结构型模式（三）

2020年4月27日　　11:24

结构型模式作业（三）

作业要求：

1、结合实例，绘制外观模式实例结构图（类图），用面向对象编程语言实现该模式，并对模式进行分析。

2、结合实例，绘制享元模式实例结构图（类图），用面向对象编程语言实现该模式，并对模式进行分析。

3、结合实例，绘制代理模式实例结构图（类图），用面向对象编程语言实现该模式，并对模式进行分析。

4、应用软件所提供的桌面快捷方式是快速启动应用程序的代理。桌面快速方式一般使用一张小图片（Picture）来表示，通过调用快捷方式的run()方法将调用应用软件（Appication）的run方法。使用代理模式模拟该过程，试绘制类图并编程实现。

要求：

1、模式结构图要求使用工具软件进行绘制，模式分析需要说明每个角色及作用。 如果可能，每个模式最好使用实际案例来说明。

2、以WORD文档/PDF格式提交。

3、提交的作业如果有多个文件，就提交多个，不要做压缩包。

- **作业与笔记github地址**：https://github.com/baobaotql/CCNU_DesignPattern

- **Q1：结合实例，绘制外观模式实例结构图（类图），用面向对象编程语言实现该模式，并对模式进行分析**

  A1：

  - **外观模式定义：**

    是一种通过为多个复杂的子系统提供一个一致的接口，而使这些子系统更加容易被访问的模式。该模式对外有一个统一接口，外部应用程序不用关心内部子系统的具体的细节，这样会大大降低应用程序的复杂度，提高了程序的可维护性。

  - 外观模式是“迪米特法则”的典型应用，他有以下**主要优点：**
    - 降低了子系统与客户端之间的耦合度，使得子系统的变化不会影响调用它的客户类。
    - 对客户屏蔽了子系统组件，减少了客户处理的对象数目，并使得子系统使用起来更加容易。
    - 降低了大型软件系统中的编译依赖性，简化了系统在不同平台之间的移植过程，因为编译一个子系统不会影响其他的子系统，也不会影响外观对象。
  - **缺点：**
    - 不能很好地限制客户使用子系统类。
    - 增加新的子系统可能需要修改外观类或客户端的源代码，违背了“开闭原则”
  - **模式结构：**
    - 外观（Facade）角色：为多个子系统对外提供一个共同的接口。
    - 子系统（Sub System）角色：实现系统的部分功能，客户可以通过外观角色访问它。
    - 客户（Client）角色：通过一个外观角色访问各个子系统的功能。
    - **模式结构图：**

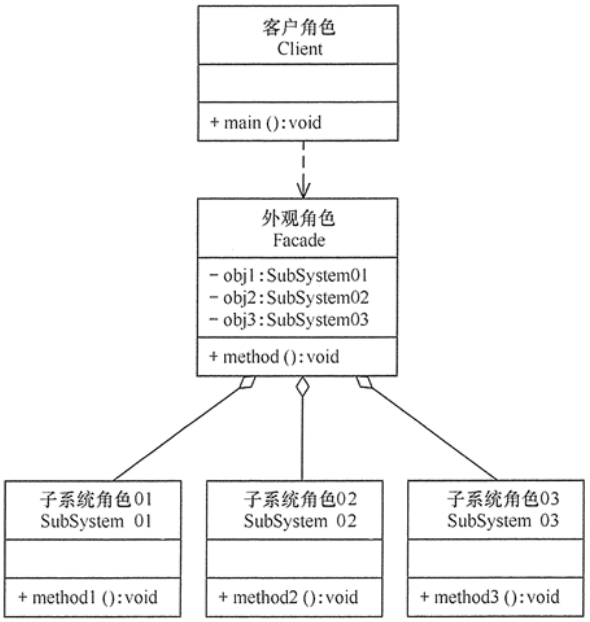


图2 外观（Facade）模式的结构图

外观（Facade）模式的结构比较简单，主要是定义了一个高层接口。它包含了对各个子系统的引用，客户端可以通过它访问各个子系统的功能。现在来分析其基本结构和实现方法。

