

设计模式 (下下)

桥接模式

在现实生活中，我们可以对不同类型的手机使用相同的功能，但是使用这个功能所进行的操作是有可能不同的。比如有折叠式手机，直立式手机，旋转式手机，显然这三者的开机，关机，接电话...等功能的使用方法都可能是不同的

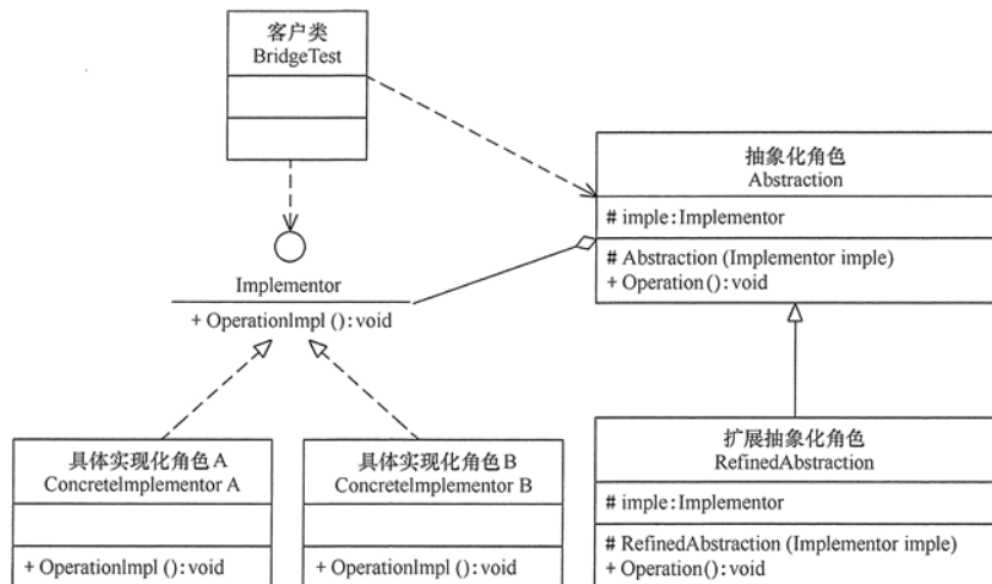
在传统的 OOP 方式下，设计可能会较为臃肿（手机下面有 折叠式手机，直立式手机，旋转式手机...甚至更多的子类，同时这些子类下面有可能因为品牌的不同而衍生出更多的子类），带来类爆炸的问题

基本介绍

桥接模式的基本定义是指：将实现与抽象放在两个不同的类层次中，使两个层次可以独立改变

桥接模式基于类的最少设计原则 (尽可能少的增加类)，通过使用封装，聚合及继承等行为让类承担不同的职责。它的主要特点是把抽象和行为实现分离，从而保持各部分的独立性以应对他们功能的拓展

桥接模式的典型 UML 类图如下



客户类：桥接模式调用者

抽象类 (Abstraction)：抽象类，并且引用 Implementor 对象

扩展抽象化类 (Refined Abstraction)：抽象类的子类，主要实现抽象类中的业务方法

行为实现接口 (Implementor)：定义实现化接口，供扩展抽象化类使用

具体行为实现类 (Concrete Implementor)：实现抽象类的具体行为

行为实现和抽象类分离开来，使得抽象类的功能独立于抽象类，而在行为实现接口中进行规范，子类进行实现。抽象类只需持有行为实现的引用，就可以进行任意的组装

小米(Brand) +翻盖机 (Phone)

小米+滑盖机

小米+直立式

案例

以上面手机为案例，使用桥接模式进行实现

我们将手机品牌作为行为实现接口，手机作为抽象类

我们假设不同手机品牌的颜色是不一样的，因此我们先写行为实现接口

```
public interface Brand {  
  
    void color();  
}
```

子类实现该接口后重写该方法即可

抽象类如下

```
abstract public class Phone {  
  
    //持有行为实现接口的引用，以使用其实现类  
    private Brand brand;  
  
    public Phone(Brand brand){  
        this.brand = brand;  
    }  
  
    public Phone(){}  
  
    public void color(){  
        brand.color();  
    }  
}
```

翻盖手机类如下

```
public class FanGaiPhone extends Phone {  
  
    public FanGaiPhone(Brand brand) {  
        super(brand);  
    }  
  
    public FanGaiPhone() {  
        super();  
    }  
  
    @Override  
    public void color() {
```

```

        super.color();
        System.out.println("并且这部还是翻盖手机");
    }
}

```

子类并不是完全重写了父类抽象类的方法，而是借用了父类的方法，而父类的方法又去调用了行为实现接口的方法 (其实本质是在调用其实现类的方法)，在这时，父类抽象类就起到了桥的作用

客户端调用

```

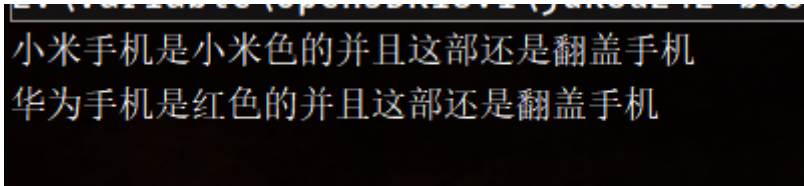
public class Client {
    public static void main(String[] args) {

        //翻盖的小米手机 (样式+品牌)
        FanGaiPhone fanGaiPhone = new FanGaiPhone(new MiBrand());
        fanGaiPhone.color();

        //翻盖的华为手机
        FanGaiPhone fanGaiPhone1 = new FanGaiPhone(new HWBrand());
        fanGaiPhone1.color();

    }
}

```



小米手机是小米色的并且这部还是翻盖手机
华为手机是红色的并且这部还是翻盖手机

在这种方式下，我如果我们想要增加一种新的手机类型，如双面屏，我们只需要写一个类去实现抽象类即可，而无需关心这个手机类型下有哪些品牌；同样，如果我们想要增加新的手机品牌，也无需关心手机类型。不用担心传统模式下类爆炸的问题

注意事项和细节

完成了抽象和行为实现部分的分离，从而提升了系统的灵活性，让抽象部分和行为实现分离开，有助于系统的分层设计，从而产生更好的结构化系统

对于系统的高层部分，只需要知道抽象部分和实现部分的接口即可，而无需知道具体细节

桥接模式替代了多层继承方案，可以减少子类个数

缺点

- 桥接模式的引入增加了系统的理解难度和设计难度，它要求开发者能够正确识别出系统中能够独立变化的两个维度。将实现接口作为聚合关系建立在抽象层，并抽出正确的行为作为实现接口

装饰者模式

案例需求

星巴克订单需求

- 咖啡种类: Espresso, ShortBlack, LongBlack, Decaf
- 调料: Milk, Soy, Chocolate

需要在扩展新的咖啡种类时, 具有良好的扩展性, 改动方便, 利于维护

客户可以只点咖啡, 也可以点咖啡+调料的组合, 并计算出价格

原理及实现

装饰者模式的定义: 动态的将新功能附加到对象上, 在对象功能扩展方面, 比继承更加有弹性, 也符合 OCP 原则

成员

Component: 抽象主体, 被装饰者和具体主体实现, 并且由装饰者持有其引用, 以装饰具体主体

ConcreteComponent: 具体主体, 被装饰者, 比如前面的咖啡种类

Decorator: 抽象装饰, 用于装饰主体, 比如各种配料

ConcreteDecorator: 具体装饰, 继承抽象装饰

当具体主体比较多时, 可以在抽象主体和其之间做一层缓冲, 作为具体主体的抽象, 减少抽象主体的成员

实现

以一个具体的订单为例: 现在有一位顾客, 点了一份 LongBlack + 一份牛奶 + 两份巧克力

我们可以看成 LongBlack 被 牛奶装饰, LongBlack + 牛奶又被一份巧克力装饰, 最后, LongBlack + 牛奶 + 一份巧克力又被一份巧克力装饰

因为我们的目的是喝咖啡, 以及计算出咖啡的价格, 因此我们可以将抽象主体定在 喝 这个动作上, 并且描述出我们喝的咖啡及其配料

```
abstract public class Drink {  
  
    //描述  
    public String dec;  
    //价格  
    private double price = 0.0;  
  
    //计算费用, 由于配料和咖啡都要计算价格, 所以抽离出来  
    abstract public double cost();  
  
    public String getDec() {  
        return dec;  
    }  
}
```

```

    public void setDec(String dec) {
        this.dec = dec;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }
}

```

然后创建抽象主体和具体主体之间的缓冲抽象类，Coffee

```

public class Coffee extends Drink {

    @Override
    public double cost() {
        return super.getPrice();
    }
}

```

创建其具体主体，这里只以 EspressoCoffee 为例

```

public class EspressoCoffee extends coffee {

    public EspressoCoffee() {
        setDec("意大利咖啡");
        setPrice(16.0);
    }
}

```

创建抽象装饰类

```

public class PeiLiaoDecorator extends Drink {

    private Drink component;

    //计算出配料的价格和咖啡的价格
    @Override
    public double cost() {
        return this.getPrice() + component.cost();
    }

    public PeiLiaoDecorator(Drink component) {
        this.component = component;
    }

    @Override
    public String getDec() {
        return dec + "" + this.getPrice() +
            "&&" + component.getDec();
    }
}

```

创建具体装饰类，这里只以 Milk 为例

```
public class MilkDecorator extends PeiLiaoDecorator {  
  
    public MilkDecorator(Drink component) {  
        super(component);  
        setDec("牛奶");  
        setPrice(3.0);  
    }  
}
```

创建客户端进行测试

```
public class Client {  
    public static void main(String[] args) {  
  
        //点了一份 LongBlack + 一份牛奶 + 两份巧克力  
  
        Drink order = new LongBlackCoffee();  
  
        order = new MilkDecorator(order);  
        order = new ChocolateDecorator(order);  
        order = new ChocolateDecorator(order);  
  
        System.out.println(order.cost());  
        System.out.println(order.getDec());  
    }  
}
```

25.0

巧克力5.000巧克力5.000牛奶3.000美式咖啡

在这种模式下，我们想要增加一个新的咖啡种类，只需要继承 Coffee 即可，然后根据需求用各种配料进行装饰即可

组合模式

比如学校：一个学校有多个学院，学院下有多个系，因此形成了一个树形结构，可以更好的实现管理操作。这就是组合模式的典型案例

介绍

- 组合模式又叫部分整体模式，该模式创建了对象组的树形结构，将对象组合成树状结构以表示“整体-部分”的层次关系
- 则和模式依据树形结构来组合对象，用来表示部分以及整体层次

