# Bean的配置

在Spring应用中，Spring IoC容器可以创建、装配和配 置应用组件，这里的应用组件对 象称为Bean。通俗一点理解，Bean就是一个对象。

1）Spring可以看作是一个大型工厂，用于生产和管理Spring容器中的Bean。如果要使 用这个工厂生成和管理Bean，需要开发者将Bean配置在Spring的配置文件中。Spring 框架支持XML和Properties两种格式的配置文件，但在实际开发中通常采用XML 格式的配置文件。

2）采用XML格式的配置文件，其XML文件的根元素是<beans>，在<beans>中可以包含 多个<bean>子元素，每个<bean>元素定义一个Bean，并描述Bean如何被装配到 Spring容器中。

3）Bean元素的常用属性及其元素如下表所示：

|  |  |
| --- | --- |
| **属性或子元素名称** | **描述** |
| id | id是Bean在BeanFactory中的唯一标识。在代码中获取Bean时以此作为参数名称。 |
| class | 指定Bean的类型，使用全类名的形式，如（dao.TestDIDaoImpl） |
| factory-bean | 使用实例工厂实例化Bean时会使用，用于指定实例工厂的引用，即类型为实例工厂的Bean的id值。需要与factory-method属性一同使用。 |
| factory-method | 使用静态工厂实例化Bean时或使用实例工厂实例化Bean时会使用该属性，用于指定静态工厂中的静态方法名称。 |
| init-method | 该属性与Bean的生命周期有关，用于指定Bean的初始化方法 |
| destroy-method | 该属性与Bean的生命周期有关，用于指定Bean的销毁方法 |
| scope | 用于指定Bean的作用域，默认为singletion（详见第三节） |
| <constructor-arg> | <bean>元素的子元素，表示使用构造方法注入，指定构造方法的参数。该元素的index属性用于指定参数的序号；ref属性用于指定对BeanFactory中其他Bean的引用关系；type属性用于指定参数类型；value属性用于指定参数的常量值（非final）。ref和value属性不可同时存在。 |
| <property> | <bean>元素的子元素，用于设置该Bean的一个属性（表示使用属性的setter方法注入）。该元素的name属性用于指定Bean实例中相应的属性名称；value属性用于指定Bean的属性值；ref属性用于指定属性对BeanFactory中其他Bean的引用关系。ref和value属性不可同时存在。 |
| <list> | <property>元素的子元素，用于将List或数组类型的对象注入给该Bean的属性成员变量。 |
| <map> | <property>元素的子元素，用于将Map类型的对象注入给该Bean的属性成员变量。 |
| <set> | <property>元素的子元素，用于将Set类型的对象注入给该Bean的属性成员变量。 |
| <entry> | <map>元素的子元素，用于设置一个键值对。拥有key属性和value属性 |

# Bean的实例化

在面向对象编程中，如果想要使用某个对象，则必须先实例化该对象。同样，在Spring 框架中，如果想要使用Spring容器中的Bean，也需要对Bean进行实例化。Spring框架 实例化Bean有3种方式：

1. 构造方法实例化；
2. 静态工厂实例化；
3. 实例化工厂实例化。

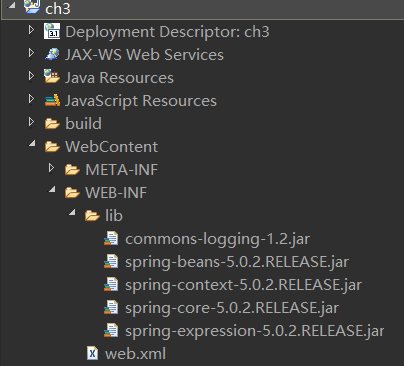
其中，构造方法实例化是最常用的。

## 构造方法实例化

在Spring框架中，Spring容器可以调用Bean对应类中的无参构造方法来实例化该Bean， 这种方式称为构造方法实例化。

1. **创建web应用ch3**

创建一个名为ch3的Web应用，并导入Spring支持和依赖的Jar包。如下图所示：



1. **创建BeanClass类**

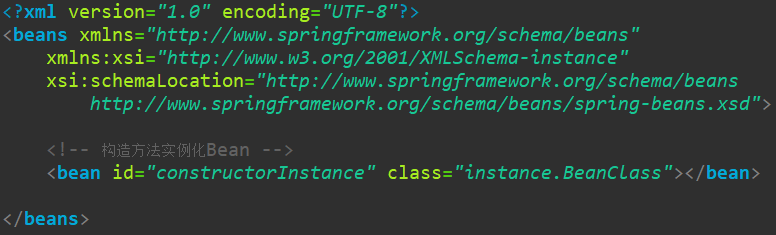
在ch3的src目录下创建instance包，并在该包中创建BeanClass类，代码如下：



