AOP（Aspect-Oriented Programming）面向切面编程

需求

spring中当大量的Bean需要使用进行统一的处理，例如日志输出等，为了降低代码的冗余，同时方便日志代码去方便维护。AOP即登上了历史的舞台。其实spring使用过程中直面切面技术的机会很少，但是spring底层的运行大量依附着AOP。

前提

aop的本质就是一种动态代理，将目标对象托付给代理对象让调度。所以学习aop前熟悉动态代理模式方便理解AOP。

aop处理过程

aop术语

- 切面(Aspect): 横切关注点(跨越应用程序多个模块的功能)被模块化的特殊对象
- 通知(Advice): 切面必须要完成的工作
- 目标(Target): 被通知的对象
- 代理(Proxy): 向目标对象应用通知之后创建的对象
- 连接点（Joinpoint）：程序执行的某个特定位置：如类某个方法调用前、调用后、方法抛出异常后等。连接点由两个信息确定：方法表示的程序执行点；相对点表示的方位。例如 ArithmethicCalculator#add() 方法执行前的连接点，执行点为 ArithmethicCalculator#add()； 方位为该方法执行前的位置
- 切点（pointcut）：每个类都拥有多个连接点：例如 ArithmethicCalculator 的所有方法实际上都是连接点，即连接点是程序类中客观存在的事务。AOP 通过切点定位到特定的连接点。类比：连接点相当于数据库中的记录，切点相当于查询条件。切点和连接点不是一对一的关系，一个切点匹配多个连接点，切点通过 org.springframework.aop.Pointcut 接口进行描述，它使用类和方法作为连接点的查询条件。

环境准备

AspectJ：Java 社区里最完整最流行的 AOP 框架. 在 Spring2.0 以上版本中, 可以使用基于 AspectJ 注解或基于 XML 配置的 AOP

要在 Spring 应用中使用 AspectJ 注解, 必须在 classpath 下包含4个类库: aopalliance.jar[spring-aop依赖jar]、aspectj.weaver.jar[aspectj的jar] 和 spring-aspects.jar[提供对spring对AspectJ的支持]，spring-aop.jar[Spring 的AOP 特性时所需的类和源码级元数据支持]。

注解版的AOP

1.配置xml文件中启动AOP：aop:aspectj-autoproxy</aop:aspectj-autoproxy> 2.在 AspectJ 注解中, 切面只是一个带有 @Aspect 注解的 Java 类. 3.通知是标注有某种注解的简单的 Java 方法。

- @Before: 前置通知, 在方法执行之前执行
- @After: 后置通知, 在方法执行之后执行
- @AfterRunning: 返回通知, 在方法返回结果之后执行
- @AfterThrowing: 异常通知, 在方法抛出异常之后
- @Around: 环绕通知, 围绕着方法执行

