Spring AOP

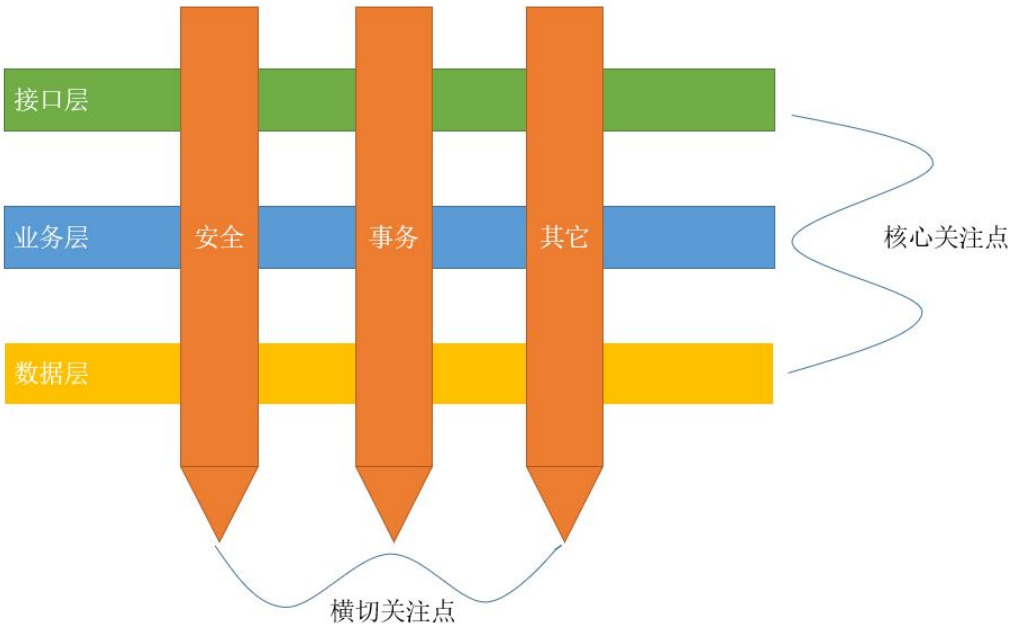
> ## 本节目标
>
> 1.理解AOP的概念 2.了解AOP的基础知识 3.掌握AOP的基础编程

1. AOP是什么?

1.1 AOP基本概念

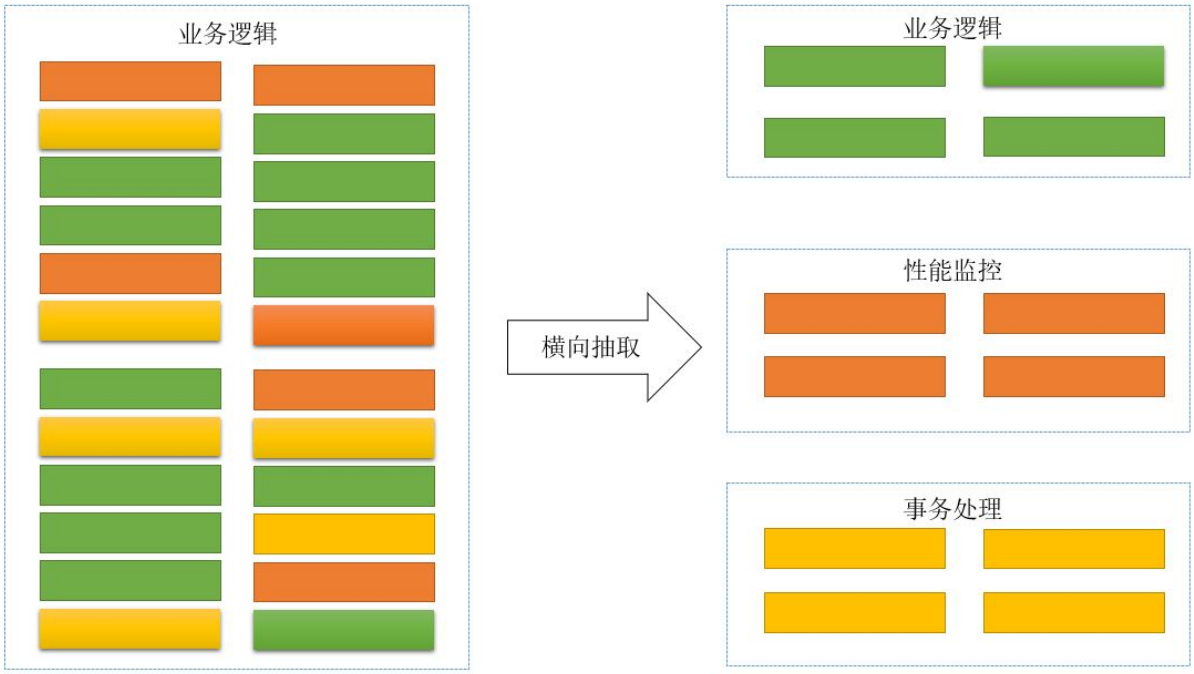
AOP是`Aspect Oriented Programing`的简称，中文最初翻译为"面向方面编程"，通常我们更喜欢称之为"面向切面编程"。

