黄勇 关注此人

关注(26) 粉丝(1038) 积分(608)

学习 • 讨论 • 总结 • 分享

.发送留言 .请教问题

**博客分类**

- 未分类(0)
- Smart(57)
- Java 那点事儿(10)
- 前端(1)
- 设计模式(2)
- 工具(13)
- 产品(4)
- 管理(1)
- 售前(1)
- 职场(1)
- 源码分析(3)
- 代码备忘(8)

**阅读排行**

1. 1. Smart - 轻量级 Java Web 开发框架
2. 2. AOP 那点事儿
3. 3. ThreadLocal 那点事儿
4. 4. Entity 映射机制实现原理
5. 5. Proxy 那点事儿
6. 6. 一个简单的 Cache 淘汰策略
7. 7. ThreadLocal 那点事儿（续集）
8. 8. 安装 CAS 服务器

**最新评论**

- @黄勇：引用来自“彭博”的评论 不敢实现这个接口，弄个... 查看»
- @彭博：不敢实现这个接口，弄个抽象类是不是更好 查看»
- @黄勇：引用来自“不愿意透露姓名的严瑾”的评论绕 确实... 查看»
- @不愿意透露姓名的严瑾：绕 查看»
- @黄勇：引用来自“ruanzy”的评论思路很好，充分利用jav... 查看»
- @ruanzy：思路很好，充分利用java的多态和面向接口编程。 ... 查看»
- @黄勇：引用来自“谢宝龙”的评论请问使用事务的时候，如... 查看»

- @EugeneQiu：很赞的分享。有意思，看来这两天得捣腾一下Smart... 查看»
- @谢宝龙：请问使用事务的时候，如何保证service的数据库链... 查看»
- @黄勇：引用来自“webit”的评论 不是故意捣乱的，不知道... 查看»

**友情链接**

- 1. 哈库呐 - Hasor
- 2. 悠悠然然 - Tiny
- 3. 黄亿华 - WebMagic
- 4. Dead_knight - Snaker

**访客统计**

- 今日访问：37
- 昨日访问：498
- 本周访问：1414
- 本月访问：4012
- 所有访问：79285



# 原 荐 AOP 那点事儿

发表于8个月前(2013-09-14 13:07) 阅读（9211）| 评论（77） 459人收藏此文章, 我要收藏
赞29

Java那点事儿 AOP Spring

又是一个周末，刚给宝宝喂完牛奶，终于让她睡着了。所以现在我才能腾出手来，坐在电脑面前给大家写这篇文章。

今天我要和大家分享的是 AOP（Aspect-Oriented Programming）这个东西，名字与 OOP 仅差一个字母，其实它是对 OOP 编程方式的一种补充，并非是取而代之。翻译过来就是“面向方面编程”，可我更倾向于翻译为“面向切面编程”。它听起有些的神秘，为什么呢？当你看完这篇文章的时候，就会知道，我们做的很重要的工作就是去写这个“切面”。那么什么是“切面”呢？

没错！就是用一把刀来切一坨面。注意，相对于面而言，我们一定是横着来切它，这简称为“横切”。可以把一段代码想象成一坨面，同样也可以用一把刀来横切它，下面要做的就是如何去实现这把刀！

需要澄清的是，这个概念不是由 Rod Johnson（老罗）提出的。其实很早以前就有了，目前最知名最强大的 Java 开源项目就是 AspectJ 了，然而它的前身是 AspectWerkz（该项目已经在 2005 年停止更新），这才是 AOP 的老祖宗。老罗（一个头发秃得和我老爸有一拼的天才）写了一个叫做 Spring 框架，从此一炮走红，成为了 Spring 之父。他在自己的 IOC 的基础之上，又实现了一套 AOP 的框架，后来仿佛发现自己越来越走进深渊里，在不能自拔的时候，有人建议他还是集成 AspectJ 吧，他在万般无奈之下才接受了该建议。于是，我们现在用得最多的想必就是 Spring + AspectJ 这种 AOP 框架了。

