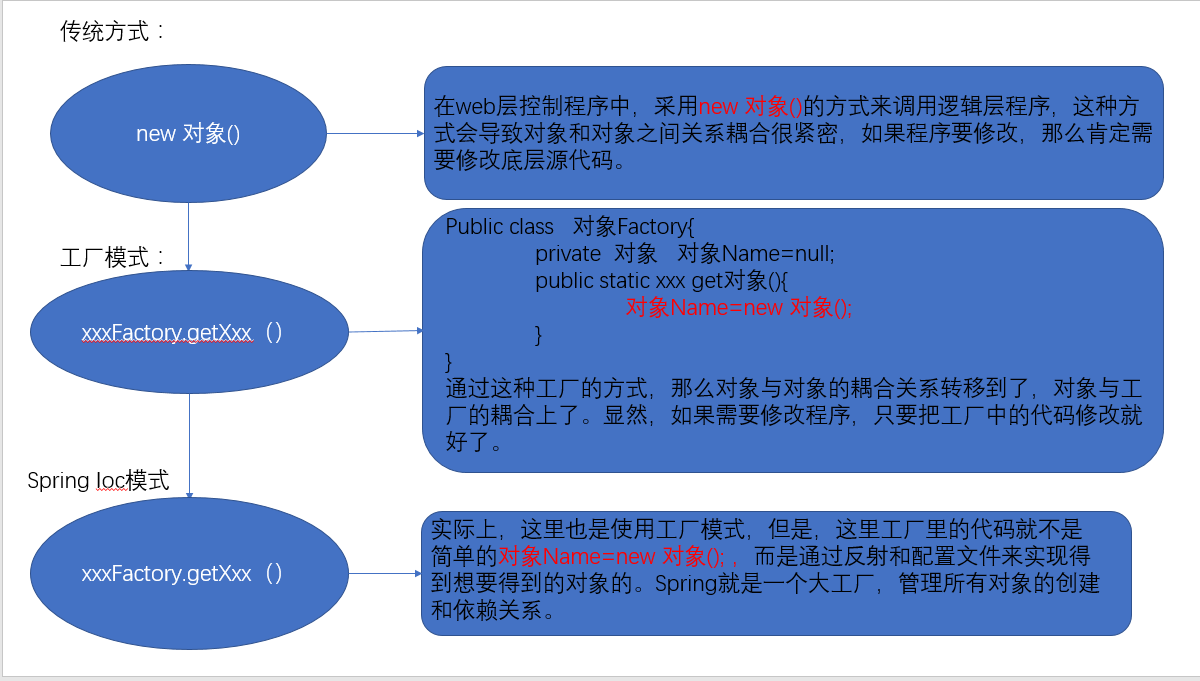
1. Spring的优点

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。（这主要是采用xml配置的方式来弄的）**

<**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

后处理bean

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.

