

参考文档的这一部分涵盖了 Spring 框架绝对必要的所有技术。

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

还提供了 Spring 与 AspectJ 的集成（目前，在功能上最丰富）以及 Java 企业领域中最成熟的 AOP 实现）。

1. IoC 容器

这一章节涵盖了 Spring 的控制反转容器

1.1. Spring IoC 容器和 Beans 的介绍

这一章涵盖了 Spring Framework IoC 原理的实现。IoC 以依赖注入而闻名。借以对象定义他们的依赖(也就是那些他们一起工作的其他类)仅仅是通过给工厂方法构造器参数, 参数或者那些在它们从工厂方法被构造或返回后设置在对象实例上的属性。当容器创造这些 Bean 的时候接下来会注入那些依赖。这个过程从根本上反转(因此叫控制反转)了 Bean 本身通过直接类构造器或者注入服务定位器模式机制来控制实例化或它的依赖。

`org.springframework.beans` 和 `org.springframework.context` 包是 Spring Framework 的 IoC 容器的基础。`BeanFactory` 接口提供了一个管理任何类型的高级的配置机制能力。`ApplicationContext` 是 `BeanFactory` 的一个子接口。它增加了:

- 与 Spring AOP 特性更简单地集成
- 消息源处理(为了在国际化中使用)
- 事件发布
- 应用层特定上下文, 比如用在 Web 应用中的 `WebApplicationContext`

简单来说, `BeanFactory` 提供了配置框架和基础功能性, `ApplicationContext` 增加了更多企业特定功能。`ApplicationContext` 是一个 `BeanFactory` 的超集, 并仅仅使用在本章 Spring IoC 容器的描述中。想获取更多的 `BeanFactory` 使用信息而不是 `ApplicationContext` 的, 看 `BeanFactory`。

在 Spring 中, 那些构成你应用主体的对象和那些被 Spring IoC 容器管理的对象被称为 beans。一个 bean 是一个被实例化组装好的并被 Spring IoC 容器管理对象。反之, 一个 bean 只是你应用程序众多对象中的一个。Beans 和依赖, 被反射在一个容器所使用的配置元数据中。

1.2. 容器概述

`org.springframework.context.ApplicationContext` 接口代表了 Spring IoC 容器的能力, 它负责实例化, 配置, 组装这些 beans。容器通过读取配置元数据来获取有关要实例化, 配置和组装哪些对象的指令。配置元数据表现为 XML, Java 注解或者 Java 代码。

它使你能够表达组成你的应用程序的对象以及这些对象之间的丰富相互依赖关系。

若干种 `ApplicationContext` 接口的实现由 Spring 实现。在独立的应用程序中，创建一个 `ClassPathXMLApplicationContext` 或者 `FileSystemXMLApplicationContext` 的实例更为普遍。当 XML 已经传统定义了配置元数据时，你可以通过提供一小部分 XML 配置指示容器将 Java 注释或代码用作元数据格式去以声明方式启用对这些其他元数据格式的支持。

在大部分应用场景中，实例化一个 Spring IoC 容器的一个或多个实例不需要显式用户代码。例如，在一个 Web 应用程序场景中，应用程序的 `web.xml` 文件中简单的八行(约)样板 Web 描述符 XML 文件通常就足够了（见 [Convenient ApplicationContext Instantiation for Web Applications](#)）。如果使用 [Spring Tools for Eclipse](#)，则只需单击几下鼠标或击键即可轻松创建此样板配置。

下图展示了一个关于 Spring 是如何工作的高级视图。你的应用程序类与配置元数据结合在了一起以便在 `ApplicationContext` 被创建和初始化之后，你有一个完成的配置好的可执行的系统或者应用。

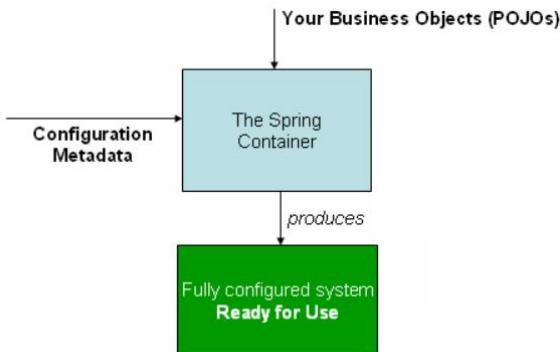


Figure 1. The Spring IoC container

1.2.1. 配置元数据

如前图所示，Spring IoC 容器接收一个配置元数据的形式。这个配置元数据表示，你，作为应用程序的开发者，告诉 Spring 容器是如何实例化，配置，装配你应用程序中的对象的。

一般来说，配置元数据以简单直观的 XML 格式提供，这是本章大部分内容用来传达 Spring IoC 容器的关键概念和功能的内容。

基于 XML 的元数据不是配置元数据的唯一允许形式。Spring IoC 容器本身与实际写入此配置元数据的格式完全脱钩。如今，许多开发人员为他们的 Spring 应用程序选择基于 Java 的配置。

有关在 Spring 容器中使用其他形式的元数据的信息，请看：

- [基于注解的配置](#): Spring 2.5 介绍了对于基于注解的元数据配置支持。
- [基于 Java 的配置](#): 从 Spring 3.0 开始许多由 Spring JavaConfig 提供的新特性变成了 Spring 框架的核心部分。因此你可以使用 Java 为你的应用程序类定义外部 bean 来替代 XML 文件。为了使用这些新特性，请看[@Configuration](#), [@Bean](#), [@Import](#), and [@DependsOn](#) 注解。

Spring 配置由至少一个，通常更多地 bean 定义组成。这些 bean 必须由 Spring 容器管理。基于 XML 的配置元数据将这些 bean 配置为顶级`<beans/>`元素内的`<bean/>`元素。Java 配置通常在[@Configuration](#) 类中使用[@Bean](#) 注解的方法。

这些 bean 的定义对应了那些实际上组成你应用程序对象。通常，你定义的服务层对象，数据访问对象，表示对象比如 Struts 的 Action 实例，基础架构对象比如 Hibernate SessionFactories, JMS Queues，以此类推。通常，不会在容器中配置细粒度的域对象，因为 DAO 和业务逻辑通常负责创建和加载域对象。然而，你可以使用集成了 AspectJ 的 Spring 去配置那些已经在 IoC 容器控制之外创建的对象。见 [Using AspectJ to dependency-inject domain object with Spring](#).

下面的例子展示了基于 XML 的元数据配置的基本结构：

```
<?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="..."> ① ②
        <!-- collaborators and configuration for this bean go here -->
    </bean>

    <bean id="..." class="...">
        <!-- collaborators and configuration for this bean go here -->
    </bean>

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

</beans>
```

① `id` 属性是标识单个 bean 定义的字符串

② `class` 属性定义 bean 的类型并使用完全限定的类名。

`id` 属性的值是指协作对象。在此示例中未显示用于引用协作对象的 XML。有关更多信息，请参见[依赖项 \(dependencies\)](#)。

1.2.2. 实例化一个容器

提供给 `ApplicationContext` 构造函数的位置路径是资源字符串，它们让容器从各种外部资源（例如本地）加载配置元数据比如本地文件系统，Java `CLASSPATH` 等。

Java

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

Kotlin

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

在你学习完关于 Spring IoC 容器之后，你或许想知道更多关于 Spring 的资源抽象（如[资源](#)中所述），它提供了一种方便的机制去读取一个输入流格式的以 URI 句法定义的位置[资源](#)。特别是[如应用程序上下文和资源路径](#)中所述，资源路径用于构造应用程序上下文。

下面的例子展示了服务层对象(`service.xml`)的配置文件：

```
<?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>
```

下面的例子展示了数据访问层 `daos.xml` 对象文件：

```

<?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>

```

在上面的例子中，服务层包括了 `PetStoreServiceImpl` 类和两个数据访问层类 `JpaAccountDao` 和 `JpaItemDao`(基于 JPA 对象关系映射标准)。属性名称(`property name`)元素引用 JavaBean 属性的名称，而 `ref` 元素引用另一个 bean 定义的名称。`id` 和 `ref` 元素之间的这种联系表示了协作对象之间的依赖性。对于配置一个对象的依赖的详细信息，见依赖(`Dependencies`)。

● 构建基于 XML 的配置元数据

使 bean 定义跨多个 XML 文件将会很有用。经常性的，在你架构中的每一个独立的 XML 配置文件表示一个逻辑层或者模块。

你可以使用应用上下文构造器去从所有的 XML 片中读取 bean 定义。如[上一节](#)中所示，此构造函数具有多个资源(`Resource`)位置。或者，使用`<import />`元素的一个或多个实例从另一个文件加载 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>

```

在上述例子中，外部 bean 定义从三个文件中被读取到了：`service.xml`, `messageSource.xml` 和 `themeSource.xml`. 所有位置路径都相对于定义文

件进行导入，因此 `services.xml` 必须与进行导入的文件位于同一目录或类路径位置，而 `messageSource.xml` 和 `themeSource.xml` 必须位于该路径下方的资源位置导入文件。如你所见，斜杠被忽略。但是，鉴于这些路径是相对的，最好不要使用任何斜线。根据 Spring Schema，导入的文件的内容（包括顶级`<beans/>`元素）必须是有效的 XML bean 定义。

可以但不建议使用相对的“`../`”路径引用父目录中的文件。这样做会创建对当前应用程序外部文件的依赖关系。特别是，不建议对 `classpath: URL`（例如，`classpath: ../ services.xml`）使用此引用，在 URL 中，运行时解析过程会选择“最近的”类路径根目录，然后查看其父目录。类路径配置的更改可能导致选择其他错误的目录。



你始终可以使用完全限定的资源位置来代替相对路径：例如，文件：`C:/config/services.xml` 或类路径：`/config/services.xml`。但是请注意，您正在将应用程序的配置耦合到特定的绝对位置。通常，最好为这样的绝对位置保留一个间接寻址-例如，通过在运行时针对 JVM 系统属性解析的“`$ {...}`”占位符。

命名空间本身提供了导入指令功能。Spring 所提供的一系列 XML 名称空间（例如，上下文和 util 名称空间）中提供了超出普通 bean 定义的其他配置功能。

● Groovy Bean 定义 DSL

作为外部化配置元数据的另一个示例，Bean 定义也可以在 Spring 的 Groovy Bean 定义 DSL 中表达，这是从 Grails 框架中得知的。通常，这种配置位于“`.groovy`”文件中，其结构如以下示例所示：

```

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

```

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

1.2.3. 使用容器

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

使用 `ApplicationContext` 可以读取 bean 定义并访问它们，如以下示例所示：

Java

```

// 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();

```

Kotlin

```
import org.springframework.beans.factory.getBean  
  
// create and configure beans  
val context = ClassPathXmlApplicationContext("services.xml", "daos.xml")  
  
// retrieve configured instance  
val service = context.getBean<PetStoreService>("petStore")  
  
// use configured instance  
var userList = service.getUsernameList()
```

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

Java

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

Kotlin

```
val context = GenericGroovyApplicationContext("services.groovy", "daos.groovy")
```

最灵活的变体是 `GenericApplicationContext` 与读取器委托结合使用，例如，与 XML 文件的 `XmlBeanDefinitionReader` 结合使用，如以下示例所示：

Java

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

Kotlin

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

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

Java

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

Kotlin

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

您可以在同一 `ApplicationContext` 上混合和匹配此类阅读器委托，从不同的配置源中读取 bean 定义。

然后可以使用 `getBean` 检索 bean 的实例。`ApplicationContext` 接口有检索 beans 的一系列方法，但是，要我说，你的应用程序不该用它。确实，您的应用程序代码应该根本不调用 `getBean()` 方法，因此完全不依赖 Spring API。例如，Spring 与 Web 框架的集成的各种 Web 框架组件（例如控制器和 JSF 托管的 Bean）提供了依赖项注入，使您可以声明一个通过元数据（例如自动装配注释）对特定 bean 的依赖。

1.3. Bean 的概述

一个 Spring IoC 容器管理着一个或多个 beans。这些 beans 通过你提供给容器的配置元数据创建（比如 XML 中`<bean/>`标签定义）。

在容器内部，这些 bean 定义被表示为 `BeanDefinition` 对象，它包含了如下元数据：

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

该元数据转换为构成每个 bean 定义的一组属性。下表描述了这些属性：

Property	Explained in...
Class	Instantiating Beans
Name	Naming Beans
Scope	Bean Scopes
Constructor arguments	Dependency Injection
Properties	Dependency Injection
Autowiring mode	Autowiring Collaborators
Lazy initialization mode	Lazy-initialized Beans
Initialization method	Initialization Callbacks
Destruction method	Destruction Callbacks

除了包含有关如何创建特定 bean 的信息的 bean 定义之外，`ApplicationContext` 实现还允许注册在容器外部（由用户）创建的现有对象。这将通过 `ApplicationContext` 的 `BeanFactory` 的 `getBeanFactory()` 方法完成，它返回 `BeanFactory` 的 `DefaultListableBeanFactory` 实现。`DefaultListableBeanFactory` 通过以下方式支持此注册：`registerSingleton(...)` 和 `registerBeanDefinition(...)` 方法。但是，典型的应用程序只能与通过常规 bean 定义元数据定义的 bean 一起使用。



Bean 元数据和手动提供的单例实例需要尽早注册，以便容器在自动装配和其他自省步骤中正确地理解/解释它们。虽然在某种程度上支持覆盖现有元数据和现有单例实例，但是在运行时（与对工厂的实时访问同时）对新 bean 的注册不被正式支持，并且可能导致并发访问异常，bean 容器中的状态不一致或全部。

1.3.1. 命名 beans

每个 bean 具有一个或多个标识符。这些标识符在承载 Bean 的容器内必须唯一。一个 bean 同行只有一个标识符，然而如果它确实有超过一个的时候，另外的一个我们可以考虑当成别名。

在基于 XML 的配置元数据上，你使用 `id` 属性，`name` 属性，或者两者皆用能够特指 bean 的标识符。`id` 属性是你能够特指确定的一个 `id`。简便点，这些命名是单词数字一类 ('myBean', 'someService' 等)但是它们也会包含特殊字符。如果你想给这些 bean 加点别名，你也可以给他们制定一个 `name` 属性，由逗号，分号，空格隔开。作为历史记录，在 Spring 3.1 之前的版本中，`id` 属性定义为 `xsd:ID` 类型，该类型限制了可能的字符。自 3.1 之后它被定义为 `xsd:String` 类型。请注意，bean ID 唯一性仍由容器强制执行，尽管

不再由 XML 解析器执行。

你不再被强制提供一个 `name` 或者 `id` 给一个 bean。如果你不明确的提供一个 `name` 或者 `id` 属性的话容器将会自动生成一个唯一名字给这个 bean 然而，如果你想通过名字 `name` 参考一个 bean 的话，比如 `ref` 元素的使用或者一个服务定位风格查找表，你必须提供一个名字 `name`。不提供名称的动机与使用内部 bean 和自动装配合作者有关。

Bean 命名约定

约定是在命名 bean 时将标准 Java 约定用于实例字段名称。也就是说，bean 名称以小写字母开头，并用驼峰式大小写。此类名称的示例包括 `accountManager`, `accountService`, `userDao`, `LoginController` 等。

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

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

● 在 Bean 的定义之外使用别名

在 bean 定义本身中，你可以通过使用 `id` 属性指定的最多一个名称和 `name` 属性中任意数量的其他名称的组合来为 bean 提供多个名称。这些名称可以是同一个 bean 的等效别名，并且在某些情况下很有用，例如，通过使用特定于该组件本身的 bean 名称，让应用程序中的每个组件都引用一个公共依赖项。但是，在实际定义 bean 的地方指定所有别名并不总是足够的。有时需要为在别处定义的 bean 引入别名。在大型系统中通常是这种情况，在大型系统中，配置在每个子系统之间分配，每个子系统都有自己的对象定义集。在基于 XML 的配置元数据中，可以使用`<alias/>`元素来完成此操作。以下示例显示了如何执行此操作：

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

在这个例子中，一个叫 `fromName` 的 bean (在同一个容器中) 在定义了别名之后也可能被参考成 `toName`。

举例说明，子系统 A 的配置元数据可能被参考为一个叫 `subsystemA-dataSource` 的 `DataSource`。子系统 B 的配置元数据可能被参考为一个叫 `subsystemB-dataSource` 的 `DataSource`。组成使用这两个子系统的主应用程序时，主应用程序通过 `myApp-dataSource` 的名称引用数据源。要使所有三个名称都引用相同的对象，可以将以下别名定义添加到配置元数据中：

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

现在每个组件和主应用程序可以使用唯一的一个 `name` 去引用数据源了，并且保证不与任何其他定义冲突（有效地创建名称空间），但它们引用的是同一 bean。

Java-配置

如果使用 `JavaConfiguration`，则 `@Bean` 注解可用于提供别名。有关详细信息，请参见 [使用 @Bean 注解](#)。

1.3.2. 实例化 Beans

`Bean` 定义实质上是创建一个或多个对象的方法。当被调用时，容器将查看命名 `bean` 的食谱（`recipe?`），并使用该 `bean` 定义封装的配置元数据来创建（或获取）实际对象。

如果使用基于 XML 的配置元数据，则在 `<bean/>` 元素的 `class` 属性中指定要实例化的对象的类型（或类）。此类属性（在内部是 `BeanDefinition` 实例的 `Class` 属性）通常 是必需的。（有关例外，请参阅 [使用实例工厂方法实例化 \(Instantiation by Using an Instance Factory Method\)](#) 和 [Bean 定义继承 \(Bean Definition Inheritance\)](#)）可以通过以下两种方式之一使用 `Class` 属性：

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

内部类命名

如果你想为一个静态内部类命名的话，你不得不使用二进制名称。

例如，一个叫 `SomeThing` 的类名在 `com.example` 的包里，这个类还有一个叫 `OtherThing` 的内部类，它的 `class` 属性就会是 `com.example.SomeThing$OtherThing`。

请注意，名称中使用 `$` 字符将嵌套的类名与外部类名分开。

● 构造器实例化

当你使用构造器方法去创建一个 bean 时，所有普通的类都可以被 Spring 使用并与之兼容。也就是说，正在开发的类不需要实现任何特定的接口或以特定的方式进行编码。只需指定 bean 类就足够了。但是，根据您用于该特定 bean 的 IoC 的类型，您可能需要一个默认（空）构造函数。

Spring IoC 容器几乎可以管理您要管理的任何类。他不只限制在管理真正的 JavaBeans。大多数 Spring 用户更喜欢实际的只有一个默认构造函数的 JavaBeans 以及根据容器中的属性建模的适当的 setter 和 getter。您还可以在容器中拥有更多奇特的非 Bean 样式类。例如，您需要使用绝对不符合 JavaBean 规范的旧式连接池，Spring 也可以对其进行管理。

通过基于 XML 的配置元数据，你可以这么来定义你的 bean 类：

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

对于有参构造机制（如果需要）和在对象被构建之后设置对象实例属性的详情，请看注入依赖([Injecting Dependencies](#))

● 静态工厂实例化

当你通过一个静态工厂方法创建了一个 bean 的时候，使用 `class` 属性能够指定一个包含静态工厂方法和一个叫做 `factory-method` 的属性的类去指定一个工厂方法本身的名字。你可以去调用它（可以传参稍后会讲）并返回一个活动的对象，随后将其视为已通过构造函数创建。这种 bean 定义的一种用法是在旧版代码中调用静态工厂。

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

下面的例子展示了一个和上面的 bean 定义一起工作的类：

Java

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

Kotlin

```
class ClientService private constructor() {  
    companion object {  
        private val clientService = ClientService()  
        fun createInstance() = clientService  
    }  
}
```

有关为工厂方法提供（可选）参数并在从工厂返回对象后设置对象实例属性的机制的详细信息，请参见详细的依赖关系和配置 [Dependencies and Configuration in Detail](#)。

● 使用实例工厂方法实例化

和通过[静态工厂方法实例化](#)类似的，使用实例工厂方法实例化会从容器中调用现有 bean 的非静态方法来创建新 bean。为了使用这个机制，设置 `class` 属性为空，在 `factory-bean` 属性，指定一个在当前容器（或父容器或祖先容器）中类的名字，它包含了一个将要创建这个对象的实例方法。设置工厂方法本身的名字在 `factory-method` 属性中。下面的这个例子展示了如何配置这么一个 bean：

```
<!-- the factory bean, which contains a method called createInstance() -->  
<bean id="serviceLocator" class="examples.DefaultServiceLocator">  
    <!-- inject any dependencies required by this locator bean -->  
</bean>  
  
<!-- the bean to be created via the factory bean -->  
<bean id="clientService"  
    factory-bean="serviceLocator"  
    factory-method="createClientServiceInstance"/>
```

下面的例子展示了对应的类：

Java

```
public class DefaultServiceLocator {  
  
    private static ClientService clientService = new ClientServiceImpl();  
  
    public ClientService createClientServiceInstance() {  
        return clientService;  
    }  
}
```

Kotlin

```
class DefaultServiceLocator {  
    companion object {  
        private val clientService = ClientServiceImpl()  
    }  
    fun createClientServiceInstance(): ClientService {  
        return clientService  
    }  
}
```

一个工厂类也能够拥有不止一个的工厂方法，就像下面这个例子：

```
<bean id="serviceLocator" class="examples.DefaultServiceLocator">  
    <!-- inject any dependencies required by this locator bean -->  
</bean>  
  
<bean id="clientService"  
    factory-bean="serviceLocator"  
    factory-method="createClientServiceInstance"/>  
  
<bean id="accountService"  
    factory-bean="serviceLocator"  
    factory-method="createAccountServiceInstance"/>
```

下面的例子展示了对应的类：

Java

```
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;  
    }  
}
```

Kotlin

```
class DefaultServiceLocator {  
    companion object {  
        private val clientService = ClientServiceImpl()  
        private val accountService = AccountServiceImpl()  
    }  
  
    fun createClientServiceInstance(): ClientService {  
        return clientService  
    }  
  
    fun createAccountServiceInstance(): AccountService {  
        return accountService  
    }  
}
```

这种方法表明，工厂 Bean 本身可以通过依赖项注入（DI）进行管理和配置。详细信息，请参见依赖性和配置 [Dependencies and Configuration in Detail](#)。



在 Spring 文档中，“factory bean”引用为一个被配置在 Spring 容器内的，能够通过一个[实例](#)或者[静态工厂方法](#)创建对象的一个 bean。相比之下，[FactoryBean](#)（请注意大小写）是指特定于 Spring 的 [FactoryBean](#) 实现类。

● 确定 Bean 的运行时类型

确定特定 bean 的运行时类型并非易事。Bean 元数据定义中的指定类只是初始类引用，可能与声明的工厂方法结合使用，或者是 [FactoryBean](#) 类，这可能导致 Bean 的运行时类型不同，或者在实例级的工厂方法情况下完全不进行设置（通过指定的 [factory-bean](#) 名称解析）。此外，AOP 代理可以使用基于接口的代理包装 Bean 实例，而目标 Bean 的实际

类型（仅是其实现的接口）的暴露程度有限。找出特定 bean 的实际运行时类型的推荐方法是对指定 bean 名称的 `BeanFactory.getType` 调用。这考虑了上述所有情况，并返回了针对相同 bean 名称的 `BeanFactory.getBean` 调用将返回的对象的类型。

1.4. 依赖

一个经典的企业级应用不会包含一个单独的对象（或者 bean 在 Spring 用语中）。即使是最简单的应用程序，也有一些对象可以协同工作，以呈现最终用户视为一致的应用程序。下面这一节将解释如何从定义多个独立的 Bean 定义到实现对象协作以实现目标的完全实现的应用程序。

1.4.1. 依赖注入

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

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

DI 存在两个主要变体：[基于构造函数的依赖注入](#)和[基于 Setter 的依赖注入](#)。

● 基于构造器的依赖注入

基于构造函数的 DI 是通过容器调用具有多个参数的构造函数来完成的，每个参数表示一个依赖项。调用带有特定参数的静态工厂方法来构造 Bean 几乎是等效的，并且本次讨论将构造函数和静态工厂方法的参数视为类似。下面的例子展示了一个仅能通过构造器注入进行依赖注入的类：

Java

```
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...  
}
```

Kotlin

```
// a constructor so that the Spring container can inject a MovieFinder  
class SimpleMovieLister(private val movieFinder: MovieFinder) {  
    // business logic that actually uses the injected MovieFinder is omitted...  
}
```

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

构造参数解析

构造函数参数解析匹配通过使用参数的类型进行。如果 Bean 定义的构造函数参数中不存在任何歧义，当 bean 正在被初始化的时候，bean 定义中构造器参数的顺序就是提供给适配的构造器的参数顺序。看看下面这个类：

Java

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

Kotlin

```
package x.y

class ThingOne(thingTwo: ThingTwo, thingThree: ThingThree)
```

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

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

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

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

当引用另一个 bean 时，类型是已知的，并且可以发生匹配（与前面的示例一样）。当使用简单类型（例如`<value> true </ value>`）时，Spring 无法确定值的类型，因此在没有帮助的情况下无法按类型进行匹配。来看下面这个类：

Java

```
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;
    }
}
```

Kotlin

```
package examples

class ExampleBean(
    private val years: Int, // Number of years to calculate the Ultimate Answer
    private val ultimateAnswer: String// The Answer to Life, the Universe, and
    Everything
)
```

构造函数参数类型匹配

在上述情况下，如果通过使用 `type` 属性显式指定构造函数参数的类型，则容器可以使用简单类型的类型匹配。来看下面这个例子：

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

构造器参数索引

您可以使用 `index` 属性来明确指定构造函数参数的索引，就像下面这个例子一样：

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

除了解决多个简单值的歧义性之外，指定索引还可以解决歧义，其中构造函数具有两个相同类型的参数。



索引从 0 开始。

构造器参数名称

你也可以使用构造器参数名来消除值得歧义，来看下面这个例子：

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

请记住，要立即使用该功能，必须在启用调试标志的情况下编译代码，以便 Spring 可以从构造函数中查找参数名称。如果您不能或不想使用 `debug` 标志编译代码，则可以使用 `@ConstructorProperties` JDK 注释显式命名构造函数参数。然后，该示例类必须如下所

示：

Java

```
package examples;

public class ExampleBean {

    // Fields omitted

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

Kotlin

```
package examples

class ExampleBean
@ConstructorProperties("years", "ultimateAnswer")
constructor(val years: Int, val ultimateAnswer: String)
```

● 基于 Setter 的构造输入

通过使用无参数构造函数或无参数静态工厂方法实例化您的 bean 之后，容器通过在 bean 上调用 `setter` 方法来完成基于 `setter` 的 DI。

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

Java

```
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...  
}
```

Kotlin

```
class SimpleMovieLister {  
  
    // a late-initialized property so that the Spring container can inject a  
    // MovieFinder  
    lateinit var movieFinder: MovieFinder  
  
    // business logic that actually uses the injected MovieFinder is omitted...  
}
```

`ApplicationContext` 支持它管理的 bean 的基于构造函数和基于 `setter` 的依赖注入。

在已经通过构造函数方法注入了某些依赖项之后，它还支持基于 `setter` 的依赖注入。您可以以 `BeanDefinition` 的形式配置依赖项，并与 `PropertyEditor` 实例结合使用以将属性从一种格式转换为另一种格式。但是，大多数 Spring 用户并不直接（即以编程方式）使用这些类，而是使用 XML bean 定义，带注解的组件（即以`@Component`, `@Controller` 等进行注解的类）或`@Bean` 方法基于 Java 配置类。然后将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。

基于构造器还是基于 `setter` 进行依赖注入？

由于可以混合使用基于构造函数的 DI 和基于 `setter` 的 DI，因此将构造函数用于强制性依赖项并将 `setter` 方法或配置方法用于可选依赖性是一个很好的经验法则。注意，可以在 `setter` 方法上使用 `@Required` 批注，以使该属性成为必需的依赖项。但是，最好使用带有参数的程序验证的构造函数注入。

Spring 团队通常提倡构造函数注入，因为它可以让您将应用程序组件实现为不可变对象，并确保所需的依赖项不为 `null`。此外，注入构造函数的组件始终以完全初始化的状态返回到客户端（调用）代码。附带说明一下，大量的构造函数参数是一种不好的代码异味，这表明该类可能承担了太多的职责，应将其重构以更好地解决关注点分离问题。

`Setter` 注入主要应仅用于可以在类中分配合理的默认值的可选依赖项。否则，必须在代码使用依赖项的任何地方执行非空检查。`setter` 注入的一个好处是，`setter` 方法使该类的对象在以后可以重新配置或重新注入。因此，通过 `JMX MBean` 进行管理是用于 `setter` 注入的引人注目的用例。

使用 DI 的风格可以使一个指定的类具有最大意义。有时，在处理您没有源代码的第三方类时，将为您做出选择。例如，如果第三方类未公开任何 `setter` 方法，则构造函数注入可能是 DI 的唯一可用形式。

● 依赖解析过程

容器执行 bean 依赖项解析，如下所示：

使用描述所有 bean 的配置元数据创建和初始化 `ApplicationContext`。可以通过 XML、Java 代码或注释来指定配置元数据。

对于每个 bean，其依赖项都以属性、构造函数参数或 `static-factory` 方法的参数（如果使用它而不是常规构造函数）的形式表示。在实际创建 Bean 时，会将这些依赖项提供给 Bean。

每个属性或构造函数参数都是要设置的值的实际定义，或者是对容器中另一个 bean 的引用。

作为值的每个属性或构造函数参数都将从其指定的格式转换为该属性或构造函数参数的实际类型。默认情况下，Spring 可以将以字符串格式提供的值转换为所有内置类型，例如 `int`, `long`, `String`, `boolean` 等。

