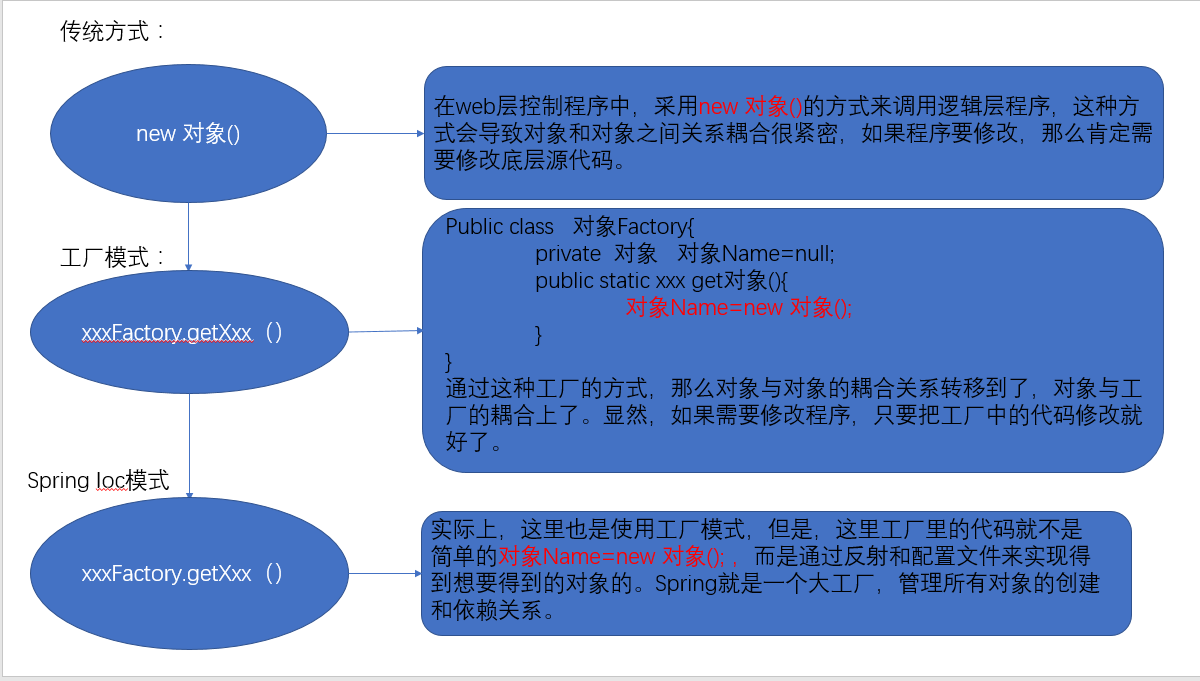
1. Spring的优点

1.1 方便解耦，简化开发：（IOC）

* Spring就是一个大工厂，可以将所有对象创建和依赖关系维护，交给Spring管理。

Web层中的controll中，一般是要调用其他对象，通常任何一个java程序都是要两个及其以上的对象配合完成的，一般来说



1.2. AOP编程的支持

* Spring提供面向切面的编程，可以方便的实现对程序进行：日志记录，性能统计，安全控制，事务处理，异常处理等等
* 主要的意图是：将日志记录，性能统计，安全控制，事务处理，异常处理等代码从业务逻辑代码中划分出来，通过对这些行为的分离，我们希望可以将它们独立到非指导业务逻辑的方法中，进而改  变这些行为的时候不影响业务逻辑的代码。  
  可以通过预编译方式和运行期动态代理实现在不修改源代码的情况下给程序动态统一添加功能的一种技术。

**假设把应用程序想成一个立体结构的话，OOP的利刃是纵向切入系统，把系统划分为很多个模块（如：用户模块，文章模块等等），而AOP的利刃是横向切入系统，提取各个模块可能都要重复操作的部分（如：权限检查，日志记录等等）。由此可见，AOP是OOP的一个有效补充。**

1.3. 其他优点：

1> 声明式事务的支持：

只需要通过配置就可以完成对事务的管理，而无需手动编程。

2> 方便程序的测试

Spring 对 Junit4支持，可以通过注解方便的测试Spring程序。

3> 方便集成各种优秀的框架。

二. Spring中IOC(控制反转)和DI(依赖注入)

