

设计模式公司荣誉出品

您的设计模式

我们的设计模式

CBF4LIFE

2009 年 5 月

我希望这本书的读者具备最基本的代码编写能力，您是一个初级的 coder, 可以从中领会到怎么设计一段优秀的代码；您是一个高级程序员，可以从中全面了解到设计模式以及 Java 的边角技术的使用；您是一个顶级的系统分析师，可以从中获得共鸣，寻找到项目公共问题的解决办法，呀，是不是把牛吹大了?!

目 录

第 1 章	策略模式【STRATEGY PATTERN】	4
第 2 章	代理模式【PROXY PATTERN】	8
第 3 章	单例模式【SINGLETON PATTERN】	12
第 4 章	多例模式【MULTITION PATTERN】	16
第 5 章	工厂方法模式【FACTORY METHOD PATTERN】	19
第 6 章	抽象工厂模式【ABSTRACT FACTORY PATTERN】	31
第 7 章	门面模式【FACADE PATTERN】	44
第 8 章	适配器模式【ADAPTER PATTERN】	51
第 9 章	模板方法模式【TEMPLATE METHOD PATTERN】	63
第 10 章	建造者模式【BUILDER PATTERN】	82
第 11 章	桥梁模式【BRIDGE PATTERN】	97
第 12 章	命令模式【COMMAND PATTERN】	112
第 13 章	装饰模式【DECORATOR PATTERN】	126
第 14 章	迭代器模式【ITERATOR PATTERN】	137
第 15 章	组合模式【COMPOSITE PATTERN】	147
第 16 章	观察者模式【OBSERVER PATTERN】	175
第 17 章	责任链模式【CHAIN OF RESPONSIBILITY PATTERN】	194
第 18 章	访问者模式【VISITOR PATTERN】	210
第 19 章	状态模式【STATE PATTERN】	236
第 20 章	原型模式【PROTOTYPE PATTERN】	255
第 21 章	中介者模式【MEDIATOR PATTERN】	268
第 22 章	解释器模式【INTERPRETER PATTERN】	286
第 23 章	享元模式【FLYWEIGHT PATTERN】	298
第 24 章	备忘录模式【MEMENTO PATTERN】	313
第 25 章	模式大 PK.....	333
第 26 章	六大设计原则	334
26.1	单一职责原则【SINGLE RESPONSIBILITY PRINCIPLE】	334
26.2	里氏替换原则【LISKOV SUBSTITUTION PRINCIPLE】	341
26.3	依赖倒置原则【DEPENDENCE	

INVERSION PRINCIPLE】 353

26.4 接口隔离原则【INTERFACE SEGREGATION PRINCIPLE】 354

26.5 迪米特法则【LOW OF DEMETER】 364

26.6 开闭原则【OPEN CLOSE PRINCIPLE】 374

第 27 章 混编模式讲解391

27.1 命令模式+责任链模式 392

第 28 章 更新记录:413

相关说明414

相关说明414

第 29 章 后序.....415

第 1 章 策略模式【Strategy Pattern】

刘备要到江东娶老婆了，走之前诸葛亮给赵云（伴郎）三个锦囊妙计，说是按天机拆开解决棘手问题，嘿，还别说，真是解决了大问题，搞到最后是周瑜陪了夫人又折兵呀，那咱们先看看这个场景是什么样子的。

先说这个场景中的要素：三个妙计，一个锦囊，一个赵云，妙计是小亮同志给的，妙计是放置在锦囊里，俗称就是锦囊妙计嘛，那赵云就是一个干活的人，从锦囊中取出妙计，执行，然后获胜，用 JAVA 程序怎么表现这个呢？我们先看类图：



三个妙计是同一类型的东东，那咱就写个接口：

```

package com.cbf4life.strategy;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 首先定一个策略接口，这是诸葛亮老人家给赵云的三个锦囊妙计的接口
 *
 */
public interface IStrategy {

    //每个锦囊妙计都是一个可执行的算法
    public void operate();

}
  
```

然后再写三个实现类，有三个妙计嘛：

```
package com.cbf4life.strategy;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 找乔国老帮忙，使孙权不能杀刘备
 */
public class BackDoor implements IStrategy {

    public void operate() {
        System.out.println("找乔国老帮忙，让吴国太给孙权施加压力");
    }

}

package com.cbf4life.strategy;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 求吴国太开个绿灯
 */
public class GivenGreenLight implements IStrategy {

    public void operate() {
        System.out.println("求吴国太开个绿灯,放行!");
    }

}

package com.cbf4life.strategy;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 孙夫人断后，挡住追兵
 */
public class BlockEnemy implements IStrategy {

    public void operate() {
```

```
        System.out.println("孙夫人断后，挡住追兵");
    }

}
```

好了，大家看看，三个妙计是有了，那需要有个地方放这些妙计呀，放锦囊呀：

```
package com.cbf4life.strategy;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 计谋有了，那还要有锦囊
 */
public class Context {
    //构造函数，你要使用那个妙计
    private IStrategy strategy;
    public Context(IStrategy strategy){
        this.strategy = strategy;
    }

    //使用计谋了，看我出招了
    public void operate(){
        this.strategy.operate();
    }
}
```

然后就是赵云雄赳赳的揣着三个锦囊，拉着已步入老年行列的、还想着娶纯情少女的、色迷迷的刘老爷子去入赘了，嗨，还别说，小亮的三个妙计还真是不错，瞅瞅：

```
package com.cbf4life.strategy;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 */
public class ZhaoYun {

    /**
     * 赵云出场了，他根据诸葛亮给他的交代，依次拆开妙计
     */
    public static void main(String[] args) {
        Context context;
```

```

//刚刚到吴国的时候拆第一个
System.out.println("-----刚刚到吴国的时候拆第一个-----");
context = new Context(new BackDoor()); //拿到妙计
context.operate(); //拆开执行
System.out.println("\n\n\n\n\n\n\n\n");

//刘备乐不思蜀了，拆第二个了
System.out.println("-----刘备乐不思蜀了，拆第二个了-----");
context = new Context(new GivenGreenLight());
context.operate(); //执行了第二个锦囊了
System.out.println("\n\n\n\n\n\n\n\n");

//孙权的小兵追了，咋办？拆第三个
System.out.println("-----孙权的小兵追了，咋办？拆第三个
-----");
context = new Context(new BlockEnemy());
context.operate(); //孙夫人退兵
System.out.println("\n\n\n\n\n\n\n\n");

/*
 * 问题来了：赵云实际不知道是哪个策略呀，他只知道拆第一个锦囊，
 * 而不知道是BackDoor这个妙计，咋办？ 似乎这个策略模式已经把计谋名称写出来了
 *
 * 错！BackDoor、GivenGreenLight、BlockEnemy只是一个代码，你写成first、second、
third, 没人会说你错！
 *
 * 策略模式的好处就是：体现了高内聚低耦合的特性呀，缺点嘛，这个那个，我回去再查查
 */
}

}

```

就这三招，搞的周郎是“陪了夫人又折兵”呀！这就是策略模式，高内聚低耦合的特点也表现出来了，还有一个就是扩展性，也就是 OCP 原则，策略类可以继续增加下去，只要修改 Context.java 就可以了，这个不多说了，自己领会吧。

第2章 代理模式【Proxy Pattern】

什么是代理模式呢？我很忙，忙的没空理你，那你要找我呢就先找我的代理人吧，那代理人总要知道被代理人能做哪些事情不能做哪些事情吧，那就是两个人具备同一个接口，代理人虽然不能干活，但是被代理的人能干活呀。

比如西门庆找潘金莲，那潘金莲不好意思答复呀，咋办，找那个王婆做代理，表现在程序上时这样的：先定义一种类型的女人：

```
package com.cbf4life.proxy;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 定义一种类型的女人，王婆和潘金莲都属于这个类型的女人
 */
public interface KindWomen {

    //这种类型的女人能做什么事情呢？
    public void makeEyesWithMan(); //抛媚眼

    public void happyWithMan(); //happy what? You know that!

}
```

一种类型嘛，那肯定是接口，然后定义潘金莲：

```
package com.cbf4life.proxy;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 定一个潘金莲是什么样的人
 */
public class PanJinLian implements KindWomen {

    public void happyWithMan() {
        System.out.println("潘金莲在和男人做那个.....");
    }

}
```



```
        public void makeEyesWithMan() {
            System.out.println("潘金莲抛媚眼");
        }
    }
}
```

再定一个丑陋的王婆:

```
package com.cbf4life.proxy;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 王婆这个人老聪明了，她太老了，是个男人都看不上，
 * 但是她有智慧有经验呀，她作为一类女人的代理！
 */
public class WangPo implements KindWomen {
    private KindWomen kindWomen;

    public WangPo(){ //默认的话，是潘金莲的代理
        this.kindWomen = new PanJinLian();
    }

    //她可以是KindWomen的任何一个女人的代理，只要你是这一类型
    public WangPo(KindWomen kindWomen){
        this.kindWomen = kindWomen;
    }

    public void happyWithMan() {
        this.kindWomen.happyWithMan(); //自己老了，干不了，可以让年轻的代替
    }

    public void makeEyesWithMan() {
        this.kindWomen.makeEyesWithMan(); //王婆这么大年龄了，谁看她抛媚眼？！
    }
}
}
```

两个女主角都上场了，男主角也该出现了:

```

package com.cbf4life.proxy;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 定义一个西门庆，这人色中饿鬼
 */
public class XiMenQing {

    /**
     * 水浒里是这样写的：西门庆被潘金莲用竹竿敲了一下难道，痴迷了，
     * 被王婆看到了，就开始撮合两人好事，王婆作为潘金莲的代理人
     * 收了不少好处费，那我们假设一下：
     * 如果没有王婆在中间牵线，这两个不要脸的能成吗？难说的很！
     */
    public static void main(String[] args) {
        //把王婆叫出来
        WangPo wangPo = new WangPo();

        //然后西门庆就说，我要和潘金莲happy，然后王婆就安排了西门庆丢筷子的那出戏：
        wangPo.makeEyesWithMan(); //看到没，虽然表面上时王婆在做，实际上爽的是潘金莲
        wangPo.happyWithMan();    }
}

```

那这就是活生生的一个例子，通过代理人实现了某种目的，如果真去掉王婆这个中间环节，直接是西门庆和潘金莲勾搭，估计很难成就武松杀嫂事件。

那我们再考虑一下，水浒里还有没有这类型的女人？有，卢俊义的老婆贾氏（就是和那个固管家苟合的那个），这名字起的：“假使”，那我们也让王婆做她的代理：

把贾氏素描出来：

```

package com.cbf4life.proxy;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 */
public class JiaShi implements KindWomen {

    public void happyWithMan() {
        System.out.println("贾氏正在Happy中.....");
    }
}

```

```
    }

    public void makeEyesWithMan() {
        System.out.println("贾氏抛媚眼");
    }
}
```

西门庆勾贾氏:

```
package com.cbf4life.proxy;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 定义一个西门庆，这人色中饿鬼
 */
public class XiMenQing {

    public static void main(String[] args) {
        //改编一下历史，贾氏被西门庆勾走：
        JiaShi jiaShi = new JiaShi();
        WangPo wangPo = new WangPo(jiaShi); //让王婆作为贾氏的代理人

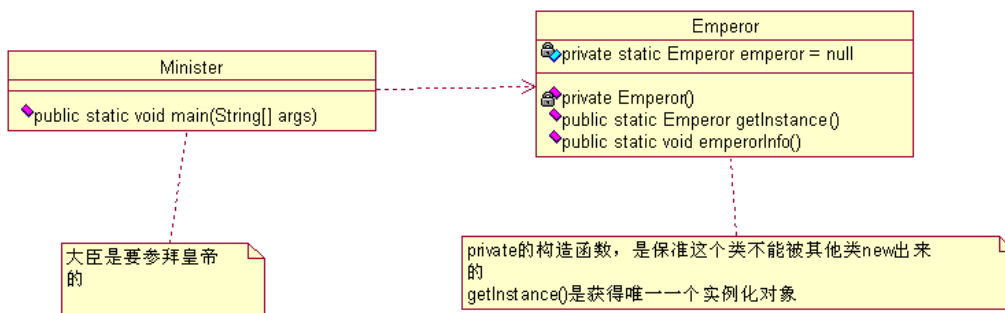
        wangPo.makeEyesWithMan();
        wangPo.happyWithMan();
    }
}
```

说完这个故事，那额总结一下，代理模式主要使用了 Java 的多态，干活的是被代理类，代理类主要是接活，你让我干活，好，我交给幕后的类去干，你满意就成，那怎么知道被代理类能不能干呢？同根就成，大家知根知底，你能做啥，我能做啥都清楚的很，同一个接口呗。

第3章 单例模式【Singleton Pattern】

这个模式是很有意思，而且比较简单，但是我还是要说因为它使用的是如此的广泛，如此的有人缘，单例就是单一、独苗的意思，那什么是独一份呢？你的思维是独一份，除此之外还有什么不能山寨的呢？我们举个比较难复制的对象：皇帝

中国的历史上很少出现两个皇帝并存的时期，是有，但不多，那我们就认为皇帝是个单例模式，在这个场景中，有皇帝，有大臣，大臣是天天要上朝参见皇帝的，今天参拜的皇帝应该和昨天、前天的一样（过渡期的不考虑，别找茬哦），大臣磕完头，抬头一看，嗨，还是昨天那个皇帝，单例模式，绝对的单例模式，先看类图：



然后我们看程序实现，先定一个皇帝：

```
package com.cbf4life.singleton1;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 中国的历史上一般都是一个朝代一个皇帝，有两个皇帝的話，必然要PK出一个皇帝出来
 */
public class Emperor {

    private static Emperor emperor = null; //定义一个皇帝放在那里，然后给这个皇帝名字

    private Emperor(){
        //世俗和道德约束你，目的就是不允许你产生第二个皇帝
    }

    public static Emperor getInstance(){
```

```
        if(emperor == null){ //如果皇帝还没有定义，那就定一个
            emperor = new Emperor();
        }
        return emperor;
    }

    //皇帝叫什么名字呀
    public static void emperorInfo(){
        System.out.println("我就是皇帝某某...");
    }
}
```

然后定义大臣：

```
package com.cbf4life.singleton1;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 大臣是天天要面见皇帝，今天见的皇帝和昨天的，前天不一样那就出问题了！
 */
@SuppressWarnings("all")
public class Minister {

    /**
     * @param args
     */
    public static void main(String[] args) {
        //第一天
        Emperor emperor1=Emperor.getInstance();
        emperor1.emperorInfo(); //第一天见的皇帝叫什么名字呢？

        //第二天
        Emperor emperor2=Emperor.getInstance();
        Emperor.emperorInfo();

        //第三天
        Emperor emperor3=Emperor.getInstance();
        emperor2.emperorInfo();

        //三天见的皇帝都是同一个人，荣幸吧！
    }
}
```

看到没，大臣天天见到的都是同一个皇帝，不会产生错乱情况，反正都是一个皇帝，是好是坏就这一个，只要提到皇帝，大家都知道指的是谁，清晰，而又明确。问题是这是通常情况，还有个例的，如同一个时期同一个朝代有两个皇帝，怎么办？

单例模式很简单，就是在构造函数中多了加一个构造函数，访问权限是 `private` 的就可以了，这个模式是简单，但是简单中透着风险，风险？什么风险？在一个 B/S 项目中，每个 HTTP Request 请求到 J2EE 的容器上后都创建了一个线程，每个线程都要创建同一个单例对象，怎么办？，好，我们写一个通用的单例程序，然后分析一下：

```
package com.cbf4life.singleton3;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 通用单例模式
 */
@SuppressWarnings("all")
public class SingletonPattern {
    private static SingletonPattern singletonPattern= null;
    //限制住不能直接产生一个实例
    private SingletonPattern(){

    }

    public SingletonPattern getInstance(){
        if(this.singletonPattern == null){ //如果还没有实例，则创建一个
            this.singletonPattern = new SingletonPattern();
        }
        return this.singletonPattern;
    }
}
```

我们来看黄色的那一部分，假如现在有两个线程 A 和线程 B，线程 A 执行到 `this.singletonPattern = new SingletonPattern()`，正在申请内存分配，可能需要 0.001 微秒，就在这 0.001 微秒之内，线程 B 执行到 `if(this.singletonPattern == null)`，你说这个时候这个判断条件是 `true` 还是 `false`？是 `true`，那然后呢？线程 B 也往下走，于是乎就在内存中就有两个 `SingletonPattern` 的实例了，看看是不是出问题了？

如果你这个单例是去拿一个序列号或者创建一个信号资源的时候，会怎么样？业务逻辑混乱！数据一致性校验失败！最重要的是你从代码上还看不出什么问题，这才是最要命的！因为这种情况基本上你是重现不了的，不寒而栗吧，那怎么修改？有很多种方案，我就说一种，能简单的、彻底解决问题的方案：

```
package com.cbf4life.singleton3;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 通用单例模式
 */
@SuppressWarnings("all")
public class SingletonPattern {
    private static final SingletonPattern singletonPattern= new
    SingletonPattern();

    // 限制住不能直接产生一个实例
    private SingletonPattern(){
    }

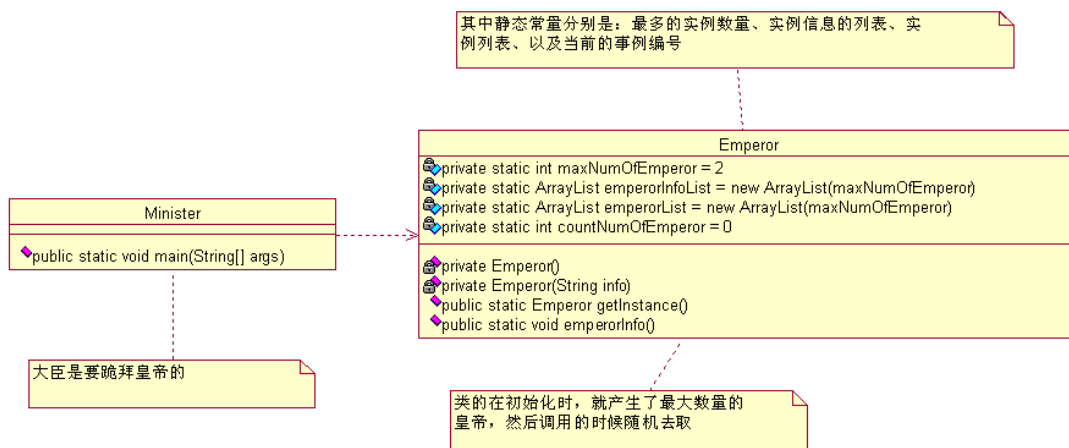
    public synchronized static SingletonPattern getInstance(){
        return singletonPattern;
    }
}
```

直接 new 一个对象传递给类的成员变量 singletonpattern，你要的时候 getInstance（）直接返回给你，解决问题！

第4章 多例模式【Multition Pattern】

这种情况有没有？有！大点声，有没有？有！，是，确实有，就出现在明朝，那三国期间的算不算，不算，各自称帝，各有各的地盘，国号不同。大家还记得那首诗《石灰吟》吗？作者是谁？于谦，他是被谁杀死的？明英宗朱祁镇，对，就是那个在土木堡之变中被瓦剌俘虏的皇帝，被俘虏后，他弟弟朱祁钰当上了皇帝，就是明景帝，估计当上皇帝后乐疯了，忘记把老哥朱祁镇削为太上皇了，我 Shit，在中国的历史上就这个时期是有 2 个皇帝，你说这期间的大臣多郁闷，两个皇帝耶，两个精神依附对象呀。

这个场景放到我们设计模式中就是叫有上限的多例模式（没上限的多例模式太容易了，和你直接 new 一个对象没啥差别，不讨论）怎么实现呢，看我出招，先看类图：



然后看程序，先把两个皇帝定义出来：

```
package com.cbf4life.singleton2;
```

```
import java.util.ArrayList;
```

```
import java.util.Random;
```

```
/**
```

```
 * @author cbf4Life cbf4life@126.com
```

```
 * I'm glad to share my knowledge with you all.
```

```
 * 中国的历史上一般都是一个朝代一个皇帝，有两个皇帝的話，必然要PK出一个皇帝出来。
```

```
 * 问题出来了：如果真在一个时间，中国出现了两个皇帝怎么办？比如明朝土木堡之变后，
```

```
 * 明英宗被俘虏，明景帝即位，但是明景帝当上皇帝后乐疯了，竟然忘记把他老哥明英宗削为太上皇，
```

```
 * 也就是在这一个多月的时间内，中国竟然有两个皇帝！
```

```
 *
```

```
 */
```



```

@SuppressWarnings("all")
public class Emperor {
    private static int maxNumOfEmperor = 2; //最多只能有连个皇帝
    private static ArrayList emperorInfoList=new ArrayList(maxNumOfEmperor); //
    皇帝叫什么名字
    private static ArrayList emperorList=new ArrayList(maxNumOfEmperor); //装皇
    帝的列表;
    private static int countNumOfEmperor =0; //正在被人尊称的是那个皇帝

    //先把2个皇帝产生出来
    static{
        //把所有的皇帝都产生出来
        for(int i=0;i<maxNumOfEmperor;i++){
            emperorList.add(new Emperor("皇"+(i+1)+"帝"));
        }
    }

    //就这么多皇帝了，不允许再推举一个皇帝(new 一个皇帝)
    private Emperor(){
        //世俗和道德约束你，目的就是不允许你产生第二个皇帝
    }

    private Emperor(String info){
        emperorInfoList.add(info);
    }

    public static Emperor getInstance(){
        Random random = new Random();
        countNumOfEmperor = random.nextInt(maxNumOfEmperor); //随机拉出一个皇帝，
    只要是个精神领袖就成
        return (Emperor)emperorList.get(countNumOfEmperor);
    }

    //皇帝叫什么名字呀
    public static void emperorInfo(){
        System.out.println(emperorInfoList.get(countNumOfEmperor));
    }
}

```

那大臣是比较悲惨了，两个皇帝呀，两个老子呀，怎么拜呀，不管了，只要是个皇帝就成：

```
package com.cbf4life.singleton2;
```

```
/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 大臣们悲惨了，一个皇帝都伺候不过来了，现在还来了两个个皇帝
 * TND，不管了，找到个皇帝，磕头，请按就成了！
 */
@SuppressWarnings("all")
public class Minister {

    /**
     * @param args
     */
    public static void main(String[] args) {

        int ministerNum =10; //10个大臣

        for(int i=0;i<ministerNum;i++){
            Emperor emperor = Emperor.getInstance();
            System.out.print("第" +(i+1) + "个大臣参拜的是: ");
            emperor.emperorInfo();
        }
    }
}
```

那各位看官就可能会不屑了：有的大臣可是有骨气，只拜一个真神，你怎么处理？这个问题太简单，懒的详细回答你，getInstance(param)是不是就解决了这个问题？！自己思考，太 Easy 了。

第5章 工厂方法模式【Factory Method Pattern】

女娲补天的故事大家都听说过吧，今天不说这个，说女娲创造人的故事，可不是“造人”的工作，这个词被现代人滥用了。这个故事是说，女娲在补了天后，下到凡间一看，哇塞，风景太优美了，天空是湛蓝的，水是清澈的，空气是清新的，太美丽了，然后就待时间长了就有点寂寞了，没有动物，这些看的到都是静态的东西呀，怎么办？

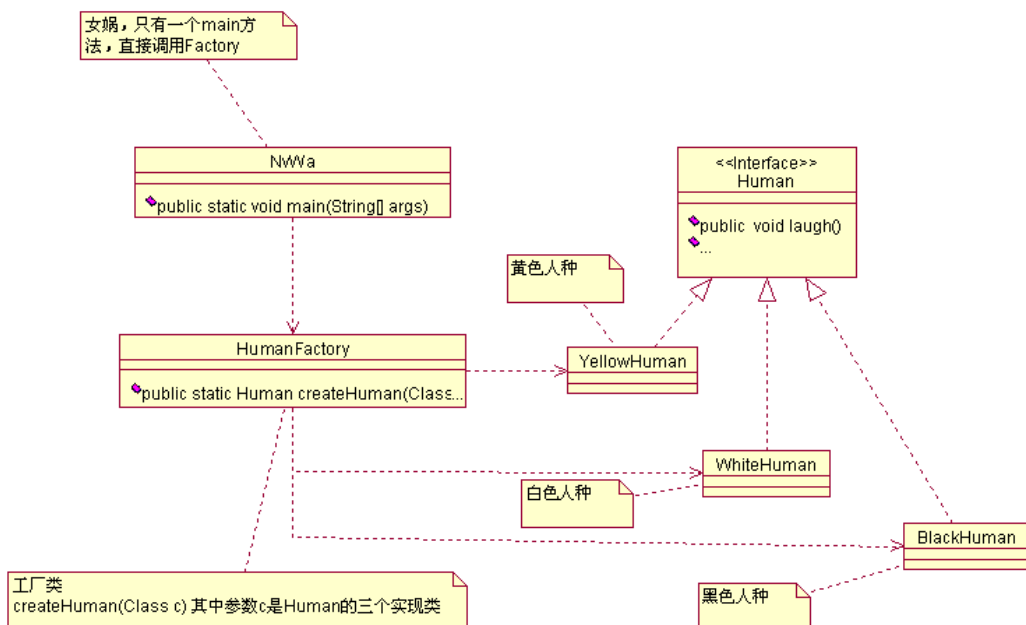
别忘了是神仙呀，没有办不到的事情，于是女娲就架起了八卦炉（技术术语：建立工厂）开始创建人，具体过程是这样的：先是泥巴捏，然后放八卦炉里烤，再扔到地上成长，但是意外总是会产生：

第一次烤泥人，兹兹兹兹~~，感觉应该熟了，往地上一扔，biu~，一个白人诞生了，没烤熟！

第二次烤泥人，兹兹兹兹兹兹兹兹~~，上次都没烤熟，这次多烤会儿，往地上一扔，嘿，熟过头了，黑人哪！

第三次烤泥人，兹~兹~兹~，一边烤一边看着，嘿，正正好，Perfect！优品，黄色人类！【备注:RB 人不属此列】

这个过程还是比较有意思的，先看看类图：（之前在论坛上有兄弟建议加类图和源文件，以后的模式都会加上去，之前的会一个一个的补充，目的是让大家看着舒服，看着愉悦，看着就想要，就像是看色情小说一样，目标，目标而已，能不能实现就看大家给我的信心了）



那这个过程我们就用程序来表现，首先定义一个人类的总称：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 定义一个人类的统称
 */
public interface Human {
    //首先定义什么是人类

    //人是愉快的，会笑的，本来是想用smile表示，想了一下laugh更合适，好长时间没有大笑了；
    public void laugh();

    //人类还会哭，代表痛苦
    public void cry();

    //人类会说话
    public void talk();
}
```

然后定义具体的人类：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 黄色人类，这个翻译的不准确，将就点吧
 */
public class YellowHuman implements Human {

    public void cry() {
        System.out.println("黄色人类会哭");
    }

    public void laugh() {
        System.out.println("黄色人类会大笑，幸福呀！");
    }

    public void talk() {
        System.out.println("黄色人类会说话，一般说的都是双字节");
    }
}
```

```
    }  
}
```

白色人类:

```
package com.cbf4life;  
  
/**  
 * @author cbf4Life cbf4life@126.com  
 * I'm glad to share my knowledge with you all.  
 * 白色人类  
 */  
public class WhiteHuman implements Human {  
  
    public void cry() {  
        System.out.println("白色人类会哭");  
    }  
  
    public void laugh() {  
        System.out.println("白色人类会大笑，侵略的笑声");  
    }  
  
    public void talk() {  
        System.out.println("白色人类会说话，一般都是但是单字节!");  
    }  
}
```

黑色人类:

```
package com.cbf4life;  
  
/**  
 * @author cbf4Life cbf4life@126.com  
 * I'm glad to share my knowledge with you all.  
 * 黑色人类，记得中学学英语，老师说black man是侮辱人的意思，不懂，没跟老外说话  
 */  
public class BlackHuman implements Human {  
  
    public void cry() {  
        System.out.println("黑人会哭");  
    }  
}
```

```

    }

    public void laugh() {
        System.out.println("黑人会笑");
    }

    public void talk() {
        System.out.println("黑人可以说话，一般人听不懂");
    }
}

```

人类也定义完毕了，那我们把八卦炉定义出来：

```

package com.cbf4life;

import java.util.List;
import java.util.Random;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 今天讲女娲造人的故事，这个故事梗概是这样的：
 * 很久很久以前，盘古开辟了天地，用身躯造出日月星辰、山川草木，天地一片繁华
 * One day，女娲下界走了一遭，哎！太寂寞，太孤独了，没个会笑的、会哭的、会说话的东东
 * 那怎么办呢？不用愁，女娲，神仙呀，造出来呀，然后捏泥巴，放八卦炉（后来这个成了太白金星的宝贝）中烤，于是就有了人：
 * 我们把这个生产人的过程用Java程序表现出来：
 */
public class HumanFactory {

    //定一个烤箱，泥巴塞进去，人就出来，这个太先进了
    public static Human createHuman(Class c){
        Human human=null; //定义一个类型的人类

        try {
            human = (Human)Class.forName(c.getName()).newInstance(); //产生一个
            人类

        } catch (InstantiationException e) { //你要是不说个人类颜色的话，没法烤，要白的
            黑，你说话了才好烤

            System.out.println("必须指定人类的颜色");
        } catch (IllegalAccessException e) { //定义的人类有问题，那就烤不出来了，这是...

```

```

        System.out.println("人类定义错误!");
    } catch (ClassNotFoundException e) { //你随便说个人类，我到哪里给你制造去?!

        System.out.println("混蛋，你指定的人类找不到!");
    }
    return human;
}

}

```

然后我们再把女娲声明出来：

```

package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 首先定义女娲，这真是额的神呀
 */
public class NvWa {

    public static void main(String[] args) {

        //女娲第一次造人，试验性质，少造点，火候不足，缺陷产品
        System.out.println("-----造出的第一批人是这样的：白人
        -----");

        Human whiteHuman = HumanFactory.createHuman(WhiteHuman.class);
        whiteHuman.cry();
        whiteHuman.laugh();
        whiteHuman.talk();

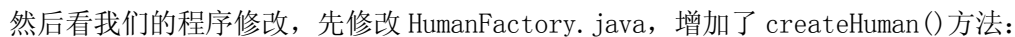
        //女娲第二次造人，火候加足点，然后又出了个次品，黑人
        System.out.println("\n\n-----造出的第二批人是这样的：黑人
        -----");

        Human blackHuman = HumanFactory.createHuman(BlackHuman.class);
        blackHuman.cry();
        blackHuman.laugh();
        blackHuman.talk();

        //第三批人了，这次火候掌握的正好，黄色人类（不写黄人，免得引起歧义），备注：RB人不属
        于此列
        System.out.println("\n\n-----造出的第三批人是这样的：黄色人类
        -----");
    }
}

```

这样这个世界就热闹起来了，人也有了，但是这样创建太累了，神仙也会累的，那怎么办？神仙就想我塞进去一团泥巴，随机出来一群人，管他是黑人、白人、黄人，只要是人就成（你看看，神仙都偷何况是我们人），先修改类图：



第 24 页


```

        try {
            human = (Human)Class.forName(c.getName()).newInstance();    //产生一个
人类

        } catch (InstantiationException e) { //你要是不说个人类颜色的话，没法烤，要白的
黑，你说话了才好烤

            System.out.println("必须指定人类的颜色");
        } catch (IllegalAccessException e) { //定义的人类有问题，那就烤不出来了，这是...

            System.out.println("人类定义错误！");
        } catch (ClassNotFoundException e) { //你随便说个人类，我到哪里给你制造去？！

            System.out.println("混蛋，你指定的人类找不到！");
        }
        return human;
    }

    //女娲生气了，把一团泥巴塞到八卦炉，哎产生啥人类就啥人类
    public static Human createHuman(){
        Human human=null;    //定义一个类型的人类

        //首先是获得有多少个实现类，多少个人类
        List<Class> concreteHumanList =
ClassUtils.getAllClassByInterface(Human.class);    //定义了多少人类
        //八卦炉自己开始想烧出什么人就什么人
        Random random = new Random();
        int rand = random.nextInt(concreteHumanList.size());

        human = createHuman(concreteHumanList.get(rand));

        return human;
    }
}

```

然后看女娲是如何做的：

```

package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 首先定义女娲，这真是额的神呀
 */

```

```

public class NvWa {

    public static void main(String[] args) {

        //女娲第一次造人，试验性质，少造点，火候不足，缺陷产品
        System.out.println("-----造出的第一批人是这样的：白人
-----");
        Human whiteHuman = HumanFactory.createHuman(WhiteHuman.class);
        whiteHuman.cry();
        whiteHuman.laugh();
        whiteHuman.talk();

        //女娲第二次造人，火候加足点，然后又出了个次品，黑人
        System.out.println("\n\n-----造出的第二批人是这样的：黑人
-----");
        Human blackHuman = HumanFactory.createHuman(BlackHuman.class);
        blackHuman.cry();
        blackHuman.laugh();
        blackHuman.talk();

        //第三批人了，这次火候掌握的正好，黄色人类（不写黄人，免得引起歧义），备注：RB人不属
        于此列
        System.out.println("\n\n-----造出的第三批人是这样的：黄色人类
-----");
        Human yellowHuman = HumanFactory.createHuman(YellowHuman.class);
        yellowHuman.cry();
        yellowHuman.laugh();
        yellowHuman.talk();

        //女娲烦躁了，爱是啥人类就是啥人类，烧吧

        for(int i=0;i<10000000000;i++){
            System.out.println("\n\n-----随机产生人类了-----" +
i);

            Human human = HumanFactory.createHuman();
            human.cry();
            human.laugh();
            human.talk();
        }

    }

}

```

哇，这个世界热闹了！，不过还没有完毕，这个程序你跑不起来，还要有这个类：

```
package com.cbf4life;

import java.io.File;
import java.io.IOException;
import java.net.URL;
import java.util.ArrayList;
import java.util.Enumeration;
import java.util.List;

/**
 * @author cbf4Life cbf4life@126.com I'm glad to share my knowledge with you
 *      all.
 *
 */
@SuppressWarnings("all")
public class ClassUtils {

    //给一个接口，返回这个接口的所有实现类
    public static List<Class> getAllClassByInterface(Class c){
        List<Class> returnClassList = new ArrayList<Class>(); //返回结果

        //如果不是一个接口，则不做处理
        if(c.isInterface()){
            String packageName = c.getPackage().getName(); //获得当前的包名
            try {
                List<Class> allClass = getClasses(packageName); //获得当前包下以及子包下的所有类

                //判断是否是同一个接口
                for(int i=0;i<allClass.size();i++){
                    if(c.isAssignableFrom(allClass.get(i))){ //判断是不是一个接口
                        if(!c.equals(allClass.get(i))){ //本身不加进去
                            returnClassList.add(allClass.get(i));
                        }
                    }
                }
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

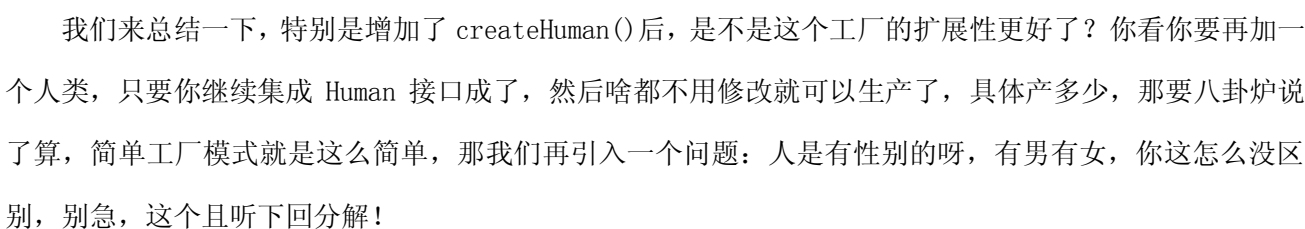
```
    }

    return returnClassList;
}

//从一个包中查找出所有的类，在jar包中不能查找
private static List<Class> getClasses(String packageName)
    throws ClassNotFoundException, IOException {
    ClassLoader classLoader = Thread.currentThread()
        .getContextClassLoader();
    String path = packageName.replace('.', '/');
    Enumeration<URL> resources = classLoader.getResources(path);
    List<File> dirs = new ArrayList<File>();
    while (resources.hasMoreElements()) {
        URL resource = resources.nextElement();
        dirs.add(new File(resource.getFile()));
    }
    ArrayList<Class> classes = new ArrayList<Class>();
    for (File directory : dirs) {
        classes.addAll(findClasses(directory, packageName));
    }
    return classes;
}

private static List<Class> findClasses(File directory, String packageName)
throws ClassNotFoundException {
    List<Class> classes = new ArrayList<Class>();
    if (!directory.exists()) {
        return classes;
    }
    File[] files = directory.listFiles();
    for (File file : files) {
        if (file.isDirectory()) {
            assert !file.getName().contains(".");
            classes.addAll(findClasses(file, packageName + "." +
file.getName()));
        } else if (file.getName().endsWith(".class")) {
            classes.add(Class.forName(packageName + '.' +
file.getName().substring(0, file.getName().length() - 6)));
        }
    }
    return classes;
}
```

告诉你了，这个 ClassUtils 可是个宝，用处可大了去了，可以由一个接口查找到所有的实现类，也可以由父类查找到所有的子类，这个要自己修改一下，动脑筋想下，简单的很！完整的类图如下：



```
package com.cbf4life.advance;
```

```
public class HumanFactory {  
    // 定义一个MAP, 初始化过的Human对象都放在这里
```

```

private static HashMap<String,Human> humans = new HashMap<String,Human>();

//定一个烤箱，泥巴塞进去，人就出来，这个太先进了
public static Human createHuman(Class c){
    Human human=null; //定义一个类型的人类

    try {
        //如果MAP中有，则直接从取出，不用初始化了
        if(humans.containsKey(c.getSimpleName())){
            human = humans.get(c.getSimpleName());
        }else{
            human = (Human)Class.forName(c.getName()).newInstance();
            //放到MAP中
            humans.put(c.getSimpleName(), human);
        }
    } catch (InstantiationException e) { //你要是不说个人类颜色的话，没法烤，要白的
        黑，你说话了才好烤

        System.out.println("必须指定人类的颜色");
    } catch (IllegalAccessException e) { //一定定义的人类有问题，那就烤不出来了，
        这是...

        System.out.println("人类定义错误!");
    } catch (ClassNotFoundException e) { //你随便说个人类，我到哪里给你制造去?!

        System.out.println("混蛋，你指定的人类找不到!");
    }
    return human;
}
}

```

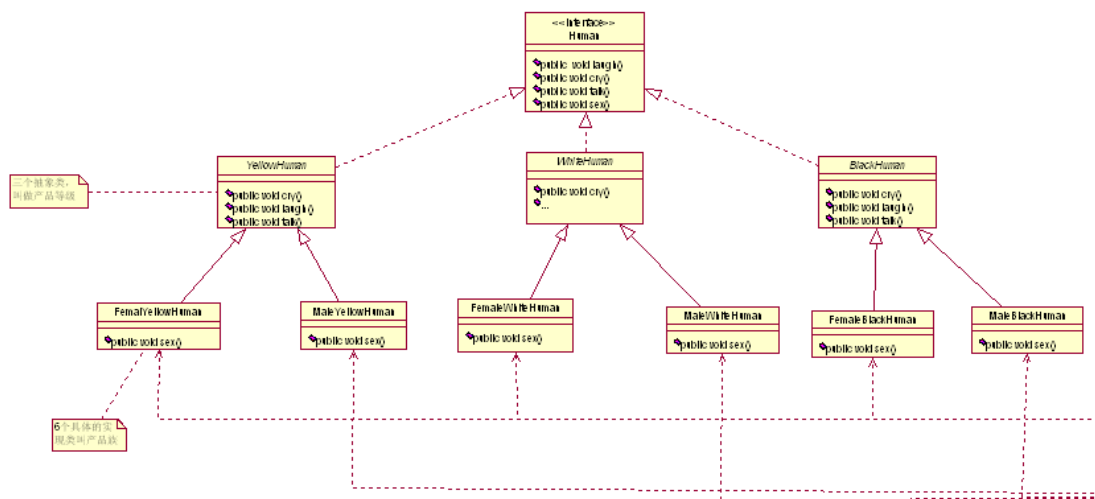
很简单，就加上了黄色那部分的代码，这个在类初始化很消耗资源的情况比较实用，比如你要连接硬件，或者是为了初始化一个类需要准备比较多条件（参数），通过这种方式可以很好的减少项目的复杂程度。

第6章 抽象工厂模式【Abstract Factory Pattern】

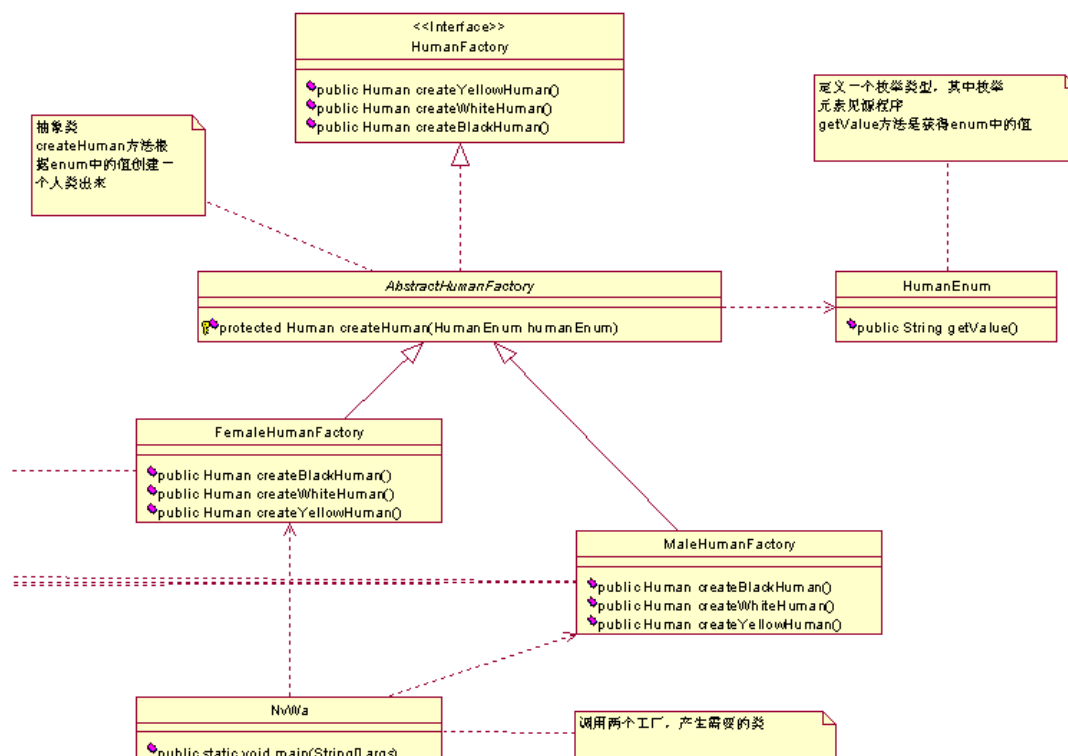
好了，我们继续上一节课，上一节讲到女娲造人，人是造出来了，世界热闹了，可是低头一看，都是清一色的类型，缺少关爱、仇恨、喜怒哀乐等情绪，人类的生命太平淡了，女娲一想，猛然一拍脑袋，Shit！忘记给人类定义性别了，那怎么办？抹掉重来，然后就把人类重新洗牌，准备重新开始制造人类。

由于先前的工作已经花费了很大的精力做为铺垫，也不想从头开始了，那先说人类（Product 产品类）怎么改吧，好，有了，给每个人类都加一个性别，然后再重新制造，这个问题解决了，那八卦炉怎么办？只有一个呀，要么生产出全都是男性，要不都是女性，那不行呀，有了，把已经有了一条生产线——八卦炉（工厂模式中的 Concrete Factory）拆开，于是女娲就使用了“八卦拷贝术”，把原先的八卦炉一个变两个，并且略加修改，就成了女性八卦炉（只生产女性，一个具体工厂的实现类）和男性八卦炉（只生产男性，又一个具体工厂的实现类），这个过程类图如下：

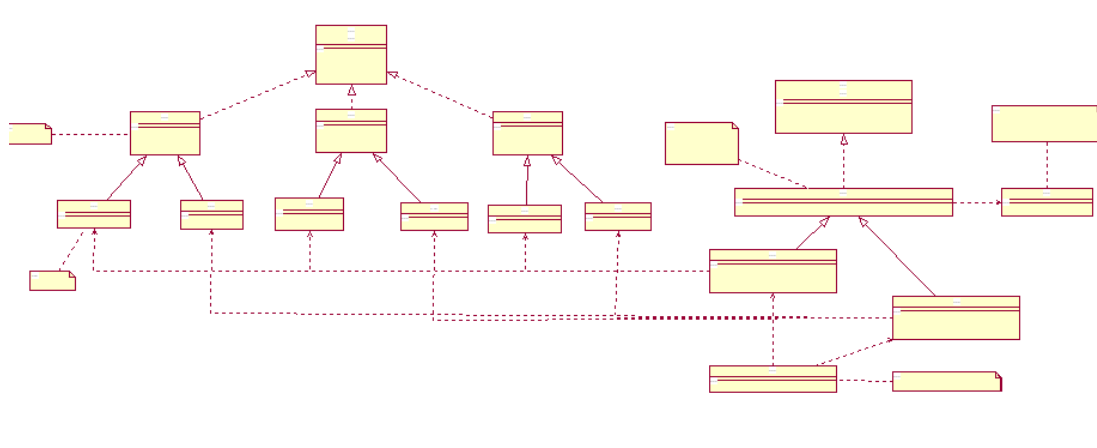
先看人类（也就是产品）的类图：



这个类图也比较简单，Java 的典型类图，一个接口，几个抽象类，然后是几个实现类，没啥多说的，其中三个抽象类在抽象工厂模式中是叫做产品等级，六个实现类是叫做产品族，这个也比较好理解，实现类嘛是真实的产品，一个叫产品，多了就叫产品族，然后再看工厂类：



其中抽象工厂只实现了一个 createHuman 的方法，目的是简化实现类的代码工作量，这个在讲代码的时候会说。这里还使用了 Jdk 1.5 的一个新特性 Enum 类型，其实这个完全可以类的静态变量来实现，但我想既然是学习就应该学有所获得，即使你对这个模式非常了解，也可能没用过 Enum 类型，也算是一个不同的知识点吧，我希望给大家讲解，每次都有新的技术点提出来，每个人都会有一点点的收获就足够了，然后在具体的项目中使用，知道有这个技术点，然后上 baidu 狗狗一下就能解决问题。话题扯远了，我们继续类图，完整的类图如下，这个看不大清楚，其实就是上面那两个类图加起来，大家可以看源码中的那个类图文件：



然后类图讲解完毕，我们来看程序实现：


```

package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 定义一个人类的统称，问题出来了，刚刚定义的时候忘记定义性别了
 * 这个重要的问题非修改不可，否则这个世界上太多太多的东西不存在了
 */
public interface Human {
    //首先定义什么是人类

    //人是愉快的，会笑的，本来是想用smile表示，想了一下laugh更合适，好长时间没有大笑了；
    public void laugh();

    //人类还会哭，代表痛苦
    public void cry();

    //人类会说话
    public void talk();

    //定义性别
    public void sex();
}

```

人类的接口定义好，然后根据接口创建三个抽象类，也就是三个产品等级，实现 laugh()、cry()、talk() 三个方法，以 AbstractYellowHuman 为例：

```

package com.cbf4life.yellowHuman;

import com.cbf4life.Human;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 为什么要修改成抽象类呢？要定义性别呀
 */
public abstract class AbstractYellowHuman implements Human {

    public void cry() {
        System.out.println("黄色人类会哭");
    }
}

```

```
public void laugh() {
    System.out.println("黄色人类会大笑，幸福呀！");
}

public void talk() {
    System.out.println("黄色人类会说话，一般说的都是双字节");
}

}
```

其他的两个抽象类 AbstractWhiteHuman 和 AbstractBlackHuman 与此类似的事项方法，不再通篇拷贝代码，大家可以看一下源代码。算了，还是拷贝，反正是电子档的，不想看就往下翻页，也成就了部分“懒人”，不用启动 Eclipse，还要把源码拷贝进来：

白种人的抽象类：

```
package com.cbf4life.whiteHuman;

import com.cbf4life.Human;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 白色人人类
 * 为了代码整洁，新建一个包，这里是白种人的天下了
 */
public abstract class AbstractWhiteHuman implements Human {

    public void cry() {
        System.out.println("白色人类会哭");
    }

    public void laugh() {
        System.out.println("白色人类会大笑，侵略的笑声");
    }

}
```

```
    public void talk() {
        System.out.println("白色人类会说话，一般都是但是单字节!");
    }
}
```

黑种人的抽象类:

```
package com.cbf4life.blackHuman;

import com.cbf4life.Human;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 黑色人类，记得中学学英语，老师说black man是侮辱人的意思，不懂，没跟老外说话
 */
public abstract class AbstractBlackHuman implements Human {

    public void cry() {
        System.out.println("黑人会哭");
    }

    public void laugh() {
        System.out.println("黑人会笑");
    }

    public void talk() {
        System.out.println("黑人可以说话，一般人听不懂");
    }

}
```

三个抽象类都实现完毕了，然后就是些实现类了。其实，你说抽象类放这里有什么意义吗？就是不允许你 new 出来一个抽象的对象呗，使用非抽象类完全就可以代替，呵呵，杀猪杀尾巴，各有各的杀法，不过既然进了 Java 这个门就要遵守 Java 这个规矩，我们看实现类:

女性黄种人的实现类:

```
package com.cbf4life.yellowHuman;
```

```
/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 女性黄种人
 */
public class YellowFemaleHuman extends AbstractYellowHuman {

    public void sex() {
        System.out.println("该黄种人的性别为女...");
    }

}
```

男性黄种人的实现类:

```
package com.cbf4life.yellowHuman;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 男性黄种人
 */
public class YellowMaleHuman extends AbstractYellowHuman {

    public void sex() {
        System.out.println("该黄种人的性别为男....");
    }

}
```

女性白种人的实现类:

```
package com.cbf4life.whiteHuman;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.\
 * 女性白种人
 */
public class WhiteFemaleHuman extends AbstractWhiteHuman {

    public void sex() {
```

```
        System.out.println("该白种人的性别为女....");
    }

}
```

男性白种人的实现类:

```
package com.cbf4life.whiteHuman;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 男性白种人
 */
public class WhiteMaleHuman extends AbstractWhiteHuman {

    public void sex() {
        System.out.println("该白种人的性别为男....");
    }

}
```

女性黑种人的实现类:

```
package com.cbf4life.blackHuman;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 女性黑种人
 */
public class BlackFemaleHuman extends AbstractBlackHuman {

    public void sex() {
        System.out.println("该黑种人的性别为女...");
    }

}
```

男性黑种人的实现类:

```
package com.cbf4life.blackHuman;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 男性黑种人
 */
public class BlackMaleHuman extends AbstractBlackHuman {

    public void sex() {
        System.out.println("该黑种人的性别为男...");
    }

}
```

抽象工厂模式下的产品等级和产品族都已经完成，也就是人类以及产生出的人类是什么样子的都已经定义好了，下一步就等着工厂开工创建了，那我们来看工厂类。

在看工厂类之前我们先看那个枚举类型，这个是很有意思的：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 世界上有哪些类型的人，列出来
 * JDK 1.5开始引入enum类型也是目的，吸引C程序员转过来
 */
public enum HumanEnum {

    //把世界上所有人类类型都定义出来
    YelloMaleHuman("com.cbf4life.yellowHuman.YellowMaleHuman"),

    YelloFemaleHuman("com.cbf4life.yellowHuman.YellowFemaleHuman"),

    WhiteFemaleHuman("com.cbf4life.whiteHuman.WhiteFemaleHuman"),

    WhiteMaleHuman("com.cbf4life.whiteHuman.WhiteMaleHuman"),

    BlackFemaleHuman("com.cbf4life.blackHuman.BlackFemaleHuman"),

    BlackMaleHuman("com.cbf4life.blackHuman.BlackMaleHuman");

}
```

```

private String value = "";
//定义构造函数，目的是Data(value)类型的相匹配
private HumanEnum(String value){
    this.value = value;
}

public String getValue(){
    return this.value;
}

/*
 * java enum类型尽量简单使用，尽量不要使用多态、继承等方法
 * 毕竟用Class完全可以代替enum
 */
}

```

我之所以引入 Enum 这个类型，是想让大家在看这本书的时候能够随时随地的学到点什么，你如果看不懂设计模式，你可以从我的程序中学到一些新的技术点，不用像我以前报着砖头似的书在那里啃，看一遍不懂，再看第二遍，然后翻了英文原本才知道，哦~，原来是这样滴，只能说有些翻译家实在不懂技术。我在讲解技术的时候，尽量少用专业术语，尽量使用大部分人类都能理解的语言。

Enum 以前我也很少用，最近在一个项目中偶然使用上了，然后才发觉它的好处，Enum 类型作为一个参数传递到一个方法中时，在 Junit 进行单元测试的时候，不用判断输入参数是否为空、长度为 0 的边界异常条件，如果方法传入的参数不是 Enum 类型的话，根本就传递不进来，你说定义一个类，定义一堆的静态变量，这也可以呀，这个不和你抬杠，上面的代码我解释一下，构造函数没啥好说的，然后是 getValue() 方法，就是获得枚举类型中一个元素的值，枚举类型中的元素也是有名称和值的，这个和 HashMap 有点类似。

然后，我们看我们的工厂类，先看接口：

```

package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 这次定一个接口，应该要造不同性别的人，需要不同的生产线
 * 那这个八卦炉必须可以制造男人和女人
 */

```

```

*/
public interface HumanFactory {

    //制造黄色人类
    public Human createYellowHuman();

    //制造一个白色人类
    public Human createWhiteHuman();

    //制造一个黑色人类
    public Human createBlackHuman();
}

```

然后看抽象类:

```

package com.cbf4life.humanFactory;

import com.cbf4life.Human;
import com.cbf4life.HumanEnum;
import com.cbf4life.HumanFactory;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 编写一个抽象类，根据enum创建一个人类出来
 */
public abstract class AbstractHumanFactory implements HumanFactory {

    /*
     * 给定一个性别人类，创建一个人类出来 专业术语是产生产品等级
     */
    protected Human createHuman(HumanEnum humanEnum) {
        Human human = null;

        //如果传递进来不是一个Enum中具体的一个Element的话，则不处理
        if (!humanEnum.getValue().equals("")) {
            try {
                //直接产生一个实例
                human = (Human)
Class.forName(humanEnum.getValue()).newInstance();
            } catch (Exception e) {
                //因为使用了enum，这个种异常情况不会产生了，除非你的enum有问题；
                e.printStackTrace();
            }
        }
    }
}

```



```

    }
    return human;
}

}

```

看到没，这就是引入 enum 的好处，createHuman(HumanEnum humanEnum)这个方法定义了输入参数必须是 HumanEnum 类型，然后直接使用 humanEnum.getValue() 方法就能获得具体传递进来的值，这个不多说了，大家自己看程序领会，没多大难度，这个抽象类的目的就是减少下边实现类的代码量，我们看实现类：

男性工厂，只创建男性：

```

package com.cbf4life.humanFactory;

import com.cbf4life.Human;
import com.cbf4life.HumanEnum;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 男性创建工厂
 */
public class MaleHumanFactory extends AbstractHumanFactory {

    //创建一个男性黑种人
    public Human createBlackHuman() {
        return super.createHuman(HumanEnum.BlackMaleHuman);
    }

    //创建一个男性白种人
    public Human createWhiteHuman() {
        return super.createHuman(HumanEnum.WhiteMaleHuman);
    }

    //创建一个男性黄种人
    public Human createYellowHuman() {
        return super.createHuman(HumanEnum.YelloMaleHuman);
    }

}

```

女性工厂，只创建女性：

```

package com.cbf4life.humanFactory;

import com.cbf4life.Human;
import com.cbf4life.HumanEnum;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.\
 * 女性创建工厂
 */
public class FemaleHumanFactory extends AbstractHumanFactory {

    //创建一个女性黑种人
    public Human createBlackHuman() {
        return super.createHuman(HumanEnum.BlackFemaleHuman);
    }

    //创建一个女性白种人
    public Human createWhiteHuman() {
        return super.createHuman(HumanEnum.WhiteFemaleHuman);
    }

    //创建一个女性黄种人
    public Human createYellowHuman() {
        return super.createHuman(HumanEnum.YelloFemaleHuman);
    }
}

```

产品定义好了，工厂也定义好了，万事俱备只欠东风，那咱就开始造吧，哦，不对，女娲开始造人了：

```

package com.cbf4life;

import com.cbf4life.humanFactory.FemaleHumanFactory;
import com.cbf4life.humanFactory.MaleHumanFactory;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 女娲建立起了两条生产线，分别是：
 * 男性生产线
 * 女性生产线
 */

```

```
public class NvWa {

    public static void main(String[] args) {

        //第一条生产线，男性生产线
        HumanFactory maleHumanFactory = new MaleHumanFactory();

        //第二条生产线，女性生产线
        HumanFactory femaleHumanFactory = new FemaleHumanFactory();

        //生产线建立完毕，开始生产人了：
        Human maleYellowHuman = maleHumanFactory.createYellowHuman();

        Human femaleYellowHuman = femaleHumanFactory.createYellowHuman();

        maleYellowHuman.cry();
        maleYellowHuman.laugh();
        femaleYellowHuman.sex();
        /*
         * .....
         * 后面你可以续了
         */
    }
}
```

两个八卦炉，一个造女的，一个造男的，开足马力，一直造到这个世界到现在这个模式为止。

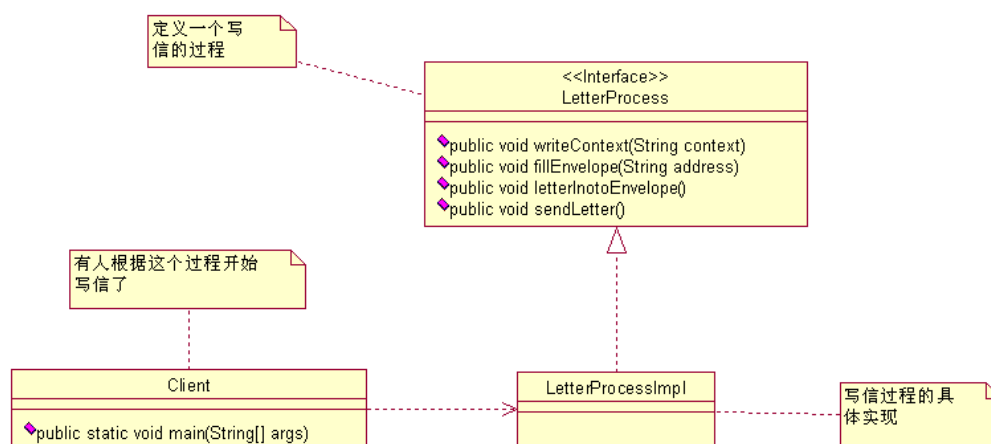
抽象工厂模式讲完了，那我们再思考一些问题：工厂模式有哪些优缺点？先说优点，我这人一般先看人优点，非常重要的有点就是，工厂模式符合 OCP 原则，也就是开闭原则，怎么说呢，比如就性别的问题，这个世界上还存在双性人，是男也是女的人，那这个就是要在我们的产品族中增加一类产品，同时再增加一个工厂就可以解决这个问题，不需要我再来实现了吧，很简单的大家自己画下类图，然后实现下。

那还有没有其他好处呢？抽象工厂模式，还有一个非常大的有点，高内聚，低耦合，在一个较大的项目组，产品是由一批人定义开发的，但是提供其他成员访问的时候，只有工厂方法和产品的接口，也就是说只需要提供 Product Interface 和 Concrete Factory 就可以产生自己需要的对象和方法，Java 的高内聚低耦合的特性表现的一览无遗，哈哈。

第 7 章 门面模式【Facade Pattern】

好，我们继续讲课。大家都是高智商的人，都写过纸质的信件吧，比如给女朋友写情书什么的，写信的过程大家都还记得吧，先写信的内容，然后写信封，然后把信放到信封中，封好，投递到信箱中进行邮递，这个过程还是比较简单的，虽然简单，这四个步骤都是要跑的呀，信多了还是麻烦，比如到了情人节，为了大海捞针，给十个女孩子发情书，都要这样跑一遍，你不要累死，更别说你要发个广告信啥的，一下子发 1 千万封邮件，那不就完蛋了？那怎么办呢？还好，现在邮局开发了一个新业务，你只要把信件的必要信息高速我，我给你发，我来做这四个过程，你就不要管了，只要把信件交给我就成了。