OOP引入**封装、继承和多态性**等概念来建立一种对象层次结构，用以模拟公共行为的一个集合。当我们需要为分散的对象引入公共行为的时候，OOP则显得无能为力。也就是说，OOP允许你定义从上到下的关系，但并不适合定义从左到右的关系。例如日志功能，日志代码往往水平地散布在所有对象层次中，而与它所散布到的对象的核心功能毫无关系。对于其它类型的代码，如安全性、异常处理也是如此。这种散布在各处的代码被称为**横切（cross-cutting）代码**，在OOP设计中，它导致了大量代码的重复，而不利于各个模块的重用，因此引入了AOP的编程理念，可以说AOP是OOP的补充和完善。



AOP把软件系统分为两个部分：**核心关注点**和**横切关注点。**业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，它们经常发生在核心关注点的多处，而各处都基本相似。比如权限认证、日志、事务处理。AOP的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。

AOP的核心思想就是**"将应用程序中的商业逻辑同对其提供支持的通用服务进行分离。"**



实现AOP的技术，主要分为两大类：

- 一是采用动态代理技术，利用截取消息的方式，对该消息进行装饰，以取代原有对象行为的执行
- 二是采用静态织入的方式，引入特定的语法创建“切面”，从而使得编译器可以在编译期间织入有关“切面”的代码

1.1 AOP思想演变

1.1.1 演变过程-1

设置一个如下场景，支付程序：

```
public interface PayService {
    void pay();
}
```

```
public class AliPayService implements PayService {

    @Override
    public void pay() {
        //支付业务逻辑
    }
}
```

```
public class WeiXinPayService implements PayService {

    @Override
    public void pay() {
        //支付业务逻辑
    }
}
```

思考如下问题：

- 需要对支付方法添加日志记录，安全检查，时间统计功能怎么做？

1.1.2 演变过程-2

解决场景-1的问题，程序可能如下：

```
public class AliPayService implements PayService {

    @Override
    public void pay() {
        //1.安全检查
        //2.日志记录
        //3.时间记录开始

        //支付业务逻辑

        //4.时间记录结束
    }
}
```

```
public class WeiXinPayService implements PayService {

    @Override
    public void pay() {
        //1.安全检查
        //2.日志记录
        //3.时间记录开始

        //支付业务逻辑

        //4.时间记录结束
    }
}
```

