**代理模式**

* 概述
* 静态代理
* jdk动态代理
* cglib动态代理

# 概述

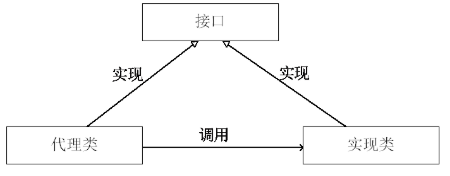
代理模式是一种设计模式，简单说即是在不改变源码的情况下，实现对目标对象的功能扩展，增强业务功能

# 静态代理

## 概述

为某个对象提供一个代理，以控制对这个对象的访问。 代理类和委托类有共同的父类或父接口，这样在任何使用委托类对象的地方都可以用代理对象替代。代理类负责请求的预处理、过滤、将请求分派给委托类处理、以及委托类执行完请求后的后续处理。





PS：由程序员创建或工具生成代理类的源码，再编译代理类。所谓静态也就是在程序运行前就已经存在代理类的字节码文件，代理类和委托类的关系在运行前就确定了

## 代码实现

### 定义接口

**public** **interface** Subject {

**void** doSomething();

}

### 定义目标对象

**public** **class** RealSubject **implements** Subject {

@Override

**public** **void** doSomething() {

System.***out***.println("目标类的业务方法");

}

}

### 定义代理对象

**public** **class** ProxySubject **implements** Subject{

//业务接口

Subject subject;

**public** ProxySubject(Subject subject) {

**this**.subject = subject;

}

**public** **void** before() {

System.***out***.println("执行业务之前的附加功能...");

}

**public** **void** after() {

System.***out***.println("执行业务之后的附加功能...");

}

@Override

**public** **void** doSomething() {

before();

subject.doSomething();

after();

}

}

### 测试

**public** **class** Test {

**public** **static** **void** main(String[] args) {

Subject subject = **new** RealSubject();

ProxySubject proxy = **new** ProxySubject(subject);

proxy.doSomething();

}

}

## 静态代理优缺点

优点：

可以做到不对目标对象进行修改的前提下，对目标对象进行功能的扩展和拦截,这是代理的共有优点。

缺点：

因为代理对象，需要实现与目标对象一样的接口，会导致代理类十分繁多，不易维护，同时一旦接口增加方法，则目标对象和代理类都需要维护

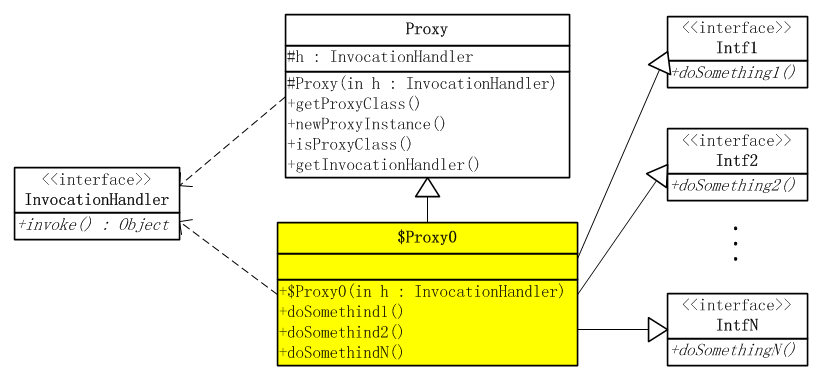
## 练习

需求：在一个类中定义一个方法，返回字符串“abcde”，在不修改源代码的基础上转为大写

# jdk动态代理

## 概述

 JDK动态代理的实现是在运行时，根据一组接口定义，使用Proxy、InvocationHandler等工具类去生成一个代理类和代理类实例,动态代理类克服了proxy需要继承专一的interface接口，并且要实现相应的method的缺陷



PS：图中高亮显示的$Proxy0即为JDK动态代理技术生成的代理类，类名的生成规则是前缀"$Proxy"加上一个序列数。这个类继承Proxy，实现一系列的接口Intf1,Intf2...IntfN

## 实现步骤

1. 创建一个实现了 InvocationHandler 接口的类，必须重写接口里的 invoke（）方法。
2. 创建被代理的类和接口
3. 通过 Proxy 的静态方法 newProxyInsatance(ClassLoader loader,Class[] interfaces,InvocationHandler,handler) 来创建一个代理
4. 通过代理来调用方法

## 代码实现

### 定义接口

**public** **interface** Subject {

**void** doSomething();

}

### 定义目标类

**public** **class** RealSubject **implements** Subject {

@Override

**public** **void** doSomething() {

System.***out***.println("目标类的业务方法");

}

}

### 定义代理工厂

**public** **class** MyInvocationHandler **implements** InvocationHandler{

Object obj;

