# 1. Spring介绍和下载

## 1.1 Spring概述

Spring是Java开发人员必须掌握的框架，因为Spring基本上提供了Java企业级项目的一整套解决方案，Spring主要的模块有：

（1）Spring Core：核心功能，如提供IoC容器，处理对象的创建和对象之间的依赖关系；

（2）Spring Web：Spring对Web支持的模块，如提供Spring和Struts整合的功能。以后还会学习其中的SpringMVC框架，这是Spring提供的基于MVC设计模式的框架，现在一般使用SpringMVC而非Struts。

（3）Spring AOP：Spring提供的面向切面编程AOP。

除此之外，Spring还提供了ORM、JDBC等的支持。

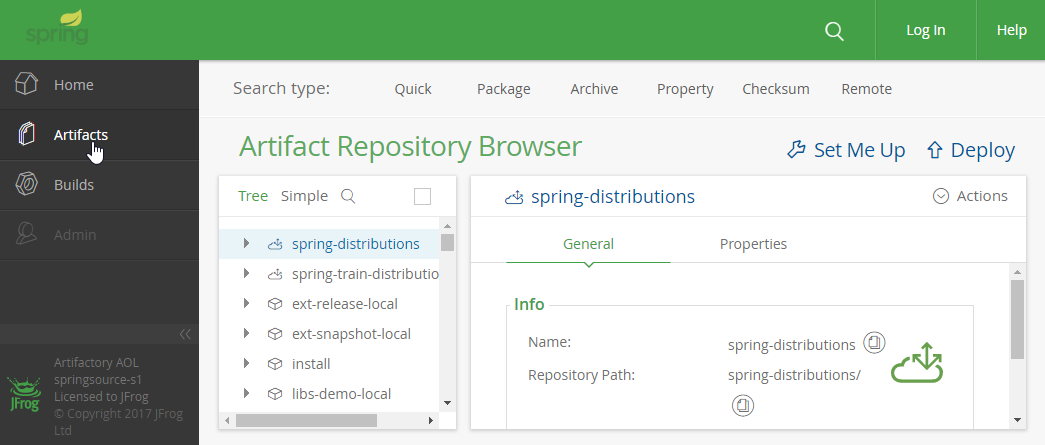
没有Spring之前，Java企业级项目的开发遵循Java EE的制定的规范，即大型网站开发使用EJB规范。但是EJB非常繁杂，这催生了Spring框架。现在使用Spring 框架非常广泛，甚至评论“Spring是Java EE的事实标准”。

掌握Spring非常重要，Spring是一个改善架构的一个框架。Spring的目的是为了简化企业级开发。并且Spring是一个非侵入性的框架，在实际使用时，你甚至可以不需要了解Spring的任何API，但项目确确实实运行在Spring容器上。

读者对上述的说明可能还是似懂非懂，可在以后的讲解中慢慢体会，总之Spring这一套东西值得研究。

## 1.2 下载Spring框架

在Spring官网上没有Spring下载地址，需要导到Spring的官方仓库上下载。网址：<https://repo.spring.io/webapp/#/home> 。首先展开页面左侧的导航栏，点击“Artifacts”，此时右边就会显示树状结构目录。比如：



然后，依次选择节点：“libs-release-local” - “org” - “springframework” - “spring” - “选择版本（这里我们选择使用较新的5.0.2.RELEASE版本）”。这样，选中其中的“spring-framework-4.3.7.RELEASE-dist.zip”文件后，就能在页面的右上角点击“Download”下载Spring的发布包了。该包基本包含了Spring的所有官方资源（包括文档、参考教程等）。

下载后解压此包，在“docs/spring-framework-reference”目录就提供了参考文档。其中提供了不同格式的文档，包含PDF和HTML的文档。

终归，还是应该学好英文，看好官方文档，才是真正的深入研究和理解。

# 2. Spring的核心——IoC容器

## 2.1 基本概念

Spring的核心功能就是创建对象并维护对象之间的依赖关系。而这些功能的实现是通过IOC容器实现的。

为何要用Spring创建对象并维护关系？这就涉及到控制反转（Inversion of Control，即IoC）。

“反转”是相对于“正向”而言的。以前，我们要在一个类中（如A类）使用一个B类的对象时，是通过new关键字自己创建出来的，这就是常规的“正向”情况。什么是反向呢？就是A类不再主动获取B实例，而是等待IOC容器获取一个B实例，然后容器将实例反向地注入到A类中以提供使用。

即控制反转（IoC）的意思就是将对象的创建交由外部容器来创建，不再自己创建所需对象。这个注入的过程就是对程序中所用到的依赖的注入过程，因此也叫作依赖注入（Dependency Injection），依赖注入就用来处理对象之间的依赖关系。

其实依赖注入和控制反转是对同一件事情的不同描述，即它们描述的角度不同。依赖注入是从应用程序的角度描述：应用程序依赖容器创建并注入它所需要的外部资源；而控制反转是从容器的角度在描述：容器控制应用程序，由容器反向的向应用程序注入应用程序所需要的外部资源。

那么到底为何要使用IoC/DI呢？

IoC/DI对编程带来的改变不仅体现在代码上，更是体现在思想上，使程序开发发生了“主从换位”的变化。原本，应用程序是“主动出击”获取所需资源，但在IoC/DI思想中，应用程序变成被动等待容器来创建并注入它所需要的资源。

