# 建立目标对象

前置增强，后置增强，环绕增强使用该目标对象

## 目标对象接口：

**package** com.yang.springaoppractice.service;

/\*\*

\* 业务模拟接口

\* **@author** Administrator

\*

\*/

**public** **interface** ServiceSimulator {

**void** select();

**void** delete();

}

## 目标对象类：

### ServiceSimulatorImpl：

**package** com.yang.springaoppractice.service;

**import** org.springframework.stereotype.Component;

/\*\*

\* 业务模拟实现类，用于作为添加动态代理的target

\*

\* **@Component**("serviceSimulator") 自动生成该类的bean，名称为"serviceSimulator"

\*

\* **@author** Administrator

\*

\*/

@Component("serviceSimulator")

**public** **class** ServiceSimulatorImpl **implements** ServiceSimulator {

/\*\*

\* 模拟select方法

\*/

@Override

**public** **void** select() {

**try** {

System.***out***.println("Simulating select......");

Thread.*currentThread*().*sleep*(500);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

/\*\*

\* 模拟delete方法

\*/

@Override

**public** **void** delete() {

**try** {

System.***out***.println("Simulating delete......");

Thread.*currentThread*().*sleep*(300);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

/\*\*

\* 模拟write方法

\*/

@Override

**public** **void** write(String string) {

System.***out***.println(string);

}

}

### ServiceSimulatorImpl2：

**package** com.yang.springaoppractice.service;

**import** org.springframework.stereotype.Component;

@Component("serviceSimulatorImpl2")

**public** **class** ServiceSimulatorImpl2 **implements** ServiceSimulator {

/\*\*

\* 模拟select方法

\*/

@Override

**public** **void** select() {

**try** {

System.***out***.println("Simulating select......");

Thread.*currentThread*().*sleep*(500);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

/\*\*

\* 模拟delete方法

\*/

@Override

**public** **void** delete() {

**try** {

System.***out***.println("Simulating delete......");

Thread.*currentThread*().*sleep*(300);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

/\*\*

\* 模拟write方法

\*/

@Override

**public** **void** write(String string) {

System.***out***.println(string);

}

}

# 建立增强（advice）

## 增强（advice）概念：

Spring通过**增强类定义横切逻辑**，同时还包括了在方法的那一点上加入横切代码的**方位**信息，所以，增强即包括了横切逻辑，还包含部分连接点信息。（简单来说就是增强（advice） = 横切逻辑 + 方位信息）

## 增强（advice）的类型：

* 前置增强
* 后置增强
* 环绕增强
* 异常抛出增强
* 引介增强

## 定义及使用示例：

### 前置增强：

定义增强代码：

**package** com.yang.springaoppractice.advice;

**import** java.lang.reflect.Method;

**import** org.springframework.aop.MethodBeforeAdvice;

**import** org.springframework.stereotype.Component;

/\*\*

\* 前置增强（MethodBeforeAdvice）

\*

\* **@Component**("methodStartReminderAdvice") 自动生成该类的bean，名称为"methodStartReminderAdvice"

\*

\* **@author** Administrator

\*

\*/

@Component("methodStartReminderAdvice")

**public** **class** MethodStartReminderAdvice **implements** MethodBeforeAdvice{

@Override

**public** **void** before(Method arg0, Object[] arg1, Object arg2) **throws** Throwable {

/\*\*

\* 该语句在目标对象方法前执行

\*/

System.***out***.println("The method "

+ arg0.getName()

+ " in the class "

+ arg2.getClass().getName()

+ " is starting......");

}

}

### 后置增强：

定义增强代码：

**package** com.yang.springaoppractice.advice;

**import** java.lang.reflect.Method;

**import** org.springframework.aop.AfterReturningAdvice;

**import** org.springframework.stereotype.Component;

/\*\*

\* 后置增强（AfterReturningAdvice）

\*

\* **@Component**("methodEndReminderAdvice") 自动生成该类的bean，名称为"methodEndReminderAdvice"

\*

\* **@author** Administrator

\*

\*/

@Component("methodEndReminderAdvice")

**public** **class** MethodEndReminderAdvice **implements** AfterReturningAdvice {

@Override

**public** **void** afterReturning(Object arg0, Method arg1, Object[] arg2, Object arg3) **throws** Throwable {

/\*\*

\* 代码在目标对象方法运行后运行

\*/

System.***out***.println("The method is end");

