# Spring核心技术 (Core Technologies)

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

其中最重要的是Spring Framework的控制反转（IoC）容器。 对Spring框架的IoC容器进行彻底的处理后，我们将全面介绍Spring的面向方面编程（AOP）技术。 Spring Framework有自己的AOP框架，它在概念上易于理解，并成功解决了Java企业编程中AOP要求的80％最佳点。

还提供了Spring与AspectJ集成的覆盖范围（目前最丰富的 - 在功能方面 - 当然也是Java企业领域中最成熟的AOP实现）。

# 1. Spring IOC容器 (The IoC Container)

本章介绍Spring Inversion of Control（IoC）容器。

## 1.1. Spring IoC容器和Bean简介 (Introduction to the Spring IoC Container and Beans)

本章介绍了控制反转（IoC）原理的Spring Framework实现。 （参见控制反转。）IoC也称为依赖注入（DI）。 这是一个过程，通过这个过程，对象只能通过构造函数参数，工厂方法的参数或在从工厂方法构造或返回的对象实例上设置的属性来定义它们的依赖关系（即，它们使用的其他对象）。。 然后容器在创建bean时注入这些依赖项。 这个过程基本上是bean本身通过使用类的直接构造或诸如服务定位器模式之类的机制来控制其依赖关系的实例化或位置的逆（因此名称，控制反转）。

org.springframework.beans和org.springframework.context包是Spring Framework的IoC容器的基础。 BeanFactory接口提供了一种能够管理任何类型对象的高级配置机制。 ApplicationContext是BeanFactory的子接口。 它补充说：

* 更容易与Spring的AOP功能集成
* 消息资源处理（用于国际化）
* 事件订阅
* 特定于应用程序层的上下文，例如WebApplicationContext，用于Web应用程序。

简而言之，BeanFactory提供配置框架和基本功能，ApplicationContext添加了更多特定于企业的功能。 ApplicationContext是BeanFactory的完整超集，在本章中仅用于Spring的IoC容器的描述。 有关使用BeanFactory而不是ApplicationContext的更多信息，请参阅BeanFactory。

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

## 1.2. Spring IoC容器概览 (Container Overview)

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

Spring提供了ApplicationContext接口的几个实现。在独立应用程序中，通常会创建ClassPathXmlApplicationContext或FileSystemXmlApplicationContext的实例。虽然XML是定义配置元数据的传统格式，但您可以通过提供少量XML配置来声明容器使用Java注释或代码作为元数据格式，以声明方式启用对这些其他元数据格式的支持。

在大多数应用程序方案中，不需要显式用户代码来实例化Spring IoC容器的一个或多个实例。例如，在Web应用程序场景中，应用程序的web.xml文件中的简单八行（左右）样板Web描述符XML通常就足够了（请参阅Web应用程序的便捷ApplicationContext实例化）。如果您使用Spring Tool Suite（基于Eclipse的开发环境），只需点击几下鼠标或按键即可轻松创建此样板配置。**ContextLoaderListener（的contextInitialized方法初始化父容器），DispatcherServlet（的父类initWebApplicationContext方法，初始化子容器）**

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



*Figure 1. The Spring IoC container*

### 1.2.1.配置元数据（SpringBean的定义方式）(Configuration Metadata)

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

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

|  |  |
| --- | --- |
|  | 基于XML的元数据不是唯一允许的配置元数据形式。 Spring IoC容器本身完全与实际编写此配置元数据的格式分离。 目前，许多开发人员为其Spring应用程序选择基于JavaConfig的配置。 |

有关在Spring容器中使用其他形式的元数据的信息，请参阅：

* 基于注释的配置：Spring 2.5引入了对基于注释的配置元数据的支持。
* 基于Java的配置：从Spring 3.0开始，Spring JavaConfig项目提供的许多功能成为核心Spring Framework的一部分。因此，您可以使用Java而不是XML文件在应用程序类外部定义bean。要使用这些新功能，请参阅@Configuration，@ Bean，@ Import和@Depend son注释。

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

这些bean定义对应于构成应用程序的实际对象。通常，您定义服务层对象，数据访问对象（DAO），表示对象（如Struts Action实例），基础结构对象（如Hibernate SessionFactories，JMS队列等）。通常，不会在容器中配置细粒度域对象，因为创建和加载域对象通常由DAO和业务逻辑负责。但是，您可以使用Spring与AspectJ的集成来配置在IoC容器控制之外创建的对象。请参阅使用AspectJ使用Spring依赖注入域对象。

以下示例显示了基于XML的配置元数据的基本结构：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans  <http://www.springframework.org/schema/beans/spring-beans.xsd>">  <bean id="..." class="..."> ❶ ❷  <!-- 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> |
| ❶id属性是一个标识单个bean定义的字符串。 |
| ❷class属性定义bean的类型并使用完全限定的classname。 |

id属性的值指的是协作对象。 在此示例中未显示用于引用协作对象的XML。 有关更多信息，请参阅依赖项。

### 1.2.2. 实例化容器 (Instantiating a Container)

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

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

|  |  |
| --- | --- |
|  | 在了解了Spring的IoC容器之后，您可能想要了解有关Spring的资源抽象的更多信息（如参考资料中所述），它提供了一种从URI语法中定义的位置读取InputStream的便捷机制。 特别是，资源路径用于构建应用程序上下文，如应用程序上下文和资源路径中所述。 |

以下示例显示了服务层对象（services.xml）配置文件：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans.xsd">  <!-- services -->  <bean id="petStore" class="org.springframework.samples.jpetstore.services.PetStoreServiceImpl">  <property name="accountDao" ref="accountDao"/>  <property name="itemDao" ref="itemDao"/>  <!-- additional collaborators and configuration for this bean go here -->  </bean>  <!-- more bean definitions for services go here -->  </beans> |

以下示例显示了数据访问对象daos.xml文件：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans.xsd">  <bean id="accountDao"  class="org.springframework.samples.jpetstore.dao.jpa.JpaAccountDao">  <!-- additional collaborators and configuration for this bean go here -->  </bean>  <bean id="itemDao" class="org.springframework.samples.jpetstore.dao.jpa.JpaItemDao">  <!-- additional collaborators and configuration for this bean go here -->  </bean>  <!-- more bean definitions for data access objects go here -->  </beans> |