为了凸显通过构造方法实例化，故意在构造方法中对message进行赋值操作。

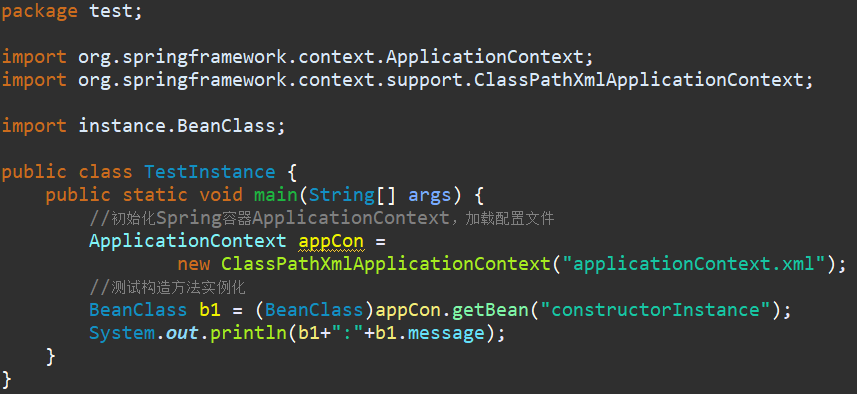
1. **创建Spring配置文件**

在ch3的src目录下创建Spring的配置文件applicationContext.xml，在配置文件中定义Bean，代码如下：

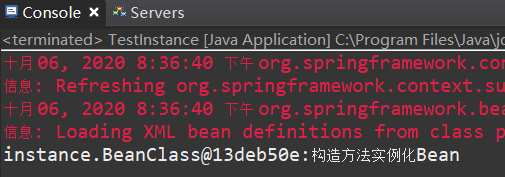


1. **创建测试类**

在ch3的src目录下创建test包，并在该包下创建测试类TestInstance，代码如下：



控制台运行结果：

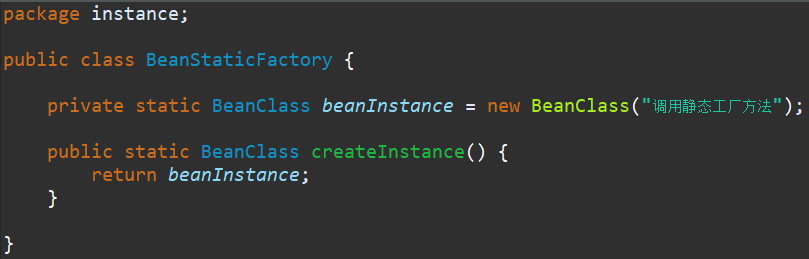


## 静态工厂实例化

使用静态工厂实例化Bean时要求开发者创建工厂类，并在工厂类中创建静态方法来创 建Bean的实例；在配置Bean时，<bean>的class属性指向静态工厂类的全类名，同时 还需要使用factory-method属性指向工厂类中的静态方法。

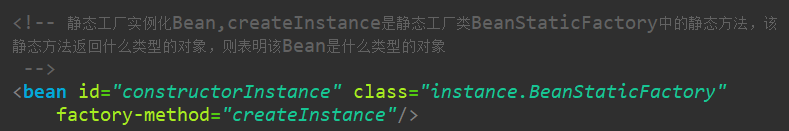
1. **创建工厂类BeanStaticFactory**

在上一小节创建的ch3应用中，在instance包中创建工厂类BeanStatic，代码如下：



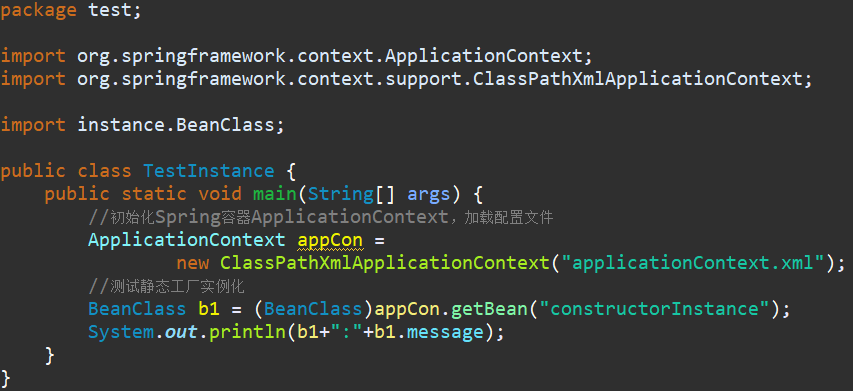
1. **编辑配置文件**

在Spring配置文件中通过静态工厂实例化，代码如下：



1. **添加测试代码**

编写测试类进行测试，代码如下：

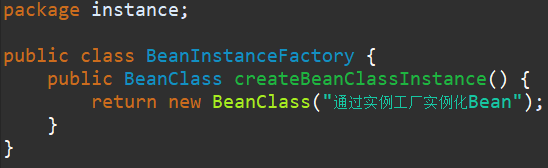


## 实例工厂实例化

使用实例工厂实例化Bean时要求开发者创建工厂类，并在工厂类中创建一个实例方法 来创建Bean的实例。在配置Bean时需要使用factory-bean属性指向配置的实例工厂， 同时还需要使用factory-method属性指向实例工厂中的实例方法。