在创建容器时，Spring 容器会验证每个 bean 的配置。但是，在实际创建 Bean 之前，不会设置 Bean 属性本身。创建容器时，将创建具有单例作用域并设置为预先实例化（默认）的 Bean。作用域在 Bean 作用域中定义。否则，仅在请求时才创建 Bean。创建和分配 Bean

的依赖关系关系及其依赖的依赖（依此类推）时，创建 Bean 可能会导致创建一个 Bean 图。请注意，这些依赖项之间的解析不匹配可能会在后期出现，即在第一次创建受影响的 bean 时。

循环依赖

如果主要使用构造函数注入，则可能会创建无法解决的循环依赖方案。

例如：类 A 通过构造函数注入需要类 B 的实例，而类 B 通过构造函数注入获取类 A 的实例。如果您配置了将类 A 和 B 相互注入的 bean，Spring IoC 容器会在运行时检测到此循环引用，并抛出 `BeanCurrentlyInCreationException`。

一种可能的解决方案是编辑某些类的源代码，这些类的源代码由设置者而不是构造函数来配置。

或者，避免构造函数注入，而仅使用 `setter` 注入。换句话说，尽管不建议这样做，但是您可以使用 `setter` 注入配置循环依赖关系。

你可以完全相信 Spring 在做正确的事情。它在容器加载时检测配置问题，例如对不存在的 Bean 的引用和循环依赖项。在实际创建 Bean 时，Spring 设置属性并尽可能晚地解决依赖关系。这意味着如果创建对象或其依赖项之一存在问题，则已正确加载的 Spring 容器稍后可以在您请求对象时生成异常-例如，由于缺少属性或无效属性，bean 引发异常。这可能会延迟某些配置问题的可见性，这就是为什么默认情况下 `ApplicationContext` 实现会预先实例化单例 bean 的原因。在实际需要这些 bean 之前先花一些时间和内存来创建它们，您会在创建 `ApplicationContext` 时发现配置问题，而不是稍后。你仍然可以覆盖此默认行为，以便单例 bean 延迟初始化，而不是预先实例化。

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

● 依赖注入的例子

以下示例将基于 XML 的配置元数据用于基于 `setter` 的依赖注入。一小部分的 Spring XML 配置文件指定了一些 bean 定义，如下所示：

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

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

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

以下示例显示了相应的 `ExampleBean` 类：

Java

```
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;
    }
}
```

Kotlin

```
class ExampleBean {
    lateinit var beanOne: AnotherBean
    lateinit var beanTwo: YetAnotherBean
    var i: Int = 0
}
```

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

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

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

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

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

以下示例显示了相应的 `ExampleBean` 类：

Java

```
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;
    }
}
```

Kotlin

```
class ExampleBean(
    private val beanOne: AnotherBean,
    private val beanTwo: YetAnotherBean,
    private val i: Int)
```

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

现在考虑该示例的一个变体，在该变体中，不是使用构造函数，而是告诉 Spring 调用静态工厂方法以返回对象的实例：

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

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

以下示例显示了相应的 `ExampleBean` 类：

Java

```
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;
    }
}
```

Kotlin

```
class ExampleBean private constructor() {
    companion object {
        // 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.
        fun createInstance(anotherBean: AnotherBean, yetAnotherBean: YetAnotherBean,
i: Int): ExampleBean {
            val eb = ExampleBean (...)
            // some other operations...
            return eb
        }
    }
}
```

静态工厂方法的参数由`<constructor-arg/>`元素提供，与实际使用构造函数的情况完全相同。`factory` 方法返回的类的类型不必与包含静态工厂方法的类的类型相同（尽管在

此示例中他们是一样的）。实例（非静态）工厂方法可以以基本上相同的方式使用（除了使用 `factory-bean` 属性代替 `class` 属性之外），因此在此不讨论这些细节。

1.4.2. 依赖和配置详情

如上一节所述，您可以将 `bean` 属性和构造函数参数定义为对其他托管 `bean`（协作者）的引用或内联定义的值。Spring 的基于 XML 的配置元数据为此目的在其 `<property/>` 和 `<constructor-arg/>` 元素中支持子元素类型。

● 直接值（原语、字符串等）

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

```
<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="misterkaoli"/>
</bean>
```

以下示例将 `p`-命名空间用于更简洁的 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="misterkaoli"/>

</beans>
```

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

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

```
<bean id="mappings"
      class="org.springframework.context.support.PropertySourcesPlaceholderConfigurer">

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

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

idref 元素

`idref` 元素只是一种防错方法，可以将容器中另一个 bean 的 `id`（字符串值-不是引用）传递给`<constructor-arg/>`或`<property/>`元素。下面的例子展示了如何使用：

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

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

前面的 bean 定义代码段（在运行时）与以下代码段完全等效：

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

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

第一种形式优于第二种形式，因为使用 `idref` 标记可使容器在部署时验证所引用的命名 Bean 实际上是否存在。在第二个变体中，不对传递给客户端 bean 的 `targetName` 属性的值执行验证。拼写错误仅在实际实例化客户端 bean 时发现（最有可能导致致命的结果）。如果客户端 Bean 是原型 Bean，则可能仅在部署容器后很长时间才发现此错字和所产生的

异常。

在 4.0 Bean XSD 中不再支持 `idref` 元素上的 `local` 属性，因为它不再提供常规 Bean 引用上的值。

升级到 4.0 模式时，将现有的 `idref local` 引用更改为 `idref bean`。

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

● 参考其他 Beans(协作者)

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

通过 `<ref/>` 标记的 `bean` 属性指定目标 bean 是最通用的形式，并且允许创建对同一容器或父容器中任何 bean 的引用，而不管它是否在同一 XML 文件中。`bean` 属性的值可以与目标 bean 的 `id` 属性相同，也可以与目标 bean 的 `name` 属性中的值之一相同。下列例子展示了如何使用 `ref` 元素：

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

通过 `parent` 属性指定目标 Bean 将创建对当前容器的父容器中的 Bean 的引用。`parent` 属性的值可以与目标 bean 的 `id` 属性或目标 bean 的 `name` 属性中的值之一相同。目标 Bean 必须位于当前容器的父容器中。 主要在具有容器层次结构并且要使用与父 bean 名称相同的代理将现有 bean 封装在父容器中时，才应使用此 bean 参考变量。以下清单对显示了如何使用 `parent` 属性：

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

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

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

● 内部 bean

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

```
<bean id="outer" class="...">
    <!-- instead of using a reference to a target bean, simply define the target bean
    inline -->
        <property name="target">
            <bean class="com.example.Person"> <!-- this is the inner bean -->
                <property name="name" value="Fiona Apple"/>
                <property name="age" value="25"/>
            </bean>
        </property>
    </bean>
```

内部 bean 定义不需要定义的 ID 或名称。如果指定，则容器不使用某个值作为标识符。容器在创建时也将忽略作用域标志，因为内部 Bean 始终是匿名的，并且始终与外部 Bean 一起创建。不可能独立地访问内部 bean 或将其注入到协作 bean 中而不是封装到封闭 bean 中。

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

● 集合

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

```

<bean id="moreComplexObject" class="example.ComplexObject">
    <!-- results in a setAdminEmails(java.util.Properties) call -->
    <property name="adminEmails">
        <props>
            <prop key="administrator">administrator@example.org</prop>
            <prop key="support">support@example.org</prop>
            <prop key="development">development@example.org</prop>
        </props>
    </property>
    <!-- results in a setSomeList(java.util.List) call -->
    <property name="someList">
        <list>
            <value>a list element followed by a reference</value>
            <ref bean="myDataSource" />
        </list>
    </property>
    <!-- results in a setSomeMap(java.util.Map) call -->
    <property name="someMap">
        <map>
            <entry key="an entry" value="just some string"/>
            <entry key ="a ref" value-ref="myDataSource"/>
        </map>
    </property>
    <!-- results in a setSomeSet(java.util.Set) call -->
    <property name="someSet">
        <set>
            <value>just some string</value>
            <ref bean="myDataSource" />
        </set>
    </property>
</bean>

```

映射键或值的值或设置值也可以是以下任意元素：

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

集合合并：

Spring 容器还支持合并集合。 应用程序开发人员可以定义父`<list/>`, `<map/>`, `<set/>`或`<props/>`元素，并具有子`<list/>`, `<map/>`, `<set/>`或`<props/>`元素。 从父集合继承并覆盖值。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，子集合的元素会覆盖父集合中指定的值。

关于合并的本节讨论了父子 bean 机制。不熟悉父级和子级 bean 定义的读者可能希望先阅读[相关部分\(1.7 Bean Definition Inheritance\)](#)，然后再继续。

下面的示例演示了集合合并：

```

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

```

注意子 bean 定义的 `adminEmails` 属性的 `<props/>` 元素上使用 `merge = true` 属性。

当 `child` bean 由容器解析并实例化后，生成的实例具有 `adminEmails Properties` 集合，其中包含将 `child` 的 `adminEmails` 集合与父对象的 `adminEmails` 集合合并的结果。下列列表展示了结果内容：

```

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

```

子属性集的值集继承了父`<props />`的所有属性元素，而支持值的子值覆盖了父集合中的值。

此合并行为类似地适用于`<list/>`, `<map/>`和`<set/>`集合类型。在`<list/>`元素的特定情况下，将维护与 `List` 集合类型关联的语义（即，值的 `有序集合` 的概念）。父级的值先于子级列表的所有值。对于“`Map`”，“`Set`”和“`Properties`”集合类型，不存在任何排序。因此，对于容器内部使用的关联 `Map`, `Set` 和 `Properties` 实现类型基础的集合类型，没有任何排序语义有效。

集合合并的局限性

您不能合并不同的集合类型（例如 `Map` 和 `List`）。如果您尝试这样做，则会引发对应的 `异常`。必须在下面的继承的子定义中指定 `merge` 属性。在父集合定义上指定 `merge` 属性

是多余的，不会导致所需的合并。

强类型集合

随着 Java 5 中泛型类型的引入，您可以使用强类型集合。也就是说，可以声明一个 `Collection` 类型，使其只能包含（例如）`String` 元素。如果使用 Spring 将强类型的集合依赖注入到 Bean 中，则可以利用 Spring 的类型转换支持，在你的强类型集合实例中的元素被添加进集合之前，预先转换为合适的类型。

以下 Java 类和 bean 定义显示了如何执行此操作：

Java

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

Kotlin

```
class SomeClass {  
    lateinit var accounts: Map<String, Float>  
}
```

```
<beans>  
    <bean id="something" class="x.y.SomeClass">  
        <property name="accounts">  
            <map>  
                <entry key="one" value="9.99"/>  
                <entry key="two" value="2.75"/>  
                <entry key="six" value="3.99"/>  
            </map>  
        </property>  
    </bean>  
</beans>
```

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

● 指向空或值为空的字符串

Spring 将属性等的空参数视为空 `String`。以下基于 XML 的配置元数据片段将 `email`

属性设置为空的 String 值（“”）。

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

前面的示例等效于以下 Java 代码：

Java

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

Kotlin

```
exampleBean.email = ""
```

`<null />`元素处理空值。以下清单显示了一个示例：

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

前面的示例等效于以下 Java 代码：

Java

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

Kotlin

```
exampleBean.email = null
```

● P 命名空间的 XML 快捷方式

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

Spring 支持带有名称空间（9.1XML Schema）的可扩展配置格式，这些名称空间基于 XML Schema 定义。本章讨论的 `bean` 配置格式在 XML Schema 文档中定义。但是，`p` 命名空间未在 XSD 文件中定义，仅存在于 Spring 的核心(core)中。

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

```

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

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

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

```

该示例显示了 p 命名空间中的一个属性，该属性在 bean 定义中称为 email。这告诉 Spring 包含一个属性声明。如前所述，p 名称空间没有架构定义，因此可以将属性名称设置为属性名称。

下一个示例包括另外两个 bean 定义，它们都引用了另一个 bean：

```

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

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

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

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

```

此示例不仅包括使用 p-namespace 的属性值，还使用特殊格式声明属性引用。第一个 bean 定义使用<property name="spouse" ref =“ jane”/>创建从 bean john 到 bean jane 的引用，而第二个 bean 定义使用 p:spouse-ref =“ jane”作为属性来执行完全一样的东西。在这种情况下，spouse 是属性名称，而-ref 部分指示这不是一个直接值，而是对另一个 bean 的引用。



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以 Ref 结尾的属性发生冲突，而标准 XML 格式则没有。我们建议您仔细选择方法，并与团队成员进行交流，以避免同时使用这三种方法生成 XML 文档。

● C 命名空间的 XML 快捷方式

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

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

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

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

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

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

</beans>
```

c:命名空间使用与 p: 相同的约定（bean 引用为尾随-ref）以按名称设置构造函数参数。同样，即使未在 XSD 架构中定义它（也存在于 Spring 内核中），也需要在 XML 文件中声明它。

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



由于 XML 语法的原因，索引符号要求前导 `_` 的存在，因为 XML 属性名称不能以数字开头（即使某些 IDE 允许）。相应的索引符号也可用于 `<constructor-arg>` 元素，但并不常用，因为声明的普通顺序通常就足够了。

● 复合属性名

设置 `bean` 属性时，可以使用复合属性名称或嵌套属性名称，只要路径中除最终属性名称以外的所有组件都不为空即可。来看下面这个 `bean` 的定义：

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

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

1.4.3. 使用 `depends-on`

如果一个 `bean` 是另一个 `bean` 的依赖项，则通常意味着将一个 `bean` 设置为另一个 `bean` 的属性。通常，您可以使用基于 XML 的配置元数据中的 `<ref/>` 元素 (References to Other Beans) 来完成此操作。但是，有时 `bean` 之间的依赖性不是直接的。一个例子是当在一个类的静态初始化块需要被触发，诸如用于数据库驱动的注册。`depends-on` 属性可以使使用这个元素的 `bean` 在初始化之前，强制一个或多个 `bean` 先被初始化。下面的示例使用 `depends-on` 属性来表示对单个 `bean` 的依赖关系：

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

要表达对多个 `bean` 的依赖关系，请提供一个 `bean` 名称列表作为 `Depends-on` 属性的值（逗号，空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" />
```



depends-on 属性既可以指定初始化时间依赖，也可以仅在单例 bean 的情况下指定相应的销毁时间依赖。与给定 bean 定义依赖关系的从属 bean 首先被销毁，然后再销毁给定 bean 本身。因此，依赖也可以控制关闭顺序。

1.4.4. 懒初始化 Beans

默认情况下，作为初始化过程的一部分，`ApplicationContext` 实现会急于创建和配置所有 `singleton(1.5.1)` bean。总的来说，这种预初始化是可取的，因为与数小时甚至数天后相比，会立即发现配置或周围环境中的错误。如果不希望使用此行为，则可以通过将 bean 定义标记为延迟初始化来防止单例 bean 的预实例化。延迟初始化的 bean 告诉 IoC 容器在首次请求时而不是在启动时创建一个 bean 实例。

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

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

当前面的配置写在 `ApplicationContext` 文件时，在 `ApplicationContext` 启动时不会急切地预实例化 `lazy` bean，而在 `not.lazy` Bean 中则急切地预实例化。

但是，当延迟初始化的 bean 是未延迟初始化的单例 bean 的依赖项时，`ApplicationContext` 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。延迟初始化的 bean 被注入到其他未延迟初始化的单例 bean 中。

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

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

1.4.5. 自动装配协作者

Spring 容器可以自动装配协作 beans 之间的关系。您可以通过检查 ApplicationContext 的内容，让 Spring 为您的 bean 自动解决协作者（其他 bean）。

自动装配有如下缺点：

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

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

Table 2. Autowiring modes

Mode	Explanation
<code>no</code>	(Default) No autowiring. Bean references must be defined by <code>ref</code> 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.
<code>byName</code>	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 <code>master</code> property (that is, it has a <code>setMaster(..)</code> method), Spring looks for a bean definition named <code>master</code> and uses it to set the property.
<code>byType</code>	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 <code>byType</code> autowiring for that bean. If there are no matching beans, nothing happens (the property is not set).
<code>constructor</code>	Analogous to <code>byType</code> but applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.

使用 `byType` 或构造函数自动装配模式，您可以连接数组和类型集合。在这种情况下，将提供容器中与期望类型匹配的所有自动装配候选，以满足相关性。如果期望的键类型为 `String`，则可以自动装配强类型 Map 实例。自动装配的 Map 实例的值包括与期望类型匹配的所有 bean 实例，并且 Map 实例的键包含相应的 bean 名称。

● 自动装配的限制和缺点

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

我们考虑一下自动装配的限制和缺点：

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

在以后的场景中，您有几种选择：

- 放弃自动装配，转而使用显示装配。
- 通过将其 bean 的 `autowire-candidate` 属性设置为 `false`，避免自动装配 bean 定义，[如下一节](#)所述。
- 通过将其`<bean/>`元素的 `primary` 属性设置为 `true`，将单个 bean 定义指定为主要候选对象。
- 如[基于注释的容器配置](#)中所述，通过基于注释的配置实现更细粒度的控件。
- **从自动装配中排除一个 Bean**

在每个 bean 的基础上，您可以从自动装配中排除一个 bean。以 Spring XML 格式，将`<bean/>`元素的 `autowire-candidate` 属性设置为 `false`。容器使特定的 bean 定义不适用于自动装配基础结构（包括注释样式配置，例如`@Autowired`）。



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

您还可以基于与 Bean 名称的模式匹配来限制自动装配候选。顶级`<beans/>`元素在其 `default-autowire-candidates` 属性内接受一个或多个模式串。例如，要将自动装配候选状态限制为名称以 `Repository` 结尾的任何 bean，请提供`*Repository` 值。要提供多种

模式，请在以逗号分隔的列表中定义它们。 Bean 定义的 autowire-candidate 属性的显式值 true 或 false 始终优先。 对于此类 bean，模式匹配规则不适用。

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

1.4.6. 方法注入

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

一个解决方案是放弃某些控制反转。您可以通过实现 ApplicationContextAware 接口，并通过对容器进行 `getBean("B")` 调用来使 bean A 知道该容器，以便每次 bean A 需要它时都请求一个（通常是新的）bean B 实例。以下示例显示了此方法：

Java

```
// 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;
    }
}
```

Kotlin

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

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

class CommandManager : ApplicationContextAware {

    private lateinit var applicationContext: ApplicationContext

    fun process(commandState: Map<*, *>): Any {
        // grab a new instance of the appropriate Command
        val command = createCommand()
        // set the state on the (hopefully brand new) Command instance
        command.state = commandState
        return command.execute()
    }

    // notice the Spring API dependency!
    protected fun createCommand() =
        applicationContext.getBean("command", Command::class.java)

    override fun setApplicationContext(applicationContext: ApplicationContext) {
        this.applicationContext = applicationContext
    }
}
```

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

You can read more about the motivation for Method Injection in [this blog entry](#).

<https://spring.io>

● 查找方法注入

查找方法注入是容器重写被容器管理的 Bean 上的方法并返回容器中另一个已命名 Bean 的查找结果的能力。查找通常涉及原型 bean，如上一节中所述。Spring 框架通过使用从 CGLIB 库生成字节码来动态生成覆盖该方法的子类来实现此方法注入。

- 为了使此动态子类起作用，Spring Bean 容器子类的类也不能是 `final`，而要覆盖的方法也不能是 `final`。
- 对具有抽象方法的类进行单元测试需要您自己对该类进行子类化，并提供该抽象方法的存根实现。
- 组件扫描也需要具体方法，这需要具体的类别。
- 另一个关键限制是，查找方法不适用于工厂方法，尤其不适用于配置类中的`@Bean` 方法，因为在这种情况下，容器不负责创建实例，因此无法创建运行时生成的动态子类。

对于前面的代码片段中的 `CommandManager` 类，Spring 容器动态地覆盖 `createCommand()` 方法的实现。如重新编写的示例所示，`CommandManager` 类没有任何 Spring 依赖项：

Java

```
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();
}
```

Kotlin

```
package fiona.apple

// no more Spring imports!

abstract class CommandManager {

    fun process(commandState: Any): Any {
        // grab a new instance of the appropriate Command interface
        val command = createCommand()
        // set the state on the (hopefully brand new) Command instance
        command.state = commandState
        return command.execute()
    }

    // okay... but where is the implementation of this method?
    protected abstract fun createCommand(): Command
}
```

在包含要注入的方法的客户端类（在本例中为 `CommandManager`）中，要注入的方法需要以下形式的声明：

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

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

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

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

每当需要新的 `myCommand` bean 实例时，标识为 `commandManager` 的 bean 就会调用其自己的 `createCommand()` 方法。如果确实需要将 `myCommand` bean 部署为原型，则必须小心。如果是单例，则每次都返回 `myCommand` bean 的相同实例。

另外，在基于注释的组件模型中，您可以通过`@Lookup` 注释声明一个查找方法，如下示例所示：

Java

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

Kotlin

```
abstract class CommandManager {  
  
    fun process(commandState: Any): Any {  
        val command = createCommand()  
        command.state = commandState  
        return command.execute()  
    }  
  
    @Lookup("myCommand")  
    protected abstract fun createCommand(): Command  
}
```

或者，更常用的是，你可以依赖于目标 bean 根据查找方法的声明的返回类型来解析：

Java

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

Kotlin

```
abstract class CommandManager {  
  
    fun process(commandState: Any): Any {  
        val command = createCommand()  
        command.state = commandState  
        return command.execute()  
    }  
  
    @Lookup  
    protected abstract fun createCommand(): Command  
}
```

请注意，通常应使用具体的存根实现声明此类带注解的查找方法，以使其与 Spring 的组件扫描规则（默认情况下抽象类会被忽略）兼容。此限制不适用于显式注册或显式导入的 Bean 类。



访问作用域不同的目标 bean 的另一种方法是 `ObjectFactory / Provider` 注入点。请参阅作用域 Bean 作为依赖项 [Scoped Beans as Dependencies\(1.5.4\)](#)。

您还可以在以下位置找到 `ServiceLocatorFactoryBean` (`org.springframework.beans.factory.config` 包) 很有用。

● 强制方法替换

与查找方法注入相比，方法注入的一种不太常用的形式是能够用另一种方法实现替换被管理 bean 中的任意方法。您可以放心地跳过本节的其余部分，直到您真正需要此功能为止。

借助基于 XML 的配置元数据，您可以使用 `replaced-method` 元素将现有的方法实现替换为已部署的 Bean。考虑以下类，该类具有一个我们要覆盖的名为 `computeValue` 的方法：

Java

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

Kotlin

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

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

Java

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

Kotlin

```
/**  
 * meant to be used to override the existing computeValue(String)  
 * implementation in MyValueCalculator  
 */  
class ReplacementComputeValue : MethodReplacer {  
  
    override fun reimplement(obj: Any, method: Method, args: Array<out Any>): Any {  
        // get the input value, work with it, and return a computed result  
        val input = args[0] as String;  
        ...  
        return ...;  
    }  
}
```

用于部署原始类并指定方法覆盖的 Bean 定义类似于以下示例：

```

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

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

```

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