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

#### 编写基于XML的配置元数据

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

您可以使用应用程序上下文构造函数从所有这些XML片段加载bean定义。此构造函数采用多个Resource位置，如上一节中所示。或者，使用一个或多个<import />元素来从另一个或多个文件加载bean定义。以下示例显示了如何执行此操作：

|  |
| --- |
| <beans>  <import resource="services.xml"/>  <import resource="resources/messageSource.xml"/>  <import resource="/resources/themeSource.xml"/>  <bean id="bean1" class="..."/>  <bean id="bean2" class="..."/>  </beans> |

在前面的示例中，外部bean定义从三个文件加载：services.xml，messageSource.xml和themeSource.xml。 所有位置路径都与执行导入的定义文件相关，因此services.xml必须与执行导入的文件位于相同的目录或类路径位置，而messageSource.xml和themeSource.xml必须位于该位置下的资源位置 导入文件。 如您所见，忽略前导斜杠。 但是，鉴于这些路径是相对的，最好不要使用斜杠。 根据Spring Schema，正在导入的文件的内容（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“../”路径引用父目录中的文件。 这样做会对当前应用程序之外的文件创建依赖关系。 特别是，不建议对classpath使用此引用：URL（例如，classpath：../ services.xml），其中运行时解析过程选择“最近的”类路径根，然后查看其父目录。 类路径配置更改可能导致选择不同的，不正确的目录。  您始终可以使用完全限定的资源位置而不是相对路径：例如，file：C：/config/services.xml或classpath：/config/services.xml。 但是，请注意您将应用程序的配置与特定的绝对位置耦合。 通常最好为这样的绝对位置保持间接 - 例如，通过在运行时针对JVM系统属性解析的“$ {...}”占位符。 |

命名空间本身提供了导入指令功能。 除了普通bean定义之外的其他配置功能在Spring提供的一系列XML命名空间中可用 - 例如，context和util命名空间。

#### Groovy Bean定义DSL

作为外化配置元数据的另一个示例，bean定义也可以在Spring的Groovy Bean Definition DSL中表示，如已知的那样：

|  |
| --- |
| 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  }  }  } |

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

### 1.2.3. 使用容器 (Use Container)

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

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

|  |
| --- |
| // create and configure beans  ApplicationContext context = new ClassPathXmlApplicationContext("services.xml", "daos.xml");  // retrieve configured instance  PetStoreService service = context.getBean("petStore", PetStoreService.class);  // use configured instance  List<String> userList = service.getUsernameList(); |

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

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

最灵活的变体是GenericApplicationContext与读者委托相结合 - 例如，使用XML文件的XmlBeanDefinitionReader，如以下示例所示：

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

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

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

您可以在同一个ApplicationContext上混合和匹配这些读者委托，从不同的配置源中读取bean定义。

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

## 1.3. Spring Bean概览 (Bean Overview)

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

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

* 包限定的类名：通常是正在定义的bean的实际实现类。
* Bean行为配置元素，说明bean在容器中的行为方式（范围，生命周期回调等）。
* 引用bean执行其工作所需的其他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。

此元数据转换为组成每个bean定义的一组属性。下表描述了这些属性：

*表1：bean的定义：*

|  |  |
| --- | --- |
| **Property** | **Explained in…​** |
| Class | Instantiating Beans |
| Name | Naming Beans |
| Scope | Bean Scopes |
| Constructor arguments | Dependency Injection |
| Properties | Dependency Injection |
| Autowiring mode | Autowriting Collaborators |
| Lazy initialization mode | Lazy-initialized Beans |
| Initializition 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。 这是通过getBeanFactory（）方法访问ApplicationContext的BeanFactory来完成的，该方法返回BeanFactory DefaultListableBeanFactory实现。 DefaultListableBeanFactory通过registerSingleton（..）和registerBeanDefinition（..）方法支持此注册。 但是，典型应用程序仅适用于通过常规bean定义元数据定义的bean。

|  |  |
| --- | --- |
|  | 需要尽早注册Bean元数据和手动提供的单例实例，以便容器在自动装配和其他内省步骤期间正确推理它们。 虽然在某种程度上支持覆盖现有元数据和现有单例实例，但是在运行时注册新bean（与对工厂的实时访问同时）并未得到官方支持，并且可能导致并发访问异常，bean容器中的状态不一致，或者 都。 |

### 1.3.1 命名Beans(Naming Beans)

每个bean都有一个或多个标识符。这些标识符在托管bean的容器中必须是唯一的。 bean通常只有一个标识符。但是，如果它需要多个，则额外的可以被视为别名。

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

您不需要为bean提供名称或ID。如果未明确提供名称或标识，则容器会为该Bean生成唯一的名称。但是，如果要通过名称引用该bean，通过使用ref元素或Service Locator样式查找，则必须提供名称。不提供名称的动机与使用内部bean和自动装配协作者有关。

|  |
| --- |
| Bean命名约定 (Bean Naming Conventions) |
| 惯例是在命名bean时使用标准Java约定作为实例字段名称。 也就是说，bean名称以小写字母开头，并从那里开始驼峰。 此类名称的示例包括accountManager，accountService，userDao，loginController等。  命名bean始终使您的配置更易于阅读和理解。 此外，如果您使用Spring AOP，那么在将建议应用于与名称相关的一组bean时，它会有很大帮助。 |

|  |  |
| --- | --- |
|  | 通过类路径中的组件扫描，Spring按照前面描述的规则为未命名的组件生成bean名称：基本上，采用简单的类名并将其初始字符转换为小写。 但是，在（不常见的）特殊情况下，当有多个字符并且第一个和第二个字符都是大写字母时，原始外壳将被保留。 这些规则与java.beans.Introspector.decapitalize（Spring在此处使用）中定义的规则相同。 |

在Bean定义之外别名Bean

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

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

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

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

例如，子系统A的配置元数据可以通过subsystemA-dataSource的名称来引用DataSource。 子系统B的配置元数据可以通过subsystemB-dataSource的名称来引用DataSource。 在编写使用这两个子系统的主应用程序时，主应用程序通过myApp-dataSource的名称引用DataSource。 要使所有三个名称引用同一对象，可以将以下别名定义添加到配置元数据中：

