation*

}

We can then use the annotation to annotate the implementation of service operations. The change to the aspect to retry only idempotent operations involves refining the pointcut expression so that only @Idempotent operations match, as follows:

@Around("com.xyz.myapp.SystemArchitecture.businessService() && " +

"@annotation(com.xyz.myapp.service.Idempotent)")

**public** Object doConcurrentOperation(ProceedingJoinPoint pjp) **throws** Throwable {

...

}

5.5. Schema-based AOP Support

If you prefer an XML-based format, Spring also offers support for defining aspects using the new aop namespace tags. The exact same pointcut expressions and advice kinds as when using the @AspectJ style are supported. Hence, in this section we focus on the new syntax and refer the reader to the discussion in the previous section ([@AspectJ support](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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/5.1.8.RELEASE/spring-framework-reference/core.html#xsd-schemas). See [the AOP schema](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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 that 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/5.1.8.RELEASE/spring-framework-reference/core.html#aop-autoproxy) mechanism. This can cause issues (such as advice not being woven) if you already use explicit auto-proxying through the use ofBeanNameAutoProxyCreator or something similar. The recommended usage pattern is to use either only the <aop:config> style or only the AutoProxyCreator style and never mix them. |

5.5.1. Declaring an Aspect

When you use the schema support, an aspect is a regular Java object defined as a bean in your Spring application context. The state and behavior are captured in the fields and methods of the object, and the pointcut and advice information are captured in the XML.

You can declare an aspect by using the <aop:aspect> element, and reference the backing bean by using the ref attribute, as the following example shows:

<aop:config>

<aop:aspect id="myAspect" ref="aBean">

...

</aop:aspect>

</aop:config>

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

...

</bean>

The bean that backs the aspect (aBean in this case) can of course be configured and dependency injected just like any other Spring bean.

5.5.2. Declaring a Pointcut

You can declare a named pointcut inside an <aop:config> element, letting the pointcut definition be shared across several aspects and advisors.

A pointcut that represents the execution of any business service in the service layer can 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/5.1.8.RELEASE/spring-framework-reference/core.html#aop-ataspectj). If you use 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 as follows:

<aop:config>

<aop:pointcut id="businessService"

expression="com.xyz.myapp.SystemArchitecture.businessService()"/>

</aop:config>

Assume that you have a SystemArchitecture aspect as described in [Sharing Common Pointcut Definitions](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-common-pointcuts).

Then declaring a pointcut inside an aspect is very similar to declaring a top-level pointcut, as the following example shows:

<aop:config>

<aop:aspect id="myAspect" ref="aBean">

<aop:pointcut id="businessService"

expression="execution(\* com.xyz.myapp.service.\*.\*(..))"/>

...

</aop:aspect>

</aop:config>

In much the same way as an @AspectJ aspect, pointcuts declared by using the schema based definition style can collect join point context. For example, the following pointcut collects the this object as the join point context and passes it to the 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, as follows:

**public** **void** monitor(Object service) {

...

}

When combining pointcut sub-expressions, && is awkward within an XML document, so you can use the and, or, and notkeywords in place of &&, ||, and !, respectively. For example, the previous pointcut can be better written as follows:

<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.

5.5.3. Declaring Advice

The schema-based AOP support uses the same five kinds of advice as 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> by using the <aop:before> element, as the following example shows:

<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, as follows:

<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 that contains 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 is 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. The following example shows how to declare it:

<aop:aspect id="afterReturningExample" ref="aBean">

<aop:after-returning

pointcut-ref="dataAccessOperation"

method="doAccessCheck"/>

...

</aop:aspect>

As in the @AspectJ style, you can get the return value within the advice body. To do so, use the returning attribute to specify the name of the parameter to which the return value should be passed, as the following example shows:

<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, you can declare the method signature as follows:

**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> by using the after-throwing element, as the following example shows:

<aop:aspect id="afterThrowingExample" ref="aBean">

<aop:after-throwing

pointcut-ref="dataAccessOperation"

method="doRecoveryActions"/>

...

</aop:aspect>

As in the @AspectJ style, you can get the thrown exception within the advice body. To do so, use the throwing attribute to specify the name of the parameter to which the exception should be passed as the following example shows:

<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 follows:

**public** **void** doRecoveryActions(DataAccessException dataAccessEx) {...

After (Finally) Advice

After (finally) advice runs no matter how a matched method execution exits. You can declare it by using the after element, as the following example shows:

<aop:aspect id="afterFinallyExample" ref="aBean">

<aop:after

pointcut-ref="dataAccessOperation"

method="doReleaseLock"/>

...

</aop:aspect>

Around Advice

The last 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 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. Do not use around advice if before advice can do the job.

You can declare around advice by 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 called with an Object[]. The values in the array are used as the arguments to the method execution when it proceeds. See [Around Advice](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-ataspectj-around-advice) for notes on calling proceed with an Object[]. The following example shows how to declare around advice in XML:

<aop:aspect id="aroundExample" ref="aBean">

<aop:around

pointcut-ref="businessService"

method="doBasicProfiling"/>

...

</aop:aspect>

The implementation of the doBasicProfiling advice can be exactly the same as in the @AspectJ example (minus the annotation, of course), as the following example shows:

**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/5.1.8.RELEASE/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), you can do so by using the arg-names attribute of the advice element, which is treated in the same manner as the argNames attribute in an advice annotation (as described in [Determining Argument Names](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-ataspectj-advice-params-names)). The following example shows how to specify an argument name in XML:

<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.

The following slightly more involved example of the XSD-based approach shows some around advice used in conjunction with a number of strongly typed parameters:

**package** x.y.service;

**public** **interface** **PersonService** {

Person getPerson(String personName, **int** age);

}

**public** **class** **DefaultFooService** **implements** FooService {

**public** Person getPerson(String name, **int** age) {

**return** **new** Person(name, age);

}

}

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, as the following example shows:

**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, the following example XML configuration effects the execution of the preceding 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 https://www.springframework.org/schema/beans/spring-beans.xsd

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

*<!-- this is the object that will be proxied by Spring's AOP infrastructure -->*

<bean id="personService" class="x.y.service.DefaultPersonService"/>

*<!-- this is the actual advice itself -->*

<bean id="profiler" class="x.y.SimpleProfiler"/>

<aop:config>

<aop:aspect ref="profiler">

<aop:pointcut id="theExecutionOfSomePersonServiceMethod"

expression="execution(\* x.y.service.PersonService.getPerson(String,int))

and args(name, age)"/>

<aop:around pointcut-ref="theExecutionOfSomePersonServiceMethod"

method="profile"/>

</aop:aspect>

</aop:config>

</beans>

Consider the following driver script:

**import** org.springframework.beans.factory.BeanFactory;

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

**import** x.y.service.PersonService;

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

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

BeanFactory ctx = **new** ClassPathXmlApplicationContext("x/y/plain.xml");

PersonService person = (PersonService) ctx.getBean("personService");

person.getPerson("Pengo", 12);

}

}

With such a Boot class, we would get output similar to the following on standard output:

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/5.1.8.RELEASE/spring-framework-reference/core.html#aop-ataspectj-advice-ordering). The precedence between aspects is determined by either adding the Order annotation to the bean that backs the aspect or by having the bean implement the Ordered interface.

5.5.4. Introductions

Introductions (known as inter-type declarations in AspectJ) let an aspect declare that advised objects implement a given interface and provide an implementation of that interface on behalf of those objects.

You can make an introduction by using the aop:declare-parents element inside an aop:aspect. You can use the aop:declare-parents element to declare that matching types have a new parent (hence the name). For example, given an interface named UsageTracked and an implementation of that interface named DefaultUsageTracked, the following aspect declares that all implementors of service interfaces also implement the UsageTracked interface. (In order to expose statistics through 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 that backs the usageTracking bean would then contain the following method:

**public** **void** recordUsage(UsageTracked usageTracked) {

usageTracked.incrementUseCount();

}

The interface to be implemented is determined by the implement-interface attribute. The value of the types-matchingattribute is an AspectJ type pattern. Any bean of a matching type implements the UsageTracked interface. Note that, in the before advice of the preceding example, service beans can be directly used as implementations of the UsageTracked interface. To access a bean programmatically, you could write the following:

UsageTracked usageTracked = (UsageTracked) context.getBean("myService");

5.5.5. 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.

5.5.6. Advisors

The concept of “advisors” comes 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/5.1.8.RELEASE/spring-framework-reference/core.html#aop-api-advice-types). Advisors can take advantage of AspectJ pointcut expressions.

Spring supports the advisor concept with the <aop:advisor> element. You most commonly see it used in conjunction with transactional advice, which also has its own namespace support in Spring. The following example shows an advisor:

<aop:config>

<aop:pointcut id="businessService"

expression="execution(\* com.xyz.myapp.service.\*.\*(..))"/>

<aop:advisor

pointcut-ref="businessService"

advice-ref="tx-advice"/>

</aop:config>

<tx:advice id="tx-advice">

<tx:attributes>

<tx:method name="\*" propagation="REQUIRED"/>

</tx:attributes>

</tx:advice>

As well as the pointcut-ref attribute used in the preceding 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.

5.5.7. An AOP Schema Example

This section shows how the concurrent locking failure retry example from [An AOP Example](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-ataspectj-example) looks when rewritten with the schema support.

The execution of business services can sometimes fail due to concurrency issues (for example, a deadlock loser). If the operation is retried, it is likely to succeed on the next try. For business services where it is appropriate to retry in such conditions (idempotent operations that do not need to go back to the user for conflict resolution), we want 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 through an aspect.

