

V

Java私塾 《深入浅出学Spring3》 ——系列精品教程

10101010101010101010101010101



配套视频和源码请到私塾在线下载。

<http://sishuok.com>



《深入浅出学Spring3开发》——系列精品教程

本节课程概览

n AOP入门

包括：是什么、能干什么、AOP基本思想的演变

真正高质量培训 签订就业协议

网 址：<http://www.javass.cn>
咨询QQ：460190900



《深入浅出学Spring3开发》——系列精品教程

第三章：AOP开发

真正高质量培训 签订就业协议

网 址：<http://www.javass.cn>
咨询QQ：460190900



AOP入门-1

n AOP是什么 (Aspect Oriented Programming)

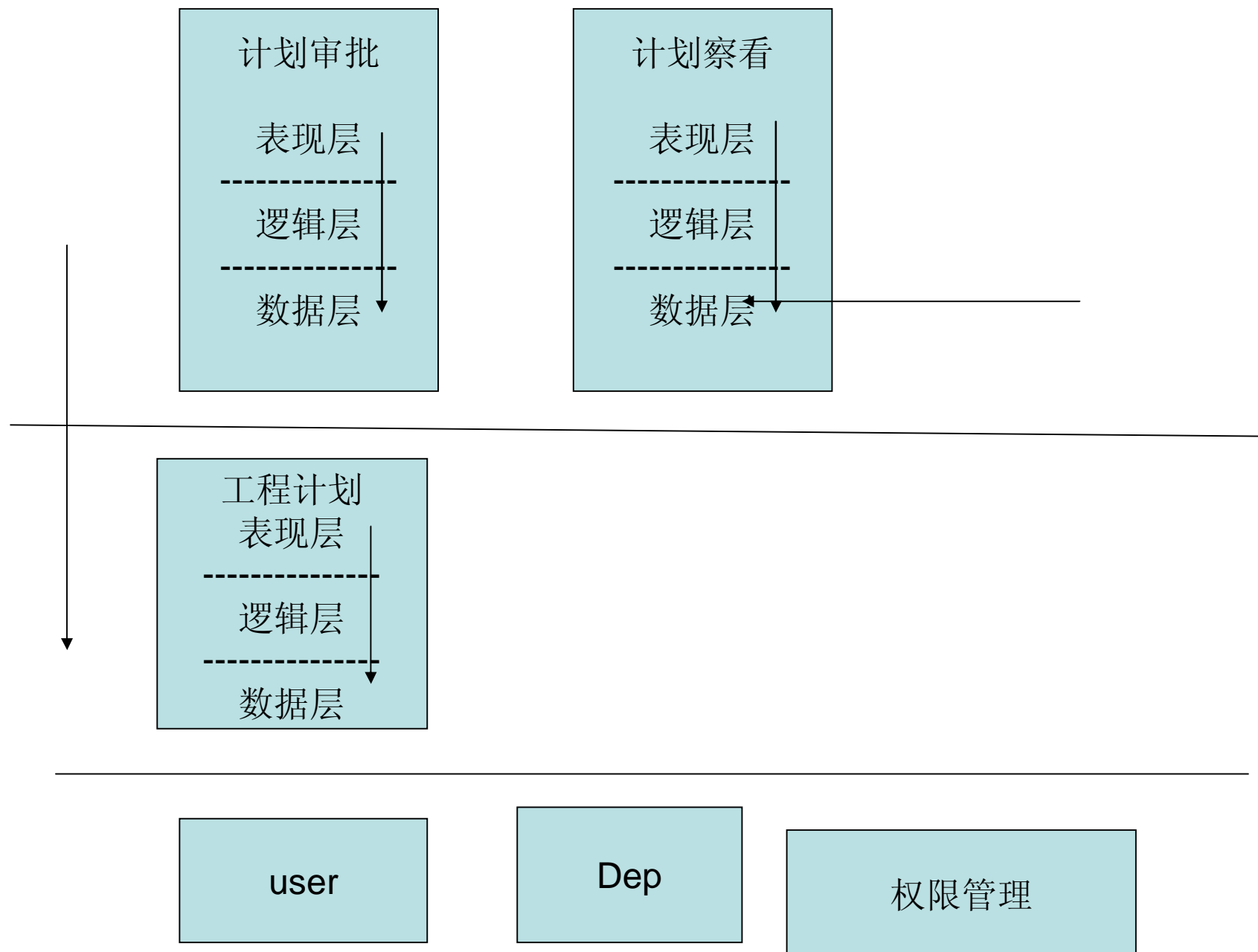
AOP是一种编程范式，提供从另一个角度来考虑程序结构以完善面向对象编程（OOP）。

AOP为开发者提供了一种描述横切关注点的机制，并能够自动将横切关注点织入到面向对象的软件系统中，从而实现了横切关注点的模块化。

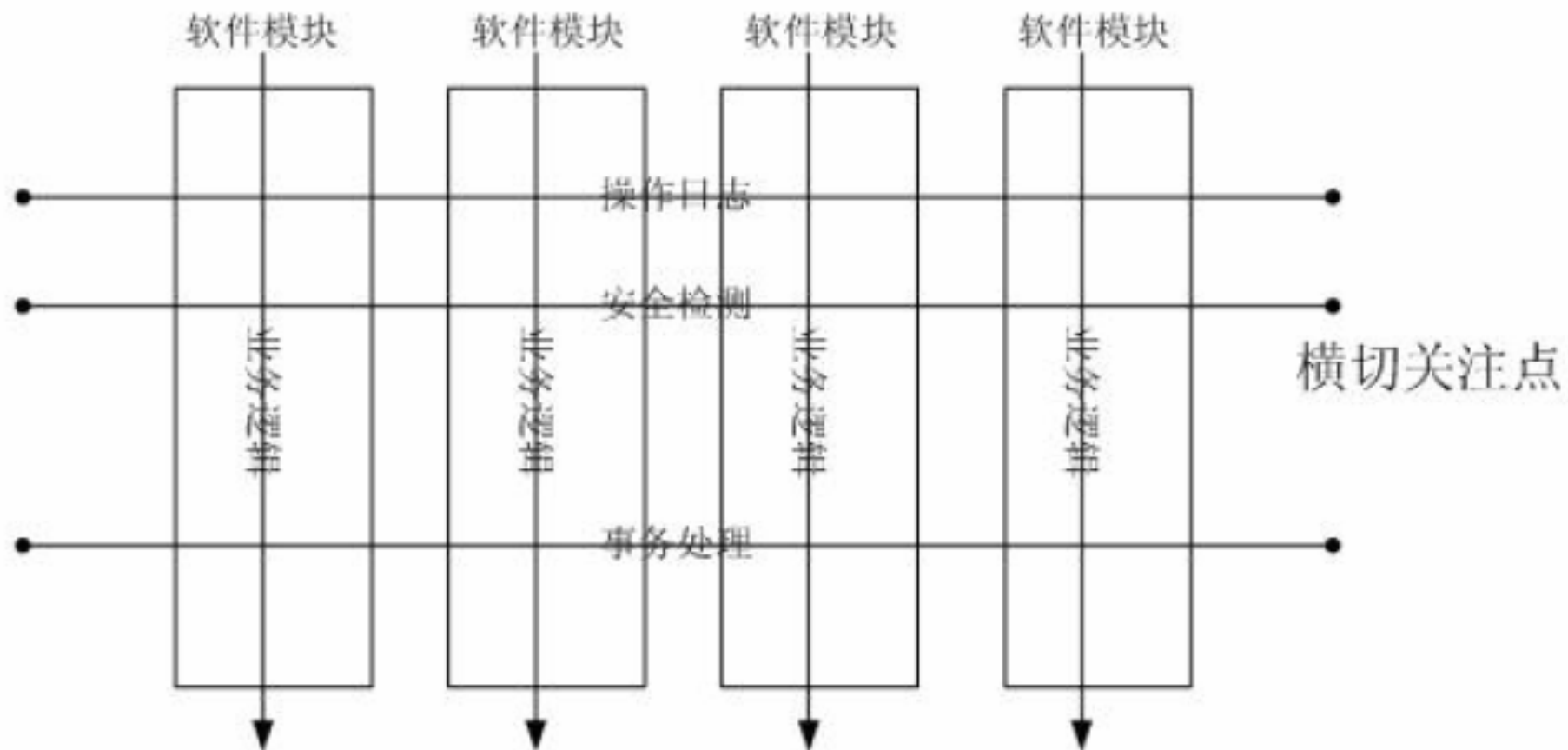
AOP能够将那些与业务无关，却为业务模块所共同调用的逻辑或责任，例如事务处理、日志管理、权限控制等，封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。

n AOP能干什么，也是AOP带来的好处

- 1: 降低模块的耦合度
- 2: 使系统容易扩展
- 3: 设计决定的迟绑定：使用AOP, 设计师可以推迟为将来的需求作决定，因为它可以把这种需求作为独立的方面很容易的实现。
- 4: 更好的代码复用性