上述改造之后的代码有大量的重复，自然想到的办法就是把公共部分提取出模板或者公共的父类方法。然后让调用的类来使用公共模板或者继承公共的父类。

1.1.3 演变过程-3

提取公共模板之后，可能的程序如下：

```
public class LogComponent {
    //日志记录
}

public class TimeComponent {
    //时间统计开始
    //时间统计结束
}
```

```
public class SecurityComponent {
    //安全检查
}

public class AliPayService implements PayService {

    @Override
    public void pay() {
        //1.使用SecurityComponent安全检查
        //2.使用LogComponent日志记录
        //3.使用TimeComponent时间记录开始

        //支付业务逻辑

        //4.使用TimeComponent时间记录结束
    }
}
```

```
public class WeiXinPayService implements PayService {

    @Override
    public void pay() {
        //1.使用SecurityComponent安全检查
        //2.使用LogComponent日志记录
        //3.使用TimeComponent时间记录开始

        //支付业务逻辑

        //4.使用TimeComponent时间记录结束
    }
}
```

1.1.4 演变过程-4

- 仍然存在问题：

我们会发现，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很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”。

- 改进的解决方案

采用装饰器模式或者代理模式来实现。

- 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
- 代理模式定义：为其它对象提供一种代理以控制对这个对象的访问

静态代理实现：

```
public class PayServiceProxy implements PayService {
    private PayService payService;
    public PayServiceProxy(PayService payService) {
```

```
        this.payService = payService;
    }
    @Override
    public void pay() {
        //1.使用SecurityComponent安全检查
        //2.使用LogComponent日志记录
        //3.使用TimeComponent时间记录开始
        //支付业务逻辑
        this.payService.pay();
        //4.使用TimeComponent时间记录结束
    }
}
```

```
PayService payService = new AliaPayService();
PayServiceProxy proxy = new PayServiceProxy(payService);
proxy.pay();
```

1.1.5 演变过程-5

JDK动态代理解决方案（比较常用，**面试考点**）

```
public class PayServiceJDKInvocationHandler implements InvocationHandler {

    //目标对象即就是被代理对象
    private Object target;

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

    //proxy代理对象
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        //1.安全检查
        //2.记录日志
        //3.时间统计开始
        Object retVal = method.invoke(target, args);
        //4.时间统计结束
        return retVal;
    }

    public static void main(String[] args) {
        PayService target=  new AlipayService();
        InvocationHandler handler =
            new PayServiceJDKInvocationHandler(target);
        PayService proxy = (PayService) Proxy.newProxyInstance(
                target.getClass().getClassLoader(),
                new Class[]{PayService.class},
                handler
        );
        proxy.pay();
    }
```