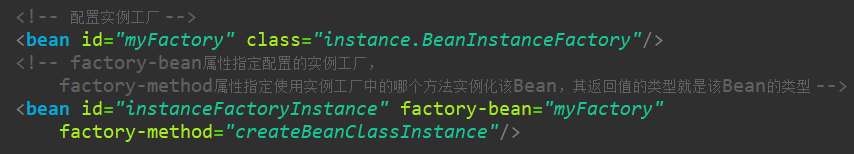
1. **创建实例工厂类BeanInstanceFactory**

在上节的ch3应用中，在instance包中创建工厂类BeanInstanceFactory，该类中有一个实例化来实例化对象，具体代码如下：



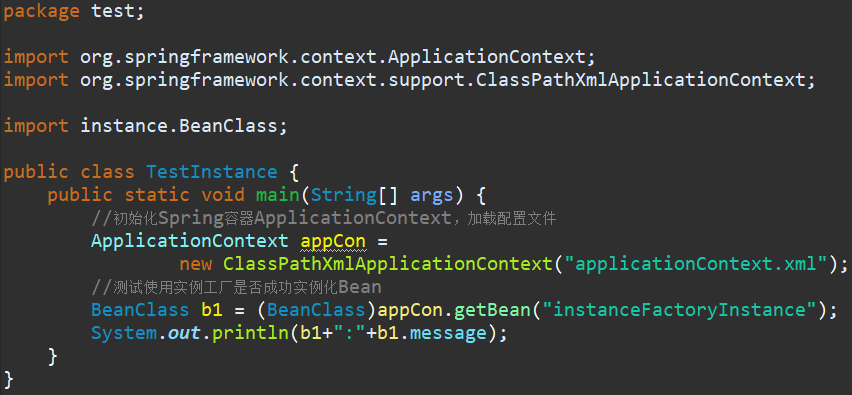
1. **编辑配置文件**

在配置文件applicationContext.xml中通过实例工厂实例化Bean，代码如下：



1. **编写测试代码**

编写测试代码，验证配置的Bean是否可以使用，代码如下：



# **Bean的作用域**

在Spring配置文件中配置Bean时，可以通过<bean>元素的可选属性scope为该Bean 指定作用域，如不指定，则默认为singleton作用域。在Spring中为Bean的实例定义了 6种作用域。如下所示：

|  |  |
| --- | --- |
| **作用域名称** | **描述** |
| singleton | Bean默认的作用域，使用singleton定义的Bean在Spring容器中只有一个Bean实例（单例设计模式的思想），Spring容器每次获取该Bean时，返回的都是同一个实例 |
| prototype | Spring容器每次获取prototype定义的Bean时，容器都将创建一个新的Bean实例，各个实例都不相同。 |
| request | 在一次HTTP请求中容器将返回一个Bean实例，不同的HTTP请求返回不同的Bean实例。仅在Web Spring应用程序上下文使用。 |
| session | 在一个HTTP Session中，容器将返回一个Bean实例。仅在Web Spring应用程序上下文使用。 |
| applicationm | 为每一个ServletContext对象创建一个实例，即同一个应用共享一个Bean实例。仅在Web Spring应用程序上下文使用。 |
| websocket | 为每一个WebSocket对象创建一个Bean实例。仅在Web Spring应用程序上下文使用。 |

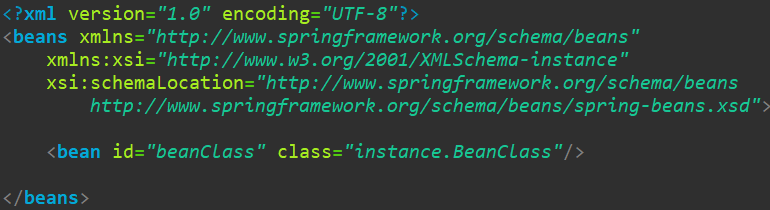
在以上6种作用域中，singleton和prototype是最常用的两种作用域。后面4种仅在 Web Spring应用程序上下文中使用。本节仅对singletion和prototype这两种作用域进行 详细讲解。

## singleton作用域

singleton作用域是Bean默认的作用域。在使用id获取Bean实例时，IoC容器将返回 同一个Bean实例。验证如下：

1. **定义singleton作用域的Bean**

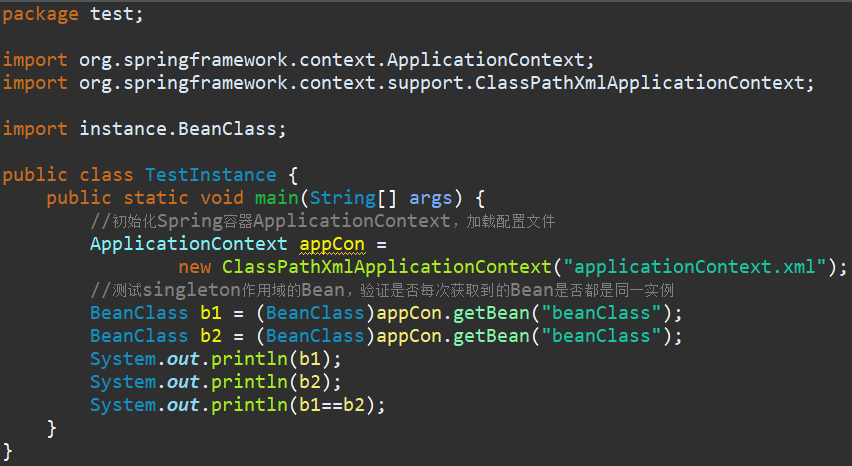
在前面创建的ch3应用中，在Spring配置文件中定义作用域为singletion的Bean。但由于singleton是Bean默认的作用域，所以可以不使用scope属性指定，当然也可以使用singleton属性指定。代码如下：



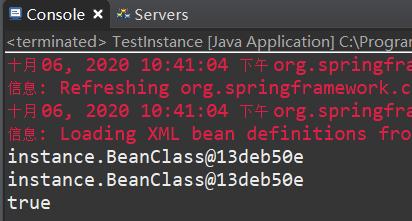
定义了一个id为beanClass，类型为instance.BeanClass，作用域为singleton的Bean。