这样的改变其实是编程思想的一个大进步，能有效的分离对象和它所需要的外部资源，降低程序的耦合性，有利于功能复用。更重要的是，这使得程序的体系结构变得非常灵活。

【注：上述描述参见网址：<http://blog.csdn.net/doris_crazy/article/details/18353197> 】

比如在项目中，BaseServiceImpl这个接口实现类需要调用DAO层对象，以往我们使用“private IBaseDao baseDao = new BaseDaoImpl();”来获得DAO层的实现类对象，其实这就产生了耦合，因为代码中要指明具体的DAO层实现类对象，也导致项目不灵活。而采用了IoC/DI后，BaseServiceImpl中只要定义下成员“private IBaseDao baseDao;”即可，至于具体对象的创建交由我们配置的IOC容器对象来完成，这样明显降低了耦合性，增加了灵活性，并且容器自身创建的对象有利于复用。

总之，利用IoC，可以把对象的创建、初始化和销毁等交给Spring管理。这里再举个例子说明这样做的好处。普通人“找对象”可以有两种方式，一种是自己在微信找对象，也可以上非诚勿扰找对象。而第二种上非诚勿扰的方式就是相当于是Spring的方式，找对象这种事别人已经帮你做好了，你只要选择即可，而不需自己花时间找。也就是说后一种方式让你的工作重心发生了转义，更加专注“找对象”。

## 2.2 Spring第一个案例——用IoC创建对象

首先新建项目，引入引入Spring的jar包。Spring的jar包在libs目录下。可以发现，Spring中每个模块的jar包都提供了三个jar包，一个是可直接使用的jar包，一个是Javadoc文档jar包，还有一个是源码jar包。我们只要导入可直接使用的jar包即可。在本例中，需要导入以下jar包，这也是Spring的基础的jar包：

|  |
| --- |
| spring-aop-5.0.2.RELEASE.jar  spring-beans-5.0.2.RELEASE.jar  spring-context-5.0.2.RELEASE.jar  spring-core-5.0.2.RELEASE.jar  spring-expression-5.0.2.RELEASE.jar |

除此之外，Spring的运行还需要一个Apache-Commons中的日志组件包“commons-logging.jar”，这在Spring的lib目录是没有提供的，需要自行下载引入。

搭建好环境（Java SE项目即可）后，就可以写第一个Spring案例了。我们用该案例实现从IoC容器中拿到一个我们配置的、指示Spring生产的一个实例对象。

（1）首先创建一个Student类：

|  |
| --- |
| **package** com.sd;  **public class** Student {  *// 默认构造* **public** Student() { }   *// 提供一个sayHello方法* **public void** sayHello() {  System.***out***.println(**"I am a student."**);  } } |

（2）在src下创建一个名为applicationContext.xml的Spring配置文件（文件名可以自定义）。该配置文件用于指示Spring可以创建的bean。

该文件的根元素是beans，其中可包括一系列的bean。那么什么是bean呢？Spring中，把一个类放到Spring容器中，这个类就叫做bean。下面是该配置文件的内容，其实就是配置bean：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"**>  *<!--  使用bean标签描述一个类。id属性是这个bean的唯一标识；class是这个bean的全限定类名  -->* <**bean id="student" class="com.sd.Student"** /> </**beans**> |

（3）在主类中通过Spring容器获得bean的对象：

|  |
| --- |
| **public static void** main(String[] args) {  *// ApplicationContext是Spring的总容器。通过它可加载配置文件启动Spring容器。* ApplicationContext ac = **new** ClassPathXmlApplicationContext(**"applicationContext.xml"**);  *// 用ac的getBean()方法，从容器中根据id提取一个这个bean的对象* Student student = (Student) ac.getBean(**"student"**);  *// 调用对象的方法* student.sayHello(); } |

ApplicationContext是一个接口，ClassPathXmlApplicationContext是它的一个实现类。ClassPathXmlApplicationContext可加载classpath路径下的指定配置文件来启动Spring容器，上述中我们就是指定加载了applicationContext.xml文件，当然也可指定加载别的文件。如果配置文件在com.sd包下，且文件名为beans.xml，则代码就写为：

|  |
| --- |
| ApplicationContext ac = **new** ClassPathXmlApplicationContext(**"com/sd/beans.xml"**); |

通过第一个例子读者要体会到对象是Spring创建的，不是自己new出来的。

# 3. 详解对象的创建

## 3.1 Spring创建对象的方式

Spring中，既可以使用构造器来创建对象，也能使用工厂类创建对象。

（1）用构造器创建对象。第一个例子中就是使用构造器创建对象的。Spring容器默认使用无参构造创建对象。可以在Student无参构造中执行一些输出，这样Spring容器创建Student对象时会执行构造器中的输出。这证明了Spring默认是调用无参构造完成对象的创建的

（2）用工厂类创建对象，这里又分为：静态工厂方法创建对象和实例工厂方法创建对象。

例如，Student学生工厂类中，分别提供了静态方法和实例方法得到创建好的Student对象：

|  |
| --- |
| **public class** StudentFactory {  *// 静态方法* **public static** Student getStaticInstance() {  **return new** Student();  }  *// 实例方法* **public** Student getInstance() {  **return new** Student();  } } |

这时，applicationContext.xml中要这样配置：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"**>  *<!-- 用静态方法创建对象。class为工厂类，用factory-method指明静态方法 -->* <**bean id="staticStudent" class="com.sd.StudentFactory" factory-method="getStaticInstance"** />   *<!-- 下面配置实例工厂方法创建对象。使用实例方法先要有对象，因此先配置一个该工厂的bean -->  <!-- 配置工厂实例 -->* <**bean id="studentFactory" class="com.sd.StudentFactory"** />  *<!-- 配置实例方法。factory-bean指向上述的工厂bean，factory-method指明使用的实例方法 -->* <**bean id="instanceStudent" factory-bean="studentFactory" factory-method="getInstance"** /> </**beans**> |

