设计模式

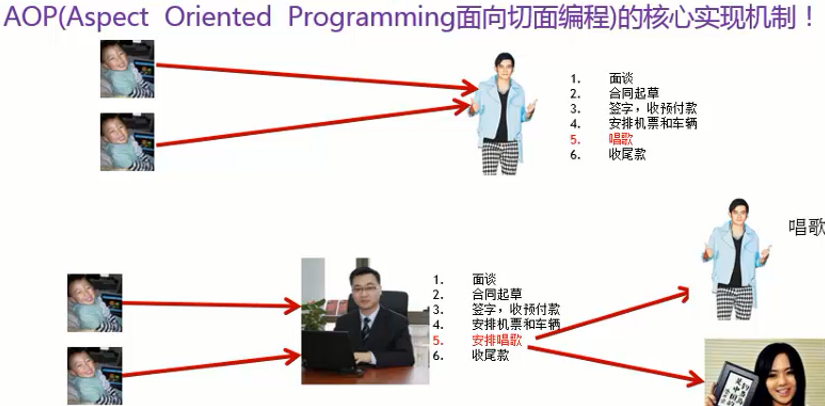
# 代理模式

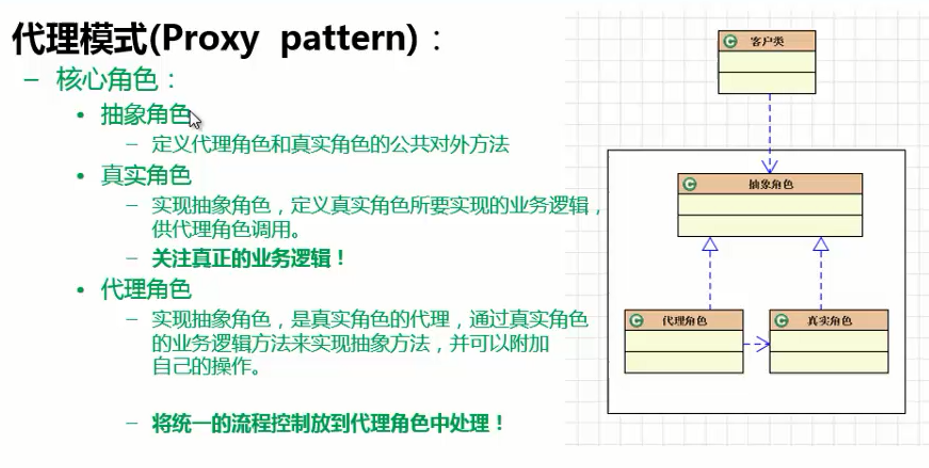
Proxy Pattern

## 1.核心作用

通过代理，控制对对象的访问。

可以详细控制访问某个（某类）对象的方法，在调用这个方法前做前置处理，调用这个方法后做后置处理（AOP的微观实现）





## 2.核心角色

1）抽象角色

定义代理角色和真实角色的公共对外方法

2）真实角色

实现抽象角色，定义真实角色所要实现的业务逻辑，供代理角色调用

**关注真正的业务逻辑**

3）代理角色

实现抽象角色，是真实角色的代理，通过真实角色的业务逻辑方法来实现抽象方法，并可以附加自己的操作

**将统一的流程控制放到代理角色中处理**

## 3.应用场景

安全代理：屏蔽对真实角色的直接访问

远程代理：通过代理类处理远程方法调用（RMI）

延迟加载：先加载轻量级的代理对象，真正需要再加载真实对象

比如要开发大文档查看软件，大文档中有大的图片，可能有100M，在打开文件是不可能将所有的图片显示出来，可以使用代理模式，当需要查看图片时，用proxy来进行大图片的打开

## 4.分类

静态代理（静态**定义**代理类）

动态代理（动态**生成**代理类）\*\*\*

JDK自带的动态代理

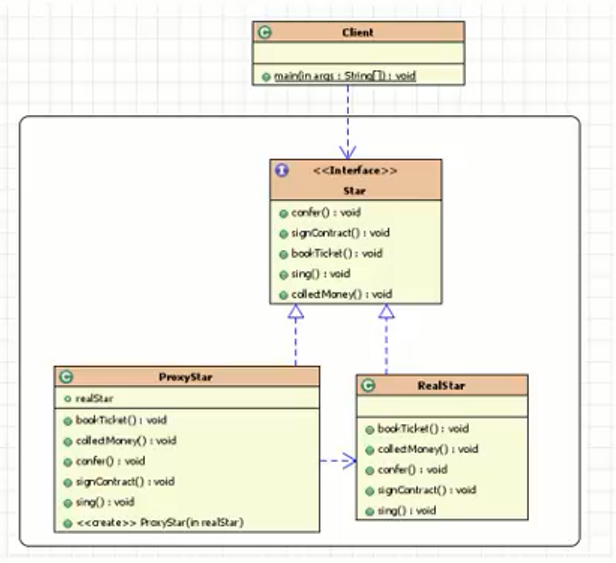
javaassist字节码操作库实现

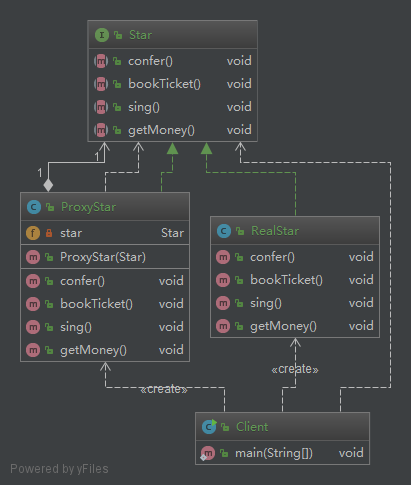
CGLIB

ASM（底层使用指令，可维护性较差）

## 静态代理

static proxy





## 动态代理

dynamic proxy，比静态代理用的更多

动态代理（动态**生成**代理类）\*\*\*

JDK自带的动态代理

javaassist字节码操作库实现

CGLIB

ASM（底层使用指令，可维护性较差）

动态代理相比静态代理的优点

抽象角色中（接口）声明的所有方法都被转移到调用处理器一个集中的方法中处理，这样我们可以更加灵活和统一的处理众多的方法

JDK自带的动态代理

java.lang.reflect.Proxy

作用：动态生成代理类和对象

java.lang.reflect.InvocationHandler（处理器接口）

可以通过invoke方法实现对真实角色的代理访问

每次通过Proxy生成代理类对象时都要指定对应的处理器对象