**2）测试singleton作用域**

编写程序，测试singleton作用域，代码如下：



运行结果：



从运行结果中可以看出，获取作用域为singleton的Bean时，容器仅返回的同一个 实例。

## prototype作用域

当将Bean的作用域设置为prototype时，Spring IoC容器将为每次请求都分别创建一个 新的实例，各实例都是不同的。验证如下：

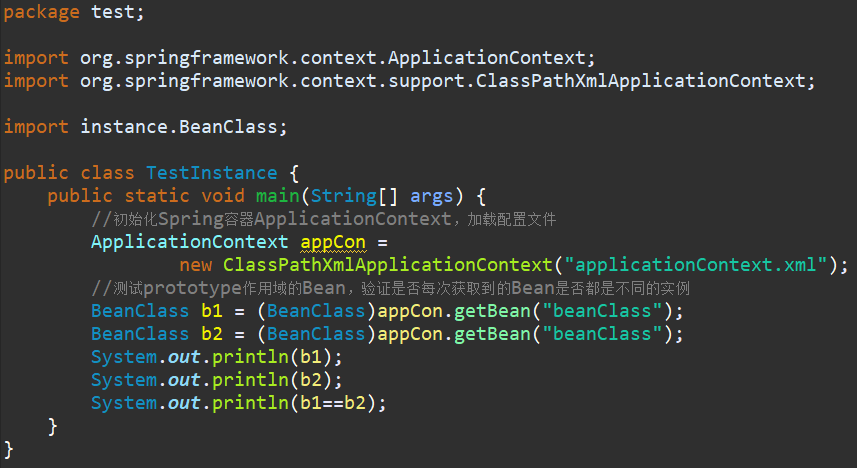
1. **定义prototype作用域的Bean**

在前面创建的ch3应用中，在Spring配置文件中定义作用域是prototype的Bean，代码如下：

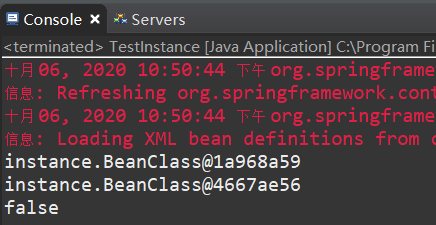


1. **测试prototype作用域**

编写程序，测试singleton作用域，代码如下：



运行结果如下：



从运行结果可以查看，每次获取到的作用域为prototype的Bean都是不同的。

# Bean的生命周期

## 认识Bean的生命周期

一个对象的生命周期包括创建（实例化与初始化）、使用以及销毁等阶段，在Spring 中，Bean对象周期也遵循这一过程，并且在Spring提供了许多对外接口，允许开发 者对3个过程（实例化、出实例化、销毁）的前后做一些操作。在Spring Bean中， 实例化是为Bean对象开辟空间，初始化则是对Bean属性的初始化。、

Spring容器虽然可以管理Bean的生命周期，但是只能管理singleton作用域的Bean。 在singleton作用域下，Spring能够精确地知道Bean何时被创建，何时初始化完成， 以及何时被销毁。而对于prototype作用域的Bean，Spring只负责创建，当容器创建 了Bean的实例后，Bean的实例就交给了客户端的代码管理，Spring容器将不再跟踪 其生命周期，并且不会管理那些被配置成prototype作用域的Bean。

Spring中的Bean的生命周期的整个过程如下：

（1）根据Bean的配置情况实例化一个Bean；

（2）根据Spring上下文对实例化的Bean进行依赖注入，即对Bean的属性进行初

始化；

（3）如果Bean实现了BeanNameAware接口，将调用它实现的setBeanName（String beanId）方法，此处参数传递的是Spring配置文件中Bean的id

（4）如果Bean实现了BeanFactoryAware接口，将调用它实现的setBeanFactory方法，

此处参数传递的是当前Spring工厂实例的引用；

（5）如果Bean实现了ApplicationContextAware接口，将调用它实现的

setApplicationContext（ApplicationContext）方法，此处参数传递的是Spring上下 文实例的引用；

（6）如果Bean关联了BeanPostProcessor接口，将调用初始化方法

postProcessBeforeInitialization（Object obj，String s）方法对Bean进行操作；

（7）如果Bean实现了InitializingBean接口，将调用afterPropertiesSet方法；

（8）如果Bean在Spring配置文件中配置了init-method属性，将自动调用其配置

的初始方法（init-method属性用于指定方法名）；

（9）如果Bean关联了BeanPostProcessor接口，将调用postProcessAfterInitialization （Object obj，String s）方法，由于是在Bean初始化结束时调用After方法，也 可用与内存或缓存技术。

注意：以上工作完成后就可以使用该Bean，由于该Bean的作用域是singleton，

所以调用的是同一个Bean实例。

（10）当Bean不再需要时将进入销毁阶段，如果Bean实现了DisposableBean接

口，则调用其实现的destroy方法将Spring中的Bean销毁；

（11）如果在配置文件中配置Bean时通过destroy-method属性指定了Bean的销

毁方法，将调用其配置的销毁方法进行销毁。

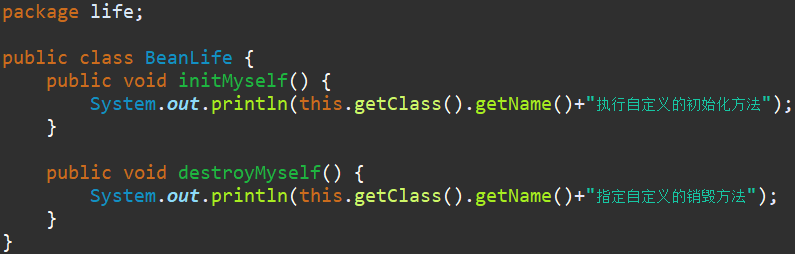
在Spring中，通过实现特定的接口或通过<bean>元素的属性设置可以对Bean的生命周 期过程产生影响。开发者可以随意地配置<bean>元素的属性，但不建议过多地让Bean 实现接口。因为这样将使代码和Spring聚合比较紧密（因为这些接口是Spring框架提 供的）。