上述分别配置了Spring分别如何得到Student对象。Java中的代码如下：

|  |
| --- |
| **public class** Demo {  **public static void** main(String[] args) {  ClassPathXmlApplicationContext ac = **new** ClassPathXmlApplicationContext(**"applicationContext.xml"**);  *// 用工厂静态方法得到对象* Student stu1 = (Student) ac.getBean(**"staticStudent"**);  System.***out***.println(stu1);  *// 用工厂实例方法得到对象* Student stu2 = (Student) ac.getBean(**"instanceStudent"**);  System.***out***.println(stu2);  } } |

上述就是通过配置工厂方法进而拿到对象。实际上这就是容器内部调用了工厂的方法，本质上这种方式不是Spring创建了对象，而是工厂中创建了对象，只不过被Spring容器调用了。

在实际使用中，我们大多数情况都是利用Spring通过构造器创建对象的方式来从Spring中拿到对象的，当然也不免有少数情况是通过让Spring调用工厂方法的方式来拿到对象的。这个要根据具体情况做。

## 3.2 使用别名alias

在配置文件中可以为bean设置别名，例如：

|  |
| --- |
| *<?***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="student" class="com.sd.Student"** />  *<!-- 给上面的student设置别名。使用alias标签  name属性值要和需要设置别名的bean的id一致  alias属性就是要设置的别名  -->* <**alias name="student" alias="stu"** /> </**beans**> |

这样就可以直接通过别名拿到对象：

|  |
| --- |
| **public static void** main(String[] args) {  ApplicationContext ac = **new** ClassPathXmlApplicationContext(**"applicationContext.xml"**);  Student stu = (Student) ac.getBean(**"stu"**);  Student stud = (Student) ac.getBean(**"stud"**);  stu.sayHello();  stud.sayHello(); } |

## 3.3 单例和多例的控制

在bean节点中，可通过scope属性设置Spring创建对象的对象是单实例的还是多实例的。

如果scope属性值可为singleton或prototype。该属性值默认就是“singleton”，表示对象是单例的，即Spring容器中只会创建该bean的一个对象。例如，无论调用多少次getBean方法，拿到的都是这个对象。例如：

|  |
| --- |
| **public static void** main(String[] args) {  ApplicationContext ac = **new** ClassPathXmlApplicationContext(**"applicationContext.xml"**);  Student student1 = (Student) ac.getBean(**"student"**);  Student student2 = (Student) ac.getBean(**"student"**);  System.***out***.println(student1 == student2); *// 输出true，表示是同一对象* } |

注意，这里的单例和以前讲的“单例模式”是不太一样的。“单例模式”是指在一个运行的JVM中始终只有这个类的一个实例存在（通过代码控制死了），而Spring中配置的单例，只是意味着通过getBean方法根据这个bean的id拿到的对象是同一个对象。若在Spring中配置了此类的多个单例bean，那么分别根据id拿到对象，它们也是不相等的。例如配置：

|  |
| --- |
| <**bean id="person1" scope="singleton" class="com.zhang.test.Person"** /> <**bean id="person2" scope="singleton" class="com.zhang.test.Person"** /> |

这样在程序中分别拿到对象，它们不相等：

|  |
| --- |
| **public static void** main(String[] args) {  ApplicationContext ac = **new** ClassPathXmlApplicationContext(**"applicationContext.xml"**);  Student student1 = (Student) ac.getBean(**"student1"**);  Student student2 = (Student) ac.getBean(**"student2"**);  System.***out***.println(student1 == student2); *// 输出false，不是同一对象* } |

和使用singleton相反，当scope属性值设置为“prototype（原型）”时，表示Spring创建对象的方式是多例的，即每次通过该bean的id拿到的实例都是不同的。如配置：

|  |
| --- |
| <**bean id="student" class="com.sd.Student" scope="prototype"** /> |

程序每次拿到的对象都不同：

|  |
| --- |
| **public static void** main(String[] args) {  ApplicationContext ac = **new** ClassPathXmlApplicationContext(**"applicationContext.xml"**);  Student student1 = (Student) ac.getBean(**"student"**);  Student student2 = (Student) ac.getBean(**"student"**);  System.***out***.println(student1 == student2); *// 输出false，不是同一对象* } |

单实例和多实例的区别和注意点：

如果对象是单实例的，那么在该类上声明的属性将会被全局共享。这样可能导致线程安全问题。使用单实例时，JVM中管理的对象较少，有助于提升性能。

如果对象是多实例的，属性就不会有上述的线程安全问题，因为每次拿到对象都是新的。但是频繁地创建和销毁对象会导致JVM性能下降。

以后，在进行WEB框架的整合时，我们会把对象的创建交由Spring管理。此时，我们一般设置DAO和Service对象是单例的，这是为了尽量减少对象的创建。而Struts的Action设置为多例的，这是因为Struts框架中，Action本身就是多例的，每次请求都会产生新的Action对象。而DAO和Service是开发者自行编写的，设置为单例后只要编程时注意点，是没有问题的，如果DAO和Service也设置成多例，那么每有一个Action请求，就会创建Service和DAO对象，消耗较大。

## 3.4 Spring创建对象的时机

Spring会在何时创建对象呢？

对于默认的单例bean，Spring会在容器启动时就创建好一个该bean的对象，后续getBean()方法会直接拿到这个对象，因此多次拿到的对象都是相等的。

而对于多例bean，Spring会在每次getBean()方法被调用时，才创建需要的bean以供使用。因此每次拿到的对象是不同的。

上述的说法都可以通过断点调试和默认构造器的输出进行验证。可自行做。

在bean中，可通过lazy-init属性设置单例对象的初始化时机。lazy-init就是用于设置单例对象的延迟加载（即用到时再创建对象）。lazy-init默认是false，当设置为true时就是延迟加载：

|  |
| --- |
| *<!-- 延迟加载单例bean -->* <**bean id="student" class="com.sd.Student" lazy-init="true"**/> |

