

# 面向切面的编程AOP

浙江大学城市学院

彭 彬

[pengb@zucc.edu.cn](mailto:pengb@zucc.edu.cn)

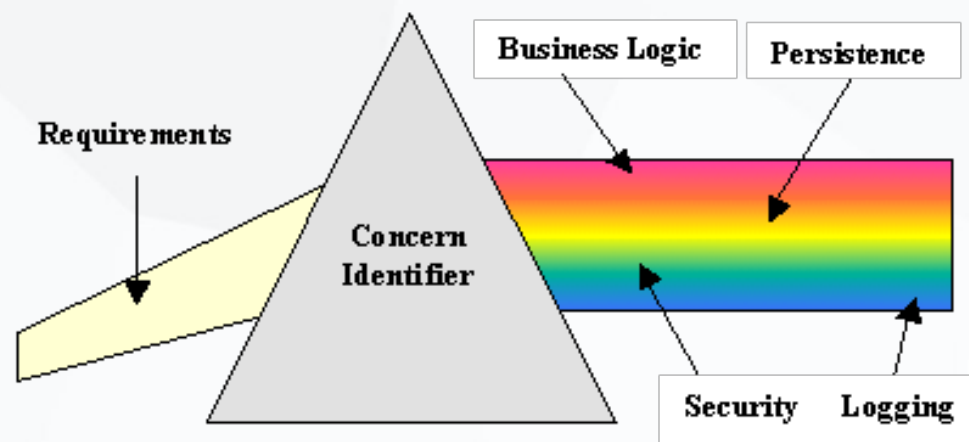
# AOP

AOP为Aspect Oriented Programming的缩写，意为：面向切面编程，可以**通过预编译方式和运行期动态代理**实现在**不修改源代码**的情况下给程序动态统一**添加功能**的一种技术。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。常用于日志记录，性能统计，安全控制，事务处理，异常处理等等。

# AOP

我们可以将系统模块划分为：核心关注点和横切关注点。对于核心关注点而言，通常来说，实现这些关注点的模块是相互独立的，他们分别完成了系统需要的商业逻辑，这些逻辑与具体的业务需求有关。而对于日志、安全、持久化等关注点而言，他们却是商业逻辑模块所共同需要的，这些逻辑分布于核心关注点的各处。在AOP中，诸如这些模块，都称为横切关注点。

如果将整个模块比喻为一个圆柱体，那么关注点识别过程可以用三棱镜法则来形容，穿越三棱镜的光束（指需求），照射到圆柱体各处，获得不同颜色的光束，最后识别出不同的关注点。