```

java.lang.String
String
Str

```

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

1.5. Bean 的作用域

创建 bean 定义时，将创建一个配方来创建该 bean 定义所定义的类的实际实例。一个 bean 定义是一个配方的想法很重要，因为它意味着与类一样，您可以从一个配方中创建许多对象实例。

您不仅可以控制要插入到从特定 bean 定义创建的对象中的各种依赖项和配置值，还可以控制从特定 bean 定义创建的对象的作用域。这种方法功能强大且灵活，因为您可以选择通过配置创建的对象的范围，而不必在 Java 类级别上烘烤对象的范围。可以将 Bean 定义为部署在多个作用域之一中。Spring 框架支持六种作用域，其中只有在使用 Web 感知的 `ApplicationContext` 时才可用。您还可以创建[自定义范围（1.5.5）](#)。

下表表述了所支持的作用域：

Table 3. Bean scopes

Scope	Description
<code>singleton</code>	(Default) Scopes a single bean definition to a single object instance for each Spring IoC container.
<code>prototype</code>	Scopes a single bean definition to any number of object instances.

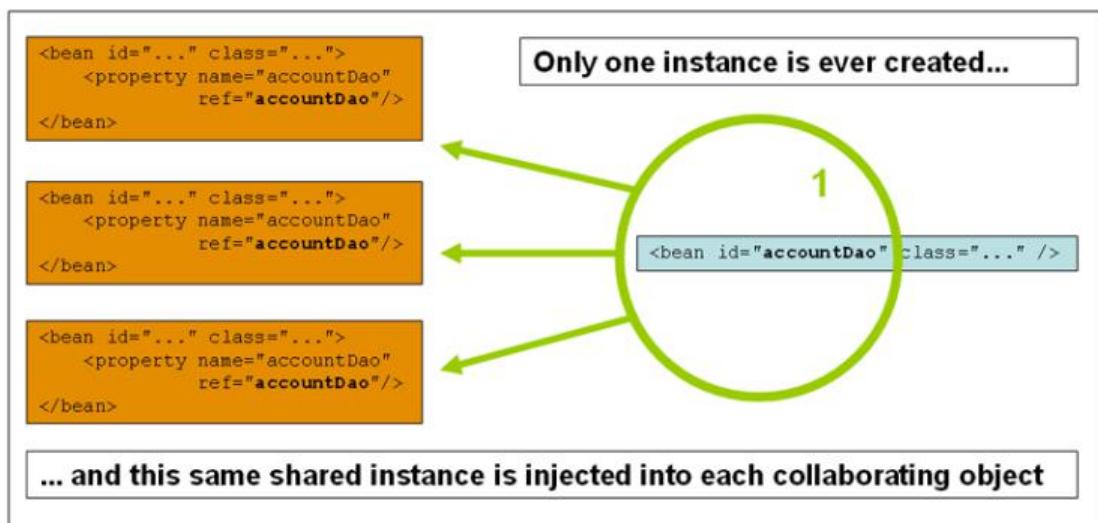
Scope	Description
request	Scopes a single bean definition to the lifecycle of a single HTTP request. That is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext .
session	Scopes a single bean definition to the lifecycle of an HTTP Session . Only valid in the context of a web-aware Spring ApplicationContext .
application	Scopes a single bean definition to the lifecycle of a ServletContext . Only valid in the context of a web-aware Spring ApplicationContext .
websocket	Scopes a single bean definition to the lifecycle of a WebSocket . Only valid in the context of a web-aware Spring ApplicationContext .

从 Spring 3.0 开始，线程作用域可用，但默认情况下未注册。有关更多信息，请参见 [SimpleThreadScope](#) 文档。有关如何注册此或任何其他自定义范围的说明，请参阅[使用自定义范围\(1.5.5\)](#)。

1.5.1. 单例作用域

仅管理一个 singleton bean 的一个共享实例，并且所有对具有 ID 或与该 bean 定义相匹配的 ID 的 bean 的请求都会导致该特定的 bean 实例由 Spring 容器返回。

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



Spring 的“单例 bean”概念与四人帮 (Gang of Four) 模式手册中定义的单例模式不

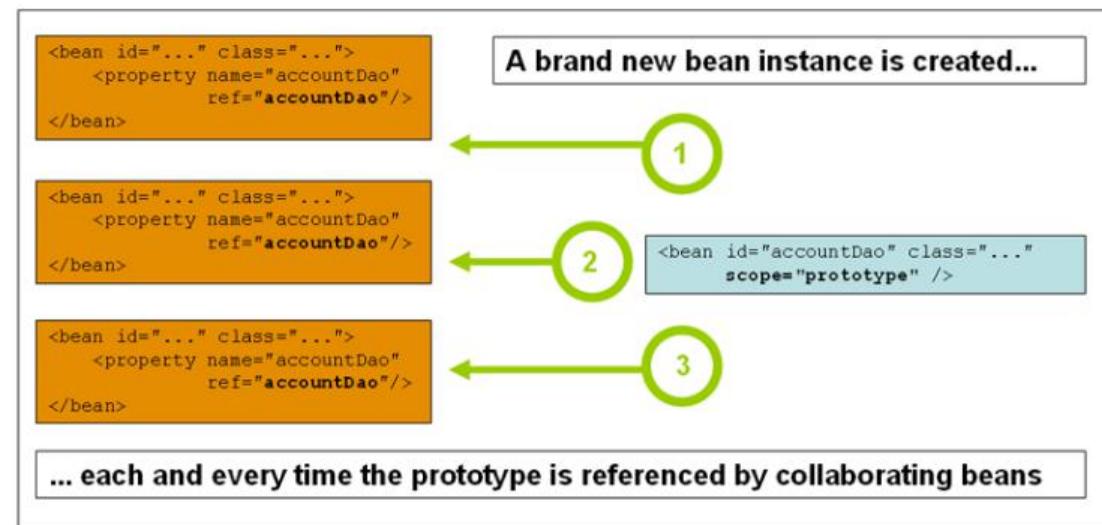
同。GoF 单例对对象的范围进行硬编码，以使每个类加载器 ClassLoader 只能创建一个特定类的一个实例。Spring 单例的作用域最好描述为每个容器和每个 bean。（这一句翻译没搞明白什么意思）。这意味着如果你在单个 Spring 容器中为特定类定义一个 bean，则 Spring 容器将根据该 bean 的定义创建有且仅有一个实例。Singleton 作用域是 Spring 中的默认作用域。要将 bean 定义为 XML 中的单例，可以定义 bean，如以下示例所示：

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

1.5.2. 原型作用域

每次对特定 bean 提出请求时，bean 部署的非单例原型作用域内都会导致创建一个新 bean 实例。也就是说，该 Bean 被注入到另一个 Bean 中，或者您可以通过容器上的 `getBean()` 方法调用来请求它。通常，应将原型作用域用于所有有状态 Bean，将单例作用域用于无状态 Bean。

下图说明了 Spring 原型作用域：



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

以下示例将 bean 定义为 XML 原型：

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

与其他作用域相反，Spring 不管理原型 Bean 的完整生命周期。容器将实例化，配置或组装原型对象，然后将其交给客户端，而无需对该原型实例的进一步记录。因此，尽管在不考虑作用域的情况下在所有对象上都调用了初始化生命周期回调方法，但在原型的情况下，不会调用已配置的销毁生命周期回调。客户端代码必须清除原型作用域内的对象，并释放原型 Bean 拥有的昂贵资源。要使 Spring 容器释放原型作用域 bean 所拥有的资源，请尝试使用自定义 bean 后处理器（1.8.1），该处理器包含对需要清除的 bean 的引用。

在某些方面，Spring 容器在原型作用域内的 bean 方面的角色是 Java new 运算符的替代。超过该时间点的所有生命周期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[生命周期回调（1.6.1）](#)。）

1.5.3. 有原型 bean 依赖的单例 Bean

当您使用对原型 bean 有依赖性的单例作用域 Bean 时，请注意，依赖关系在实例化时已解析。因此，如果你把一个原型作用域 bean 注入到一个单例作用域 bean 中时，一个新的原型作用域实例将被创建并在稍后注入到单例 bean 中，原型实例是曾经提供给单例范围的 bean 的唯一实例。

但是，假设您希望单例作用域的 bean 在运行时重复获取原型作用域的 bean 的新实例。你不能将原型作用域的 bean 依赖项注入到您的单例 bean 中，因为当 Spring 容器实例化单例 bean 并解析并注入其依赖项时，该注入仅发生一次。如果在运行时不止一次需要原型 bean 的新实例，请参见[方法注入（1.4.6）](#)。

1.5.4. 请求、会话、应用程序和 WebSocket 作用域

仅当您使用可感知 Web 的 Spring ApplicationContext 实现（例如 XmlWebApplicationContext）时，request，session，application 和 websocket 范围才可用。如果将这些作用域与常规的 Spring IoC 容器（例如 ClassPathXmlApplicationContext）一起使用，则会抛出一个 IllegalStateException 异常，该错误抱怨未知的 bean 作用域。

● 初始化 Web 配置

为了在 request，session，application 和 websocket 级别（网络范围的 Bean）上支持 Bean 的作用域，在定义 Bean 之前，需要一些较小的初始配置。（对于标准范围：单例和原型，不需要此初始设置。）

如何完成此初始设置取决于您的特定 Servlet 环境。

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

如果您使用 Servlet 2.5 Web 容器，并且在 Spring 的 `DispatcherServlet` 外部处理请求（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 `WebApplicationInitializer` 接口以编程方式完成此操作。或者，或者对于较旧的容器，将以下声明添加到 Web 应用程序的 `web.xml` 文件中：

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

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

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

`DispatcherServlet`, `RequestContextListener` 和 `RequestContextFilter` 都做完全相同的事情，即将 HTTP 请求对象绑定到为该请求提供服务的线程。这使得在请求链和会话范围内的 Bean 可以在调用链的更下游使用。

● 请求作用域

考虑以下 XML 配置来定义 bean：

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

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

当使用注解驱动的组件或 Java 配置时，`@RequestScope` 注解可用于将组件分配给 **请求作用域**。以下示例显示了如何执行此操作：

Java

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

Kotlin

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

● 会话作用域

考虑以下 XML 配置来定义 bean：

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

Spring 容器通过在单个 HTTP **会话** 的生存期内使用 `userPreferences` bean 定义来创建 `UserPreferences` bean 的新实例。换句话说，`userPreferences` bean 作用域在 HTTP **会话** 级别。与请求作用域的 Bean 一样，您可以根据需要任意更改所创建实例的内部状态，知道其他也在使用从同一 `userPreferences` Bean 定义创建的实例的 HTTP `Session` 实例不会看到这些状态更改，因为它们特定于单个 HTTP **会话**。当 HTTP 会话最终被丢弃时，作用于该特定 HTTP 会话的 bean 也将被丢弃。

使用注解驱动的组件或 Java 配置时，可以使用 `@SessionScope` 注解将组件分配给 **会话作用域**。

Java

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

Kotlin

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

● 应用程序作用域

考虑以下 XML 配置来定义 bean:

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

Spring 容器通过对整个 Web 应用程序使用一次 `appPreferences` bean 定义来创建 `AppPreferences` bean 的新实例。也就是说，`appPreferences` bean 的作用域位于 `ServletContext` 级别，并存储为常规 `ServletContext` 属性。这有点类似于 Spring 单例 bean，但是有两个重要的区别：它是每个 `ServletContext` 的单例，而不是每个 Spring 'ApplicationContext' 的单例（在任何给定的 Web 应用程序中可能都有多个），并且它实际上是公开的，因此 可见为 `ServletContext` 属性。

使用注解驱动的组件或 Java 配置时，可以使用 `@ApplicationScope` 注解将组件分配给 **应用程序作用域**。以下示例显示了如何执行此操作：

Java

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

Kotlin

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

● 作用域 Bean 作为依赖

Spring IoC 容器不仅管理对象（bean）的实例化，而且还管理协作者（或依赖项）的连接。如果要将 HTTP 会话作用域的 bean 注入（例如）到作用域更长的另一个 bean 中，则可以选择注入 AOP 代理来代替作用域的 bean。也就是说，你需要注入暴露出一样的公共接口的作用域内的对象的代理对象，但也可以检索相关作用域的真正目标对象（如 HTTP 请求），并委托方法调用到真正的对象。

您还可以在作用域为单例的 bean 之间使用 `<aop: scoped-proxy/>`，然后引用通过可序列化的中间代理进行，因此能够在反序列化时重新获得目标单例 bean。

当针对作用域原型的 bean 声明 `<aop: scoped-proxy/>` 时，共享代理上的每个方法调用都会导致创建新的目标实例，然后将该调用转发到该目标实例。

同样，作用域代理不是以生命周期安全的方式从较小的作用域访问 bean 的唯一方法。您也可以将注入点（即构造函数或 `setter` 参数或自动装配的字段）声明为 `ObjectFactory <MyTargetBean>`，从而允许 `getObject()` 调用在需要时按需检索当前实例。实例或将其单独存储。

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

JSR-330 的这种变体称为 `Provider`，并与 `Provider <MyTargetBean>` 声明和每次检索尝试的相应 `get()` 调用一起使用。有关 JSR-330 更多详细信息，请参见 [此处\(1.11\)](#)。

以下示例中的配置仅一行，但是了解其背后的“原因”和“方式”很重要：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           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/> ①
    </bean>

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

① 定义代理的行。

要创建这样的代理，请将[`<aop: scoped-proxy />`](#)子元素插入到作用域 bean 定义中（请参阅[选择要创建的代理类型](#)和[基于 XML Schema 的配置 9.1](#)）。为什么在 `request`, `session` 和 `custom` 作用域级别定义的 bean 定义都需要[`<aop: scopedproxy />`](#)元素？考虑以下单例 bean 定义，并将其与需要为上述范围定义的内容进行对比（请注意，以下 `userPreferences` bean 定义不完整）：

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

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

在前面的示例中，单例 bean (`userManager`) 注入了对 HTTP 会话作用域 bean (`userPreferences`) 的引用。这里的重点是 `userManager` bean 是单例的：每个容器仅实例化一次，并且它的依赖项（在这种情况下，只有一个，`userPreferences` bean）也只注入一次。这意味着 `userManager` bean 仅在完全相同的 `userPreferences` 对象（即最初与之注入对象）上操作。

将生命周期短的作用域 bean 注入生命周期较长的作用域 bean 时，这不是想要的行为（例如，将 HTTP 会话范围的协作 bean 作为依赖项注入到 singleton bean 中）。相反，您只需要一个 `userManager` 对象，并且在 HTTP 会话的生存期内，您需要一个特定于 HTTP 会话的 `userPreferences` 对象。因此，容器创建了一个对象，该对象公开与 `UserPreferences` 类完全相同的公共接口（理想情况下是一个 `UserPreferences` 实例的对象），该对象可以从作用域机制（`HTTP request`, `Session` 等）中获取实际的 `UserPreferences` 对象。容器将此代理对象注入到 `userManager` bean 中，而后者不知道此 `UserPreferences` 引用是代理。在此示例中，当 `UserManager` 实例在注入依赖项的 `UserPreferences` 对象上调用方法时，实际上是在代理上调用方法。然后，代理从 HTTP 会话（在本例中）获取真实的 `UserPreferences` 对象，并将方法调用委托给检索到的真实的 `UserPreferences` 对象。

因此，在将请求范围和会话范围的 bean 注入到协作对象中时，您需要以下（正确和完整）配置，如以下示例所示：

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

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

选择要创建的代理类型

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



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

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

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

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

1.5.5. 自定义作用域

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

● 创建一个自定义作用域

要将自定义范围集成到 Spring 容器中，需要实现 `org.springframework.beans.factory.config.Scope` 接口，本节对此进行了介绍。有关如何实现自己的作用域的想法，请参阅 Spring 框架本身提供的 `Scope` 实现和 `Scope javadoc`，其中详细说明了需要实现的方法。

作用域接口有四种方法可以从作用域中获取对象，将它们从作用域中删除，然后销毁它

们。

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

Java

```
Object get(String name, ObjectFactory<?> objectFactory)
```

Kotlin

```
fun get(name: String, objectFactory: ObjectFactory<*>): Any
```

会话作用域的实现，例如，从基础会话中删除了会话作用域的 bean。应该返回该对象，但是如果找不到具有指定名称的对象，则可以返回 null。以下方法从基础作用域中删除该对象：

Java

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

Kotlin

```
fun remove(name: String): Any
```

以下方法注册一个回调，当作用域被销毁或作用域中的指定对象被销毁时，作用域应调用该回调：

Java

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

Kotlin

```
fun registerDestructionCallback(name: String, destructionCallback: Runnable)
```

有关销毁回调的更多信息，请参见 [javadoc](#) 或 Spring 作用域实现。以下方法获取基础范围的会话标识符：

Java

```
String getConversationId()
```

Kotlin

```
fun getConversationId(): String
```

每个作用域的标识符都不相同。对于会话作用域的实现，此标识符可以是会话标识符。

● 使用一个自定义作用域

在编写和测试一个或多个自定义作用域实现之后，需要使 Spring 容器检测到您的新作用域。以下方法是在 Spring 容器中注册新范围的主要方法：

Java

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

Kotlin

```
fun registerScope(scopeName: String, scope: Scope)
```

该方法在 `ConfigurableBeanFactory` 接口上声明，该接口可通过 Spring 附带的大多数 `ApplicationContext` 的具体实现上的 `BeanFactory` 属性获得。

`registerScope(..)` 方法的第一个参数是与作用域相关联的唯一名称。Spring 容器本身中的此类名称的示例包括 `singleton` 和 `prototype`。`registerScope(..)` 方法的第二个参数是你希望注册和使用的自定义作用域实现的实际实例。

假设您编写了自定义的 `Scope` 实现，然后注册它，如下面的示例所示。



下一个示例使用 `SimpleThreadScope`，它包含在 Spring 中，但默认情况下未注册。对于您自己的自定义作用域实现，说明将是相同的。

Java

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

Kotlin

```
val threadScope = SimpleThreadScope()
beanFactory.registerScope("thread", threadScope)
```

然后，您可以按照您的自定义作用域的作用域规则创建 bean 定义，如下所示：

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

使用自定义作用域实现，您不仅限于以编程方式注册作用域。您还可以使用 `CustomScopeConfigurer` 类以声明方式进行作用域注册，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>



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

1.6. 自定义 Bean 的性质

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

- [生命周期回调\(1.6.1\)](#)
- [ApplicationContextAware 和 BeanNameAware\(1.6.2\)](#)
- [其他 Aware 接口\(1.6.3\)](#)

1.6.1. 生命周期回调

为了与容器对 bean 生命周期的管理进行交互，您可以实现 `Spring InitializingBean` 和 `DisposableBean` 接口。容器为前者调用 `afterPropertiesSet()` 并为后者调用 `destroy()`，以使 Bean 在初始化和销毁 Bean 时执行某些操作。



通常，在现代 Spring 应用程序中，JSR-250 @PostConstruct 和 @PreDestroy 注解被认为是接收生命周期回调的最佳实践。使用这些注解意味着您的 bean 没有耦合到特定于 Spring 的接口。有关详细信息，请参见[使用@PostConstruct 和@PreDestroy\(1.9.9\)](#)。

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

在内部，Spring 框架使用 `BeanPostProcessor` 实现来处理它可以找到的任何回调接口并调用适当的方法。如果您需要自定义特性或其他生命周期行为，默认情况下 Spring 不提供，则您可以自己实现 `BeanPostProcessor`。有关更多信息，请参见[容器扩展点 \(1.8\)](#)。

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

本节介绍了生命周期回调接口。

● 初始化回调

使用 `org.springframework.beans.factory.InitializingBean` 接口，容器在容器上设置了所有必要的属性后，即可执行初始化工作。`InitializingBean` 接口指定一个方法：

Java

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

Kotlin

```
fun afterPropertiesSet()
```

我们建议您不要使用 `InitializingBean` 接口，因为它不必要地将代码耦合到 Spring。另外，我们建议使用 `@PostConstruct` 批注或指定 POJO 初始化方法。对于基于 XML 的配置元数据，可以使用 `init-method` 属性指定具有无效无参数签名的方法的名称。通过 Java 配置，可以使用 `@Bean` 的 `initMethod` 属性。请参阅[接收生命周期回调\(1.12.3/152,153\)](#)。请看以下示例：

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

Java

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

Kotlin

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

前面的示例与下面的示例（由两个列表组成）几乎具有完全相同的效果：

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

Java

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

Kotlin

```
class AnotherExampleBean : InitializingBean {  
  
    override fun afterPropertiesSet() {  
        // do some initialization work  
    }  
}
```

但是，前面两个示例中的第一个示例并未将代码耦合到 Spring。

● 回调销毁

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

Java

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

Kotlin

```
fun destroy()
```

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

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

Java

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

Kotlin

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

前面的示例与下面的示例（由两个列表组成）几乎具有完全相同的效果：

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

Java

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

Kotlin

```
class AnotherExampleBean : DisposableBean {  
  
    override fun destroy() {  
        // do some destruction work (like releasing pooled connections)  
    }  
}
```

但是，前面两个示例中的第一个示例并未将代码耦合到 Spring。



您可以为<bean>标签的 `destroy-method` 属性分配一个特殊的（推断的）值，该值指示 Spring 自动检测特定 bean 类上的公共 `close` 或 `shutdown` 方法。（因此，任何实现 `java.lang.AutoCloseable` 或 `java.io.Closeable` 的类都将匹配。）您还可以在<beans>元素的 `default-destroy-method` 属性上设置此特殊（推断）值，以将此行为应用于整个 bean 集（请参见[默认初始化和销毁方法\(如下\)](#)）。请注意，这是 Java 配置的默认行为。

● 默认的初始化和销毁方法

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

您可以配置 Spring 容器以“look”命名每个 bean 上的 `initialization` 和 `destroy` 回调方法名称。这意味着，作为应用程序开发人员，您可以编写应用程序类并使用称为 `init()` 的初始化回调，而不必为每个 bean 定义配置 `init-method=“init”` 属性。Spring IoC 容器在创建 bean 时（并根据前面描述的标准生命周期回调协定）调用该方法。此功能还对 `initialization` 和 `destroy` 方法回调强制执行一致的命名约定。

假设您的初始化回调方法命名为 `init()`，而 `destroy` 回调方法命名为 `destroy()`。

然后，您的类类似于以下示例中的类：

Java

```
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.");
        }
    }
}
```

Kotlin

```
class DefaultBlogService : BlogService {

    private var blogDao: BlogDao? = null

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

然后，您可以在类似于以下内容的 Bean 中使用该类：

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

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

</beans>
```

顶级`<beans/>`标签属性上存在 `default-init-method` 属性，导致 Spring IoC 容器将 Bean 类上称为 `init` 的方法识别为初始化方法回调。在创建和组装 bean 时，如果 bean 类具有这种方法，则会在适当的时间调用它。

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

如果现有的 Bean 类已经具有按惯例命名的回调方法，你可以通过使用`<bean/>`本身的

`init-method` 和 `destroy-method` 属性指定（在 XML 中）方法名称来覆盖默认值。

Spring 容器保证在为 bean 提供所有依赖项后立即调用配置的初始化回调。因此，在原始 bean 引用上调用了初始化回调，这意味着 AOP 拦截器等尚未应用于 bean。首先完全创建目标 bean，然后应用带有其拦截器链的 AOP 代理（例如）。如果目标 Bean 和代理分别定义，则您的代码甚至可以绕过代理与原始目标 Bean 进行交互。因此，将拦截器应用于 `init` 方法将是不一致的，因为这样做会将目标 Bean 的生命周期耦合到其代理或拦截器，并且当您的代码直接与原始目标 Bean 进行交互时会留下奇怪的语义。

● 结合生命周期机制

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

- `InitializingBean` 和 `DisposableBean` 回调接口
- 自定义 `init()` 和 `destroy()` 方法
- `@PostConstruct` 和 `@PreDestroy` 注解（1.9.9）。

你可以结合这些机制去控制一个给出的 bean。

如果为一个 bean 配置了多个生命周期机制，并且为每个机制配置了不同的方法名称，则每个配置的方法都将按照此注释后列出的顺序运行。
但是，如果为多个生命周期机制中的多个生命周期配置了相同的方法名称（例如，为初始化方法使用 `init()`），则该方法将运行一次，如上一节所述。

为同一个 bean 配置的具有不同初始化方法的多种生命周期机制如下：

1. `@PostConstruct` 方法注解
2. 由 `InitializingBean` 回调接口定义的 `afterPropertiesSet()`。
3. 一个自定义配置的 `init()` 方法

销毁方法以同样的顺序被调用：

1. `@PostDestroy` 方法注解
2. 由 `DisposableBean` 回调接口定义的 `destroy()`。
3. 一个自定义配置的 `destroy()` 方法

● 启动和关闭回调

`Lifecycle` 接口为具有自己的生命周期要求（例如启动和停止某些后台进程）的任何

对象定义了基本方法：

Java

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

Kotlin

```
interface Lifecycle {  
    fun start()  
    fun stop()  
    val isRunning: Boolean  
}
```

任何 Spring 管理的对象都可以实现 `Lifecycle` 接口。然后，当 `ApplicationContext` 本身接收到启动和停止信号时（例如，对于运行时的停止/重新启动场景），它将把这些调用级联到在该上下文中定义的所有 `Lifecycle` 实现。它通过委派给 `LifecycleProcessor` 来做到这一点，如以下列表所示：

Java

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

Kotlin

```
interface LifecycleProcessor : Lifecycle {  
    fun onRefresh()  
    fun onClose()  
}
```

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



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

另外，请注意，不能保证会在销毁之前发出停止通知。在常规关闭时，在传播常规销毁回调之前，所有 `Lifecycle` bean 都会首先收到停止通知。但是，在上下文生存期内进行热刷新或停止刷新尝试时，仅调用 `destroy` 方法。

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

Java

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

Kotlin

```
interface Phased {  
    val phase: Int  
}
```

以下列表显示了 `SmartLifecycle` 接口的定义：

Java

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

Kotlin

```
interface SmartLifecycle : Lifecycle, Phased {  
    val isAutoStartup: Boolean  
    fun stop(callback: Runnable)  
}
```

启动时，相位最低的对象首先启动。停止时，这个顺序就会倒过来。因此，实现 `SmartLifecycle` 并且其 `getPhase()` 方法返回 `Integer.MIN_VALUE` 的对象将是第一个启动且最后一个停止的对象。在这组的另一端，相位值 `Integer.MAX_VALUE` 表示该对象应最后启动并首先停止（可能是因为它依赖于正在运行的其他进程）。在考虑阶段值时，重要的是要知道，任何未实现 `SmartLifecycle` 的“普通” `Lifecycle` 对象的默认阶段为 `0`。因此，任何负相位值都表明对象应在这些标准组件之前开始（并在它们之后停止）。对于任何正相位值，反之亦然。

`SmartLifecycle` 定义的 `stop` 方法接受回调。任何实现都必须在该实现的关闭过程完成后调用该回调的 `run()` 方法。这将在必要时启用异步关闭，因为 `LifecycleProcessor` 接口的默认实现 `DefaultLifecycleProcessor` 会等待每个阶段内的对象组的超时值等待调用该回调。默认的每阶段超时为 30 秒。您可以通过在上下文中定义一个名为 `lifecycleProcessor` 的 bean 来覆盖默认的生命周期处理器实例。如果只想修改超时，则定义以下内容即可：

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

如前所述，`LifecycleProcessor` 接口还定义了用于刷新和关闭上下文的回调方法。稍后驱动关闭处理，就好像已经显式调用了 `stop()` 一样，但是它在上下文关闭时发生。另一方面，“refresh”回调启用 `SmartLifecycle` bean 的另一个功能。刷新上下文时（在所有对象都被实例化和初始化之后），该回调将被调用。届时，默认生命周期处理器将检查每个 `SmartLifecycle` 对象的 `isAutoStartup()` 方法返回的布尔值。如果为 `true`，则从那时开始启动该对象，而不是等待上下文或它自己的 `start()` 方法的显式调用（与上下文刷新不同，对于标准上下文实现，上下文启动不会自动发生）。相位值和任何“依赖”关系决定了启动顺序，如前所述。

● 在非 Web 应用程序中优雅地关闭 Spring IoC 容器



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

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

要注册关闭挂钩，请调用在 `ConfigurableApplicationContext` 接口上声明的 `registerShutdownHook()` 方法，如以下示例所示：

Java

```
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...
    }
}
```

Kotlin

```
import org.springframework.context.support.ClassPathXmlApplicationContext

fun main() {
    val ctx = ClassPathXmlApplicationContext("beans.xml")

    // add a shutdown hook for the above context...
    ctx.registerShutdownHook()

    // app runs here...

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

1.6.2. ApplicationContextAware 和 BeanNameAware 接口

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

Java

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

Kotlin

```
interface ApplicationContextAware {  
  
    @Throws(BeansException::class)  
    fun setApplicationContext(applicationContext: ApplicationContext)  
}
```

因此，bean 可以通过 `ApplicationContext` 接口或通过将引用转换为该接口的已知子类（例如 `ConfigurableApplicationContext`，它公开了其他功能）来以编程方式操纵创建它们的 `ApplicationContext`。一种用途是通过编程方式检索其他 bean。有时，此功能很有用。但是，通常应避免使用它，因为它将代码耦合到 Spring，并且不遵循控制反转样式，在该样式中，将协作者作为属性提供给 bean。`ApplicationContext` 的其他方法提供对文件资源的访问，发布应用程序事件以及访问 `MessageSource`。这些附加功能在 `ApplicationContext` 的其他功能（1.15）中进行了描述。

自动装配是获得对 `ApplicationContext` 的引用的另一种选择。传统的构造函数和 `byType` 自动装配模式（如“[自动装配协作器（1.4.5）](#)”中所述）可以分别为构造函数参数或 `setter` 方法参数提供 `ApplicationContext` 类型的依赖项。要获得更大的灵活性，包括能够自动连接字段和使用多个参数方法，请使用基于注释的自动装配功能。如果这样做，则将 `ApplicationContext` 自动连装配需要使用 `ApplicationContext` 类型的字段，构造函数参数或方法参数中（如果有问题的字段，构造函数或方法带有`@Autowired` 批注）。有关更多信息，请参见[使用`@Autowired`（1.9.2）](#)。

当 `ApplicationContext` 创 建 一 个 实 现
org.springframework.beans.factory.BeanNameAware 接口，该类提供了对在其关联

对象定义中定义的名称的引用。以下列表显示了 BeanNameAware 接口的定义：

Java

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

Kotlin

```
interface BeanNameAware {  
  
    @Throws(BeansException::class)  
    fun setBeanName(name: String)  
}
```

回调应该在一个普通 bean 的属性填充后，但也应该在一个初始化回调前，比如 InitializingBean, afterPropertiesSet, 或者一个自定义初始化方法(init-method)。

1.6.3. 其他 Aware 接口

除了 ApplicationContextAware 和 BeanNameAware(前面已经讨论过)之外，Spring 还提供了多种 Aware 回调接口，这些接口使 Bean 向容器指示它们需要某种基础结构依赖性。通常，名称表示依赖项类型。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
ApplicationEventPublisherAware	Event publisher of the enclosing ApplicationContext.	Additional Capabilities of the ApplicationContext
BeanClassLoaderAware	Class loader used to load the bean classes.	Instantiating Beans
BeanFactoryAware	Declaring BeanFactory.	ApplicationContextAware and BeanNameAware
BeanNameAware	Name of the declaring bean.	ApplicationContextAware and BeanNameAware
BootstrapContextAware	Resource adapter BootstrapContext the container runs in. Typically available only in JCA-aware ApplicationContext instances.	JCA CCI
LoadTimeWeaverAware	Defined weaver for processing class definition at load time.	Load-time Weaving with AspectJ in the Spring Framework

Name	Injected Dependency	Explained in...
<code>MessageSourceAware</code>	Configured strategy for resolving messages (with support for parametrization and internationalization).	Additional Capabilities of the ApplicationContext
<code>NotificationPublisherAware</code>	Spring JMX notification publisher.	Notifications
<code>ResourceLoaderAware</code>	Configured loader for low-level access to resources.	Resources
<code>ServletConfigAware</code>	Current <code>ServletConfig</code> the container runs in. Valid only in a web-aware Spring <code>ApplicationContext</code> .	Spring MVC
<code>ServletContextAware</code>	Current <code>ServletContext</code> the container runs in. Valid only in a web-aware Spring <code>ApplicationContext</code> .	Spring MVC

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

1. 7. Bean 定义继承

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

如果您以编程方式使用 `ApplicationContext` 接口，则子 bean 定义由 `ChildBeanDefinition` 类表示。大多数用户不在此级层级上与他们合作。相反，它们在诸如 `ClassPathXmlApplicationContext` 之类的类中声明性地配置 Bean 定义。当使用基于 XML 的配置元数据时，可以通过使用父属性来指示子 Bean 定义，并指定父 Bean 作为该属性的值。以下示例显示了如何执行此操作：当使用基于 XML 的配置元数据时，可以通过使用父属性来指示子 Bean 定义，并指定父 Bean 作为该属性的值。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"> ①
    <property name="name" value="override"/>
    <!-- the age property value of 1 will be inherited from parent -->
</bean>

```

① 注意 `parent` 属性

如果未指定子 bean 定义，则使用父定义中的 bean 类，但也可以覆盖它。在后一种情况下，子 bean 类必须与父类兼容（也就是说，它必须接受父类的属性值）。

子 bean 定义从父对象继承作用域，构造函数参数值，属性值和方法覆写，并可以选择添加新值。你指定的任何作用域，初始化方法，`destroy` 方法或静态工厂方法设置都会覆盖相应的父设置。

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

前面的示例使用 `abstract` 属性将父 bean 定义显式标记为抽象类。如果父定义未指定为一个类，则需要将父 bean 定义显式标记为抽象类，如以下示例所示：

```

<bean id="inheritedTestBeanWithoutClass" abstract="true">
  <property name="name" value="parent"/>
  <property name="age" value="1"/>
</bean>

