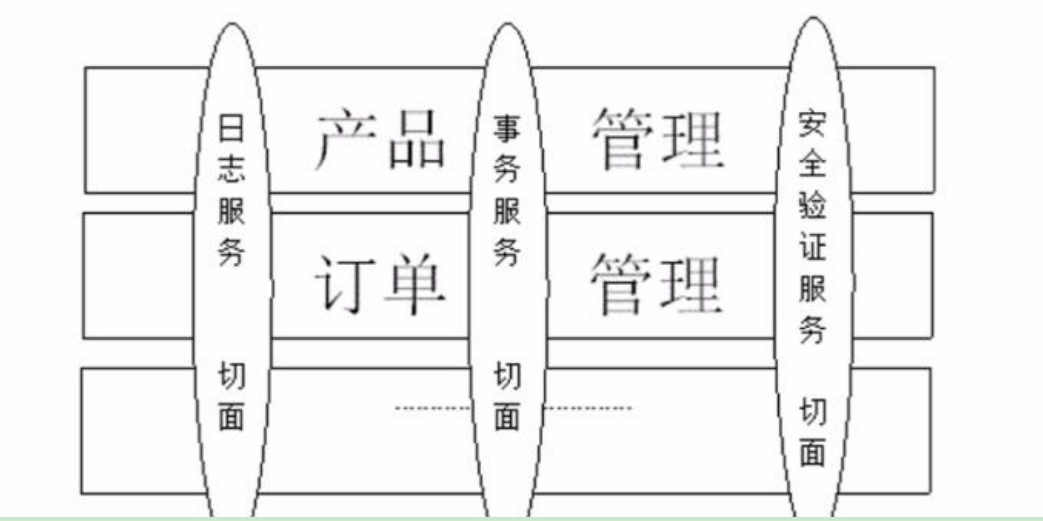
Spring AOP

什么是AOP及实现方式
AOP基本概念
Spring中的AOP
Schema-based AOP
Spring AOP API
AspectJ

https://zh.wikipedia.org/wiki/%E9%9D%A2%E5%90%91%E4%BE%A7%E9%9D%A2%E7%9A%84%E7%A8%8B%E5%BA%8F%E8%AE%BE%E8%AE%A1这是AOP的基本概念，是面向切面的编程

- **AOP：Aspect Oriented Programming的缩写，意为：面向切面编程，通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术**
- **主要的功能是：日志记录，性能统计，安全控制，事务处理，异常处理等等**

这是个切面



AOP实现方式
预编译
AspectJ
运行期动态代理（JDK动态代理，CGLib动态代理）
SpringAOP，JbossAOP

名称	说明
切面(Aspect)	一个关注点的模块化，这个关注点可能会横切多个对象
连接点(Joinpoint)	程序执行过程中的某个特定的点
通知(Advice)	在切面的某个特定的连接点上执行的动作
切入点(Pointcut)	匹配连接点的断言，在AOP中通知和一个切入点表达式关联
引入(Introduction)	在不修改类代码的前提下，为类添加新的方法和属性
目标对象(Target Object)	被一个或者多个切面所通知的对象
AOP代理(AOP Proxy)	AOP框架创建的对象，用来实现切面契约(aspect contract)(包括通知方法执行等功能)
织入（Weaving）	把切面连接到其它的应用程序类型或者对象上，并创建一个被通知的对象，分为：编译时织入、类加载时织入、执行时织入

Advice的类型

Advice的类型

名称	说明
前置通知(Before advice)	在某连接点（join point）之前执行的通知，但不能阻止连接点前的执行（除非它抛出一个异常）
返回后通知(After returning advice)	在某连接点（join point）正常完成后执行的通知
抛出异常后通知(After throwing advice)	在方法抛出异常退出时执行的通知
后通知(After(finally)advice)	当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）
环绕通知(Around Advice)	包围一个连接点（join point）的通知

Spring框架中AOP的用途

1 提供了声明式的企业服务，特别是**EJB的替代服务的声明，这种服务是可以用于任何地方的，比如互联网也OK,可以使用Spring定义自己的切面**
因为切面，可以与面向对象，结合使用，可以使用自己的功能。
2 允许**用户定制自己的方面（切面）**，以完成OOP与AOP的互补使用
OOP与AOP的关注点事不一样的 ，OOP是面向对象编程 ，就不多说了， AOP,就是横切 。

