

# Spring配置： 详解

## SpringIOC:

按照我个人的理解来说，SpringIOC（Inversion of Control）控制反转：指就是说原先在对象中要使用另一个对象就必须显式的去创建另一个对象的实例，例如通过构造方法或者是调用工厂方法（工厂方法最终也是需要new，因为这是Java创建对象所必须的）来获得。而Spring提供了IOC容器来帮我们生成所需要的对象。也就是说在我们原先的对象中有用到其他对象的地方Spring会帮我们注入。不用我们再去考虑这些问题。这边注入的实现手段就是DI（Dependency Injection）依赖注入。DI的方式有两种，一种是通过配置文件XML，一种是通过注解Annotation。下面会为两种进行说明。

## SpringIOC容器:

容器是SpringIOC的核心它主要有两种:

- **BeanFactory** : BeanFactory为IOC容器提供了基础功能，Spring文档中提到，当前该类仅仅是为了向后兼容老的版本，除非你有更好的原因否则就应该使用第二种容器。
- **ApplicationContext** : 通过API文档可以知道，ApplicationContext是

BeanFactory的子接口，并且从文档中也可以看到ApplicationContext除了包含有BeanFactory的所有功能还支持了更多的功能。

ApplicationContext的实现有四种方式:

1. **FileSystemXmlApplicationContext** : 加载配置文件的时候采用的是项目的路径。
2. **ClassPathXmlApplicationContext** : 加载配置文件的时候根据ClassPath位置。
3. **XmlWebApplicationContext** : 在Web环境下初始化监听器的时候会加载该类。
4. **AnnotationConfigApplicationContext** : 根据注解的方式启动Spring 容器。

Table 6.9. Feature Matrix

Feature	BeanFactory	ApplicationContext
Bean instantiation/wiring	Yes	Yes
Automatic <code>BeanPostProcessor</code> registration	No	Yes
Automatic <code>BeanFactoryPostProcessor</code> registration	No	Yes
Convenient <code>MessageSource</code> access (for i18n)	No	Yes
<code>ApplicationEvent</code> publication	No	Yes

<https://blog.csdn.net/u010890358>

## SpringDI的方式:

Spring提供了三种方式来依赖注入:

1. 构造方法注入
2. setter方法注入
3. 接口注入

其中Spring以往推荐使用Setter的方法现在改成推荐构造方法注入。使用构造方法注入需要注意的一点就是要避免循环依赖。所谓的循环依赖指的就是在A对象的构造方法中Spring要注入B, 而在B对象中Spring要注入A。这个时候会形成一个闭环因为Spring不知道该先注入哪一个接着会抛出异常。而Spring建议的处理方式是说如果遇到这种情况的话就改用Setter方式注入。

## SpringIOC之Xml:

### Bean:

省去了所有的测试环境的搭建本次使用Junit进行单元测试使用ClassPathXmlApplication来管理bean。首先从把类交给Spring Ioc管理开始一步步深入。从下面实例可以看到我们要让Spring帮我们管理对象的生成需要在配置文件中注册。每一个标签bean对应的就一个类对象, 通过id(或者Name)以及Class来定位一个类, 这边Class要求是完全限定类名。因为Spring生成对象的过程就是使用反射机制所以你必要要提供一个正确的路径否则就会抛出异常。除了这两个属性外, Spring还提供了其他的属性来对Bean进行设置下面将对所有的属性进行测试。

```
<?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"
       xmlns:aop="http://www.springframework.org/schema/aop"
       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
                           http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd">

    <bean id="userDaoImpl" class="com.ctc.DaoImpl.UserDaoImpl"></bean>

    <bean id="userServiceImpl" class="com.ctc.ServiceImpl.UserServiceImpl" autowire="byType">
    </bean>

</beans>
```

Finished after 0.762 seconds

Runs: 1/1 Errors: 0 Failures: 0

Xml [Runner: JUnit 4] (0.011 s)

```
1* import java.lang.reflect.InvocationHandler;
15
16 public class SpringTest {
17
18     private static ApplicationContext cx ;
19
20
21     @BeforeClass
22     public static void getInstance(){
23         if(cx == null){
24             cx = new ClassPathXmlApplicationContext("AnnotationBean.xml");
25             //cx = new ClassPathXmlApplicationContext("bean.xml");
26         }
27     }
28
29     @Test
30     public void Xml(){
31         UserService userService = (UserService) cx.getBean("userServiceImpl");
32         UserDao userDao = (UserDao) cx.getBean("userDaoImpl");
33     }
34 }
```

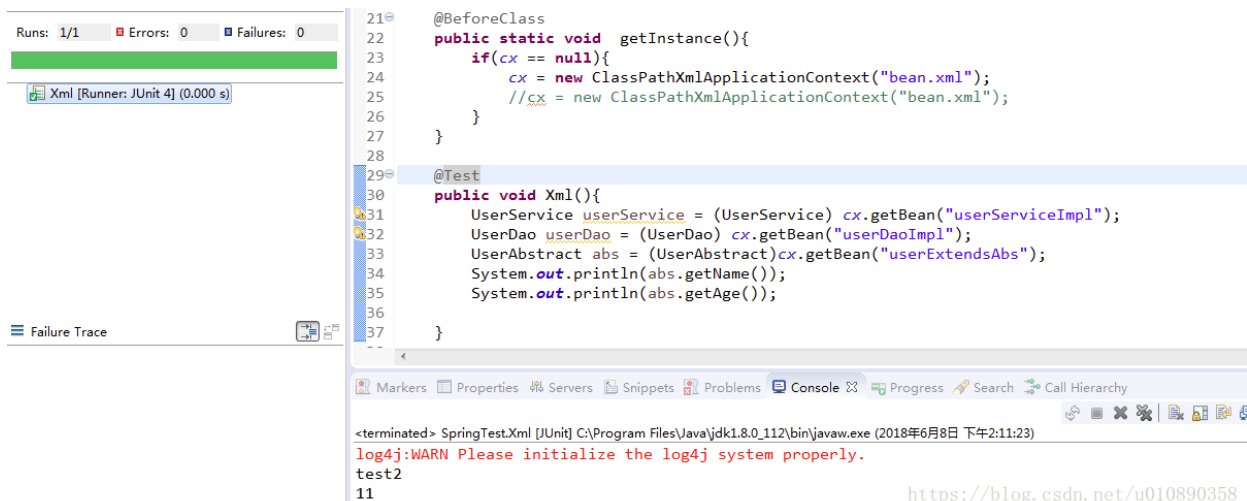
## abstract以及parent:

abstract以及parent是搭配使用的，如果bean定义了属性abstract=true，那么这个类就无法通过Spring容器拿到，而是作为一个模板存在。有点类似于设计模式中的模板方法。这个bean对应的类它定义的属性，方法可以被复用。通过Parent属性在子类中可以指向被abstract定义的bean。用法包括集合类的并集，类的继承等等。下面给出栗子，可以看到定义了一个父类UserAbstract还有它的子类UserExtendsAbs。运行的时候子类可以直接拿到父类中定义的属性name以及age。此外Spring允许我们对父类中的值进行覆盖，所以Name输出变成了test2。

```
<bean id="userDaoImpl" class="com.ctc.DaoImpl.UserDaoImpl"></bean>

<bean id="userServiceImpl" class="com.ctc.ServiceImpl.UserServiceImpl" autowire="byType" >
</bean>

<bean id="userAbs" abstract="true" class="com.ctc.Abstract.UserAbstract">
<property name="name" value="test"></property>
<property name="age" value="11" ></property>
</bean>
<bean id="userExtendsAbs" parent="userAbs" class="com.ctc.Abstract.UserExtendsAbs">
<property name="name" value="test2"></property>
</bean>
```



## lazy-init, init-method, destroy-method:

默认情况下容器中的所有bean在初始化容器的时候就被加载，但是我们可以指定lazy-init属性来让Spring实现懒加载。只有在调用的时候才添加进容器。而init-method以及destroy-method顾名思义就是初始化以及销毁的时候调用，类似于Servlet的init以及destroy。下面举个栗子。因为UserServiceImpl跟UserDaoImpl有关联关系所以注释掉。可以看到程序开始容器初始化的时候没有对userDaoImpl初始化，而当调用了userDaoImpl的时候才会被加入容器里面。此时也可以看到init方法被调用，在容器关闭的时候destroy也被调用了。

```

<bean id="userDaoImpl" class="com.ctc.DaoImpl.UserDaoImpl"
init-method="init" destroy-method="destroy" lazy-init="true"></bean>

```

```

!-- <bean id="userServiceImpl" class="com.ctc.ServiceImpl.UserServiceImpl" autowire="b
</bean> -->

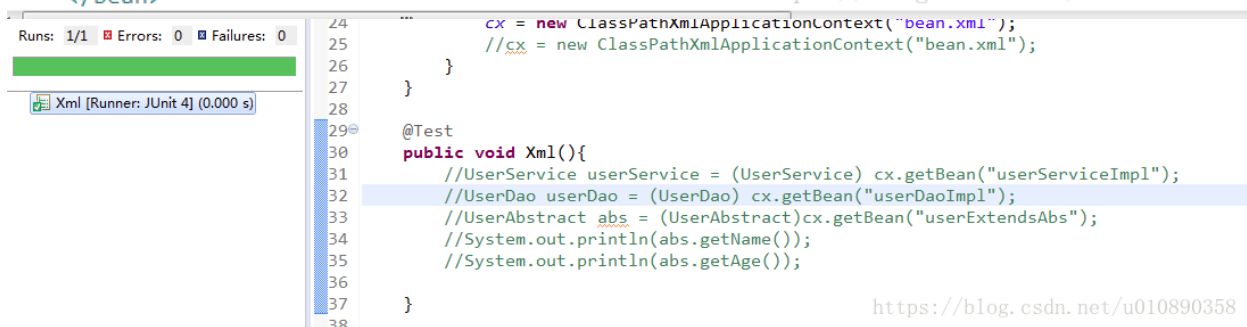
```

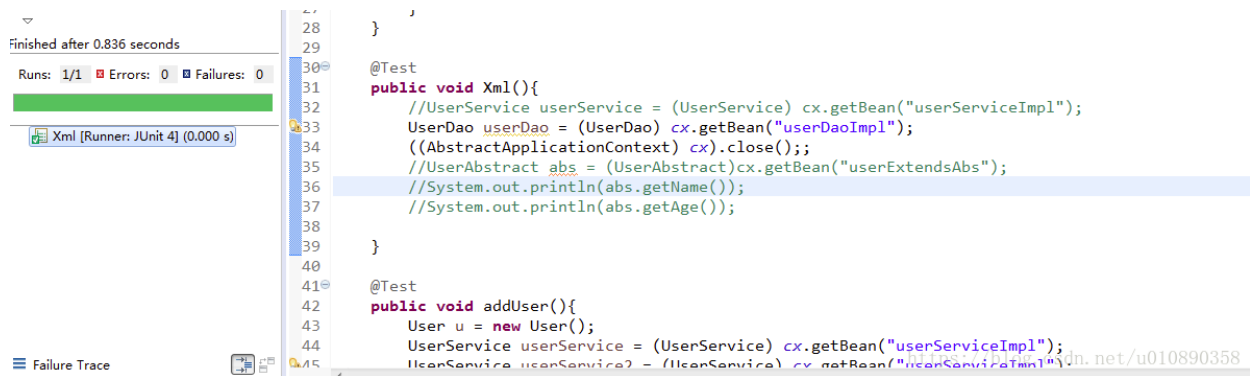
```

<bean id="userAbs" abstract="true" class="com.ctc.Abstract.UserAbstract">
<property name="name" value="test"></property>
<property name="age" value="11" ></property>
</bean>
<bean id="userExtendsAbs" parent="userAbs" class="com.ctc.Abstract.UserExtendsAbs">
<property name="name" value="test2"></property>
</bean>

```

<https://blog.csdn.net/u010890358>





## depends-on:

一般情况下如果两个对象存在依赖的时候我们会通过上述提到的三种注入方式来注入。但是如果两个对象的关系没有那么直接，例如A对象在初始化的时候要调用B对象的一个静态参数，而这个静态参数只有B初始化的时候才会更新。而它们直接的依赖关系并没有那么直接。那么这个时候我们就可以使用 `depends-on`。举个栗子。可以看到默认情况下 `userDaoImpl` 拿到 `number` 为0，而添加了 `depends-on` 后，Spring 会先初始化 `userDaoImpl2` 然后再初始化 `userDaoImpl`，所以 `number` 的值变成了10。

```
public class UserDaoImpl implements UserDao {
```

```
    public UserDaoImpl(){
        System.out.println(UserDaoImpl2.number);
        System.out.println("constructor init");
    }
```

