代理模式

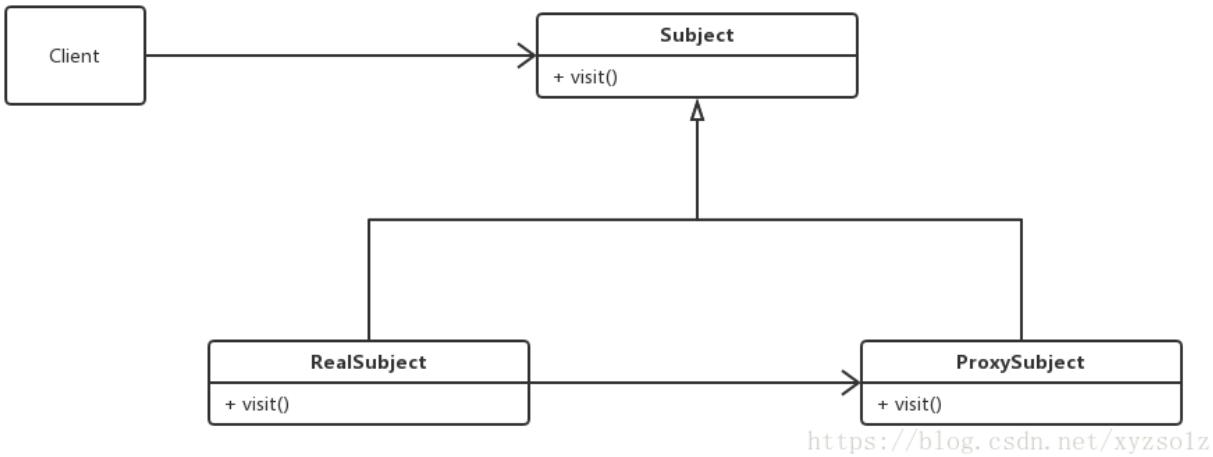
1.代理模式的定义

为其他对象提供一种代理以控制对这个对象的访问。

2.代理模式的使用场景

当无法或不想直接访问某个对象或访问某个对象存在困难时可以通过一个代理对象来间接访问，为了保证客户端的透明性，委托对象与代理对象需要实现相同的接口。

3.代理模式的UML类图



角色介绍：

- Subject:抽象主题类。

  该类主要职责是声明真实主题与代理的共同接口方法，该类既可以是一个抽象类也可以是一个接口。

- RealSubject:真实主题类。

  该类也称为被委托类或被代理类，该类定义了代理所有表示的真实对象，由其执行具体的业务逻辑方法，而客户端则通过代理类间接地调用真实主题类中定义的方法

- ProxySubject:代理类。

  该类也称为委托类或代理类，该类持有一个对真实主题类的引用，在其所实现的接口方法中调用真实主题类中相应的接口方法执行，以此起到代理的作用。

- Client:客户类，即使用代理类的类型。

4.代理类的简单实现

小明以前在公司上班时，就遇到过被老板拖欠工资甚至克扣工资的情况，这种情况下小明还是通过法律途径来解决问题，一旦小明选择走法律途径解决该纠纷，那么不可避免地就需要请一个律师来作为自己的诉讼代理人，我们将诉讼

的流程抽象在一个接口类中。

诉讼接口：

```
package staticproxy;

//诉讼接口类
public interface ILawsuit {
        // 提交申请
        void submit();

        // 进行举证
        void burden();

        // 开始辩论
        void defend();

        // 诉讼完成
        void finish();
}
```

4个方法很简单，都是诉讼流程

具体诉讼人：

```
package staticproxy;

//具体诉讼人
public class XiaoMing implements ILawsuit {

        @Override
        public void submit() {
                // 老板欠小明工资 小明只好申请仲裁
                System.out.println("老板拖欠工资！特此申请仲裁");
        }

        @Override
        public void burden() {
                // 小明证据充足，不怕告不赢
                System.out.println("这是合同和过去一年的银行流水!");
        }

        @Override
        public void defend() {
                // 铁证如山，辩护也没什么好说的
                System.out.println("证据确凿!,不需要再说什么!");
        }

        @Override
        public void finish() {
                // 结果也是肯定赢了
                System.out.println("诉讼成功!判决老板即日起七天内结算工资");
        }
```

```
}
```

如上所示，该类实现了ILawsuit并对其中4个方法作出具体的实现逻辑，逻辑很简单，都只是输出一段话而已，当然，小明自己是不会去大官司的，于是小明请了一个律师代替自己诉讼。

代理律师：

```
package staticproxy;

//代理律师
public class Lawyer implements ILawsuit {

        private ILawsuit mLawsuit;// 持有一个具体被代理者的引用

        public Lawyer(ILawsuit lawsuit) {
                mLawsuit = lawsuit;
        }

        @Override
        public void submit() {
                mLawsuit.submit();
        }

        @Override
        public void burden() {
                mLawsuit.burden();
        }

        @Override
        public void defend() {
                mLawsuit.burden();
        }

        @Override
        public void finish() {
                mLawsuit.finish();
        }

}
```