- 组合模式使得用户对单个对象 (叶子结点) 和组合对象 (非叶子结点) 的访问具有一致性。组合模式能让客户端以一致的方式

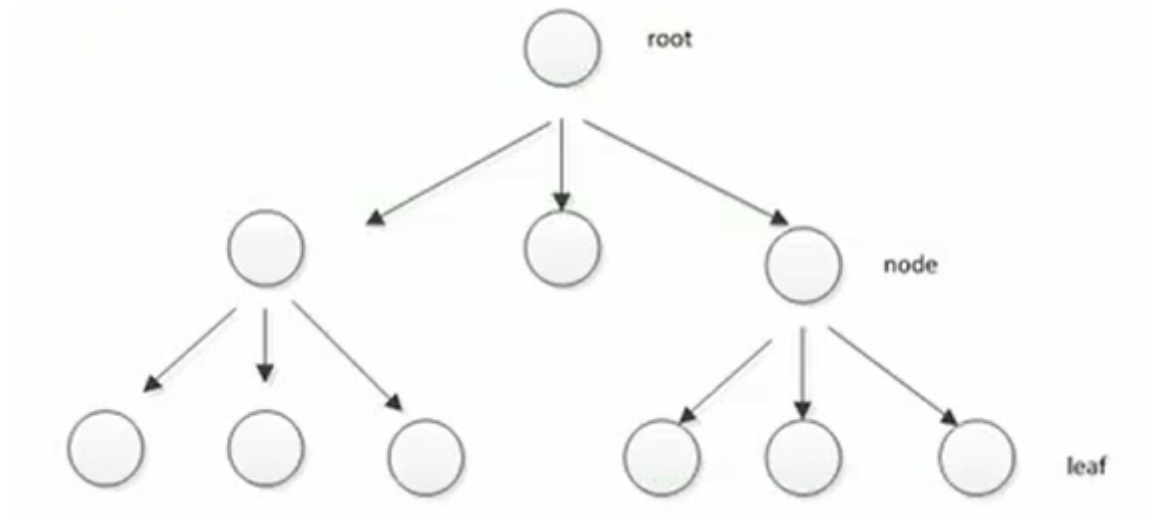
处理个别对象以及组合对象

模式结构

Component:: 它的主要作用是为叶子结点和非叶子结点声明公共接口，并实现它们的默认行为

Composite: 非叶子结点，重写 Component 中关于子结点的操作，如 add, remove, get, 等

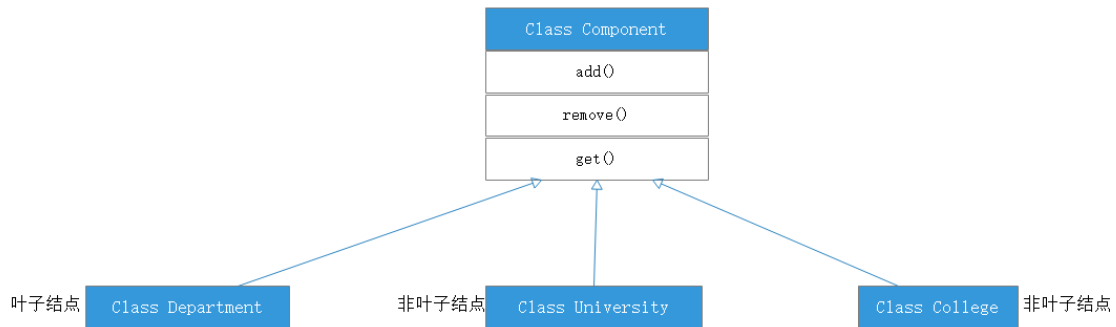
Leaf: 叶子结点



组合模式能够解决：当我们对树上的结点和叶子进行操作时，能够使用统一的操作方式，而不用考虑其实结点还是叶子

案例分析及实现

学校，学院，系，其实都可以算做一种组织，所以我们可以将“组织”这个概念作为 Component 抽象出来



学校管理学院，学院管理系。
因此学校会组合学院，学院会组合系
但是在聚合时其实聚合的是
Component，作为多态来调用其子结点

代码实现

先写 Component，有点像所谓的根结点，不过这个根结点是所有的非叶子结点和叶子结点的抽象

```
abstract public class OrganizationComponent {

    //名字
    private String name;
    //描述
    private String des;

    /**
     * 默认实现，对于叶子结点而言，没有可以操作的子结点，
     * 所以它是不用重写 add 方法的
     * */
    protected void add(OrganizationComponent component){

    }

    protected void remove(OrganizationComponent component){

    }

    public OrganizationComponent(String name, String des) {
        this.name = name;
        this.des = des;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDes() {
```



```

        return des;
    }

    public void setDes(String des) {
        this.des = des;
    }

    //所有子类都需要实现
    abstract public void printf();
}

```

Composite 非叶子结点 University

```

public class University extends OrganizationComponent {

    //聚合子结点，这里的List存的就是学院
    List<OrganizationComponent> sonNode = new ArrayList<>();

    public University(String name, String des) {
        super(name, des);
    }

    //重写 add
    @Override
    protected void add(OrganizationComponent component) {
        sonNode.add(component);
    }

    //重写 remove
    @Override
    protected void remove(OrganizationComponent component) {
        sonNode.remove(component);
    }

    //输出 university 中包含的学院
    @Override
    public void printf() {
        System.out.println("-----"+getName()+"-----");
        for (OrganizationComponent component : sonNode) {
            component.printf();
        }
    }
}

```

非叶子结点 College

```

public class College extends OrganizationComponent {

    //这个 List 存的其实是 department
    List<OrganizationComponent> sonNode = new ArrayList<>();

    public College(String name, String des) {
        super(name, des);
    }
}

```

```

//重写 add
@Override
protected void add(OrganizationComponent component) {
    sonNode.add(component);
}

//重写 remove
@Override
protected void remove(OrganizationComponent component) {
    sonNode.remove(component);
}

//输出 university 中包含的学院
@Override
public void printf() {
    System.out.println("-----"+getName()+"-----");
    for (OrganizationComponent component : sonNode) {
        component.printf();
    }
}
}

```

叶子结点 Department

```

public class Department extends OrganizationComponent {

    public Department(String name, String des) {
        super(name, des);
    }

    @Override
    public void printf() {
        System.out.println(getName()+":"+getDes());
    }
}

```

客户端

```

public class Client {
    public static void main(String[] args) {

        //创建学校
        OrganizationComponent university = new University("贵理工", "nb");
        //创建学院
        OrganizationComponent computerCollege = new College("计算机学院", "有这个学院? ");
        OrganizationComponent wakuangCollege = new College("挖矿学院", "黄金矿工");

        //创建系
        computerCollege.add(new Department("带数据", "找不到工作"));
        computerCollege.add(new Department("云计算", "学 jsp 的云计算"));
    }
}

```

```

    waKuangCollege.add(new Department("挖钻石", "挖锤子"));
    waKuangCollege.add(new Department("挖黄金", "挖peach"));

    //将学院加入学校
    university.add(computerCollege);
    university.add(waKuangCollege);

    university.printf();
}
}

```

```

-----贵理工-----
-----计算机学院-----
带数据::找不到工作
云计算::学 jsp 的云计算
-----挖矿学院-----
挖钻石::挖锤子
挖黄金::挖peach

```

如果我们还想加班级这种类，可以直接组合到树状结构中作为叶子结点，那我们只需将原来的叶子结点进行方法的增加，新的叶子结点写法同原来的叶子结点即可；如果我们要组合一个类作为非叶子结点，则直接聚合进去即可

注意事项和细节

组合模式简化了客户端操作，只需面对一致的对象而不需考虑叶子结点和非叶子结点的问题

具有较强的扩展性，当我们需要更改组合对象时，只需要调整内部的层次关系，客户端不用做出任何改动

缺点

如果非叶子结点和叶子结点具有很大的差异性，比如很多方法和属性都不一样，则不适合使用组合模式

外观模式

需求

假设在工厂中，我们有很多的设备，每次来上班需要将它们按顺序启动，下班又需要将它们按顺序关闭，如果每个设备都需要亲自去操作一番，这显然是很麻烦的，因此，我们可以提供一个“面板类”，来对这些设备进行统一的管理，这样我们(客户端)只需要跟这个面板发生调用关系即可。这就是外观模式

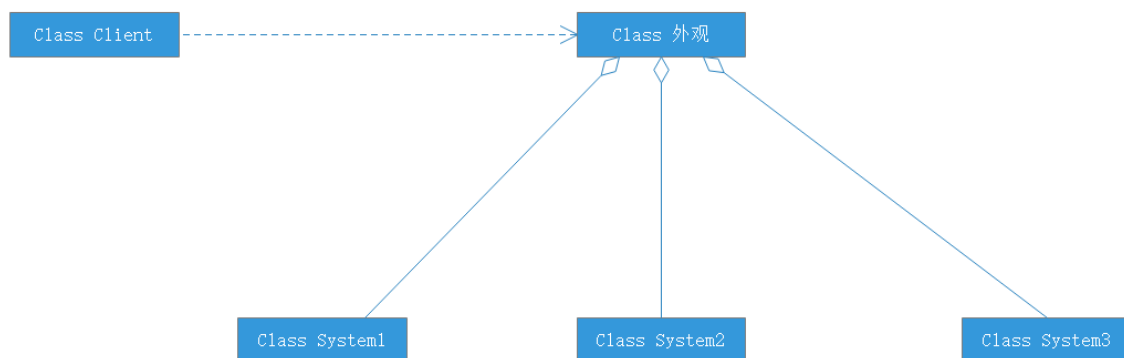
基本介绍

外观模式为子系统中的一组接口提供一个一致的界面，此模式通过定义一个高层接口，该接口使得子系统更容易被使用，并且不需要关心子系统的内部细节

模式结构

外观类有、类似于 Gateway 的功能

1. 只将具体的主机 (子系统) 组合进来
2. 然后转发客户端请求到对应的主机 (子系统) 上：在外观类中通过方法 "转发"，然后根据需求在转发方法中组合子系统的方法完成功能的实现
3. 具体处理由主机完成



外观类：为客户端提供统一的调用接口，但是并不能很好地限制客户端使用哪些子系统

System：子系统，处理外观类指派的任务 (客户端发来的请求)。相当于实现功能

Client：客户端，调用外观类

代码实现

建立第一个子系统类，TVSystem

```
public class TVSystem2 {

    //单例
    private static TVSystem2 tvSystem = new TVSystem2();

    private TVSystem2(){
        if(tvSystem!=null){
            throw new RuntimeException("不允许以该方法创建单例对象");
        }
    }

    public static TVSystem2 getInstance(){
        return tvSystem;
    }

    public void start(){
        System.out.println("电视打开");
    }
}
```

```

    }

    public void show(){
        System.out.println("电视放日剧");
    }

    public void halt(){
        System.out.println("电脑关机");
    }
}

```

然后其他子系统类

```

public class MachineSystem {

    private static MachineSystem machineSystem = new MachineSystem();

    private MachineSystem(){
        if(machineSystem!=null){
            throw new RuntimeException("不允许该方式创建单例对象");
        }
    }

    public MachineSystem getInstance(){
        return machineSystem;
    }

    public void start(){
        System.out.println("机器人打开");
    }

    public void move(){
        System.out.println("机器人炒菜");
    }

    public void wash(){
        System.out.println("机器人睡午觉");
    }

    public void halt(){
        System.out.println("机器人关机");
    }
}

```

```

public class ComputerSystem {

    //单例
    private static ComputerSystem computerSystem = new ComputerSystem();

    private ComputerSystem(){
        if(computerSystem!=null){
            throw new RuntimeException("单例对象不允许如此创建");
        }
    }

    public ComputerSystem getInstance() {

```

```

        return computerSystem;
    }

    public void start(){
        System.out.println("电脑打开");
    }

    public void music(){
        System.out.println("电脑播放音乐");
    }

    public void halt(){
        System.out.println("电脑关机");
    }
}

```

建立外观类

我们假设工厂在一天中需要进行如下阶段的操作

ready: 电脑和电视开机

play: 电脑放音乐，电视放日剧

pause: 机器人开机并进行炒菜的工作，然后睡觉

end: 关闭所有子系统

```

public class SystemFacade {

    /**
     * 引用各个子系统对象
     */
    private ComputerSystem computerSystem;
    private TVSystem2 tvSystem;
    private MachineSystem machineSystem;

    public SystemFacade() {
        this.computerSystem = ComputerSystem.getInstance();
        this.tvSystem = TVSystem2.getInstance();
        this.machineSystem = MachineSystem.getInstance();
    }

    public void ready(){
        computerSystem.start();
        tvSystem.start();
    }

    public void play(){
        computerSystem.music();
        tvSystem.show();
    }

    public void pause(){
        machineSystem.start();
        machineSystem.move();
        machineSystem.wash();
        machineSystem.halt();
    }
}

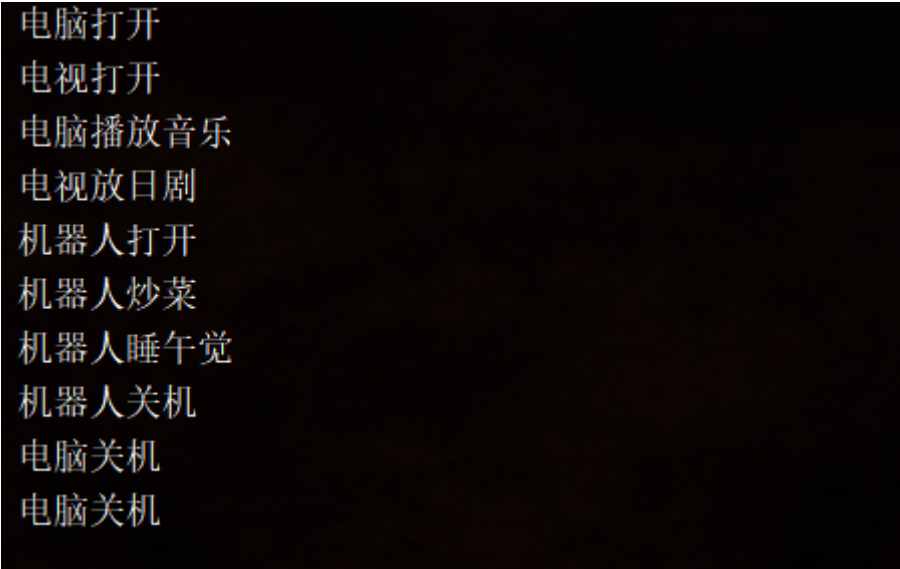
```

```
public void end(){
    computerSystem.halt();
    tvSystem.halt();
}
}
```

客户端

```
public class Client {
    public static void main(String[] args) {

        SystemFacade systemFacade = new SystemFacade();
        systemFacade.ready();
        systemFacade.play();
        systemFacade.pause();
        systemFacade.end();
    }
}
```