此时，单例对象不是在容器启动时创建，而是和多例bean一样，在调用getBean()方法时才创建对象：

|  |
| --- |
| **public static void** main(String[] args) {  ApplicationContext ac = **new** ClassPathXmlApplicationContext(**"applicationContext.xml"**); *// 此时不创建对象* Student student = (Student) ac.getBean(**"student"**); *// 此时才创建单例对象* } |

一般，使用单例bean时不设置“懒加载”，因为容器启动时就初始化对象既可以在初始化时就检查Spring配置文件有无出错，又可以事先创建好对象以供使用，提高效率。

需要注意，init-lazy属性的设置对多例bean是没有效果的，多例对象的创建总是在getBean()方法被调用时才创建对象。这是因为容器并不知道应用需要多少个多例对象。

另外，Spring容器支持在一个对象创建之后和销毁之前，执行指定的方法。通过下面属性配置：

通过设置 init\_method="方法名" 来指定对象创建之后需要执行的方法；

通过设置 destroy\_method="方法名" 来指定对象销毁之前需要执行的方法。

比如在Student类中增加m1和m2两个无参实例方法，然后配置：

|  |
| --- |
| <**bean id="student" class="com.sd.Student" init-method="m1" destroy-method="m2"** /> |

测试程序：

|  |
| --- |
| **public class** Demo {  **public static void** main(String[] args) {  ClassPathXmlApplicationContext ac = **new** ClassPathXmlApplicationContext(**"applicationContext.xml"**);  Student stu = (Student)ac.getBean(**"student"**);  *// 只有手动销毁容器，才会执行对象销毁前要执行的方法(destroy-method)  // 并且，只有ClassPathXmlApplicationContext类有destroy方法，它的接口ApplicationContext是没有这个方法的  // 所以我们这里用了实现类ClassPathXmlApplicationContext* ac.destroy(); *// 销毁容器* ac.close(); *// 关闭容器* } } |

可以看到两个方法分别在Student对象创建之后和销毁之前被执行了。也可以执行类中的静态方法，但无论是静态方法还是实例方法，都不能执行带有参数的方法。

小结：本章介绍了Spring如何创建对象的以及其中的一些细节。

# 4. 依赖注入（DI）

如果说Spring仅仅提供IoC功能来创建对象的话，可能给人的感觉是用处不大，完全可以自己做好对象的创建。但Spring强大的是还提供了依赖注入功能DI。实际上DI通俗来说，就一句话：给对象的属性赋值。因为类的属性其实就是类的依赖，DI就是名字看上去高深一点。

在三层架构中就体现了对象的依赖关系，例如Service层依赖DAO层的资源，体现在Service类中有DAO层接口类型的属性成员存在，例如“private IBaseDao baseDao;”。

Spring提供了依赖注入功能，因此我们可以利用依赖注入来维护对象间的关系，直观的讲，就是给类的成员“赋值”，提供类的实例所需要的资源，如上述的Service实现类中，需要我们提供IBaseDao资源。

Spring中，如果用XML方式配置bean的依赖关系（即给属性赋值），主要有两种方式，分别是使用getter方法和构造函数。

## 4.1 setter方式实现DI

利用setter方法注入，就是使用类成员提供的setter方法来注入该资源。这就要求需要注入的对象成员一定要写setter方法（getter方法可省略，但为了方便使用，一般都写上）。

一般，我们常用的数据类型有：基本类型及其包装类、字符串、引用类型、List集合、Set集合、Map集合、数组和Properties类（这个Properties类在Spring中常用，特别是配置框架信息的时候）。

DI就是研究如何给上述这些类型的属性赋值，利用Spring配置都能做到。在Spring中，给bean的属性赋值的过程又叫“bean的装配”。

现在就完成bean装配的一个例子。为了充分地演示Spring能装配上述这些类型的属性，下面专门新建一个类Person，类中的属性涵盖了上述的类型：

|  |
| --- |
| **package** com.sd;  **import** java.util.List; **import** java.util.Map; **import** java.util.Properties; **import** java.util.Set;  **public class** Person {  **private** Long **id**;  **private** String **name**;  **private** Student **student**;  **private** List **lists**;  **private** Set **sets**;  **private** Map **map**;  Object[] **objects**;  Properties **properties**;  *// 写上setter getter方法。其实赋值只要setter就行了......* } |

