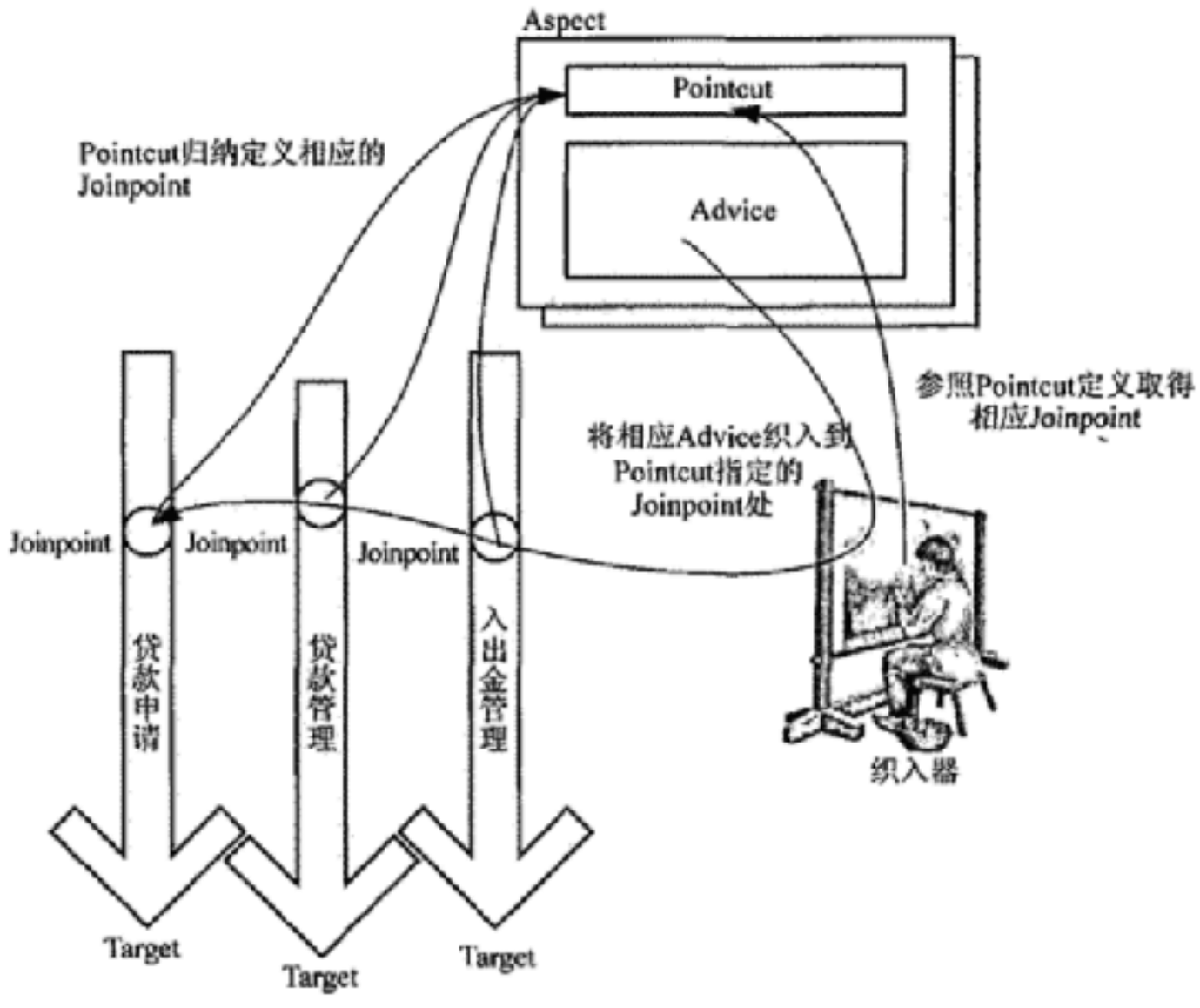
AOP

more than spring

AOP公民

OOP用来解决模块化问题，但对于系统性需求无法很好的通用化。
AOP将系统性需求(日志记录/权限管控等)独立出来，弥补OOP的不足。



AOP公民

JoinPoint

决定在哪些点进行织入操作。
方法调用/方法执行/字段设置/
字段读取/构造方法执行/
异常处理执行/类初始化(static 代码块)