<bean id="inheritsWithClass" class="org.springframework.beans.DerivedTestBean"
      parent="inheritedTestBeanWithoutClass" init-method="initialize">
  <property name="name" value="override"/>
  <!-- age will inherit the value of 1 from the parent bean definition-->
</bean>

```

父 bean 不能单独实例化，因为它不完整，并且还被明确标记为抽象。当定义是抽象的时，它只能用作纯模板 bean 定义，用作子定义的父定义。如果尝试使用这么一个抽象父类 bean 于它自身，比如咱引用它自己作为另一个 bean 的 `ref` 属性或者使用父类的 bean ID 显式的调用 `getBean()` 方法将会返回一个错误。类似的，容器内部的 `preInstantiateSingletons()` 方法将会忽略那些被定义为抽象的 bean。



`ApplicationContext` 默认的将所有单例进行预初始化。因此，有个很重要（至少是对单例 bean）的事就是你有一个仅仅打算作为模板的（父类）bean 定义并且它被指定了一个类，你必须确定将 `abstract` 属性设为 `true`，否则应用上下文将会实际（或尝试）去预编译它。

1.8. 容器扩展点

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

1.8.1. 通过使用 BeanPostProcessor 自定义 Beans

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

您可以配置多个 `BeanPostProcessor` 实例，并且可以通过设置 `order` 属性来控制这些 `BeanPostProcessor` 实例的运行顺序。仅当 `BeanPostProcessor` 实现 `Ordered` 接口时，才可以设置此属性。如果编写自己的 `BeanPostProcessor`，则也应该考虑实现 `Ordered` 接口。有关更多详细信息，请参见 `BeanPostProcessor` 和 `Ordered` 接口的 javadoc。另请参见有关[以编程方式注册 BeanPostProcessor 实例](#)的说明。

`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` 实例完成其工作。



`BeanPostProcessor` 实例是按容器划分作用域的。仅在使用容器层次结构时，这才有意义。如果在一个容器中定义 `BeanPostProcessor`，它将仅对该容器中的 bean 进行后处理。换句话说，一个容器中定义的 bean 不会由另一个容器中定义的 `BeanPostProcessor` 进行后处理，即使这两个容器是同一层次结构的一部分。

要更改实际的 bean 定义（即定义 bean 的蓝图），您需要使用 `BeanFactoryPostProcessor`，如使用 [BeanFactoryPostProcessor 自定义配置元数据\(1.8.2\)](#) 中所述。

`org.springframework.beans.factory.config.BeanPostProcessor` 接口恰好由

两个回调方法组成。将此类注册为容器的 post-processor 时，对于容器创建的每个 bean 实例，后处理器都会在容器初始化方法（例如 `InitializingBean.afterPropertiesSet()` 或任何声明的 `init` 方法），并在任何 bean 初始化之后被调用。post-processor 可以对 bean 实例执行任何操作，包括完全忽略回调。Bean 后处理器通常检查回调接口，或者可以用代理包装 Bean。一些 Spring AOP 基础结构类被实现为 bean 后处理器，以提供代理包装逻辑。

`ApplicationContext` 自动检测配置元数据中实现 `BeanPostProcessor` 接口的定义的所有 bean。`ApplicationContext` 将这些 bean 注册为 post-processor，以便以后在 bean 创建时可以调用它们。Bean 后处理器可以与其他任何 Bean 相同的方式部署在容器中。

请注意，在配置类上使用 `@Bean` 工厂方法声明 `BeanPostProcessor` 时，工厂方法的返回类型应该是实现类本身，或者至少是 `org.springframework.beans.factory.config.BeanPostProcessor` 接口。指示该 bean 的后处理器性质。否则，`ApplicationContext` 无法在完全创建之前按类型自动检测它。由于需要早期实例化 `BeanPostProcessor` 以便将其应用于上下文中的其他 Bean 初始化，因此这种早期类型检测至关重要。

以编程方式注册 `BeanPostProcessor` 实例

尽管建议的 `BeanPostProcessor` 注册方法是通过 `ApplicationContext` 自动检测（如前所述），但是您可以使用 `addBeanPostProcessor` 方法以编程方式针对 `ConfigurableBeanFactory` 注册它们。当您需要在注册之前评估条件逻辑，甚至需要跨层次结构的上下文复制 `Beanpost-processor` 时，这将非常有用。但是请注意，以编程方式添加的 `BeanPostProcessor` 实例不遵守 `Ordered` 接口。这样，注册的顺序决定了执行的顺序。还要注意，以编程方式注册的 `BeanPostProcessor` 实例始终在通过自动检测注册的 `BeanPostProcessor` 实例之前进行处理，而不考虑任何明确的顺序。

`BeanPostProcessor` 实例和 AOP 自动代理

实现 `BeanPostProcessor` 接口的类是特殊的，并且容器对它们的处理方式有所不同。作为 `ApplicationContext` 特殊启动阶段的一部分，所有 `BeanPostProcessor` 实例和它们直接引用的 Bean 都会在启动时实例化。接下来，以排序方式注册所有 `BeanPostProcessor` 实例，并将其应用于容器

中的所有其他 bean。因为 AOP 自动代理是作为 BeanPostProcessor 本身实现的，所以 BeanPostProcessor 实例或它们直接引用的 bean 都没有资格进行自动代理，因此也没有编织方面的内容。

对于任何此类 Bean，您都应该看到一条信息日志消息：Bean someBean 不适合所有 BeanPostProcessor 接口进行处理（例如：不具备自动代理功能）。

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

下面的示例演示如何在 ApplicationContext 中编写，注册和使用 BeanPostProcessor 实例。

● 例子：Hello World, BeanPostProcessor-style

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

以下列表显示了自定义 BeanPostProcessor 实现类的定义：

Java

```
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;
    }
}
```

Kotlin

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

class InstantiationTracingBeanPostProcessor : BeanPostProcessor {

    // simply return the instantiated bean as-is
    override fun postProcessBeforeInitialization(bean: Any, beanName: String): Any? {
        return bean // we could potentially return any object reference here...
    }

    override fun postProcessAfterInitialization(bean: Any, beanName: String): Any? {
        println("Bean '$beanName' created : $bean")
        return bean
    }
}
```

以下 bean 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>
```

注意，如何单纯定义 `InstantiationTracingBeanPostProcessor`。它甚至没有名称，并且因为它是 Bean，所以可以像注入其他任何 Bean 一样对其进行依赖注入。（前面的配置还定义了一个由 Groovy 脚本支持的 bean。Spring 动态语言支持在标题为 [Dynamic Language Support](#) 的一章中有详细介绍。）

下面的 Java 应用程序运行前面的代码和配置：

Java

```
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 = ctx.getBean("messenger", Messenger.class);
        System.out.println(messenger);
    }

}
```

Kotlin

```
import org.springframework.beans.factory.getBean

fun main() {
    val ctx = ClassPathXmlApplicationContext("scripting/beans.xml")
    val messenger = ctx.getBean<Messenger>("messenger")
    println(messenger)
}
```

前面的应用程序的输出类似于以下内容：

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

● 例子：RequiredAnnotationBeanPostProcessor

将回调接口或注解与自定义 `BeanPostProcessor` 实现结合使用是扩展 Spring IoC 容器的常用方法。Spring 的 `RequiredAnnotationBeanPostProcessor` 是一个示例，它是 Spring 发行版附带的 `BeanPostProcessor` 实现，可以确保标有（任意）注解的 bean 上的 JavaBean 属性实际上（被配置为）依赖注入了一个值。

1.8.2. 使用 BeanFactoryPostProcessor 自定义配置元数据

我们 要 看 的 下 一 个 扩 展 点 是
`org.springframework.beans.factory.config.BeanFactoryPostProcessor`。该接口的语义类似于 `BeanPostProcessor` 的语义，但有一个主要区别：
`BeanFactoryPostProcessor` 对 Bean 配置元数据进行操作。也就是说，Spring IoC 容器允许 `BeanFactoryPostProcessor` 读取配置元数据，并有可能在容器实例化除 `beanFactoryPostProcessor` 实例以外的任何 Bean 之前更改它。

您可以配置多个 `BeanFactoryPostProcessor` 实例，并且可以通过设置 `order` 属性来控制这些 `BeanFactoryPostProcessor` 实例的运行顺序。但是，仅当 `BeanFactoryPostProcessor` 实现 `Ordered` 接口时，才可以设置此属性。如果您编写自己的 `BeanFactoryPostProcessor`，也应该考虑实现 `Ordered` 接口。有关更多详细信息，请参见 `BeanFactoryPostProcessor` 和 `Ordered` 接口的 javadoc。

如果要更改实际的 bean 实例（即从配置元数据创建的对象），则需要使用 `BeanPostProcessor`（在前面的 [使用 BeanPostProcessor 自定义 Bean](#) 中进行了介绍）。从技术上讲，可以在 `BeanFactoryPostProcessor` 中使用 Bean 实例（例如，通过使用 `BeanFactory.getBean()`），但是这样做会导致过早的 Bean 实例化，从而违反了标准容器的生命周期。这可能会导致负面影响，例如绕过 bean 后处理。



同样，`BeanFactoryPostProcessor` 实例是按容器划分作用域的。仅在使用容器层次结构时才有意义。如果在一个容器中定义 `BeanFactoryPostProcessor`，则仅将其应用于该容器中的 Bean 定义。一个容器中的 Bean 定义不会由另一个容器中的 `BeanFactoryPostProcessor` 实例进行后处理，即使两个容器都在同一层次结构也是如此。

Bean 工厂后处理器在 `ApplicationContext` 中声明时会自动运行，以便将更改应用于定义容器的配置元数据。Spring 包含许多预定义的 bean 工厂 post-processors，例如 `PropertyOverrideConfigurer` 和 `PropertySourcesPlaceholderConfigurer`。您也可以使用自定义 `BeanFactoryPostProcessor`，例如注册自定义属性编辑器。



与 `BeanPostProcessors` 一样，您通常不想配置 `BeanFactoryPostProcessors` 用于延迟初始化。如果没有其他 bean 引用 Bean (Factory) PostProcessor，则该后处理器将完全不会实例化。因此，将其标记为延迟初始化将被忽略，即使您在 `<beans/>` 标签的声明中将 `default-lazy-init` 属性设置为 `true`，`Bean (Factory) PostProcessor` 也会被实例化。

● 例子：类名替换 PropertySourcePlaceholderConfiguration

您可以使用 `PropertySourcesPlaceholderConfigurer` 来通过使用标准 Java 属性格

式将豆定义中的属性值外部化到单独的文件中。这样做使部署应用程序的人员可以自定义特定于环境的属性，例如数据库 URL 和密码，而不会为修改容器的一个或多个主要 XML 定义文件带来复杂性或风险。考虑以下基于 XML 的配置元数据片段，其中定义了具有占位符值的 DataSource：

```
<bean  
    class="org.springframework.context.support.PropertySourcesPlaceholderConfigurer">  
        <property name="locations" value="classpath:com/something/jdbc.properties"/>  
    </bean>  
  
<bean id="dataSource" destroy-method="close"  
    class="org.apache.commons.dbcp.BasicDataSource">  
    <property name="driverClassName" value="${jdbc.driverClassName}"/>  
    <property name="url" value="${jdbc.url}"/>  
    <property name="username" value="${jdbc.username}"/>  
    <property name="password" value="${jdbc.password}"/>  
    </bean>
```

该示例显示了从外部属性文件配置的属性。在运行时，将 PropertySourcesPlaceholderConfigurer 应用于替换数据源某些属性的元数据。将要替换的值指定为 \${property-name} 格式的占位符，该格式遵循 Ant 和 log4j 和 JSP EL 样式。

实际值来自标准 Java 属性格式的另一个文件：

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

因此，\${jdbc.username} 字符串在运行时将被替换为值“sa”，并且与属性文件中的键匹配的其他占位符值也是如此。PropertySourcesPlaceholderConfigurer 检查 Bean 定义的大多数属性和属性中的占位符。此外，您可以自定义占位符前缀和后缀。

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

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

PropertySourcesPlaceholderConfigurer 不仅在您指定的属性文件中查找属性 properties。默认情况下，如果无法在指定的属性文件中找到属性，则会检查 Spring Environment 属性和常规 Java System 属性。

您可以使用 `PropertySourcesPlaceholderConfigurer` 替换类名称，这在您必须在运行时选择特定的实现类时有时很有用。下面这个图讲了如何去做：



```
<bean  
class="org.springframework.beans.factory.config.PropertySourcesPlacehol  
derConfigurer">  
    <property name="locations">  
        <value>classpath:com/something/strategy.properties</value>  
    </property>  
    <property name="properties">  
        <value>custom.strategy.class=com.something.DefaultStrategy</value>  
    </property>  
</bean>  
  
<bean id="serviceStrategy" class="${custom.strategy.class}" />
```

如果无法在运行时将该类解析为有效的类，则在将要创建该 bean 时（在非延迟初始化 bean 的 `ApplicationContext` 的 `preInstantiateSingletons()` 阶段期间），该 bean 的解析将失败。

● 例子：`PropertyOverrideConfigurer`

另一个 bean 工厂 post-processor 程序 `PropertyOverrideConfigurer` 类似于 `PropertySourcesPlaceholderConfigurer`，但是与后者不同，原始定义对于 bean 属性可以具有默认值或完全没有值。如果覆盖的属性文件没有某个 bean 属性的条目，则使用默认的上下文定义。

注意，bean 定义不知道会被覆盖，因此在覆盖配置器正在被使用的 XML 定义文件中不能立即看出。在多个对同一个 bean 属性定义了不同值的 `PropertyOverrideConfigurer` 实例，由于覆盖机制，只有最后一个能够生效。

属性配置按照下列格式：

```
beanName.property=value
```

下列列出一个例子：

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

此示例文件可与包含一个名为 `dataSource` 的 bean 的容器定义一起使用，该 bean 具

有驱动程序和 url 属性。

复合属性名也被支持，只要除了最终属性被覆写之外每个路径上的组件都已经为非空（大概是由构造函数初始化的）。在下面的示例中，tom bean 的 fred 属性的 bob 属性的 sammy 属性设置为标量值 123：

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



指定的替代值始终是字面值。它们不会转换为 bean 引用。当 XML bean 定义中的原始值指定 bean 引用时，此约定也适用。

使用 Spring 2.5 中引入的[上下文\(context\)](#)名称空间，可以使用专用配置元素配置属性覆盖，如以下示例所示：

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

1.8.3. 使用 FactoryBean 自定义实例化逻辑

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

`FactoryBean` 接口是可插入 Spring IoC 容器的实例化逻辑的一点。如果您有复杂的初始化代码，而不是（可能）冗长的 XML 量，可以用 Java 更好地表达，则可以创建自己的 `FactoryBean`，在该类中编写复杂的初始化，然后将自定义 `FactoryBean` 插入容器。

`FactoryBean` 接口提供了三个方法：

- `Object getObject()`：返回一个工厂创建的实例，并根据工厂返回的是否是 `Singleton` 或者 `prototype`，这个实力可能能被共享。
- `boolean isSingleton()`：如果工厂返回单例则返回 `true`，否则返回 `false`。
- `Class getObjectType()`：根据 `getObject()` 方法返回的对象返回类类型，或者位置的情况下返回 `null`。

Spring 框架中的许多地方都使用了 `FactoryBean` 概念和接口。它起码有 50 个实现。

当您需要向容器请求一个实际的 `FactoryBean` 实例本身而不是由它产生的 bean 时，请在调用 `ApplicationContext` 的 `getBean()` 方法时在该 bean 的 ID 前面加上一个 & 符号。所以对于给定的一个 id 为 `myBean` 的 `FactoryBean` 时，调用 `getBean("myBean")` 会得到 `FactoryBean` 的产品，如果调用 `getBean("&myBean")` 则返回 `FactoryBean` 实例本身。

1.9. 基于注解的容器配置

对于配置 Spring 注解比 XML 要好吗？

基于注释的配置的引入提出了一个问题，即这种方法是否比 XML“更好”。简短的答案是“各有所长”。长话短说，每种方法都有其优缺点，通常，由开发人员决定哪种策略更适合他们。由于定义方式的不同，注释在声明中提供了很多上下文，从而使配置更短，更简洁。但是，XML 擅长装配组件组件而不接触其源代码或重新编译它们。一些开发人员更喜欢将在源代码中装配类，而另一些开发人员则认为带注解的类不再是 POJO，而且，该配置变得分散且难以控制。

无论选择如何，Spring 都可以容纳两种样式，甚至可以将它们混合在一起。值得指出的是，通过其 [JavaConfig\(1.12\)](#) 选项，Spring 允许以非侵入方式使用注释，而无需接触目标组件的源代码，并且就工具而言，[Spring Tools for Eclipse](#) 支持所有配置样式。

基于注释的配置提供了 XML 设置的替代方法，该配置依赖字节码元数据来装配组件，而不是标签声明。通过使用相关类，方法或字段声明上的注解，开发人员无需使用 XML 来描述 bean 的装配，而是将配置转入组件类本身。如[示例：RequiredBeansProcessor\(1.8.2 上面\)](#)，结合使用 [BeanPostProcessor](#) 和注释，是扩展 Spring IoC 容器的常用方法。例如，Spring2.0 引入了使用@Required 注释强制执行必需属性的可能性。Spring2.5 使得可以遵循相同的通用方法来驱动 Spring 的依赖项注入。本质上，@Autowired 注解提供的功能与[自动装配协作器（1.4.5）](#) 中所述的功能相同，但具有更细粒度的控制和更广泛的适用性。Spring2.5 还添加了对 JSR-250 批注的支持，例如@PostConstruct 和@PreDestroy。Spring3.0 增加了对 [javax.inject](#) 包中包含的 JSR-330(Java 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，请参见[相关章节\(1.11\)](#)。



注释注入在 XML 注入之前执行。因此，通过两种方法装配的属性 XML 配置将覆盖注解。

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

```

(隐式注册的 post-processors 包括 `AutowiredAnnotationBeanPostProcessor`, `CommonAnnotationBeanPostProcessor`, `PersistenceAnnotationBeanPostProcessor` 和前述的 `RequiredAnnotationBeanPostProcessor`。)

 `<context: annotation-config/>` 仅在定义它的相同应用程序上下文中查找关于 bean 的注解。就是说，如果将 `<context : annotation-config/>` 放在 `DispatcherServlet` 的 `WebApplicationContext` 中，它将仅检查控制器中的`@Autowired` bean，而不检查服务。

1.9.1. @Required 注解

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

Java

```

public class SimpleMovieLister {

    private MovieFinder movieFinder;

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

    // ...
}

```

Kotlin

```
class SimpleMovieLister {  
  
    @Required  
    lateinit var movieFinder: MovieFinder  
  
    // ...  
}
```

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



从 Spring Framework 5.1 开始，`@Required` 批注已正式弃用，转而使用构造函数注入进行必需的设置（或 `InitializingBean.afterPropertiesSet()` 的自定义实现以及 bean 属性 `setter` 方法）。

1.9.2. 使用`@Autowired`



在本节中的示例中，可以使用 JSR 330 的`@Inject` 注释代替 Spring 的`@Autowired` 注释。有关更多详细信息，请参见[此处\(1.11\)](#)。

您可以将`@Autowired` 注释应用于构造函数，如以下示例所示：

Java

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

Kotlin

```
class MovieRecommender @Autowired constructor(  
    private val customerPreferenceDao: CustomerPreferenceDao)
```

从 Spring Framework 4.3 开始，如果目标 bean 仅定义一个构造函数作为开始，则不再需要在此类构造函数上使用 @Autowired 注解。但是，如果有多个构造函数可用，并且没有主/默认构造函数，则至少一个构造函数必须使用 @Autowired 注释，以指示容器使用哪个。有关详细信息，请参见有关[构造函数解析](#)的讨论。

您还可以将 @Autowired 注解应用于传统的 setter 方法，如以下示例所示：

Java

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

Kotlin

```
class SimpleMovieLister {  
  
    @Autowired  
    lateinit var movieFinder: MovieFinder  
  
    // ...  
}
```

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：

Java

```
public class MovieRecommender {  
  
    private MovieCatalog movieCatalog;  
  
    private CustomerPreferenceDao customerPreferenceDao;  
  
    @Autowired  
    public void prepare(MovieCatalog movieCatalog,  
                        CustomerPreferenceDao customerPreferenceDao) {  
        this.movieCatalog = movieCatalog;  
        this.customerPreferenceDao = customerPreferenceDao;  
    }  
  
    // ...  
}
```

Kotlin

```
class MovieRecommender {  
  
    private lateinit var movieCatalog: MovieCatalog  
  
    private lateinit var customerPreferenceDao: CustomerPreferenceDao  
  
    @Autowired  
    fun prepare(movieCatalog: MovieCatalog,  
               customerPreferenceDao: CustomerPreferenceDao) {  
        this.movieCatalog = movieCatalog  
        this.customerPreferenceDao = customerPreferenceDao  
    }  
  
    // ...  
}
```

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

Java

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

Kotlin

```
class MovieRecommender @Autowired constructor(  
    private val customerPreferenceDao: CustomerPreferenceDao) {  
  
    @Autowired  
    private lateinit var movieCatalog: MovieCatalog  
  
    // ...  
}
```

确保目标组件（例如，MovieCatalog 或 CustomerPreferenceDao）由用于@Autowired 注解的注入点的类型一致地声明。否则，注入可能会由于运行时出现“找不到类型匹配(notype match found)”错误而失败。



对于通过类路径扫描找到的 XML 定义的 bean 或组件类，容器通常预先知道具体的类型。但是，对于@Bean 工厂方法，您需要确保声明的返回类型足够明显。对于实现多个接口的组件或可能由其实现类型引用的组件，请考虑在工厂方法中声明最具体的返回类型（至少根据引用您的 bean 的注入点的要求进行声明）。

您还可以通过将@Autowired 注解添加到需要该类型数组的字段或方法中，指示 Spring 从 ApplicationContext 提供特定类型的所有 bean，如以下示例所示：

Java

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

Kotlin

```
class MovieRecommender {  
  
    @Autowired  
    private lateinit var movieCatalogs: Array<MovieCatalog>  
  
    // ...  
}
```

如以下示例所示，这同样适用于泛型集合：

Java

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

Kotlin

```
class MovieRecommender {  
  
    @Autowired  
    lateinit var movieCatalogs: Set<MovieCatalog>  
  
    // ...  
}
```

你的目标 beans 可以实现 org.springframework.core.Ordered 接口或者使用 @Order 或者标准 @Priority 注解，如果你希望你的数组或列表中的项按照指定顺序排列。否则，他们的顺序将按照容器中相关目标 bean 的注册顺序进行。



您可以在目标类级别和 @Bean 方法上声明 @Order 注解，这可能适用于单个 bean 定义（如果使用同一 bean 类的多个定义）。@Order 值可能会影响注入点的优先级，但请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。

注意，标准 javax.annotation.Priority 注解在 @Bean 级别不可用，因为无法在方法上声明它。它的语义可以通过 @Order 值与 @Primary 结合在每种类型的单个 bean 上进行建模。

只要预期的键类型为 String，甚至输入类型的 Map 实例也可以自动装配。映射值包含所有预期类型的 bean，并且键包含相应的 bean 名称，如以下示例所示：

Java

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

Kotlin

```
class MovieRecommender {  
  
    @Autowired  
    lateinit var movieCatalogs: Map<String, MovieCatalog>  
  
    // ...  
}
```

默认情况下，当给定注入点没有匹配的候选 bean 可用时，自动装配将失败。对于声明的数组、集合或映射，至少应有一个匹配元素。

默认行为是将带注解的方法和字段视为指示所需的依赖项。您可以按照以下示例中所示

的方式更改此行为,从而使框架可以通过将其标记为不需要来跳过不满足要求的注入点(即,通过将`@Autowired` 中的 `required` 属性设置为 `false`) :

Java

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

Kotlin

```
class SimpleMovieLister {  
  
    @Autowired(required = false)  
    var movieFinder: MovieFinder? = null  
  
    // ...  
}
```

如果不需要的方法(或者在多个参数的情况下,其中一个依赖项)不可用,则根本不会调用该方法。在这种情况下,完全不需要填充非必需字段,而保留其默认值。

注入的构造函数和工厂方法参数是一种特殊情况,因为由于 Spring 的构造函数解析算法可能会处理多个构造函数,因此`@Autowired` 中的 `required` 属性的含义有所不同。默认情况下,有效地需要构造函数和工厂方法参数,但是在单构造函数场景中有一些特殊规则,例如,如果没有可用的匹配 bean,则多元素注入点(数组,集合,映射)解析为空实例。这允许一种通用的实现模式,其中所有依赖项都可以在唯一的多参数构造函数中声明-例如,声明为不带`@Autowired` 批注的单个公共构造函数。



对于任何给定 bean 类只有一个构造函数能声明@Autowired，并将必选属性设置为 true，这表示在用作 Spring bean 时可以进行自动装配的构造函数。因此，如果必填属性保留为其默认值 vtrue，则@Autowired 只能注释单个构造函数。如果多个构造函数声明了注解，则它们都必须声明 required = false 才能被视为自动装配的候选对象（类似于 XML 中的 autowire=constructor）。可以满足匹配 Spring 容器中 bean 的带有最多数量依赖项的构造函数将被选中。如果没有一个候选者满足要求，则将使用主/默认构造函数（如果存在）。同样，如果一个类声明了多个构造函数，但都没有使用@Autowired 进行注释，则将使用主/默认构造函数（如果存在）。如果一个类只声明一个单一的构造函数开始，即使没有注释，也将始终使用它。请注意，带注解的构造函数不必是公共的。

建议在 setter 方法上使用@Autowired 的 required 属性，而不推荐使用已弃用的@Required 注解。将 required 属性设置为 false 表示该属性对于自动装配不是必需的，并且如果无法自动装配，则将忽略该属性。另一方面，@Required 更为强大，因为它可以通过容器支持的任何方式强制设置属性，并且如果未定义任何值，则会引发相应的异常。

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

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

从 Spring Framework5.0 开始，还可以用@Nullable 注解(在任何包中任何形式，例 JSR-305 中的 `javax.annotation.Nullable`)或仅利用 Kotlin 内置的 nullsafety 支持：

Java

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

Kotlin

```
class SimpleMovieLister {  
  
    @Autowired  
    var movieFinder: MovieFinder? = null  
  
    // ...  
}
```

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

Java

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

Kotlin

```
class MovieRecommender {  
  
    @Autowired  
    lateinit var context: ApplicationContext  
  
    // ...  
}
```

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

1.9.3. 使用 @Primary 基于注解的自动装配的调整

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

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

Java

```
@Configuration
public class MovieConfiguration {

    @Bean
    @Primary
    public MovieCatalog firstMovieCatalog() { ... }

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

    // ...
}
```

Kotlin

```
@Configuration
class MovieConfiguration {

    @Bean
    @Primary
    fun firstMovieCatalog(): MovieCatalog { ... }

    @Bean
    fun secondMovieCatalog(): MovieCatalog { ... }

    // ...
}
```

通过前面的配置，以下 `MovieRecommender` 与 `firstMovieCatalog` 自动装配：

Java

```
public class MovieRecommender {  
  
    @Autowired  
    private MovieCatalog movieCatalog;  
  
    // ...  
}
```

Kotlin

```
class MovieRecommender {  
  
    @Autowired  
    private lateinit var movieCatalog: MovieCatalog  
  
    // ...  
}
```

相关 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. 使用 Qualifiers 基于注解的自动装配的调整

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

Java

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

Kotlin

```
class MovieRecommender {  
  
    @Autowired  
    @Qualifier("main")  
    private lateinit var movieCatalog: MovieCatalog  
  
    // ...  
}
```

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

Java

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

Kotlin

```
class MovieRecommender {

    private lateinit var movieCatalog: MovieCatalog

    private lateinit var customerPreferenceDao: CustomerPreferenceDao

    @Autowired
    fun prepare(@Qualifier("main") movieCatalog: MovieCatalog,
               customerPreferenceDao: CustomerPreferenceDao) {
        this.movieCatalog = movieCatalog
        this.customerPreferenceDao = customerPreferenceDao
    }

    // ...
}
```

下面的例子展示了相关 bean 的定义：

```
<?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"/> ①

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

    <bean class="example.SimpleMovieCatalog">
        <qualifier value="action"/> ②

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

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

</beans>
```

① 值为 `main` 的 Bean 与限定有相同值的构造函数参数关联。

② 值为 `action` 的 Bean 与限定有相同值的构造函数参数关联。

对于 `fallback` 匹配，`bean` 名称被视为默认的限定符值。因此，可以定义一个 `id` 为 `main` 而不是嵌套的 `qualifier` 元素定义 `bean`，从而得到相同的匹配结果。尽管可以使用此约定按名称引用特定的 `bean`，但 `@Autowired` 基本上是关于带有可选语义限定符的类型

驱动的注入。这意味着，即使带有 Bean 名称 `fallback` 的限定符值，在类型匹配集中也始终具有狭窄的语义。它们没有在语义上表示对唯一 `bean id` 的引用。好的限定符值是主要的或 EMEA 的或持久的，表示特定组件的特性，该特性独立于 `bean id`，在使用匿名 bean 定义（例如上例中的定义）的情况下，可以自动生成这些特性。

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



在类型匹配的候选对象中，让限定符值根据目标 Bean 名称进行选择，在注入点不需要 `@Qualifier` 注释。如果没有其他解决方案指示符（例如限定符或主标记），则对于非唯一依赖性情况，Spring 将注入点名称（即字段名称或参数名称）与目标 Bean 名称进行匹配，然后选择同名候选（如果有）。

就是说，如果您打算按名称表示注释驱动的注入，则即使它能够在类型匹配的候选对象中按 bean 名称进行选择，也不要主要使用 `@Autowired`。而是使用 JSR-250 `@Resource` 注解，该批注的语义定义是通过其唯一名称标识特定目标组件，而声明的类型与匹配过程无关。`@Autowired` 具有不同的语义：在按类型选择候选 bean 之后，仅在那些类型选择的候选中考虑指定的 String 限定符值（例如，将 `account` 限定符与标记有相同限定符标签的 bean 进行匹配）。

对于本身定义为集合、`Map` 或数组类型的 bean，`@Resource` 是一个很好的解决方案，通过唯一名称引用特定的集合或数组 bean。就是说，从 4.3 版本开始，只要元素类型信息保留在 `@Bean` 返回类型签名或集合继承层次结构中，就可以通过 Spring 的 `@Autowired` 类型匹配算法来匹配 `Map` 和数组类型。在这种情况下，您可以使用限定符值在同类型的集合中进行选择，如上一段所述。

从 4.3 开始，`@Autowired` 还考虑了自我引用以进行注入（即，引用回当前注入的 Bean）。请注意，自我注入是一个 fallback。对其他组件的常规依赖始终优先。从这个意义上说，自我引用不参与常规的候选选择，因此尤其是绝不是主要的。相反，它们总是以最低优先级结束。实际上，应该仅将自我引用用作最后的手段（例如，通过 Bean 的事务代理在同一实例上调用其他方法）。在这种情况下，请考虑将受影响的方法分解为单独的委托 bean。

或者，您可以使用`@Resource`，它可以通过其唯一名称获取返回到当前 bean 的代理。



尝试将`@Bean` 方法的结果注入相同的配置类也实际上是一种自引用方案。要么在实际需要的方法签名中懒解析此类引用（与配置类中的自动装配字段相对），要么将受影响的`@Bean` 方法声明为静态，将其与包含的配置类实例及其生命周期脱钩。否则，仅在回退阶段考虑此类 Bean，而将其他配置类上的匹配 Bean 选作主要候选对象（如果可用）。