电脑打开
电视打开
电脑播放音乐
电视放日剧
机器人打开
机器人炒菜
机器人睡午觉
机器人关机
电脑关机
电脑关机

注意事项和细节

外观模式屏蔽了子系统的细节，对客户端而言是透明的，降低了客户端与子系统的耦合度

增加新的子系统可能需要修改外观类或客户端的代码，违反了 ocp 原则

享元模式

需求

小明给客户 A 做了一个产品展示网站，A 的朋友们觉得网站效果不错，也希望做一个产品展示网站，但是要求都有些不同

- 有客户要求以新闻的形式展示
- 有客户要求以博客的形式展示
- 有客户要求以微笑公众号的形式展示

传统解决方案：cv 一份原来的，然后根据需要改一改，再给每个网站租一台服务器即可

传统解决方案的不足

- 需要的网站结构相似度个很高，而且都不是高访问量的网站，如果用多台服务器就亏了

解决方案：整合到一个环境中，共享其相关的代码和数据，通过不同域名访问不同的网站即可。对于硬盘，内存，CPU，数据库等资源来说，都可以达成共享

对于这种实现了对象共享的模式，就是享元模式

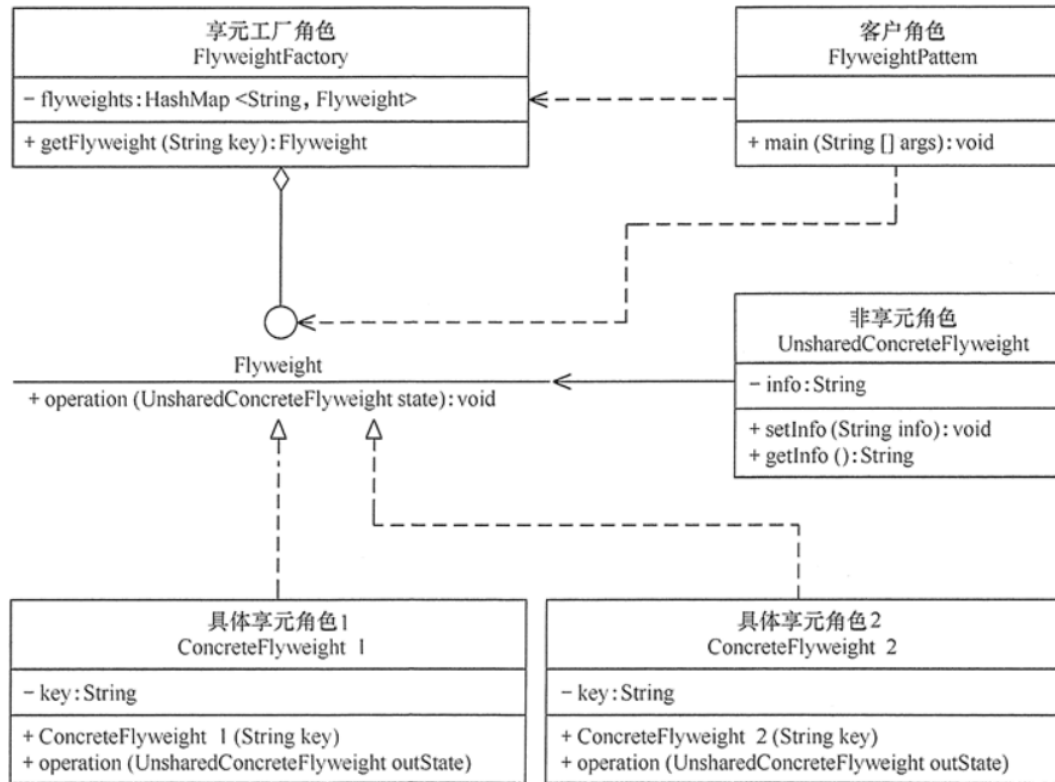
基本介绍

享元模式也叫蝇量模式，运用共享技术有效的支持大量细粒度的对象

享元模式的主要目的是实现对象的共享，即共享池，当系统中对象多的时候可以减少内存的开销，通常与工厂模式一起使用。当一个客户端请求时，工厂需要检查当前对象池中是否有符合条件的对象，如果有，就返回已经存在的对象，如果没有，则创建一个新对象

享元模式常用于系统底层开发，解决系统的性能问题，例如数据库连接池：在池中都是创建好的连接对象，我们在使用时只需要去连接池中取用即可，如果想要取得的连接对象不存在，则在池中创建一个新的连接对象即可

模式结构



抽象享元角色 (Flyweight)：是所有的具体享元类的基类，为具体享元角色规范需要实现的公共接口，同时定义对象的外部状态和内部状态

具体享元角色 (Concrete Flyweight)：实现抽象享元角色中所规定的接口。

非享元角色 (Unsharable Flyweight)：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，而不会出现在享元工厂中

享元工厂角色 (Flyweight Factory)：负责创建和管理容器 (集合) 中的享元角色。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

外部状态和内部状态

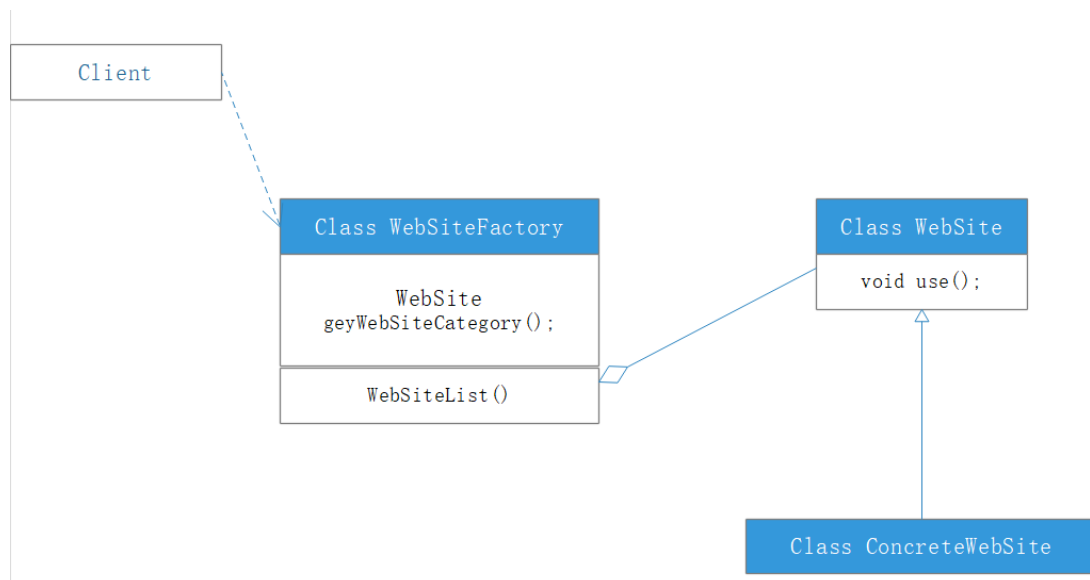
享元模式提出了两个需求：细粒度和共享对象。共享对象的信息分为两个部分，即内部状态和外部状态

内部状态：指对象共享出来的信息，存储在享元对象内部且不会随着环境而改变

外部状态：对象依赖的标记，随着环境的改变而改变，不可共享

代码实现

我们先来实现一下我们所说的网站案例的类图



先写抽象享元角色

```

abstract public class WebSite {

    abstract public void use();
}
  
```

然后具体享元角色

```

public class ConcreteWebSite extends WebSite {

    //具体的网站类型
    private String type;

    public ConcreteWebSite(String type){
        this.type = type;
    }

    @Override
    public void use() {
        System.out.println("使用"+type+"展示网页");
    }
}
  
```

享元工厂

```

/**
 * 网站工厂类，根据需求返回具体网站 concreteWebSite 的对象
 */
public class WebSiteFactory {

    //集合池
    private Map<String, ConcreteWebSite> webSitePool = new HashMap<>();

    //根据网站类型，从池中返回一个网站；如果没有则创建该网站并放入池中，并返回
    public WebSite getConcreteCategory(String type){
        //判断是否有对应对象在池中
        if(!webSitePool.containsKey(type)){
            webSitePool.put(type, new ConcreteWebSite(type));
        }
    }
}
  
```

```

        return webSitePool.get(type);
    }
    return webSitePool.get(type);
}

//获取网站有哪些类型
public void getPoolSize(){
    System.out.println(webSitePool.size());
}
}

```

客户端

```

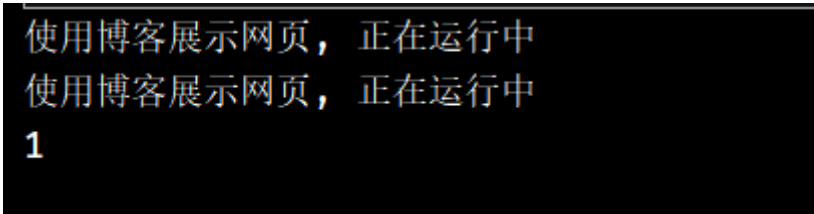
public class Client {
    public static void main(String[] args) {

        //创建工厂类
        webSiteFactory webSiteFactory = new webSiteFactory();

        //客户要一个博客形式的网站
        webSite blog = webSiteFactory.getConcreteCategory("博客");
        blog.use();
        //第二个客户也要一个博客形式的网站
        webSite blog2 = webSiteFactory.getConcreteCategory("博客");
        blog2.use();

        webSiteFactory.getPoolSize();
    }
}

```



```

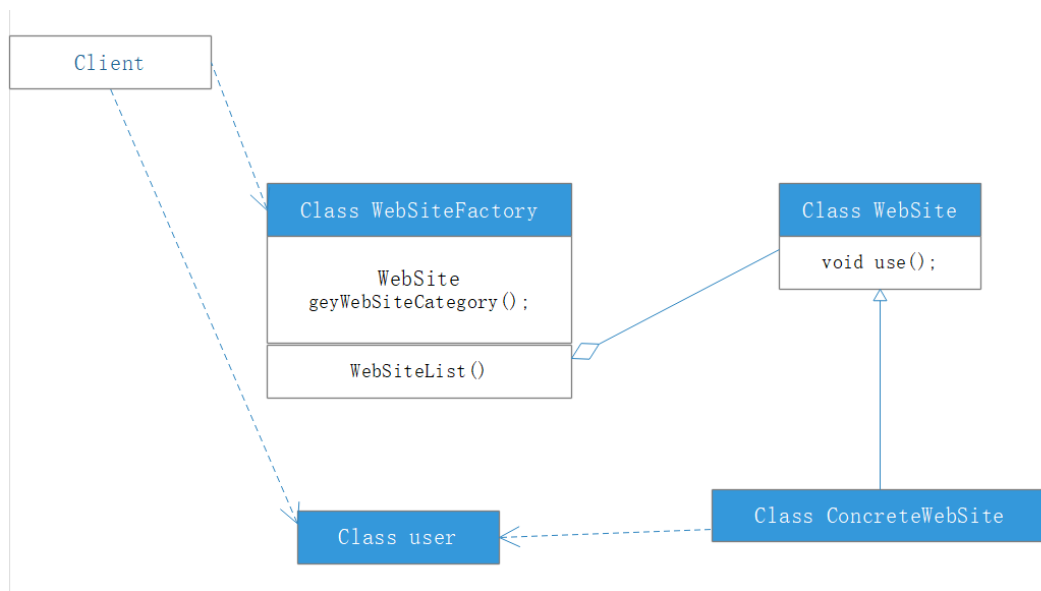
使用博客展示网页，正在运行中
使用博客展示网页，正在运行中
1

```

可以发现，虽然我们创建了两个 webSite 对象，但是在享元工厂中其实只有1个博客类型的网站，除了第一次创建外，后面再创建 key 为博客类型的网站时直接返回即可

type 可以说是网站的内部状态，但是网站也应该有外部状态：比如说用户就是一种外部状态

因此可以在类图中增加一个角色



新增用户类

```

public class User {

    private String name;

    public User(String name) {
        this.name = name;
    }

    public User(){}

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

}

```

然后将用户这个外部状态以参数注入的方式注入到享元角色中

```

abstract public class website {

    //什么用户去使用网站
    abstract public void use(User user);

}

```

```

public class Concretewebsite extends website {

    //具体的网站类型
    private String type;

    public Concretewebsite(String type){
        this.type = type;
    }

    @Override

```

```

    public void use(User user) {
        System.out.println("用户: "+user.getName()+"使用"+type+"网页", 正在运行
中");
    }
}

```

客户端

```

public class Client {
    public static void main(String[] args) {

        //创建工厂类
        webSiteFactory webSiteFactory = new WebSiteFactory();

        //客户要一个博客形式的网站
        webSite blog = webSiteFactory.getConcreteCategory("博客");
        blog.use(new User("张三"));
        //第二个客户也要一个博客形式的网站
        webSite blog2 = webSiteFactory.getConcreteCategory("博客");
        blog2.use(new User("李四"));

        webSiteFactory.getPoolSize();
    }
}

```

```

用户: 张三使用博客网页, 正在运行中
用户: 李四使用博客网页, 正在运行中
1

```

在享元模式中，我们不仅减少了对对象的创建（相同对象在对象池中只存在一份），避免了资源浪费，同时共享了对象的内部状态

Integer 中有趣的享元模式

我们先来看如下这段代码

```

Integer x = Integer.valueOf(127);
Integer y = Integer.valueOf(127);

Integer z = Integer.valueOf(256);
Integer o = Integer.valueOf(256);

System.out.println(x==y);
System.out.println(z==o);