我们的类图还是从最原始的状态开始：



在这中环境下，最累的是写信的人，为了发送一封信出去要有四个步骤，而且这四个步骤还不能颠倒，你不可能没写信就把信放到信封吧，写信的人要知道这四个步骤，而且还要知道这四个步骤的顺序，恐怖吧，我们先看看这个过程如何表现出来的：

先看写信的过程接口，定义了写信的四个步骤：

```
package com.cbf4life.facade;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 定义一个写信的过程
 */
public interface LetterProcess {
```

```
//首先要写信的内容
public void writeContext(String context);

//其次写信封
public void fillEnvelope(String address);

//把信放到信封里
public void letterInotoEnvelope();

//然后邮递
public void sendLetter();

}
```

写信过程的具体实现:

```
package com.cbf4life.facade;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 写信的具体实现了
 */
public class LetterProcessImpl implements LetterProcess {

    //写信
    public void writeContext(String context) {
        System.out.println("填写信的内容...." + context);
    }

    //在信封上填写必要的信息
    public void fillEnvelope(String address) {
        System.out.println("填写收件人地址及姓名...." + address);
    }

    //把信放到信封中，并封好
    public void letterInotoEnvelope() {
        System.out.println("把信放到信封中....");
    }

    //塞到邮箱中，邮递
    public void sendLetter() {
        System.out.println("邮递信件...");
    }
}
```

```
}
```

然后就有人开始用这个过程写信了：

```
package com.cbf4life.facade;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 我开始给朋友写信了
 */
public class Client {

    public static void main(String[] args) {

        //创建一个处理信件的过程
        LetterProcess letterProcess = new LetterProcessImpl();

        //开始写信
        letterProcess.writeContext("Hello,It's me,do you know who I am? I'm your
old lover. I'd like to....");

        //开始写信封
        letterProcess.fillEnvelope("Happy Road No. 666,God Province,Heaven");

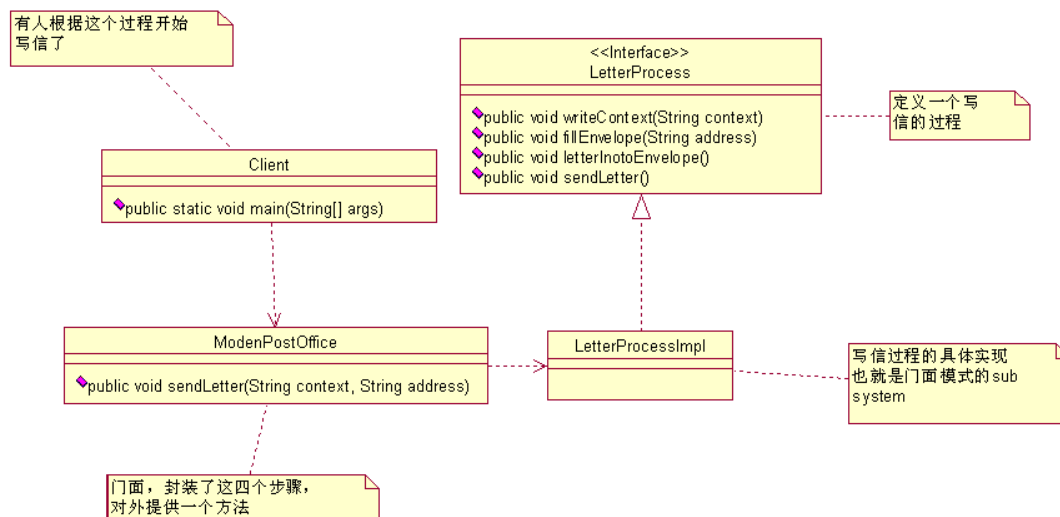
        //把信放到信封里，并封装好
        letterProcess.letterInotoEnvelope();

        //跑到邮局把信塞到邮箱，投递
        letterProcess.sendLetter();

    }

}
```

那这个过程与高内聚的要求相差甚远，你想，你要知道这四个步骤，而且还要知道这四个步骤的顺序，一旦出错，信就不可能邮寄出去，那我们如何来改进呢？先看类图：



这就是门面模式，还是比较简单的，Sub System 比较复杂，为了让调用者更方便的调用，就对 Sub System 进行了封装，增加了一个门面，Client 调用时，直接调用门面的方法就可以了，不用了解具体的实现方法以及相关的业务顺序，我们来看程序的改变，LetterProcess 接口和实现类都没有改变，只是增加了一个 ModenPostOffice 类，我们这个 java 程序清单如下：

```

package com.cbf4life.facade;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 */
public class ModenPostOffice {
    private LetterProcess letterProcess = new LetterProcessImpl();

    //写信，封装，投递，一体化了
    public void sendLetter(String context,String address){

        //帮你写信
        letterProcess.writeContext(context);

        //写好信封
        letterProcess.fillEnvelope(address);

        //把信放到信封中
        letterProcess.letterInotoEnvelope();

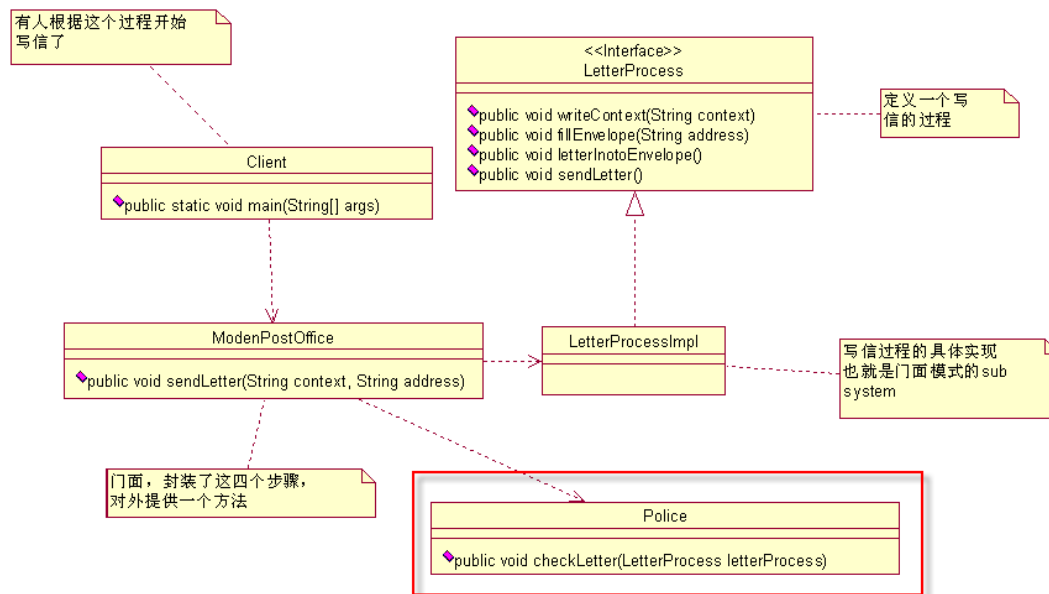
        //邮递信件
    }
}
  
```

```
        letterProcess.sendLetter();  
    }  
}
```

这个类是什么意思呢，就是说现在又一个叫 Hell Road PostOffice（地狱路邮局）提供了一种新型的服务，客户只要把信的内容以及收信地址给他们，他们就会把信写好，封好，并发送出去，这种服务提出时大受欢迎呀，这简单呀，客户减少了很多工作，那我们看看客户是怎么调用的，Client.java 的程序清单如下：

```
package com.cbf4life.facade;  
  
/**  
 * @author cbf4Life cbf4life@126.com  
 * I'm glad to share my knowledge with you all.  
 * 我开始给朋友写信了  
 */  
public class Client {  
  
    public static void main(String[] args) {  
        //现代化的邮局，有这项服务，邮局名称叫Hell Road  
        ModenPostOffice hellRoadPostOffice = new ModenPostOffice();  
  
        //你只要把信的内容和收信人地址给他，他会帮你完成一系列的工作；  
        String address = "Happy Road No. 666,God Province,Heaven"; //定义一个地址  
        String context = "Hello,It's me,do you know who I am? I'm your old lover.  
I'd like to....";  
        hellRoadPostOffice.sendLetter(context, address);  
    }  
}
```

看到没，客户简单了很多，提供这种模式后，系统的扩展性也有了很大的提高，突然一个非常时期，寄往 God Province（上帝省）的邮件都必须进行安全检查，那我们这个就很好处理了，看类图：



看这个红色的框，只增加了这一部分，其他部分在类图上都不需要改动，那我们来看源码：

```

package com.cbf4life.facade;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 */
public class ModenPostOffice {
    private LetterProcess letterProcess = new LetterProcessImpl();
    private Police letterPolice = new Police();

    //写信，封装，投递，一体化了
    public void sendLetter(String context,String address){

        //帮你写信
        letterProcess.writeContext(context);

        //写好信封
        letterProcess.fillEnvelope(address);

        //警察要检查信件了
        letterPolice.checkLetter(letterProcess);

        //把信放到信封中
        letterProcess.letterInotoEnvelope();

        //邮递信件
    }
}

```

```
        letterProcess.sendLetter();  
  
    }  
}
```

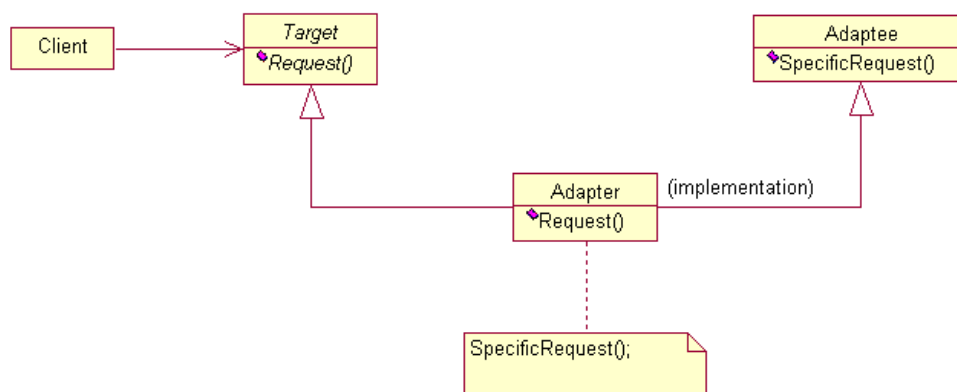
只是增加了一个 letterPolice 变量的声明以及一个方法的调用，那这个写信的过程就变成了这样：先写信，然后写信封，然后警察开始检查，然后才把信放到信封，然后发送出去，那这个变更对客户来说，是透明的，他根本就看不到有人在检查他的邮件，他也不用了解，反正现代化的邮件都帮他做了，这也是他乐意的地方。

门面模式讲解完毕，这是一个很好的封装方法，一个子系统比较复杂的实话，比如算法或者业务比较复杂，就可以封装出一个或多个门面出来，项目的结构简单，而且扩展性非常好。还有，在一个较大项目中的时候，为了避免人员带来的风险，也可以使用这个模式，技术水平比较差的成员，尽量安排独立的模块（Sub System），然后把他写的程序封装到一个门面里，尽量让其他项目成员不用看到这些烂人的代码，看也看不懂，我也遇到过一个“高人”写的代码，private 方法、构造函数、常量基本都不用，你要一个 public 方法，好，一个类里就一个 public 方法，所有代码都在里面，然后你就看吧，一大坨的程序，看着能把人逼疯，使用门面模式后，对门面进行单元测试，约束项目成员的代码质量，对项目整体质量的提升也是一个比较好的帮助。

第 8 章 适配器模式【Adapter Pattern】

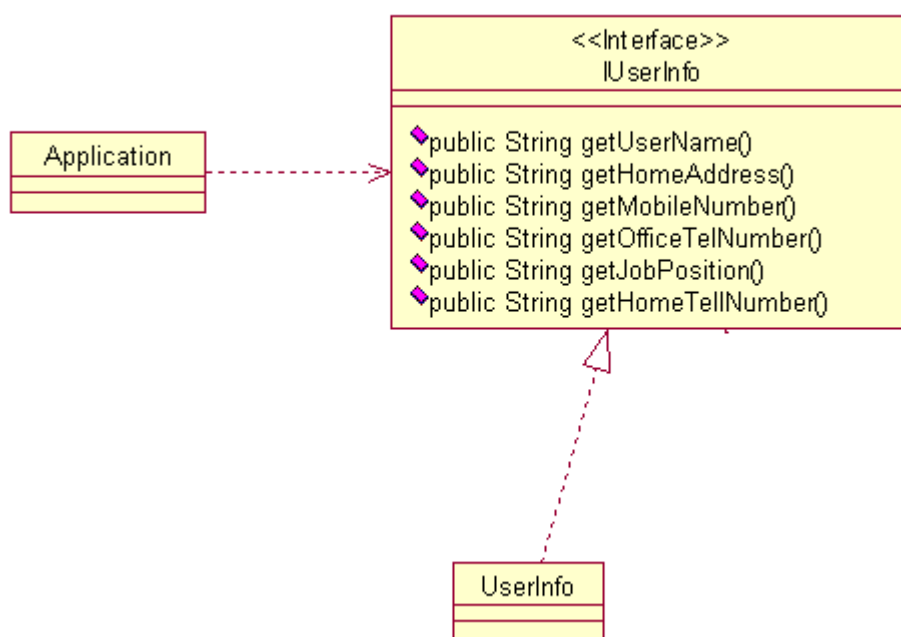
好，请安静，后排聊天的同学别吵醒前排睡觉的同学了，大家要相互理解嘛。今天讲适配器模式，这个模式也很简单，你笔记本上的那个拖在外面的黑盒子就是个适配器，一般你在中国能用，在日本也能用，虽然两个国家的电源电压不同，中国是 220V，日本是 110V，但是这个适配器能够把这些不同的电压转换为你需要的 36V 电压，保证你的笔记本能够正常运行，那我们在设计模式中引入这个适配器模式是不是也是这个意思呢？是的，一样的作用，两个不同接口，有不同的实现，但是某一天突然上帝命令你把 B 接口转换为 A 接口，怎么办？继承，能解决，但是比较傻，而且还违背了 OCP 原则，怎么办？好在我们还有适配器模式。

适配器的通用类图是这个样子滴：



首先声明，这个不是我画的，这是从 Rational Rose 的帮助文件中截取的，这个类图也很容易理解，Target 是一个类（接口），Adaptee 是一个接口，通过 Adapter 把 Adaptee 包装成 Target 的一个子类（实现类），注意了这里用了个名词包装（Wrapper），那其实这个模式也叫做包装模式（Wrapper），但是包装模式可不知一个，还包括了以后要讲的装饰模式。我来讲个自己的一个经验，让大家可以更容易了解这个适配器模式，否则纯讲技术太枯燥了，技术也可以有娱乐的嘛。

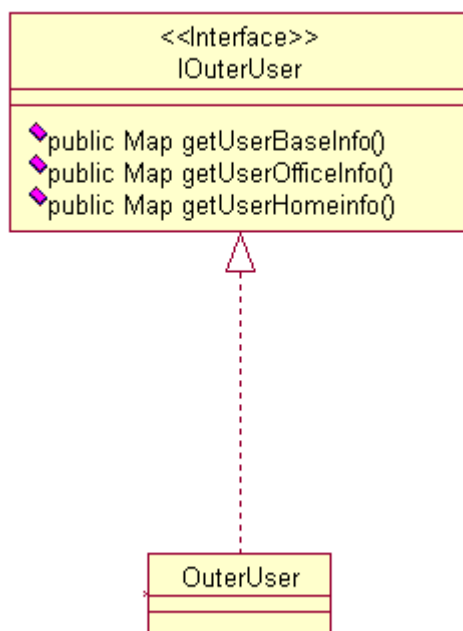
我在 2004 年的时候带了一个项目，做一个人力资源管理，该项目是我们总公司发起的项目，公司一共有 700 多号人，包括子公司，这个项目还是比较简单的，分为三大模块：人员信息管理，薪酬管理，职位管理，其中人员管理这块就用到了适配器模式，是怎么回事呢？当时开发时明确的指明：人员信息管理的对象是所有员工的所有信息，然后我们就这样设计了一个类图：



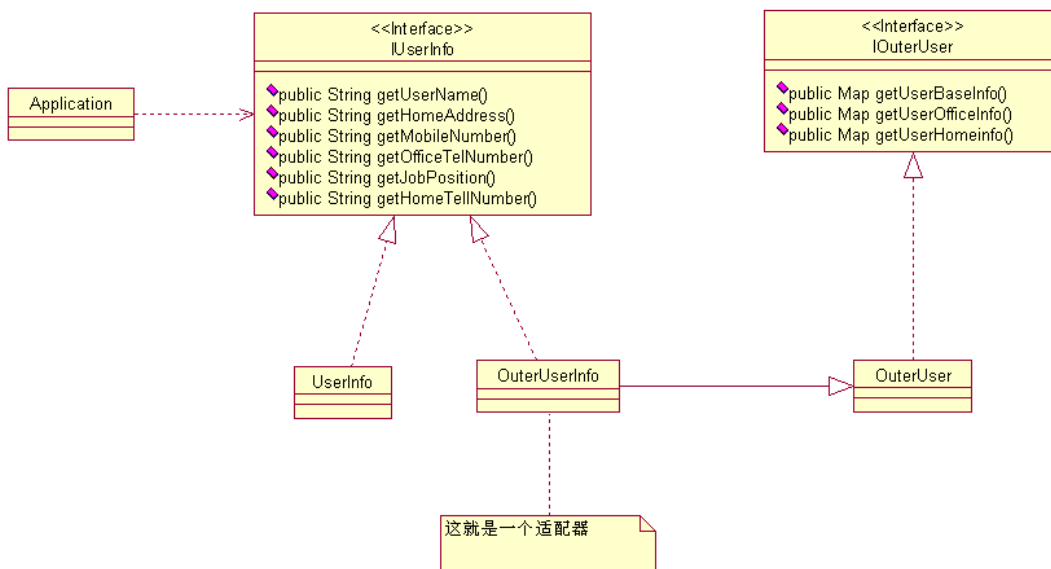
还是比较简单的，有一个对象 `UserInfo` 存储用户的所有信息（实际系统上还有很多子类，不多说了），也就是 `BO`（Business Object），这个对象设计为贫血对象（Thin Business Object），不需要存储状态以及相关的关系，而且我也是反对使用充血对象（Rich Business Object），这里说了两个名词贫血对象和充血对象，这两个名词很简单，在领域模型中分别叫做贫血领域模型和充血领域模型，有什么区别呢？在一个对象中不存储实体状态以及对象之间的关系的就叫做贫血对象，上升到领域模型中就是贫血领域模型，有实体状态和对象关系的模型的就是充血领域模型，是不是太技术化了？这个看不懂没关系，都是糊弄人的东西，属于专用名词，这本书写完了，我再折腾本领域模型的文章，揭露领域模型中糊弄人的专用名词，这个绝对是专用名词的堆砌，呵呵。扯远了，我们继续说适配器模式，这个 `UserInfo` 对象，在系统中很多地方使用，你可以查看自己的信息，也可以做修改，当然这个对象是有 `setter` 方法的，我们这里用不到就隐藏掉了。

这个项目是 04 年年底投产的，运行到 05 年年底还是比较平稳的，中间修修补补也很正常，05 年年底不知道是那股风吹的，很多公司开始使用借聘人员的方式招聘人员，我们公司也不例外，从一个人力资源公司借用了一大批的低技术、低工资的人员，分配到各个子公司，总共有将近 200 号人，然后就找我们部门老大谈判，说要增加一个功能借用人员管理，老大一看有钱赚呀，一拍大腿，做！

我带人过去一调研，不是这么简单，人力资源公司有一套自己的人员管理系统，我们公司需要把我们使用到的人员信息传输到我们的系统中，系统之间的传输使用 `RMI`（Remote Method Invocation，远程对象调用）的方式，但是有一个问题人力资源公司的人员对象和我们系统的对象不相同呀，他们的对象是这样的：



人员资源公司是把人的信息分为了三部分：基本信息，办公信息和个人家庭信息，并且都放到了 `HashMap` 中，比如人员的姓名放到 `BaseInfo` 信息中，家庭地址放到 `HomeInfo` 中，这咱不好说他们系统设计的不好，那问题是咱的系统要和他们系统有交互，怎么办？使用适配器模式，类图如下：



大家可能会问，这两个对象都不在一个系统中，你如何使用呢？简单！RMI 已经帮我们做了这件事情，只要有接口，就可以把远程的对象当成本地的对象使用，这个大家有时间可以去看一下 RMI 文档，不多说了。通过适配器，把 `OuterUser` 伪装成我们系统中一个 `IUserInfo` 对象，这样，我们的系统基本不用修改什么程序员，所有的人员查询、调用跟本地一样样的，说的口干舌燥，那下边我们来看具体的代码实现：

首先看 IUserInfo.java 的代码:

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 用户信息对象
 */
public interface IUserInfo {

    //获得用户姓名
    public String getUsername();

    //获得家庭地址
    public String getHomeAddress();

    //手机号码, 这个太重要, 手机泛滥呀
    public String getMobileNumber();

    //办公电话, 一般式座机
    public String getOfficeTelNumber();

    //这个人的职位是啥
    public String getJobPosition();

    //获得家庭电话, 这个有点缺德, 我是不喜欢打家庭电话讨论工作
    public String getHomeTelNumber();
}
```

然后看这个接口的实现类:

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 */
public class UserInfo implements IUserInfo {

    /*
     * 获得家庭地址, 下属送礼也可以找到地方
     */
}
```

```
*/
public String getHomeAddress() {
    System.out.println("这里是员工的家庭地址....");
    return null;
}

/*
 * 获得家庭电话号码
 */
public String getHomeTelNumber() {
    System.out.println("员工的家庭电话是....");
    return null;
}

/*
 * 员工的职位，是部门经理还是小兵
 */
public String getJobPosition() {
    System.out.println("这个人的职位是BOSS....");
    return null;
}

/*
 * 手机号码
 */
public String getMobileNumber() {
    System.out.println("这个人的手机号码是0000....");
    return null;
}

/*
 * 办公室电话，烦躁的时候最好“不小心”把电话线踢掉，我经常这么干，对己对人都有好处
 */
public String getOfficeTelNumber() {
    System.out.println("办公室电话是....");
    return null;
}

/*
 * 姓名了，这个老重要了
 */
public String getUserName() {
    System.out.println("姓名叫做...");
    return null;
}
```

```

    }

}

```

可能有人要问了，为什么要把电话号码、手机号码都设置成 String 类型，而不是 int 类型，大家觉的呢？题外话，这个绝对应该是 String 类型，包括数据库也应该是 varchar 类型的，手机号码有小灵通带区号的，比如 02100001，这个你用数字怎么表示？有些人要在手机号码前加上 0086 后再保存，比如我们公司的印度阿三就是这样，喜欢在手机号码前 0086 保存下来，呵呵，我是想到啥就说啥，啰嗦了点。继续看我们的代码，下面看我们系统的应用如何调用 UserInfo 的信息：

```

package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 这就是我们具体的应用了，比如老板要查所有的20-30的女性信息
 */
public class App {

    public static void main(String[] args) {
        //没有与外系统连接的时候，是这样写的
        IUserInfo youngGirl = new UserInfo();
        //从数据库中查到101个
        for(int i=0;i<101;i++){
            youngGirl.getMobileNumber();
        }

    }

}

```

这老板，比较那个，为什么是 101，是男生都应该知道吧，111 代表男生，101 代表女生，呵呵，是不是比较色呀。从数据库中生成了 101 个 UserInfo 对象，直接打印出来就成了。那然后增加了外系统的人员信息，怎么处理呢？下面是 IOuterUser.java 的源代码：

```

package com.cbf4life;

import java.util.Map;

```



```
/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 外系统的人员信息
 */
@SuppressWarnings("all")
public interface IOuterUser {

    //基本信息，比如名称，性别，手机号码了等
    public Map getUserBaseInfo();

    //工作区域信息
    public Map getUserOfficeInfo();

    //用户的家庭信息
    public Map getUserHomeInfo();

}
```

我们再看看外系统的用户信息的具体实现类：

```
package com.cbf4life;

import java.util.HashMap;
import java.util.Map;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 外系统的用户信息的实现类
 */
@SuppressWarnings("all")
public class OuterUser implements IOuterUser {

    /*
     * 用户的基本信息
     */
    public Map getUserBaseInfo() {
        HashMap baseInfoMap = new HashMap();

        baseInfoMap.put("userName", "这个员工叫混世魔王....");
        baseInfoMap.put("mobileNumber", "这个员工电话是....");
    }
}
```

```
        return baseInfoMap;
    }

    /**
     * 员工的家庭信息
     */
    public Map getUserHomeInfo() {
        HashMap homeInfo = new HashMap();

        homeInfo.put("homeTelNumbner", "员工的家庭电话是...");
        homeInfo.put("homeAddress", "员工的家庭地址是...");

        return homeInfo;
    }

    /**
     * 员工的工作信息，比如职位了等
     */
    public Map getUserOfficeInfo() {
        HashMap officeInfo = new HashMap();

        officeInfo.put("jobPosition", "这个人的职位是BOSS...");
        officeInfo.put("officeTelNumber", "员工的办公电话是...");

        return officeInfo;
    }
}
```

那怎么把外系统的用户信息包装成我们公司的人员信息呢？看下面的 OuterUserInfo 类源码，也就是我们的适配器：

```
package com.cbf4life;

import java.util.Map;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 把OuterUser包装成UserInfo
 */
```

```
@SuppressWarnings("all")
public class OuterUserInfo extends OuterUser implements IUserInfo {

    private Map baseInfo = super.getUserBaseInfo(); //员工的基本信息
    private Map homeInfo = super.getUserHomeInfo(); //员工的家庭 信息
    private Map officeInfo = super.getUserOfficeInfo(); //工作信息

    /*
     * 家庭地址
     */
    public String getHomeAddress() {
        String homeAddress = (String)this.homeInfo.get("homeAddress");
        System.out.println(homeAddress);
        return homeAddress;
    }

    /*
     * 家庭电话号码
     */
    public String getHomeTelNumber() {
        String homeTelNumber = (String)this.homeInfo.get("homeTelNumber");
        System.out.println(homeTelNumber);
        return homeTelNumber;
    }

    /*
     * 职位信息
     */
    public String getJobPosition() {
        String jobPosition = (String)this.officeInfo.get("jobPosition");
        System.out.println(jobPosition);
        return jobPosition;
    }

    /*
     * 手机号码
     */
    public String getMobileNumber() {
        String mobileNumber = (String)this.baseInfo.get("mobileNumber");
        System.out.println(mobileNumber);
        return mobileNumber;
    }

    /*
```

```
    * 办公电话
    */
    public String getOfficeTelNumber() {
        String officeTelNumber = (String)this.officeInfo.get("officeTelNumber");
        System.out.println(officeTelNumber);
        return officeTelNumber;
    }

    /*
    * 员工的名称
    */
    public String getUserName() {
        String userName = (String)this.baseInfo.get("userName");
        System.out.println(userName);
        return userName;
    }
}
```

大家看到没？这里有很多的强制类型转换，就是(String)这个东西，如果使用泛型的话，完全就可以避免这个转化，这节课啰嗦的太多就不再讲了，下次找个时间再讲。这个适配器的作用就是做接口的转换，那然后我们再来看看我们的业务是怎么调用的：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 这就是我们具体的应用了，比如老板要查所有的20-30的女性信息
 */
public class App {

    public static void main(String[] args) {
        //没有与外系统连接的时候，是这样写的
        //IUserInfo youngGirl = new UserInfo();

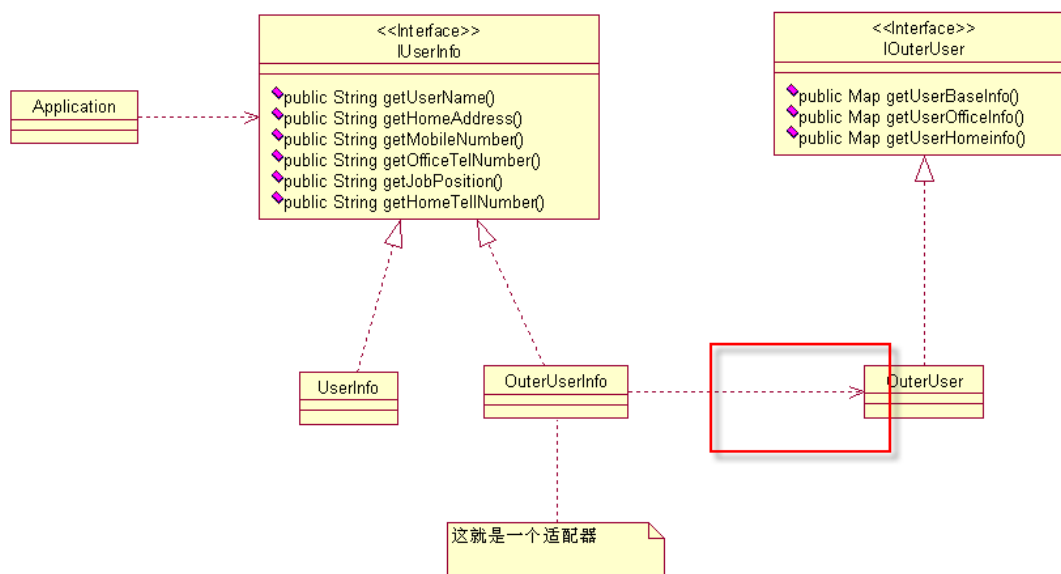
        //老板一想不对呀，兔子不吃窝边草，还是找人力资源的员工好点
        IUserInfo youngGirl = new OuterUserInfo(); //我们只修改了这一句好
        //从数据库中查到101个
        for(int i=0;i<101;i++){
            youngGirl.getMobileNumber();
        }

    }

}
```

大家看，使用了适配器模式只修改了一句话，其他的业务逻辑都不用修改就解决了系统对接的问题，而且在我们实际系统中只是增加了一个业务类的继承，就实现了可以查本公司的员工信息，也可以查人力资源公司的员工信息，尽量少的修改，通过扩展的方式解决了该问题。

适配器模式分为类适配器和对象适配器，这个区别不大，上边的例子就是类适配器，那对象适配器是什么样子的呢？对象适配器的类图是这个样子滴：



看到没？和上边的类图就一个箭头的图形的差异，一个是继承，一个是关联，就这么多区别，只要把我们上面的程序稍微修改一下就成了类适配器，这个大家自己考虑一下，简单的很。

适配器模式不适合在系统设计阶段采用，没有一个系统分析师会在做详设的时候考虑使用适配器模式，这个模式使用的主要场景是扩展应用中，就像我们上面的那个例子一样，系统扩展了，不符合原有设计的时候才考虑通过适配器模式减少代码修改带来的风险。

在论坛上有网友建议我加上模式的优缺点，这个我是不建议加上去，你先掌握了怎么使用，然后再想怎么更好的使用，而且我想你既然想到了你要使用这 23 个模式，肯定是有权衡的吧，那这个模式的优缺点是不是由你自己来总结会更好的呢？

第 9 章 模板方法模式【Template Method Pattern】

周三，9:00，我刚刚坐到位置，打开电脑准备开始干活。

“小三，小三，叫一下其它同事，到会议室，开会”老大跑过来吼，带着淫笑。还不等大家坐稳，老大就开讲了，

“告诉大家一个好消息，昨天终于把牛叉模型公司的口子打开了，要我们做悍马模型，虽然是第一个车辆模型，但是我们有能力，有信心做好，我们一定要…（中间省略 20 分钟的讲话，如果你听过领导人的讲话，这个你应该能够续上）”

动员工作做完了，那就开始压任务了，“这次时间是非常紧张的，只有一个星期的时间，小三，你负责在一个星期的时间把这批 10 万车模（注：车模是车辆模型的意思，不是香车美女那个车模）建设完成…”

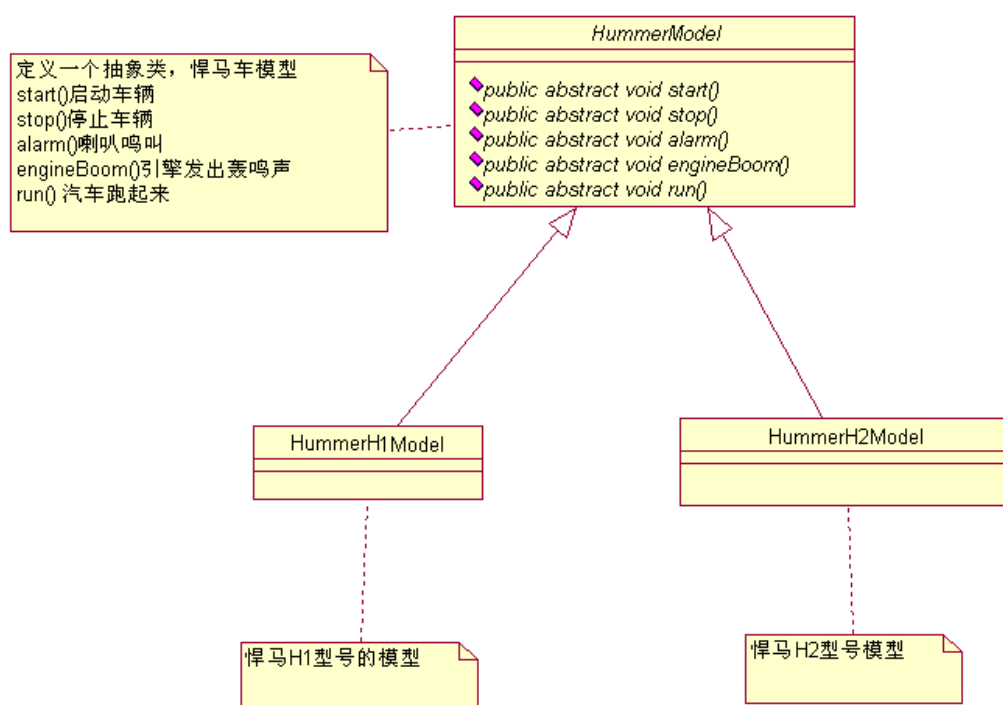
“一个星期？这个…，是真做不完，要做分析，做模板，做测试，还要考虑扩展性、稳定性、健壮性等，时间实在是太少了”还没等老大说完，我就急了，再不急我的小命就折在上面了！

“那这样，你只做实现，不考虑使用设计模式，扩展性等都不用考虑”老大又把我压回去了。

“不考虑设计模式？那…”

哎，领导已经布置任务了，那就开始死命的做吧，命苦不能怨政府，点背不能怪社会呀，然后就开始准备动手做，在做之前先介绍一下我们公司的背景，我们公司是做模型生产的，做过桥梁模型、建筑模型、机械模型，甚至是一些政府、军事的机密模型，这个不能说，就是把真实的实物按照一定的比例缩小或放大，用于试验、分析、量化或者是销售等等，上面提到的牛叉模型公司专门销售车辆模型的公司，自己不生产，我们公司是第一次从牛叉模型公司接单，那我怎么着也要把活干好，可时间很紧张呀，怎么办？

既然领导都说了，不考虑扩展性，那好办，我先设计个类图：



非常简单的实现，你要悍马模型，我就给你悍马模型，先写个抽象类，然后两个不同型号模型实现类，那我们把这个程序实现出来：

HummerModel 抽象类的程序清单如下：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * Hummer Model是悍马车辆模型的意思，不是悍马美女车模
 */
public abstract class HummerModel {

    /**
     * 首先，这个模型要能够被发动起来，别管是手摇发动，还是电力发动，反正
     * 是要能够发动起来，那这个实现要在实现类里了
     */
    public abstract void start();

    //能发动，那还要能停下来，那才是真本事
    public abstract void stop();
```



```
//喇叭会出声音，是滴滴叫，还是哔哔叫
public abstract void alarm();

//引擎会轰隆隆的响，不响那是假的
public abstract void engineBoom();

//那模型应该会跑吧，别管是人推的，还是电力驱动，总之要会跑
public abstract void run();
}
```

H1 型号悍马的定义如下：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 悍马车是每个越野者的最爱，其中H1最接近军用系列
 */
public class HummerH1Model extends HummerModel {

    @Override
    public void alarm() {
        System.out.println("悍马H1鸣笛...");
    }

    @Override
    public void engineBoom() {
        System.out.println("悍马H1引擎声音是这样在...");
    }

    @Override
    public void start() {
        System.out.println("悍马H1发动...");
    }

    @Override
    public void stop() {
        System.out.println("悍马H1停车...");
    }

    /*
```

```
    * 这个方法是很有意思的，它要跑，那肯定要启动，停止了等，也就是要调其他方法
    */
    @Override
    public void run() {

        //先发动汽车
        this.start();

        //引擎开始轰鸣
        this.engineBoom();

        //然后就开始跑了，跑的过程中遇到一条狗挡路，就按喇叭
        this.alarm();

        //到达目的地就停车
        this.stop();
    }
}
```

然后看悍马 H2 型号的实现：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * H1和H2有什么差别，还真不知道，真没接触过悍马
 */
public class HummerH2Model extends HummerModel {

    @Override
    public void alarm() {
        System.out.println("悍马H2鸣笛...");
    }

    @Override
    public void engineBoom() {
        System.out.println("悍马H2引擎声音是这样在...");
    }

    @Override
    public void start() {
```

```
        System.out.println("悍马H2发动...");
    }

    @Override
    public void stop() {
        System.out.println("悍马H1停车...");
    }

    /*
     * H2要跑，那肯定要启动，停止了等，也就是要调其他方法
     */
    @Override
    public void run() {

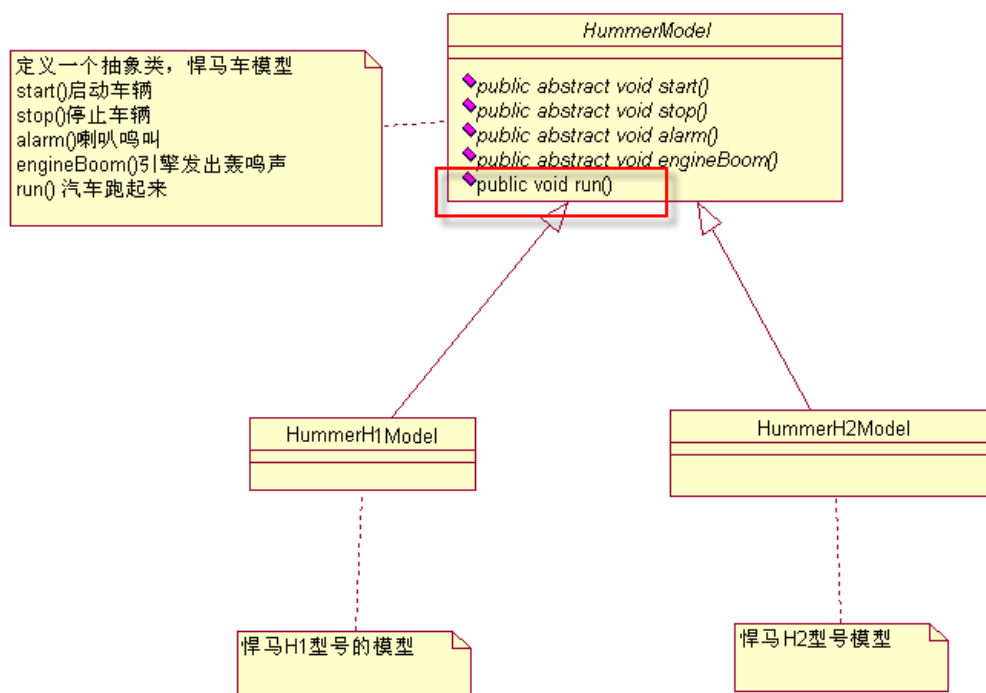
        //先发动汽车
        this.start();

        //引擎开始轰鸣
        this.engineBoom();

        //然后就开始跑了，跑的过程中遇到一条狗挡路，就按喇叭
        this.alarm();

        //到达目的地就停车
        this.stop();
    }
}
```

然后程序写到这里，你就看到问题了，run 方法的实现应该在抽象类上，不应该在实现类上，好，我们修改一下类图 and 实现：



就把 run 方法放到了抽象类中，那代码也相应的改变一下，先看 HummerModel.java:

```
package com.cbf4life;
```

```
/**
```

```
 * @author cbf4Life cbf4life@126.com
```

```
 * I'm glad to share my knowledge with you all.
```

```
 * Hummer Model是悍马车辆模型的意思，不是悍马美女车模
```

```
 */
```

```
public abstract class HummerModel {
```

```
    /*
```

```
     * 首先，这个模型要能够被发动起来，别管是手摇发动，还是电力发动，反正
```

```
     * 是要能够发动起来，那这个实现要在实现类里了
```

```
     */
```

```
    public abstract void start();
```

```
    //能发动，那还要能停下来，那才是真本事
```

```
    public abstract void stop();
```

```
    //喇叭会出声音，是滴滴叫，还是哔哔叫
```

```
    public abstract void alarm();
```

```
    //引擎会轰隆隆的响，不响那是假的
```

```
public abstract void engineBoom();

//那模型应该会跑吧，别管是人退的，还是电力驱动，总之要会跑
public void run() {

    //先发动汽车
    this.start();

    //引擎开始轰鸣
    this.engineBoom();

    //然后就开始跑了，跑的过程中遇到一条狗挡路，就按喇叭
    this.alarm();

    //到达目的地就停车
    this.stop();
}
}
```

下面是 HummerH1Model.java 程序清单：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 悍马车是每个越野者的最爱，其中H1最接近军用系列
 */
public class HummerH1Model extends HummerModel {

    @Override
    public void alarm() {
        System.out.println("悍马H1鸣笛...");
    }

    @Override
    public void engineBoom() {
        System.out.println("悍马H1引擎声音是这样在...");
    }

    @Override
    public void start() {
        System.out.println("悍马H1发动...");
    }
}
```

```
    }

    @Override
    public void stop() {
        System.out.println("悍马H1停车...");
    }
}
```

下面是 HummerH2Model.java 的程序清单：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * H1和H2有什么差别，还真不知道，真没接触过悍马
 */
public class HummerH2Model extends HummerModel {

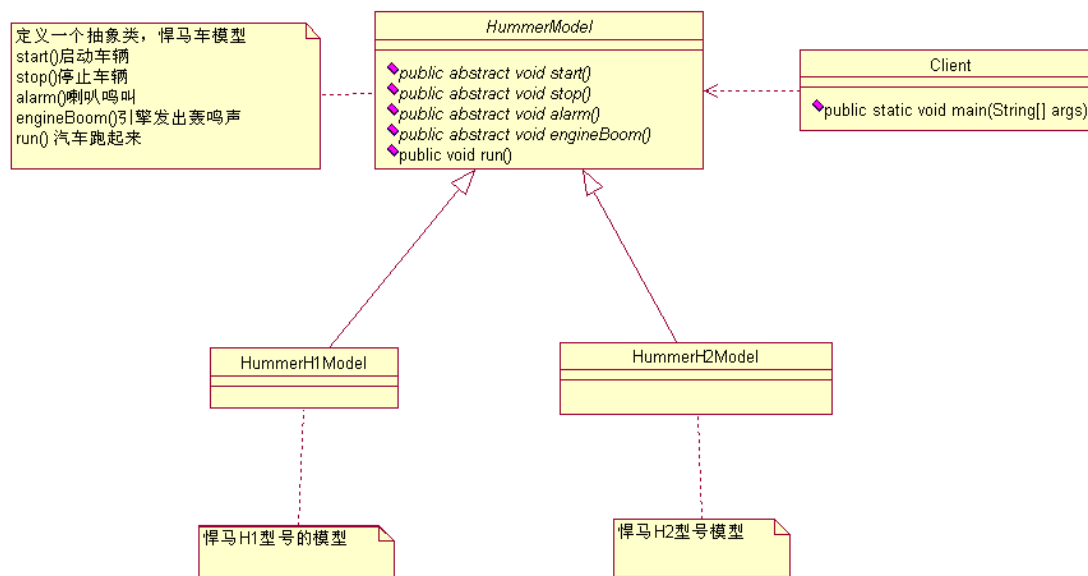
    @Override
    public void alarm() {
        System.out.println("悍马H2鸣笛...");
    }

    @Override
    public void engineBoom() {
        System.out.println("悍马H2引擎声音是这样在...");
    }

    @Override
    public void start() {
        System.out.println("悍马H2发动...");
    }

    @Override
    public void stop() {
        System.out.println("悍马H2停车...");
    }
}
```

类图修改完毕了，程序也该好了，提交给老大，老大一看，挺好，就开始生产了，并提交给客户使用了，那客户是如何使用的呢？类图上增加一个 Client 类，就是客户，我们这个是用 main 函数来代替他使用，类图如下：



然后看增加的 Client.java 程序，非常的简单：

```

package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 客户开始使用这个模型
 */
public class Client {

    public static void main(String[] args) {
        //客户开着H1型号，出去遛弯了
        HummerModel h1 = new HummerH1Model();
        h1.run(); //汽车跑起来了;

        //客户开H2型号，出去玩耍了
        HummerModel h2 = new HummerH2Model();
        h2.run();
    }
}
  
```

运行的结果如下：

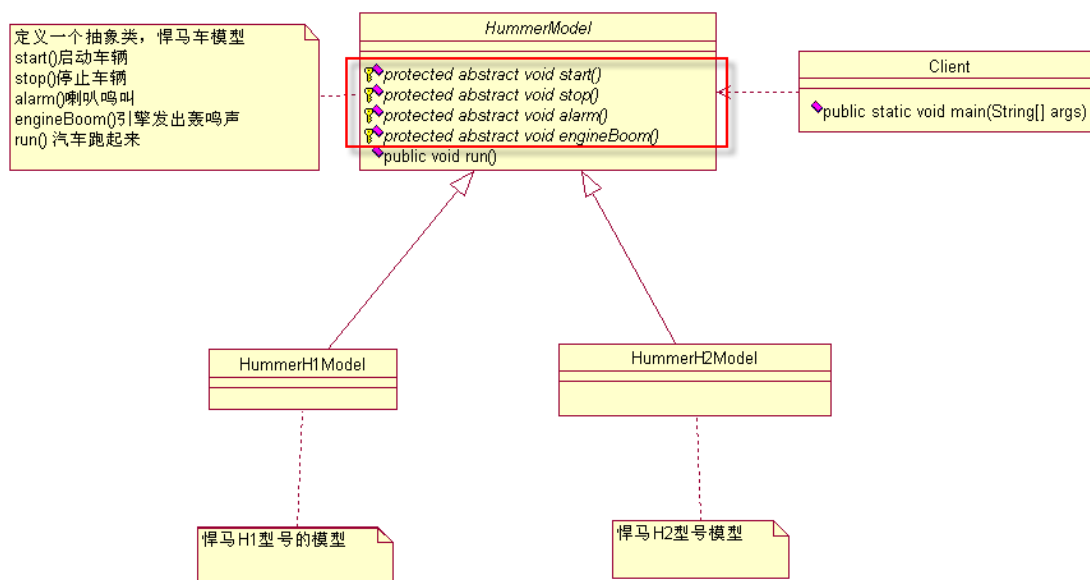
```

悍马H1发动...
悍马H1引擎声音是这样在...
悍马H1鸣笛...
悍马H1停车...
悍马H2发动...
悍马H2引擎声音是这样在...
悍马H2鸣笛...
悍马H2停车...

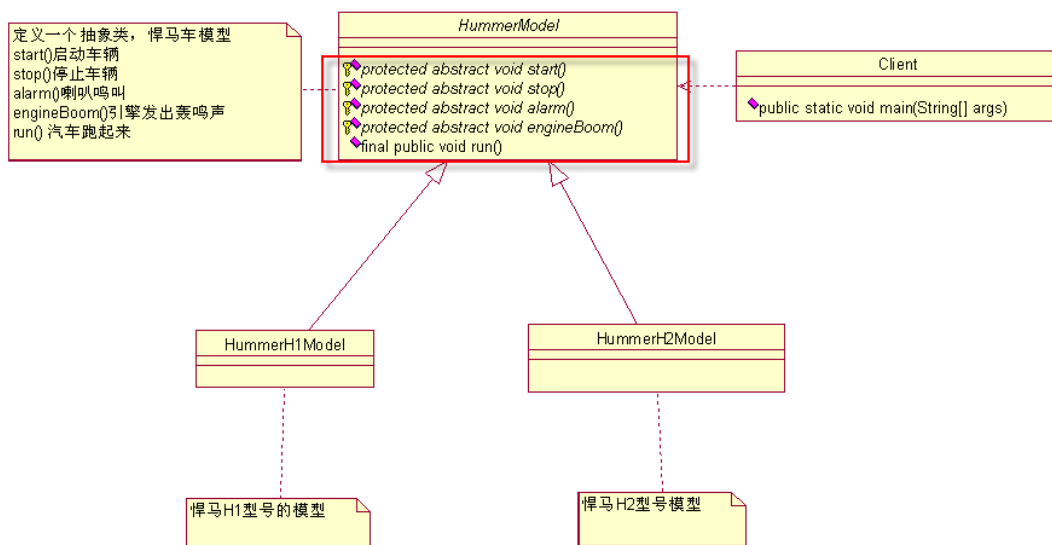
```

非常非常的简单，那如果我告诉这就是模板方法模式你会不会很不屑呢？就这模式，太简单了，我一直在使用呀，是的，你经常在使用，但你不知道这是模板方法模式，那些所谓的高手就可以很牛 X 的说“用模板方法模式就可以实现…”，你还要很崇拜的看着，哇，牛人，模板方法模式是什么呀？

然后我们继续回顾我们这个模型，回头一想，不对呀，需求分析的有点问题，客户要关心模型的启动，停止，鸣笛，引擎声音吗？他只要在 run 的过程中，听到或看都成了呀，暴露那么多的方法干啥？好了，我们重新修改一下类图：



把抽象类上的四个方法设置为 protected 访问权限，好了，既然客户不关心这几个方法，而且这四个方法都是由子类来实现的，那就设置成 protected 模式。咦~，那还有个缺陷，run 方法既然子类都不修改，那是不是可以设置成 final 类型呢？是滴是滴，类图如下：



好了，这才是模板方法模式，就是这个样子，我们只要修改抽象类代码就可以了，HummerModel.java 程序清单如下：

```

package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * Hummer Model是悍马车辆模型的意思，不是悍马美女车模
 */
public abstract class HummerModel {

    /**
     * 首先，这个模型要能够被发动起来，别管是手摇发动，还是电力发动，反正
     * 是要能够发动起来，那这个实现要在实现类里了
     */
    protected abstract void start();

    //能发动，那还要能停下来，那才是真本事
    protected abstract void stop();

    //喇叭会出声音，是滴滴叫，还是哗哗叫
    protected abstract void alarm();

    //引擎会轰隆隆的响，不响那是假的
    protected abstract void engineBoom();

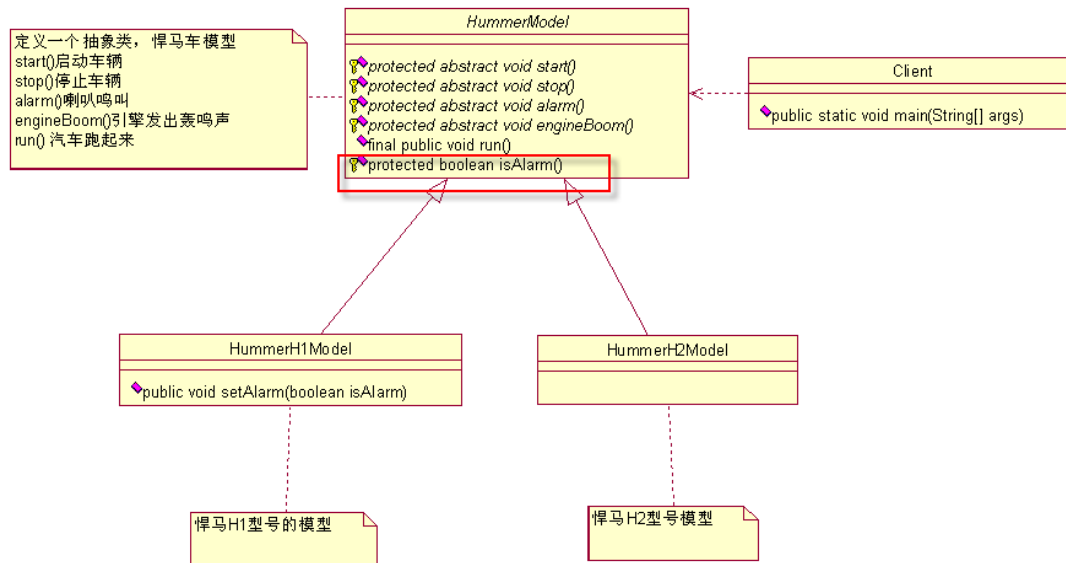
    //那模型应该会跑吧，别管是人退的，还是电力驱动，总之要会跑
  
```

```
final public void run() {  
  
    //先发动汽车  
    this.start();  
  
    //引擎开始轰鸣  
    this.engineBoom();  
  
    //然后就开始跑了，跑的过程中遇到一条狗挡路，就按喇叭  
    this.alarm();  
  
    //到达目的地就停车  
    this.stop();  
}  
}
```

其他的子类都不用修改(如果要修改，就是把四个方法的访问权限由 public 修改 protected)，大家请看这个 run 方法，他定义了调用其他方法的顺序，并且子类是不能修改的，这个叫做模板方法；start、stop、alarm、engineBoom 这四个方法是子类必须实现的，而且这四个方法的修改对应了不同的类，这个叫做基本方法，基本方法又分为三种：在抽象类中实现了的基本方法叫做具体方法；在抽象类中没有实现，在子类中实现了叫做抽象方法，我们这四个基本方法都是抽象方法，由子类来实现的；还有一种叫做钩子方法，这个等会讲。

到目前为止，这两个模型都稳定的运行，突然有一天，老大又找到了我，

“客户提出新要求了，那个喇叭想让它响就响，你看你设计的模型，车子一启动，喇叭就狂响，赶快修改一下”，确实是设计缺陷，呵呵，不过是我故意的，那我们怎么修改呢？看修改后的类图：



增加一个方法，isAlarm()，喇叭要不要响，这就是钩子方法(Hook Method)，那我们只要修改一下抽象类就可以了：

```

package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * Hummer Model是悍马车辆模型的意思，不是悍马美女车模
 */
public abstract class HummerModel {

    /**
     * 首先，这个模型要能够被发动起来，别管是手摇发动，还是电力发动，反正
     * 是要能够发动起来，那这个实现要在实现类里了
     */
    protected abstract void start();

    //能发动，那还要能停下来，那才是真本事
    protected abstract void stop();

    //喇叭会出声音，是滴滴叫，还是哔哔叫
    protected abstract void alarm();

    //引擎会轰隆隆的响，不响那是假的
    protected abstract void engineBoom();

    //那模型应该会跑吧，别管是人推的，还是电力驱动，总之要会跑

```

```
final public void run() {

    //先发动汽车
    this.start();

    //引擎开始轰鸣
    this.engineBoom();

    //喇叭想让它响就响，不想让它响就不响
    if(this.isAlarm()){
        this.alarm();
    }

    //到达目的地就停车
    this.stop();
}

//钩子方法，默认喇叭是会响的
protected boolean isAlarm(){
    return true;
}
}
```

钩子方法模式是由抽象类来实现的，子类可以重写的，H2 型号的悍马是不会叫的，喇叭是个摆设，看 HummerH2Model.java 代码：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * H1和H2有什么差别，还真不知道，真没接触过悍马
 */
public class HummerH2Model extends HummerModel {

    @Override
    protected void alarm() {
        System.out.println("悍马H2鸣笛...");
    }

    @Override
    protected void engineBoom() {
```

```

        System.out.println("悍马H2引擎声音是这样在...");
    }

    @Override
    protected void start() {
        System.out.println("悍马H2发动...");
    }

    @Override
    protected void stop() {
        System.out.println("悍马H1停车...");
    }

    //默认没有喇叭的
    @Override
    protected boolean isAlarm() {
        return false;
    }
}

```

那 H2 型号模型都没有喇叭，就是按了喇叭也没有声音，那客户端这边的调用没有任何修改，出来的结果就不同，我们先看 Client.java 程序：

```

package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 客户开始使用这个模型
 */
public class Client {

    public static void main(String[] args) {
        HummerH2Model h2 = new HummerH2Model();
        h2.run(); //H2型号的悍马跑起来
    }
}

```

运行的出来的结果是这样的：

```
悍马H2发动...
悍马H2引擎声音是这样在...
悍马H1停车...
```

那 H1 又有所不同了，它的喇叭要不要响是由客户来决定，其实在类图上已经标明了 setAlarm 这个方法，我们看 HummerH1Model.java 的代码：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 悍马车是每个越野者的最爱，其中H1最接近军用系列
 */
public class HummerH1Model extends HummerModel {
    private boolean alarmFlag = true; //是否要响喇叭

    @Override
    protected void alarm() {
        System.out.println("悍马H1鸣笛...");
    }

    @Override
    protected void engineBoom() {
        System.out.println("悍马H1引擎声音是这样在...");
    }

    @Override
    protected void start() {
        System.out.println("悍马H1发动...");
    }

    @Override
    protected void stop() {
        System.out.println("悍马H1停车...");
    }
}
```

```

@Override
protected boolean isAlarm() {
    return this.alarmFlag;
}

//要不要响喇叭，是有客户的来决定的
public void setAlarm(boolean isAlarm){
    this.alarmFlag = isAlarm;
}
}

```

这段代码呢修改了两个地方，一是重写了父类的 isAlarm() 方法，一是增加了一个 setAlarm 方法，由调用者去决定是否要这个功能，也就是喇叭要不要滴滴答答的响，哈哈，那我们看看 Client.java 的修改：

```

package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 客户开始使用这个模型
 */
public class Client {

    public static void main(String[] args) {
        //客户开着H1型号，出去遛弯了
        HummerH1Model h1 = new HummerH1Model();
        h1.setAlarm(true);
        h1.run(); //汽车跑起来了;

    }

}

```

运行的结果如下：

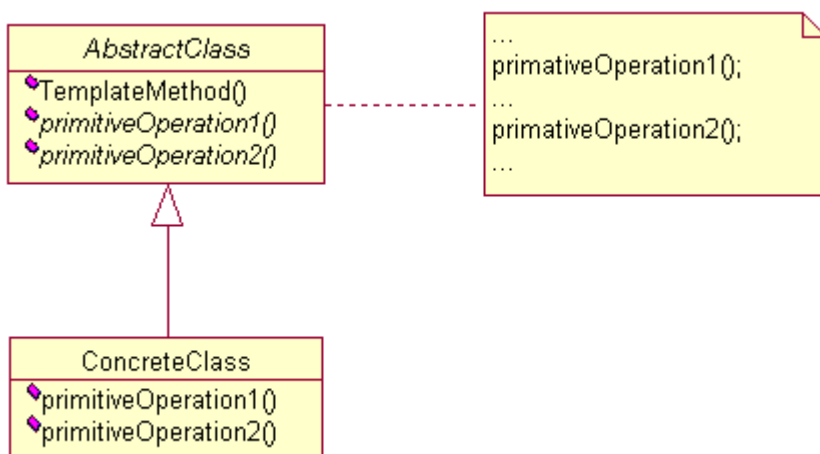
```

悍马H1发动...
悍马H1引擎声音是这样在...
悍马H1鸣笛...
悍马 H1 停车...