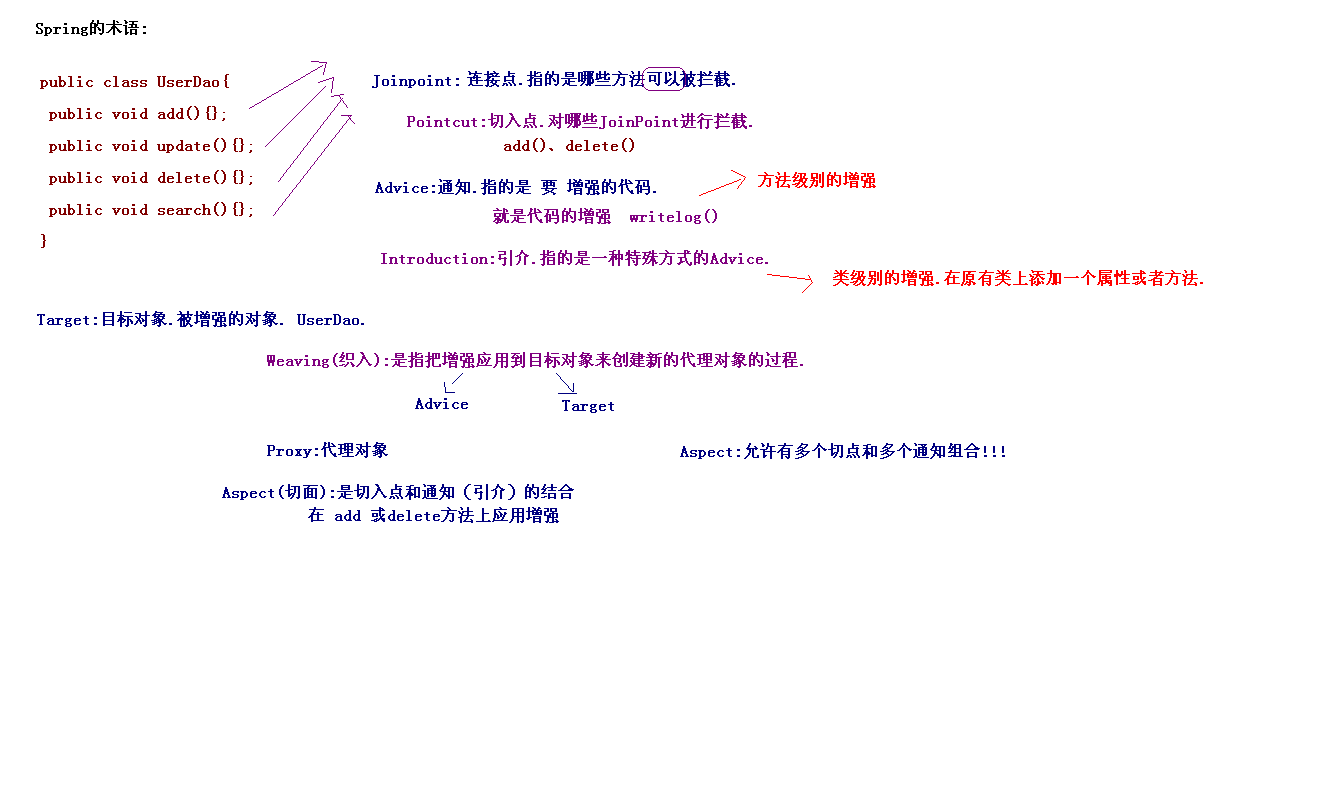
\* spring-test-3.2.0.RELEASE.jar

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

原理：不对接口类生成代理，其实是生成了一个真实对象的子类。

CGLIB(Code Generation Library)是一个开源项目！是一个强大的，高性能，高质量的Code生成类库，它可以在运行期扩展Java类与实现Java接口。 Hibernate支持它来实现PO(Persistent Object 持久化对象)字节码的动态生成

Hibernate生成持久化类的javassist.

CGLIB生成代理机制:其实生成了一个真实对象的子类.

下载cglib的jar包.

\* 现在做cglib的开发,可以不用直接引入cglib的包.已经在spring的核心中集成cglib.

public class CGLibProxy implements MethodInterceptor{

private ProductDao productDao;

public CGLibProxy(ProductDao productDao) {

super();

this.productDao = productDao;

}

public ProductDao createProxy(){

// 使用CGLIB生成代理:

// 1.创建核心类:

Enhancer enhancer = new Enhancer();

// 2.为其设置父类:

enhancer.setSuperclass(productDao.getClass());

// 3.设置回调:

enhancer.setCallback(this);

// 4.创建代理:

return (ProductDao) enhancer.create();

}

public Object intercept(Object proxy, Method method, Object[] args,

MethodProxy methodProxy) throws Throwable {

if("add".equals(method.getName())){

System.out.println("日志记录==============");

Object obj = methodProxy.invokeSuper(proxy, args);

return obj;

}

return methodProxy.invokeSuper(proxy, args);

}

}

**结论:Spring框架,如果类实现了接口,就使用JDK的动态代理生成代理对象,如果这个类没有实现任何接口,使用CGLIB生成代理对象.**

6.2 Spring中的传统Aop开发

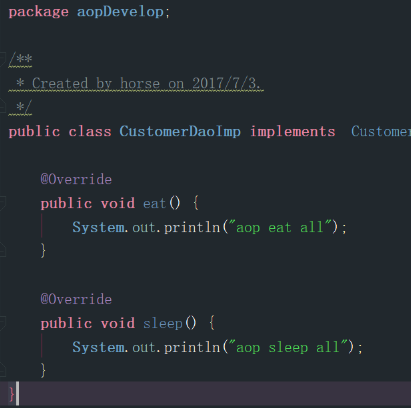
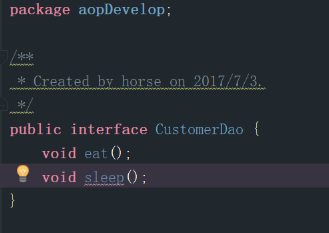
要在Spring中使用Aop开发，首先要引入相应的jar包：