<https://blog.csdn.net/u010890358>

```
public class UserDaoImpl2 implements UserDao {
```

```
    public static int number = 0;
```

```
    public UserDaoImpl2(){
        System.out.println("constructor UserDaoImpl2");
        number = 10;
    }
```

<https://blog.csdn.net/u010890358>

```

Runs: 1/1 Errors: 0 Failures: 0
Xml [Runner: JUnit 4] (0.000 s)

24
25
26
27
28
29
30
31
32
33
34
35
36
37
38

@Test
public void Xml(){
    //UserService userService = (UserService) cx.getBean("userServiceImpl");
    //UserDao userDao = (UserDao) cx.getBean("userDaoImpl");
    //((AbstractApplicationContext) cx).close();
    //UserAbstract abs = (UserAbstract)cx.getBean("userExtendsAbs");
    //System.out.println(abs.getName());
    //System.out.println(abs.getAge());
}

<terminated> SpringTest.Xml [JUnit] C:\Program Files\Java\jdk1.8.0_112\bin\javaw.exe (2018年6月8日 下午2:55:38)
0
constructor init
constructor UserDaoImpl2

```

```

Runs: 1/1 Errors: 0 Failures: 0
Xml [Runner: JUnit 4] (0.000 s)

4
5
6
7
8
9
10
11
12
13
14
15
16
17
18

xmlns:context="http://www.springframework.org/schema/context"
xmlns:aop="http://www.springframework.org/schema/aop"
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
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop."
<!-- <bean autowire="default" autowire-candidate="default" depends-on="" factory-bean="" factory-
/bean> -->
<bean id="userDaoImpl" class="com.ctc.DaoImpl.UserDaoImpl" depends-on="userDaoImpl2"></bean>
<bean id="userDaoImpl2" class="com.ctc.DaoImpl.UserDaoImpl2" ></bean>
<!-- <bean id="userServiceImpl" class="com.ctc.ServiceImpl.UserServiceImpl" autowire="byType" >

Source Namespaces Overview aop beans context
<terminated> SpringTest.Xml [JUnit] C:\Program Files\Java\jdk1.8.0_112\bin\javaw.exe (2018年6月8日 下午2:56:14)
10
constructor UserDaoImpl2
constructor init

```

## factory-bean, factory-method:

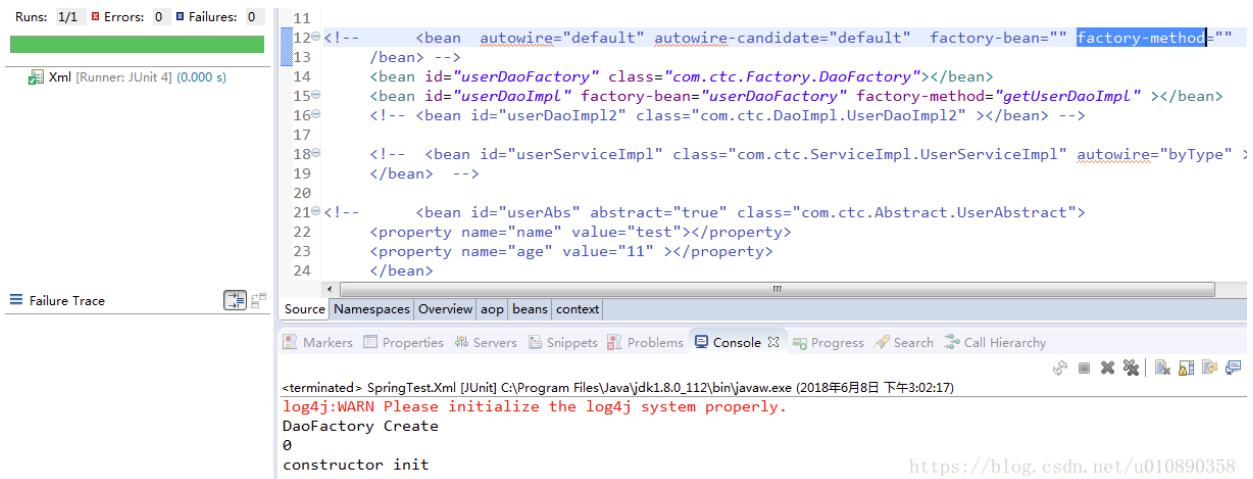
Spring除了通过class来拿到对象，同时也允许对象通过工厂的方法来获取。举个栗子。可以看到userDaoImpl通过DaoFactory工厂拿到。

```

6 public class DaoFactory {
7
8     public UserDao getUserDaoImpl(){
9         System.out.println("DaoFactory Create");
10        return new UserDaoImpl();
11    }
12
13 }
14