Because we want to retry the operation, we need to use around advice so that we can call proceed multiple times. The following listing shows the basic aspect implementation (which is a regular Java class that uses 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 that 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 are both configured by Spring. The main action happens in the doConcurrentOperation around advice method. We try to proceed. If we fail with a PessimisticLockingFailureException, we 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 as follows:

<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 retries only genuinely idempotent operations, by introducing an Idempotent annotation and using the annotation to annotate the implementation of service operations, as the following example shows:

@Retention(RetentionPolicy.RUNTIME)

**public** @interface Idempotent {

*// marker annotation*

}

The change to the aspect to retry only idempotent operations involves refining the pointcut expression so that only @Idempotent operations match, as follows:

<aop:pointcut id="idempotentOperation"

expression="execution(\* com.xyz.myapp.service.\*.\*(..)) and

@annotation(com.xyz.myapp.service.Idempotent)"/>

5.6. 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, the @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.

5.6.1. 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, Spring AOP is the right choice. If you need to advise objects not managed by the Spring container (such as domain objects, typically), you need to use AspectJ. You 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 you use AspectJ, you have the choice of the AspectJ language syntax (also known as the “code style”) or the @AspectJ annotation style. Clearly, if you do not use Java 5+, 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, the AspectJ language syntax is the preferred option. It is cleaner and simpler because the language was purposefully designed for writing aspects. If you do not use Eclipse or have only a few aspects that do not play a major role in your application, you may want to consider using the @AspectJ style, sticking with regular Java compilation in your IDE, and adding an aspect weaving phase to your build script.

5.6.2. @AspectJ or XML for Spring AOP?

If you have chosen to use Spring AOP, you have a choice of @AspectJ or XML style. There are various tradeoffs to consider.

The XML style may most familiar to existing Spring users, and it is backed by genuine POJOs. When using AOP as a tool to configure enterprise services, XML can be a good choice (a good test is whether you consider the pointcut expression to be a part of your configuration that you might want to change independently). With the XML style, it is arguably clearer from your configuration which aspects are present in the system.

The XML style has two disadvantages. First, 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 you use the @AspectJ style, this information is encapsulated in a single module: the aspect. 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 the following:

@Pointcut("execution(\* get\*())")

**public** **void** propertyAccess() {}

@Pointcut("execution(org.xyz.Account+ \*(..))")

**public** **void** operationReturningAnAccount() {}

@Pointcut("propertyAccess() && operationReturningAnAccount()")

**public** **void** accountPropertyAccess() {}

In the XML style you 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 that 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, you can easily migrate to a classic AspectJ setup. On balance, the Spring team prefers the @AspectJ style for custom aspects beyond simple configuration of enterprise services.

5.7. Mixing Aspect Types

It is perfectly possible to mix @AspectJ style aspects by using the auto-proxying support, schema-defined <aop:aspect>aspects, <aop:advisor> declared advisors, and even proxies and interceptors in other styles in the same configuration. All of these are implemented by using the same underlying support mechanism and can co-exist without any difficulty.

5.8. Proxying Mechanisms

Spring AOP uses either JDK dynamic proxies or CGLIB to create the proxy for a given target object. JDK dynamic proxies are built into the JDK, whereas CGLIB is a common open-source class definition library (repackaged into spring-core).

If the target object to be proxied implements at least one interface, a JDK dynamic proxy is used. All of the interfaces implemented by the target type are proxied. If the target object does not implement any interfaces, a CGLIB proxy is created.

If you want to force the use of CGLIB proxying (for example, to proxy every method defined for the target object, not only those implemented by its interfaces), you can do so. However, you should consider the following issues:

* With CGLIB, final methods cannot be advised, as they cannot be overridden in runtime-generated subclasses.
* As of Spring 4.0, the constructor of your proxied object is NOT called twice anymore, since the CGLIB proxy instance is created through 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, as follows:

<aop:config proxy-target-class="true">

*<!-- other beans defined here... -->*

</aop:config>

To force CGLIB proxying when you use the @AspectJ auto-proxy support, set the proxy-target-class attribute of the <aop:aspectj-autoproxy> element to true, as follows:

<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 forces the use of CGLIB proxies *for all three of them*. |

5.8.1. 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 the following code snippet shows:

**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 the following image and listing show:



**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(..) method of the Main class has a reference to the proxy. This means that method calls on that object reference are calls on the proxy. As a result, the proxy can 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. This does entail some work on your part, but it is the best, least-invasive approach. The next approach is absolutely horrendous, and we hesitate to point it out, precisely because it is so horrendous. You can (painful as it is to us) totally tie the logic within your class to Spring AOP, as the following example shows:

**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, as the following example shows:

**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.

5.9. Programmatic Creation of @AspectJ Proxies

In addition to declaring aspects in your configuration by using either <aop:config> or <aop:aspectj-autoproxy>, it is also possible to programmatically create proxies that advise target objects. For the full details of Spring’s AOP API, see the [next chapter](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-api). Here, we want to focus on the ability to automatically create proxies by using @AspectJ aspects.

You can use the org.springframework.aop.aspectj.annotation.AspectJProxyFactory class to create a proxy for a target object that is advised by one or more @AspectJ aspects. The basic usage for this class is very simple, as the following example shows:

*// 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();

See the [javadoc](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/aop/aspectj/annotation/AspectJProxyFactory.html) for more information.

5.10. Using AspectJ with Spring Applications

Everything we have covered so far in this chapter is pure Spring AOP. In this section, we look at how you can use the AspectJ compiler or 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 stand-alone in your distribution as spring-aspects.jar. You 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/5.1.8.RELEASE/spring-framework-reference/core.html#aop-atconfigurable) and [Other Spring aspects for AspectJ](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-ajlib-other) discuss the content of this library and how you can use it. [Configuring AspectJ Aspects by Using Spring IoC](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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/5.1.8.RELEASE/spring-framework-reference/core.html#aop-aj-ltw) provides an introduction to load-time weaving for Spring applications that use AspectJ.

5.10.1. 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 that contains the configuration to be applied.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 with the new operator or by an ORM tool as a result of a database query.

The @Configurable annotation marks a class as being eligible for Spring-driven configuration. In the simplest case, you can use purely it as a marker annotation, as the following example shows:

**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 configures new instances of the annotated type (Account, in this case) by 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 to omit the id attribute, as the following example shows:

<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, as the following example shows:

**package** com.xyz.myapp.domain;

**import** org.springframework.beans.factory.annotation.Configurable;

@Configurable("account")

**public** **class** **Account** {

*// ...*

}

Spring now looks for a bean definition named account and uses 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. You can specify either @Configurable(autowire=Autowire.BY\_TYPE) or@Configurable(autowire=Autowire.BY\_NAME for autowiring by type or by name, respectively. As an alternative, it is preferable to specify explicit, annotation-driven dependency injection for your @Configurable beans through @Autowired or @Inject at the field or method level (see [Annotation-based Container Configuration](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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, Spring validates after configuration that all properties (which are not primitives or collections) have been set.

Note that using the annotation on its own does nothing. 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 (for example, objects instantiated with the newoperator) as well as to Serializable objects that are undergoing deserialization (for example, through [readResolve()](https://docs.oracle.com/javase/8/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” are fine. In this context, “after initialization” means that the dependencies are injected after the object has been constructed. This means that the dependencies are not 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, you need to define this on the @Configurable declaration, as follows:  @Configurable(preConstruction = true)  You can find 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/5.1.8.RELEASE/spring-framework-reference/core.html#aop-aj-ltw)). The AnnotationBeanConfigurerAspect itself needs to be configured by Spring (in order to obtain a reference to the bean factory that is to be used to configure new objects). If you use Java-based configuration, you can add @EnableSpringConfigured to any @Configuration class, as follows:

@Configuration

@EnableSpringConfigured

**public** **class** **AppConfig** {

}

If you prefer XML based configuration, the Spring [context namespace](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#xsd-schemas-context) defines a convenient context:spring-configuredelement, which you can use as follows:

<context:spring-configured/>

Instances of @Configurable objects created before the aspect has been configured 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. The following example shows how to use the depends-on attribute:

<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 that are registered as regular Spring beans with the container. Doing so results in double initialization, 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, the annotation has no affect during unit testing. You can set mock or stub property references in the object under test and proceed as normal. If @Configurable types have been woven by AspectJ, you can still unit test outside of the container as normal, but you see a warning message each time that you construct a @Configurable object indicating that it has not been configured by Spring.

Working with Multiple Application Contexts

The AnnotationBeanConfigurerAspect that is 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: 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 application configuration that has a shared parent application context that defines common business services, everything needed to support those services, and one child application context for each servlet (which contains definitions particular to that servlet). All of these contexts co-exist within the same classloader hierarchy, and so the AnnotationBeanConfigurerAspect can hold a reference to only 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 by using the @Configurable mechanism (which is probably not something you want to do anyway).

When deploying multiple web applications within the same container, ensure that each web application loads the types in spring-aspects.jar by using its own classloader (for example, by placing spring-aspects.jar in 'WEB-INF/lib'). If spring-aspects.jar is added only to the container-wide classpath (and hence loaded by the shared parent classloader), all web applications share the same aspect instance (which is probably not what you want).