Spring的AOP实现

- 纯java实现，无需特殊的编译过程，不需要控制类加载器层次
- 目前只支持方法执行连接点（通知Spring Bean的方法执行）
- 不是为了提供最完整的AOP实现（尽管它非常强大）;而是侧重于提供一种AOP实现和Spring IoC容器之间的整合，用于帮助解决企业应用中的常见问题
- Spring AOP不会与AspectJ竞争，从而提供综合全面的AOP解决方案

上面的第二段说的话呢，
AspectJ是一个完整的全面的AOP解决方案
Spring AOP呢，只是一种与IOC容器之间的整合，来提供一种切实有用的，解决企业应用的场景问题。虽然不是一个完整的实现，但是是一个非常有用的实现，

有接口与无接口的区别

有接口和无接口的Spring AOP实现区别

- Spring AOP默认使用标准的JavaSE动态代理作为AOP代理，这使得任何接口（或者接口集）都可以被代理
- Spring AOP中也可以使用CGLIB代理（如果一个业务对象并没有实现一个接口）

就是有接口的话，就实现上面的默认JavaSe动态代理，如果没有接口的话，默认就是实现下面的CGLIB代理，

Schema ---based AOP
Spring所有的切面和通知器都必须放在一个<aop:config>内（可以配置包含多个<aop:config>元素），每一个<aop:config>可以包含pointcut，advisor和aspect元素（他们必须按照这个顺序进行相应的声明）
<aop:config> 风格的配置大量使用到了spring的自动代理机制

- <aop:config>
- <aop:aspect>

```
<aop:config>
    <aop:aspect id="myAspect" ref="aBean">
        ...
    </aop:aspect>
</aop:config>

<bean id="aBean" class="...">
    ...
</bean>
```

<apo:config>具体使用的是<aop:aspect>这个配置元素

- target(com.xyz.service.AccountService) （only in Spring AOP） target 用于匹配当前目标对象类型的执行方法
- args(java.io.Serializable) （only in Spring AOP）
- @target(org.springframework.transaction.annotation.Transactional) （only in Spring AOP）
- @within(org.springframework.transaction.annotation.Transactional) （only in Spring AOP）
- @annotation(org.springframework.transaction.annotation.Transactional) （only in Spring AOP）

args 用于匹配当前执行的方法传入的参数为指定类型的执行方法

- execution(public * *(..)) 切入点为执行所有public方法时
- execution(* set*(..)) 切入点为执行所有set开始的方法时
- execution(* com.xyz.service.AccountService.*(..)) 切入点为执行AccountService类中的所有方法时
- execution(* com.xyz.service..(..)) 切入点为执行com.xyz.service包下的所有方法时
- execution(* com.xyz.service...(..)) 切入点为执行com.xyz.service包及其子包下的所有方法时

关于Spring AOP execution表达式的博客
http://sishuok.com/forum/posts/list/281.html

```
<aop:config>

    <aop:pointcut id="businessService"
        expression="execution(* com.xyz.myapp.service..(..))"/>

</aop:config>
```

这就是匹配该指定的包下所有的类的类型方法的时候

```
<aop:config>

    <aop:pointcut id="businessService"
        expression="com.xyz.myapp.SystemArchitecture.businessService()"/>

</aop:config>
```

这个是只是执行这个类的这个方法的时候

根据需求来选择

http://www.cnblogs.com/65702708/archive/2012/08/20/2647793.html这是关于通知的人家的博客

这是个Before 是跟这个表面意思一样的意思，就是之前的方法切面

Before advice

```
<aop:aspect id="beforeExample" ref="aBean">

    <aop:before
        pointcut-ref="dataAccessOperation"
        method="doAccessCheck"/>

    ...

</aop:aspect>
```

```
<aop:aspect id="beforeExample" ref="aBean">

    <aop:before
        pointcut="execution(* com.xyz.myapp.dao..(..))"
        method="doAccessCheck"/>

    ...

</aop:aspect>
```

```
<bean id="moocAspect" class="aop.schema.advice.MoocAspect"></bean>

<bean id="aspectBiz" class="aop.schema.advice.biz.AspectBiz"></bean>

<aop:config>
   <aop:aspect id="moocAspectAOP" ref="moocAspect" >
        <aop:pointcut id="moocPiontuct" expression="execution(* aop.schema.advice.b
iz.AspectBiz.*(..))" />
          <aop:before method="before" pointcut-ref="moocPiontuct"/>
   </aop:aspect>
</aop:config>
```