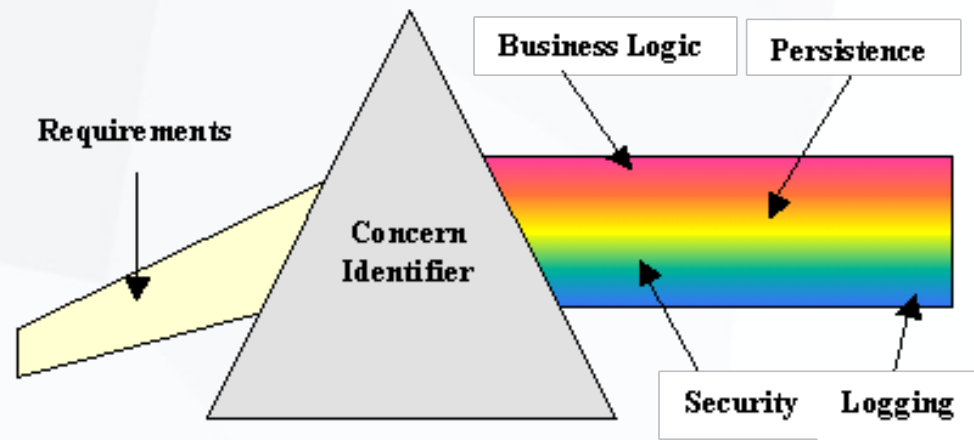
# AOP分析及实现的基本思路

上图识别出来的关注点中，Business Logic属于核心关注点，它会调用到Security，Logging，Persistence等横切关注点。

```
public class BusinessLogic
{
    public void SomeOperation()
    {
        //验证安全性；Security关注点；
        //执行前记录日志；Logging关注点；

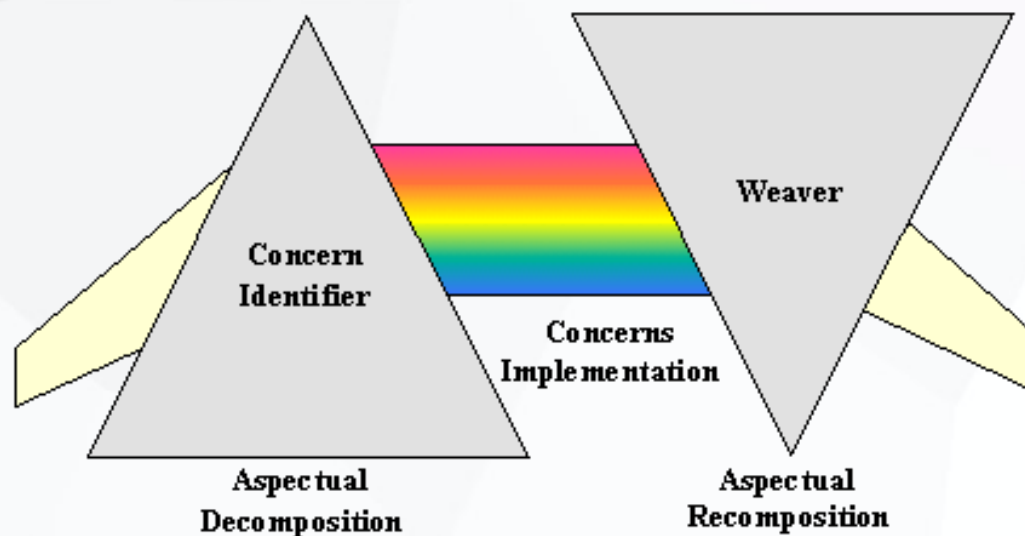
        DoSomething();

        //保存逻辑运算后的数据；Persistence关注点；
        //执行结束记录日志；Logging关注点；
    }
}
```



# AOP分析及实现的基本思路

AOP的目的，就是要将诸如Logging之类的横切关注点从BusinessLogic类中分离出来。利用AOP技术，可以对相关的横切关注点封装，形成单独的“Aspect”。这就保证了横切关注点的复用。由于BusinessLogic类中不再包含横切关注点的逻辑代码，为达到调用横切关注点的目的，可以利用横切技术，截取BusinessLogic类中相关方法的消息，例如SomeOperation()方法，然后将这些“Aspect”织入到该方法中。



# AOP分析及实现的基本思路

AOP原理及其实现比IoC可能更困难一些，我们可以将如何实现AOP用这样的语言来描述：“我们将通过一种方法，将一些实现共同功能的代码织入到我们需要的地方”，织入的方法就是使用“代理对象”。

设想我们要给每个方法的调用入口增加一个日志功能，这个“每个方法调用入口”就是切面（Aspect），要“织入”的方法就是“日志功能”，而我们要实现这种织入，就使用“代理对象”来完成。

我们按照下面的逻辑来尝试向大家讲解。

代理模式→静态代理→动态代理→Java实现方法→AOP实现→示例

# 代理模式

举一个现实生活中的例子：歌星或者明星都有一个自己的经纪人，这个经纪人就是他们的代理人，当我们需要找明星表演时，不能直接找到该明星，只能是找明星的代理人。比如刘德华在现实生活中非常有名，会唱歌，会跳舞，会拍戏，当我们需要找刘德华表演时，不能直接找到刘德华了(刘德华说，你找我代理人商谈具体事宜吧!)，只能是找刘德华的代理人，因此刘德华这个代理人存在的价值就是拦截我们对刘德华的直接访问！