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

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

|  |
| --- |
| Java注解配置Beans (Java-configuration) |
| 如果使用Javaconfiguration，则@Bean注释可用于提供别名。 有关详细信息，请参阅使用@Bean批注。 |

### 1.3.2. 实例化Beans (Instantiating Beans)

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

如果使用基于XML的配置元数据，则指定要在<bean />元素的class属性中实例化的对象的类型（或类）。此类属性（在内部，是BeanDefinition实例上的Class属性）通常是必需的。 （有关例外，请参阅使用实例工厂方法和Bean定义继承进行实例化。）您可以使用以下两种方法之一来使用Class属性：

* 通常，在容器本身通过反向调用其构造函数直接创建bean的情况下指定要构造的bean类，稍微等同于使用new运算符的Java代码。
* 要指定包含为创建对象而调用的静态工厂方法的实际类，在不太常见的情况下，容器在类上调用静态工厂方法来。

|  |
| --- |
| 内部类名 (Inner class names) |
| 如果要为静态嵌套类配置bean定义，则必须使用嵌套类的二进制名称。  例如，如果在com.example包中有一个名为SomeThing的类，并且此SomeThing类具有一个名为OtherThing的静态嵌套类，则bean定义上的class属性值将为com.example.SomeThing $ OtherThing。  请注意，在名称中使用$字符可以将嵌套类名与外部类名分开。 |

#### 使用构造函数实例化（Instantiation with a Constructor）

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

Spring IoC容器几乎可以管理您希望它管理的任何类。它不仅限于管理真正的JavaBeans。大多数Spring用户更喜欢实际的JavaBeans，只有一个默认（无参数）构造函数，并且在容器中的属性之后建模了适当的setter和getter。您还可以在容器中拥有更多异国情调的非bean样式类。例如，如果您需要使用绝对不符合JavaBean规范的旧连接池，那么Spring也可以对其进行管理。

使用基于XML的配置元数据，您可以按如下方式指定bean类：

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

有关为构造函数提供参数的机制（如果需要）以及在构造对象后设置对象实例属性的详细信息，请参阅注入依赖项。

#### 使用静态工厂方法实例化 （Instantiation by Using an Instance Factory Method）

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

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

|  |
| --- |
| <bean id="clientService"  class="examples.ClientService"  factory-method="createInstance"/> |

以下示例显示了一个可以使用前面的bean定义的类：

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

有关在从工厂返回对象后为工厂方法提供（可选）参数和设置对象实例属性的机制的详细信息，请参阅依赖关系和详细配置。

#### 使用实例工厂方法实例化（Instantiation by Using an Instance Factory Method）

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

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

以下示例显示了相应的Java类：

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

一个工厂类也可以包含多个工厂方法，如以下示例所示：

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

以下示例显示了相应的Java类：

|  |
| --- |
| public class DefaultServiceLocator {  private static ClientService clientService = new ClientServiceImpl();  private static AccountService accountService = new AccountServiceImpl();  public ClientService createClientServiceInstance() {  return clientService;  }  public AccountService createAccountServiceInstance() {  return accountService;  }  } |

这种方法表明可以通过依赖注入（DI）来管理和配置工厂bean本身。 请参阅详细信息中的依赖关系和配置。

|  |  |
| --- | --- |
|  | 在Spring文档中，“工厂bean”是指在Spring容器中配置并通过实例或静态工厂方法创建对象的bean。 相比之下，FactoryBean（注意大小写）是指Spring特定的FactoryBean。 |

## 1.4. Bean的依赖 （Dependencies）

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

### 1.4.1. Bean的依赖注入（Dependency Injection）

依赖注入（DI）是一个过程，通过这个过程，对象只能通过构造函数参数，工厂方法的参数或在构造对象实例后在对象实例上设置的属性来定义它们的依赖关系（即，它们使用的其他对象）。从工厂方法返回。然后容器在创建bean时注入这些依赖项。这个过程基本上是bean本身的反向（因此名称，控制反转），通过使用类的直接构造或服务定位器模式来控制其依赖项的实例化或位置。

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

DI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。

#### Bean基于构造函数的依赖注入（Constructor-based Dependency Injection）

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

|  |
| --- |
| public class SimpleMovieLister {  // the SimpleMovieLister has a dependency on a MovieFinder  private MovieFinder movieFinder;  // a constructor so that the Spring container can inject a MovieFinder  public SimpleMovieLister(MovieFinder movieFinder) {  this.movieFinder = movieFinder;  }  // business logic that actually uses the injected MovieFinder is omitted...  } |

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

**构造函数参数解析**

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

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

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

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

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

|  |
| --- |
| package examples;  public class ExampleBean {  // Number of years to calculate the Ultimate Answer  private int years;  // The Answer to Life, the Universe, and Everything  private String ultimateAnswer;  public ExampleBean(int years, String ultimateAnswer) {  this.years = years;  this.ultimateAnswer = ultimateAnswer;  }  } |

**构造函数参数类型匹配**

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

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

**构造函数参数索引**

您可以使用index属性显式指定构造函数参数的索引，如以下示例所示：

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

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

|  |  |
| --- | --- |
|  | 索引从0开始。 |

**构造函数参数名称**

您还可以使用构造函数参数名称进行值消歧，如以下示例所示：

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

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

|  |
| --- |
| package examples;  public class ExampleBean {  // Fields omitted  @ConstructorProperties({"years", "ultimateAnswer"})  public ExampleBean(int years, String ultimateAnswer) {  this.years = years;  this.ultimateAnswer = ultimateAnswer;  }  } |

#### Bean基于Setter的依赖注入（Setter-based Dependency Injection）

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

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

|  |
| --- |
| public class SimpleMovieLister {  // the SimpleMovieLister has a dependency on the MovieFinder  private MovieFinder movieFinder;  // a setter method so that the Spring container can inject a MovieFinder  public void setMovieFinder(MovieFinder movieFinder) {  this.movieFinder = movieFinder;  }  // business logic that actually uses the injected MovieFinder is omitted...  } |