```

看到没，这个模型 run 起来就有声音了，那当然把 `h1.setAlarm(false)` 运行起来喇叭就没有声音了，钩子方法的作用就是这样滴。

那我们总结一下模板方法模式，模板方法模式就是在模板方法中按照一个的规则和顺序调用基本方法，具体到我们上面那个例子就是 `run` 方法按照规定的顺序(先调用 `start`, 然后再调用 `engineBoom`, 再调用 `alarm`, 最后调用 `stop`)调用本类的其他方法，并且由 `isAlarm` 方法的返回值确定 `run` 中的执行顺序变更，通用类图如下：



其中 `TemplateMethod` 就是模板方法，`operation1` 和 `operation2` 就是基本方法，模板方法是通过汇总或排序基本方法而产生的结果集。模板方法在一些开源框架中应用很多，它提供了一个抽象类，然后开源框架写了一堆子类，在《XXX In Action》中就说明了，如果你需要扩展功能，可以继承了这个抽象类，然后修改 `protected` 方法，再然后就是调用一个类似 `execute` 方法，就完成你的扩展开发，确实是一种简单的模式。

初级程序员在写程序的时候经常会问高手“父类怎么调用子类的方法”，这个问题很有普遍性，反正我是被问过好几回，那么父类是否可以调用子类的方法呢？我的回答是能，但强烈的、极度的不建议，怎么做呢？

- ✧ 把子类传递到父类的有参构造中，然后调用；
- ✧ 使用反射的方式调用，你使用了反射还有谁不能调用的？！
- ✧ 父类调用子类的静态方法。

这三种都是父类直接调用子类的方法，好用不？好用！解决问题了吗？解决了！项目中允许使用不？不允许！我就一直没有搞懂为什么要父类调用子类的方法，如果一定要调用子类，那为什么要继承它呢？

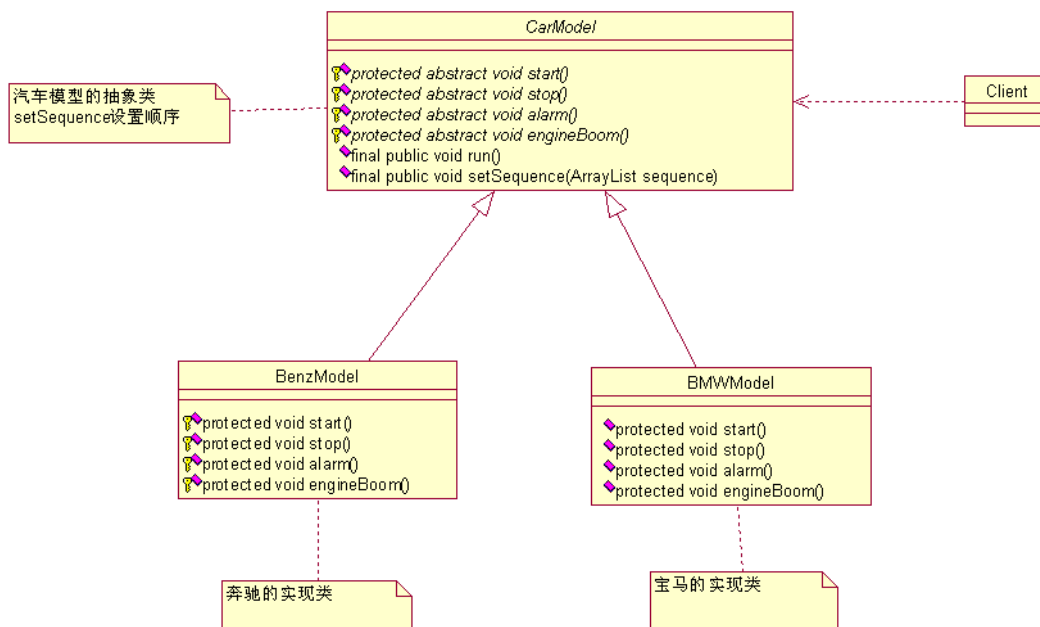
搞不懂。其实这个问题可以换个角度去理解，在重写了父类部分方法后，再调用从父类继承的方法，产生不同的结果（而这正是模板方法模式），这是不是也可以理解为父类调用了子类的方法呢？你修改了子类，影响了父类的结果，模板方法模式就是这样效果。

第 10 章 建造者模式【Builder Pattern】

又是一个周三，快要下班了，老大突然又拉住我，喜滋滋的告诉我“牛叉公司很满意我们做的模型，又签订了一个合同，把奔驰、宝马的车辆模型都交给我我们公司制作了，不过这次又额外增加了一个新需求：汽车的启动、停止、喇叭声音、引擎声音都有客户自己控制，他想什么顺序就什么顺序，这个没问题吧？”。

看着老大殷切的目光，我还能说啥，肯定的点头，“没问题！”，加班加点做呗，“再苦再累就当自己二百五 再难再险就当自己二皮脸 与君共勉！”这句话说出了我的心声。那任务是接下来，我们怎么做实现呢？

首先我们想了，奔驰、宝马都是一个产品，他们有共有的属性，牛叉公司关心的是单个模型，奔驰模型 A 是先有引擎声音，然后再启动；奔驰模型 B 呢是先启动起来，然后再有引擎声音，这才是牛叉公司要关心的，那到我们老大这边呢，就是满足人家的要求，要什么顺序就立马能产生什么顺序的模型出来，我呢就负责把老大的要求实现掉，而且还要是批量的，看不懂？没关系，继续看下去，首先由我生产出 N 多个奔驰和宝马车辆模型，这些车辆模型的都有 run 方法，但是具体到每一个模型的 run 方法可能中间的执行任务的顺序是不同的，老大说要啥顺序，我就给啥顺序，最终客户买走后只能是既定的模型，还是没听明白，我们继续，我们先把我们最基本的对象 Product 在类图中表明出来：



我们定义了一个 CarModel 的抽象类，其中 run 和 setSequence 是由抽象类实现的，其他都是子类自己实现，那这个是否可以解决这个问题呢？应该可以，我们把代码实现出来，先看 CarModel.java 程序：

```
package com.cbf4life;

import java.util.ArrayList;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 定义一个车辆模型的抽象类，所有的车辆模型都继承这里类
 */
public abstract class CarModel {

    //这个参数是各个基本方法执行的顺序
    private ArrayList<String> sequence = new ArrayList<String>();

    /**
     * 模型是启动开始跑了
     */
    protected abstract void start();

    //能发动，那还要能停下来，那才是真本事
    protected abstract void stop();

    //喇叭会出声音，是滴滴叫，还是哗哗叫
    protected abstract void alarm();

    //引擎会轰隆隆的响，不响那是假的
    protected abstract void engineBoom();

    //那模型应该会跑吧，别管是人推的，还是电力驱动，总之要会跑
    final public void run() {

        //循环一遍，谁在前，就先执行谁
        for(int i=0;i<this.sequence.size();i++){
            String actionName = this.sequence.get(i);

            if(actionName.equalsIgnoreCase("start")){ //如果是start关键字，
                this.start(); //开启汽车
            }else if(actionName.equalsIgnoreCase("stop")){ //如果是stop关键字
                this.stop(); //停止汽车
            }else if(actionName.equalsIgnoreCase("alarm")){ //如果是alarm关键字
                this.alarm(); //喇叭开始叫了
            }else if(actionName.equalsIgnoreCase("engine boom")){ //如果是engine
boom关键字
                this.engineBoom(); //引擎开始轰鸣
            }
        }
    }
}
```

```

    }

    }

}

//把传递过来的值传递到类内
final public void setSequence(ArrayList<String> sequence){
    this.sequence = sequence;
}
}

```

其中 `setSequence` 方法是允许客户自己设置一个顺序，是要先跑起来在有引擎声音还是先有引擎声音再跑起来，还是说那个喇叭就不要响，对于一个具体的模型永远都固定的，那这个事由牛叉告诉我们，有 1w 件事这样的，1w 件事那样的顺序，目前的设计都能满足这个要求。

`run` 方法使用了一个数组的遍历，确定那个是先执行，程序比较简单，不多说了，我们继续看两个实现类，先看 `BenzModel.java` 程序：

```

package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 奔驰车模型
 */
public class BenzModel extends CarModel {

    @Override
    protected void alarm() {
        System.out.println("奔驰车的喇叭声音是这个样子的...");
    }

    @Override
    protected void engineBoom() {
        System.out.println("奔驰车的引擎是这个声音的...");
    }

    @Override

```

```
protected void start() {  
    System.out.println("奔驰车跑起来是这个样子的...");  
}  
  
@Override  
protected void stop() {  
    System.out.println("奔驰车应该这样停车...");  
}  
}
```

没啥特别的，不多说，再看 BMWModel.java 程序：

```
package com.cbf4life;  
  
/**  
 * @author cbf4Life cbf4life@126.com  
 * I'm glad to share my knowledge with you all.  
 * 宝马车模型  
 */  
public class BMWModel extends CarModel {  
  
    @Override  
    protected void alarm() {  
        System.out.println("宝马车的喇叭声音是这个样子的...");  
    }  
  
    @Override  
    protected void engineBoom() {  
        System.out.println("宝马车的引擎是这个声音的...");  
    }  
  
    @Override  
    protected void start() {  
        System.out.println("宝马车跑起来是这个样子的...");  
    }  
  
    @Override
```

```
protected void stop() {  
    System.out.println("宝马车应该这样停车...");  
}  
  
}
```

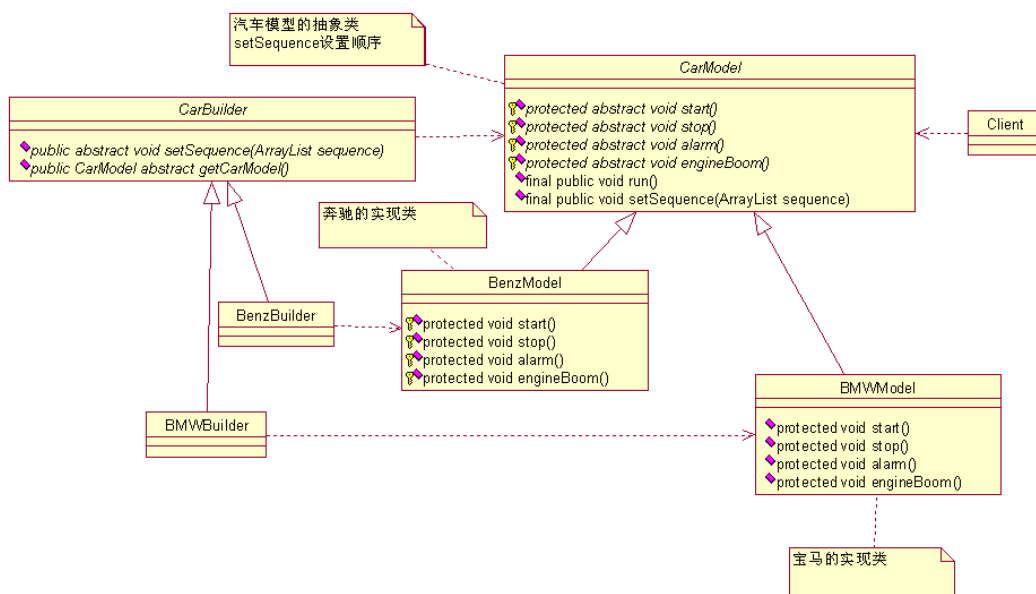
两个实现类都完成，我们再来看牛叉公司要的要求，我先要 1 个奔驰的模型，这个模型的要求是跑的时候，先发动引擎，然后再挂档启动，然后停下来，不需要喇叭，那怎么实现呢：

```
package com.cbf4life;  
  
import java.util.ArrayList;  
  
/**  
 * @author cbf4Life cbf4life@126.com  
 * I'm glad to share my knowledge with you all.  
 * 最终客户开始使用这个模型  
 */  
public class Client {  
  
    public static void main(String[] args) {  
        /*  
         * 客户告诉牛叉公司，我要这样一个模型，然后牛叉公司就告诉我老大  
         * 说要这样一个模型，这样一个顺序，然后我就来制造  
         */  
        BenzModel benz = new BenzModel();  
        ArrayList<String> sequence = new ArrayList<String>(); //存放run的顺序  
  
        sequence.add("engine boom"); //客户要求，run的时候先发动引擎  
        sequence.add("start"); //启动起来  
        sequence.add("stop"); //开了一段就停下来  
  
        //然后我们把这个顺序给奔驰车：  
        benz.setSequence(sequence);  
        benz.run();  
    }  
}
```

运行结果是这样的：

奔驰车的引擎是这个声音的...
 奔驰车跑起来是这个样子的...
 奔驰车应该这样停车...

看，满足了牛叉公司的需求了，满足完了，还要下一个需求呀，然后是第 2 件宝马模型，只要启动，停止，其他的什么都不要，第 3 件模型，先喇叭，然后启动，然后停止，第 4 件...，直到把你逼疯为止，那怎么办？我们修改程序，满足这种变态需求，好，看我如何修改，先修改类图：



增加了一个 CarBuilder 的抽象类，以及两个实现类，其目的是你要什么排列顺序的车，我就给你什么顺序的车，那我们先看 CarBuilder.java 抽象类的程序：

```

package com.cbf4life;

import java.util.ArrayList;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 要什么顺序的车，你说，我给建造出来
 */
public abstract class CarBuilder {

    //建造一个模型，你要给我一个顺序要，就是组装顺序
    public abstract void setSequence(ArrayList<String> sequence);
  
```

```
//设置完毕顺序后，就可以直接拿到这个车辆模型
public abstract CarModel getCarModel();
}
```

这个抽象类比较简单，程序上也以后注释说明，不多说，我们看两个实现类，先看 BenzBuilder.java 的程序代码：

```
package com.cbf4life;

import java.util.ArrayList;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 各种设施都给了，我们按照一定的顺序制造一个奔驰车
 */
public class BenzBuilder extends CarBuilder {
    private BenzModel benz = new BenzModel();

    @Override
    public CarModel getCarModel() {
        return this.benz;
    }

    @Override
    public void setSequence(ArrayList<String> sequence) {
        this.benz.setSequence(sequence);
    }
}
```

下面是 BMWBuilder.java 程序代码：

```
package com.cbf4life;

import java.util.ArrayList;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
```



```

* 给定一个顺序，返回一个宝马车
*/
public class BMWBuilder extends CarBuilder {
    private BMWModel bmw = new BMWModel();

    @Override
    public CarModel getCarModel() {
        return this.bmw;
    }

    @Override
    public void setSequence(ArrayList<String> sequence) {
        this.bmw.setSequence(sequence);
    }
}

```

程序很简单，很实用，这就是我最希望的，简单而又实用，我欣赏这样的程序员。那我们再来看我们 Client.java 程序的修改：

```

package com.cbf4life;

import java.util.ArrayList;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 最终客户开始使用这个模型
 */
public class Client {

    public static void main(String[] args) {
        /*
         * 客户告诉牛叉公司，我要这样一个模型，然后牛叉公司就告诉我老大
         * 说要这样一个模型，这样一个顺序，然后我就来制造
         */
        ArrayList<String> sequence = new ArrayList<String>(); //存放run的顺序
        sequence.add("engine boom"); //客户要求，run的时候先发动引擎
        sequence.add("start"); //启动起来
        sequence.add("stop"); //开了一段就停下来

        //要一个奔驰车：
    }
}

```

```

        BenzBuilder benzBuilder = new BenzBuilder();
        //把顺序给这个builder类，制造出这样一个车出来
        benzBuilder.setSequence(sequence);
        //制造出一个奔驰车
        BenzModel benz = (BenzModel)benzBuilder.getCarModel();
        //奔驰车跑一下看看
        benz.run();

    }

}

```

运行结果如下：

```

奔驰车的引擎是这个声音的...
奔驰车跑起来是这个样子的...
奔驰车应该这样停车...

```

那如果我再想要个同样顺序的宝马车呢？很简单，Client.java 程序如下：

```

package com.cbf4life;

import java.util.ArrayList;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 最终客户开始使用这个模型
 */
public class Client {

    public static void main(String[] args) {

        ArrayList<String> sequence = new ArrayList<String>(); //存放run的顺序
        sequence.add("engine boom"); //客户要求，run的时候先发动引擎
        sequence.add("start"); //启动起来
        sequence.add("stop"); //开了一段就挺下来

        //要一个奔驰车：
        BenzBuilder benzBuilder = new BenzBuilder();
        //把顺序给这个builder类，制造出这样一个车出来
        benzBuilder.setSequence(sequence);
    }
}

```

```
//制造出一个奔驰车
BenzModel benz = (BenzModel)benzBuilder.getCarModel();
//奔驰车跑一下看看
benz.run();

//按照同样的顺序，我再要一个宝马
BMWBuilder bmwBuilder = new BMWBuilder();
bmwBuilder.setSequence(sequence);
BMWModel bmw = (BMWModel)bmwBuilder.getCarModel();
bmw.run();

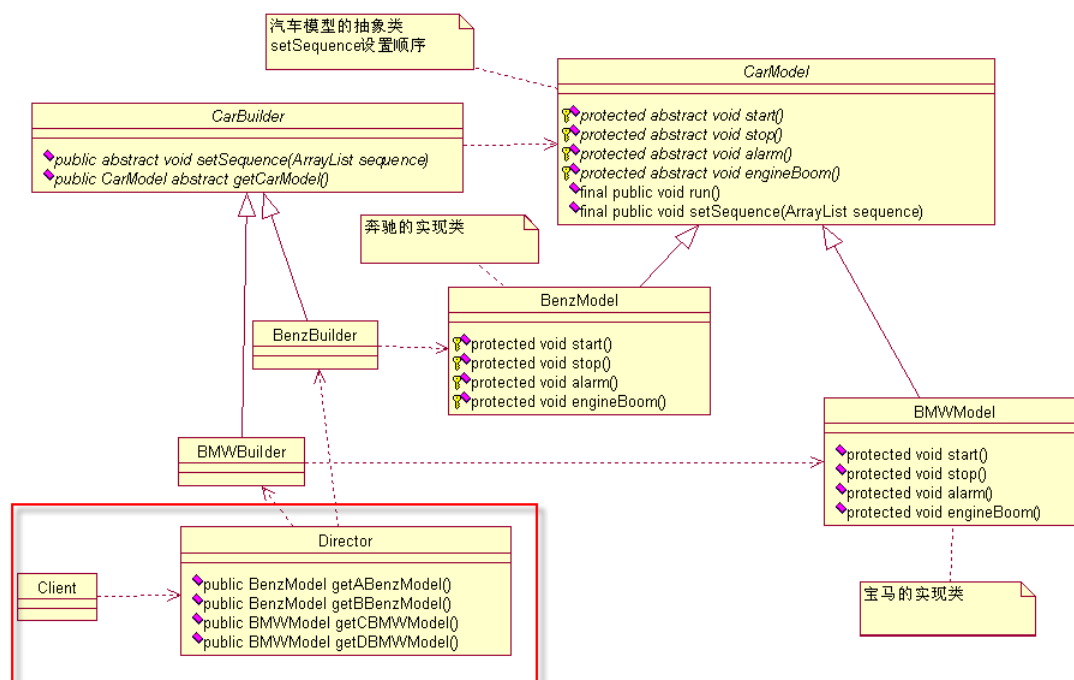
}

}
```

运行结果如下：

```
奔驰车的引擎是这个声音的...
奔驰车跑起来是这个样子的...
奔驰车应该这样停车...
宝马车的引擎是这个声音的...
宝马车跑起来是这个样子的...
宝马车应该这样停车...
```

看，是不是比刚开始直接访问产品类(Product)简单了很多吧，那这有个这样的需求，这四个过程(start,stop,alarm,engineBoom)按照排列组合有很多种，那我们怎么满足这种需求呢？也就是要有个类来安排这几个方法组合，就像导演安排演员一样，那个先出场那个后出场，那个不出场，我们这个也叫导演类，那我们修改一下类图：



增加了 Director 类，这个类是做了层封装，类中的方法说明如下：

A 型号的奔驰车辆模型是只有启动（start）、停止(stop)方法，其他的引擎声音、喇叭都没有；

B 型号的奔驰车是先发动引擎（engine boom），然后启动(start)，再然后停车(stop)，没有喇叭；

C 型号的宝马车是先喇叭叫一下（alarm），然后（start），再然后是停车(stop)，引擎不轰鸣；

D 型号的宝马车就一个启动(start)，然后一路跑到黑，永动机，没有停止方法，没有喇叭，没有引擎轰鸣；

E 型号、F 型号...等等，可以有很多，启动(start)、停止(stop)、喇叭(alarm)、引擎轰鸣(engine boom)

这四个方法在这个类中可以随意的自由组合，有几种呢？好像是排列组合，这个不会算，高中数学没学好，反正有很多种了，这里都可以实现。

我们看一下代码实现，Director.java 代码如下：

```

package com.cbf4life;

import java.util.ArrayList;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 导演安排顺序，生产车辆模型
 */
public class Director {
    private ArrayList<String> sequence = new ArrayList();
    private BenzBuilder benzBuilder = new BenzBuilder();
}

```

```
private BMWBuilder bmwBuilder = new BMWBuilder();

/*
 * A类型的奔驰车模型，先start,然后stop,其他什么引擎了，喇叭一概没有
 */
public BenzModel getABenzModel(){
    //清理场景，这里是一些初级程序员不注意的地方
    this.sequence.clear();

    //这只ABenzModel的执行顺序
    this.sequence.add("start");
    this.sequence.add("stop");

    //按照顺序返回一个奔驰车
    this.benzBuilder.setSequence(this.sequence);
    return (BenzModel)this.benzBuilder.getCarModel();
}

/*
 * B型号的奔驰车模型，是先发动引擎，然后启动，然后停止，没有喇叭
 */
public BenzModel getBBenzModel(){
    this.sequence.clear();

    this.sequence.add("engine boom");
    this.sequence.add("start");
    this.sequence.add("stop");

    this.benzBuilder.setSequence(this.sequence);
    return (BenzModel)this.benzBuilder.getCarModel();
}

/*
 * C型号的宝马车是先按下喇叭（炫耀嘛），然后启动，然后停止
 */
public BMWModel getCBMWModel(){
    this.sequence.clear();

    this.sequence.add("alarm");
    this.sequence.add("start");
    this.sequence.add("stop");

    this.bmwBuilder.setSequence(this.sequence);
```

```

        return (BMWModel) this.bmwBuilder.getCarModel();
    }

    /**
     * D类型的宝马车只有一个功能，就是跑，启动起来就跑，永远不停止，牛叉
     */
    public BMWModel getDBMWModel(){
        this.sequence.clear();

        this.sequence.add("start");

        this.bmwBuilder.setSequence(this.sequence);
        return (BMWModel) this.benzBuilder.getCarModel();
    }

    /**
     * 这里还可以有很多方法，你可以先停止，然后再启动，或者一直停着不动，静态的嘛
     * 导演类嘛，按照什么顺序是导演说了算
     */
}

```

大家看一下程序中有很多 this 调用，这个我一般是这样要求项目组成员的，如果你要调用类中的成员变量或方法，需要在前面加上 this 关键字，不加也能正常的跑起来，但是不清晰，加上 this 关键字，我就是要调用本类中成员变量或方法，而不是本方法中的一个变量，还有 super 方法也是一样，是调用父类的成员变量或者方法，那就加上这个关键字，不要省略，这要靠约束，还有就是程序员的自觉性，他要是死不悔改，那咱也没招。

上面每个方法都一个 this.sequence.clear()，这个估计你一看就明白，但是做为一个系统分析师或是技术经理一定要告诉项目成员，ArrayList 和 HashMap 如果定义成类的成员变量，那你在方法中调用一定要做一个 clear 的动作，防止数据混乱，这个如果你发生过一次问题的话，比如 ArrayList 中出现一个“出乎意料”的数据，而你又花费了几个通宵才解决这个问题，那你会有很深刻的印象。

然后 Client 程序就只与 Director 打交道了，牛叉公司要 A 类型的奔驰车 1W 辆，B 类型的奔驰车 100W 辆，C 类型的宝马车 1000W 辆，D 类型的我不要：

```

package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.

```

```

* 这里是牛叉公司的天下，他要啥我们给啥
*/
public class Client {

    public static void main(String[] args) {
        Director director = new Director();

        //1w辆A类型的奔驰车
        for(int i=0;i<10000;i++){
            director.getABenzModel().run();
        }

        //100w辆B类型的奔驰车
        for(int i=0;i<1000000;i++){
            director.getBBenzModel().run();
        }

        //1000w辆C类型的宝马车
        for(int i=0;i<10000000;i++){
            director.getCBMWModel().run();
        }
    }
}

```

清晰，简单吧，我们写程序重构的最终目的就是这个，简单，清晰，代码是让人看的，不是写完就完事了，我一直在教育我带的团队，Java 程序不是像我们前辈写那个二进制代码、汇编一样，写完基本上就自己能看懂，别人看就跟看天书一样，现在的高级语言，要像写中文汉字一样，你写的，别人能看懂。

整个程序编写完毕，而且简洁明了，这就是建造者模式，中间有几个角色需要说明一下：

Client 就是牛叉公司，这个到具体的应用中就是其他的模块或者页面；

CarModel 以及两个实现类 BenzModel 和 BMWModel 叫做产品类(Product Class)，这个产品类实现了模板方法模式，也就是有模板方法和基本方法，这个参考上一节的模板方法模式；

CarBuilder 以及两个实现类 BenzBuilder 和 BMWBuilder 叫做建造者(Builder Class)，在上面的那个例子中就是我和我的团队，负责建造 Benz 和 BMW 车模，按照指定的顺序；

Director 类叫做导演类(Director Class)，负责安排已有模块的顺序，然后告诉 Builder 开始建造，在上面的例子中就是我们的老大，Client 找到老大，说我要这个，这个，那个类型的车辆模型，然后老大就把命令传递给我，我和我的团队就开始拼命的建造，于是一个项目建设完毕了。

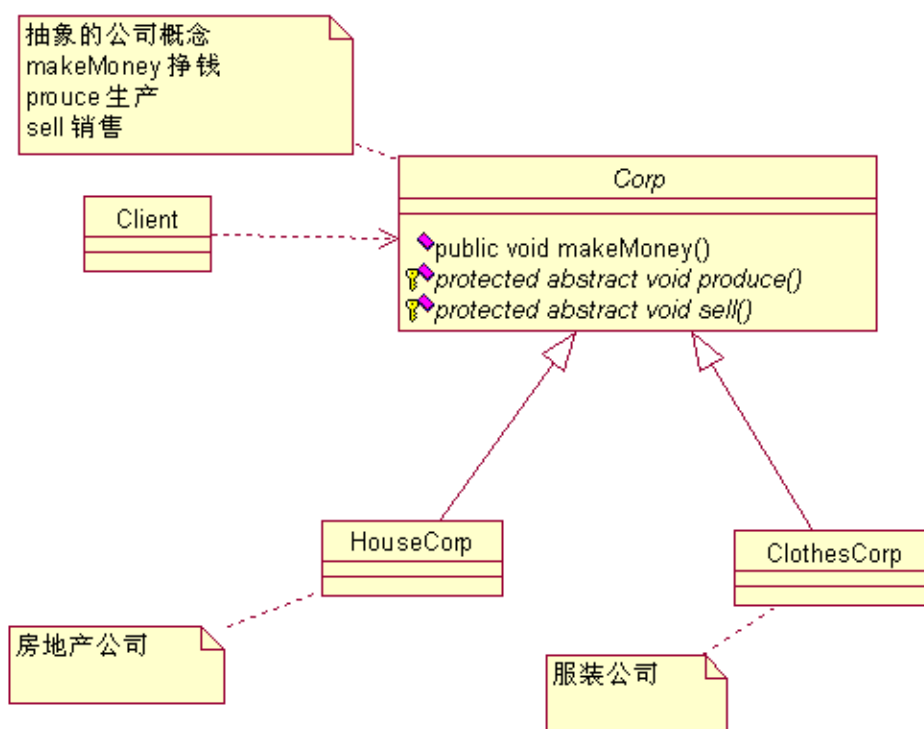
大家看到这里估计就开始犯嘀咕了，这个建造者模式和工厂模式非常相似呀，yes，是的，是非常相似，但

是记住一点你就可以游刃有余的使用了：建造者模式最主要功能是基本方法的调用顺序安排，也就是这些基本方法已经实现了；而工厂方法则重点是创建，你要什么对象我创建一个对象出来，组装顺序则不是他关心的。

建造者模式使用的场景，一是产品类非常的复杂，或者产品类中的调用顺序不同产生了不同的效能，这个时候使用建造者模式是非常合适，我曾在一个银行交易类项目中遇到了这个问题，一个产品的定价计算模型有 n 多种，每个模型有固定的计算步骤，计算非常复杂，项目中就使用了建造者模式；二是“在对象创建过程中会使用到系统中的一些其它对象，这些对象在产品对象的创建过程中不易得到”，这个是我没有遇到过的，创建过程中不易得到？那为什么在设计阶段不修正这个问题，创建的时候都不易得到耶！

第 11 章 桥梁模式【Bridge Pattern】

今天我要说说我自己，梦想中的我自己，我身价过亿，有两个大公司，一个是房地产公司，一个是服装制造业，这两个公司都很赚钱，天天帮我在累加财富，其实是什么公司我倒是不关心，我关心的是是不是在赚钱，赚了多少钱，这才是我关心的，我是商人呀，唯利是图是我的本性，偷税漏税是我的方法，欺上瞒下、压榨员工血汗我是的手段嘛，我先用类图表示一下我这两个公司：



类图很简单，声明了一个 **Corp** 抽象类，定义一个公司的抽象模型，公司首要是赚钱的，不赚钱谁开公司，做义务或善举那也是有背后利益支撑的，我还是赞成这句话“天下熙熙，皆为利来；天下壤壤，皆为利往”，那我们先看 **Corp** 类的源代码：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 定义一个公司的抽象类
 */
public abstract class Corp {
```

```

/*
 * 是公司就应该有生产把，甭管是什么软件公司还是制造业公司
 * 那每个公司的生产的东西都不一样，所以由实现类来完成
 */
protected abstract void produce();

/*
 * 有产品了，那肯定要销售呀，不销售你公司怎么生存
 */
protected abstract void sell();

//公司是干什么的？赚钱的呀，不赚钱傻子才干
public void makeMoney(){

    //每个公司都是一样，先生产
    this.produce();

    //然后销售
    this.sell();

}

}

```

合适的方法存在合适的类中，这个基本上是每本 Java 基础书上都会讲的，但是到实际的项目中应用的时候就不是这么回事儿了，正常的很。我们继续看两个实现类如何实现的，先看 HouseCorp 类，这个是最赚钱的公司：

```

package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 房地产公司，按照翻译来说应该叫realty corp，这个是比较准确的翻译
 * 但是我问你房地产公司翻译成英文，你第一反应什么？对嘛还是house corp!
 */
public class HouseCorp extends Corp {

    //房地产公司就是盖房子
    protected void produce() {
        System.out.println("房地产公司盖房子...");
    }

}

```

```
//房地产卖房子，自己住那可不赚钱
protected void sell() {
    System.out.println("房地产公司出售房子...");
}

//房地产公司很High了，赚钱，计算利润
public void makeMoney(){
    super.makeMoney();
    System.out.println("房地产公司赚大钱了...");
}

}
```

然后看服装公司，虽然不景气，但好歹也是赚钱的：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 服装公司，这个行当现在不怎么样
 */
public class ClothesCorp extends Corp {

    //服装公司生产的就是衣服了
    protected void produce() {
        System.out.println("服装公司生产衣服...");
    }

    //服装公司买服装，可只卖服装，不买穿衣服的模特
    protected void sell() {
        System.out.println("服装公司出售衣服...");
    }

    //服装公司不景气，但怎么说也是赚钱行业也
    public void makeMoney(){
        super.makeMoney();
        System.out.println("服装公司赚小钱...");
    }

}
```

两个公司都有了，那肯定有人会关心两个公司的运营情况呀，我自己怎么也要知道它是生产什么的，赚多少钱吧，那看看这个 Client.java 是什么样子的：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 我要关心我自己的公司了
 */
public class Client {

    public static void main(String[] args) {

        System.out.println("-----房地产公司是这个样子运行的-----");
        //先找到我的公司
        HouseCorp houseCorp =new HouseCorp();
        //看我怎么挣钱
        houseCorp.makeMoney();
        System.out.println("\n");

        System.out.println("-----服装公司是这样运行的-----");
        ClothesCorp clothesCorp = new ClothesCorp();
        clothesCorp.makeMoney();

    }
}
```

看，这个代码很简单，运行结果如下：

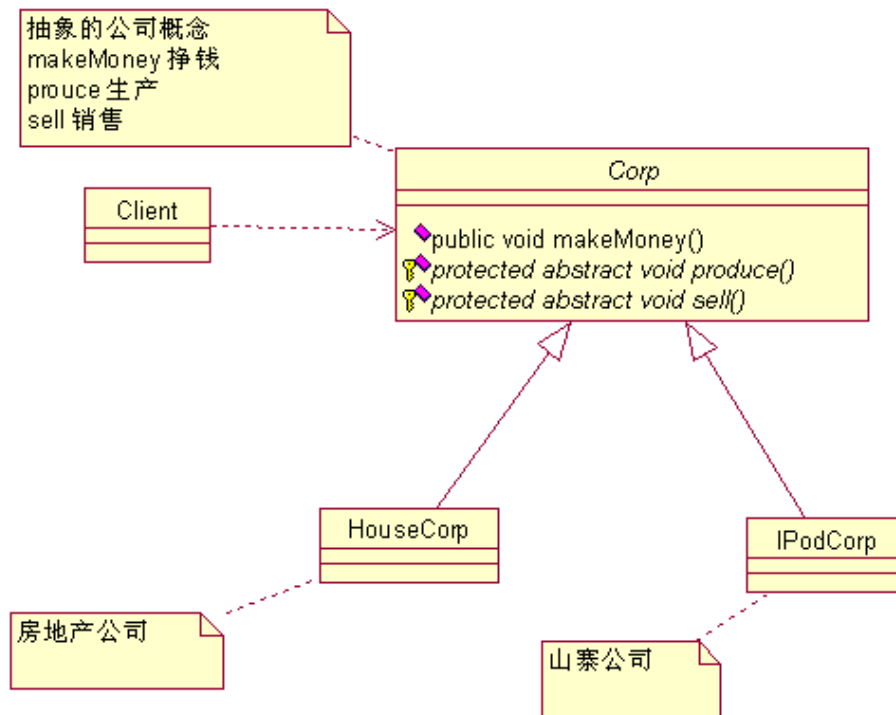
```
-----房地产公司是这个样子运行的-----
房地产公司盖房子...
房地产公司出售房子...
房地产公司赚大钱了...

-----服装公司是这样运行的-----
服装公司生产衣服...
服装公司出售衣服...
服装公司赚小钱...
```

上述代码完全可以描述我现在的公司，但是你要知道万物都是运动的，你要用运动的眼光看问题，我公司也

会发展，终于在有一天我觉得赚钱速度太慢，于是我上下疏通，左右打关系，终于开辟了一条赚钱的康庄大道：生产山寨产品，什么产品呢？就是市场上什么牌子的东西火爆我生产什么牌子的东西，甭管是打火机还是电脑，只要它火爆，我就生产，赚过了高峰期就换个产品，打一枪换一个牌子，不承担售后成本、也不担心销路问题，我只有正品的十分之一的价格，你买不买？哈哈，赚钱呀！

企业的方向定下来了，通过调查，市场上前段时间比较火爆的是苹果 iPod，那咱就生产这个，把服装厂该成 iPod 生产厂，看类图的变化：



好，我的企业改头换面了，开始生产 iPod 产品了，看我 IPodCorp 类的实现：

```

package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 我是山寨老大，你流行啥我就生产啥
 * 现在流行iPod
 */
public class IPodCorp extends Corp {

    //我开始生产iPod了
    protected void produce() {

```

```

        System.out.println("我生产iPod...");
    }

    //山寨的iPod很畅销，便宜呀
    protected void sell() {
        System.out.println("iPod畅销...");
    }

    //狂赚钱
    public void makeMoney(){
        super.makeMoney();
        System.out.println("我赚钱呀...");
    }
}

```

这个厂子修改完毕了，我这个董事长还要去看看到底生产什么的，看这个 Client.java 程序：

```

package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 我要关心我自己的公司了
 */
public class Client {

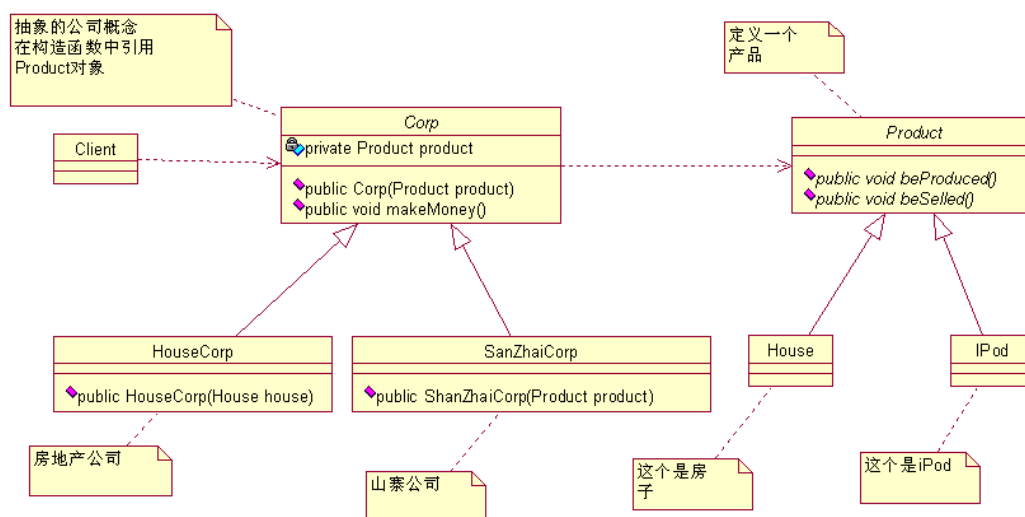
    public static void main(String[] args) {
        System.out.println("-----房地产公司是这个样子运行的-----");
        //先找到我的公司
        HouseCorp houseCorp =new HouseCorp();
        //看我怎么挣钱
        houseCorp.makeMoney();
        System.out.println("\n");

        System.out.println("-----山寨公司是这样运行的-----");
        IPodCorp iPodCorp = new IPodCorp();
        iPodCorp.makeMoney();
    }
}

```

确实，只用修改了几句话，我的服装厂就开始变成山寨 iPod 生产车间，然后就看我的财富在积累积累，你想呀山寨的东西不需要特别的销售渠道（正品到哪里我就到哪里），不需要维修成本（大不了给你换个，你还想咋地，过了高峰期我就改头换面了你找谁维修去，投诉？投诉谁呢？），不承担广告成本（正品在打广告，我还需要吗？需要吗？），但是我也有犯愁的时候，我这是个山寨工厂，要及时的生产出市场上流行产品，转型要快，要灵活，今天从生产 iPod 转为生产 MP4，明天再转为生产上网本，这个都需要灵活的变化，不要限制的太死，那问题来了，每次我的厂房，我的工人，我的设备都在，不可能每次我换个山寨产品我的厂子就彻底不要了，这不行，成本忒高了点，那怎么办？

Thinking, Thinking..., I got an idea!, 这样设计:



Corp 类和 Product 类建立一个关联关系，可以彻底解决我以后山寨公司生产产品的问题，看程序说话，先看 Product 抽象类：

```
package com.cbf4life.implementor;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 这是我整个集团公司的产品类
 */
public abstract class Product {

    //甭管是什么产品它总要是能被生产出来
    public abstract void beProduced();
}
```

```
//生产出来的东西，一定要销售出去，否则亏本呀
public abstract void beSelled();

}
```

简单，忒简单了，看 House 产品类：

```
package com.cbf4life.implementor;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 这是我集团公司盖的房子
 */
public class House extends Product {

    //豆腐渣就豆腐渣呗，好歹也是个房子
    public void beProducted() {
        System.out.println("生产出的房子是这个样子的...");
    }

    //虽然是豆腐渣，也是能够销售出去的
    public void beSelled() {
        System.out.println("生产出的房子卖出去了...");
    }

}
```

不多说，看 Clothes 产品类：

```
package com.cbf4life.implementor;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 我集团公司生产的衣服
 */
public class Clothes extends Product {

    public void beProducted() {
```



```
        System.out.println("生产出的衣服是这个样子的...");
    }

    public void beSelled() {
        System.out.println("生产出的衣服卖出去了...");
    }
}
```

下面是 iPod 产品类:

```
package com.cbf4life.implementor;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 生产iPod了
 */
public class IPod extends Product {

    public void beProducted() {
        System.out.println("生产出的iPod是这个样子的...");
    }

    public void beSelled() {
        System.out.println("生产出的iPod卖出去了...");
    }
}
```

产品类是有了, 那我们再看 Corp 抽象类:

```
package com.cbf4life.abstraction;

import com.cbf4life.implementor.Product;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 定义一个公司的抽象类
 */
```

```
public abstract class Corp {

    //定义一个产品对象，抽象的了，不知道具体是什么产品
    private Product product;

    //构造函数，由子类定义传递具体的产品进来
    public Corp(Product product){
        this.product = product;
    }

    //公司是干什么的？赚钱的呀，不赚钱傻子才干
    public void makeMoney(){

        //每个公司都是一样，先生产
        this.product.beProducted();

        //然后销售
        this.product.beSelled();

    }

}
```

这里多了个有参构造，其目的是要继承的子类都必选重写自己的有参构造函数，把产品类传递进来，再看子类 HouseCorp 的实现：

```
package com.cbf4life.abstraction;

import com.cbf4life.implementor.House;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 房地产公司，按照翻译来说应该叫realty corp，这个是比较准确的翻译
 * 但是我问你房地产公司翻译成英文，你第一反应什么？对嘛还是house corp!
 */
public class HouseCorp extends Corp {

    //定义传递一个House产品进来
    public HouseCorp(House house){
        super(house);
    }

}
```

```

    }

    //房地产公司很High了，赚钱，计算利润
    public void makeMoney(){
        super.makeMoney();
        System.out.println("房地产公司赚大钱了...");
    }
}

```

理解上没有多少难度，不多说，继续看山寨公司的实现：

```

package com.cbf4life.abstraction;

import com.cbf4life.implementor.Product;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 我是山寨老大，你流行啥我就生产啥
 */
public class ShanZhaiCorp extends Corp {

    //产什么产品，不知道，等被调用的才知道
    public ShanZhaiCorp(Product product){
        super(product);
    }

    //狂赚钱
    public void makeMoney(){
        super.makeMoney();
        System.out.println("我赚钱呀...");
    }
}

```

HouseCorp 类和 ShanZhaiCorp 类的区别是在有参构造的参数类型上，HouseCorp 类比较明确，我就是只要 House 类，所以直接定义传递进来的必须是 House 类，一个类尽可能少的承担职责，那方法也是一样，既然 HouseCorp 类已经非常明确只生产 House 产品，那为什么不定义成 House 类型呢？ShanZhaiCorp 就不同了，它是确定不了生产什么类型。

好了，两大对应的阵营都已经产生了，那我们再看 Client 程序：

```
package com.cbf4life;

import com.cbf4life.abstraction.HouseCorp;
import com.cbf4life.abstraction.ShanZhaiCorp;
import com.cbf4life.implementor.Clothes;
import com.cbf4life.implementor.House;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 我要关心我自己的公司了
 */
public class Client {

    public static void main(String[] args) {

        House house = new House();
        System.out.println("-----房地产公司是这个样子运行的-----");
        //先找到我的公司
        HouseCorp houseCorp =new HouseCorp(house);
        //看我怎么挣钱
        houseCorp.makeMoney();
        System.out.println("\n");

        //山寨公司生产的产品很多，不过我只要指定产品就成了
        System.out.println("-----山寨公司是这样运行的-----");
        ShanZhaiCorp shanZhaiCorp = new ShanZhaiCorp(new Clothes());
        shanZhaiCorp.makeMoney();

    }
}
```

运行结果如下：

```
-----房地产公司是这个样子运行的-----
生产出的房子是这个样子的...
生产出的房子卖出去了...
房地产公司赚大钱了...

-----山寨公司是这样运行的-----
```

生产出的衣服是这个样子的...

生产出的衣服卖出去了...

我赚钱呀...

这个山寨公司的前身是生产衣服的，那我现在要修改一下，生产 iPod，看如下的变化：

```
package com.cbf4life;

import com.cbf4life.abstraction.HouseCorp;
import com.cbf4life.abstraction.ShanZhaiCorp;
import com.cbf4life.implementor.House;
import com.cbf4life.implementor.iPod;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 我要关心我自己的公司了
 */
public class Client {

    public static void main(String[] args) {
        House house = new House();
        System.out.println("-----房地产公司是这个样子运行的-----");
        //先找到我的公司
        HouseCorp houseCorp = new HouseCorp(house);
        //看我怎么挣钱
        houseCorp.makeMoney();
        System.out.println("\n");

        //山寨公司生产的产品很多，不过我只要制定产品就成了
        System.out.println("-----山寨公司是这样运行的-----");
        //ShanZhaiCorp shanZhaiCorp = new ShanZhaiCorp(new Clothes());
        ShanZhaiCorp shanZhaiCorp = new ShanZhaiCorp(new iPod());
        shanZhaiCorp.makeMoney();
    }
}
```

运行结果如下：

-----房地产公司是这个样子运行的-----

生产出的房子是这个样子的...

生产出的房子卖出去了...

房地产公司赚大钱了...

-----山寨公司是这样运行的-----

生产出的iPod是这个样子的...

生产出的iPod卖出去了...

我赚钱呀...

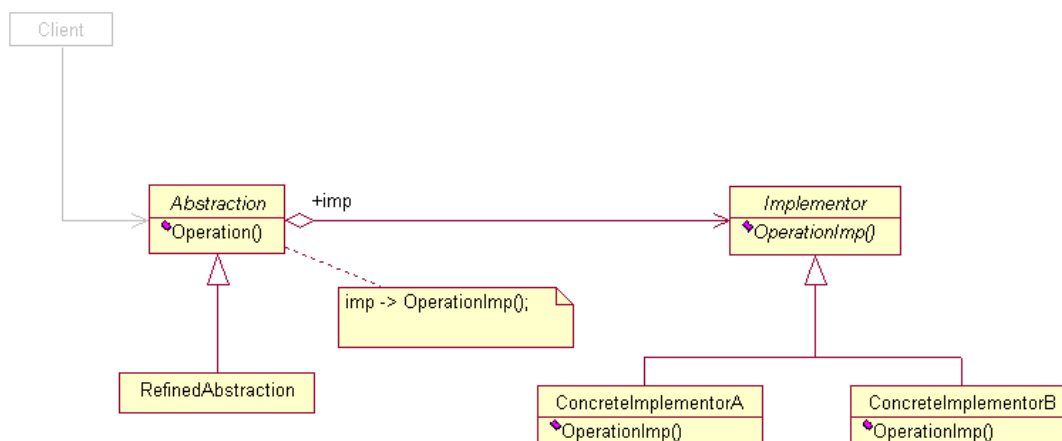
看代码上的黄色底色的代码，就修改了这一句话代码就完成了生产产品的转换。那我们深入的思考一下，既然万物都是运动的，我现在只有房地产公司和山寨公司，那以后我会不会增加一些其他的公司呢？或者房地产公司会不会对业务进行细化，比如分为公寓房公司，别墅公司，以及商业房公司等等呢？那我告诉你，会的，绝对的会的，但是你发觉没，这种变化对我们上面的类图没有大的修改，充其量是扩展，你看呀：

增加公司，你要么继承 Corp 类，要么继承 HouseCorp 或 ShanZhaiCorp，不用再修改原有的类了；

增加产品，继承 Product 类，或者继承 House 类，你要把房子分为公寓房、别墅、商业用房等等；

你都是在扩展，唯一你要修改的就是 Client 类，你类都增加了哪能不修改调用呢，也就是说 Corp 类和 Product 类都可以自由的扩展，而不会对整个应用产生太大的变更，这就是桥梁模式。

为什么叫桥梁模式？我们看一下桥梁模式的通用类图：



看到中间那根带箭头的线了吗？是不是类似一个桥，连接了两个类？所以就叫桥梁模式。我们再把桥梁模式的几个概念熟悉一下，大家有没有注意到我把 Corp 类以及它的两个实现类放到了 Abstraction 包中，把 House 以及相关的三个实现类放到了 Implementor 包中，这两个包分别对应了桥梁模式的业务抽象角色（Abstraction）和业务实现角色（Implementor），这两个角色我估计没几个人能说的明白，特别是看了“四人帮”的书或者是那本非常有名的、比砖头还要厚的书，你会越看越糊涂，忒专业化，有点像看政府的红头文件，什么都说了，可好像又什么都没有说。这两个角色大家只要记住一句话就成：业务抽象角色引用业务实现角色，

或者说业务抽象角色的部分实现是由业务实现角色完成的，很简单，别想那么复杂了。

桥梁模式的优点就是类间解耦，我们上面已经提到，两个角色都可以自己的扩展下去，不会相互影响，这个也符合 OCP 原则。

今天说到桥梁模式，那就多扯几句，大家对类的继承有什么看法吗？继承的优点有很多，可以把公共的方法或属性抽取，父类封装共性，子类实现特性，这是继承的基本功能，缺点有没有？有，强关联关系，父类有个方法，你子类也必须有这个方法，是不可选择的，那这会带来扩展性的问题，我举个简单的例子来说明这个问题：Father 类有一个方法 A，Son 继承了这个方法，然后 GrandSon 也继承了这个方法，问题是突然有一天 Son 要重写父类的这个方法，他敢做吗？绝对不敢！GrandSon 可是要用从 Father 继承过来的方法 A，你修改了，那就要修改 Son 和 GrandSon 之间的关系，那这个风险就大了去。

今天讲的这个桥梁模式就是这一问题的解决方法，桥梁模式描述了类间弱关联关系，还说上面的那个例子，Fater 类完全可以把可能会变化的方法放出去，Son 子类要有这个方法很简答，桥梁搭过去，获得这个方法，GrandSon 也一样，即使你 Son 子类不想使用这个方法了，也没关系，对 GrandSon 不产生影响，他不是从你 Son 中继承来的方法！

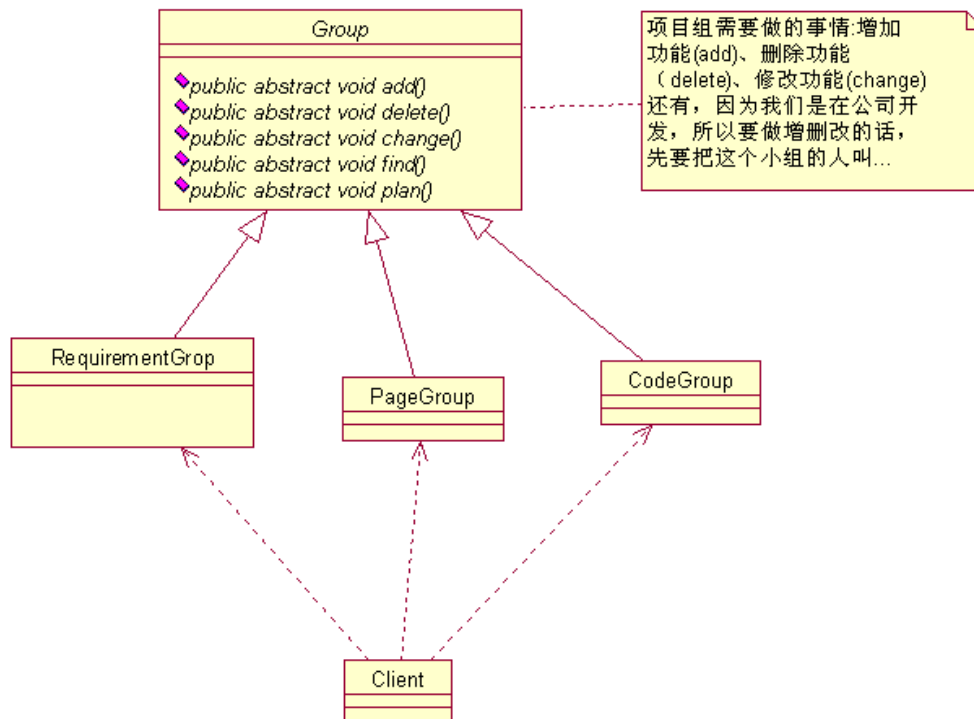
继承不能说它不好，非常好，但是有缺点的，我们可以扬长避短，对于比较明确不发生变化的，则通过继承来完成，若不能确定是否会发生变化的，那就认为是会发生变化，则通过桥梁模式来解决，这才是一个完美的世界。

第 12 章 命令模式【Command Pattern】

今天讲命令模式，这个模式从名字上看就很简单，命令嘛，老大发命令，小兵执行就是了，确实是这个意思，但是更深化了，用模式来描述真是世界的命令情况。正在看这本书的你，我猜测分为两类：已经工作的和没有工作的，先说没有工作的，那你为啥要看这本书，为了以后工作呗，只要你参见工作，你肯定会待在项目组，那今天我们就以项目组为例子来讲述命令模式。

我是我们部门的项目经理，就是一个项目的头，在中国做项目，项目经理就是什么都要懂，什么都要管，做好了项目经理能分到一杯羹，做不好都是你项目经理的责任，这个是绝对的，我带过太多的项目，行政命令一压下来，那就一条道，做完做好！我们虽然是一个集团公司，但是我们部门是独立核算的，就是说呀，我们部门不仅仅为我们集团服务，还可以为其他甲方服务，赚取更多的外快，所以俺们的工资才能是中上等。在 2007 年我带领了一个项目，比较小，但是钱可不少，是做什么的呢？为一家旅行社建立一套内部管理系统，管理他的客户、旅游资源、票务以及内部管理，整体上类似一个小型的 ERP 系统，门店比较多，员工也比较多，但是需求比较明确，因为他们之前有一套自己购买的内部管理系统，这次变动部分模块基本上是翻版，而且旅行社有自己的 IT 部门，比较好相处，都是技术人员，没有交流鸿沟嘛。

这个项目的成员分工也是采用了常规的分工方式，分为需求组（Requirement Group，简称 RG）、美工组（Page Group，简称 PG）、代码组（我们内部还有一个比较优雅的名字：逻辑实现组，这里使用大家经常称呼的名称吧，英文缩写叫 Code Group，简称 CG），总共加上我这个项目经理正好十个人，刚开始的时候客户（也就是旅行社，甲方）还是很乐意和我们每个组探讨，比如和需求组讨论需求，和美工讨论页面，和代码组讨论实现，告诉他们修改这里，删除这里，增加这些等等，这是一种比较常见的甲乙双方合作模式，甲方深入到乙方的项目开发中，我们把这个模式用类图表示一下：



这个类图很简单, 客户和三个组都有交流, 这也合情合理, 那我们看看这个的实现, 首先看抽象类, 我们是面向接口或抽象类编程的嘛:

```

package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 项目组分成了三个组, 每个组还是要接受增删改的命令
 */
public abstract class Group {

    //甲乙双方分开办公, 你要和那个组讨论, 你首先要找到这个组
    public abstract void find();

    //被要求增加功能
    public abstract void add();

    //被要求删除功能
    public abstract void delete();

    //被要求修改功能
    public abstract void change();
  
```

```
//被要求给出所有的变更计划
public abstract void plan();

}
```

大家看抽象类中的每个方法，是不是每个都是一个命令？找到它，增加，删除，给我计划！是不是命令，这也就是命令模式中的命令接收者角色(Receiver)，等会细讲。我们再看三个实现类，需求组最重要，没有需求你还设计个P呀，看 RequirementGroup 类的实现：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 需求组的职责是和客户谈定需求，这个组的人应该都是业务领域专家
 */
public class RequirementGroup extends Group {

    //客户要求需求组过去和他们谈
    public void find() {
        System.out.println("找到需求组...");
    }

    //客户要求增加一项需求
    public void add() {
        System.out.println("客户要求增加一项需求...");
    }

    //客户要求修改一项需求
    public void change() {
        System.out.println("客户要求修改一项需求...");
    }

    //客户要求删除一项需求
    public void delete() {
        System.out.println("客户要求删除一项需求...");
    }

    //客户要求出变更计划
    public void plan() {
        System.out.println("客户要求需求变更计划...");
    }
}
```

```
}  
  
}
```

需求组有了，我们再看美工组，美工组也很重要，是项目的脸面，客户最终接触到的还是界面，这个非常重要，看 PageGroup 的实现：

```
package com.cbf4life;  
  
/**  
 * @author cbf4Life cbf4life@126.com  
 * I'm glad to share my knowledge with you all.  
 * 美工组的职责是设计出一套漂亮、简单、便捷的界面  
 */  
public class PageGroup extends Group {  
  
    //首先这个美工组应该被找到吧，要不你跟谁谈？  
    public void find() {  
        System.out.println("找到美工组...");  
    }  
  
    //美工被要求增加一个页面  
    public void add() {  
        System.out.println("客户要求增加一个页面...");  
    }  
  
    //客户要求对现有界面做修改  
    public void change() {  
        System.out.println("客户要求修改一个页面...");  
    }  
  
    //甲方是老大，要求删除一些页面  
    public void delete() {  
        System.out.println("客户要求删除一个页面...");  
    }  
  
    //所有的增删改那要给出计划呀  
    public void plan() {  
        System.out.println("客户要求页面变更计划...");  
    }  
}
```

最后看代码组，这个组的成员一般都是比较闷骚型的，不多说话，但多做事儿，比较沉闷，这是这个组的典型特点，我们来看看这个 CodeGroup 类：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 代码组的职责是实现业务逻辑，当然包括数据库设计了
 */
public class CodeGroup extends Group {

    //客户要求代码组过去和他们谈
    public void find() {
        System.out.println("找到代码组...");
    }

    //客户要求增加一项功能
    public void add() {
        System.out.println("客户要求增加一项功能...");
    }

    //客户要求修改一项功能
    public void change() {
        System.out.println("客户要求修改一项功能...");
    }

    //客户要求删除一项功能
    public void delete() {
        System.out.println("客户要求删除一项功能...");
    }

    //客户要求出变更计划
    public void plan() {
        System.out.println("客户要求代码变更计划...");
    }
}
```

整个项目的三个支柱都已经产生了，那看客户怎么和我们谈。客户刚刚开始给了我们一份他们自己写的一份需求，还是比较完整的，需求组根据这份需求写了一份分析说明书，客户一看，不对，要增加点需求，看程序：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 客户就是甲方，给我们钱的一方，是老大
 */
public class Client {

    public static void main(String[] args) {

        //首先客户找到需求组说，过来谈需求，并修改
        System.out.println("-----客户要求增加一个需求-----");
        Group rg = new RequirementGroup();
        //找到需求组
        rg.find();

        //增加一个需求
        rg.add();

        //要求变更计划
        rg.plan();

    }
}
```

运行的结果如下：

```
-----客户要求增加一个需求-----
找到需求组...
客户要求增加一项需求...
客户要求需求变更计划...
```

好的，客户的需求达到了，需求刚开始没考虑周全，增加需求是在所难免的嘛，理解理解。然后又过了段时间，客户说“界面多画了一个，过来谈谈”，于是：

```
package com.cbf4life;

/**
```

```
* @author cbf4Life cbf4life@126.com
* I'm glad to share my knowledge with you all.
* 客户就是甲方，给我们钱的一方，是老大
*/
public class Client {

    public static void main(String[] args) {

        //首先客户找到美工组说，过来谈页面，并修改
        System.out.println("-----客户要求删除一个页面-----");
        Group pg = new PageGroup();
        //找到需求组
        pg.find();

        //增加一个需求
        pg.delete();

        //要求变更计划
        pg.plan();

    }
}
```

运行结果如下：

```
-----客户要求增加一个页面-----
找到美工组...
客户要求删除一个页面...
客户要求页面变更计划...
```

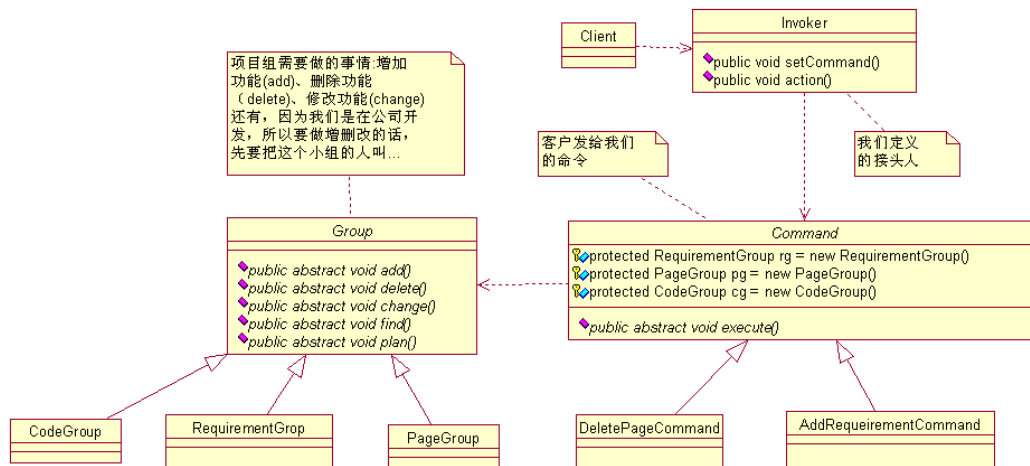
好了，界面也谈过了，应该没什么大问题了吧。过了一天后，客户又让代码组过去，说是数据库设计问题，然后又叫美工过去，布置了一堆命令，这个我就不一一写了，大家应该能够体会到，你做过项目的话，这种体会更深，客户让修改你不修改？项目不想做了你！

但是问题来了，我们修改可以，但是每次都是叫一个组去，布置个任务，然后出计划，次次都这样，如果让你当甲方也就是客户，你烦不烦？而且这种方式很容易出错误呀，而且还真发生过，客户把美工叫过去了，要删除，可美工说需求是这么写的，然后客户又命令需求组过去，一次次的折腾，客户也烦躁了，于是直接抓住我这个项目经理说：

“我不管你们内部怎么安排，你就给我找个接头人，我告诉他怎么做，删除页面了，增加功能了，你们内部

怎么处理，我就告诉他我要干什么就成了...”