现在就在applicationContext.xml中装配bean：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"**>  *<!-- 1. 先把student加入bean中 -->* <**bean id="stu" class="com.sd.Student"** />  *<!-- 2. 再把person加入bean中。 -->* <**bean id="person" class="com.sd.Person"**>  *<!-- 在bean中，使用property标签来利用setter方法给相应属性注入值。其中：  name属性表示要注入的属性名称  value表示要注入的值（如果属性类型是基本类型及其包装类，或者是字符串的话）。  -->  <!-- 3. 例如给id和name属性赋值 -->* <**property name="id" value="10001"** />  <**property name="name" value="张三"** />  *<!--  4. 这里给student属性赋值。该属性类型是普通的引用类型Student，可以直接引用已经配置的bean资源  stu。使用ref（表示引用）属性指向stu这个bean即可。这表示使用Spring容器内部创建的stu  对象给该student属性赋值。  -->* <**property name="student" ref="stu"**/>  *<!-- 5. 装配lists属性 -->* <**property name="lists"**>  *<!-- 装配list类型属性，要使用list标签，标签中的每个元素就是注入到该集合中的元素 -->* <**list**>  *<!-- 其中可使用value，表示在集合中加入基本类型和字符串等 -->* <**value**>1</**value**>  <**value**>hello world</**value**>  *<!-- 也可使用ref标签，表示引用bean资源 -->  <!-- 这里表示向集合中加入一个stu对象引用的元素 -->* <**ref bean="stu"** />  </**list**>  </**property**>  *<!-- 6. 装配sets属性-->* <**property name="sets"**>  *<!-- 其中和list集合的配置大同小异。只是这里使用set标签 -->* <**set**>  <**value**>1</**value**>  <**value**>hello world</**value**>  <**ref bean="stu"** />  </**set**>  </**property**>  *<!-- 7. 装配map属性 -->* <**property name="map"**>  *<!-- 装配map属性使用map标签 -->* <**map**>  *<!-- 其中用entry标签表示每对键值对  key属性表示键，value属性表示值。  如果key或value需要引用bean，则使用key-ref属性和value-ref属性。  -->  <!-- 键是k1字符串，值是v1字符串 -->* <**entry key="k1" value="v1"** />  *<!-- 键是k2字符串，键引用stu -->* <**entry key="k2" value-ref="stu"** />  </**map**>  </**property**>  *<!-- 8. 装配properties属性 -->* <**property name="properties"**>  *<!-- 使用props标签 -->* <**props**>  *<!-- 其中再用prop标签表示一个键值对  properties键值对和map不同，properties由于相当于是配置文件信息，  因此键和值只允许是String字符串类型，因此直接写key和在标签中写值即可，  不允许对象的引用等其他类型的选择。  -->* <**prop key="p1"**>value1</**prop**>  <**prop key="p2"**>value2</**prop**>  </**props**>  </**property**>  *<!-- 9. 装配数组属性 -->* <**property name="objects"**>  *<!-- 数组的装配用list方式即可 -->* <**list**>  <**value**>数据</**value**>  <**ref bean="stu"** />  </**list**>  </**property**>  </**bean**> </**beans**> |

说明：

（1）使用property利用setter方法注入属性值。对于基本类型及其包装类，和字符串类型，用value属性赋值即可，会自动应用合适的数据和类型。

（2）ref属性表示引用配置好的bean资源，即直接引用Spring容器中的资源。当然也可用ref标签实现，例如student属性的注入也可这样写：

|  |
| --- |
| <**property name="student"**>  <**ref bean="stu"** /> </**property**> |

ref属性和标签就是引用的意思，在很多地方都会这样使用（一般常使用简洁的ref属性），例如引用Spring配置中已有的对象和字符串等资源等。

（3）配置list、set、map类型属性使用分别使用list、set和map标签，其中引用（ref）bean可正常使用。在map的entry中，也可通过value或ref标签设置map的值：

|  |
| --- |
| <**entry key="k1"**>  <**value**>v1</**value**> </**entry**> <**entry key="k2"**>  <**ref bean="stu"** /> </**entry**> |

同样，还是推荐用属性的方式配置。

另外，数组也可用list方式装配。而properties的装配只允许字符串类型的键值。

装配好后，运行程序代码：

|  |
| --- |
| **public static void** main(String[] args) {  ApplicationContext ac = **new** ClassPathXmlApplicationContext(**"applicationContext.xml"**); *// 此时不创建对象* Person person = (Person) ac.getBean(**"person"**);  person.getStudent().say();  System.***out***.println(person.getLists().size()); } |

根据输出内容，可证明属性已经被注入了，如果要看详细信息，断点调试即可。

上述就讲完了对象装配的过程。将来在项目中，可能并不是我们每需要一个对象就从Spring容器中拿一个，且要这样配置依赖注入。但在配置项目的基础架构和利用Spring进行框架的整合时，我们就很可能需要进行这样的装配配置，使得Spring能管理其他框架的对象。到时配置时，就会知道这些配置大概是注入了什么信息，而且了解了bean的装配后，我们自己看官方文档也能进行灵活的配置，这也是学习装配的原因之一。

那么IoC和DI有什么意义呢？它们的真正意义就是：让Java代码端进行完全的面向接口编程。

以三层架构举例，我们Service等层中都使用了接口，在写具体Controller或Service的实现时，可以说是“面向接口编程”，这种方式让开发者无需关心具体的实现。但是以前写的Service和DAO是“不完全的面向接口编程”。为什么呢？这是因为在实现类中，必须先使用一个实现类创建接口对象。例如在某Service中：

|  |
| --- |
| **package** com.sd;  **public class** StudentServiceImpl {  *// 需要使用实现类创建对象* **private** IStudentDao **studentDao** = **new** MySQLStudentDao();  **public void** addStudent(Student student) {  *// 方法中是“面向接口编程”* **studentDao**.insert(student);  } } |

也就是说使用接口对象前必须要在代码中使用一个实现类创建对象。就是这样的做法导致了“不完全”的面向接口编程，使Service依赖了具体的实现。

现在，如果采用了Spring的IoC和依赖注入，Service和Controller中完全不需要关心实现类是什么，Service中只要维护一个该接口的属性并提供setter方法即可，至于该属性是什么实现类，根本不需要关心，这完全由Spring的配置决定，在程序运行时对属性进行注入。因此Service只要改成这样：

|  |
| --- |
| **package** com.sd;  **public class** StudentServiceImpl {  *// 无需实现* **private** IStudentDao **studentDao**;  *// 提供setter方法以便注入* **public void** setStudentDao(IStudentDao studentDao) {  **this**.**studentDao** = studentDao;  }   **public void** addStudent(Student student) {  *// “面向接口编程”* **studentDao**.insert(student);  } } |

