Spring AOP

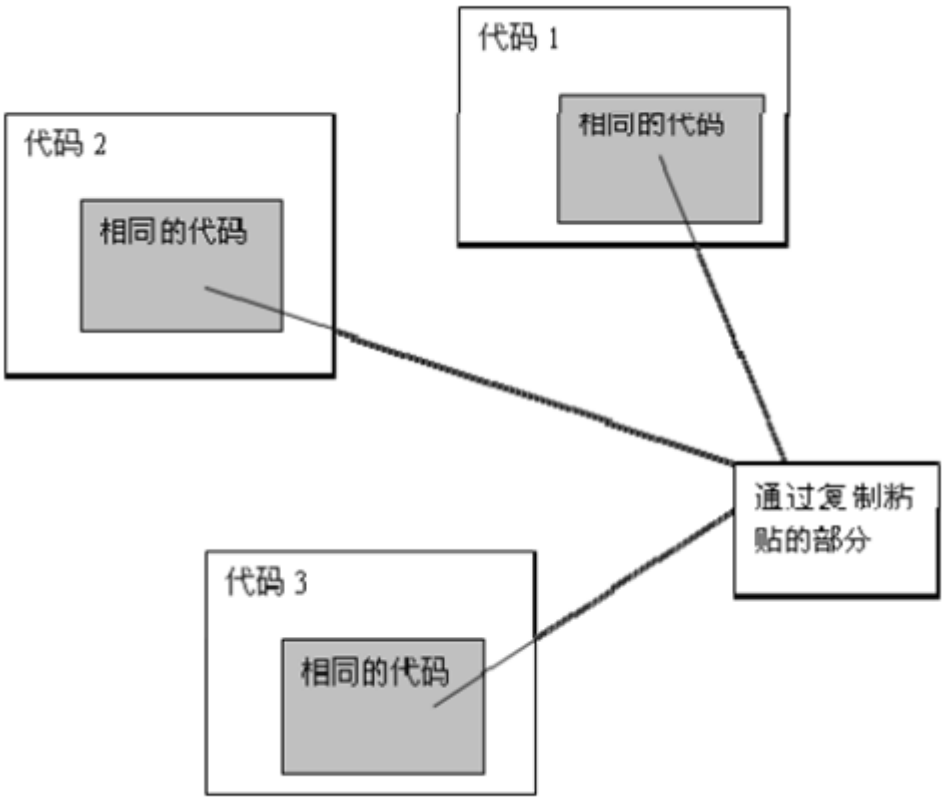
1 需求分析:

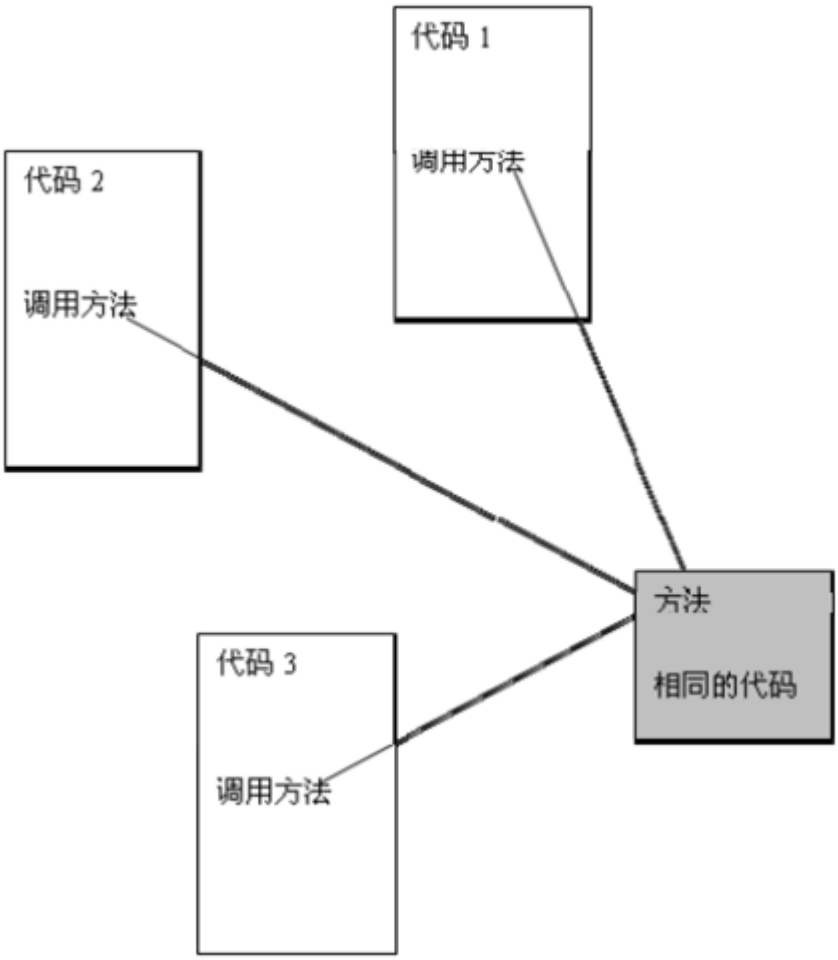
日志记录功能，监视用户对特殊功能操作的每一个流程和步骤。

2 理解DRY规则

在软件开发领域，有一个非常重要的规则：Don’t Repeat Yourself，就是所谓的DRY规则，

意思就是不要写重复的代码





为什么要使用方法调用呢，而不是在三个地方使用重复的代码呢?