这里上的<aop:pointcut>的具体解释在下面的链接中有讲，Pointcut 是指那些方法需要被执行"AOP",是由"Pointcut Expression"来描述
http://blog.csdn.net/kkdelta/article/details/7441829
<aop:before>是前置通知，就是我们指定我们切面的类

所有上面pointcut指定的类，都会被执行下面的这个before的前置通知

在这个配置中，上面的**aop pointcut上指定的ID与下面的Before是同一个 pointcut-ref是一个**，这样才能在一起配对使用，这是个认证（寻找匹配）的标识

advice

- **After returning advice**

```
<aop:aspect id="afterReturningExample" ref="aBean">
    <aop:after-returning
        pointcut-ref="dataAccessOperation"
        method="doAccessCheck"/>
    ...
</aop:aspect>
```

```
<aop:aspect id="afterReturningExample" ref="aBean">
    <aop:after-returning
        pointcut-ref="dataAccessOperation"
        returning="retVal"
        method="doAccessCheck"/>
    ...
```

这个after 是指定返回值

AfterReturning 增强处理将在目标方法正常完成后被织入。

使用@AfterReturning可指定如下两个属性：

可以限制返回值

① pointcut / value：两者都用于指定该切入点对应的切入表达式

② **returning**：指定一个返回值形参名，增强处理定义的方法可通过该行参名来访问目标方法的返回值。

- **After throwing advice**

```
<aop:aspect id="afterThrowingExample" ref="aBean">
    <aop:after-throwing
        pointcut-ref="dataAccessOperation"
        method="doRecoveryActions"/>
    ...
</aop:aspect>
```

- **使用throwing属性来指定可被传递的异常的参数名称**

```
<aop:aspect id="afterThrowingExample" ref="aBean">
    <aop:after-throwing
        pointcut-ref="dataAccessOperation"
        throwing="dataAccessEx"
        method="doRecoveryActions"/>
```

跟上面这个**AfterReturning** 的**returning** 长得很像，用法也差不多，都是用来指定返回值的名称

这个是抛出异常用的

这个时候，刚才那个AfterReturning不见了，因为这个时候也是后置通知，那个是正常的返回情况下的，但是现在是出错的情况下，所以会导致这样的情况也不足为奇

- **After (finally) advice**

```
<aop:aspect id="afterFinallyExample" ref="aBean">

    <aop:after
        pointcut-ref="dataAccessOperation"
        method="doReleaseLock"/>

    ...

</aop:aspect>
```

这个是after默认的方法，这个是默认执行的，是始终被执行的 。在返回之后，方法结束前的最后一行代码，去执行after adivce，这个是默认执行的，不因为前面的 after -throwing抛出异常而改变
无论方法是否正常结束，这个after都会被执行。
在下面贴代码：

```
<bean id="moocAspect" class="aop.schema.advice.MoocAspect"></bean>

<bean id="aspectBiz" class="aop.schema.advice.biz.AspectBiz"></bean>

<aop:config>
   <aop:aspect id="moocAspectAOP" ref="moocAspect" >
         <aop:pointcut id="moocPiontuct" expression="execution(* aop.schema.advice.biz.AspectBiz.*(..))" />
            <aop:before method="before"  pointcut-ref="moocPiontuct"/>
      <aop:after-returning method="afterReturning" pointcut-ref="moocPiontuct"/>
      <aop:after-throwing method="afterThrowing" pointcut-ref="moocPiontuct"/>
      <aop:after method="after" pointcut-ref="moocPiontuct"/>
      <aop:around method="around" pointcut-ref="moocPiontuct"/>
   </aop:aspect>
</aop:config>
```

- **Around advice**
- **通知方法的第一个参数必须是ProceedingJoinPoint类型**

```
<aop:aspect id="aroundExample" ref="aBean">

    <aop:around
        pointcut-ref="businessService"
        method="doBasicProfiling"/>

    ...

</aop:aspect>
```

```
public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {
    // start stopwatch
    Object retVal = pjp.proceed();
    // stop stopwatch
    return retVal;
```