```

结果如下

```
true  
false
```

我们进入 `valueOf` 这个方法中，看一看为什么会出现这种情况

```
public static Integer valueOf(int i) {  
    if (i >= IntegerCache.low && i <= IntegerCache.high)  
        return IntegerCache.cache[i + (-IntegerCache.low)];  
    return new Integer(i);  
}
```

可以看见，如果参数 `i` 在 `low` 和 `high` 之间，则直接返回缓存池中的对象，那 `low` 和 `high` 又是多少呢？跳转到 `low` 定义处

```
static final int low = -128;  
static final int high;  
static final Integer cache[];  
  
static {  
    // high value may be configured by property  
    int h = 127;  
    String integerCacheHighPropValue =  
        sun.misc.VM.getSavedProperty(key: "java.lang.Integer.IntegerCache.high");  
    if (integerCacheHighPropValue != null) {  
        try {  
            int i = parseInt(integerCacheHighPropValue);  
            i = Math.max(i, 127);  
            // Maximum array size is Integer.MAX_VALUE  
            h = Math.min(i, Integer.MAX_VALUE - (-low) - 1);  
        } catch (NumberFormatException nfe) {  
            // If the property cannot be parsed into an int, ignore  
        }  
    }  
    high = h;  
  
    cache = new Integer[(high - low) + 1];  
    int j = low;  
    for(int k = 0; k < cache.length; k++)  
        cache[k] = new Integer(j++);  
  
    // range [-128, 127] must be interned (JLS7 5.1.7)  
    assert IntegerCache.high >= 127;  
}  
  
private IntegerCache() {}  
}
```

可以看见，low 到 high 的范围是 -128~127，在此范围内使用了享元模式，直接返回缓存池中的对象；如果不在此范围，则会 new 一个 Integer 对象并返回。所以在上面的例子中，前者是 true，后者是 false

注意事项和细节

在享元模式中，享表示共享，元表示对象

当系统中有大量对象，这些对象消耗大量内存，并且对象的状态大部分可以外部化时，可以考虑使用享元模式

用唯一标识码判断享元对象是否存在，如果在内存中有，则返回这个唯一标识所表示的对象，一般使用 Map 集合

享元模式减少了对对象的创建，降低了程序内存的占用

享元模式的经典场景是需要缓存池，比如 String 常量池，数据库连接池

缺点

享元模式提高了系统的复杂度，需要分离出内部状态和外部状态，而内部状态具有固化特性，不应随着外部状态的改变而改变

享元模式需要注意区分内部状态和外部状态，并且常和工厂模式搭配使用，由享元对象工厂来管理和创建享元对象

代理模式

基本介绍

为一个对象提供一个替身，以控制对这个对象的访问，即通过代理对象访问目标对象。这样做的好处是：可以在目标对象实现的基础上，增加额外的功能操作，即扩展目标对象的功能

被代理的对象可以是远程对象，**创建开销大的**对象或需要**安全控制**的对象

代理模式有不同的形式，主要的有三种

- 静态代理
- 动态 (JDK 代理) 代理
- CGLib 代理 (可以在内存中动态的创建对象，而不需要实现接口)

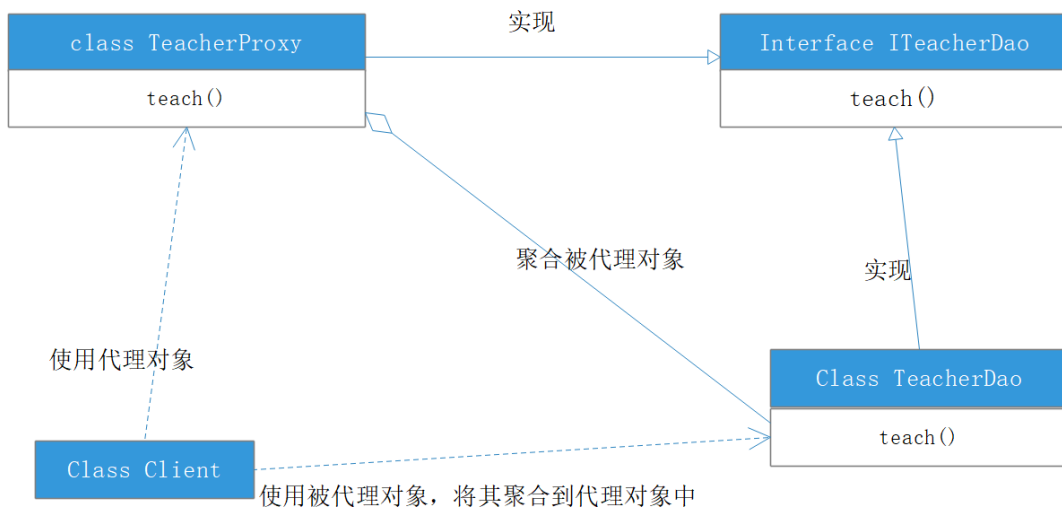
静态代理

基本介绍

静态代理在使用时，需要定义接口或者父类，被代理对象(目标对象)与代理对象一起实现相同的接口或者是继承相同的父类，然后通过调用代理对象中的方法来调用目标对象中相同的方法

案例

假设我们的老师生病了，没办法亲自来到教室上课，于是委托我来讲课，老师远程告诉我应该讲什么内容，此时我就是代理对象，老师就是被代理对象，我们二者需要实现同一个接口



接口，规范的是代理对象应该代理的方法

```
public interface ITeacherDao {

    void teach();

}
```

被代理对象

```
public class TeacherDao implements ITeacherDao {
    @Override
    public void teach() {
        System.out.println("老师授课中");
    }
}
```

代理对象

```
public class TeacherProxy implements ITeacherDao {

    //聚合被代理对象
    private ITeacherDao teacher;

    public TeacherProxy(ITeacherDao teacher){
        this.teacher = teacher;
    }

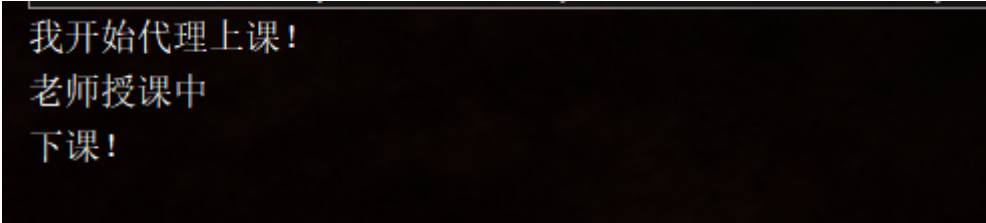
    @Override
    public void teach() {
        System.out.println("我开始代理上课!");
        teacher.teach();
    }
}
```



```
        System.out.println("下课! ");  
    }  
}
```

客户端

```
public class Client {  
  
    public static void main(String[] args) {  
  
        //创建被代理对象  
        TeacherDao teacherDao = new TeacherDao();  
        //创建代理对象，同时将被代理对象传入  
        TeacherProxy teacherProxy = new TeacherProxy(teacherDao);  
        teacherProxy.teach();  
    }  
}
```



```
我开始代理上课!  
老师授课中  
下课!
```

优缺点

优点：在不修改被代理对象功能的前提下，能直接在代理对象中进行功能的扩展

缺点：因为代理对象需要与被代理对象实现一样的接口，所以在有很多代理对象的情况下，一大能加接口方法，被代理对象和众多的代理对象都需要进行修改

JDK 代理

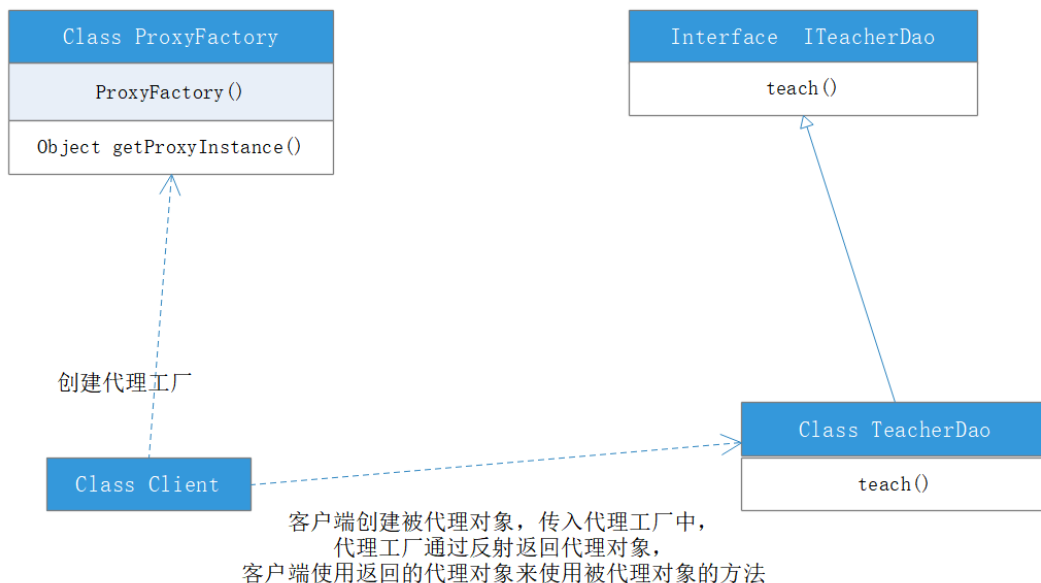
基本介绍

代理对象不需要实现接口，但是被代理对象需要实现接口

代理对象的生成，需要利用 JDK 的 API，动态的在内存中构建代理对象所以叫 JDK 代理

案例

我们依据静态代理的案例来进行修改，类图如下



按照相同的步骤，我们先创建接口，但在JDK代理中这个接口只会被被代理对象实现

```
public interface ITeacherDao {

    void teach();

}
```

被代理对象

```
public class TeacherDao implements ITeacherDao {

    @Override
    public void teach() {
        System.out.println("老师正在授课");
    }

}
```

代理对象工厂

```
public class ProxyFactory {

    //聚合被代理对象
    private Object target;

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

    //根据被代理对象生成代理对象
    public Object getProxyInstance(){

        return Proxy.newProxyInstance(target.getClass().getClassLoader(),
            target.getClass().getInterfaces(),
            new InvocationHandler() {
                @Override
                public Object invoke(Object proxy, Method method, Object[]
args) throws Throwable {

            
```

```

        System.out.println("动态代理开始");
        //调用 method 对象代表的方法
        Object re = method.invoke(target, args);
        System.out.println("动态代理结束");
        return re;
    }
}
});
}
}

```

```

public static Object newProxyInstance(ClassLoader loader,
                                     Class[] interfaces,
                                     InvocationHandler h)

```

newProxyInstance 中的参数说明

- loader: 指定 **当前被代理对象的类加载器**
- interfaces: **被代理对象实现的接口们**
- InvocationHandler h: 事件处理器，监听接口的方法的调用，可以在方法调用的时候，追加一些其他操作

```

public Object invoke(Object proxy, Method method, Object[] args)
    throws Throwable;

```

- proxy: 调用方法的代理对象
- method: 当使用生成的代理对象调用被代理对象的方法时，这个参数就是那个方法的反射
- args: 如果调用的被代理对象的方法有参数，则参数会放入这里
- 返回值: 根据上面三个参数生成的代理对象

客户端

```

public class Client {
    public static void main(String[] args) {

        TeacherDao target = new TeacherDao();
        //创建代理对象
        ProxyFactory factory = new ProxyFactory(target);
        ITeacherDao instance = (ITeacherDao)factory.getProxyInstance();

        instance.teach();
    }
}

```

动态代理开始
老师正在授课

注意事项和细节

在 JDK 代理模式中，我们不需要使得代理类去继承接口，降低了与被代理对象的耦合度
当接口增加方法时，也不用再修改代理类，简化了操作

CGlib 代理

静态代理和动态代理都要求被代理对象实现一个接口，但是这无疑也增加了维护的复杂度。当我们的被代理对象并没有实现任何接口时，我们可以使用被代理对象的子类来实现代理，这就是 CGlib 代理

CGlib 代理也叫作子类代理，它是在内存中构建一个子类对象从而实现对被代理对象功能的扩展。

CGlib 是一个强大的，高性能的代码生成包，可以在运行期扩展 Java 类和实现接口，底层是通过使用字节码处理框架 ASM 来转换字节码并生成新的类

代理模式的选择

- 被代理对象，需要抽象出一层接口，并实现其中的方法时，可以使用 JDK 代理来创建其代理对象（因为 newProxyInstance 要求必须传入被代理对象实现的接口）
- 被代理对象不需要抽象出，并实现接口，使用 CGlib 代理
- CGlib 代理也可以创建实现了接口的被代理对象的代理对象，但是在 jdk1.8 中，效率没有 JDK 代理高