我一听，好呀，这也正是我想要的，我项目组的兄弟们也已经受不了了，于是我改变了一下我的处理方式，看看类图：



类图中增加了不少，看着也比较清晰，比较简单的（Command 抽象类与 Group 抽象类是没有关联关系的，与 Group 的三个实现类是有关联关系，为了线条不交叉就直接画上父类有关系），增加的几个类说明如下：

Command 抽象类：客户发给我们的命令，定义三个工作组的成员变量，供子类使用；定义一个抽象方法 execute，由子类来实现；

Invoker 实现类：项目接头人，setComand 接受客户发给我我们的命令，action 方法是执行客户的命令（方法名写成是 action 是与 command 的 execute 区分开，避免混淆）

我们先看 Command 抽象类代码：

```

package com.cbf4life.command;

import com.cbf4life.receiver.CodeGroup;
import com.cbf4life.receiver.PageGroup;
import com.cbf4life.receiver.RequirementGroup;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 命令的抽象类，我们把客户发出的命令定义成一个一个的对象
 */
public abstract class Command {

    //把三个组都定义好，子类可以直接使用
    protected RequirementGroup rg = new RequirementGroup(); //需求组
    
```

```

    protected PageGroup pg = new PageGroup(); //美工组
    protected CodeGroup cg = new CodeGroup(); //代码组;

    //只要一个方法，你要我做什么事情
    public abstract void execute();

}

```

这个简单，看两个具体的实现类，先看 AddRequeirementCommand 类，这个类的作用就是增加一项需求。

```

package com.cbf4life.command;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 增加一项需求
 */
public class AddRequirementCommand extends Command {
    //执行增加一项需求的命令
    public void execute() {
        //找到需求组
        super.rg.find();

        //增加一份需求
        super.rg.add();

        //给出计划
        super.rg.plan();
    }
}

```

看删除一个页面的命令,DeletePageCommand 类:

```

package com.cbf4life.command;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 删除一个页面的命令
 */

```



```
public class DeletePageCommand extends Command {

    //执行删除一个页面的命令
    public void execute() {
        //找到页面组
        super.pg.find();

        //删除一个页面
        super.rg.delete();

        //给出计划
        super.rg.plan();
    }
}
```

Command 抽象类还可以有很多的子类，比如增加一个功能命令（AddCodeCommand），删除一份需求命令（DeleteRequirementCommand）等等，这里就不用描述了，都很简单。

我们再看我们的接头人，就是 Invoker 类的实现：

```
package com.cbf4life.invoker;

import com.cbf4life.command.Command;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 接头人的职责就是接收命令，并执行
 */
public class Invoker {
    //什么命令
    private Command command;

    //客户发出命令
    public void setCommand(Command command){
        this.command = command;
    }

    //执行客户的命令
    public void action(){
        this.command.execute();
    }
}
```

```
}
```

这个是更简单了，简单是简单，可以帮我们解决很多问题，我们再看一下客户提出变更的过程：

```
package com.cbf4life;

import com.cbf4life.command.AddRequirementCommand;
import com.cbf4life.command.Command;
import com.cbf4life.invoker.Invoker;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 客户就是甲方，给我们钱的一方，是老大
 */
public class Client {

    public static void main(String[] args) {
        //定义我们的接头人
        Invoker xiaoSan = new Invoker(); //接头人就是我小三

        //客户要求增加一项需求
        System.out.println("-----客户要求增加一项需求-----");
        //客户给我们下命令来
        Command command = new AddRequirementCommand();

        //接头人接收到命令
        xiaoSan.setCommand(command);

        //接头人执行命令
        xiaoSan.action();
    }
}
```

运行结果如下：

```
-----客户要求增加一项需求-----
找到需求组...
客户要求增加一项需求...
客户要求需求变更计划...
```

那我们看看，如果客户要求删除一个页面，那我们的修改有多大呢？想想，Look：

```
package com.cbf4life;

import com.cbf4life.command.Command;
import com.cbf4life.command.DeletePageCommand;
import com.cbf4life.invoker.Invoker;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 客户就是甲方，给我们钱的一方，是老大
 */
public class Client {

    public static void main(String[] args) {
        //定义我们的接头人
        Invoker xiaoSan = new Invoker(); //接头人就是我小三

        //客户要求增加一项需求
        System.out.println("-----客户要求删除一个页面-----");
        //客户给我们下命令来
        //Command command = new AddRequirementCommand();
        Command command = new DeletePageCommand();

        //接头人接收到命令
        xiaoSan.setCommand(command);

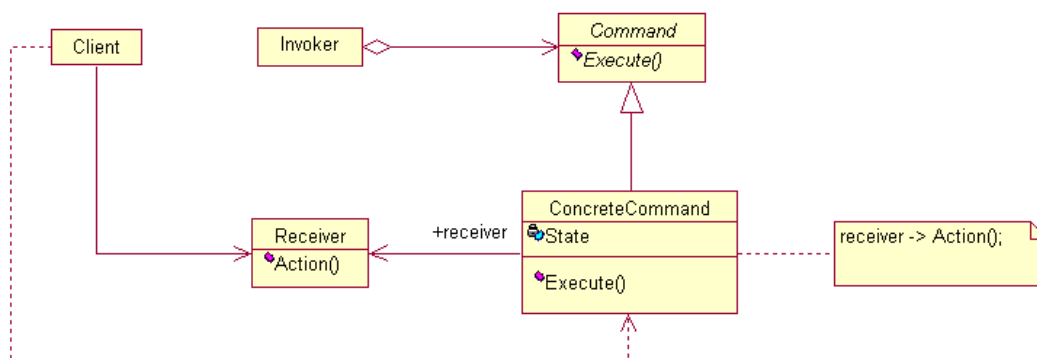
        //接头人执行命令
        xiaoSan.action();
    }
}
```

运行结果如下：

```
-----客户要求删除一个页面-----
找到美工组...
客户要求删除一项需求...
客户要求需求变更计划...
```

看到上面打黄色的代码来吗？就修改了这么多，就完成了命令的，是不是很简单，而且客户也不用知道到底要谁来修改，这个他不需要知道的，高内聚的要求体现出来了，这就是命令模式。

命令模式的通用类图如下：



在这个类图中，我们看到三个角色：

Receiver 角色：这个就是干活的角色，命令传递到这里是应该被执行的，具体到上面我们的例子中就是 Group 的三个实现类；

Command 角色：就是命令，需要我执行的所有命令都在这里声明；

Invoker 角色：调用者，接收到命令，并执行命令，例子中我这里项目经理就是这个角色；

命令模式比较简单，但是在项目中使用是非常频繁的，封装性非常好，因为它把请求方（Invoker）和执行方（Receiver）分开了，扩展性也有很好的保障。但是，命令模式也是有缺点的，你看 Command 的子类没有，那个如果我要写下去的可不是几个，而是几十个，这个类膨胀的非常多，这个就需要大家在项目中自己考虑使用了。

上面的例子我还没有说完，我们想想，客户要求增加一项需求，那是不是页面也增加，同时功能也要增加呢？如果不使用命令模式，客户就需要先找需求组，然后找美工组，然后找代码组，这个...，你想让客户跳楼呀！使用命令模式后，客户只管发命令模式，你增加一项需求，没问题，我内部调动三个组通力合作，然后反馈你结果，这也正是客户需要的，那这个要怎么修改呢？想想看，很简单的，在 `AddRequirementCommand` 类的 `execute` 方法中增加对 `PageGroup` 和 `CodePage` 的调用就成了，修改后的代码如下：

```
package com.cbf4life.command;
```

```
/**
```

```
 * @author cbf4Life cbf4life@126.com
```

```
* I'm glad to share my knowledge with you all.
* 增加一项需求
*/
public class AddRequirementCommand extends Command {
    // 执行增加一项需求的命令
    public void execute() {
        // 找到需求组
        super.rg.find();

        // 增加一份需求
        super.rg.add();

        // 页面也要增加
        super.pg.add();

        // 功能也要增加
        super.cg.add();

        // 给出计划
        super.rg.plan();
    }
}
```

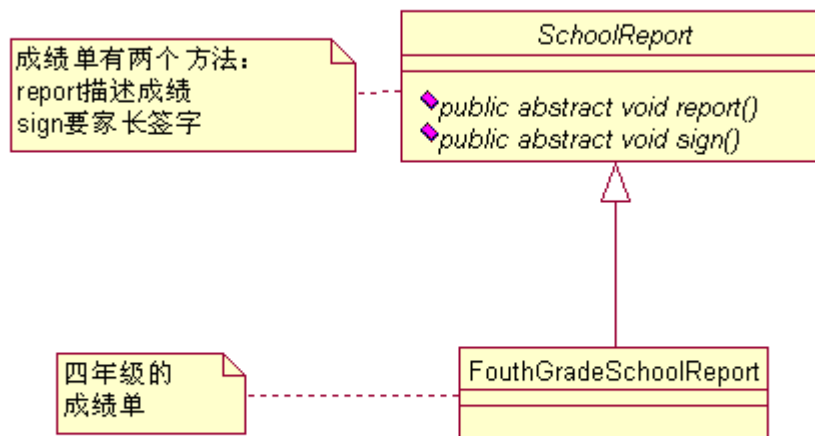
看看，是不是就解决问题了？命令模式做了一层非常好的封装。那还有一个问题需要大家考虑：客户发出命令，那要是撤回怎么办？就类似你使用Ctrl+Z组合键（undo 功能），发出一个命令，在没有执行或执行后撤回（执行后撤回是状态变更）该怎么实现呢？想想看，简单，非常简单，undo 也是一个命令嘛！

第 13 章 装饰模式【Decorator Pattern】

Ladies and gentlemen, May I get your attention, Please?, Now I' m going to talk about decorator pattern. 装饰模式在中国使用的那实在是多，中国的文化是中庸文化，说话或做事情都不能太直接，需要有技巧的，比如说话吧，你要批评一个人，你不能一上来就说你这个做的不对，那个做的不对，你要先肯定他的成绩，表扬一下优点，然后再指出瑕疵，指出错误的地方，最后再来个激励，你修改了这些缺点后有那些好处，比如你能带更多的小兵，到个小头目等等，否则你一上来就是一顿批评，你瞅瞅看，肯定是不服气，顶撞甚至是直接“此处不养爷，自有养爷处”开溜哇。这是说话，那做事情也有很多，在山寨产品流行之前，假货很是比较盛行的，我在 2002 年买了个手机，当时老板吹的是天花乱坠，承诺这个手机是最新的，我看着也像，壳子是崭新的，包装是崭新的，没有任何瑕疵，就是比正品便宜了一大截，然后我买了，缺钱哪，用来 3 个月，坏了，一送修，检查，说这是个新壳装旧机，我晕！拿一个旧手机的线路板，找个新的外壳、屏幕、包装就成了新手机，装饰模式害人不浅呀！

我们不说不开心的事情，今天举一个什么例子呢？就说说我上小学的糗事吧。我上小学的时候学习成绩非常的差，班级上 40 多个同学，我基本上都是在排名 45 名以后，按照老师给我的定义就是“不是读书的料”，但是我老爸管的很严格，明知道我不是这块料，还是往赶鸭子上架，每次考试完毕我都是战战兢兢的，“竹笋炒肉”是肯定少不了的，能少点就少点吧，肉可是自己的呀。四年级期末考试考完，学校出来个很损的招儿（这招儿现在很流行的），打印出成绩单，要家长签字，然后才能上五年级，我那个恐惧呀，不过也就是几秒钟的时间，玩起来什么都忘记了。

我们先看看这个成绩单的类图：



成绩单的抽象类，然后有一个四年级的成绩单实现类，先看抽象类：

```

package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 成绩单的抽象类
 */
public abstract class SchoolReport {

    //成绩单的主要展示的就是你的成绩情况
    public abstract void report();

    //成绩单要家长签字，这个是最要命的
    public abstract void sign();
}
  
```

然后看我们的实现类 FouthGradSchoolReport：

```

package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
  
```

```

* 四年级的成绩单，这个是我们学校第一次实施，以前没有干过
* 这种“缺德”事。
*/
public class FouthGradeSchoolReport extends SchoolReport {

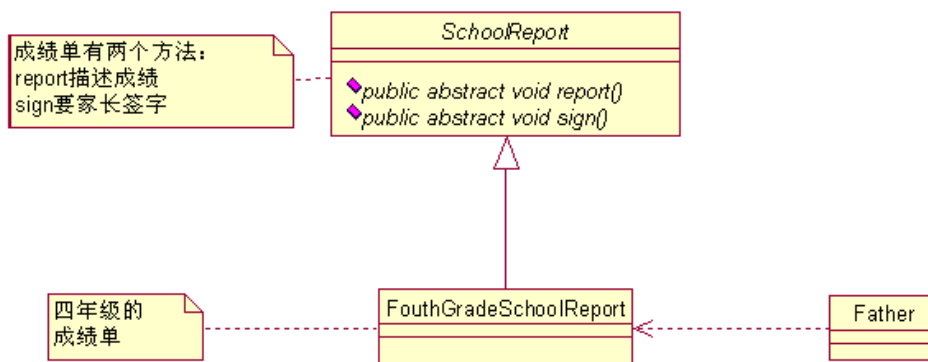
    //我的成绩单
    public void report() {
        //成绩单的格式是这个样子的
        System.out.println("尊敬的xxx家长:");
        System.out.println("    .....");
        System.out.println("  语文 62  数学65  体育 98  自然  63");
        System.out.println("    .....");
        System.out.println("                    家长签名:      ");
    }

    //家长签名
    public void sign(String name) {
        System.out.println("家长签名为: "+name);
    }

}

```

成绩单出来，你别看什么 62，65 之类的成绩，你要知道在小学低于 90 分基本上就是中下等了，唉，爱学习的人太多了！怎么着，那我把这个成绩单给老爸看看？好，我们修改一下类图，成绩单给老爸看：



老爸开始看成绩单，这个成绩单可是最真实的，啥都没有动过，原装，看 Father 类：


```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 老爸看成绩单了
 */
public class Father {

    public static void main(String[] args) {
        //成绩单拿过来
        SchoolReport sr = new FouthGradeSchoolReport();

        //看成绩单
        sr.report();

        //签名? 休想!
    }
}
```

运行结果如下:

尊敬的xxx家长:

.....

语文 62 数学65 体育 98 自然 63

.....

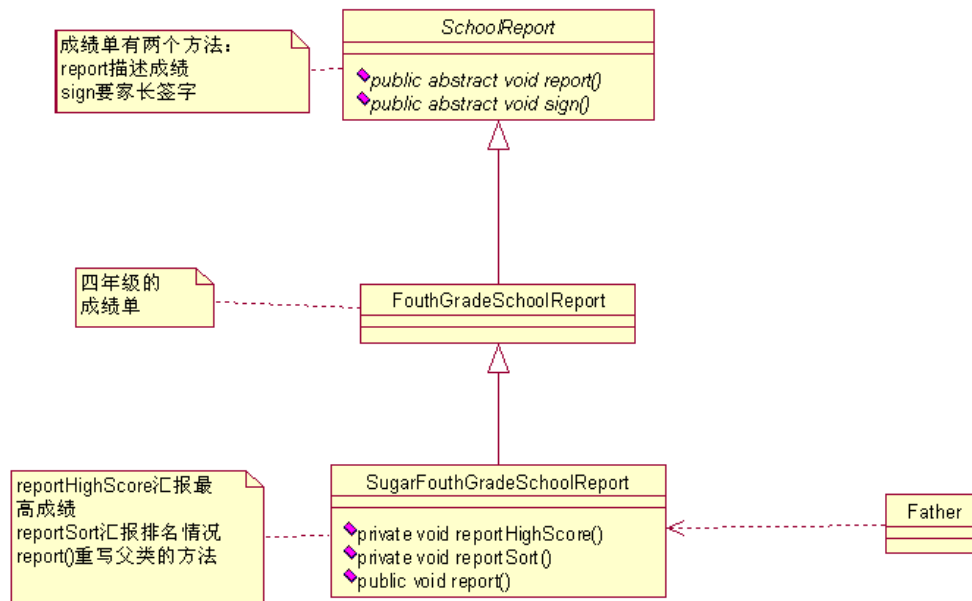
家长签名:

就这成绩还要我签字?! 老爸就开始找笞帚, 我的屁股已经做好了准备, 肌肉要绷紧, 要不那个太疼了! 哈哈, 幸运的是, 这个不是当时的真实情况, 我没有直接把成绩单交给老爸, 而是在交给他之前做了点技术工作, 我要把成绩单封装一下, 封装分类两步走:

第一步:跟老爸说各个科目的最高分, 语文最高是 75, 数学是 78, 自然是 80, 然老爸觉的我成绩与最高分数相差不多, 这个是实情, 但是不知道是什么原因, 反正期末考试都考的不怎么样, 但是基本上都集中在 70 分以上, 我这 60 多分基本上还是垫底的角色;

第二步: 在老爸看成绩单后, 告诉他我是排名第 38 名, 全班, 这个也是实情, 为啥呢? 有将近十个同学退学了! 这个情况我是不说的。不知道是不是当时第一次发成绩单, 学校没有考虑清楚, 没有写上总共有多少同学, 排名第几名等等, 反正是被我钻了个空子。

那修饰是说完了, 我们看看类图如何修改:



我想这是你最容易想到的类图，通过直接增加了一个子类，重写 report 方法，很容易的解决了这个问题，是不是这样？是的，确实是，确实是一个很好的办法，我们来看具体的实现：

```

package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 对这个成绩单进行美化
 * Sugar这个词太好了，名词是糖的意思，动词就是美化
 * 给你颗糖你还不美去
 */
public class SugarFouthGradeSchoolReport extends FouthGradeSchoolReport {

    //首先要定义你要美化的方法，先给老爸说学校最高成绩
    private void reportHighScore(){
        System.out.println("这次考试语文最高是75，数学是78，自然是80");
    }

    //在老爸看完成绩单后，我再汇报学校的排名情况
    private void reportSort(){
        System.out.println("我是排名第38名...");
    }

    //由于汇报的内容已经发生变更，那所以要重写父类
    @Override
    public void report(){
  
```

```

        this.reportHighScore(); //先说最高成绩
        super.report(); //然后老爸看成绩单
        this.reportSort(); //然后告诉老爸学习学校排名
    }
}

```

然后 Father 类稍做修改就可以看到美化后的成绩单，看代码如下：

```

package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 老爸看成绩单了
 */
public class Father {

    public static void main(String[] args) {
        //美化过的成绩单拿过来
        SchoolReport sr= new SugarFouthGradeSchoolReport();

        //看成绩单
        sr.report();

        //然后老爸，一看，很开心，就签名了
        sr.sign("老三"); //我叫小三，老爸当然叫老三
    }
}

```

运行结果如下：

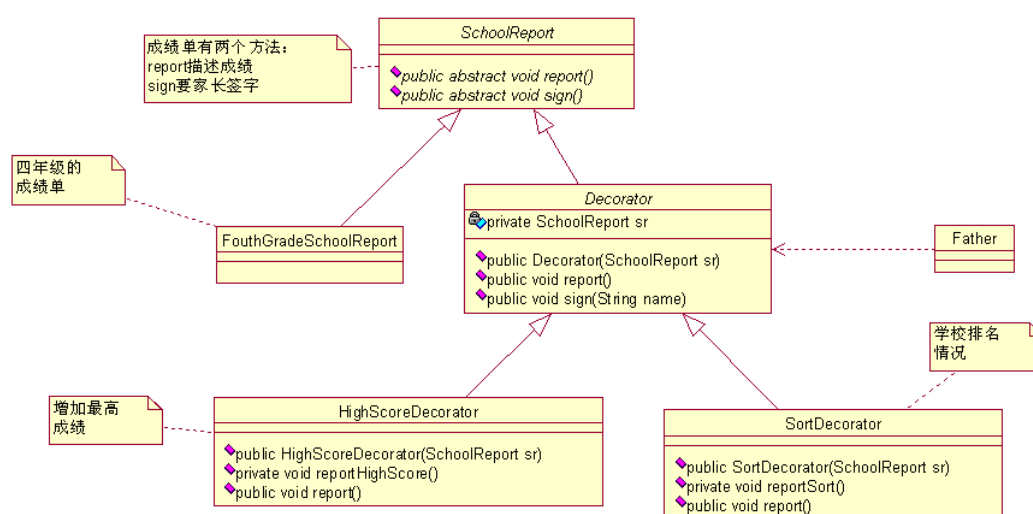
```

这次考试语文最高是75，数学是78，自然是80
尊敬的xxx家长：
.....
语文 62  数学65  体育 98  自然  63
.....
                家长签名：
我是排名第38名...
家长签名为：老三

```

通过继承确实能够解决这个问题，老爸看成绩单很开心，然后就给签字了，但是现实的情况很复杂的，可能老爸听我汇报最高成绩后，就直接乐开花了，直接签名了，后面的排名就没必要了，或者老爸要先听排名情况，那怎么办？继续扩展类？你能扩展多少个类？这还是一个比较简单的场景，一旦需要装饰的条件非常的多，比如 20 个，你还通过继承来解决，你想想的子类有多少个？你是不是马上就要崩溃了！

好，你也看到通过继承情况确实出现了问题，类爆炸，类的数量激增，光写这些类不累死你才怪，而且还要想想以后维护怎么办，谁愿意接收这么一大堆类的维护哪？并且在面向对象的设计中，如果超过 2 层继承，你应该想想是不是出设计问题了，是不是应该重新找一条道了，这是经验值，不是什么绝对的，继承层次越多你以后的维护成本越多，问题这么多，那怎么办？好办，装饰模式出场来解决这些问题，我们先来看类图：



增加一个抽象类和两个实现类，其中 **Decorator** 的作用是封装 **SchoolReport** 类，看源代码：

```

package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 装饰类，我要把我的成绩单装饰一下
 */
public abstract class Decorator extends SchoolReport {

    //首先我要知道是那个成绩单
    private SchoolReport sr;

    //构造函数，传递成绩单过来
    public Decorator(SchoolReport sr) {

```

```

        this.sr = sr;
    }

    //成绩单还是要被看到的
    public void report(){
        this.sr.report();
    }

    //看完毕还是要签名的
    public void sign(String name){
        this.sr.sign(name);
    }
}

```

Decorator 抽象类的目的很简单，就是要让子类来对封装 SchoolReport 的子类，怎么封装？重写 report 方法！先看 HighScoreDecorator 实现类：

```

package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 我要把我学校的最高成绩告诉老爸
 */
public class HighScoreDecorator extends Decorator {

    //构造函数
    public HighScoreDecorator(SchoolReport sr){
        super(sr);
    }

    //我要汇报最高成绩
    private void reportHighScore(){
        System.out.println("这次考试语文最高是75，数学是78，自然是80");
    }

    //最高成绩我要做老爸看成绩单前告诉他，否则等他一看，就抡起笤帚有揍我，我那还有机会说呀
    @Override
    public void report(){
        this.reportHighScore();
        super.report();
    }
}

```

```

    }
}

```

重写了 `report` 方法，先调用具体装饰类的装饰方法 `reportHighScore`，然后再调用具体构件的方法，我们再来看怎么回报学校排序情况 `SortDecorator` 代码：

```

package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 学校排名的情况汇报
 */
public class SortDecorator extends Decorator {

    //构造函数
    public SortDecorator(SchoolReport sr){
        super(sr);
    }

    //告诉老爸学校的排名情况
    private void reportSort(){
        System.out.println("我是排名第38名...");
    }

    //老爸看成绩单后再告诉他，加强作用
    @Override
    public void report(){
        super.report();
        this.reportSort();
    }
}

```

然后看看我老爸怎么看成绩单的：

```

package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 老爸看成绩单了

```

```

*/
public class Father {

    public static void main(String[] args) {
        //成绩单拿过来
        SchoolReport sr;
        sr = new FouthGradeSchoolReport(); //原装的成绩单

        //加 了最高分说明的成绩单
        sr = new HighScoreDecorator(sr);

        //又加了成绩排名的说明
        sr = new SortDecorator(sr);

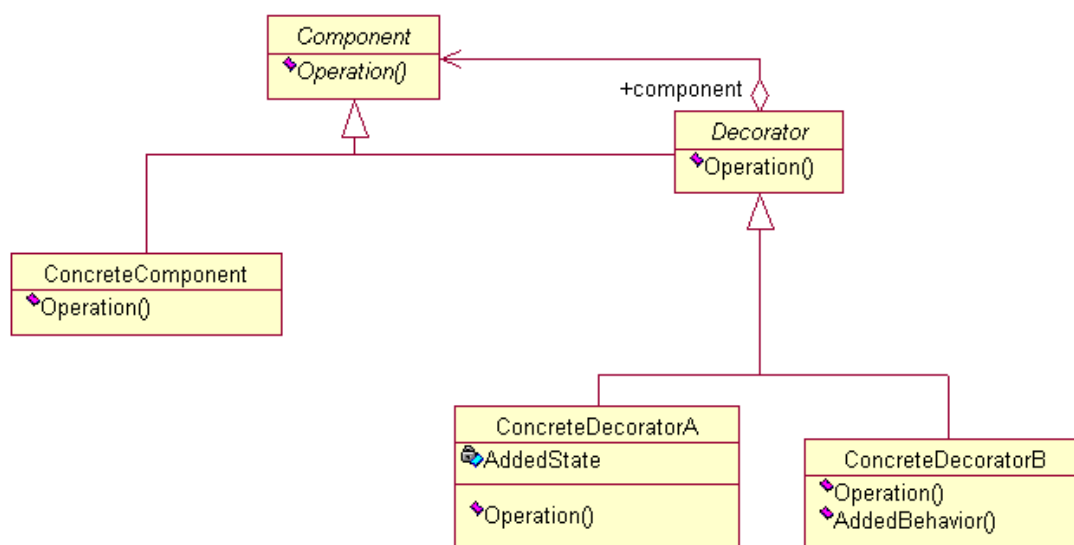
        //看成绩单
        sr.report();

        //然后老爸，一看，很开心，就签名了
        sr.sign("老三"); //我叫小三，老爸当然叫老三
    }
}

```

老爸一看成绩单，听我这么一说，非常开心，儿子有进步呀，从 40 多名进步到 30 多名，进步很大，躲过了一顿海扁。

这就是装饰模式，装饰模式的通用类图如下：



看类图，Component 是一个接口或者是抽象类，就是定义我们最核心的对象，也就是最原始的对象，比如上面的成绩单，记住在装饰模式中，必然有一个被提取出来最核心、最原始、最基本的接口或抽象类，

就是 Component。

ConcreteComponent 这个事最核心、最原始、最基本的接口或抽象类的实现，你要装饰的就是这个东东。

Decorator 一般是一个抽象类，做什么用呢？实现接口或者抽象方法，它里面可不一定有抽象的方法呀，在它的属性里必然有一个 private 变量指向 Component。

ConcreteDecoratorA 和 ConcreteDecoratorB 是两个具体的装饰类，你要把你最核心的、最原始的、最基本的东西装饰成啥东西，上面的例子就是把一个比较平庸的成绩单装饰成家长认可的成绩单。

装饰模式是对继承的有力补充，你要知道继承可不是万能的，继承可以解决实际的问题，但是在项目中你要考虑诸如易维护、易扩展、易复用等，而且在一些情况下（比如上面那个成绩单例子）你要是用继承就会增加很多类，而且灵活性非常的差，那当然维护也不容易了，也就是说装饰模式可以替代继承，解决我们类膨胀的问题，你要知道继承是静态的给类增加功能，而装饰模式则是动态的给增加功能，你看上面的那个例子，我不想要 SortDecorator 这层的封装也很简单呀，直接在 Father 中去掉就可以了，如果你用继承就必须修改程序。

装饰模式还有一个非常好的优点，扩展性非常好，在一个项目中，你会有非常多因素考虑不到，特别是业务的变更，时不时的冒出一个需求，特别是提出一个令项目大量延迟的需求时候，那种心情是…，真想骂娘！装饰模式可以给我们很好的帮助，通过装饰模式重新封装一个类，而不是通过继承来完成，简单点说，三个继承关系 Father, Son, GrandSon 三个类，我要再 Son 类上增强一些功能怎么办？我想你会坚决的顶回去！不允许，对了，为什么呢？你增强的功能是修改 Son 类中的方法吗？增加方法吗？对 GrandSon 的影响哪？特别是 GrandSon 有多个的情况，你怎么办？这个评估的工作量就是够你受的，所以这个是不允许的，那还是要解决问题的呀，怎么办？通过建立 SonDecorator 类来修饰 Son，等于说是创建了一个新的类，这个对原有程序没有变更，通过扩充很好的完成了这次变更。

第 14 章 迭代器模式【Iterator Pattern】

周五下午，我正在看技术网站，第六感官发觉有人在身后，扭头一看，我 c，老大站在背后，赶忙站起来，

“王经理，你找我？”我说。

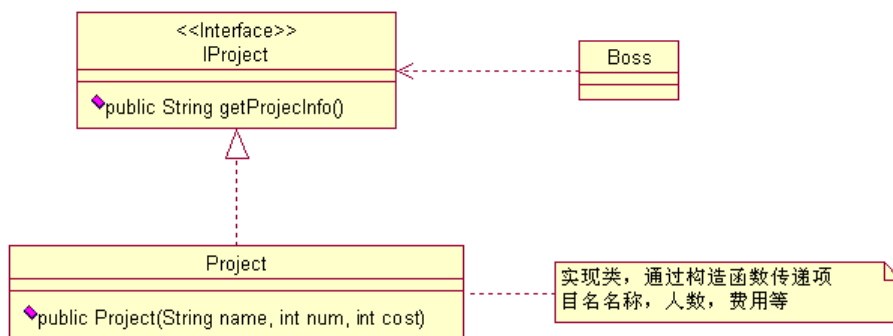
“哦，在看技术呀。有个事情找你谈一下，你到我办公室来一下。”老大说。

到老大办公室，

“是这样，刚刚我在看季报，我们每个项目的支出费用都很高，项目情况复杂，人员情况也不简单，我看着也有点糊涂，你看，这是我们现在还在开发或者维护的 103 个项目，你能不能先把这些项目信息重新打印一份给我，咱们好查查到底有什么问题。”老大说。

“这个好办，我马上去办”我爽快的答复道。

很快我设计了一个类图，并开始实施：



类图非常简单，是个程序员都能实现，我们来看看简单的东西：

```
package com.cbf4life;
```

```
/**
```

```
 * @author cbf4Life cbf4life@126.com
```

```
 * I'm glad to share my knowledge with you all.
```

```
 * 定义一个接口，所有的项目都是一个接口
```

```
 */
```

```
public interface IProject {
```

```
    //从老板这里看到的就项目信息
```

```
    public String getProjectInfo();
```

```
}
```

定义了一个接口，面向接口编程嘛，当然要定义接口了，然后看看实现类：

```
package com.cbf4life;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 所有项目的信息类
 */
public class Project implements IProject {
    //项目名称
    private String name = "";

    //项目成员数量
    private int num = 0;

    //项目费用
    private int cost = 0;

    //定义一个构造函数，把所有老板需要看到的信息存储起来
    public Project(String name,int num,int cost){
        //赋值到类的成员变量中
        this.name = name;
        this.num = num;
        this.cost=cost;
    }

    //得到项目的信息
    public String getProjectInfo() {
        String info = "";

        //获得项目的名称
        info = info+ "项目名称是: " + this.name;
        //获得项目人数
        info = info + "\t项目人数: "+ this.num;
        //项目费用
        info = info+ "\t 项目费用: "+ this.cost;

        return info;
    }
}
```

实现类也是比较简单的，通过构造函数传递过来要显示的数据，然后放到 `getProjectInfo` 中显示，这太 easy 了！，然后我们老大要看看结果了：

```
package com.cbf4life;

import java.util.ArrayList;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 老板来看项目信息了
 */
public class Boss {

    public static void main(String[] args) {
        //定义一个List，存放所有的项目对象
        ArrayList<IProject> projectList = new ArrayList<IProject>();

        //增加星球大战项目
        projectList.add(new Project("星球大战项目",10,100000));
        //增加扭转时空项目
        projectList.add(new Project("扭转时空项目",100,10000000));
        //增加超人改造项目
        projectList.add(new Project("超人改造项目",10000,1000000000));

        //这边100个项目
        for(int i=4;i<104;i++){
            projectList.add(new Project("第"+i+"个项目",i*5,i*1000000));
        }

        //遍历一下ArrayList，把所有的数据都取出
        for(IProject project:projectList){
            System.out.println(project.getProjectInfo());
        }
    }
}
```

然后看一下我们的运行结果：

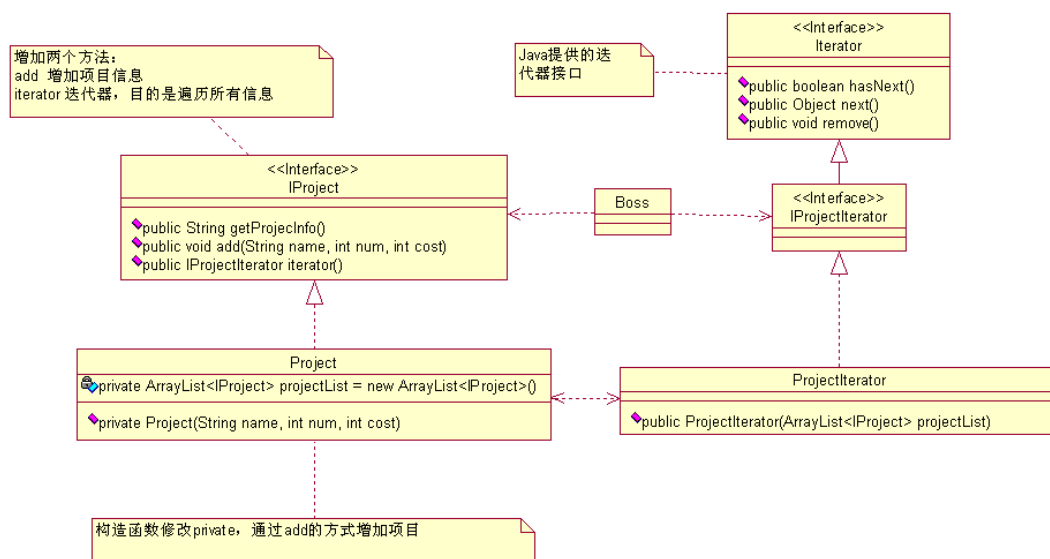
```
项目名称是：星球大战项目 项目人数： 10    项目费用： 100000
```

```

项目名称是：扭转时空项目 项目人数：100    项目费用：10000000
项目名称是：超人改造项目 项目人数：10000  项目费用：1000000000
项目名称是：第4个项目    项目人数：20     项目费用：4000000
项目名称是：第5个项目    项目人数：25     项目费用：5000000
.
.
.

```

老大一看，非常 Happy，这么快就出结果了，大大的把我夸奖了一番，然后就去埋头去研究那堆枯燥的报表了，然后我回到座位上，又看了一遍程序（心里很乐，就又想看看自己的成果），想了想，应该还有另外一种实现方式，因为是遍历嘛，让我想到的就是迭代器模式，我先把类图画出来：



看着是不是复杂了很多？是的，是有点复杂了，这个我等会说明原因，我们看代码实现，先 IProject 接口：

```

package com.cbf4life.pattern;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 定义一个接口，所有的项目都是一个接口
 */
public interface IProject {

    //增加项目
    public void add(String name,int num,int cost);
}

```

```
//从老板这里看到的就是项目信息
public String getProjectInfo();

//获得一个可以被遍历的对象
public IProjectIterator iterator();
}
```

这里多了两个方法，一个是 add 方法，这个方法是增加项目，也就是说产生了一个对象后，直接使用 add 方法增加项目信息。我们再来看实现类：

```
package com.cbf4life.pattern;

import java.util.ArrayList;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 所有项目的信息类
 */
@SuppressWarnings("all")
public class Project implements IProject {
    //定义一个项目列表，说有的项目都放在这里
    private ArrayList<IProject> projectList = new ArrayList<IProject>();

    //项目名称
    private String name = "";

    //项目成员数量
    private int num = 0;

    //项目费用
    private int cost = 0;

    public Project(){

    }

    //定义一个构造函数，把所有老板需要看到的信息存储起来
    private Project(String name,int num,int cost){
        //赋值到类的成员变量中
        this.name = name;
        this.num = num;
    }
}
```

```

        this.cost=cost;
    }

    //增加项目
    public void add(String name,int num,int cost){
        this.projectList.add(new Project(name,num,cost));
    }

    //得到项目的信息
    public String getProjectInfo() {
        String info = "";

        //获得项目的名称
        info = info+ "项目名称是: " + this.name;
        //获得项目人数
        info = info + "\t项目人数: "+ this.num;
        //项目费用
        info = info+ "\t 项目费用: "+ this.cost;

        return info;
    }

    //产生一个遍历对象
    public IProjectIterator iterator(){
        return new ProjectIterator(this.projectList);
    }
}

```

项目信息类已经产生，我们再来看看我们的迭代器是如何实现的，先看接口：

```

package com.cbf4life.pattern;

import java.util.Iterator;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 定义个Iterator接口
 */
@SuppressWarnings("all")
public interface IProjectIterator extends Iterator {

```

```
}
```

大家可能很奇怪，你定义的这个接口方法、变量都没有，有什么意义呢？有意义，所有的 Java 书上都一直说是面向接口编程，你的接口是对一个事物的描述，也就是说我通过接口就知道这个事物有哪些方法，哪些属性，我们这里的 `IProjectIterator` 是要建立一个指向 `Project` 类的迭代器，目前暂时定义的就是一个通用的迭代器，可能以后会增加 `IProjectIterator` 的一些属性或者方法。当然了，你也可以在实现类上实现两个接口，一个是 `Iterator`，一个是 `IProjectIterator`（这时候，这个接口就不用继承 `Iterator`），杀猪杀尾巴，各有各的杀发。我的习惯是：如果我要实现一个容器或者其他 API 提供接口时，我一般都自己先写一个接口继承，然后再继承自己写的接口，保证自己的实现类只用实现自己写的接口（接口传递，当然也要实现顶层的接口），程序阅读也清晰一些。我们继续看迭代器的实现类：

```
package com.cbf4life.pattern;

import java.util.ArrayList;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 定义一个迭代器
 */
public class ProjectIterator implements IProjectIterator {

    //所有的项目都放在这里ArrayList中
    private ArrayList<IProject> projectList = new ArrayList<IProject>();

    private int currentItem = 0;

    //构造函数传入projectList
    public ProjectIterator(ArrayList<IProject> projectList){
        this.projectList = projectList;
    }

    //判断是否还有元素，必须实现
    public boolean hasNext() {
        //定义一个返回值
        boolean b = true;
        if(this.currentItem>=projectList.size() ||
this.projectList.get(this.currentItem) == null){
            b =false;
        }
    }
}
```

```
    }  
    return b;  
}  
  
//取得下一个值  
public IProject next() {  
    return (IProject) this.projectList.get(this.currentItem++);  
}  
  
//删除一个对象  
public void remove() {  
    //暂时没有使用到  
}  
  
}
```

都写完毕了，然后看看我们的 Boss 类有多少改动：

```
package com.cbf4life.pattern;  
  
/**  
 * @author cbf4Life cbf4life@126.com  
 * I'm glad to share my knowledge with you all.  
 * 老板来看项目信息了  
 */  
public class Boss {  
  
    public static void main(String[] args) {  
        //定义一个List，存放所有的项目对象  
        IProject project = new Project();  
  
        //增加星球大战项目  
        project.add("星球大战项目", 10, 100000);  
        //增加扭转时空项目  
        project.add("扭转时空项目", 100, 10000000);  
        //增加超人改造项目  
        project.add("超人改造项目", 10000, 1000000000);  
  
        //这边100个项目  
        for(int i=4; i<104; i++){  
            project.add("第"+i+"个项目", i*5, i*1000000);  
        }  
    }  
}
```



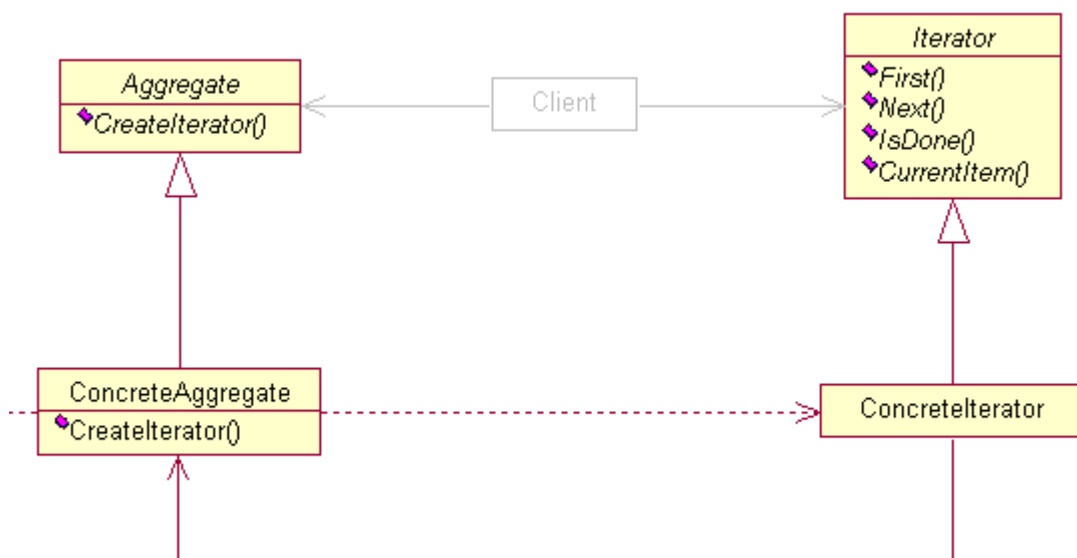
```
//遍历一下ArrayList，把所有的数据都取出
IProjectIterator projectIterator = project.iterator();
while(projectIterator.hasNext()){
    IProject p = (IProject)projectIterator.next();
    System.out.println(p.getProjectInfo());
}
}
```

运行结果如下：

```
项目名称是：星球大战项目 项目人数：10    项目费用：100000
项目名称是：扭转时空项目 项目人数：100    项目费用：10000000
项目名称是：超人改造项目 项目人数：10000 项目费用：1000000000
项目名称是：第4个项目    项目人数：20    项目费用：4000000
项目名称是：第5个项目    项目人数：25    项目费用：5000000
.
.
.
```

上面的程序增加了复杂性，但是从面向对象的开发上来看，`project.add()`增加一个项目是不是更友好一些？

上面的例子就使用了迭代器模式，我们来看看迭代器的通用类图：



类图是很简单，但是你看用起来就很麻烦，就比如上面例子的两个实现方法，你觉的那个简单？当然是第一个了！23 个设计模式是为了简化我们代码和设计的复杂度、耦合程度，为什么我们用了这个迭代器模式程序会复杂了一些呢？这是为什么？因为从 JDK 1.2 版本开始增加 `java.util.Iterator` 这个接口，并逐步把 `Iterator` 应用到各个聚集类(`Collection`)中，我们来看 JDK 1.5 的 API 帮助文件，你会看到有一个叫 `java.util.Iterable` 的接口，看看有多少个接口继承了它：

```
java.lang
```

接口 `Iterable<T>`

所有已知子接口：

[BeanContext](#), [BeanContextServices](#), [BlockingQueue<E>](#), [Collection<E>](#), [List<E>](#), [Queue<E>](#),
[Set<E>](#), [SortedSet<E>](#)

所有已知实现类：

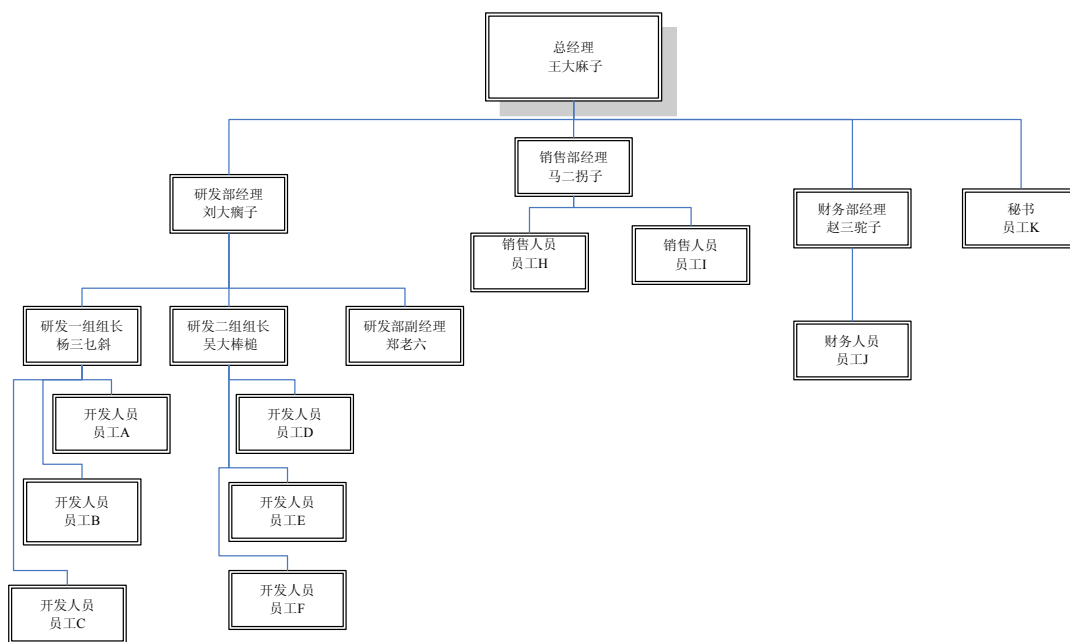
[AbstractCollection](#), [AbstractList](#), [AbstractQueue](#), [AbstractSequentialList](#), [AbstractSet](#),
[ArrayBlockingQueue](#), [ArrayList](#), [AttributeList](#), [BeanContextServicesSupport](#),
[BeanContextSupport](#), [ConcurrentLinkedQueue](#), [CopyOnWriteArrayList](#), [CopyOnWriteArraySet](#),
[DelayQueue](#), [EnumSet](#), [HashSet](#), [JobStateReasons](#), [LinkedBlockingQueue](#), [LinkedHashSet](#),
[LinkedList](#), [PriorityBlockingQueue](#), [PriorityQueue](#), [RoleList](#), [RoleUnresolvedList](#), [Stack](#),
[SynchronousQueue](#), [TreeSet](#), [Vector](#)

`java.util.Iterable` 接口只有一个方法：`iterator()`，也就说通过 `iterator()` 这个方法去遍历聚集类中的所有方法或属性，基本上现在所有的高级的语言都有 `Iterator` 这个接口或者实现，Java 已经把迭代器给我们准备了，我们再去写迭代器，是不是“六指儿抓痒，多一道子”？所以呀，这个迭代器模式也有点没落了，基本上很少有项目再独立写迭代器了，直接使用 `List` 或者 `Map` 就可以完整的解决问题。

第 15 章 组合模式【Composite Pattern】

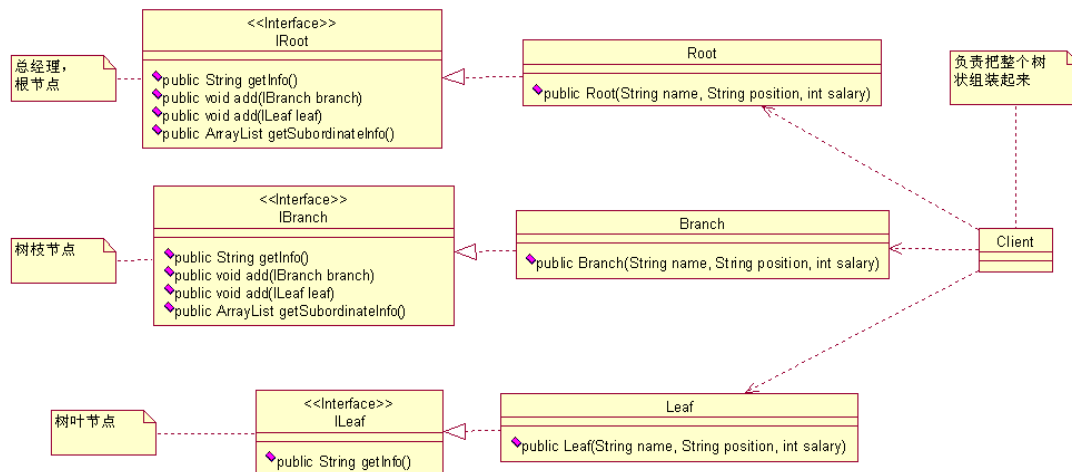
大家在上学的时候应该都学过“数据结构”这门课程吧，还记得其中有一节叫“二叉树”吧，我们上学那会儿这一章节是必考内容，左子树，右子树，什么先序遍历后序遍历什么，重点就是二叉树的遍历，我还记得当时老师就说，考试的时候一定有二叉树的构建和遍历，现在想起来还是觉的老师是正确的，树状结果在实际项目应用的非常广泛。

咱就先说个最常见的例子，公司的人事管理就是一个典型的树状结构，你想想你公司的结构是不是这样：



从最高的老大，往下一层一层的管理，最后到我们这层小兵，很典型的树状结构（说明一下，这不是二叉树，有关二叉树的定义可以翻翻以前的教科书），我们今天的任务就是要把这个树状结构实现出来，并且还要把它遍历一遍，你要确认你建立的树是否有问题呀。

从这个树状结构上分析，有两种节点：有分支的节点（如研发部经理）和无分支的节点（如员工 A、员工 D 等），我们增加一点学术术语上去，总经理叫做根节点（是不是想到 XML 中的那个根节点 root，那就对了），类似研发部经理有分支的节点叫做树枝节点，类似员工 A 的无分支的节点叫做树叶节点，都很形象，三个类型的节点，那是不是定义三个类就可以？好，我们按照这个思路走下去，先看我们自己设计的类图：



这个类图是初学者最容易想到的类图（如果你已经看明白这个类图的缺陷了，就可以不看下边的实现了，我是循序渐进的讲课，呵呵），我那来看这个实现：

先看最高级别的根节点的实现：

```

package com.cbf4life.common;

import java.util.ArrayList;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 定义一个根节点，就为总经理服务
 */
public interface IRoot {

    //得到总经理的信息
    public String getInfo();

    //总经理下边要有小兵，那要能增加小兵，比如研发部总经理，这是个树枝节点
    public void add(IBranch branch);

    //那要能增加树叶节点
    public void add(ILeaf leaf);

    //既然能增加，那要还要能够遍历，不可能总经理不知道他手下有哪些人
    public ArrayList getSubordinateInfo();

}
  
```

这个根节点就是我们的总经理 CEO，然后看实现类：

```
package com.cbf4life.common;

import java.util.ArrayList;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 根节点的实现类
 */
@SuppressWarnings("all")
public class Root implements IRoot {
    //保存根节点下的树枝节点和树叶节点，Subordinate的意思是下级
    private ArrayList subordinateList = new ArrayList();
    //根节点的名称
    private String name = "";
    //根节点的职位
    private String position = "";
    //根节点的薪水
    private int salary = 0;

    //通过构造函数传递进来总经理的信息
    public Root(String name,String position,int salary){
        this.name = name;
        this.position = position;
        this.salary = salary;
    }

    //增加树枝节点
    public void add(IBranch branch) {
        this.subordinateList.add(branch);
    }

    //增加叶子节点，比如秘书，直接隶属于总经理
    public void add(ILeaf leaf) {
        this.subordinateList.add(leaf);
    }

    //得到自己的信息
    public String getInfo() {
        String info = "";
        info = "名称: " + this.name;;
    }
}
```

```

        info = info + "\t职位: " + this.position;
        info = info + "\t薪水: " + this.salary;
        return info;
    }

    //得到下级的信息
    public ArrayList getSubordinateInfo() {
        return this.subordinateList;
    }
}

```

很简单，通过构造函数传入参数，然后获得信息，还可以增加子树枝节点（部门经理）和叶子节点（秘书）。我们再来看 IBranch.java:

```

package com.cbf4life.common;

import java.util.ArrayList;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 树枝节点，也就是各个部门经理和组长的角色
 */
public interface IBranch {

    //获得信息
    public String getInfo();

    //增加数据节点，例如研发部下的研发一组
    public void add(IBranch branch);

    //增加叶子节点
    public void add(ILeaf leaf);

    //获得下级信息
    public ArrayList getSubordinateInfo();
}

```

下面是树枝节点的实现类:

```
package com.cbf4life.common;

import java.util.ArrayList;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 所有的树枝节点
 */
@SuppressWarnings("all")
public class Branch implements IBranch {
    //存储子节点的信息
    private ArrayList subordinateList = new ArrayList();

    //树枝节点的名称
    private String name="";
    //树枝节点的职位
    private String position = "";
    //树枝节点的薪水
    private int salary = 0;

    //通过构造函数传递树枝节点参数
    public Branch(String name,String position,int salary){
        this.name = name;
        this.position = position;
        this.salary = salary;
    }

    //增加一个子树枝节点
    public void add(IBranch branch) {
        this.subordinateList.add(branch);
    }

    //增加一个叶子节点
    public void add(ILeaf leaf) {
        this.subordinateList.add(leaf);
    }

    //获得自己树枝节点的信息
    public String getInfo() {
        String info = "";
        info = "名称: " + this.name;
        info = info + "\t职位: "+ this.position;
        info = info + "\t薪水: "+this.salary;
    }
}
```

```

        return info;
    }

    //获得下级的信息
    public ArrayList getSubordinateInfo() {
        return this.subordinateList;
    }
}

```

最后看叶子节点，也就是员工的接口：

```

package com.cbf4life.common;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 叶子节点，也就是最小的小兵了，只能自己干活，不能指派别人了
 */
public interface ILeaf {

    //获得自己的信息呀
    public String getInfo();

}

```

下面是叶子节点的实现类：

```

package com.cbf4life.common;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 最小的叶子节点
 */
@SuppressWarnings("all")
public class Leaf implements ILeaf {
    //叶子叫什么名字
    private String name = "";
    //叶子的职位
    private String position = "";
    //叶子的薪水
    private int salary=0;
}

```



```

//通过构造函数传递信息
public Leaf(String name,String position,int salary){
    this.name = name;
    this.position = position;
    this.salary = salary;
}
//最小的小兵只能获得自己的信息了
public String getInfo() {
    String info = "";
    info = "名称: " + this.name;
    info = info + "\t职位: " + this.position;
    info = info + "\t薪水: " + this.salary;
    return info;
}
}

```

好了，所有的根节点，树枝节点和叶子节点都已经实现了，从总经理、部门经理到最终的员工都已经实现了，然后的工作就是组装成一个树状结构和遍历这个树状结构，看 Client.java 程序：

```

package com.cbf4life.common;

import java.util.ArrayList;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * Client的作用是组装这棵树，并遍历一遍
 */
@SuppressWarnings("all")
public class Client {

    public static void main(String[] args) {
        //首先产生了一个根节点
        IRoot ceo = new Root("王大麻子", "总经理", 100000);

        //产生三个部门经理，也就是树枝节点
        IBranch developDep = new Branch("刘大瘸子", "研发部门经理", 10000);
        IBranch salesDep = new Branch("马二拐子", "销售部门经理", 20000);
        IBranch financeDep = new Branch("赵三驼子", "财务部经理", 30000);

        //再把三个小组长产生出来
        IBranch firstDevGroup = new Branch("杨三乜斜", "开发一组组长", 5000);
    }
}

```

```
IBranch secondDevGroup = new Branch("吴大棒槌", "开发二组组长", 6000);
```

```
//剩下的及时我们这些小兵了,就是路人甲,路人乙
```

```
ILeaf a = new Leaf("a", "开发人员", 2000);
```

```
ILeaf b = new Leaf("b", "开发人员", 2000);
```

```
ILeaf c = new Leaf("c", "开发人员", 2000);
```

```
ILeaf d = new Leaf("d", "开发人员", 2000);
```

```
ILeaf e = new Leaf("e", "开发人员", 2000);
```

```
ILeaf f = new Leaf("f", "开发人员", 2000);
```

```
ILeaf g = new Leaf("g", "开发人员", 2000);
```

```
ILeaf h = new Leaf("h", "销售人员", 5000);
```

```
ILeaf i = new Leaf("i", "销售人员", 4000);
```

```
ILeaf j = new Leaf("j", "财务人员", 5000);
```

```
ILeaf k = new Leaf("k", "CEO秘书", 8000);
```

```
ILeaf zhengLaoLiu = new Leaf("郑老六", "研发部副总", 20000);
```

```
//该产生的人都产生出来了,然后我们怎么组装这棵树
```

```
//首先是定义总经理下有三个部门经理
```

```
ceo.add(developDep);
```

```
ceo.add(salesDep);
```

```
ceo.add(financeDep);
```

```
//总经理下还有一个秘书
```

```
ceo.add(k);
```

```
//定义研发部门 下的结构
```

```
developDep.add(firstDevGroup);
```

```
developDep.add(secondDevGroup);
```

```
//研发部经理下还有一个副总
```

```
developDep.add(zhengLaoLiu);
```

```
//看看开发两个开发小组下有什么
```

```
firstDevGroup.add(a);
```

```
firstDevGroup.add(b);
```

```
firstDevGroup.add(c);
```

```
secondDevGroup.add(d);
```

```
secondDevGroup.add(e);
```

```
secondDevGroup.add(f);
```

```
//再看销售部下的人员情况
```

```
salesDep.add(h);
```

```
salesDep.add(i);
```

```
//最后一个财务
```

```
financeDep.add(j);
```

```
//树状结构写完毕，然后我们打印出来
System.out.println(ceo.getInfo());

//打印出来整个树形
getAllSubordinateInfo(ceo.getSubordinateInfo());

}

//遍历所有的树枝节点，打印出信息
private static void getAllSubordinateInfo(ArrayList subordinateList){
    int length = subordinateList.size();
    for(int m=0;m<length;m++){ //定义一个ArrayList长度，不要在for循环中每次计算
        Object s = subordinateList.get(m);
        if(s instanceof Leaf){ //是个叶子节点，也就是员工
            ILeaf employee = (ILeaf)s;
            System.out.println(((Leaf) s).getInfo());
        }else{
            IBranch branch = (IBranch)s;
            System.out.println(branch.getInfo());
            //再递归调用
            getAllSubordinateInfo(branch.getSubordinateInfo());
        }
    }
}
}
```

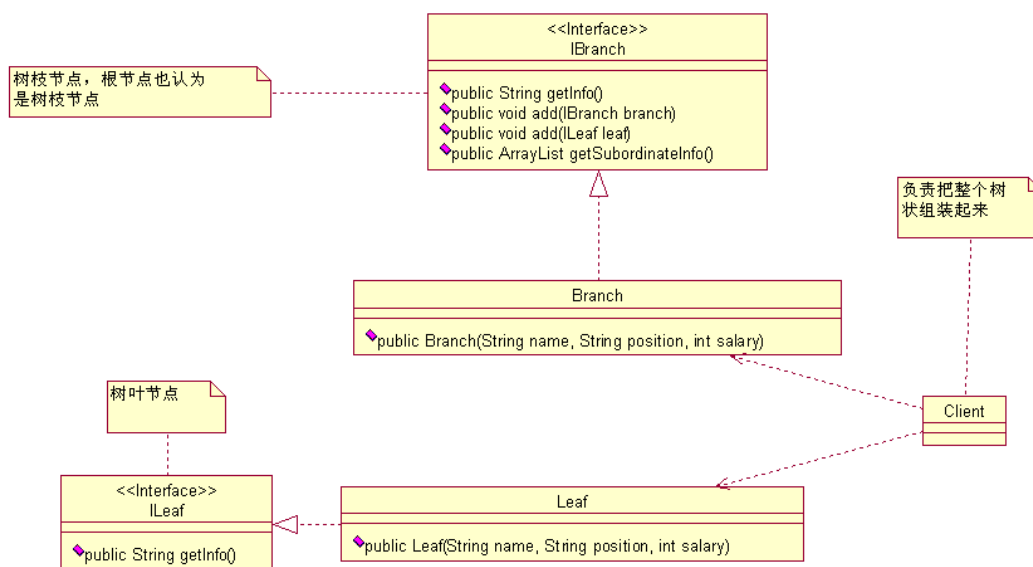
这个程序比较长，如果是在我们的项目中有这样的程序，肯定是被拉出来做典型的，你写一大坨的程序给谁呀，以后还要维护的，程序是要短小精悍！幸运的是，我们是这为案例来讲解，而且就是指出这样组装这棵树是有问题，等会我们深入讲解，先看运行结果：

```
名称：王大麻子   职位：总经理 薪水：100000
名称：刘大瘸子   职位：研发部门经理 薪水：10000
名称：杨三乜斜   职位：开发一组组长 薪水：5000
名称：a   职位：开发人员 薪水：2000
名称：b   职位：开发人员 薪水：2000
名称：c   职位：开发人员 薪水：2000
名称：吴大棒槌   职位：开发二组组长 薪水：6000
名称：d   职位：开发人员 薪水：2000
名称：e   职位：开发人员 薪水：2000
名称：f   职位：开发人员 薪水：2000
名称：郑老六   职位：研发部副总 薪水：20000
```

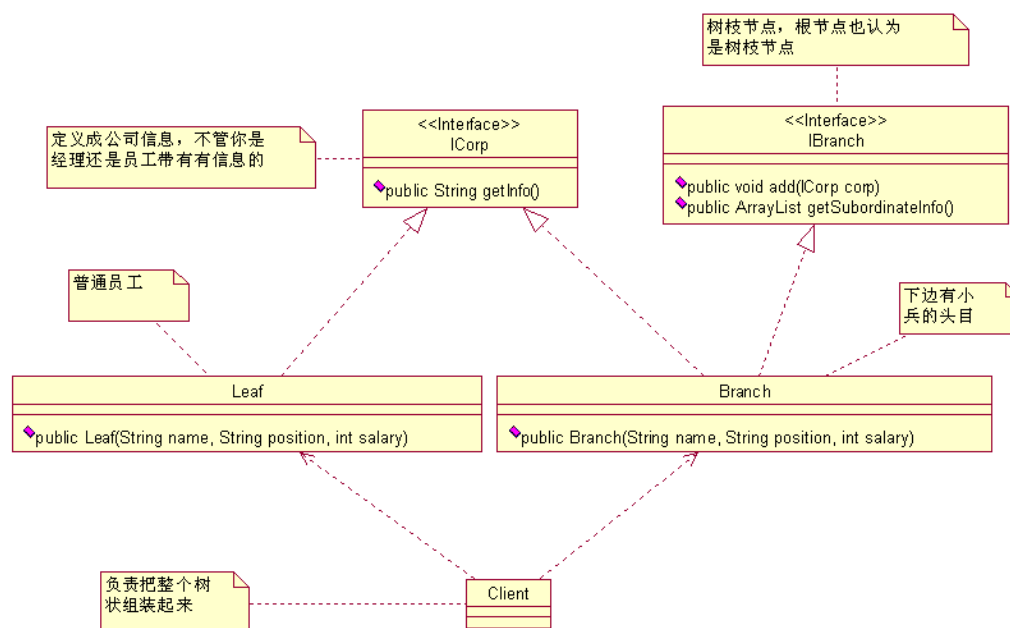
名称：马二拐子 职位：销售部门经理 薪水：20000
 名称：h 职位：销售人员 薪水：5000
 名称：i 职位：销售人员 薪水：4000
 名称：赵三驼子 职位：财务部经理 薪水：30000
 名称：j 职位：财务人员 薪水：5000
 名称：k 职位：CEO秘书 薪水：8000

和我们期望要的结果一样，一棵完整的树就生成了，而且我们还能够遍历。看类图或程序的时候，你有没有发觉有问题？getInfo 每个接口都有为什么不能抽象出来？Root 类和 Branch 类有什么差别？为什么要定义成两个接口两个类？如果我要加一个任职期限，你是不是每个类都需要修改？如果我要后序遍历（从员工找到他的上级领导）能做吗？——彻底晕菜了！

问题很多，我们一个一个解决，先说抽象的问题，确实可以吧 IBranch 和 IRoot 合并成一个接口，这个我们先肯定下来，这是个比较大的改动，我们先画个类图：



这个类图还是有点问题的，接口的作用是什么？定义共性，那 ILeaf 和 IBranch 是不是也有共性呢？有 getInfo ()，我们是不是要把这个共性也已经封装起来呢？好，我们再修改一下类图：



类图上有两个接口，`ICorp` 是公司所有人员的信息的接口类，不管你是经理还是员工，你都有名字，职位，薪水，这个定义成一个接口没有错，`IBranch` 有没有必要呢？我们先实现出来然后再说。

先看 `ICorp.java` 源代码：

```

package com.cbf4life.advance;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 公司类，定义每个员工都有信息
 */
public interface ICorp {

    //每个员工都有信息，你想隐藏，门儿都没有！
    public String getInfo();
}

```

接口很简单，只有一个方法，就是获得员工的信息，我们再来看实现类：

```

package com.cbf4life.advance;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.