2.1 原理

IOC控制反转就是1.1中讲的那样通过配置xml文件，把所有对象的使用，交给spring工厂。这样耦合度很低。

而DI依赖注入，是在IOC基础上的一个概念。

首先，我们要明白java中对象与对象的关系：\* 继承

\* 依赖

\* 聚集

继承好理解，那么依赖是怎么样的一种情况呢？

|  |
| --- |
| public class B{}  public class A{  private B b; //这种情况就叫，A对象依赖B对象  } |

Q：那么spring中的DI 是指什么呢？

A：在我们通过spring工厂建立A对象的时候，spring工厂也会把对像b随着A的创建而创建，这就叫依赖注入。

2.2 IOC 和 DI 的使用

Ioc：

*<?*xml version="1.0" encoding="UTF-8"*?>*<**beans** xmlns="http://www.springframework.org/schema/beans"  
 xmlns:p="http://www.springframework.org/schema/p"  
 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="demo1" class="springDemo.SpringDemo" >  
 </**bean**>  
</**beans**>

然后在需要使用SpringDemo的时候

//创建一个工厂类  
**ApplicationContext** applicationContext=  
 **new** ClassPathXmlApplicationContext("demoContext.xml");  
**SpringDemo** springDemo= (**SpringDemo**) applicationContext.getBean("demo2");  
springDemo.sayHello();

DI的使用：

spring的依赖注入也是采用配置的方式。

<**bean** id="demo1" class="springDemo.SpringDemo" ></**bean**>  
  
 <!--采用静态工厂来创造-->  
<**bean** id="demo2" class="springDemo.SpringDemoFactory" factory-method="getSpringBean">  
 <**property** name="string" value="周楷"></**property**>  
</**bean**>

<property >这个标签就是用来 填需要注入的属性的，如果需要注入的不是对象，那就可以直接用value来表示值。如果注入的是对象，那么用ref属性。

但，记住，要使用注入的话，那么必须在springDemo中对String string这个属性，添加get，set方法。

三. Spring 细节

3.1 BeanFactory 和 ApplicationContext 区别：

ApplicationContext 继承了 BeanFactory接口

ApplicationContext 一旦加载就会创立xml中所有配置的对象。

BeanFactory 只有在使用 getBean(String id)才会配置需要的对象。

3.2 bean类的作用范围

spring中bean的作用范围指的是，创造的类是什么样的？是根据bean上的scope元素规定的，**实际开发用的还是singleton、prototype**

<!--创建的是单例模式的bean 这是默认的-->  
<**bean** id="demo3" class="springDemo.SpringDemo" scope="singleton" ></**bean**>  
  
<!--创建的是一般模式的bean,即调用一次就重新创建一个对象-->  
<**bean** id="demo4" class="springDemo.SpringDemo" scope="prototype" ></**bean**>  
  
<!--创建存入request域的bean类-->  
<**bean** id="demo5" class="springDemo.SpringDemo" scope="request" ></**bean**>  
  
<!--创建存入session域的bean类-->  
<**bean** id="demo5" class="springDemo.SpringDemo" scope="session" ></**bean**>  
  
<!--request、session Only valid in the context of a web-aware Spring ApplicationContext.-->

**注意：在spring中，一个类可以有多个bean，只要他们在bean标签配置的id不一样就可以了。**

<**bean** id="demo3" class="springDemo.SpringDemo" scope="singleton" ></**bean**>

<**bean** id="demo4" class="springDemo.SpringDemo" scope="prototype" ></**bean**>

3.3. Spring的生命周期：

配置Bean的初始化和销毁的方法:

配置初始化和销毁的方法:

\* init-method=”setup”

\* destroy-method=”teardown”

执行销毁的时候,必须手动关闭工厂,而且只对scope=”**singleton**”有效.

