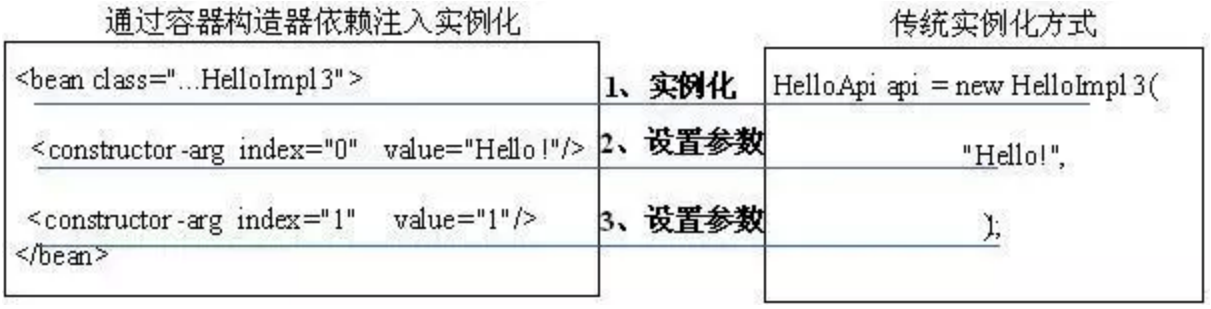
Spring配置：属性注入

构造器注入:

通过配置构造器参数实现，构造器参数就是依赖。除了构造器方式，还有静态工厂、实例工厂方法可以进行构造器注入。



构造器注入可以根据参数索引注入、参数类型注入或Spring3支持的参数名注入，但参数名注入是有限制的，需要使用在编译程序时打开调试模式（即在编译时使用“javac -g:vars”在class文件中生成变量调试信息，默认是不包含变量调试信息的，从而能获取参数名字，否则获取不到参数名字）或在构造器上使用@ConstructorProperties（java.beans.ConstructorProperties）注解来指定参数名。

一、根据参数索引注入，使用标签“<constructor-arg index="1" value="1"/>”来指定注入的依赖，其中“index”表示索引，从0开始，即第一个参数索引为0，“value”来指定注入的常量值。

二、根据参数类型进行注入，使用标签“<constructor-arg type="java.lang.String" value="Hello World!"/>”来指定注入的依赖，其中“type”表示需要匹配的参数类型，可以是基本类型也可以是其他类型，如“int”、“java.lang.String”，“value”来指定注入的常量值。

三、根据参数名进行注入，使用标签“<constructor-arg name="message" value="Hello World!"/>”来指定注入的依赖，其中“name”表示需要匹配的参数名字，“value”来指定注入的常量值。

除了可以使用以上方式，还可以通过在构造器上添加@java.beans.ConstructorProperties({"message", "index"})注解来指定参数名字。

静态工厂类

```
//静态工厂类
package cn.javass.spring.chapter3;
import cn.javass.spring.chapter2.helloworld.HelloApi;
public class DependencyInjectByStaticFactory {
    public static HelloApi newInstance(String message, int index) {
        return new HelloImpl3(message, index);
```

```
    }
}
--
<bean id="byIndex"
   class="cn.javass.spring.chapter3.DependencyInjectByStaticFactory"
      factory-method="newInstance">
   <constructor-arg index="0" value="Hello World!"/>
   <constructor-arg index="1" value="1"/>
</bean>
```

实例工厂类

```
//实例工厂类
package cn.javass.spring.chapter3;
import cn.javass.spring.chapter2.helloworld.HelloApi;
public class DependencyInjectByInstanceFactory {
   public HelloApi newInstance(String message, int index) {
      return new HelloImpl3(message, index);
   }
}
--
<bean id="instanceFactory"
class="cn.javass.spring.chapter3.DependencyInjectByInstanceFactory"/>

<bean id="byIndex"
      factory-bean="instanceFactory"  factory-method="newInstance">
      <constructor-arg index="0" value="Hello World!"/>
      <constructor-arg index="1" value="1"/>
</bean>
```

静态工厂方式和实例工厂方式根据参数名字注入的方式只支持通过在class文件中添加“变量调试信息”方式才能运行，ConstructorProperties注解方式不能工作，它只对构造器方式起作用，不建议使用根据参数名进行构造器注入。