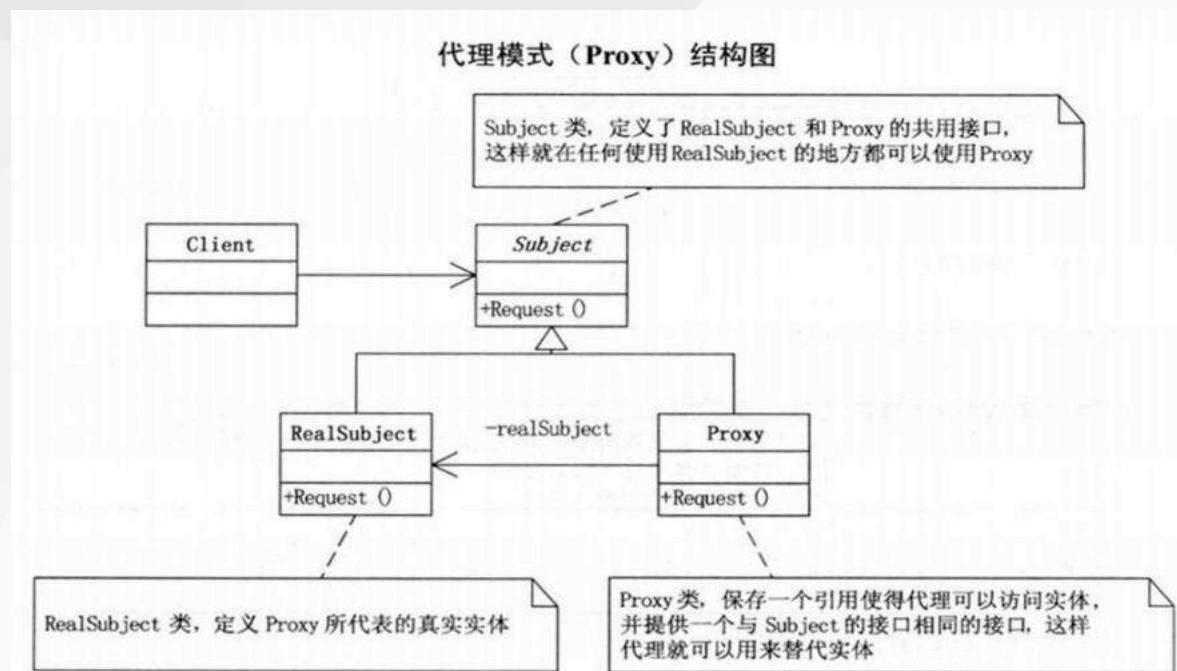
明确代理对象的两个概念：

- 1、代理对象存在的价值主要用于拦截对真实业务对象的访问。
- 2、代理对象应该具有和目标对象(真实业务对象)相同的方法。


刘德华(真实业务对象)会唱歌，会跳舞，会拍戏，刘德华有什么业务，代理人就代理什么业务，我们要找刘德华唱歌，跳舞，拍戏，先找他的经纪人，交钱给他的经纪人，然后经纪人再让刘德华去唱歌，跳舞，拍戏。

# 代理模式

代理模式是常用的java设计模式，他的特征是代理类与委托类有同样的接口，代理类主要负责为委托类预处理消息、过滤消息、把消息转发给委托类，以及事后处理消息等。代理类与委托类之间通常会存在关联关系，一个代理类的对象与一个委托类的对象关联，代理类的对象本身并不真正实现服务，而是通过调用委托类的对象的相关方法，来提供特定的服务。







# 代理模式实现方式

实现代理模式有两种方法：

静态代理：是由程序员创建或特定工具自动生成源代码，再对其编译。在程序员运行之前，代理类.class文件就已经被创建了。

动态代理：是在程序运行时通过反射机制动态创建代理对象。

# 静态代理模式实现方式

静态代理模式

接口

代理对象

```
public interface IService {  
    public void doA();  
    public void doB();  
}
```

业务实现对象

```
public class ServiceImpl implements IService {  
    @Override  
    public void doA() {  
        System.out.println("doA");  
    }  
  
    @Override  
    public void doB() {  
        System.out.println("doB");  
    }  
}
```

```
public class ServiceProxy implements IService {  
    private ServiceImpl svcImpl;  
  
    public ServiceProxy(ServiceImpl si){  
        this.svcImpl = si;  
    }  
  
    @Override  
    public void doA() {  
        System.out.println("log:invoke doA");  
        this.svcImpl.doA();  
    }  
  
    @Override  
    public void doB() {  
        System.out.println("log:invoke doB");  
        this.svcImpl.doB();  
    }  
}
```