```

```

* Leaf是树叶节点，在这里就是我们这些小兵
*/
@SuppressWarnings("all")
public class Leaf implements ICorp {
    //小兵也有名称
    private String name = "";
    //小兵也有职位
    private String position = "";
    //小兵也有薪水，否则谁给你干
    private int salary = 0;

    //通过一个构造函数传递小兵的信息
    public Leaf(String name,String position,int salary){
        this.name = name;
        this.position = position;
        this.salary = salary;
    }

    //获得小兵的信息
    public String getInfo() {
        String info = "";
        info = "姓名: " + this.name;
        info = info + "\t职位: " + this.position;
        info = info + "\t薪水: " + this.salary;
        return info;
    }
}

```

小兵就只有这些信息了，我们是具体干活的，我们是管理不了其他同事的，我们来看看那些经理和小组长是怎么实现的，先看接口：

```

package com.cbf4life.advance;

import java.util.ArrayList;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 这些下边有小兵或者是经理等风云人物
 */
public interface IBranch {

```

```

//能够增加小兵(树叶节点)或者是经理(树枝节点)
public void addSubordinate(ICorp corp);

//我还要能够获得下属的信息
public ArrayList<ICorp> getSubordinate();

/*本来还应该有一个方法delSubordinate(ICorp corp), 删除下属
 * 这个方法我们没有用到就不写进来了
 */
}

```

接口也是很简单的，下面是实现类：

```

package com.cbf4life.advance;

import java.util.ArrayList;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 这些树枝节点也就是这些领导们既要有自己的信息，还要知道自己的下属情况
 */
@SuppressWarnings("all")
public class Branch implements IBranch, ICorp {
    //领导也是人，也有名字
    private String name = "";
    //领导和领导不同，也是职位区别
    private String position = "";
    //领导也是拿薪水的
    private int salary = 0;
    //领导下边有那些下级领导和小兵
    ArrayList<ICorp> subordinateList = new ArrayList<ICorp>();

    //通过构造函数传递领导的信息
    public Branch(String name, String position, int salary) {
        this.name = name;
        this.position = position;
        this.salary = salary;
    }

    //增加一个下属，可能是小头目，也可能是个小兵

```

```

    public void addSubordinate(ICorp corp) {
        this.subordinateList.add(corp);
    }

    //我有哪些下属
    public ArrayList<ICorp> getSubordinate() {
        return this.subordinateList;
    }

    //领导也是人，他也有信息
    public String getInfo() {
        String info = "";
        info = "姓名: " + this.name;
        info = info + "\t职位: " + this.position;
        info = info + "\t薪水: " + this.salary;
        return info;
    }
}

```

实现类也很简单，不多说，程序写的好不好，就看别人怎么调用了，我们看 Client.java 程序：

```

package com.cbf4life.advance;

import java.util.ArrayList;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 组装这个树形结构，并展示出来
 */
@SuppressWarnings("all")
public class Client {

    public static void main(String[] args) {
        //首先是组装一个组织结构出来
        Branch ceo = compositeCorpTree();

        //首先把CEO的信息打印出来：
        System.out.println(ceo.getInfo());

        //然后是所有员工信息
        System.out.println(getTreeInfo(ceo));
    }
}

```



```
}

//把整个树组装出来
public static Branch compositeCorpTree(){
    //首先产生总经理CEO
    Branch root = new Branch("王大麻子", "总经理", 100000);
    //把三个部门经理产生出来
    Branch developDep = new Branch("刘大瘸子", "研发部门经理", 10000);
    Branch salesDep = new Branch("马二拐子", "销售部门经理", 20000);
    Branch financeDep = new Branch("赵三驼子", "财务部经理", 30000);

    //再把三个小组长产生出来
    Branch firstDevGroup = new Branch("杨三乜斜", "开发一组组长", 5000);
    Branch secondDevGroup = new Branch("吴大棒槌", "开发二组组长", 6000);

    //把所有的小兵都产生出来
    Leaf a = new Leaf("a", "开发人员", 2000);
    Leaf b = new Leaf("b", "开发人员", 2000);
    Leaf c = new Leaf("c", "开发人员", 2000);
    Leaf d = new Leaf("d", "开发人员", 2000);
    Leaf e = new Leaf("e", "开发人员", 2000);
    Leaf f = new Leaf("f", "开发人员", 2000);
    Leaf g = new Leaf("g", "开发人员", 2000);
    Leaf h = new Leaf("h", "销售人员", 5000);
    Leaf i = new Leaf("i", "销售人员", 4000);
    Leaf j = new Leaf("j", "财务人员", 5000);
    Leaf k = new Leaf("k", "CEO秘书", 8000);
    Leaf zhengLaoLiu = new Leaf("郑老六", "研发部副经理", 20000);

    //开始组装
    //CEO下有三个部门经理和一个秘书
    root.addSubordinate(k);
    root.addSubordinate(developDep);
    root.addSubordinate(salesDep);
    root.addSubordinate(financeDep);

    //研发部经理
    developDep.addSubordinate(zhengLaoLiu);
    developDep.addSubordinate(firstDevGroup);
    developDep.addSubordinate(secondDevGroup);
```

```

//看看开发两个开发小组下有什么
firstDevGroup.addSubordinate(a);
firstDevGroup.addSubordinate(b);
firstDevGroup.addSubordinate(c);
secondDevGroup.addSubordinate(d);
secondDevGroup.addSubordinate(e);
secondDevGroup.addSubordinate(f);

//再看销售部下的人员情况
salesDep.addSubordinate(h);
salesDep.addSubordinate(i);

//最后一个财务
financeDep.addSubordinate(j);

return root;
}

//遍历整棵树,只要给我根节点,我就能遍历出所有的节点
public static String getTreeInfo(Branch root){
    ArrayList<ICorp> subordinateList = root.getSubordinate();
    String info = "";
    for(ICorp s :subordinateList){
        if(s instanceof Leaf){ //是员工就直接获得信息
            info = info + s.getInfo()+"\n";
        }else{ //是个小头目
            info = info + s.getInfo() + "\n" + getTreeInfo((Branch)s);
        }
    }

    return info;
}
}

```

运行结果如下:

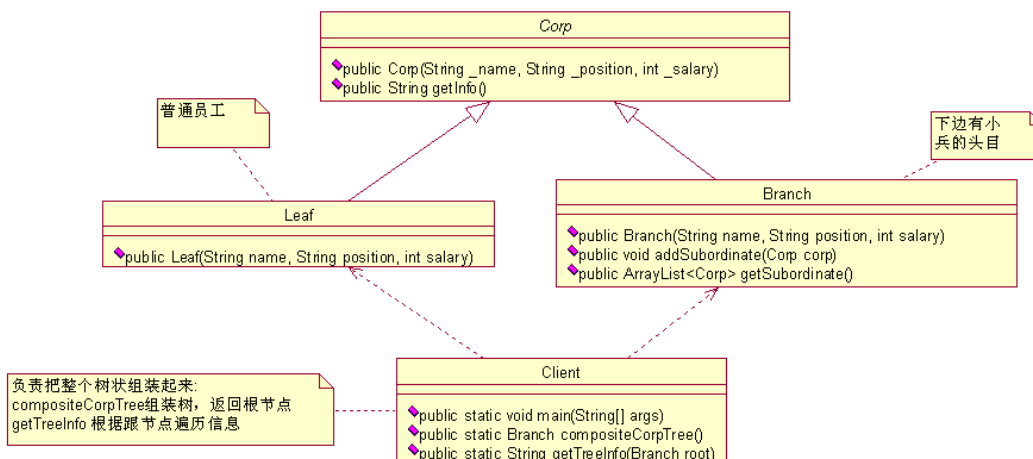
```

姓名: 王大麻子  职位: 总经理 薪水: 100000
姓名: k  职位: CEO秘书  薪水: 8000
姓名: 刘大瘸子  职位: 研发部门经理  薪水: 10000
姓名: 郑老六  职位: 研发部副经理  薪水: 20000
姓名: 杨三乜斜  职位: 开发一组组长  薪水: 5000
姓名: a  职位: 开发人员  薪水: 2000

```

姓名: b 职位: 开发人员 薪水: 2000
 姓名: c 职位: 开发人员 薪水: 2000
 姓名: 吴大棒槌 职位: 开发二组组长 薪水: 6000
 姓名: d 职位: 开发人员 薪水: 2000
 姓名: e 职位: 开发人员 薪水: 2000
 姓名: f 职位: 开发人员 薪水: 2000
 姓名: 马二拐子 职位: 销售部门经理 薪水: 20000
 姓名: h 职位: 销售人员 薪水: 5000
 姓名: i 职位: 销售人员 薪水: 4000
 姓名: 赵三驼子 职位: 财务部经理 薪水: 30000
 姓名: j 职位: 财务人员 薪水: 5000

一个非常清理的树状人员资源管理图出现了,那我们的程序是否还可以优化? 可以!你看 Leaf 和 Branch 中都有 getInfo 信息,是否可以抽象,好,我们抽象一下:



你一看这个图,乐了,能不乐嘛,减少很多工作量了,接口没有了,改成抽象类了,IBranch 接口也没有了,直接把方法放到了实现类中了,那我们先来看抽象类:

```
package com.cbf4life.perfect;
```

```
/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 定义一个公司的人员的抽象类
 */
@SuppressWarnings("all")
public abstract class Corp {
    //公司每个人都有名称
```

```
private String name = "";
//公司每个人都职位
private String position = "";
//公司每个人都有薪水
private int salary =0;

/*通过接口的方式传递，我们改变一下习惯，传递进来的参数名以下划线开始
 * 这个在一些开源项目中非常常见，一般构造函数都是这么定义的
 */
public Corp(String _name,String _position,int _salary){
    this.name = _name;
    this.position = _position;
    this.salary = _salary;
}

//获得员工信息
public String getInfo(){
    String info = "";
    info = "姓名: " + this.name;
    info = info + "\t职位: " + this.position;
    info = info + "\t薪水: " + this.salary;
    return info;
}
}
```

抽象类嘛，就应该抽象出一些共性的东西出来，然后看两个具体的实现类：

```
package com.cbf4life.perfect;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 普通员工很简单，就写一个构造函数就可以了
 */
public class Leaf extends Corp {

    //就写一个构造函数，这个是必须的
    public Leaf(String _name,String _position,int _salary){
        super(_name,_position,_salary);
    }
}
```

这个改动比较多，就几行代码就完成了，确实就应该这样，下面是小头目的实现类：

```
package com.cbf4life.perfect;

import java.util.ArrayList;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 节点类，也简单了很多
 */
public class Branch extends Corp {
    //领导下边有那些下级领导和小兵
    ArrayList<Corp> subordinateList = new ArrayList<Corp>();

    //构造函数是必须的
    public Branch(String _name,String _position,int _salary){
        super(_name,_position,_salary);
    }

    //增加一个下属，可能是小头目，也可能是个小兵
    public void addSubordinate(Corp corp) {
        this.subordinateList.add(corp);
    }

    //我有哪些下属
    public ArrayList<Corp> getSubordinate() {
        return this.subordinateList;
    }
}
```

也缩减了很多，再看 Client.java 程序，这个就没有多大变化了：

```
package com.cbf4life.perfect;

import java.util.ArrayList;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 组装这个树形结构，并展示出来
 */
```

```
*/
@SuppressWarnings("all")
public class Client {

    public static void main(String[] args) {
        //首先是组装一个组织结构出来
        Branch ceo = compositeCorpTree();

        //首先把CEO的信息打印出来:
        System.out.println(ceo.getInfo());

        //然后是所有员工信息
        System.out.println(getTreeInfo(ceo));
    }

    //把整个树组装出来
    public static Branch compositeCorpTree(){
        //首先产生总经理CEO
        Branch root = new Branch("王大麻子", "总经理", 100000);
        //把三个部门经理产生出来
        Branch developDep = new Branch("刘大瘸子", "研发部门经理", 10000);
        Branch salesDep = new Branch("马二拐子", "销售部门经理", 20000);
        Branch financeDep = new Branch("赵三驼子", "财务部经理", 30000);

        //再把三个小组长产生出来
        Branch firstDevGroup = new Branch("杨三乜斜", "开发一组组长", 5000);
        Branch secondDevGroup = new Branch("吴大棒槌", "开发二组组长", 6000);

        //把所有的小兵都产生出来
        Leaf a = new Leaf("a", "开发人员", 2000);
        Leaf b = new Leaf("b", "开发人员", 2000);
        Leaf c = new Leaf("c", "开发人员", 2000);
        Leaf d = new Leaf("d", "开发人员", 2000);
        Leaf e = new Leaf("e", "开发人员", 2000);
        Leaf f = new Leaf("f", "开发人员", 2000);
        Leaf g = new Leaf("g", "开发人员", 2000);
        Leaf h = new Leaf("h", "销售人员", 5000);
        Leaf i = new Leaf("i", "销售人员", 4000);
        Leaf j = new Leaf("j", "财务人员", 5000);
        Leaf k = new Leaf("k", "CEO秘书", 8000);
        Leaf zhengLaoLiu = new Leaf("郑老六", "研发部副经理", 20000);

        //开始组装
        //CEO下有三个部门经理和一个秘书
```

```

root.addSubordinate(k);
root.addSubordinate(developDep);
root.addSubordinate(salesDep);
root.addSubordinate(financeDep);

//研发部经理
developDep.addSubordinate(zhengLaoLiu);
developDep.addSubordinate(firstDevGroup);
developDep.addSubordinate(secondDevGroup);

//看看开发两个开发小组下有什么
firstDevGroup.addSubordinate(a);
firstDevGroup.addSubordinate(b);
firstDevGroup.addSubordinate(c);
secondDevGroup.addSubordinate(d);
secondDevGroup.addSubordinate(e);
secondDevGroup.addSubordinate(f);

//再看销售部下的人员情况
salesDep.addSubordinate(h);
salesDep.addSubordinate(i);

//最后一个财务
financeDep.addSubordinate(j);

return root;
}

//遍历整棵树,只要给我根节点,我就能遍历出所有的节点
public static String getTreeInfo(Branch root){
    ArrayList<Corp> subordinateList = root.getSubordinate();
    String info = "";
    for(Corp s :subordinateList){
        if(s instanceof Leaf){ //是员工就直接获得信息
            info = info + s.getInfo()+"\n";
        }else{ //是个小头目
            info = info + s.getInfo() + "\n" + getTreeInfo((Branch)s);
        }
    }

    return info;
}

```

```
}  
  
}
```

就是把用到 ICorp 接口的地方修改为 Corp 抽象类就成了，就上面黄色的部分作了点修改，其他保持不变，运行结果还是保持一样：

```
姓名：王大麻子  职位：总经理 薪水：100000  
姓名：k  职位：CEO秘书 薪水：8000  
姓名：刘大瘸子  职位：研发部门经理 薪水：10000  
姓名：郑老六  职位：研发部副经理 薪水：20000  
姓名：杨三乜斜  职位：开发一组组长 薪水：5000  
姓名：a  职位：开发人员 薪水：2000  
姓名：b  职位：开发人员 薪水：2000  
姓名：c  职位：开发人员 薪水：2000  
姓名：吴大棒槌  职位：开发二组组长 薪水：6000  
姓名：d  职位：开发人员 薪水：2000  
姓名：e  职位：开发人员 薪水：2000  
姓名：f  职位：开发人员 薪水：2000  
姓名：马二拐子  职位：销售部门经理 薪水：20000  
姓名：h  职位：销售人员 薪水：5000  
姓名：i  职位：销售人员 薪水：4000  
姓名：赵三驼子  职位：财务部经理 薪水：30000  
姓名：j  职位：财务人员 薪水：5000
```

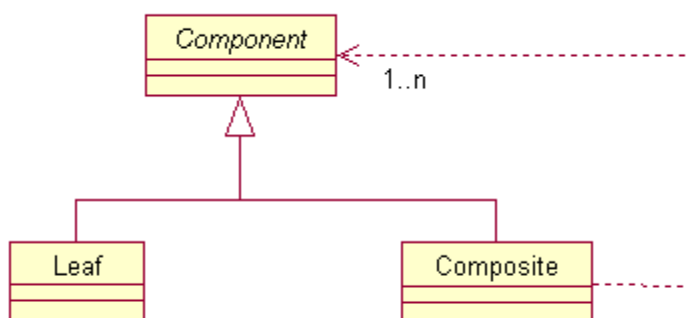
确实是类、接口减少了很多，而且程序也简单很多，但是大家可能还是很迷茫，这个 Client 程序并没有改变多少呀，非常正确，树的组装你是跑不了的，你要知道在项目中使用数据库来存储这些信息的，你从数据库中提出出来哪些人要分配到树枝，哪些人要分配到树叶，树枝与树枝、树叶的关系，这些都需要人去定义，通常这里使用一个界面去配置，在数据库中是一个标志信息，例如定义这样一张表：

主键	唯一编码	名称	是否是叶子节点	父节点
1	CEO	总经理	否	
2	developDep	研发部经理	否	CEO
3	salesDep	销售部经理	否	CEO
4	financeDep	财务部经理	否	CEO
5	k	总经理秘书	是	CEO

6	a	员工 A	是	developed
7	b	员工 B	是	Developed

从这张表中已经定义一个树形结构，我们要做的就是从数据库中读取出来，然后展现到前台上，这个读取就用个 for 循环加上递归是不是就可以把一棵树建立起来？我们程序中其实还包涵了数据的读取和加工，用了数据库后，数据和逻辑已经在表中定义好了，我们直接读取放到树上就可以了，这个还是比较容易做了的，大家不妨自己考虑一下。

上面我们讲到的就是组合模式（也叫合成模式），有时又叫做部分—整体模式（Part-Whole），主要是用来描述整体与部分的关系，用的最多的地方就是树形结构。组合模式通用类图如下：



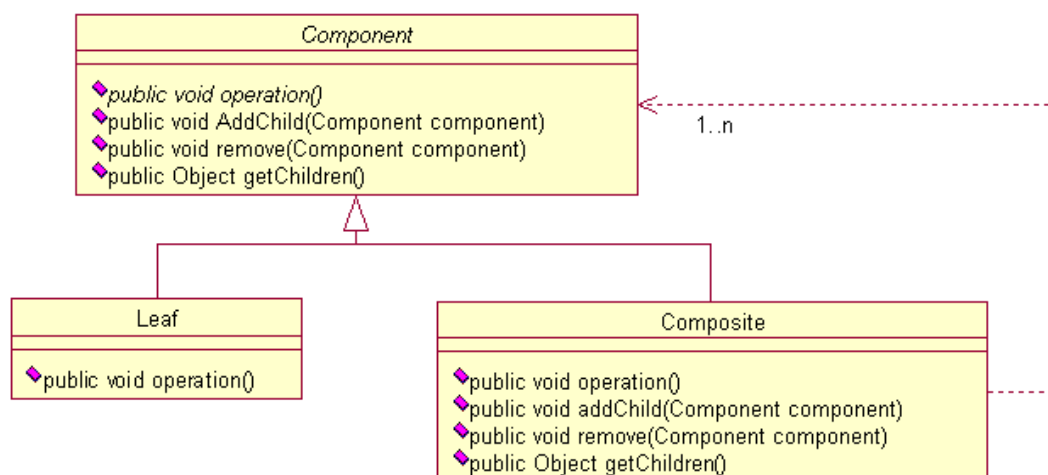
我们先来说说组合模式的几个角色：

抽象构件角色 (Component)：定义参加组合的对象的共有方法和属性，可以定义一些默认的行为或属性；比如我们例子中的 `getInfo` 就封装到了抽象类中。

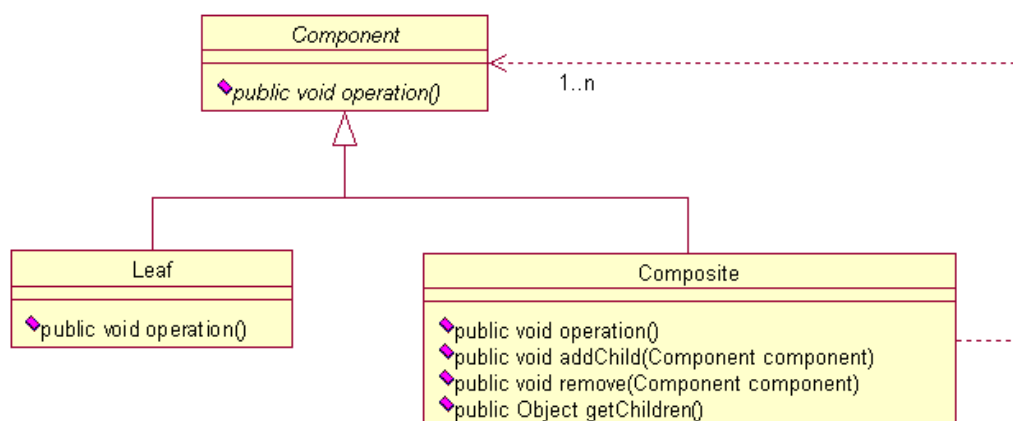
叶子构件 (Leaf)：叶子对象，其下再也没有其他的分支。

树枝构件 (Composite)：树枝对象，它的作用是组合树枝节点和叶子节点；

组合模式有两种模式，透明模式和安全模式，这两个模式有什么区别呢？先看类图：



透明模式类图



安全模式类图

从类图上大家应该能看清楚了，这两种模式各有优缺点，透明模式是把用来组合使用的方法放到抽象类中，比如 `add()`, `remove()` 以及 `getChildren` 等方法（顺便说一下，`getChildren` 一般返回的结果为 `Iterable` 的实现类，很多，大家可以看 JDK 的帮助），不管叶子对象还是树枝对象都有相同的结构，通过判断是 `getChildren` 的返回值确认是叶子节点还是树枝节点，如果处理不当，这个会在运行期出现问题的，不是很建议的方式；安全模式就不同了，它是把树枝节点和树叶节点彻底分开，树枝节点单独拥有用来组合的方法，这种方法比较安全，我们的例子使用了安全模式。

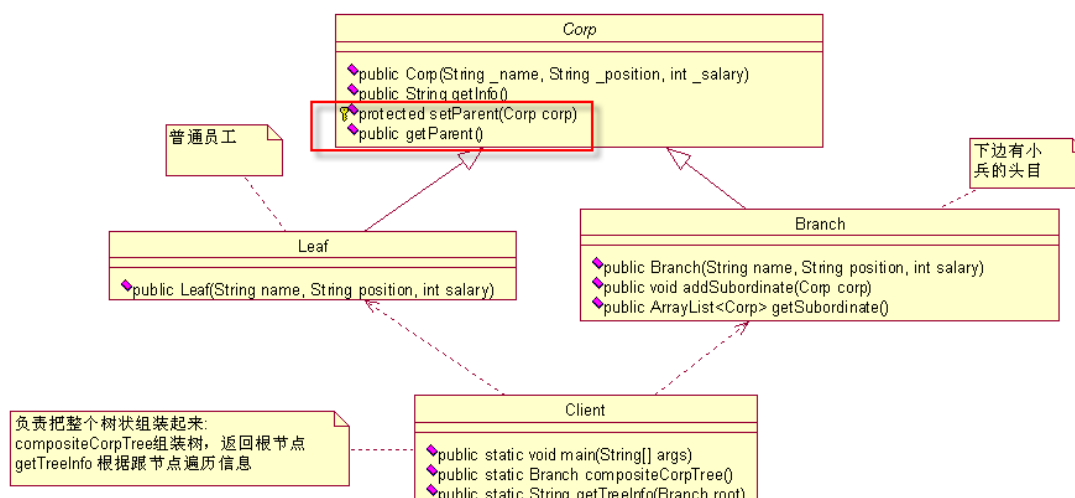
组合模式的优点有哪些呢？第一个优点只要是树形结构，就要考虑使用组合模式，这个一定记住，只要是要体现局部和整体的关系的时候，而且这种关系还可能比较深，考虑一下组合模式吧。组合模式有一个非常明显的缺点，看到我们在 `Client.java` 中的定义了树叶和树枝使用时的定义了吗？如下：

```
Branch developDep = new Branch("刘大瘸子", "研发部门经理", 10000);
...
Leaf g = new Leaf("g", "开发人员", 2000);
```

发现什么问题了吗？直接使用了实现类！这个在面向接口编程上是很不恰当的，这个在使用的时候要考虑清楚。

组合模式在项目中到处都有，比如现在的页面结构一般都是上下结构，上面放系统的 Logo，下边分为两部分：左边是导航菜单，右边是展示区，左边的导航菜单一般都是树形的结构，比较清晰，这个 JavaScript 有很多例子，大家可以到网上搜索一把；还有，我们的自己也是一个树状结构，根据我，能够找到我的父母，根据父亲又能找到爷爷奶奶，根据母亲能够找到外公外婆等等，很典型的树形结构，而且还很规范（这个要是不规范那肯定是乱套了）。

我们在上面也还提到了一个问题，就是树的遍历问题，从上到下遍历没有问题，但是我要是从下往上遍历呢？比如在人力资源这颗树上，我从中抽取一个用户，要找到它的上级有哪些，下级有哪些，怎么处理？想想，~~~，再想想！想出来了，我们对下答案，先看类图：



看类图中的红色方框，只要增加两个方法就可以了，一个是设置父节点是谁，一个是查找父节点是谁，我们来看一下程序的改变：

```
package com.cbf4life.extend;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
```

```

* 定义一个公司的人员的抽象类
*/
@SuppressWarnings("all")
public abstract class Corp {
    //公司每个人都有名称
    private String name = "";
    //公司每个人都有职位
    private String position = "";
    //公司每个人都有薪水
    private int salary = 0;
    //父节点是谁
    private Corp parent = null;

    /* 通过接口的方式传递，我们改变一下习惯，传递进来的参数名以下划线开始
    * 这个在一些开源项目中非常常见，一般构造函数都是定义的
    */
    public Corp(String _name, String _position, int _salary){
        this.name = _name;
        this.position = _position;
        this.salary = _salary;
    }

    //获得员工信息
    public String getInfo(){
        String info = "";
        info = "姓名: " + this.name;
        info = info + "\t职位: " + this.position;
        info = info + "\t薪水: " + this.salary;
        return info;
    }

    //设置父节点
    protected void setParent(Corp _parent){
        this.parent = _parent;
    }

    //得到父节点
    public Corp getParent(){
        return this.parent;
    }
}

```

就增加了黄色部分，然后我们再来看看 Branch.java 的改变:

```

package com.cbf4life.extend;

import java.util.ArrayList;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 节点类，也简单了很多
 */
public class Branch extends Corp {
    // 领导下边有那些下级领导和小兵
    ArrayList<Corp> subordinateList = new ArrayList<Corp>();

    // 构造函数是必须的
    public Branch(String _name, String _position, int _salary) {
        super(_name, _position, _salary);
    }

    // 增加一个下属，可能是小头目，也可能是个小兵
    public void addSubordinate(Corp corp) {
        corp.setParent(this); // 设置父节点
        this.subordinateList.add(corp);
    }

    // 我有哪些下属
    public ArrayList<Corp> getSubordinate() {
        return this.subordinateList;
    }
}

```

增加了黄色部分，看懂程序了吗？就是在每个节点甭管是树枝节点还是树叶节点，都增加了一个属性：父节点对象，这样在树枝节点增加子节点或叶子的时候设置父节点，然后你看整棵树就除了根节点外每个节点都有一个父节点，剩下的事情还不好处理吗？每个节点上都有父节点了，你要往上找，那就找呗！Client 程序我就不写了，今天已经拷贝的代码实在有点多，大家自己考虑一下，写个 find 方法，然后一个一个往上找，最简单的方法了！

有了这个 parent 属性，什么后序遍历（从下往上找）、中序遍历（从中间某个环节往上或往下遍历）都解决了，这个就不多说了。

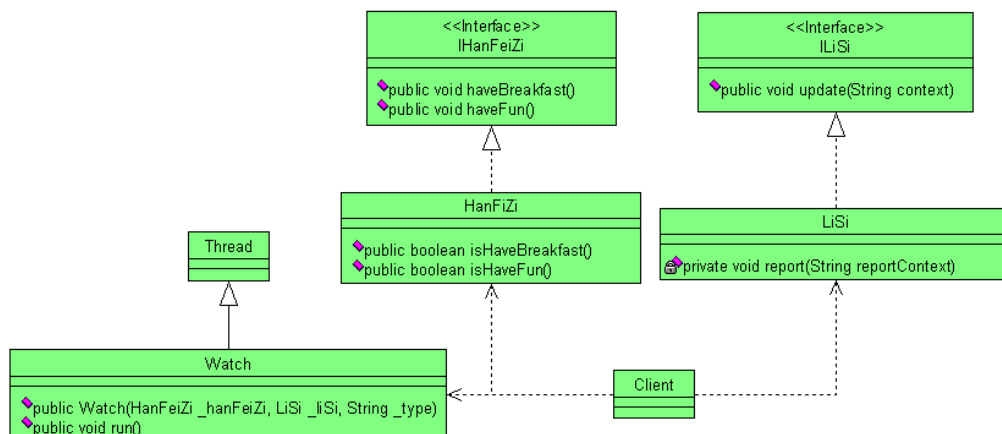
再提一个问题，树叶节点和树枝节点是有顺序的，你不能乱排的，怎么办？比如我们上面的例子，研

发一组下边有三个成员，这三个成员是要进行排序的呀，你如何处理？问我呀，问你呢，好好想想，以后用到着的！

第 16 章 观察者模式【Observer Pattern】

《孙子兵法》有云：“知彼知己，百战不殆；不知彼而知己，一胜一负；不知彼，不知己，每战必殆”，那怎么才能知己知彼呢？知己是很容易的，自己的军队嘛，很容易知道，那怎么知彼呢？安插间谍是很好的一個办法，我们今天就来讲一个间谍的故事。

韩非子大家都应该记得吧，法家的代表人物，主张建立法制社会，实施重罚制度，真是非常有远见呀，看看现在社会在呼吁什么，建立法制化的社会，在 2000 多年前就已经提出了。大家可能还不知道，法家还有一个非常重要的代表人物，李斯，对，就是李斯，秦国的丞相，最终被残忍的车裂的那位，李斯和韩非子都是荀子的学生，李斯是师兄，韩非子是师弟，若干年后，李斯成为最强诸侯秦国的上尉，致力于统一全国，于是安插了间谍到各个国家的重要人物的身边，以获取必要的信息，韩非子作为韩国的重量级人物，身边自然没少间谍了，韩非子早饭吃的什么，中午放了几个 P，晚上在做什么娱乐，李斯都了如指掌，那可是相隔千里！怎么做到的呢？间谍呀！好，我们先通过程序把这个过程展现一下，看看李斯是怎么监控韩非子，先看类图：



看惯了清一色的谈黄色类图，咱换个颜色，写程序是一个艺术创作过程，我的一个同事就曾经把一个类图画成一个小乌龟的形状，超级牛 X。这个类图应该是程序员最容易想到得，你要监控，我就给你监控，正确呀，我们来看程序的实现，先看我们的主角韩非子的接口（类似于韩非子这样的人，被观察者角色）：

```
package com.cbf4life.common;
```

```
/**
```

```
 * @author cbf4Life cbf4life@126.com
```

```
 * I'm glad to share my knowledge with you all.
```

```
 * 类似韩非子这花样的人，被监控起来了还不知道
```

```

*/
public interface IHanFeiZi {

    //韩非子也是人，也要吃早饭的
    public void haveBreakfast();

    //韩非之也是人，是人就要娱乐活动，至于活动时啥，嘿嘿，不说了
    public void haveFun();
}

```

然后看韩非子的实现类 HanFeiZi.java:

```

package com.cbf4life.common;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 韩非子，李斯的师弟，韩国的重要人物
 */
public class HanFeiZi implements IHanFeiZi{
    //韩非子是否在吃饭，作为监控的判断标准
    private boolean isHaveBreakfast = false;
    //韩非子是否在娱乐
    private boolean isHaveFun = false;

    //韩非子要吃饭了
    public void haveBreakfast(){
        System.out.println("韩非子:开始吃饭了...");
        this.isHaveBreakfast =true;
    }

    //韩非子开始娱乐了,古代人没啥娱乐，你能想到的就那么多
    public void haveFun(){
        System.out.println("韩非子:开始娱乐了...");
        this.isHaveFun = true;
    }

    //以下是bean的基本方法，getter/setter，不多说
    public boolean isHaveBreakfast() {
        return isHaveBreakfast;
    }

    public void setHaveBreakfast(boolean isHaveBreakfast) {

```



```

        this.isHaveBreakfast = isHaveBreakfast;
    }

    public boolean isHaveFun() {
        return isHaveFun;
    }

    public void setHaveFun(boolean isHaveFun) {
        this.isHaveFun = isHaveFun;
    }
}

```

其中有两个 getter/setter 方法，这个就没有在类图中表示出来，比较简单，通过 isHaveBreakfast 和 isHaveFun 这两个布尔型变量来判断韩非子是否在吃饭或者娱乐，韩非子是属于被观察者，那还有观察者李斯，我们来看李斯这类人的接口：

```

package com.cbf4life.common;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 类似于李斯的这种人，现代嘛叫做偷窥狂
 */
public interface ILiSi {

    //一发现别人有动静，自己也要行动起来
    public void update(String context);
}

```

李斯这类人比较简单，一发现自己观察的对象发生了变化，比如吃饭，娱乐了，自己立刻也要行动起来，那怎么行动呢？看实现类：

```

package com.cbf4life.common;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 李斯这个人，是个观察者，只要韩非子一有动静，这边就知道
 */

```

```

public class LiSi implements ILiSi{

    //首先李斯是个观察者，一旦韩非子有活动，他就知道，他就要向老板汇报
    public void update(String str){
        System.out.println("李斯:观察到韩非子活动，开始向老板汇报了...");
        this.reportToQiShiHuang(str);
        System.out.println("李斯: 汇报完毕，秦老板赏给他两个萝卜吃吃...\n");
    }

    //汇报给秦始皇
    private void reportToQiShiHuang(String reportContext){
        System.out.println("李斯: 报告，秦老板！韩非子有活动了--->" + reportContext);
    }
}

```

韩非子是秦始皇非常崇拜的人物，甚至说过见韩非子一面死又何憾！不过，韩非子还真是被秦始皇干掉的，历史呀上演过太多这样的悲剧。这么重要的人物有活动，你李斯敢不向老大汇报？！

两个重量级的人物都定义出来了，那我们就来看看要怎么监控，先写个监控程序：

```

package com.cbf4life.common;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 监控程序
 */
class Watch extends Thread{
    private HanFeiZi hanFeiZi;
    private LiSi liSi;
    private String type;

    //通过构造函数传递参数，我要监控的是谁，谁来监控，要监控什么
    public Watch(HanFeiZi _hanFeiZi, LiSi _liSi, String _type){
        this.hanFeiZi = _hanFeiZi;
        this.liSi = _liSi;
        this.type = _type;
    }

    @Override
    public void run(){
        while(true){
            if(this.type.equals("breakfast")){ //监控是否在吃早餐

```

```

        //如果发现韩非子在吃饭，就通知李斯
        if(this.hanFeiZi.isHaveBreakfast()){
            this.liSi.update("韩非子在吃饭");
            //重置状态，继续监控
            this.hanFeiZi.setHaveBreakfast(false);
        }
    }else{//监控是否在娱乐
        if(this.hanFeiZi.isHaveFun()){
            this.liSi.update("韩非子在娱乐");
            this.hanFeiZi.setHaveFun(false);
        }
    }
}
}
}
}

```

监控程序继承了 `java.lang.Thread` 类，可以同时启动多个线程进行监控，Java 的多线程机制还是比较简单的，继承 `Thread` 类，重写 `run()` 方法，然后 `new SubThread()`，再然后 `subThread.start()` 就可以启动一个线程了，我们继续往下看：

```

package com.cbf4life.common;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 这个Client就是我们，用我们的视角看待这段历史
 */
public class Client {

    public static void main(String[] args) throws InterruptedException {
        //定义出韩非子和李斯
        LiSi liSi = new LiSi();
        HanFeiZi hanFeiZi = new HanFeiZi();

        //观察早餐
        Watch watchBreakfast = new Watch(hanFeiZi,liSi,"breakfast");
        //开始启动线程，监控
        watchBreakfast.start();

        //观察娱乐情况
        Watch watchFun = new Watch(hanFeiZi,liSi,"fun");
        watchFun.start();
    }
}

```

```
//然后这里我们看看韩非子在干什么
Thread.sleep(1000); //主线程等待1秒后再往下执行
hanFeiZi.haveBreakfast();

//韩非子娱乐了
Thread.sleep(1000);
hanFeiZi.haveFun();

    }
}
```

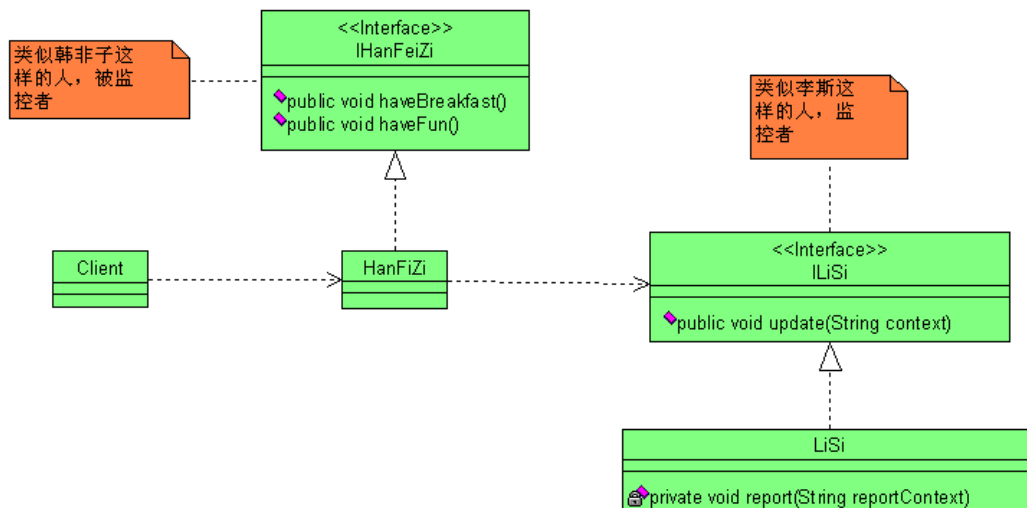
运行结果如下：

```
韩非子:开始吃饭了...
李斯:观察到李斯活动, 开始向老板汇报了...
李斯: 报告, 秦老板! 韩非子有活动了--->韩非子在吃饭
李斯: 汇报完毕, 秦老板赏给他两个萝卜吃吃...

韩非子:开始娱乐了...
李斯:观察到李斯活动, 开始向老板汇报了...
李斯: 报告, 秦老板! 韩非子有活动了--->韩非子在娱乐
李斯: 汇报完毕, 秦老板赏给他两个萝卜吃吃...
```

结果出来，韩非子一吃早饭李斯就知道，韩非子一娱乐李斯也知道，非常正确！结果正确但并不表示你有成绩，我告诉你：你的成绩是 0，甚至是负的，你有没有看到你的 CPU 飙升，Eclipse 不响应状态？看到了？看到了你还不想为什么？！看看上面的程序，别的就不多说了，使用了一个 while(true) 这样一个死循环来做监听，你要是用到项目中，你要多少硬件投入进来？你还让不让别人的程序也 run 起来？！一台服务器就跑你这一个程序就完事了，错，绝对的错！

错误也看到了，我们必须修改，这个没有办法应用到项目中去呀，而且这个程序根本就不是面向对象的程序，这完全是面向过程的（我写出这样的程序也不容易呀，安慰一下自己），不改不行，怎么修改呢？我们来想，既然韩非子一吃饭李斯就知道了，那我们为什么不把李斯这个类聚集到韩非子这里类上呢？说改就改，立马动手，我们来看修改后的类图：



类图非常简单，就是在 HanFeiZi 类中引用了 ILiSi 这个接口，看我们程序代码怎么修改，IHanFeiZi 接口完全没有修改，我们来看 HanFeiZi 这个实现类：

```

package com.cbf4life.advance;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 韩非子，李斯的师弟，韩国的重要人物
 */
public class HanFeiZi implements IHanFeiZi {
    //把李斯声明出来
    private ILiSi liSi = new LiSi();

    //韩非子要吃饭了
    public void haveBreakfast() {
        System.out.println("韩非子:开始吃饭了...");
        //通知李斯
        this.liSi.update("韩非子在吃饭");
    }

    //韩非子开始娱乐了,古代人没啥娱乐，你能想到的就那么多
    public void haveFun() {
        System.out.println("韩非子:开始娱乐了...");
        this.liSi.update("韩非子在娱乐");
    }
}

```

韩非子 HanFeiZi 实现类就把接口的两个方法实现就可以了，在每个方法中调用 LiSi.update() 方法，完成李斯观察韩非子任务，李斯的接口和实现类都没有任何改变，我们再来看看 Client 程序的变更：

```
package com.cbf4life.advance;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 这个Client就是我们，用我们的视角看待这段历史
 */
public class Client {

    public static void main(String[] args) {

        //定义出韩非子
        HanFeiZi hanFeiZi = new HanFeiZi();

        //然后这里我们看看韩非子在干什么
        hanFeiZi.haveBreakfast();

        //韩非子娱乐了
        hanFeiZi.haveFun();

    }
}
```

李斯都不用在 Client 中定义了，非常简单，运行结果如下：

```
韩非子:开始吃饭了...
李斯:观察到韩非子活动，开始向老板汇报了...
李斯:报告，秦老板！韩非子有活动了--->韩非子在吃饭
李斯:汇报完毕，秦老板赏给他两个萝卜吃吃...

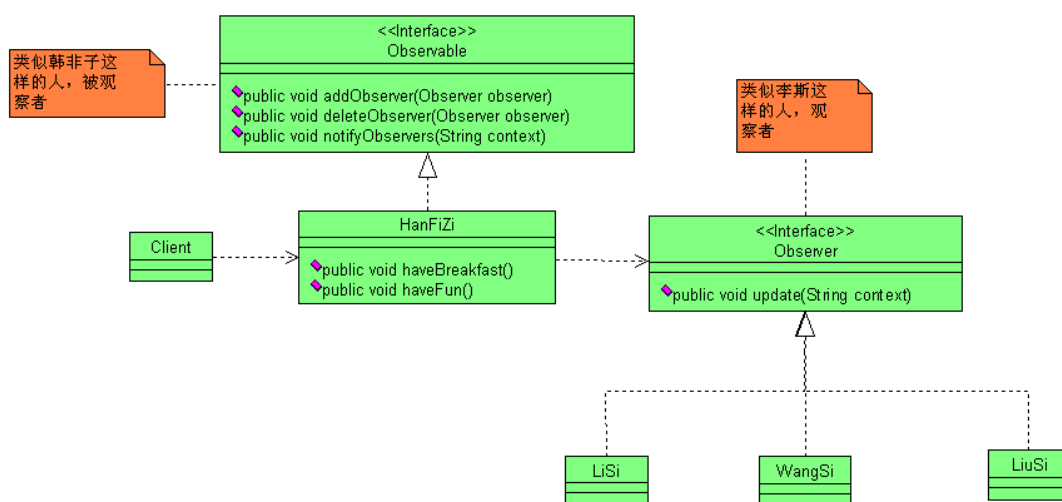
韩非子:开始娱乐了...
李斯:观察到韩非子活动，开始向老板汇报了...
李斯:报告，秦老板！韩非子有活动了--->韩非子在娱乐
李斯:汇报完毕，秦老板赏给他两个萝卜吃吃...
```

运行结果正确，效率也比较高，是不是应该乐呵乐呵了？大功告成了？稍等等，你想在战国争雄的时候，韩非子这么有名望（法家代表）、有实力（韩国的公子，他老爹参与过争夺韩国王位）的人，就只有秦

国一个国家关心他吗？想想也不可能呀，肯定有一大帮的各国的类似李斯这样的人在看着他，监视着一举一动，但是看看我们的程序，你在 HanFeiZi 这个类中定义：

```
private ILiSi liSi =new LiSi();
```

一下子就敲死了，只有李斯才能观察到韩非子，这是不对的，也就是说韩非子的活动只通知了李斯一个人，这不可能；再者，李斯只观察韩非子的吃饭，娱乐吗？政治倾向不关心吗？思维倾向不关心吗？杀人放火不关心吗？也就说韩非子的一系列活动都要通知李斯，那可怎么办？要按照上面的例子，我们不是要修改疯掉了吗？这和开闭原则严重违背呀，我们的程序有问题，怎么修改，来看类图：



我们把接口名称修改了一下，这样显得更抽象化，Observable 是被观察者，就是类似韩非子这样的人，Observer 接口是观察者，类似李斯这样的，同时还有其他国家的比如王斯、刘斯等，在 Observable 接口中有三个比较重要的方法，分别是 addObserver 增加观察者，deleteObserver 删除观察者，notifyObservers 通知所有的观察者，这是什么意思呢？我这里有一个信息，一个对象，我可以允许有多个对象来察看，你观察也成，我观察也成，只要是观察者就成，也就是说我的改变或动作执行，会通知其他的对象，看程序会更明白一点，先看 Observable 接口：

```
package com.cbf4life.advance2;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 所有被观察者者，通用接口
 */
public interface Observable {

    //增加一个观察者
}
```

```

    public void addObserver(Observer observer);

    //删除一个观察者，——我不想让你看了
    public void deleteObserver(Observer observer);

    //既然要观察，我发生改变了他也应该用所动作——通知观察者
    public void notifyObservers(String context);
}

```

这是一个通用的被观察者接口，所有的被观察者都可以实现这个接口。再来看韩非子的实现类：

```

package com.cbf4life.advance2;

import java.util.ArrayList;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 韩非子，李斯的师弟，韩国的重要人物
 */
public class HanFeiZi implements Observable{
    //定义个变长数组，存放所有的观察者
    private ArrayList<Observer> observerList = new ArrayList<Observer>();

    //增加观察者
    public void addObserver(Observer observer){
        this.observerList.add(observer);
    }

    //删除观察者
    public void deleteObserver(Observer observer){
        this.observerList.remove(observer);
    }

    //通知所有的观察者
    public void notifyObservers(String context){
        for(Observer observer:observerList){
            observer.update(context);
        }
    }

    //韩非子要吃饭了
    public void haveBreakfast(){

```



```

        System.out.println("韩非子:开始吃饭了...");
        //通知所有的观察者
        this.notifyObservers("韩非子在吃饭");
    }

    //韩非子开始娱乐了,古代人没啥娱乐,你能想到的就那么多
    public void haveFun(){
        System.out.println("韩非子:开始娱乐了...");
        this.notifyObservers("韩非子在娱乐");
    }
}

```

再来看观察者接口 Observer.java:

```

package com.cbf4life.advance2;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 所有观察者,通用接口
 */
public interface Observer {

    //一发现别人有动静,自己也要行动起来
    public void update(String context);
}

```

然后是三个很无耻的观察者,偷窥狂嘛:

```

package com.cbf4life.advance2;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 李斯这个人,是个观察者,只要韩非子一有动静,这边就知道
 */
public class LiSi implements Observer{

    //首先李斯是个观察者,一旦韩非子有活动,他就知道,他就要向老板汇报
    public void update(String str){

```

```

        System.out.println("李斯: 观察到李斯活动, 开始向老板汇报了...");
        this.reportToQiShiHuang(str);
        System.out.println("李斯: 汇报完毕, 秦老板赏给他两个萝卜吃吃...\n");
    }

    //汇报给秦始皇
    private void reportToQiShiHuang(String reportContext){
        System.out.println("李斯: 报告, 秦老板! 韩非子有活动了--->" + reportContext);
    }
}

```

李斯是真有其人, 以下两个观察者是杜撰出来的:

```

package com.cbf4life.advance2;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 王斯, 也是观察者, 杜撰的人名
 */
public class WangSi implements Observer{

    //王斯, 看到韩非子有活动, 自己就受不了
    public void update(String str){
        System.out.println("王斯: 观察到韩非子活动, 自己也开始活动了...");
        this.cry(str);
        System.out.println("王斯: 真真的哭死了...\n");
    }

    //一看李斯有活动, 就哭, 痛哭
    private void cry(String context){
        System.out.println("王斯: 因为" + context + ", 所以我悲伤呀!");
    }
}

package com.cbf4life.advance2;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 刘斯这个人, 是个观察者, 只要韩非子一有动静, 这边就知道
 * 杜撰的人名

```

```

*/
public class LiuSi implements Observer{

    //刘斯，观察到韩非子活动后，自己也做一定得事情
    public void update(String str){
        System.out.println("刘斯： 观察到韩非子活动，开始动作了...");
        this.happy(str);
        System.out.println("刘斯： 真被乐死了\n");
    }

    //一看韩非子有变化，他就快乐
    private void happy(String context){
        System.out.println("刘斯： 因为" +context+"，—所以我快乐呀！ " );
    }
}

```

所有的历史人物都在场了，那我们来看看这场历史闹剧是如何演绎的：

```

package com.cbf4life.advance2;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 这个Client就是我们，用我们的视角看待这段历史
 */
public class Client {

    public static void main(String[] args) {
        //三个观察者产生出来
        Observer liSi = new LiSi();
        Observer wangSi = new WangSi();
        Observer liuSi = new LiuSi();

        //定义出韩非子
        HanFeiZi hanFeiZi = new HanFeiZi();

        //我们后人根据历史，描述这个场景，有三个人在观察韩非子
        hanFeiZi.addObserver(liSi);
        hanFeiZi.addObserver(wangSi);
        hanFeiZi.addObserver(liuSi);

        //然后这里我们看看韩非子在干什么
        hanFeiZi.haveBreakfast();
    }
}

```

```

    }
}

```

运行结果如下：

```

韩非子:开始吃饭了...
李斯: 观察到李斯活动, 开始向老板汇报了...
李斯: 报告, 秦老板! 韩非子有活动了--->韩非子在吃饭
李斯: 汇报完毕, 秦老板赏给他两个萝卜吃吃...

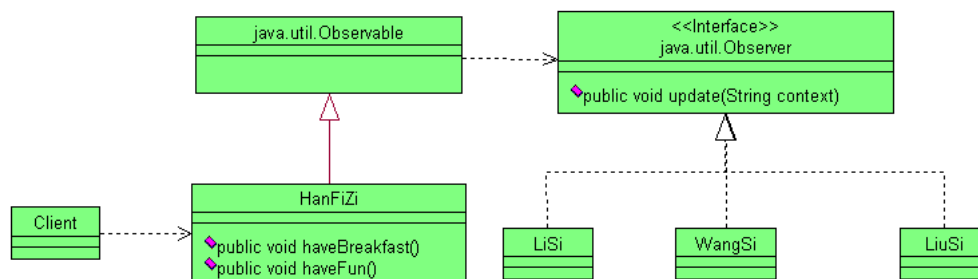
王斯: 观察到韩非子活动, 自己也开始活动了...
王斯: 因为韩非子在吃饭, —所以我悲伤呀!
王斯: 真真的哭死了...

刘斯: 观察到韩非子活动, 开始动作了...
刘斯: 因为韩非子在吃饭, —所以我快乐呀!
刘斯: 真被乐死了

```

好了，结果也正确了，也符合开闭原则了，也同时实现类间解耦，想再加观察者？好呀，继续实现 Observer 接口就成了，这时候必须修改 Client 程序，因为你业务都发生了变化。

细心的你可能已经发现，HanFeiZi 这个实现类中应该抽象出一个父类，父类完全实现接口，HanFeiZi 这个类只实现两个方法 haveBreakfast 和 haveFun 就可以了，是的，是的，确实是应该这样，那先稍等等，我们打开 JDK 的帮助文件看看，查找一下 Observable 是不是已经有这个类了？JDK 中提供了：java.util.Observable 实现类和 java.util.Observer 接口，也就是说我们上面写的那个例子中的 Observable 接口可以改换成 java.util.Observable 实现类了，看如下类图：



是不是又简单了很多？那就对了！然后我们看一下我们程序的变更，先看 HanFeiZi 的实现类：

```
package com.cbf4life.perfect;

import java.util.ArrayList;
import java.util.Observable;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 韩非子，李斯的师弟，韩国的重要人物
 */
public class HanFeiZi extends Observable{

    //韩非子要吃饭了
    public void haveBreakfast(){
        System.out.println("韩非子:开始吃饭了...");
        //通知所有的观察者
        super.setChanged();
        super.notifyObservers("韩非子在吃饭");
    }

    //韩非子开始娱乐了,古代人没啥娱乐，你能想到的就那么多
    public void haveFun(){
        System.out.println("韩非子:开始娱乐了...");
        super.setChanged();
        this.notifyObservers("韩非子在娱乐");
    }
}
```

改变的不多，引入了一个 `java.util.Observable` 对象，删除了增加、删除观察者的方法，简单了很多，那我们再来看观察者的实现类：

```
package com.cbf4life.perfect;

import java.util.Observable;
import java.util.Observer;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
```

```

* 李斯这个人，是个观察者，只要韩非子一有动静，这边就知道
*/
public class LiSi implements Observer{

    //首先李斯是个观察者，一旦韩非子有活动，他就知道，他就要向老板汇报
    public void update(Observable observable, Object obj){
        System.out.println("李斯：观察到李斯活动，开始向老板汇报了...");
        this.reportToQiShiHuang(obj.toString());
        System.out.println("李斯：汇报完毕，秦老板赏给他两个萝卜吃吃...\n");
    }

    //汇报给秦始皇
    private void reportToQiShiHuang(String reportContext){
        System.out.println("李斯：报告，秦老板！韩非子有活动了--->" + reportContext);
    }
}

```

就改变了黄色的部分，应该 java.util.Observer 接口要求 update 传递过来两个变量, Observable 这个变量我们没用到，就不处理了。其他两个观察者实现类也是相同的改动，如下代码：

```

package com.cbf4life.perfect;
import java.util.Observable;
import java.util.Observer;
/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 王斯，也是观察者，杜撰的人名
 */
public class WangSi implements Observer{

    //王斯，看到韩非子有活动，自己就受不了
    public void update(Observable observable, Object obj){
        System.out.println("王斯：观察到韩非子活动，自己也开始活动了...");
        this.cry(obj.toString());
        System.out.println("王斯：真真的哭死了...\n");
    }

    //一看李斯有活动，就哭，痛哭
    private void cry(String context){
        System.out.println("王斯：因为" + context + "，—所以我悲伤呀！");
    }
}

```

```

package com.cbf4life.perfect;

import java.util.Observable;
import java.util.Observer;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 刘斯这个人，是个观察者，只要韩非子一有动静，这边就知道
 * 杜撰的人名
 */
public class LiuSi implements Observer{

    //刘斯，观察到韩非子活动后，自己也做一定得事情
    public void update(Observable observable, Object obj){
        System.out.println("刘斯：观察到韩非子活动，开始动作了...");
        this.happy(obj.toString());
        System.out.println("刘斯：真被乐死了\n");
    }

    //一看韩非子有变化，他就快乐
    private void happy(String context){
        System.out.println("刘斯：因为" +context+"，—所以我快乐呀！" );
    }
}

```

然后再来看 Client 程序：

```

package com.cbf4life.perfect;

import java.util.Observer;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 这个Client就是我们，用我们的视角看待这段历史
 */
public class Client {

    public static void main(String[] args) {
        //三个观察者产生出来
    }
}

```

```
Observer liSi = new LiSi();
Observer wangSi = new WangSi();
Observer liuSi = new LiuSi();

//定义出韩非子
HanFeiZi hanFeiZi = new HanFeiZi();

//我们后人根据历史，描述这个场景，有三个人在观察韩非子
hanFeiZi.addObserver(liSi);
hanFeiZi.addObserver(wangSi);
hanFeiZi.addObserver(liuSi);

//然后这里我们看看韩非子在干什么
hanFeiZi.haveBreakfast();
}
}
```

程序体内没有任何变更，只是引入了一个接口而已，运行结果如下：

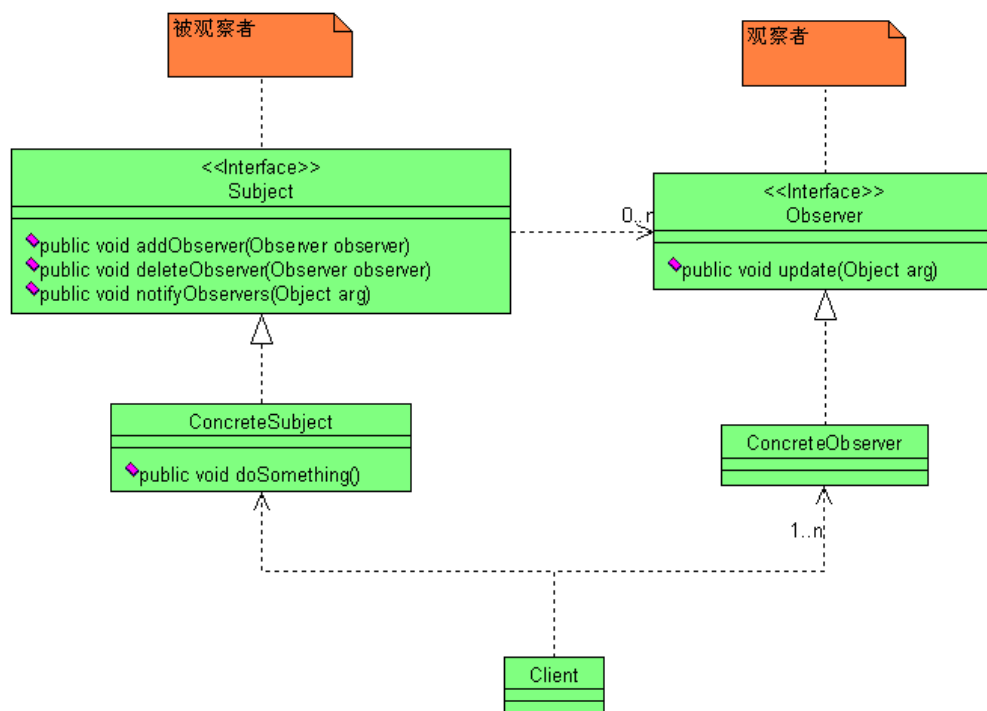
```
韩非子:开始吃饭了...
刘斯: 观察到韩非子活动, 开始动作了...
刘斯: 因为韩非子在吃饭, —所以我快乐呀!
刘斯: 真被乐死了

王斯: 观察到韩非子活动, 自己也开始活动了...
王斯: 因为韩非子在吃饭, —所以我悲伤呀!
王斯: 真真的哭死了...

李斯: 观察到李斯活动, 开始向老板汇报了...
李斯: 报告, 秦老板! 韩非子有活动了--->韩非子在吃饭
李斯: 汇报完毕, 秦老板赏给他两个萝卜吃吃...
```

运行结果一样，只是通知的先后顺序不同而已，程序已经简约到极致了。

以上讲解的就是观察者模式，这个模式的通用类图如下：



观察者模式在实际项目的应用中非常常见，比如你到 ATM 机器上取钱，多次输错密码，卡就会被 ATM 吞掉，吞卡动作发生的时候，会触发哪些事件呢？第一摄像头连续快拍，第二，通知监控系统，吞卡发生；第三，初始化 ATM 机屏幕，返回最初状态，你不能因为就吞了一张卡，整个 ATM 都不能用了吧，一般前两个动作都是通过观察者模式来完成的。

观察者模式有一个变种叫做发布/订阅模型 (Publish/Subscribe)，如果你做过 EJB (Enterprise JavaBean) 的开发，这个你绝对不会陌生。EJB2 是个折腾死人不偿命的玩意儿，写个 Bean 要实现，还要继承，再加上那一堆的配置文件，小项目还凑活，你要知道用 EJB 开发的基本上都不是小项目，到最后是每个项目成员都在骂 EJB 这个忽悠人的东西；但是 EJB3 是个非常优秀的框架，还是算比较轻量级，写个 Bean 只要加个 Annotation 就成了，配置文件减少了，而且也引入了依赖注入的概念，虽然只是 EJB2 的翻版，但是毕竟还是前进了一步，不知道以后 EJB 的路会怎么样。在 EJB 中有三个类型的 Bean: Session Bean、Entity Bean 和 MessageDriven Bean, 我们这里来说一下 MessageDriven Bean (一般简称为 MDB)，消息驱动 Bean，消息的发布者 (Provider) 发布一个消息，也就是一个消息驱动 Bean, 通过 EJB 容器（一般是 Message Queue 消息队列）通知订阅者做出回应，从原理上看很简单，就是观察者模式的升级版。

那观察者模式在什么情况下使用呢？观察者可以实现消息的广播，一个消息可以触发多个事件，这是观察者模式非常重要的功能。使用观察者模式也有两个重点问题要解决：

广播链的问题。如果你做过数据库的触发器，你就应该知道有一个触发器链的问题，比如表 A 上写了一个触发器，内容是一个字段更新后更新表 B 的一条数据，而表 B 上也有个触发器，要更新表 C，表 C 也有

触发器…, 完蛋了, 这个数据库基本上就毁掉了! 我们的观察者模式也是一样的问题, 一个观察者可以有双重身份, 即使观察者, 也是被观察者, 这没什么问题呀, 但是链一旦建立, 这个逻辑就比较复杂, 可维护性非常差, 根据经验建议, 在一个观察者模式中最多出现一个对象既是观察者也是被观察者, 也就是说消息最多转发一次 (传递两次), 这还是比较好控制的;

异步处理问题。这个 EJB 是一个非常好的例子, 被观察者发生动作了, 观察者要做出回应, 如果观察者比较多, 而且处理时间比较长怎么办? 那就用异步呗, 异步处理就要考虑线程安全和队列的问题, 这个大家有时间看看 Message Queue, 就会有更深的了解。

我们在来回顾一下我们写的程序, 观察者增加了, 我就必须修改业务逻辑 Client 程序, 这个是必须得吗? 回顾一下我们以前讲到工厂方法模式的时候用到了 ClassUtils 这个类, 其中有一个方法就是根据接口查找到所有的实现类, 问题解决了! 我可以查找到所有的观察者, 然后全部加进来, 以后要是新增加观察者也没有问题呀, 程序那真是一点都不用改了!