/\*\*

\* 获取目标对象

\*/

**public** Object getTarget(Object obj) {

**this**.obj = obj;

**return** Proxy.*newProxyInstance*(

obj.getClass().getClassLoader(),//这是找到需要代理的类

obj.getClass().getInterfaces(), //这是找到需要代理类继承的接口

**this**);//this代表的当前的类 也就是 JDKProxy对象

}

@Override

**public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {

// **TODO** Auto-generated method stub

System.***out***.println("执行业务之前添加附加功能.......");

Object result = method.invoke(obj, args);

System.***out***.println("执行业务之后添加附加功能.......");

**return** result;

}

}

### 测试类

**public** **class** Test {

**public** **static** **void** main(String[] args) {

//创建目标对象

Subject subject = **new** RealSubject();

//创建代理对象

Subject proxy = (Subject)**new** MyInvocationHandler().getTarget(subject);

//调用方法

proxy.doSomething();

}

}

## jdk动态代理优缺点

优点:

接口中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（InvocationHandler.invoke）。这样，在接口方法数量比较多的时候，我们可以进行灵活处理，而不需要像静态代理那样每一个方法进行中转

缺点：

JDK 动态代理只能对实现了接口的目标类进行代理，而不实现接口的类就不能使用 JDK 的动态代理

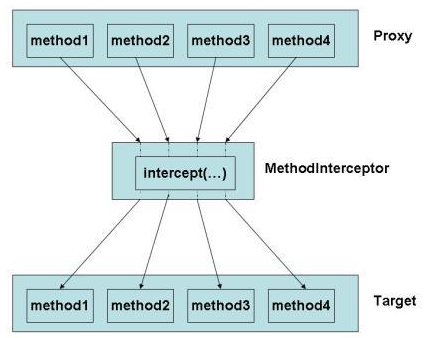
# cglib动态代理

## 概述

1. CGLIB基于ASM实现。提供比反射更为强大的动态特性。使用CGLIB可以非常方便的实现的动态代理
2. 使用 JDK 的 Proxy 实现代理，要求目标类与代理类实现相同的接口。若目标类不存在接口，则无法使用该方式实现。
3. 但对于无接口的类，要为其创建动态代理，就要使用 CGLIB 来实现。CGLIB 代理的生成原理是生成目标类的子类，而子类是增强过的，这个子类对象就是代理对象。所以，使用CGLIB 生成动态代理，要求目标类必须能够被继承，即不能是 final 的类。

## 基本原理

CGLIB动态代理的原理就是用Enhancer生成一个原有类的子类，并且设置好callback到proxy， 则原有类的每个方法调用都会转为调用实现了MethodInterceptor接口的proxy的intercept() 函数,如图所示



**PS:  使用CGLIB代理最核心类Enhancer生成代理对象**

基本代码：

**public** Object createProxy(Class targetClass) {

    Enhancer enhancer = **new** Enhancer();

    enhancer.setSuperclass(targetClass);

    enhancer.setCallback(**new** MethodInterceptorImpl ());

**return** enhancer.create();

}

createProxy方法返回值是targetClass的一个实例的代理。

## 实现步骤

1. 引入CGLib包
2. 定义目标类
3. 定义工厂生成方法拦截器
4. 在测试类中生成代理对象调用方法

## 代码实现

### 引入CGLib包



### 定义目标类

**public** **class** RealSubject **implements** Subject {

@Override

**public** **void** doSomething() {

System.***out***.println("目标类的业务方法");

}

}

### 定义工厂类生成方法拦截器

**public** **class** MycglibFactory **implements** MethodInterceptor {

**private** Object target;// 业务类对象，供代理方法中进行真正的业务方法调用

// 相当于JDK动态代理中的绑定

**public** Object getInstance(Object target) {

**this**.target = target; // 给业务对象赋值

Enhancer enhancer = **new** Enhancer(); // 创建加强器，用来创建动态代理类

enhancer.setSuperclass(**this**.target.getClass()); // 为加强器指定要代理的业务类（即：为下面生成的代理类指定父类）

// 设置回调：对于代理类上所有方法的调用，都会调用CallBack，而Callback则需要实现intercept()方法进行拦截

enhancer.setCallback(**this**);

// 创建动态代理类对象并返回

**return** enhancer.create();

}

// 实现回调方法

**public** Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) **throws** Throwable {

System.***out***.println("预处理——————");

proxy.invokeSuper(obj, args); // 调用业务类（父类中）的方法

System.***out***.println("调用后操作——————");

**return** **null**;

}

}

### 在测试类中生成代理对象调用方法

**public** **class** Test {

**public** **static** **void** main(String[] args) {

//创建目标对象

RealSubject target = **new** RealSubject();

//创建代理对象

RealSubject proxy = (RealSubject) **new** MycglibFactory().getInstance(target);

//代理对象调用方法

proxy.doSomething();

}

}