5.10.2. Other Spring aspects for AspectJ

In addition to the @Configurable aspect, spring-aspects.jar contains an AspectJ aspect that you can use 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 you use this aspect, you must annotate the implementation class (or methods within that class or both), 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 who want to use the Spring configuration and transaction management support but do not 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 by 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);

}

5.10.3. Configuring AspectJ Aspects by Using Spring IoC

When you use AspectJ aspects with Spring applications, it is natural to both want and expect to be able to configure such aspects with Spring. The AspectJ runtime itself is responsible for aspect creation, and the means of configuring the AspectJ-created aspects through 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 easy. You can create a bean definition that references the aspect type as normal and include the factory-method="aspectOf" bean attribute. This ensures that Spring obtains the aspect instance by asking AspectJ for it rather than trying to create an instance itself. The following example shows how to use the factory-method="aspectOf" attribute:

<bean id="profiler" class="com.xyz.profiler.Profiler"

factory-method="aspectOf">

<property name="profilingStrategy" ref="jamonProfilingStrategy"/>

</bean>

|  |  |
| --- | --- |
|  | Note the factory-method="aspectOf" attribute |

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 in Spring, you need to tell the Spring AOP @AspectJ auto-proxying support which exact subset of the @AspectJ aspects defined in the configuration should be used for auto-proxying. 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 are used for Spring AOP auto-proxy configuration. The following example shows how to use <include/> elements:

<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 results in the creation of Spring AOP proxies. The @AspectJ style of aspect declaration is being used here, but the AspectJ runtime is not involved. |

5.10.4. 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 a general introduction to LTW. For full details on the specifics of LTW and configuring LTW with only 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 that the Spring Framework brings to AspectJ LTW is in enabling much finer-grained control over the weaving process. 'Vanilla' AspectJ LTW is effected by 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 is often a little too coarse. Spring-enabled LTW lets you switch on LTW on a per-ClassLoader basis, which 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/5.1.8.RELEASE/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 is needed to add -javaagent:path/to/aspectjweaver.jar or (as we describe later in this section) -javaagent:path/to/spring-instrument.jar. Developers configure 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 that uses Spring, followed by detailed specifics about elements introduced in the example. For a complete example, see the [Petclinic sample application](https://github.com/spring-projects/spring-petclinic).

A First Example

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, we are going to switch on a simple profiling aspect that lets us quickly get some performance metrics. We can then apply a finer-grained profiling tool to that specific area immediately afterwards.

|  |  |
| --- | --- |
|  | The example presented here uses XML configuration. You can also configure and use @AspectJ with [Java configuration](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-java). Specifically, you can use the @EnableLoadTimeWeaving annotation as an alternative to <context:load-time-weaver/> (see [below](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-aj-ltw-spring) for details). |

The following example shows the profiling aspect, which is not fancy. It is a time-based profiler that uses 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 also need to create an META-INF/aop.xml file, to inform the AspectJ weaver that we want to weave our ProfilingAspect into our classes. This file convention, namely the presence of a file (or files) on the Java classpath called META-INF/aop.xml is standard AspectJ. The following example shows the aop.xml file:

<!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN" "https://www.eclipse.org/aspectj/dtd/aspectj.dtd">

<aspectj>

<weaver>

*<!-- only weave classes in our application-specific packages -->*

<include within="foo.\*"/>

</weaver>

<aspects>

*<!-- weave in just this aspect -->*

<aspect name="foo.ProfilingAspect"/>

</aspects>

</aspectj>

Now we can move on to the Spring-specific portion of the configuration. We need to configure a LoadTimeWeaver (explained later). 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 (there are some more options that you can specify, but these are detailed later), as can be seen in the following example:

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

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

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

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

xsi:schemaLocation="

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

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

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

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

*<!-- 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 (the aspect, the META-INF/aop.xml file, and the Spring configuration) are in place, we can create the following 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();

}

}

We have 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, for this example, we use a Java agent (supplied with Spring) to switch on LTW. We use the following command to run the Main class shown earlier:

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 that run on the JVM](https://docs.oracle.com/javase/8/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 preceding example.

The output from the execution of the Main program looks something like the next example. (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). The following listing shows the output we got when we ran our profiler:

Calculating entitlement

StopWatch 'ProfilingAspect': running time (millis) = 1234

------ ----- ----------------------------

ms % Task name

------ ----- ----------------------------

01234 100% calculateEntitlement

Since this LTW is effected by using full-blown AspectJ, we are not limited only to advising Spring beans. The following slight variation on the Main program yields 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 preceding program, we bootstrap the Spring container and then create a new instance of the StubEntitlementCalculationService totally outside the context of Spring. The profiling advice still gets woven in.

Admittedly, the example is simplistic. However, the basics of the LTW support in Spring have all been introduced in the earlier example, and the rest of this section explains 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 and then 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. You can write them in either the AspectJ language itself, or you can write your aspects in the @AspectJ-style. 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 by 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 LTW part of the [AspectJ reference documentation](https://www.eclipse.org/aspectj/doc/released/devguide/ltw-configuration.html). Because the aop.xmlfile is 100% AspectJ, we do not describe it further here.

Required libraries (JARS)

At minimum, you need the following libraries to use the Spring Framework’s support for AspectJ LTW:

* spring-aop.jar
* aspectjweaver.jar

If you use the [Spring-provided agent to enable instrumentation](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-aj-ltw-environment-generic), you 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, see the javadoc API documentation for the java.lang.instrument package before continuing. While that documentation is not comprehensive, at least you can see the key interfaces and classes (for reference as you read through this section). |

Configuring a LoadTimeWeaver for a particular ApplicationContext can be as easy as adding one line. (Note that you almost certainly need to use an ApplicationContext as your Spring container — typically, a BeanFactory is not enough because the LTW support uses BeanFactoryPostProcessors.)

To enable the Spring Framework’s LTW support, you need to configure a LoadTimeWeaver, which typically is done by using the @EnableLoadTimeWeaving annotation, as follows:

@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. The following example shows how to use <context:load-time-weaver/>:

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

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

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

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

xsi:schemaLocation="

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

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

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

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

<context:load-time-weaver/>

</beans>

The preceding configuration automatically defines and registers a number of LTW-specific infrastructure beans, such as a LoadTimeWeaver and an AspectJWeavingEnabler, for you. The default LoadTimeWeaver is the DefaultContextLoadTimeWeaverclass, which attempts to decorate an automatically detected LoadTimeWeaver. The exact type of LoadTimeWeaver that is “automatically detected” is dependent upon your runtime environment. The following table summarizes various LoadTimeWeaverimplementations:

| *Table 13. DefaultContextLoadTimeWeaver LoadTimeWeavers* | |
| --- | --- |
| **Runtime Environment** | LoadTimeWeaver**implementation** |
| Running in [Apache Tomcat](https://tomcat.apache.org/) | TomcatLoadTimeWeaver |
| Running in [GlassFish](https://glassfish.dev.java.net/) (limited to EAR deployments) | GlassFishLoadTimeWeaver |
| Running in Red Hat’s [JBoss AS](https://www.jboss.org/jbossas/) or [WildFly](https://www.wildfly.org/) | JBossLoadTimeWeaver |
| Running in IBM’s [WebSphere](https://www-01.ibm.com/software/webservers/appserv/was/) | WebSphereLoadTimeWeaver |
| Running in Oracle’s [WebLogic](https://www.oracle.com/technetwork/middleware/weblogic/overview/index-085209.html) | WebLogicLoadTimeWeaver |
| JVM started with Spring InstrumentationSavingAgent (java -javaagent:path/to/spring-instrument.jar) | InstrumentationLoadTimeWeaver |
| Fallback, expecting the underlying ClassLoader to follow common conventions (namely addTransformer and optionally a getThrowawayClassLoader method) | ReflectiveLoadTimeWeaver |

Note that the table lists only the LoadTimeWeavers that are autodetected when you use the DefaultContextLoadTimeWeaver. You can specify exactly which LoadTimeWeaver implementation to use.

To specify a specific LoadTimeWeaver with Java configuration, implement the LoadTimeWeavingConfigurer interface and override the getLoadTimeWeaver() method. The following example specifies a ReflectiveLoadTimeWeaver:

@Configuration

@EnableLoadTimeWeaving

**public** **class** **AppConfig** **implements** LoadTimeWeavingConfigurer {

@Override

**public** LoadTimeWeaver getLoadTimeWeaver() {

**return** **new** ReflectiveLoadTimeWeaver();

}

}

If you use XML-based configuration, you can specify the fully qualified classname as the value of the weaver-class attribute on the <context:load-time-weaver/> element. Again, the following example specifies a ReflectiveLoadTimeWeaver:

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

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

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

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

xsi:schemaLocation="

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

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

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

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

<context:load-time-weaver

weaver-class="org.springframework.instrument.classloading.ReflectiveLoadTimeWeaver"/>

</beans>

The LoadTimeWeaver that is defined and registered by the configuration can be later retrieved from the Spring container by using the well known name, loadTimeWeaver. Remember that the LoadTimeWeaver exists only 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 javadoc of theClassPreProcessorAgentAdapter class for further details, because the specifics of how the weaving is actually effected is beyond the scope of this document.

There is one final attribute of the configuration left to discuss: the aspectjWeaving attribute (or aspectj-weaving if you use XML). This attribute controls whether LTW is enabled or not. It accepts one of three possible values, with the default value beingautodetect if the attribute is not present. The following table summarizes the three possible values:

| *Table 14. AspectJ weaving attribute values* | | |
| --- | --- | --- |
| **Annotation Value** | **XML Value** | **Explanation** |
| ENABLED | on | AspectJ weaving is on, and aspects are woven at load-time as appropriate. |
| DISABLED | off | LTW is off. No aspect is 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. Otherwise, it is off. This is the default value. |

Environment-specific Configuration

This last section contains any additional settings and configuration that you need when you use Spring’s LTW support in environments such as application servers and web containers.

Tomcat, JBoss, WebSphere, WebLogic

Tomcat, JBoss/WildFly, IBM WebSphere Application Server and Oracle WebLogic Server all provide a general app ClassLoaderthat is capable of local instrumentation. Spring’s native LTW may leverage those ClassLoader implementations to provide AspectJ weaving. You can simply enable load-time weaving, as [described earlier](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-using-aspectj). Specifically, you do not need to modify the JVM launch script to add -javaagent:path/to/spring-instrument.jar.