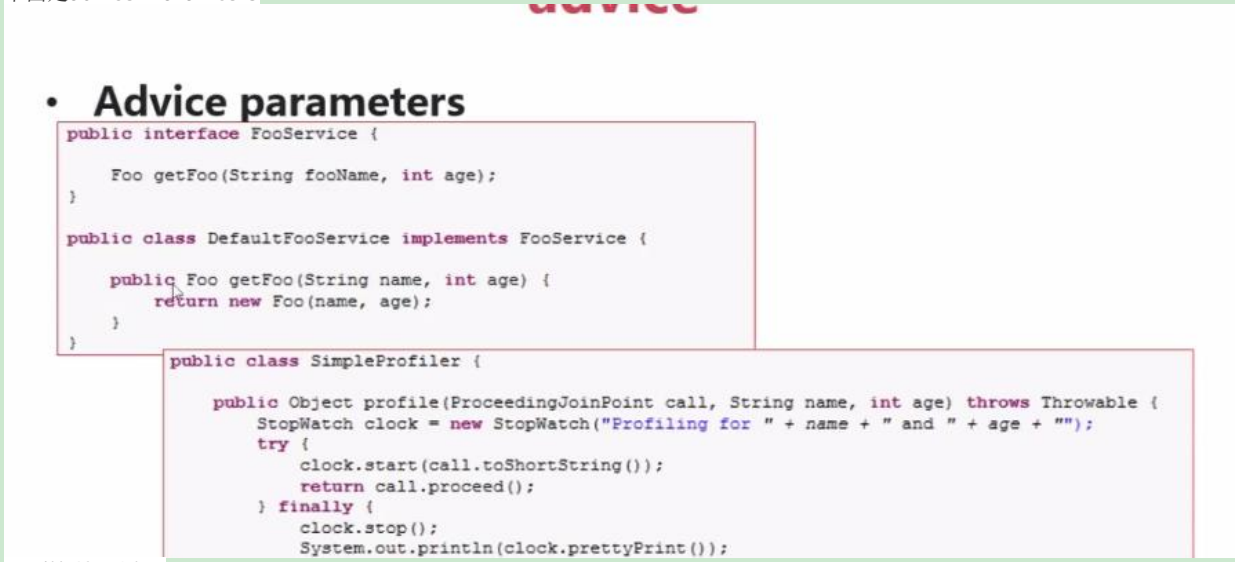
around advice是环绕通知，这个通知有一个特点，就是第一个参数，必须是ProceedingJoinPoint类型
下面贴代码：

```
public  Object around(ProceedingJoinPoint pjp) throws Throwable {
   System.out.println(" MoocAspect  around 1");
Object obj=pjp.proceed();
System.out.println(" MoocAspect  around 2");
   return  obj;
}
```

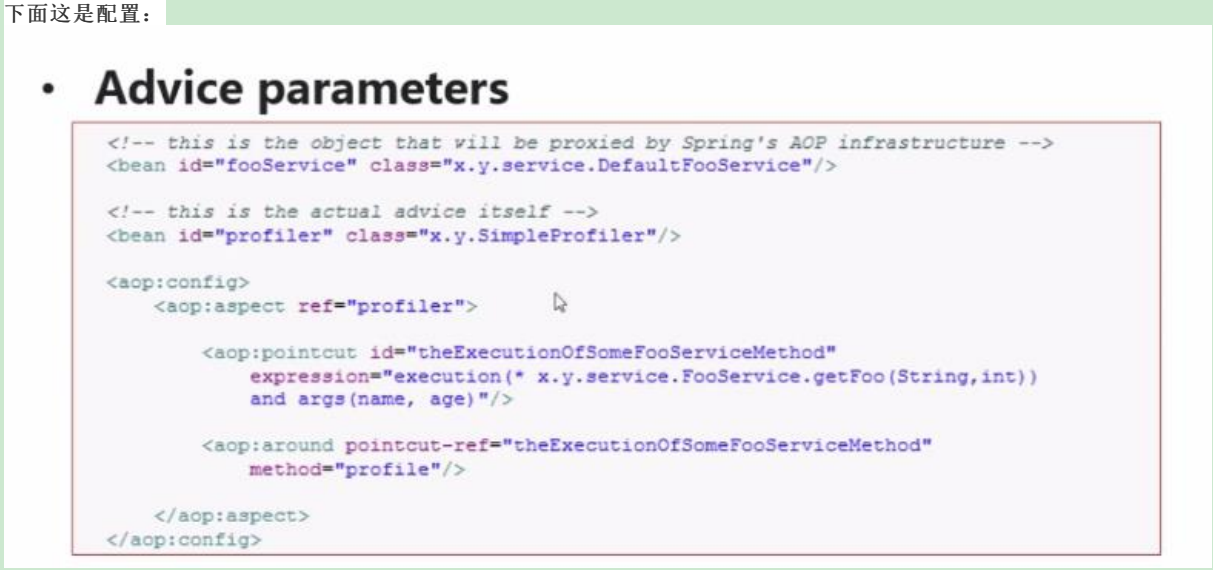
Around通知是最后一种通知类型。Around通知在匹配方法运行期的"周围"执行。 它有机会在目标方法的前面和后面执行，并决定什么时候运行，怎么运行，甚至是否运行。 **Around**通知经常在需要在一个方法执行前或后共享状态信息，并且是线程安全的情况下使用（启动和停止一个计时器就是一个例子）。 注意选择能满足你需求的最简单的通知类型（**i.e.**如果简单的**before**通知就能做的事情绝对不要使用**around**通知）。