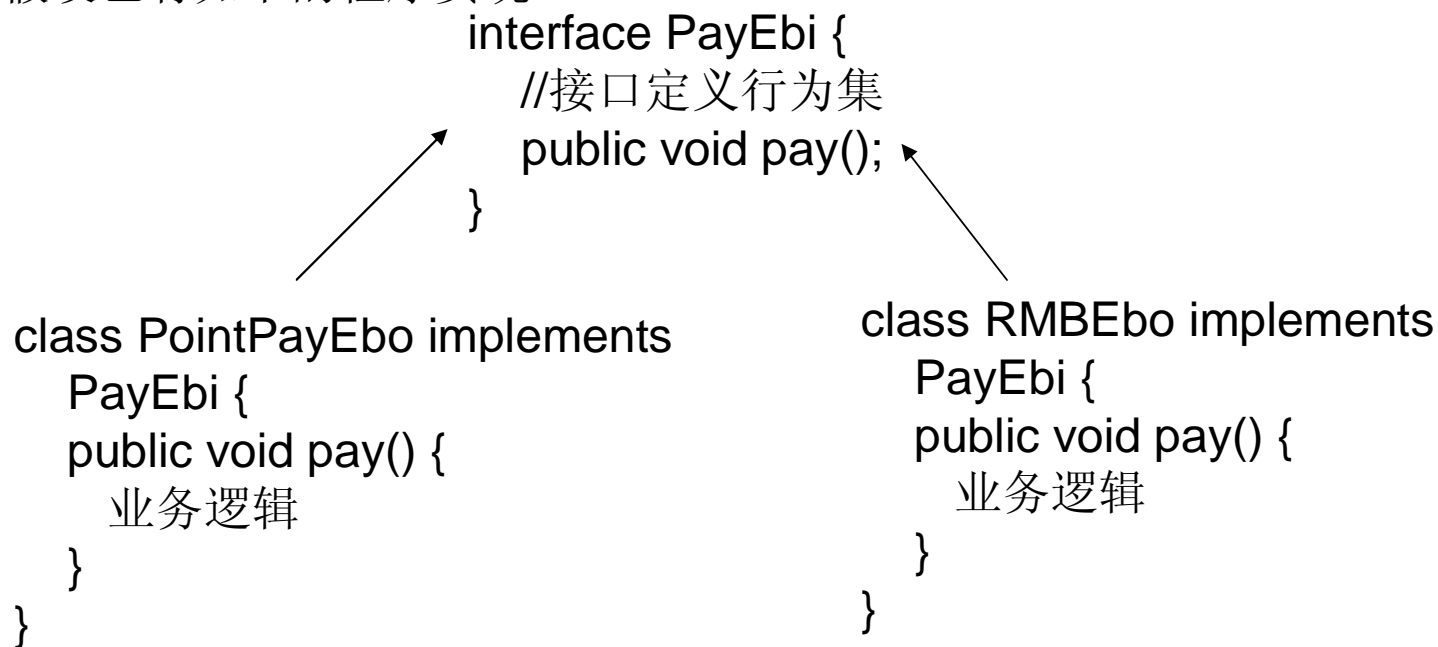
AOP入门-2





AOP基本思想的演变过程-1

n 假设已有如下的程序实现



n 考虑这样一个问题:

要对系统中的某些方法添加日志记录、性能监测、安全控制等功能, 这种需要添加功能的方法散布在很多个类中。面对这种需求, 应该怎么办呢?



AOP基本思想的演变过程-2

n 一个很自然的实现方式，可能如下：

```
class PointPayEbo implements
    PayEbi {
    public void pay() {
        //1. 记录日志开始
        //2. 时间统计开始
        //3. 安全检查

        业务逻辑

        //4. 时间统计结束
    }
}
```

```
class RMBPayEbo implements
    PayEbi {
    public void pay() {
        //1. 记录日志开始
        //2. 时间统计开始
        //3. 安全检查

        业务逻辑

        //4. 时间统计结束
    }
}
```

n 很快，大家发现里面有很多重复代码，一个自然的改进方式是：

把公共部分提出来，做成公共模块或者是公共的父类，然后让应用调用这些公共模块或者是继承公共的父类。可能的示例如下：



《深入浅出学Spring3开发》——系列精品教程

AOP基本思想的演变过程-3

```
class LogUtil{
    //记录日志
}

class TimeUtil{
    //时间统计开始
    //时间统计结束
}

class SecurityUtil{
    //安全检查
}
```

```
class PointPayEbo implements
    PayEbi {
    public void pay() {
    //1. 使用LogUtil 记录日志
    //2. 使用TimeUtil 开始时间统计
    //3. 使用SecurityUtil 进行安全检查
```

业务逻辑

```
    //4. 使用TimeUtil 结束时间统计
    }
}
```

```
class RMBPayEbo implements
    PayEbi {
    public void pay() {
    //1. 使用LogUtil 记录日志
    //2. 使用TimeUtil 开始时间统计
    //3. 使用SecurityUtil 进行安全检查
```

业务逻辑

```
    //4. 使用TimeUtil 结束时间统计
    }
}
```



AOP基本思想的演变过程-4

n 仍然存在问题:

大家会发现，需要修改的地方分散在很多个文件中，如果需要修改的文件多那么修改的量会很大，这无疑会增加出错的几率，并且加大系统维护的难度。

而且，如果添加功能的需求是在软件开发的后期才提出的话，这样大量修改已有的文件，也不符合基本的“开-闭原则”。

n 改进的解决方案

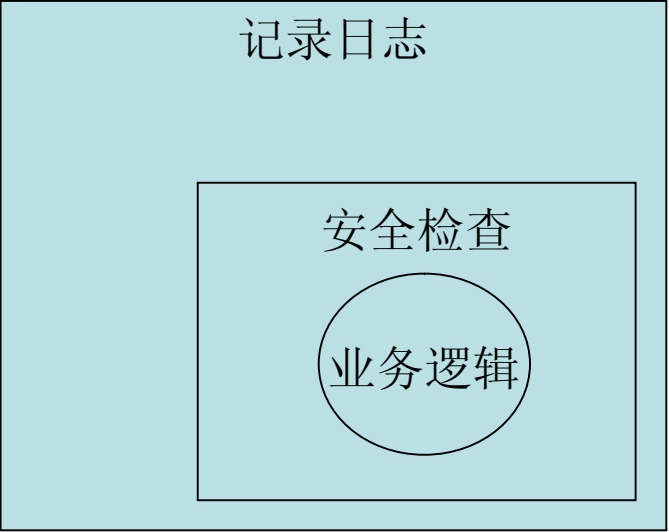
采用装饰器模式或者代理模式来实现。

n 装饰器模式定义

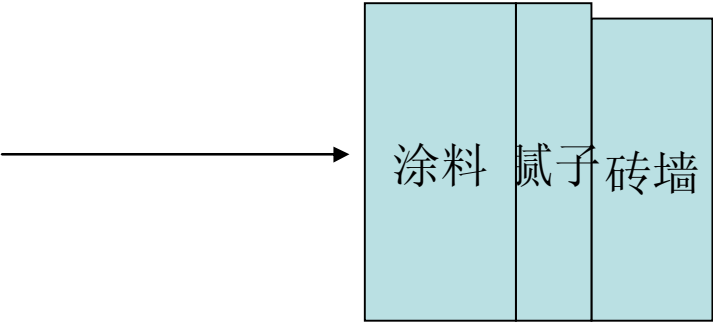
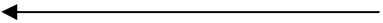
动态地给一个对象添加一些额外的职责。就增加功能来说，装饰器模式相比生成子类更为灵活。

n 代理模式定义

为其他对象提供一种代理以控制对这个对象的访问



记录日志+安全检查+业务逻辑





《深入浅出学Spring3开发》——系列精品教程

AOP基本思想的演变过程-5

nJDK动态代理解决方案（比较通用的解决方案）

```
public class MyInvocationHandler implements InvocationHandler {  
    private Object target;  
    public MyInvocationHandler(Object target) {  
        this.target = target;  
    }  
    public Object invoke(Object proxy, Method method, Object[] args)  
throws Throwable {  
        //1. 记录日志    2. 时间统计开始    3. 安全检查  
        Object retVal = method.invoke(target, args);  
        //4. 时间统计结束  
        return retVal;  
    }  
    public Object proxy() {  
        return Proxy.newProxyInstance(target.getClass().getClassLoader(),  
            target.getClass().getInterfaces(), new MyInvocationHandler(target));  
    }  
}
```



AOP基本思想的演变过程-6

n CGLIB动态代理的解决方案:

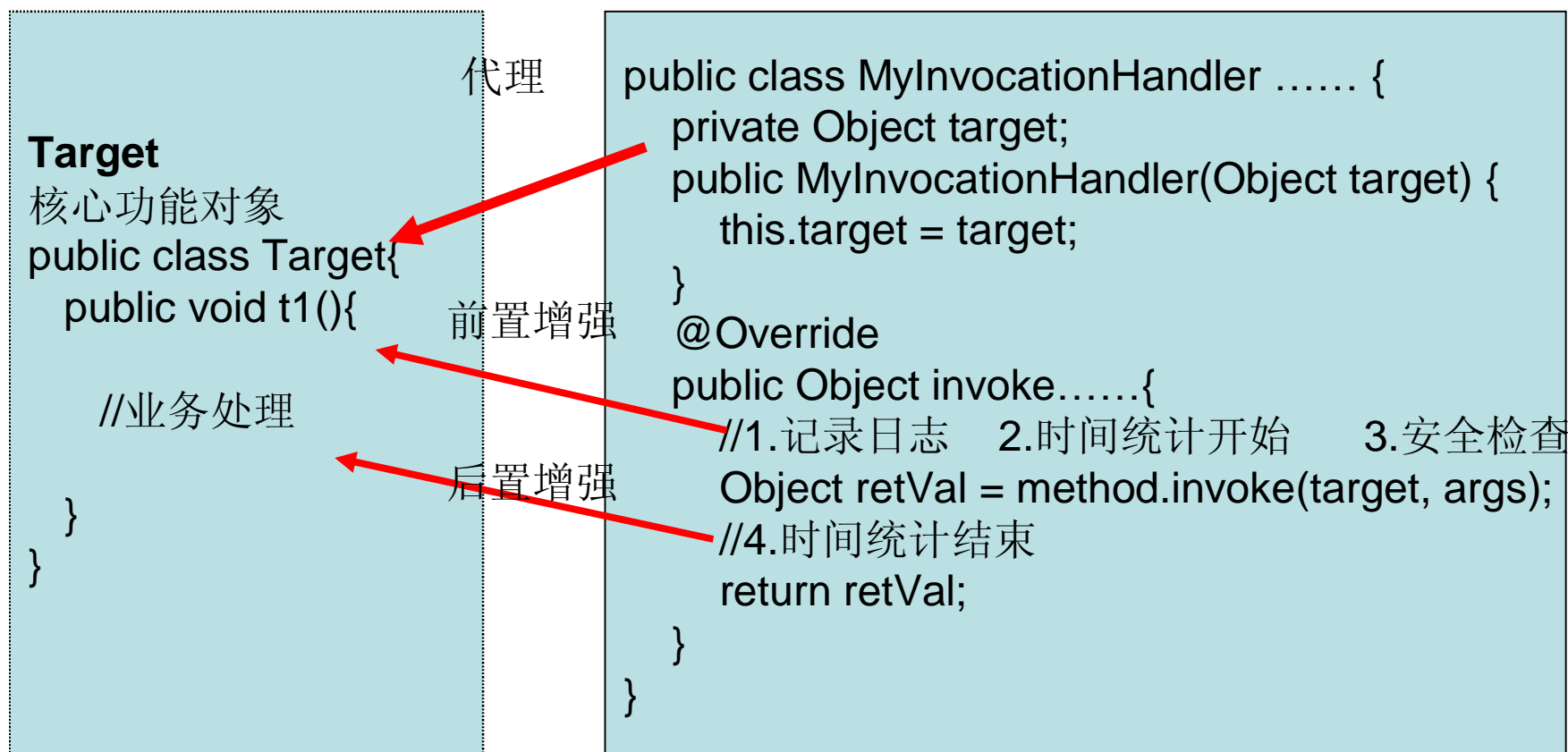
```
public class MyInterceptor implements MethodInterceptor {  
    private Object target;  
    public MyInterceptor(Object target) {  
        this.target = target;  
    }  
    public Object intercept(Object proxy, Method method, Object[] args,  
                           MethodProxy invocation) throws Throwable {  
        //1. 记录日志 2. 时间统计开始 3. 安全检查  
        Object retVal = invocation.invoke(target, args);  
        //4. 时间统计结束  
        return retVal;  
    }  
    public Object proxy() {  
        return Enhancer.create(target.getClass(), new MyInterceptor(target));  
    }  
}
```



