代理模式

在代理模式（Proxy Pattern）中，一个类代表另一个类的功能。这种类型的设计模式属于结构型模式。

在代理模式中，我们创建具有现有对象的对象，以便向外界提供功能接口。

介绍

意图：为其他对象提供一种代理以控制对这个对象的访问。

主要解决：在直接访问对象时带来的问题，比如说：要访问的对象在远程的机器上。在面向对象系统中，有些对象由于某些原因（比如对象创建开销很大，或者某些操作需要安全控制，或者需要进程外的访问），直接访问会给使用者或者系统结构带来很多麻烦，我们可以在访问此对象时加上一个对此对象的访问层。

何时使用：想在访问一个类时做一些控制。

如何解决：增加中间层。

关键代码：实现与被代理类组合。

应用实例： 1、Windows 里面的快捷方式。 2、猪八戒去找高翠兰结果是孙悟空变的，可以这样理解：把高翠兰的外貌抽象出来，高翠兰本人和孙悟空都实现了这个接口，猪八戒访问高翠兰的时候看不出来这个是孙悟空，所以说孙悟空是高翠兰代理类。 3、买火车票不一定在火车站买，也可以去代售点。 4、一张支票或银行存单是账户中资金的代理。支票在市场交易中用来代替现金，并提供对签发人账号上资金的控制。 5、spring aop。

优点： 1、职责清晰。 2、高扩展性。 3、智能化。

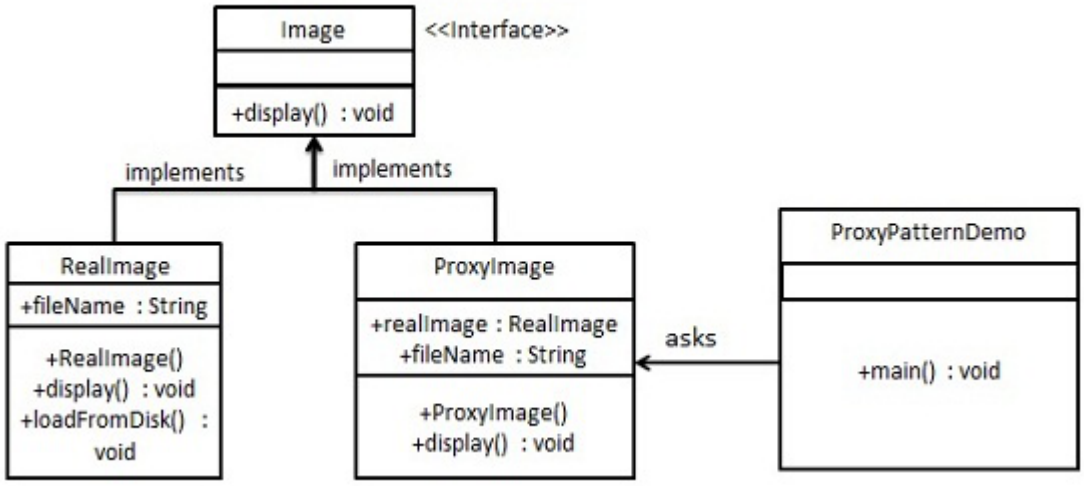
缺点： 1、由于在客户端和真实主题之间增加了代理对象，因此有些类型的代理模式可能会造成请求的处理速度变慢。 2、实现代理模式需要额外的工作，有些代理模式的实现非常复杂。

使用场景：按职责来划分，通常有以下使用场景： 1、远程代理。 2、虚拟代理。 3、Copy-on-Write 代理。 4、保护（Protect or Access）代理。 5、Cache代理。 6、防火墙（Firewall）代理。 7、同步化（Synchronization）代理。 8、智能引用（Smart Reference）代理。

注意事项： 1、和适配器模式的区别：适配器模式主要改变所考虑对象的接口，而代理模式不能改变所代理类的接口。 2、和装饰器模式的区别：装饰器模式为了增强功能，而代理模式是为了加以控制。

实现

我们将创建一个 *Image* 接口和实现了 *Image* 接口的实体类。*ProxyImage* 是一个代理类，减少 *RealImage* 对象加载的内存占用。*ProxyPatternDemo*，我们的演示类使用 *ProxyImage* 来获取要加载的 *Image* 对象，并按照需求进行显示。



反馈

步骤 1

创建一个接口。

Image.java

```
public interface Image {
   void display();
}
```

步骤 2

创建实现接口的实体类。

RealImage.java

```
public class RealImage implements Image {

   private String fileName;

   public RealImage(String fileName){
      this.fileName = fileName;
      loadFromDisk(fileName);
   }

   @Override
   public void display() {
      System.out.println("Displaying " + fileName);
   }

   private void loadFromDisk(String fileName){
      System.out.println("Loading " + fileName);
   }
}
```

ProxyImage.java

```
public class ProxyImage implements Image{

   private RealImage realImage;
   private String fileName;

   public ProxyImage(String fileName){
```

```
      this.fileName = fileName;
   }

   @Override
   public void display() {
      if(realImage == null){
         realImage = new RealImage(fileName);
      }
      realImage.display();
   }
}
```