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

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

Java

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

    String value();
}
```

Kotlin

```
@Target(AnnotationTarget.FIELD, AnnotationTarget.VALUE_PARAMETER)
@Retention(AnnotationRetention.RUNTIME)
@Qualifier
annotation class Genre(val value: String)
```

然后，您可以在自动连接的字段和参数上提供自定义限定符，如以下示例所示：

Java

```
public class MovieRecommender {  
  
    @Autowired  
    @Genre("Action")  
    private MovieCatalog actionCatalog;  
  
    private MovieCatalog comedyCatalog;  
  
    @Autowired  
    public void setComedyCatalog(@Genre("Comedy") MovieCatalog comedyCatalog) {  
        this.comedyCatalog = comedyCatalog;  
    }  
  
    // ...  
}
```

Kotlin

```
class MovieRecommender {  
  
    @Autowired  
    @Genre("Action")  
    private lateinit var actionCatalog: MovieCatalog  
  
    private lateinit var comedyCatalog: MovieCatalog  
  
    @Autowired  
    fun setComedyCatalog(@Genre("Comedy") comedyCatalog: MovieCatalog) {  
        this.comedyCatalog = comedyCatalog  
    }  
  
    // ...  
}
```

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

```

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

```

在“[类路径扫描和托管组件\(1.10\)](#)”中，您可以看到基于注释的替代方法，以 XML 形式提供限定符元数据。具体来说，请参阅[为限定符元数据提供注解\(1.10.8\)](#)。

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

Java

```

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

```

Kotlin

```
@Target(AnnotationTarget.FIELD, AnnotationTarget.VALUE_PARAMETER)
@Retention(AnnotationRetention.RUNTIME)
@Qualifier
annotation class Offline
```

然后将注释添加到要自动装配的字段或属性，如以下示例所示：

Java

```
public class MovieRecommender {

    @Autowired
    @Offline ①
    private MovieCatalog offlineCatalog;

    // ...
}
```

① 这行加入`@Offline` 注解

Kotlin

```
class MovieRecommender {

    @Autowired
    @Offline ①
    private lateinit var offlineCatalog: MovieCatalog

    // ...
}
```

① 这行加入`@Offline` 注解

现在 bean 的定义只需要一个先定了，如下列所展示的：

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

① 这个元素指定了一个限定符

您还可以定义自定义限定符注解，该注解除了简单值属性之外或代替简单值属性，都接受命名属性。如果随后在要自动装配的字段或参数上指定了多个属性值，则 bean 定义必须与所有此类属性值匹配才能被视为自动装配候选。例如，请看以下注释定义：

Java

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

    String genre();

    Format format();
}
```

Kotlin

```
@Target(AnnotationTarget.FIELD, AnnotationTarget.VALUE_PARAMETER)
@Retention(AnnotationRetention.RUNTIME)
@Qualifier
annotation class MovieQualifier(val genre: String, val format: Format)
```

在这种情况下，**Format** 是一个枚举，定义如下：

Java

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

Kotlin

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

要自动装配的字段将用自定义限定符进行注解，并包括两个属性的值：体裁和格式，如以下示例所示：

Java

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

Kotlin

```
class MovieRecommender {  
  
    @Autowired  
    @MovieQualifier(format = Format.VHS, genre = "Action")  
    private lateinit var actionVhsCatalog: MovieCatalog  
  
    @Autowired  
    @MovieQualifier(format = Format.VHS, genre = "Comedy")  
    private lateinit var comedyVhsCatalog: MovieCatalog  
  
    @Autowired  
    @MovieQualifier(format = Format.DVD, genre = "Action")  
    private lateinit var actionDvdCatalog: MovieCatalog  
  
    @Autowired  
    @MovieQualifier(format = Format.BLURAY, genre = "Comedy")  
    private lateinit var comedyBluRayCatalog: MovieCatalog  
  
    // ...  
}
```

最后，bean 定义应包含匹配的限定符值。此示例还演示了可以使用 bean 元属性代替 `<qualifier/>` 元素。如果可用，`<qualifier/>` 元素及其属性优先，但是如果不存在这样的限定符，则自动装配机制将退回到`<meta/>` 标记内提供的值，如以下示例中的最后两个 bean 定义：

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        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. 使用泛型作为自动装配限定符

除了@Qualifier注解之外，您还可以将Java泛型类型用作限定符的隐式形式。例如，假设您具有以下配置：

Java

```
@Configuration
public class MyConfiguration {

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

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

Kotlin

```
@Configuration
class MyConfiguration {

    @Bean
    fun stringStore() = StringStore()

    @Bean
    fun integerStore() = IntegerStore()
}
```

假设前面的 bean 实现了通用接口（即 `Store <String>` 和 `Store <Integer>`），则可以`@Autowired` `Store` 接口，并且泛型被用作了一个限定符，如以下示例所示：

Java

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

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

Kotlin

```
@Autowired  
private lateinit var s1: Store<String> // <String> qualifier, injects the stringStore  
bean  
  
@Autowired  
private lateinit var s2: Store<Integer> // <Integer> qualifier, injects the  
integerStore bean
```

当自动装配列表，Map 实例和数组是，泛型限定符也可以使用，比如下面的例子自动装配了一个泛型列表 List：

Java

```
// 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;
```

Kotlin

```
// Inject all Store beans as long as they have an <Integer> generic  
// Store<String> beans will not appear in this list  
@Autowired  
private lateinit var s: List<Store<Integer>>
```

1.9.6. 使用 CustomAutoWireConfigurer

CustomAutoWireConfigurer 是 BeanFactoryPostProcessor，即使您没有使用 Spring 的 @Qualifier 注解来注解自己的自定义限定符注释类型，也可以使用它来注册。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>
```

AutowireCandidateResolver 通过以下方式确定自动依赖注入：

- 每个 bean 定义的 autowire-candidate 值
- <beans/> 元素上 default-autowire-candidates 模式
- @Qualifier 注解以及在 CustomAutoWireConfigurer 中注册的所有自定义注解的存储在

当多个 bean 符合自动装配候选条件时，“primary”的确定如下：如果候选对象中恰好有一个 bean 定义的 `primary` 属性设置为 `true`，则将其选中。

1.9.7. 使用`@Resource` 注入

Spring 还通过对字段或 bean 属性 `setter` 方法使用 JSR-250 `@Resource` 注解 (`javax.annotation.Resource`) 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 也为 Spring 管理的对象支持此模式。

`@Resource` 具有 `name` 属性。默认情况下，Spring 将该值解释为要注入的 Bean 名称。换句话说，它遵循 `name` 语义，如以下示例所示：

Java

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

① 这行插入一个`@Resource`。

Kotlin

```
class SimpleMovieLister {  
  
    @Resource(name="myMovieFinder") ①  
    private lateinit var movieFinder:MovieFinder  
}
```

① 这行插入一个`@Resource`。

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

Java

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

Kotlin

```
class SimpleMovieLister {  
  
    @Resource  
    private lateinit var movieFinder: MovieFinder  
}
```



配置注解的 name 通过 `CommonAnnotationBeanPostProcessor` 的 `ApplicationContext` 发现并解析为 Bean 名称。如果您明确配置 Spring 的 `SimpleJndiBeanFactory`, 则可以通过 JNDI 解析名称。但是, 我们建议您依赖默认行为, 并使用 Spring 的 JNDI 查找功能来保留间接级别。

在`@Resource` 用法(未指定显式名称)的特殊情况下, 类似于`@Autowired`, `@Resource` 查找主类型匹配而不是特定的命名 Bean, 并解析众所周知的可解决依赖项: `BeanFactory`, `ApplicationContext` , `ResourceLoader` , `ApplicationEventPublisher` 和 `MessageSource` 接口。

因此, 在以下示例中, `customerPreferenceDao` 字段首先查找名为“`customerPreferenceDao`”的 bean, 然后回退到类型为 `CustomerPreferenceDao` 的 primary 类型匹配项:

Java

```
public class MovieRecommender {  
  
    @Resource  
    private CustomerPreferenceDao customerPreferenceDao;  
  
    @Resource  
    private ApplicationContext context; ①  
  
    public MovieRecommender() {  
    }  
  
    // ...  
}
```

① `context` 字段是基于已知的可解决依赖类型：`ApplicationContext` 注入的。

Kotlin

```
class MovieRecommender {  
  
    @Resource  
    private lateinit var customerPreferenceDao: CustomerPreferenceDao  
  
    @Resource  
    private lateinit var context: ApplicationContext ①  
  
    // ...  
}
```

① `context` 字段是基于已知的可解决依赖类型：`ApplicationContext` 注入的。

1.9.8. 使用@Value

`@Value` 通常用于注入外部属性：

Java

```
@Component
public class MovieRecommender {

    private final String catalog;

    public MovieRecommender(@Value("${catalog.name}") String catalog) {
        this.catalog = catalog;
    }
}
```

Kotlin

```
@Component
class MovieRecommender(@Value("\${catalog.name}") private val catalog: String)
```

伴随下列配置：

Java

```
@Configuration
@PropertySource("classpath:application.properties")
public class AppConfig { }
```

Kotlin

```
@Configuration
@PropertySource("classpath:application.properties")
class AppConfig
```

还有下列 application.properties 文件：

```
catalog.name=MovieCatalog
```

Spring 提供了一个默认的宽松内嵌值解析器。 它将尝试解析属性值，如果无法解析，则将属性名称（例如`$ {catalog.name}`）作为值注入。如果要严格控制不存在的值，则应声明一个 `PropertySourcesPlaceholderConfigurer` bean，如以下示例所示：

Java

```
@Configuration  
public class AppConfig {  
  
    @Bean  
    public static PropertySourcesPlaceholderConfigurer  
propertyPlaceholderConfigurer() {  
        return new PropertySourcesPlaceholderConfigurer();  
    }  
}
```

Kotlin

```
@Configuration  
class AppConfig {  
  
    @Bean  
    fun propertyPlaceholderConfigurer() = PropertySourcesPlaceholderConfigurer()  
}
```



使用 `JavaConfig` 配置 `PropertySourcesPlaceholderConfigurer` 时，`@bean` 方法必须为静态。

如果任何`${}`占位符都没被解析出来的话，`name` 使用上述的配置会使 Spring 初始化失败。也可以使用 `setPlaceholderPrefix`，`setPlaceholderSuffix` 或 `setValueSeparator` 之类的方法来自定义占位符。



Spring boot 默认配置一个 `PropertySourcesPlaceholderConfigurer` bean 时，将会从 `application.properties` 和 `application.yml` 文件中获取属性值。

Spring 提供的内置转换器支持允许自动处理简单的类型转换（例如，转换为 `Integer` 或 `int`）。多个逗号分隔的值可以自动转换为 `String` 数组，而无需付出额外的努力。

可以提供如下默认值：

Java

```
@Component  
public class MovieRecommender {  
  
    private final String catalog;  
  
    public MovieRecommender(@Value("${catalog.name:defaultCatalog}") String catalog) {  
        this.catalog = catalog;  
    }  
}
```

Kotlin

```
@Component
class MovieRecommender(@Value("\${catalog.name:defaultCatalog}") private val catalog:
String)
```

Spring `BeanPostProcessor` 在后台使用 `ConversionService` 处理将`@Value` 中的 `String` 值转换为目标类型的过程。如果要为自己的自定义类型提供转换支持，则可以提供自己的 `ConversionService` bean 实例，如以下示例所示：

Java

```
@Configuration
public class AppConfig {

    @Bean
    public ConversionService conversionService() {
        DefaultFormattingConversionService conversionService = new
DefaultFormattingConversionService();
        conversionService.addConverter(new MyCustomConverter());
        return conversionService;
    }
}
```

Kotlin

```
@Configuration
class AppConfig {

    @Bean
    fun conversionService(): ConversionService {
        return DefaultFormattingConversionService().apply {
            addConverter(MyCustomConverter())
        }
    }
}
```

当`@Value` 包含 SpEL 表达式(第 4 章)时，该值将在运行时动态计算，如以下示例所示：

Java

```
@Component
public class MovieRecommender {

    private final String catalog;

    public MovieRecommender(@Value("#{systemProperties['user.catalog'] + 'Catalog' }") String catalog) {
        this.catalog = catalog;
    }
}
```

Kotlin

```
@Component
class MovieRecommender(
    @Value("#{systemProperties['user.catalog'] + 'Catalog' }") private val catalog: String)
```

SpEL 还支持使用更复杂的数据结构：

Java

```
@Component
public class MovieRecommender {

    private final Map<String, Integer> countOfMoviesPerCatalog;

    public MovieRecommender(
        @Value("#{{'Thriller': 100, 'Comedy': 300}}") Map<String, Integer> countOfMoviesPerCatalog) {
        this.countOfMoviesPerCatalog = countOfMoviesPerCatalog;
    }
}
```

Kotlin

```
@Component
class MovieRecommender(
    @Value("#{{'Thriller': 100, 'Comedy': 300}}") private val countOfMoviesPerCatalog: Map<String, Int>)
```

1.9.9. 使用@PostConstruct 和@PreDestroy

CommonAnnotationBeanPostProcessor 不仅可以识别@Resource 注解，还可以识别 JSR-250 生命周期注解：javax.annotation.PostConstruct 和 javax.annotation.PreDestroy。在 Spring 2.5 中引入的对这些注解的支持提供了初始

化回调和销毁回调中描述的生命周期回调机制的替代方法。假设 `CommonAnnotationBeanPostProcessor` 在 `Spring ApplicationContext` 中注册，则在生命周期中与相应的 Spring 生命周期接口方法或显式声明的回调方法在同一点调用带有这些注解之一的方法。在以下示例中，缓存在初始化时预先填充，并在销毁时清除：

Java

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

Kotlin

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

有关组合各种生命周期机制的效果的详细信息，请参见[结合生命周期机制\(69页\)](#)。



与 `@Resource` 一样，`@PostConstruct` 和 `@PreDestroy` 批注解类型是从 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，并最终在 JDK 11 中删除。如果需要，现在需要通过 Maven Central 获取 `javax.annotation-api` 工件，只需像其他任何库一样将其添加到应用程序的类路径中即可。

1.10. 类路径扫描和托管组件

本章中的大多数示例都使用 XML 来指定在 Spring 容器中生成每个 `BeanDefinition` 的配置元数据。上一节（[基于注解的容器配置](#)）演示了许多如何通过源码层面的注解提供配

置元数据。但是，即使在这些示例中，“base” bean 定义也已在 XML 文件中明确定义，而注解仅驱动依赖项注入。本节介绍了通过扫描类路径来隐式检测候选组件的选项。候选组件是与过滤条件匹配的类，并在容器中注册了相应的 Bean 定义。这消除了使用 XML 进行 bean 注册的需要。而是可以使用注解（例如，`@Component`），AspectJ 类型表达式或您自己的自定义过滤条件来选择哪些类已向容器注册了 bean 定义。



从 Spring 3.0 开始，Spring JavaConfig 项目提供的许多功能是 Spring Framework 核心的一部分。这使您可以使用 Java 而不是使用传统的 XML 文件来定义 bean。请查看[@Configuration](#), [@Bean](#), [@Import](#) 和 [@DependsOn](#) 注解，以获取有关如何使用这些新特性的示例。

1.10.1. `@Component` 以及更多的原型注释

`@Repository` 注解是任何实现数据库的角色或原型（也称为数据访问对象或 DAO）的类的标记。这些标记的使用将获自动转化异常，详情见[异常转换](#)的描述。

Spring 提供了进一步的原型注解：`@Component`, `@Service` 和 `@Controller`。
`@Component` 是任何 Spring 托管组件的通用原型。`@Repository`, `@Service` 和 `@Controller` 是`@Component` 的特定例，用于更特定的用例（分别在持久层，服务层和表示层中）。因此，您可以使用`@Component` 来注解标识组件类，但是通过使用`@Repository`, `@Service` 或 `@Controller` 注解来标识组件类，您的类更适合被工具或与相关联的切面处理。例如，这些原型注解成为切入点的理想目标。`@Repository`, `@Service` 和 `@Controller` 在 Spring 框架的将来版本中也可以带有其他语义。因此，如果在服务层使用`@Component` 或 `@Service` 之间进行选择，则`@Service` 显然是更好的选择。同样，如前所述，`@Repository` 已被支持作为持久层中自动异常转换的标记。

1.10.2. 使用元注解和合成注解

Spring 提供的许多注解都可以在您自己的代码中用作元注解。元注解是可以应用于另一个注解的注解。例如，前面提到的`@Service` 注解使用`@Component` 进行元注解，如以下示例所示：

Java

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

    // ...
}
```

① Component 导致@Service 的处理方式与@Component 相同。

Kotlin

```
@Target(AnnotationTarget.TYPE)
@Retention(AnnotationRetention.RUNTIME)
@MustBeDocumented
@Component ①
annotation class Service {

    // ...
}
```

① Component 导致@Service 的处理方式与@Component 相同。

您还可以组合元注释来创建“组合注解”。例如，Spring MVC 中的@RestController 注解由@Controller 和@ResponseBody 组成。

此外，组合注解可以选择从元注解中重新声明属性，以允许自定义。当您只希望公开元注解属性的子集时，此功能特别有用。例如，Spring 的@SessionScope 注解将作用域名称硬编码为 session，但仍允许自定义 proxyMode。以下列表显示了 SessionScope 注解的定义：

Java

```
@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;

}
```

Kotlin

```
@Target(AnnotationTarget.TYPE, AnnotationTarget.FUNCTION)
@Retention(AnnotationRetention.RUNTIME)
@MustBeDocumented
@Scope(WebApplicationContext.SCOPE_SESSION)
annotation class SessionScope(
    @get:AliasFor(annotation = Scope::class)
    val proxyMode: ScopedProxyMode = ScopedProxyMode.TARGET_CLASS
)
```

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

Java

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

Kotlin

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

您还可以覆盖 `proxyMode` 的值，如以下示例所示：

Java

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

Kotlin

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

有关更多详细信息，请参见 [Spring Annotation programming model](#) Wiki 页面。

1.10.3. 自动扫描类和注册 Bean 定义

Spring 可以自动检测原型类，并向 [ApplicationContext](#) 注册相应的 [BeanDefinition](#) 实例。例如，以下两个类别能被自动检测：

Java

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

Kotlin

```
@Service  
 class SimpleMovieLister(private val movieFinder: MovieFinder)
```

Java

```
@Repository  
 public class JpaMovieFinder implements MovieFinder {  
     // implementation elided for clarity  
 }
```

Kotlin

```
@Repository  
class JpaMovieFinder : MovieFinder {  
    // implementation elided for clarity  
}
```

要自动检测这些类并注册相应的 bean，需要将 `@ComponentScan` 添加到 `@Configuration` 类中，其中 `basePackages` 属性是这两个类的公共父包。（或者，您可以指定一个逗号分隔，分号分隔或空格分隔的列表，其中包括每个类的父包。）

Java

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

Kotlin

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



为简便起见，前面的示例可能使用了注解的 `value` 属性（即，`@ComponentScan("org.example")`）。

以下替代方法使用 XML：

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



`<context:component-scan>` 的 使用 隐 式 启 用
`<context:annotation-config>`。当使用 `<context:component-scan>` 的时候一般不需要启用 `<context:annotation-config>`。



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，请确保未激活 JAR 任务的仅文件。此外，在某些环境中，基于安全策略可能不会公开类路径目录。例如，JDK1.7.0_45 及更高版本上的独立应用程序。（这需要在清单中设置“TrustedLibrary”请参见 <https://stackoverflow.com/questions/19394570/java-jre-7u45-breaks-classloader-getresources>）。

在 JDK 9 的模块路径（Jigsaw）上，Spring 的类路径扫描通常可以按预期进行。但是，请确保将组件类导出到模块信息描述符中。如果您希望 Spring 调用类的非公共成员，请确保它们是“opened”（也就是说，它们在模块信息描述符中使用了 opens 声明而不是 export 声明）。

此外，当您使用 component-scan 元素时，将隐式包括 `AutowiredAnnotationBeanPostProcessor` 和 `CommonAnnotationBeanPostProcessor`。这意味着两个组件将被自动扫描并装配在一起，而所有这些都不需要 XML 中提供任何 bean 配置元数据。



您可以禁用注册 `AutowiredAnnotationBeanPostProcessor` 和 `CommonAnnotationBeanPostProcessor` 通过包含带有 false 值的注释配置属性。

1.10.4. 使用过滤器自定义扫描

默认情况下，仅使用 `@Component`, `@Repository`, `@Service`, `@Controller`, `@Configuration` 进行注释的类或使用 `@Component` 进行注解的自定义注解是唯一检测到的候选组件。但是，您可以通过应用自定义过滤器来修改和扩展此行为。将它们添加为 `@ComponentScan` 注解的 `includeFilters` 或 `excludeFilters` 属性（或作为 XML 配置中 `<context:component-scan>` 元素的 `<context:include-filter/>` 或 `<context:exclude-filter />` 子元素）。每个过滤器元素都需要类型和表达式属性。下表描述了过滤选项：

Table 5. Filter Types

Filter Type	Example Expression	Description
annotation (default)	<code>org.example.SomeAnnotation</code>	An annotation to be <i>present</i> or <i>meta-present</i> at the type level in target components.
assignable	<code>org.example.SomeClass</code>	A class (or interface) that the target components are assignable to (extend or implement).

Filter Type	Example Expression	Description
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 <code>org.springframework.core.type.TypeFilter</code> interface.

以下示例显示了忽略所有`@Repository`注解并改为使用“存根”库的配置：

Java

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

Kotlin

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

下方列表展示了等效的 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>
```



您也可以通过在注解上设置 `useDefaultFilters=false` 或通过将 `use-default-filters="false"` 作为 `<component-scan/>` 元素的属性来禁用默认过滤器。这有效地禁用了对使用 `@Component`, `@Repository`, `@Service`, `@Controller`, `@RestController` 或 `@Configuration` 进行注释或元注释的类的自动检测。

1.10.5. 在组件中定义 Bean 元数据

Spring 组件还可以将 bean 定义元数据贡献给容器。您可以使用与 `@Bean` 注解类相同的 `@Configuration` 注解来定义 Bean 元数据。以下示例显示了如何执行此操作：

Java

```
@Component
public class FactoryMethodComponent {

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

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

Kotlin

```
@Component
class FactoryMethodComponent {

    @Bean
    @Qualifier("public")
    fun publicInstance() = TestBean("publicInstance")

    fun doWork() {
        // Component method implementation omitted
    }
}
```

上面的类是一个，在其 `doWork()` 方法中具有特定于应用程序的代码的 Spring 组件。但是，它也提供了一个 bean 定义，该定义具有引用方法 `publicInstance()` 的工厂方法。`@Bean` 注解标识工厂方法和其他 bean 定义属性，例如通过 `@Qualifier` 注解的限定符值。可以指定的其他方法层级的注解是 `@Scope`, `@Lazy` 和自定义限定符注解。



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

如前所述，支持自动装配的字段和方法，并自动装配`@Bean`方法。以下示例显示了如何执行此操作：

Java

```
@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);
    }
}
```

Kotlin

```
@Component
class FactoryMethodComponent {

    companion object {
        private var i: Int = 0
    }

    @Bean
    @Qualifier("public")
    fun publicInstance() = TestBean("publicInstance")

    // use of a custom qualifier and autowiring of method parameters
    @Bean
    protected fun protectedInstance(
        @Qualifier("public") spouse: TestBean,
        @Value("#{privateInstance.age}") country: String) =
    TestBean("protectedInstance", 1).apply {
        this.spouse = spouse
        this.country = country
    }

    @Bean
    private fun privateInstance() = TestBean("privateInstance", i++)

    @Bean
    @RequestScope
    fun requestScopedInstance() = TestBean("requestScopedInstance", 3)
}
```

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

从 Spring Framework 4.3 开始，您还可以声明类型为 `InjectionPoint` 的工厂方法参数（或更具体的子类：`DependencyDescriptor`），以访问触发当前 bean 创建的请求注入点。注意，这仅适用于实际创建 bean 实例，而不适用于注入现有实例。因此，此功能对原型范围的 bean 最有意义。对于其他作用域，`factory` 方法仅看到在给定作用域中触发创建新 bean 实例的注入点（例如，触发创建惰性单例 bean 的依赖项）。在这种情况下，可以将提供的注入点元数据与语义一起使用。以下示例显示如何使用 `InjectionPoint`：

Java

```
@Component
public class FactoryMethodComponent {

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

Kotlin

```
@Component
class FactoryMethodComponent {

    @Bean
    @Scope("prototype")
    fun prototypeInstance(injectionPoint: InjectionPoint) =
        TestBean("prototypeInstance for ${injectionPoint.member}")
}
```

常规 Spring 组件中的 `@Bean` 方法的处理方式与 Spring `@Configuration` 类中的 `@Bean` 方法不同。区别在于，使用 CGLIB 不能增强 `@Component` 类，以拦截方法和字段的调用。CGLIB 代理是通过调用 `@Configuration` 类中的 `@Bean` 方法中的方法或字段来创建对协作对象的 Bean 元数据引用的方法。此类方法不是使用普通的 Java 语义调用的，而是通过容器进行的，以提供一般的 Spring Bean 生命周期管理和代理，即使通过 `@Bean` 方法的编程调用引用其他 Bean 时也是如此。相反，在普通 `@Component` 类内的 `@Bean` 方法中调用方法或字段具有标准 Java 语义，而无需特殊的 CGLIB 处理或其他约束。

您可以将@Bean 方法声明为静态方法，从而允许在不将其包含配置类创建为实例的情况下调用它们。在定义 post-processor Bean（例如 BeanFactoryPostProcessor 或 BeanPostProcessor 类型）时，这特别有意义，因为此类 Bean 在容器生命周期的早期进行了初始化，并且应避免在那时触发配置的其他部分。

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



@Bean 方法的 Java 语言可见性不会对 Spring 容器中的最终 bean 定义产生直接影响。您可以在非@Configuration 类中自由声明自己的工厂方法，也可以在任何地方声明静态方法。但是，@Configuration 类中的常规@Bean 方法必须是可重写的—即，不得将它们声明为 private 或 final。

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

最后，一个类可以为同一个 bean 保留多个@Bean 方法，这取决于在运行时可用的依赖关系，从而可以使用多个工厂方法。这与在其他配置方案中选择“最贪婪”的构造函数或工厂方法的算法相同：在构造时会选择具有最大可满足依赖关系的变量，类似于容器如何在多个@Autowired 构造函数之间进行选择。

1.10.6. 命名自动检测组件

在扫描过程中自动检测到组件时，其 bean 名称由该扫描程序已知的 BeanNameGenerator 策略生成。默认情况下，任何包含名称值的 Spring 构造型注释（@Component, @Repository, @Service 和 @Controller）都会将该名称提供给相应的 bean 定义。

如果这样的注解不包含名称值，或者不包含任何其他扫描到的组件（例如，由自定义过滤器发现的组件），则缺省 bean 名称生成器将返回不使用大写字母的非限定类名称。例如，如果检测到以下组件类，则名称将为 myMovieLister 和 movieFinderImpl：

Java

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

Kotlin

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

Java

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

Kotlin

```
@Repository
class MovieFinderImpl : MovieFinder {
    // ...
}
```

如果不想依赖默认的 bean 命名策略，则可以提供自定义的命名策略。首先，实现 BeanNameGenerator 接口，并确保包括默认的无参构造函数。然后，在配置扫描器时提供完全限定的类名，如以下示例注解和 Bean 定义所示。



如果由于多个自动检测到的组件具有相同的非限定类名称（即，具有相同名称但位于不同程序包中的类）而导致命名冲突，则可能需要为了生成的 bean 名称配置一个 BeanNameGenerator，其默认值为全限定标准名称。从 Spring Framework 5.2.3 开始，在 org.springframework.context.annotation 包中的 FullyQualifiedAnnotationBeanNameGenerator 可以用于此类目的。

Java

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

Kotlin

```
@Configuration  
@ComponentScan(basePackages = ["org.example"], nameGenerator = MyNameGenerator::class)  
class AppConfig {  
    // ...  
}
```

```
<beans>  
    <context:component-scan base-package="org.example"  
        name-generator="org.example.MyNameGenerator" />  
</beans>
```

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

1.10.7. 为自动扫描的组件提供一个作用域

通常，与 Spring 管理的组件一样，自动检测到的组件的默认范围也是最常见的范围是 **单例**。但是，有时您需要使用 **@Scope** 注解指定的其他作用域。您可以在注解中提供作用域的名称，如以下示例所示：

Java

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

Kotlin

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

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

有关特定于 Web 的作用域的详细信息，例如 Spring 上下文中的“request”或“session”，请参阅 [Request, Session, Application 和 WebSocket 作用域](#)。与这些作用域的预构建注解一样，您也可以使用 Spring 的元注解方法来编写自己的作用域注解：例如，使用 **@Scope ("prototype")** 进行元注解的自定义注解，也可能会声明自定义的作用域。

域代理模式。

要提供用于作用域解析的自定义策略，而不是依赖于基于注解的方法，可以实现 `ScopeMetadataResolver` 接口。确保包括默认的无参数构造函数。然后，可以在配置扫描程序时提供完全限定的类名，如以下注释和 Bean 定义示例所示：

Java

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

Kotlin

```
@Configuration  
@ComponentScan(basePackages = ["org.example"], scopeResolver = MyScopeResolver::class)  
class AppConfig {  
    // ...  
}
```

```
<beans>  
    <context:component-scan base-package="org.example" scope-  
    resolver="org.example.MyScopeResolver"/>  
</beans>
```

使用某些非单例作用域时，可能有必要为作用域对象生成代理。原因请在[作用域 Bean 作为依赖项](#)中查看。为此，在 `component-scan` 元素上可以使用 `scopedProxy` 属性。三个可能的值是：`no`, `interface` 和 `targetClass`。例如，以下配置产生标准的 JDK 动态代理：