AOP基本思想的演变过程-7

- n JDK动态代理的特点
 - 不能代理类，只能代理接口
- n CGLIB动态代理的特点
 - 能代理类和接口，不能代理final类
- n 动态代理的本质
 - 用来实现对目标对象进行增强，最终表现为类，只不过是动态创建子类，不用手工生成子类。
- n 动态代理的限制
 - 只能在父类方法被调用之前或之后进行增强（功能的修改），不能在中间进行修改，要想在方法调用中增强，需要ASM(一个Java字节码操作和分析框架)

AOP基本思想的演变过程-8





AOP基本思想的演变过程-9

n 更好的解决方案——AOP提供

人们认识到，传统的程序经常表现出一些不能自然地适合跨越多个程序模块的行为，例如日志记录、对上下文敏感的错误处理等等，人们将这种行为称为“横切关注点（CrossCuttingConcern）”，因为它跨越了给定编程模型中的典型职责界限。

如果使用过用于密切关注点的代码，您就会知道缺乏模块性所带来的问题。因为横切行为的实现是分散的，开发人员发现这种行为难以作逻辑思维、实现和更改。因此，面向方面的编程AOP应运而生。

回过头来再次理解AOP的概念。



《深入浅出学Spring3开发》——系列精品教程

本节课程小结

n AOP入门

包括：是什么、能干什么、AOP基本思想的演变

- n 作业：复习和掌握这些理论知识，尤其是AOP基本思想的演变，这能更好的理解AOP思想。



《深入浅出学Spring3开发》——系列精品教程

本节课程概览

n AOP开发

包括：AOP的基本概念，AOP的基本运行流程，还有AOP的HelloWorld。



《深入浅出学Spring3开发》——系列精品教程

第三章：AOP开发

真正高质量培训 签订就业协议

网 址：<http://www.javass.cn>
咨询QQ：460190900



AOP基本概念-1

n 关注点

就是所关注的公共功能，比如像事务管理，就是一个关注点。表示“要做什么”

n 连接点（Joinpoint）：

在程序执行过程中某个特定的点，通常在这些点需要添加关注点的功能，比如某方法调用的时候或者处理异常的时候。在Spring AOP中，一个连接点总是代表一个方法的执行。表示“在什么地方做”

n 通知（Advice）：

在切面的某个特定的连接点（Joinpoint）上执行的动作。

通知有各种类型，其中包括“around”、“before”和“after”等通知。通知的类型将在后面部分进行讨论。许多AOP框架，包括Spring，都是以拦截器做通知模型，并维护一个以连接点为中心的拦截器链。表示“具体怎么做”

n 切面/方面（Aspect）：

一个关注点的模块化，这个关注点可能会横切多个对象。综合表示“在什么地方，要做什么，以及具体如何做”



AOP基本概念-2

n 切入点（Pointcut）：

匹配连接点的断言。通知和一个切入点表达式关联，并在满足这个切入点的连接点上运行（例如，当执行某个特定名称的方法时）。切入点表达式如何和连接点匹配是AOP的核心：Spring缺省使用AspectJ切入点语法。

n 目标对象（Target Object）：

被一个或者多个切面所通知（advise）的对象。也有人把它叫做 被通知（advised）对象。既然Spring AOP是通过运行时代理实现的，这个对象永远是一个 被代理（proxied）对象。

n AOP代理（AOP Proxy）：

AOP框架使用代理模式创建的对象，从而实现在连接点处插入通知（即应用切面），就是**通过代理来对目标对象应用切面**。在Spring中，AOP代理可以用JDK动态代理或CGLIB代理实现，而通过拦截器模型应用切面。注意：Spring引入的基于模式（schema-based）风格和@AspectJ注解风格的切面声明，对于使用这些风格的用户来说，代理的创建是透明的。



AOP基本概念-3

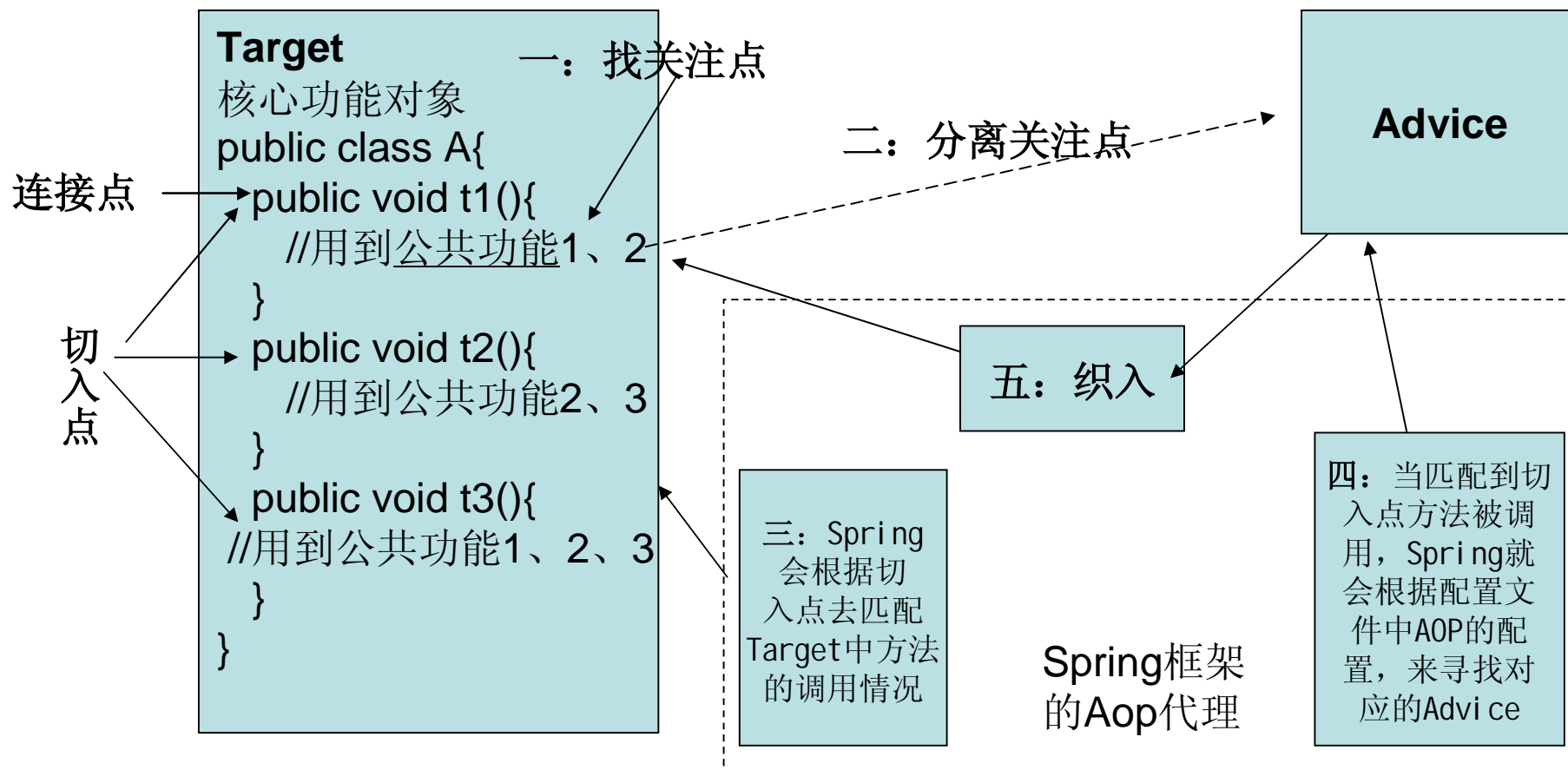
n 织入 (Weaving) :

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的过程。也就是说织入是一个过程，是将切面应用到目标对象从而创建出AOP代理对象的过程。这些可以在编译时（例如使用AspectJ编译器），类加载时和运行时完成。Spring和其他纯Java AOP框架一样，在运行时完成织入。

n 引入 (Introduction) :

也被称为内部类型声明 (inter-type declaration)。为已有的类声明额外的方法或者某个类型的字段。Spring允许引入新的接口（以及一个对应的实现）到任何被代理的对象。例如，你可以使用一个引入来使bean实现 `IsModified` 接口，以便简化缓存机制。