> 此举不是是为了编程简单，代码简洁。实际上，这是次要的。最重要的原因是为了软件后期的升级及维护。 （相同的地方需要修改，有什么好处？）

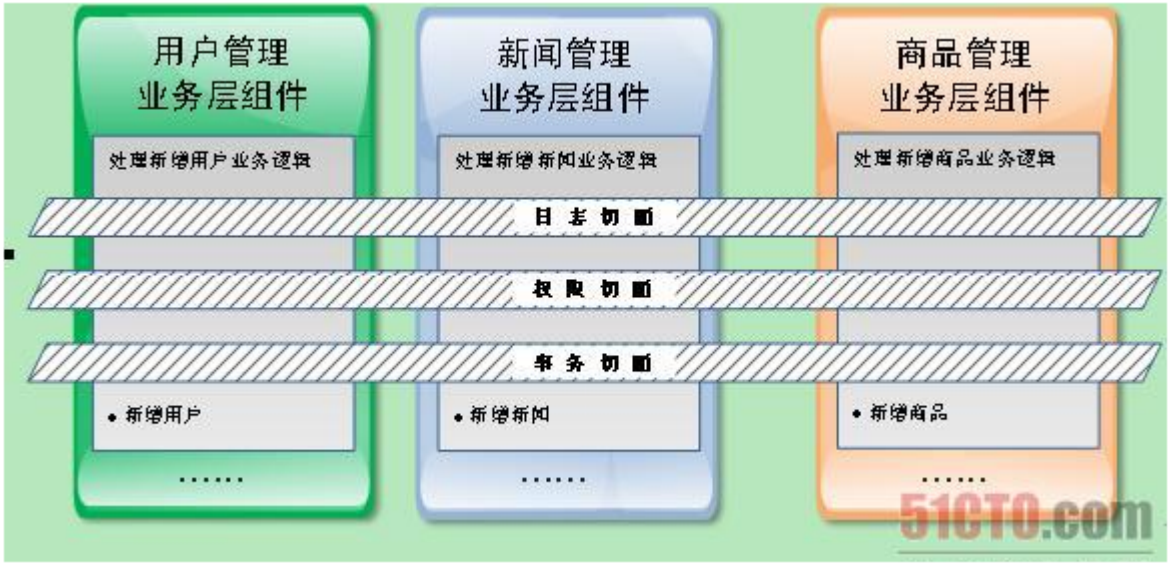
3 简介

AOP为Aspect Oriented Programming的缩写，意为：面向切面编程

AOP提供了另外一种思考程序结构的角度，弥补了OOP（面向对象编程）的不足。就像刚开始理解OO概念一样，对于新手来说AOP也是非常抽象难以理解的，不能仅从一个概念上去定义AOP。假如我们有一个系统，分为好多个模块，每个模块都负责处理一项重要的功能。但是每个模块都需要一些相似的辅助功能如安全、日志输出等等。这就是一种交叉业务，而这种“交叉”非常适合用AOP来解决。

Aspect切面

AOP它利用一种称为“横切”的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其名为“Aspect”，即切面。

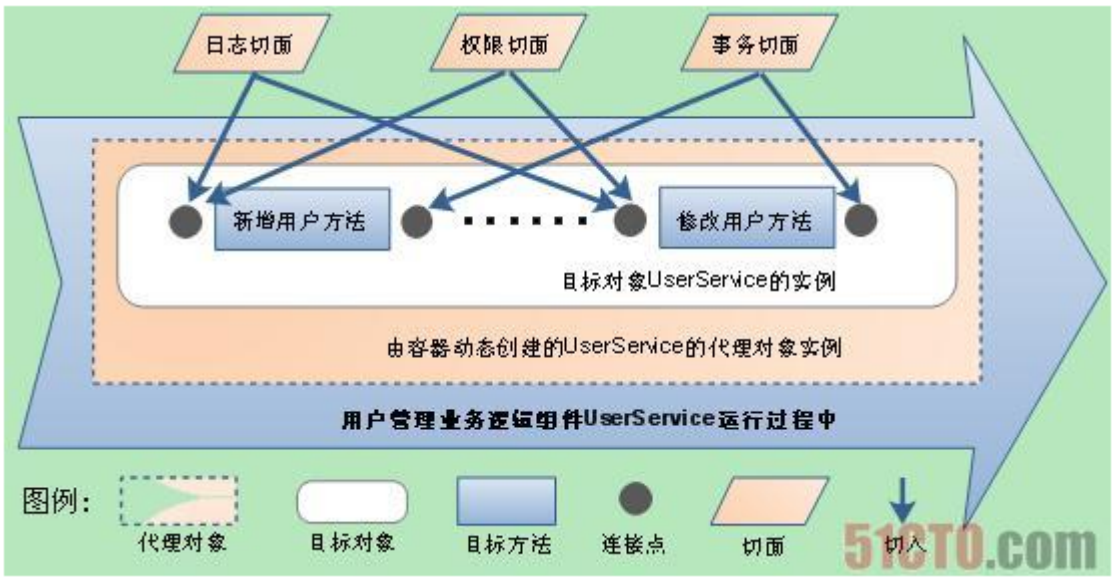


所谓**“切面”**，简单地说，就是将那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来。

切面就是横切面，代表的是一个普遍存在的共有功能。

原理

AOP技术是建立在Java语言的反射机制与动态代理机制之上的



业务逻辑组件在运行过程中，AOP容器会动态创建一个代理对象供使用者调用。该代理对象已经按程序员的意图将切面成功切入到目标方法的连接点上，从而使切面的功能与业务逻辑的功能同时得以执行。