Note that on JBoss, you may need to disable the app server scanning 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 are not supported by specific LoadTimeWeaver implementations, a JVM agent is the general solution. For such cases, Spring provides InstrumentationLoadTimeWeaver which requires a Spring-specific (but very general) JVM agent, spring-instrument.jar, autodetected by common @EnableLoadTimeWeaving and <context:load-time-weaver/> setups.

To use it, you must start the virtual machine with the Spring agent by supplying the following JVM options:

-javaagent:/path/to/spring-instrument.jar

Note that this requires modification of the JVM launch script, which may prevent you from using this in application server environments (depending on your server and your operation policies). That said, for one-app-per-JVM deployments such as standalone Spring Boot applications, you typically control the entire JVM setup in any case.

5.11. Further Resources

More information on AspectJ can be found on the [AspectJ website](https://www.eclipse.org/aspectj).

*Eclipse AspectJ* by Adrian Colyer et. al. (Addison-Wesley, 2005) provides a comprehensive introduction and reference for the AspectJ language.

*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

The previous chapter described the Spring’s support for AOP with @AspectJ and schema-based aspect definitions. In this chapter, we discuss the lower-level Spring AOP APIs. For common applications, we recommend the use of Spring AOP with AspectJ pointcuts as described in the previous chapter.

6.1. Pointcut API in Spring

This section describes how Spring handles the crucial pointcut concept.

6.1.1. Concepts

Spring’s pointcut model enables pointcut reuse independent of advice types. You can target different advice with 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 follows:

**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 are matched. The following listing shows the ClassFilter interface definition:

**public** **interface** **ClassFilter** {

**boolean** matches(Class clazz);

}

The MethodMatcher interface is normally more important. The complete interface follows:

**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 ever matches 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 two-argument matches method returns true for a given method, and the isRuntime() method for the MethodMatcher returns true, the three-argument matches method is invoked on every method invocation. This lets a pointcut look at the arguments passed to the method invocation immediately before the target advice is to execute.

Most MethodMatcher implementations are static, meaning that their isRuntime() method returns false. In this case, the three-argument matches method is never 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. |

6.1.2. Operations on Pointcuts

Spring supports operations (notably, union and intersection) on pointcuts.

Union means the methods that either pointcut matches. Intersection means the methods that both pointcuts match. Union is usually more useful. You can compose pointcuts by using the static methods in theorg.springframework.aop.support.Pointcuts class or by using the ComposablePointcut class in the same package. However, using AspectJ pointcut expressions is usually a simpler approach.

6.1.3. 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](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop) for a discussion of supported AspectJ pointcut primitives.

6.1.4. Convenience Pointcut Implementations

Spring provides several convenient pointcut implementations. You can use some of them directly. Others are intended to be subclassed in application-specific pointcuts.

Static Pointcuts

Static pointcuts are based on the method and the target class and cannot take into account the method’s arguments. Static pointcuts suffice — and are best — for most usages. Spring can 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.

The rest of this section describes some of the static pointcut implementations that are 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 that uses the regular expression support in the JDK.

With the JdkRegexpMethodPointcut class, you can provide a list of pattern strings. If any of these is a match, the pointcut evaluates to true. (So, the result is effectively the union of these pointcuts.)

The following example shows how to use JdkRegexpMethodPointcut:

<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 named RegexpMethodPointcutAdvisor, which lets us also reference an Advice (remember that an Advice can be an interceptor, before advice, throws advice, and others). Behind the scenes, Spring uses a JdkRegexpMethodPointcut. Using RegexpMethodPointcutAdvisor simplifies wiring, as the one bean encapsulates both pointcut and advice, as the following example shows:

<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>

You can use RegexpMethodPointcutAdvisor 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 and that 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 by 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 five times that of other dynamic pointcuts. |

6.1.5. Pointcut Superclasses

Spring provides useful pointcut superclasses to help you to implement your own pointcuts.

Because static pointcuts are most useful, you should probably subclass StaticMethodMatcherPointcut. This requires implementing only one abstract method (although you can override other methods to customize behavior). The following example shows how to subclass StaticMethodMatcherPointcut:

**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.

6.1.6. Custom Pointcuts

Because pointcuts in Spring AOP are Java classes rather than language features (as in AspectJ), you can declare custom pointcuts, whether static or dynamic. Custom pointcuts in Spring can be arbitrarily complex. However, we recommend using the AspectJ pointcut expression language, if you can.

|  |  |
| --- | --- |
|  | 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.” |

6.2. Advice API in Spring

Now we can examine how Spring AOP handles advice.

6.2.1. Advice Lifecycles

Each advice is a Spring bean. An advice instance can be shared across all advised objects or be 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.

You can use a mix of shared and per-instance advice in the same AOP proxy.

6.2.2. Advice Types in Spring

Spring provides several advice types and is extensible to support arbitrary advice types. This section describes 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 that uses method interception. Classes that implement MethodInterceptor and that implement around advice should also 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.

The following example shows a simple MethodInterceptor implementation:

**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 proceed() method of MethodInvocation. This proceeds down the interceptor chain towards the join point. Most interceptors 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 do not want to do this without good reason.

|  |  |
| --- | --- |
|  | MethodInterceptor implementations 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 is called only 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 following listing shows the MethodBeforeAdvice interface:

**public** **interface** **MethodBeforeAdvice** **extends** BeforeAdvice {

**void** before(Method m, Object**[]** args, Object target) **throws** Throwable;

}

(Spring’s API design would allow for field before advice, although the usual objects apply to field interception and it is unlikely for Spring to ever implement it.)

Note that 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, it aborts further execution of the interceptor chain. The exception propagates back up the interceptor chain. If it is unchecked or on the signature of the invoked method, it is passed directly to the client. Otherwise, it is wrapped in an unchecked exception by the AOP proxy.

The following example shows 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 following form:

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 next two listing show classes that are examples of throws advice.

The following advice is invoked if a RemoteException is thrown (including from subclasses):

**public** **class** **RemoteThrowsAdvice** **implements** ThrowsAdvice {

**public** **void** afterThrowing(RemoteException ex) **throws** Throwable {

*// Do something with remote exception*

}

}

Unlike the preceding advice, the next example declares four arguments, so that it has access to the invoked method, method arguments, and target object. The following advice is invoked if a ServletException is thrown:

**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 that handles both RemoteException and ServletException. Any number of throws advice methods can be combined in a single class. The following listing shows the final example:

**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 overrides the original exception (that is, it changes the exception thrown to the user). The overriding exception is typically a RuntimeException, which is is compatible with any method signature. However, if a throws-advice method throws a checked exception, it must 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, which the following listing shows:

**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), the invoked method, the method’s arguments, and the 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 does not change the execution path. If it throws an exception, it is 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 that implement 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 the class, rather than the 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.

Consider an example from the Spring test suite and 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.

First, we 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 interfaces, concealing the use of interception to do so. You can set the delegate to any object using a constructor argument. The default delegate (when the no-argument constructor is used) is this. Thus, in the next example, 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 supports introductions against any of them. Subclasses such as LockMixin can 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, theIntroductionAdvisor used controls which interfaces are actually exposed. An introduced interface conceals 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 do not 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.

The following example shows the example LockMixin class:

**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, you need not override the invoke() method. The DelegatingIntroductionInterceptor implementation (which calls the delegate method if the method is introduced, otherwise proceeds towards the join point) usually suffices. In the present case, we need to add a check: no setter method can be invoked if in locked mode.

The required introduction only needs to hold a distinct LockMixin instance and specify the introduced interfaces (in this case, only Lockable). A more complex example might take a reference to the introduction interceptor (which would be defined as a prototype). In this case, there is no configuration relevant for a LockMixin, so we create it by using new. The following example shows our LockMixinAdvisor class:

**public** **class** **LockMixinAdvisor** **extends** DefaultIntroductionAdvisor {

**public** LockMixinAdvisor() {

super(**new** LockMixin(), Lockable.class);

}

}

We can apply this advisor very simply, because it requires no configuration. (However, it is 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 by using the Advised.addAdvisor() method or (the recommended way) in XML configuration, as any other advisor. All proxy creation choices discussed below, including “auto proxy creators,” correctly handle introductions and stateful mixins.

6.3. The Advisor API in Spring

In Spring, an Advisor is an aspect that contains only 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. 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 an interception around advice, throws advice, and before advice in one proxy configuration. Spring automatically creates the necessary interceptor chain.

6.4. Using the ProxyFactoryBean to Create AOP Proxies

If you use the Spring IoC container (an ApplicationContext or BeanFactory) for your business objects (and you should be!), you want to use one of Spring’s AOP FactoryBean implementations. (Remember that a factory bean introduces a layer of indirection, letting it 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, any advice that applies, and their ordering. However, there are simpler options that are preferable if you do not need such control.

6.4.1. Basics

The ProxyFactoryBean, like other Spring FactoryBean implementations, introduces a level of indirection. If you define a ProxyFactoryBean named foo, objects that reference foo do not see the ProxyFactoryBean instance itself but an object created by the implementation of the getObject() method in the ProxyFactoryBean . This method creates an AOP proxy that wraps a target object.

One of the most important benefits of using a ProxyFactoryBean or another IoC-aware class to create AOP proxies is 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.