setter注入

setter注入，是通过在通过构造器、静态工厂或实例工厂实例好Bean后，通过调用Bean类的setter方法进行注入依赖。

通过容器setter注入		传统setter方式
<bean class="...HelloImpl4">	1、实例化	HelloApi api = new HelloImpl4 ();
<property name="message" value="Hello"/>	2、设置属性	Api. setMessage("Hello");
<property name="index" value="1"/>	3、设置属性	Api. setIndex (" Hello");
</bean>		

```
package cn.javass.spring.chapter3;
import cn.javass.spring.chapter2.helloworld.HelloApi;
public class HelloImpl4 implements HelloApi {
   private String message;
   private int index;
   //setter方法
   public void setMessage(String message) {
      this.message = message;
   }
   public void setIndex(int index) {
      this.index = index;
   }
   @Override
   public void sayHello() {
      System.out.println(index + ":" + message);
   }
}
--
<!-- 通过setter方式进行依赖注入 -->
<bean id="bean" class="cn.javass.spring.chapter3.HelloImpl4">
   <property name="message" value="Hello World!"/>
   <property name="index">
      <value>1</value>
   </property>
</bean>
--
@Test
public void testSetterDependencyInject() {
   BeanFactory beanFactory =
      new ClassPathXmlApplicationContext("chapter3/setterDependencyInject.xml");
   HelloApi bean = beanFactory.getBean("bean", HelloApi.class);
   bean.sayHello();
}
```

知道如何配置了，但Spring如何知道setter方法？如何将值注入进去的呢？其实方法名是要遵守约定的，setter注入的方法名要遵循“JavaBean getter/setter 方法命名约定”：

JavaBean：是本质就是一个POJO类，但具有一下限制：

- 该类必须要有公共的无参构造器，如public HelloImpl4() {}；
- 属性为private访问级别，不建议public，如private String message;
- 属性必要时通过一组setter（修改器）和getter（访问器）方法来访问；
- setter方法，以“set” 开头，后跟首字母大写的属性名，如“setMesssage”，简单属性一般只有一个方法参数，方法返回值通常为“void”；
- getter方法，一般属性以“get”开头，对于boolean类型一般以“is”开头，后跟首字母大写的属性名，如“getMesssage” ，“isOk” ；

- 还有一些其他特殊情况，比如属性有连续两个大写字母开头，如“URL”,则setter/getter方法为：“setURL”和“getURL”，其他一些特殊情况请参看“Java Bean”命名规范。

注入常量

```
<property name="message" value="Hello World!"/>
```

或

```
<property name="index">
   <value>1</value>
</property>
```

以上两种方式都可以，从配置来看第一种更简洁。注意此处“value”中指定的全是字符串，由Spring容器将此字符串转换成属性所需要的类型，如果转换出错，将抛出相应的异常。

Spring容器目前能对各种基本类型把配置的String参数转换为需要的类型。

注：Spring类型转换系统对于boolean类型进行了容错处理，除了可以使用“true/false”标准的Java值进行注入，还能使用“yes/no”、“on/off”、“1/0”来代表“真/假”，所以大家在学习或工作中遇到这种类似问题不要觉得是人家配置错了，而是Spring容错做的非常好。

注入Bean ID

用于注入Bean的ID，ID是一个常量不是引用，且类似于注入常量，但提供错误验证功能

```
<property name="id">
   <idref bean="bean1"/>
</property>
```

上述配置本质上在运行时等于如下方式

```
<property name="id" ref="bean1"/>
```

第二种方式（<idref bean=″bean1″/>）可以在容器初始化时校验被引用的Bean是否存在，如果不存在将抛出异常，而第一种方式（<idref local=″bean2″/>）只有在Bean实际使用时才能发现传入的Bean的ID是否正确，可能发生不可预料的错误。因此如果想注入Bean的ID，推荐使用第二种方式。

```
package cn.javass.spring.chapter3.bean
public class IdRefTestBean {
   private String id;
   public String getId() {
      return id;
   }
   public void setId(String id) {
      this.id = id;
   }
}
```

--

```
<bean id="bean1" class="java.lang.String">
<constructor-arg index="0" value="test"/>
</bean>
<bean id="bean2" class="java.lang.String">
   <constructor-arg index="0" value="test"/>
</bean>
```

--

```
<bean id="idrefBean1" class="cn.javass.spring.chapter3.bean.IdRefTestBean">
      <property name="id"><idref bean="bean1"/></property>
</bean>
<bean id="idrefBean2" class="cn.javass.spring.chapter3.bean.IdRefTestBean">
   <property name="id"><idref local="bean2"/></property>
</bean>
```

从配置中可以看出，注入的Bean的ID是一个java.lang.String类型，即字符串类型，因此注入的同样是常量，只是具有校验功能。

<idref bean="……"/>将在容器初始化时校验注入的ID对于的Bean是否存在，如果不存在将抛出异常。

<idref local="……"/>将在XML解析时校验注入的ID对于的Bean在当前配置文件中是否存在，如果不存在将抛出异常，它不同于<idref bean="……"/>，<idref local="……"/>是校验发生在XML解析式而非容器初始化时，且只检查当前配置文件中是否存在相应的Bean。