AOP基本运行流程





AOP的Advice类型

n 前置通知 (Before advice) :

在某连接点之前执行的通知，但这个通知不能阻止连接点前的执行（除非它抛出一个异常）。

n 返回后通知 (After returning advice) :

在某连接点正常完成后执行的通知：例如，一个方法没有抛出任何异常，正常返回。

n 抛出异常后通知 (After throwing advice) :

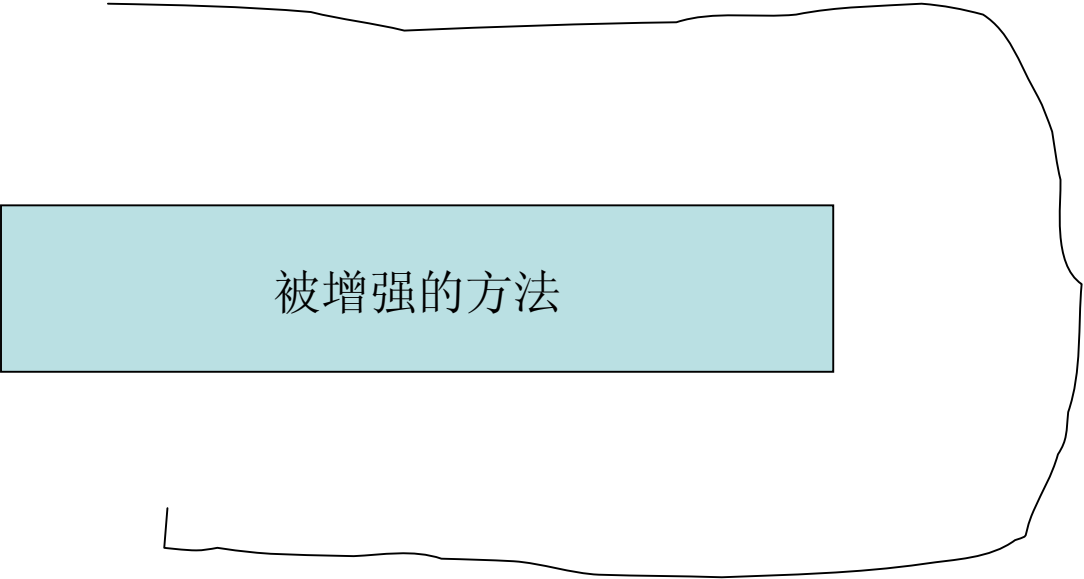
在方法抛出异常退出时执行的通知。

n 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）。

n 环绕通知 (Around Advice) :

包围一个连接点的通知，如方法调用。这是最强大的一种通知类型。环绕通知可以在方法调用前后完成自定义的行为。它也会选择是否继续执行连接点或直接返回它们自己的返回值或抛出异常来结束执行。



被增强的方法



《深入浅出学Spring3开发》——系列精品教程

HelloWorld-1

n 构建环境和前面一样

n 定义一个接口如下：

```
package cn.javass.Spring3.aop;  
public interface Api {  
    public String testAop();  
}
```

n 写一个类实现这个接口

```
public class Impl implements Api {  
    public String testAop() {  
        System.out.println("test aop");  
        return "aop is ok";  
    }  
}
```

n 写一个类作为Before的Advice，没有任何特殊要求，就是一个普通类

```
public class MyBefore {  
    public void b1(){  
        System.out.println("now befoer----->");  
    }  
}
```

真正高质量培训 签订就业协议

网 址：<http://www.javass.cn>
咨询QQ：460190900



《深入浅出学Spring3开发》——系列精品教程

HelloWorld-2

n 配置文件，要注意配置的时候要保证一定要有命名空间aop的定义，如下：

```
<?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/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-3.0.xsd"
">
```



《深入浅出学Spring3开发》——系列精品教程

HelloWorld-3

```
<bean id="testAopApi" class="cn.javass.spring3.aop.Impl"></bean>
<bean id="myBefore" class="cn.javass.spring3.aop.MyBefore"></bean>

<aop:config>
  <aop:aspect id="abcd" ref="myBefore">
    <aop:pointcut id="myPointcut"
      expression="execution(* cn.javass.spring3.aop.*(..))"/>
    <aop:before
      pointcut-ref="myPointcut"
      method="b1"/>
  </aop:aspect>
</aop:config>
</beans>
```



《深入浅出学Spring3开发》——系列精品教程

HelloWorld-4

n 客户端如下:

```
public class Client {  
    public static void main(String[] args) {  
        ApplicationContext context = new ClassPathXmlApplicationContext(  
            new String[] {"applicationContext.xml"});  
        Api api = (Api)context.getBean("testAopApi");  
        String s = api.testAop();  
        System.out.println("s==" + s);  
    }  
}
```

n 测试结果如下:

```
now befoer----->  
test aop  
s==aop is ok
```



《深入浅出学Spring3开发》——系列精品教程

本节课程小结

n AOP开发

包括：AOP的基本概念，AOP的基本运行流程，还有AOP的HelloWorld。

n 作业：

- 1：复习和掌握这些理论知识，尤其是AOP的基本运行流程
- 2：动手去实现AOP的HelloWorld，体会一下AOP的开发和运行流程。



《深入浅出学Spring3开发》——系列精品教程

本节课程概览

n AOP开发

包括：@AspectJ支持

真正高质量培训 签订就业协议

网 址：<http://www.javass.cn>
咨询QQ：460190900



《深入浅出学Spring3开发》——系列精品教程

第三章：AOP开发

真正高质量培训 签订就业协议

网 址：<http://www.javass.cn>
咨询QQ：460190900



《深入浅出学Spring3开发》——系列精品教程

@AspectJ支持-1

n 启用@AspectJ支持

通过在你的Spring的配置中引入下列元素来启用Spring对@AspectJ的支持:

```
<aop: aspectj-autoproxy/>
```

n 声明一个方面

在application context中定义的任意带有一个@Aspect切面（拥有@Aspect注解）的bean都将被Spring自动识别并用于配置在Spring AOP。

配置如: `<bean id="myAspect" class="org.xyz.NotVeryUsefulAspect">`

```
<!-- configure properties of aspect here as normal -->
```

```
</bean>
```

Java类:

@Aspect

```
public class NotVeryUsefulAspect {  
}
```



@AspectJ支持-2

n 声明一个切入点 (pointcut)

使用 @Pointcut 注解来表示，示例如下：

```
@Pointcut("execution(* transfer(..))")  
private void anyOldTransfer() {}
```

n 切入点指定者的支持

Spring AOP 支持在切入点表达式中使用如下的AspectJ切入点指定者：

- 1: **execution**: 匹配方法执行的连接点，这是你将会用到的Spring的最主要的切入点指定者。
- 2: **within**: 限定匹配特定类型的连接点（在使用Spring AOP的时候，在匹配的类型中定义的方法的执行）。
- 3: **this**: 限定匹配特定的连接点（使用Spring AOP的时候方法的执行），其中bean reference（Spring AOP 代理）是指定类型的实例。
- 4: **target**: 限定匹配特定的连接点（使用Spring AOP的时候方法的执行），其中目标对象（被代理的application object）是指定类型的实例。



@AspectJ支持-3

- 5: args: 限定匹配特定的连接点（使用Spring AOP的时候方法的执行），其中参数是指定类型的实例。
- 6: @target: 限定匹配特定的连接点（使用Spring AOP的时候方法的执行），其中执行的对象的类已经有指定类型的注解。
- 7: @args: 限定匹配特定的连接点（使用Spring AOP的时候方法的执行），其中实际传入参数的运行时类型有指定类型的注解。
- 8: @within: 限定匹配特定的连接点，其中连接点所在类型已指定注解（在使用Spring AOP的时候，所执行的方法所在类型已指定注解）。
- 9: @annotation: 限定匹配特定的连接点（使用Spring AOP的时候方法的执行），其中连接点的主题有某种给定的注解