# 动态代理模式实现方式

动态代理：是在程序运行时通过反射机制动态创建代理对象。

利用Java反射机制构建动态代理对象

```
public class DynamicServiceProxy implements InvocationHandler {
    private Object svcImpl;

    public DynamicServiceProxy(Object obj) {
        this.svcImpl = obj;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) {
        System.out.println("log:dynamic proxy invoke:" + method.getName());
        Object result = null;
        try {
            result = method.invoke(svcImpl, args);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
        return result;
    }
}
```

# 动态代理模式实现方式

动态代理：是在程序运行时通过反射机制动态创建代理对象。

利用Java反射  
Proxy类创建动态  
代理对象实例

```
//动态代理
```

```
IService svc = (IService)Proxy.newProxyInstance(  
    IService.class.getClassLoader(),  
    new Class[]{IService.class},  
    new DynamicServiceProxy(new cn.edu.zucc.pb.dynamicproxy.ServiceImpl()))  
svc.doA();  
svc.doB();
```

注意Proxy.newProxyInstance()方法接受三个参数：

ClassLoader loader:指定当前目标对象使用的类加载器,获取加载器的方法是固定的

Class<?>[] interfaces:指定目标对象实现的接口的类型,使用泛型方式确认类型

InvocationHandler:指定动态处理器，执行目标对象的方法时,会触发事件处理器的方法

# 动态代理模式实现方式--CGLib

利用Java的动态代理有一个问题：无法摆脱仅支持interface代理的限制，对于没有接口的类，如何实现动态代理呢，这就需要CGLib了



Byte Code Generation Library is high level API to generate and transform JAVA byte code. It is used by AOP, testing, data access frameworks to generate dynamic proxy objects and intercept field access. <https://github.com/cglib/cglib/wiki>

CGLib采用了非常底层的字节码技术，其原理是通过字节码技术为一个类创建子类，并在子类中采用方法拦截的技术拦截所有父类方法的调用，顺势织入横切逻辑。但因为采用的是继承，所以不能对final修饰的类进行代理。JDK动态代理与CGLib动态代理均是实现Spring AOP的基础。

<https://github.com/cglib/cglib>

# 动态代理模式实现方式--CGLib

使用CGLib创建“字节码”级别的子类，并设置拦截器

```
public Object getInstance(final Object target) {  
    this.svcImpl = target;  
    Enhancer enhancer = new Enhancer();  
    enhancer.setSuperclass(this.svcImpl.getClass());  
    enhancer.setCallback(this);  
    return enhancer.create();  
}
```

@Override

```
public Object intercept(Object object, Method method, Object[] args,  
    MethodProxy methodProxy) throws Throwable {  
    System.out.println("log:cglib proxy invoke:" + method.getName());  
    Object result = null;  
    try {  
        result = methodProxy.invokeSuper(object, args);  
    } catch (IllegalAccessException e) {  
        e.printStackTrace();  
    } catch (InvocationTargetException e) {  
        e.printStackTrace();  
    }  
    return result;  
}
```

# 动态代理模式实现方式--CGLib

使用CGLib进行代理的动态创建并调用

//CGLib动态代理

```
cn.edu.zucc.pb.cgproxy.IService cgSvc = new cn.edu.zucc.pb.cgproxy.ServiceImpl();  
CglibProxy cglibProxy = new CglibProxy();  
cn.edu.zucc.pb.cgproxy.ServiceImpl svcImplProxy =  
    (cn.edu.zucc.pb.cgproxy.ServiceImpl) cglibProxy.getInstance(cgSvc);  
svcImplProxy.doA();  
svcImplProxy.doB();
```

CGLIB创建的动态代理对象比JDK创建的动态代理对象的性能更高，但是CGLIB创建代理对象时所花费的时间却比JDK多得多。原因是CGLIB是通过运行时动态创建“子类”代码的方式进行创建的。



## 参考内容

感谢下面参考内容的作者。

<https://www.cnblogs.com/jingzhishen/p/4980551.html>

[https://blog.csdn.net/m0\\_38039437/article/details/77970633](https://blog.csdn.net/m0_38039437/article/details/77970633)

<https://www.cnblogs.com/daniels/p/8242592.html>

<https://www.cnblogs.com/gonjan-blog/p/6685611.html>

<https://github.com/cglib/cglib>





# END

---

Pb&Lois