```

<https://blog.csdn.net/u010890358>

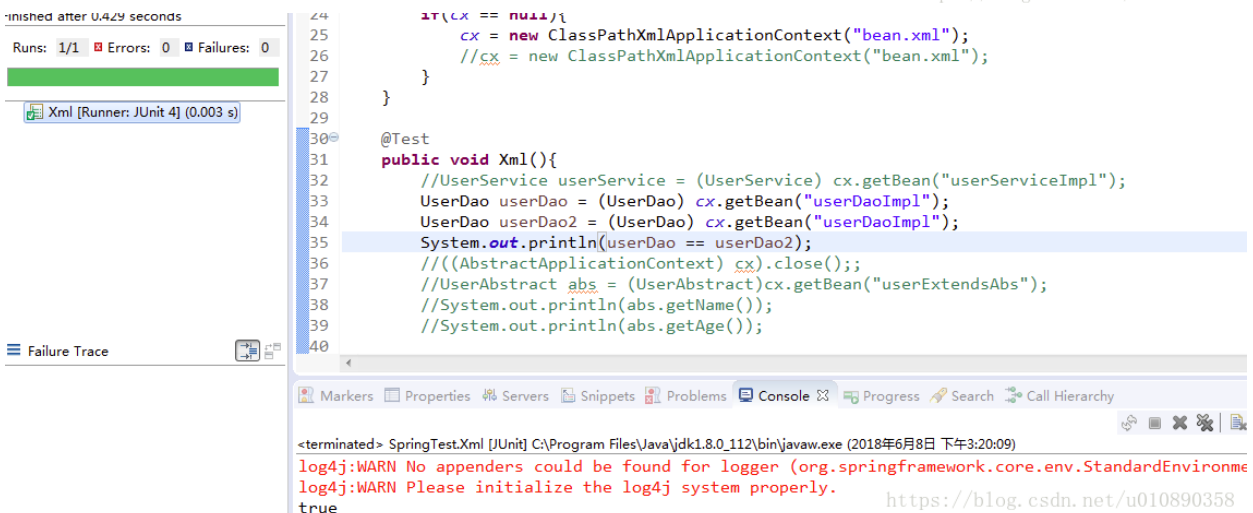


## Scope:

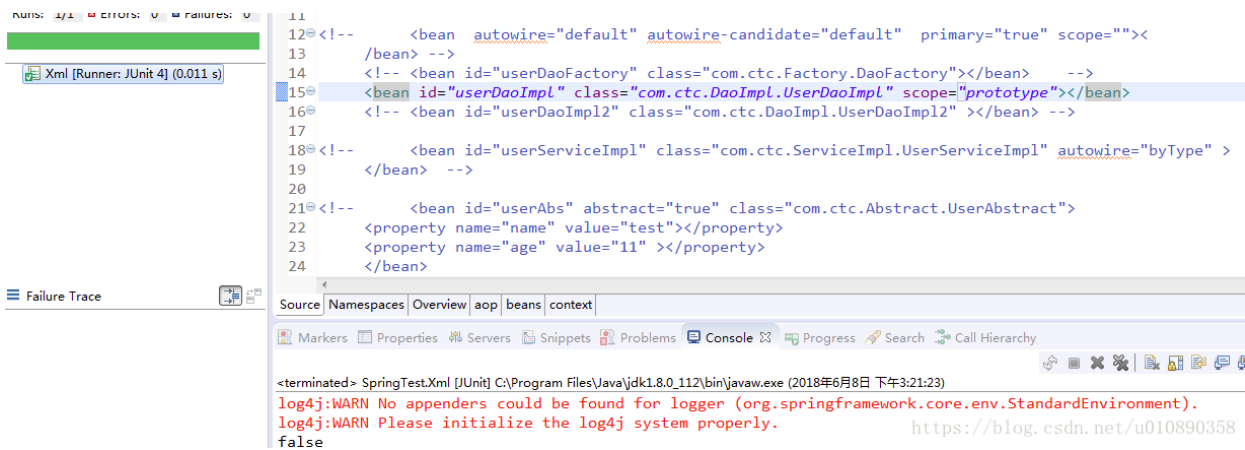
Spring的bean可以设置scope属性来确认它的作用域，可以看到默认情况下是singleton也就是单例，也可以设置成多例。而其他的几种情况都是在web环境下使用。举个栗子。可以看到默认情况下每次取出来的userDaoImpl都是同一个，而我们如果设置成prototype那么每次拿到的对象都是不一样的。

Table 6.3. Bean scopes

Scope	Description
singleton	(Default) Scopes a single bean definition to a single object instance per Spring IoC container.
prototype	Scopes a single bean definition to any number of object instances.
request	Scopes a single bean definition to the lifecycle of a single HTTP request; that is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring <code>ApplicationContext</code> .
session	Scopes a single bean definition to the lifecycle of an HTTP <code>Session</code> . Only valid in the context of a web-aware Spring <code>ApplicationContext</code> .
global session	Scopes a single bean definition to the lifecycle of a global HTTP <code>Session</code> . Typically only valid when used in a portlet context. Only valid in the context of a web-aware Spring <code>ApplicationContext</code> .
application	Scopes a single bean definition to the lifecycle of a <code>ServletContext</code> . Only valid in the context of a web-aware Spring <code>ApplicationContext</code> .







在介绍剩下的autowire, autowire-candidate以及primary属性之前, 先介绍了xml中如何实现依赖注入。

## 构造器注入:

直接举个栗子。现在我们要在对象userServiceImpl中注入对象userDaoImpl。如果我们没有指定的话, 默认情况下是调用默认的构造方法来初始化。可以看到属性值通过构造方法成功注入, 这边Spring允许你通过name来对应构造器参数, 也可以使用index表示对应的位置, 从0开始。或者使用type, 指定类型。

```

public class UserServiceImpl implements UserService {

    private UserDao userDaoImpl;

    private int age;

    private String name;

    public UserServiceImpl(UserDao userDaoImpl , int age , String name) {
        this.userDaoImpl = userDaoImpl;
        this.age = age;
        this.name = name;
    }
}

```

<https://blog.csdn.net/u010890358>

```

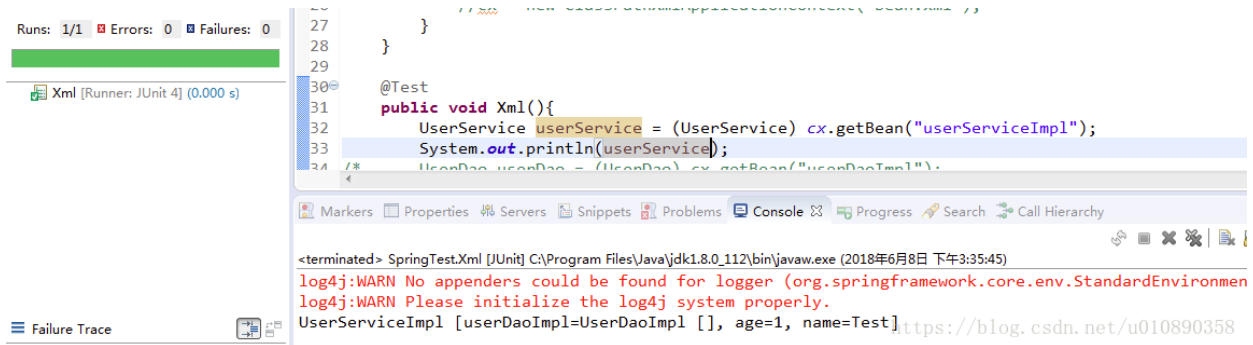
<bean id="userDaoImpl" class="com.ctc.DaoImpl.UserDaoImpl"></bean>
<!-- <bean id="userDaoImpl2" class="com.ctc.DaoImpl.UserDaoImpl2" ></bean> -->

<bean id="userServiceImpl" class="com.ctc.ServiceImpl.UserServiceImpl">
<constructor-arg name="userDaoImpl" ref="userDaoImpl" />
<constructor-arg name="age" value="1" />
<constructor-arg name="name" value="Test" />
</bean>