Bean的生命周期的11个步骤:

1.instantiate bean对象实例化

public Custome(){

//生命周期第一步

}

2.populate properties 封装属性

private String str；

public void setStr(){

//生命周期第二步，注入属性

}

3.如果Bean实现BeanNameAware 执行 setBeanName

4.如果Bean实现BeanFactoryAware 或者 ApplicationContextAware 设置工厂 setBeanFactory 或者上下文对象 setApplicationContext

**5.如果存在类实现 BeanPostProcessor（后处理Bean） ，执行postProcessBeforeInitialization**

6.如果Bean实现InitializingBean 执行 afterPropertiesSet

7.调用<bean init-method="init"> 指定初始化方法 init

**8.如果存在类实现 BeanPostProcessor（处理Bean） ，执行postProcessAfterInitialization**

9.执行业务处理

10.如果Bean实现 DisposableBean 执行 destroy

11.调用<bean destroy-method="customerDestroy"> 指定销毁方法 customerDestroy

四. IOC装配Bean的方式 和 相应的属性注入DI方式：

关于xml名称空间与引入约束的说明：

**命名空间：**

命名空间声明的一般形式为：第一部分是一个关键字xmlns:，第二部分是命名空间的前缀，第三部分是一个等号，第四部分是双引号，将第五部分的命名空间标识URI包括起来。需要注意的是，命名空间的前缀不能为xml，因为在XML中这个字符串是保留作特殊用途的。例：  
     xmlns:xs="<http://www.w3.org/2001/XMLSchema>"    //其中xs为前缀。  
     还可以隐式声明命名空间，即省略掉冒号和命名空间前缀。例：  
     xmlns="[http://www.w3school.com.cn](http://www.w3school.com.cn/)"   //注意在一个文档中只能有一个隐式声明的命名空间

**.xsd文件中的targetNamespace:**

xsd文件中定义了一个targetNameSpace后，其内部定义的元素，属性，类型等都属于该targetNameSpace,其自身或外部xsd文件使用这些元素，属性等都必须从定义的targetNameSpace中找。targetNamespace="http://www.tjbsl.com.cn",这里的targetNamespace属性表示了该shema所对应的命名空间的URI。也就是说在引用该Schema的其它文档中要声明命名空间，其URI应该是targetNamespace的属性值。

     targetNamespace定义了Schema定义的新元素与属性的命名空间。而"<http://www.w3.org/2001/XMLSchema>"命名空间则定义了element, attribute, complexType, group, simpleType等元素。

**总结：**

XML和Schema的命名空间标签使用格式是相同的（这也是Schema相对与DTD的优势），但XML和Schema都有各自的独特的属性，这也是由他们不同的功能决定的，Schema主要给XML提供服务，所以会规定好targetNameSpace来声明命名空间的名字，而XML需要使用schema的服务，所以需要SchemaLocation来声明使用的命名空间。

所以下面的<http://www.springframework.org/schema/beans>

在.xsd文件中是放在targetNamespace中，

在xml文件就是放在xmlns:xxx中，xxx可以自定义。当使用这个目录下约束的标签的时候，

需要<xxx: >来使用。

比如在 <http://zzzzzk.abc> 中约束了<zk>这个标签，那么当你在xml中这样引用

xmnls:zz=” <http://zzzzzk.abc>” ,那么使用<zk>标签就必须

<zz:zk></zk>

**正文开始了：**

**装配bean的意思就是创建bean对象的方式，spring中bean装配有两大方式：**

**4.1 通过xml配置装配方式**

首先xml装配的方式需要引入的名称空间和约束如下：

<**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  
 ">

xmlns是xxx.xml文件用到的命名空间  
xmlns:xsi是指xxx.xml遵守xml规范  
xsi:schemaLocation是指具体用到的schema资源  //这里指具体用到的资源位置。

其中Spring中通过xml配置装配又有三种方式：

1> 通过无参数构造

<!--默认是采用无参构造器创造对象的-->  
<**bean** id="demo1" class="springDemo.SpringDemo" ></**bean**>

2> 通过工厂静态构造

<!--采用静态工厂来创造-->  
<**bean** id="demo2" class="springDemo.SpringDemoFactory" factory-method="getSpringBean">  
 <**property** name="string" value="周楷"></**property**>  
</**bean**>

工厂类：

**public class SpringDemoFactory** {  
 **public static SpringDemo** getSpringBean(){  
 // return new SpringDemo("zk");  
 //通过工厂，可以创建有参数的对象。  
 **return new** SpringDemo();  
 }  
}

3> 通过工厂实例创造(了解)

就是通过工厂的非静态方法创建对象。

相应的，xml方式也有相应的属性DI注入方式：

属性分别有：基本类型属性，对象属性，集合属性

**对于基本类型属性：**

<!--配置基本属性-->  
<**bean** id="SpringDemo" class="springDemo.SpringDemo" scope="prototype">  
 <**property** name="string" value="周楷"></**property**>  
</**bean**>

