一、对AOP的初印象

首先先给出一段比较专业的术语（来自百度）：

在软件业，AOP为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。AOP是OOP的延续，是软件开发中的一个

热点，也是Spring框架中的一个重要内容，是函数式编程的一种衍生范型。利用AOP可以对业务逻辑

的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高

了开发的效率。

1

2

3

4

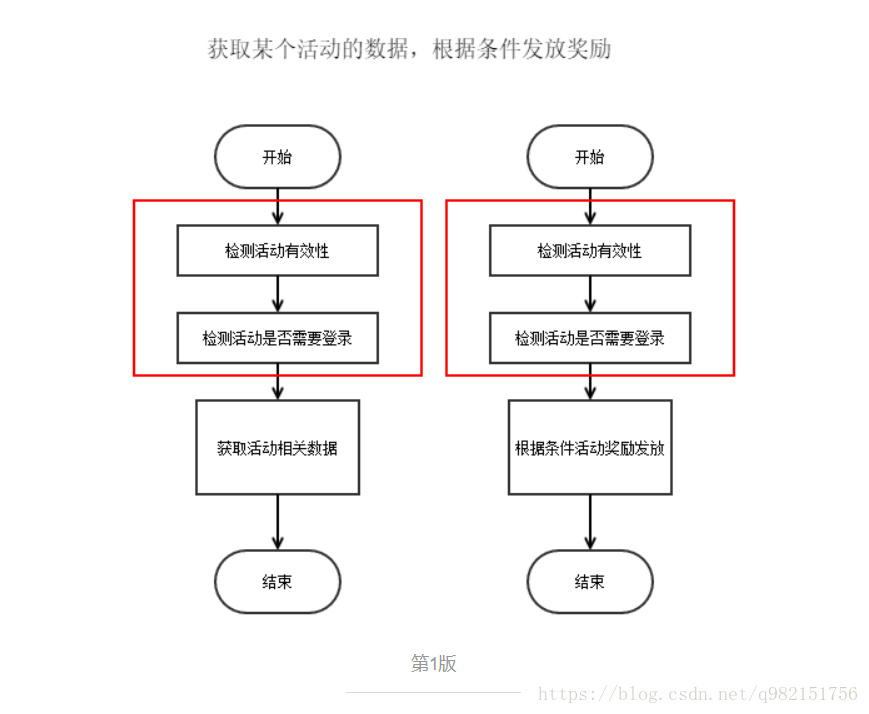
5

然后我们举一个比较容易理解的例子（来自：Spring 之 AOP）：

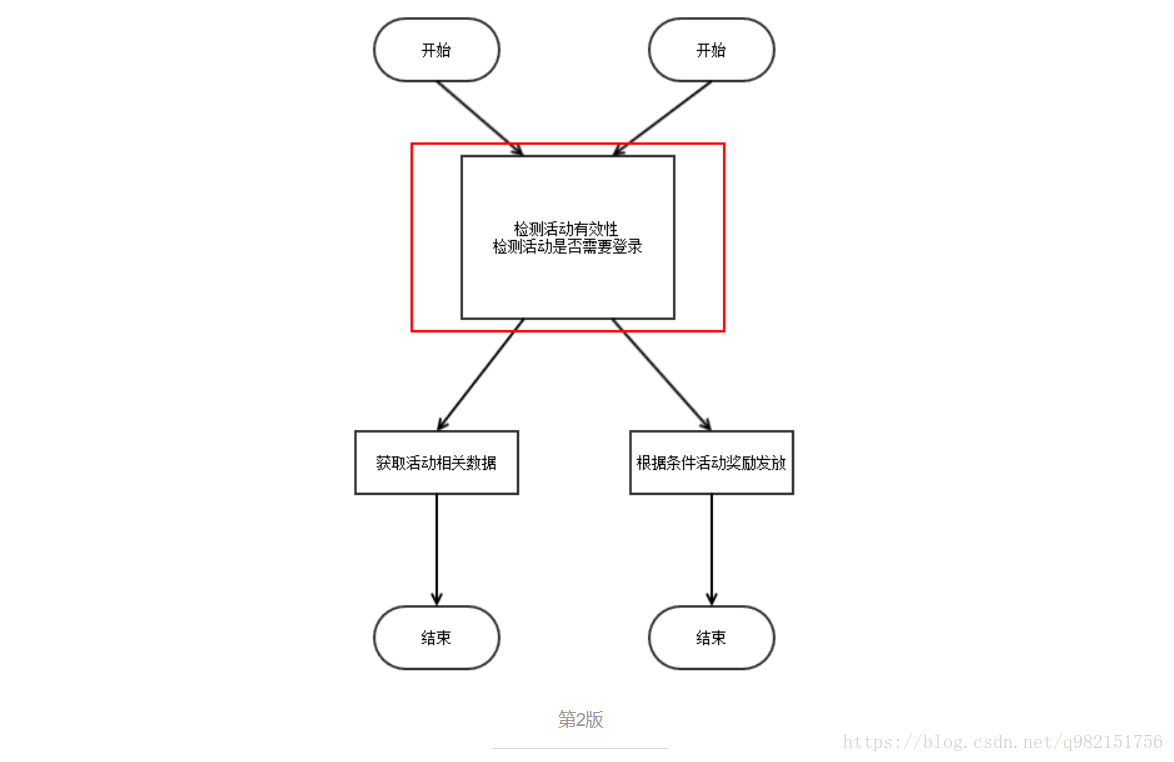
要理解切面编程，就需要先理解什么是切面。用刀把一个西瓜分成两瓣，切开的切口就是切面；炒菜，锅与炉子共同来完成炒菜，锅与炉子就是切面。web层级设计中，web层->网关层->服务层->数据层，每一层之间也是一个切面。编程中，对象与对象之间，方法与方法之间，模块与模块之间都是一个个切面。