Aspect

对系统中横切关注点进行模块化封装的概念实体。可以包括多个pointcut及advice定义。

PointCut

JointPoint的表述方式。需要参照pointcut描述的jointpoint信息才能确定往系统哪些jointpoint上
执行织入逻辑。

Advice

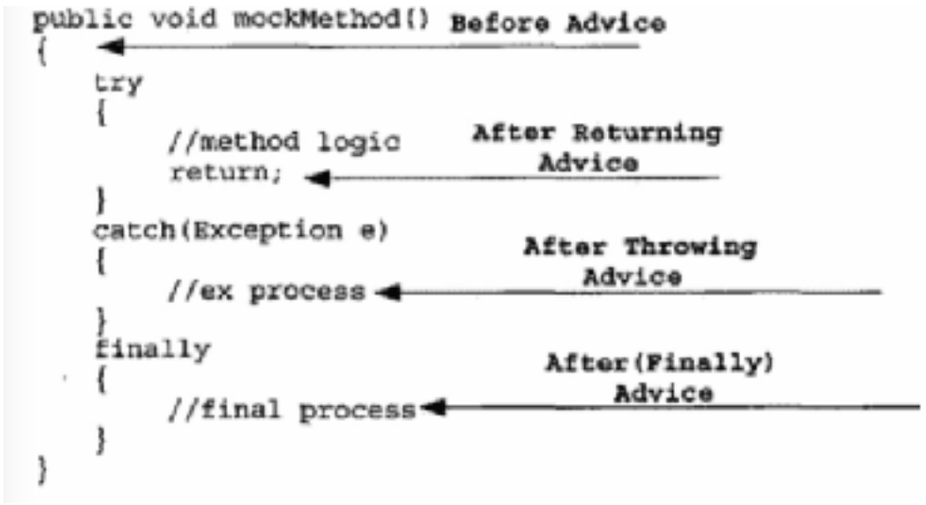
单一横切点关注的载体。
代表将会植入到JointPoint处的横切逻辑。

Weaving

织入器，将aspect跟目标对象链接起来。可以在运行期间/编译期间/加载期间执行。

AOP公民

Advice



BeforeAdvice
在jointpoint之前执行

AfterReturningAdvice
jointpoint处逻辑正常完成之后执行

AfterThrowingAdvice
jointpoint处抓住异常之后执行

AfterFinallyAdvice
jointpoint处必然要执行

范围最小化原则，尽量选用最小范围的advice。比如需要在方法正常返回后去更新缓存，则应该优先选用afterReturningAdvice。比如去取得系统的一些公共资源则优先选用beforeAdvice。AroundAdvice多用于执行一些系统的公用逻辑。

AroundAdvice
对其附加上的jointpoint进行包裹，可以在之前/之后执行相关逻辑，甚至中断或者忽略或者重复jointpoint处原来程序的执行流程。所以有些功能跟其他advice重复。

1：在线程安全前提下，在jp前后共享一些状态，比如计时
2：Retry的场景，需要多次执行process

@PointCut

Part1： Expression

用表达式指明匹配规则

支持 || && ！ 逻辑运算

Part2： Signature

expression的载体/标识符，返回值类型必须为void

在expression的复合定义中用signature来取代重复的表达式定义

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

方法执行jp(修饰符模式？ 返回值模式 类路径模式？方法名模式 参数模式 异常模式？)

