为什么要有Spring AOP？

上一篇从Web开发演进过程的一个侧面简述了一下为什么会有Spring？事实上只介绍了为什么会有Spring IOC（控制反转/依赖注入）以及Spring IOC的雏形。我们知道Spring的两个核心知识点是：IOC和AOP。因此，这一篇还是以Web开发演进过程为线索继续探讨一下为什么会有Spring AOP？等介绍完这两个核心的知识点之后，才会进一步展开对Spring核心原理的探讨！

一、Web开发演进到一定阶段的痛点

我们在初学习Java Web的时候，应该都经历了以下的阶段：

（1）一个主函数main中包含了所有的方法；

（2）将主函数中的方法进行拆分封装，抽取为一个个的方法；

（3）按照每一个方法不同的功能分为一个个的类；

（4）有了MVC模型之后，我们按照MVC的思想将我们的代码拆分为三层，每层负责不同的功能，进行分门别类的管理；

很多程序的功能还可以通过继承关系而得到重用，进一步提高了开发效率。再后来，又出现了各种各样的设计模式，使设计程序功能变得得心应手。

在面向对象的大环境下，我们可以很好地组织代码，通过继承、封装和多态的思想去设计一个个比较让人满意的类，但是我们慢慢的发现，我们的代码中逐渐多了很多重复性的代码，有人可能会想到，把这些重复性的代码抽取出来不就好了吗？是这样的，我们看一下这种思路的一个实例：

```java
public class UserServiceImpl implements UserService {

    public User getById(String userId) {
        LogUtils.log("传入的参数是: " + userId);
        //执行具体的数据访问操作，假设执行了
        User user = new User();
        LogUtils.log("得到的结果是: " + user.toString());
        return null;
    }

    public int add(User user) {
        LogUtils.log("传入的参数是: " + user.toString());
        //执行具体的数据访问操作，假设执行了
        LogUtils.log("得到的结果是: " + user.toString());
        return 0;
    }

    public boolean delete(String userId) {
        LogUtils.log("传入的参数是: " + userId);
        //执行具体的数据访问操作，假设执行了
        boolean isSuccess = true;
        LogUtils.log("得到的结果是: " + isSuccess);
        return false;
    }
}
```


可以看到，上述代码功能上确实可以实现，但是我们的业务代码已经被这些非核心的代码所混淆，并且占据了大量的空间！显然这种显示的调用过程成为了我们开发过程中的一个痛点，如何将类似这种的非核心的代码剥离出去成为一个迫切需要解决的问题！

不仅如此，假设我们要控制每一个方法的访问权限，只允许一部分用户进行访问，在不考虑过滤器的情况下，我们是不是需要在每一个方法开始的时候判断用户是否具有该权限，如果有的话就可以进行访问，如果没有的话，就不允许进行访问！

诸如此类，还有数据库事务的控制，数据库连接的创建和关闭等等，这些都充斥这大量重复性的模板代码！一个很现实的问题，假如有一天，业务需求不需要进行日志记录了，那岂不是我们需要把以前写的代码，全部删掉！想想都是一件很可怕的事情！

二、使用设计模式进行一次改进

如果你对设计模式玩的比较熟的话，这个时候你可能会想到使用**JDK动态代理设计模式**（动态代理设计模式可以在原有的方法前后添加判断、选择或其他逻辑）对上述代码进行改进，（关于什么是JDK动态代理，这里不再详细赘述，有不懂的的可以查阅相关资料具体了解一下！）修改后的代码如下：

```java
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

public class ProxyFactory implements InvocationHandler {

    private Object target;

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

    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        System.out.println("执行---LogUtils.log(\"传入的参数是: \" + args.toString())");
        Object result = method.invoke(target, args);
        System.out.println("执行---LogUtils.log(\"得到的结果是: \" + result)");
        return result;
    }

    public Object getProxy() {
        return Proxy.newProxyInstance(target.getClass().getClassLoader(),
                target.getClass().getInterfaces(), this);
    }
}
```


上述为代理类，红色框中圈出的表示以前业务中的模板代码，这里直接输出表示方法执行的过程，以前的UserServiceImpl修改为如下（直接用输出的方式表示方法执行了）：

```java
public class UserServiceImpl implements UserService {

    public User getById(String userId) {
        System.out.println("执行了getById方法");
        return null;
    }

    public int add(User user) {
        System.out.println("执行了add方法");
        return 0;
    }

    public boolean delete(String userId) {
        System.out.println("执行了delete方法");
        return false;
    }
}
```