n 合并切入点表达式

切入点表达式可以使用 '&&', '||' 和 '!' 来合并. 还可以通过名字来指向切入点表达式。



@AspectJ支持-4

n 切入点表达式的基本语法

Spring AOP 用户可能会经常使用 execution pointcut designator。执行表达式的格式如下：

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?  
name-pattern(param-pattern) throws-pattern?)
```

除了返回类型模式（上面代码片断中的ret-type-pattern），名字模式和参数模式以外，所有的部分都是可选的。返回类型模式决定了方法的返回类型必须依次匹配一个连接点。

n 类型匹配模式

- 1: *: 匹配任何数量字符；比如模式 (*,String) 匹配了一个接受两个参数的方法，第一个可以是任意类型，第二个则必须是String类型
- 2: ..: 匹配任何数量字符的重复，如在类型模式中匹配任何数量包；而在方法参数模式中匹配任何数量参数，可以使零到多个。
- 3: +: 匹配指定类型的子类型；仅能作为后缀放在类型模式后边。



《深入浅出学Spring3开发》——系列精品教程

@AspectJ支持-5

n 类型匹配模式示例

- 1: `java.lang.String` 匹配String类型;
- 2: `java.*.String` 匹配java包下的任何“一级子包”下的String类型;
如匹配`java.lang.String`, 但不匹配`java.lang.ss.String`
- 3: `java..*` 匹配java包及任何子包下的任何类型;
如匹配`java.lang.String`、`java.lang.annotation.Annotation`
- 4: `java.lang.*ing` 匹配任何java.lang包下的以ing结尾的类型;
- 5: `java.lang.Number+` 匹配java.lang包下的任何Number的子类型;
如匹配`java.lang.Integer`, 也匹配`java.math.BigInteger`

n 切入点表达式的基本示例, 使用`execution`

- 1: `public **(..)`
任何公共方法的执行
- 2: `* cn.javass..IPointcutService.*()`
`cn.javass`包及所有子包下`IPointcutService`接口中的任何无参方法
- 3: `* cn.javass..*.*(..)`
`cn.javass`包及所有子包下任何类的任何方法



@AspectJ支持-6

4: * cn.javass..IPointcutService.*(*)

cn.javass包及所有子包下IPointcutService接口的任何只有一个参数方法

5: * (!cn.javass..IPointcutService+).*(..)

非“cn.javass包及所有子包下IPointcutService接口及子类型”的任何方法

6: * cn.javass..IPointcutService+. *()

cn.javass包及所有子包下IPointcutService接口及子类型的的任何无参方法

7: * cn.javass..IPointcut*.test*(java.util.Date)

cn.javass包及所有子包下IPointcut前缀类型的的以test开头的只有一个参数类型为java.util.Date的方法，注意该匹配是根据方法签名的参数类型进行匹配的，而不是根据执行时传入的参数类型决定的如定义方法：public void test(Object obj);即使执行时传入java.util.Date，也不会匹配的。

8: * cn.javass..IPointcut*.test*(..) throws IllegalArgumentException, ArrayIndexOutOfBoundsException

cn.javass包及所有子包下IPointcut前缀类型的的任何方法，且抛出IllegalArgumentException和ArrayIndexOutOfBoundsException异常



《深入浅出学Spring3开发》——系列精品教程

@AspectJ支持-7

9: * (cn.javass..IPointcutService+ && java.io.Serializable+).*(..)

任何实现了cn.javass包及所有子包下IPointcutService接口和
java.io.Serializable接口的类型的任何方法

10: @java.lang.Deprecated * *(..)

任何持有@java.lang.Deprecated注解的方法

11: @java.lang.Deprecated @cn.javass..Secure * *(..)

任何持有@java.lang.Deprecated和@cn.javass..Secure注解的方法

12: @(java.lang.Deprecated || cn.javass..Secure) * *(..)

任何持有@java.lang.Deprecated或@cn.javass..Secure注解的方法

13: (@cn.javass..Secure *) *(..)

任何返回值类型持有@cn.javass..Secure的方法

14: * (@cn.javass..Secure *).*(..)

任何定义方法的类型持有@cn.javass..Secure的方法

15: * *(@cn.javass..Secure (*), @cn.javass..Secure (*))

任何签名带有两个参数的方法，且这个两个参数都被@Secure标记了，如public void
test(@Secure String str1, @Secure String str1);



@AspectJ支持-8

16: `* *((@ cn.javass..Secure *))或* *(@ cn.javass..Secure *)`

任何带有一个参数的方法，且该参数类型持有@ cn.javass..Secure；如public void test(Model model); 且Model类上持有@Secure注解

17: `* *(@cn.javass..Secure (@cn.javass..Secure *) , @ cn.javass..Secure (@cn.javass..Secure *))`

任何带有两个参数的方法，且这两个参数都被@ cn.javass..Secure标记了；且这两个参数的类型上都持有@ cn.javass..Secure；

18: `* *(java.util.Map<cn.javass..Model, cn.javass..Model>, ...)`

任何带有一个java.util.Map参数的方法，且该参数类型是以<cn.javass..Model, cn.javass..Model>为泛型参数；注意只匹配第一个参数为java.util.Map, 不包括子类型；如public void test(HashMap<Model, Model> map, String str); 将不匹配，必须使用“`* *(java.util.HashMap<cn.javass..Model, cn.javass..Model>, ...)`”进行匹配；而public void test(Map map, int i); 也将不匹配，因为泛型参数不匹配

19: `* *(java.util.Collection<@cn.javass..Secure *>)`

任何带有一个参数（类型为java.util.Collection）的方法，且该参数类型是有一个泛型参数，该泛型参数类型上持有@cn.javass..Secure注解；如public void test(Collection<Model> collection); Model类型上持有@cn.javass..Secure



@AspectJ支持-9

n 切入点表达式的基本示例，使用`within`匹配指定类型内的方法

1: `within(cn.javass..*)`

cn.javass包及子包下的任何方法执行

2: `within(cn.javass..IPointcutService+)`

cn.javass包或所有子包下IPointcutService类型及子类型的任何方法

3: `within(@cn.javass..Secure *)`

持有cn.javass..Secure注解的任何类型的任何方法必须是在目标对象上声明这个注解，在接口上声明的对它不起作用

n 切入点表达式的基本示例，使用`this`

使用“`this(类型全限定名)`”匹配当前AOP代理对象类型的执行方法；注意是AOP代理对象的类型匹配，这样就可能包括引入接口方法也可以匹配；注意`this`中使用的表达式必须是类型全限定名，不支持通配符

1: `this(cn.javass.spring.chapter6.service.IPointcutService)`

当前AOP对象实现了 IPointcutService接口的任何方法

2: `this(cn.javass.spring.chapter6.service.IIntroductionService)`

当前AOP对象实现了 IIntroductionService接口的任何方法也可能是引入接口



@AspectJ支持-10

n 切入点表达式的基本示例，使用target

使用“target(类型全限定名)”匹配当前目标对象类型的执行方法；注意是目标对象的类型匹配，这样就不包括引入接口也类型匹配；注意target中使用的表达式必须是类型全限定名，不支持通配符

1: target(cn.javass.spring.chapter6.service.IPointcutService)

当前目标对象（非AOP对象）实现了 IPointcutService接口的任何方法

2: target(cn.javass.spring.chapter6.service.IIntroductionService)

当前目标对象（非AOP对象） 实现了IIntroductionService 接口的任何方法不可能是引入接口

n 切入点表达式的基本示例，使用args

使用“args(参数类型列表)”匹配当前执行的方法传入的参数为指定类型的执行方法；注意是匹配传入的参数类型，不是匹配方法签名的参数类型；参数类型列表中的参数必须是类型全限定名，通配符不支持；args属于动态切入点，这种切入点开销非常大，非特殊情况最好不要使用

1: args (java.io.Serializable,...)

任何一个以接受“传入参数类型为 java.io.Serializable” 开头，且其后可跟任意个任意类型的参数的方法执行，args指定的参数类型是在运行时动态匹配的



@AspectJ支持-11

n 切入点表达式的基本示例，使用@within

使用“@within(注解类型)”匹配所以持有指定注解类型内的方法；注解类型也必须是全限定类型名

1: @within (cn.javass.spring.chapter6. Secure)

任何目标对象对应的类型持有Secure注解的类方法；必须是在目标对象上声明这个注解，在接口上声明的对它不起作用

n 切入点表达式的基本示例，使用@target

使用“@target(注解类型)”匹配当前目标对象类型的执行方法，其中目标对象持有指定的注解；注解类型也必须是全限定类型名

1: @target (cn.javass.spring.chapter6. Secure)

任何目标对象持有Secure注解的类方法；必须是在目标对象上声明这个注解，在接口上声明的对它不起作用

n 切入点表达式的基本示例，使用@args

使用“@args(注解列表)”匹配当前执行的方法传入的参数持有指定注解的执行；注解类型也必须是全限定类型名

1: @args (cn.javass.spring.chapter6. Secure)

任何一个只接受一个参数的方法，且方法运行时传入的参数持有注解cn.javass.spring.chapter6. Secure；动态切入点，类似于arg指示符；



@AspectJ支持-12

n 切入点表达式的基本示例，使用@annotation

使用“@annotation(注解类型)”匹配当前执行方法持有指定注解的方法；注解类型也必须是全限定类型名

1: @annotation(cn.javass.spring.chapter6.Secure)

当前执行方法上持有注解 cn.javass.spring.chapter6.Secure将被匹配

n 切入点表达式的基本示例，使用bean

使用“bean(Bea n id或名字通配符)”匹配特定名称的Bean对象的执行方法；Spring AOP扩展的，在AspectJ中无相应概念

1: bean(*Service)

匹配所有以Service命名（id或name）结尾的Bean

n 切入点表达式的基本示例，使用reference pointcut

引用其他命名切入点，只有@AspectJ风格支持，Schema风格不支持，如下所示：

```
@Pointcut(value="bean(*Service)") //命名切入点1
```

```
private void pointcut1(){} 
```

```
@Pointcut(value="@args(cn.javass.spring.chapter6.Secure)") //命名切入点2
```

```
private void pointcut2(){} 
```

```
@Before(value = "pointcut1() && pointcut2()") //引用命名切入点
```

```
public void referencePointcutTest1(JoinPoint jp) {
```

```
    dump("pointcut1() && pointcut2()", jp);
```

```
}
```



《深入浅出学Spring3开发》——系列精品教程

本节课程小结

- n AOP开发
包括：@AspectJ支持
- n 作业：复习和练习这些知识



《深入浅出学Spring3开发》——系列精品教程

本节课程概览

n AOP开发

包括：@AspectJ支持的第二部分

真正高质量培训 签订就业协议

网 址：<http://www.javass.cn>
咨询QQ：460190900



《深入浅出学Spring3开发》——系列精品教程

第三章：AOP开发

真正高质量培训 签订就业协议

网 址：<http://www.javass.cn>
咨询QQ：460190900



@AspectJ支持-13

n 声明通知

通知是跟一个切入点表达式关联起来的，并且在切入点匹配的方法执行之前或者之后或者之前和之后运行。切入点表达式可能是指向已命名的切入点的简单引用或者是一个已经声明过的切入点表达式。

n 前置通知（Before advice），使用 @Before 注解声明

@Aspect

```
public class BeforeExample {  
    @Before("execution(* com.xyz.myapp.dao.*(..))")  
    public void doAccessCheck() {  
        // ...  
    }  
}
```

n 返回后通知（After returning advice），使用 @AfterReturning注解声明

n 后通知（After (finally) advice），使用 @After注解声明



《深入浅出学Spring3开发》——系列精品教程

@AspectJ支持-14

n 抛出后通知 (After throwing advice) , 使用 @AfterThrowing注解声明
可以将抛出的异常绑定到通知的一个参数上 , 如下:

```
@AfterThrowing(  
    pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()",  
    throwing="ex")  
public void doRecoveryActions(Exception ex) {  
    // ...  
}
```

n 环绕通知 (Around Advice) , 使用 @Around注解声明

```
@Around("com.xyz.myapp.SystemArchitecture.businessService()")  
public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable  
{  
    // start stopwatch  
    Object retVal = pjp.proceed();  
    // stop stopwatch  
    return retVal;  
}
```



@AspectJ支持-15

n 给Advice传递参数

通常情况下，Advice都需要获取一定的参数，比如：Before需要截获传入的参数，而After需要获取方法的返回值等等。下面介绍两种方式

n 方式一：使用JoinPoint来给Advice的方法传递参数

Spring AOP提供使用org.aspectj.lang.JoinPoint类型获取连接点数据，任何通知方法的第一个参数都可以是JoinPoint(环绕通知是ProceedingJoinPoint, JoinPoint子类)，当然第一个参数位置也可以是JoinPoint.StaticPart类型，这个只返回连接点的静态部分。

1: JoinPoint: 提供访问当前被通知方法的目标对象、代理对象、方法参数等数据

```
public interface JoinPoint {  
    String toString();           //连接点所在位置的相关信息  
    String toShortString();      //连接点所在位置的简短相关信息  
    String toLongString();       //连接点所在位置的全部相关信息  
    Object getThis();            //返回AOP代理对象  
    Object getTarget();          //返回目标对象  
    Object[] getArgs();          //返回被通知方法参数列表  
    Signature getSignature();    //返回当前连接点签名  
    SourceLocation getSourceLocation(); //返回连接点方法所在类文件中的位置  
    String getKind();            //连接点类型  
    StaticPart getStaticPart();  //返回连接点静态部分  
}
```



《深入浅出学Spring3开发》——系列精品教程

@AspectJ支持-16

2: ProceedingJoinPoint: 用于环绕通知, 使用proceed()方法来执行目标方法

```
public interface ProceedingJoinPoint extends JoinPoint {  
    public Object proceed() throws Throwable;  
    public Object proceed(Object[] args) throws Throwable;  
}
```

3: JoinPoint.StaticPart: 提供访问连接点的静态部分, 如被通知方法签名、连接点类型等

```
public interface StaticPart {  
    Signature getSignature();    //返回当前连接点签名  
    String getKind();            //连接点类型  
    int getId();                 //唯一标识  
    String toString();           //连接点所在位置的相关信息  
    String toShortString();      //连接点所在位置的简短相关信息  
    String toLongString();       //连接点所在位置的全部相关信息  
}
```

4: 示例: 使用如下方式在通知方法上声明, 必须是在第一个参数, 然后使用jp.getArgs()就能获取到被通知方法参数:

```
@Before(value="execution(* sayBefore(*))")  
public void before(JoinPoint jp) {}  
@Before(value="execution(* sayBefore(*))")  
public void before(JoinPoint.StaticPart jp) {}
```



@AspectJ支持-17

n 方式二：使用args来给Advice的方法传递参数，示例如：

```
@Before(value="execution(* test(*)) && args(param)", argNames="param")
public void before1(String param) {
    System.out.println("===param: " + param);
}
```

切入点表达式`execution(* test(*)) && args(param)`：

- (1) 首先`execution(* test(*))`匹配任何方法名为`test`，且有一个任何类型的参数；
- (2) `args(param)`将首先查找通知方法上同名的参数，并在方法执行时（运行时）匹配传入的参数是使用该同名参数类型，即`java.lang.String`；如果匹配将把该被通知参数传递给通知方法上同名参数。

n 参数匹配的策略

1：可以通过“`argNames`”属性指定参数名，示例如：

```
@Before(value=" args(param)", argNames="param") //明确指定了
public void before1(String param) {
    System.out.println("===param: " + param);
}
```

2：如果第一个参数类型是`JoinPoint`、`ProceedingJoinPoint`或`JoinPoint.StaticPart`类型，应该从“`argNames`”属性省略掉该参数名（可选，写上也对），这些类型对象会自动传入的，但必须作为第一个参数。示例如：



《深入浅出学Spring3开发》——系列精品教程

@AspectJ支持-18

```
@Before(value=" args(param)", argNames="param") //明确指定了  
public void before1(JoinPoint jp, String param) {  
    System.out.println("===param: " + param);  
}
```

3: 如果“class文件中含有变量调试信息”，Spring将可以使用这些方法签名中的参数名来确定参数名，示例如下：

```
@Before(value=" args(param)") //不需要argNames了  
public void before1(JoinPoint jp, String param) {  
    System.out.println("===param: " + param);  
}
```

4: 如果没有“class文件中含有变量调试信息”，将尝试自己的参数匹配算法，如果发现参数绑定有二义性将抛出AmbiguousBindingException异常；对于只有一个绑定变量的切入点表达式，而通知方法只接受一个参数，说明绑定参数是明确的，从而能配对成功，示例如下：

```
@Before(value=" args(param)")  
public void before1(JoinPoint jp, String param) {  
    System.out.println("===param: " + param);  
}
```

5: 以上策略失败将抛出IllegalArgumentException。



《深入浅出学Spring3开发》——系列精品教程

@AspectJ支持-19

n 处理参数

```
@Around("execution(List<Account> find*(..)) &&" +  
        "com.xyz.SystemArchitecture.inDataAccessLayer() &&" +  
        "args(accountHolderNamePattern)")  
public Object preProcessQueryPattern(ProceedingJoinPoint pjp, String  
        accountHolderNamePattern)  
throws Throwable {  
    String newPattern = preProcess(accountHolderNamePattern);  
    return pjp.proceed(new Object[] {newPattern});  
}
```

n 获取方法运行后的返回值

```
@AfterReturning(  
    pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()",  
    returning="retVal")  
public void doAccessCheck(Object retVal) {  
    // ...  
}
```




Advice的执行顺序-1

n Advice执行的顺序

如果有多个通知想要在同一连接点运行会发生什么？Spring AOP 的执行通知的顺序跟AspectJ的一样。在“进入”连接点的情况下，最高优先级的通知会先执行（所以上面给出的两个前置通知（before advice）中，优先级高的那个会先执行）。在“退出”连接点的情况下，最高优先级的通知会最后执行。只需要记住通知是按照定义的顺序来执行的就可以了。

当定义在 **不同的** 切面里的两个通知都需要在一个相同的连接点中运行，那么除非你指定，否则执行的顺序是未知的。你可以通过指定优先级来控制执行顺序。在Spring中可以在切面类中实现 `org.springframework.core.Ordered` 接口做到这一点。在两个切面中，`Ordered.getValue()` 方法返回值较低的那个有更高的优先级。



《深入浅出学Spring3开发》——系列精品教程

Advice的执行顺序-2

n 默认无序

n 指定顺序

1: 同一切面中通知执行顺序:

- 1、前置通知/环绕通知 (proceed方法执行之前) --- 执行顺序不确定
- 2、被通知方法
- 3、后置通知/环绕通知 (proceed之后) --- 执行顺序不确定

2: 不同切面中的通知执行顺序

Aspect1 (order:1)

- 1、前置通知/环绕通知 (proceed方法执行之前)
- 2、被通知方法
- 3、后置通知/环绕通知 (proceed之后)

Aspect2 (order:2)

- 1、前置通知/环绕通知 (proceed方法执行之前)
- 2、被通知方法
- 3、后置通知/环绕通知 (proceed之后)

最终执行顺序 (order较小值拥有较高优先级)

- 1、Aspect1.前置通知/环绕通知 (proceed方法执行之前)
- 2、Aspect2.前置通知/环绕通知 (proceed方法执行之前)
- 3、被通知方法
- 4、Aspect2.后置通知/环绕通知 (proceed之后)
- 5、Aspect1.后置通知/环绕通知 (proceed之后)

高优先级

低优先级

低优先级

高优先级

真正高质量培训 签订就业协议

网 址: <http://www.javass.cn>
咨询QQ: 460190900



《深入浅出学Spring3开发》——系列精品教程

本节课程小结

n AOP开发

包括：@AspectJ支持的第二部分

n 作业：

- 1: 复习和练习这些知识，理解和掌握Advice的执行顺序
- 2: 从前面项目中挑选一个模块，使用AOP实现统计Ebo中的每个方法每次运行的时间，并记录日志



《深入浅出学Spring3开发》——系列精品教程

本节课程概览

n AOP开发

包括：Schema风格的AOP支持、AOP声明风格的选择、AOP的代理机制、Spring1.x中的AOP的APIs以及基本使用、AOP开发和设计考虑

真正高质量培训 签订就业协议

网 址：<http://www.javass.cn>
咨询QQ：460190900



《深入浅出学Spring3开发》——系列精品教程

第三章：AOP开发

真正高质量培训 签订就业协议

网 址：<http://www.javass.cn>
咨询QQ：460190900



Schema风格的AOP支持-1

n 声明一个切面 : `<aop: aspect>`

```
<aop: config>
```

```
  <aop: aspect id="myAspect" ref="adviceBean">
```

```
    ...
```

```
  </aop: aspect>
```

```
</aop: config>
```

```
<bean id="adviceBean" class="...">
```

```
  ...
```

```
</bean>
```

n 声明一个切入点 : `<aop: pointcut>`

```
<aop: config>
```

```
  <aop: pointcut id="businessService"
```

```
    expression="execution(* com.xyz.myapp.service.*(..))"/>
```

```
</aop: config>
```

在切面里面声明一个切入点和声明一个顶级的切入点非常类似:



Schema风格的AOP支持-2

```
<aop: config>
  <aop: aspect id="myAspect" ref="aBean">
    <aop: pointcut id="businessService"
expression="execution(* com.xyz.myapp.service.*(..))"/>
    ...
  </aop: aspect>
</aop: config>
```

当需要连接子表达式的时候, '&' 在XML中用起来非常不方便, 所以关键字 'and', 'or' 和 'not' 可以分别用来代替 '&', '||' 和 '!'。

n 声明通知Advice, Before Advice