那么 AOP 到底是什么？如何去使用它？本文将逐步带您进入 AOP 的世界，让您感受到前所未有的畅快！

不过在开始讲解 AOP 之前，我想有必要回忆一下这段代码：

## 1. 写死代码

先来一个接口：

```
1 public interface Greeting {
2
```

```
    void sayHello(String name);
}
```

还有一个实现类：

```
public class GreetingImpl implements Greeting {

    @Override
    public void sayHello(String name) {
        before();
        System.out.println("Hello! " + name);
        after();
    }

    private void before() {
        System.out.println("Before");
    }

    private void after() {
        System.out.println("After");
    }
}
```

before() 与 after() 方法写死在 sayHello() 方法体中了，这样的代码的味道非常不好。如果哪位仁兄大量写了这样的代码，肯定要被你的架构师骂个够呛。

比如：我们要统计每个方法的执行时间，以对性能作出评估，那是不是要在每个方法的一头一尾都做点手脚呢？

再比如：我们要写一个 JDBC 程序，那是不是也要在方法的开头去连接数据库，方法的末尾去关闭数据库连接呢？

这样的代码只会把程序员累死，把架构师气死！

一定要想办法对上面的代码进行重构，首先给出三个解决方案：

## 2. 静态代理

最简单的解决方案就是使用静态代理模式了，我们单独为 GreetingImpl 这个类写一个代理类：

```
public class GreetingProxy implements Greeting {

    private GreetingImpl greetingImpl;

    public GreetingProxy(GreetingImpl greetingImpl) {
        this.greetingImpl = greetingImpl;
    }

    @Override
    public void sayHello(String name) {
        before();
        greetingImpl.sayHello(name);
        after();
    }

    private void before() {
        System.out.println("Before");
    }

    private void after() {
        System.out.println("After");
    }
}
```

就用这个 GreetingProxy 去代理 GreetingImpl，下面看看客户端如何来调用：

```
public class Client {

    public static void main(String[] args) {
        Greeting greetingProxy = new GreetingProxy(new GreetingImpl());
        greetingProxy.sayHello("Jack");
    }
}
```

这样写没错，但是有个问题，XxxProxy 这样的类会越来越多，如何才能将这些代理类尽可能减少呢？最好只有一个代理类。

这时我们就需要使用 JDK 提供的动态代理了。

### 3. JDK 动态代理

```
public class JDKDynamicProxy implements InvocationHandler {

    private Object target;

    public JDKDynamicProxy(Object target) {
        this.target = target;
    }

    @SuppressWarnings("unchecked")
    public <T> T getProxy() {
        return (T) Proxy.newProxyInstance(
            target.getClass().getClassLoader(),
            target.getClass().getInterfaces(),
            this
        );
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        before();
        Object result = method.invoke(target, args);
        after();
        return result;
    }

    private void before() {
        System.out.println("Before");
    }

    private void after() {
        System.out.println("After");
    }
}
```

客户端是这样调用的：

```
public class Client {

    public static void main(String[] args) {
        Greeting greeting = new JDKDynamicProxy(new GreetingImpl()).getProxy();
        greeting.sayHello("Jack");
    }
}
```

这样所有的代理类都合并到动态代理类中了，但这样做仍然存在一个问题：JDK 给我们提供的动态代理只能代理接口，而不能代理没有接口的类。有什么方法可以解决呢？

### 4. CGLib 动态代理

我们使用开源的 CGLib 类库可以代理没有接口的类，这样就弥补了 JDK 的不足。CGLib 动态代理类是这样玩的：

```
public class CGLibDynamicProxy implements MethodInterceptor {

    private static CGLibDynamicProxy instance = new CGLibDynamicProxy();

    private CGLibDynamicProxy() {
    }

    public static CGLibDynamicProxy getInstance() {
        return instance;
    }

    @SuppressWarnings("unchecked")
    public <T> T getProxy(Class<T> cls) {
        return (T) Enhancer.create(cls, this);
```

```
    }

    @Override
    public Object intercept(Object target, Method method, Object[] args, MethodProxy proxy) throws Throwable {
        before();
        Object result = proxy.invokeSuper(target, args);
        after();
        return result;
    }

    private void before() {
        System.out.println("Before");
    }

    private void after() {
        System.out.println("After");
    }
}
```