测试代码如下：

```java
public class ProxyDemo {

    public static void main(String[] args) {
        UserService userService = new UserServiceImpl();
        UserService proxy =
                (UserService) new ProxyFactory(userService).getProxy();
        proxy.add(new User());
        proxy.delete("");
        proxy.getById("");
    }
}
```

```
xyDemo
"C:\Program Files\Java\jdk1.8.0_151\bin\java" ...
执行---LogUtils.log("传入的参数是: " + args.toString())
执行了add方法
执行---LogUtils.log("得到的结果是: " + result)
执行---LogUtils.log("传入的参数是: " + args.toString())
执行了delete方法
执行---LogUtils.log("得到的结果是: " + result)
执行---LogUtils.log("传入的参数是: " + args.toString())
执行了getById方法
执行---LogUtils.log("得到的结果是: " + result)
```


上述的执行结果可以看出，每次调用一个方法的时候前后都会调用我们期望的代码，实现了我们期望的标准！

通过JDK动态代理的方式，让我们彻底的解放出来了！

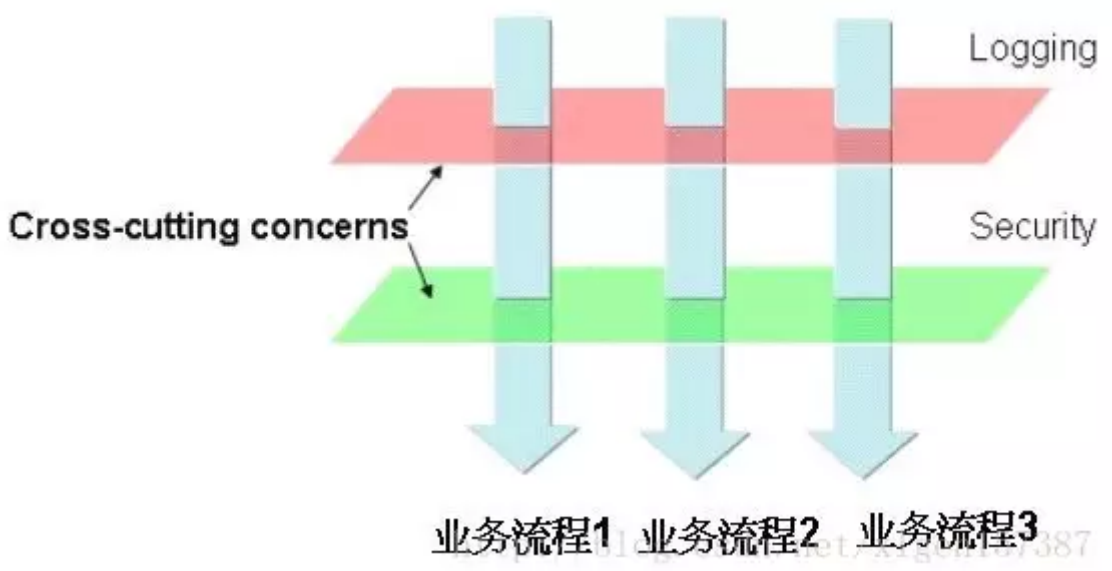
三、撕开披在AOP身上的一层薄纱

上述过程中，我们看到在**动态代理的**invoke方法里边，我们相当于在原有方法的调用前后“**植入**”了我们的通用日志记录代码，如果你看到这一层的话，那么恭喜你！你已经领悟到了AOP思想最核心的东西了！

上述抽取公共代码其实就是AOP中**横切**的过程，代理对象中在方法调用前后“**植入**”自己写的通用日志记录代码其实就是AOP中**织入**的过程！这个织入的代码也就是**横切逻辑**，织入代码的过程其实就是在原有的方法前后**增强**　原方法的过程！总的来说，我们想解决我们开发中的痛点，然后就出现了一种技术，这种技术手段就是AOP。