public修饰的可以在其他aspect中直接引用
private修饰的只能在本aspect中使用

*** 代表任意返回值类型**

. 代表当前包
.. 当前包和子包

*** 可以匹配相邻的多个字符**

*** 代表一个参数**
.. 代表0个或多个参数

PointCut Designator

execution
匹配拥有指定方法签名的jp
（方法级别）

within
匹配指定类型下的jp
（类级别-类下所有方法）

this
匹配目标对象的代理对象下的jp
（类级别-类下所有方法）

target
匹配目标对象下的jp
（类级别-类下所有方法）

args
匹配指定参数类型的jp
（参数级别-所有方法）

bean
匹配指定bean下的jp
（实例级别）

@within
匹配标注了某个注解的对象

@target
匹配标注了某个注解的目标对象

@args
匹配当前方法参数类型

@annotation
匹配所有对象方法级是否标注指定注解

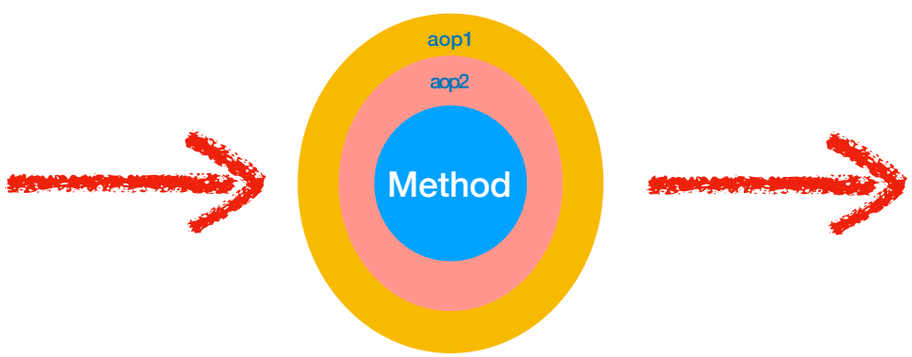
@PointCut

```java
@Component
@Aspect
public class AspectE {

    @Pointcut("execution(* transfer(..))") // the pointcut expression
    private void anyOldTransfer() {} // the pointcut signature

    @Pointcut("execution(public * *(..))")
    private void anyPublicOperation() {}

    @Pointcut("within(cn.deepclue.bolt.search..*)")
    private void inSearch() {}

    @Pointcut("anyPublicOperation() && inSearch()")
    private void searchOperation() {}

    @Pointcut("execution(* set*(..))")
    public void setMethod() {}

    @Pointcut("execution(* cn.deepclue.service.AccountService.*(..))")
    public void anyAccountServiceMethod() {}

    @Pointcut("execution(* cn.deepclue.service.*.*(..))")
    public void anyServiceMethod() {}

    @Pointcut("execution(* cn.deepclue.service..*.*(..))")
    public void anyServiceAndInnerServiceMethod() {}

    @Pointcut("this(cn.deepclue.service.BuyService)")
    public void anyBuyServiceImplMethod() {}

    @Pointcut("target(cn.deepclue.service.BuyService)")
    public void anyTargetBuyServiceImplMethod() {}

    @Pointcut("args(java.io.Serializable)")
    public void anySingleSerializableArgMethod() {}

    @Pointcut("bean(BuyService)")
    public void anyBuyServiceMethod() {}
```

```java
//@Before("cn.deepclue.bolt.SystemArchitecture.dataAccessOperation()")
@Before("execution(* cn.deepclue.bolt.dao.*.*(..))")
public void doAccessCheck() {
    //...
}

@AfterReturning(pointcut = "cn.deepclue.bolt.SystemArchitecture.dataAccessOperation()", returning = "retVal")
public void doResultProcess(Object retVal) {
    //...
}

@AfterThrowing(pointcut = "cn.deepclue.bolt.SystemArchitecture.dataAccessOperation()", throwing = "ex")
public void doRecoveryActions(/*DataAccess*/Exception ex) {
    //...
}

@After("cn.deepclue.bolt.SystemArchitecture.dataAccessOperation()")
public void doReleaseLock() {
    //...
}

@Around("execution(List<Account> find*(..)) && cn.deepclue.bolt.SystemArchitecture.inDataAccessLayer() && args(accountHolderNamePattern)")
public Object preProcessQueryPattern(ProceedingJoinPoint pjp, String accountHolderNamePattern) throws Throwable {

    String newPattern = accountHolderNamePattern.concat("vvv");

    return pjp.proceed(new Object[] {newPattern});
}
```

@advisor

```java
//@Before("cn.deepclue.bolt.SystemArchitecture.dataAccessOperation()")
@Before("execution(* cn.deepclue.bolt.dao.*.*(..))")
public void doAccessCheck() {
    //...
}

@AfterReturning(pointcut = "cn.deepclue.bolt.SystemArchitecture.dataAccessOperation()", returning = "retVal")
public void doResultProcess(Object retVal) {
    //...
}

@AfterThrowing(pointcut = "cn.deepclue.bolt.SystemArchitecture.dataAccessOperation()", throwing = "ex")
public void doRecoveryActions(/*DataAccess*/Exception ex) {
    //...
}

@After("cn.deepclue.bolt.SystemArchitecture.dataAccessOperation()")
public void doReleaseLock() {
    //...
}

@Around("execution(List<Account> find*(..)) && cn.deepclue.bolt.SystemArchitecture.inDataAccessLayer() && args(accountHolderNamePattern)")
public Object preProcessQueryPattern(ProceedingJoinPoint pjp, String accountHolderNamePattern) throws Throwable {

    String newPattern = accountHolderNamePattern.concat("vvv");

    return pjp.proceed(new Object[] {newPattern});
}
```