## 示例

通过一个示例来演示Bean的生命周期。

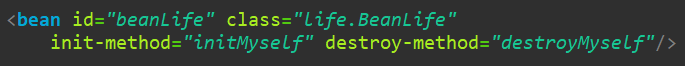
1. **创建Bean的实现类**

在先前创建的ch3应用中的src目录下创建life包，在life包下创建BeanLife类，在BeanLife类中定义两个方法，一个演示初始化过程，一个演示销毁过程。具体代码如下：



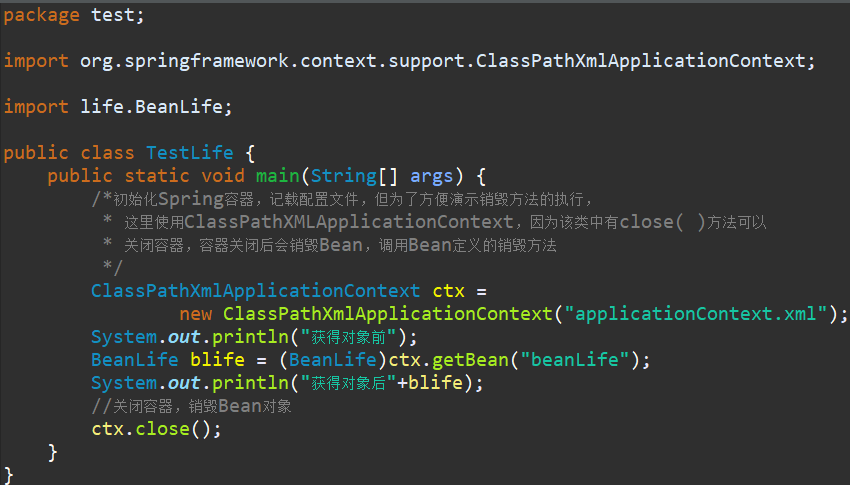
1. **配置Bean**

在Spring配置文件中配置该Bean，并指定该Bean的初始化方法和销毁方法。代码如下：

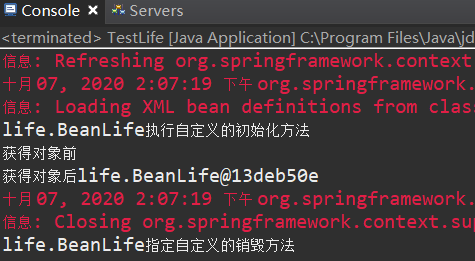


1. **测试Bean的生命周期**

在ch3应用的test包中创建测试类TestLife，测试Bean的生命周期。代码如下：



运行结果如下：



从运行结果可以看出，在加载配置文件时执行了Bean的初始化方法initMyself；在 获得对象后，调用close( )方法关闭容器时，执行了Bean的销毁方法destroyMyself。

# Bean的装配方式

Bean的装配方式指的是将Bean注入Spring IoC容器中的方式，也可以理解为配置 Bean的方式。在前面的学习中，我们都是通过XML文件即配置Bean，而实际上，Spring 容器支持基于XML配置的装配、基于注解的装配以及自动装配等多种装配方式。其中 最受青睐的装配方式是基于注解的装配。

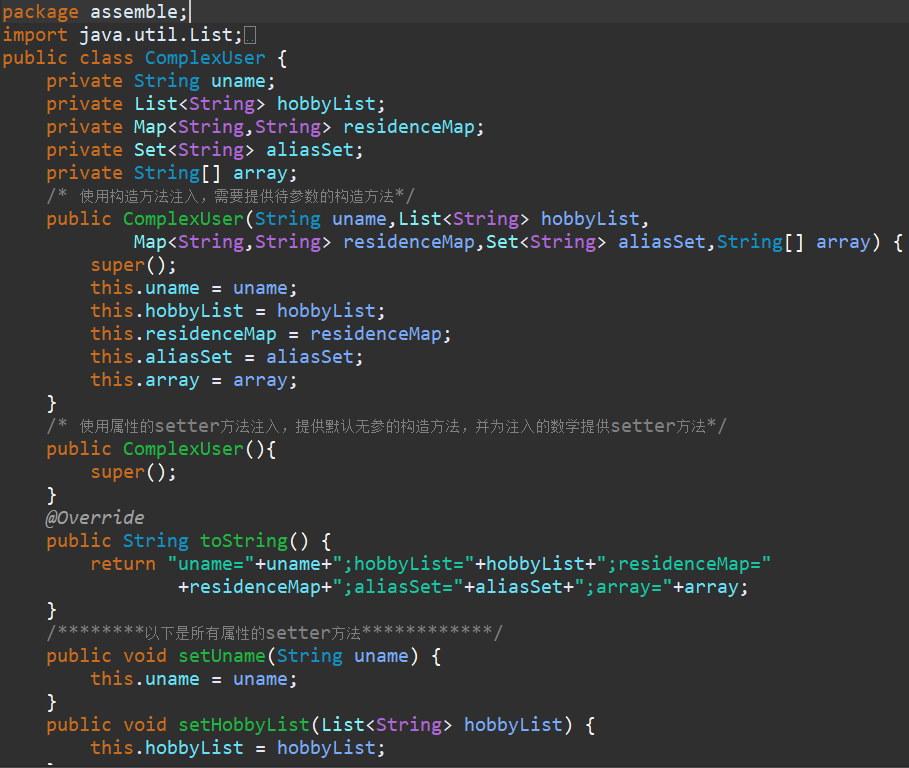
## 基于XML配置的装配

基于XML配置的装配方式是指在Spring配置文件中通过<bean>标签来配置一个Bean， 将Bean装配到Spring容器中。

示例如下：

1. **创建Bean的实现类**

在ch3应用的src目录中创建assemble包，在assemble包下创建ComplexUser类，在ComplexUser类中提供构造方法和属性成员的setter方法，用于分别演示使用构造方法注入和使用属性的setter方法注入。具体代码如下：