6.4.2. 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 (described later and see also [JDK- and CGLIB-based proxies](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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 the following:

* proxyTargetClass: true if the target class is to be proxied, rather than the target class’s interfaces. If this property value is set to true, then CGLIB proxies are created (but see also [JDK- and CGLIB-based proxies](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-pfb-proxy-types)).
* optimize: Controls whether or not aggressive optimizations are applied to proxies created through CGLIB. You should not blithely use this setting unless you fully understand 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, changes to the configuration are no longer allowed. This is useful both as a slight optimization and for those cases when you do not want callers to be able to manipulate the proxy (through 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 the following:

* proxyInterfaces: An array of String interface names. If this is not supplied, a CGLIB proxy for the target class is used (but see also [JDK- and CGLIB-based proxies](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-pfb-proxy-types)).
* interceptorNames: A 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 is 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 cannot mention bean references here, since doing so results in the ProxyFactoryBean ignoring the singleton setting of the advice.

You can append an interceptor name with an asterisk (\*). Doing so results in the application of all advisor beans with names that start with the part before the asterisk to be applied. You can find an example of using this feature in [Using “Global” Advisors](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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.

6.4.3. JDK- and CGLIB-based proxies

This section serves as the definitive documentation on how the ProxyFactoryBean chooses to create either a JDK-based proxy or a CGLIB-based proxy for a particular target object (which 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) does not implement any interfaces, a CGLIB-based proxy is created. This is the easiest scenario, because JDK proxies are interface-based, and no interfaces means JDK proxying is not even possible. You can plug in the target bean and specify the list of interceptors by setting the interceptorNames property. Note that a CGLIB-based proxy is created even if the proxyTargetClass property of theProxyFactoryBean has been set to false. (Doing so 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, 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, a CGLIB-based proxy is created. This makes sense and is in keeping with the principle of least surprise. Even if the proxyInterfaces property of the ProxyFactoryBean has been set to one or more fully qualified interface names, the fact that the proxyTargetClass property is set to true causes 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, a JDK-based proxy is created. The created proxy implements 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 are not 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, the ProxyFactoryBean auto-detects the fact that the target class does actually implement at least one interface, and a JDK-based proxy is created. The interfaces that are actually proxied are all of the interfaces that the target class implements. In effect, this is the same as supplying a list of each and every interface that the target class implements to the proxyInterfacesproperty. However, it is significantly less work and less prone to typographical errors.

6.4.4. Proxying Interfaces

Consider a simple example of ProxyFactoryBean in action. This example involves:

* A target bean that is proxied. This is the personTarget bean definition in the example.
* An Advisor and an Interceptor used to provide advice.
* An AOP proxy bean definition to specify the target object (the personTarget bean), the interfaces to proxy, and the advices to apply.

The following listing shows the example:

<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, which holds the bean names of the interceptors or advisors in the current factory. You can use advisors, interceptors, before, after returning, and throws advice objects. The ordering of advisors is significant.

|  |  |
| --- | --- |
|  | You might be wondering why the list does not hold bean references. The reason for this is that, if the singleton property of the ProxyFactoryBean 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 is necessary to be able to obtain an instance of the prototype from the factory. Holding a reference is not sufficient. |

The person bean definition shown earlier 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. The following example shows how to do so:

<bean id="personUser" class="com.mycompany.PersonUser">

<property name="person"><ref bean="person"/></property>

</bean>

The PersonUser class in this example exposes a property of type Person. As far as it is 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 later).

You can conceal the distinction between target and proxy by using an anonymous inner bean. Only the ProxyFactoryBeandefinition is different. The advice is included only for completeness. The following example shows how to use an anonymous inner bean:

<bean id="myAdvisor" class="com.mycompany.MyAdvisor">

<property name="someProperty" value="Custom string property value"/>

</bean>

<bean id="debugInterceptor" class="org.springframework.aop.interceptor.DebugInterceptor"/>

<bean id="person" class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="proxyInterfaces" value="com.mycompany.Person"/>

*<!-- Use inner bean, not local reference to target -->*

<property name="target">

<bean class="com.mycompany.PersonImpl">

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

<property name="age" value="51"/>

</bean>

</property>

<property name="interceptorNames">

<list>

<value>myAdvisor</value>

<value>debugInterceptor</value>

</list>

</property>

</bean>

Using an anonymous inner bean has the advantage that there is only one object of type Person. This is 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 is 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).

6.4.5. Proxying Classes

What if you need to proxy a class, rather than one or more interfaces?

Imagine that in our earlier example, there was no Person interface. We needed to advise a class called Person that did not implement any business interface. In this case, you can configure Spring to use CGLIB proxying rather than dynamic proxies. To do so, set the proxyTargetClass property on the ProxyFactoryBean shown earlier to true. While it is best to program to interfaces rather than classes, the ability to advise classes that do not implement interfaces can be useful when working with legacy code. (In general, Spring is not 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 cannot be advised, as they cannot 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 works “out of the box”, as do JDK dynamic proxies.

There is little performance difference between CGLIB proxying and dynamic proxies. Performance should not be a decisive consideration in this case.

6.4.6. Using “Global” Advisors

By appending an asterisk to an interceptor name, all advisors with bean names that match the part before the asterisk are added to the advisor chain. This can come in handy if you need to add a standard set of “global” advisors. The following example defines two 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"/>

6.5. 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, we create a parent, template, bean definition for the proxy, as follows:

<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 is never instantiated itself, so it can actually be incomplete. Then, each proxy that needs to be created is a child bean definition, which wraps the target of the proxy as an inner bean definition, since the target is never used on its own anyway. The following example shows such a child bean:

<bean id="myService" parent="txProxyTemplate">

<property name="target">

<bean class="org.springframework.samples.MyServiceImpl">

</bean>

</property>

</bean>

You can override properties from the parent template. In the following example, we override 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 parent bean example, we explicitly marked the parent bean definition as being abstract by setting the abstractattribute to true, as described [previously](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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), by default, pre-instantiate all singletons. Therefore, it is important (at least for singleton beans) that, if you have a (parent) bean definition that 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 actually tries to pre-instantiate it.

6.6. Creating AOP Proxies Programmatically with the ProxyFactory

It is easy to create AOP proxies programmatically with Spring. This lets you use Spring AOP without dependency on Spring IoC.

The interfaces implemented by the target object are automatically proxied. The following listing shows creation of a proxy for a target object, with one interceptor and one advisor:

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 preceding example, or specify the interfaces to be proxied in an alternate constructor.

You can add advices (with interceptors as a specialized kind of advice), advisors, or both 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) that let you 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 you should in general. |

6.7. Manipulating Advised Objects

However you create AOP proxies, you can manipulate them BY using the org.springframework.aop.framework.Advisedinterface. Any AOP proxy can be cast to this interface, no matter which 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 returns 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 is the object that you added. If you added an interceptor or other advice type, Spring wrapped this in an advisor with a pointcut that always returns true. Thus, if you added a MethodInterceptor, the advisor returned for this index is a DefaultPointcutAdvisor that returns your MethodInterceptor and 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 is the generic DefaultPointcutAdvisor, which you can use with any advice or pointcut (but not for introductions).

By default, it is possible to add or remove advisors or interceptors even once a proxy has been created. The only restriction is that it is impossible to add or remove an introduction advisor, as existing proxies from the factory do not show the interface change. (You can obtain a new proxy from the factory to avoid this problem.)

The following example shows 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 is questionable whether it is 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). We 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 that we want to test. (For example, the advice can get inside a transaction created for that method, perhaps 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 that case, the Advised isFrozen() method returns true, and any attempts to modify advice through addition or removal results 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).

6.8. Using the "auto-proxy" facility

So far, we have considered explicit creation of AOP proxies by using a ProxyFactoryBean or similar factory bean.

Spring also lets us use “auto-proxy” bean definitions, which can automatically proxy selected bean definitions. This is built on Spring’s “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 lets you declare the targets eligible for auto-proxying. You need not use ProxyFactoryBean.

There are two ways to do this:

* By 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.

6.8.1. Auto-proxy Bean Definitions

This section covers the auto-proxy creators provided by the org.springframework.aop.framework.autoproxy package.

BeanNameAutoProxyCreator

The BeanNameAutoProxyCreator class is a BeanPostProcessor that automatically creates AOP proxies for beans with names that match literal values or wildcards. The following example shows how to create a BeanNameAutoProxyCreator bean:

<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">

<property name="beanNames" value="jdk\*,onlyJdk"/>

<property name="interceptorNames">

<list>

<value>myInterceptor</value>

</list>

</property>

</bean>

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 preceding example, are plain old bean definitions with the target class. An AOP proxy is automatically created by the BeanNameAutoProxyCreator. The same advice is applied to all matching beans. Note that, if advisors are used (rather than the interceptor in the preceding example), the pointcuts may apply differently to different beans.

DefaultAdvisorAutoProxyCreator

A more general and extremely powerful auto-proxy creator is DefaultAdvisorAutoProxyCreator. This automagically applies 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 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 automatically evaluates 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 is not proxied. As bean definitions are added for new business objects, they are automatically proxied if necessary.

Auto-proxying 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 returns an AOP proxy, not the target business object. (The “inner bean” idiom shown earlier also offers this benefit.)

The following example creates a DefaultAdvisorAutoProxyCreator bean and the other elements discussed in this section:

<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 add new business objects without including specific proxy configuration. You can also easily drop in additional aspects (for example, tracing or performance monitoring aspects) with minimal change to configuration.

The DefaultAdvisorAutoProxyCreator offers support for filtering (by using a naming convention so that only certain advisors are evaluated, which allows the 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 preceding example has a configurable order value. The default setting is unordered.

6.9. Using TargetSource Implementations

Spring offers the concept of a TargetSource, expressed in the org.springframework.aop.TargetSource interface. This interface is responsible for returning the “target object” that implements the join point. The TargetSource implementation is asked for a target instance each time the AOP proxy handles a method invocation.

