1. 设计模式之代理模式（Proxy Pattern）

https://www.jianshu.com/p/4de2e59e763e

1.1 What:

> 为其他对象提供一种代理以控制对这个对象的访问。

1.2 Why:

优点：

1.增强目标对象。可以在执行目标对象方法的前后或者其他地方加上验证、日志等等代码；（Spring框架中的 AOP）
2.将调用对象和被调用对象分离，一定程度上降低了耦合度。扩展性好；
3.保护目标对象；
4.职责清晰。目标对象就是实现实际的业务逻辑，不用关心其他非本职责的事务，通过后期的代理完成，附带的结果就是编程简洁清晰。

缺点：

1.对象与对象调用之间增加了一层代理，可能会导致执行的速度变慢；
2.实现代理的代码有时会很复杂，添加了额外的工作量；
3.增加系统的复杂度。

Where:

1.需要保护目标类，不希望直接被调用；
2.设置权限。类似Spring AOP的使用。

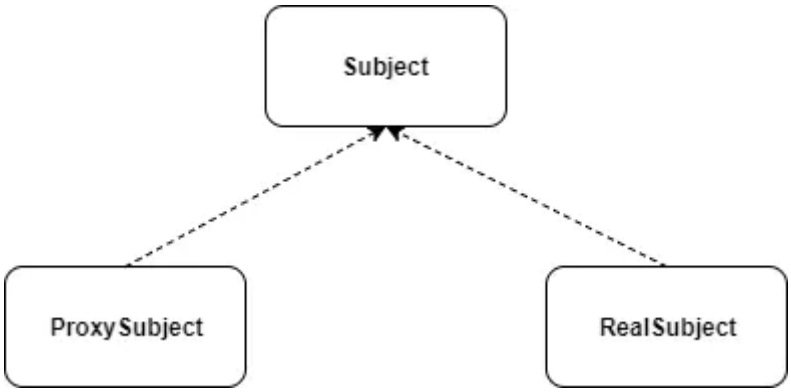
How:

代理模式有三个角色：

Subject（抽象角色）： 通过接口或抽象类声明真实角色实现的业务方法。

RealSubject（目标角色）： 实现抽象角色，定义目标角色所要实现的业务逻辑，供代理角色调用。

Proxy（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。



1.3 代理模式有三种实现方式：

静态代理

代理类在程序运行前就编译好，因此称为静态代理。代理对象和被代理对象都要实现相同的接口或者继承相同的父类(除Object之外)，一旦接口或者父类添加、修改方法，子类都要统一更改，违背开闭原则，因此静态代理具有一定的局限性。

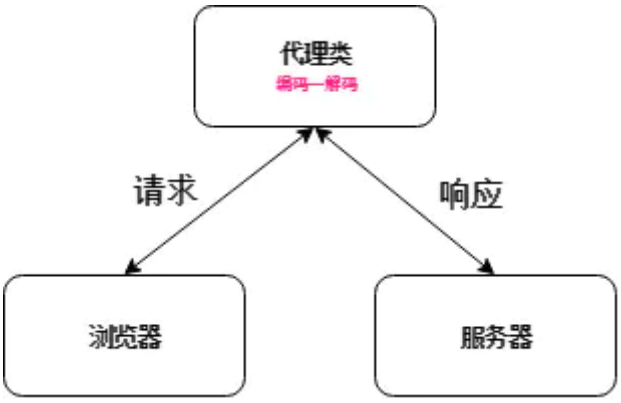
动态代理

利用JDK的API,动态的在内存中构建代理对象。不需要继承父类，可扩展性高。

Cglib代理

上面两种代理方式目标对象都需要实现接口，但有时候并不是所有目标对象都要实现接口，因此可以使用Cglib框架实现普通类的代理。

以下都是以浏览器访问服务器的简单举例：模拟浏览器向服务器发生消息，经过代理转码后服务器收到请求消息并响应信息，经过代理记录日志，最后浏览器才接收到响应信息。



1.4 静态代理示例

IHttpInvoke(抽象角色):

```
public interface IHttpInvoke {
    String invoke(String request);
}
```

Server(目标对象):

```
public class Server implements IHttpInvoke {

    Logger logger = Logger.getLogger(String.valueOf(getClass()));

    @Override
    public String invoke(String request) {
        String response = "没有的，不存在的!";
        return response;
    }
}
```