我们一般做活动的时候，一般对每一个接口都会做活动的有效性校验（是否开始、是否结束等等）、以及这个接口是不是需要用户登录。

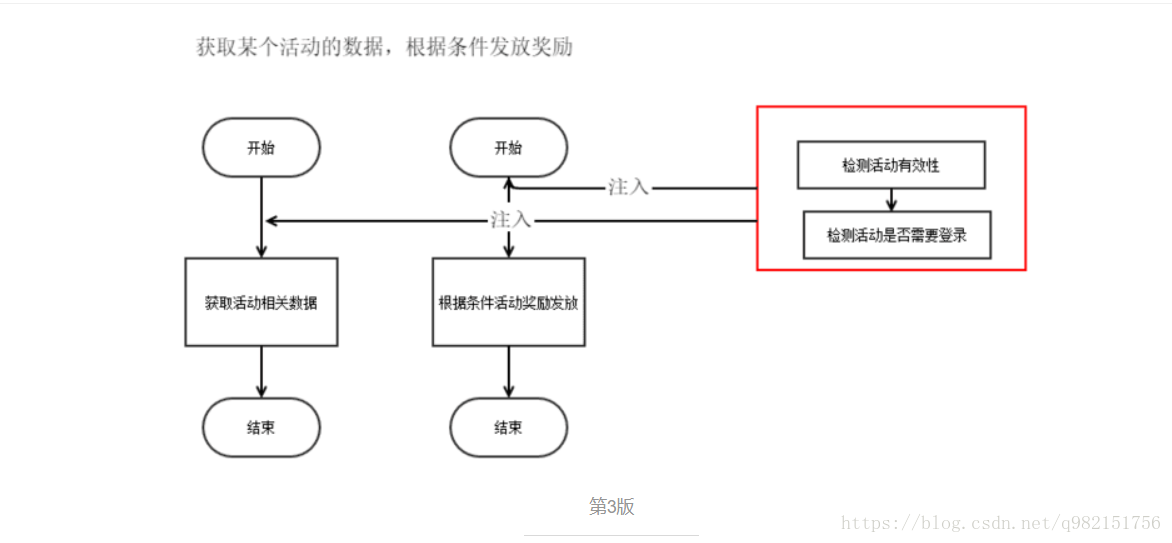
按照正常的逻辑，我们可以这么做。



这有个问题就是，有多少接口，就要多少次代码copy。对于一个“懒人”，这是不可容忍的。好，提出一个公共方法，每个接口都来调用这个接口。这里有点切面的味道了。



同样有个问题，我虽然不用每次都copy代码了，但是，每个接口总得要调用这个方法吧。于是就有了切面的概念，我将方法注入到接口调用的某个地方（切点）。



这样接口只需要关心具体的业务，而不需要关注其他非该接口关注的逻辑或处理。

红框处，就是面向切面编程。

二、AOP中的相关概念

看过了上面的例子，我想大家脑中对AOP已经有了一个大致的雏形，但是又对上面提到的切面之类的术语有一些模糊的地方，接下来就来讲解一下AOP中的相关概念，了解了AOP中的概念，才能真正的掌握AOP的精髓。