Around通知使用 aop:around 元素来声明。 **通知方法的第一个参数的类型必须是 ProceedingJoinPoint 类型**。 在通知的主体中，调用ProceedingJoinPoint的proceed() 方法来执行真正的方法。 proceed 方法也可能会被调用并且传入一个 Object[] 对象 - 该数组将作为方法执行时候的参数。

下面是advice Paramters



下面这是配置：



Introductions

允许一个切面 声明一个实现指定接口 的通知对象 ，并且提供了一个接口实现类来代表这些对象。


```
<aop:aspect id="usageTrackerAspect" ref="usageTracking">

    <aop:declare-parents
        types-matching="com.xzy.myapp.service.*+"
        implement-interface="com.xyz.myapp.service.tracking.UsageTracked"
        default-impl="com.xyz.myapp.service.tracking.DefaultUsageTracked"/>

    <aop:before
        pointcut="com.xyz.myapp.SystemArchitecture.businessService()
            and this(usageTracked)"
        method="recordUsage"/>

</aop:aspect>
```

```
<aop:declare-parents types-matching="aop.schema.advice.biz.*(+)"
implement-interface="aop.schema.advice.Fit" default-impl="aop.schema.advice.FitImpl"
/>
```

这里是配置的地方，

```
ApplicationContext context=new
```

```
ClassPathXmlApplicationContext("spring-aop-schema-advice.xml");
Fit fit = (Fit) context.getBean("aspectBiz");
fit.filter();
```

这里是实现的地方，这里的话，用获取aspectBiz就可以了

types-matching 匹配的类型
implement-interface 实现指定接口实现类的 接口
default-impl 实现指定接口实现类的 类
这个的作用 我想就是给我们指定的类 类似的设置一个父类，然后我们代理这个对象，这个东西很有意思，然后我们就可以代理这个对象生成一些方法
属于代理模式中的静态代理。作用就是通过proxy代理了impl。实现并可以加强imple中的功能！假如说impl中只有一个方法a（），那么proxy就可以代理a（）并对a添加附加功能/设定访问权限等等
需要注意的是 这里代理出来的方法不再受到切面的影响 也就是不再享受到切面的before之类的功能
这是其他人的笔记
在这里呢，我们需要重新说明一下，前面的概念<aop:aspect>的是定义一个切面　　其中的属性ref是添加依赖，没有依赖的话，只要下面不引用的话也没关系，
我们在获取的时候，已经被转换完成了，这样才能被能被强制类型转换
所有基于配置文件的**aspect** 只支持单例模式

- schema-defined aspects只支持singleton model

Advisors
Advisor：充当Advice和Pointcut的适配器，类似使用Aspect的@Aspect注解的类 。一般有advice和pointcut属性。
advisor只持有一个Pointcut和一个advice，而aspect可以多个pointcut和多个advice
advisor就像一个小的自含包的方面，只有一个advice
切面自身通过一个Bean表示，并且必须实现某个advice接口，同时，**advisor也可以很好的利用AspectJ的切入点表达式**
Spring通过配置文件中<aop:advisor>元素支持advisor实际使用中，大多数情况下它会和transactional advice配合使用

