

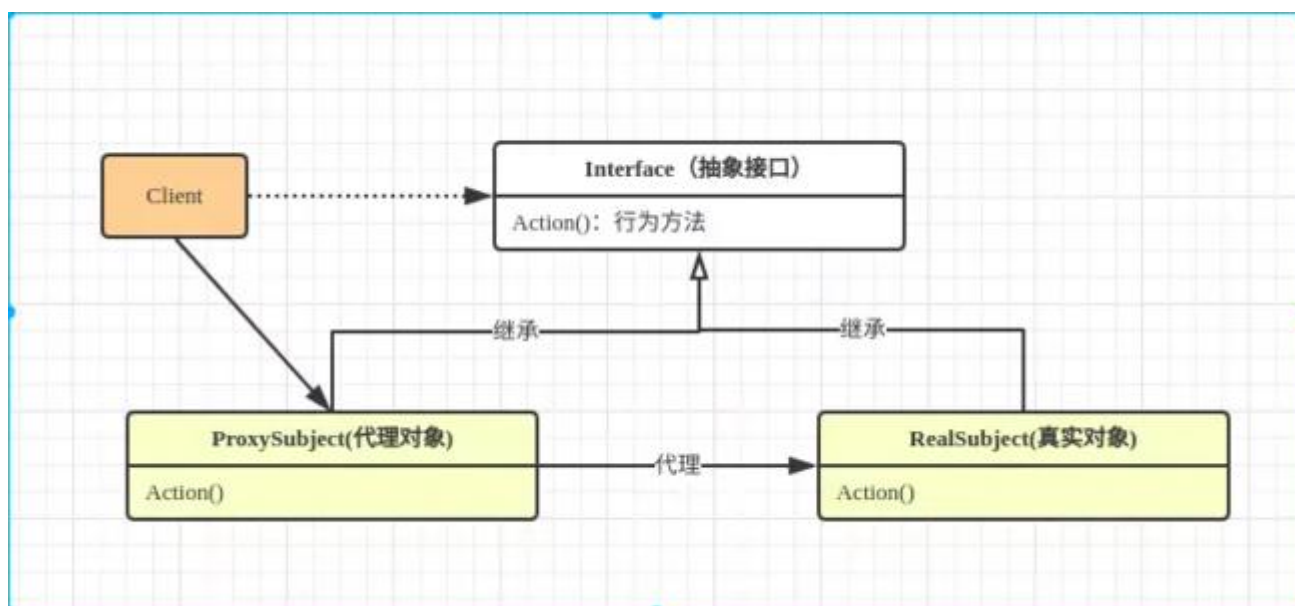
动态代理原理

静态代理

代理模式给某一个对象提供一个代理对象，并由代理对象控制对原对象的引用。通俗的来讲代理模式就是我们生活中常见的中介。比如你按照小卡片上的电话打过去寻求服务，一般不是由本人，可能是一个成年雄性接听电话，然而真正做事的可能是另一个小姐姐。

目的：（1）通过引入代理对象的方式来间接访问目标对象，防止直接访问目标对象给系统带来的不必要复杂性；
（2）通过代理对象对访问进行控制；

代理模式一般会有三个角色：



抽象角色：指代理角色和真实角色对外提供的公共方法，一般为一个接口

真实角色：需要实现抽象角色接口，定义了真实角色所要实现的业务逻辑，以便供代理角色调用。也就是真正的业务逻辑在此。

代理角色：需要实现抽象角色接口，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。将统一的流程控制都放到代理角色中处理！

静态代理在使用时,需要定义接口或者父类,被代理对象与代理对象一起实现相同的接口或者是继承相同父类。一般来说，被代理对象和代理对象是一一对一的关系，当然一个代理对象对应多个被代理对象也是可以的。

静态代理，一对一则会出现时静态代理对象量多、代码量大，从而导致代码复杂，可维护性差的问题，一对多则代理对象会出现扩展能力差的问题。

动态代理

在运行时再创建代理类和其实例，因此显然效率更低。要完成这个场景，需要在运行期动态创建一个Class。JDK提供了 `Proxy` 来完成这件事情。基本使用如下：

```
//抽象角色
interface Api {
    void test(String a);
}

//真实角色
class ApiImpl{
    @Override
    public void test(String a) {
        System.out.println("真实实现: " + a);
    }
}

//创建真实角色实例
ApiImpl api = new ApiImpl();

//JDK动态代理：
Proxy.newProxyInstance(getClass().getClassLoader(),
    new Class[]{Api.class}, //JDK实现只能代理接口
    new InvocationHandler() {
        @Override
        public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
            //执行真实对象方法
            return method.invoke(api, args);
        }
    });
```

实际上，`Proxy.newProxyInstance` 会创建一个Class，与静态代理不同，这个Class不是由具体的.java源文件编译而来，即没有真正的文件，只是在内存中按照Class格式生成了一个Class。

```
String name = Api.class.getName()+"$Proxy0";
//生成代理指定接口的Class数据
byte[] bytes = ProxyGenerator.generateProxyClass(name, new Class[]{Api.class});
FileOutputStream fos = new FileOutputStream("lib/" + name+".class");
fos.write(bytes);
fos.close();
```

然后可以在生成的文件中查看我们的代理类：

```
com.enjoy.lib.Api$Proxy0.class x
Decompiled .class file, bytecode version: 49.0 (Java 5)

56         } catch (RuntimeException | Error var2) {
57             throw var2;
58         } catch (Throwable var3) {
59             throw new UndeclaredThrowableException(var3);
60         }
61     }
62
63     static {
64         try {
65             m1 = Class.forName("java.lang.Object").getMethod(s: "equals", Class.forName("java.lang.Object"));
66             m2 = Class.forName("java.lang.Object").getMethod(s: "toString");
67             m3 = Class.forName("com.enjoy.lib.Api").getMethod(s: "test", Class.forName("java.lang.String"));
68             m0 = Class.forName("java.lang.Object").getMethod(s: "hashCode");
69         } catch (NoSuchMethodException var2) {
70             throw new NoSuchMethodError(var2.getMessage());
71         } catch (ClassNotFoundException var3) {
72             throw new NoClassDefFoundError(var3.getMessage());
73         }
74     }
75 }
76
```

在初始化时，获得 `method` 备用。而这个代理类中所有方法的实现变为：

```
public final void test(String var1) throws {
    try {
        super.h.invoke(o: this, m3, new Object[]{var1});
    } catch (RuntimeException | Error var3) {
        throw var3;
    } catch (Throwable var4) {
        throw new UndeclaredThrowableException(var4);
    }
}
```

这里的 `h` 其实就是 `InvocationHandler` 接口，所以我们在使用动态代理时，传递的 `InvocationHandler` 就是一个监听，在代理对象上执行方法，都会由这个监听回调出来。