第 17 章 责任链模式【Chain of Responsibility Pattern】

中国古代对妇女制定了“三从四德”的道德规范,“三从”是指“未嫁从父、既嫁从夫、夫死从子”, 也就是说一个女性, 在没有结婚的时候要听从于父亲, 结了婚后听从于丈夫, 丈夫死了还要听儿子的, 举个例子来说, 一个女的要出去逛街, 同样这样的一个请求, 在她没有出嫁前她必须征得父亲的同意, 出嫁之后必须获得丈夫的许可, 那丈夫死了怎么办? 一般都是男的比女的死的早, 还要问问儿子是否允许自己出去逛街, 估计你下边马上要问要是没有儿子怎么办? 请示小叔子、侄子等等, 在父系社会中, 妇女只占从属地位, 现在想想中国的妇女还是比较悲惨的, 逛个街还要请示来请示去, 而且作为父亲、丈夫、儿子只有两种选择: 要不承担起责任来告诉她允许或不允许逛街, 要不就让她请示下一个人, 这是整个社会体系的约束, 应用到我们项目中就是业务规则, 那我们来看怎么把“三从”通过我们的程序来实现, 需求很简单: 通过程序描述一下古代妇女的“三从”制度, 好我们先看类图:



非常简单的类图，这个实现也非常简单，我们先看 IWomen 接口：

```
package com.cbf4life.common;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 古代悲哀女性的总称
 */
public interface IWomen {

    //获得个人状况
    public int getType();

    //获得个人请示，你要干什么？出去逛街？约会？还是看电影
    public String getRequest();
}
```

女性就两个参数，一个是当前的个人状况，是结婚了呢还是没结婚，丈夫有没有去世，另外一个是要请示的内容，要出去逛街呀还是吃饭，我们看实现类：

```
package com.cbf4life.common;

/**
 * @author cbf4Life cbf4life@126.com
```

```

* I'm glad to share my knowledge with you all.
* 古代女性的总称
*/
public class Women implements IWomen{
    /*
    * 通过一个int类型的参数来描述妇女的个人状况
    * 1---未出嫁
    * 2---出嫁
    * 3---夫死
    */
    private int type=0;

    //妇女的请示
    private String request = "";

    //构造函数传递过来请求
    public Women(int _type,String _request){
        this.type = _type;
        this.request = _request;
    }

    //获得自己的状况
    public int getType(){
        return this.type;
    }

    //获得妇女的请求
    public String getRequest(){
        return this.request;
    }
}

```

我们使用数字来代表女性的不同状态，1 是未结婚，2 是已经结婚的，而且丈夫建在，3 是丈夫去世了。我们再来看有处理权的人员接口：

```

package com.cbf4life.common;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 父系社会，那就是男性有至高权利，handler控制权
 */

```

```
public interface IHandler {  
  
    //一个女性（女儿，妻子或者是母亲）要求逛街，你要处理这个请求  
    public void HandleMessage(IWomen women);  
  
}
```

父亲、丈夫、儿子都是这个 IHandler 接口的实现者：

```
package com.cbf4life.common;  
  
/**  
 * @author cbf4Life cbf4life@126.com  
 * I'm glad to share my knowledge with you all.  
 * 父亲  
 */  
public class Father implements IHandler {  
  
    //未出嫁女儿来请示父亲  
    public void HandleMessage(IWomen women) {  
        System.out.println("女儿的请示是: "+women.getRequest());  
        System.out.println("父亲的答复是: 同意");  
    }  
  
}  
  
package com.cbf4life.common;  
  
/**  
 * @author cbf4Life cbf4life@126.com  
 * I'm glad to share my knowledge with you all.  
 * 丈夫类  
 */  
public class Husband implements IHandler {  
  
    //妻子向丈夫请示  
    public void HandleMessage(IWomen women) {  
        System.out.println("妻子的请示是: "+women.getRequest());  
        System.out.println("丈夫的答复是: 同意");  
    }  
  
}
```

```
package com.cbf4life.common;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 儿子类
 */
public class Son implements IHandler {

    //目前向儿子请示
    public void HandleMessage(IWomen women) {
        System.out.println("母亲的请示是: "+women.getRequest());
        System.out.println("儿子的答复是: 同意");
    }

}
```

这三个类非常非常的简单，就一个方法，处理女儿、妻子、母亲提出的请求，再来看 Client 是怎么组装的：

```
package com.cbf4life.common;

import java.util.ArrayList;
import java.util.Random;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 我们后人来看这样的社会道德
 */
@SuppressWarnings("all")
public class Client {

    public static void main(String[] args) {
        //随机挑选几个女性
        Random rand = new Random();
        ArrayList<IWomen> arrayList = new ArrayList();
        for(int i=0;i<5;i++){
            arrayList.add(new Women(rand.nextInt(4),"我要出去逛街"));
        }
    }
}
```

```

//定义三个请示对象
IHandler father = new Father();
IHandler husband = new Husband();
IHandler son = new Son();

for(IWomen women:arrayList){
    if(women.getType() ==1){ //未婚少女，请示父亲
        System.out.println("\n-----女儿向父亲请示-----");
        father.HandleMessage(women);
    }else if(women.getType() ==2){ //已婚少妇，请示丈夫
        System.out.println("\n-----妻子向丈夫请示-----");
        husband.HandleMessage(women);
    }else if(women.getType() == 3){ //母亲请示儿子
        System.out.println("\n-----母亲向儿子请示-----");
        son.HandleMessage(women);
    }else{
        //暂时啥也不做
    }
}
}
}

```

首先是通过随机方法产生了 5 个古代妇女的对象，然后看她们是如何就逛街这件事去请示的，运行结果如下：（由于是随机的，您看到得结果可能和这里有所不同）

```

-----女儿向父亲请示-----
女儿的请示是：我要出去逛街
父亲的答复是：同意

-----妻子向丈夫请示-----
妻子的请示是：我要出去逛街
丈夫的答复是：同意

-----妻子向丈夫请示-----
妻子的请示是：我要出去逛街
丈夫的答复是：同意

-----女儿向父亲请示-----
女儿的请示是：我要出去逛街
父亲的答复是：同意

```

“三从四德”的旧社会规范已经完整的表现出来了，你看谁向谁请示都定义出来了，但是你是不是发现这个程序写的有点不舒服？有点别扭？有点想重构它的感觉？那就对了！这段代码有以下几个问题：

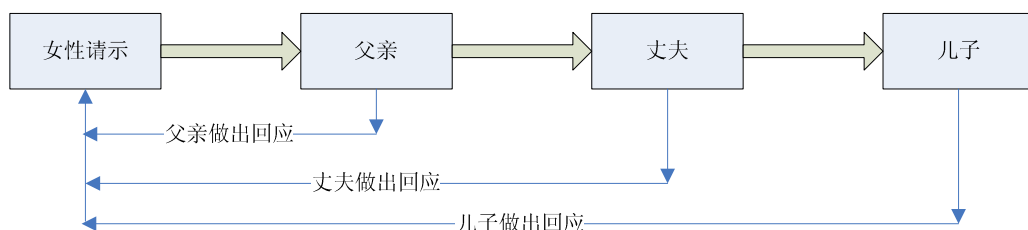
失去面向对象的意义。对女儿提出的请示，应该在父亲类中做出决定，父亲这个类应该是知道女儿的请求应该自己处理，而不是在 Client 类中进行组装出来，也就是说原本应该是父亲这个类做的事情抛给了其他类进行处理；

迪米特法则相违背。我们在 Client 类中写了 if...eles 的判断条件，你看这个条件体内都是一个接口 IHandler 的三个实现类，谁能处理那个请求，怎么处理，直接在实现类中定义好不就结了吗？你的类我知道的越少越好，别让我猜测你类中的逻辑，想想看，把这段 if...else 移动到三个实现类中该怎么做？

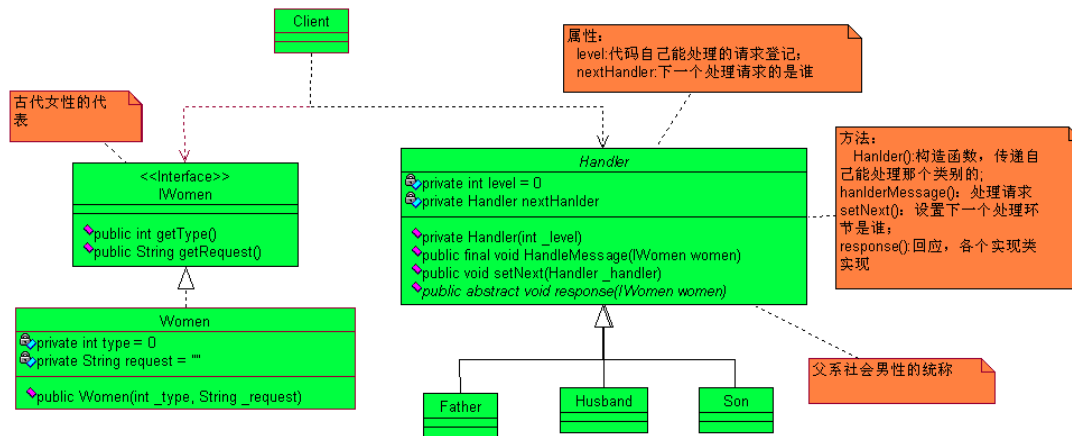
耦合过重。这个什么意思呢，我们要根据 Women 的 type 来决定使用 IHandler 的那个实现类来处理请求，我问你，如果 IHandler 的实现类继续扩展怎么办？修改 Client 类？与开闭原则违背喽！

异常情况没有考虑。妻子只能向丈夫请示吗？如果妻子向自己的父亲请示了，父亲应该做何处理？我们的程序上可没有体现出来。

既然有这么多的问题，那我们要想办法来解决这些问题，我们可以抽象成这样一个结构，女性的请求先发送到父亲类，父亲类一看是自己要处理的，就回应处理，如果女儿已经出嫁了，那就要把这个请求转发到女婿来处理，那女婿一旦去天国报道了，那就由儿子来处理这个请求，类似于这样请求：



父亲、丈夫、儿子每个节点有两个选择：要么承担责任，做出回复；要么把请求转发到后序环节。结构分析的已经清楚了，那我们看怎么来实现这个功能，先看类图：



从类图上看，三个实现类 Father、Husband、Son 只要实现构造函数和父类中的抽象方法就可以了，具体怎么处理这些请求，都已经转移到了 Handler 抽象类中，我们来看 Handler 怎么实现：

```

package com.cbf4life.advance;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 父系社会，那就是男性有至高权利，handler控制权
 */
public abstract class Handler {
    //能处理的级别
    private int level = 0;

    //责任传递，下一个人责任人是谁
    private Handler nextHandler;

    //每个类都要说明一下自己能处理哪些请求
    public Handler(int _level){
        this.level = _level;
    }

    //一个女性（女儿，妻子或者是母亲）要求逛街，你要处理这个请求
    public final void handleMessage(IWomen women){
        if(women.getType() == this.level){
            this.response(women);
        }else{
            if(this.nextHandler != null){ //有后续环节，才把请求往后递送
                this.nextHandler.handleMessage(women);
            }else{ //已经没有后续处理人了，不用处理了
                System.out.println("-----没地方请示了，不做处理!-----\n");
            }
        }
    }
}
  
```

```

        }

    }
}

/*
 * 如果你属于你处理的返回，你应该让她找下一个环节的人，比如
 * 女儿出嫁了，还向父亲请示是否可以逛街，那父亲就应该告诉女儿，应该找丈夫请示
 */
public void setNext(Handler _handler){
    this.nextHanlder = _handler;
}

//有请示那当然要回应
public abstract void response(IWomen women);
}

```

有没有看到，其实在这里也用到模版方法模式，在模版方法中判断请求的级别和当前能够处理的级别，如果相同则调用基本方法，做出反馈；如果不相等，则传递到下一个环节，由下一环节做出回应。基本方法 response 要各个实现类都要实现，我们来看三个实现类：

```

package com.cbf4life.advance;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 父亲
 */
public class Father extends Handler {
    //父亲只处理女儿的请求
    public Father(){
        super(1);
    }

    //父亲的答复
    @Override
    public void response(IWomen women) {
        System.out.println("-----女儿向父亲请示-----");
        System.out.println(women.getRequest());
        System.out.println("父亲的答复是：同意\n");
    }
}

```

```
}

package com.cbf4life.advance;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 丈夫类
 */
public class Husband extends Handler {
    //丈夫只处理妻子的请求
    public Husband(){
        super(2);
    }

    //丈夫请示的答复
    @Override
    public void response(IWomen women) {
        System.out.println("-----妻子向丈夫请示-----");
        System.out.println(women.getRequest());
        System.out.println("丈夫的答复是：同意\n");
    }
}

package com.cbf4life.advance;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 儿子类
 */
public class Son extends Handler {
    //儿子只处理母亲的请求
    public Son(){
        super(3);
    }

    //儿子的答复
    public void response(IWomen women) {
        System.out.println("-----母亲向儿子请示-----");
        System.out.println(women.getRequest());
        System.out.println("儿子的答复是：同意\n");
    }
}
```

```
}
```

这三个类都很简单，构造方法那是你必须实现的，父类已经定义了，子类不实现编译不通过，通过构造方法我们设置了各个类能处理的请求类型，Father 只能处理请求类型为 1（也就是女儿）的请求；Husband 只能处理请求类型类 2（也就是妻子）的请求；儿子只能处理请求类型为 3（也就是目前）的请求。

Women 类的接口没有任何变化，实现类少有变化，看代码：

```
package com.cbf4life.advance;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 古代女性的总称
 */
public class Women implements IWomen{
    /*
     * 通过一个int类型的参数来描述妇女的个人状况
     * 1---未出嫁
     * 2---出嫁
     * 3---夫死
     */
    private int type=0;

    //妇女的请示
    private String request = "";

    //构造函数传递过来请求
    public Women(int _type,String _request){
        this.type = _type;

        //为了显示好看点，我在这里做了点处理
        switch(this.type){
            case 1:
                this.request = "女儿的请求是: " + _request;
                break;
            case 2:
                this.request = "妻子的请求是: " + _request;
                break;
            case 3:
                this.request = "母亲的请求是: " + _request;
```

```

    }

    //获得自己的状况
    public int getType(){
        return this.type;
    }

    //获得妇女的请求
    public String getRequest(){
        return this.request;
    }
}

```

就是为了展示结果清晰一点，Women 类做了一点改变，看黄色部分。我们再来看 Client 类是怎么描述古代这一个礼节的：

```

package com.cbf4life.advance;

import java.util.ArrayList;
import java.util.Random;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 我们后人来看这样的社会道德
 */
@SuppressWarnings("all")
public class Client {

    public static void main(String[] args) {
        //随机挑选几个女性
        Random rand = new Random();
        ArrayList<IWomen> arrayList = new ArrayList();
        for(int i=0;i<5;i++){
            arrayList.add(new Women(rand.nextInt(4), "我要出去逛街"));
        }

        //定义三个请示对象
        Handler father = new Father();
        Handler husband = new Husband();
        Handler son = new Son();
    }
}

```

```
//设置请示顺序
father.setNext(husband);
husband.setNext(son);

for(IWomen women:arrayList){
    father.HandleMessage(women);
}

}

}
```

通过在 Client 中设置请求的传递顺序，解决了请求到底谁来回应的问题，运行结果如下：

```
-----妻子向丈夫请示-----
妻子的请求是：我要出去逛街
丈夫的答复是：同意

-----女儿向父亲请示-----
女儿的请求是：我要出去逛街
父亲的答复是：同意

-----母亲向儿子请示-----
母亲的请求是：我要出去逛街
儿子的答复是：同意

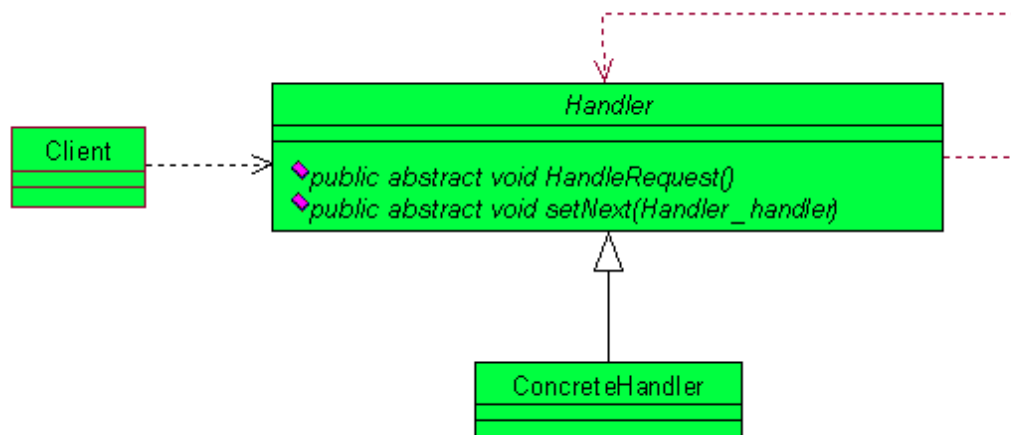
-----妻子向丈夫请示-----
妻子的请求是：我要出去逛街
丈夫的答复是：同意

-----母亲向儿子请示-----
母亲的请求是：我要出去逛街
儿子的答复是：同意
```

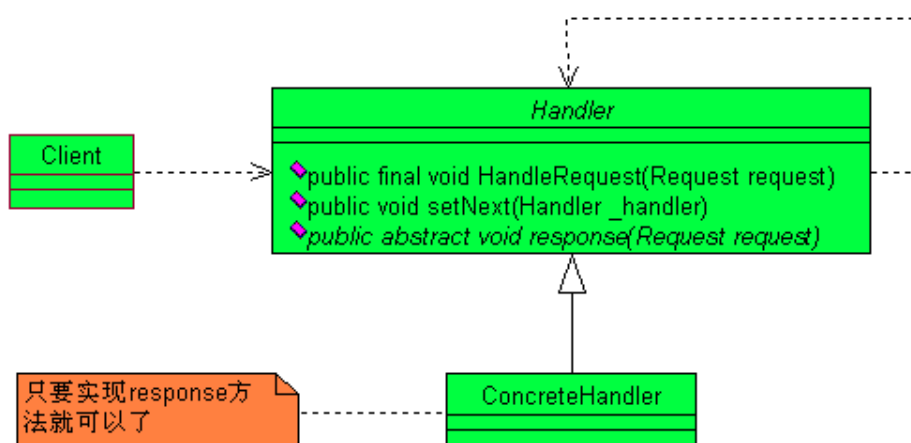
结果也正确，业务调用类 Client 也不用去做判断到底是需要谁去处理，而且 Handler 抽象类的子类以后可以继续增加下去，只是我们这个传递链增加而已，调用类可以不用了解变化过程，甚至是谁在处理这个请求都不用知道。

以上讲解的就是责任链模式，你看 Father、Husband、Son 这三个类的处理女性的请求时是不是在传递呀，每个环节只有两个选项：要么承担责任做出回应，要么向下传递请求，最终会有环节做出回应，通用

类图如下：



在通用类图中 **Handler** 是一个接口或者是抽象类，每个实现类都有两个方法 `HandleRequest` 是处理请求，`setNext` 是设置当前环节怎么把不属于自己处理的请求扔给谁，对于这个类图我觉得需要改变，融合进来模版方法模式，类图如下：



想想单一职责法则和迪米特法则吧，通过融合模版方法模式，各个实现类只要关注的自己业务逻辑就成了，至于说什么事要自己处理，那就让父类去决定好了，也就是说父类实现了请求传递的功能，子类实现请求的处理，符合单一职责法则，各个类只作一个动作或逻辑，也就是只有一个原因引起类的改变，我建议大家在使用的时候用这种方法，好处是非常明显的了，子类的实现非常简单，责任链的建立也非常的灵活。

这里顺便插一句话，在论坛上有网友不赞成我这种写法，说是没有抓住 XX 模式的核心，我想请问你一下，XX 模式的核心是什么？就拿今天讲的责任链模式来说，GOF 是这样说的：

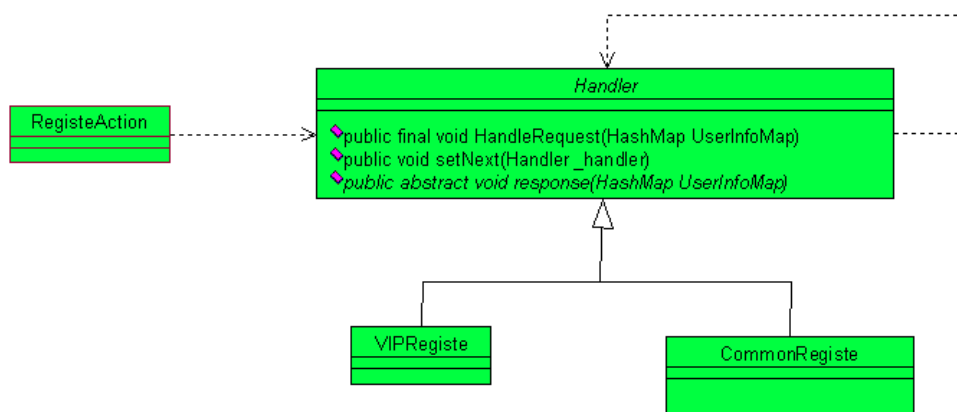
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

这是 GOF 的原话，我想请问大侠，告诉我这句话是什么意思，如果你来给读者讲解，你想怎么讲？翻译成中文？拿着彦 X 的那本书给读者讲？照本宣科？你觉的还有人来看吗？模式是在经验中积累的，是知识的结晶，而不是死板硬套的模子！不要因为一些写模式的书比较流行就膜拜了，自己动脑筋想想这模式真是就只能是这样吗？是不是有更优秀的方式来替代呢？别照抄别人的！

责任链模式屏蔽了请求的处理过程，你发起一个请求到底是谁处理的，这个你不用关心，只要你把请求抛给责任链的第一个处理者，最终会返回一个处理结果（当然也可以不做任何处理），作为请求者可以不用知道到底是需要谁来处理的，这是责任链模式的核心；同时责任链模式也可以做为一种补救模式来使用，举个简单例子，如项目开发的时候，需求确认是这样的：一个请求（比如银行客户存款的币种），一个处理者（只处理人民币），但是随着业务的发展（改革开放了嘛，还要处理美元、日元等等），处理者的数量和类型都有所增加，那这时候就可以在第一个处理者后面建立一个链，也就是责任链来处理请求，你是人民币，好，还是第一个业务逻辑来处理，你是美元，好，传递到第二个业务逻辑来处理，日元，欧元…，这些都不用在对原有的业务逻辑产生很大改变，通过扩展实现类就可以很好的解决这些需求变更的问题。

责任链有一个缺点是大家在开发的时候要注意：调试不是很方便，特别是链条比较长，环节比较多的时候，由于采用了类似递归的方式，调试的时候逻辑可能比较复杂。

责任链在实际的项目中使用也是比较多的，我曾经做过这样一个项目，界面上有一个用户注册功能，注册用户分两种，一种是 VIP 用户，也就是在该单位办理过业务的，一种是普通用户，一个用户的注册要填写一堆信息，VIP 用户只比普通用户多了一个输入项：VIP 序列号，注册后还需要激活，VIP 和普通用户的激活流程也是不同的，VIP 是自动发送邮件到用户的邮箱中就算激活了，普通用户要发送短信才能激活，为什么呢？获得手机号码以后好发广告短信呀！这个功能项目组就采用了责任链模式，甭管从前台传递过来的是 VIP 用户信息还是普通用户信息，统一传递到一个处理入口，通过责任链来完成任务的处理，类图如下：



其中 RegisterAction 是继承了 Strust2 中的 ActionSupport，实现 HTTP 传递过来的对象组装，组装出一个 HashMap 对象 UserInfoMap，传递给 handler 的两个实现类，具体是那个实现类来处理的，就由 HashMap 上的用户标识来做决定了，这个和上面我们举的例子很类似，代码大家自己实现。

还有一个问题需要和大家说明一下，观察者模式也可以实现请求的传递，比如一个事件发生了，通知了观察者，同时观察者又作为一个被观察者，通知了另外一个观察者，这也形成了一个事件广播链，这和我们今天讲的责任链是有区别的：

受众数量不同。观察者广播链式可以 1: N 的方式广播，而责任链则要求是的 1:1 的传递，必然有一个且只有一个类完成请求的处理；

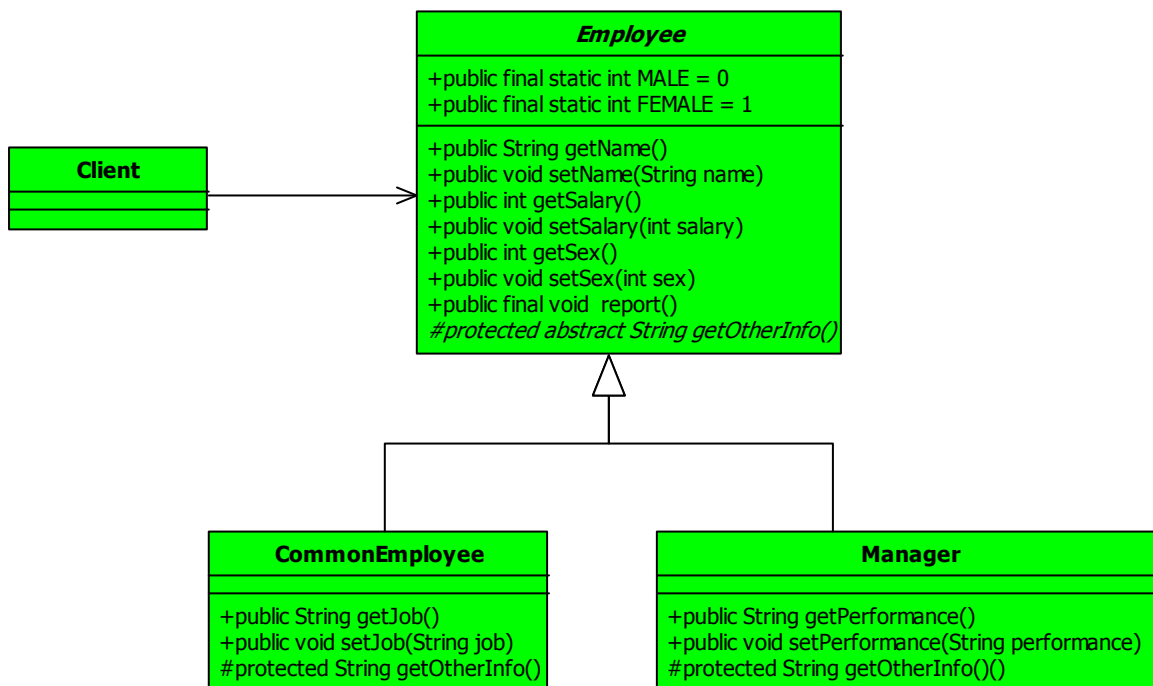
请求内容不同。观察者广播链中的信息可以在传播中改变，但是责任链中的请求是不可改变的；

处理逻辑不通。观察者广播链主要用于触发联动动作，而责任链则是对一个类型的请求按照既定的规则进行处理。

第 18 章 访问者模式【Visitor Pattern】

今天天气不错，绝对是晴空万里，骄阳似火呀，好，我们今天来讲访问者模式，我们在前面讲了组合模式和迭代器模式，通过组合模式我们能够把一个公司的人员组织机构树搭建起来，给管理带来非常大的便利，通过迭代器模式我们可以把每一个员工都遍历一遍，看看是不是有“人去世了还在领退休金”，“拿高工资而不干活的尸位素餐”等情况，那我们今天的要讲访问者模式是做什么用的呢？

我们公司有七百多技术人员，分布在全国各地，组织架构你在组合模式中也看到了，很常见的家长领导型模式，每个技术人员的岗位都是固定的，你在组织机构在那棵树下，充当的是什么叶子节点都是非常明确的，每一个员工的信息比如名字、性别、薪水等都是记录在数据库中，现在有这样一个需求，我要把公司中的所有人员信息都打印汇报上去，很简单吧，我们来看类图：



这个类图还是比较简单的，使用了一个模版方法模式，把所要的信息都打印出来，我们先来看一下抽象类：

```
package com.cbf4life.common;
```

```
/**
```

```
 * @author cbf4Life cbf4life@126.com
```

```
 * I'm glad to share my knowledge with you all.
```

```
 * 在一个单位里谁都是员工，甭管你是部门经理还是小兵
```

```
*/
public abstract class Employee {
    public final static int MALE = 0; //0代表是男性
    public final static int FEMALE = 1; //1代表是女性
    //甭管是谁，都有工资
    private String name;

    //只要是员工那就有薪水
    private int salary;

    //性别很重要
    private int sex;

    //以下是简单的getter/setter，不多说
    public String getName() {
        return name;
    }

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

    public int getSalary() {
        return salary;
    }

    public void setSalary(int salary) {
        this.salary = salary;
    }

    public int getSex() {
        return sex;
    }

    public void setSex(int sex) {
        this.sex = sex;
    }

    //打印出员工的信息
    public final void report(){
        String info = "姓名: " + this.name + "\t";
        info = info + "性别: " + (this.sex == FEMALE?"女":"男") + "\t";
        info = info + "薪水: " + this.salary + "\t";
    }
}
```

```

        //获得员工的其他信息
        info = info + this.getOtherInfo();
        System.out.println(info);
    }

    //拼装员工的其他信息
    protected abstract String getOtherInfo();
}

```

再看小兵的实现类，越卑微的人物越能引起共鸣，因为我们有共同的经历、思维和苦难，呵呵，看实现类：

```

package com.cbf4life.common;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 普通员工，也就是最小的小兵
 */
public class CommonEmployee extends Employee {

    //工作内容，这个非常重要，以后的职业规划就是靠这个了
    private String job;

    public String getJob() {
        return job;
    }

    public void setJob(String job) {
        this.job = job;
    }

    protected String getOtherInfo(){
        return "工作: " + this.job + "\t";
    }

}

```

在来看领导阶层：

```
package com.cbf4life.common;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 经理级人物
 */
public class Manager extends Employee {

    //这类人物的职责非常明确：业绩
    private String performance;

    public String getPerformance() {
        return performance;
    }

    public void setPerformance(String performance) {
        this.performance = performance;
    }

    protected String getOtherInfo(){
        return "业绩: " + this.performance + "\t";
    }
}
```

Performance 这个单词在我们技术人员的眼里就是代表性能，在实际商务英语中可以有 Sales Performance 销售业绩，performance evaluation 业绩评估等等。然后我们来看一下我们的 invoker 类：

```
package com.cbf4life.common;

import java.util.ArrayList;
import java.util.List;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 */
public class Client {

    public static void main(String[] args) {
        for(Employee emp:mockEmployee()){

```

```
        emp.report();
    }
}

//模拟出公司的人员情况，我们可以想象这个数据室通过持久层传递过来的
public static List<Employee> mockEmployee(){
    List<Employee> empList = new ArrayList<Employee>();

    //产生张三这个员工
    CommonEmployee zhangSan = new CommonEmployee();
    zhangSan.setJob("编写Java程序，绝对的蓝领、苦工加搬运工");
    zhangSan.setName("张三");
    zhangSan.setSalary(1800);
    zhangSan.setSex(Employee.MALE);
    empList.add(zhangSan);

    //产生李四这个员工
    CommonEmployee liSi = new CommonEmployee();
    liSi.setJob("页面美工，审美素质太不流行了！");
    liSi.setName("李四");
    liSi.setSalary(1900);
    liSi.setSex(Employee.FEMALE);
    empList.add(liSi);

    //再产生一个经理
    Manager wangWu = new Manager();
    wangWu.setName("王五");
    wangWu.setPerformance("基本上是负值，但是我会拍马屁呀");
    wangWu.setSalary(18750);
    wangWu.setSex(Employee.MALE);
    empList.add(wangWu);

    return empList;
}
}
```

先通过 mockEmployee 来模拟出一个数组，当然了在实际项目中这个数组应该是从持久层产生过来的。

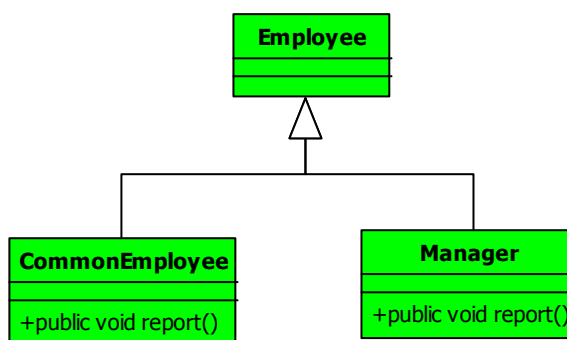
我们来看运行结果：

```
姓名：张三   性别：男 薪水：1800  工作：编写Java程序，绝对的蓝领、苦工加搬运工
姓名：李四   性别：女 薪水：1900  工作：页面美工，审美素质太不流行了！
姓名：王五   性别：男 薪水：18750 业绩：基本上是负值，但是我会拍马屁呀
```

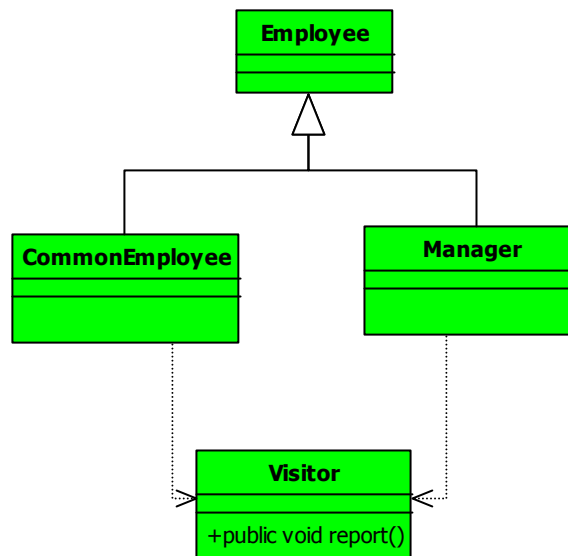
结果出来了，也非常正确。我们来想一想这个实际的情况，人力资源部门拿这份表格会给谁看呢？那当然是大老板了，大老板关心的是什么？关心部门经理的业绩！小兵的情况不是他要了解的，就像二战的时候一位将军（巴顿？艾森豪威尔？记不清楚了）说的“我一想到我的士兵也有孩子、妻子、父母，我就痛心疾首，…但是这是战场，我只能认为他们是一群机器…”，是呀，其实我们也一样呀，那问题就出来了：

- ◆ 大老板就看部门经理的报表，小兵的报表可看可不看；
- ◆ 多个大老板，“嗜好”是不同的，主管销售的，则主要关心的营销情况；主管会计的，则主要关心企业的整体财务运行状态；主管技术的，则主要看技术的研发情况；

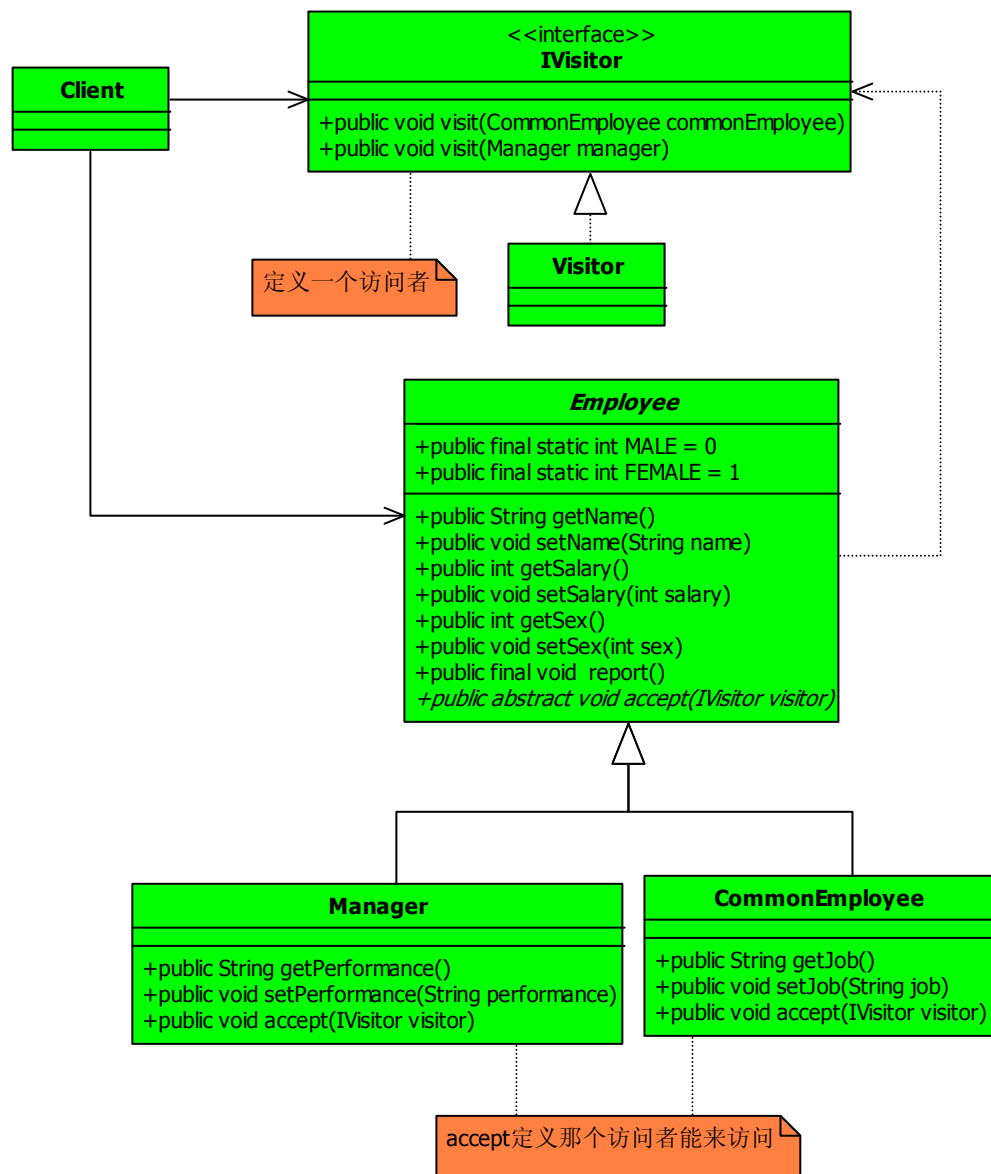
综合成一句话，这个报表会有修改：数据的修改以及报表的展现修改，按照开闭原则，项目分析的时候已经考虑到这些可能引起变更的隐私，就需要在设计时考虑通过扩展来避开未来需求变更而引起的代码修改风险。我们来想一想，每个普通员工类和经理类都有一个方法 `report`，那是否可以把这个方法提取到另外一个类中来实现呢？原有的示意图如下：



这两个类都有一个相同的方法 `report()`，但是要实现的内容不相同，而且还有可能会发生变动，那就让其他类来实现这个 `report` 方法，好，看示意图图的变更：



两个类的 report 方法都不需要了，只有 Visitor 类来实现了 report 的方法，这个猛一看还真有点委托（intergration）的意味，我们实现下来你就知道这和委托有非常大的差距，我们来看类图：



在抽象类 `Employee` 中增加了 `accept` 方法，这个方法是定义我这个类可以允许被谁来访问，也就定义一类访问者，在具体的实现类中调用访问者的方法。我们先看访问者接口 `IVisitor` 程序：

```
package com.cbf4life.common2;
```

```
/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 访问者，我要去访问人家的数据了
 */
public interface IVisitor {
```

```

//首先定义我可以访问普通员工
public void visit(CommonEmployee commonEmployee);

//其次定义，我还可以访问部门经理
public void visit(Manager manager);

}

```

如下是访问者的实现类：

```

package com.cbf4life.common2;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 */
public class Visitor implements IVisitor {

    //访问普通员工，打印出报表
    public void visit(CommonEmployee commonEmployee) {
        System.out.println(this.getCommonEmployee(commonEmployee));
    }

    //访问部门经理，打印出报表
    public void visit(Manager manager) {
        System.out.println(this.getManagerInfo(manager));
    }

    //组装出基本信息
    private String getBasicInfo(Employee employee){
        String info = "姓名: " + employee.getName() + "\t";
        info = info + "性别: " + (employee.getSex() == Employee.FEMALE?"女":"男")
+ "\t";
        info = info + "薪水: " + employee.getSalary() + "\t";

        return info;
    }

    //组装出部门经理的信息
    private String getManagerInfo(Manager manager){
        String basicInfo = this.getBasicInfo(manager);
        String otherInfo = "业绩: "+manager.getPerformance() + "\t";
    }
}

```

```

        return basicInfo + otherInfo;
    }

    //组装出普通员工信息
    private String getCommonEmployee(CommonEmployee commonEmployee){
        String basicInfo = this.getBasicInfo(commonEmployee);
        String otherInfo = "工作: "+commonEmployee.getJob()+"\t";
        return basicInfo + otherInfo;
    }
}

```

在具体的实现类中，定义了两个私有方法，作用就是产生需要打印的数据和格式，然后在访问者访问相关的对象是，产生这个报表。继续看 Employee 抽象类：

```

package com.cbf4life.common2;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 在一个单位里谁都是员工，甭管你是部门经理还是小兵
 */
public abstract class Employee {
    public final static int MALE = 0; //0代表是男性
    public final static int FEMALE = 1; //1代表是女性
    //甭管是谁，都有工资
    private String name;

    //只要是员工那就有薪水
    private int salary;

    //性别很重要
    private int sex;

    //以下是简单的getter/setter，不多说
    public String getName() {
        return name;
    }

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

```

```

    public int getSalary() {
        return salary;
    }

    public void setSalary(int salary) {
        this.salary = salary;
    }

    public int getSex() {
        return sex;
    }

    public void setSex(int sex) {
        this.sex = sex;
    }

    //我允许一个访问者过来访问
    public abstract void accept(IVisitor visitor);
}

```

删除了 report 方法，增加了 accept 方法，需要实现类来实现。继续看实现类：

```

package com.cbf4life.common2;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 普通员工，也就是最小的小兵
 */
public class CommonEmployee extends Employee {

    //工作内容，这个非常重要，以后的职业规划就是靠这个了
    private String job;

    public String getJob() {
        return job;
    }

    public void setJob(String job) {
        this.job = job;
    }

    //我允许访问者过来访问

```

```

    @Override
    public void accept(IVisitor visitor){
        visitor.visit(this);
    }
}

```

上面是普通员工的实现类，这个实现类的 accept 方法很简单，这个类就把自身传递过去，也就是让访问者访问本身这个对象。再看 Manager 类：

```

package com.cbf4life.common2;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 经理级人物
 */
public class Manager extends Employee {

    //这类人物的职责非常明确：业绩
    private String performance;

    public String getPerformance() {
        return performance;
    }

    public void setPerformance(String performance) {
        this.performance = performance;
    }

    //部门经理允许访问者访问
    @Override
    public void accept(IVisitor visitor){
        visitor.visit(this);
    }
}

```

所有的业务定义都已经完成，我们来看看怎么调用这个逻辑：

```

package com.cbf4life.common2;

```

```
import java.util.ArrayList;
import java.util.List;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 */
public class Client {

    public static void main(String[] args) {
        for(Employee emp:mockEmployee()){
            emp.accept(new Visitor());
        }
    }

    //模拟出公司的人员情况，我们可以想象这个数据室通过持久层传递过来的
    public static List<Employee> mockEmployee(){
        List<Employee> empList = new ArrayList<Employee>();

        //产生张三这个员工
        CommonEmployee zhangSan = new CommonEmployee();
        zhangSan.setJob("编写Java程序，绝对的蓝领、苦工加搬运工");
        zhangSan.setName("张三");
        zhangSan.setSalary(1800);
        zhangSan.setSex(Employee.MALE);
        empList.add(zhangSan);

        //产生李四这个员工
        CommonEmployee liSi = new CommonEmployee();
        liSi.setJob("页面美工，审美素质太不流行了!");
        liSi.setName("李四");
        liSi.setSalary(1900);
        liSi.setSex(Employee.FEMALE);
        empList.add(liSi);

        //再产生一个经理
        Manager wangWu = new Manager();
        wangWu.setName("王五");
        wangWu.setPerformance("基本上是负值，但是我会拍马屁呀");
        wangWu.setSalary(18750);
        wangWu.setSex(Employee.MALE);
        empList.add(wangWu);

        return empList;
    }
}
```

```
}  
}
```

改动非常少，就黄色那么一行的改动，我们看运行结果：

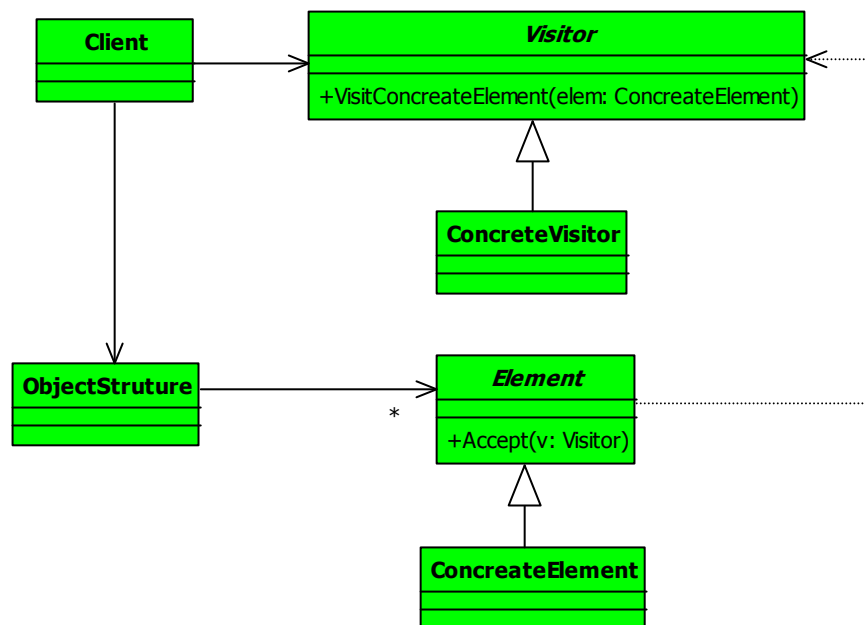
```
姓名：张三  性别：男 薪水：1800  工作：编写Java程序，绝对的蓝领、苦工加搬运工  
姓名：李四  性别：女 薪水：1900  工作：页面美工，审美素质太不流行了！  
姓名：王五  性别：男 薪水：18750 业绩：基本上是负值，但是我会拍马屁呀
```

运行结果也完全相同，那回过头我们来看看这个程序是怎么实现的：

- ◆ 首先通过循环遍历所有元素；
- ◆ 其次，每个员工对象都定义了一个访问者；
- ◆ 再其次，员工对象把自己做为一个参数调用访问者 visit 方法；
- ◆ 然后，访问者调用自己内部的计算逻辑，计算出相应的数据和表格元素；
- ◆ 最后，访问者打印出报表和数据；

事情的经过就是这个样子滴~，那我们再来看看上面提到的数据和报表格式都会改变的情况，首先数据的改变，数据改那当然都要改，这个没跑，说不上两个方案有什么优劣；其次报表格式修改，这个方案绝对是有优势的，你看我只要再产生一个 Visitor 就可以产生一个新的报表格式，而其他的类都不用修改，再如果你是用 Spring 开发的话，那就更爽了，在 Spring 的配置文件中使用的是接口注入，我只要把配置文件<property name="xxx" ref=" " />中的 ref 修改一下就行，别的什么都不用修改了！

以上讲的就是访问者模式，这个模式的通用类图如下：



看了这个通用类图，大家可能要犯迷糊了，这里怎么有一个 ObjectStruture 这个类呢？你刚刚举得例子就没有呢？真没有嘛？我们不是定义了一个 List 了吗？这就是一个 ObjectStruture，我们来看这几个角色的职责：

抽象访问者（Visitor）：抽象类或者接口，声明访问者可以访问哪些元素，具体到程序中就是 visit 方法的参数定义哪些对象是可以被访问的；

具体访问者（ConcreteVisitor）：访问者访问到一个类后该怎么干（哎，这个别读歪了），要做什么事情；

抽象元素（Element）：接口或者抽象类，声明接受那一类型的访问者访问，程序上是通过 accept 方法中的参数来定义；

具体元素：（ConcreteElement）：实现 accept 方法，通常是 visitor.visit(this)，基本上都形成了一个套路了；

结构对象（ObjectStruture）：容纳多个不同类、不同接口的容器，比如 List、Set、Map 等，在项目中，一般很少抽象出来这个角色；

大家可以这样理解访问者模式，我作为一个访客（Visitor）到朋友家（Visited Class）去拜访，朋友之间聊聊天，喝喝酒，再相互吹捧吹捧，炫耀炫耀，这都正常，聊天的时候，朋友告诉我，他今年加官进爵了，工资也涨了 30%，准备再买套房子，那我就在心里盘算（Visitor-self-method）“你个龟儿子，这么有钱，老子去年要借 10W 你都不借”，我根据被朋友的信息，执行了自己的一个方法。

接下来我们来思考一下，访问者可以用在什么地方。在这种地方你一定要考虑到使用访问者模式：业

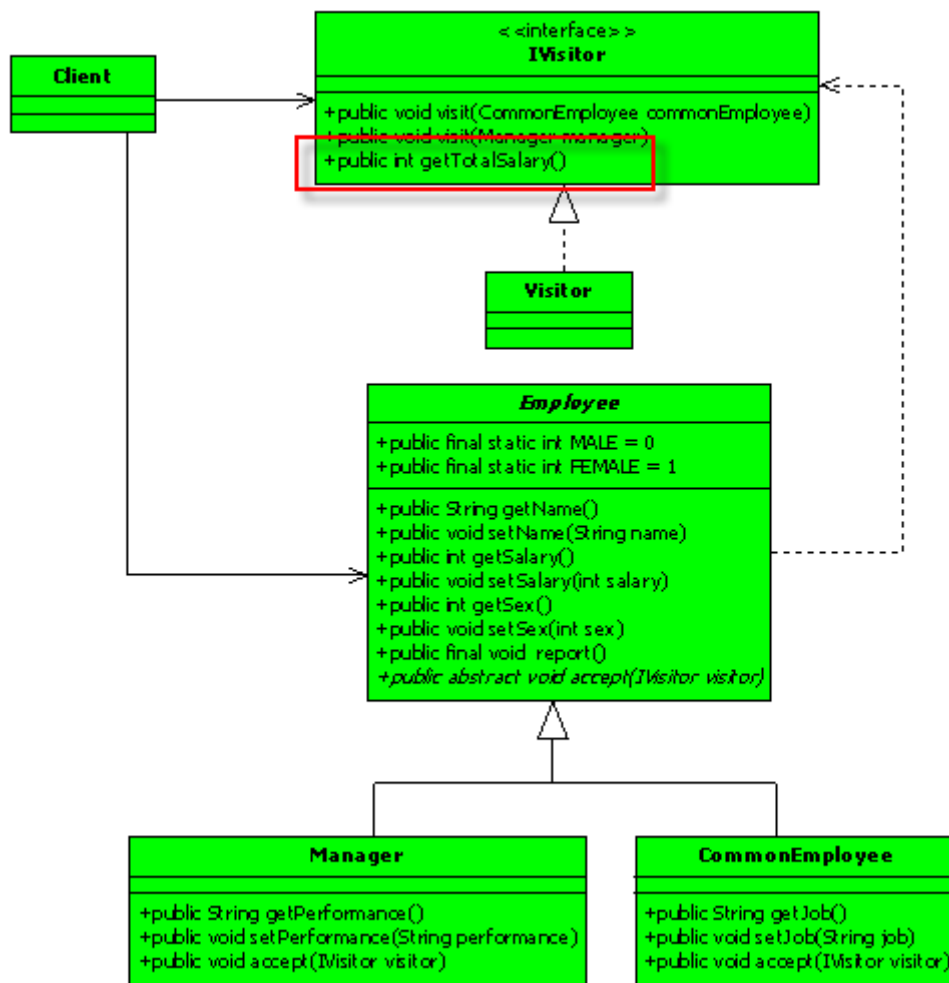
务规则要求遍历多个不同的对象。这本身也是访问者模式出发点，迭代器模式只能访问同类或同接口的数据，（当然了，你使用 `instanceof` 的话，能访问所有的数据，这个不争论），而访问者模式是对迭代器模式的扩充，可以遍历不同的对象，然后执行不同的操作，也就是针对访问的对象不同，执行不同的操作。访问者模式还有一个用途，就是充当拦截器（Interceptor）角色，这个我们在后边来讲。

访问者模式有哪些优点呢？首先是符合单一职责原则，具体元素角色也就是 `Employee` 这个类的两个子类负责数据的加载，而 `Visitor` 类则负责报表的展现，两个不同的职责非常明确的分离开来，各自演绎而变化；其次，由于职责分开，继续增加对数据的操作是非常快捷的，例如现在要增加一个给最大老板的一份报表，这份报表格式又有所不同，容易处理吧，直接在 `Visitor` 中增加一个方法，传递过来数据后进行整理打印；最后，数据汇总，就以刚刚我们说的 `Employee` 的例子，如果我现在要统计所有员工的工资之和，怎么计算？把所有人的工资 `for` 循环加一遍？是个办法，那我再提个问题，员工工资*1.2，部门经理*1.4，总经理*1.8，然后把这些工资加起来，你如何处理？1.2，1.4，1.8 是什么？我 K，你没看到领导不论什么时候都比你拿的多，工资奖金就不说了，就是过节发个慰问券也比你多，就是这个系数在作祟。我们继续说你先怎么统计？使用 `for` 循环，然后使用 `instanceof` 来判断是员工还是经理？可以解决，但不是个好办法，好办法是通过访问者模式来实现，把数据扔给访问者，由访问者来进行统计计算。

访问者模式的缺点也很明显，访问者要访问一个类就必然要求这个类公布一些方法，也就是说访问者关注了其他类的内部细节，这是迪米特法则所不建议的；还有一个缺点就是，具体角色的增加删除修改都是比较苦难的，就上面那个例子，你想想，你要是想增加一个成员变量，比如年龄 `age`，`Visitor` 就需要修改，如果 `Visitor` 是一个还好说，多个呢？业务逻辑再复杂点呢？访问者模式是有缺点的，是事物都有缺点，但是这仍然掩盖不了它的光芒，访问者模式结合其他模式比如模版方法模式、状态模式、解释器模式、代理模式等就会非常强大，这个我们放在模式混编中来讲解。

访问者模式是会经常用到的模式，虽然你不注意，有可能你起的名字也不是什么 `Visitor`，但是这个是非常容易使用到的，在这里我提出三个扩展的功能共大家参考。

统计功能。在访问者模式中的使用中我也提到访问者的统计功能，汇总和报表是金融类企业非常常用的功能，基本上都是一堆的计算公式，然后出一个报表，很多项目是采用了数据库的存储过程来实现，这个我不是很推荐，除非海量数据处理，一个晚上要上亿、几十亿条的数据跑批处理，这个除了存储过程来处理没有其他办法的，你要是用应用服务器来处理，连接数据库的网络就是处于 100% 占用状态，一个晚上也未必跑得完这批数据！除了这种海量数据外，我建议数据统计和报表的批处理通过访问者模式来处理会比较简单。好，那我们来统计一下公司人员的工资，先看类图：



就在接口上增加了一个 `getTotalSalary` 方法，在 `Visitor` 实现类中实现该方法，我们先看接口：

```

package com.cbf4life.extend;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 访问者，我要去访问人家的数据了
 */
public interface IVisitor {

    // 首先定义我可以访问普通员工
    public void visit(CommonEmployee commonEmployee);

    // 其次定义，我还可以访问部门经理
    public void visit(Manager manager);
}

