代理模式

基本概念

代理模式在不改变原始类(或者说，被代理类)的结构和功能的前提下，通过添加一个代理类来拓展原始类的功能

```
class User {}

interface IUserService {
    User saveUser(User user);
    User updateUser(User user);
}

public class UserService implements IUserService{
    @Override
    public User saveUser(User user) {
        /* 调用UserDao.saveUser()方法将User保存至底层存储数据库中 */
        return null;
    }

    @Override
    public User updateUser(User user) {
        /* 调用UserDao.updateUser()方法将User更新至底层存储数据库中 */
        return null;
    }
}
```

这是一个非常简单的UserService，用于保存和更新用户数据库数据

但是天有不测风云，老板提出了新的需求: 虽然底层MySQL的binlog能够记录数据的创建与更新，但是查询起来很不方便，所以要求在重要数据创建或者更新后，记录更新前与更新后的数据，便于后续的审计工作

程序正常运行，一切都显得岁月静好，甚至还想点被卡布奇诺享受一下人生

绕头想了想，这不就是操作日志的记录吗。思量再三，决定使用Elasticsearch来保存操作日志，并将对应的类和方法设计完毕，但是方法调用该放在哪里?

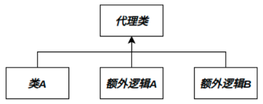
如果放入进UserService中，那么两个不太相关的类就耦合在了一起，后续进行拓展时就会很麻烦

再后来，你使用了代理模式解决了这个问题

```
class UserServiceProxy implements IUserService {
    private UserService userService;

    public UserServiceProxy(UserService userService) {
        // 通过依赖注入的方式降低耦合性
        this.userService = userService;
    }

    @Override
    public User saveUser(User user) {
        // 实际的创建用户动作仍然由UserService完成，只不过在后面添加一些了一些逻辑而已
        User createdUser = userService.saveUser(user);
        /* 将createdUser序列化成Json后，并添加一些额外描述信息发送给ES */
        return createdUser;
    }

    @Override
    public User updateUser(User user) {
        User updatedUser = userService.updateUser(user);
        /* 将updatedUser序列化成Json后，并添加一些额外描述信息发送给ES */
        return updatedUser;
    }
}
```

为了不变动原有UserService类的行为，以及使用UserService的代码，UserServiceProxy 需实现相同的接口，并实现接口中的所有方法

实际的方法调用其实还是UserService，只不过UserServiceProxy加了点儿额外逻辑而已



如果原有的类A并没有实现相关的接口，或者说类A的接口不对外暴露，那么可以让代理类直接继承类A，重写类A需要做拓展的相关方法

实际上，代理模式就像一个夹心饼干一样

这种类似于"夹心饼干"的逻辑，在不同的语言以及不同的框架下随处可见

代理模式的应用

代理模式通常会用在为原有的业务逻辑添加非业务功能上，例如限流，鉴权，日志记录，事务，统计，监控等场景下

对于框架设计而言，提供AOP的切面入口，通常也会采用代理模式实现

Python代理模式

在Python中，两个比较著名的Web框架: Django和Flask，均使用了代理模式作为请求前以及请求后的逻辑处理

Flask

before_request　在View函数调用之前需调用的函数列表

after_request　在View函数调用之后需调用的函数列表，常用此函数来注册日志记录方法

```
import time
from flask import Flask, jsonify, g

app = Flask("ZMind")


@app.before_request
def record_request_time():
    g.start_time = time.time()


@app.after_request
def record_request_log(response, *args, **kwargs):
    start_time = getattr(g, "start_time", None)

    if start_time:
        print("elapsed: {}".format(time.time() - start_time))

    return response


if __name__ == "__main__":
    app.run(port=13955, debug=True)
```

Django　Django则采用定义于settings.py中的MIDDLEWARE数组实现的面向切面编程，用户需在自定义的中间件中实现process_request以及process_response两个方法

动态代理

实现代理模式的方式通常有两种

实现被代理类的接口，将被代理类包裹在代理类中　这样一来代理类必须要实现接口中的所有方法，大多数场景中可能只需要在少数方法上进行拓展

直接使用继承的方式在原有方法上进行拓展

使用继承会导致子类和父类的代码存在割裂感，因为大部分情况下不会覆盖父类所有方法，而是少量的方法

并且采用继承的实现每次只能为一个类添加额外行为

静态代理模式

静态代理模式一个比较大的局限所在就是当系统中存在较多需要被代理的类时，需要事先创建出大量的代理类，然后在原有使用被代理类的地方替换成代理类　一是工作量很大，二是系统中会存在大量相似的代码

当系统中存在大量需要被代理的类时，通常会采用反射的机制，在运行时创建出代理类，并替换掉原有类，通常称为动态代理

```
class OperationLogService {
    public OperationLogService() {}
    public void insertOperationLog(String objectJson) {/* 逻辑省略 */}
}

class OperationLogProxy {
    private OperationLogService operationLogService;

    public OperationLogProxy() {
        this.operationLogService = new OperationLogService();
    }

    private class DynamicProxyHandler implements InvocationHandler {

        private Object proxiedObject;

        public DynamicProxyHandler(Object proxiedObject) { this.proxiedObject = proxiedObject; }

        @Override
        public Object invoke(Object o, Method method, Object[] objects) throws Throwable {
            Object result = method.invoke(proxiedObject);
            // 插入日志记录，写入日志 */
            operationLogService.insertOperationLog("");
            return result;
        }
    }

    public Object createProxy(Object proxiedObject) {
        Class[] interfaces = proxiedObject.getClass().getInterfaces();
        DynamicProxyHandler handler = new DynamicProxyHandler(proxiedObject);
        return Proxy.newProxyInstance(proxiedObject.getClass().getClassLoader(), interfaces, handler);
    }

    public static void main(String[] args) {
        // 使用动态代理类
        OperationLogProxy proxy = new OperationLogProxy();
        IUserService userService = (IUserService) proxy.createProxy(new UserService());
    }
}
```

动态代理通常使用反射的机制实现，但是也可以使用其它动态加载字节码的手段实现

动态代理在Python和Golang中都不常见，Python由于本身是弱类型语言的关系，可以随意的动态加载。而Golang本身则弱化了反射机制，更多的还是使用静态代理