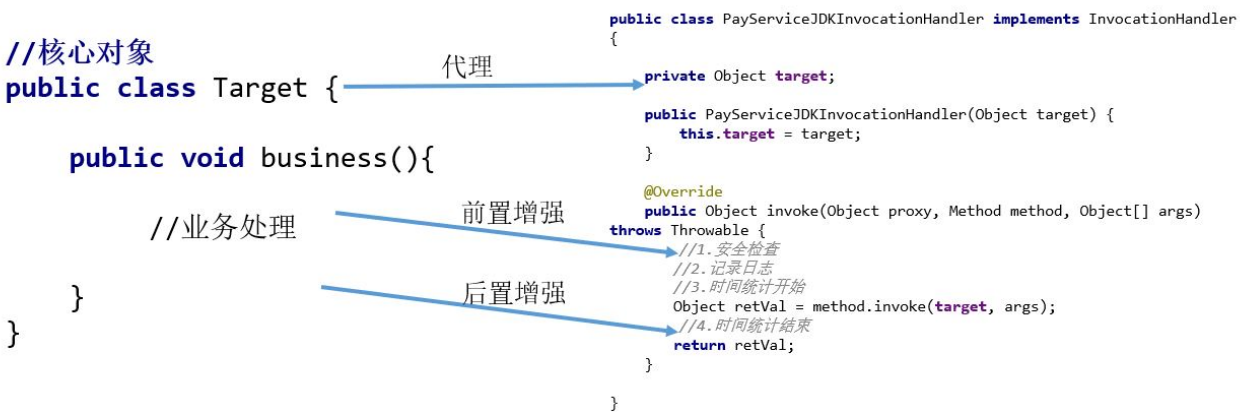
```
}
```

1.1.6 演变过程-6

CGLIB动态代理的解决方案：

```
public class PayServiceCGLIBInterceptor implements MethodInterceptor {

    private Object target;

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

    @Override
    public Object intercept(Object o, Method method, Object[] args, MethodProxy
methodProxy) throws Throwable {
        //1.安全检查
        //2.记录日志
        //3.时间统计开始
        Object retVal = methodProxy.invoke(target, args);
        //4.时间统计结束
        return retVal;
    }

    public static void main(String[] args) {
        PayService target=  new AlipayService();
        PayService proxy= (PayService)Enhancer.create(target.getClass(),new
PayServiceCGLIBInterceptor(target));
        proxy.pay();
    }
}
```

1.1.7 演变过程-7



1.1.8 演变过程-8

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（Cross Cutting Concern）”，因为它跨越了给定编程模型中的典型职责界限。 如果使用过于密切关注点的代码，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。

2.AOP能干什么?

切面关注通用的逻辑，下面是在项目中常用到切面的场景：

- Authentication 权限
- Caching 缓存
- Context passing 内容传递
- Error handling 错误处理
- Lazy loading　懒加载
- Debugging　调试
- logging，tracing， profiling and monitoring　记录跟踪，优化，校准
- Performance optimization　性能优化
- Persistence 持久化
- Resource pooling 资源池
- Synchronization　同步
- Transactions 事务