Developers who use Spring AOP do not normally need to work directly with TargetSource implementations, but this provides a powerful means of supporting pooling, hot swappable, and other sophisticated targets. For example, a pooling TargetSourcecan return a different target instance for each invocation, by using a pool to manage instances.

If you do not specify a TargetSource, a default implementation is used to wrap a local object. The same target is returned for each invocation (as you would expect).

The rest of this section describes 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. |

6.9.1. Hot-swappable Target Sources

The org.springframework.aop.target.HotSwappableTargetSource exists to let the target of an AOP proxy be switched while letting callers keep their references to it.

Changing the target source’s target takes effect immediately. The HotSwappableTargetSource is thread-safe.

You can change the target by using the swap() method on HotSwappableTargetSource, as the follow example shows:

HotSwappableTargetSource swapper = (HotSwappableTargetSource) beanFactory.getBean("swapper");

Object oldTarget = swapper.swap(newTarget);

The following example shows the required XML definitions:

<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 preceding swap() call changes the target of the swappable bean. Clients that hold a reference to that bean are unaware of the change but immediately start hitting the new target.

Although this example does not add any advice (it is not necessary to add advice to use a TargetSource), any TargetSource can be used in conjunction with arbitrary advice.

6.9.2. 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 support for Commons Pool 2.2, which provides a fairly efficient pooling implementation. You need the commons-pool Jar on your application’s classpath to use this feature. You can also subclassorg.springframework.aop.target.AbstractPoolingTargetSource to support any other pooling API.

|  |  |
| --- | --- |
|  | Commons Pool 1.5+ is also supported but is deprecated as of Spring Framework 4.2. |

The following listing shows an example configuration:

<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 preceding example) must be a prototype. This lets the PoolingTargetSource implementation create new instances of the target to grow the pool as necessary. See the [javadoc ofAbstractPoolingTargetSource](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframeworkaop/target/AbstractPoolingTargetSource.html) and the concrete subclass you wish to use for information about its properties. maxSize is the most basic and is 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, you need not specify interceptors to use pooling. If you want only pooling and no other advice, do not set the interceptorNamesproperty at all.

You can configure Spring to be able to cast any pooled object to the org.springframework.aop.target.PoolingConfig interface, which exposes information about the configuration and current size of the pool through an introduction. You need to define an advisor similar to the following:

<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 that exposes the pooled object.

The cast is defined 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 do not 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 by using auto-proxying. You can set the TargetSource implementations used by any auto-proxy creator.

6.9.3. Prototype Target Sources

Setting up a “prototype” target source is similar to setting up a pooling TargetSource. In this case, a new instance of the target is created on every method invocation. Although the cost of creating a new object is not high in a modern JVM, the cost of wiring up the new object (satisfying its IoC dependencies) may be more expensive. Thus, you should not use this approach without very good reason.

To do this, you could modify the poolTargetSource definition shown earlier as follows (we also changed the name, for clarity):

<bean id="prototypeTargetSource" class="org.springframework.aop.target.PrototypeTargetSource">

<property name="targetBeanName" ref="businessObjectTarget"/>

</bean>

The only property is 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.

6.9.4. 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 provides a JDK-wide facility to transparently store a resource alongside a thread. Setting up aThreadLocalTargetSource is pretty much the same as was explained for the other types of target source, as the following example shows:

<bean id="threadlocalTargetSource" class="org.springframework.aop.target.ThreadLocalTargetSource">

<property name="targetBeanName" value="businessObjectTarget"/>

</bean>

|  |  |
| --- | --- |
|  | ThreadLocal instances come with serious issues (potentially resulting in memory leaks) when incorrectly using them in multi-threaded and multi-classloader environments. You should always consider wrapping a threadlocal in some other class and never directly use the ThreadLocal itself (except in the wrapper class). Also, you should always remember to correctly set and unset (where the latter simply involves a call to ThreadLocal.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 ThreadLocal instances without other proper handling code. |

6.10. 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 interception around advice, before, throws advice, and after returning advice.

The org.springframework.aop.framework.adapter package is an SPI package that lets support for new custom advice types 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.

See the [org.springframework.aop.framework.adapter](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/aop/framework/adapter/package-frame.html) javadoc for further information.

7. Null-safety

Although Java does not let you express null-safety with its type system, the Spring Framework now provides the following annotations in the org.springframework.lang package to let you declare nullability of APIs and fields:

* [@Nullable](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/lang/Nullable.html): Annotation to indicate that a specific parameter, return value, or field can be null.
* [@NonNull](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/lang/NonNull.html): Annotation to indicate that a specific parameter, return value, or field cannot be null (not needed on parameters / return values and fields where @NonNullApi and @NonNullFields apply, respectively).
* [@NonNullApi](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/lang/NonNullApi.html): Annotation at the package level that declares non-null as the default semantics for parameters and return values.
* [@NonNullFields](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/lang/NonNullFields.html): Annotation at the package level that declares non-null as the default semantics for fields.

The Spring Framework itself leverages 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 declarations are expected to be fine-tuned between Spring Framework releases, including minor ones. Nullability of types used inside method bodies is outside of the scope of this feature.

|  |  |
| --- | --- |
|  | Other common libraries such as Reactor and Spring Data provide null-safe APIs that use a similar nullability arrangement, delivering a consistent overall experience for Spring application developers. |

7.1. Use cases

In addition to providing an explicit declaration for Spring Framework API nullability, these annotations can be used by an IDE (such as IDEA or Eclipse) to provide useful warnings related to null-safety in order to avoid NullPointerException at 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 the [Kotlin support documentation](https://docs.spring.io/spring/docs/5.1.8.RELEASE/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 wide-spread JSR). JSR-305 meta-annotations let tooling vendors like IDEA or Kotlin 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 a JSR-305 dependency to the project classpath to take advantage of Spring null-safe API. Only projects such as Spring-based libraries that use null-safety annotations in their codebase should add com.google.code.findbugs:jsr305:3.0.2 with compileOnly Gradle configuration or Maven provided scope to avoid compile warnings.

8. Data Buffers and Codecs

Java NIO provides ByteBuffer but many libraries build their own byte buffer API on top, especially for network operations where reusing buffers and/or using direct buffers is beneficial for performance. For example Netty has the ByteBuf hierarchy, Undertow uses XNIO, Jetty uses pooled byte buffers with a callback to be released, and so on. The spring-core module provides a set of abstractions to work with various byte buffer APIs as follows:

* [DataBufferFactory](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#databuffers-factory) abstracts the creation of a data buffer.
* [DataBuffer](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#databuffers-buffer) represents a byte buffer, which may be [pooled](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#databuffers-buffer-pooled).
* [DataBufferUtils](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#databuffers-utils) offers utility methods for data buffers.
* [Codecs](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#codecs) decode or encode streams data buffer streams into higher level objects.

8.1. DataBufferFactory

DataBufferFactory is used to create data buffers in one of two ways:

1. Allocate a new data buffer, optionally specifying capacity upfront, if known, which is more efficient even though implementations of DataBuffer can grow and shrink on demand.
2. Wrap an existing byte[] or java.nio.ByteBuffer, which decorates the given data with a DataBuffer implementation and that does not involve allocation.

Note that WebFlux applications do not create a DataBufferFactory directly but instead access it through the ServerHttpResponse or the ClientHttpRequest on the client side. The type of factory depends on the underlying client or server, e.g. NettyDataBufferFactory for Reactor Netty, DefaultDataBufferFactory for others.

8.2. DataBuffer

The DataBuffer interface offers similar operations as java.nio.ByteBuffer but also brings a few additional benefits some of which are inspired by the Netty ByteBuf. Below is a partial list of benefits:

* Read and write with independent positions, i.e. not requiring a call to flip() to alternate between read and write.
* Capacity expanded on demand as with java.lang.StringBuilder.
* Pooled buffers and reference counting via [PooledDataBuffer](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#databuffers-buffer-pooled).
* View a buffer as java.nio.ByteBuffer, InputStream, or OutputStream.
* Determine the index, or the last index, for a given byte.

8.3. PooledDataBuffer

As explained in the Javadoc for [ByteBuffer](https://docs.oracle.com/javase/8/docs/api/java/nio/ByteBuffer.html), byte buffers can be direct or non-direct. Direct buffers may reside outside the Java heap which eliminates the need for copying for native I/O operations. That makes direct buffers particularly useful for receiving and sending data over a socket, but they’re also more expensive to create and release, which leads to the idea of pooling buffers.

PooledDataBuffer is an extension of DataBuffer that helps with reference counting which is essential for byte buffer pooling. How does it work? When a PooledDataBuffer is allocated the reference count is at 1. Calls to retain() increment the count, while calls to release() decrement it. As long as the count is above 0, the buffer is guaranteed not to be released. When the count is decreased to 0, the pooled buffer can be released, which in practice could mean the reserved memory for the buffer is returned to the memory pool.

Note that instead of operating on PooledDataBuffer directly, in most cases it’s better to use the convenience methods in DataBufferUtils that apply release or retain to a DataBuffer only if it is an instance of PooledDataBuffer.

8.4. DataBufferUtils

DataBufferUtils offers a number of utility methods to operate on data buffers:

* Join a stream of data buffers into a single buffer possibly with zero copy, e.g. via composite buffers, if that’s supported by the underlying byte buffer API.
* Turn InputStream or NIO Channel into Flux<DataBuffer>, and vice versa a Publisher<DataBuffer> into OutputStream or NIO Channel.
* Methods to release or retain a DataBuffer if the buffer is an instance of PooledDataBuffer.
* Skip or take from a stream of bytes until a specific byte count.