ApplicationContext支持它管理的bean的基于构造函数和基于setter的DI。 在通过构造函数方法注入了一些依赖项之后，它还支持基于setter的DI。 您可以以BeanDefinition的形式配置依赖项，并将其与PropertyEditor实例结合使用，以将属性从一种格式转换为另一种格式。 但是，大多数Spring用户不直接使用这些类（即以编程方式），而是使用XML bean定义，带注释的组件（即使用@Component，@ Controller等注释的类）或@Bean方法。 基于Java的@Configuration类。 然后，这些源在内部转换为BeanDefinition的实例，并用于加载整个Spring IoC容器实例。

|  |
| --- |
| 基于构造函数或基于setter的DI？ (Constructor-based or setter-based DI?) |
| 由于您可以混合基于构造函数和基于setter的DI，因此将构造函数用于强制依赖项和setter方法或可选依赖项的配置方法是一个很好的经验法则。请注意，在setter方法上使用@Required注释可用于使属性成为必需的依赖项。  Spring团队通常提倡构造函数注入，因为它允许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，构造函数注入的组件总是以完全初始化的状态返回到客户端（调用）代码。作为旁注，大量的构造函数参数是一个糟糕的代码气味，暗示该类可能有太多的责任，应该重构以更好地解决关注点的正确分离。  Setter注入应主要仅用于可在类中指定合理默认值的可选依赖项。否则，必须在代码使用依赖项的任何位置执行非空检查。 setter注入的一个好处是setter方法使该类的对象可以在以后重新配置或重新注入。因此，通过JMX MBean进行管理是二次注入的一个引人注目的用例。  使用对特定类最有意义的DI样式。有时，在处理您没有源的第三方类时，会选择您。例如，如果第三方类没有公开任何setter方法，那么构造函数注入可能是唯一可用的DI形式。 |

#### Bean的依赖性解决过程（Dependency Resolution Process）

容器执行bean依赖性解析，如下所示：

* ***ApplicationContext是使用描述所有bean的配置元数据创建和初始化的。配置元数据可以由XML，Java代码或注释指定。***
* ***对于每个bean，它的依赖关系以属性，构造函数参数或static-factory方法的参数的形式表示（如果使用它而不是普通的构造函数）。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。***

Spring容器在创建容器时验证每个bean的配置。但是，在实际创建bean之前，不会设置bean属性本身。创建容器时会创建单例作用域并设置为预先实例化（默认值）的Bean。范围在Bean范围中定义。否则，仅在请求时才创建bean。创建bean可能会导致创建bean的图形，因为bean的依赖关系及其依赖关系（依此类推）被创建和分配。请注意，这些依赖项之间的解决方案不匹配可能会显示较晚 - 也就是说，首次创建受影响的bean时。

|  |
| --- |
| 基于构造函数或基于setter的DI？ (Constructor-based or setter-based DI?) |
| 如果您主要使用构造函数注入，则可以创建无法解析的循环依赖关系场景。  例如：类A通过构造函数注入需要类B的实例，而类B通过构造函数注入需要类A的实例。如果为A类和B类配置bean以便相互注入，则Spring IoC容器会在运行时检测此循环引用，并抛出BeanCurrentlyInCreationException。  一种可能的解决方案是编辑由setter而不是构造函数配置的某些类的源代码。或者，避免构造函数注入并仅使用setter注入。换句话说，尽管不推荐使用，但您可以使用setter注入配置循环依赖关系。  与典型情况（没有循环依赖）不同，bean A和bean B之间的循环依赖强制其中一个bean在完全初始化之前被注入另一个bean（经典的鸡与鸡蛋场景）。 |

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

如果不存在循环依赖，当一个或多个协作bean被注入依赖bean时，每个协作bean在被注入依赖bean之前完全配置。 这意味着，如果bean A依赖于bean B，则Spring IoC容器在调用bean A上的setter方法之前完全配置bean B.换句话说，bean被实例化（如果它不是预先实例化的单例 ），设置其依赖项，并调用相关的生命周期方法（如配置的init方法或InitializingBean回调方法）。

#### Bean的依赖注入的示例(Examples of Dependency Injection)

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

|  |
| --- |
| <bean id="exampleBean" class="examples.ExampleBean">  <!-- setter injection using the nested ref element -->  <property name="beanOne">  <ref bean="anotherExampleBean"/>  </property>  <!-- setter injection using the neater ref attribute -->  <property name="beanTwo" ref="yetAnotherBean"/>  <property name="integerProperty" value="1"/>  </bean>  <bean id="anotherExampleBean" class="examples.AnotherBean"/>  <bean id="yetAnotherBean" class="examples.YetAnotherBean"/> |

以下示例显示了相应的ExampleBean类：

|  |
| --- |
| public class ExampleBean {  private AnotherBean beanOne;  private YetAnotherBean beanTwo;  private int i;  public void setBeanOne(AnotherBean beanOne) {  this.beanOne = beanOne;  }  public void setBeanTwo(YetAnotherBean beanTwo) {  this.beanTwo = beanTwo;  }  public void setIntegerProperty(int i) {  this.i = i;  }  } |

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

|  |
| --- |
| <bean id="exampleBean" class="examples.ExampleBean">  <!-- constructor injection using the nested ref element -->  <constructor-arg>  <ref bean="anotherExampleBean"/>  </constructor-arg>  <!-- constructor injection using the neater ref attribute -->  <constructor-arg ref="yetAnotherBean"/>  <constructor-arg type="int" value="1"/>  </bean>  <bean id="anotherExampleBean" class="examples.AnotherBean"/>  <bean id="yetAnotherBean" class="examples.YetAnotherBean"/> |

以下示例显示了相应的ExampleBean类：

|  |
| --- |
| public class ExampleBean {  private AnotherBean beanOne;  private YetAnotherBean beanTwo;  private int i;  public ExampleBean(  AnotherBean anotherBean, YetAnotherBean yetAnotherBean, int i) {  this.beanOne = anotherBean;  this.beanTwo = yetAnotherBean;  this.i = i;  }  } |