```
<aop: aspect id="beforeExample" ref="aBean">
  <aop: before
    pointcut="execution(* com.xyz.myapp.dao.*(..))"
    method="doAccessCheck"/>
  ...
</aop: aspect>
```



Schema风格的AOP支持-3

n 返回后通知 (After returning advice)

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

n 抛出异常后通知 (After throwing advice)

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



Schema风格的AOP支持-4

n 后通知 (After (finally) advice)

```
<aop: aspect id="afterFinallyExample" ref="aBean">
  <aop: after
    pointcut-ref="dataAccessOperation"
    method="doReleaseLock"/>
  ...
</aop: aspect>
```

n Around通知

```
<aop: aspect id="aroundExample" ref="aBean">
  <aop: around
    pointcut-ref="businessService"
    method="doBasicProfiling"/>
  ...
</aop: aspect>
```



Schema风格的AOP支持-5

n 通知的参数，可以通过 `arg-names` 属性来实现

```
<aop:before  
  pointcut="com.xyz.lib.Pointcuts.anyPublicMethod()"  
  method="audit"  
  arg-names="auditable"/>
```

n `Advisor`

“advisors”这个概念来自Spring1.2对AOP的支持，在AspectJ中是没有等价的概念。`advisor`就像一个小的自包含的切面，这个切面只有一个通知。

```
<aop:config>  
  <aop:pointcut id="businessService"  
    expression="execution(* com.xyz.myapp.service.*(..))"/>  
  <aop:advisor  
    pointcut-ref="businessService"  
    advice-ref="tx-advice"/>  
</aop:config>
```




AOP声明风格的选择

n Spring AOP还是完全用AspectJ?

优先选用Spring的AOP，因为它不是一个人在战斗，Spring除了AOP，还有IOC，还有事务等其他功能。

如果你需要通知domain对象或其它没有在Spring容器中管理的任意对象，那么你需要使用AspectJ。

n Spring AOP中使用@AspectJ还是XML?

优先使用@AspectJ。

XML风格有两个缺点。第一是它不能完全将需求实现的地方封装到一个位置。DRY原则中说系统中的每一项知识都必须具有单一、无歧义、权威的表示。当使用XML风格时，如何实现一个需求的知识被分割到支撑类的声明中以及XML配置文件中。当使用@AspectJ风格时就只有一个单独的模块 -切面- 信息被封装了起来。第二是XML风格同@AspectJ风格所能表达的内容相比有更多的限制：仅仅支持"singleton"切面实例模型，并且不能在XML中组合命名连接点的声明。



AOP的代理机制

- n Spring AOP使用JDK动态代理或者CGLIB来为目标对象创建代理。如果被代理的目标对象实现了至少一个接口，则会使用JDK动态代理。所有该目标类型实现的接口都将被代理。若该目标对象没有实现任何接口，则创建一个CGLIB代理。
- n 如果你希望强制使用CGLIB代理，（例如：希望代理目标对象的所有方法，而不只是实现自接口的方法）那也可以。但是需要考虑以下问题：
 - 1: 无法通知（advise）Final 方法，因为他们不能被覆写。
 - 2: 将CGLIB 2二进制发行包放在classpath下面，JDK本身就提供了动态代理
 - 3: 强制使用CGLIB代理需要将 `<aop:config>` 的 `proxy-target-class` 属性设为 true:
`<aop:config proxy-target-class="true"></aop:config>`
 - 4: 当需要使用CGLIB代理和@AspectJ自动代理支持，请按照如下的方式设置
`<aop:aspectj-autoproxy>` 的 `proxy-target-class` 属性:
`<aop:aspectj-autoproxy proxy-target-class="true"/>`



AOP的API -1

n 切入点的API

`org.springframework.aop.Pointcut` 是最核心的接口，用来将 通知应用于特定的类和方法，完整的接口定义如下：

```
public interface Pointcut {  
    ClassFilter getClassFilter();  
    MethodMatcher getMethodMatcher();  
}
```

- 1: 将`Pointcut`接口分割成有利于重用类和方法匹配的两部分，以及进行更细粒度的操作组合（例如与另一个方法匹配实现进行“或操作”）。
- 2: `ClassFilter`接口用来将切入点限定在一个给定的类集合中。如果`matches()`方法总是返回`true`，所有目标类都将被匹配：

```
public interface ClassFilter {  
    boolean matches(Class clazz);  
}
```



AOP的API -2

3: MethodMatcher接口通常更重要，完整的接口定义如下：

```
public interface MethodMatcher {  
    boolean matches(Method m, Class targetClass);  
    boolean isRuntime();  
    boolean matches(Method m, Class targetClass, Object[] args);  
}
```

matches(Method, Class)方法用来测试这个切入点是否匹配目标类的指定方法。这将在AOP代理被创建的时候执行，这样可以避免在每次方法调用的时候都执行。如果matches(Method, Class)对于一个给定的方法返回true，并且isRuntime()也返回true，那么matches(Method, Class , Object[])将在每个方法调用的时候被调用。这使得切入点在通知将被执行前可以查看传入到方法的参数。

大多数MethodMatcher是静态的，这意味着isRuntime()方法返回 false。在这种情况下，matches(Method, Class , Object[])永远不会被调用。



AOP的API -3

n 静态切入点

静态切入点基于方法和目标类进行切入点判断而不考虑方法的参数。在多数情况下，静态切入点是高效的、最好的选择。Spring只在第一次调用方法时执行静态切入点：以后每次调用这个方法时就不需要再执行。

1: 正则表达式切入点

正则表达式是最常用的描述静态切入点的方式，多数AOP框架都支持这种方式。org.springframework.aop.support.Perl5RegexMethodPointcut是一个最基本的正则表达式切入点，它使用Perl 5正则表达式语法。示例如下：

```
<bean id="settersAndAbsquatulatePointcut"
      class="org.springframework.aop.support.Perl5RegexMethodPointcut">
  <property name="patterns">
    <list>
      <value>.*set.*</value>
      <value>.*absquatulate</value>
    </list>
  </property>
</bean>
```



AOP中Advice的API - 1

n 拦截around通知

Spring里使用方法拦截的around通知兼容AOP联盟接口。实现around通知的MethodInterceptor应当实现下面的接口：

```
public interface MethodInterceptor extends Interceptor {  
    Object invoke(MethodInvocation invocation) throws Throwable;  
}
```

提示：在invoke方法里面不要忘记调用：invocation.proceed();

n 前置通知

```
public interface MethodBeforeAdvice extends BeforeAdvice {  
    void before(Method m, Object[] args, Object target) throws Throwable;  
}
```

n 后置通知

```
public interface AfterReturningAdvice extends Advice {  
    void afterReturning(Object returnValue, Method m, Object[] args,  
        Object target) throws Throwable;  
}
```




AOP中Advice的API -2

n 异常通知

如果连接点抛出异常，异常通知（throws advice）将在连接点返回后被调用。

Spring提供类型检查的异常通知，这意味着org.springframework.aop.ThrowsAdvice接口不包含任何方法：它只是一个标记接口用来标识 所给对象实现了一个或者多个针对特定类型的异常通知方法。这些方法应当满足下面的格式

afterThrowing([Method], [args], [target], subclassOfThrowable)

只有最后一个参数是必须的。因此异常通知方法对方法及参数的需求，方法的签名将从一到四个参数之间变化。下面是一些throws通知的例子。

当一个RemoteException（包括它的子类）被抛出时，下面的通知会被调用：

```
public class RemoteThrowsAdvice implements ThrowsAdvice {  
    public void afterThrowing(RemoteException ex) throws Throwable {  
        // Do something with remote exception  
    }  
}
```

n 引入通知

```
public interface IntroductionInterceptor extends MethodInterceptor {  
    boolean implementsInterface(Class intf);  
}
```



ProxyFactoryBean-1

n 使用ProxyFactoryBean创建AOP代理

使用ProxyFactoryBean或者其它IoC相关类带来的最重要的好处之一就是创建AOP代理，这意味着通知和切入点也可以由IoC来管理。这是一个强大的功能并使得某些特定的解决方案成为可能

下面描述ProxyFactoryBean的属性：

- 1: proxyTargetClass: 这个属性为true时，目标类本身被代理而不是目标类的接口。如果这个属性值被设为true，CGLIB代理将被创建
- 2: optimize: 用来控制通过CGLIB创建的代理是否使用激进的优化策略。除非完全了解AOP代理如何处理优化，否则不推荐用户使用这个设置。目前这个属性仅用于CGLIB代理；对于JDK动态代理（缺省代理）无效。
- 3: frozen: 用来控制代理工厂被配置之后，是否还允许修改通知。缺省值为false（即在代理被配置之后，不允许修改代理的配置）。
- 4: exposeProxy: 决定当前代理是否被保存在一个ThreadLocal 中以便被目标对象访问。（目标对象本身可以通过MethodInvocation来访问，因此不需要ThreadLocal。） 如果个目标对象需要获取代理而且exposeProxy属性被设为true，目标对象可以使用AopContext.currentProxy()方法。



ProxyFactoryBean-2

- 5: **aopProxyFactory**: 使用AopProxyFactory的实现。这提供了一种方法来自定义是否使用动态代理，CGLIB或其它代理策略。 缺省实现将根据情况选择动态代理或者CGLIB。一般情况下应该没有使用这个属性的需要；它是被设计来在Spring 1.1中添加新的代理类型的。
- 6: **proxyInterfaces**: 需要代理的接口名的字符串数组。如果没有提供，将为目标类使用一个CGLIB代理
- 7: **interceptorNames**: Advisor的字符串数组，可以包括拦截器或其它通知的名字。顺序是很重要的，排在前面的将被优先服务。就是说列表里的第一个拦截器将能够第一个拦截调用。

这里的名字是当前工厂中bean的名字，包括父工厂中bean的名字。这里你不能使用bean的引用因为这会导致ProxyFactoryBean忽略通知的单例设置。

你可以把一个拦截器的名字加上一个星号作为后缀（*）。这将导致这个应用程序里所有名字以星号之前部分开头的advisor都被应用。



ProxyFactoryBean-3

n 让我们看一个关于ProxyFactoryBean的简单例子，这个是对接口进行代理