- 外观模式的应用场景
  - 对分层结构系统构建时，使用外观模式定义子系统中每层的入口点可以简化子系统之间的依赖关系。
  - 当一个复杂系统的子系统很多时，外观模式可以为系统设计一个简单的接口供外界访问。
  - 当客户端与多个子系统之间存在很大的联系时，引入外观模式可将它们分离，从而提高子系统的独立性和可移植性。
- **模式代码：**

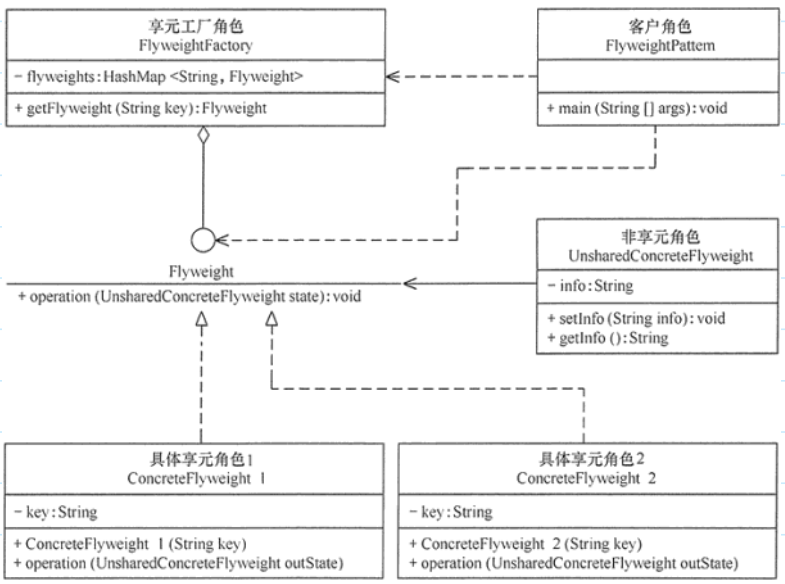
```
package facade;

public class FacadePattern
{
    public static void main(String[] args)
    {
        Facade f=new Facade();
        f.method();
    }
}
//外观角色
class Facade
{
    private SubSystem01 obj1=new SubSystem01();
    private SubSystem02 obj2=new SubSystem02();
    private SubSystem03 obj3=new SubSystem03();
    public void method()
    {
        obj1.method1();
        obj2.method2();
        obj3.method3();
    }
}
//子系统角色
class SubSystem01
{
    public  void method1()
    {
        System.out.println("子系统01的method1()被调用！");
    }
```

```
}
//子系统角色
class SubSystem02
{
    public void method2()
    {
        System.out.println("子系统02的method2()被调用！");
    }
}
//子系统角色
class SubSystem03
{
    public void method3()
    {
        System.out.println("子系统03的method3()被调用！");
    }
}
```

- **Q2：结合实例，绘制享元模式实例结构图（类图），用面向对象编程语言实现该模式，并对模式进行分析**

  A2：

  - **享元模式的定义：**
    运用共享技术来有效地支持大量细粒度对象的复用。它通过共享已经存在的又橡来大幅度减少需要创建的对象数量、避免大量相似类的开销，从而提高系统资源的利用率
  - **优点：**
    - 相同对象只要保存一份，这降低了系统中对象的数量，从而降低了系统中细粒度对象给内存带来的压力
  - **缺点：**
    - 为了使对象可以共享，需要将一些不能共享的状态外部化，这将增加程序的复杂性。
    - 读取享元模式的外部状态会使得运行时间稍微变长。
  - **模式结构**
    - 抽象享元角色（Flyweight）:是所有的具体享元类的基类，为具体享元规范需要实现的公共接口，非享元的外部状态以参数的形式通过方法传入。
    - 具体享元（Concrete Flyweight）角色：实现抽象享元角色中所规定的接口。
    - 非享元（Unsharable Flyweight)角色：是不可以共享的外部状态，它以参数的形式注入具体享元的相关方法中。
    - 享元工厂（Flyweight Factory）角色：负责创建和管理享元角色。当客户对象请求一个享元对象时，享元工厂检查系统中是否存在符合要求的享元对象，如果存在则提供给客户；如果不存在的话，则创建一个新的享元对象。
    - **模式结构图：**