这里还是先给出一个比较专业的概念定义：

Aspect（切面）： Aspect 声明类似于 Java 中的类声明，在 Aspect 中会包含着一些 Pointcut 以及相应的 Advice。

Joint point（连接点）：表示在程序中明确定义的点，典型的包括方法调用，对类成员的访问以及异常处理程序块的执行等等，它自身还可以嵌套其它 joint point。

Pointcut（切点）：表示一组 joint point，这些 joint point 或是通过逻辑关系组合起来，或是通过通配、正则表达式等方式集中起来，它定义了相应的 Advice 将要发生的地方。

Advice（增强）：Advice 定义了在 Pointcut 里面定义的程序点具体要做的操作，它通过 before、after 和 around 来区别是在每个 joint point 之前、之后还是代替执行的代码。

Target（目标对象）：织入 Advice 的目标对象.。

Weaving（织入）：将 Aspect 和其他对象连接起来, 并创建 Adviced object 的过程

然后举一个容易理解的例子：

看完了上面的理论部分知识, 我相信还是会有不少朋友感觉到 AOP 的概念还是很模糊, 对 AOP 中的各种概念理解的还不是很透彻. 其实这很正常, 因为 AOP 中的概念是在是太多了, 我当时也是花了老大劲才梳理清楚的.

下面我以一个简单的例子来比喻一下 AOP 中 Aspect, Joint point, Pointcut 与 Advice之间的关系.

让我们来假设一下, 从前有一个叫爪哇的小县城, 在一个月黑风高的晚上, 这个县城中发生了命案. 作案的凶手十分狡猾, 现场没有留下什么有价值的线索. 不过万幸的是, 刚从隔壁回来的老王恰好在这时候无意中发现了凶手行凶的过程, 但是由于天色已晚, 加上凶手蒙着面, 老王并没有看清凶手的面目, 只知道凶手是个男性, 身高约七尺五寸. 爪哇县的县令根据老王的描述, 对守门的士兵下命令说: 凡是发现有身高七尺五寸的男性, 都要抓过来审问. 士兵当然不敢违背县令的命令, 只好把进出城的所有符合条件的人都抓了起来.

来让我们看一下上面的一个小故事和 AOP 到底有什么对应关系.

首先我们知道, 在 Spring AOP 中 Joint point 指代的是所有方法的执行点, 而 point cut 是一个描述信息, 它修饰的是 Joint point, 通过 point cut, 我们就可以确定哪些 Joint point 可以被织入 Advice. 对应到我们在上面举的例子, 我们可以做一个简单的类比, Joint point 就相当于 爪哇的小县城里的百姓,pointcut 就相当于 老王所做的指控, 即凶手是个男性, 身高约七尺五寸, 而 Advice 则是施加在符合老王所描述的嫌疑人的动作: 抓过来审问.

为什么可以这样类比呢?