总结

面向切面编程（AOP）就是对软件系统不同关注点的分离，开发者通过拦截方法调用并在方法调用前后

添加辅助代码。

拦截器（Interceptor）是动态拦截Action调用的对象。它提供了一种机制使开发者可以在一个Action执行之前或执行之后插入需要的代码。

4 AOP常用术语

1. **切面（Aspect）**：切面是你要实现的交叉功能。它是应用系统模块化的一个切面或领域。
2. **连接点（Join point）**：连接点是应用程序执行过程中插入切面的地点（在程序执行过程中某个特定的点）在Spring AOP中一个连接点代表一个方法的执行。这个点可以是方法调用，异常抛出或者是要修改的字段。通过申明一个import org.aspectj.lang.JoinPoint类型的参数可以使通知（Advice）的主体部分获得连接点信息。
3. **通知（Advice）**：通知切面的实际实现。它通知应用系统新的行为。通知包括好多种类，在后面单独列出。
4. **切入点（Pointcut）**：切入点定义了通知应该应用在哪些连接点。通知（Advice）可以应用到AOP的任何连接点。通知（Advice）将和一个切入点表达式关联，并在满足这个连接点的切入点上运行（例如：在执行一个特定名称的方法时）切入点表达式如何和连接点匹配是AOP的核心Spring使用缺省的AspectJ切入点的语法。
5. **引入（Introduction）**：引入允许你为已经存在的类添加新的方法和属性。Spring允许引入新的接口（以及一个对应的实现）到任何被代理的对象。
6. **目标对象（Target Object）**：目标对象是被通知对象。Spring AOP是运行时代理实现的，所以这个对象永远是一个被代理对象。
7. **AOP代理（AOP Proxy）**：代理是将通知（Advice）应用到目标对象后创建的对象。
8. **织入（Weaving）**：把切面（Aspect）连接到其它的应用程序类型或者对象上来创建一个被通知（advised）的对象。可以在编译时做这件事（例如使用AspectJ编译器），也可以在类加载或运行时完成。 Spring和其他纯Java AOP框架一样， 在运行时完成织入。

4.1 通知（Advice）类型：

前置通知（Before Advice）：在一个连接点之前执行的通知，但这个通知不能阻止连接点前的执行（除非它抛出异常）

后置通知（After Advice）：

返回后通知（After returning advice）：在一个连接点正常完成后执行的通知。例如：一个方法正常返回，没有抛出任何异常。

抛出后通知（After throwing advice）：在一个方法抛出异常时执行的通知。

环绕通知（Around advice）：包围一个连接点的通知，就像方法调用。这是最强大的一种通知类型。环绕通知可以在方法调用前后完成自定义的行为。它也会选择是否继续执行连接点或直接返回他们自己的返回值域或抛出异常来结束执行。

5 AOP开发示例

导入需要的依赖 jar

Spring Aop:

实现开发方式

1、编程方式 2、配置方式 3、**基于注解方式@AspectJ注解驱动的切面**

基于注解的Spring AOP的配置和使用日志记录

1 创建切面

```
/**
 * 定义一个切面
 * 日志管理
 */
@Component  //将bean注册到Spring IOC容器中
@Aspect   //将该类声明为一个切面 Aspect
public class LogAspect {
    /**
     * @Before 通知类型（前置通知）
     * execution 切入点 Pointcut
     * 第1个* 代表方法返回值   2个*代表类名称 3个*代表方法名称   最后..代表所有的参数
     */
    @Before("execution(* com.qfjy.service.impl.*.*(..))")
    public void before(){
        System.out.println("---目标方法前执行---");
    }
}
```

2 在applicationContext.xml中加入配置

```
    <!-- Spring aop aspectj  激活自动完成代理对象操作 -->
    <aop:aspectj-autoproxy></aop:aspectj-autoproxy>
```

3 PointCut execution 切入点

切入点（Pointcut）：切入点定义了通知应该应用在哪些连接点。通知（Advice）可以应用到AOP的任何连接点。通知（Advice）将和一个切入点表达式关联，并在满足这个连接点的切入点上运行（例如：在执行一个特定名称的方法时）切入点表达式如何和连接点匹配是AOP的核心Spring使用缺省的AspectJ切入点的语法。