1. **配置Bean**

在Spring配置文件中配置类型为ComplexUser的Bean；根据依赖注入的两种方式，配置两个示例。代码如下：

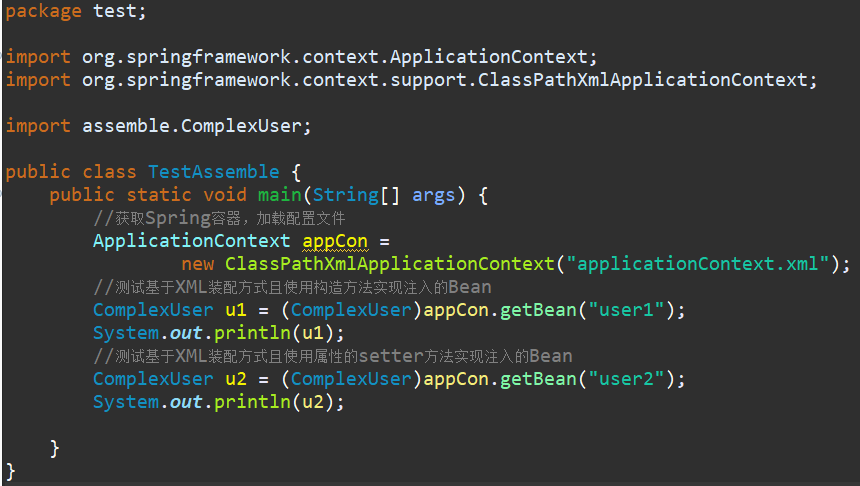
1. 使用构造方法实现注入，配置Bean，如下：

1. 使用属性的setter方法注入，配置Bean，代码如下：

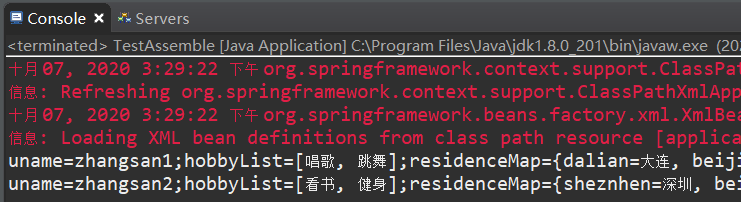


1. **测试基于XML配置的装配方式**

在ch3应用的test包中创建测试类TestAssemble，具体代码如下：



运行结果如下：



成功获取Bean。

## 基于注解的装配

### 认识

1）在Spring框架中，尽管使用XML配置文件可以很简单地配置Bean，但如果应用中 有大量的Bean需要装配，会导致XML配置文件过于庞大，不方便日后的升级与维 护，因此更多的时候推荐开发者使用注解 ( annotatino )的方式去装配Bean。

1. 在使用注解的方式装配Bean之后，Spring容器并不知道去哪里扫描Bean对象，所

以还需要在Spring配置文件中使用<context:component-scan>标签扫描包，使得包中类中的注解生效。

但在Spring4.0以上的版本，<context:component-scan>需要spring-aop-5.0.2.RELEAS E.jar包的支持。所以还需要将spring-aop-5.0.2.RELEASE.jar导入 WEB-INF/lib目录 中。

1. Spring中定义了一系列的注解，用于装配Bean和实现依赖注入。

|  |  |
| --- | --- |
| **注解名称** | **描述** |
| @Component | 定义一个Bean |
| @Repository | 与@Component作用相同，专门用于标注数据访问层 |
| @Service | 与@Compoennt作用相同，专门用于标注业务层 |
| @Controller | 与@Component作用相同，专门用于标注控制层 |
| @Autowired | 实现依赖注入，根据类型自动注入。如需要手动注入，需要结合@Qualifier一起使用 |
| @Resource | 实现依赖注入，根据给定的名称去寻找Bean实现注入 |
| @Qualifier | 实现依赖注入，与@Autowired配合使用，根据指定Bean的名称实现注入 |
| @Value | 实现依赖注入，标注在属性上，如@Value(“张三”)，用于基本数据类型（包括对应的包装类）和字符串类型的数据 |

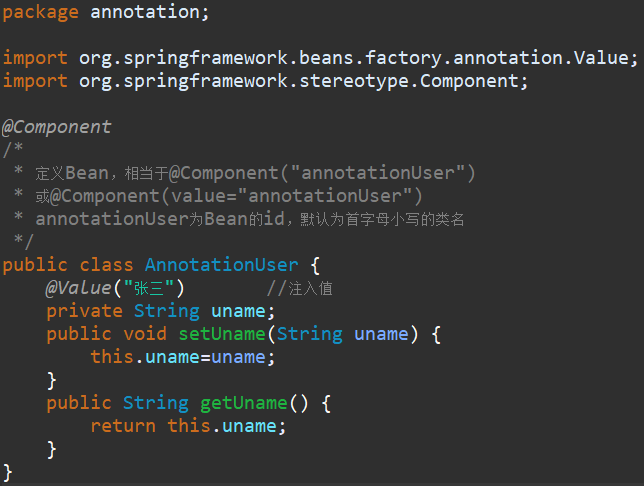
虽然在基于注解的装配方式中提供了实现依赖注入的注解，如@Autowired、@Value 等，但对于一些复杂对象的注入目前还解决不了，如注入数组、List、Map等类 型的对象。