以上代码中了 Singleton 模式，那么客户端调用也更加轻松了：

```
public class Client {

    public static void main(String[] args) {
        Greeting greeting = CGLibDynamicProxy.getInstance().getProxy(GreetingImpl.class);
        greeting.sayHello("Jack");
    }
}
```

到此为止，我们能做的都做了，问题似乎全部都解决了。但事情总不会那么完美，而我们一定要追求完美！

老罗搞出了一个 AOP 框架，能否做到完美而优雅呢？请大家继续往下看吧！

## 5. Spring AOP：前置增强、后置增强、环绕增强（编程式）

在 Spring AOP 的世界里，与 AOP 相关的术语实在太多，往往也是我们的“拦路虎”，不管是看那本书或是技术文档，在开头都要将这些术语逐个灌输给读者。我想这完全是在吓唬人了，其实没那么复杂的，大家放轻松一点。

我们上面例子中提到的 before() 方法，在 Spring AOP 里就叫 **Before Advice（前置增强）**。有些人将 Advice 直译为“通知”，我想这是不太合适的，因为它根本就没有“通知”的含义，而是对原有代码功能的一种“增强”。再说，CGLib 中也有一个 Enhancer 类，它就是一个增强类。

此外，像 after() 这样的方法就叫 **After Advice（后置增强）**，因为它放在后面来增强代码的功能。

如果能把 before() 与 after() 合并在一起，那就叫 **Around Advice（环绕增强）**，就像汉堡一样，中间夹一根火腿。

这三个概念是不是轻松地理解了呢？如果是，那就继续吧！

我们下面要做的就是去实现这些所谓的“增强类”，让他们横切到代码中，而不是将这些写死在代码中。

先来一个前置增强类吧：

```
public class GreetingBeforeAdvice implements MethodBeforeAdvice {

    @Override
    public void before(Method method, Object[] args, Object target) throws Throwable {
        System.out.println("Before");
    }
}
```

注意：这个类实现了 org.springframework.aop.MethodBeforeAdvice 接口，我们将需要增强的代码放入其中。

再来一个后置增强类吧：

```
public class GreetingAfterAdvice implements AfterReturningAdvice {

    @Override
```

```
    public void afterReturning(Object result, Method method, Object[] args, Object target) throws Throwable {
        System.out.println("After");
    }
}
```

类似地，这个类实现了 org.springframework.aop.AfterReturningAdvice 接口。

最后用一个客户端来把它们集成起来，看看如何调用吧：

```
public class Client {

    public static void main(String[] args) {
        ProxyFactory proxyFactory = new ProxyFactory();     // 创建代理工厂
        proxyFactory.setTarget(new GreetingImpl());         // 射入目标类对象
        proxyFactory.addAdvice(new GreetingBeforeAdvice()); // 添加前置增强
        proxyFactory.addAdvice(new GreetingAfterAdvice());  // 添加后置增强

        Greeting greeting = (Greeting) proxyFactory.getProxy(); // 从代理工厂中获取代理
        greeting.sayHello("Jack");                              // 调用代理的方法
    }
}
```

请仔细阅读以上代码及其注释，您会发现，其实 Spring AOP 还是挺简单的，对吗？

当然，我们完全可以只定义一个增强类，让它同时实现 MethodBeforeAdvice 与 AfterReturningAdvice 这两个接口，如下：

```
public class GreetingBeforeAndAfterAdvice implements MethodBeforeAdvice, AfterReturningAdvice {

    @Override
    public void before(Method method, Object[] args, Object target) throws Throwable {
        System.out.println("Before");
    }

    @Override
    public void afterReturning(Object result, Method method, Object[] args, Object target) throws Throwable {
        System.out.println("After");
    }
}
```

这样我们只需要使用一行代码，同时就可以添加前置与后置增强：

```
proxyFactory.addAdvice(new GreetingBeforeAndAfterAdvice());
```