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

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

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

以下示例显示了相应的ExampleBean类：

|  |
| --- |
| public class ExampleBean {  // a private constructor  private ExampleBean(...) {  ...  }  // a static factory method; the arguments to this method can be  // considered the dependencies of the bean that is returned,  // regardless of how those arguments are actually used.  public static ExampleBean createInstance (  AnotherBean anotherBean, YetAnotherBean yetAnotherBean, int i) {  ExampleBean eb = new ExampleBean (...);  // some other operations...  return eb;  }  } |

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

### 1.4.2. Bean的依赖关系和配置细节(Dependencies and Configuration in Detail)

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，或者作为内联定义的值。 Spring的基于XML的配置元数据为此目的支持其<property />和<constructor-arg />元素中的子元素类型。

#### Bean的属性使用直接数据(Straight Values (Primitives, Strings, and so on))

<property />元素的value属性将属性或构造函数参数指定为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String转换为属性或参数的实际类型。 以下示例显示了要设置的各种值：

|  |
| --- |
| <bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">  <!-- results in a setDriverClassName(String) call -->  <property name="driverClassName" value="com.mysql.jdbc.Driver"/>  <property name="url" value="jdbc:mysql://localhost:3306/mydb"/>  <property name="username" value="root"/>  <property name="password" value="masterkaoli"/>  </bean> |

以下示例使用p命名空间进行更简洁的XML配置：

|  |
| --- |
| <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:p="http://www.springframework.org/schema/p"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans.xsd">  <bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"  destroy-method="close"  p:driverClassName="com.mysql.jdbc.Driver"  p:url="jdbc:mysql://localhost:3306/mydb"  p:username="root"  p:password="masterkaoli"/>  </beans> |

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

您还可以配置java.util.Properties实例，如下所示：

|  |
| --- |
| <bean id="mappings"  class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">  <!-- typed as a java.util.Properties -->  <property name="properties">  <value>  jdbc.driver.className=com.mysql.jdbc.Driver  jdbc.url=jdbc:mysql://localhost:3306/mydb  </value>  </property>  </bean> |

Spring容器通过使用JavaBeans PropertyEditor机制将<value />元素内的文本转换为java.util.Properties实例。 这是一个很好的快捷方式，并且是Spring团队支持在value属性样式上使用嵌套<value />元素的少数几个地方之一。

##### 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 - 而不是引用）传递给<constructor-arg />或<property />元素。 以下示例显示了如何使用它：

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

前面的bean定义代码段与以下代码段完全等效（在运行时）：

|  |
| --- |
| <bean id="theTargetBean" class="..." />  <bean id=" client" class="... StraightValuesBean ">  <property name="targetName" value="theTargetBean"/>  </bean>  @Data  public class StraightValuesBean {  public String targetName;  } |

第一种形式优于第二种形式，因为使用idref标记允许容器在部署时验证引用的命名bean实际存在。 在第二个变体中，不对传递给客户端bean的targetName属性的值执行验证。 当客户端bean实际被实例化时，才会发现错别字（最可能是致命的结果）。 如果客户端bean是原型bean，则只能在部署容器后很长时间才能发现此错误和产生的异常。

|  |  |
| --- | --- |
|  | 4.0 beans XSD中不再支持idref元素的local属性，因为它不再提供常规bean引用的值。 升级到4.0架构时，将现有的idref本地引用更改为idref bean。 |

<idref />元素带来值的一个常见位置（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 />元素可以防止拼写错误的拦截器ID。

##### 重点实战

|  |
| --- |
| package com.mfm.learn.spring;  import org.apache.commons.dbcp2.BasicDataSource;  import org.apache.commons.logging.Log;  import org.apache.commons.logging.LogFactory;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import lombok.Data;  @Data  public class StraightValuesBean {  private static final Log LOGGER = LogFactory  .getLog(StraightValuesBean.class);  public String targetName;  public StraightValuesBean() {  }  public StraightValuesBean(String targetName) {  this.targetName = targetName;  }  @SuppressWarnings("resource")  public static void main(String[] args) {  ApplicationContext context = new ClassPathXmlApplicationContext(  "straight-values-bean.xml");  //1.<property />元素和p:命名空间以及使用properties文件注入简单值  BasicDataSource myDataSource1 = context.getBean("myDataSource1",  BasicDataSource.class);  BasicDataSource myDataSource2 = context.getBean("myDataSource2",  BasicDataSource.class);  BasicDataSource myDataSource3 = context.getBean("myDataSource3",  BasicDataSource.class);  StraightValuesBean.LOGGER.info(myDataSource1.getUrl());  StraightValuesBean.LOGGER.info(myDataSource2.getUrl());  StraightValuesBean.LOGGER.info(myDataSource3.getUrl());  //2.idref元素使用  StraightValuesBean theClientBean = context.getBean("theClientBean",  StraightValuesBean.class);  //当需要把bean名称注入到某个bean中时候，这种方式很好，会校验bean是否实际存在，如果不存在会报错的，比如myDataSource3改为myDataSource333未运行程序就会报错  StraightValuesBean.LOGGER.info(theClientBean.getTargetName());  //等价于  StraightValuesBean theClientBean2 = context.getBean("theClientBean2",  StraightValuesBean.class);  //当需要把bean名称注入到某个bean中时候，这种方式不会校验，就算写成myDataSource333也可以正常输出  StraightValuesBean.LOGGER.info(theClientBean2.getTargetName());  //应用：<idref />元素带来值的一个常见位置（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 />元素可以防止拼写错误的拦截器ID。  }  } |