```

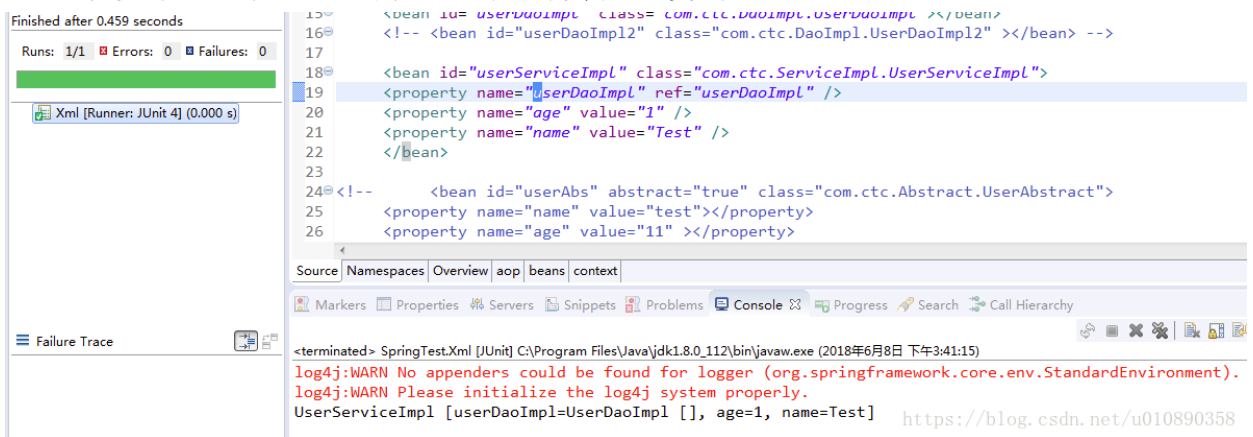
<https://blog.csdn.net/u010890358>





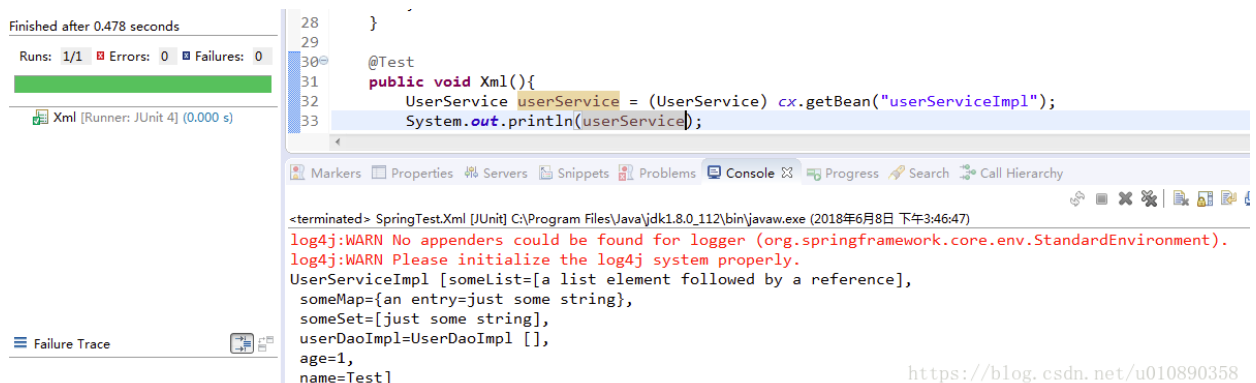
## setter方法注入：

直接举个栗子。之前的例子通过构造方法注入，现在我们直接使用标签property的方式来注入。可以看到一样可以注入成功。但是切记，所有的成员变量必须实现它的setter方法，否则会抛出异常。



## Collection注入：

在对象中经常要使用到Collection集合类型，Spring也提供了对象的方式来注入。举个栗子。



```

public class UserServiceImpl implements UserService {

    private List<String> someList;

    private Map<String,String> someMap;

    private Set<String> someSet;

    private UserDao userDaoImpl;

    private int age;

    private String name;

    public UserServiceImpl(UserDao userDaoImpl , int age , String name) {
        this.userDaoImpl = userDaoImpl;
        this.age = age;
        this.name = name;
    }
}

```

<https://blog.csdn.net/u010890358>

```

<bean id="userServiceImpl" class="com.ctc.ServiceImpl.UserServiceImpl">
<property name="userDaoImpl" ref="userDaoImpl" />
<property name="age" value="1" />
<property name="name" value="Test" />
<!-- results in a setSomeList(java.util.List) call -->
<property name="someList">
    <list>
        <value>a list element followed by a reference</value>
    </list>
</property>
<!-- results in a setSomeMap(java.util.Map) call -->
<property name="someMap">
    <map>
        <entry key="an entry" value="just some string"/>
    </map>
</property>
<!-- results in a setSomeSet(java.util.Set) call -->
<property name="someSet">
    <set>
        <value>just some string</value>
    </set>
</property>
</bean>

```

<https://blog.csdn.net/u010890358>

Finished after 0.478 seconds  
Runs: 1/1 Errors: 0 Failures: 0  
Xml [Runner: JUnit 4] (0.000 s)

```

28     }
29
30     @Test
31     public void Xml(){
32         UserService userService = (UserService) cx.getBean("userServiceImpl");
33         System.out.println(userService);

```

Markers Properties Servers Snippets Problems Console Progress Search Call Hierarchy

```

<terminated> SpringTest.Xml [JUnit] C:\Program Files\Java\jdk1.8.0_112\bin\javaw.exe (2018年6月8日 下午3:46:47)
log4j:WARN No appenders could be found for logger (org.springframework.core.env.StandardEnvironment).
log4j:WARN Please initialize the log4j system properly.
UserServiceImpl [someList=[a list element followed by a reference],
someMap={an entry=just some string},
someSet=[just some string],
userDaoImpl=UserDaoImpl [],
age=1,
name=Test]