\* spring-aop-3.2.0.RELEASE.jar

\* com.springsource.org.aopalliance-1.0.0.jar

6.2.1 不带切点的切面开发（即：对所有的方法进行增强）

第一步：编写接口以及实现类



第二步：编写需要增强的代码类，以及增强的位置

在spring的advice中有很多种：

前置通知 org.springframework.aop.MethodBeforeAdvice

\* 在目标方法执行前实施增强

后置通知 org.springframework.aop.AfterReturningAdvice

\* 在目标方法执行后实施增强

环绕通知 org.aopalliance.intercept.MethodInterceptor

\* 在目标方法执行前后实施增强

异常抛出通知 org.springframework.aop.ThrowsAdvice

\* 在方法抛出异常后实施增强

引介通知 org.springframework.aop.IntroductionInterceptor(课程不讲.)

\* 在目标类中添加一些新的方法和属性

编写的时候就实现相应的接口就好了：

|  |
| --- |
| **package aopDevelop**;  **import org.springframework.aop.**MethodBeforeAdvice;  **import java.lang.reflect.Method**;  /\*\*  \* Created by horse on 2017/7/3.  \*/ **public class ZkBeforeAdvice implements** MethodBeforeAdvice{   @Override  **public void** before(**Method** method, **Object**[] objects, **Object** o) **throws Throwable** {  **System**.out.println("前置增强");  } } |

第三步：在spring配置文件中，通过配置生成代理

spring中并不需要我们自己去写代理，可以通过配置生成。

|  |
| --- |
| <!--定义目标对象-->  <**bean** id="customerDao" class="aopDevelop.CustomerDaoImp"></**bean**>  <!--定义通知-->  <**bean** id="beforeAdvice" class="aopDevelop.ZkBeforeAdvice"></**bean**>  <!—配置生成代理-->  <**bean** id="customerDaoProxy" class="org.springframework.aop.framework.ProxyFactoryBean">  <!--设置目标对象-->  <**property** name="target" ref="customerDao"/>  <!--设置实现的接口，value中写接口的全路径-->  <**property** name="proxyInterfaces" value="aopDevelop.CustomerDao"/>  <!--需要实现的什么增强代理 l拦截器的名字-->  <**property** name="interceptorNames" value="beforeAdvice"/>  </**bean**> |