}

}

### 环绕增强：

定义增强代码：

**package** com.yang.springaoppractice.advice;

**import** java.lang.reflect.Method;

**import** org.aopalliance.intercept.MethodInterceptor;

**import** org.aopalliance.intercept.MethodInvocation;

**import** org.springframework.cglib.proxy.MethodProxy;

**import** org.springframework.stereotype.Component;

/\*\*

\* 环绕增强（MethodInterceptor）

\*

\* **@Component**("methodPerformanceCheck") 自动生成该类的bean，名称为"methodPerformanceCheck"

\*

\* **@author** Administrator

\*

\*/

@Component("methodPerformanceCheck")

**public** **class** MethodPerformanceCheck **implements** MethodInterceptor{

@Override

**public** Object invoke(MethodInvocation arg0) **throws** Throwable {

/\*\*

\* 代码在目标对象方法运行前运行

\*/

System.***out***.println("The method has started......");

**long** start = System.*currentTimeMillis*();

/\*\*

\* 此处代码实质为调用目标对象方法

\*/

Object result = arg0.proceed();

/\*\*

\* 代码在目标对象方法运行前运行

\*/

**long** end = System.*currentTimeMillis*();

System.***out***.println("The method has ended");

**long** cost = end - start;

System.***out***.println("The method cost " + cost + " millisecond");

**return** result;

}

}

### 异常抛出增强：

#### 异常抛出增强目标对象类：

**package** com.yang.springaoppractice.service;

**import** java.sql.SQLException;

**import** org.springframework.stereotype.Component;

/\*\*

\* 异常抛出增强目标对象类,模拟抛出异常

\*

\* **@Component**("throwExceptionSimulator") 自动生成该类的bean，名称为"throwExceptionSimulator"

\*

\* **@author** Administrator

\*

\*/

@Component("throwExceptionSimulator")

**public** **class** ThrowExceptionSimulator {

/\*\*

\* 该方法模拟运行时抛出RuntimeException

\*/

**public** **void** remove(){

**throw** **new** RuntimeException("This is the RuntimeException that I throw");

}

/\*\*

\* 该方法模拟运行时抛出SQLException异常

\* **@throws** SQLException

\*/

**public** **void** update() **throws** SQLException{

**throw** **new** SQLException("This is the SQLException that I throw");

}

}

#### 定义增强代码：

**package** com.yang.springaoppractice.advice;

**import** java.lang.reflect.Method;

**import** org.springframework.aop.ThrowsAdvice;

**import** org.springframework.stereotype.Component;

/\*\*

\* 异常抛出增强（ThrowsAdvice）

\*

\* **@Component**("exceptionInterceptor") 自动生成该类的bean，名称为"exceptionInterceptor"

\*

\* **@author** Administrator

\*

\*/

@Component("exceptionInterceptor")

**public** **class** ExceptionInterceptor **implements** ThrowsAdvice {

**public** **void** afterThrowing(Method method, Object[] args, Object target,Exception ex){

/\*\*

\* 目标对象方法抛出异常时运行

\*/

System.***out***.println("Now this is where I going to complain about the Exceptions");

}

}

### **引介增强**：

引介增强是相当奇特的增强，有点相当于为代理对象增加新功能

此处的引介增强代码单独在spring-introduce-advice 项目中

#### Note：

貌似引介增强不能和注释自动注入一起使用，否则会遇到cast（强转）问题

使用引介增强还会遇到其他很奇怪的cast（强转）问题，不知道原因是什么

#### 引介增强目标对象类：

**package** com.yang.springaoppractice.advice;

**import** org.springframework.stereotype.Component;

/\*\*

\*

\* 引介增强的目标对象

\*

\* **@author** Ryan

\*

\*/

**public** **class** ForumServiceForIntroduce {

**public** **void** removeTopic(**int** topicId) {

System.***out***.println("模拟删除Topic记录:"+topicId);

**try** {

Thread.*currentThread*().*sleep*(20);

} **catch** (Exception e) {

**throw** **new** RuntimeException(e);

}

}

**public** **void** removeForum(**int** forumId) {

System.***out***.println("模拟删除Forum记录:"+forumId);

**try** {

Thread.*currentThread*().*sleep*(40);

} **catch** (Exception e) {

**throw** **new** RuntimeException(e);

}

}

}

#### 引介增强的横切逻辑代码类：

**package** com.yang.springaoppractice.advice;

/\*\*

\* 模拟方法的性能

\*

\* 封装“调用开始时间”和“调用结束时间”