HttpInvokeProxy(代理对象)

```
public class HttpInvokeProxy implements IHttpInvoke {

    private Logger logger = Logger.getLogger(String.valueOf(getClass()));

    private IHttpInvoke iHttpInvoke;

    public HttpInvokeProxy(IHttpInvoke iHttpInvoke) {
        this.iHttpInvoke = iHttpInvoke;
    }
```

```
    @Override
    public String invoke(String request) {
        String req = before(request);
        String response = iHttpInvoke.invoke(req);
        after(response);
        return response;
    }

    public String before(String request){
        logger.info("请求数据: " + request);
        byte[] req = request.getBytes();
        String requestData = null;
        try {
            requestData = new String(req,"GBK");
            logger.info("转码成功，返回请求数据");
        } catch (UnsupportedEncodingException e) {
            e.printStackTrace();
        }
        return requestData;
    }

    public void after(String response){
        logger.info("响应数据: " + response);
    }
}
```

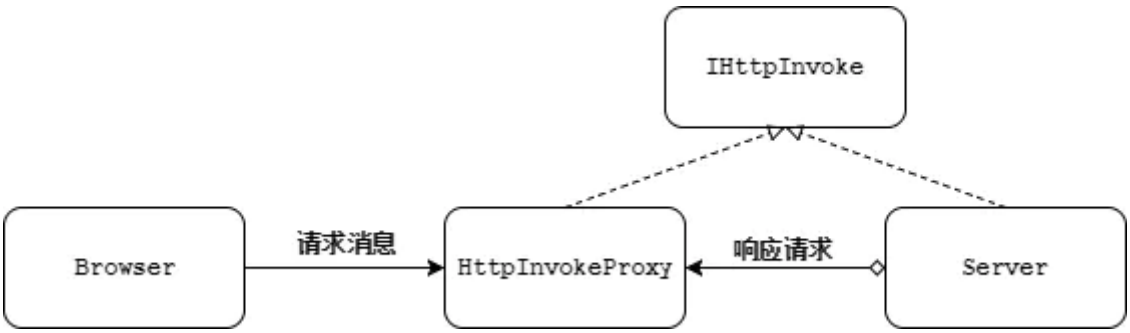
Browser(浏览器客户端)

```
public class Browser {
    public static void main(String[] args) {
        String request = "给我一个女朋友！";
        IHttpInvoke httpInvokeProxy = new HttpInvokeProxy(new Server());
        httpInvokeProxy.invoke(request);
    }
}
```

输出结果

```
五月 16, 2019 10:25:02 上午 Structural.ProxyPattern.StaticProxy.HttpInvokeProxy
before
信息: 请求数据:给我一个女朋友!
五月 16, 2019 10:25:03 上午 Structural.ProxyPattern.StaticProxy.HttpInvokeProxy
before
信息: 转码成功，返回请求数据
五月 16, 2019 10:25:03 上午 Structural.ProxyPattern.StaticProxy.HttpInvokeProxy
after
信息: 响应数据: 没有的，不存在的!
```

静态代理的UML



1.5 动态代理示例

IHttpInvoke(抽象角色):

```
public interface IHttpInvoke {
    String invoke(String request);
}
```

Server(目标对象):

```
public class Server implements IHttpInvoke {

    Logger logger = Logger.getLogger(String.valueOf(getClass()));

    @Override
    public String invoke(String request) {
        String response = "没有的，不存在的!";
        return response;
    }
}
```

HttpInvokeProxy(代理对象):

```
public class HttpInvokeProxy implements InvocationHandler {

    private Logger logger = Logger.getLogger(String.valueOf(getClass()));

    private Object obj;

    public HttpInvokeProxy(Object obj) {
        this.obj = obj;
    }

    public String before(String request){
        logger.info("请求数据：" + request);
        byte[] req = request.getBytes();
        String requestData = null;
        try {
            requestData = new String(req,"GBK");
            logger.info("转码成功，返回请求数据");
        } catch (UnsupportedEncodingException e) {
            e.printStackTrace();
        }
        return requestData;
    }
```

```
    public void after(String response){
        logger.info("响应数据: " + response);
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        before((String) args[0]);
        Object object = method.invoke(obj,args);
        after((String) object);
        return object;
    }
}
```

Browser(浏览器客户端):

```
public class Browser {
    public static void main(String[] args) {
        String request = "给我一个女朋友! ";
        Server server = new Server();
        InvocationHandler invocationHandler = new HttpInvokeProxy(server);
        Class cls = server.getClass();
        IHttpInvoke httpInvoke = (IHttpInvoke)
Proxy.newProxyInstance(cls.getClassLoader(),cls.getInterfaces(),invocationHandle
r);
        httpInvoke.httpInvoke(request);
    }
}
```