实现 CGlib 代理

1. 引入所需要的 jar 包

```
<dependency>
  <groupId>org.ow2.asm</groupId>
  <artifactId>asm</artifactId>
  <version>7.2</version>
</dependency>

<dependency>
  <groupId>org.ow2.asm</groupId>
  <artifactId>asm-commons</artifactId>
  <version>7.2</version>
</dependency>

<dependency>
  <groupId>org.ow2.asm</groupId>
  <artifactId>asm-tree</artifactId>
  <version>7.2</version>
</dependency>

<dependency>
  <groupId>cglib</groupId>
  <artifactId>cglib</artifactId>
  <version>3.3.0</version>
</dependency>
```

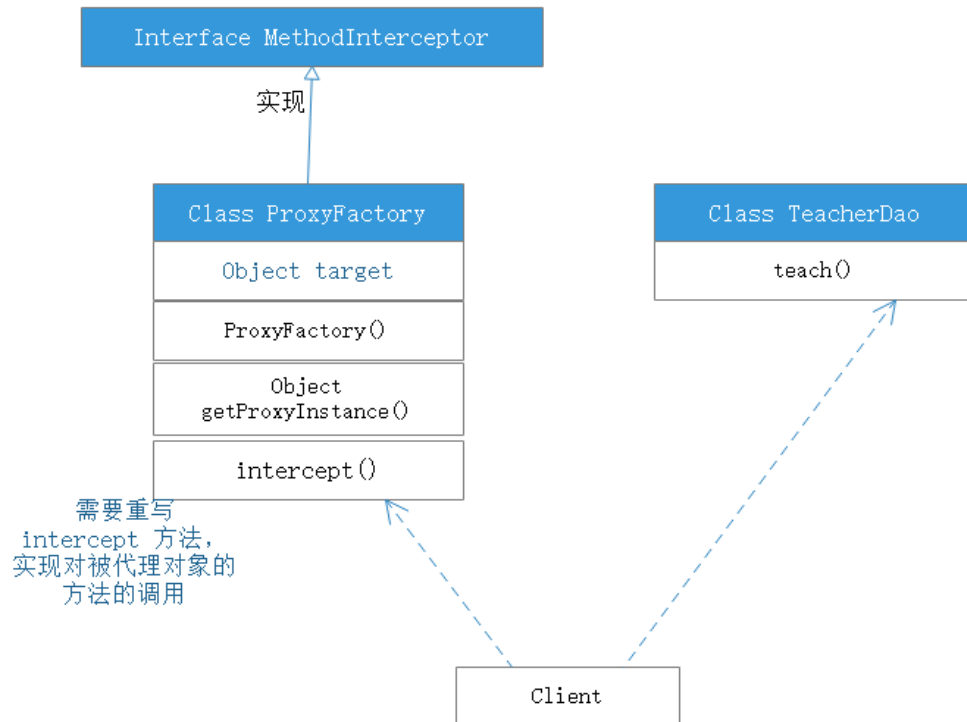
CGlib 代理的几个注意点

- 被代理类不能使 final

- 被代理类的方法如果是 final / static，则该方法不会被拦截，即不能对该方法进行切面编程（增加额外功能）

2. 代码实现

先看下类图



我们先创建被代理对象

```
public class TeacherDao {

    void teach(){
        System.out.println("老师上课中");
    }
}
```

代理工厂类

```
public class ProxyFactory2 implements MethodInterceptor {

    private Object target;

    public ProxyFactory2(Object target){
        this.target = target;
    }

    //返回一个代理对象，是 target 的代理对象
    public Object getProxyInstance(){
        //1. 创建工具类
        Enhancer enhancer = new Enhancer();
        //2. 设置父类，即被代理对象
        enhancer.setSuperclass(target.getClass());
        //3. 设置回调函数
    }
}
```

```

        enhancer.setCallback(this);
        //4. 创建子类对象，即代理对象
        return enhancer.create();
    }

    //重写 intercept，调用代理对象的方法时触发
    public Object intercept(Object o, Method method, Object[] objects,
        MethodProxy methodProxy) throws Throwable {
        System.out.println("cglib代理开始");
        Object re = method.invoke(target, objects);
        System.out.println("cglib代理结束");
        return re;
    }
}

```

客户端

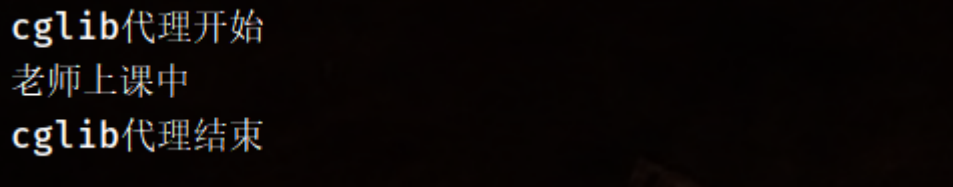
```

public class Client {

    public static void main(String[] args) {

        //创建被代理对象
        TeacherDao teacherDao = new TeacherDao();
        //获取代理对象
        TeacherDao instance = (TeacherDao)new
        ProxyFactory2(teacherDao).getProxyInstance();
        //执行代理对象的方法，以触发执行被代理对象中对应的方法
        instance.teach();
    }
}

```



```

cglib代理开始
老师上课中
cglib代理结束

```

可以看见调用了 intercept 方法

其它代理模式

1. 防火墙代理

内网通过代理穿透防火墙，实现对公网的访问

2. 缓存代理

比如，当请求图片文件等资源时，先到缓存代理里取，如果取到资源，返回即可；如果没有取到资源，再到公网或者数据库取，然后进行缓存

3. 远程代理

远程对象的本地代理，通过它可以把远程对象当做本地对象来调用（RPC 通信）。远程代理通过网络和真正的远程对象进行沟通

4. 同步代理

在多线程编程中，完成多线程间同步工作。比如多个人找同一个黄牛去代买票

总结

Cglib 代理也常被认为是动态代理的一种，其和 JDK 代理的区别如下

	Cglib	JDK
是否提供子类代理	是	否
是否提供接口代理	是	是
区别	1.必须依赖于 CGLib 的类库，基于 Enhancer 来创建被代理对象的子类，用这个子类作为被代理对象 2. 代理对象工厂必须实现 MethodInterceptor，重写 intercept 方法，在这个方法中调用被代理对象的方法。该方法在调用代理对象的方法时触发	1.实现 InvocationHandler 2. 使用 Proxy.newProxyInstance 产生代理对象 3. 被代理的对象必须要实现接口

模板方法模式

需求

有一个制作咖啡的流程，大致如下

- 1. 选择咖啡豆
- 2. 研磨
- 3. 蒸煮

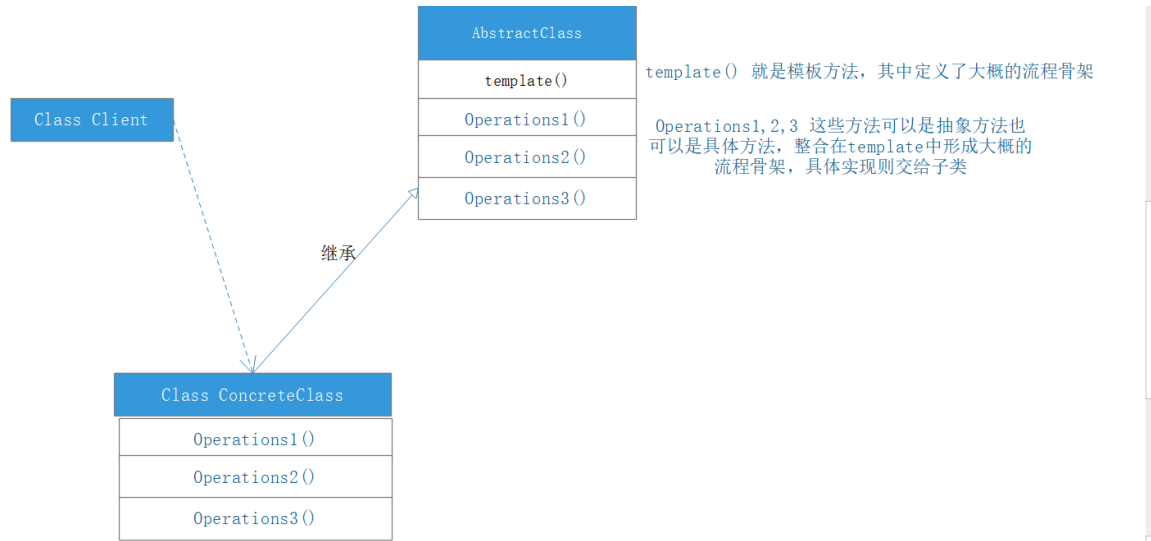
4. 添加配料

通过添加不同的配料，可以做出不同种类的咖啡；咖啡豆的不同种类也会影响咖啡的味道

我们假设：研磨，蒸煮这两个步骤对于制作每一种咖啡来说都是一样的。那如果我们希望上一杯阿拉比卡种咖啡豆制作的卡布奇诺，和一杯罗伯氏特种咖啡中制作的低咖，需要如何去做呢

基本介绍

定义一个操作中的流程骨架，而将流程中的一些不固定的步骤延迟到子类中，使得子类可以不改变该流程结构的情况下重定义该流程的某些特定步骤



代码实现

根据需求构建抽象类

建造抽象类，从我们的需求中可以知道其中有两个步骤是固定的，因此我们可以将固定的步骤直接实现在抽象类中

```
abstract public class CoffeeAbstractClass {

    final public void template(){
        coffeeBean();
        grind();
        cook();
        batching();
    }
    //咖啡豆种类
    abstract protected void coffeeBean();

    private void grind(){
        System.out.println("研磨咖啡豆");
    }

    private void cook(){
        System.out.println("蒸煮咖啡豆");
    }
    //配料
```



```
    abstract protected void batching();  
}
```

建造其实现类

```
public class ArabicaCoffee extends CoffeeAbstractClass {  
    @Override  
    protected void coffeeBean() {  
        System.out.println("使用阿拉比卡咖啡豆");  
    }  
  
    @Override  
    protected void batching() {  
        System.out.println("使用卡布奇诺调料");  
    }  
}
```

```
public class RobustaCoffee extends CoffeeAbstractClass {  
    @Override  
    protected void coffeeBean() {  
        System.out.println("使用罗伯氏特咖啡豆");  
    }  
  
    @Override  
    protected void batching() {  
        System.out.println("使用低咖配料");  
    }  
}
```

客户端

```
public class Client {  
    public static void main(String[] args) {  
  
        CoffeeAbstractClass arabicaCoffee = new ArabicaCoffee();  
        arabicaCoffee.template();  
        System.out.println("-----");  
        CoffeeAbstractClass robustaCoffee = new RobustaCoffee();  
        robustaCoffee.template();  
    }  
}
```

使用阿拉比卡咖啡豆

研磨咖啡豆

蒸煮咖啡豆

使用卡布奇诺调料

使用罗伯氏特咖啡豆

研磨咖啡豆

蒸煮咖啡豆

使用低咖配料

钩子方法

在模板方法模式的抽象类中，我们可以定义一个方法，该方法默认不做任何事情，模板方法只在特定点调用钩子方法。子类视情况而言决定是否复写钩子方法，触发锚点上的钩子方法可能能够在一定程度上改变流程骨架。该方法称为钩子方法，特定点称为锚点

现在，我们需要一杯阿拉比卡的纯咖啡，让我们看下代码应该如何改进

在抽象类中新增一个默认返回 true 的钩子方法

```
abstract public class CoffeeAbstractClass {

    final public void template(){
        coffeeBean();
        grind();
        cook();
        if(hookHasBatching())
            batching();
    }

    abstract protected void coffeeBean();

    private void grind(){
        System.out.println("研磨咖啡豆");
    }

    private void cook(){
        System.out.println("蒸煮咖啡豆");
    }

    abstract protected void batching();

    //钩子方法：决定是否添加配料，默认是要加的
    protected boolean hookHasBatching(){
        return true;
    };
}
```

新增一个纯咖啡类，实现如下

```
public class OnlyCoffee extends CoffeeAbstractClass{
    @Override
    protected void coffeeBean() {
        System.out.println("使用阿拉比卡咖啡豆");
    }

    @Override
    protected void batching() {

    }

    //复写钩子方法，当被模板方法调用时在锚点处返回了false，便不会执行 batching() 方法
    @Override
    protected boolean hookHasBatching() {
        return false;
    }
}
```