```

<https://blog.csdn.net/u010890358>

现在回过头来说说前面搁置的几个属性。

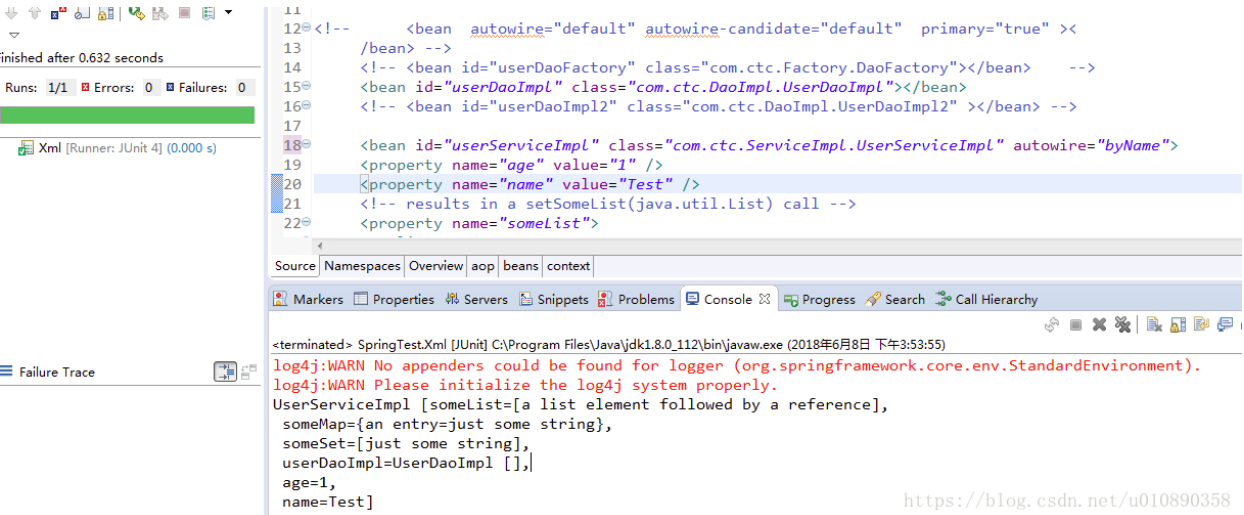
autowire:

Spring有个自动装配的机制会帮我们自动的把userDaoImpl注入到userServiceImpl中，通过配置autowire可以实现。直接举个栗子。可以看到我们在配置文件中删掉了userDaoImpl,但是Spring一样会帮我们成功注入。但是要注意的是，如果我们选择的是byType,那么对于存在两个相同Class的userDaoImpl Spring会不知道要注入哪一个，因此会抛出异常。

Table 6.2. Autowiring modes

Mode	Explanation
no	(Default) No autowiring. Bean references must be defined via a <code>&lt;ref&gt;</code> element. Changing the default setting is not recommended for larger deployments, because specifying collaborators explicitly gives greater control and clarity. To some extent, it documents the structure of a system.
byName	Autowiring by property name. Spring looks for a bean with the same name as the property that needs to be autowired. For example, if a bean definition is set to autowire by name, and it contains a <code>master</code> property (that is, it has a <code>setMaster(..)</code> method), Spring looks for a bean definition named <code>master</code> , and uses it to set the property.
byType	Allows a property to be autowired if exactly one bean of the property type exists in the container. If more than one exists, a fatal exception is thrown, which indicates that you may not use <i>byType</i> autowiring for that bean. If there are no matching beans, nothing happens; the property is not set.
constructor	Analogous to <i>byType</i> , but applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.

<https://blog.csdn.net/u010890358>



```
11
12<!-- <bean autowire="default" autowire-candidate="default" primary="true" ><
13 />bean> -->
14 <!-- <bean id="userDaoFactory" class="com.ctc.Factory.DaoFactory"></bean> -->
15 <bean id="userDaoImpl" class="com.ctc.DaoImpl.UserDaoImpl"></bean>
16 <!-- <bean id="userDaoImpl2" class="com.ctc.DaoImpl.UserDaoImpl2" ></bean> -->
17
18 <bean id="userServiceImpl" class="com.ctc.ServiceImpl.UserServiceImpl" autowire="byName">
19 <property name="age" value="1" />
20 <property name="name" value="Test" />
21 <!-- results in a setSomeList(java.util.List) call -->
22 <property name="someList">
```

Source Namespaces Overview aop beans context

Markers Properties Servers Snippets Problems Console Progress Search Call Hierarchy

<terminated> SpringTest.Xml [JUnit] C:\Program Files\Java\jdk1.8.0\_112\bin\javaw.exe (2018年6月8日 下午3:53:55)

log4j:WARN No appenders could be found for logger (org.springframework.core.env.StandardEnvironment).

log4j:WARN Please initialize the log4j system properly.

UserServiceImpl {someList=[a list element followed by a reference],

someMap={an entry=just some string},

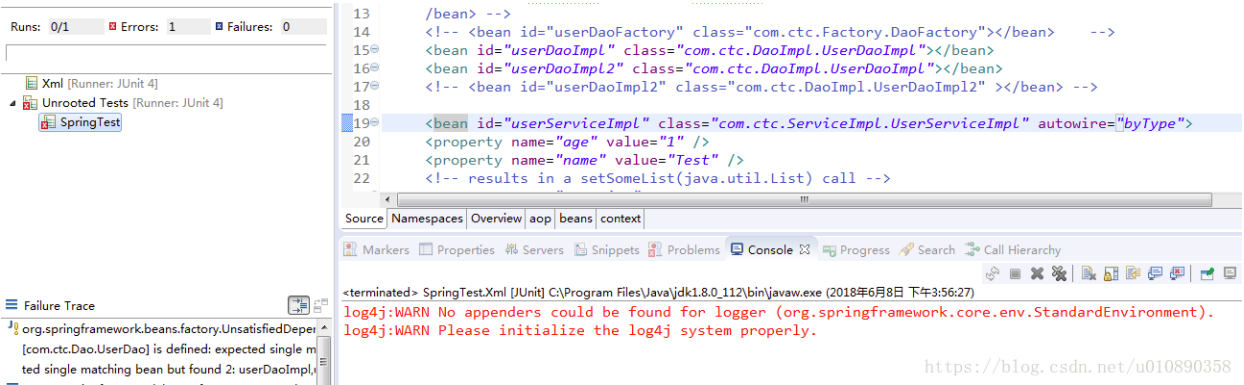
someSet=[just some string],

userDaoImpl=userDaoImpl [],

age=1,

name=Test]