4.利用方法签名编写 AspectJ 切入点表达式

- execution * com.atguigu.spring.ArithmeticCalculator.*(..): 匹配 ArithmeticCalculator 中声明的所有方法,第一个 * 代表任意修饰符及任意返回值. 第二个 * 代表任意方法. .. 匹配任意数量的参数. 若目标类与接口与该切面在同一个包中, 可以省略包名.
- execution public * ArithmeticCalculator.*(..): 匹配 ArithmeticCalculator 接口的所有公有方法.
- execution public double ArithmeticCalculator.*(..): 匹配 ArithmeticCalculator 中返回 double 类型数值的方法
- execution public double ArithmeticCalculator.*(double, ..): 匹配第一个参数为 double 类型的方法, .. 匹配任意数量任意类型的参数
- execution public double ArithmeticCalculator.*(double, double): 匹配参数类型为 double, double 类型的方法.

```
//指定切面的执行顺序，数值越小越先执行
@Order(value=1)
@Aspect
@Repository
public class LoggingAspect {
```

```
 /**
  *
  * @Title: loggingOperation
  * @Description: TODO(这里用一句话描述这个方法的作用)
  * @param: execution(* aop.yxd.server.*.*(..))
  * ,任何返回值类型，该包下的任意class文件的任何方法，任何参数
  * @return: void
  * @throws
  */
 @Pointcut("execution(* aop.yxd.server.*.*(..))")
 public void loggingOperation() {

 }
 @Before("loggingOperation()")
 // JoinPoint:是aspectJ下的包
 public void beforeLogging(JoinPoint jPoint) {
  System.out.println("方法名：" + jPoint.getSignature().getName());
  System.out.println("参数：" + Arrays.asList(jPoint.getArgs()));
  System.out.println("前置通知方法方法");
 }
 @After("loggingOperation()")
 public void afterLogging(JoinPoint jPoint) {

  System.out.println("后置通知方法");
 }
 /**
  *
  * @Title: afterReturningLogging
  * @Description: 正常返回的时候就是值，不正常返回就是空NULL
  * @param: @param jPoint
  * @param: @param result
  * @return: void
  * @throws
  */
 @AfterReturning(value = "loggingOperation()", returning = "result")
 public void afterReturningLogging(JoinPoint jPoint, Object result) {
  System.out.println("方法名：" + jPoint.getSignature().getName());
  System.out.println("返回通知返回结果为：" + result);
  System.out.println("返回通知方法");
 }
 @AfterThrowing(value = "loggingOperation()", throwing = "e")
 public void afterThrowingLogging(JoinPoint jPoint, Exception e) {
  System.out.println("方法名：" + jPoint.getSignature().getName());
  System.out.println("异常通知返回结果为：" + e);
  System.out.println("异常通知方法");
 }
 @Around(value = "loggingOperation()")
 public Object aroundLogging(ProceedingJoinPoint proceedingJoinPoint) {
  Object result = null;
  String methodName = proceedingJoinPoint.getSignature().getName();
  try {
   System.out.println("前置通知" + methodName);
   result = proceedingJoinPoint.proceed();
   System.out.println("结果：" + result);
   System.out.println("后置通知" + methodName);
  } catch (Throwable e) {
   System.out.println("异常通知" + methodName);
   e.printStackTrace();
  }
  System.out.println("返回通知" + methodName);
  return result;

 }
}
```

- 在 AspectJ 切面中, 可以通过 @Pointcut 注解将一个切入点声明成简单的方法. 切入点的方法体通常是空的, 因为将切入点定义与应用程序逻辑混在一起是不合理的.

AOP基于xml配置版

步骤：

1. 注册切面Bean，和目标对象
2. 设定切点 <aop:pointcut expression="execution(* .(..))" id="pointcut" />
3. 配置切面（需要：注入切面）
4. 配置通知（需要： 1.通知 在哪？[导入通知。method] ;2.通知给谁[加入切点。pointcut-ref]）

```
<!-- 代理对象 -->
 <bean id="arithmetioc" class="aop.yxd.server.Arithmetioc"></bean>

 <!-- 切面 -->
 <bean id="loggingAspect" class="aop.yxd.aspect.LoggingAspect"></bean>
 <!-- 非注解的 配置aop -->
 <aop:config>
  <!-- 配置切点的表达式 ，注册在spring下，且 aop扫描到的java对象，都会被切到。 -->
  <aop:pointcut expression="execution(* *.*(..))" id="pointcut" />
  <!-- 配置切面及通知 ; ref : 导入切面 ;order:优先级; -->
  <aop:aspect ref="loggingAspect" order="1">
   <!-- 配置通知 -->
   <!-- method:配置通知; pointcut-ref : 将切点去配置进去。 -->
   <aop:before method="beforeLogging" pointcut-ref="pointcut" />
   <aop:before method="afterLogging" pointcut-ref="pointcut" />
   <aop:after-throwing method="afterThrowingLogging"
    pointcut-ref="pointcut" throwing="e" />
  </aop:aspect>

  <!-- 可以在这里配置多个切面，同时，切面里还可以有多个通知[前置，后置....] -->

 </aop:config>
```

补充：

更多的时刻，我们使用基于xml配置的方式使用aop。技术的发展是粒度逐渐加大，向标准的高内聚，低耦合走的。这一点从AOP就可以看出来，自己理解一下。