**对于对象属性：**

首先看SpringDemo中有一个对象属性 SpringZK

|  |
| --- |
| **public class SpringDemo implements** ISpringDemo1{  **private SpringZk** springZk;  **private String** string; |

再看配置文件中需要这样配置：

<!--配置对象属性-->  
<**bean** id="zzz" class="springDemo.SpringZk" >  
 <**property** name="aname" value="吴恒"></**property**>  
 <**property** name="user" value="新用户"></**property**>  
</**bean**> <!--要先配置需要注入的对象的bean-->  
<**bean** id="SpringDemo" class="springDemo.SpringDemo" scope="prototype">  
 <**property** name="string" value="周楷"></**property**>  
 <**property** name="springZk" ref="zzz"></**property**>  
 <!--然后，再用ref引用配置的需要注入的对象 -->  
</**bean**>

**对于集合属性（List ,set ,Map）：**

<**bean** id="SpringDemo" class="springDemo.SpringDemo" scope="prototype">  
 <**property** name="string" value="周楷"></**property**>  
 <!-- 注入List集合 -->  
 <**property** name="springZks">  
 <**list**>  
  
 <**ref** bean="zzz"></**ref**>  
 </**list**>  
 </**property**>  
  
 <!-- 注入set集合 -->  
 <**property** name="set">  
 <**set**>  
 <**value**>杜宏</**value**>  
 <**value**>如花</**value**>  
 </**set**>  
 </**property**>  
  
 <!-- 注入map集合 -->  
 <**property** name="map">  
 <**map**>  
 <**entry** key="刚刚" value="111"/>  
 <**entry** key="娇娇" value="333"/>  
 </**map**>  
 </**property**>  
 <!--注入properties文件-->  
 <**property** name="properties">  
 <**props**>  
 <**prop** key="username">root</**prop**>  
 <**prop** key="password">123</**prop**>  
 </**props**>  
 </**property**>  
  
</**bean**>

**在property中，引用List集合，如果List中是一般类型就用value=“”，**

**如果是对象就用<ref bean=” ”> 这个来引用bean**

4.2 通过注解的方式装配 和 相应的DI注入

首先要使用注解方式，那么必须在xml中引入约束

<**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">  
  
 <**bean**>

然后在beans目录下，使用：

<context:annotation-config/>

<**context:component-scan base-package**="springDemo"/>  
<!--明确在哪个包下去检测，使用了注解-->

然后在bean类上使用

@Component("SpringDemo")  
**public class SpringDemo implements** ISpringDemo1 {  
 @Override  
 **public void** sayHello() {  
  
 **System**.out.println("zzzzzk");  
 }  
}

Component 代表这个类是bean ，而且Spring在后面分层的时候，为了好区分，还是分别用了三个和Component一样功效的注解：

Repoistory 用来给Dao层的类使用

Service 用来给service层的类使用

Controller 用来给web层的类使用

**相应的，通过注解方式注入属性的方式是：**

**普通属性：**

@Value("是弱智")  
**private String** str;  
@Override  
**public void** sayHello() {  
  
 **System**.out.println("zzzzzk"+str);  
}

**对象属性：**

对象属性和复杂属性都是采用：

**public class SpringDemo implements** ISpringDemo1 {  
 @Autowired

@Qualifier("springZk")   
 **private SpringZk** springZk;

这里可以看到我们需要注入的对象上有一个 错误下划线

这是因为，我们并没有在SpringZk上加入Component注解。

那么我们采用两种方式引入SpringZk都是可以的，

<**bean** id="springZk" class="springDemo.SpringZk" >  
 <**property** name="user" value="zzz">  
 </**property**>  
 <**property** name="aname" value="lalal"></**property**>  
</**bean**>

@Repository("springZk")  
**public class SpringZk** {  
 @Value("zzz")  
 **private String** aname;  
 @Value("kkk")  
 **private String** user;

但是默认会采用第一种方式，而且这里我们使用了Repository注解。注意在

Repository(“名字相等”) 中要等于 前面的Qualifier(“名字相等”)