而三层架构中的Spring就可这样配置（使用ref引用）：

|  |
| --- |
| *<!-- action -> service -> dao 的案例（以操作student为例） --> <!-- 先配置student对象。 -->* <**bean id="studentDao" class="project.StudentDao"** /> *<!-- 再配置service对象 -->* <**bean id="studentService" class="project.StudentService"**>  *<!-- 其中的studentDao属性要引用dao对象 -->* <**property name="studentDao" ref="studentDao"** /> </**bean**> *<!-- 最后配置action -->* <**bean id="studentAction" class="project.StudentAction"**>  *<!-- 其中的studentService属性要引用service对象 -->* <**property name="studentService" ref="studentService"** /> </**bean**> |

具体的三层结构的整合后面会详细说。总之，利用IoC和DI能实现完全面向接口的编程，使开发者无需关心实现类。

## 4.2 构造器方式实现DI

Spring容器默认调用无参构造创建对象，如果指定Spring在创建对象时使用带参构造，那么也能利用这种方式实现属性的注入，即在创建对象时“顺便”进行DI。

例如，给Student加上构造器：

|  |
| --- |
| **public** Student(String name, **int** age) {  **this**.**name** = name;  **this**.**age** = age; } |

这时可这样配置bean，通过构造器进行DI：

|  |
| --- |
| <**bean id="student" class="com.sd.Student"**>  *<!-- 用constructor-arg配置构造参数值。*  *index代表参数的索引，从0开始，*  *type表示类型（对于引用类型，必须写全名，如java.lang.String）*  *value表示参数值 -->* <**constructor-arg index="0" type="java.lang.String" value="张三"** />  <**constructor-arg index="1" type="int" value="100"** /> </**bean**> |

获取对象的方式和以前是一样的，这时得到的对象就有指定的值了。

当然，也可使用ref进行资源的引用。例如：

|  |
| --- |
| *<!-- 先配置一个String资源。这里，该字符串资源也利用了构造器进行了注入。（实际是没必要这样做的，为了演示） -->* <**bean id="zs" class="java.lang.String"**>  <**constructor-arg index="0" type="java.lang.String" value="张三"** /> </**bean**>  <**bean id="student" class="com.sd.Student"**>  *<!-- Student中，构造参数的值引用了上述字符串bean资源（使用ref）。 -->* <**constructor-arg index="0" type="java.lang.String" ref="zs"** />  <**constructor-arg index="1" type="int" value="100"** /> </**bean**> |

在程序中：

|  |
| --- |
| **public class** Demo {  **public static void** main(String[] args) {  ClassPathXmlApplicationContext ac = **new** ClassPathXmlApplicationContext(**"applicationContext.xml"**);  *// 获取Student对象* Student stu = (Student)ac.getBean(**"student"**);  System.***out***.println(stu); *// Student{name='张三', age=100}  // 也可拿到该字符串资源* String zs = (String)ac.getBean(**"zs"**); *// 张三* System.***out***.println(zs);  } } |

说明：

（1）如果想要用setter注入，一定要提供默认构造器，否则Spring无法创建对象。一般，默认构造器和setter、getter方法是一个普通类都应该提供的，方便使用。

（2）最好有这样的条件反射：看到一个类提供了构造器和setter方法，马上就能想到利用Spring可以实现什么属性的注入。就像看到美女条件反射要多看几眼。如果能达到这种条件反射的境界，说明Spring的DI已经深入人心。

## 4.3 Spring容器中bean的继承

例如，现在Person类中，有name属性：

|  |
| --- |
| **package** com.sd;  **public class** Person {  **private** String **name**; *// name属性   // setter和getter...* } |

Dancer类继承Person：

|  |
| --- |
| **package** com.sd;  **public class** Dancer **extends** Person {  } |

如果现在applicationContext.xml中配置这两个bean：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"**>  *<!-- 先把Person加入到bean，并注入name属性 -->* <**bean id="person" class="com.sd.Person"**>  <**property name="name" value="张三"** />  </**bean**>  *<!-- 再把Dancer加入到bean -->* <**bean id="dancer" class="com.sd.Dancer"** /> </**beans**> |

那么从Spring容器中拿到Dancer对象后，该对象能拿到继承的父类的name属性“张三”吗？

|  |
| --- |
| **package** com.sd;  **import** org.springframework.context.ApplicationContext; **import** org.springframework.context.support.ClassPathXmlApplicationContext;  **public class** Main {   **public static void** main(String[] args) {  ApplicationContext ac = **new** ClassPathXmlApplicationContext(**"applicationContext.xml"**);  Dancer dancer = (Dancer) ac.getBean(**"dancer"**); *// 拿到dancer* System.***out***.println(dancer.getName()); *// 输出getName().* } } |

运行程序，输出是null，说明属性值没有被继承。这是因为Student和Person都是Spring容器单独创建出来的，Java中本身也没有“属性的继承”一说，只会继承方法，看上去是“继承了属性”，实际上是方法的输出就是那样。

如果想在Spring中“继承父类的属性”，即使用父类bean的属性值，只要在子类bean中使用parent属性指向父类bean即可。例如：

|  |
| --- |
| <**bean id="dancer" class="com.sd.Dancer" parent="person"** /> |

这样再运行程序，就会输出父类bean注入的属性值。用parent就实现了Spring容器内部的继承关系。

实际上，可直接在Dancer中注入name属性，这是因为Dancer继承了Person，会继承getName方法，既然也就“有了”name这个“属性”，在Spring中是可配置的：

|  |
| --- |
| *<?***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="dancer" class="com.sd.Dancer"**>  <**property name="name" value="张三"** />  </**bean**> </**beans**> |