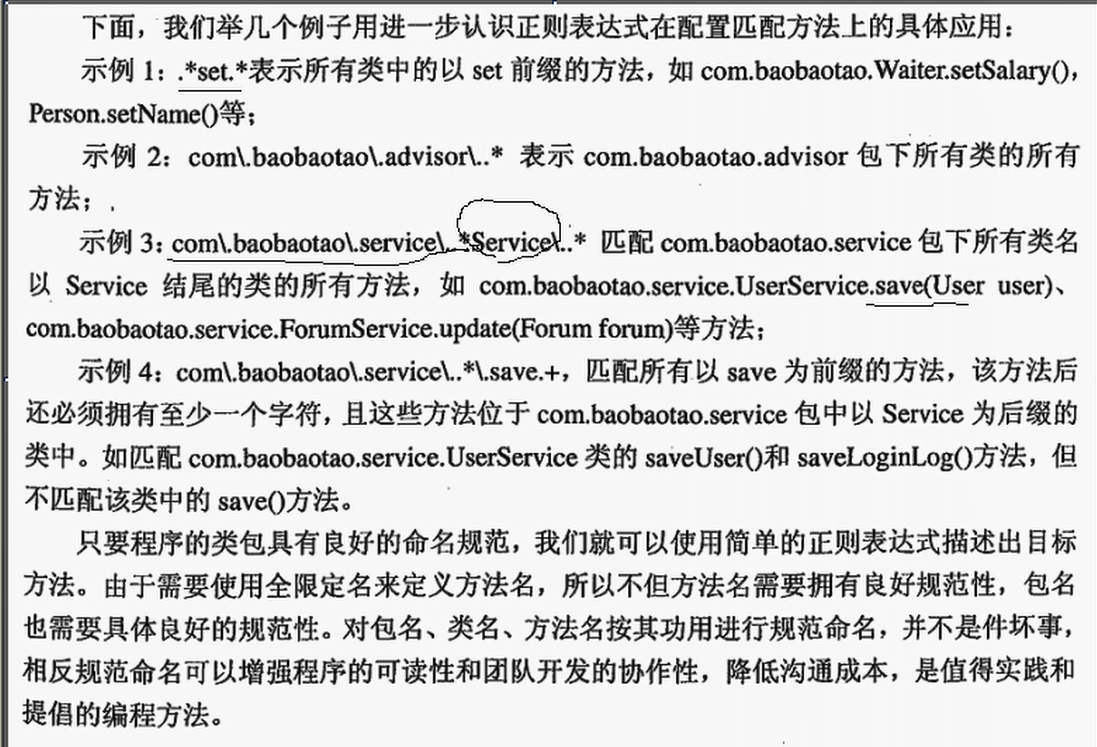
第四步：使用代理！

在spring中使用代理，是采用给对象 注入 它的代理对象来完成。

|  |
| --- |
| @RunWith(**SpringJUnit4ClassRunner**.**class**) @ContextConfiguration(locations = "classpath:applicationContext.xml") **public class AopTest** {  @Autowired  @Qualifier("customerDaoProxy")  **private** CustomerDao customerDao;  @Test  **public void** test1(){  */\*  \* 测试不带切点的切面的使用  \* \*/* customerDao.eat();  } } |

6.2.2 带切点的切面

正则表达式：



采用cglib生成代理（spring会自动选择是采用cglib还是proxy）

第一步：定义增强目标

|  |
| --- |
| **package aopDevelop.pointAdvice**;  /\*\*  \* Created by horse on 2017/7/3.  \*/ **public class OrderDao** {  **public void** drink(){  **System**.out.println("drink point");  }  **public void** down(){  **System**.out.println("down point");  } } |

第二步：定义增强类型类

|  |
| --- |
| **import org.aopalliance.intercept**.MethodInterceptor; **import org.aopalliance.intercept**.MethodInvocation;  /\*\*  \* Created by horse on 2017/7/3.  \*/ **public class ZkAroundAdvice implements** MethodInterceptor{  @Override  **public Object** invoke(MethodInvocation methodInvocation) **throws Throwable** {  **System**.out.println("环绕前增强。。");  **Object** result=methodInvocation.proceed(); //执行目标方法  **System**.out.println("后增强");  **return** result;  } } |

第三步：在applicationContext中配置

|  |
| --- |
| <!--带切点的增强--> <!-- 定义目标对象 --> <**bean** id="orderDao" class="aopDevelop.pointAdvice.OrderDao"></**bean**> <!-- 定义增强 --> <**bean** id="aroundAdvice" class="aopDevelop.pointAdvice.ZkAroundAdvice"></**bean**> <!-- 定义切点切面: --> <**bean** id="myPointcutAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">  <!-- 定义表达式,规定哪些方法执行拦截 -->  <!-- . 任意字符 \* 任意个 -->  <!--定义切点-->  <**property** name="pattern" value=".\*drink.\*"></**property**>  <!-- 应用增强 -->  <**property** name="advice" ref="aroundAdvice"/> </**bean**> <!-- 定义生成代理对象 --> <**bean** id="orderDaoProxy" class="org.springframework.aop.framework.ProxyFactoryBean">  <!-- 配置目标 -->  <**property** name="target" ref="orderDao"></**property**>  <!-- 针对类的代理 -->  <**property** name="proxyTargetClass" value="true"></**property**>  <!-- 在目标上应用增强 -->  <**property** name="interceptorNames" value="myPointcutAdvisor"></**property**> </**bean**> |

6.2.3 实际开发以及自动代理

前面介绍的两种都是自己在xml中配置，一种增强就要写一堆的代码，通过ProxyFactoryBean来织入切面的，但这样很麻烦。实际开发中，是通过自动代理来实现的。