```

```
//统计所有员工工资总和
public int getTotalSalary();
}
```

就多了一个 `getTotalSalary` 方法，我们再来看实现类：

```
package com.cbf4life.extend;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 */
public class Visitor implements IVisitor {
    //部门经理的工资系数是5
    private final static int MANAGER_COEFFICIENT = 5;

    //员工的工资系数是2
    private final static int COMMONEMPLOYEE_COEFFICIENT = 2;

    //普通员工的工资总和
    private int commonTotalSalary = 0;

    //部门经理的工资总和
    private int managerTotalSalary = 0;

    //访问普通员工，打印出报表
    public void visit(CommonEmployee commonEmployee) {
        System.out.println(this.getCommonEmployee(commonEmployee));
        //计算普通员工的薪水总和
        this.calCommonSalary(commonEmployee.getSalary());
    }

    //访问部门经理，打印出报表
    public void visit(Manager manager) {
        System.out.println(this.getManagerInfo(manager));
        //计算部门经理的工资总和
        this.calManagerSalary(manager.getSalary());
    }

    //组装出基本信息
    private String getBasicInfo(Employee employee){
        String info = "姓名: " + employee.getName() + "\t";
        info = info + "性别: " + (employee.getSex() == Employee.FEMALE?"女":"男")
}
```

```

+ "\t";
    info = info + "薪水: " + employee.getSalary() + "\t";

    return info;
}

// 组装出部门经理的信息
private String getManagerInfo(Manager manager){
    String basicInfo = this.getBasicInfo(manager);
    String otherInfo = "业绩: "+manager.getPerformance() + "\t";
    return basicInfo + otherInfo;
}

// 组装出普通员工信息
private String getCommonEmployee(CommonEmployee commonEmployee){
    String basicInfo = this.getBasicInfo(commonEmployee);
    String otherInfo = "工作: "+commonEmployee.getJob()+"\t";
    return basicInfo + otherInfo;
}

// 计算部门经理的工资总和
private void calManagerSalary(int salary){
    this.managerTotalSalary = this.managerTotalSalary + salary
*MANAGER_COEFFICIENT ;
}

// 计算普通员工的工资总和
private void calCommonSalary(int salary){
    this.commonTotalSalary = this.commonTotalSalary +
salary*COMMONEMPLOYEE_COEFFICIENT;
}

// 获得所有员工的工资总和
public int getTotalSalary(){
    return this.commonTotalSalary + this.managerTotalSalary;
}
}

```

程序比较长，但是还是比较简单的，分别计算普通员工和经理级员工的工资总和，然后加起来。注意我们在实现时已经考虑员工工资和经理工资的系数不同。

我们再来看 Client 类的调用：

```
package com.cbf4life.extend;

import java.util.ArrayList;
import java.util.List;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 */
public class Client {

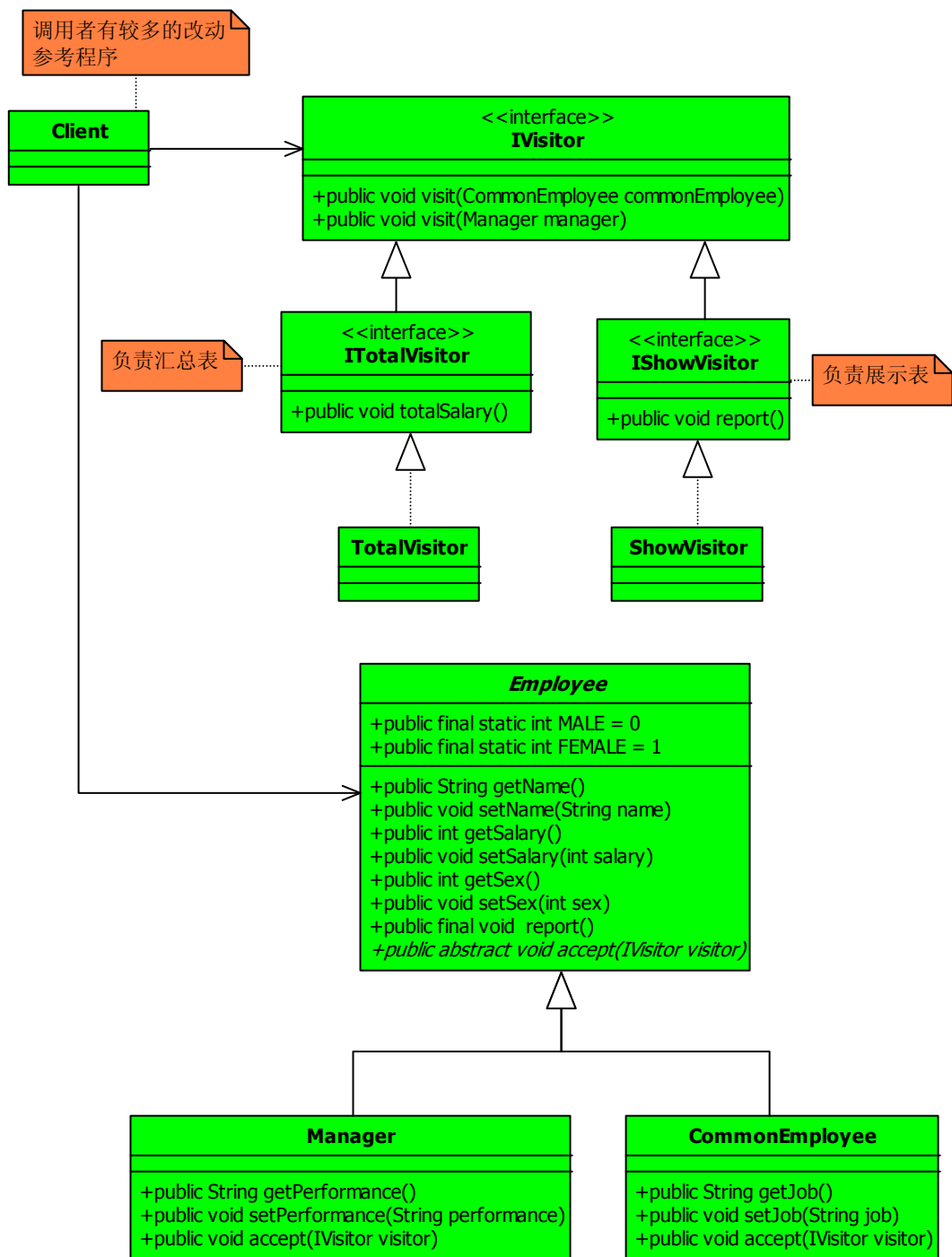
    public static void main(String[] args) {
        IVisitor visitor = new Visitor();
        for(Employee emp:mockEmployee()){
            emp.accept(visitor);
        }
        System.out.println(" 本公司的月工资总额是: "+visitor.getTotalSalary());
    }
}
```

其中 `mockEmployee` 静态方法没有任何改动，就没有拷贝上去。我们来看运行的结果：

```
姓名：张三   性别：男 薪水：1800  工作：编写Java程序，绝对的蓝领、苦工加搬运工
姓名：李四   性别：女 薪水：1900  工作：页面美工，审美素质太不流行了！
姓名：王五   性别：男 薪水：18750 业绩：基本上是负值，但是我会拍马屁呀
本公司的月工资总额是：101150
```

然后你想修改工资的系数，没有问题！想换个展示格式，也没有问题！自己练习一下吧

多个访问者。在实际的项目中，一个对象，多个访问者的情况非常多。其实我们上面例子就应该是两个访问者，为什么呢？报表分两种，一种是展示表，通过数据库查询，把结果展示出来，这个就类似于我们的那个列表；第二种是汇总表，这个是需要通过模型或者公式计算出来的，一般都是批处理结果，这个类似于我们计算工资总额，这两种报表格式是对同一堆数据的两种处理方式，从程序上看，一个类就有个不同的访问者了，那我们修改一下类图：



类图看着挺恐怖，其实也没啥复杂的，多了两个接口和两个实现类，分别负责展示表和汇总表的业务处理，我们先看 IVisitor 接口程序：

```

package com.cbf4life.extend2;

/**
 * @author cbf4Life cbf4life@126.com

```

```

* I'm glad to share my knowledge with you all.
* 访问者，我要去访问人家的数据了
*/
public interface IVisitor {

    //首先定义我可以访问普通员工
    public void visit(CommonEmployee commonEmployee);

    //其次定义，我还可以访问部门经理
    public void visit(Manager manager);

}

```

该接口定义其下的实现类能够访问哪些类，子接口定义了具体访问者的任务和责任，先看 *ITotalVisitor* 接口：

```

package com.cbf4life.extend2;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 负责统计报表的产生
 */
public interface ITotalVisitor extends IVisitor {
    //统计所有员工工资总和
    public void totalSalary();
}

```

就一句话，非常简单，我们再来看展示表：

```

package com.cbf4life.extend2;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 负责展示报表的产生
 */
public interface IShowVisitor extends IVisitor {
    //展示报表
    public void report();
}

```

也是就一句话，我们展示表访问者的实现类：

```
package com.cbf4life.extend2;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 展示报表，该访问者的工作就是看到什么数据展示什么数据
 */
public class ShowVisitor implements IShowVisitor {
    private String info = "";

    //打印出报表
    public void report() {
        System.out.println(this.info);
    }

    //访问普通员工，组装信息
    public void visit(CommonEmployee commonEmployee) {
        this.info = this.info + this.getBasicInfo(commonEmployee)+ "工作:"
        "+commonEmployee.getJob()+"\t\n";
    }

    //访问经理，然后组装信息
    public void visit(Manager manager) {
        this.info = this.info + this.getBasicInfo(manager) + "业绩:"
        "+manager.getPerformance() + "\t\n";
    }

    //组装出基本信息
    private String getBasicInfo(Employee employee){
        String info = "姓名: " + employee.getName() + "\t";
        info = info + "性别: " + (employee.getSex() == Employee.FEMALE?"女":"男")
        + "\t";
        info = info + "薪水: " + employee.getSalary() + "\t";

        return info;
    }
}
```

下面是汇总表访问者：


```

package com.cbf4life.extend2;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 汇总表, 该访问者起汇总作用, 把容器中的数据一个一个遍历, 然后汇总
 */
public class TotalVisitor implements ITotalVisitor {
    // 部门经理的工资系数是5
    private final static int MANAGER_COEFFICIENT = 5;

    // 员工的工资系数是2
    private final static int COMMONEMPLOYEE_COEFFICIENT = 2;

    // 普通员工的工资总和
    private int commonTotalSalary = 0;

    // 部门经理的工资总和
    private int managerTotalSalary = 0;

    public void totalSalary() {
        System.out.println(" 本公司的月工资总额是" + (this.commonTotalSalary +
this.managerTotalSalary));
    }

    // 访问普通员工, 计算工资总额
    public void visit(CommonEmployee commonEmployee) {
        this.commonTotalSalary = this.commonTotalSalary +
commonEmployee.getSalary()*COMMONEMPLOYEE_COEFFICIENT;
    }

    // 访问部门经理, 计算工资总额
    public void visit(Manager manager) {
        this.managerTotalSalary = this.managerTotalSalary + manager.getSalary()*MANAGER_COEFFICIENT ;
    }
}

```

然后看 *Client* 类的修改:

```
package com.cbf4life.extend2;

import java.util.ArrayList;
import java.util.List;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 */
public class Client {

    public static void main(String[] args) {
        //展示报表访问者
        IShowVisitor showVisitor = new ShowVisitor();
        //汇总报表的访问者
        ITotalVisitor totalVisitor = new TotalVisitor();

        for(Employee emp:mockEmployee()){
            emp.accept(showVisitor); //接受展示报表访问者
            emp.accept(totalVisitor); //接受汇总表访问者
        }

        //展示报表
        showVisitor.report();
        //汇总报表
        totalVisitor.totalSalary();
    }
}
```

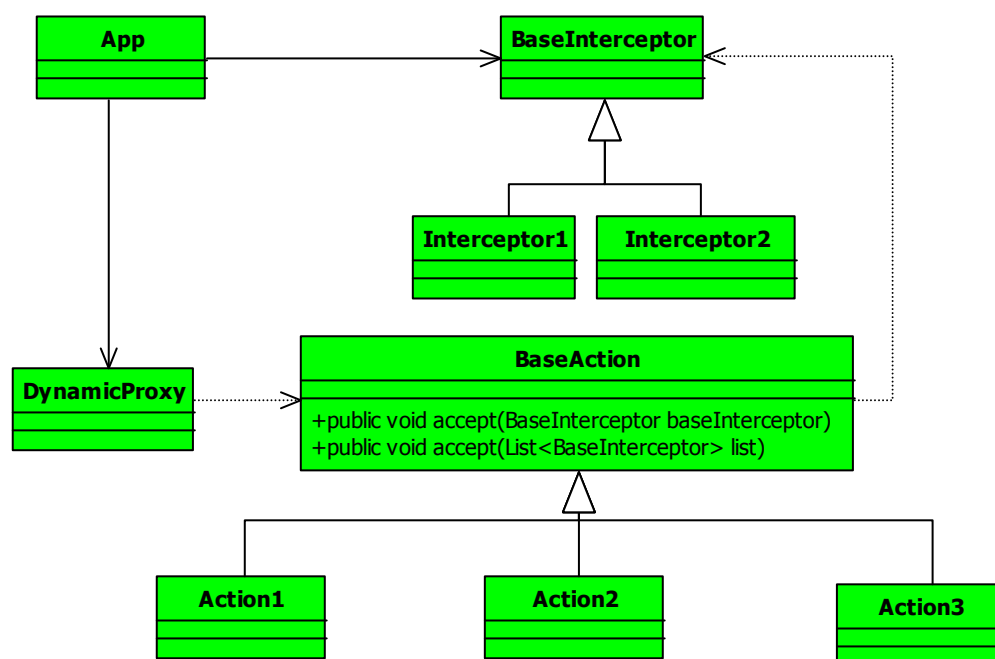
运行结果如下:

```
姓名: 张三  性别: 男 薪水: 1800 工作: 编写Java程序, 绝对的蓝领、苦工加搬运工
姓名: 李四  性别: 女 薪水: 1900 工作: 页面美工, 审美素质太不流行了!
姓名: 王五  性别: 男 薪水: 18750 业绩: 基本上是负值, 但是我会拍马屁呀

本公司的月工资总额是101150
```

大家可以再深入的想象，一堆数据从几个角度来分析，那是什么？数据挖掘（Data Mining），数据的上切、下钻等等处理，大家有兴趣看可以翻看数据挖掘或者商业智能（BI）的书。

拦截器。你如果用过 Struts2，对拦截器绝对不会陌生，我们先想一下拦截器有什么作用，拦截器的核心作用是“围墙”作用，拦截器对被拦截的对象进行检查，符合规则的对象则开门放进去，继续执行下一个逻辑，不符合规则的则弹回（其实这也是过滤器的作用）；拦截器还有一个作用是修改数据，对于符合规则数据可以进行修改，以便继续后序的逻辑。具备了这两个功能，拦截器的雏形就有了，访问者模式就可以实现简单的拦截器角色，我们来看类图：

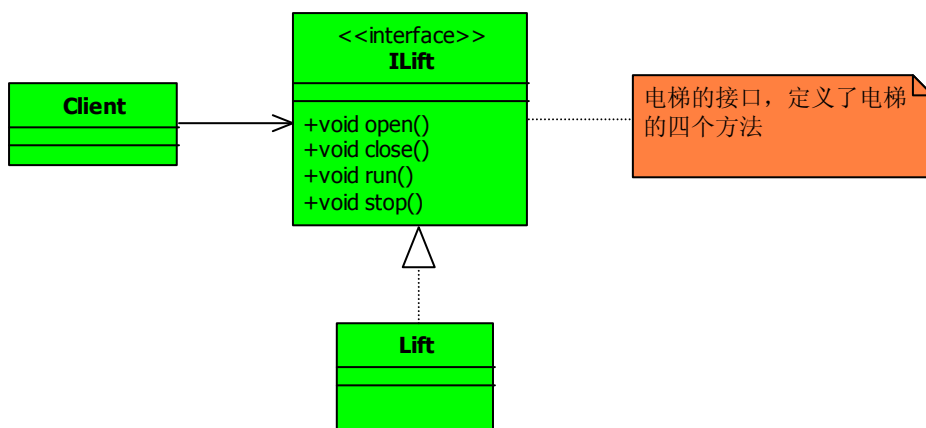


看着是不是和访问者模式的通用类图很类似？两个 `accept` 方法，其中参数为 `List` 类型的则实现了拦截器栈的作用，`DynamicProxy` 类使用了动态代理和反射模式。拦截器实现起来也不复杂，今天就不实现了，这个作为作业，请大家自己来实现。计划在混编模式中一起探讨。

第 19 章 状态模式【State Pattern】

现在城市发展很快，百万级人口的城市一堆一堆的，那其中有两个东西的发明在城市的发展中起到非常重要的作用：一个是汽车，一个呢是...，猜猜看，是什么？是电梯！汽车让城市可以横向扩展，电梯让城市可以纵向延伸，向空中伸展。汽车对城市的发展我们就不说了，电梯，你想想看，如果没有电梯，每天你需要爬 10 层楼梯，你是不是会崩溃掉？建筑师设计了一个没有电梯的建筑，那投资家肯定不愿意投资，那也是建筑师的耻辱呀，今天我们就用程序表现一下这个电梯是怎么运作的。

我们每天都在乘电梯，那我们来看看电梯有哪些动作（映射到 Java 中就是有多少方法）：开门、关门、运行、停止，就这四个动作，好，我们就用程序来实现一下电梯的动作，先看类图设计：



非常简单的类图，定义一个接口，然后是一个实现类，然后业务类 **Client** 就可以调用，并运行起来，简单也来看看我们的程序，先看接口：

```
package com.cbf4life.common;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 定义一个电梯的接口
 */
public interface ILift {

    //首先电梯门开启动作
    public void open();

    //电梯门有开启，那当然也就有关闭了
```

```
public void close();

//电梯要能上能下，跑起来
public void run();

//电梯还要能停下来，停不下来那就扯淡了
public void stop();
}
```

然后看实现类：

```
package com.cbf4life.common;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 电梯的实现类
 */
public class Lift implements ILift {

    //电梯门关闭
    public void close() {
        System.out.println("电梯门关闭...");
    }

    //电梯门开启
    public void open() {
        System.out.println("电梯门开启...");
    }

    //电梯开始跑起来
    public void run() {
        System.out.println("电梯上下跑起来...");
    }

    //电梯停止
    public void stop() {
        System.out.println("电梯停止了...");
    }
}
```

电梯的开、关、跑、停都实现了，开看业务是怎么调用的：

```
package com.cbf4life.common;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 模拟电梯的动作
 */
public class Client {

    public static void main(String[] args) {
        ILift lift = new Lift();
        //首先是电梯门开启，人进去
        lift.open();

        //然后电梯门关闭
        lift.close();

        //再然后，电梯跑起来，向上或者向下
        lift.run();

        //最后到达目的地，电梯挺下来
        lift.stop();
    }
}
```

运行的结果如下：

```
电梯门开启...
电梯门关闭...
电梯上下跑起来...
电梯停止了...
```

太简单的程序了，是个程序员都会写这个程序，这么简单的程序还拿出来 show，是不是太小看我们的智商了？！非也，非也，我们继续往下分析，这个程序有什么问题，你想呀电梯门可以打开，但不是随时都可以开，是有前提条件的，你不可能电梯在运行的时候突然开门吧？！电梯也不会出现停止了但是不开门的情况吧？！那要是有也是事故嘛，再仔细想想，电梯的这四个动作的执行都是有前置条件，具体点说说在特定状态下才能做特定事，那我们来分析一下电梯有什么那些特定状态：

门敞状态---按了电梯上下按钮，电梯门开，这中间有 5 秒的时间（当然你也可以用身体挡住电梯门，那就不是 5 秒了），那就是门敞状态；在这个状态下电梯只能做的动作是关门动作，做别的动作？那就危险

喽

门闭状态---电梯门关闭了，在这个状态下，可以进行的动作是：开门（我不想坐电梯了）、停止（忘记按路层号了）、运行

运行状态---电梯正在跑，上下窜，在这个状态下，电梯只能做的是停止；

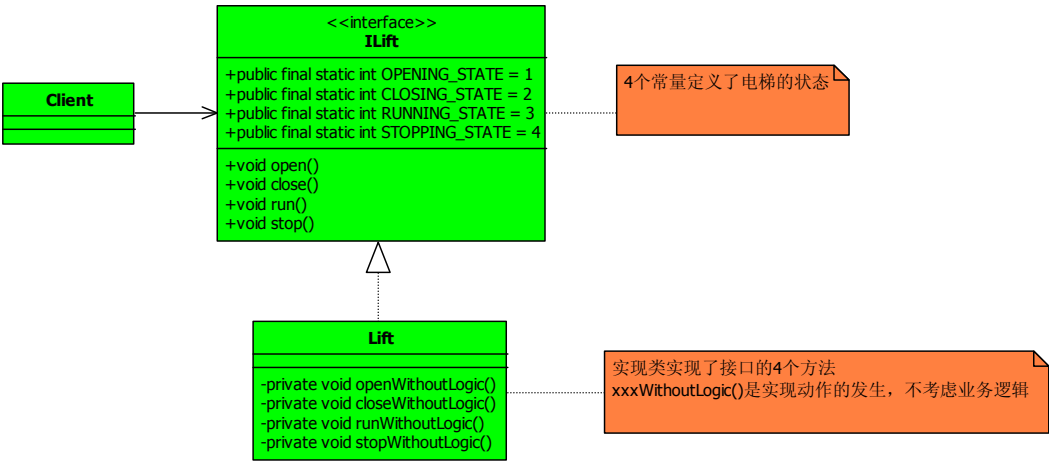
停止状态---电梯停止不动，在这个状态下，电梯有两个可选动作：继续运行和开门动作；

我们用一张表来表示电梯状态和动作之间的关系：

	开门（open）	关门（close）	运行(run)	停止(stop)
门敞状态	○	☆	○	○
门闭状态	☆	○	☆	☆
运行状态	○	○	○	☆
停止状态	☆	○	☆	○

电梯状态和动作对应表（○表示不允许，☆表示允许动作）

看到这张表后，我们才发觉，哦~~，我们的程序做的很不严谨，好，我们来修改一下，先看类图：



增加了状态的类图

在接口中定义了四个常量，分别表示电梯的四个状态：门敞状态、关闭状态、运行状态、停止状态，然后在实现类中电梯的每一次动作发生都要对状态进行判断，判断是否运行执行，也就是动作的执行是否符合业务逻辑，实现类中的四个私有方法是仅仅实现电梯的动作，没有任何的前置条件，因此这四个方法是不能为外部类调用的，设置为私有方法。我们先看接口的改变：

```
package com.cbf4life.common2;
```

```
/**
```

```
* @author cbf4Life cbf4life@126.com
* I'm glad to share my knowledge with you all.
* 定义一个电梯的接口
*/
public interface ILift {
    //电梯的四个状态
    public final static int OPENING_STATE = 1; //门敞状态
    public final static int CLOSING_STATE = 2; //门闭状态
    public final static int RUNNING_STATE = 3; //运行状态
    public final static int STOPPING_STATE = 4; //停止状态;

    //设置电梯的状态
    public void setState(int state);

    //首先电梯门开启动作
    public void open();

    //电梯门有开启，那当然也就有关闭了
    public void close();

    //电梯要能上能下，跑起来
    public void run();

    //电梯还要能停下来，停不下来那就扯淡了
    public void stop();
}
```

增加了四个静态常量，增加了一个方法 setState，设置电梯的状态。我们再来看实现类是如何实现的：

```
package com.cbf4life.common2;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 电梯的实现类
 */
public class Lift implements ILift {
    private int state;

    public void setState(int state) {
        this.state = state;
    }
}
```



```
//电梯门关闭
public void close() {
    //电梯在什么状态下才能关闭
    switch(this.state){
        case OPENING_STATE: //如果是则可以关门，同时修改电梯状态
            this.closeWithoutLogic();
            this.setState(CLOSING_STATE);
            break;
        case CLOSING_STATE: //如果电梯就是关门状态，则什么都不做
            //do nothing;
            break;
        case RUNNING_STATE: //如果是正在运行，门本来就是关闭的，也说明都不做
            //do nothing;
            break;
        case STOPPING_STATE: //如果是停止状态，本也是关闭的，什么也不做
            //do nothing;
            break;
    }
}

//电梯门开启
public void open() {
    //电梯在什么状态才能开启
    switch(this.state){
        case OPENING_STATE: //如果已经在门敞状态，则什么都不做
            //do nothing;
            break;
        case CLOSING_STATE: //如是电梯时关闭状态，则可以开启
            this.openWithoutLogic();
            this.setState(OPENING_STATE);
            break;
        case RUNNING_STATE: //正在运行状态，则不能开门，什么都不做
            //do nothing;
            break;
        case STOPPING_STATE: //停止状态，淡然要开门了
            this.openWithoutLogic();
            this.setState(OPENING_STATE);
            break;
    }
}

//电梯开始跑起来
```

```
public void run() {
    switch(this.state){
        case OPENING_STATE: //如果已经在门敞状态，则不你能运行，什么都不做
            //do nothing;
            break;
        case CLOSING_STATE: //如是电梯时关闭状态，则可以运行
            this.runWithoutLogic();
            this.setState(RUNNING_STATE);
            break;
        case RUNNING_STATE: //正在运行状态，则什么都不做
            //do nothing;
            break;
        case STOPPING_STATE: //停止状态，可以运行
            this.runWithoutLogic();
            this.setState(RUNNING_STATE);
    }
}

//电梯停止
public void stop() {
    switch(this.state){
        case OPENING_STATE: //如果已经在门敞状态，那肯定要先停下来的，什么都不做
            //do nothing;
            break;
        case CLOSING_STATE: //如是电梯时关闭状态，则当然可以停止了
            this.stopWithoutLogic();
            this.setState(CLOSING_STATE);
            break;
        case RUNNING_STATE: //正在运行状态，有运行当然那也就有停止了
            this.stopWithoutLogic();
            this.setState(CLOSING_STATE);
            break;
        case STOPPING_STATE: //停止状态，什么都不做
            //do nothing;
            break;
    }
}

//纯粹的电梯关门，不考虑实际的逻辑
private void closeWithoutLogic(){
    System.out.println("电梯门关闭...");
}

//纯粹的店门开，不考虑任何条件
```

```

private void openWithoutLogic(){
    System.out.println("电梯门开启...");
}

//纯粹的运行，不考虑其他条件
private void runWithoutLogic(){
    System.out.println("电梯上下跑起来...");
}

//单纯的停止，不考虑其他条件
private void stopWithoutLogic(){
    System.out.println("电梯停止了...");
}
}

```

程序有点长，但是还是很简单的，就是在每一个接口定义的方法中使用 `witch...case` 来进行判断，是否运行运行指定的动作。我们来 Client 程序的变更：

```

package com.cbf4life.common2;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 模拟电梯的动作
 */
public class Client {

    public static void main(String[] args) {
        ILift lift = new Lift();

        //电梯的初始条件应该是停止状态
        lift.setState(ILift.STOPPING_STATE);

        //首先是电梯门开启，人进去
        lift.open();

        //然后电梯门关闭
        lift.close();

        //再然后，电梯跑起来，向上或者向下
        lift.run();
    }
}

```

```
//最后到达目的地，电梯挺下来  
lift.stop();  
}  
}
```

业务调用的方法中增加了电梯状态判断，电梯要开门不是随时都可以开的，必须满足了一定条件你才能开门，人才能走进去，我们设置电梯的起始是停止状态，看运行结果：

```
电梯门开启...  
电梯门关闭...  
电梯上下跑起来...  
电梯停止了...
```

我们来想一下，这段程序有什么问题，首先 Lift.java 这个文件有点长，长的原因是我们在程序中使用了大量的 switch...case 这样的判断（if...else 也是一样），程序中只要你有这样的判断就避免不了加长程序，同步的在业务比较复杂的情况下，程序体会更长，这个就不是一个很好的习惯了，较长的方法或者类的维护性比较差，毕竟程序是给人来阅读的；其次，扩展性非常的不好，大家来想想，电梯还有两个状态没有加，是什么？通电状态和断电状态，你要是在程序再增加这两个方法，你看看 Open()、Close()、Run()、Stop() 这四个方法都要增加判断条件，也就是说 switch 判断体中还要增加 case 项，也就说与开闭原则相违背了；再其次，我们来思考我们的业务，电梯在门敞开状态下就不能上下跑了吗？电梯有没有发生过只有运行没有停止状态呢（从 40 层直接坠到 1 层嘛）？电梯故障嘛，还有电梯在检修的时候，可以在 stop 状态下不开门，这也是正常的业务需求呀，你想想看，如果加上这些判断条件，上面的程序有多少需要修改？虽然这些都是电梯的业务逻辑，但是一个类有且仅有一个原因引起类的变化，单一职责原则，看看我们的类，业务上的任务一个小小增加或改动都对我们的这个电梯类产生了修改，这是在项目开发上是有很大风险的。既然我们已经发现程序上有以上问题，我们怎么来修改呢？

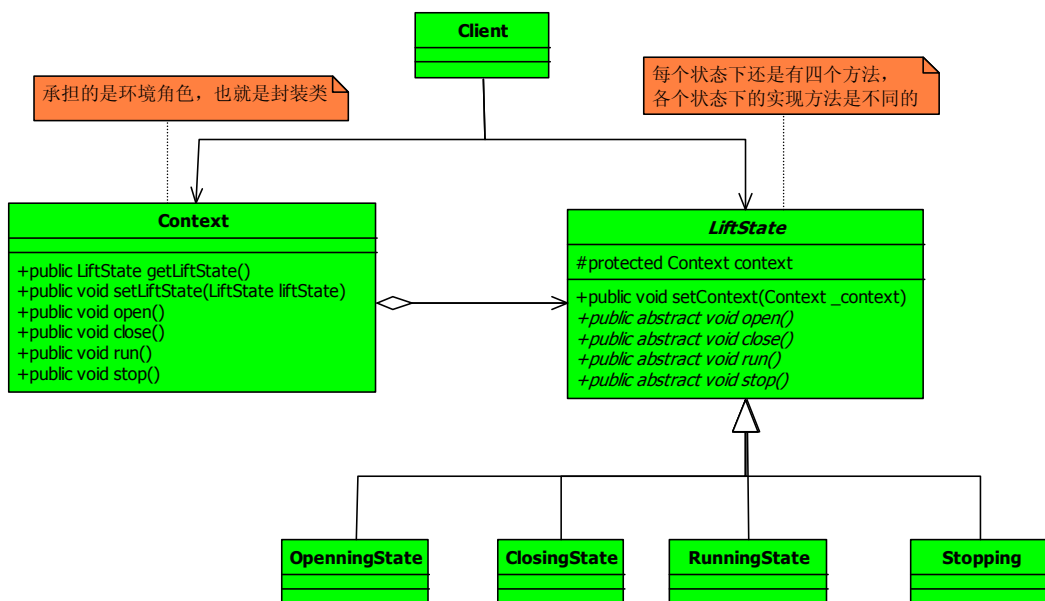
刚刚我们是从电梯的有哪些方法以及这些方法执行的条件去分析，现在我们换个角度来看问题，我们来想电梯在具有这些状态的时候，能够做什么事情，也就是说在电梯处于一个具体状态时，我们来思考这个状态是由什么动作触发而产生以及在这个状态下电梯还能做什么事情，举个例子来说，电梯在停止状态时，我们来思考两个问题：

第一、这个停止状态时怎么来的，那当然是由于电梯执行了 stop 方法而来的；

第二、在停止状态下，电梯还能做什么动作？继续运行？开门？那当然都可以了。

我们再来分析其他三个状态，也都是一样的结果，我们只要实现电梯在一个状态下的两个任务模型就

可以了：这个状态是如何产生的以及在这个状态下还能做什么其他动作（也就是这个状态怎么过渡到其他状态），既然我们以状态为参考模型，那我们就先定义电梯的状态接口，思考过后我们来看类图：



以状态作为导向的类图

在类图中，定义了一个 **LiftState** 抽象类，声明了一个受保护的类型 **Context** 变量，这个是串联我们各个状态的封装类，封装的目的很明显，就是电梯对象内部状态的变化不被调用类知晓，也就是迪米特法则了，我的类内部情节你知道越少越好，并且还定义了四个具体的实现类，承担的是状态的产生以及状态间的转换过渡，我们先来看 **LiftState** 程序：

```

package com.cbf4life.advance;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 定义一个电梯的接口
 */
public abstract class LiftState{

    //定义一个环境角色，也就是封装状态的变换引起的功能变化
    protected Context context;

    public void setContext(Context _context){
        this.context = _context;
    }
}
  
```

```

//首先电梯门开启动作
public abstract void open();

//电梯门有开启，那当然也就有关闭了
public abstract void close();

//电梯要能上能下，跑起来
public abstract void run();

//电梯还要能停下来，停不下来那就扯淡了
public abstract void stop();
}

```

抽象类比较简单，我们来先看一个具体的实现，门敞状态的实现类：

```

package com.cbf4life.advance;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 在电梯门开启的状态下能做什么事情
 */
public class OpenningState extends LiftState {

    //开启当然可以关闭了，我就想测试一下电梯门开关功能
    @Override
    public void close() {
        //状态修改
        super.context.setLiftState(Context.closeingState);
        //动作委托为CloseState来执行
        super.context.getLiftState().close();
    }

    //打开电梯门
    @Override
    public void open() {
        System.out.println("电梯门开启...");
    }

    //门开着电梯就想跑，这电梯，吓死你！
    @Override
    public void run() {

```

```

        //do nothing;
    }

    //开门还不停止?
    public void stop() {
        //do nothing;
    }
}

```

我来解释一下这个类的几个方法，Opening 状态是由 open() 方法产生的，因此这个方法中有一个具体的业务逻辑，我们是用 print 来代替了；在 Opening 状态下，电梯能过渡到其他什么状态呢？按照现在的定义的是只能过渡到 Closing 状态，因此我们在 Close() 中定义了状态变更，同时把 Close 这个动作也委托了给 CloseState 类下的 Close 方法执行，这个可能不好理解，我们再看看 Context 类就可能好理解一点：

```

package com.cbf4life.advance;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 */
public class Context {
    //定义出所有的电梯状态
    public final static OpeningState openingState = new OpeningState();
    public final static ClosingState closeingState = new ClosingState();
    public final static RunningState runningState = new RunningState();
    public final static StoppingState stoppingState = new StoppingState();

    //定一个当前电梯状态
    private LiftState liftState;

    public LiftState getLiftState() {
        return liftState;
    }

    public void setLiftState(LiftState liftState) {
        this.liftState = liftState;
        //把当前的环境通知到各个实现类中
        this.liftState.setContext(this);
    }
}

```

```

    public void open(){
        this.liftState.open();
    }

    public void close(){
        this.liftState.close();
    }

    public void run(){
        this.liftState.run();
    }

    public void stop(){
        this.liftState.stop();
    }
}

```

结合以上三个类，我们可以这样理解，Context 是一个环境角色，它的作用是串联各个状态的过渡，在 LiftState 抽象类中我们定义了并把这个环境角色聚合进来，并传递到了子类，也就是四个具体的实现类中自己根据环境来决定如何进行状态的过渡。我们把其他的三个具体实现类阅读完毕，下面是关闭状态：

```

package com.cbf4life.advance;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 电梯门关闭以后，电梯可以做哪些事情
 */
public class ClosingState extends LiftState {

    //电梯门关闭，这是关闭状态要实现的动作
    @Override
    public void close() {
        System.out.println("电梯门关闭...");
    }

    //电梯门关了再打开，逗你玩呢，那这个允许呀
    @Override
    public void open() {
        super.context.setLiftState(Context.openningState); //置为门敞状态
        super.context.getLiftState().open();
    }
}

```



```

    }

    //电梯门关了就跑，这是再正常不过了
    @Override
    public void run() {
        super.context.setLiftState(Context.runningState); //设置为运行状态;
        super.context.getLiftState().run();
    }

    //电梯门关着，我就不按楼层
    @Override
    public void stop() {
        super.context.setLiftState(Context.stoppingState); //设置为停止状态;
        super.context.getLiftState().stop();
    }
}

```

下面是电梯的运行状态：

```

package com.cbf4life.advance;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 电梯在运行状态下能做哪些动作
 */
public class RunningState extends LiftState {

    //电梯门关闭？这是肯定了
    @Override
    public void close() {
        //do nothing
    }

    //运行的时候开电梯门？你疯了！电梯不会给你开的
    @Override
    public void open() {
        //do nothing
    }

    //这是在运行状态下要实现的方法
    @Override

```

```

    public void run() {
        System.out.println("电梯上下跑...");
    }

    //这个事绝对是合理的，光运行不停止还有谁敢做这个电梯？！估计只有上帝了
    @Override
    public void stop() {
        super.context.setLiftState(Context.stoppingState); //环境设置为停止状态；
        super.context.getLiftState().stop();
    }
}

```

下面是停止状态：

```

package com.cbf4life.advance;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 在停止状态下能做什么事情
 */
public class StoppingState extends LiftState {

    //停止状态关门？电梯门本来就是关着的！
    @Override
    public void close() {
        //do nothing;
    }

    //停止状态，开门，那是要的！
    @Override
    public void open() {
        super.context.setLiftState(Context.openningState);
        super.context.getLiftState().open();
    }

    //停止状态再跑起来，正常的很
    @Override
    public void run() {
        super.context.setLiftState(Context.runningState);
        super.context.getLiftState().run();
    }
}

```

```
//停止状态是怎么发生的呢？当然是停止方法执行了
@Override
public void stop() {
    System.out.println("电梯停止了...");
}

}
```

业务逻辑都已经实现了，我们来看看 Client 怎么实现：

```
package com.cbf4life.advance;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 模拟电梯的动作
 */
public class Client {

    public static void main(String[] args) {
        Context context = new Context();
        context.setLiftState(new ClosingState());

        context.open();
        context.close();
        context.run();
        context.stop();
    }
}
```

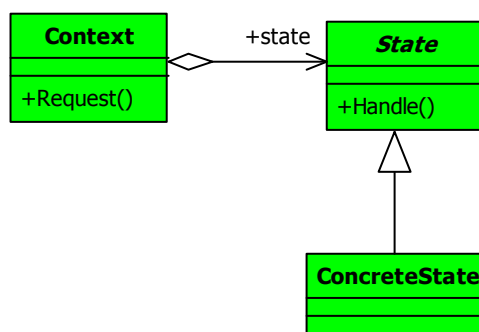
Client 调用类太简单了，只要定义个电梯的初始状态，然后调用相关的方法，就完成了，完全不用考虑状态的变更，看运行结果：

```
电梯门开启...
电梯门关闭...
电梯上下跑起来...
电梯停止了...
```

我们再来回顾一下我们刚刚批判上一段的代码，首先我们说人家代码太长，这个问题我们解决了，通

过各个子类来实现，每个子类的代码都很短，而且也取消了的 switch...case 条件的判断；其次，说人家不符合开闭原则，那如果在我们这个例子中要增加两个状态怎么加？增加两个子类，一个是通电状态，一个是断电状态，同时修改其他实现类的相应方法，因为状态要过渡呀，那当然要修改原有的类，只是在原有类中的方法上增加，而不去做修改；再其次，我们说人家不符合迪米特法则，我们现在呢是各个状态是单独的一个类，只有与这个状态的有关的因素修改了这个类才修改，符合迪米特法则，非常完美！

上面例子中多次提到状态，那我们这节讲的就是状态模式，什么是状态模式呢？**当一个对象内在状态改变时允许其改变行为，这个对象看起来像是改变了其类。**说实话，这个定义的后半句我也没看懂，看过 GOF 才明白是怎么回事：Allow an object to alter its behavior when its internal state changes. The object will appear to change its class. [GoF, p305]，也就是说状态模式封装的非常好，状态的变更引起了行为的变更，从外部看起来就好像这个对象对应的类发生了改变一样。状态模式的通用实现类如下：



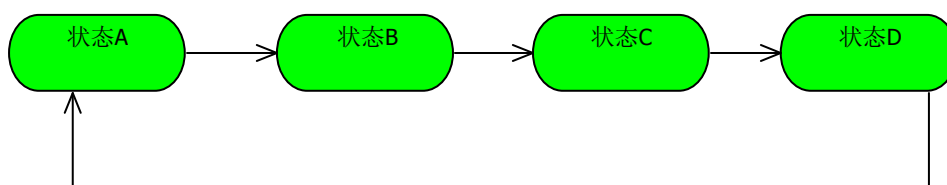
状态模式通用类图

状态模式中有什么优点呢？首先是避免了过多的 switch...case 或者 if...else 语句的使用，避免了程序的复杂性；其次是很好的使用体现了开闭原则和单一职责原则，每个状态都是一个子类，你要增加状态就增加子类，你要修改状态，你只修改一个子类就可以了；最后一个好处就是封装性非常好，这也是状态模式的基本要求，状态变换放置到了类的内部来实现，外部的调用不用知道类内部如何实现状态和行为的变换。

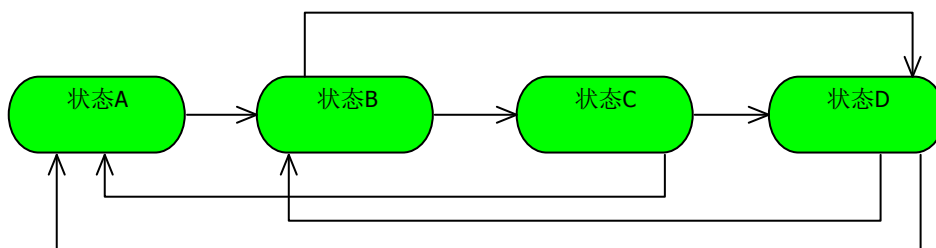
状态模式既然有优点，那当然有缺点了，只有一个缺点，子类会太多，也就是类膨胀，你想一个事物有七八、十来个状态也不稀奇，如果完全使用状态模式就会有太多的子类，不好管理，这个需要大家在项目自己衡量。其实有很大方式解决这个状态问题，比如在数据库中建立一个状态表，然后根据状态执行相应的操作，这个也不复杂，看大家的习惯和嗜好了。

状态模式使用于当某个对象在它的状态发生改变时，它的行为也随着发生比较大的变化，也就是说行为是受状态约束的情况下可以使用状态模式，而且状态模式使用时对象的状态最好不要超过五个，防止你写子类写疯掉。

上面的例子可能比较复杂，请各位看官耐心的看，看完我想肯定有所收获。我翻遍了所有能找到的资料（至少也有十几本，其中有几本原文的书还是很很不错的，我举这个电梯的例子也是从《Design Pattern for Dummies》这本书来激发出来的），基本（基本哦，还是有几本讲的不错）上没有一本把这个状态模式讲透彻的，我不敢说我就讲的透彻，大家都只讲了一个状态到另一个状态过渡，状态间的过渡是固定的，举个简单的例子：



这个状态图是很多书上都有的，状态A只能变更到状态B，状态B再变更到状态C，例子举的最多的就是TCP监听的例子，TCP有三个状态：等待，连接，断开，然后这三个状态中按照顺序循环变更，按照这个状态变更来讲解状态模式，我认为是不太合适的，为什么呢？你在项目中太少看到一个状态只能过渡到另一个状态情形，项目中遇到的大多数情况都是一个状态可以转换为几种状态，如下图：



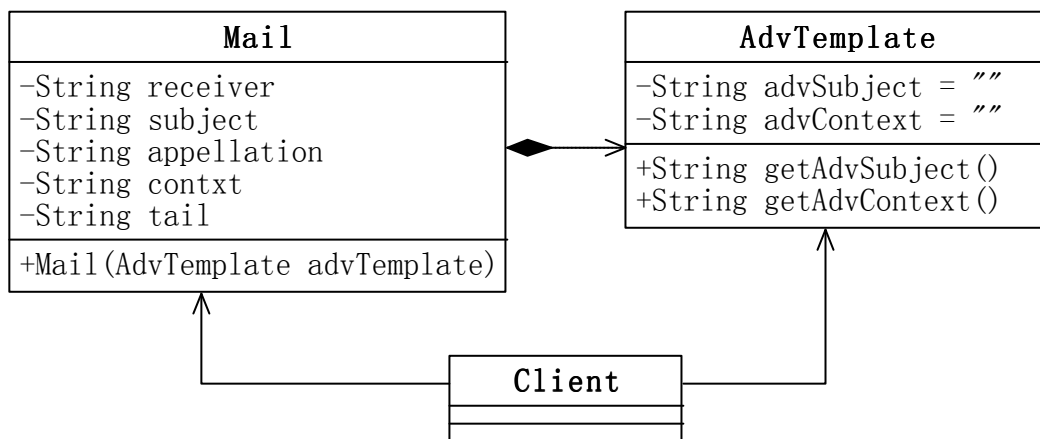
状态B可以转换为状态C也可以转换为状态D，而状态D呢也可以转换为状态A或状态B，这在项目分析过程中有一个叫状态图可以完整的展示这种蜘蛛网结构，举个实际例子来说，一些收费网站的用户就有很多状态，比如普通用户，普通会员，VIP会员，白金级用户等等，这个状态的变更你不允许跳跃？！这不可能，所以我在例子中就举了一个比较复杂的应用，基本上可以实现状态间自由切换，这才是最经常用到的状态模式。然后我再提问个问题，状态间的自由切换，那会有很多种呀，你要一个一个的牢记一遍吗？比如上面那个电梯的例子，我要一个正常的电梯运行逻辑，规则是开门->关门->运行->停止；还要一个紧急状态（比如火灾）下的运行逻辑，关门->停止，紧急状态电梯当然不能用了；再要一个维修状态下的运行逻辑，这个状态任何情况都可以，开着门电梯运行？可以！门来回开关？可以！永久停止不动？可以！那这怎么实现呢？需要我们把已有的几种状态按照一定的顺序再重新组装一下，那这个是什么模式？什么模式？大声点！建造者模式！对，建造模式+状态模式会起到非常好的封装作用。

再往深里面扯几句，应该有部分读者做过 workflow 开发，如果不是土制框架的话，就应该有个状态机管理（即使是土制框架也应该有），比如一个 *Activity*（节点）有初始化状态（*Initialized State*）、挂起状态（*Suspended State*）、完成状态（*Completed State*）等等，流程实例也是有这么多状态，那这些状态怎么管理呢？通过状态机（*State Machine*）来管理，那状态机是个什么东西呢？就是我们上面提到的 *Context* 类的升级变态 BOSS！

第 20 章 原型模式【Prototype Pattern】

今天我们来讲原型模式，这个模式的简单程度是仅次于单例模式和迭代器模式，非常简单，但是要使用好这个模式还有很多注意事项。我们通过一个例子来解释一下什么是原型模式。

现在电子账单越来越流行了，比如你的信用卡，到月初的时候银行就会发一份电子邮件到你邮箱中，说你这个月消费了多少，什么时候消费的，积分是多少等等，这个是每个月发一次，但是还有一种也是银行发的邮件你肯定有印象：广告信，现在各大银行的信用卡部门都在拉拢客户，电子邮件是一种廉价、快捷的通讯方式，你用纸质的广告信那个费用多高呀，比如我今天推出一个信用卡刷卡抽奖活动，通过电子账单系统可以一个晚上发送给 600 万客户，为什么要用电子账单系统呢？直接找个发垃圾邮件不就解决问题了吗？是个好主意，但是这个方案在金融行业是行不通的，银行发这种邮件是有要求的，一是一般银行都要求个性化服务，发过去的邮件上总有一些个人信息吧，比如“XX 先生”，“XX 女士”等等，二是邮件的到达率有一定的要求，由于大批量的发送邮件会被接收方邮件服务器误认为是垃圾邮件，因此在邮件头要增加一些伪造数据，以规避被反垃圾邮件引擎误认为是垃圾邮件；从这两方面考虑广告信的发送也是电子账单系统（电子账单系统一般包括：账单分析、账单生成器、广告信管理、发送队列管理、发送机、退信处理、报表管理等）的一个子功能，我们今天就来考虑一下广告信这个模块是怎么开发的。那既然是广告信，肯定需要一个模版，然后再从数据库中把客户的信息一个一个的取出，放到模版中生成一份完整的邮件，然后扔给发送机进行发送处理，我们来看类图：



在类图中 **AdvTemplate** 是广告信的模板，一般都是从数据库取出，生成一个 BO 或者是 DTO，我们这里使用一个静态的值来做代表；**Mail** 类是一封邮件类，发送机发送的就是这个类，我们先来看看我们的程序：

```
public class AdvTemplate {
```

```
//广告信名称
private String advSubject = "XX银行国庆信用卡抽奖活动";

//广告信内容
private String advContext = "国庆抽奖活动通知：只要刷卡就送你1百万! ....";

//取得广告信的名称
public String getAdvSubject(){
    return this.advSubject;
}

//取得广告信的内容
public String getAdvContext(){
    return this.advContext;
}
}
```

我们再来看邮件类：

```
public class Mail {
    //收件人
    private String receiver;

    //邮件名称
    private String subject;

    //称谓
    private String appellation;

    //邮件内容
    private String context;

    //邮件的尾部，一般都是加上“xxx版权所有”等信息
    private String tail;

    //构造函数
    public Mail(AdvTemplate advTemplate){
        this.context = advTemplate.getAdvContext();
        this.subject = advTemplate.getAdvSubject();
    }

    //以下为getter/setter方法
    public String getReceiver() {
```



```
        return receiver;
    }

    public void setReceiver(String receiver) {
        this.receiver = receiver;
    }

    public String getSubject() {
        return subject;
    }

    public void setSubject(String subject) {
        this.subject = subject;
    }

    public String getAppellation() {
        return appellation;
    }

    public void setAppellation(String appellation) {
        this.appellation = appellation;
    }

    public String getContxt() {
        return contxt;
    }

    public void setContxt(String contxt) {
        this.contxt = contxt;
    }

    public String getTail() {
        return tail;
    }

    public void setTail(String tail) {
        this.tail = tail;
    }
}
```

Mail 就是一个业务对象，我们再来看业务场景类是怎么调用的：

```

public class Client {
    //发送账单的数量，这个值是从数据库中获得
    private static int MAX_COUNT = 6;

    public static void main(String[] args) {
        //模拟发送邮件
        int i=0;
        //把模板定义出来，这个是从数据库中获得
        Mail mail = new Mail(new AdvTemplate());
        mail.setTail("XX银行版权所有");
        while(i<MAX_COUNT){
            //以下是每封邮件不同的地方
            mail.setAppellation(getRandString(5)+" 先生（女士）");
            mail.setReceiver(getRandString(5) + "@" + getRandString(8)+".com");

            //然后发送邮件
            sendMail(mail);
            i++;
        }
    }

    //发送邮件
    public static void sendMail(Mail mail){
        System.out.println("标题: "+mail.getSubject() + "\t收件人: "+mail.getReceiver()+"\t....发送成功! ");
    }

    //获得指定长度的随机字符串
    public static String getRandString(int maxLength){
        String source = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
        StringBuffer sb = new StringBuffer();
        Random rand = new Random();
        for(int i=0;i<maxLength;i++){
            sb.append(source.charAt(rand.nextInt(source.length())));
        }
        return sb.toString();
    }
}

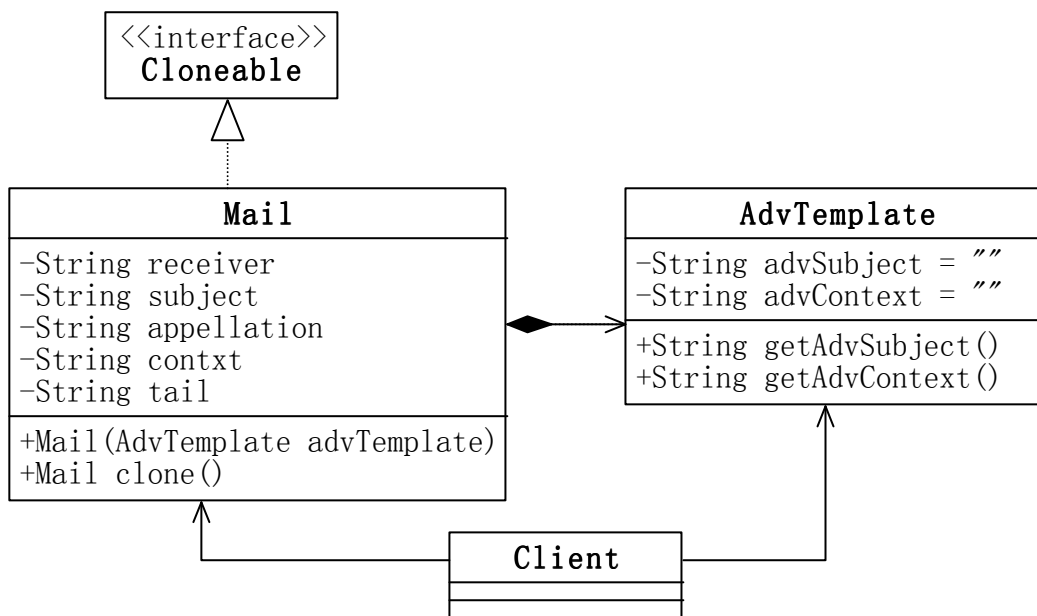
```

运行结果如下：

标题: XX银行国庆信用卡抽奖活动	收件人: fjQUm@ZnkyPSsL.com发送成功!
标题: XX银行国庆信用卡抽奖活动	收件人: ZIKnC@NOKdloNM.com发送成功!
标题: XX银行国庆信用卡抽奖活动	收件人: zNkMI@HpMMSZaz.com发送成功!
标题: XX银行国庆信用卡抽奖活动	收件人: oMTFA@uBwkRjxa.com发送成功!
标题: XX银行国庆信用卡抽奖活动	收件人: TquWT@TLLLVNFja.com发送成功!
标题: XX银行国庆信用卡抽奖活动	收件人: rkQbp@mfATHDQH.com发送成功!

由于是随机数, 每次运行都由所差异, 不管怎么样, 我们这个电子账单发送程序时写出来了, 也能发送出来了, 我们再来仔细的想想, 这个程序是否有问题? 你看, 你这是一个线程在运行, 也就是你发送是单线程的, 那按照一封邮件发出去需要 0.02 秒(够小了, 你还要到数据库中取数据呢), 600 万封邮件需要...我算算(掰指头计算中...), 恩, 是 33 个小时, 也就是一个整天都发送不完, 今天发送不完, 明天的账单又产生了, 积累积累, 激起甲方人员一堆抱怨, 那怎么办?

好办, 把 sendMail 修改为多线程, 但是你把 sendMail 修改为多线程还是有问题的呀, 你看哦, 产生第一封邮件对象, 放到线程 1 中运行, 还没有发送出去; 线程 2 呢也启动了, 直接就把邮件对象 mail 的收件人地址和称谓修改掉了, 线程不安全了, 好了, 说到这里, 你会说这有 N 多种解决办法, 我们不多说, 我们今天就说一种, 使用原型模式来解决这个问题, 使用对象的拷贝功能来解决这个问题, 类图稍作修改, 如下图:



增加了一个 Cloneable 接口, Mail 实现了这个接口, 在 Mail 类中重写了 clone() 方法, 我们来看 Mail 类的改变:

```
public class Mail implements Cloneable{
    //收件人
    private String receiver;

    //邮件名称
    private String subject;

    //称谓
    private String appellation;

    //邮件内容
    private String contxt;

    //邮件的尾部，一般都是加上“xxx版权所有”等信息
    private String tail;

    //构造函数
    public Mail(AdvTemplate advTemplate){
        this.contxt = advTemplate.getAdvContext();
        this.subject = advTemplate.getAdvSubject();
    }

    @Override
    public Mail clone(){
        Mail mail =null;
        try {
            mail = (Mail)super.clone();
        } catch (CloneNotSupportedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        return mail;
    }

    //以下为getter/setter方法
    public String getReceiver() {
        return receiver;
    }

    public void setReceiver(String receiver) {
        this.receiver = receiver;
    }

    public String getSubject() {
```

```
        return subject;
    }

    public void setSubject(String subject) {
        this.subject = subject;
    }

    public String getAppellation() {
        return appellation;
    }

    public void setAppellation(String appellation) {
        this.appellation = appellation;
    }

    public String getContxt() {
        return contxt;
    }

    public void setContxt(String contxt) {
        this.contxt = contxt;
    }

    public String getTail() {
        return tail;
    }

    public void setTail(String tail) {
        this.tail = tail;
    }
}
```

就黄色体部分做了修改，大家可能看着这个类有点奇怪，先保留你的好奇，我们继续讲下去，我会给你解答的，看 Client 类的改变：

```
public class Client {
    //发送账单的数量，这个值是从数据库中获得
    private static int MAX_COUNT = 6;

    public static void main(String[] args) {
        //模拟发送邮件
    }
}
```

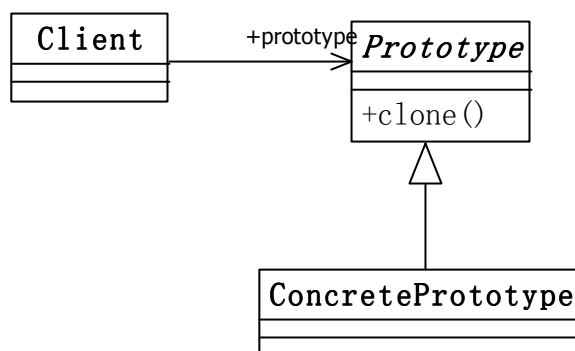
```

int i=0;
//把模板定义出来，这个是从数据中获得
Mail mail = new Mail(new AdvTemplate());
mail.setTail("xx银行版权所有");
while(i<MAX_COUNT){
    //以下是每封邮件不同的地方
    Mail cloneMail = mail.clone();
    cloneMail.setAppellation(getRandString(5)+" 先生(女士)");
    cloneMail.setReceiver(getRandString(5) + "@" +
getRandString(8)+".com");

    //然后发送邮件
    sendMail(cloneMail);
    i++;
}
}
}

```

运行结果不变，一样完成了电子广告信的发送功能，而且 sendMail 即使是多线程也没有关系，看到 mail.clone() 这个方法了吗？把对象拷贝一份，产生一个新的对象，和原有对象一样，然后再修改细节的数据，如设置称谓，设置收件人地址等等。这种不通过 new 关键字来产生一个对象，而是通过对象拷贝来实现的模式就叫做原型模式，其通用类图如下：



这个模式的核心是一个 clone 方法，通过这个方法进行对象的拷贝，Java 提供了一个 Cloneable 接口来标示这个对象是可拷贝的，为什么说是“标示”呢？翻开 JDK 的帮助看看 Cloneable 是一个方法都没有的，这个接口只是一个标记作用，在 JVM 中具有这个标记的对象才有可能被拷贝，那怎么才能从“有可能被拷贝”转换为“可以被拷贝”呢？方法是覆盖 clone() 方法，是的，你没有看错是重写 clone() 方法，看看我们上面 Mail 类：

```
@Override
public Mail clone(){}

```

在 clone() 方法上增加了一个注解 @Override, 没有继承一个类为什么可以重写呢? 在 Java 中所有类的老祖宗是谁? 对嘛, Object 类, 每个类默认都是继承了这个类, 所以这个用上重写是非常正确的。

原型模式虽然很简单, 但是在 Java 中使用原型模式也就是 clone 方法还是有一些注意事项的, 我们通过几个例子一个一个解说 (如果你对 Java 不是很感冒的话, 可以跳开以下部分)。

对象拷贝时, 类的构造函数是不会被执行的。 一个实现了 Cloneable 并重写了 clone 方法的类 A, 有一个无参构造或有参构造 B, 通过 new 关键字产生了一个对象 S, 再然后通过 S.clone() 方式产生了一个新的对象 T, 那么在对象拷贝时构造函数 B 是不会被执行的, 我们来写一小段程序来说明这个问题:

```
public class Thing implements Cloneable{
    public Thing(){
        System.out.println("构造函数被执行了...");
    }

    @Override
    public Thing clone(){
        Thing thing=null;
        try {
            thing = (Thing)super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return thing;
    }
}

```

然后我们再来写一个 Client 类, 进行对象的拷贝:

```
public class Client {

    public static void main(String[] args) {
        //产生一个对象
        Thing thing = new Thing();

        //拷贝一个对象
    }
}

```

```
        Thing cloneThing = thing.clone();

    }

}
```

运行结果如下：

构造函数被执行了...