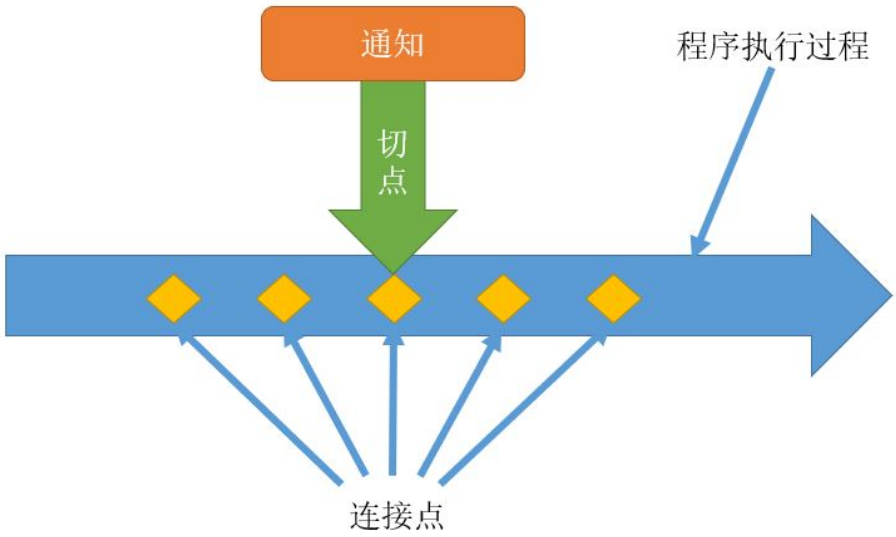
AOP的带来的好处：

- 降低模块耦合度
- 使系统容易扩展
- 延迟设计决定：使用AOP，设计师可以推迟为将来的需求作决定，因为需求作为独立的方面很容易实现
- 更好的代码复用性

3.AOP术语

和大多数技术一样，AOP已经形成了自己的术语。描述切面的常用术语有通知（advice）,切点（pointcut）和连接点（joinpoint）。

下图展示了AOP常用的几个概念是如何关联在一起的。



在一个或者多个连接点上，可以把切面的功能（通知）织入到程序的执行过程中

遗憾的是，大多数描述AOP功能的术语很不直观，尽管如此，这些术语已经成为AOP行话的组成部分，因此我们还是需要了解这些术语。

3.1 通知（Advice）

切面也是有目标的 ——它必须完成的工作。在AOP术语中，**切面的工作被称之为通知。**

通知：定义了切面是什么，何时使用，其描述了切面要完成的工作，还解决何时执行这个工作的问题。

Spring切面可以应用5种类型的通知：

- 前置通知(Before): 在目标方法被调用之前调用通知功能；
- 后置通知(After)： 在目标方法完成之后调用通知，此时不会关心方法的输出是什么；
- 返回通知(After-returning)：在目标方法成功执行之后调用通知；
- 异常通知(After-throwing): 在目标方法抛出异常后通知调用；
- 环绕通知(Around)：通知包裹了被通知的方法，在被通知的方法通知之前和调用之后执行自定义的行为。

3.2 连接点（Join Point）

连接点：是在应用执行过程中能够插入切面的一个点，这个点可以是方法调用时，抛出异常时，甚至修改字段时。切面代码可以利用这些点插入到应用的正常流程之中，并添加新的行为。

3.3 切点（Pointcut）

切点：是缩小切面所通知的连接点的范围。如果说通知定义了切面**是什么**和**何时**的话，那么切点就定义了**何处。**

切点的定义会匹配通知所要织入的一个或者多个连接点。我们通常会使用明确的类和方法名称，或者利用正则表达式定义所匹配的类和方法名来指定这些切点。

3.4 切面（Aspect）

切面：是通知和切点的结合。通知和切点共同定义了切面的全部内容————它是什么，在何时，在何处完成其功能。

3.5 织入（Weaving）

织入是把切面应用到目标对象并创建新的代理对象的过程，切面在指定的连接点被织入到目标对象中。

在目标对象的生命周期里有多个点可以进行织入：

- **编译期**：

  切面在目标类编译时被织入。这种方式需要特殊的编译器。AspectJ的织入编译器就是以这种方式织入切面的。

- **类加载器**：

  切面在目标类加载到JVM时被织入。这种方式需要特殊的类加载器（ClassLoader）,它可以在目标类被引入应用之前增强该目标类的字节码。AspectJ5的加载时织入（load-time weaving. LTW）就支持以这种方式织入切面。

- **运行期**：

  切面在应用运行的某一时刻被织入。一般情况下，在织入切面时，AOP容器会为目标对象动态创建一个代理对象。SpringAOP就是以这种方式织入切面的。

3.7 目标对象（Target Object）

被一个或者多个切面所通知（Advice）的对象，也把它叫做被通知的对象（Advised）。既然SpringAOP是通过运行时代理实现的，那么这个对象永远是一个被代理（Proxied）对象。

4.Spring对AOP的支持

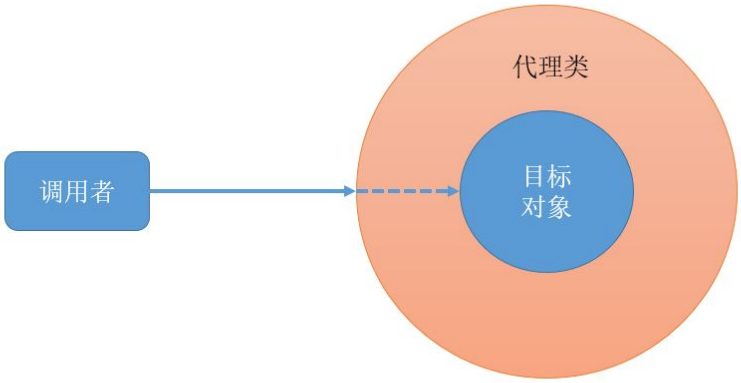
4.1 SpringAOP支持方式

- 基于代理的经典AOP支持
- 纯POJO切面
- @AspectJ注解驱动切面
- 注入式AspectJ切面（适用于Spring各个版本）

前三种都是SpringAOP的实现变体，SpringAOP构建在**动态代理**基础上，因此Spring对AOP的支持局限于方法拦截。Spring经典的AOP直接使用ProxyFactoryBean，这是比较笨重和繁琐的方式。现在Spring提供了更简洁和干净的面向切面编程方式，引入简单的**声明式**和**基于注解**的AOP。