输出结果:

```
五月 16, 2019 10:53:01 上午 Structural.ProxyPattern.DynamicProxy.HttpInvokeProxy
before
信息: 请求数据: 给我一个女朋友!
五月 16, 2019 10:53:02 上午 Structural.ProxyPattern.DynamicProxy.HttpInvokeProxy
before
信息: 转码成功，返回请求数据
五月 16, 2019 10:53:02 上午 Structural.ProxyPattern.DynamicProxy.HttpInvokeProxy
after
信息: 响应数据: 没有的，不存在的!
```

1.6 Cglib代理

```
Cglib是一个强大的高性能的代码生成包,它可以在运行期扩展java类与实现java接口.它广泛的被许多AOP
的框架使用,例如Spring AOP和synaop,为他们提供方法的interception(拦截)
```

使用Cglib代理需要引入Cglib的jar包，而且还需要引入asm-all的jar包。注意两个包的版本号，因为不同的版本可能不兼容导致报错。

```
<dependency>
    <groupId>cglib</groupId>
    <artifactId>cglib</artifactId>
    <version>2.2.2</version>
</dependency>
<dependency>
    <groupId>asm</groupId>
    <artifactId>asm-all</artifactId>
    <version>3.3</version>
</dependency>
```

Server(目标对象):

```
public class Server{

    Logger logger = Logger.getLogger(String.valueOf(getClass()));

    public String receiveRequest(String request) {
        String response = "没有的，不存在的!";
        return response;
    }
}
```

HttpInvokeProxy代理类

```
public class HttpInvokeProxy implements MethodInterceptor {

    private Logger logger = Logger.getLogger(String.valueOf(getClass()));

    private Object obj;

    public HttpInvokeProxy() {
    }

    public HttpInvokeProxy(Object obj) {
        this.obj = obj;
    }

    public String before(String request){
        logger.info("请求数据: " + request);
        byte[] req = request.getBytes();
        String requestData = null;
        try {
            requestData = new String(req,"GBK");
            logger.info("转码成功，返回请求数据");
        } catch (UnsupportedEncodingException e) {
            e.printStackTrace();
        }
        return requestData;
    }

    public void after(String response){
        logger.info("响应数据: " + response);
    }

    public Object getProxy(){
```

```
        //1.工具类
        Enhancer enhancer = new Enhancer();
        //2.设置父类
        enhancer.setSuperclass(Server.class);
        //3.设置回调函数
        enhancer.setCallback(this);
        //4.创建子类(代理对象)
        Object object = enhancer.create();
        return object;
    }

    @Override
    public Object intercept(Object o, Method method, Object[] objects,
MethodProxy methodProxy) throws Throwable {
        before((String) objects[0]);
        Object obj = methodProxy.invokeSuper(o,objects);
        after((String) obj);
        return obj;
    }
}
```

Browser(浏览器客户端):

```
public class Browser {
    public static void main(String[] args) {
        String request = "给我一个女朋友! ";
        Server server = new Server();
        Server proxy = (Server) new HttpInvokeProxy(server).getProxy();
        proxy.receiveRequest(request);
    }
}
```

输出结果:

```
五月 16, 2019 11:52:42 上午 Structural.ProxyPattern.CglibProxy.HttpInvokeProxy
before
信息: 请求数据:给我一个女朋友!
五月 16, 2019 11:52:42 上午 Structural.ProxyPattern.CglibProxy.HttpInvokeProxy
before
信息: 转码成功，返回请求数据
五月 16, 2019 11:52:42 上午 Structural.ProxyPattern.CglibProxy.HttpInvokeProxy
after
信息: 响应数据: 没有的，不存在的!
```

1.7 总结

静态代理 作为原始的代理模式设计，代理对象和目标对象都实现同一个接口，在代理对象中指向的是目标对象的实例，这样对外暴露的是代理对象而真正调用的是目标对象。其优点是保护目标对象，提高安全性；但缺点也显而易见，不同的接口要有不同的代理类实现，代码量增加，系统冗余。

动态代理 弥补了静态代理需要子类继承或者实现父类的缺点，利用反射机制创建对象，进一步降低了耦合度。动态代理必须依赖接口的实现，但并不是所有的目标对象都会继承父类或者实现接口。

Cglib代理 弥补了JDK动态代理不足。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，来完成动态代理的实现。但由于CGLib采用动态创建子类的方法，对于final方法，无法进行代理。

每种代理方式都有优缺点，我们应该根据实际的开发场景选择动态代理或者使用Cglib代理。

了解更多设计模式：

设计模式系列

参考资料：

https://www.cnblogs.com/cenyu/p/6289209.html