刚才有提到“环绕增强”，其实这个东西可以把“前置增强”与“后置增强”的功能给合并起来，无需让我们同时实现以上两个接口。

```
public class GreetingAroundAdvice implements MethodInterceptor {

    @Override
    public Object invoke(MethodInvocation invocation) throws Throwable {
        before();
        Object result = invocation.proceed();
        after();
        return result;
    }

    private void before() {
        System.out.println("Before");
    }

    private void after() {
        System.out.println("After");
    }
}
```

环绕增强类需要实现 org.aopalliance.intercept.MethodInterceptor 接口。注意，这个接口不是 Spring 提供的，它是 AOP 联盟（一个很牛逼的联盟）写的，Spring 只是借用了它。

在客户端中同样也需要将该增强类的对象添加到代理工厂中：

```
proxyFactory.addAdvice(new GreetingAroundAdvice());
```

好了，这就是 Spring AOP 的基本用法，但这只是“编程式”而已。Spring AOP 如果只是这样，那就太傻逼了，它

曾经也是一度宣传用 Spring 配置文件的方式来定义 Bean 对象，把代码中的 new 操作全部解脱出来。

## 6. Spring AOP：前置增强、后置增强、环绕增强（声明式）

先看 Spring 配置文件是如何写的吧：

```
<?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"
       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">

    <!-- 扫描指定包（将 @Component 注解的类自动定义为 Spring Bean） -->
    <context:component-scan base-package="aop.demo"/>

    <!-- 配置一个代理 -->
    <bean id="greetingProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="interfaces" value="aop.Greeting"/> <!-- 需要代理的接口 -->
        <property name="target" ref="greetingImpl"/>       <!-- 接口实现类 -->
        <property name="interceptorNames">                 <!-- 拦截器名称（也就是增强类名称，
Spring Bean 的 id） -->
            <list>
                <value>greetingAroundAdvice</value>
            </list>
        </property>
    </bean>

</beans>
```

一定要阅读以上代码的注释，其实使用 ProxyFactoryBean 就可以取代前面的 ProxyFactory，其实它们俩就一回事儿。我认为 interceptorNames 应该改名为 adviceNames 或许会更容易让人理解，不就是往这个属性里面添加增强类吗？

此外，如果只有一个增强类，可以使用以下方法来简化：

```
...

    <bean id="greetingProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="interfaces" value="aop.Greeting"/>
        <property name="target" ref="greetingImpl"/>
        <property name="interceptorNames" value="greetingAroundAdvice"/> <!-- 注意这行配置 -->
    </bean>

...
```

还需要注意的是，这里使用了 Spring 2.5+ 的特性“Bean 扫描”，这样我们就无需在 Spring 配置文件里不断地定义 <bean id="xxx" class="xxx"/> 了，从而解脱了我们的双手。

看看这是有多么的简单：

```
@Component
public class GreetingImpl implements Greeting {

    ...
}
```

```
@Component
public class GreetingAroundAdvice implements MethodInterceptor {

    ...
}
```

最后看看客户端吧：

```
public class Client {

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("aop/demo/spring.xml"); // 获
取 Spring Context
        Greeting greeting = (Greeting) context.getBean("greetingProxy");                    // 从 Context 中
```

```
        根据 id 获取 Bean 对象（其实就是一个代理）
        greeting.sayHello("Jack");                                    // 调用代理的
方法
    }
}
```

代码量确实少了，我们将配置性的代码放入配置文件，这样也有助于后期维护。更重要的是，代码只关注于业务逻辑，而将配置放入文件中。这是一条最佳实践！

除了上面提到的那三类增强以外，其实还有两类增强也需要了解一下，关键的时候您要能想得到它们才行。

## 7. Spring AOP：抛出增强

程序报错，抛出异常了，一般的做法是打印到控制台或日志文件中，这样很多地方都得去处理，有没有一个一劳永逸的方法呢？那就是 **Throws Advice（抛出增强）**，它确实很强，不信你就继续往下看：

```
@Component
public class GreetingImpl implements Greeting {

    @Override
    public void sayHello(String name) {
        System.out.println("Hello! " + name);

        throw new RuntimeException("Error"); // 故意抛出一个异常，看看异常信息能否被拦截到
    }
}
```