律师类表示代理者律师，在该类里面会持有一个被代理者（这里也就是上面的Aige类）的引用，律师所执行的方法实质就是简单低调用被代理者中的方法，下面来看看客户类中具体的调用执行关系。

客户类：

```
package staticproxy;

//客户类
public class Client {
        public static void main(String[] args) {
                // 构造一个小明
                ILawsuit xiaoming = new XiaoMing();
                // 构造一个代理律师并将小明作为构造参数传递出去
```

```
                ILawsuit lawyer = new Lawyer(xiaoming);

                // 律师提交诉讼申请
                lawyer.submit();
                // 律师进行举证
                lawyer.burden();
                // 律师代替小明进行辩护
                lawyer.defend();
                // 完成诉讼
                lawyer.finish();
        }
}
```

其实代理模式也很简单，其主要还是一种委托机制，真实对象将方法的执行委托给代理对象，而且委托得干净利落毫不做作，这也是为什么代理模式也称为委托模式的原因，相信大家不难理解。其实气门的代理类完全可以代理多个被代理类，就像上面的例子一样，一个律师可以代理多个人打官司，这是没有任何问题的，而具体到底是代理的哪个人，这就要看代理类中所持有的实际对象类型，上述的例子中实际对象类型是Xiaoming ,也就是代理的XIaoming。

代理可以分为两大部分，一是静态代理，二是动态代理。静态代理。静态代理如上述示例那样，代理者的代码有程序员自己或通过一些自动化工具生成固定的代码在对其进行编译，也就是说在我们的代码运行前代理类的class编译文件就已存在；而动态代理则与静态代理相反，通过反射机制动态地生成代理者的对象，也就是说我们在code阶段压根不需要知道代理谁，代理谁我们将会在执行阶段决定。而Java也给我们提供了一个便捷的动态代理接口InvocationHandler，实现该接口需要重写其调用方法invoke.

我们主要通过invoke方法来调用具体的被代理方法，也就是真实的方法。动态代理可以使我们的代码逻辑更简洁，不过在这之前我们得先完善动态代理类。

动态代理类：

```
package dynamicproxy;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

public class DynamicProxy implements InvocationHandler {

        private Object obj;// 被代理的类引用

        public DynamicProxy(Object object) {
                this.obj = object;
        }

        @Override
        public Object invoke(Object proxy, Method method, Object[] args)
                        throws Throwable {
                // 调用被代理类对象的方法
                Object result = method.invoke(obj, args);
                return result;
        }

}
```

如上代码所示，我们声明一个Object的引用，该引用将指向被代理类，而我们调用被代理类的具体方法则在invoke方法中执行，也就是说我们原来有代理类所做的工作现在有InvocationHandler来处理，不再需要关心到底代理谁。

修改后的客户端：

```
package dynamicproxy;

import java.lang.reflect.Proxy;


public class Client {
        public static void main(String[] args) {
                //构造一个小明
                ILawsuit xiaoming=new XiaoMing();

                //构造一个动态代理
                DynamicProxy proxy=new DynamicProxy(xiaoming);

                //获取被代理类小明的ClassLoader
                ClassLoader loader=xiaoming.getClass().getClassLoader();

                //动态构造一个代理者律师

            ILawsuit lawyer=(ILawsuit)Proxy.newProxyInstance(loader, new Class[]{ILawsuit.class}, p
                    //律师提交诉讼申请
                lawyer.submit();
                //律师进行举证
                lawyer.burden();
                //律师代替小明进行辩护
                lawyer.defend();
                //完成诉讼
                lawyer.finish();
        }

}
```

由此可见动态代理通过一个代理类来代理N多个被代理类，其实质时对代理者与被代理者进行解耦，使两者直接没有直接的耦合关系。相对而言静态代理则只能给定接口下的实现类做代理，如果接口不同那么就需要重新定义不同代理类，较为复杂，但是静态代理更符合面向对象原则。在开发是具体使用哪种方式来实现代理，就看自己的偏好了。

静态代理和动态代理是从code方面来区分代理模式的两种方式，我们也可以从起始于适用范围来区分不同类型的代理实现：

- 远程代理（Remote Proxy）:为某个对象在不同的内存地址空间提供局部代理。贷系统可以将Server部分的实现隐藏，以便Client可以不考虑Server的存在。
- 虚拟代理（Virtual Proxy）:使用一个代理对象表示一个十分耗资源的对象并在真正需要时才创建。
- 保护代理（Protection Proxy）：使用代理控制对原始对象的访问。该类型的代理常被用于原始对象有不同访问权限的情况。
- 只能引用（Smart Reference）：在访问原始对象的时执行一些自己的附加操作并对指向原始对象的引用计数。

这里注意的是，静态代理和动态代理都可以应用于上述4种情形，两者是各自独立的变化。