8.5. Codecs

The org.springframework.core.codec package provides the following strategy interfaces:

* Encoder to encode Publisher<T> into a stream of data buffers.
* Decoder to decode Publisher<DataBuffer> into a stream of higher level objects.

The spring-core module provides byte[], ByteBuffer, DataBuffer, Resource, and String encoder and decoder implementations. The spring-web module adds Jackson JSON, Jackson Smile, JAXB2, Protocol Buffers and other encoders and decoders. See [Codecs](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/web-reactive.html#webflux-codecs) in the WebFlux section.

8.6. Using DataBuffer

When working with data buffers, special care must be taken to ensure buffers are released since they may be [pooled](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#databuffers-buffer-pooled). We’ll use codecs to illustrate how that works but the concepts apply more generally. Let’s see what codecs must do internally to manage data buffers.

A Decoder is the last to read input data buffers, before creating higher level objects, and therefore it must release them as follows:

1. If a Decoder simply reads each input buffer and is ready to release it immediately, it can do so via DataBufferUtils.release(dataBuffer).
2. If a Decoder is using Flux or Mono operators such as flatMap, reduce, and others that prefetch and cache data items internally, or is using operators such as filter, skip, and others that leave out items, thendoOnDiscard(PooledDataBuffer.class, DataBufferUtils::release) must be added to the composition chain to ensure such buffers are released prior to being discarded, possibly also as a result an error or cancellation signal.
3. If a Decoder holds on to one or more data buffers in any other way, it must ensure they are released when fully read, or in case an error or cancellation signals that take place before the cached data buffers have been read and released.

Note that DataBufferUtils#join offers a safe and efficient way to aggregate a data buffer stream into a single data buffer. Likewise skipUntilByteCount and takeUntilByteCount are additional safe methods for decoders to use.

An Encoder allocates data buffers that others must read (and release). So an Encoder doesn’t have much to do. However an Encoder must take care to release a data buffer if a serialization error occurs while populating the buffer with data. For example:

DataBuffer buffer = factory.allocateBuffer();

**boolean** release = true;

**try** {

*// serialize and populate buffer..*

release = false;

}

**finally** {

**if** (release) {

DataBufferUtils.release(buffer);

}

}

**return** buffer;

The consumer of an Encoder is responsible for releasing the data buffers it receives. In a WebFlux application, the output of the Encoder is used to write to the HTTP server response, or to the client HTTP request, in which case releasing the data buffers is the responsibility of the code writing to the server response, or to the client request.

Note that when running on Netty, there are debugging options for [troubleshooting buffer leaks](https://github.com/netty/netty/wiki/Reference-counted-objects#troubleshooting-buffer-leaks).

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 so forth. 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 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 https://www.springframework.org/schema/beans/spring-beans.xsd

*http://www.springframework.org/schema/util https://www.springframework.org/schema/util/spring-util.xsd"*>

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

</beans>

Using <util:constant/>

Consider the following bean definition:

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

<property name="isolation">

<bean id="java.sql.Connection.TRANSACTION\_SERIALIZABLE"

class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean" />

</property>

</bean>

The preceding 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 verbose and (unnecessarily) exposes Spring’s internal plumbing to the end user.

The following XML Schema-based version is more concise, clearly expresses the developer’s intent (“inject this constant value”), and it 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 Argument from a Field Value

[FieldRetrievingFactoryBean](https://docs.spring.io/spring-framework/docs/5.1.8.RELEASE/javadoc-api/org/springframework/beans/factory/config/FieldRetrievingFactoryBean.html) is a FactoryBean that 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 argument for another bean.

The following example shows how a static field is exposed, by using the [staticField](https://docs.spring.io/spring-framework/docs/5.1.8.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, as the following example shows:

<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 also has 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 does not have to be specified for the bean reference, as the following example shows:

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

<property name="isolation">

<bean id="java.sql.Connection.TRANSACTION\_SERIALIZABLE"

class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean" />

</property>

</bean>

You can also 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/5.1.8.RELEASE/javadoc-api/org/springframework/beans/factory/config/FieldRetrievingFactoryBean.html) class.

Injecting enumeration values into beans as either property or constructor arguments is easy to do in Spring. You do not actually have to do anything or know anything about the Spring internals (or even about classes such as the FieldRetrievingFactoryBean). The following example enumeration shows how easy injecting an enum value is:

**package** javax.persistence;

**public** **enum** PersistenceContextType {

TRANSACTION,

EXTENDED

}

Now consider the following setter of type PersistenceContextType and the corresponding bean definition:

**package** example;

**public** **class** **Client** {

**private** PersistenceContextType persistenceContextType;

**public** **void** setPersistenceContextType(PersistenceContextType type) {

this.persistenceContextType = type;

}

}

<bean class="example.Client">

<property name="persistenceContextType" value="TRANSACTION"/>

</bean>

Using <util:property-path/>

Consider the following example:

*<!-- target bean to be referenced by name -->*

<bean id="testBean" class="org.springframework.beans.TestBean" scope="prototype">

<property name="age" value="10"/>

<property name="spouse">

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

<property name="age" value="11"/>

</bean>

</property>

</bean>

*<!-- results in 10, which is the value of property 'age' of bean 'testBean' -->*

<bean id="testBean.age" class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>

The preceding 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.

Now consider the following example, which adds a <util:property-path/> element:

*<!-- target bean to be referenced by name -->*

<bean id="testBean" class="org.springframework.beans.TestBean" scope="prototype">

<property name="age" value="10"/>

<property name="spouse">

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

<property name="age" value="11"/>

</bean>

</property>

</bean>

*<!-- results in 10, which is the value of property 'age' of bean 'testBean' -->*

<util:property-path id="name" path="testBean.age"/>

The value of the path attribute of the <property-path/> element follows the form of beanName.beanProperty. In this case, it picks up the age property of the bean named testBean. The value of that age property is 10.

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 by a bean name. You can then use this value in another bean definition as a property value or constructor argument.

The following example shows a path being 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>

// results in 11, which is the value of property 'spouse.age' of bean 'person'

<bean id="theAge"

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

<property name="targetBeanName" value="person"/>

<property name="propertyPath" value="spouse.age"/>

</bean>

In the following example, a path is evaluated against an inner bean:

*<!-- results in 12, which is the value of property 'age' of the inner bean -->*

<bean id="theAge"

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

<property name="targetObject">

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

<property name="age" value="12"/>

</bean>

</property>

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

</bean>

There is also a shortcut form, where the bean name is the property path. The following example shows the shortcut form:

*<!-- results 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 also has to use the same id, which is the path. If used as an inner bean, there is no need to refer to it at all, as the following example shows:

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

<property name="age">

<bean id="person.age"

class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>

</property>

</bean>

You can specifically set the result type in the actual definition. This is not necessary for most use cases, but it can sometimes be useful. See the javadoc for more info on this feature.

Using <util:properties/>

Consider the following example:

*<!-- 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 preceding 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/5.1.8.RELEASE/spring-framework-reference/core.html#resources) location).

The following example uses a util:properties element to make a more concise representation:

*<!-- creates a java.util.Properties instance with values loaded from the supplied location -->*

<util:properties id="jdbcConfiguration" location="classpath:com/foo/jdbc-production.properties"/>

Using <util:list/>

Consider the following example:

*<!-- 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 preceding configuration uses a Spring FactoryBean implementation (the ListFactoryBean) to create a java.util.Listinstance and initialize it with values taken from the supplied sourceList.

The following example uses a <util:list/> element to make a more concise representation:

*<!-- 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 is instantiated and populated by using the list-class attribute 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, the container chooses a List implementation.

Using <util:map/>

Consider the following example:

*<!-- 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 preceding configuration uses a Spring FactoryBean implementation (the MapFactoryBean) to create a java.util.Mapinstance initialized with key-value pairs taken from the supplied 'sourceMap'.

The following example uses a <util:map/> element to make a more concise representation:

*<!-- 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 is instantiated and populated by using 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, the container chooses a Map implementation.

Using <util:set/>

Consider the following example:

*<!-- 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 preceding configuration uses a Spring FactoryBean implementation (the SetFactoryBean) to create a java.util.Setinstance initialized with values taken from the supplied sourceSet.

The following example uses a <util:set/> element to make a more concise representation:

*<!-- 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 is instantiated and populated by using 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, the container chooses a Set implementation.

9.1.2. The aop Schema

The aop tags deal with configuring all things AOP in Spring, including 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/5.1.8.RELEASE/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 snippet references the correct schema so that the tags in the aop 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:aop="http://www.springframework.org/schema/aop"* xsi:schemaLocation="

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

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

*<!-- bean 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 the “grunt” work in Spring, such as BeanfactoryPostProcessors. The following snippet references the correct schema so that the elements 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 https://www.springframework.org/schema/beans/spring-beans.xsd

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

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

</beans>

Using <property-placeholder/>

This element activates the replacement of ${…​} placeholders, which are resolved against a specified properties file (as a [Spring resource location](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#resources)). This element is a convenience mechanism that sets up a [PropertyPlaceholderConfigurer](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-placeholderconfigurer) for you. If you need more control over the PropertyPlaceholderConfigurer, you can explicitly define one yourself.