下面是抛出增强类的代码：

```
@Component
public class GreetingThrowAdvice implements ThrowsAdvice {

    public void afterThrowing(Method method, Object[] args, Object target, Exception e) {
        System.out.println("---------- Throw Exception ----------");
        System.out.println("Target Class: " + target.getClass().getName());
        System.out.println("Method Name: " + method.getName());
        System.out.println("Exception Message: " + e.getMessage());
        System.out.println("-------------------------------------");
    }
}
```

抛出增强类需要实现 org.springframework.aop.ThrowsAdvice 接口，在接口方法中可获取方法、参数、目标对象、异常对象等信息。我们可以把这些信息统一写入到日志中，当然也可以持久化到数据库中。

这个功能确实太棒了！但还有一个更厉害的增强。如果某个类实现了 A 接口，但没有实现 B 接口，那么该类可以调用 B 接口的方法吗？如果您没有看到下面的内容，一定不敢相信原来这是可行的！

## 8. Spring AOP：引入增强

以上提到的都是对方法的增强，那能否对类进行增强呢？用 AOP 的行话来讲，对方法的增强叫做 **Weaving（织入）**，而对类的增强叫做 **Introduction（引入）**。而 **Introduction Advice（引入增强）**就是对类的功能增强，它也是 Spring AOP 提供的最后一种增强。建议您一开始千万不要去看《Spring Reference》，否则您一定会后悔的。因为当您看了以下的代码示例后，一定会彻底明白什么才是引入增强。

定义了一个新接口 Apology（道歉）：

```
public interface Apology {

    void saySorry(String name);
}
```

但我不想在代码中让 GreetingImpl 直接去实现这个接口，我想在程序运行的时候动态地实现它。因为假如我实现了这个接口，那么我就一定要改写 GreetingImpl 这个类，关键是我不想改它，或许在真实场景中，这个类有1万行代码，我实在是不敢动了。于是，我需要借助 Spring 的引入增强。这个有点意思了！

```
@Component
```

```java
public class GreetingIntroAdvice extends DelegatingIntroductionInterceptor implements Apology {

    @Override
    public Object invoke(MethodInvocation invocation) throws Throwable {
        return super.invoke(invocation);
    }

    @Override
    public void saySorry(String name) {
        System.out.println("Sorry! " + name);
    }
}
```

以上定义了一个引入增强类，扩展了 org.springframework.aop.support.DelegatingIntroductionInterceptor 类，同时也实现了新定义的 Apology 接口。在类中首先覆盖了父类的 invoke() 方法，然后实现了 Apology 接口的方法。我就是想用这个增强类去丰富 GreetingImpl 类的功能，那么这个 GreetingImpl 类无需直接实现 Apology 接口，就可以在程序运行的时候调用 Apology 接口的方法了。这简直是太神奇的！

看看是如何配置的吧：

```xml
<?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"
       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">

    <context:component-scan base-package="aop.demo"/>

    <bean id="greetingProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="interfaces" value="aop.demo.Apology"/>          <!-- 需要动态实现的接口 -->
        <property name="target" ref="greetingImpl"/>                    <!-- 目标类 -->
        <property name="interceptorNames" value="greetingIntroAdvice"/> <!-- 引入增强 -->
        <property name="proxyTargetClass" value="true"/>                <!-- 代理目标类（默认为 false，代理接口） -->
    </bean>

</beans>
```

需要注意 proxyTargetClass 属性，它表明是否代理目标类，默认为 false，也就是代理接口了，此时 Spring 就用 JDK 动态代理。如果为 true，那么 Spring 就用 CGLib 动态代理。这简直就是太方便了！Spring 封装了这一切，让程序员不在关心那么多的细节。我们要向老罗同志致敬，您是我们心中永远的 idol！

当您看完下面的客户端代码，一定会完全明白以上的这一切：

```java
public class Client {

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("aop/demo/spring.xml");
        GreetingImpl greetingImpl = (GreetingImpl) context.getBean("greetingProxy"); // 注意：转型为目标类，而并非它的 Greeting 接口
        greetingImpl.sayHello("Jack");

        Apology apology = (Apology) greetingImpl; // 将目标类强制向上转型为 Apology 接口（这是引入增强给我们带来的特性，也就是“接口动态实现”功能）
        apology.saySorry("Jack");
    }
}
```