Java

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

Kotlin

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

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

1.10.8. 使用组件提供限定符元数据

`@Qualifier` 注解在[使用 Qualifiers 基于注解的自动装配的调整中讨论过\(1.9.4\)](#)。

该部分中的示例演示了`@Qualifier` 注解和自定义限定符注解的使用，以在解析自动装配候选时提供细粒度的控制。由于这些示例基于 XML bean 定义，因此通过使用 XML 中 bean 元素的限定符或 meta 子元素，在候选 bean 定义上提供了限定符元数据。当依靠类路径扫描来自动检测组件时，可以在候选类上为限定符元数据提供类型级别的注解。下面的三个示例演示了此技术：

Java

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

Kotlin

```
@Component  
@Qualifier("Action")  
class ActionMovieCatalog : MovieCatalog
```

Java

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

Kotlin

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

Java

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

Kotlin

```
@Component  
@Offline  
class CachingMovieCatalog : MovieCatalog {  
    // ...  
}
```

与大多数基于注释的替代方法一样，请记住，注解元数据绑定到类定义本身，而 XML 的使用允许相同类型的多个 bean 提供其限定符元数据的变体，因为每个元数据由每个实例提供而不是每个类。

1.10.9. 生成候选组件的索引

尽管类路径扫描非常快，但可以通过在编译时创建候选静态列表来提高大型应用程序的启动性能。在这种模式下，作为组件扫描目标的所有模块都必须使用此机制。

现存的@ComponentScan 或<context: component-scan 指令必须保留原样，以请求上下文扫描某些软件包中的候选对象。当 ApplicationContext 检测到这样的索引时，它将自动使用它，而不是扫描类路径。

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

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context-indexer</artifactId>
        <version>5.2.8.RELEASE</version>
        <optional>true</optional>
    </dependency>
</dependencies>
```

对于 Gradle 4.5 和更早版本，依赖关系应在 `compileOnly` 配置中声明，如以下示例所示：

```
dependencies {
    compileOnly "org.springframework:spring-context-indexer:5.2.8.RELEASE"
}
```

在 Gradle 4.6 及更高版本中，依赖性应在注释处理器配置中声明，如以下示例所示：

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

该过程将生成一个包含在 jar 文件中的 `META-INF/spring.components` 文件。



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



在以下位置找到 `META-INF / spring.components` 时会自动启用索引类路径。如果某些库（或用例）的索引部分可用，但无法为整个应用程序构建，你可以通过设置 `spring.index.ignore` 为 `true` 来 fallback 一个常规类路径安排（好像根本没有索引）作为系统属性或在 `classpath` 根目录下的 `spring.properties` 文件中。

1.11. 使用 JSR 330 标准注解

从 Spring 3.0 开始，Spring 提供对 JSR-330 标准注解（依赖注入）的支持。这些注解的扫描方式与 Spring 注解的扫描方式相同。要使用它们，您需要在类路径中有相关的 jar 包。

如果使用 Maven，则标准 Maven 存储库（<https://repo1.maven.org/maven2/javax/inject/javax.inject/1/>）中提供了 `javax.inject` 工件。您可以将以下依赖项添加到文件 `pom.xml` 中：



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

1.11.1. 使用`@Inject` 和`@Named` 进行依赖注入

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

Java

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

Kotlin

```
import javax.inject.Inject

class SimpleMovieLister {

    @Inject
    lateinit var movieFinder: MovieFinder

    fun listMovies() {
        movieFinder.findMovies(...)
        // ...
    }
}
```

与`@Autowired`一样，您可以在字段层面、方法层面和构造函数参数层面使用`@Inject`。此外，您可以将注入点声明为`Provider`，从而允许按需访问范围较小的bean，或者通过`Provider.get()`调用来懒访问其他bean。下示例提供了上述示例的变体：

Java

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

Kotlin

```
import javax.inject.Inject

class SimpleMovieLister {

    @Inject
    lateinit var movieFinder: MovieFinder

    fun listMovies() {
        movieFinder.findMovies(...)
        // ...
    }
}
```

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

Java

```
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;
    }

    // ...
}
```

Kotlin

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

class SimpleMovieLister {

    private lateinit var movieFinder: MovieFinder

    @Inject
    fun setMovieFinder(@Named("main") movieFinder: MovieFinder) {
        this.movieFinder = movieFinder
    }

    // ...
}
```

与 `@Autowired` 一样，`@Inject` 也可以与 `java.util.Optional` 或 `@Nullable` 一起使用。这在这里更加适用，因为 `@Inject` 没有必需的属性。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) {
        // ...
    }
}
```

Java

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

Kotlin

```
class SimpleMovieLister {  
  
    @Inject  
    var movieFinder: MovieFinder? = null  
}
```

1.11.2. **@Named** 和**@ManagedBean**:和**@Component** 标准等效注解

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

Java

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

Kotlin

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

@Named("movieListener") // @ManagedBean("movieListener") could be used as well
class SimpleMovieLister {

    @Inject
    lateinit var movieFinder: MovieFinder

    // ...
}
```

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

Java

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

@Named
public class SimpleMovieLister {

    private MovieFinder movieFinder;

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

    // ...
}
```

Kotlin

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

@Named
class SimpleMovieLister {

    @Inject
    lateinit var movieFinder: MovieFinder

    // ...
}
```

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

Java

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

Kotlin

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



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

1.11.3. JSR-330 标准注解的局限

当使用标准注释时，您应该知道一些重要功能是不可用，如下表所示：

Table 6. Spring component model elements versus JSR-330 variants

Spring	javax.inject.*	javax.inject restrictions / comments
<code>@Autowired</code>	<code>@Inject</code>	<code>@Inject</code> has no 'required' attribute. Can be used with Java 8's <code>Optional</code> instead.
<code>@Component</code>	<code>@Named</code> / <code>@ManagedBean</code>	JSR-330 does not provide a composable model, only a way to identify named components.

Spring	javax.inject.*	javax.inject restrictions / comments
@Scope("singleton")	@Singleton	The JSR-330 default scope is like Spring's <code>prototype</code> . However, in order to keep it consistent with Spring's general defaults, a JSR-330 bean declared in the Spring container is a <code>singleton</code> by default. In order to use a scope other than <code>singleton</code> , you should use Spring's <code>@Scope</code> annotation. <code>javax.inject</code> also provides a <code>@Scope</code> annotation. Nevertheless, this one is only intended to be used for creating your own annotations.
@Qualifier	@Qualifier / @Named	<code>javax.inject.Qualifier</code> is just a meta-annotation for building custom qualifiers. Concrete <code>String</code> qualifiers (like Spring's <code>@Qualifier</code> with a value) can be associated through <code>javax.inject.Named</code> .
@Value	-	no equivalent
@Required	-	no equivalent
@Lazy	-	no equivalent
ObjectFactory	Provider	<code>javax.inject.Provider</code> is a direct alternative to Spring's <code>ObjectFactory</code> , only with a shorter <code>get()</code> method name. It can also be used in combination with Spring's <code>@Autowired</code> or with non-annotated constructors and setter methods.

1.12. 基于 Java 的容器配置

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`（1.12.1）
- 使用 `AnnotationConfigApplicationContext` 实例化 Spring 容器（1.12.2）
- 使用 `@Bean` 注解（1.12.3）
- 使用 `@Configuration` 注解（1.12.4）
- 组成基于 Java 的配置（1.12.5）
- Bean 定义配置文件（1.13.1）
- `PropertySource` 抽象（1.13.2）

- 使用@PropertySource (1.13.3)
- 声明中的占位符解析 (1.13.4)

1.12.1. 基本概念:@Bean 和@Configuration

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

@Bean 注解用于指示方法实例化，配置和初始化要由 Spring IoC 容器管理的新对象。对于那些熟悉 Spring 的<beans/> XML 配置的人来说，@ Bean 注解与<bean/>元素具有相同的作用。您可以将@Bean 注解的方法与任何 Spring @Component 一起使用。但是，它们最常与@Configuration bean 一起使用。

用@Configuration 注解表示该类的主要目的是作为 Bean 定义的来源。此外，
@Configuration 类允许通过调用同一类中的其他@Bean 方法来定义 Bean 之间的依赖关系。最简单的@Configuration 类的内容如下：

Java

```
@Configuration
public class AppConfig {

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

Kotlin

```
@Configuration
class AppConfig {

    @Bean
    fun myService(): MyService {
        return MyServiceImpl()
    }
}
```

前面的 `AppConfig` 类等同于下面的 Spring <beans/> XML：

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

完整的@Configuration 与“精简”@Bean 模式之争？

如果在未使用@Configuration 注解的类中声明@Bean 方法，则将它们称为以“精简”模式进行处理。在@Component 或普通类(plain old class)中声明的 Bean 方法被认为是“精简版”，其中包含类的主要目的不同，而@Bean 方法在那里具有某种优势。例如，服务组件可以通过每个适用组件类上的其他@Bean 方法将管理视图公开给容器。在这种情况下，@Bean 方法是一种通用的工厂方法机制。

与完整的@Configuration 不同，精简@Bean 方法无法声明 Bean 之间的依赖关系。取而代之的是，它们在其包含组件的内部状态上进行操作，并且还可以根据其可能声明的参数进行操作。因此，此类@Bean 方法不应调用其他@Bean 方法。实际上，每个这样的方法仅是用于特定 bean 的字面上的工厂方法，而没有任何特殊的运行时语义。这里的积极反作用是，不必在运行时应用 CGLIB 子类，因此在类设计方面没有任何限制(即，包含类可能是 final 类，依此类推)。

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

以下各节将详细讨论@Bean 和@Configuration 注解。但是，首先，我们介绍了使用基于 Java 的配置来创建 spring 容器的各种方法。

1.12.2. 通过使用 AnnotationConfigApplicationContext 实例化 Spring 容器

以下各节介绍了 Spring 3.0 中引入的 Spring 的 AnnotationConfigApplicationContext。这种通用的 ApplicationContext 实现不仅可以接受@Configuration 类作为输入，还可以接受普通的@Component 类以及带有 JSR-330 元数据注解的类。

当提供@Configuration 类作为输入时，@Configuration 类本身将注册为 Bean 定义，并且该类中所有已声明的@Bean 方法也将注册为 Bean 定义。

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

● 简单结构

与实例化 `ClassPathXmlApplicationContext` 时将 Spring XML 文件用作输入的方式几乎相同，实例化 `AnnotationConfigApplicationContext` 时可以将`@Configuration`类用作输入。如下面的示例所示，这允许完全不使用 XML 来使用 Spring 容器：

Java

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

Kotlin

```
import org.springframework.beans.factory.getBean

fun main() {
    val ctx = AnnotationConfigApplicationContext(AppConfig::class.java)
    val myService = ctx.getBean<MyService>()
    myService.doStuff()
}
```

如前所述，`AnnotationConfigApplicationContext` 不限于仅与`@Configuration`类一起使用。可以将任何`@Component` 或 JSR-330 带注解的类作为输入提供给构造函数，如以下示例所示：

Java

```
public static void main(String[] args) {
    ApplicationContext ctx = new
    AnnotationConfigApplicationContext(MyServiceImpl.class, Dependency1.class,
    Dependency2.class);
    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
}
```

Kotlin

```
import org.springframework.beans.factory.getBean

fun main() {
    val ctx = AnnotationConfigApplicationContext(MyServiceImpl::class.java,
    Dependency1::class.java, Dependency2::class.java)
    val myService = ctx.getBean<MyService>()
    myService.doStuff()
}
```

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

- 通过使用 `register(Class<?>...)` 以编程方式构建容器

您可以使用无参构造函数实例化 `AnnotationConfigApplicationContext`，然后使用 `register()` 方法对 其 进 行 配 置 。 以 编 程 方 式 构 建 `AnnotationConfigApplicationContext` 时，此方法特别有用。以下示例显示了怎么做：

Java

```
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();  
}
```

Kotlin

```
import org.springframework.beans.factory.getBean  
  
fun main() {  
    val ctx = AnnotationConfigApplicationContext()  
    ctx.register(AppConfig::class.java, OtherConfig::class.java)  
    ctx.register(AdditionalConfig::class.java)  
    ctx.refresh()  
    val myService = ctx.getBean<MyService>()  
    myService.doStuff()  
}
```

- 使用 `scan(String)` 启用扫描组件扫描

要启用组件扫描，可以按如下方式对`@Configuration` 类进行注解：

Java

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

① 此注解启用组件扫描。

Kotlin

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

- ① 此注解启用组件扫描。

经验丰富的 Spring 用户可能熟悉 Spring 的 context：名称空间中的 XML 声明，如以下示例所示：



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

在前面的示例中，对 `com.acme` 包进行了扫描以查找任何 `@Component` 注解的类，并将这些类注册为容器内的 Spring bean 定义。`AnnotationConfigApplicationContext` 公开了 `scan (String...)` 方法以允许相同的组件扫描功能，如以下示例所示：

Java

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

Kotlin

```
fun main() {  
    val ctx = AnnotationConfigApplicationContext()  
    ctx.scan("com.acme")  
    ctx.refresh()  
    val myService = ctx.getBean<MyService>()  
}
```



请记住，`@Configuration` 类使用 `@Component` 进行元注解，因此它们是组件扫描的候选对象。在前面的示例中，假定 `AppConfig` 在 `com.acme` 包（或下面的任何包）中声明，则在调用 `scan()` 时将其抓取。根据 `refresh()`，其所有 `@Bean` 方法都将被处理并注册为容器内的 Bean 定义。

● 使用 `AnnotationConfigWebApplicationContext` 支持 Web 应用

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

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

`@Bean` 是方法层面的注解，很像 XML `<bean/>` 标签。注解支持`<bean/>`提供的某些属性，例如`*init-method(1.6.1)*destroy-method(1.6.1)*autowiring(1.4.5) *name`。

您可以在`@Configuration` 注解或`@Component` 注解的类中使用`@Bean` 注解。

● 声明一个 Bean

要声明一个 bean，可以用`@Bean` 注解来标识一个方法。您可以使用此方法在类型指定为该方法的返回值的`ApplicationContext` 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：

Java

```
@Configuration
public class AppConfig {

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

Kotlin

```
@Configuration
class AppConfig {

    @Bean
    fun transferService() = TransferServiceImpl()
}
```

前面的配置与下面的 Spring XML 完全等效：

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

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

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

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

Java

```
@Configuration
public class AppConfig {

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

Kotlin

```
@Configuration
class AppConfig {

    @Bean
    fun transferService(): TransferService {
        return TransferServiceImpl()
    }
}
```

但是，这将高级类型预测的可见性限制为指定的接口类型（`TransferService`）。然后，使用仅一次知道容器的完整类型（`TransferServiceImpl`），实例化受影响的单例 bean。非懒加载单例 bean 根据其声明顺序实例化，因此您可能会看到不同的类型匹配结果，具体取决于另一个组件何时尝试通过未声明的类型进行匹配（例如 `@Autowired` `TransferServiceImpl`，仅当 `transferService` bean 具有被实例化）。



如果您通过声明的服务接口一致地引用类型，则@Bean 返回类型可以安全地加入该设计决策。但是，对于实现多个接口的组件或对于可能由其实现类型引用的组件，声明尽可能最具体的返回类型（至少与引用您的 bean 的注入点所要求的具体类型一样）是比较安全的。

● Bean 依赖

@Bean 注解的方法可以具有任意数量的参数，这些参数描述构建该 bean 所需的依赖关系。例如，如果我们的 TransferService 需要一个 AccountRepository，则可以使用方法参数来实现该依赖关系，如以下示例所示：

Java

```
@Configuration
public class AppConfig {

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

Kotlin

```
@Configuration
class AppConfig {

    @Bean
    fun transferService(accountRepository: AccountRepository): TransferService {
        return TransferServiceImpl(accountRepository)
    }
}
```

解析机制与基于构造函数的依赖注入几乎相同。有关更多详细信息，请参见[相关部分 \(1.4.1\)](#)。

● 接收生命周期回调

任何使用@Bean 注解定义的类都支持常规的生命周期回调，并且可以使用 JSR-250 中的@PostConstruct 和@PreDestroy 注解。详情请见 [JSR-250 注解 \(1.9.9\)](#)。

还完全支持常规的 Spring 生命周期 (1.6.1) 回调。如果 bean 实现了 InitializingBean, DisposableBean 或 Lifecycle，则容器将调用它们各自的方法。

还完全支持标准的*Aware 接口集（例如 BeanFactoryAware (1.16), BeanNameAware, MessageSourceAware (1.6.2), ApplicationContextAware (1.6.2) 等）。

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

Java

```
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();  
    }  
}
```

Kotlin

```
class BeanOne {

    fun init() {
        // initialization logic
    }
}

class BeanTwo {

    fun cleanup() {
        // destruction logic
    }
}

@Configuration
class AppConfig {

    @Bean(initMethod = "init")
    fun beanOne() = BeanOne()

    @Bean(destroyMethod = "cleanup")
    fun beanTwo() = BeanTwo()
}
```

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

默认情况下，您可能要对通过 JNDI 获取的资源执行此操作，因为其生命周期是在应用程序外部进行管理的。特别是，请确保始终对数据源执行此操作，因为在 Java EE 应用程序服务器上已知这是有问题的。下面的示例显示如何防止对数据源的自动销毁回调：

Java

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



Kotlin

```
@Bean(destroyMethod = "")  
fun dataSource(): DataSource {  
    return jndiTemplate.lookup("MyDS") as DataSource  
}
```

同样，通过`@Bean` 方法，通常使用程序化 JNDI 查找，方法是使用 Spring 的 `JndiTemplate` 或 `JndiLocatorDelegate` 帮助器，或者直接使用 JNDI `InitialContext` 用法，而不是 `JndiObjectFactoryBean` 变体（这将迫使您将返回类型声明为 `FactoryBean` 类型，而不是实际的目标类类型，因此很难在打算引用此处提供的资源的其他`@Bean` 方法中用于交叉引用调用）。

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

Java

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

Kotlin

```
@Configuration  
class AppConfig {  
  
    @Bean  
    fun beanOne() = BeanOne().apply {  
        init()  
    }  
  
    // ...  
}
```



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

● 指定 Bean 作用域

Spring 包含 `@Scope` 注解，以便您可以指定 bean 的作用域。

- 使用 `@Scope` 注解

您可以指定使用 `@Bean` 注解定义的 bean 应该具有特定作用域。您可以使用 [Bean Scopes \(1.5\)](#) 部分中指定的任何标准作用域。

默认范围是单例，但是您可以使用 `@Scope` 注释覆盖它，如以下示例所示：

Java

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

Kotlin

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

@Scope 和 scoped-proxy

Spring 提供了一种通过[作用域代理（58 页）](#)使用作用域依赖性的便捷方法。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 注解在 Java 中配置 bean，可以通过 proxyMode 属性提供同等的支持。默认值为无代理（`ScopedProxyMode.NO`），但您可以设置为 `ScopedProxyMode.TARGET_CLASS` 或 `ScopedProxyMode.INTERFACES`。

如果使用 Java 从 XML 参考文档（请参阅[作用域代理（58 页）](#)）将作用域代理示例移植到我们的@Bean，则它类似于以下内容：

Java

```
// 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;
}
```

Kotlin

```
// an HTTP Session-scoped bean exposed as a proxy
@Bean
@SessionScope
fun userPreferences() = UserPreferences()

@Bean
fun userService(): Service {
    return SimpleUserService().apply {
        // a reference to the proxied userPreferences bean
        setUserPreferences(userPreferences())
    }
}
```

● 自定义 Bean 名称

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：

Java

```
@Configuration
public class AppConfig {

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

Kotlin

```
@Configuration  
class AppConfig {  
  
    @Bean("myThing")  
    fun thing() = Thing()  
}
```

● Bean 别名

如在[命名 Bean \(上一节\)](#) 中讨论的那样，有时希望为单个 Bean 提供多个名称，否则称为 Bean 别名。为此，`@Bean` 注解的 `name` 属性接受一个 `String` 数组。以下示例说明如何为 bean 设置多个别名：

Java

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

Kotlin

```
@Configuration  
class AppConfig {  
  
    @Bean("dataSource", "subsystemA-dataSource", "subsystemB-dataSource")  
    fun dataSource(): DataSource {  
        // instantiate, configure and return DataSource bean...  
    }  
}
```

● Bean 描述

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

要向`@Bean` 添加描述，可以使用`@Description` 注释，如下所示：

Java

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

Kotlin

```
@Configuration  
class AppConfig {  
  
    @Bean  
    @Description("Provides a basic example of a bean")  
    fun thing() = Thing()  
}
```

1.12.4. 使用@Configuration 注解

`@Configuration` 是类级别的注解，指示对象是 Bean 定义的源。`@Configuration` 类通过公共`@Bean` 注解方法声明 bean。对`@Configuration` 类的`@Bean` 方法的调用也可以用于定义 Bean 之间的依赖关系。有关一般介绍，请参见 [基本概念：@Bean 和 @Configuration\(1.12.1\)](#)。

● 注入 bean 间的依赖关系

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

Java

```
@Configuration  
public class AppConfig {  
  
    @Bean  
    public BeanOne beanOne() {  
        return new BeanOne(beanTwo());  
    }  
  
    @Bean  
    public BeanTwo beanTwo() {  
        return new BeanTwo();  
    }  
}
```

Kotlin

```
@Configuration  
class AppConfig {  
  
    @Bean  
    fun beanOne() = BeanOne(beanTwo())  
  
    @Bean  
    fun beanTwo() = BeanTwo()  
}
```

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



仅当在 `@Configuration` 类中声明 `@Bean` 方法时，此声明 bean 间依赖关系的方法才有效。您不能使用简单的 `@Component` 类声明 Bean 间的依赖关系。

● 查找方法注入

如前所述，[查找方法注入\(1.4.6\)](#)是一项高级功能，您应该很少使用。在单例作用域的 bean 依赖于原型作用域的 bean 的情况下，这很有用。将 Java 用于这种类型的配置为实现此模式提供了原生的方法。以下示例显示如何使用查找方法注入：

Java

```
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();  
}
```

Kotlin

```
abstract class CommandManager {  
    fun process(commandState: Any): Any {  
        // grab a new instance of the appropriate Command interface  
        val 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 fun createCommand(): Command  
}
```

通过使用 Java 配置，可以创建 `CommandManager` 的子类，在该子类中，抽象的 `createCommand()` 方法将被覆盖，以便它查找新的（原型）命令对象。以下示例显示了如何执行此操作：

Java

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

Kotlin

```
@Bean  
@Scope("prototype")  
fun asyncCommand(): AsyncCommand {  
    val command = AsyncCommand()  
    // inject dependencies here as required  
    return command  
}  
  
@Bean  
fun commandManager(): CommandManager {  
    // return new anonymous implementation of CommandManager with createCommand()  
    // overridden to return a new prototype Command object  
    return object : CommandManager() {  
        override fun createCommand(): Command {  
            return asyncCommand()  
        }  
    }  
}
```

● 关于基于 Java 配置在内部如何工作的更多信息

考虑以下示例，该示例显示了一个@Bean 注解方法被调用两次：

Java

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

Kotlin

```
@Configuration  
class AppConfig {  
  
    @Bean  
    fun clientService1(): ClientService {  
        return ClientServiceImpl().apply {  
            clientDao = clientDao()  
        }  
    }  
  
    @Bean  
    fun clientService2(): ClientService {  
        return ClientServiceImpl().apply {  
            clientDao = clientDao()  
        }  
    }  
  
    @Bean  
    fun clientDao(): ClientDao {  
        return ClientDaoImpl()  
    }  
}
```

`clientDao()` 在 `clientService1()` 中被调用一次，并在 `clientService2()` 中被调用一次。由于此方法会创建一个 `ClientDaoImpl` 的新实例并返回它，因此通常希望有两个实例（每个服务一个）。那肯定是有问题的：在 Spring 中，实例化的 bean 默认情况下具有单例作用域。这就是神奇的地方：所有 `@Configuration` 类在启动时都使用 `CGLIB` 进行了子类化。在子类中，子方法在调用父方法并创建新实例之前，首先检查容器中是否有任何缓存（作用域）的 bean。



根据 bean 的作用范围，行为可能会有所不同。 我们在这里讨论单例模式。



从 Spring 3.2 开始，不再需要将 `CGLIB` 添加到您的类路径中，因为 `CGLIB` 类已经在 `org.springframework.cglib` 下重新打包并直接包含在 `spring-core` JAR 中。

由于 CGLIB 在启动时会动态添加功能，因此存在一些限制。特别是，配置类不能是 final。但是，从 4.3 版本开始，配置类中允许使用任何构造函数，包括使用 @Autowired 或单个非默认构造函数声明进行默认注入。



如果您希望避免任何 CGLIB 施加的限制，请考虑在非@Configuration 类（例如，在普通的@Component 类上）声明@Bean 方法。然后，不会截获 @Bean 方法之间的跨方法调用，因此您必须专门依赖那里的构造函数或方法级别的依赖项注入。

1.12.5. 构建基于 Java 的配置

Spring 的基于 Java 的配置功能使您可以撰写注解，从而降低配置的复杂性。

● 使用@Import 注解

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

Java

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

Kotlin

```
@Configuration  
class ConfigA {  
  
    @Bean  
    fun a() = A()  
}  
  
@Configuration  
@Import(ConfigA::class)  
class ConfigB {  
  
    @Bean  
    fun b() = B()  
}
```

现在，无需在实例化上下文时同时指定 `ConfigA.class` 和 `ConfigB.class`，只需显式提供 `ConfigB`，如以下示例所示：

Java

```
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);  
}
```

Kotlin

```
import org.springframework.beans.factory.getBean  
  
fun main() {  
    val ctx = AnnotationConfigApplicationContext(ConfigB::class.java)  
  
    // now both beans A and B will be available...  
    val a = ctx.getBean<A>()  
    val b = ctx.getBean<B>()  
}
```

这种方法简化了容器的实例化，因为只需要处理一个类，而不是要求您在构造过程中记住潜在的大量`@Configuration` 类。



从 Spring Framework 4.2 开始，`@Import` 还支持对常规组件类的引用，类似于 `AnnotationConfigApplicationContext.register` 方法。如果要通过使用一些配置类作为入口点来显式定义所有组件，从而避免组件扫描，则此功能特别有用。

在导入的`@Bean` 定义上注入依赖项

前面的示例有效，但过于简单。在大多数实际情况下，Bean 在配置类之间相互依赖。使用 XML 时，这不是问题，因为不涉及任何编译器，并且您可以声明 `ref="someBean"` 并信任 Spring 在容器初始化期间进行处理。使用`@Configuration` 类时，Java 编译器会在配置模型上施加约束，因为对其他 bean 的引用必须是有效的 Java 语法。

幸运的是解决问题非常简单。正如[我们已经讨论的](#)，`@Bean` 方法可以具有任意数量的参数来描述 Bean 的依赖关系。考虑以下具有多个`@Configuration` 类的更真实的场景，每个类取决于在其他类中声明的 bean：

Java

```
@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");
}
```

Kotlin

```
import org.springframework.beans.factory.getBean

@Configuration
class ServiceConfig {

    @Bean
    fun transferService(accountRepository: AccountRepository): TransferService {
        return TransferServiceImpl(accountRepository)
    }
}

@Configuration
class RepositoryConfig {

    @Bean
    fun accountRepository(dataSource: DataSource): AccountRepository {
        return JdbcAccountRepository(dataSource)
    }
}

@Configuration
@Import(ServiceConfig::class, RepositoryConfig::class)
class SystemTestConfig {

    @Bean
    fun dataSource(): DataSource {
        // return new DataSource
    }
}

fun main() {
    val ctx = AnnotationConfigApplicationContext(SystemTestConfig::class.java)
    // everything wires up across configuration classes...
    val transferService = ctx.getBean<TransferService>()
    transferService.transfer(100.00, "A123", "C456")
}
```

还有另一种方法可以达到相同的结果。请记住，`@Configuration` 类最终仅是容器中的另一个 bean：这意味着它们可以利用`@Autowired` 和`@Value` 注入以及与任何其他 bean 相同的其他功能。

确保以这种方式注入的依赖项只是最简单的一种。`@Configuration` 类在上下文的初始化过程中很早就被处理，并且强制以这种方式注入依赖项可能导致意外的过早初始化。如上例所示，请尽可能使用基于参数的注入。



另外，通过 `@Bean` 使用 `BeanPostProcessor` 和 `BeanFactoryPostProcessor` 定义时要特别小心。通常应将这些声明为静态 `@Bean` 方法，而不触发其包含的配置类的实例化。否则，`@Autowired` 和 `@Value` 可能不适用于配置类本身，因为可以将其创建为比 `AutowiredAnnotationBeanPostProcessor` 早的 bean 实例。

下面的例子展示了一个 bean 如何被装配为另一个 bean：

Java

```
@Configuration
public class ServiceConfig {

    @Autowired
    private AccountRepository accountRepository;

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

@Configuration
public class RepositoryConfig {

    private final DataSource dataSource;

    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");
}
```

Kotlin

```
import org.springframework.beans.factory.getBean

@Configuration
class ServiceConfig {

    @Autowired
    lateinit var accountRepository: AccountRepository

    @Bean
    fun transferService(): TransferService {
        return TransferServiceImpl(accountRepository)
    }
}

@Configuration
class RepositoryConfig(private val dataSource: DataSource) {

    @Bean
    fun accountRepository(): AccountRepository {
        return JdbcAccountRepository(dataSource)
    }
}

@Configuration
@Import(ServiceConfig::class, RepositoryConfig::class)
class SystemTestConfig {