\*

\* 二者相减，得到调用方法花费的总时间

\*

\* **@author** Ryan

\*

\*/

**public** **class** MethodPerformace {

/\*\*

\* 开始时间

\*/

**private** **long** begin;

/\*\*

\* 结束时间

\*/

**private** **long** end;

/\*\*

\* 方法名称

\*/

**private** String serviceMethod;

/\*\*

\* 构造

\*/

**public** MethodPerformace(String serviceMethod){

reset(serviceMethod);

}

/\*\*

\* 打印性能

\*/

**public** **void** printPerformace(){

end = System.*currentTimeMillis*();

**long** elapse = end - begin;

System.***out***.println(serviceMethod+"花费"+elapse+"毫秒。");

}

/\*\*

\* 重设

\* **@param** serviceMethod

\*/

**public** **void** reset(String serviceMethod){

**this**.serviceMethod = serviceMethod;

**this**.begin = System.*currentTimeMillis*();

}

}

#### 引介增强的引介功能接口：

**package** com.yang.springaoppractice.advice;

/\*\*

\*

\* 提供给代理使用，用于扩展代理的功能

\*

\* **@author** Ryan

\*

\*/

**public** **interface** Monitorable {

/\*\*

\* 是否开启方法的监听

\* **@param** active

\*/

**void** setMonitorActive(**boolean** status);

}

#### 定义增强代码：

**package** com.yang.springaoppractice.advice;

**import** org.aopalliance.intercept.MethodInvocation;

**import** org.springframework.aop.support.DelegatingIntroductionInterceptor;

**import** org.springframework.stereotype.Component;

/\*\*

\*

\* 引介增强：

\*

\* （1）引介增强是一种比较特殊的的增强类型，它不是在目标方法周围织入增强，而是为目标类创建新的方法和属性，所以引介增强的连接点是类级别的，而非方法级别。

\* （2）通过引介增强，可以为目标类增加一个接口的实现，即原来目标类未实现某个接口，通过引介增强可以为目标类创建实现该接口的代理。

\*

\* **@author** Ryan

\*

\*/

**public** **class** ControllablePerformaceMonitor **extends**

DelegatingIntroductionInterceptor **implements** Monitorable {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** **boolean** status;

**public** **void** setMonitorActive(**boolean** status) {

}

**public** Object invoke(MethodInvocation mi) **throws** Throwable {

Object obj = **null**;

/\*\*

\* 只有在监听打开的时候，才启用方法的性能检测

\*/

**if** (status) {

/\*\*

\* 横切逻辑

\*/

PerformanceMonitor.*begin*(mi.getClass().getName() + "."

+ mi.getMethod().getName());

obj = **super**.invoke(mi);

PerformanceMonitor.*end*();

} **else** {

obj = **super**.invoke(mi);

}

**return** obj;

}

}

#### Beans.xml的配置：

<?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:p=*"http://www.springframework.org/schema/p"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.0.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-3.0.xsd"*>

<!-- 引介增强 -->

<bean id=*"pmonitor"* class=*"com.yang.springaoppractice.advice.ControllablePerformaceMonitor"* />

<!-- 代理的目标对象，基于类的动态代理，CGLib动态代理-->

<bean id=*"forumServiceTarget"* class=*"com.yang.springaoppractice.advice.ForumServiceForIntroduce"* />

<!-- 代理对象 -->

<bean id=*"forumService"* class=*"org.springframework.aop.framework.ProxyFactoryBean"*

p:interfaces=*"com.yang.springaoppractice.advice.Monitorable"*

p:target-ref=*"forumServiceTarget"*

p:interceptorNames=*"pmonitor"*

p:proxyTargetClass=*"true"* />

</beans>

#### Junit单元测试：

**package** com.yang.springaoppractice.advice;

**import** org.junit.Test;

**import** org.junit.runner.RunWith;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.test.context.ContextConfiguration;

**import** org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

/\*\*

\* 测试引介增强

\*

\* **@author** Ryan

\*

\*/

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration(locations = { "/beans.xml" })

**public** **class** IntroduceAdviceTests {

@Autowired

**private** ApplicationContext context;

@Test

**public** **void** testIntroduceAdvice() {

/\*\*

\* 方法性能监控关闭（默认）

\*/

ForumServiceForIntroduce forumService = (ForumServiceForIntroduce) context.getBean("forumService");

forumService.removeForum(10);