### @Component

该注解仅仅表示一个组件对象（Bean），可以作用在任何层次，相当于XML配置文件中 的<bean>元素的作用。Bean的名称默认是首字母小写的类名；也可以自定义Bean名称， 如**@Component(“test”)或Component(value=”test”)**，表示Bean的名称为”test”。

1. **创建Bean的实现类**

在前面创建的ch3应用中的src目录下创建annotatio包，在该包下创建Bean的实现类AnnotationUser，代码如下：

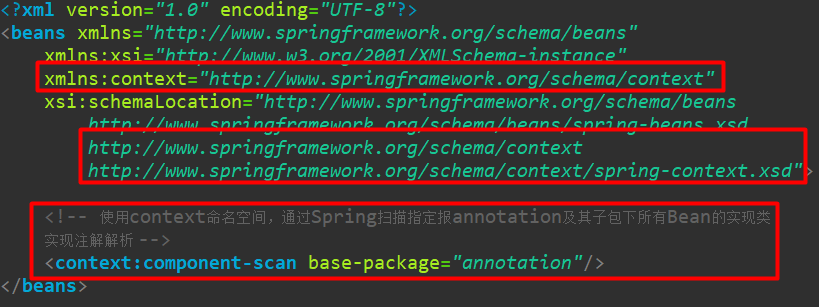


1. **配置注解（扫描包）**

使用注解配置Bean后，Spring容器并不知道去哪里扫描Bean对象，需要在配置文件中配置注解，语法如下：

**<context:component-scan base-package=”Bean所在的包路径”/>**

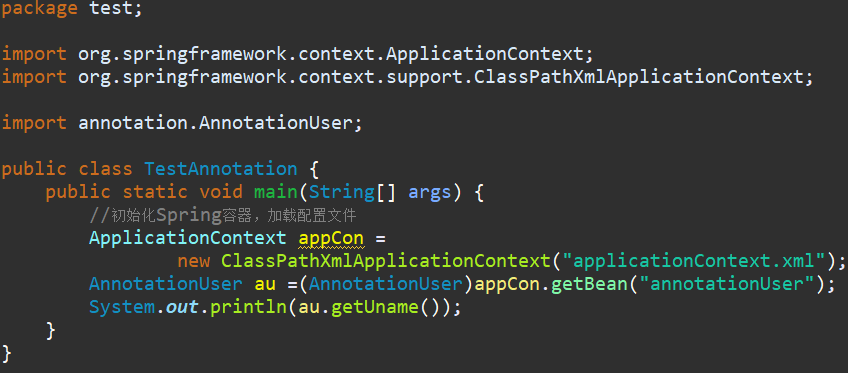
在Spring配置文件中扫描，代码如下：



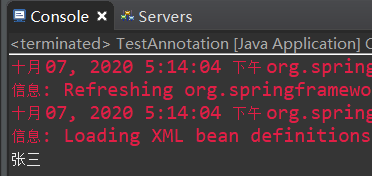
扫描包需要引入context命令空间且要注意定义顺序，同时扫描包还需要将 spring-aop.RELEASE.jar包放入WEB-INF/lib目录中。

1. **测试Bean实例**

在test包中创建测试类TestAnnotation，测试通过注解方式装配的Bean是否可用，具体代码如下：



运行结果如下：



### 分层注解

分层注解指的是@Repository、@Service、@Controller这三个注解。这三个注解的功 能与@Component注解的功能相同，不同的是这三个注解倾向于分层，使类的标注更加 清晰和层次化，让开发者一眼便知这个类是属于哪一个层的。如下：

1）@Repository：用于注解持久层；

2）@Service：用于标注业务逻辑层；

3）@Controller：用于标注控制组件类（Spring MVC的Controller）。

### 依赖注入注解

1）@Autowired：该注解可以对类成员变量、方法及构造方法进行标注，完成自动装配

的工作。通过使用@Autowired来消除setter和getter方法。默认按照

Bean的类型进行装配。

2）@Resource：该注解与@Autowired注解功能相同，区别在于该注解默认是按照名称

来装配注入的，只有当找不到与名称匹配的Bean时才会按照类型来装

配注入；而@Autowired默认按照Bean的类型进行装配，如果想按照名

称来装配注入，则需要和@Qualifier注解一起使用。

该注解有两个属性：name属性和type属性；name属性用于指定Bean 的id名 称，即按照名称来装配注入；type属性指定Bean类型，即按 照Bean的类型进行装配。

3）@Qualifier：该注解与@Autowired注解配合使用。当@Autowired注解需要按照名称

来装配注入时需要和该注解一起使用，Bean的实例名称由@Qualifier 注解的参数指定。

4）@Value：该注解用于注入基本数据类型（包括对应的包装类）和String类型数据。

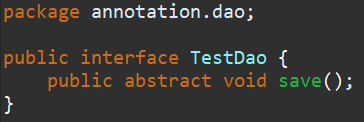
### 使用示例

通过一个示例来演示如何使用这些注解。

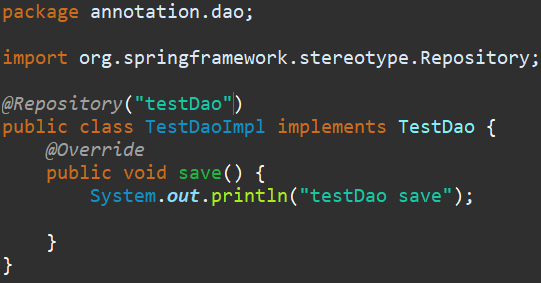
1. **创建DAO层**

在前面创建的ch3应用中的src目录中创建annotation.dao包，在该包下创建TestDao接口和TestDaoImpl实现类，并将实现类TestDaoImpl使用@Repository注解标注为数据访问层。