straight-values-bean.xml

|  |
| --- |
| <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:p="http://www.springframework.org/schema/p"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans.xsd">    <bean id="myDataSource1" class="org.apache.commons.dbcp2.BasicDataSource" destroy-method="close">  <!-- results in a setDriverClassName(String) call -->  <property name="driverClassName" value="com.mysql.jdbc.Driver"/>  <property name="url" value="jdbc:mysql://10.255.33.108:3306/keplerdb"/>  <property name="username" value="root"/>  <property name="password" value="123456"/>  </bean>  <bean id="myDataSource2" class="org.apache.commons.dbcp2.BasicDataSource"  destroy-method="close"  p:driverClassName="com.mysql.jdbc.Driver"  p:url="jdbc:mysql://10.255.33.108:3306/keplerdb"  p:username="root"  p:password="123456"  />  <bean id="myDataSource3" class="org.apache.commons.dbcp2.BasicDataSource"  destroy-method="close"  p:driverClassName="${jdbc.driver.className}"  p:url="${jdbc.url}"  p:username="root"  p:password="123456"  />  <bean id="mappings"  class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">  <!-- typed as a java.util.Properties -->  <!--  <property name="locations" value="classpath:/resources/datasource.properties" /> 可将key=vaule配置到properties文件中  <property name="placeholderPrefix" value="${"></property> 默认  <property name="placeholderSuffix" value="}"></property> 默认  -->  <property name="properties">  <value>  jdbc.driver.className=com.mysql.jdbc.Driver  jdbc.url=jdbc:mysql://10.255.33.108:3306/keplerdb  </value>  </property>  </bean>      <bean id="theClientBean" class="com.mfm.learn.spring.StraightValuesBean">  <!-- <property name="targetName">  <idref bean="myDataSource3"/>  </property> -->  <constructor-arg>  <idref bean="myDataSource3"/>  </constructor-arg>  </bean>  <bean id="theClientBean2" class="com.mfm.learn.spring.StraightValuesBean">  <property name="targetName" value="myDataSource33" />  </bean>    </beans> |

#### 关联其他Bean (References to Other Beans (Collaborators) )

ref元素是<constructor-arg />或<property />定义元素中的最后一个元素。在这里，您将bean的指定属性的值设置为对容器管理的另一个bean（协作者）的引用。引用的bean是要设置其属性的bean的依赖项，并且在设置该属性之前根据需要对其进行初始化。 （如果协作者是单例bean，它可能已经被容器初始化。）所有引用最终都是对另一个对象的引用。范围和验证取决于您是通过bean，local或parent属性指定其他对象的ID还是名称。

通过<ref />标记的bean属性指定目标bean是最常用的形式，并允许创建对同一容器或父容器中的任何bean的引用，而不管它是否在同一XML文件中。 bean属性的值可以与目标bean的id属性相同，或者与目标bean的name属性中的值之一相同。以下示例显示如何使用ref元素：

|  |
| --- |
| <ref bean="someBean"/> |

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

|  |
| --- |
| <!-- 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> | | |
|  | 4.0 beans XSD不再支持ref元素的local属性，因为它不再提供常规bean引用的值。 升级到4.0架构时，将现有的ref本地引用更改为ref bean。 |

#### 内部Beans（Inner Beans）

<property />或<constructor-arg />元素中的<bean />元素定义了内部bean，如下例所示：

|  |
| --- |
| <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，则容器不使用此类值作为标识符。 容器还会在创建时忽略scope标志，因为内部bean始终是匿名的，并且始终使用外部bean创建。 不可能独立访问内部bean或将它们注入协作bean而不是封闭bean。

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

#### Bean的集合属性配置（Collections）

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

|  |
| --- |
| <bean id="moreComplexObject" class="example.ComplexObject">  <!-- results in a setAdminEmails(java.util.Properties) call -->  <property name="adminEmails">  <props>  <prop key="administrator">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何元素：

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

**Collection Merging**

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

关于合并的这一部分讨论了父子bean机制。 不熟悉父母和子bean定义的读者可能希望在继续之前阅读相关部分。

以下示例演示了集合合并：

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

请注意在子bean定义的adminEmails属性的<props />元素上使用merge = true属性。 当容器解析并实例化子bean时，生成的实例有一个adminEmails属性集合，其中包含将子项的adminEmails集合与父项的adminEmails集合合并的结果。 以下清单显示了结果：

administrator=administrator@example.com

sales=sales@example.com

support=support@example.co.uk

子属性集合的值集继承父<props />的所有属性元素，子值的支持值覆盖父集合中的值。

此合并行为同样适用于<list />，<map />和<set />集合类型。 在<list />元素的特定情况下，保持与List集合类型相关联的语义（即，有序的值集合的概念）。 父级的值位于所有子级列表的值之前。 对于Map，Set和Properties集合类型，不存在排序。 因此，对于作为容器内部使用的关联Map，Set和Properties实现类型的基础的集合类型，没有排序语义有效。

**集合合并的局限性**

您无法合并不同的集合类型（例如Map和List）。 如果您尝试这样做，则会抛出相应的异常。 必须在较低的继承子定义上指定merge属性。 在父集合定义上指定merge属性是多余的，并且不会导致所需的合并。

**带泛型的集合**

通过在Java 5中引入泛型类型，您可以使用强类型集合。 也就是说，可以声明Collection类型，使其只能包含（例如）String元素。 如果使用Spring将强类型集合依赖注入到bean中，则可以利用Spring的类型转换支持，以便强类型Collection实例的元素在添加到类型之前转换为适当的类型。 采集。 以下Java类和bean定义显示了如何执行此操作：

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

|  |  |
| --- | --- |
| <beans>  <bean id="something" class="x.y.SomeClass">  <property name="accounts">  <map>  <entry key="one" value="9.99"/>  <entry key="two" value="2.75"/>  <entry key="six" value="3.99"/>  </map>  </property>  </bean>  </beans> | public class SomeClass {  private Map<String, Float> accounts;  public void setAccounts(Map<String, Float> accounts) {  this.accounts = accounts;  }  } |

当bean的accounts属性准备好进行注入时，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。

#### Bean的空值和NULL属性配置(Null and Empty String Values)

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

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

上面的示例等效于以下Java代码：

|  |
| --- |
| exampleBean.setEmail(""); |

该<null/>元素处理null的值。以下清单显示了一个示例：

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

上述配置等同于以下Java代码：

|  |
| --- |
| exampleBean.setEmail(null); |

#### 带有p命名空间的XML快捷方式（XML Shortcut with the p-namespace）

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

Spring支持具有命名空间的可扩展配置格式，这些命名空间基于XML Schema定义。beans本章中讨论的配置格式在XML Schema文档中定义。但是，p-namespace未在XSD文件中定义，仅存在于Spring的核心中。

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

|  |
| --- |
| <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:p="http://www.springframework.org/schema/p"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans.xsd">  <bean name="classic" class="com.example.ExampleBean">  <property name="email" value="someone@somewhere.com"/>  </bean>  <bean name="p-namespace" class="com.example.ExampleBean"  p:email="someone@somewhere.com"/>  </beans> |