示例：

```
例:  execution (public  * com.qfjy.service..*.*(..))

整个表达式可以分为五个部分:

1、execution():: 表达式主体。
2、第一个*号:表示返回类型,  *号表示所有的类型。
3、包名:表示需要拦截的包名,后面的两个句点表示当前包和当前包的所有子包,com.qfjy.service包、子孙包下所有类的方法。
4、第二个*号:表示类名,*号表示所有的类。
5、*(..):最后这个星号表示方法名,*号表示所有的方法,后面括弧里表示方法的参数,两个句点表示任何参数0或多个
6、public可以省略。代表访问public方法
```

PointCut可以有下列方式来定义或者通过&& || 和!的方式进行组合

```
("execution(* com.qfjy.controller.*.*(..)) || execution(* com.qfjy.service.*.*(..))")
```

```
例如:
1.execution(* *(..));  代表所有类中的所有方法,
参数说明:第一个*[返回类型为任意],第二*[方法名为任意],(..)表示的是方法参数为任意
2.execution(* set*(..));   任何一个以“set”开始的方法的执行:
3.execution(* *(*,String)); 方法参数第一个为任意参数类型,第二个为String
4.execution(* cn.qfjy.service.*.*(..));   service包下的所有类的所有方法
```

4 JoinPoint连接点

连接点（Join point）：连接点是应用程序执行过程中插入切面的地点（在程序执行过程中某个特定的点）在Spring AOP中一个连接点代表一个方法的执行。这个点可以是方法调用，异常抛出或者是要修改的字段。通过申明一个 import org.aspectj.lang.JoinPoint类型的参数可以使通知（Advice）的主体部分获得连接点信息。

希望日志方法记录信息更为详细(JoinPoint)

AspectJ使用org.aspectj.lang.JoinPoint接口表示目标类连接点对象，**如果是环绕增强时，使用org.aspectj.lang.ProceedingJoinPoint**表示连接点对象，该类是JoinPoint的子接口。任何一个增强方法都可以通过将第一个入参声明为JoinPoint访问到连接点上下文的信息。这两个接口的主要方法:

```
JoinPoint:
java.lang.Object[] getArgs():获取连接点方法运行时的入参列表(入参的参数值);
Signature getSignature() :获取连接点的方法签名对象(方法名称);
java.lang.Object getTarget() :获取连接点所在的目标对象;
java.lang.Object getThis() :获取代理对象本身;
```

```
@Before("execution(* com.qfjy.service.impl.*.*(..))")
public void before(JoinPoint join){
    Object[] objs=join.getArgs();//获取方法入参列表
    for(Object o:objs){
        System.out.println(o);
    }
    String methodName=join.getSignature().getName();//获取方法名称
    Object obj=join.getTarget();//目标对象
    String className = join.getTarget().getClass().getName();//类的全限定名
    System.out.println("---目标方法前执行---"+methodName);
}
```

通知

通知（Advice）类型:

前置通知（Before Advice）：在一个连接点之前执行的通知，但这个通知不能阻止连接点前的执行（除非它抛出异常）

后置通知（After Advice）：在目标方法执行后（无论是否出现异常）执行的通知

返回后通知（After returning advice）：在一个连接点正常完成后执行的通知。例如：一个方法正常返回，没有抛出任何异常。

抛出后通知（After throwing advice）：在一个方法抛出异常时执行的通知。

环绕通知（Around advice）：包围一个连接点的通知，就像方法调用。这是最强大的一种通知类型。环绕通知可以在方法调用前后完成自定义的行为。它也会选择是否继续执行连接点或直接返回他们自己的返回值域或抛出异常来结束执行。

后置通知（After Advice）

后置通知（After Advice）：在目标方法执行后（无论是否出现异常）执行的通知

```
@After("execution(* com.qfjy..*.*(..))")
public void after(JoinPoint join){
    String methodName=join.getSignature().getName();//方法名称
    Object obj=join.getTarget();//目标对象
    System.out.println("---后置通知---"+obj+methodName);
}
```