注意事项和细节

模板方法一般都需要加上 final 关键字，以防止被子类重写

模板方法模式的基本思路是：算法流程只存在父类中，便于修改。需要修改流程步骤时，只要修改父类的模板方法或修改已经实现的步骤，子类就会继承这些修改，实现了代码的复用

适用场景

- 当要完成某样东西，完成这个东西需要一系列步骤，这些步骤中大部分步骤基本相同，个别步骤可能不同时，考虑使用模板方法模式

缺点：

- 每一个实现都需要一个子类，导致了系统的复杂性

命令模式

需求

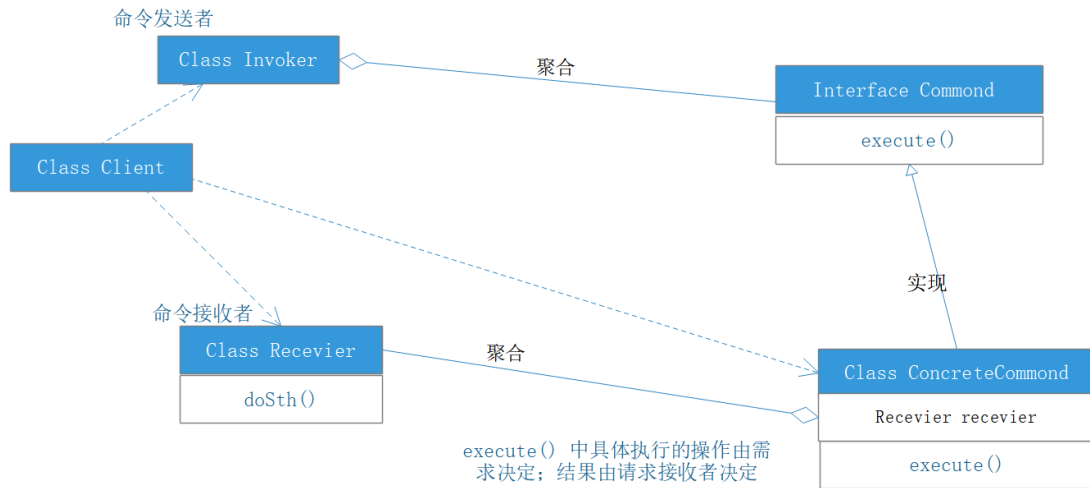
假设：家里新增了一堆智能家电，但是这些智能家电来自于不同的厂商，我们希望有这样一款 APP 能够操作所有的家电。要想办到这件事，每个智能家电厂商都需要提供一个统一的接口家电，来让 APP 能够调用

基本介绍

在软件设计过程中，我们经常需要向某些对象发送请求，但是并不知道请求的接收者是谁，也不知道请求后执行的具体操作是什么，我们只需要指定具体的请求即可。在这种情况下，可以考虑使用命令模式

命令模式使得发送方和接收方解耦，发送方不必关心请求具体给了谁去执行，而只关心发出了什么样的命令

命令模式中，会将一个请求封装成一个对象，以便使用不同的参数来表示不同的命令



ConcreteCommand：抽象命令类（或接口）的具体实现类，聚合接收者对象，并通过调用接收者的方法来完成命令要执行的操作

Receiver：执行命令功能的相关操作，是具体命令对象业务的真正实现者

代码实现

假设我们有如下一个遥控器



建立命令接口

```
public interface Command {  
  
    //执行命令  
    void execute();  
    //撤销命令  
    void undo();  
}
```

建立命令接收者，我们假设 TV 只有关和开两个动作

```

public class TVRecevier {

    public void on(){
        System.out.println("打开电视");
    }

    public void close(){
        System.out.println("关闭电视");
    }

}

```

建立具体的命令，打开电视和关闭电视

```

public class TVOnCommond implements Commond {

    private TVRecevier tvRecevier;

    public TVOnCommond(TVRecevier tvRecevier){
        this.tvRecevier = tvRecevier;
    }

    @Override
    public void execute() {
        tvRecevier.on();
    }

    @Override
    public void undo() {
        tvRecevier.close();
    }

}

```

```

public class TVCloseCommond implements Commond{

    private TVRecevier tvRecevier;

    public TVCloseCommond(TVRecevier tvRecevier){
        this.tvRecevier = tvRecevier;
    }

    @Override
    public void execute() {
        tvRecevier.close();
    }

    @Override
    public void undo() {
        tvRecevier.on();
    }

}

```

现在来看下命令发送者，显然在需求中，我们的命令发送者是遥控器，通过上面假设的遥控板，我们可以知道有不同的电器，**每一个电器有开和关两个命令**，因此，我们可以把开和关两个命令做成两个数组，通过给两个数组的特定位置上赋值命令来达到设置电器命令的作用

同时，我们可以保存每一次的命令，当执行撤销操作时，我们执行保存下来的命令的相反操作

```
public class RemoteController {

    private Command[] onCommands;
    private Command[] closeCommands;
    private Command undoCommands;

    public RemoteController(){
        onCommands = new Command[6];
        closeCommands = new Command[6];
    }

    //设置一组命令，num代表第是给第num个电器设置命令
    public void setCommand(int num, Command onCommand,
                           Command closeCommand){
        this.onCommands[num] = onCommand;
        this.closeCommands[num] = closeCommand;
    }

    //按下了第num个电器的开按钮
    public void onButton(int num){
        onCommands[num].execute();
        //记录本次操作
        undoCommands = onCommands[num];
    }

    public void closeButton(int num){
        closeCommands[num].execute();
        undoCommands = closeCommands[num];
    }

    //按下撤销按钮的处理，如果保存的是开的命令，那么就会执行关的命令
    public void undoButton(){
        undoCommands.undo();
    }
}
```

比如在 set 中，我们给 onCommands[0] 设置了 TV 的 onCommand，closeCommands[0] 设置了 TV 的 closeCommand，那么这就是一组命令，这一组命令构成了遥控器上对 TV 的操作

最后编写 Client，进行测试

```
public class Client {
    public static void main(String[] args) {

        //创建命令接收者
        TVReceiver tvReceiver = new TVReceiver();
        //创建命令
        TVOnCommand tvOnCommand = new TVOnCommand(tvReceiver);
        TVCloseCommand tvCloseCommand = new TVCloseCommand(tvReceiver);
        //组装到遥控器上，并设置相关命令
        RemoteController remoteController = new RemoteController();
        remoteController.setCommand(0, tvOnCommand, tvCloseCommand);
        remoteController.onButton(0);
        remoteController.closeButton(0);
        remoteController.undoButton();
    }
}
```

```
}  
}
```

打开电视
关闭电视
打开电视

在这种模式下，如果我们想要再增加对冰箱的操作，我们需要再增加一个冰箱的开命令对象和关命令对象并实现命令接口，然后增加冰箱这个命令接收者，但是我们无需改动原来的代码，满足了 OCP 原则

宏命令模式

命令模式也可以用来和组合模式联合使用，这样就构成了宏命令模式，也叫组合命令模式。宏命令包含了一组命令，其充当了具体命令实现者与发送者的双重角色，调用宏命令将会递归执行其所有的命令（这种特殊的命令的 execute 方法内部是顺序调用其它若干命令的 execute 方法）

现在我们假设我们的冰箱很高级，当我们按下遥控器上打开冰箱的按钮时，冰箱将会有三个阶段

打开冰箱门->弹出食品柜->关闭冰箱门

现在开始 coding

首先我们创建命令接口

```
public interface Command {  
  
    void execute();  
}
```

然后创建宏命令接口

```
public interface MacroCommand {  
  
    //添加命令  
    void addCommand(Command command);  
  
    //删除命令  
    void deleteCommand(Command command);  
  
    //宏命令执行  
    void execute();  
}
```

然后创建宏命令实现类

```
public class FridgeMacroCommand implements MacroCommand {  
  
    List<Command> commands;  
  
    public FridgeMacroCommand(){
```



```

        commons = new ArrayList<>();
    }

    @Override
    public void addCommond(Commond command) {
        commons.add(command);
    }

    @Override
    public void deleCommond(Commond command) {
        commons.remove(command);
    }

    //执行所有的子命令
    public void execute(){
        for (Commond command : commons) {
            command.execute();
        }
    }
}

```

继续，创建接收者

```

public class FridgeRecevier {

    public void open(){
        System.out.println("打开冰箱");
    }

    public void pushFood(){
        System.out.println("弹出食品柜");
    }

    public void close(){
        System.out.println("关闭冰箱");
    }
}

```

接下来是三个命令对象

```

public class OpenFridgeCommond implements Commond {

    private FridgeRecevier fridge;

    public OpenFridgeCommond(FridgeRecevier fridge){
        this.fridge = fridge;
    }

    @Override
    public void execute() {
        fridge.open();
    }
}

```

```

public class PushFoodCommond implements Commond {

    FridgeRecevier fridge;

    public PushFoodCommond(FridgeRecevier fridge){
        this.fridge = fridge;
    }

    @Override
    public void execute() {
        fridge.pushFood();
    }
}

```

```

public class CloseFridgeCommond implements Commond {

    private FridgeRecevier fridge;

    public CloseFridgeCommond(FridgeRecevier fridge){
        this.fridge = fridge;
    }

    @Override
    public void execute() {
        fridge.close();
    }
}

```

然后客户端调用

```

public class Client {
    public static void main(String[] args) {

        //创建命令接收者
        FridgeRecevier fridge = new FridgeRecevier();
        //创建命令对象
        Commond openFridge = new OpenFridgeCommond(fridge);
        Commond pushFood = new PushFoodCommond(fridge);
        Commond closeFridge = new CloseFridgeCommond(fridge);

        //命令控制对象，充当了调用者和具体命令的双重对象
        MacroCommond macro = new FridgeMacroCommond();
        macro.addCommond(openFridge);
        macro.addCommond(pushFood);
        macro.addCommond(closeFridge);
        //执行宏命令
        macro.execute();
    }
}

```

打开冰箱
弹出食品柜
关闭冰箱

可以看见，我们只执行了一次命令，但是其中却包含了多条命令，这就是宏命令模式

注意事项和细节

命令模式的价值在于将发起请求的对象和执行请求的对象解耦，调用者只需要调用命令对象的 `execute` 方法就可以让接收者工作，并且不需要知道内部实现细节。命令对象会负责让接收者执行命令的动作，命令对象起到了连通调用者和接收者的作用

缺点：

- 可能导致具体命令类的爆炸

访问者模式

需求

我们将观众分为男人和女人，当某个选手表演完成后，他们将对歌手进行不同的评价，评价可以是成功或失败

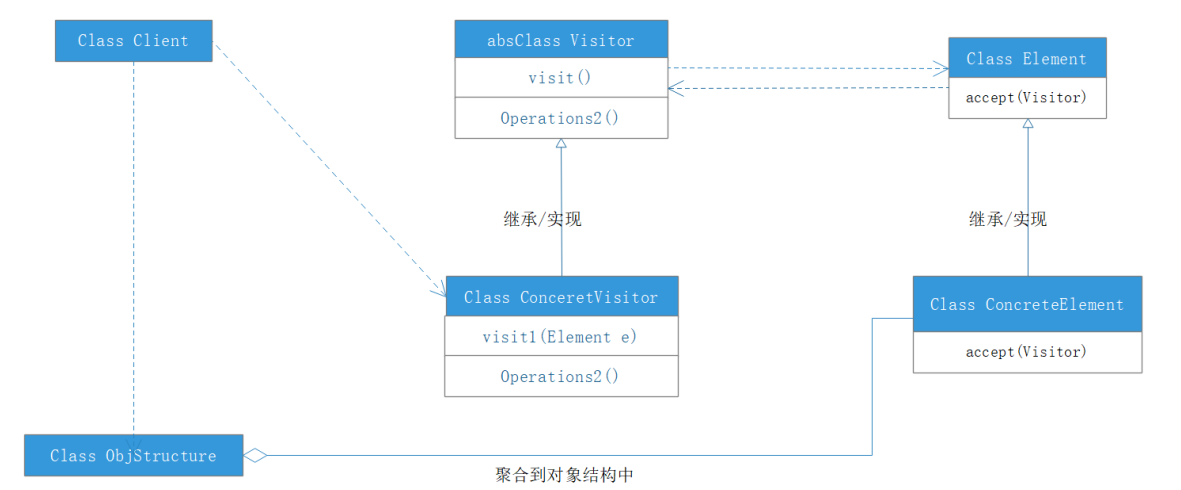
基本介绍

表示一个作用于其对象结构中的各元素的操作，它使你可以在不改变各元素类的前提下定义作用于这些元素的新操作

该模式主要 **将数据结构与数据操作分离**

工作原理：在被访问的类里面增加对外提供接待访问者的接口

本质：**预留通路，回调实现**。它的实现主要就是通过预先定义好调用的通路，在被访问的对象上定义 `accept` 方法，在访问者的对象上定义 `visit` 方法；然后在调用真正发生的时候，通过两次分发的，利用预先定义好的通路，回调到访问者具体的实现上



Visitor：抽象访问者，它定义了对每一个元素（Element）访问的行为，它的参数就是可以访问的元素，它的方法个数理论上讲与元素个数（Element的实现类个数）是一样的，从这点不难看出，访问者模式要求元素类的个数不能改变（不能改变的意思是说，如果元素类的个数经常改变，则说明不适合使用访问者模式）

Concrete Visitor：给出对每一个元素类访问时所产生的具体行为

Element：抽象元素类，定义了一个接受访问者（accept）的方法，其意义是指，每一个元素都要可以被访问者访问

Concrete Element：提供接受访问方法的具体实现，而这个具体的实现，通常情况下是使用访问者提供的访问该元素类的方法

ObjStructure：这个便是定义当中所提到的对象结构，对象结构是一个抽象表述，具体点可以理解为一个具有容器性质或者复合对象特性的类，它会含有一组元素（Element），并且可以迭代这些元素，供访问者访问