★为了定义一个advisor的优先级以便让advice可以有序，可以使用order属性来定义advisor的顺序

下面是Advisor的配置

Advisors

```
<aop:config>

    <aop:pointcut id="businessService"
        expression="execution(* com.xyz.myapp.service..(..))"/>

    <aop:advisor
        pointcut-ref="businessService"
        advice-ref="tx-advice"/>

</aop:config>

<tx:advice id="tx-advice">
    <tx:attributes>
        <tx:method name="*" propagation="REQUIRED"/>
    </tx:attributes>
</tx:advice>
```

第一次复习留下的

```
<?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" xmlns:tx="http://www.springframework.org/schema/tx"
xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/bea
ns/spring-beans.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/c
ontext/spring-context.xsd http://www.springframework.org/schema/aop http://www.springframework.org/schema
/aop/spring-aop.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spr
```

```
ing-tx.xsd">



<context:component-scan base-package="com.Spring.Aop.Test"/>

       <aop:config>
           <aop:pointcut id="businessService" expression="execution(* com.Spring.Aop.Test.*.*(..))"/>
           <aop:aspect id="MyAspect" ref="myAspect">
                  <aop:around method="around" pointcut-ref="businessService"/>
                     <aop:before method="before" pointcut-ref="businessService"  />
                       <aop:after method="After" pointcut-ref="businessService"/>
                      <aop:after-returning method="AfterReturing"  pointcut-ref="businessService" returni
ng="retval"/>
                  <aop:after-throwing method="AfterThrowing" pointcut-ref="businessService" />
                 <aop:declare-parents types-matching="com.Spring.Aop.Test.Test02" implement-interface="co
m.Spring.Aop.Test.iTest01"
default-impl="com.Spring.Aop.Test.Test01"
/>

              </aop:aspect>

       </aop:config>


           <bean id="throwingTest" class="com.Spring.Aop.ThrowingTest"/>
       <bean id="myAspect" class="com.Spring.Aop.MyAspect"/>




</beans>
```

切面类 Myaspect类

```
package com.Spring.Aop;

import org.aopalliance.intercept.Joinpoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * Created by han on 2016/8/31.
 */
public class MyAspect {

public  void before(  ){
System.out.println("我是 before。。");
    }

public  void  After(){
System.out.println("我是After...");
    }

public  void AfterReturing (int retval){
System.out.println("我是 afterReturning"+"  "+retval);
    }

public  void AfterThrowing(){
System.out.println("我是 AfterThrowing。。。");
    }



public  Object around(ProceedingJoinPoint pjp) throws Throwable {
System.out.println("环绕之前的吊样");
Object obj=pjp.proceed();

System.out.println("环绕之后的吊样....");
return obj;
    }
}
```

iTest1 这个接口 主要是因为我使用的是@configuration这个注解来标注Bean才会用到接口

```
public interface iTest01{
public  void Test01();
public  int  Test012Z();

public  void  Test013(int re);
}
```

具体的Bean类

```
package com.Spring.Aop.Test;

import com.Spring.Aop.MyAspect;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * Created by han on 2016/8/31.
 */
@Configuration
public class Test01 implements iTest01 {


public  void Test01(){
System.out.println("我是Test01。。。");
    }

@Override
public int Test012Z() {
System.out.println("我是Test012 ....");
return 10;
    }

@Override
public void Test013(int re) {
System.out.println("我是Test013"+re);
    }

public static void main(String[] args) {
ApplicationContext context = new ClassPathXmlApplicationContext("Spring/applicationcontext3.xml");
iTest01 test01 = (iTest01) context.getBean("test01");
test01.Test01();
test01.Test012Z();
test01.Test013(1);

    }
}
```

用来测试Around环绕的Test02类

```
@Component
public class Test02 {

public static void main(String[] args) {
ApplicationContext context = new ClassPathXmlApplicationContext("Spring/applicationcontext3.xml");
iTest01   test02 =(iTest01)context.getBean("test02");
test02.Test01();

    }
}
```