    @Bean
    fun dataSource(): DataSource {
        // return new DataSource
    }
}

fun main() {
    val ctx = AnnotationConfigApplicationContext(SystemTestConfig::class.java)
    // everything wires up across configuration classes...
    val transferService = ctx.getBean<TransferService>()
    transferService.transfer(100.00, "A123", "C456")
}
```



`@Configuration` 类的构造器注入方法只在 Spring 4.3 之后被支持，

同样需要注意的是如果目标 bean 定义只有一个构造器的时候不需要特指
`@Autowired`。

对导航场景的全限定 bean 导入

在前面的场景中，使用`@Autowired` 可以很好地工作并提供所需的模块化，但是确定一定以及肯定在何处声明自动装配的 Bean 定义仍然有些模棱两可。例如，当开发人员查看 `ServiceConfig` 时，您如何确切知道`@Autowired AccountRepository` bean 的声明位置？它在代码中并不是显式的，这可能看上去很美好。请记住，[Spring Tools for Eclipse](#)

提供了可以渲染图形的工具，这些图形显示了所有事物的装配方式，这也许就是您所需要的。另外，您的 Java IDE 可以轻松找到 `AccountRepository` 类型的所有声明和使用，并快速向您显示返回该类型的`@Bean` 方法的位置。

如果这种歧义是不可接受的，并且您希望从 IDE 内部直接从一个`@Configuration` 类导航到另一个`@Configuration` 类，请考虑自动装配配置类本身。以下示例显示了如何执行此操作：

Java

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

Kotlin

```
@Configuration
class ServiceConfig {

    @Autowired
    private lateinit var repositoryConfig: RepositoryConfig

    @Bean
    fun transferService(): TransferService {
        // navigate 'through' the config class to the @Bean method!
        return TransferServiceImpl(repositoryConfig.accountRepository())
    }
}
```

在上述情况下，定义 `AccountRepository` 的位置是完全明确的。但是，`ServiceConfig` 现在与 `RepositoryConfig` 紧密耦合，那就很坑了。所以通过使用基于接口或基于抽象类的`@Configuration` 类可以稍微缓解这种紧密耦合。请看以下示例：

Java

```
@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");
}
```

Kotlin

```
import org.springframework.beans.factory.getBean

@Configuration
class ServiceConfig {

    @Autowired
    private lateinit var repositoryConfig: RepositoryConfig

    @Bean
    fun transferService(): TransferService {
        return TransferServiceImpl(repositoryConfig.accountRepository())
    }
}

@Configuration
interface RepositoryConfig {

    @Bean
    fun accountRepository(): AccountRepository
}

@Configuration
class DefaultRepositoryConfig : RepositoryConfig {

    @Bean
    fun accountRepository(): AccountRepository {
        return JdbcAccountRepository(...)
    }
}

@Configuration
@Import(ServiceConfig::class, DefaultRepositoryConfig::class) // import the concrete config!
class SystemTestConfig {

    @Bean
    fun dataSource(): DataSource {
        // return DataSource
    }
}

fun main() {
    val ctx = AnnotationConfigApplicationContext(SystemTestConfig::class.java)
    val transferService = ctx.getBean<TransferService>()
    transferService.transfer(100.00, "A123", "C456")
}
```

现在，`ServiceConfig` 与具体的 `DefaultRepositoryConfig` 是松耦合，并且内置的 IDE 工具仍然有用：您可以轻松地获得 `RepositoryConfig` 实现的类型层次结构。以这种方式，指向`@Configuration` 类及其依赖项与指向基于接口的代码的通常过程没有什么不同。



如果要影响某些 bean 的启动创建顺序，请考虑将其中一些声明为 `@Lazy`（用于首次访问而不是在启动时创建）或声明为`@DependsOn` 某些其他 bean（确保其他特定 bean 在当前 bean 创建之前，意味着它们是直接依赖的）。

● 有条件地包括`@Configuration` 类或 Bean 方法

根据某些系统状态，有条件地启用或禁用完整的`@Configuration` 类甚至单个`@Bean` 方法通常很有用。一个常见的示例是仅在 Spring 环境中启用了特定的配置文件时才使用 `@Profile` 注解来激活 Bean（有关详细信息，请参见 [Bean 定义配置文\(1.13.1\)](#)）。

`@Profile` 注解实际上通过使用更灵活的调用`@Conditional` 注解来实现的。
`@Conditional` 注解在 `@Bean` 注册之前应引用的特定 `org.springframework.context.annotation.Condition` 实现。

`Condition` 接口的实现提供了一个 `matches(...)` 方法，该方法返回 `true` 或 `false`。例如，以下列表显示了用于`@Profile` 的实际 `Condition` 实现：

Java

```
@Override
public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
    // 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;
}
```

Kotlin

```
override fun matches(context: ConditionContext, metadata: AnnotatedTypeMetadata): Boolean {
    // Read the @Profile annotation attributes
    val attrs = metadata.getAllAnnotationAttributes(Profile::class.java.name)
    if (attrs != null) {
        for (value in attrs["value"]!!) {
            if (context.environment.acceptsProfiles(Profiles.of(*value as
Array<String>)))) {
                return true
            }
        }
        return false
    }
    return true
}
```

有关更多详细信息，请参见[@Conditional](#) 的 javadoc。

● 结合 Java 和 XML 配置

Spring 的[@Configuration](#) 类支持并非旨在完全替代 Spring XML。某些工具（例如 Spring XML 名称空间）仍然是配置容器的理想方法。在使用 XML 方便或有必要的情况下，你可以选择：使用例如以“XML 为中心”的方式凭借“[ClassPathXmlApplicationContext](#)”实例化容器，或以“Java 为中心”的方式使用 [AnnotationConfigApplicationContext](#) 实例化容器。[@ImportResource](#) 注解可根据需要导入 XML。

以 XML 中心使用[@Configuration](#) 类

或许更偏好从 XML 引导 Spring 容器，并包括用特定方式创建的[@Configuration](#) 类。例如，在使用 Spring XML 的大型现有代码库中，根据需要创建[@Configuration](#) 类并从现有 XML 文件中将它们包含在内会变得更加容易。在稍后章节，我们将介绍在这种“以 XML 为中心”的情况下使用[@Configuration](#) 类的情况。

将[@Configuration](#) 类声明为纯 Spring [`<bean/>`](#)元素

请记住，[@Configuration](#) 类最终是容器中的 bean 定义。在本系列示例中，我们创建一个名为 [AppConfig](#) 的 [@Configuration](#) 类，并将其使用 [`<bean/>`](#) 定义包含在 [systemtest-config.xml](#) 中。因为[`<context:annotation-config/>`](#)已打开，所以容器可以识别[@Configuration](#) 注解并正确处理 [AppConfig](#) 中声明的[@Bean](#) 方法。

以下示例显示了 Java 中的普通配置类：

Java

```
@Configuration
public class AppConfig {

    @Autowired
    private DataSource dataSource;

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

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

Kotlin

```
@Configuration
class AppConfig {

    @Autowired
    private lateinit var dataSource: DataSource

    @Bean
    fun accountRepository(): AccountRepository {
        return JdbcAccountRepository(dataSource)
    }

    @Bean
    fun transferService() = TransferService(accountRepository())
}
```

以下示例显示了 `system-test-config.xml` 文件的一部分：

```

<beans>
    <!-- enable processing of annotations such as @Autowired and @Configuration -->
    <context:annotation-config/>
    <context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>

    <bean class="com.acme.AppConfig"/>

    <bean class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>
</beans>

```

以下示例显示了可能的 `jdbc.properties` 文件：

```

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

```

Java

```

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

```

Kotlin

```

fun main() {
    val ctx = ClassPathXmlApplicationContext("classpath:/com/acme/system-test-
config.xml")
    val transferService = ctx.getBean<TransferService>()
    // ...
}

```

在 `system-test-config.xml` 文件中，`AppConfig <bean >/>` 没有声明 `id` 元素。尽管这样做是可以接受的，但是由于没有其他 `bean` 曾经引用过它，因此这是不必要的，并且不太可能通过名称从容器中显式获取。同样，`DataSource` `bean` 只能按类型自动装配，因此也不严格要求显式 `bean id`。

使用 `<context: component-scan/>` 抓取 `@Configuration` 类

由于 `@Configuration` 使用 `@Component` 进行元注解，因此 `@Configuration` 注解的类自动成为组件扫描的候选对象。使用与先前示例中描述的场景相同的场景，我们可以重新定义 `system-test-config.xml` 以利用组件扫描的优势。请注意，在这种情况下，我们无需

显式声明`<context:annotation-config/>`，因为`<context:component-scan/>`可启用相同的功能。

以下示例显示了修改后的`system-test-config.xml`文件：

```
<beans>
    <!-- picks up and registers AppConfig as a bean definition -->
    <context:component-scan base-package="com.acme"/>
    <context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>

    <bean class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>
</beans>
```

@Configuration 以类为中心的 XML 与@ImportResource 的使用

在`@Configuration`类是配置容器的主要机制的应用程序中，仍然有必要多少用点 XML。在这些情况下，您可以使用`@ImportResource`并仅定义所需的 XML。这样做实现了“以 Java 为中心”的方法来配置容器，并使 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`注解来实现按需使用 XML 的以 Java 为中心的配置：

Java

```
@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);
    }
}
```

Kotlin

```
@Configuration
@ImportResource("classpath:/com/acme/properties-config.xml")
class AppConfig {

    @Value("\${jdbc.url}")
    private lateinit var url: String

    @Value("\${jdbc.username}")
    private lateinit var username: String

    @Value("\${jdbc.password}")
    private lateinit var password: String

    @Bean
    fun dataSource(): DataSource {
        return DriverManagerDataSource(url, username, password)
    }
}
```

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

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

Java

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

Kotlin

```
import org.springframework.beans.factory.getBean

fun main() {
    val ctx = AnnotationConfigApplicationContext(AppConfig::class.java)
    val transferService = ctx.getBean<TransferService>()
    // ...
}
```

1.13. 环境抽象概念

Environment 接口是集成在容器中的抽象概念，可对应用程序环境的两个关键方面进行建模：[概要文件\(1.13.1\)](#)和[属性\(1.13.2\)](#)。

概要文件是仅在给定概要文件处于活动状态时才向容器注册的 Bean 定义的命名逻辑组。可以将 Bean 分配给概要文件，无论是以 XML 定义还是带有注解。与配置文件相关的环境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，以及哪些配置文件（如果有）在默认情况下应处于活动状态。

属性在几乎所有应用程序中都起着重要作用，并且可能源自各种来源：属性文件，JVM 系统属性，系统环境变量，JNDI，Servlet 上下文参数，特殊属性对象，Map 对象等。环境对象与属性相关的作用是为用户提供方便的服务界面，用于配置属性源并从中解析属性。

1.13.1. Bean 定义文件

Bean 定义配置文件在核心容器中提供了一种机制，该机制允许在不同环境中注册不同的 Bean。”environment”一词对不同的用户可能具有不同的含义，并且此功能可以帮助解决许多用例，包括：

- 在开发中针对内存中的数据源进行工作，而不是在进行 QA 或生产时从 JNDI 查找相同的数据源。
- 仅在将应用程序部署到性能环境中时注册监视基础架构。
- 为消费者 A 针对消费者 B 部署注册 bean 的自定义实现。

在需要数据源的实际应用中考虑第一个用例。在测试环境中，配置可能类似于以下内容：

Java

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

Kotlin

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

现在，假设该应用程序的数据源已在生产应用程序服务器的 JNDI 目录中注册，请考虑如何将该应用程序部署到 QA 或生产环境中。现在，我们的 `dataSource` bean 看起来像下面的列表：

Java

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

Kotlin

```
@Bean(destroyMethod = "")  
fun dataSource(): DataSource {  
    val ctx = InitialContext()  
    return ctx.lookup("java:comp/env/jdbc/datasource") as DataSource  
}
```

问题是如何根据当前环境在使用这两种变体之间进行切换。随着时间的流逝，Spring 用户已经设计出多种方法来完成此任务，通常依赖于系统环境变量和包含`$ {placeholder}` 占位符的 XML `<import />`语句的组合，这个 XML 根据环境变量值解析为正确的配置文件路径。Bean 定义配置文件是一项核心容器功能，可提供此问题的解决方案。

如果我们概括前面特定于环境的 Bean 定义示例中所示的用例，那么最终需要在某些上下文中而不是在其他上下文中注册某些 Bean 定义。你可能会说你要在情况 A 中注册一个特

定的 bean 定义配置文件，在情况 B 中注册一个不同的配置文件。我们首先更新配置以反映这种需求。

● 使用 @profile 注解

`@Profile` 注解可让你在一个或多个指定的配置文件处于活动状态时指定有资格注册的组件。使用前面的示例，我们可以如下重写 `dataSource` 配置：

Java

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

Kotlin

```
@Configuration  
@Profile("development")  
class StandaloneDataConfig {  
  
    @Bean  
    fun dataSource(): DataSource {  
        return EmbeddedDatabaseBuilder()  
            .setType(EmbeddedDatabaseType.HSQL)  
            .addScript("classpath:com/bank/config/sql/schema.sql")  
            .addScript("classpath:com/bank/config/sql/test-data.sql")  
            .build()  
    }  
}
```

Java

```
@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");  
     }  
 }
```

Kotlin

```
@Configuration  
 @Profile("production")  
 class JndiDataConfig {  
  
     @Bean(destroyMethod = "")  
     fun dataSource(): DataSource {  
         val ctx = InitialContext()  
         return ctx.lookup("java:comp/env/jdbc/datasource") as DataSource  
     }  
 }
```

如前所述，对于 `@Bean` 方法，通常选择使用程序化 JNDI 查找，方法是使用 Spring 的 `JndiTemplate / JndiLocatorDelegate` 帮助器或前面显示的直接 JNDI `InitialContext` 用法，而不是 `JndiObjectFactoryBean` 变体，这将强制你将返回类型声明为 `FactoryBean` 类型。

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

- `!`：配置文件的逻辑“非”
- `&`：配置文件的逻辑“与”
- `|`：配置文件的逻辑“或”

您不能不使用括号的运算符的情况下将 `&` 和 `|` 混合使用。 例如，

 `production&us-east| eu-central` 不是有效的表达式。 它必须表示为 `production&(us-east|eu-central)`。

您可以将 `@Profile` 用作元注解，以创建自定义的组合注解。以下示例定义了一个自定义 `@Production` 注解，您可以将其用作 `@Profile("production")` 的替代品：

Java

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

Kotlin

```
@Target(AnnotationTarget.TYPE)
@Retention(AnnotationRetention.RUNTIME)
@Profile("production")
annotation class Production
```



如果 `@Configuration` 类用 `@Profile` 标记，则除非该类中的一个或多个指定的配置文件处于活动状态，否则所有与该类关联的 `@Bean` 方法和 `@Import` 注解都会被忽略。如果 `@Component` 或 `@Configuration` 类标记有 `@Profile({"p1", "p2"})`，则除非已激活概要文件 'p1' 或 'p2'，否则不会注册或处理该类。如果给定的配置文件以 NOT 运算符 (!) 为前缀，则仅在该配置文件未激活时才注册带该注解的元素。例如，给定 `@Profile({"p1", "! p2"})`，如果配置文件 'p1' 处于活动状态或如果配置文件 'p2' 未处于活动状态，则会进行注册。

也可以在方法级别将 `@Profile` 声明为仅包含配置类的一个特定 bean（例如，特定 bean 的替代变体），如以下示例所示：

Java

```
@Configuration
public class AppConfig {

    @Bean("dataSource")
    @Profile("development") ①
    public DataSource standaloneDataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.HSQL)
            .addScript("classpath:com/bank/config/sql/schema.sql")
            .addScript("classpath:com/bank/config/sql/test-data.sql")
            .build();
    }

    @Bean("dataSource")
    @Profile("production") ②
    public DataSource jndiDataSource() throws Exception {
        Context ctx = new InitialContext();
        return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");
    }
}
```

- ① `standaloneDataSource` 方法仅在 `development` 配置文件中可用。
- ② `jndiDataSource` 方法仅在 `production` 配置文件中可用。

Kotlin

```
@Configuration
class AppConfig {

    @Bean("dataSource")
    @Profile("development") ①
    fun standaloneDataSource(): DataSource {
        return EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.HSQL)
            .addScript("classpath:com/bank/config/sql/schema.sql")
            .addScript("classpath:com/bank/config/sql/test-data.sql")
            .build()
    }

    @Bean("dataSource")
    @Profile("production") ②
    fun jndiDataSource() =
        InitialContext().lookup("java:comp/env/jdbc/datasource") as DataSource
}
```

- ① `standaloneDataSource` 方法仅在 `development` 配置文件中可用。
- ② `jndiDataSource` 方法仅在 `production` 配置文件中可用。

在 @Bean 方法上使用 @Profile 时，可能会出现特殊情况：如果 Java 方法名称相同的重载 @Bean 方法（类似于构造函数重载），则必须在所有重载方法上一致声明 @Profile 条件。如果条件不一致，则仅重载方法中第一个声明的条件很重要。因此，@Profile 不能用于选择具有另一个参数签名的重载方法。在创建时，同一 bean 的所有工厂方法之间的解析都遵循 Spring 的构造函数解析算法。



如果要使用不同的概要文件条件定义备用 bean，请使用 @Bean name 属性使用指向相同 bean 名称的不同 Java 方法名称，如前面的示例所示。如果参数签名都相同（例如，所有变体都具有无参工厂方法），则这是首先在有效 Java 类中表示这种排列的唯一方法（因为只能有一个特定名称和参数签名的方法）。

● XML Bean 定义配置文件

XML 对应项是 <beans> 元素的 profile 属性。我们前面的示例配置可以用两个 XML 文件重写，如下所示：

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

    <jdbc:embedded-database id="dataSource">
        <jdbc:script location="classpath:com/bank/config/sql/schema.sql"/>
        <jdbc:script location="classpath:com/bank/config/sql/test-data.sql"/>
    </jdbc:embedded-database>
</beans>
```

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

    <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>
</beans>
```

也可以避免在同一文件中拆分和嵌套 <beans/> 元素，如以下示例所示：

```

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

    <!-- other bean definitions -->

    <beans profile="development">
        <jdbc:embedded-database id="dataSource">
            <jdbc:script location="classpath:com/bank/config/sql/schema.sql"/>
            <jdbc:script location="classpath:com/bank/config/sql/test-data.sql"/>
        </jdbc:embedded-database>
    </beans>

    <beans profile="production">
        <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>
    </beans>
</beans>

```

spring-bean.xsd 已被限制为仅允许作为文件中的最后一个元素，这应有助于提供灵活性而不会在 XML 文件中造成混乱。

XML 对应项不支持前面描述的配置文件表达式。但是，可以使用！操作符取消配置文件。也可以通过嵌套配置文件来应用逻辑“与”，如以下示例所示：



```

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

    <!-- other bean definitions -->

    <beans profile="production">
        <beans profile="us-east">
            <jee:jndi-lookup id="dataSource" jndi-
name="java:comp/env/jdbc/datasource"/>
        </beans>
    </beans>
</beans>

```

在前面的示例中，如果 **production** 和 **us-east** 配置文件都处于活动状态，则将 **dataSource** bean 对外暴露。

● 激活一个 Profile

现在，我们已经更新了配置，我们仍然需要指示 Spring 哪个配置文件处于活动状态。如果我们现在启动示例应用程序，将会看到抛出 `NoSuchBeanDefinitionException` 的异常消息，因为容器找不到名为 `dataSource` 的 Spring bean。

可以通过多种方式来激活配置文件，但是最直接的方法是针对可通过 `ApplicationContext` 获得的 `Environment API` 以编程方式进行配置，下面的示例演示了如何进行配置：

Java

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
ctx.getEnvironment(). setActiveProfiles("development");
ctx.register(SomeConfig.class, StandaloneDataConfig.class, JndiDataConfig.class);
ctx.refresh();
```

Kotlin

```
val ctx = AnnotationConfigApplicationContext().apply {
    environment.setActiveProfiles("development")
    register(SomeConfig::class.java, StandaloneDataConfig::class.java,
    JndiDataConfig::class.java)
    refresh()
}
```

此外，您还可以通过 `spring.profiles.active` 属性以声明方式激活配置文件，该属性可以通过系统环境变量，JVM 系统属性，`web.xml` 中的 `servlet` 上下文参数来指定，甚至可以作为 JNDI 中的条目来指定(请参阅 [PropertySource Abstraction\(1.13.2\)](#))。在集成测试中，可以使用 `spring-test` 模块中的`@ActiveProfiles` 注解来声明活动配置文件(请参阅[环境配置文件的上下文配置](#))。

请注意，配置文件不是“非此即彼”的问题。你可以一次激活多个配置文件。通过编程，可以为 `setActiveProfiles()` 方法提供多个配置文件名称，该方法接受字符串参数。以下示例激活多个配置文件：

Java

```
ctx.getEnvironment(). setActiveProfiles("profile1", "profile2");
```

Kotlin

```
ctx.getEnvironment(). setActiveProfiles("profile1", "profile2")
```

声明性地，`spring.profiles.active` 可以接受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"
```

● 默认 Profile

默认配置文件表示默认情况下启用的配置文件。 请看以下示例：

Java

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

Kotlin

```
@Configuration  
@Profile("default")  
class DefaultDataConfig {  
  
    @Bean  
    fun dataSource(): DataSource {  
        return EmbeddedDatabaseBuilder()  
            .setType(EmbeddedDatabaseType.HSQL)  
            .addScript("classpath:com/bank/config/sql/schema.sql")  
            .build()  
    }  
}
```

如果没有配置文件处于活动状态，那么将创建 dataSource。 你可以看到这是为一个或多个 bean 提供默认定义的一种方法。 如果启用了任何配置文件，则默认配置文件将不适用。

您可以通过在 `Environment` 上使用 `setDefaultProfiles()` 或声明性地使用 `spring.profiles.default` 属性来更改默认配置文件的名称。

1.13.2. PropertySource 抽象

Spring 的环境抽象概念提供了可配置属性源层次结构上的搜索操作。请看以下列表：

Java

```
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);
```

Kotlin

```
val ctx = GenericApplicationContext()
val env = ctx.environment
val containsMyProperty = env.containsProperty("my-property")
println("Does my environment contain the 'my-property' property? $containsMyProperty")
```

在前面的代码片段中，我们看到了一种高级方式来询问 Spring 是否为当前环境定义了 `my-property` 属性。为了回答这个问题，环境对象在一组 `PropertySource` 对象上执行搜索。`PropertySource` 是对任何键值对源的简单抽象，并且 Spring 的 `StandardEnvironment` 配置有两个 `PropertySource` 对象—一个代表 JVM 系统属性的集合 (`System.getProperties()`) 和一个代表系统环境变量的集合 (`System.getenv()`)。



这些默认属性源适用于 `StandardEnvironment`，可在独立应用程序中使用。`StandardServletEnvironment` 填充了其他默认属性源，包括 `servlet` 配置和 `servlet` 上下文参数。它可以选择启用 `JndiPropertySource`。有关详细信息，请参见 javadoc。

具体来说，当您使用 `StandardEnvironment` 时，如果在运行时存在 `my-property` 系统属性或 `my-property` 环境变量，则对 `env.containsProperty("my-property")` 的调用将返回 `true`。

执行的搜索是分层的。默认情况下，系统属性优先于环境变量。因此，如果在调用 `env.getProperty ("my-property")` 时在两个地方都同时设置了 `my-property` 属性，则系统属性值“胜出”并返回。请注意，属性值不会合并，而是会被前面的条目完全覆盖。



对于常见的 `StandardServletEnvironment`，完整层次结构如下，下列表优先级自上而下：

1. `ServletConfig` 参数（如果适用，例如在 `DispatcherServlet` 上下文中）
2. `ServletContext` 参数（`web.xml` 上下文参数条目）
3. JNDI 环境变量（`java:comp/env/`条目）
4. JVM 系统属性（`-D` 命令行参数）
5. JVM 系统环境（操作系统环境变量）

最重要的是，整个机制是可配置的。也许您具有要集成到此搜索中的自定义属性源。为此，实现并实例化您自己的 `PropertySource` 并将其添加到当前环境的 `PropertySources` 集中。以下示例显示了如何执行此操作：

Java

```
ConfigurableApplicationContext ctx = new GenericApplicationContext();
MutablePropertySources sources = ctx.getEnvironment().getPropertySources();
sources.addFirst(new MyPropertySource());
```

Kotlin

```
val ctx = GenericApplicationContext()
val sources = ctx.environment.propertySources
sources.addFirst(MyPropertySource())
```

在前面的代码中，已在搜索中以最高优先级添加了 `MyPropertySource`。如果它包含 `my-property` 属性，则将检测并返回该属性，以支持任何其他 `PropertySource` 中的 `my-property` 属性。`MutablePropertySources` API 公开了许多方法，这些方法允许对属性源集进行精确操作。

1.13.3. 使用 `@PropertySource`

`@PropertySource` 注解为将 `PropertySource` 添加到 Spring 的 `Environment` 中提供了一种方便的声明性机制。

给定一个名为 `app.properties` 的文件，其中包含键值对 `testbean.name =`

`myTestBean`, 下面的`@Configuration`类使用`@PropertySource`, 以这样的方式调用`testBean.getName()`会返回`myTestBean`:

Java

```
@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;  
     }  
 }
```

Kotlin

```
@Configuration  
 @PropertySource("classpath:/com/myco/app.properties")  
 class AppConfig {  
  
     @Autowired  
     private lateinit var env: Environment  
  
     @Bean  
     fun testBean() = TestBean().apply {  
         name = env.getProperty("testbean.name")!!  
     }  
 }
```

`@PropertySource` 资源位置中存在的任何`...{}`占位符都是根据已经针对该环境注册的一组属性源来解析的, 如以下示例所示:

Java

```
@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;
    }
}
```

Kotlin

```
@Configuration
@PropertySource("classpath:/com/\${my.placeholder:default/path}/app.properties")
class AppConfig {

    @Autowired
    private lateinit var env: Environment

    @Bean
    fun testBean() = TestBean().apply {
        name = env.getProperty("testbean.name")!!
    }
}
```

假设 `my.placeholder` 存在于已注册的属性源之一（例如，系统属性或环境变量）中，则将占位符解析为相应的值。如果不是，则将 `default/path` 用作默认值。如果未指定默认值并且无法解析属性，则抛出 `IllegalArgumentException`。



根据 Java 8 约定，`@PropertySource` 注解是可重复的。但是，所有此类`@PropertySource` 注解都需要在同一层面上声明，可以直接在配置类上声明，也可以在同一自定义注解中声明为元注解。

1.13.4. 语句中占位符解析

回首过去，元素中占位符的值只能根据 JVM 系统属性或环境变量来解析。今非昔比已不再是这种情况。由于环境抽象是在整个容器中集成的，因此很容易通过它来路由占位符的解析。这意味着您可以按照自己喜欢的任何方式配置解析过程。您可以更改搜索系统属性和环境变量的优先级，也可以完全删除它们。您还可以根据需要将自己的属性源添加到组合中。

具体而言，以下语句无论在何处定义 `customer` 属性，只要在 `Environment` 中可用，该语句就起作用：

```
<beans>
    <import resource="com/bank/service/${customer}-config.xml"/>
</beans>
```

1.14. 注册一个 LoadTimeWeaver

Spring 使用 `LoadTimeWeaver` 在将类加载到 Java 虚拟机（JVM）中时对其进行动态转换。

要启用加载时织入，可以将`@EnableLoadTimeWeaving` 添加到您的`@Configuration` 类之一，如以下示例所示：

Java

```
@Configuration
@EnableLoadTimeWeaving
public class AppConfig { }
```

Kotlin

```
@Configuration
@EnableLoadTimeWeaving
class AppConfig
```

另外，对于 XML 配置，可以使用 `context:load-time-weaver` 元素：

```
<beans>
    <context:load-time-weaver/>
</beans>
```

为 `ApplicationContext` 配置后，该 `ApplicationContext` 中的任何 bean 都可以实现 `LoadTimeWeaverAware`，从而接收对加载时 weaver 实例的引用。与 Spring 的 JPA 支持结合使用时，该功能特别有用，因为在 JPA 类转换中可能需要进行加载时编织。咨询 `LocalContainerEntityManagerFactoryBean`. 有关 AspectJ 加载时编织的更多信息，请参见使用 Spring 框架中的 AspectJ(5.10.4)。

1.15. ApplicationContext 的其他功能

如本章介绍(第一章开头)中所讨论的, `org.springframework.beans.factory` 包提供了用于管理和操作 bean 的基本功能, 包括以编程方式。`org.springframework.context` 包添加了 `ApplicationContext` 接口, 该接口扩展了 `BeanFactory` 接口, 此外还扩展了其他接口以提供更多面向应用程序框架的样式的附加功能。许多人以完全声明性的方式使用 `ApplicationContext`, 甚至没有以编程方式创建它, 而是依靠诸如 `ContextLoader` 之类的支持类来自动实例化 `ApplicationContext`, 作为 Java EE Web 应用程序正常启动过程的一部分。

为了以更加面向框架的方式增强 `BeanFactory` 的功能, 上下文包还提供以下功能:

- 通过 `MessageSource` 接口访问 i18n 样式的消息。
- 通过 `ResourceLoader` 接口访问资源, 例如 URL 和文件。
- 通过使用 `ApplicationEventPublisher` 接口, 将事件发布到实现 `ApplicationListener` 接口的 bean。
- 加载多个(分层)上下文, 使每个上下文都通过 `HierarchicalBeanFactory` 接口集中在一个特定层上, 例如应用程序的 Web 层。