代码实现

访问者角色：观众

具体访问者角色：男人和女人

抽象元素角色：投票种类

具体元素角色：成功票，失败票

先创建访问者角色

```
/**
 * 观众
 */
abstract public class Watcher {

    /**
     * 成功票
     */
    abstract public void toAccess(AccessTicketElement accessTicket);

    /**
     * 失败票
     */
    abstract public void toFail(FailTicketElement failTicket);
}
```

然后创建抽象元素及其具体元素

```
abstract public class TicketElement {

    /**
     * “接受”一个观众投票
     */
    abstract public void accept(Watcher watcher);
}
```

```

public class AccessTicketElement extends TicketElement {
    @Override
    public void accept(Watcher watcher) {
        watcher.toAccess(this);
        System.out.println("获得一张成功票");
    }
}

```

```

public class FailTicketElement extends TicketElement {
    @Override
    public void accept(Watcher watcher) {
        watcher.toFail(this);
        System.out.println("获得一张失败票");
    }
}

```

创建具体访问者

```

public class ManWatcher extends Watcher {
    @Override
    public void toAccess(TicketElement accessTicket) {
        System.out.println("男人投了成功票");
    }

    @Override
    public void toFail(TicketElement failTicket) {
        System.out.println("男人投了失败票");
    }
}

```

```

public class WomanWatcher extends Watcher {

    @Override
    public void toAccess(TicketElement accessTicket) {
        System.out.println("女人投了成功票");
    }

    @Override
    public void toFail(TicketElement failTicket) {
        System.out.println("女人投了成功票");
    }
}

```

结构对象

```

/**
 * 结构对象
 */
public class ObjStructure {

    private List<TicketElement> tickets;

    public ObjStructure(){
        tickets = new ArrayList<>();
    }
}

```

```

}

/**
 * 添加一种种类的票
 * @param ticketElement
 */
public void addTicketType(TicketElement ticketElement){
    tickets.add(ticketElement);
}

/**
 * 输出访问者可以访问的所有票
 * @param watcher
 */
public void action(Watcher watcher){
    for (TicketElement ticket : tickets) {
        ticket.accept(watcher);
    }
}
}
}

```

客户端

```

public class Client {
    public static void main(String[] args) {

        ObjStructure objStructure = new ObjStructure();
        objStructure.addTicketType(new AccessTicketElement());
        objStructure.addTicketType(new FailTicketElement());

        objStructure.action(new ManWatcher());
        objStructure.action(new WomanWatcher());
    }
}

```

男人投了成功票
 获得一张成功票
 男人投了失败票
 获得一张失败票
 女人投了成功票
 获得一张成功票
 女人投了成功票
 获得一张失败票

在这种情况下，我们不难发现，如果我们想要增加一种票的种类，这将是十分麻烦的，我们需要在抽象访问者中增加一个访问对应票的方法，同时所有的具体访问者又要去实现它；但是我们增加一个访问者种类却是很方便的

因此，在使用该模式时，我们应该将不常被增加新子类的抽象类作为抽象元素角色，如人；而将经常增加新子类的抽象类作为抽象访问角色，如票

所以上述代码在逻辑上是存在一定问题的，但这里留给各位读者去自行修改

分派的概念

变量被声明时的类型叫做变量的静态类型(Static Type)，有些人又把静态类型叫做明显类型(Apparent Type)；而变量所引用的对象的真实类型又叫做变量的实际类型(Actual Type)

```
Map map = null;

map = new HashMap();
```

声明了一个变量 map，它的静态类型是 Map，实际类型是 HashMap

根据对象的类型而对方法进行的选择，就是分派(Dispatch)，分派(Dispatch)又分为两种，即静态分派和动态分派

静态分派(Static Dispatch)发生在编译时期，分派根据静态类型信息发生。静态分派对于我们来说并不陌生，方法重载就是静态分派。

动态分派(Dynamic Dispatch)发生在运行时期，动态分派动态地置换掉某个方法。

静态分派：Java通过方法重载支持静态分派

动态分派：Java通过方法的重写支持动态分派

双分派

Java 是一种静态多分派，动态单分派的语言，Java 不支持动态的双分派，但是可以通过设计模式来实现动态双分派

在上面的访问者模式中，我们就用到了双分派技术。所谓双分派技术，就是**在选择一个方法的时候，不仅仅要根据消息接收者的运行时类型来区别，还要根据参数的运行时类型来区别。**

在访问者模式中

```
for (TicketElement ticket : tickets) {
    ticket.accept(watcher);
}
```

所有 ticket 的 accept() 方法的接收参数类型都是 watcher，所以方法的参数类型不会影响虚拟机对方法的选择。虚拟机具体是调用 FailTicket 的 accept() 方法还是 AccessTicket 的 accept() 方法，是由 ticket 的实际类型来决定的，这里就是第一次动态单分派

```
watcher.toFail(this);
```

在运行时根据 this 的具体类型来选择是要调用访问者的 toFail() 还是 toAccess() 方法，在此完成了又一次的动态单分派

两次动态单分派结合起来,就完成了一次伪动态双分派,先确定了调用哪个 Ticket 的 accept() 方法,然后再确定了调用 Watcher 中的针对哪个 Ticket 的 toFail() / toAccess() 方法,这就是伪动态双分派

在 accept 的方法实现中,传递 this 进去 具体的 Ticket 根据this的类型,又完成了一次分派,找到了需要调用的方法

注意事项和细节

优点

- 符合单一职责原则
- 对功能进行了统一，适用于数据结构相对稳定的系统

缺点

- 对具体元素角色的扩展支持性极差
- 违背了依赖倒置原则，访问者依赖的是具体元素而不是抽象元素

迭代器模式

需求

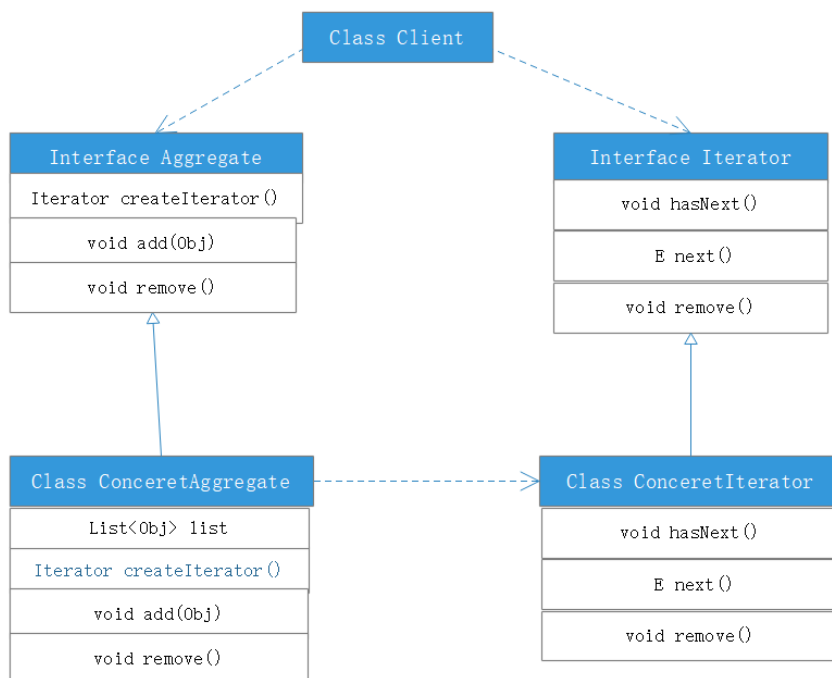
学校下面有学院，学院下有系，要求展示出学校的院系组成。我们尝试使用迭代器模式进行实现

基本介绍

定义：提供一种方法顺序访问一个聚合对象中各个元素, 而又无须暴露该对象的内部表示

如果我们的元素是用不同的容器存储的，有数组，也有集合类，甚至其他方式时，如果客户端想要遍历这些元素集合，就需要使用多种遍历方式，甚至还会暴露元素内部的结构。在这时我们就可以使用迭代器模式

迭代器模式将数据与遍历分离，无需关心数据内部



抽象容器 Aggregate：一般是一个接口，提供一个获得迭代器的方法，和增加，删除等功能

具体容器 ConcreteAggregate：抽象容器的具体实现类，负责实现遍历方式

抽象迭代器 Iterator：定义遍历元素所需要的方法，一般来说会有这么三个方法：取得下一个元素的方法 next()，判断是否遍历结束的方法 hasNext()，移出当前对象的方法 remove()

具体迭代器 ConcretelIterator：持有被聚合对象的集合，并提供一个方法，返回对应集合的迭代器。从理论上来说，一种 ConcretAggregate 对应一种 Concreteliterator。负责存储数据

实现

我们假设我们学校只有计算机学院和理学院，两个学院存储下属院系时，前者用集合存储，后者用数组存储

我们先来写抽象迭代器

```
public interface Iterator {  
  
    boolean hasNext();  
  
    Object next();  
}
```

然后写抽象容器 College

```
public interface College {  
  
    Iterator createIterator();  
    void add(Object object);  
    void remove(Object object);  
    //简写  
    void getName();  
}
```

然后写一个 Department 类，这个类的对象就是被聚合对象

```
public class Department {  
  
    private String name;  
    private String des;  
  
    public Department(String name, String des) {  
        this.name = name;  
        this.des = des;  
    }  
  
    public Department() {}  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public String getDes() {
```

```

        return des;
    }

    public void setDes(String des) {
        this.des = des;
    }
}

```

写具体迭代器，我们的两个学院使用不同的存储方式存储系，所以我们需要两个迭代器

```

public class MathCollegeIterator implements Iterator {

    //规定数据的存放形式，与对应的容器中数据的存放方式相同
    private List<Department> mathDepartments;
    //游标
    private int cur;

    public MathCollegeIterator(List<Department> mathDepartments) {
        this.mathDepartments = mathDepartments;
    }

    public MathCollegeIterator(){}

    @Override
    public boolean hasNext() {
        if(cur >= mathDepartments.size()){
            return false;
        }
        return true;
    }

    @Override
    public Object next() {
        if (hasNext()){
            return mathDepartments.get(cur++);
        }
        return null;
    }
}

```

```

public class ComputerCollegeIterator implements Iterator {

    //规定数据的存放形式，与对应的容器中数据的存放方式相同
    private Department[] departments;
    private int cur;

    public ComputerCollegeIterator(Department[] departments) {
        this.departments = departments;
    }

    public ComputerCollegeIterator(){}

    @Override
    public boolean hasNext() {
        if(cur >= departments.length || departments[cur] == null) {
            return false;
        }
    }
}

```

```

        return true;
    }

    @Override
    public Object next() {
        if (hasNext()){
            return departments[cur++];
        }
        return null;
    }
}

```

然后写具体容器，同样，也是两个

```

public class ComputerCollege implements College {

    private Department[] departments = new Department[5];
    private int index = 0;

    @Override
    public Iterator createIterator() {
        return new ComputerCollegeIterator(departments);
    }

    @Override
    public void add(Object object) {
        departments[index++] = (Department)object;
    }

    @Override
    public void remove(Object object) {

    }

    @Override
    public void getName() {
        System.out.println("计算机学院");
    }
}

```

```

public class MathCollege implements College {

    private List<Department> departments = new ArrayList<>();

    //返回创建好的迭代器
    @Override
    public Iterator createIterator() {
        return new MathCollegeIterator(departments);
    }

    //增加元素
    @Override
    public void add(Object object) {
        departments.add((Department) object);
    }
}

```

```

//删除元素
@Override
public void remove(Object object) {
    departments.remove(object);
}

@Override
public void getName() {
    System.out.println("理学院");
}
}

```

最后，编写客户端进行测试

```

public class Client {
    public static void main(String[] args) {

        //计算机学院
        ComputerCollege computerCollege = new ComputerCollege();
        computerCollege.add(new Department("大数据", "大数据专业"));
        computerCollege.add(new Department("ss", "dwwd"));
        computerCollege.getName();

        Iterator iterator = computerCollege.createIterator();
        while(iterator.hasNext()){
            Department next = (Department)iterator.next();
            System.out.println(next.getName()+"::"+next.getDes());
        }

        System.out.println("-----");
        //理学院
        MathCollege mathCollege = new MathCollege();
        mathCollege.add(new Department("朴素数学", "2222"));
        mathCollege.add(new Department("理论数学", "223455"));
        mathCollege.getName();

        Iterator iterator1 = mathCollege.createIterator();

        while(iterator1.hasNext()){
            Department next = (Department)iterator1.next();
            System.out.println(next.getName()+"::"+next.getDes());
        }
    }
}

```

```

计算机学院
大数据::大数据专业
ss::dwwd
-----
理学院
朴素数学::2222
理论数学::223455

```

迭代器模式与集合是紧密联系的，可以说密不可分，我们只要实现一个集合，就需要同时提供这个集合的迭代器，就像Java中的Collection，List、Set、Map等，这些集合都有自己的迭代器