<https://blog.csdn.net/u010890358>



```
13 />bean> -->
14 <!-- <bean id="userDaoFactory" class="com.ctc.Factory.DaoFactory"></bean> -->
15 <bean id="userDaoImpl" class="com.ctc.DaoImpl.UserDaoImpl"></bean>
16 <bean id="userDaoImpl2" class="com.ctc.DaoImpl.UserDaoImpl"></bean>
17 <!-- <bean id="userDaoImpl2" class="com.ctc.DaoImpl.UserDaoImpl2" ></bean> -->
18
19 <bean id="userServiceImpl" class="com.ctc.ServiceImpl.UserServiceImpl" autowire="byType">
20 <property name="age" value="1" />
21 <property name="name" value="Test" />
22 <!-- results in a setSomeList(java.util.List) call -->
```

Source Namespaces Overview aop beans context

Markers Properties Servers Snippets Problems Console Progress Search Call Hierarchy

<terminated> SpringTest.Xml [JUnit] C:\Program Files\Java\jdk1.8.0\_112\bin\javaw.exe (2018年6月8日 下午3:56:27)

log4j:WARN No appenders could be found for logger (org.springframework.core.env.StandardEnvironment).

log4j:WARN Please initialize the log4j system properly.

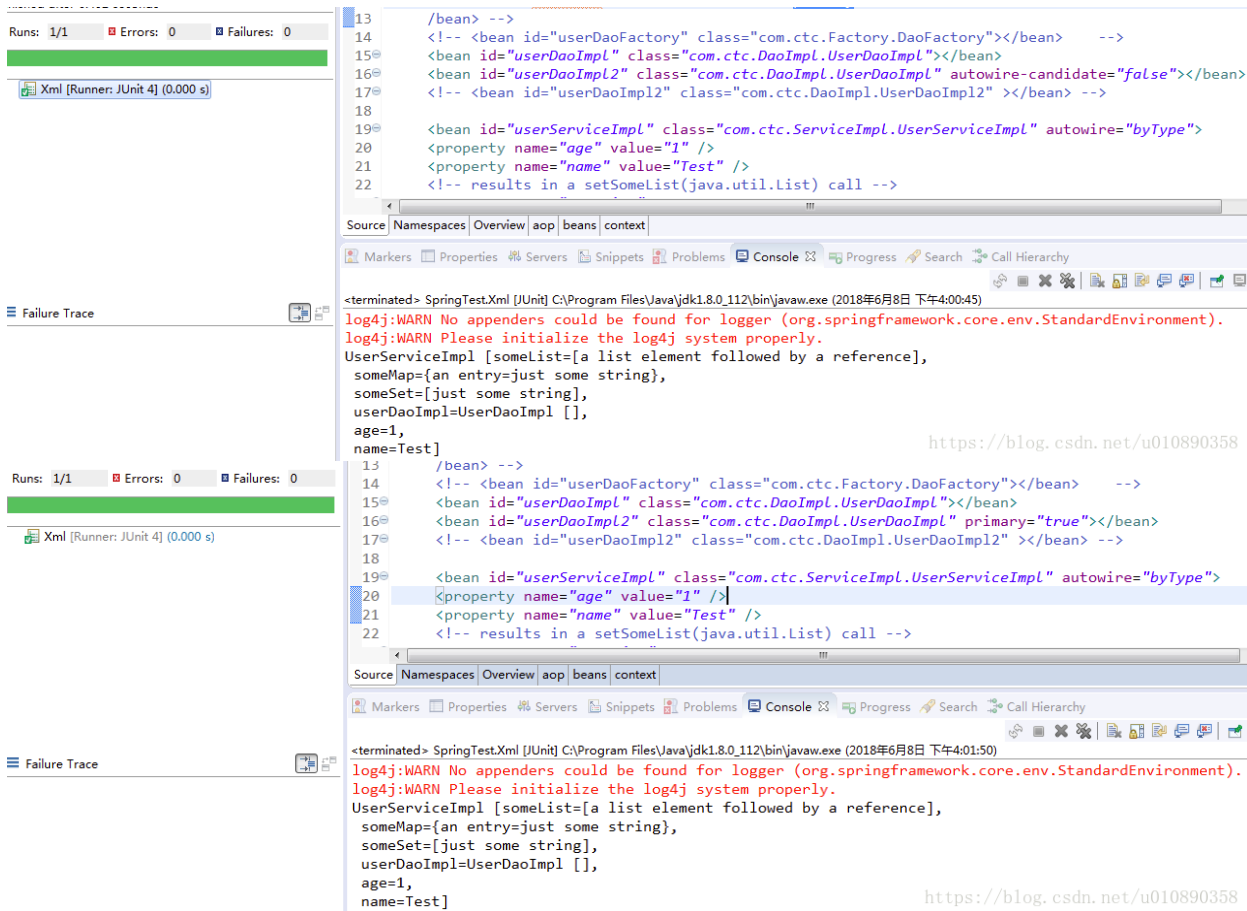
org.springframework.beans.factory.UnsatisfiedDependencyException: [com.ctc.DaoImpl.UserDao] is defined: expected single matching bean but found 2: userDaoImpl, userDaoImpl2

<https://blog.csdn.net/u010890358>

## autowire-candidate, primary:

针对上述的问题，Spring提供了两个方法来解决。第一种如果你在id为userDaoImpl2上面添加了autowire-candidate，那么就说明如果自动装配的时候

出现冲突，就会忽略当前这个bean即userDaoImpl2。而primary则是反过来，它表示的是一种优先级，如果遇到冲突情况优先使用带有该属性的bean。举个栗子。



## SpringIOC之Annotation注解:

使用注解的方式的话就不需要在配置文件中写这么多了，相对Ioc而言我认为使用注解更为方便。如果要使用Annotation的方式只需要在配置文件中添加`<context:component-scan base-package="com.ctc" />`，那么你的基本要求就可以实现了，Spring就根据base-package指定的路径去扫描下面所有的类。我们只需要在业务层数据层上的类添加上注解就可以了。Spring提供了多种注解：`@Component`，`@Service`（业务层），`@Repository`（持久层），`@Controller`（表现层）。当前我们可以在所有的类上面直接使用`@Component`一样可以实现。但是Spring建议你根据需要分开定义，因为在以后的版本中Spring可能会对他们添加一个特性，当前`@Repository`已经支持了针对异常的新特性。