**不能注入集合属性**

另外，bean在xml上还有很多属性，比如scope ，init-method，destory-method，对应的在类上的注解分别是 scope， PostInit ，Predestroy。

但是有个很重要的问题，采用注解的方法不能一个类不同属性的注入。即注解不能创建一个类的注入不同对象（即，不能注入不同属性）。

如果需要注入不同对象，最好的办法就是采用xml配置。

<**bean** id="zk1" class="springDemo.SpringZk">  
 <**property** name="aname" value="周楷1号"></**property**>  
 <**property** name="user" value="朱佳伟1号"></**property**>  
</**bean**>  
<**bean** id="zk2" class="springDemo.SpringZk" >  
 <**property** name="aname" value="周楷2号"></**property**>  
 <**property** name="user" value="朱佳伟2号"></**property**>  
</**bean**>  
<**bean** id="zk3" class="springDemo.SpringZk">  
 <**property** name="aname" value="周楷3号"></**property**>  
 <**property** name="user" value="朱佳伟3号"></**property**>  
</**bean**>  
<**bean** id="springDemo2" class="springDemo.SpringDemo"></**bean**>  
  
<**bean** id="springDemo" class="springDemo.SpringDemo">  
 <**property** name="springZks" >  
 <**list**>  
 <**ref** bean="zk1"></**ref**>  
 <**ref** bean="zk2"></**ref**>  
 <**ref** bean="zk3"></**ref**>  
 </**list**>  
 </**property**>  
</**bean**>

4.3 总结：

虽然，开发推荐 通过xml装配bean，通过注解注入属性，两者结合起来。但是，因为实际开发并没有用过，并不明白，为什么只需要一种属性注入，因此还是觉得xml使用舒服。

五. Spring整合其他框架

5.1 Spring整合web的原理

在实际开发中，Spring都是用来和web开发打交道的，因此在web项目中，使用Spring，难道要每次使用Spring的时候，都要使用

ApplicationContext applicationContext=  
 **new** ClassPathXmlApplicationContext("demoContext.xml");

来加载配置文件吗？

有没有可能在web项目启动的时候，就加载一次且只加载一次？

我们回想，web层最基本的三个东西：servlet，listener，filter。任何基于web层开发的框架都是使用更高明的编程技巧来合理的使用这三样东西。

那么，servlet中有一个servletContext域对象，可以把对象存入到其中，而且最关键的是，servletContext是项目一加载就会创建的，然后，这个对象是贯穿整个项目的。

所以，我们可以把加载Spring配置得到的对象，放入到ServletContext域中。那么加载配置的代码写在哪呢？

再回想，listener监听器中，主要监听ServletContext、Request、Session的创建和销毁，因此我们就可以把代码卸载监听ServletContext的创建方法中。

第一步：我们创建一个ServletContext监听器：

**public class MyListener implements** ServletContextListener {  
 @Override  
 **public void** contextInitialized(**ServletContextEvent** servletContextEvent) {  
 //在这个里面实现加载spring配置的代码。  
 ApplicationContext applicationContext=  
 **new** ClassPathXmlApplicationContext("demoContext.xml");  
 ServletContext servletContext=servletContextEvent.getServletContext();  
 servletContext.setAttribute("applicationContext",applicationContext);  
 }  
 @Override  
 **public void** contextDestroyed(**ServletContextEvent** servletContextEvent) {

}  
}

第二步，配置Spring的bean

<**bean** id="**userService**" class="service.UserService"></**bean**>

第三步，通过spring工厂调用bean，实现业务代码

**protected void** doPost(HttpServletRequest request, HttpServletResponse response) **throws ServletException**, **IOException** {  
 //1.得到ServletContext中的Spring配置对象ApplicationContext  
 ApplicationContext context= (ApplicationContext) request.getServletContext().getAttribute("applicationContext");  
 //2.通过工厂加载需要的对象  
 **UserService** service= (**UserService**) context.getBean("userService");  
 //3.实现业务代码  
 service.eat();  
}

以上，就是Spring整合web的原理。但实际上，Spring已经提供了相应的api给我们。

|  |
| --- |
| 导入;spring-web-3.2.0.RELEASE.jar  在web.xml中配置:  <listener>  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  </listener>  <context-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:applicationContext.xml</param-value>  </context-param>  修改程序的代码:  **WebApplicationContext** applicationContext =  WebApplicationContextUtils.getWebApplicationContext(getServletContext()); **UserService** userService= (**UserService**) applicationContext.getBean("userService"); userService.eat(); |