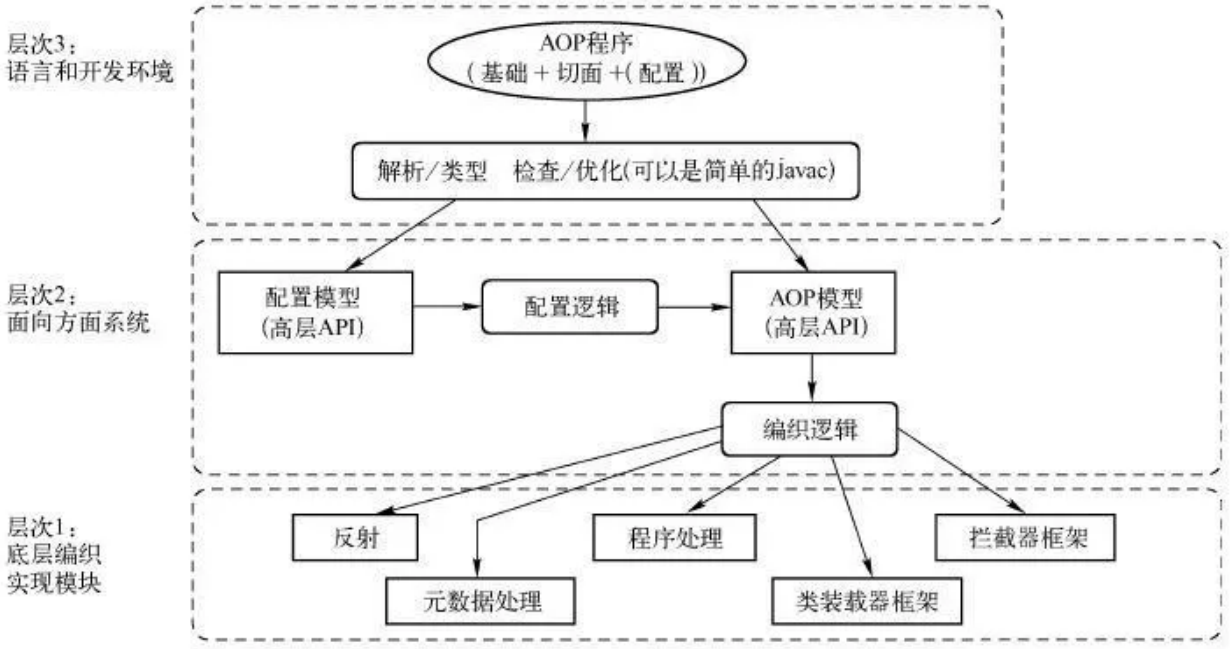
AOP书面表述如下：

AOP（Aspect Oriented Programming）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（Spring核心之一），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。



四、AOP与Spring AOP的关系

AOP是一种思想，不同的厂商或企业可能有不同的实现方式，为了更好的应用AOP技术，技术专家们成立了AOP联盟来探讨AOP的标准化，AOP联盟定义的AOP体系结构把与AOP相关的概念大致分为由高到低、从使用到实现的三层关系，AOP联盟定义的AOP体系结构如下图：



AOP 联盟定义的 AOP 体系结构

在AOP联盟定义的AOP体系结构下有很多的实现者，例如：AspectJ、AspectWerkz、JBoss AOP、Spring AOP等。Spring AOP就是在此标准下产生的，这里不再深入Spring AOP的其他概念，这些概念会在后期探讨。

五、其他问题

上述通过动态代理的方式实现了简单的AOP，但是值得注意的是，我们的代理目标对象必须实现一个接口，要是一个接口的实现类，这是因为再生成Proxy对象的时候这个方法需要一

个目标对象的接口：

```
public Object getProxy() {
    return Proxy.newProxyInstance(target.getClass().getClassLoader(),
            target.getClass().getInterfaces(), this);
}
```


显然，有些特殊的场景使用JDK动态代理技术的话，已经不能够满足我们的场景了，又遇到痛点了！凡事不劳我们操心的Spring框架已经替我们想到了，既然你有这种需求，我就使用一种技术帮你实现就行了，Spring在这里使用CGLib动态代理的方式实现了我们的这种诉求。

CGLib采用底层的字节码技术，可以为一个类创建子类，在子类中采用方法拦截的技术拦截所有父类方法的调用并顺势的织入横切逻辑。

看到这里，我们会想以后会不会还有CGLib解决不了得问题啊？我们已经很清楚的知道了对于Spring AOP来说，使用到了JDK动态代理技术和CGLib动态代理技术，这两种方式已经实现了我们绝大多数的场景，如果还有不能满足的需求，迫切需要解决的痛点，我相信Spring还会采用相应的技术来满足这些场景。

六、总结

上述的过程，大致从一个侧面探讨了一下我们为什么需要AOP，AOP与Spring AOP的关系以及Spring AOP两种实现的方式（JDK动态代理和CGLib动态代理）。

Spring不尝试提供最为完善的AOP实现，它更侧重于提供一种和Spring IOC容器整个的AOP实现，用于解决实际的问题，在Spring中无缝的整合了Spring AOP、Spring IOC和AspectJ。

当然，Spring AOP的内容不仅仅有这些！例如：我们在使用Spring AOP的时候只是简单的配置了一下（通过XML或注解进行配置），没有像ProxyDemo测试类中的那样，还需要我们手动的调用ProxyFactory 来创建代理对象，然后调用我们的目标方法，其实Spring AOP在内部已经帮我们把这些事情做好了，具体的原理后期会继续探讨。另外，Spring如何整合Spring IOC和AOP的，这一点也会在后期探讨。

最后补充一下！动态代理或者设计模式重要吗？很重要！Spring AOP用到了动态代理，Spring事务管理用到了动态代理，MyBatis数据库连接池用到了动态代理，MyBatis创建Mapper用到了动态代理等等，你说重要不！要想踏进这些高层框架原理的大门，设计模式首先是我们的第一段台阶！