/\*\*

\* 方法性能监控打开

\*/

Monitorable moniterable = (Monitorable) forumService;

moniterable.setMonitorActive(**true**);

forumService.removeForum(10);

}

}

### 增强的代理对象配置及使用：

Advice的代理对象配置在示例项目advicebeans.xml中

Advice的使用可参照示例项目junit测试AdviceTest.java

# 建立切面（advisor）

## Note：

Spring的aop jar表下有很多定义好的切点，一般情况下是不需要自己定义切点的，下面的代码模拟了aop 如何实现，不可作为使用示例

## 基本概念：

一般切面（Advisor）：增强(Advice)就是一个一般切面(Advisor)，但是因为**横切面太宽泛**，因为他包含了横切连接点是所有类的所有方法。**很少使用**。

**切点切面**（PointcutAdvisor）：包含Advice和Pointcut。提供更具灵活特性的切面。具有实际意义的切面

引介切面（IntroductionAdvisor）：和引介增强搭配使用。专门配合引介增强使用。

习惯上提及切面时都是指**切点切面**

生成advisor bean的配置有两种：请参照笔记-生成Advisor（切面）bean的两种通用方法

## 静态普通方法名匹配切点切面（可过滤class，method）

### 定义切点：

**package** com.yang.springaoppractice.advisor;

**import** java.lang.reflect.Method;

**import** org.springframework.aop.support.StaticMethodMatcherPointcut;

**import** com.yang.springaoppractice.service.ServiceSimulatorImpl;

/\*\*

\* 静态普通方法名匹配切点切面的切点（可过滤class，method）

\*

\* 定义切点（point cut）

\*/

**public** **class** MethodStartReminderPointcut **extends** StaticMethodMatcherPointcut {

/\*\*

\* 定义只有当目标对象class为ServiceSimulatorImpl.class及调用的方法为select()时，

\* 才触发代理

\*/

@Override

**public** **boolean** matches(Method method, Class<?> targetClass) {

**if**("select".equals(method.getName())

&& ServiceSimulatorImpl.**class**.isAssignableFrom(targetClass)){

**return** **true**;

}

**else**{

**return** **false**;

}

}

}

### 配置xml生成advisor及代理对象的beans：

<!-- 静态普通方法名匹配切点切面 -->

<!-- 生成advisor bean的配置有两种：请参照笔记-

生成Advisor（切面）bean的两种通用方法-->

<!-- 配置生成advisor（增强） bean：

通过加入 MethodStartReminderAdvice增强的配置-->

<!-- <bean id="methodStartReminderAdvisor"

class="com.yang.springaoppractice.advisor.MethodStartReminderAdvisor"

p:advice-ref="methodStartReminderAdvice"/> -->

<!-- 配置生成advisor（增强） bean：

通过加入 MethodStartReminderAdvice增强

及MethodStartReminderPointcut切点的配置 -->

<bean id=*"methodStartReminderAdvisor"*

class=*"org.springframework.aop.support.DefaultPointcutAdvisor"*

p:advice-ref=*"methodStartReminderAdvice"*>

<property name=*"pointcut"*>

<bean class=*"com.yang.springaoppractice.advisor.MethodStartReminderPointcut"*/>

</property>

</bean>

<!-- 配置代理对象bean，

配置两个 代理对象bean是为了对比-->

<bean id=*"parentMethodStartReminderAdvisor"*

class=*"org.springframework.aop.framework.ProxyFactoryBean"*

p:interceptorNames=*"methodStartReminderAdvisor"*

p:proxyInterfaces=*"com.yang.springaoppractice.service.ServiceSimulator"*/>

<bean id=*"proxyMethodStartReminderAdvisor"*

parent=*"parentMethodStartReminderAdvisor"*

p:target-ref=*"serviceSimulator"*/>

<bean id=*"proxyMethodStartReminderAdvisor2"*

parent=*"parentMethodStartReminderAdvisor"*

p:target-ref=*"serviceSimulatorImpl2"*/>

### Test：

自动注入beans：

@Autowired

**private** ServiceSimulator proxyMethodStartReminderAdvisor;

@Autowired

**private** ServiceSimulator proxyMethodStartReminderAdvisor2;

测试代码：

@Test

**public** **void** testMethodStartReminderAdvisor(){

System.***out***.println("Begin testMethodStartReminderAdvisor():");