步骤 **3**

当被请求时，使用 *ProxyImage* 来获取 *RealImage* 类的对象。

ProxyPatternDemo.java

```
public class ProxyPatternDemo {

   public static void main(String[] args) {
      Image image = new ProxyImage("test_10mb.jpg");

      // 图像将从磁盘加载
      image.display();
      System.out.println("");
      // 图像不需要从磁盘加载
      image.display();
   }
}
```

步骤 **4**

执行程序，输出结果：

```
Loading test_10mb.jpg
Displaying test_10mb.jpg

Displaying test_10mb.jpg
```

□ 享元模式　　责任链模式 □

2 篇笔记　□ 写笔记

JDK 自带的动态代理

- java.lang.reflect.Proxy:生成动态代理类和对象；
- java.lang.reflect.InvocationHandler（处理器接口）：可以通过invoke方法实现对真实角色的代理访问。

每次通过 Proxy 生成的代理类对象都要指定对应的处理器对象。

代码：

a) 接口：Subject.java

```
**
 * @author gnehcgnaw
 * @date 2018/11/5 19:29
 */
public interface Subject {
    public int sellBooks();

    public String speak();
}
```

b)真实对象：RealSubject.java

```
/**
 * @author gnehcgnaw
 * @date 2018/11/5 18:54
 */
public class RealSubject implements Subject{
    @Override
    public int sellBooks() {
        System.out.println("卖书");
        return 1 ;
    }

    @Override
    public String speak() {
        System.out.println("说话");
        return "张三";
    }
}
```

c)处理器对象：MyInvocationHandler.java

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

/**
 * 定义一个处理器
 * @author gnehcgnaw
 * @date 2018/11/5 19:26
 */
public class MyInvocationHandler implements InvocationHandler {
    /**
     * 因为需要处理真实角色，所以要把真实角色传进来
     */
    Subject realSubject ;

    public MyInvocationHandler(Subject realSubject) {
        this.realSubject = realSubject;
    }

    /**
     *
```

```
     * @param proxy    代理类
     * @param method    正在调用的方法
     * @param args      方法的参数
     * @return
     * @throws Throwable
     */
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        System.out.println("调用代理类");
        if(method.getName().equals("sellBooks")){
            int invoke = (int)method.invoke(realSubject, args);
            System.out.println("调用的是卖书的方法");
            return invoke ;
        }else {
            String string = (String) method.invoke(realSubject,args) ;
            System.out.println("调用的是说话的方法");
            return  string ;
        }
    }
}
```

d)调用端：Main.java

```
import java.lang.reflect.Proxy;

/**
 * 调用类
 * @author gnehcgnaw
 * @date 2018/11/7 20:26
 */
public class Client {
    public static void main(String[] args) {
        //真实对象
        Subject realSubject =  new RealSubject();

        MyInvocationHandler myInvocationHandler = new MyInvocationHandler(realSubject);
        //代理对象
        Subject proxyClass = (Subject) Proxy.newProxyInstance(ClassLoader.getSystemClassLoad
er(), new Class[]{Subject.class}, myInvocationHandler);

        proxyClass.sellBooks();

        proxyClass.speak();
    }
}
```

gnehcgnaw 7个月前 (11-13)

Cglib 动态代理是针对代理的类, 动态生成一个子类, 然后子类覆盖代理类中的方法, 如果是private或是final类修饰的方法,则不会被重写。

CGLIB是一个功能强大，高性能的代码生成包。它为没有实现接口的类提供代理，为JDK的动态代理提供了很好的补充。通常可以使用Java的动态代理创建代理，但当要代理的类没有实现接口或者为了更好

的性能，CGLIB是一个好的选择。

CGLIB作为一个开源项目，其代码托管在github，地址为：https://github.com/cglib/cglib

需要代理的类:

```
package cn.cpf.pattern.structure.proxy.cglib;
public class Engineer {
    // 可以被代理
    public void eat() {
        System.out.println("工程师正在吃饭");
    }

    // final 方法不会被生成的字类覆盖
    public final void work() {
        System.out.println("工程师正在工作");
    }

    // private 方法不会被生成的字类覆盖
    private void play() {
        System.out.println("this engineer is playing game");
    }
}
```

CGLIB 代理类:

```
package cn.cpf.pattern.structure.proxy.cglib;

import net.sf.cglib.proxy.Enhancer;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;
import java.lang.reflect.Method;

public class CglibProxy implements MethodInterceptor {
    private Object target;

    public CglibProxy(Object target) {
        this.target = target;
    }

    @Override
    public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodPro
xy) throws Throwable {
        System.out.println("###   before invocation");
        Object result = method.invoke(target, objects);
        System.out.println("###   end invocation");
        return result;
    }

    public static Object getProxy(Object target) {
        Enhancer enhancer = new Enhancer();
        // 设置需要代理的对象
        enhancer.setSuperclass(target.getClass());
        // 设置代理人
```

```
        enhancer.setCallback(new CglibProxy(target));
        return enhancer.create();
    }
}
```

测试方法:

```
import java.lang.reflect.Method;
import java.util.Arrays;

public class CglibMainTest {
    public static void main(String[] args) {
        // 生成 Cglib 代理类
        Engineer engineerProxy = (Engineer) CglibProxy.getProxy(new Engineer());
        // 调用相关方法
        engineerProxy.eat();
    }
}
```

运行结果:

```
###   before invocation
工程师正在吃饭
###   end invocation
```

更多内容可以参考：CGLIB(Code Generation Library) 介绍与原理