4.2 SpringAOP的特点

- Spring通知是使用Java编写
- Spring在运行时通知对象
- Spring只支持方法级别的连接点



Spring的切面由包裹了目标对象的代理类实现。代理类处理方法的调用，执行额外的切面逻辑，并调用目标方法。

通过在代理类中包裹切面，Spring在运行期把切面织入到Spring管理的Bean中。如上图，代理类封装了目标类，并且拦截被通知方法的调用，再把调用转发给真正的目标Bean。当代理拦截到方法调用时，在调用目标Bean方法过程中，会执行切面逻辑。

4.3 使用注解创建切面

4.3.1 演示示例

- 添加依赖

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aspects</artifactId>
</dependency>
```

- 定义切面

```
package com.bittech.springcore.aop.aspect;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;

@Aspect
//使用注解的方式定义Bean
@Component
public class PayAspect {
```

```
    /**
     * 支付连接点
     */
    @Pointcut(value = "execution(* com.bittech.aop.service.PayService.pay())")
    public void payPointCut() {

    }

    @Before(value = "payPointCut()")
    public void security() {
        System.out.println("Before >> Security check!!");
    }

    @Before(value = "payPointCut()")
    public void log() {
        System.out.println("Before >> Log record .");
    }

    @Before(value = "payPointCut()")
    public void timeStart() {
        System.out.println("Before >> Time record start.");
    }

    @After(value = "payPointCut()")
    public void timeEnd() {
        System.out.println("After >> Time record end.");
    }

    @AfterReturning(value = "payPointCut()")
    public void payReturn() {
        System.out.println("AfterReturning >> Pay Returning.");
    }

    @AfterThrowing(value = "payPointCut()")
    public void payThrowing() {
        System.out.println("AfterThrowing >> Refund !");
    }

    @Around(value = "payPointCut()")
    public Object payAround(ProceedingJoinPoint joinPoint) {
        Object retVal = null;
        System.out.println("Around A>> Pay around .");
        try {
            retVal = joinPoint.proceed();
        } catch (Throwable throwable) {
            throwable.printStackTrace();
        }
        System.out.println("Around B>> Pay around .");
        return retVal;
    }
}
```

- XML配置使用切面

在Spring的XML中装配Bean的话，需要使用Spring aop名命空间中的`<aop:aspectj-autoproxy/>`元素。下面是XML配置展现如何完成该功能：

```
<?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:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd">

    <!--启动Aspectj自动代理-->
    <aop:aspectj-autoproxy/>

    <!--扫描Bean的定义-->
    <context:component-scan base-package="com.bittech.springcore.aop"/>

</beans>
```

```
public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("application-
aop.xml");
    PayService aliPayService = context.getBean(PayService.class);
    aliPayService.pay();
 }
```

- JavaConfig配置使用切面

```
@EnableAspectJAutoProxy
@Configuration
@ComponentScan(basePackages = {"com.bittech.aop"})
public class AopApplication {

    @Bean
    public PayService payService() {
        return new AliPayService();
    }

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AopApplication.class);
        PayService payService = context.getBean(PayService.class);
        payService.pay();
    }
}
```

- 程序运行结果