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

下一个示例包括另外两个bean定义，它们都引用了另一个bean：

|  |
| --- |
| <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:p="http://www.springframework.org/schema/p"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans.xsd">  <bean name="john-classic" class="com.example.Person">  <property name="name" value="John Doe"/>  <property name="spouse" ref="jane"/>  </bean>  <bean name="john-modern"  class="com.example.Person"  p:name="John Doe"  **p:spouse-ref**="jane"/>  <bean name="jane" class="com.example.Person">  <property name="name" value="Jane Doe"/>  </bean>  </beans> |

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

|  |  |
| --- | --- |
|  | 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结束的属性冲突Ref，而标准XML格式则不然。我们建议您仔细选择您的方法并将其传达给您的团队成员，以避免生成同时使用所有三种方法的XML文档。 |

#### 带有c命名空间的XML快捷方式(XML Shortcut with the c-namespace)

与带有p-namespace的XML Shortcut类似，Spring 3.1中引入的c-namespace允许使用内联属性来配置构造函数参数，而不是嵌套的constructor-arg元素。

以下示例使用c：namespace来执行与基于Constructor的依赖注入相同的操作：

|  |
| --- |
| <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:c="http://www.springframework.org/schema/c"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans.xsd">  <bean id="thingOne" class="x.y.ThingTwo"/>  <bean id="thingTwo" class="x.y.ThingThree"/>  <!-- traditional declaration -->  <bean id="thingOne" class="x.y.ThingOne">  <constructor-arg ref="thingTwo"/>  <constructor-arg ref="thingThree"/>  <constructor-arg value="something@somewhere.com"/>  </bean>  <!-- c-namespace declaration -->  <bean id="thingOne" class="x.y.ThingOne" c:thingTwo-ref="thingTwo" c:thingThree-ref="thingThree" c:email="something@somewhere.com"/>  </beans> |

该c:命名空间使用相同的约定作为p:一个（尾部-ref的bean引用），供他们的名字设置构造函数的参数。类似地，它需要声明，即使它没有在XSD架构中定义（它存在于Spring核心内）。

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

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

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

实际上，构造函数解析 机制在匹配参数方面非常有效，因此除非您确实需要，否则我们建议在整个配置中使用名称表示法。

#### 复合属性名称(Compound Property Names)

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

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

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

### 1.4.3. Bean的依赖控制（Using depends-on）

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

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

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

|  |
| --- |
| <bean id="beanOne" class="ExampleBean" depends-on="manager,accountDao">  <property name="manager" ref="manager" />  </bean>  <bean id="manager" class="ManagerBean" />  <bean id="accountDao" class="x.y.jdbc.JdbcAccountDao" /> |

|  |  |
| --- | --- |
|  | 该depends-on属性既可以指定初始化时间依赖性，也可以指定单独的 bean，相应的销毁时间依赖性。depends-on在给定的bean本身被销毁之前，首先销毁定义与给定bean 的关系的从属bean 。这样，depends-on也可以控制销毁顺序。 |

### 1.4.4. 懒初始化Bean（Lazy-initialized Beans）

默认情况下，ApplicationContext实现会急切地创建和配置所有 单例 bean作为初始化过程的一部分。通常，这种预先实例化是可取的，因为配置或周围环境中的错误是立即发现的，而不是几小时甚至几天后。如果不希望出现这种情况，可以通过将bean定义标记为延迟初始化来阻止单例bean的预实例化。延迟初始化的bean告诉IoC容器在第一次请求时创建bean实例，而不是在启动时创建。

在XML中，此行为由 元素lazy-init上的属性控制<bean/>，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"/> |

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

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

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

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

### 1.4.5. Bean的自动装配与协作（Autowiring Collaborators）

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

自动装配可以显着减少指定属性或构造函数参数的需要。（在本章其他地方讨论的其他机制，如bean模板 ，在这方面也很有价值。）

自动装配可以随着对象的发展更新配置。例如，如果需要向类添加依赖项，则可以自动满足该依赖项，而无需修改配置。因此，自动装配在开发期间尤其有用，而不会在代码库变得更稳定时否定切换到显式布线的选项。

使用基于XML的配置元数据（请参阅依赖注入）时，可以使用元素的autowire属性为 bean定义指定autowire模式<bean/>。自动装配功能有四种模式。您指定每个bean的自动装配，因此可以选择要自动装配的那些。下表描述了四种自动装配模式：

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

使用byType或constructor自动装配模式，您可以连接阵列和键入的集合。在这种情况下，提供容器内与预期类型匹配的所有autowire候选者以满足依赖性。Map如果预期的键类型是，则可以自动装配强类型实例String。自动装配Map 实例的值由与预期类型匹配的所有bean实例组成， Map实例的键包含相应的bean名称。

自动装配的局限和缺点

自动装配在项目中一致使用时效果最佳。如果一般不使用自动装配，那么开发人员使用它来连接一个或两个bean定义可能会让人感到困惑。

#### 考虑自动装配的局限和缺点：

* 显式依赖项property和constructor-arg设置始终覆盖自动装配。您不能自动装配简单属性，例如基元 Strings，和Classes（以及此类简单属性的数组）。这种限制是按设计的。
* 自动装配不如显式布线精确。虽然如前面的表中所述，Spring谨慎地避免在可能产生意外结果的模糊性的情况下进行猜测。您不再明确记录Spring管理对象之间的关系。
* 可能无法为可能从Spring容器生成文档的工具提供接线信息。
* 容器中的多个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，这不一定是个问题。但是，对于期望单个值的依赖关系，这种模糊性不是任意解决的。如果没有可用的唯一bean定义，则抛出异常。

在后一种情况下，您有几种选择：

* 放弃自动装配以支持显式布线。
* 通过将其autowire-candidate属性设置为bean，避免对bean定义进行自动装配false，如下一节所述。
* 通过将primary其<bean/>元素的属性设置为，将单个bean定义指定为主要候选者 true。
* 实现基于注释的配置可用的更细粒度控制，如基于注释的容器配置中所述。