Using <annotation-config/>

This element activates the Spring infrastructure to detect annotations in bean classes:

* Spring’s [@Configuration](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-metadata) model
* [@Autowired/@Inject](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-annotation-config) and @Value
* JSR-250’s @Resource, @PostConstruct and @PreDestroy (if available)
* JPA’s @PersistenceContext and @PersistenceUnit (if available)
* Spring’s [@EventListener](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#context-functionality-events-annotation)

Alternatively, you can choose to explicitly activate the individual BeanPostProcessors for those annotations.

|  |  |
| --- | --- |
|  | This element does not activate processing of Spring’s [@Transactional](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/data-access.html#transaction-declarative-annotations) annotation; you can use the [<tx:annotation-driven/>](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/data-access.html#tx-decl-explained) element for that purpose. Similarly, Spring’s [caching annotations](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/integration.html#cache-annotations) need to be explicitly[enabled](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/integration.html#cache-annotation-enable) as well. |

Using <component-scan/>

This element is detailed in the section on [annotation-based container configuration](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-annotation-config).

Using <load-time-weaver/>

This element is detailed in the section on [load-time weaving with AspectJ in the Spring Framework](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-aj-ltw).

Using <spring-configured/>

This element is detailed in the section on [using AspectJ to dependency inject domain objects with Spring](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#aop-atconfigurable).

Using <mbean-export/>

This element is detailed in the section on [configuring annotation-based MBean export](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/integration.html#jmx-context-mbeanexport).

9.1.4. The Beans Schema

Last but not least, we have the elements in the beans schema. These elements have been in Spring since the very dawn of the framework. Examples of the various elements 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/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-properties-detailed) (and, indeed, in that entire [chapter](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans)).

Note that you can 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 write your own custom elements as described in the appendix entitled [XML Schema Authoring](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#xml-custom)).

The following example shows the <meta/> element in the context of a surrounding <bean/> (note that, without any logic to interpret it, the metadata is effectively useless as it stands).

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

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

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

xsi:schemaLocation="

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

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

<meta key="cacheName" value="foo"/>

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

</bean>

</beans>

|  |  |
| --- | --- |
|  | This is the example meta element |

In the case of the preceding example, you could assume that there is some logic that consumes the bean definition and sets up some caching infrastructure that uses the supplied metadata.

9.2. XML Schema Authoring

Since version 2.0, Spring has featured a mechanism for adding schema-based extensions to the basic Spring XML format for defining and configuring beans. This section covers how to write your own custom XML bean definition parsers and integrate such parsers into the Spring IoC container.

To facilitate authoring configuration files that use 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, you should first read the appendix entitled [[xsd-config]](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#xsd-config).

To create new XML configuration extensions:

1. [Author](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#xsd-custom-schema) an XML schema to describe your custom element(s).
2. [Code](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#xsd-custom-namespacehandler) a custom NamespaceHandler implementation.
3. [Code](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#xsd-custom-parser) one or more BeanDefinitionParser implementations (this is where the real work is done).
4. [Register](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#xsd-custom-registration) your new artifacts with Spring.

For a unified example, we create an XML extension (a custom XML element) that lets us configure objects of the typeSimpleDateFormat (from the java.text package). When we are done, we will be able to define bean definitions of type SimpleDateFormat as follows:

<myns:dateformat id="dateFormat"

pattern="yyyy-MM-dd HH:mm"

lenient="true"/>

(We include much more detailed examples follow later in this appendix. The intent of this first simple example is to walk you through the basic steps of making a custom extension.)

9.2.1. 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. For our example, we use the following schema 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 indicated line contains an extension base for all identifiable tags (meaning they have an id attribute that we can use as the bean identifier in the container). We can use this attribute because we imported the Spring-providedbeans namespace. |

The preceding schema lets us configure SimpleDateFormat objects directly in an XML application context file by using the <myns:dateformat/> element, as the following example shows:

<myns:dateformat id="dateFormat"

pattern="yyyy-MM-dd HH:mm"

lenient="true"/>

Note that, after we have created the infrastructure classes, the preceding snippet of XML is essentially the same as the following XML snippet:

<bean id="dateFormat" class="java.text.SimpleDateFormat">

<constructor-arg value="yyyy-HH-dd HH:mm"/>

<property name="lenient" value="true"/>

</bean>

The second of the two preceding snippets creates a bean in the container (identified by the name dateFormat of typeSimpleDateFormat) with a couple of properties set.

|  |  |
| --- | --- |
|  | The schema-based approach to creating configuration format allows for tight integration with an IDE that has a schema-aware XML editor. By using a properly authored schema, you can use autocompletion to let a user choose between several configuration options defined in the enumeration. |

9.2.2. Coding a NamespaceHandler

In addition to the schema, we need a NamespaceHandler to parse all elements of this specific namespace that Spring encounters while parsing configuration files. For this example, the NamespaceHandler should take care of the parsing of the myns:dateformat element.

The NamespaceHandler interface features three methods:

* init(): Allows for initialization of the NamespaceHandler and is 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 itself register bean definitions, return a bean definition, or both.
* 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[scopes that Spring supports](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#beans-factory-scopes). We start by highlighting a simple example, without using decoration, after which we show decoration in a somewhat more advanced example.

Although you can 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 the following example, we 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());

}

}

You may notice that there is not 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 delegates to when it needs to parse an element in its namespace. This clean separation of concerns lets a NamespaceHandlerhandle 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 contains only the logic for parsing a single custom element, as we can see in the next step.

9.2.3. Using BeanDefinitionParser

A BeanDefinitionParser is used if the NamespaceHandler encounters an XML element of the type that has been mapped to the specific bean definition parser (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' have access to the XML element (and thus to its subelements, too) so that we can parse our custom XML content, as you can see 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 BeanDefinitionrepresents. |

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.

9.2.4. Registering the Handler and the Schema

The coding is finished. All that remains to be done is to make the Spring XML parsing infrastructure aware of our custom element. We do so 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 automatically picks up your new extension by consuming these special properties files, the formats of which are detailed in the next two sections.

Writing META-INF/spring.handlers

The properties file called spring.handlers contains a mapping of XML Schema URIs to namespace handler classes. 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, so : 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 exactly match exactly the value of the targetNamespace attribute, as specified in your custom XSD schema.

Writing '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 searches for the schema (in this case, myns.xsd in the org.springframework.samples.xml package) on the classpath. The following snippet shows the line we need to add for our custom schema:

http\://www.mycompany.com/schema/myns/myns.xsd=org/springframework/samples/xml/myns.xsd

(Remember that the : character must be escaped.)

You are encouraged to deploy your XSD file (or files) right alongside the NamespaceHandler and BeanDefinitionParser classes on the classpath.

9.2.5. 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. The following example uses 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 https://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>

|  |  |
| --- | --- |
|  | Our custom bean. |

9.2.6. More Detailed Examples

This section presents some more detailed examples of custom XML extensions.

Nesting Custom Elements within Custom Elements

The example presented in this section shows how you to write 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 https://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 preceding configuration nests custom extensions within each other. The class that is actually configured by the <foo:component/> element is the Component class (shown in the next example). 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 by using setter injection. The following listing shows the Component class:

**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. The following listing shows such a custom FactoryBean:

**package** com.foo;

**import** org.springframework.beans.factory.FactoryBean;

**import** java.util.List;

**public** **class** **ComponentFactoryBean** **implements** FactoryBean<Component> {

**private** Component parent;

**private** List<Component> children;

**public** **void** setParent(Component parent) {

this.parent = parent;

}

**public** **void** setChildren(List<Component> children) {

this.children = children;

}

**public** Component getObject() **throws** Exception {

**if** (this.children != null && this.children.size() > 0) {

**for** (Component child : children) {

this.parent.addComponent(child);

}

}

**return** this.parent;

}

**public** Class<Component> getObjectType() {

**return** Component.class;

}

**public** **boolean** isSingleton() {

**return** true;

}

}

This works nicely, but it 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/5.1.8.RELEASE/spring-framework-reference/core.html#xsd-custom-introduction), we start off by creating the XSD schema to define the structure of our custom tag, as the following listing shows:

<?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>

Again following [the process described earlier](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html#xsd-custom-introduction), we 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 we are creating a BeanDefinition that describes a ComponentFactoryBean. The following listing shows our custom BeanDefinitionParser implementation:

**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);

}

}

Finally, the various artifacts need to be registered with the Spring XML infrastructure, by modifying the META-INF/spring.handlers and META-INF/spring.schemas files, as follows:

# 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 is not hard. However, it is sometimes not the right thing to do. Consider a scenario where you need to add metadata to already existing bean definitions. In this case, you certainly do not want to have to write your own entire custom extension. Rather, you merely want to add an additional attribute to the existing bean definition element.

By way of another example, suppose that you define a bean definition for a service object that (unknown to it) accesses 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. The following listing shows such a definition:

<bean id="checkingAccountService" class="com.foo.DefaultCheckingAccountService"

jcache:cache-name="checking.account">

*<!-- other dependencies here... -->*

</bean>

We can then create another BeanDefinition when the 'jcache:cache-name' attribute is parsed. This BeanDefinition then initializes the named JCache for us. We can also modify the existing BeanDefinition for the 'checkingAccountService' so that it has a dependency on this new JCache-initializing BeanDefinition. The following listing shows our JCacheInitializer:

**package** com.foo;

**public** **class** **JCacheInitializer** {

**private** String name;

**public** JCacheInitializer(String name) {

this.name = name;

}

**public** **void** initialize() {

*// lots of JCache API calls to initialize the named cache...*

}

}

Now we can move onto the custom extension. First, we need to author the XSD schema that describes the custom attribute, as follows:

<?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, we need to create the associated NamespaceHandler, as follows:

**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, we need to create the parser. Note that, in this case, because we are going to parse an XML attribute, we write a BeanDefinitionDecorator rather than a BeanDefinitionParser. The following listing shows our BeanDefinitionDecoratorimplementation:

**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;

}

}

Finally, we need to register the various artifacts with the Spring XML infrastructure by modifying the META-INF/spring.handlersand META-INF/spring.schemas files, as follows:

# 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