没想到 saySorry() 方法原来是可以被 greetingImpl 对象来直接调用的，只需将其强制转换为该接口即可。

我们再次感谢 Spring AOP，感谢老罗给我们提供了这么强大的特性！

其实，Spring AOP 还有很多精彩的地方，下一篇将介绍更多更有价值的 AOP 技术，让大家得到更多的收获。

**未完，待续...**

**AOP 那点儿事（续集）**

源码下载

分享到： 新浪微博 weibo.com 腾讯微博 t.qq.com 29赞




- « 上一篇
- 下一篇 »




# 评论77

- 1楼：城南往事 发表于 2013-09-14 13:10 回复此评论
  写的不错，继续分享，谢谢！
- 
  2楼：ToB蓝波湾 发表于 2013-09-14 16:32 回复此评论
  我怎么感觉静态代理跟装饰模式有一点相似。。。装饰模式也可以用来实现静态代理的功能吧。。。楼主是怎么理解的啊？
- 3楼：黄勇 发表于 2013-09-15 10:29 回复此评论

  **引用来自“雅典娜拉”的评论**

  
  我怎么感觉静态代理跟装饰模式有一点相似。。。装饰模式也可以用来实现静态代理的功能吧。。。楼主是怎么理解的啊？

  没错，确实有些类似。不妨将装饰模式理解为代理模式的一种扩展，代理模式到代理类就把活给干了，而装饰模式中有一个装饰器，结构上十分类似于代理类，但它是一个抽象类，需要子类来完成具体的装饰行为。不知道我这样的回答，您满意吗？
- 
  4楼：ToB蓝波湾 Android 发表于 2013-09-15 10:57 回复此评论
  哦，多谢指点
- 
  5楼：无争 发表于 2013-09-16 18:30 回复此评论
  楼主很勤奋啊，赞
- 
  6楼：谭又中 Android 发表于 2013-09-16 23:45 回复此评论
  楼主很强大，佩服...
- 7楼：jorneyr 发表于 2013-09-17 08:24 回复此评论

  **引用来自“雅典娜拉”的评论**

我怎么感觉静态代理跟装饰模式有一点相似。。。装饰模式也可以用来实现静态代理的功能吧。。。楼主是怎么理解的啊？

其实差不太多，但是细微上有点区别。
1. 代理使用的是继承，装饰使用的是组合。
2. 组合更方便，可以动态的增加更多的装饰。
3. 代理是控制动作的权限，而装饰是改变了原来的动作。

- 8楼：ToB蓝波湾 Android 发表于 2013-09-17 08:28 回复此评论

  **引用来自“jorneyr”的评论**

  **引用来自“雅典娜拉”的评论**

  

  我怎么感觉静态代理跟装饰模式有一点相似。。。装饰模式也可以用来实现静态代理的功能吧。。。楼主是怎么理解的啊？

  其实差不太多，但是细微上有点区别。
  1. 代理使用的是继承，装饰使用的是组合。
  2. 组合更方便，可以动态的增加更多的装饰。
  3. 代理是控制动作的权限，而装饰是改变了原来的动作。

  嗯，有道理，多谢

- 

  9楼：BugTermina 发表于 2013-09-17 08:42 回复此评论
  非常感谢楼主，俺正为考试有AOP的内容发愁，没学过Spring，老师给出的书看得我似懂非懂，还是楼主用中文写出来看的明白……

- 

  10楼：晋哥哥 发表于 2013-09-17 09:08 回复此评论
  总结的很好


- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- >


插入： 表情 开源软件

发表评论

关闭插入表情
**关闭相关文章阅读**

- 2013/09/14 AOP 那点事儿（续集）
- 2013/02/21 SPRING AOP
- 2013/11/14 Spring AOP的实现
- 2013/09/22 Spring AOP
- 2013/09/13 Spring AOP（1）相关概念