- 图中的 UnsharedConcreteFlyweight 是与淳元角色，里面包含了非共享的外部状态信息 info；
- 而 Flyweight 是抽象享元角色，里面包含了享元方法 operation(UnsharedConcreteFlyweight state)，非享元的外部状态以参数的形式通过该方法传入；
- ConcreteFlyweight 是具体享元角色，包含了关键字 key，它实现了抽象享元接口；
- FlyweightFactory 是享元工厂角色，它逝关键字 key 来管理具体享元；
- 客户角色通过享元工厂获取具体享元，并访问具体享元的相关方法。

- **模式代码：**

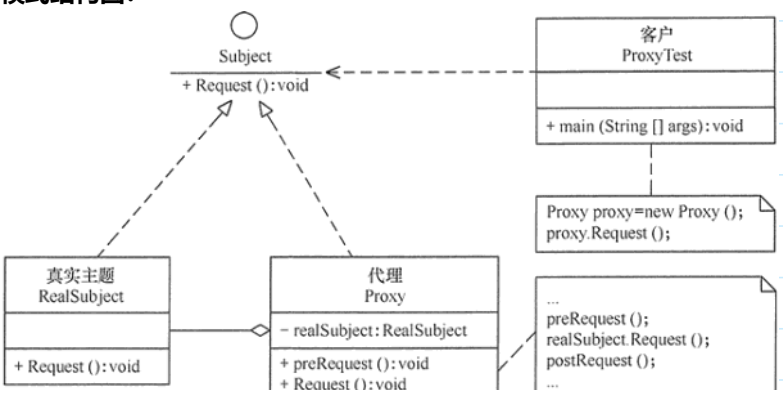
```
package flyweight;

import java.util.HashMap;

public class FlyweightPattern
{
    public static void main(String[] args)
    {
        FlyweightFactory factory=new FlyweightFactory();
        Flyweight f01=factory.getFlyweight("a");
        Flyweight f02=factory.getFlyweight("a");
        Flyweight f03=factory.getFlyweight("a");
        Flyweight f11=factory.getFlyweight("b");
        Flyweight f12=factory.getFlyweight("b");
        f01.operation(new UnsharedConcreteFlyweight("第1次调用a。"));
        f02.operation(new UnsharedConcreteFlyweight("第2次调用a。"));
        f03.operation(new UnsharedConcreteFlyweight("第3次调用a。"));
        f11.operation(new UnsharedConcreteFlyweight("第1次调用b。"));
        f12.operation(new UnsharedConcreteFlyweight("第2次调用b。"));
    }
}
//非享元角色
class UnsharedConcreteFlyweight
{
    private String info;
    UnsharedConcreteFlyweight(String info)
    {
        this.info=info;
    }
    public String getInfo()
    {
        return info;
    }
    public void setInfo(String info)
    {
        this.info=info;
    }
}
//抽象享元角色
interface Flyweight
{
    public void operation(UnsharedConcreteFlyweight state);
}
//具体享元角色
class ConcreteFlyweight implements Flyweight
{
    private String key;
    ConcreteFlyweight(String key)
    {
        this.key=key;
        System.out.println("具体享元"+key+"被创建！");
    }
```

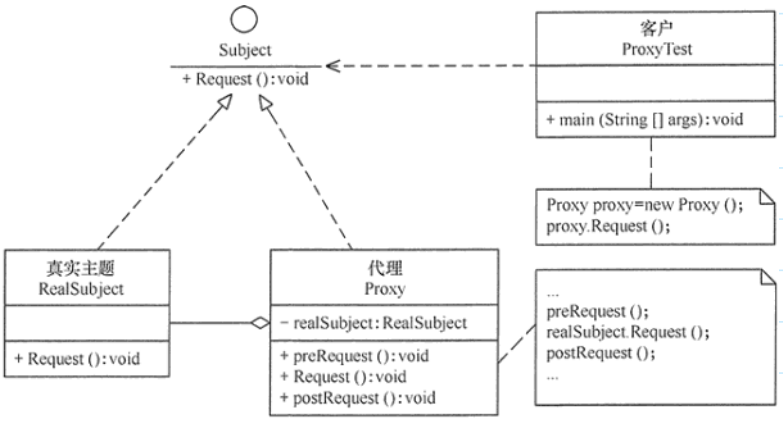
```
    public void operation(UnsharedConcreteFlyweight outState)
    {
        System.out.print("具体享元"+key+"被调用，");
        System.out.println("非享元信息是:"+outState.getInfo());
    }
}
//享元工厂角色
class FlyweightFactory
{
    private HashMap<String, Flyweight> flyweights=new HashMap<String, Flyweight>();
    public Flyweight getFlyweight(String key)
    {
        Flyweight flyweight=(Flyweight)flyweights.get(key);
        if(flyweight!=null)
        {
            System.out.println("具体享元"+key+"已经存在，被成功获取！");
        }
        else
        {
            flyweight=new ConcreteFlyweight(key);
            flyweights.put(key, flyweight);
        }
        return flyweight;
    }
}
```