TestDao接口代码如下：



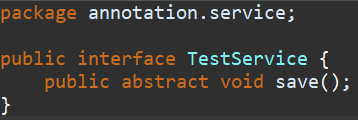
TestDaoImpl接口实现类如下：



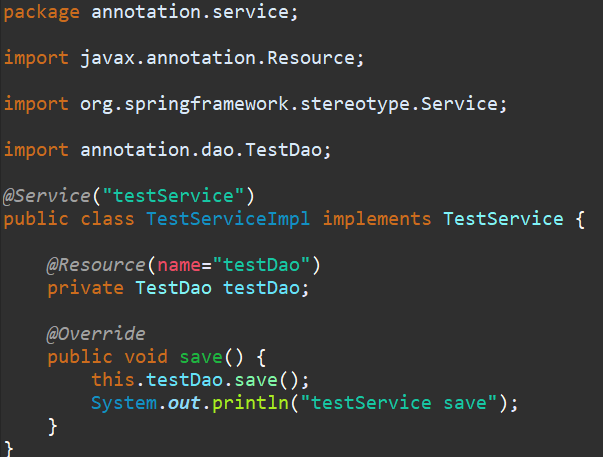
1. **创建Service层**

在先前创建的ch3应用中的src目录下创建annotation.service包，在该包下创建TestService接口和TestServiceImpl实现类，并将实现类TestServiceImpl实现@Service注解标注为业务逻辑层。

TestService接口代码如下：



TestServiceImpl实现类代码如下：

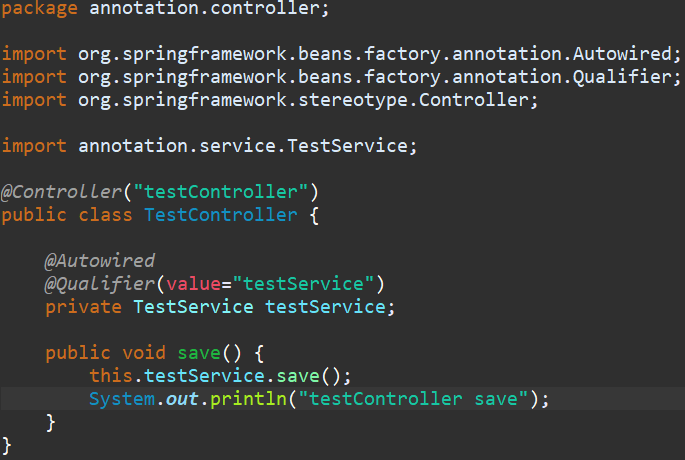


在TestServiceImpl类中使用@Resource注解进行了注入。

1. **创建Controller层**

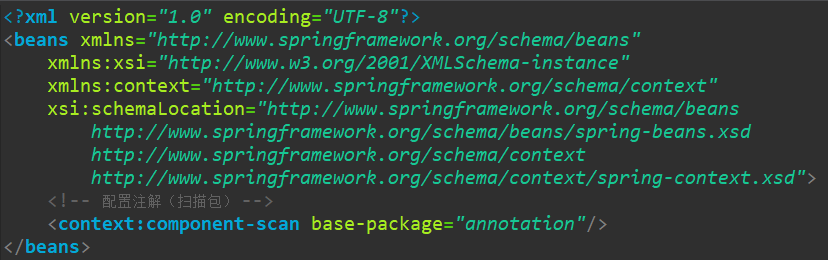
在先前创建的ch3应用中的src目录下创建annotation.controler包，在该包中创建TestController类，并将TestController类使用@Controller注解标注为控制层。

TestController的代码如下：



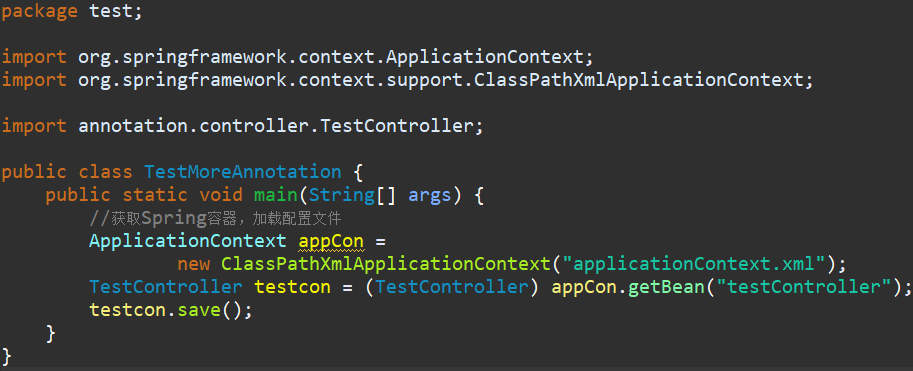
1. **配置注解（扫描包）**

在Spring配置文件中配置注解，扫描包，由于Dao层、Service层、Controller层都在annotation包下，所以只需要要扫描annotation包即可，因为它也会自动扫描其下的所有子包。



1. **创建测试类**

在src目录下的test包中创建测试类TestMoreAnnotation，代码如下：



运行结果如下：