### @Required:

如果我们在`userServiceImpl`中在针对`userDaoImpl`方法上面添加了`Required`那么说明你必须显示的配置文件中指定这个属性。否则就会抛出异常。而Spring的解释是说，这种情况是为了防止出现NPE。举个栗子。可以看到，一

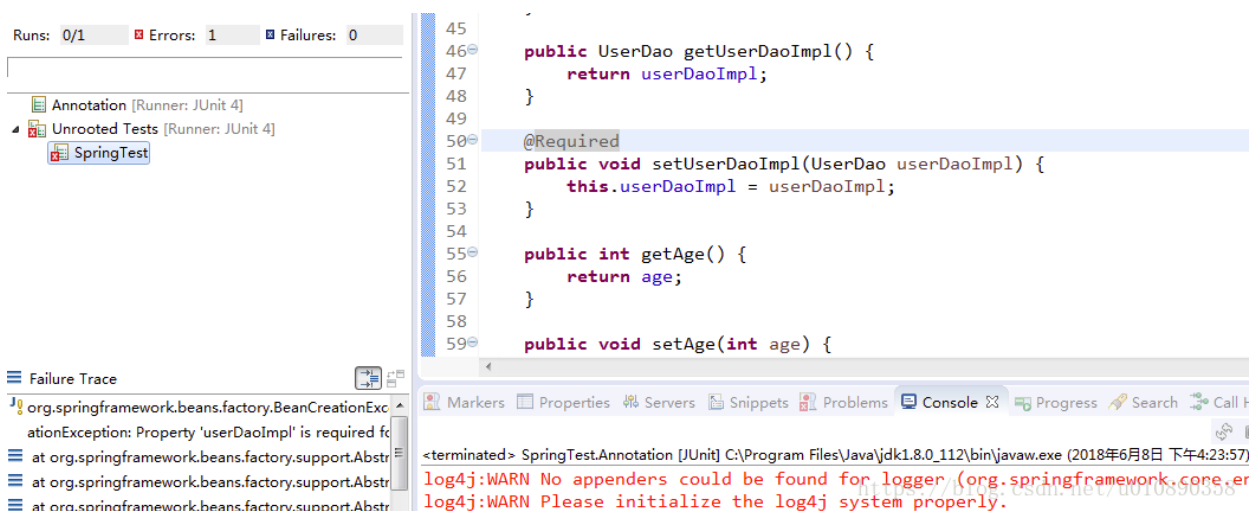
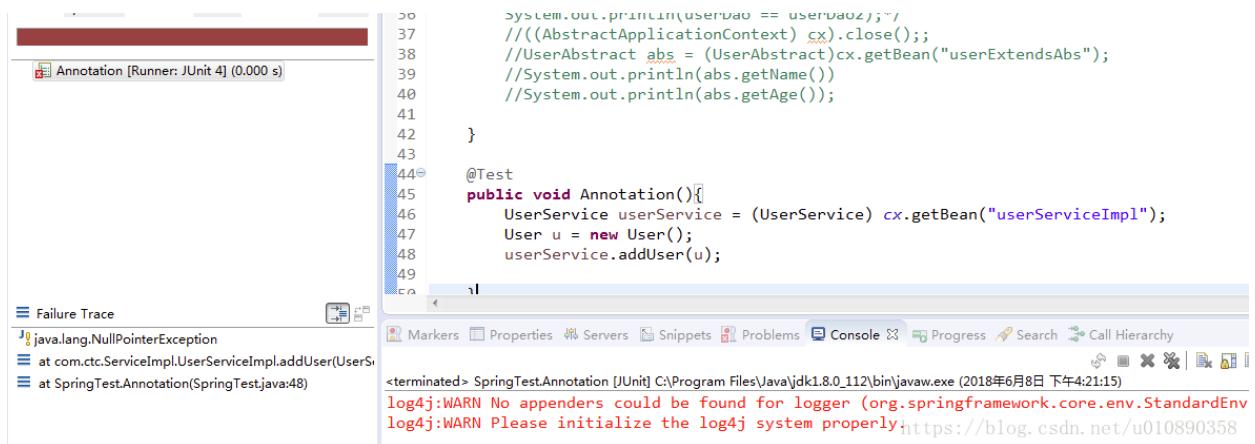
开始我们没有配置Required，Spring没有出现异常这个时候 userDaoImpl 为空，因此在调用 addUser 的时候出现了 NPE。但是如果加入了该注解，可以看到这个时候 Spring 抛出异常了，说明你必须要在配置文件配置，否则无法进行下一步。

```
public void setUserDaoImpl(UserDao userDaoImpl) {  
    this.userDaoImpl = userDaoImpl;  
}
```

<https://blog.csdn.net/u010890358>

```
11  
12     <context:component-scan base-package="com.ctc" />  
13     <bean id="userDaoImpl" class="com.ctc.DaoImpl.UserDaoImpl" />  
14     <bean id="userServiceImpl" class="com.ctc.ServiceImpl.UserServiceImpl">  
15     </bean>  
16 </beans>
```

<https://blog.csdn.net/u010890358>



## @Autowired, @Resource;

@Autowired跟@Resource都可以作用在构造方法，set方法，以及属性值上面注入。它们的区别在于：

1. Autowired是Spring的规范，而Resource是java的规范；

## 2. Autowired默认按类型匹配但是可以搭配@Qualifier来指定名称，而Resource默认按名称；

因为太简单了，就不提供栗子了~此外xml中提到了属性在annotation中都可以实现。需要的话自己查文档吧~

## Spring在3.X之后使用Java来代替XML配置：

Spring在3.X之后推荐使用Java 类的方式来代替XML的配置。下面将进行一个简单的实例。首先创建一个SpringConfig对应一个XML文件。这边Configuration表明当前这个类对应一个XML文件，另外在JUnit中我们不再使用ClassPathXMLApplicationContext，而是改用了AnnotationConfigApplicationContext。因为我们不再通过XML的方式来获取Bean。

@Configuration

@ComponentScan(basePackages="com.ctc")

public class SpringConfig {

}