**自动代理和上面根据配置代理的原理是不同的：**

**根据配置代理是在生成bean之后，通过cglib或者jdkProxy来生成相应的代理。**

**自动代理是用 后处理bean 的方法时候，通过cglib或者jdkProxy来生成相应的代理。**

**采用cglib还是jdkProxy，spring会根据被代理是否有接口自动选择的。**

spring中采用配置方法的主要采用两种方式来实现自动代理：

1.根据名称来生成代理类：

org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator

第一步：创建被代理对象。

第二步：创建增强类。

第三步（配置代码简化）：xml配置

|  |
| --- |
| <!--不带切点的增强--> <!--定义目标对象--> <**bean** id="customerDao" class="aopDevelop.aopAll.CustomerDaoImp"></**bean**>  <!--带切点的增强--> <!-- 定义目标对象 --> <**bean** id="orderDao" class="aopDevelop.pointAdvice.OrderDao"></**bean**> <!-- 定义增强 --> <**bean** id="aroundAdvice" class="aopDevelop.pointAdvice.ZkAroundAdvice"></**bean**> <!--定义增强--> <**bean** id="beforeAdvice" class="aopDevelop.aopAll.ZkBeforeAdvice"></**bean**>  <!--采用根据名称的自动代理--> <**bean** class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">  <**property** name="beanNames" value="\*Dao"/>  <**property** name="interceptorNames" value="beforeAdvice"/> </**bean**> |

可以看到，相比上面一堆的代码，这里只要简单的一个BeanNameAutoProxyCreator类就可以了。

插播小知识：

|  |
| --- |
| <**property** name="beanNames" value="\*Dao"/>  <**property** name="pattern" value=".\*d.\*"/>  这里两个\* 意义是不同的：  第一个是通配符中的\* ， 意义是，匹配0个字符或多个字符。  第二个是正则表达式中 \* ，意义是，匹配该字符的前一个字符的0次或多次出现。 |

2.根据切面切点信息来生成代理类：

org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator

和上面只有第三步不同：

|  |
| --- |
| <!--采用根据切点切面信息的自动代理--> <!-- 定义一个带有切点的切面 --> <**bean** id="myPointcutAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">  <**property** name="pattern" value=".\*d.\*"/>  <**property** name="advice" ref="aroundAdvice"/> </**bean**>  <**bean** id="myPointcutAdvisor2" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">  <**property** name="pattern" value=".\*eat.\*"/>  <**property** name="advice" ref="aroundAdvice"/> </**bean**>  <!-- 自动生成代理 --> <**bean** class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"></**bean**> |

只要配置一个DefaultAdvisorAutoProxyCreator 类就可以了

6.2.4 总结：

目标对象：一个类

连接点：目标对象中的所有方法

通知：前置增强，后置增强，环绕增强 （通过继承相应的接口来实现）

定义切点：通过配置，来把通知织入相应的方法，织入通知的方法被成为切点。

最后通过配置生成代理或者自动生成代理。

6.3 Spring中的AspectJ的AOP开发

第一步：引入aspectJ的相应jar包

com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar

spring-aspects-3.2.0.RELEASE.jar

第二步：编写目标对象类。

|  |
| --- |
| **public class UserDao** {  **public void** add(){  **System**.out.println("添加用户");  }  **public int** update(){  **System**.out.println("修改用户");  **return** 1;  }  **public void** delete(){  **System**.out.println("删除用户");  }  **public void** find(){  **System**.out.println("查询用户");  **int** d = 1/ 0;  } } |