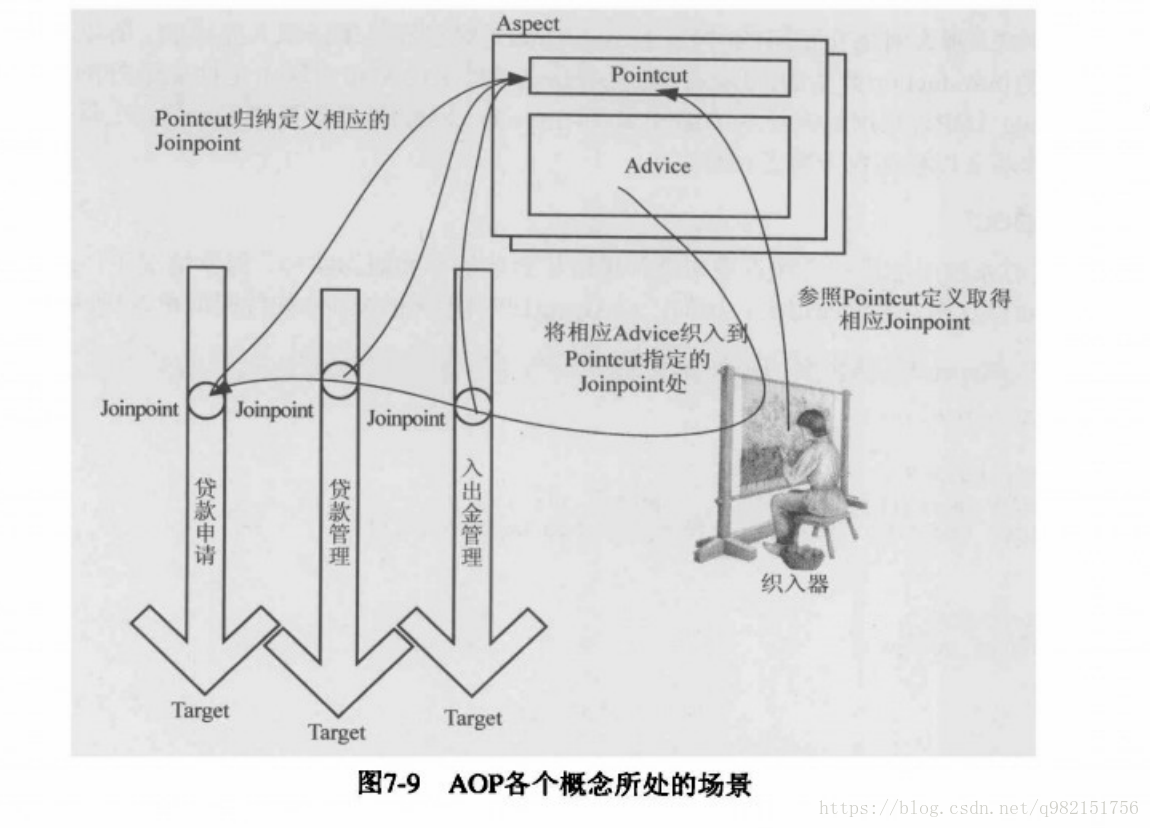
Joint point ： 爪哇的小县城里的百姓: 因为根据定义, Joint point 是所有可能被织入 Advice 的候选的点, 在 Spring AOP中, 则可以认为所有方法执行点都是 Joint point. 而在我们上面的例子中, 命案发生在小县城中, 按理说在此县城中的所有人都有可能是嫌疑人.

Pointcut ：男性, 身高约七尺五寸: 我们知道, 所有的方法(joint point) 都可以织入 Advice, 但是我们并不希望在所有方法上都织入 Advice, 而 Pointcut 的作用就是提供一组规则来匹配joinpoint, 给满足规则的 joinpoint 添加 Advice. 同理, 对于县令来说, 他再昏庸, 也知道不能把县城中的所有百姓都抓起来审问, 而是根据凶手是个男性, 身高约七尺五寸, 把符合条件的人抓起来. 在这里 凶手是个男性, 身高约七尺五寸 就是一个修饰谓语, 它限定了凶手的范围, 满足此修饰规则的百姓都是嫌疑人, 都需要抓起来审问.

Advice ：抓过来审问, Advice 是一个动作, 即一段 Java 代码, 这段 Java 代码是作用于 point cut 所限定的那些 Joint point 上的. 同理, 对比到我们的例子中, 抓过来审问 这个动作就是对作用于那些满足 男性, 身高约七尺五寸 的爪哇的小县城里的百姓.

Aspect:：Aspect 是 point cut 与 Advice 的组合, 因此在这里我们就可以类比: “根据老王的线索, 凡是发现有身高七尺五寸的男性, 都要抓过来审问” 这一整个动作可以被认为是一个 Aspect.

最后是一个描述这些概念之间关系的图：



三、其他的一些内容

AOP中的Joinpoint可以有多种类型：构造方法调用，字段的设置和获取，方法的调用，方法的执行，异常的处理执行，类的初始化。也就是说在AOP的概念中我们可以在上面的这些Joinpoint上织入我们自定义的Advice，但是在Spring中却没有实现上面所有的joinpoint，确切的说，Spring只支持方法执行类型的Joinpoint。

Advice 的类型

before advice, 在 join point 前被执行的 advice. 虽然 before advice 是在 join point 前被执行, 但是它并不能够阻止 join point 的执行, 除非发生了异常(即我们在 before advice 代码中, 不能人为地决定是否继续执行 join point 中的代码)

after return advice, 在一个 join point 正常返回后执行的 advice

after throwing advice, 当一个 join point 抛出异常后执行的 advice

after(final) advice, 无论一个 join point 是正常退出还是发生了异常, 都会被执行的 advice.