返回后通知 After returning advice

返回后通知（After returning advice）：在一个连接点正常完成后执行的通知。例如：一个方法正常返回，没有抛出任何异常。(得到方法返回值通知)

```
/**
     * 返回后通知
     * pointcut  或 value属性 设定切入点。
     * retuning 定义返回值的 变量名
     */
    @AfterReturning(pointcut="execution(* com.qfjy.service.impl.*.*
(..))",returning="result")
    public void afterReturn(JoinPoint join,Object result){
        String methodName=join.getSignature().getName();//方法名称
        Object obj=join.getTarget();//目标对象
        System.out.println("---方法返回后通知---"+obj+methodName+"--返回值是：+result);
    }
```

抛出后通知（After throwing advice)

抛出后通知（After throwing advice）：称为异常通知。在一个方法抛出异常时执行的通知。

```
异常通知：在方法出现异常后，会执行的通知代码。
能够访问到异常对象、同时可通过指定特定异常，执行相应通知代码
```

```
@AfterThrowing(pointcut="execution(* com.qfjy.service.impl.*.*(..))",throwing="ex")
public void afterThrowing(JoinPoint join,Exception ex){
    String methodName=join.getSignature().getName();//方法名称
    System.out.println("---异常通知---"+methodName+"--出现的异常是：+ex);
}
```

可以通过改变 Exception 异常类型，来对特定异常，执行相应的通知代码

```
@AfterThrowing(pointcut="execution(* com.qfjy.service.impl.*.*(..))",throwing="ex")
public void afterThrowing(JoinPoint join,ArrayIndexOutOfBoundsException ex){
    String methodName=join.getSignature().getName();//方法名称
    System.out.println("---异常通知---"+methodName+"--出现的异常是数组下标越界：+ex);
}
@AfterThrowing(pointcut="execution(* com.qfjy.service.impl.*.*(..))",throwing="ex")
public void afterThrowingNullPointerException(JoinPoint join,NullPointerException ex){
    String methodName=join.getSignature().getName();//方法名称
    System.out.println("---异常通知---"+methodName+"--出现的异常是空指针：+ex);
}
```

环绕通知

环绕通知（Around advice）：包围一个连接点的通知，就像方法调用。这是最强大的一种通知类型。环绕通知可以在方法调用前后完成自定义的行为。它也会选择是否继续执行连接点或直接返回他们自己的返回值域或抛出异常来结束执行

```
    /**
     * 环绕通知
     * 如果是环绕增强时，使用org.aspectj.lang.ProceedingJoinPoint表示连接点对象，
        该类是JoinPoint的子接口
     */
```

```
    @Around(value="execution(* com.qfjy.service.impl.*.*(..))")
    public Object aroundAdvice(ProceedingJoinPoint join){
        System.out.println("-----环绕通知 --前置通知");
        Object obj=null;
        try {
            obj = join.proceed();//执行目标方法
            System.out.println("-----环绕通知--返回后通知");
        } catch (Throwable e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
            System.out.println("---环绕通知--异常通知--");
        }
        System.out.println("----环绕通知--后置通知");
        return obj;
    }
```

技巧:

为了便于对PointCut的管理。减少每个类上方的内容匹配

可通过定义切入点来实现

```
    @Pointcut(value="execution(* com.qfjy.service.impl.UserServiceImpl.*(..))")
    public void userPointCut(){}
```

对应通知方法切入点如下:

```
//前置通知
@Before("userPointCut()")
public void before(JoinPoint join){
    //代码省略....
}
//环绕通知
@Around(value="userPointCut()")
public Object aroundAdvice(ProceedingJoinPoint join){
    //代码省略....
}
```

总结:

Spring AOP的特点：功能非常强大，开发编写非常容易。

在实际开发中，常用于应用场景:

日志的记录管理、统一的异常管理、事务的管理等。