注入集合、数组和字典

Spring不仅能注入简单类型数据，还能注入集合（Collection、无序集合Set、有序集合List）类型、数组(Array)类型、字典(Map)类型数据、Properties类型数据。

引用其它Bean

```
package cn.javass.spring.chapter3.bean;
import cn.javass.spring.chapter2.helloworld.HelloApi;
public class HelloApiDecorator implements HelloApi {
   private HelloApi helloApi;
   //空参构造器
   public HelloApiDecorator() {
   }
   //有参构造器
   public HelloApiDecorator(HelloApi helloApi) {
      this.helloApi = helloApi;
   }
   public void setHelloApi(HelloApi helloApi) {
      this.helloApi = helloApi;
   }
   @Override
```

```
    public void sayHello() {
        System.out.println("==========装饰一下==========");
        helloApi.sayHello();
        System.out.println("==========装饰一下==========");
    }
}
--
<!-- 定义依赖Bean -->
<bean id="helloApi" class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
<!-- 通过构造器注入 -->
<bean id="bean1" class="cn.javass.spring.chapter3.bean.HelloApiDecorator">
    <constructor-arg index="0" ref="helloApi"/>
</bean>
<!-- 通过构造器注入 -->
<bean id="bean2" class="cn.javass.spring.chapter3.bean.HelloApiDecorator">
    <property name="helloApi"><ref bean=" helloApi"/></property>
</bean>
--
@Test
public void testBeanInject() {
    BeanFactory beanFactory =
        new ClassPathXmlApplicationContext("chapter3/beanInject.xml");
    //通过构造器方式注入
    HelloApi bean1 = beanFactory.getBean("bean1", HelloApi.class);
    bean1.sayHello();
    //通过setter方式注入
    HelloApi bean2 = beanFactory.getBean("bean2", HelloApi.class);
    bean2.sayHello();
}
```

最后再来提升下理论知识吧！

依赖和依赖注入

传统应用程序设计中所说的依赖一般指“类之间的关系”，那先让我们复习一下类之间的关系：

1. 泛化：表示类与类之间的继承关系、接口与接口之间的继承关系；

2. 实现：表示类对接口的实现；

3. 依赖：当类与类之间有使用关系时就属于依赖关系，不同于关联关系，依赖不具有“拥有关系”，而是一种“相识关系”，只在某个特定地方（比如某个方法体内）才有关系。

4. 关联：表示类与类或类与接口之间的依赖关系，表现为“拥有关系”；具体到代码可以用实例变量来表示；

5. 聚合：属于是关联的特殊情况，体现部分-整体关系，是一种弱拥有关系；整体和部分可以有不一样的生命周期；是一种弱关联；

6. 组合：属于是关联的特殊情况，也体现了体现部分-整体关系，是一种强“拥有关系”；整体与部分有相同的生命周期，是一种强关联；

Spring IoC容器的依赖有两层含义：

Bean依赖容器和容器注入Bean的依赖资源：

- **Bean依赖容器：也就是说Bean要依赖于容器，这里的依赖是指容器负责创建Bean并管理Bean的生命周期，正是由于由容器来控制创建Bean并注入依赖，也就是控制权被反转了，这也正是IoC名字的由来，此处的有依赖是指Bean和容器之间的依赖关系。**
- **容器注入Bean的依赖资源：容器负责注入Bean的依赖资源，依赖资源可以是Bean、外部文件、常量数据等，在Java中都反映为对象，并且由容器负责组装Bean之间的依赖关系，此处的依赖是指Bean之间的依赖关系，可以认为是传统类与类之间的“关联”、“聚合”、“组合”关系。**

为什么要应用依赖注入，应用依赖注入能给我们带来哪些好处呢？

- **动态替换Bean依赖对象，程序更灵活：替换Bean依赖对象，无需修改源文件：应用依赖注入后，由于可以采用配置文件方式实现，从而能随时动态的替换Bean的依赖对象，无需修改java源文件；**
- **更好实践面向接口编程，代码更清晰：在Bean中只需指定依赖对象的接口，接口定义依赖对象完成的功能，通过容器注入依赖实现；**
- **更好实践优先使用对象组合，而不是类继承：因为IoC容器采用注入依赖，也就是组合对象，从而更好的实践对象组合。**
- **增加Bean可复用性：依赖于对象组合，Bean更可复用且复用更简单；**
- **降低Bean之间耦合：由于我们完全采用面向接口编程，在代码中没有直接引用Bean依赖实现，全部引用接口，而且不会出现显示的创建依赖对象代码，而且这些依赖是由容器来注入，很容易替换依赖实现类，从而降低Bean与依赖之间耦合；**
- **代码结构更清晰：要应用依赖注入，代码结构要按照规约方式进行书写，从而更好的应用一些最佳实践，因此代码结构更清晰。**