around advice, 在 join point 前和 joint point 退出后都执行的 advice. 这个是最常用的 advice.

introduction，introduction可以为原有的对象增加新的属性和方法。

在Spring中，通过动态代理和动态字节码技术实现了AOP，这些内容，我们将在以后进行讲解。

————————————————

版权声明：本文为CSDN博主「Jivan2233」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/q982151756/article/details/80513340>

@Aspect

public class ControllerLogAspect {

    private Logger logger = LogManager.getLogger(ControllerLogAspect.class);

    ThreadLocal<Long> startTime = new ThreadLocal<>();

    ThreadLocal<String> uuid = new ThreadLocal<>();

    ThreadLocal<Boolean> serviceLog= new ThreadLocal<>();

@Pointcut("execution(public \* com.lilanz.microservice.cooperativeoffice.controller.\*.\*(..)) && !execution(public \* com.lilanz.microservice.cooperativeoffice.controller.CertificateController.getData(..))")

/\*这个注解包含两部分，PointCut表达式和PointCut签名。表达式是拿来确定切入点的位置的，说白了就是通过一些规则来确定，哪些方法是要增强的，也就是要拦截哪些方法。

@PointCut(...........)括号里面那些就是表达式。这里的execution是其中的一种匹配方式，还有：

execution: 匹配连接点

within: 某个类里面

this: 指定AOP代理类的类型

target:指定目标对象的类型

args: 指定参数的类型

bean:指定特定的bean名称，可以使用通配符（Spring自带的）

@target： 带有指定注解的类型

@args: 指定运行时传的参数带有指定的注解

@within: 匹配使用指定注解的类

@annotation:指定方法所应用的注解

\*/

    public void controllerLog(){

        System.out.println("AOP参数日志切面");

    }

    @Pointcut("execution(public \* com.lilanz.microservice.cooperativeoffice.service.\*.\*(..))")

    public void serviceLog(){

    }

    @Before("controllerLog()")

    public void doBefore(JoinPoint joinPoint) throws Throwable {

        startTime.set(System.currentTimeMillis());

        serviceLog.set(false);

        uuid.set(UUID.randomUUID().toString());

        // 接收到请求，记录请求内容

        ServletRequestAttributes attributes = (ServletRequestAttributes) RequestContextHolder.getRequestAttributes();

        HttpServletRequest request = attributes.getRequest();

        // 通过这获取到方法的所有参数名称的字符串数组

        String url = request.getRequestURL().toString();

        String time = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(new Date());

        Signature signature = joinPoint.getSignature();

        MethodSignature methodSignature = (MethodSignature) signature;

        // 获取所有的参数名

        String[] parameterNames = methodSignature.getParameterNames();

        // 获取所有的参数值

        Object[] args = joinPoint.getArgs();

        // 键列表

        List<String> keyList = new ArrayList<String>();

        // 值列表

        List<Object> valueList = new ArrayList<Object>();

        for(int i = 0;i < args.length; i++) {

            // 防止直接输出request对象值异常、加了判断处理

            if(args[i] instanceof HttpServletRequest) {

                Map<String, String[]> parameterMap = ((HttpServletRequest)args[i]).getParameterMap();

                for(String key:parameterMap.keySet()) {

                    keyList.add(key);

                    valueList.add(parameterMap.get(key));

                }

            }else {

                keyList.add(parameterNames[i]);

                valueList.add(args[i]);

            }

        }

        // 记录下请求内容

        logger.info("Time : " + time +

                "\nUUID : " + uuid.get() +

                "\nURL : " + url +

                "\nParamName : " + JSON.toJSONString(keyList) +

                "\nParamValue : " + JSON.toJSONString(valueList));

    }

    /\*\*

     \* 打印并输出日志（执行后）

     \* 内容为：返回值、执行时间

     \* @param ret 返回值

     \* @throws Throwable

     \*/

    @AfterReturning(returning = "ret", pointcut = "controllerLog()")

    public void doAfterReturning(Object ret) throws Throwable {

    }

    @AfterThrowing(throwing = "ret", pointcut = "serviceLog()")

public void afterThrowingLog(Throwable  ret) {

}

可以通过[预编译](https://baike.baidu.com/item/%E9%A2%84%E7%BC%96%E8%AF%91)方式和运行其动态代理实现在不修改[源代码](https://baike.baidu.com/item/%E6%BA%90%E4%BB%A3%E7%A0%81)的情况下给程序动态统一添加某种特定功能的一种技术。AOP实际是GoF设计模式的延续，设计模式孜孜不倦追求的是调用者和被调用者之间的解耦,提高代码的灵活性和可扩展性，AOP可以说也是这种目标的一种实现。