Order越小优先级越高
0以下order为spring内部使用

思考

this 和 target的区别

args中的参数限定 vs execution中参数限定

一个良好的pointcut的写法

spring aop vs Full AspectJ

Java平台的aop实现

动态代理

对接口生成代理对象
在代理对象中织入逻辑
因为使用反射机制
相对静态代理性能要差一些

自定义类加载器

在classloader加载class到jvm期间
使用自定义的classloader读取外部织入逻辑
将织入逻辑放入class中，将改动后的class
交给jvm运行

动态字节码增强

在需要织入横切逻辑的类运行期间
通过动态字节码增强生成相应子类
在子类中织入横切逻辑并运行子类

AOL扩展

定义语言

spring aop 采用动态代理和动态字节码增强技术来实现
如果对象有接口则优先采用动态代理技术，否则采用动态字节码增强技术
可是设置proxy-target-class=true强制使用字节码增强技术

代理模式 vs 字节码增强

代理模式

隐藏委托类的实现，将代理类跟委托类解耦合
在不改变委托类前提下在代理类中添加适配逻辑
缺陷：依赖于接口

静态代理

优点：性能高
缺点：接口继承/接口中方法很多情形下，代理类代码冗余

动态代理

缺点：性能低于静态代理
优点：通过统一扩展的InvokeHandler类去掉了冗余代码

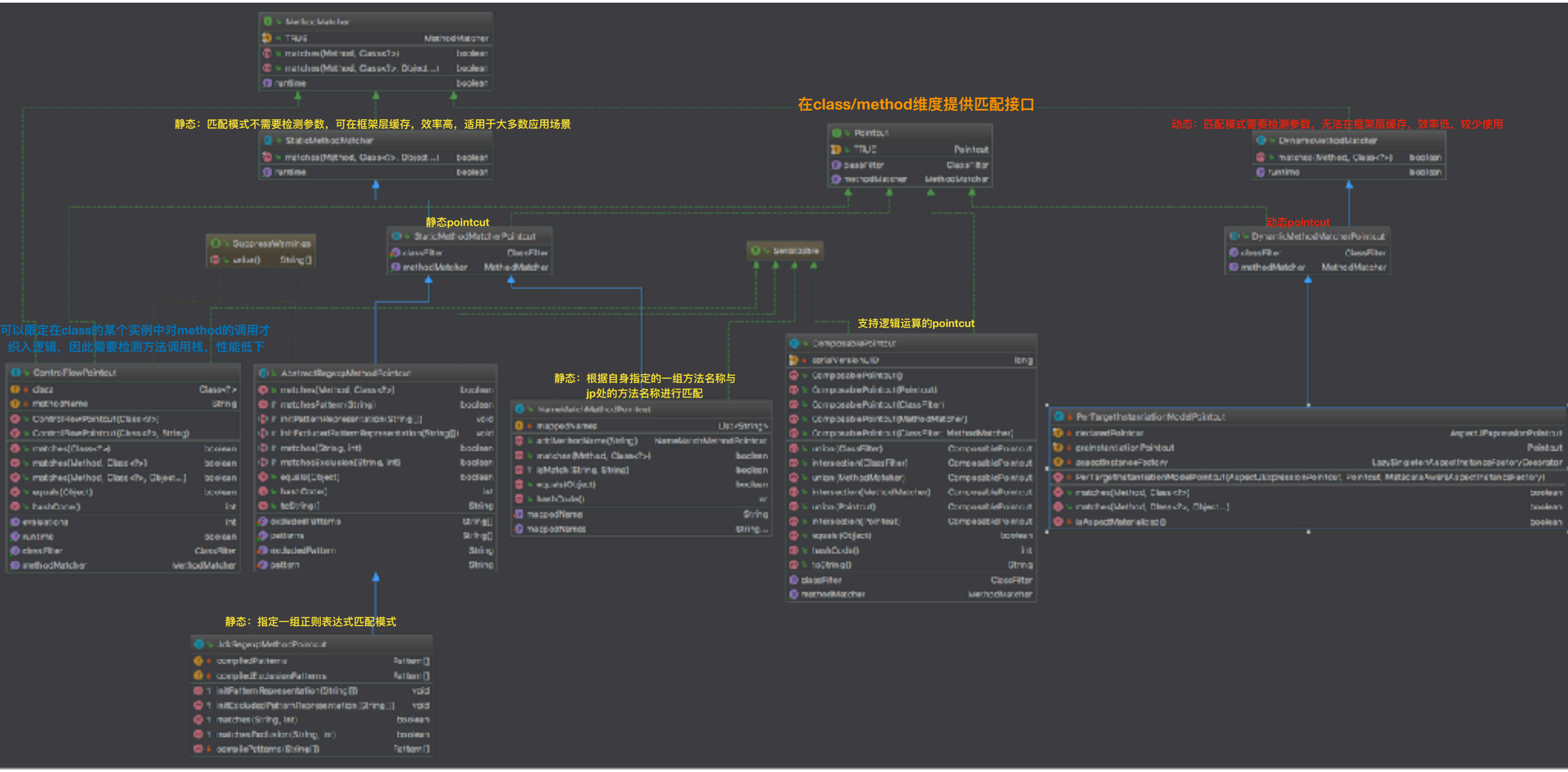
字节码增强

不依赖于接口，即使类没有实现任何接口也可以扩展
缺陷：需要借助CGLIB之类的动态字节码生成库

假设接口A有10个方法，代理类和具体实现类都必须实现这10个方法