注意事项和细节

优点

- 提供统一的方法遍历聚合对象 (容器)，而不用考虑这些对象是用什么结构聚合在一起的
- 满足单一职责原则。在迭代器模式中，我们将实际存储数据，操作数据的部分和对数据进行遍历的部分分开成了两部分。这样一来，当我们想要修改遍历方式时，只需要修改对应的迭代器类

缺点

- 每种聚合对象 (容器) 都需要一个迭代器，如果聚合对象过多则迭代器也会过多，不好管理

观察者模式

需求

气象站可以将每天测量到的温度，湿度，气压

其它第三方能够通过气象站的开放 API 来接入气象站，获得数据

测量数据更新时，要能够实时的通知给第三方

普通方案

接入网站可以**手动获取**最新的数据，也可以让气象站向接入网站**推送**最新数据

我们现在以推送方式实现需求上面的需求

接入类

```
/**
 * 显示当前天气情况
 */
public class Conditions {

    //温度，气压，湿度
    private double temperature;
    private double pressure;
    private double humidity;

    //更新天气情况，在推送下，由 wetherdata 调用
    public void update(double temperature, double pressure, double humidity){
        this.temperature = temperature;
        this.pressure = pressure;
        this.humidity = humidity;
        display();
    }
}
```

```

    }

    public void display(){
        System.out.println("现在的温度为: "+temperature);
        System.out.println("现在的气压为: "+pressure);
        System.out.println("现在的湿度为: "+humidity);
    }
}

```

天气数据类

```

/**
 * 包含最新的天气数据
 * 包含接收推送信息的对象
 * 当数据有更新时，就主动调用接收推送消息的对象的 update 方法，
 * 将最新的数据传入
 */
public class WeatherData {

    //温度，气压，湿度，接收推送信息的网站
    private double temperature;
    private double pressure;
    private double humidity;
    private Conditions condition;

    public WeatherData(Conditions condition){
        this.condition = condition;
    }

    public double getTemperature() {
        return temperature;
    }

    public double getPressure() {
        return pressure;
    }

    public double getHumidity() {
        return humidity;
    }

    public void dataChange(){
        condition.update(getTemperature(), getPressure(), getHumidity());
    }

    //三个数据同时更新时，调用接收者的 update 方法
    public void setData(double temperature, double pressure, double humidity){
        this.temperature = temperature;
        this.pressure = pressure;
        this.humidity = humidity;
        dataChange();
    }
}

```

客户端

```

public class Client {
    public static void main(String[] args) {

        Conditions condition = new Conditions();
        WeatherData weatherData = new WeatherData(condition);
        weatherData.setData(43.2, 34.1, 45.2);

        System.out.println("-----天气发生变化-----");

        weatherData.setData(100.2, 12.3, 44.23);

    }
}

```

```

现在的温度为: 43.2
现在的气压为: 34.1
现在的湿度为: 45.2
-----天气发生变化-----
现在的温度为: 100.2
现在的气压为: 12.3
现在的湿度为: 44.23

```

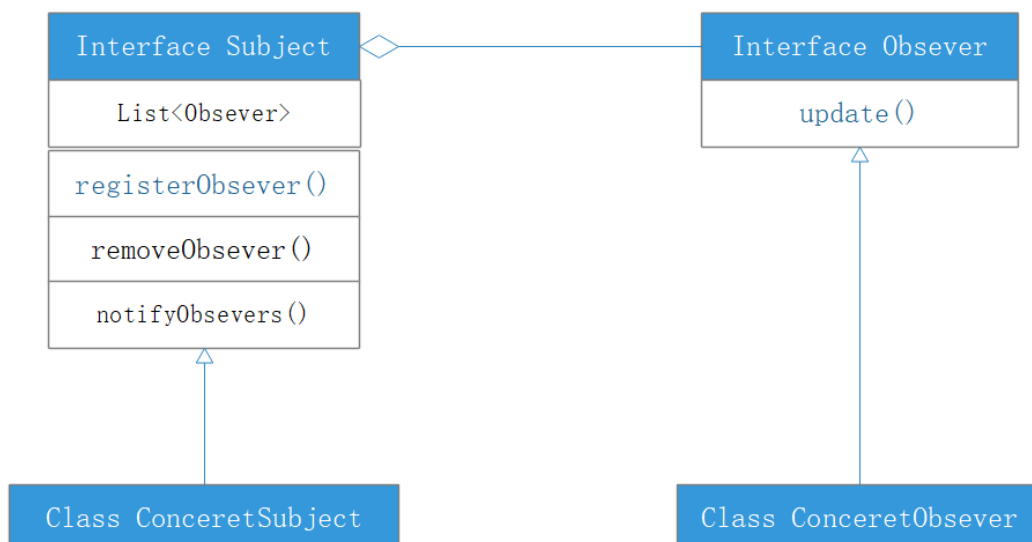
该方案存在的问题

- 没有办法在运行时动态添加第三方，想要添加第三方网站就必须要修改天气数据类，不符合开闭原则

基本介绍

定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新

观察者模式主要解决了当一个对象（目标对象）的状态发生改变，如何让所有的依赖对象（观察者对象）都将得到通知的问题



Subject 接口：具备被观察者的通用方法，如添加观察者，删除观察者，通知观察者等

ConceretSubject: 具体实现添加观察者, 删除观察者, 通知观察者等方法。视主题不同可能具备其它方法或方法实现不同 (如牛奶站和气象局的添加, 删除, 通知等方法内部逻辑可能有差异)

Observer 接口: 为所有的具体观察者定义一个更新接口, 在得到被观察者的 notify 时更新自己

代码实现

先创建 Subject 接口

```
public interface Subject {  
  
    void registerObsever(Obsever obsever);  
    void removeObsever(Obsever obsever);  
    void notifyObsever();  
}
```

然后创建 Obsever 接口

```
public interface Obsever {  
  
    void update(WeatherData weatherData);  
}
```

接口的 update() 方法需要传入具体的被观察者

创建具体的 Subject

```
package 观察者模式.访问者方式;  
  
import java.util.ArrayList;  
import java.util.List;  
  
/**  
 * @Author: antigenMHC  
 * @Date: 2020/7/13 10:50  
 * @Version: 1.0  
 **/  
public class WeatherData implements Subject {  
  
    //温度, 气压, 湿度, 接收推送信息的网站  
    private double temperature;  
    private double pressure;  
    private double humidity;  
    private List<Obsever> obsevers;  
  
    public WeatherData(){  
        obsevers = new ArrayList<>();  
    }  
  
    @Override  
    public void registerObsever(Obsever obsever) {  
        obsevers.add(obsever);  
    }  
}
```



```

@Override
public void removeObserver(Observer observer) {
    observers.remove(observer);
}

//循环，通知所有的观察者，并传入自己的最新状态
@Override
public void notifyObserver() {
    int i = 0;
    for (Observer observer : observers) {
        System.out.println("第"+((i++)+1)+"个网站");
        observer.update(this);
    }
}

//三个数据更新后，调用 Notify，更新所有的观察者
public void setData(double temperature, double pressure, double humidity){
    this.temperature = temperature;
    this.pressure = pressure;
    this.humidity = humidity;
    notifyObserver();
}

public double getTemperature() {
    return temperature;
}

public double getPressure() {
    return pressure;
}

public double getHumidity() {
    return humidity;
}

public void setTemperature(double temperature) {
    this.temperature = temperature;
}

public void setPressure(double pressure) {
    this.pressure = pressure;
}

public void setHumidity(double humidity) {
    this.humidity = humidity;
}
}

```

然后创建对应的具体的观察者

```

public class WeatherObserver implements Observer {

    //温度，气压，湿度
    private double temperature;
    private double pressure;

```

```

private double humidity;

//更新天气情况并展示
@Override
public void update(WeatherData weatherData){
    this.temperature = weatherData.getTemperature();
    this.pressure = weatherData.getPressure();
    this.humidity = weatherData.getHumidity();
    display();
}

public void display(){
    System.out.println("现在的温度为: "+temperature);
    System.out.println("现在的气压为: "+pressure);
    System.out.println("现在的湿度为: "+humidity);
}
}

```

最后创建客户端进行测试

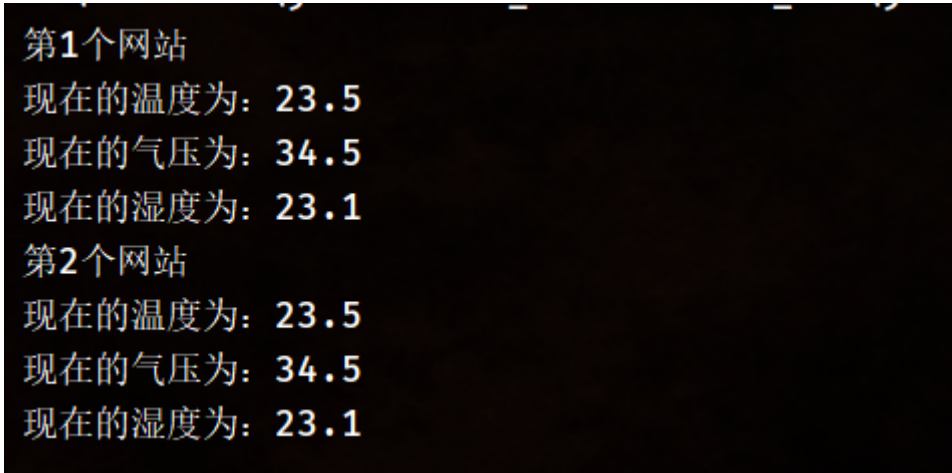
```

public class Client {
    public static void main(String[] args) {

        WeatherData weatherData = new WeatherData();
        weatherData.registerObserver(new WeatherObserver());
        weatherData.registerObserver(new WeatherObserver());

        weatherData.setData(23.5, 34.5, 23.1);
    }
}

```



```

第1个网站
现在的温度为: 23.5
现在的气压为: 34.5
现在的湿度为: 23.1
第2个网站
现在的温度为: 23.5
现在的气压为: 34.5
现在的湿度为: 23.1

```

从代码中不难看出调用逻辑

1. 创建被观察者，将观察者聚合到被观察者中
2. 当被观察者状态发生变化时，观察者自动更新展示最新的数据

在这种模式下，如果我们想要新增新的第三方网站(观察者)，只需要实现 Observer 接口即可，然后聚合进被观察者即可，满足了 OCP 原则

我们新增一个接入的第三方网站，阿里

```

public class AlibabaObserver implements Observer {
    //温度，气压，湿度
}

```

```

private double temperature;
private double pressure;
private double humidity;

//更新天气情况
@Override
public void update(WeatherData weatherData){
    this.temperature = weatherData.getTemperature();
    this.pressure = weatherData.getPressure();
    this.humidity = weatherData.getHumidity();
    display();
}

public void display(){
    System.out.println("Alibaba显示现在的温度为: "+temperature);
    System.out.println("Alibaba现在的气压为: "+pressure);
    System.out.println("Alibaba现在的湿度为: "+humidity);
}
}

```

然后只需要在客户端中将这个观察者聚合到被观察者中

```

public class Client {
    public static void main(String[] args) {

        //创建被观察者
        WeatherData weatherData = new WeatherData();
        //注册观察者
        weatherData.registerObserver(new WeatherObserver());
        weatherData.registerObserver(new WeatherObserver());
        weatherData.registerObserver(new AlibabaObserver());
        //更新数据
        weatherData.setData(23.5, 34.5, 23.1);
    }
}

```

第1个网站

现在的温度为: 23.5

现在的气压为: 34.5

现在的湿度为: 23.1

第2个网站

现在的温度为: 23.5

现在的气压为: 34.5

现在的湿度为: 23.1

第3个网站

Alibaba显示现在的温度为: 23.5

Alibaba现在的气压为: 34.5

Alibaba现在的湿度为: 23.1

注意事项和细节

优点

- （观察者和被观察者是抽象耦合的）原本被观察者对象在状态改变的时候，需要直接调用所有的观察者对象，但是抽象出观察者接口过后，被观察者和观察者就只是在抽象层面上耦合了，也就是说被观察者只是知道观察者接口，并不知道具体的观察者的类，从而实现被观察者类和具体的观察者类之间解耦
- （实现了动态联动）联动就是做一个操作会引起其它相关的操作。由于观察者模式对观察者注册实行管理，那就可以在运行期间，通过动态的控制注册的观察者，来控制某个动作的联动范围，从而实现动态联动
- （支持广播通信）由于被观察者发送通知给观察者是面向所有注册的观察者，所以每次被观察者通知的信息就要对所有注册的观察者进行广播。当然，也可以通过在被观察者上添加新的功能来限制广播的范围

缺点

- 如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间
- 如果在观察者和被观察者之间有循环依赖的话，被观察者会触发它们形成循环调用，可能导致系统崩溃
- 观察者模式没有相应的机制让观察者知道被观察者对象是怎么发生变化的，而仅仅只是知道被观察者发生了变化