- **Q3:结合实例，绘制代理模式实例结构图（类图），用面向对象编程语言实现该模式，并对模式进行分析**

  A3：
  - **代理模式定义**

    由于某些原因需要给某对象提供一个代理以控制对该对象的访问。这时，访问对象不适合或者不能直接引用目标对象，代理对象作为访问对象和目标对象之间的中介
  - **优点**
    - 代理模式在客户端与目标对象之间起到一个中介作用和保护目标对象的作用；
    - 代理对象可以扩展目标对象的功能；
    - 代理模式能将客户端与目标对象分离，在一定程度上降低了系统的耦合度；
  - **缺点**
    - 在客户端和目标对象之间增加一个代理对象，会造成请求处理速度变慢；
    - 增加了系统的复杂度
  - **模式结构**
    - 抽象主题（Subject）类：通过接口或抽象类声明真实主题和代理对象实现的业务方法。
    - 真实主题（Real Subject）类：实现了抽象主题中的具体业务，是代理对象所代表的真实对象，是最终要引用的对象。
    - 代理（Proxy）类：提供了与真实主题相同的接口，其内部含有对真实主题的引用，它可以访问、控制或扩展真实主题的功能。
    - **模式结构图：**

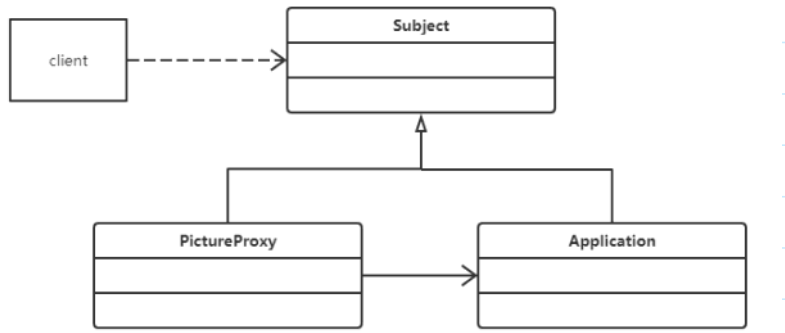





```
package proxy;
public class ProxyTest
{
    public static void main(String[] args)
    {
        Proxy proxy=new Proxy();
        proxy.Request();
    }
}
//抽象主题
interface Subject
{
    void Request();
}
//真实主题
class RealSubject implements Subject
{
    public void Request()
    {
        System.out.println("访问真实主题方法...");
    }
}
//代理
class Proxy implements Subject
{
    private RealSubject realSubject;
    public void Request()
    {
        if (realSubject==null)
        {
            realSubject=new RealSubject();
        }
        preRequest();
        realSubject.Request();
        postRequest();
    }
    public void preRequest()
    {
        System.out.println("访问真实主题之前的预处理。");
    }
    public void postRequest()
    {
        System.out.println("访问真实主题之后的后续处理。");
    }
}
```

- **Q4：应用软件所提供的桌面快捷方式是快速启动应用程序的代理。桌面快速方式一般使用一张小图片（Picture）来表示，通过调用快捷方式的run()方法将调用应用软件（Appication）的run方法。使用代理模式模拟该过程，试绘制类图并编程实现。**

  A4:




```
public class Application implements Subject {
	public void run() {
		System.out.println("Application is openning");
	}
}
public class PictureProxy implements Subject {
	Application application=new Application();
	public void run() {
		System.out.println("PictureProxy is called");
		application.run();
	}
}
public class Client {
	public static void main(String[] args) {
		Subject subject=new PictureProxy();
		subject.run();
	}
}
```