```
Around A>> Pay around .
Before >> Log record .
Before >> Security check!!
Before >> Time record start.
Ali zhifubao pay.
Around B>> Pay around .
After >> Time record end.
AfterReturning >> Pay Returning.
```

4.3.2 切点表达式

Spring借助AspectJ的切点表达式语言来定义Spring切面

- arg() : 限制连接点匹配参数为指定类型的执行方法
- @args(): 限制连接点匹配参数由指定注解标注的执行方法
- **execution() : 用于匹配时连接点的执行方法**
- this() : 限制连接点匹配AOP代理的bean引用为指定类型的类
- target : 限制连接点匹配目标对象为指定类型的类
- @target : 限制连接点匹配特定的执行对象，这些对象对应的类要具有指定类型的注解
- within() : 限制连接点匹配指定的类型
- @within() : 限制连接点匹配指定注解所标注的类型（当使用Spring AOP时，方法定义在由指定的注解所标注的类里）
- @annotation() : 限定匹配带有指定注解的连接点

4.3.3 切面注解

Spring使用AspectJ注解来声明通知方法

- @After : 通知方法会在目标方法返回或者抛出异常后调用
- @AfterReturning : 通知方法会在目标方法返回后调用
- @AfterThrowing : 通知方法会在目标方法抛出异常后调用
- @Around : 通知方法会将目标方法封装起来
- @Before : 通知方法会在目标方法调用之前执行

4.4 在XML中声明切面

4.4.1 演示示例

我们把PayAspect中的切面相关的注解都去掉，在XML的中的配置：

```
<aop:config>
    <aop:pointcut id="pay" expression="execution(*
com.bittech.aop.service.PayService.pay())"/>
    <aop:aspect ref="payAspect">
        <aop:before method="security" pointcut-ref="pay"/>
        <aop:before method="log" pointcut-ref="pay"/>
        <aop:before method="timeStart" pointcut-ref="pay"/>
        <aop:after method="timeEnd" pointcut-ref="pay"/>
        <aop:after-returning method="payReturn" pointcut-ref="pay"/>
        <aop:around method="payAround" pointcut-ref="pay"/>
        <aop:after-throwing method="payThrowing" pointcut-ref="pay"/>
    </aop:aspect>
</aop:config>
```

```
public class AopXmlApplication {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("spring-
aop.xml");
        PayService payService = context.getBean(PayService.class);
        payService.pay();
    }
}
```

4.4.2 配置元素

Spring的AOP配置元素能够以**非侵入性**的方式声明切面

- aop:aspectj-autoproxy : **启用@AspectJ注解驱动的切面**
- aop:config : 顶层的AOP配置元素。大多数的aop:*元素必须包含在aop:config元素内
- aop:pointcut : 定义一个切点
- aop:aspect : 定义一个切面
- aop:advisor : 定义AOP通知器
- aop:after : 定义AOP后置通知
- aop:after-returning : 定义AOP返回通知
- aop:after-throwing : 定义AOP异常通知
- aop:around : 定义AOP环绕通知
- aop:before : 定义一个AOP前置通知

4.5 AOP开发步骤

- 抽取横切：分析需求，提取出横切关注点
- 实现横切：独立实现横切关注点
- 切面组装：配置横切关注点与核心关注点的关系

4.6 AOP设计考虑

- **是否拦截字段**

SpringAOP仅支持方法拦截。通过AOP来修改属性不是一个好的做法，因为属性通常应该在类的内部进行访问，它体现了类的封装性，如果拦截字段并修改就会破坏这种封装性。

- **创建更多的切面**

或许我们会认为AOP特别强大而到处使用，以至于一个方法被调用时，都难以说清楚究竟执行了哪些代码，这是很可怕的，也是典型的**过度设计**。

通常如下情况才会考虑设计成切面：

（1）大部分模块都需要使用的通用功能，包括系统级或者模块级别。

（2）预计目前的实现会在今后进行功能扩展的可能性很大的地方。

- **正交性**

如果多个切面相互影响，造成一些无法预测的结果，该怎么办？

如果多个切入点的功能实现叠加，甚至造成错误，又该怎么办？

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，请保持设计的**切面的独立性**，**切面功能实现的独立性**，**切面执行顺序的独立性**（即：不依赖于多个方面的执行先后顺）。

- **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。

但是需要注意不要在细粒度的对象上使用AOP，比如对某个实体对象。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进行反射处理，严重影响性能。

总结

知识块	知识点	分类	掌握程度
AOP思想	1.AOP概念和思想	理解性	了解
AOP实现	1.AOP各种不同的实现（静态，动态代理）	实战型	掌握
Spring AOP	1. SpringAOP的使用 2. AOP设计	理解型	了解