1.15.1. Internationalization using MessageSource

`ApplicationContext` 接口扩展了一个称为 `MessageSource` 的接口, 因此提供了国际化 (“i18n”) 功能。Spring 还提供了 `HierarchicalMessageSource` 接口, 该接口可以分层解析消息。这些接口一起提供了 Spring 效果消息解析的基础。这些接口上定义的方法包括:

- `String getMessage(String code, Object[] args, String default, Locale loc)`: 从 `MessageSource` 检索消息的基本方法。如果找不到针对指定语言环境的消息, 则使用默认消息。使用标准库提供的 `MessageFormat` 功能, 传入的所有参数都将成为替换值。
- `String getMessage(String code, Object[] args, Locale loc)`: 与先前的方法基本相同, 但有一个区别: 不能指定默认消息。如果找不到该消息, 则抛出 `NoSuchMessageException`。
- `String getMessage(MessageSourceResolvable resolvable, Locale`

`locale`): 前述方法中使用的所有属性也都包装在一个名为 `MessageSourceResolvable` 的类中，您可以将其与该方法一起使用。

加载 `ApplicationContext` 时，它将自动搜索在上下文中定义的 `MessageSource` bean。Bean 必须具有名称 `messageSource`。如果找到了这样的 bean，则对先前方法的所有调用都将委派给消息源。两者都实现 `HierarchicalMessageSource` 以便进行嵌套消息传递。`StaticMessageSource` 很少使用，但是提供了将消息添加到源中的编程方式。下面的示例显示 `ResourceBundleMessageSource`:

```
<beans>
    <bean id="messageSource"
          class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basenames">
            <list>
                <value>format</value>
                <value>exceptions</value>
                <value>windows</value>
            </list>
        </property>
    </bean>
</beans>
```

该示例假定您在类路径中定义了三个资源包，分别称为 `format`、`exception` 和 `windows`。解析消息的任何请求都通过 `ResourceBundle` 对象用 JDK 标准的解析消息来处理。就本示例而言，假定上述两个资源束文件的内容如下：

```
# in format.properties
message=Alligators rock!
```

```
# in exceptions.properties
argument.required=The {0} argument is required.
```

下一个示例显示了运行 `MessageSource` 功能的程序。请记住，所有 `ApplicationContext` 实现也是 `MessageSource` 实现，因此可以转换为 `MessageSource` 接口。

Java

```
public static void main(String[] args) {
    MessageSource resources = new ClassPathXmlApplicationContext("beans.xml");
    String message = resources.getMessage("message", null, "Default", Locale.ENGLISH);
    System.out.println(message);
}
```

Kotlin

```
fun main() {
    val resources = ClassPathXmlApplicationContext("beans.xml")
    val message = resources.getMessage("message", null, "Default", Locale.ENGLISH)
    println(message)
}
```

以上程序的结果输出如下：

```
Alligators rock!
```

总而言之，`MessageSource` 是在名为 `beans.xml` 的文件中定义的，该文件位于类路径的根目录下。`messageSource` bean 定义通过其 `basenames` 属性引用了许多资源包。列表中传递给 `basenames` 属性的三个文件在类路径的根目录下以文件形式存在，分别称为 `format.properties`, `exceptions.properties` 和 `windows.properties`。

下一个示例显示了传递给消息查找的参数。这些参数将转换为 `String` 对象，并插入到查找消息中的占位符中。

```
<beans>

    <!-- this MessageSource is being used in a web application -->
    <bean id="messageSource"
          class="org.springframework.context.support.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>
```

Java

```
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", Locale.ENGLISH);  
        System.out.println(message);  
    }  
}
```

Kotlin

```
class Example {  
  
    lateinit var messages: MessageSource  
  
    fun execute() {  
        val message = messages.getMessage("argument.required",  
            arrayOf("userDao"), "Required", Locale.ENGLISH)  
        println(message)  
    }  
}
```

调用 `execute()` 方法的结果输出如下：

```
The userDao argument is required.
```

关于国际化（“i18n”），Spring 的各种 `MessageSource` 实现遵循与标准 JDK `ResourceBundle` 相同的语言环境解析和后备规则。简而言之，并继续前面定义的示例 `messageSource`，如果您想根据英国（en-GB）语言环境解析消息，则可以分别创建名为 `format_en_GB.properties`，`exceptions_en_GB.properties` 和 `windows_en_GB.properties` 的文件。

通常，语言环境解析由应用程序的周围环境管理。在以下示例中，手动指定了针对其解析（英国）消息的语言环境：

```
# in exceptions_en_GB.properties  
argument.required=Ebagum lad, the ''{0}'' argument is required, I say, required.
```

Java

```
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);  
}
```

Kotlin

```
fun main() {  
    val resources = ClassPathXmlApplicationContext("beans.xml")  
    val message = resources.getMessage("argument.required",  
        arrayOf("userDao"), "Required", Locale.UK)  
    println(message)  
}
```

上述程序运行的结果如下：

```
Ebagum lad, the 'userDao' argument is required, I say, required.
```

您还可以使用 `MessageSourceAware` 接口获取对已定义的任何 `MessageSource` 的引用。创建和配置 bean 时，在 `ApplicationContext` 中定义的实现 `MessageSourceAware` 接口的任何 bean 都会与应用程序上下文的 `MessageSource` 一起注入。



作为 `ResourceBundleMessageSource` 的替代，Spring 提供了 `ReloadableResourceBundleMessageSource` 类。此变体支持相同的包文件格式，但比基于标准 JDK 的 `ResourceBundleMessageSource` 实现要灵活得多。特别是，它允许从任何 Spring 资源位置（不仅从类路径）读取文件，并支持热重载捆绑属性文件（同时在它们之间进行有效缓存）。有关详细信息，请参见 `ReloadableResourceBundleMessageSource` Javadoc。

1.15.2. 标准和自定义事件

通过 `ApplicationEvent` 类 和 `ApplicationListener` 接口 提供 `ApplicationContext` 中的事件处理。如果将实现 `ApplicationListener` 接口的 bean 部署到上下文中，则每次将 `ApplicationEvent` 发布到 `ApplicationContext` 时，都会通知该 bean。本质上，这是标准的观察者模式。



从 Spring 4.2 开始，事件基础架构显著改进，并提供了基于注解的模型(207 页)以及发布任意事件（即不一定从 `ApplicationEvent` 扩展的

对象) 的功能。发布此类对象后, 我们会为您包装一个事件。

下表描述了 Spring 提供的标准事件:

事件	说明
ContextRefreshedEvent	在初始化或刷新 ApplicationContext 时发布 (例如, 通过使用 ConfigurableApplicationContext 接口上的 refresh() 方法)。在这里, “initialized”是指所有 Bean 都已加载, 检测到并激活了后处理器 Bean, 已预先实例化单例并且可以使用 ApplicationContext 对象。只要尚未关闭上下文, 只要选定的 ApplicationContext 实际上支持这种“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。
ContextStrartedEvent	在 ConfigurableApplicationContext 接口上使用 start() 方法启动 ApplicationContext 时发布。在这里, “Strated”是指所有 Lifecycle bean 都收到一个明确的启动信号。通常, 此信号用于在显式停止后重新启动 Bean, 但也可以用于启动尚未配置为自动启动的组件(例如, 尚未在初始化时启动的组件)。
ContextStoppedEvent	通过 使用 ConfigurableApplicationContext 接口上的 stop() 方法停止 ApplicationContext 时发布。在这里, “Stopped”表示所有 Lifecycle bean 都收到一个明确的停止信号。 停止的上下文可以通过 start() 调用重新启动。
ContextClosedEvent	通过 使用 ConfigurableApplicationContext 接口上的 close()方法关闭 ApplicationContext 或通过 JVM shutdown 钩子时发布。在这里, “Closed”意味着所有单例 bean 将被销毁。 关闭上下文后, 它将完成生命周期, 无法刷新或重新启动。
RequestHandledEvent	一个特定于 Web 的事件, 告诉所有 bean HTTP 请求已经被执行。 服务。请求完成后, 将发布此事件。此事件仅适用于使用 Spring 的 DispatcherServlet 的 Web 应用程序。
ServletRequestHandledEvent	RequestHandledEvent 的子类, 添加了特定于 Servlet 的上下文信息。

您还可以创建和发布自己的自定义事件。以下示例显示了一个简单的类，该类扩展了 Spring 的 ApplicationEvent 基类：

Java

```
public class BlockedListEvent extends ApplicationEvent {  
  
    private final String address;  
    private final String content;  
  
    public BlockedListEvent(Object source, String address, String content) {  
        super(source);  
        this.address = address;  
        this.content = content;  
    }  
  
    // accessor and other methods...  
}
```

Kotlin

```
class BlockedListEvent(source: Any,  
                      val address: String,  
                      val content: String) : ApplicationEvent(source)
```

若要发布自定义 ApplicationEvent，请在 ApplicationEventPublisher 上调用 publishEvent() 方法。通常，这是通过创建一个实现 ApplicationEventPublisherAware 的类并将其注册为 Spring Bean 来完成的。以下示例显示了此类：

Java

```
public class EmailService implements ApplicationEventPublisherAware {  
  
    private List<String> blockedList;  
    private ApplicationEventPublisher publisher;  
  
    public void setBlockedList(List<String> blockedList) {  
        this.blockedList = blockedList;  
    }  
  
    public void setApplicationEventPublisher(ApplicationEventPublisher publisher) {  
        this.publisher = publisher;  
    }  
  
    public void sendEmail(String address, String content) {  
        if (blockedList.contains(address)) {  
            publisher.publishEvent(new BlockedListEvent(this, address, content));  
            return;  
        }  
        // send email...  
    }  
}
```

Kotlin

```
class EmailService : ApplicationEventPublisherAware {  
  
    private lateinit var blockedList: List<String>  
    private lateinit var publisher: ApplicationEventPublisher  
  
    fun setBlockedList(blockedList: List<String>) {  
        this.blockedList = blockedList  
    }  
  
    override fun setApplicationEventPublisher(publisher: ApplicationEventPublisher) {  
        this.publisher = publisher  
    }  
  
    fun sendEmail(address: String, content: String) {  
        if (blockedList!!.contains(address)) {  
            publisher!!.publishEvent(BlockedListEvent(this, address, content))  
            return  
        }  
        // send email...  
    }  
}
```

在配置时，Spring 容器检测到 `EmailService` 实现了 `ApplicationEventPublisherAware` 并自动调用 `setApplicationEventPublisher()`。实际上，传入的参数是 Spring 容器本身。您正在通过其 `ApplicationEventPublisher` 接口与应用程序上下文进行交互。

要接收自定义 `ApplicationEvent`，可以创建一个实现 `ApplicationListener` 的类，并将其注册为 Spring Bean。以下示例显示了此类：

Java

```
public class BlockedListNotifier implements ApplicationListener<BlockedListEvent> {

    private String notificationAddress;

    public void setNotificationAddress(String notificationAddress) {
        this.notificationAddress = notificationAddress;
    }

    public void onApplicationEvent(BlockedListEvent event) {
        // notify appropriate parties via notificationAddress...
    }
}
```

Kotlin

```
class BlockedListNotifier : ApplicationListener<BlockedListEvent> {

    lateinit var notificationAddress: String

    override fun onApplicationEvent(event: BlockedListEvent) {
        // notify appropriate parties via notificationAddress...
    }
}
```

注意，`ApplicationListener` 通常使用您的自定义事件的类型（在前面的示例中为 `BlockedListEvent`）进行参数化。这意味着 `onApplicationEvent()` 方法可以保持类型安全，从而避免了向下转换的需要。您可以根据需要注册任意数量的事件监听器，但是请注意，默认情况下，事件监听器会同步接收事件。这意味着 `publishEvent()` 方法将阻塞，直到所有监听器都已完成对事件的处理为止。这种同步和单线程方法的一个优点是，当监听器收到事件时，如果有可用的事务上下文，它将在发布者的事务上下文内部进行操作。如果有必要采用其他发布事件的策略，请参见 Spring 的 `ApplicationEventMulticaster` 接口的 `javadoc` 和 `SimpleApplicationEventMulticaster` 实现的配置选项。

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="blockedList">
        <list>
            <value>known.spammer@example.org</value>
            <value>known.hacker@example.org</value>
            <value>john.doe@example.org</value>
        </list>
    </property>
</bean>

<bean id="blockedListNotifier" class="example.BlockedListNotifier">
    <property name="notificationAddress" value="blockedlist@example.org"/>
</bean>

```

将所有内容放在一起，当调用 `emailService` bean 的 `sendEmail()` 方法时，如果有任何应阻止的电子邮件，则会发布 `BlockedListEvent` 类型的自定义事件。`blockedListNotifier` bean 被注册为 `ApplicationListener` 并接收 `BlockedListEvent`，这时它可以通知适当的参与者。



Spring 的事件机制旨在在同一应用程序上下文内在 Spring bean 之间进行简单的通信。但是，对于更复杂的企业集成需求，单独维护的 [Spring Integration](#) 项目为基于著名的 Spring 编程模型构建轻量级，面向模式，事件驱动的架构提供了完整的支持。

● 基于注解的事件监听器

从 Spring 4.2 开始，您可以使用 `@EventListener` 注解在托管 Bean 的任何公共方法上注册事件侦听器。`BlockedListNotifier` 可以重写如下：

Java

```

public class BlockedListNotifier {

    private String notificationAddress;

    public void setNotificationAddress(String notificationAddress) {
        this.notificationAddress = notificationAddress;
    }

    @EventListener
    public void processBlockedListEvent(BlockedListEvent event) {
        // notify appropriate parties via notificationAddress...
    }
}

```

Kotlin

```
class BlockedListNotifier {  
  
    lateinit var notificationAddress: String  
  
    @EventListener  
    fun processBlockedListEvent(event: BlockedListEvent) {  
        // notify appropriate parties via notificationAddress...  
    }  
}
```

方法签名再次声明其侦听的事件类型，但是这次使用灵活的名称并且没有实现特定的监听器接口。只要实际事件类型在其实现层次结构中解析您的通用参数，也可以通过通用类型来缩小事件类型。

如果您的方法应该侦听多个事件，或者您想完全不使用任何参数来定义它，则事件类型也可以在注解本身上指定。以下示例显示了如何执行此操作：

Java

```
@EventListener({ContextStartedEvent.class, ContextRefreshedEvent.class})  
public void handleContextStart() {  
    // ...  
}
```

Kotlin

```
@EventListener(ContextStartedEvent::class, ContextRefreshedEvent::class)  
fun handleContextStart() {  
    // ...  
}
```

还可以通过使用定义 [SpEL 表达式\(第 4 章\)](#) 的注解的 `condition` 属性来添加其他运行时过滤，该注释应匹配以针对特定事件实际调用该方法。以下示例显示了仅当事件的 `content` 属性等于 `my-event` 时，才可以重写我们的通知程序以进行调用：

Java

```
@EventListener(condition = "#blEvent.content == 'my-event'")  
public void processBlockedListEvent(BlockedListEvent blockedListEvent) {  
    // notify appropriate parties via notificationAddress...  
}
```

Kotlin

```
@EventListener(condition = "#blEvent.content == 'my-event'")  
fun processBlockedListEvent(blockedListEvent: BlockedListEvent) {  
    // notify appropriate parties via notificationAddress...  
}
```

每个 SpEL 表达式都会根据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：

名称	位置	描述	示例
Event	root object	实际的 ApplicationEvent。	#root.event 或者 event
Arguments array	root object	用于调用方法的参数（作为对象数组）。	# root.args 或 args; args [0] 访问第一个参数，依此类推。
Argument name	evaluation context	任何方法参数的名称。 如果由于某种原因这些名称不可用（例如，由于在编译的字节码中没有调试信息），则还可以使用#a <#arg> 语法（其中<#arg>代表参数索引（从 0 开始）。	#blEvent 或 # a0 (您也可以使用 # p0 或 #p <#arg> 参数表示法作为别名)

请注意，即使您的方法签名实际上引用了已发布的任意对象，#root.event 也使您可以访问基础事件。如果由于处理另一个事件而需要发布一个事件，则可以更改方法签名以返回应发布的事件，如以下示例所示：

Java

```
@EventListener  
public ListUpdateEvent handleBlockedListEvent(BlockedListEvent event) {  
    // notify appropriate parties via notificationAddress and  
    // then publish a ListUpdateEvent...  
}
```

Kotlin

```
@EventListener  
fun handleBlockedListEvent(event: BlockedListEvent): ListUpdateEvent {  
    // notify appropriate parties via notificationAddress and  
    // then publish a ListUpdateEvent...  
}
```



异步监听器不支持此功能。

此新方法为上述方法处理的每个 `BlockedListEvent` 发布一个新的 `ListUpdateEvent`。如果需要发布多个事件，则可以返回事件的集合。

● 异步监听器

如果希望特定的监听器异步处理事件，则可以重用[常规的@Async 支持](#)。以下示例显示了如何执行此操作：

Java

```
@EventListener  
@Async  
public void processBlockedListEvent(BlockedListEvent event) {  
    // BlockedListEvent is processed in a separate thread  
}
```

Kotlin

```
@EventListener  
@Async  
fun processBlockedListEvent(event: BlockedListEvent) {  
    // BlockedListEvent is processed in a separate thread  
}
```

使用异步事件时，请注意以下限制：

- 如果异步事件监听器引发 `Exception`，则不会将其传播到调用方。有关更多详细信息，请参见 [AsyncUncaughtExceptionHandler](#)。
- 异步事件监听器方法无法通过返回值来发布后续事件。如果您需要发布另一个事件作为处理的结果，请注入 `ApplicationEventPublisher` 以手动发布事件。

● 指定监听器顺序

如果需要先调用一个监听器，则可以将[@Order](#) 注解添加到方法声明中，如以下示例所示：

Java

```
@EventListener  
 @Order(42)  
 public void processBlockedListEvent(BlockedListEvent event) {  
     // notify appropriate parties via notificationAddress...  
 }
```

Kotlin

```
@EventListener  
 @Order(42)  
 fun processBlockedListEvent(event: BlockedListEvent) {  
     // notify appropriate parties via notificationAddress...  
 }
```

● 一般事件

您还可以使用泛型来进一步定义事件的结构。考虑使用 `EntityCreatedEvent<T>`，其中 `T` 是已创建的实际实体的类型。例如，您可以创建以下监听器定义以仅接收 `Person` 的 `EntityCreatedEvent`：

Java

```
@EventListener  
 public void onPersonCreated(EntityCreatedEvent<Person> event) {  
     // ...  
 }
```

Kotlin

```
@EventListener  
 fun onPersonCreated(event: EntityCreatedEvent<Person>) {  
     // ...  
 }
```

由于类型擦除，只有在触发的事件解析了事件侦听器所依据的通用参数（即类似 `PersonCreatedEvent` 的类扩展 `EntityCreatedEvent <Person> {...}`）时，此方法才起作用。

在某些情况下，如果所有事件都遵循相同的结构，这可能会变得很乏味（就像前面示例中的事件一样）。在这种情况下，您可以实现 `ResolvableTypeProvider` 来指定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：

Java

```
public class EntityCreatedEvent<T> extends ApplicationEvent implements ResolvableTypeProvider {

    public EntityCreatedEvent(T entity) {
        super(entity);
    }

    @Override
    public ResolvableType getResolvableType() {
        return ResolvableType.forClassWithGenerics(getClass(),
ResolvableType.forInstancegetSource());
    }
}
```

Kotlin

```
class EntityCreatedEvent<T>(entity: T) : ApplicationEvent(entity),
ResolvableTypeProvider {

    override fun getResolvableType(): ResolvableType? {
        return ResolvableType.forClassWithGenerics(javaClass,
ResolvableType.forInstancegetSource())
    }
}
```



这个玩意不只是对 `ApplicationEvent` 而是你只要当做事件发送的任何对象他都管事。

1.15.3. 获取低级资源的简便方法

为了获得最佳用法和对应用程序上下文的理解，您应该熟悉 Spring 的 `Resource` 抽象，如[资源（第 2 章）](#)所述。

应用程序上下文是 `ResourceLoader`，可用于加载 `Resource` 对象。`Resource` 本质上是 JDK `java.net.URL` 类的功能更丰富的版本。实际上，`Resource` 的实现现在适当的地方包装了 `java.net.URL` 的实例。资源可以以透明的方式从几乎任何位置获取低级资源，包括从类路径，文件系统位置，可使用标准 URL 描述的任何位置以及其他一些变体。如果资源位置字符串是没有任何特殊前缀的简单路径，则这些资源的来源是特定的，并且适合于实际的应用程序上下文类型。

您可以配置部署到应用程序上下文中的 Bean，以实现特殊的回调接口 `ResourceLoaderAware`，以便在初始化时自动调用，并将应用程序上下文本身作为 `ResourceLoader` 传入。您还可以公开 `Resource` 类型的属性，以用于访问静态资源。它们像其他任何属性一样注入其中。您可以将那些 `Resource` 属性指定为简单的 `String` 路径，

并在部署 bean 时依靠从这些文本字符串到实际 Resource 对象的自动转换。

提供给 `ApplicationContext` 构造函数的一个或多个位置路径实际上是资源字符串，并且根据特定的上下文实现以简单的形式对其进行适当处理。例如，`ClassPathXmlApplicationContext` 将简单的位置路径视为类路径位置。您也可以使用带有特殊前缀的位置路径（资源字符串）来强制从类路径或 URL 中加载定义，而不管实际的上下文类型如何。

1.15.4. 对于 Web 应用的简便应用上下文的实例化

您可以使用例如 `ContextLoader` 声明性地创建 `ApplicationContext` 实例。当然，您也可以使用 `ApplicationContext` 实现之一以编程方式创建 `ApplicationContext` 实例。

您可以使用 `ContextLoaderListener` 注册 `ApplicationContext`，如以下示例所示：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/daoContext.xml /WEB-INF/applicationContext.xml</param-value>
</context-param>

<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-
class>
</listener>
```

监听器检查 `contextConfigLocation` 参数。如果参数不存在，那么监听器将使用 `/WEB-INF/applicationContext.xml` 作为默认值。当参数确实存在时，侦听器将使用预定义的定界符（逗号，分号和空格）来分隔 `String`，并将这些值用作搜索应用程序上下文的位置。还支持 Ant-style 的路径模式。示例包括 `/WEB-INF/*Context.xml`（适用于所有名称以 `Context.xml` 结尾且位于 `WEB-INF` 目录中的文件）和 `/WEB-INF/**/*Context.xml`（适用于所有此类文件）文件在 `WEB-INF` 的任何子目录中）。

1.15.5. 将一个 Spring 应用上下文部署成一个 Java EE RAR 文件

可以将 Spring `ApplicationContext` 部署为 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。这等效于引导独立的 `ApplicationContext`（仅托管在 Java EE 环境中）能够访问 Java EE 服务器功能。RAR 部署是部署无头 WAR 文件的方案的一种更自然的选择-实际上，这种 WAR 文件没有任何 HTTP 入口点，仅用于在 Java EE 环境中引导 `Spring ApplicationContext`。

对于不需要 HTTP 入口点而仅由消息端点和计划的作业组成的应用程序上下文，RAR 部

署是理想的选择。在这样的上下文中，Bean 可以使用应用程序服务器资源，例如 JTA 事务管理器和 JNDI 绑定的 JDBC `DataSource` 实例以及 JMS `ConnectionFactory` 实例，并且还可以在平台的 JMX 服务器上注册 - 整个过程都通过 Spring 的标准事务管理以及 JNDI 和 JMX 支持工具进行。应用程序组件还可以通过 Spring 的 `TaskExecutor` 抽象与应用程序服务器的 JCA `WorkManager` 进行交互。

有关 RAR 部署中涉及的配置详细信息，请参见 `SpringContextResourceAdapter` 类的 `javadoc`。

对于将 Spring `ApplicationContext` 作为 Java EE RAR 文件的简单部署：

- ① 将所有应用程序类打包到 RAR 文件（这是具有不同文件扩展名的标准 JAR 文件）中。将所有必需的库 JAR 添加到 RAR 归档文件的根目录中。添加一个 `META-INF / ra.xml` 部署描述符（如 `SpringContextResourceAdapter` 的 `javadoc` 中所示）和相应的 Spring XML `bean` 定义文件（通常为 `META-INF / applicationContext.xml`）。
- ② 将生成的 RAR 文件拖放到应用程序服务器的部署目录中。



此类 RAR 部署单元通常是独立的。它们不会将组件暴露给外界，甚至不会暴露给同一应用程序的其他模块。与基于 RAR 的 `ApplicationContext` 的交互通常是通过与其他模块共享的 JMS 目标进行的。例如，基于 RAR 的 `ApplicationContext` 还可以安排一些作业或对文件系统（或类似文件）中的新文件做出反应。如果需要允许从外部进行同步访问，则可以（例如）导出 RMI 端点，该端点可以由同一台计算机上的其他应用程序模块使用。

1.16. Bean 工厂

`BeanFactory` API 为 Spring 的 IoC 功能提供了基础。它的特定作用主要用于与 Spring 的其他部分以及相关的第三方框架集成，并且其 `DefaultListableBeanFactory` 实现是更高级别的 `GenericApplicationContext` 容器中的关键委托。

`BeanFactory` 和相关接口（例如 `BeanFactoryAware`, `InitializingBean`, `DisposableBean`）是其他框架组件的重要集成点。不需要任何注解甚至反射，它们甚至可以在容器及其组件之间进行非常有效的交互。应用级 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程序配置进行声明式依赖注入。

请注意，核心 `BeanFactory` API 级别及其 `DefaultListableBeanFactory` 实现不对

配置格式或要使用的任何组件注解进行假设。所有这些 flavors 都是通过扩展（例如 `XmlBeanDefinitionReader` 和 `AutowiredAnnotationBeanPostProcessor`）引入的，并以核心元数据表示形式对共享 `BeanDefinition` 对象进行操作。这就是使 Spring 的容器如此灵活和可扩展的本质。

1.16.1. BeanFactory 还是 ApplicationContext?

本节说明 `BeanFactory` 和 `ApplicationContext` 容器级别之间的区别以及对引导的影响。

除非有充分的理由，否则应使用 `ApplicationContext`，将 `GenericApplicationContext` 及其子类 `AnnotationConfigApplicationContext` 作为自定义引导的常见实现。这些是所有常见目的 Spring 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（自 5.0 版本起）注册功能性 Bean 定义。

因为 `ApplicationContext` 包含 `BeanFactory` 的所有功能，所以通常建议在普通 `BeanFactory` 上使用，除非需要对 Bean 处理的完全控制。在 `ApplicationContext`（例如 `GenericApplicationContext` 实现）中，按照约定（即，按 Bean 名称或 Bean 类型（尤其是后处理器））检测到几种 Bean，而普通的 `DefaultListableBeanFactory` 不知道任何特殊的 Bean。

对于许多扩展的容器功能，例如注释处理和 AOP 代理，`BeanPostProcessor` 扩展点至关重要。如果仅使用普通的 `DefaultListableBeanFactory`，则默认情况下不会检测和激活此类后处理器。这种情况可能会造成混淆，因为您的 bean 配置实际上并没有错。而是在这种情况下，需要通过其他设置完全引导容器。下表列出了 `BeanFactory` 和 `ApplicationContext` 接口和实现所提供的功能。

特性	BeanFactory	ApplicationContext
Bean 初始化/装配	Yes	Yes
集成生命周期管理	No	Yes
自动化 <code>BeanPostProcessor</code> 注册	No	Yes
自动化 <code>BeanFactoryPostProcessor</code> 注册	No	Yes
简便 <code>MessageSource</code> 访问（为了初始化）	No	Yes
内置 <code>ApplicationEvent</code> 发布机制	No	Yes

要向 `DefaultListableBeanFactory` 显式注册 Bean 后处理器，需要以编程方式调用 `addBeanPostProcessor`，如以下示例所示：

Java

```
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
```

Kotlin

```
val factory = DefaultListableBeanFactory()
// populate the factory with bean definitions

// now register any needed BeanPostProcessor instances
factory.addBeanPostProcessor(AutowiredAnnotationBeanPostProcessor())
factory.addBeanPostProcessor(MyBeanPostProcessor())

// now start using the factory
```

要将 `BeanFactoryPostProcessor` 应用于普通的 `DefaultListableBeanFactory`，您需要调用其 `postProcessBeanFactory` 方法，如以下示例所示：

Java

```
DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(factory);
reader.loadBeanDefinitions(new FileSystemResource("beans.xml"));

// bring in some property values from a Properties file
PropertySourcesPlaceholderConfigurer cfg = new PropertySourcesPlaceholderConfigurer();
cfg.setLocation(new FileSystemResource("jdbc.properties"));

// now actually do the replacement
cfg.postProcessBeanFactory(factory);
```

Kotlin

```
val factory = DefaultListableBeanFactory()
val reader = XmlBeanDefinitionReader(factory)
reader.loadBeanDefinitions(FileSystemResource("beans.xml"))

// bring in some property values from a Properties file
val cfg = PropertySourcesPlaceholderConfigurer()
cfg.setLocation(FileSystemResource("jdbc.properties"))

// now actually do the replacement
cfg.postProcessBeanFactory(factory)
```

在这两种情况下，显式的注册步骤都不方便，这就是为什么在 Spring 支持的应用程序中，各种 ApplicationContext 变量比普通的 DefaultListableBeanFactory 更为可取的原因，尤其是在典型企业设置中依赖 BeanFactoryPostProcessor 和 BeanPostProcessor 实例来扩展容器功能时。



AnnotationConfigApplicationContext 已注册了所有常见的注解的后处理器，并且可以通过诸如@EnableTransactionManagement之类的配置注解在后台引入其他处理器。在 Spring 基于注解的配置模型的抽象级别上，bean 后处理器的概念仅是内部容器详细信息。