5.2 Spring整合Junit

实际上，可以看到

**import org.junit.**Test;  
**import org.springframework.context.**ApplicationContext;  
**import org.springframework.context.support.ClassPathXmlApplicationContext**;  
/\*\*  
 \* Created by horse on 2017/6/13.  
 \*/  
**public class SpringTest** {  
 //spring方式  
 @Test  
 **public void** demo2(){  
 //创建一个工厂类  
 ApplicationContext applicationContext=  
 **new** ClassPathXmlApplicationContext("demoContext.xml");  
 **SpringDemo** springDemo= (**SpringDemo**) applicationContext.getBean("springDemo");  
 springDemo.sayHello();  
 }  
}

只要引入了Junit-4.1.2.jar包，那么就可以直接在方法上，通过注解@Test对方法进行测试。

但，这只是一个小的测试，如果测试多起来，这样测试是很麻烦的。

因此Spring也有整合Junit的jar包

环境：程序中有Junit环境.

导入一个jar包.spring与junit整合jar包.

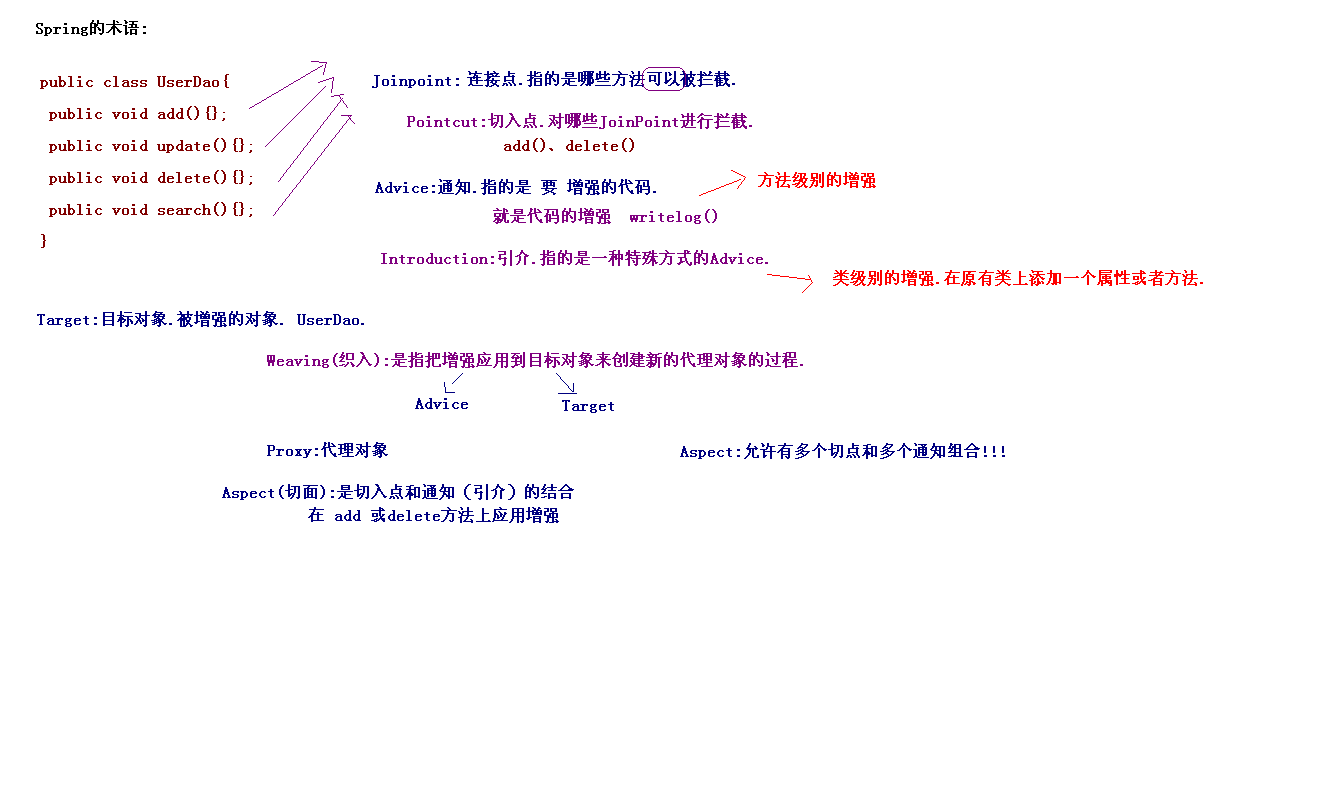
\* spring-test-3.2.0.RELEASE.jar

这样，我们的测试就可以这样进行：

@RunWith(**SpringJUnit4ClassRunner**.**class**) //固定写法  
@ContextConfiguration(locations = "classpath:demoContext.xml")

//这是指明配置文件的位置。  
**public class SpringJunit** {  
 @Autowired  
 @Qualifier("springDemo")  
 **private SpringDemo** springDemo;  
 @Test  
 **public void** prin(){  
 springDemo.sayHello();  
 }  
}

六.Spring中的aop编程



6.1 在Spring中AOP编程的两种方式

6.1.1 Jdk动态代理Aop编程：

在学动态代理的时候，就知道了动态代理是Aop切面编程的基础。下面我们用一个更具体的例子来理解：

首先Jdk原生的Proxy动态代理有个缺点，就是生成的代理对象必须是接口接收，所以先定义要被代理类的接口：

**public interface** UserDao {  
 **public void** add();  
 **public void** delete();  
}

然后通过一个子类，实现其中的方法：

**public class UserDaoImp implements** UserDao {  
 @Override  
 **public void** add() {  
 **System**.out.println("add 方法");  
 }  
 @Override  
 **public void** delete() {  
 **System**.out.println("delete 方法");  
 }  
}

然后，我们再定义一个类来创建它的Proxy代理对象：

**public class JdkProxy implements** InvocationHandler{  
 **private** UserDao userDao;  
 **public** JdkProxy(UserDao dao){  
 **this**.userDao=dao;  
 }  