### 主要功能

日志记录，性能统计，安全控制，事务处理，[异常处理](https://baike.baidu.com/item/%E5%BC%82%E5%B8%B8%E5%A4%84%E7%90%86)等等。

### 主要意图

将日志记录，性能统计，安全控制，事务处理，[异常处理](https://baike.baidu.com/item/%E5%BC%82%E5%B8%B8%E5%A4%84%E7%90%86)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。

**好了现在注解都介绍完了，这里还要提到上面用到的一个类：RequestContextHolder**

比如说，有个需求需要在service中获得request和response，我们一般会（我就是）直接在controller那把request或response作为参数传到service，这就很不美观。后来知道，原来SpringMVC提供了个很强大的类ReqeustContextHolder，通过他你就可以获得request和response什么的。

[复制代码](javascript:void(0);)

//下面两个方法在没有使用JSF的项目中是没有区别的

RequestAttributes requestAttributes = RequestContextHolder.currentRequestAttributes();

// RequestContextHolder.getRequestAttributes();

//从session里面获取对应的值

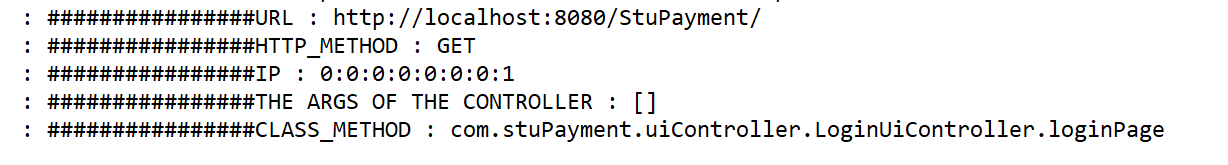
String str = (String) requestAttributes.getAttribute("name",RequestAttributes.SCOPE\_SESSION);

HttpServletRequest request = ((ServletRequestAttributes)requestAttributes).getRequest();

HttpServletResponse response = ((ServletRequestAttributes)requestAttributes).getResponse();

[复制代码](javascript:void(0);)

 好了完成了这个切面的编程后，你就成功把日志功能切入到各个controller中了。看个效果图。



**最后，再记录一下各个不同的advice的拦截顺序的问题。**

情况一，只有一个Aspect类：

　　无异常：@Around（proceed()之前的部分） → @Before → 方法执行 → @Around（proceed()之后的部分） → @After → @AfterReturning

　　有异常：@Around（proceed(之前的部分)） → @Before → 扔异常ing → @After → @AfterThrowing    （大概是因为方法没有跑完抛了异常，没有正确返回所有@Around的proceed()之后的部分和@AfterReturning两个注解的加强没有能够织入）

情况二，同一个方法有多个@Aspect类拦截：

　　单个Aspect肯定是和只有一个Aspect的时候的情况是一样的，但不同的Aspect里面的advice的顺序呢？？答案是不一定，像是线程一样，没有谁先谁后，除非你给他们分配优先级，同样地，在这里你也可以为@Aspect分配优先级，这样就可以决定谁先谁后了。

优先级有两种方式：

* 实现org.springframework.core.Ordered接口，实现它的getOrder()方法
* 给aspect添加@Order注解，该注解全称为：org.springframework.core.annotation.Order

不管是哪种，都是order的值越小越先执行：

[复制代码](javascript:void(0);)

@Order(5)

@Component

@Aspect

public class Aspect1 {

// ...

}

@Order(6)

@Component

@Aspect

public class Aspect2 {

// ...

}

[复制代码](javascript:void(0);)

这样Aspect1就永远比Aspect2先执行了。

**注意点：**

* 如果在同一个 aspect 类中，针对同一个 pointcut，定义了两个相同的 advice(比如，定义了两个 @Before)，那么这两个 advice 的执行顺序是无法确定的，哪怕你给这两个 advice 添加了 @Order 这个注解，也不行。这点切记。
* 对于@Around这个advice，不管它有没有返回值，但是必须要方法内部，调用一下 pjp.proceed();否则，Controller 中的接口将没有机会被执行，从而也导致了 @Before这个advice不会被触发