## 4.4 使用注解

使用注解能简化XML配置，Spring对注解提供了很好的支持，配置方便。

### 4.4.1 依赖注入注解

一般使用@Resource注解来进行依赖的注入，可以减少xml中的配置。

现有Student类如下：

|  |
| --- |
| **package** com.sd;  **public class** Student {  **public void** say() {  System.***out***.println(**"I am a student."**);  } } |

而Person类中的属性依赖Student：

|  |
| --- |
| **package** com.sd;  **public class** Person {  **private** Student **student**;  *// 提供了一个方法用于获得student对象* **public** Student getAStudentInstance() {  **return student**;  } } |

现想用Spring容器装配Person。如果用XML方式，则一般在Person中提供setter方法，在XML中配置bean并注入。现在如果用注解方式，则直接使用@Resource注解来注入依赖即可，这样XML只要配置bean的创建即可。

因此，Person类改为这样即可：

|  |
| --- |
| **package** com.sd;  **import** javax.annotation.Resource;  **public class** Person {  @Resource  **private** Student **student**;    **public** Student getAStudentInstance() {  **return student**;  } } |

而额外注意的是，使用注解注入依赖，在XML中还要配置启动依赖注入的注解解析器。因此applicationContext.xml中内容如下：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd"**>  *<!-- 1. 配置Student和Person的bean -->* <**bean id="student" class="com.sd.Student"** />  <**bean id="person" class="com.sd.Person"** />  *<!-- 2. 启动依赖注入的注解解析器(一个空标签) -->* <**context:annotation-config** /> </**beans**> |

（说明：上述xml中使用了context命名空间，也引入了相应的xsd约束）

测试代码：

|  |
| --- |
| **public static void** main(String[] args) {  ApplicationContext ac = **new** ClassPathXmlApplicationContext(**"applicationContext.xml"**);  Person person = (Person) ac.getBean(**"person"**); *// 拿到dancer* person.getAStudentInstance().say(); } |

上述代码成功运行，说明student属性注入是成功的。这也说明了，使用@Resource注入是不需要属性提供setter和getter方法的（但一般提供）。

下面详细解释下上述程序执行的流程是怎么样的：

（1）当Spring容器启动时，读取applicationContext.xml文件；

（2）根据配置的两个bean，Spring容器先创建了两个对象；

（3）当Spring解析到xml中配置了依赖注入的注解解析器时，Spring就会在Spring容器管理的bean范围内查找这些类的属性上是否加了@Resource注解，如果加了该注解，则进行下一步的处理（也就是@Resource所达到的效果）；

（4）Spring解析@Resource的name属性。

若没有写该属性，Spring会得到注解所在的类属性student，Spring就会把容器中id为student的bean注入到类的该属性student中，如果Spring容器中没有id为student的bean，则Spring会再按照注解所在的属性的类型（即Student类）进行匹配（首字母小写，同样也是student），如果再找不到bean，则报错。因此我们直接写@Resource是可注入成功的，因为属性名是student，会找到id为student的bean，如果属性名不是student，也会找到，因为属性类型是Student。

若@Resource注解中写了name属性值，则按照该直接的name属性值去Spring中匹配对应的bean，如果找不到，则报错“NoSuchBeanDefinitionException: No bean named 'xxx' available”。所以，如果这样写的话：

|  |
| --- |
| @Resource(name = **"mystu"**) **private** Student **student**; |

程序运行将报错，因为容器中没有id匹配的bean。

也就是说，@Resource用于依赖注入，如果注解中不写name属性，则会按照属性名和类型名到Spring容器中匹配相应的bean注入，如果写了name，就会根据name值去容器中匹配相应的bean注入。

除了@Resource外，Spring框架本身还提供了@Autowired和@Qualifier两个注解，这两个注解的结合使用就相当于@Resource注解。什么意思呢？

（1）如果在属性上单纯使用@Autowired，这和使用@Resource没有区别，都是先按照属性名再按照类型名去Spring容器中匹配bean。

（2）但@Autowired无法指定name属性，即无法指定匹配的bean。这时就需要和@Qualifier结合使用了，@Qualifier中使用value值指定要匹配的bean，相当于@Resource中的name属性。单独使用@Qualifier是无法注入的，一般就是结合@Autowired使用，也就是说@Qualifier的作用就是指定匹配的bean。例如：

|  |
| --- |
| @Autowired @Qualifier(**"mystu"**) **private** Student **student**; |

这和使用@Resource(name="mystu")的效果是一样的。

说明：

（1）@Resource只能用于注入引用类型的属性，如果想注入基本类型的属性，可以使用@Value注解。例如：

|  |
| --- |
| **public class** Person {  @Value(**"开发部"**)  **private** String **dept**; } |

程序运行时可能拿到注入的值。@Value注解的value属性是字符串类型，但Spring会根据真实类型进行转换，比如应用到int类型时，会自动将属性值“12”转为整型12存储。

（2）注解写法比较简单，但是效率比较低（因为要扫描和解析类）。XML写法比较复杂，但是效率比较高，控制灵活。其实效率在这里影响不大，基本是在启动时完成的。

### 4.4.2 类扫描注解

使用类扫描注解能把类加入到bean，让Spring进行管理，再结合@Resource等注解进行属性的注入，这样的话XML中基本就不需要什么配置，大部分都能使用注解来代替。因此注解和XML是两种选择，开发中会结合使用。