//进入切面引入的增强逻辑

proxyMethodStartReminderAdvisor.select();

//不会进入切面引入的增强逻辑

proxyMethodStartReminderAdvisor.delete();

//不会进入切面引入的增强逻辑

proxyMethodStartReminderAdvisor2.select();

//不会进入切面引入的增强逻辑

proxyMethodStartReminderAdvisor2.delete();

}

## 静态正则表达式方法匹配切点切面（只能过滤method）

### Note：

使用静态正则表达式方法匹配切点切面无需在类中定义切点或切面，这些可直接用xml配置完成

### 配置xml生成advisor及代理对象的beans：

<!-- 静态正则表达式方法匹配切点切面

在此处定义切点（point cut）及

加入MethodEndReminderAdvice增强

目前情况看不能过滤类

ps:切面（advisor）= 切点（point cut）+ 增强（advice）

-->

<bean id=*"methodEndReminderAdvisor"*

class=*"org.springframework.aop.support.RegexpMethodPointcutAdvisor"*

p:advice-ref=*"methodEndReminderAdvice"*>

<!-- 定义正则表达式切点 -->

<property name=*"patterns"*>

<list>

<value>.\*delete.\*</value>

</list>

</property>

</bean>

<!-- 配置代理对象bean -->

<bean id=*"proxyMethodEndReminderAdvisor"*

class=*"org.springframework.aop.framework.ProxyFactoryBean"*

p:target-ref=*"serviceSimulator"*

p:interceptorNames=*"methodEndReminderAdvisor"*

p:proxyInterfaces=*"com.yang.springaoppractice.service.ServiceSimulator"*/>

### Test：

自动注入beans

@Autowired

**private** ServiceSimulator proxyMethodEndReminderAdvisor;

测试代码：

@Test

**public** **void** testMethodEndReminderAdvisor(){

System.***out***.println("Begin testMethodEndReminderAdvisor():");

//不会进入切面引入的增强逻辑

proxyMethodEndReminderAdvisor.select();

//进入切面引入的增强逻辑

proxyMethodEndReminderAdvisor.delete();

}

## 动态切点切面（可过滤class，method，argument）

### Note：

动态切点切面无法直接定义切面

### 定义切点：

**package** com.yang.springaoppractice.advisor;

**import** java.lang.reflect.Method;

**import** java.util.ArrayList;

**import** org.springframework.aop.support.DynamicMethodMatcher;

**import** org.springframework.aop.support.DynamicMethodMatcherPointcut;

**import** org.springframework.stereotype.Component;

**import** com.yang.springaoppractice.service.ServiceSimulatorImpl;

/\*\*

\* 动态切点切面（可过滤class，method，argument）

\*

\* 定义切点

\*

\* 动态切点切面无法直接定义切面

\*

\* ps:切面（advisor）= 切点（point cut）+ 增强（advice）

\*

\* **@author** Administrator

\*

\*/

@Component("methodPerformanceCheckPointcut")

**public** **class** MethodPerformanceCheckPointcut **extends** DynamicMethodMatcherPointcut {

/\*\*

\* 可触发代理的String参数的集合

\*/

**private** **static** ArrayList<String> *theList* = **new** ArrayList();

**static**{

*theList*.add("hey");

*theList*.add("hallo");

}

/\*\*

\* 定义只有当目标对象class为ServiceSimulatorImpl.class及调用的方法为write(String string)，

\* 并且参数string在theList中时，触发代理

\*/

@Override

**public** **boolean** matches(Method method, Class<?> targetClass, Object[] args) {

String string = (String) args[0];

**if**(ServiceSimulatorImpl.**class**.isAssignableFrom(targetClass)

&& "write".equals(method.getName())