```
<bean id="personTarget" class="com.mycompany.PersonImpl"></bean>
<bean id="myAdvisor" class="com.mycompany.MyAdvisor"></bean>
<bean id="debugInterceptor" class="org.DebugInterceptor"></bean>
<bean id="person"
      class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="proxyInterfaces"><value>com.Person</value></property>
  <property name="target"><ref local="personTarget"/></property>
  <property name="interceptorNames">
    <list>
      <value>myAdvisor</value>
      <value>debugInterceptor</value>
    </list>
  </property>
</bean>
```

n 如果要对类进行代理，只需要把proxyTargetClass属性设为true



AOP的API 使用示例-1

n 定义如下的接口:

```
public interface Api {  
    public String t(int a);  
}
```

n 写实现类如下:

```
public class Impl implements Api {  
    public String t(int a) {  
        System.out.println("now in impl a="+a);  
        return a+" test";  
    }  
}
```

n 写一个Before的Advice实现

```
public class MyBefore implements org.springframework.aop.MethodBeforeAdvice {  
    public void before(Method method, Object[] args, Object target)  
        throws Throwable {  
        System.out.println("现在调用的是: "+target.getClass().getName()  
            +", 方法是: "+method.getName() +", 参数是: "+args[0]);  
    }  
}
```



AOP的API 使用示例-2

n 配置文件如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="myTarget" class="cn.javass.Spring3.aop1.Impl"/>
    <bean id="myBefore" class="cn.javass.Spring3.aop1.MyBefore"/>
    <bean id="myIn" class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxylInterfaces">
            <value>cn.javass.Spring3.aop1.Api</value>
        </property>
        <property name="target"><ref local="myTarget"/></property>
        <property name="interceptorNames">
            <list>
                <value>beforeAdvisor</value>
            </list>
        </property>
    </bean>
```



《深入浅出学Spring3开发》——系列精品教程

AOP的API 使用示例-3

```
<bean id="beforeAdvisor"
      class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
  <property name="advice">
    <ref local="myBefore"/>
  </property>
  <property name="patterns">
    <list>
      <value>.*</value>
    </list>
  </property>
</bean>
</beans>
```

n 客户端文件如下:

```
public class Client {
  public static void main(String[] args) {
    ApplicationContext ctx = new ClassPathXmlApplicationContext(
      new String[] {"applicationContext-1.xml"});
    Api api = (Api)ctx.getBean("myIn");
    String s = api.t(123);
    System.out.println("sssss="+s);
  }
}
```



AOP与设计模式

n AOP与设计模式

AOP实际是设计模式的一种扩展，设计模式所追求的是降低代码之间的耦合度，增加程序的灵活性和可重用性，AOP实际上就是设计模式所追求的目标的一种实现。

所谓的分离关注就是将某一通用的需求功能从不相关的类之中分离出来；同时，能够使得很多类共享一个行为，一旦行为发生变化，不必修改很多类，只要修改这个行为就可以，从而达到更好的可扩展性和复用性。



《深入浅出学Spring3开发》——系列精品教程

AOP开发步骤

n AOP包括三个清晰的开发步骤：

- 1: **功能横切**：分解需求提取出横切关注点。
- 2: **实现分离**：各自独立的实现这些关注点所需要完成的功能。
- 3: **功能回贴**：在这一步里，方面集成器通过创建一个模块单元——方面来指定重组的规则。重组过程——也叫织入或结合——则使用这些信息来构建最终系统。

AOP与OOP开发的不同关键在于它处理横切关注点的方式，在AOP中，每个关注点的实现都不知道其它关注点是否会‘关注’它，如信用卡处理模块并不知道其它的关注点实现正在为它做日志和验证操作。



AOP设计的考虑-1

n 应该拦截字段吗？

我们在实现AOP的时候，想要修改某个属性。者通常不是一种好的做法，因为属性通常应该在类的内部进行访问，他体现了类的封装性，如果拦截字段并修改它，那么就会破坏这种封装性。

而拦截方法不会破坏封装性，因为这些方法本身就是能够被外界访问的。

如果确实需要拦截字段，通常的做法是对这个字段定义getXXX和setXXX方法，然后拦截方法，进行相应的操作。

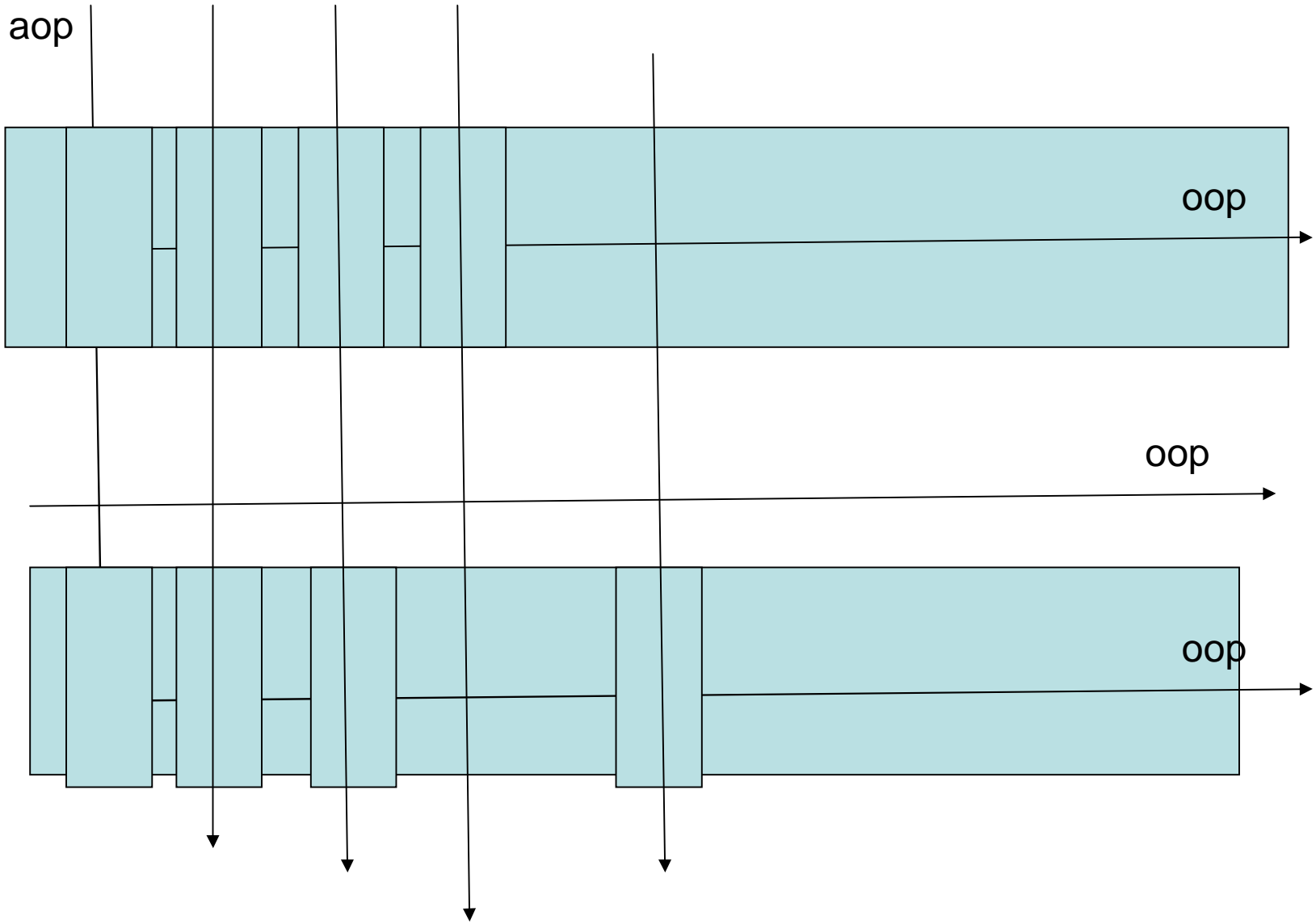
n 太多的方面？

也许你会认为AOP实在是太强大了，可以到处使用，从而设计出太多的方面，以至于当一个方法被调用时，都难以说清楚究竟执行了哪些代码。

这是很可怕的，“过犹不及”，这是典型的过度设计的毛病。通常情况下，很少有一个对象实例需要5个方面以上。况且过多的方面会影响性能。

在设计的时候，通常如下情况会设计成方面：

- (1) 大部分模块都需要使用的通用功能，包括系统级或模块级的功能
- (2) 预计目前的实现，在今后进行功能扩展的可能性很大的地方





AOP设计的考虑-2

n 正交性

如果多个方面互相影响，造成一些无法预测的结果，该怎么办？多个切入点的功能实现叠加起来，甚至造成错误？

对于这种情况，在设计AOP的时候要特别注意，要遵循连接点的正交模型：不同种类的连接点和不同种类的实现应该能够以任何顺序组合使用。

换句话说，请保持你设计的方面的独立性，功能实现的独立性，不依赖于多个方面的执行先后顺序。

n 对粗粒度对象使用AOP

AOP通常用来对粗粒度的对象进行功能增强，比如对业务逻辑对象，拦截某个业务方法，进行功能增强，从而提高系统的可扩展性。

注意不要在细粒度的对象上使用AOP，比如对某个实体描述对象。在运行时，这种细粒度对象通常实例很多，比如可能有多条数据，这种情况下，使用AOP，会有大量的方法拦截带来的反射处理，严重影响性能。



《深入浅出学Spring3开发》——系列精品教程

本节课程小结

n AOP开发

包括：Schema风格的AOP支持、AOP声明风格的选择、AOP的代理机制、Spring1.x中的AOP的APIs以及基本使用、AOP开发和设计考虑

n 作业：

1：复习和练习这些理论知识