如果想在每个方法调用前增加一段逻辑，则必须调整代理类的10个方法
而实际添加的逻辑在大多数情形下都是重复逻辑

假设ISubject接口和IRequest接口都继承了IMethod中的request方法
如果在request方法执行前后加入逻辑，则必须扩展两个新类实现以上两个接口
如果有更多接口，则需要扩展更多类

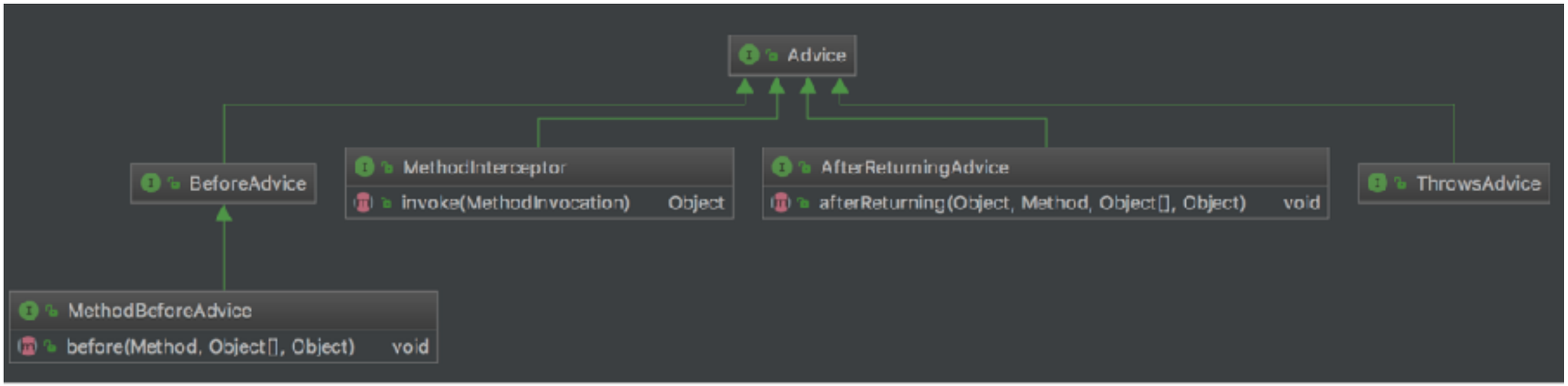
Spring Pointcut impl



Spring Advice impl

perClass类型

该类型的实例可以在目标类的所有实例之间共享
通常只提供方法拦截功能，不会为目标对象类保存任何状态



BeforeAdvice

获取一些初始化资源

ThrowAdvice

对系统中他特定的异常类型进行监控，以统一的方式处理异常

AfterReturningAdvice

可以访问当前jp的方法/返回值/参数/目标对象
但是无法对返回值进行进一步处理

AroundAdvice

可以在jp前后加入逻辑，可以控制方法是否执行

系统性能检测
附加行为添加

安全验证/检查
日志记录

Spring Advice impl

perInstance类型

该类型的实例不会再目标类所有对象之间共享
为不同实例保存它们各自的状态及逻辑

Spring aop中 Introduction是唯一一种perInstance类型的advice。
Introduction可以在不改变目标类的前提下，为目标类增加新特性或行为。
spring中增加新特性必须声明相应接口并实现，然后通过拦截器将新的接口定义
及实现类中的逻辑附加到目标对象上，之后目标对象(确切的说是目标对象的代理对象)
就有了新的行为，这个拦截器就是IntroductionIntercepter