&& *theList*.contains(string)){

**return** **true**;

}

**else**{

**return** **false**;

}

}

}

### 配置xml生成advisor及代理对象的beans：

<!-- 动态切点切面

在此处加入MethodPerformanceCheck增强

及加入MethodPerformanceCheckPointcut切点-->

<bean id=*"methodPerformanceCheckAdvisor"*

class=*"org.springframework.aop.support.DefaultPointcutAdvisor"*

p:advice-ref=*"methodPerformanceCheck"*

p:pointcut-ref=*"methodPerformanceCheckPointcut"*>

</bean>

<!-- 配置代理对象bean，

配置两个 代理对象bean是为了对比-->

<bean id=*"parentMethodPerformanceCheckAdvisor"*

class=*"org.springframework.aop.framework.ProxyFactoryBean"*

p:interceptorNames=*"methodPerformanceCheckAdvisor"*

p:proxyInterfaces=*"com.yang.springaoppractice.service.ServiceSimulator"*/>

<bean id=*"proxyMethodPerformanceCheckAdvisor"*

parent=*"parentMethodPerformanceCheckAdvisor"*

p:target-ref=*"serviceSimulator"*/>

<bean id=*"proxyMethodPerformanceCheckAdvisor2"*

parent=*"parentMethodPerformanceCheckAdvisor"*

p:target-ref=*"serviceSimulatorImpl2"*/>

### Test：

自动注入beans：

@Autowired

**private** ServiceSimulator proxyMethodPerformanceCheckAdvisor;

@Autowired

**private** ServiceSimulator proxyMethodPerformanceCheckAdvisor2;

测试代码：

@Test

**public** **void** testMethodPerformanceCheckAdvisor(){

System.***out***.println("Begin testMethodPerformanceCheckAdvisor():");

//进入切面引入的增强逻辑

proxyMethodPerformanceCheckAdvisor.write("hey");

//不会进入切面引入的增强逻辑

proxyMethodPerformanceCheckAdvisor.write("fuck");

//不会进入切面引入的增强逻辑

proxyMethodPerformanceCheckAdvisor2.write("hey");

//不会进入切面引入的增强逻辑

proxyMethodPerformanceCheckAdvisor2.write("hallo");

}

## 流程控制切点切面（ControlFlowPointcut）

### Note：

使用静态正则表达式方法匹配切点切面无需在类中定义切点或切面，这些可直接用xml配置完成

### 定义流程控制切点所作用的类

**package** com.yang.springaoppractice.service;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Component;

/\*\*

\* 流程控制切点（ControlFlowPointcut）所作用的类

\*

\* **@Component**("controlFlowService") 自动生成该类的bean，名称为"controlFlowService"

\*

\* **@author** Administrator

\*

\*/

@Component("controlFlowService")

**public** **class** ControlFlowService {

/\*\*

\* 注入代理对象，如此才可以调用代理对象方法

\*/

@Autowired

**private** ServiceSimulator proxyControlFlowAdvisor;

/\*\*

\* 通过调用该方法，间接调用目标对象上方法，触发代理

\*/

**public** **void** methodBeProxy(){

proxyControlFlowAdvisor.select();

proxyControlFlowAdvisor.delete();

}

}

### 配置xml生成advisor及代理对象的beans：

<!-- 流程切点切面 -->

<!-- 配置生成流程切点切面的切点 的bean-->

<bean id=*"pointcutOnControlFlowService"*

class=*"org.springframework.aop.support.ControlFlowPointcut"*>

<!-- 指定流程切点切面所作用的类 -->

<constructor-arg type=*"java.lang.Class"*

value=*"com.yang.springaoppractice.service.ControlFlowService"*/>

<!-- 指定流程切点切面所作用的方法 -->

<constructor-arg type=*"java.lang.String"*

value=*"methodBeProxy"*/>

</bean>

<!-- 配置生成流程切点切面 的bean-->

<bean id=*"controlFlowAdvisor"*

class=*"org.springframework.aop.support.DefaultPointcutAdvisor"*

p:pointcut-ref=*"pointcutOnControlFlowService"*

p:advice-ref=*"methodPerformanceCheck"*/>

<!-- 配置生成代理对象 -->

<bean id=*"proxyControlFlowAdvisor"*

class=*"org.springframework.aop.framework.ProxyFactoryBean"*

p:target-ref=*"serviceSimulator"*

p:interceptorNames=*"controlFlowAdvisor"*

p:proxyInterfaces=*"com.yang.springaoppractice.service.ServiceSimulator"*/>

### Test:

自动注入bean

@Autowired

**private** ServiceSimulator proxyControlFlowAdvisor;

@Autowired

**private** ControlFlowService controlFlowService;

测试代码：

@Test

**public** **void** testControlFlowAdvisor(){

System.***out***.println("Begin testControlFlowAdvisor():");

//不会进入切面引入的增强逻辑

proxyControlFlowAdvisor.select();

//不会进入切面引入的增强逻辑

proxyControlFlowAdvisor.delete();

//会进入切面引入的增强逻辑，因为切面的切点作用在ControlFlowService类上的methodBeProxy()方法

controlFlowService.methodBeProxy();

}