第三步：编写集合通知类

|  |
| --- |
| **import org.aspectj.lang.**JoinPoint; **import org.aspectj.lang.**ProceedingJoinPoint; **import org.aspectj.lang.annotation.**\*; @Aspect **public class MyAspect**{  @Before("execution(\* test.UserDao.\*(..) )")  **public void** before(**JoinPoint** joinPoint){  **System**.out.println("lalalal "+joinPoint);  //没有办法阻止目标方法执行的  }  @AfterReturning(value = "execution(\* test.UserDao.update(..) )",returning = "returnVal")  **public void** afterReturning(**Object** returnVal){  **System**.out.println("后置增强 方法的返回值："+returnVal);  //后置增强的特性就是可以接收方法的返回值。  }   //@Around 环绕通知。  @Around(value = "execution(\* test.UserDao.find(..))")  **public Object** around(ProceedingJoinPoint proceedingJoinPoint) **throws Throwable** {  **System**.out.println("环绕前增强。。");  **Object** object=proceedingJoinPoint.proceed();  **System**.out.println("环绕后增强。。");  **return** object;   //环绕增强可以阻止方法的执行，就像拦截器（过滤器）一样，比如，不执行这个方法。返回一个null值。  }   @AfterThrowing(value = "MyAspect.myPointcut()",throwing = "e")  **public void** afterThrowing(**Throwable** e){  **System**.out.println("不好了，出异常了！！！"+e.getMessage());  }   //但是是否execute(value = "execution(\* test.UserDao.find(..))") 这个代码重复了很多次。  //那么我们可以用这个定义切点。  @Pointcut("execution(\* test.UserDao.find(..))")  **private void** myPointcut(){} }  **可以看出来，aspectJ的方式来实现aop与传统方式不同：**  **传统方式是一个通知就是一个类（继承相应的通知接口），所以形成的切面是一个通知多个切点的形式**  **aspectJ中一个通知类，就可以包括多个不同的通知，并且可以定义不同切点，所以形成的切面是多个通知多个切点的形式。** |

第四步：在xml中配置，并且开启aspectJ注解支持

|  |
| --- |
| *<?*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:aop="http://www.springframework.org/schema/aop"  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/aop http://www.springframework.org/schema/aop/spring-aop.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">  这里开启aspectJ  <**aop:aspectj-autoproxy** proxy-target-class="true"/>   <**bean** id="userDao" class="test.UserDao"></**bean**>  这里要注入通知集合类  <**bean** id="myAspect" class="test.MyAspect"></**bean**> </**beans**> |
|  |

七.spring中的事务以及事务回滚

7.1 java的异常体系

java异常体系主要是分为

throwable 分为 error 和 exception

而其中exception 又分为 非检查异常和检查异常

**1.检查型异常（Checked Exception）**

**个人理解：**所谓检查（**Checked**）是指**编译器**要检查这类异常，检查的目的一方面是因为该类异常的发生难以避免，另一方面就是让开发者去解决掉这类异常，所以称为必须处理（try ...catch）的异常。如果不处理这类异常，集成开发环境中的编译器一般会给出错误提示。

**例如：**一个读取文件的方法代码逻辑没有错误，但程序运行时可能会因为文件找不到而抛出FileNotFoundException，如果不处理这些异常，程序将来肯定会出错。所以编译器会提示你要去捕获并处理这种可能发生的异常，不处理就不能通过编译。

**2.非检查型异常（Unchecked Exception）**

**个人理解：**所谓非检查（**Unchecked**）是指**编译器**不会检查这类异常，不检查的则开发者在代码的编辑编译阶段就不是必须处理，这类异常一般可以避免，因此无需处理（try ...catch）。如果不处理这类异常，集成开发环境中的编译器也不会给出错误提示。

**例如：**你的程序逻辑本身有问题，比如数组越界、访问null对象，这种错误你自己是可以避免的。编译器不会强制你检查这种异常。

而spring事务默认只对非检查型异常造成的错误进行事务回滚。那么一旦出现检查型异常，例如io异常的话，事务不会发生回滚。这样是非常不好的。

|  |
| --- |
| public void test(){  try{  service.saveDao(User user);  File file=new File(“D:\\text.txt”);  }catch(IOException e){  //发生异常之后的处理。  }  }  在这个例子中，读取文件的时候会发生io异常，按照java处理规范，这种检查异常一定要用try catch 捕获，或者直接抛出。  一直很不理解，异常的作用是啥，为什么需要异常，现在依然不是很理解。但对于java处理异常的两种方式有了新的理解。try catch是捕获异常，捕获之后，可以不做任何处理，让程序继续进行下去； 而抛出是把异常扔给它的调用者。因此上面这种情况，如果catch中不做任何处理，那么事务完全不会回滚，因为spring根本没有接收到异常的信息，异常已经被程序捕获了，没有暴露出来。**如果是把异常抛出去，那么spring框架就可以知道这个信息，就会根据异常的类型判断是否事务回滚。**  默认的如果是非检查异常的话，就会事务回滚。但如果给@Transactional(rollbackFor = **Exception**.**class**)  那么只要是得到异常信息的话，就会事务回滚。但是，对于检查型异常，一般来说，都会用try catch 捕获起来。那要怎么通知spring呢，在catch中手动加一个事务回滚。 |