#### 从自动装配中排除Bean

在每个bean的基础上，您可以从自动装配中排除bean。在Spring的XML格式中，将元素的autowire-candidate属性设置<bean/>为false。容器使特定的bean定义对自动装配基础结构不可用（包括注释样式配置等@Autowired）。

|  |  |
| --- | --- |
|  | 该autowire-candidate属性旨在仅影响基于类型的自动装配。它不会影响名称的显式引用，即使指定的bean未标记为autowire候选，也会解析它。因此，如果名称匹配，则按名称自动装配会注入bean。 |

您还可以根据针对bean名称的模式匹配来限制autowire候选者。顶级<beans/>元素在其default-autowire-candidates属性中接受一个或多个模式 。例如，要将autowire候选状态限制为名称以其结尾的任何bean Repository，请提供值\*Repository。要提供多个模式，请在逗号分隔的列表中定义它们。bean定义的属性的显式值 true或优先级始终优先。对于此类bean，模式匹配规则不适用。falseautowire-candidate

这些技术对于您永远不希望通过自动装配注入其他bean的bean非常有用。这并不意味着使用自动装配本身不能配置排除的bean。相反，bean本身不是自动装配其他bean的候选者。

### 1.4.6.方法注入(Method Injection)

在大多数应用程序场景中，容器中的大多数bean都是 单例。当单例bean需要与另一个单例bean协作或非单例bean需要与另一个非单例bean协作时，通常通过将一个bean定义为另一个bean的属性来处理依赖关系。当bean生命周期不同时会出现问题。假设单例bean A需要使用非单例（原型）bean B，可能是在A上的每个方法调用上。容器只创建一次单独的bean A，因此只有一次机会来设置属性。每次需要时，容器都不能为bean A提供bean B的新实例。

解决方案是放弃一些控制反转。你可以做一个豆意识到容器通过实现ApplicationContextAware接口，并通过制作getBean("B")到容器调用请求（典型新）bean B实例的实例每次豆A需要它。以下示例显示了此方法：

A solution is to forego some inversion of control. You can make bean A aware of the container by implementing the ApplicationContextAware interface, and by making a getBean("B") call to the container ask for (a typically new) bean B instance every time bean A needs it. The following example shows this approach:

|  |
| --- |
| // a class that uses a stateful Command-style class to perform some processing  package fiona.apple;  // Spring-API imports  import org.springframework.beans.BeansException;  import org.springframework.context.ApplicationContext;  import org.springframework.context.ApplicationContextAware;  public class CommandManager implements ApplicationContextAware {  private ApplicationContext applicationContext;  public Object process(Map commandState) {  // grab a new instance of the appropriate Command  Command command = createCommand();  // set the state on the (hopefully brand new) Command instance  command.setState(commandState);  return command.execute();  }  protected Command createCommand() {  // notice the Spring API dependency!  return this.applicationContext.getBean("command", Command.class);  }  public void setApplicationContext(  ApplicationContext applicationContext) throws BeansException {  this.applicationContext = applicationContext;  }  } |

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

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

##### 查找方法注入(Lookup Method Injection)

Lookup方法注入是容器覆盖容器管理bean上的方法并返回容器中另一个命名bean的查找结果的能力。查找通常涉及原型bean，如上一节中描述的场景。Spring Framework通过使用CGLIB库中的字节码生成来动态生成覆盖该方法的子类来实现此方法注入。

|  |  |
| --- | --- |
|  | * 要使这个动态子类工作，Spring bean容器子类不能成为的类final，以及要重写的方法也不能final。 * 对具有abstract方法的类进行单元测试需要您自己对类进行子类化并提供该abstract方法的存根实现。 * 组件扫描也需要具体的方法，这需要具体的类来获取。 * 另一个关键限制是查找方法不适用于工厂方法，特别是@Bean配置类中的方法，因为在这种情况下，容器不负责创建实例，因此无法创建运行时生成的子类苍蝇 |

对于CommandManager前面代码片段中的类，Spring容器动态地覆盖createCommand() 方法的实现。该CommandManager班没有任何Spring的依赖，因为返工例所示：

|  |
| --- |
| package fiona.apple;  // no more Spring imports!  public abstract class CommandManager {  public Object process(Object commandState) {  // grab a new instance of the appropriate Command interface  Command command = createCommand();  // set the state on the (hopefully brand new) Command instance  command.setState(commandState);  return command.execute();  }  // okay... but where is the implementation of this method?  protected abstract Command createCommand();  } |

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

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

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

|  |
| --- |
| <!-- a stateful bean deployed as a prototype (non-singleton) -->  <bean id="myCommand" class="fiona.apple.AsyncCommand" scope="prototype">  <!-- inject dependencies here as required -->  </bean>  <!-- commandProcessor uses statefulCommandHelper -->  <bean id="commandManager" class="fiona.apple.CommandManager">  <lookup-method name="createCommand" bean="myCommand"/>  </bean> |

只要需要bean 的新实例，标识为bean的bean 就会commandManager调用自己的createCommand()方法myCommand。myCommand如果实际需要，您必须小心将bean 部署为原型。如果是单例，myCommand 则每次都返回相同的bean 实例。

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

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

或者，更具惯用性，您可以依赖于针对查找方法的声明返回类型解析目标bean：

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

|  |
| --- |
|  |

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

|  |  |
| --- | --- |
|  | 访问不同范围的目标bean的另一种方法是ObjectFactory/ Provider注入点。将Scoped Beans视为依赖关系。  您也可能会发现ServiceLocatorFactoryBean（在 org.springframework.beans.factory.config包中）有用。 |

##### 任意方法替换(Arbitrary Method Replacement)

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

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

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

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

|  |
| --- |
| /\*\*  \* meant to be used to override the existing computeValue(String)  \* implementation in MyValueCalculator  \*/  public class ReplacementComputeValue implements MethodReplacer {  public Object reimplement(Object o, Method m, Object[] args) throws Throwable {  // get the input value, work with it, and return a computed result  String input = (String) args[0];  ...  return ...;  }  } |

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"/> |

您可以使用<arg-type/>元素中的一个或多个元素<replaced-method/> 来指示被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，才需要参数的签名。为方便起见，参数的类型字符串可以是完全限定类型名称的子字符串。例如，以下所有匹配 java.lang.String：

|  |
| --- |
| java.lang.String  String  Str |

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