 **public** UserDao creatProxy(){  
 UserDao dao= (UserDao) **Proxy**.newProxyInstance(userDao.getClass().getClassLoader(),  
 userDao.getClass().getInterfaces(), **this**);  
 **return** dao;  
 }  
 **public Object** invoke(**Object** proxy, **Method** method, **Object**[] args) **throws Throwable** {  
 **if**(method.getName().equals("add")){  
 **System**.out.println("增强add");  
 **Object** result=method.invoke(userDao,args);  
 **return** result;  
 }  
 **return** method.invoke(userDao,args);  
 }  
}

这里有个细节问题，在动态代理文档中，Proxy.newProxyInstance()的第三个参数是一个匿名函数

**new** InvocationHandler() {  
 @Override  
 **public Object** invoke(**Object** proxy, **Method** method, **Object**[] args) **throws Throwable** {  
 **if**(method.getName().equals("add")){  
 **System**.out.println("增强add");  
 **Object** result=method.invoke(userDao,args);  
 **return** result;  
 }  
 **return** method.invoke(userDao,args);  
 }

而这里我们的处理是，把这个类本身实现InvocationHandler这个接口，因此我们可以把它作为第三个参数，this，然后重写其中的invoke方法。

在最后，我设想，如果每一个类都需要使用代理，总不可能全部采用

**package jdkAop**;  
  
**import java.lang.reflect.**InvocationHandler;  
**import java.lang.reflect.Method**;  
**import java.lang.reflect.Proxy**;  
  
/\*\*  
 \* Created by horse on 2017/6/16.  
 \*/  
**public class JdkProxy**<T> {  
 **private** T t;  
 **public** JdkProxy(T t){  
 **this**.t=t;  
 }  
 **public** T creatProxy(){  
 T t1= (T) **Proxy**.newProxyInstance(t.getClass().getClassLoader(),  
 t.getClass().getInterfaces(), **new** InvocationHandler() {  
 @Override  
 **public Object** invoke(**Object** proxy, **Method** method, **Object**[] args) **throws Throwable** {  
 /\*

这里写通用的权限验证代码。

\*/

**return** method.invoke(t,args);  
 }  
 });  
 **return** t1;  
 }  
}

通过这个泛型类，我们就可以根据传入的不同类型，生成相应的代理。

6.1.2 cglib动态代理