如果想把某类加入到Spring容器的bean中，则使用@Component注解。

@Component用在类上，指定把该类加入到IOC容器中，该注解相当于XML中的bean节点。该注解只有一个value属性值，用于标识这个bean，相当于XML中为bean设置id属性。如果@Component不写value属性，Spring会默认将id为类名（类名首字母小写）的bean注入到容器中。

当然，首先必须要在Spring的配置文件中开启注解扫描。开启注解扫描就是指定对哪些包中的类进行注解扫描，这样，这些类中的注解才会生效。开启了注解扫描，原本配置的依赖注入的注解解析器就可去掉了，因为注解扫描本身对@Resource也有效。

即我们在applicationContext.xml中只需要这样配置即可：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd"**>  *<!-- 开启注解扫描  base-package是指定开始扫描的位置。这里指定为com.zhang  即com.zhang下的类及其子包下的类都能被扫描到*  *（把一个类放入到spring容器中,该类就是一个component）  -->* <**context:component-scan base-package="com.sd"** /> </**beans**> |

其余XML中就什么也不需配置了，因为接下来我们用全部在类中使用注解配置。

例子：现在有两个类：学生类Student和地址类Address。学生类的addr属性类型是Address，即Student依赖Address。下面用注解进行bean的配置和属性的注入（也使用@Value注解注入基本类型数据）。

Address类：

|  |
| --- |
| **package** com.sd; **import** org.springframework.beans.factory.annotation.Value; **import** org.springframework.stereotype.Component;  *// @Component只有value属性，所以可直接写该属性值 // @Component会自动在容器中注册一个名为“address”的bean // 当然，直接写@Component()也行，Spring会自动把id为类名首字母小写的bean，即address加入到容器中* @Component(**"address"**) **public class** Address {  *// 设置值* @Value(**"家庭住址"**)  **private** String **name**;  @Value(**"帝王大街"**)  **private** String **detail**;  *// setter和getter* } |

Student类：

|  |
| --- |
| **package** com.sd;  **import** org.springframework.stereotype.Component; **import** javax.annotation.Resource;  *// 把id为student的bean加入到容器中* @Component() **public class** Student {  *// 注入address属性，将容器中id为address的bean注入。这在Address类中配置过了。  // 当然也可直接写@Resource()，Spring自动先根据属性名和类型名注入* @Resource(name = **"address"**)  **private** Address **address**;  *// setter和getter* } |

最后，主类这样测试即可：

|  |
| --- |
| **public static void** main(String[] args) {  ApplicationContext ac = **new** ClassPathXmlApplicationContext(**"applicationContext.xml"**);  Student student = (Student) ac.getBean(**"student"**); *// 拿到bean的id为student的实例* System.***out***.println(student.getAddress().getDetail()); } |

最终测试成功，输出“帝王大街”。

在实际的项目中，为了区分各层的component，在DAO层、Service层和Controller层分别可以用@Repository、@Service和@Controller注解，它们的作用和@Component是相同的，只是名字不同而已。这些在以后整合项目时会用到，需要知道。

## 4.5 其他的装配方式

上述讲的几种方式都是在开发中比较常用的。其实还有其他的装配bean方式。下面再介绍两种。

### 4.5.1 使用内部bean注入依赖

同样，以实现三层架构为例：

|  |
| --- |
| <**bean id="userAction" class="project.UserAction"**>  *<!-- 下面都是嵌套的内部bean -->* <**property name="userService"**>  <**bean class="cproject.UserService"**>  <**property name="userDao"**>  <**bean class="project.UserDao"**></**bean**>  </**property**>  </**bean**>  </**property**> </**bean**> |

这样虽然能减少配置，是层次的结构，但是看起来较复杂。内部bean也能配置自己的property和constructor-arg等节点来设置值。当然，使用内部bean也要求成员属性拥有setter方法。

### 4.5.2 使用p名称空间进行DI

使用此方法需要引入p名称空间，实例如下：

|  |
| --- |
| *<?***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:p="http://www.springframework.org/schema/p"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"**>  *<!-- 正常配置所需资源 -->* <**bean id="str" class="java.lang.String"**>  <**constructor-arg index="0" value="张三"**/>  </**bean**>  *<!-- 只要在需注入资源的对象上配置相关引用属性。 -->  <!-- 该属性名规则：“p:对象属性名-ref”，使用了p名称空间，属性值指向所引用资源 -->* <**bean id="student" class="com.sd.Student" p:name-ref="str"**/> </**beans**> |

另外，可直接给属性赋值，例如：

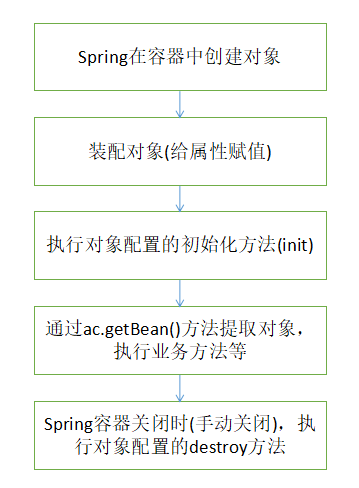
|  |
| --- |
| <**bean id="student" class="com.zhang.test.Student" p:name="张三" p:age="12"**></**bean**> |

三层架构例子：

|  |
| --- |
| <**bean id="userDao" class="project.UserDao"**></**bean**> <**bean id="userService" class="project.UserService" p:userDao-ref="userDao"**></**bean**> <**bean id="userAction" class="project.UserAction" p:userService-ref="userService"**></**bean**> |

# 5. 总结：Spring创建对象的一般流程

一般流程如下图所示：



可自行编写代码断点调试进行验证。