对象拷贝时确实构造函数没有被执行，这个从原理来讲也是可以讲得通的，Object 类的 clone 方法的原理是从内存中（具体的说就是堆内存）以二进制流的方式进行拷贝，重新分配一个内存块，那构造函数没有被执行也是非常正常的了。

浅拷贝和深拷贝问题。在解释什么是浅拷贝什么是拷贝前，我们先来看个例子：

```
public class Thing implements Cloneable{
    //定义一个私有变量
    private ArrayList<String> arrayList = new ArrayList<String>();

    @Override
    public Thing clone(){
        Thing thing=null;
        try {
            thing = (Thing)super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return thing;
    }

    //设置HashMap的值
    public void setValue(String value){
        this.arrayList.add(value);
    }

    //取得arrayList的值
    public ArrayList<String> getValue(){
        return this.arrayList;
    }
}
```



```
}
```

在 Thing 类中增加一个私有变量 arrayLis, 类型为 ArrayList, 然后通过 setValue 和 getValue 分别进行设置和取值, 我们来看场景类:

```
public class Client {  
  
    public static void main(String[] args) {  
        //产生一个对象  
        Thing thing = new Thing();  
        //设置一个值  
        thing.setValue("张三");  
  
        //拷贝一个对象  
        Thing cloneThing = thing.clone();  
        cloneThing.setValue("李四");  
  
        System.out.println(thing.getValue());  
    }  
}
```

大家猜想一下运行结果应该是什么? 是就一个“张三”吗? 运行结果如下:

```
[张三, 李四]
```

怎么会有李四呢? 是因为 Java 做了一个偷懒的拷贝动作, Object 类提供的方法 clone 只是拷贝本对象, 其对象内部的数组、引用对象等都不拷贝, 还是指向原生对象的内部元素地址, 这种拷贝就叫做浅拷贝, 确实是非常浅, 两个对象共享了一个私有变量, 你改我改大家都能改, 是一个种非常不安全的方式, 在实际项目中使用还是比较少的。你可能会比较奇怪, 为什么在 Mail 那个类中就可以使用使用 String 类型, 而不会产生由浅拷贝带来的问题呢? 内部的数组和引用对象才不拷贝, 其他的原始类型比如 int, long, String (Java 就希望你把 String 认为是基本类型, String 是没有 clone 方法的) 等都会被拷贝的。

浅拷贝是有风险的, 那怎么才能深入的拷贝呢? 我们修改一下我们的程序:

```
public class Thing implements Cloneable{
```

```
//定义一个私有变量
private ArrayList<String> arrayList = new ArrayList<String>();

@Override
public Thing clone(){
    Thing thing=null;
    try {
        thing = (Thing)super.clone();
        thing.arrayList = (ArrayList<String>)this.arrayList.clone();
    } catch (CloneNotSupportedException e) {
        e.printStackTrace();
    }
    return thing;
}
}
```

仅仅增加了黄色部分，Client 类没有任何改变，运行结果如下：

[张三]

这个实现了完全的拷贝，两个对象之间没有任何的瓜葛了，你修改你的，我修改我的，不相互影响，这种拷贝就叫做深拷贝，深拷贝还有一种实现方式就是通过自己写二进制流来操作对象，然后实现对象的深拷贝，这个大家有时间自己实现一下。

深拷贝和浅拷贝建议不要混合使用，一个类中某些引用使用深拷贝某些引用使用浅拷贝，这是一种非常差的设计，特别是在涉及到类的继承，父类有几个引用的情况就非常的复杂，建议的方案深拷贝和浅拷贝分开实现。

Clone 与 final 两对冤家。对象的 clone 与对象内的 final 属性是由冲突的，我们举例来说明这个问题：

```
public class Thing implements Cloneable{
    //定义一个私有变量
    private final ArrayList<String> arrayList = new ArrayList<String>();

    @Override
    public Thing clone(){
        Thing thing=null;
        try {
```

```
        thing = (Thing)super.clone();
        this.arrayList = (ArrayList<String>)this.arrayList.clone();
    } catch (CloneNotSupportedException e) {
        e.printStackTrace();
    }
    return thing;
}
}
```

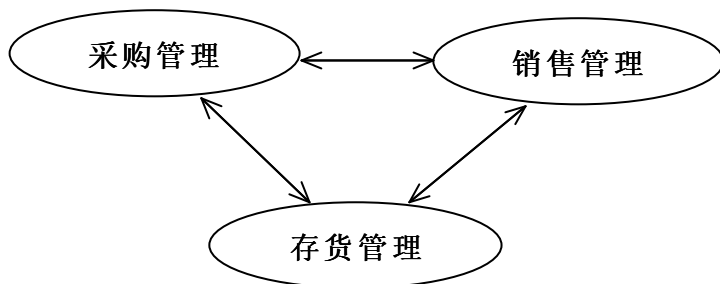
黄色的部分仅仅增加了一个 final 关键字，然后编译器就报灰色的部分错误，正常呀，final 类型你还想重写设值呀！完蛋了，你要实现深拷贝的梦想在 final 关键字的威胁下破灭了，路总是有的，我们来想想怎么修改这个方法：删除掉 final 关键字，这是最便捷最安全最快速的方式，你要使用 clone 方法就在类的成员变量上不要增加 final 关键字。

原型模式适合在什么场景使用？一是类初始化需要消化非常多的资源，这个资源包括数据、硬件资源等；二是通过 new 产生一个对象需要非常繁琐的数据准备或访问权限，则可以使用原型模式；三是一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多个对象供调用者使用。在实际项目中，原型模式很少单独出现，一般是和工厂方法模式一起出现，通过 clone 的方法创建一个对象，然后由工厂方法提供给调用者。

原型模式先产生出一个包含大量共有信息的类，然后可以拷贝出副本，修正细节信息，建立了一个完整的个性对象。不知道大家有没有看过施瓦辛格演的《第六日》这个电影，电影的主线也就是一个人被复制，然后正本和副本对招，我们今天讲的原型模式也就是由一个正本可以创建多个副本的概念，可以这样理解一个对象的产生可以不由零开始，直接从一个已经具备一定雏形的对象克隆，然后再修改为一个生产需要的对象。也就是说，产生一个人，可以不从 1 岁长到 2 岁，再 3 岁…，也可以直接找一个人，从其身上获得 DNS，然后克隆一个，直接修改一下就是 3 岁的了！，我们讲的原型模式也就是这样的功能，是紧跟时代潮流的。

第 21 章 中介者模式【Mediator Pattern】

各位好，大家都是来自五湖四海，都要生存，于是都找了个靠山——公司，给你发薪水的地方，那公司就要想尽办法盈利赚钱，盈利方法则不尽相同，但是作为公司都有相同三个环节：采购、销售和库存，这个怎么说呢？比如一个软件公司，要开发软件，需要开发环境吧，Windows 操作系统，数据库产品等，这你得买吧，那就是采购，开发完毕一个产品还要把产品推销出去，推销出去了大家才有钱赚，不推销出去大家都去喝西北风呀，既然有产品就必然有库存，软件产品也有库存，你总要拷贝吧，虽然是不需要占用库房空间，那也是要占用光盘或硬盘，这也是库存，再比如做咨询服务的公司，它要采购什么？采购知识，采购经验，这是这类企业的生存之本，销售的也是知识和经验，库存同样是知识和经验。尽然进销存是这么的重要，我们今天来讲讲它的原理和设计，我相信很多人都已经开发过这种类型的软件，基本上都形成了固定套路，不管是单机版还是网络版，一般的做法都是通过数据库来完成相关产品的管理，相对来说还是比较简单的项目，三个模块之间的示意图如下：



我们从这个示意图上可以看出，三个模块是相互依赖的，基本上是你中有我，我中有你，为什么呢？我们就以一个终端销售商（什么是终端销售商？就是以服务最终客户为目标的企业，比如 XX 超市，国美电器等等）为例子，比如采购部门要采购 IBM 型号的电脑了，它是根据什么来决定采购的呢？根据两个要素：

销售情况。销售部门要反馈销售情况，畅销就多采购，滞销就不采购；

库存情况。即使是畅销产品，库存都有 1000 台了，每天才卖出去 10 台，还要采购吗？！

销售模块是企业的盈利核心，也是对其他两个模块有影响的：

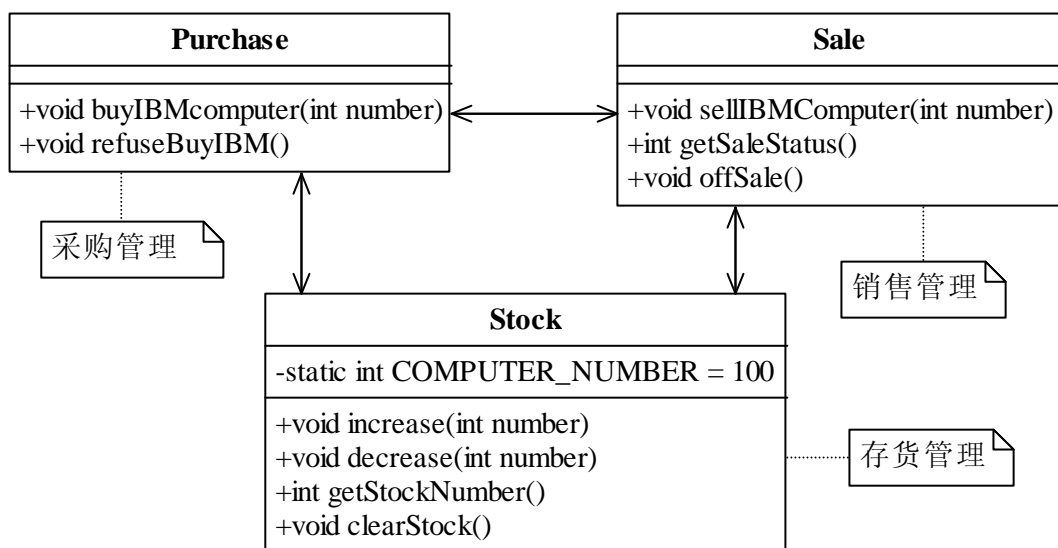
库存情况。库房有货，才能销售，没货空手套白狼是不行的；

督促采购。在特殊情况下，比如一个企业客户一下子要卖 100 台电脑，你库存里自由 80 台，怎么办？催采购部门赶快采购呀！

同样的，库存管理也对其他两个模块有影响，库房是有容积限制的，不可能无限大，所以就有了清仓处理，那就要求采购部门别采购了，同时销售部门赶快打折销售。

从以上分析来看，这三个模块都是有自己的行为，并且与其他模块之间的行为产生关联关系，就类似

我们在办公室中的同事，大家各干各的活，但是彼此之间还是有交叉的，于是乎大家之间就产生紧耦合，也就是一个团队。我们先来实现这个进销存，先看类图：



Purchase 负责采购管理，buyIBMComputer 是指定了采购 IBM 电脑，refuseBuyIBM 是不再采购 IBM 了，源代码如下：

```
public class Purchase {

    //采购IBM型号的电脑
    public void buyIBMcomputer(int number){
        //访问库存
        Stock stock = new Stock();
        //访问销售
        Sale sale = new Sale();

        //电脑的销售情况
        int saleStatus = sale.getSaleStatus();

        if(saleStatus>80){ //销售情况良好
            System.out.println("采购IBM电脑："+number + "台");
            stock.increase(number);
        }else{ //销售情况不好
            int buyNumber = number/2; //折半采购
            System.out.println("采购IBM电脑："+buyNumber+ "台");
        }
    }

    //不再采购IBM电脑
}
```

```
public void refuseBuyIBM(){
    System.out.println("不再采购IBM电脑");
}
}
```

Purchase 定义了采购电脑的一个标准，如果销售情况比较好，大于 80 分，你让我采购多少我就采购多少；销售情况不好，你让我采购 100 台，我就采购 50 台，对折采购。电脑采购完毕了，肯定要放到库房中，因此要调用库存的方法，增加库存电脑数量。我们继续来看库房 Stock 类：

```
public class Stock {
    //刚开始有100台电脑
    private static int COMPUTER_NUMBER =100;

    //库存增加
    public void increase(int number){
        COMPUTER_NUMBER = COMPUTER_NUMBER + number;
        System.out.println("库存数量为: "+COMPUTER_NUMBER);
    }

    //库存降低
    public void decrease(int number){
        COMPUTER_NUMBER = COMPUTER_NUMBER - number;
        System.out.println("库存数量为: "+COMPUTER_NUMBER);
    }

    //获得库存数量
    public int getStockNumber(){
        return COMPUTER_NUMBER;
    }

    //存货压力大了，就要通知采购人员不要采购，销售人员要尽快销售
    public void clearStock(){
        Purchase purchase = new Purchase();
        Sale sale = new Sale();
        System.out.println("清理存货数量为: "+COMPUTER_NUMBER);
        //要求折价销售
        sale.offSale();
        //要求采购人员不要采购
        purchase.refuseBuyIBM();
    }
}
```

库房中的货物数量肯定有增加和减少了，同时库房还有一个容量显示，达到一定的容量后就要求对一些商品进行折价处理，腾出更多的空间容纳新产品，于是就有了 clearStock 方法，既然是清仓处理肯定就要折价销售了，于是在 Sale 这个类中就有了 offSale 方法，我们来看 Sale 源代码：

```
public class Sale {

    //销售IBM型号的电脑
    public void sellIBMComputer(int number){
        //访问库存
        Stock stock = new Stock();
        //访问采购
        Purchase purchase = new Purchase();

        if(stock.getStockNumber()<number){ //库存数量不够销售
            purchase.buyIBMcomputer(number);
        }
        System.out.println("销售IBM电脑"+number+"台");
        stock.decrease(number);
    }

    //反馈销售情况,0—100之间变化,0代表根本就没人卖,100代表非常畅销,出一个卖一个
    public int getSaleStatus(){
        Random rand = new Random(System.currentTimeMillis());
        int saleStatus = rand.nextInt(100);
        System.out.println("IBM电脑的销售情况为: "+saleStatus);
        return saleStatus;
    }

    //折价处理
    public void offSale(){
        //库房有多少卖多少
        Stock stock = new Stock();
        System.out.println("折价销售IBM电脑"+stock.getStockNumber()+"台");
    }
}
```

Sale 类中的 getSaleStatus 是获得销售情况，这个当然要出现在 Sale 类中了，记住恰当的类放到恰当的类中，销售情况当然只有销售人员才能反馈出来了，通过百分制的机制衡量销售情况。我们再来看场景类是怎么运行的：

```
public class Client {  
  
    public static void main(String[] args) {  
        //采购人员采购电脑  
        System.out.println("-----采购人员采购电脑-----");  
        Purchase purchase = new Purchase();  
        purchase.buyIBMcomputer(100);  
  
        //销售人员销售电脑  
        System.out.println("\n-----销售人员销售电脑-----");  
        Sale sale = new Sale();  
        sale.sellIBMComputer(1);  
  
        //库房管理人员管理库存  
        System.out.println("\n-----库房管理人员清库处理-----");  
        Stock stock = new Stock();  
        stock.clearStock();  
    }  
}
```

我们在场景类中模拟了三种人员类型的活动：采购人员采购电脑，销售人员销售电脑，库管员管理库存，运行结果如下：

-----采购人员采购电脑-----

IBM 电脑的销售情况为：95

采购 IBM 电脑:100 台

库存数量为：200

-----销售人员销售电脑-----

销售 IBM 电脑 1 台

库存数量为：199

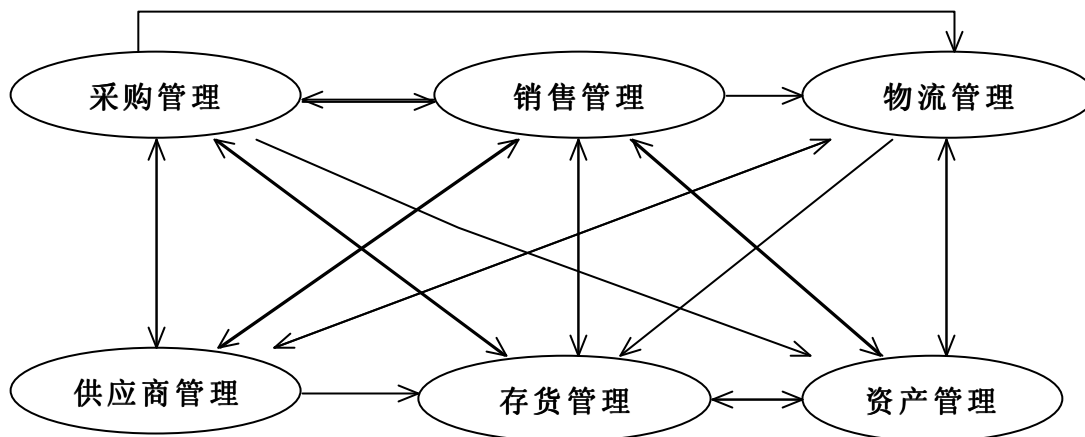
-----库房管理人员清库处理-----

清理存货数量为：199

折价销售 IBM 电脑 199 台

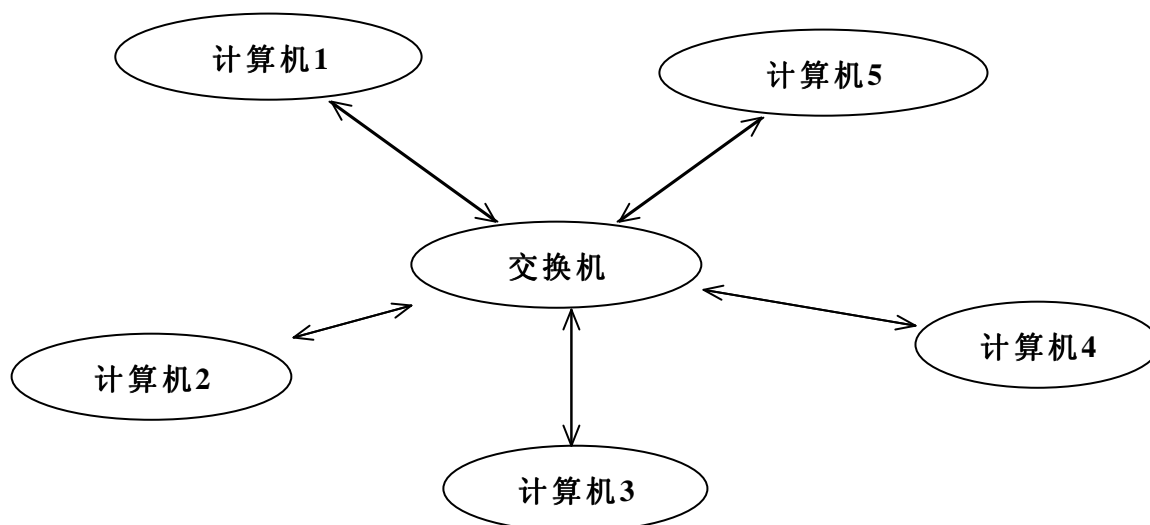
不再采购 IBM 电脑

运行结果也是我们期望的，三个不同类型的参与者完成了各自的活动。但是…但是…，你难道就没有发现这三个类间是彼此关联的吗？每个类都与其他两个类产生了关联关系，迪米特法则教育我们“每个类只和朋友类交流”，这个朋友类可不是越多越好，越多耦合性越大，改一个对象而要修改一片对象，这可不是面向对象设计所期望的，而且这还是就三个模块的情况，比较简单的一个小项目，如果有十个八个这样的模块是不是就要歇菜了，我们把进销存扩充一下，如下图的情况：

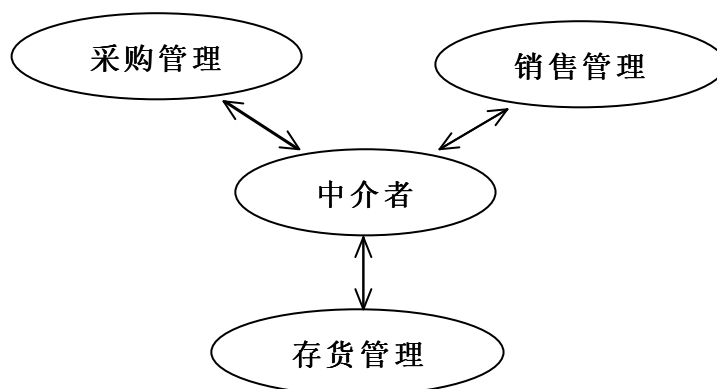


是不是看到一个蜘蛛网的结构，这个别说是编写程序了，就是给人看估计能让一大批的人昏倒！每个对象都要和其他的几个对象交流，对象越多，每个对象要交流的成本也就越多了，就单独维护这些对象的交流基本上就能让一大批程序员望而却步，这明摆着不是人干的活嘛！从这方面来，我们已经发现设计的缺陷，作为一个架构师，发现缺陷就要想办法来修改，我们思考一下，怎么来修改。

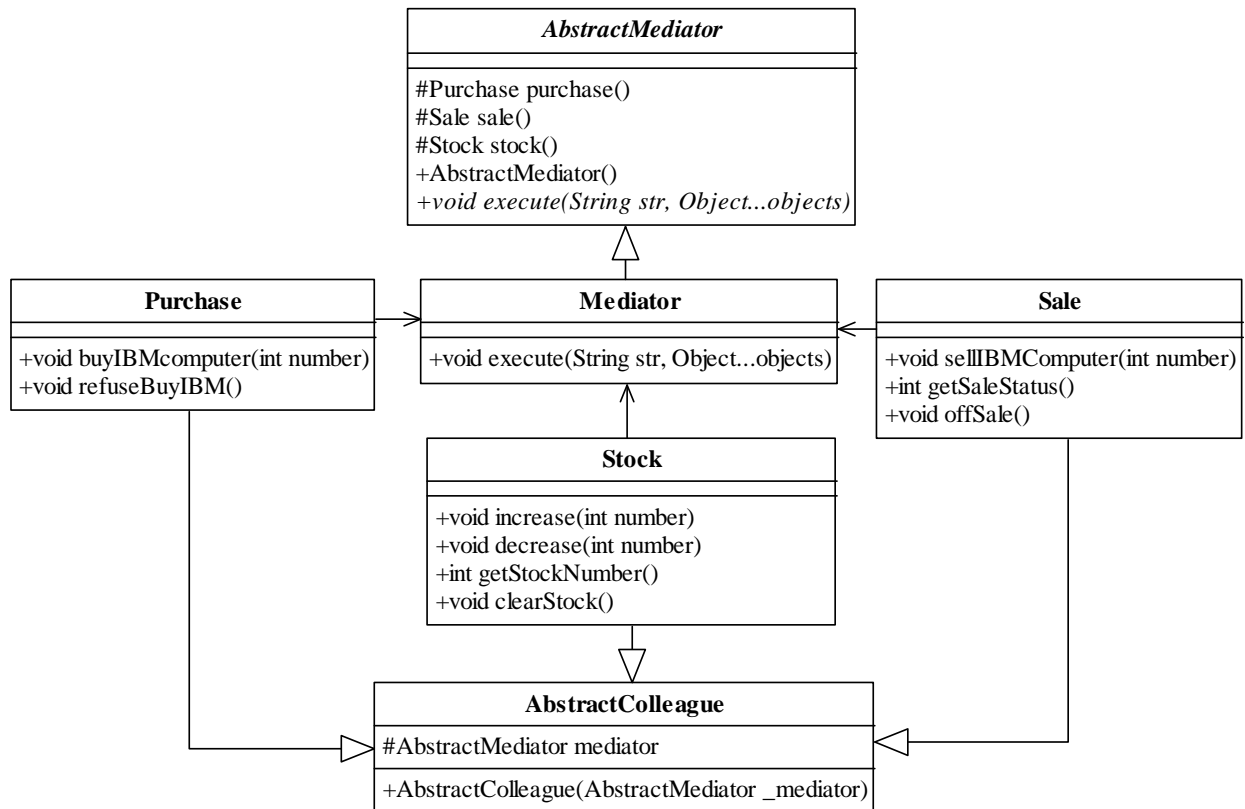
大家都是学计算机的，应该在上学的时候讲过一些网络的基本知识，还记得网络拓扑有几种类型吗？总线型，环型，星型，（什么？想不起来？！惩罚一下自己去），我们来想想星型拓扑是什么什么样子的，如下图：



星型网络拓扑中每个计算机通过交换机和其他计算机进行数据交换，各个计算机之间并没有直接出现交互的情况，结构简单，而且稳定，只要中间那个交换机不瘫痪，整个网络就不会发生大的故障，公司和网吧一般都采用星型网络，那也说明星型拓扑是深得民心，那我们来想想是不是可以把这种星型结构引入到我们的设计中呢？说干就干，我们先画一个示意图：



加入了一个中介者作为三个模块的交流核心，每个模块之间不再相互交流，要交流就通过中介者进行，每个模块只负责自己的业务逻辑，不属于自己的则丢给中介者来处理，看类图：



建立了两个抽象类 `AbstractMediator` 和 `AbstractColleague`，每个对象只是与中介者 `Mediator` 之间产生依赖，与其他对象之间没有直接的关系，`AbstractMediator` 的作用是把中介者的抽象定义，定义了一个抽象方法 `execute`，我们来看源代码：

```

public abstract class AbstractMediator {
    protected Purchase purchase;
    protected Sale sale;
    protected Stock stock;

    //构造函数
    public AbstractMediator(){
        purchase = new Purchase(this);
        sale = new Sale(this);
        stock = new Stock(this);
    }

    //中介者最重要的方法，叫做事件方法，处理多个对象之间的关系
    public abstract void execute(String str, Object...objects);
}

```

我们再来看具体的中介者，中介者可以根据业务的要求产生多个中介者（一般情况只有一个中介者），划分各个终结者的职责。我们来看 Mediator 源码：

```
public class Mediator extends AbstractMediator {

    //中介者最重要的方法
    public void execute(String str,Object...objects){
        if(str.equals("purchase.buy")){ //采购电脑
            this.buyComputer((Integer)objects[0]);
        }else if(str.equals("sale.sell")){ //销售电脑
            this.sellComputer((Integer)objects[0]);
        }else if(str.equals("sale.offsell")){ //折价销售
            this.offSell();
        }else if(str.equals("stock.clear")){ //清仓处理
            this.clearStock();
        }
    }

    //采购电脑
    private void buyComputer(int number){
        int saleStatus = super.sale.getSaleStatus();
        if(saleStatus>80){ //销售情况良好
            System.out.println("采购IBM电脑:"+number + "台");
            super.stock.increase(number);
        }else{ //销售情况不好
            int buyNumber = number/2; //折半采购
            System.out.println("采购IBM电脑: "+buyNumber+ "台");
        }
    }

    //销售电脑
    private void sellComputer(int number){
        if(super.stock.getStockNumber()<number){ //库存数量不够销售
            super.purchase.buyIBMcomputer(number);
        }
        super.stock.decrease(number);
    }

    //折价销售电脑
    private void offSell(){
        System.out.println("折价销售IBM电脑"+stock.getStockNumber()+"台");
    }
}
```

```
//清仓处理
private void clearStock(){
    //要求清仓销售
    super.sale.offSale();
    //要求采购人员不要采购
    super.purchase.refuseBuyIBM();
}
}
```

中介者 Mediator 有定义了多个 Private 方法，其目标是处理各个对象之间的依赖关系，即是说把原有一个对象要依赖多个对象的情况移到中介者的 Private 方法中实现，在实际项目中，一般的做法是中介者按照职责进行划分，每个中介者处理一个或多个类似的关联请求。

我们再来看 AbstractColleague 源码：

```
public abstract class AbstractColleague {
    protected AbstractMediator mediator;
    public AbstractColleague(AbstractMediator _mediator){
        this.mediator = _mediator;
    }
}
```

采购类 Purchase 的源码如下：

```
public class Purchase extends AbstractColleague{

    public Purchase(AbstractMediator _mediator){
        super(_mediator);
    }
    //采购IBM型号的电脑
    public void buyIBMcomputer(int number){
        super.mediator.execute("purchase.buy", number);
    }

    //不在采购IBM电脑
    public void refuseBuyIBM(){
        System.out.println("不再采购IBM电脑");
    }
}
```

Purchase 类是不是简化了很多，看着也清晰了很多，处理自己的职责，与外界有关系的事件处理则交给了中介者来完成。再来看 Stock 类：

```
public class Stock extends AbstractColleague {
    public Stock(AbstractMediator _mediator){
        super(_mediator);
    }
    //刚开始有100台电脑
    private static int COMPUTER_NUMBER =100;

    //库存增加
    public void increase(int number){
        COMPUTER_NUMBER = COMPUTER_NUMBER + number;
        System.out.println("库存数量为: "+COMPUTER_NUMBER);
    }

    //库存降低
    public void decrease(int number){
        COMPUTER_NUMBER = COMPUTER_NUMBER - number;
        System.out.println("库存数量为: "+COMPUTER_NUMBER);
    }

    //获得库存数量
    public int getStockNumber(){
        return COMPUTER_NUMBER;
    }

    //存货压力大了，就要通知采购人员不要采购，销售人员要尽快销售
    public void clearStock(){
        System.out.println("清理存货数量为: "+COMPUTER_NUMBER);
        super.mediator.execute("stock.clear");
    }
}
```

以下为 Sale 类的源码：

```
public class Sale extends AbstractColleague {
    public Sale(AbstractMediator _mediator){
        super(_mediator);
    }
}
```

```

//销售IBM型号的电脑
public void sellIBMComputer(int number){
    super.mediator.execute("sale.sell", number);
    System.out.println("销售IBM电脑"+number+"台");
}

//反馈销售情况,0—100之间变化,0代表根本就没人卖,100代表非常畅销,出1一个卖一个
public int getSaleStatus(){
    Random rand = new Random(System.currentTimeMillis());
    int saleStatus = rand.nextInt(100);
    System.out.println("IBM电脑的销售情况为: "+saleStatus);
    return saleStatus;
}

//折价处理
public void offSale(){
    super.mediator.execute("sale.offsell");
}
}

```

再来看场景类的变化:

```

public class Client {

    public static void main(String[] args) {
        AbstractMediator mediator = new Mediator();
        //采购人员采购电脑
        System.out.println("-----采购人员采购电脑-----");
        Purchase purchase = new Purchase(mediator);
        purchase.buyIBMcomputer(100);

        //销售人员销售电脑
        System.out.println("\n-----销售人员销售电脑-----");
        Sale sale = new Sale(mediator);
        sale.sellIBMComputer(1);

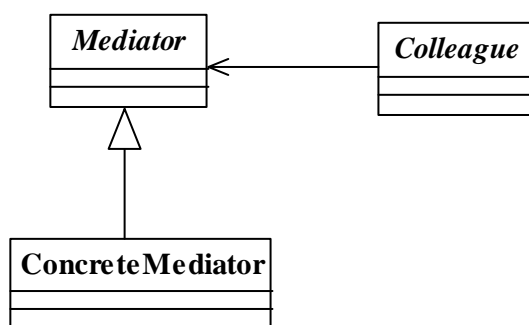
        //库房管理人员管理库存
        System.out.println("\n-----库房管理人员清库处理-----");
        Stock stock = new Stock(mediator);
        stock.clearStock();
    }
}

```

```
}
```

在场景类中增加了一个中介者，然后分别传递到三个同事类中，三个类都具有相同的特性：只负责处理自己的活动（行为），与自己无关的活动就丢给中介者处理，程序运行的结果是相同的。从项目设计上来看，加入了中介者，设计结构清晰了很多，而且类间的耦合性大大减少，代码质量也有了很大的提升。

以上讲解的模式就是中介者模式，在多个对象依赖的情况喜爱，通过加入中介者角色，取消了多个对象的关联或依关系，减少了对对象的耦合性，其通用类图为：



从类图中看，中介者模式有以下几部分组成：

抽象中介者（Mediator）角色：抽象中介者角色定义统一的接口用于各同事角色之间的通信。

具体中介者（Concrete Mediator）角色：具体中介者角色通过协调各同事角色实现协作行为，因此它必须依赖于各个同事角色。

同事（Colleague）角色：每一个同事角色都知道中介者角色，而且与其他的同事角色通信的时候，一定要通过中介者角色协作。每个同事类的行为分为两种：一种是同事本身的行为，比如改变对象本身的状态，处理自己的行为等等，这种方法叫做自发行为(Self-Method)，与其他的同事类或中介者没有任何的依赖；第二种是是必须依赖中介者才能完成的行为，叫做依赖方法（Dep-Method）。

在前几张的讲解中，每个模式的类都已经给出，但是却没有给出通用的源代码，感觉这种方式不是很好，因此从本章开始，都加入通用源代码。中介者的通用源码结构如下所示，先看中介者 Mediator 类：

```
public abstract class Mediator {
    //定义同事类
    protected ConcreteColleague1 c1;
    protected ConcreteColleague2 c2;

    //通过getter/setter方法把同事类注入进来
    public ConcreteColleague1 getC1() {
        return c1;
    }
}
```



```

    }
    public void setC1(ConcreteColleague1 c1) {
        this.c1 = c1;
    }
    public ConcreteColleague2 getC2() {
        return c2;
    }
    public void setC2(ConcreteColleague2 c2) {
        this.c2 = c2;
    }

    //中介者模式的业务逻辑
    public abstract void doSomething1();
    public abstract void doSomething2();
}

```

在 Mediator 抽象类中我们只定义了同事类的注入，为什么使用同事实现类注入而不使用抽象类注入呢？那是因为同事类虽然有抽象，但是这个抽象也太抽象了，根本就没有每个同事类所必须要完成的业务方法，当然如果每个同事类都有相同的方法，比如 execute, handler 等，那当然注入抽象类，做到依赖倒转。

具体的中介者一般只有一个，其源码如下：

```

public class ConcreteMediator extends Mediator {

    @Override
    public void doSomething1() {
        //调用同事类的方法，只要是public方法都可以调用
        super.c1.selfMethod1();
        super.c2.selfMethod2();
    }

    public void doSomething2() {
        super.c1.selfMethod1();
        super.c2.selfMethod2();
    }
}

```

中介者所具有的方法 doSomething1 和 doSomething2 都是比较复杂的业务逻辑，都是为同事类服务的，

其实现是依赖了各个同事类来完成。

同事类的基类如下：

```
public abstract class Colleague {  
    protected Mediator mediator;  
    public Colleague(Mediator _mediator){  
        this.mediator = _mediator;  
    }  
}
```

这个基类也是非常简单的，一般来说中介者模式中的抽象都是比较简单，是为了建立这个中介而服务的，同事实现类的源码如下：

```
public class ConcreteColleague1 extends Colleague {  
  
    //通过构造函数传递中介者  
    public ConcreteColleague1(Mediator _mediator){  
        super(_mediator);  
    }  
  
    //自有方法 self-method  
    public void selfMethod1(){  
        //处理自己的业务逻辑  
    }  
  
    //依赖方法 dep-method  
    public void depMethod1(){  
        //处理自己的业务逻辑  
        //自己不能处理的业务逻辑，委托给中介者处理  
        super.mediator.doSomething1();  
    }  
}
```

```
public class ConcreteColleague2 extends Colleague {  
  
    //通过构造函数传递中介者  
    public ConcreteColleague2(Mediator _mediator){  
        super(_mediator);  
    }  
}
```

```
}

//自有方法 self-method
public void selfMethod2(){
    //处理自己的业务逻辑
}

//依赖方法 dep-method
public void depMethod2(){
    //处理自己的业务逻辑
    //自己不能处理的业务逻辑，委托给中介者处理
    super.mediator.doSomething2();
}

}
```

为什么同事类要使用构造函数注入中介者而中介者使用 getter/setter 方式注入同事类呢？想过没有？那是因为同事类必须有中介者，而中介者可以只有部分同事类。

中介者模式的优点就是减少类间的依赖，把原有的一对多的依赖变成了一对一的依赖，同事类只依赖中介者，减少了依赖，当然也同时减低了类间的耦合。它的缺点呢就是中介者会膨胀的很大，而且逻辑会很复杂，因为所有的原本 N 个对象直接的相互依赖关系转换为中介者和同事类的依赖关系，同事类越多，中介者的逻辑就复杂。

中介者模式简单，但是简单不代表容易使用，这是 23 个模式中最容易被误用的模式。我们知道在面向对象的编程中，对象和对象之间必然会有依赖关系，如果你写了一个类，这个类和其他类没有任何的依赖关系，其他类也不依赖这个类，那这个类就是一个“孤岛”嘛，在项目中就没有必要存在！这就像是人类，如果一个人永远独立生活，与任何人都没有关系，那这个人基本上就算是野人了——排除在人类这个定义之外。类之间的依赖关系是必然存在的，一个类依赖多个类的情况也是存在的，存在即合理，那是否可以说只要有多个依赖关系就考虑使用中介者模式呢？答案是否定的，中介者模式未必就能帮你把原本凌乱的逻辑整理的清清楚楚，而且中介者模式也是有缺点的，这个缺点在不当的使用时会被放大，比如原本就简单的几个对象依赖关系，如果为了使用模式而加入了中介者，必然导致中介者的逻辑复杂化，因此中介者模式的使用需要“量力而行”，那在什么环境下才使用中介者模式呢？中介者模式适用于多个对象之间紧密耦合，耦合的标准可以这样来衡量：在类图中出现了蜘蛛网状结构，在这种情况下一定要考虑使用中介者模式，有利于把蜘蛛网梳理为一个星型结构，使原本复杂混乱关系变得清晰简单。

中介者模式也叫做调停者模式，是什么意思呢？一个对象要和 N 多个对象交流，是不是就像对象间的战争，很混乱，你中有我，我中有你，那怎么才能调停这种矛盾呢？加入一个中心，所有的类都和中心交

流，中心说怎么处理就这么处理，我们举一些在开发和生活中经常会碰到例子，再举四个例子如下：

机场调度中心。大家在每个机场都会看到有一个叫做“XX 机场调度中心”，这个是做什么用的呢？就是具体的中介者，调度每一架要降落和起飞的飞机，一架分机（同事类）飞到机场上空了，就询问（同事类方法）调度中心（中介者）“我是否可以降落”，“降落到那个跑道”，然后调度中心（中介者）查看其他飞机（同事类）情况，通知飞机降落，我们来设想一下，如果没有机场调度中心会是什么样子的：飞机到机场了，自己要先看看有没有飞机和自己一起降落的，有没有空跑道，停机位是否具备等等情况，这不是要了飞行员老命才怪！

MVC 框架。大家都应该使用过 Struts 吧，MVC 框架，其中的 C（Controller）就是一个中介者，叫做前端控制器（Front Controller），它的作用就是把 M（Model，业务逻辑）和 V（View，视图）隔离开，协调 M 和 V 协同工作，把 M 运行的结果和 V 代表的视图融合成一个前端可以展示的页面，减少 M 和 V 的依赖关系。MVC 框架已经成为一个非常流行、成熟的开发框架，这也是中介者模式优秀的一个体现。

C/S 结构。C/S 结构的应用也是一个典型的中介者模式，比如 MSN，张三发一个消息给李四，其过程应该是这样的：张三发送消息，MSN 服务器（中介者）接受到消息，查找李四，把消息发送到李四，同时通知张三，消息已经发送，在这里 MSN 服务器就是一个中转站，负责协调两个客户端的信息交流，与此相反的就是 IPMSG（也叫飞鸽）没有使用中介者，直接使用了 UDP 广播的方式，每个客户端既是客户端也是服务端。

中介服务。现在中介服务非常多，比如租房中介，出国中介，这些也都是中介模式的具体体现，比如你去租房子，如果没有房屋中介，你就必须一个一个小区的找，看看有没有空房子，有没有适合自己的房子，找到房子后还要和房东签合约，自己检查房屋的家具、水电煤等，有了中介后，你就省心多了，找中介，然后安排看房子，看中了，签合约，中介帮你检查房屋家具、水电煤等等。这也是中介模式的实际应用。

不知道大家有没有发觉，这章讲的中介者模式很少用到接口或者抽象类，这与依赖倒转原则是冲突的，确实是，这是什么原因呢？首先说同事类，既然是同事类而不是兄弟类（有相同的血缘）那也说明这些类之间是协作关系，完成不同的任务，处理不同的业务，所以不能在抽象类或接口中严格定义同事类必须具有的方法（从这点也可以看出继承是侵入性的），这是不合适的，就像你我是同事，虽然我们大家都是朝九晚五的上班，但是你跟我干的活肯定不同，不可能抽象出一个父类统一定义同事所必须有的方法，当然也有办法定义，每个同事都要吃饭，都要上厕所，那把这些最基本的信息封装到抽象中是可以的，但问题是这些最基本行为或属性是中介者模式要关心的嘛？如果两个对象不能提炼出共性，那就不要刻意的去追求两者的抽象，抽象只要定义出模式需要的角色即可。其次是中介者的原因，在一个项目中中介者模式可能被多个模块采用，每个中介者所围绕的同事类各不相同，你能抽象出一个具有共性的中介者吗？不可能，一个中介者抽象类一般只有一个实现者，除非中介者逻辑非常大，代码量非常高，这时才会出现多个中介

者的情况，所有，对中介者来说，抽象已经没有太多的必要。

中介者模式是一个非常好的封装模式，也是一个很容易被滥用的模式，一个对象依赖几个对象是再正常不过的事情，但是纯理论家就会要求使用中介者模式来封装这种依赖关系，这是非常危险的信号，使用中介模式就必然会带来中介者的膨胀问题，这在一个项目中时很不恰当的，那到底在什么情况下使用中介者模式呢？大家可以在如下的情况下尝试使用中介者模式：

- 1、N 个对象之间产生了相互的依赖关系，其中 N 大于 2，注意是相互的依赖；
- 2、多个对象有依赖关系，但是依赖的行为尚不确定或者有发生改变的可能，在这种情况下一般建议采用中介者模式，降低变更引起的风险扩散；
- 3、产品开发。其中一个明显的例子就是 MVC 框架，把这个应用到产品中，可以提升产品的性能和扩展性，但是作为项目开发就未必，项目是以交付投产为目标，而产品以稳定、高效、扩展为宗旨。

第 22 章 解释器模式【Interpreter Pattern】

在银行、证券类项目中，经常会有一些模型运算，通过对现有数据的统计、分析而预测不可知或未来可能发生的商业行为，模型运算大部分都是对海量数据的批量运算的结果，例如建立一个模型公式，分析一个城市的消费倾向，进而影响银行的营销和业务扩张方向，一般的模型运算都有一个或多个运算公式，通常是加减乘除四则运算，偶尔也有指数、开方等复杂运算。具体到一个金融业务中，类似的模型公式是非常复杂的，是，确实只有加减乘除四则运算，但是公式有可能有十多个参数，而且上百个业务品各有不同的取参路径，同时关联表的数据数量一般都大于百万级，呵呵，复杂了吧，不复杂那就不叫金融业务。我们今天来讲讲这个模型公式怎么实现。

需求：输入一个模型公式，然后输入模型中的参数，运算出结果；

设计要求：

- 1、公式可以修改；符合正常算术书写方式，例如 $a+b-c$ ；
- 2、高扩展性，未来增加指数、开方、极限、求导等运算逻辑时，减少改动量；
- 3、效率可以不用考虑，晚间批量运算；

需求不复杂，仅仅采用四则运算，每个程序员都可以写出来，非常简单。增加模型公式就复杂了，先解释一下为什么需要公式，而不采用直接计算的方法，例如有三个公式：

业务种类 1 要求的公式： $a+b+c-d$ ；

业务种类 2 要求的公式： $a+b+e-d$ ；

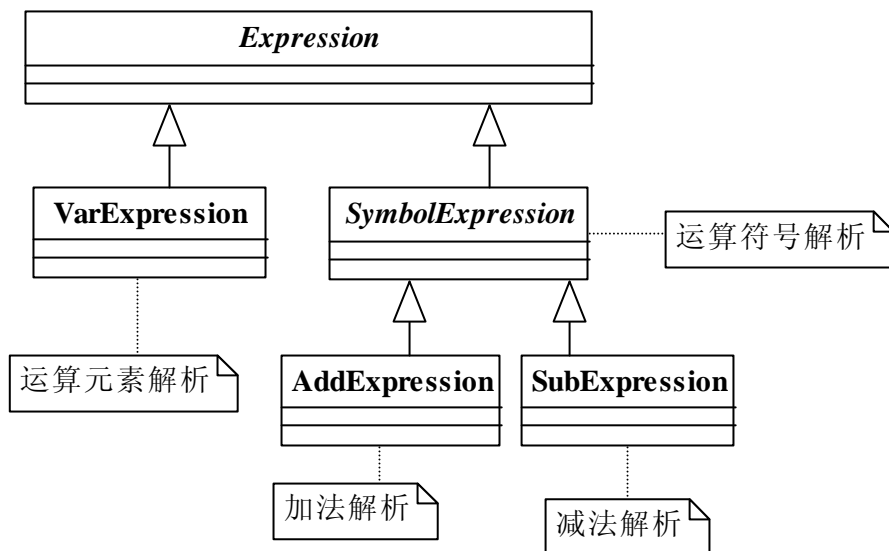
业务种类 3 要求的公式： $a-f$ ；

其中 a 、 b 、 c 、 d 、 e 、 f 参数的值都可以取得，如果使用直接计算数值的方法需要为每个品种写一个算法，现在仅仅是三个业务种类，那上百个品种呢？歇菜了吧！建立公式，然后通过公式运算才是王道。

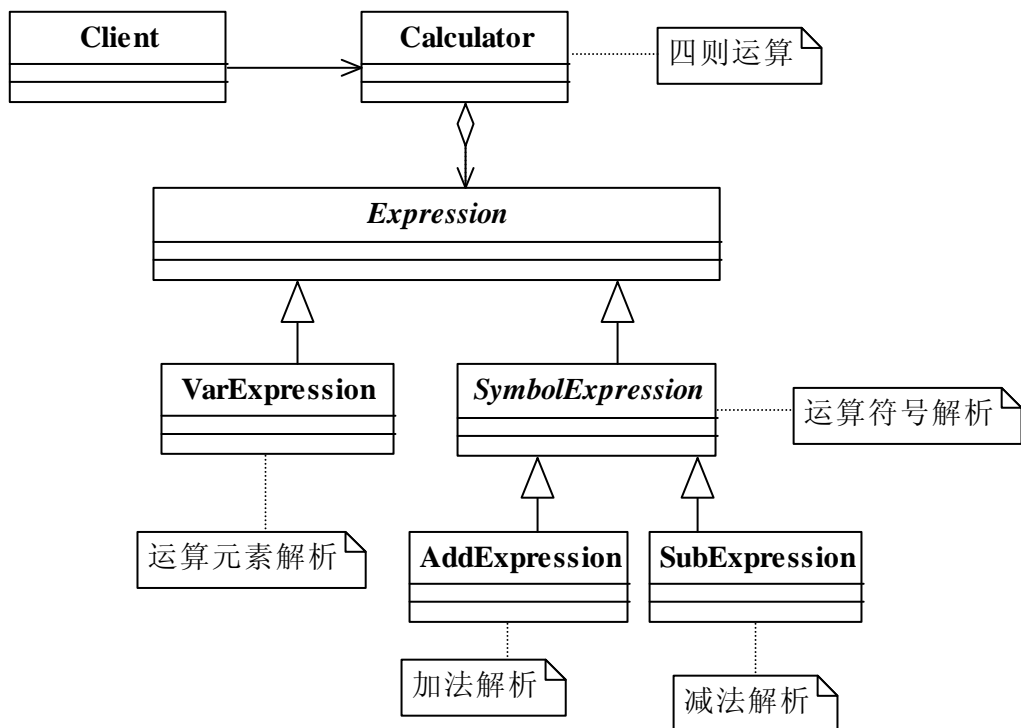
我们以实现加减算法的公式为例，讲解如果解析一个固定语法的逻辑。由于使用语法解析的场景比较少，而且一些商业公司（比如 SAS、SPSS 等统计分析软件）都支持类似的规则运算，大家发挥的空间越来越小了，我就带领大家一步一步的实现这个过程。

我们来想，公式中有什么？就有两类元素：运算元素和运算符号，运算元素就是指 a 、 b 、 c 等符号，需要在具体赋值的对象，也叫做终结符号，为什么叫终结符号呢？因为这些元素除了需要赋值外，不需要做任何处理，所有运算元素都对应一个具体的业务参数，这是语法中最小的单元逻辑，不可再拆分；运算符号就是加减符号，是需要我们编写算法进行处理，每个运算符号都要对应处理单元，否则公式无法运行，运算符号也叫做非终结符号。两个元素的共同点都是要被解析的，不同点是所有的运算元素都一个具有相同的功能，可以用一个类表示，而运算符号，则是需要分别进行解释，加法需要解释，减法也需要解释。

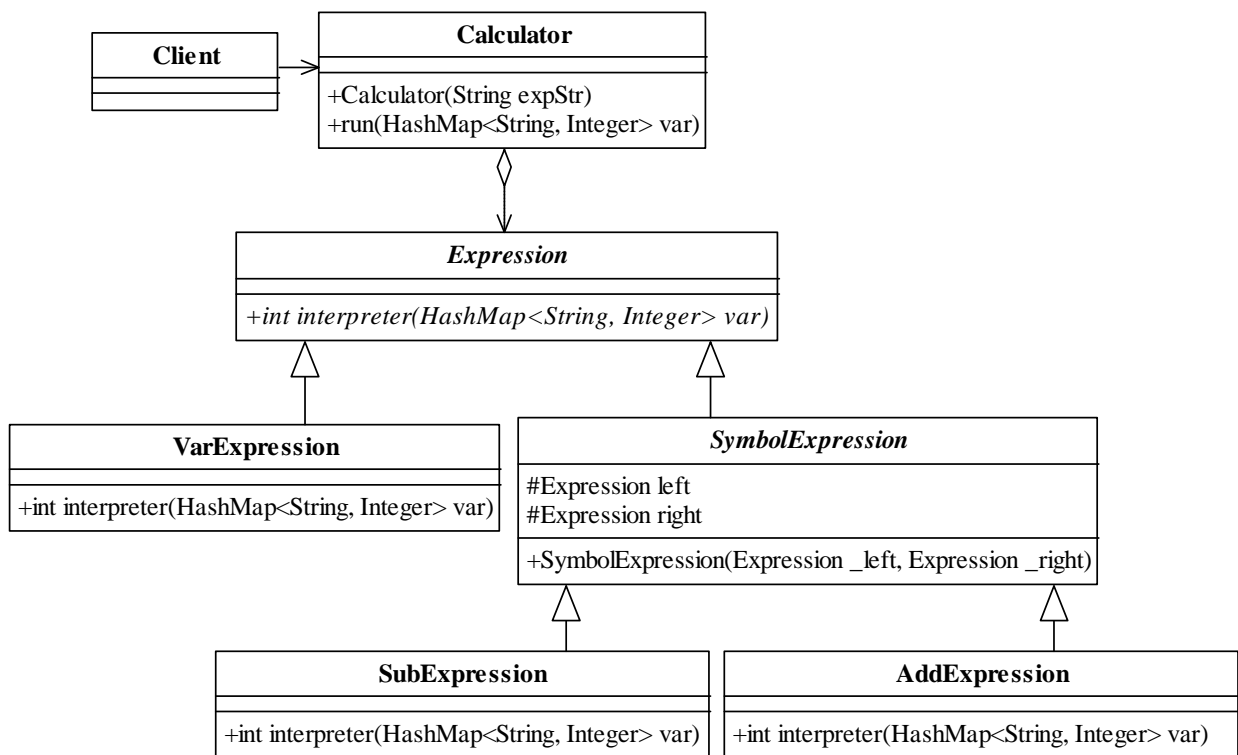
分析到这里，我们就可以先画一个简单的类图：



很简单的一个类图，VarExpression 用来解析运算元素，各个公式的运算元素的数量是不同的，每个运算元素对应了一个 VarExpression 对象，SymbolExpression 是负责运算符号解析的，分别有两个子类 AddExpression（负责加法运算）和 SubExpression（负责减法运算）来实现。解析的工作完成了，我们还需要把安排运行的先后顺序（加减法是不用考虑，但是乘除法呢？注意扩展性），并且还要返回结果，因此我们需要增加一个封装类来处理进行封装，由于我们只作运算，暂时还不与业务有挂钩，我们就定义为 Calculator 类，分析到这里，思路就比较清晰了，我们再来看类图：



Calculator 的作用就是起到封装的作用，根据迪米特原则，Client 只和直接的朋友 Calculator 交流，与其他类没关系。整个类图的结构也比较清晰，于是我们开始填充类图方法。最终类图如下：



每个类实现的职责各不相同，我们先来看 Expression 抽象类：


```
public abstract class Expression {  
  
    //解析公式和数值,其中var中的key值是公式中的参数, value值是具体的数字  
    public abstract int interpreter(HashMap<String,Integer> var);  
}
```

抽象类非常简单, 就一个方法 `interpreter`, 负责对传递进来的参数和值进行解析和匹配, 其中输入参数为 `HashMap` 类型, `key` 值为模型中的参数, 如 `a`、`b`、`c` 等, `value` 为运算时取得的具体数字。变量的解析器如下:

```
public class VarExpression extends Expression {  
    private String key;  
  
    public VarExpression(String _key){  
        this.key = _key;  
    }  
  
    //从map中取之  
    public int interpreter(HashMap<String, Integer> var) {  
        return var.get(this.key);  
    }  
}
```

以下为运算符解析器:

```
public abstract class SymbolExpression extends Expression {  
    protected Expression left;  
    protected Expression right;  
  
    //所有的解析公式都应只关心自己左右两个表达式的结果  
    public SymbolExpression(Expression _left, Expression _right){  
        this.left = _left;  
        this.right = _right;  
    }  
}
```

这个解析过程还是比较有意思的，每个运算符号，都只和自己左右两个数字有关系，但左右两个数字有可能也是一个解析的结果，无论如何，都是 Expression 的实现类，于是在对所有的运算符解析的子类中就增加了一个构造函数，传递左右两个表达式。加减法的解析器如下：

```
public class AddExpression extends SymbolExpression {

    public AddExpression(Expression _left, Expression _right){
        super(_left, _right);
    }

    //把左右两个表达式运算的结果加起来
    public int interpreter(HashMap<String, Integer> var) {
        return super.left.interpreter(var) + super.right.interpreter(var);
    }

}

public class SubExpression extends SymbolExpression {

    public SubExpression(Expression _left, Expression _right){
        super(_left, _right);
    }

    //左右两个表达式相减
    public int interpreter(HashMap<String, Integer> var) {
        return super.left.interpreter(var) - super.right.interpreter(var);
    }

}
```

解析的工作已经完成了，但是还有很大一段代码没有实现，仅仅做这样的工作还是不够的，我们还需要封装，Calculator 类的代码如下：

```
public class Calculator {
    //定义的表达式
    private Expression expression;

    //构造函数传参,并解析
    public Calculator(String expStr){
        //定义一个堆栈，安排运算的先后顺序
    }
}
```

```

Stack<Expression> stack = new Stack<Expression>();

//表达式拆分为字符数组
char[] charArray = expStr.toCharArray();

//运算
Expression left = null;
Expression right = null;
for(int i=0;i<charArray.length;i++){

    switch(charArray[i]) {
        case '+': //加法
            //加法结果放到堆栈中
            left = stack.pop();
            right = new VarExpression(String.valueOf(charArray[++i]));
            stack.push(new AddExpression(left,right));
            break;
        case '-':
            left = stack.pop();
            right = new VarExpression(String.valueOf(charArray[++i]));
            stack.push(new SubExpression(left,right));
            break;
        default: //公式中的变量
            stack.push(new VarExpression(String.valueOf(charArray[i])));
    }
}

//把运算结果抛出来
this.expression = stack.pop();
}

//开始运算
public int run(HashMap<String,Integer> var){
    return this.expression.interpreter(var);
}
}

```

方法比较长，我们来分析一下，Calculator 构造函数接受一个表达式，然后把表达式转化为 char 数组，并判断运算符号，如果是“+”则进行加法运算，把左边的数（left 变量）和右边的数（right 变量）加起来就可以了，那左边的数为什么是在堆栈中呢？例如这样的公式：a+b-c，根据 for 循环，首先被压入堆栈中的应该是有 a 元素生成的 VarExpression 对象，然后判断到加号时，把 a 元素的对象 VarExpression 从堆栈中弹出，与右边的数组 b 进行相加，b 又是怎么得来的呢？当前的数组游标下移一个单元格即可，同

时为了防止该元素再次被遍历，则通过++i 的方式跳过下一个遍历。——于是一个加法的运行结束，减法也是相同的运行原理。

为了满足业务要求，我们设置了一个 Client 类来模拟用户情况，用户要求可以扩展，可以修改公式，那就通过接受键盘事件来处理，Client 类的源代码如下：

```
public class Client {

    //运行四则运算
    public static void main(String[] args) throws IOException{
        String expStr = getExpStr();
        //赋值
        HashMap<String,Integer> var = getValue(expStr);

        Calculator cal = new Calculator(expStr);
        System.out.println("运算结果为: "+expStr +"="+cal.run(var));
    }

    //获得表达式
    public static String getExpStr() throws IOException{
        System.out.print("请输入表达式: ");
        return (new BufferedReader(new
InputStreamReader(System.in))).readLine();
    }

    //获得值映射
    public static HashMap<String,Integer> getValue(String exprStr) throws
IOException{
        HashMap<String,Integer> map = new HashMap<String,Integer>();

        //解析有几个参数要传递
        for(char ch:exprStr.toCharArray()){
            if(ch != '+' && ch != '-'){
                if(!map.containsKey(String.valueOf(ch))){ //解决重复参数的问题
                    System.out.print("请输入"+ch+"的值:");
                    String in = (new BufferedReader(new
InputStreamReader(System.in))).readLine();
                    map.put(String.valueOf(ch),Integer.valueOf(in));
                }
            }
        }

        return map;
    }
}
```

```
}  
}
```

其中 `getExpStr` 是从键盘事件中获得表达式，`getValue` 方法是从键盘事件中获得表达式中的元素映射值。运行过程如下：

首先，要求输入公式：

请输入表达式： `a+b-c`

其次，要求输入公式中的参数：

请输入a的值：`100`

请输入b的值：`20`

请输入c的值：`40`

最后运行出结果：

运算结果为：`a+b-c=80`

看，要求我们输入一个公式，然后输入参数，运行结果出来了！那我们是不是可以修改公式？当然可以了，我们只要输入公式，然后输入相应的值就可以了，公式是在运行期定义的，而不是在运行前就制定好的，是不是类似、初中学习的“代数”这门课？先公式，然后赋值，运算出结果。

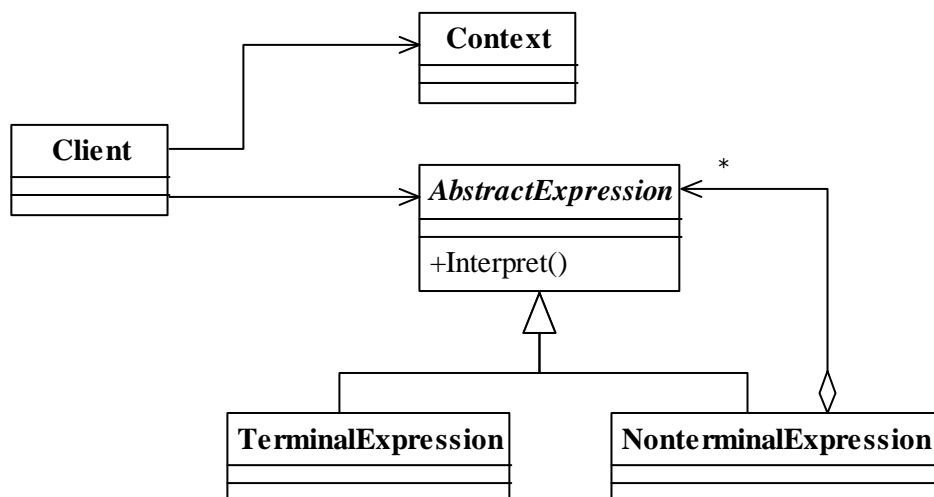
需求已经开发完毕，公式可以自由定义，只要符合规则（有变量有运算符符合）就可以运算出结果；若需要扩展也非常容易，只要扩展增加 `SymbolExpression` 的子类就可以了。这就是解释器模式。

解释器模式的定义

解释器模式（Interpreter Pattern）是一种按照规定语法进行解析的模式，在现在项目中使用较少（谁没事干会去写一个 PHP 或者 RUBY 的解析器），其定义如下：

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language。给定一个语言，定义它的文法的一种表示，并定义一个解释器，该解释器使用该表示来解释语言中的句子。

解释器模式的通用类图如下：



AbstractExpression: 抽象解释器，具体的解释任务由各个实现类完成，具体的解释器分为两大类：

TerminalExpression: 终结符表达式，实现与文法中的元素相关联的解释操作，通常一个解释器模式中只有一个终结符表达式，但有多实例，对应不同的终结符。具体到我们例子就是 VarExpression 类，表达式中的每个终结符都在堆栈中产生了一个 VarExpression 对象。

NonterminalExpression: 非终结符表达式，文法中的每条规则对应一个非终结表达式，具体到我们的例子就是加减法规则分别对应到 AddExpression 和 SubExpression 两个类。非终结符表达式根据逻辑的复杂程度而增加，原则上每个文法规则都对应一个非终结符表达式。

Context: 环境角色，具体到我们的例子中是采用 HashMap 代替。

解释器是一个比较少用的模式，以下为其通用源码，可以做为参考。抽象表达式通常只有一个方法，如下：

```
public abstract class Expression {

    //每个表达式必须有一个解析任务
    public abstract Object interpreter(Context ctx);
}
```

抽象表达式是生成语法集合（也叫做语法树）的关键，每个语法集合完成指定语法解析任务，它是通过迭代调用的方式，最终由最小的语法单元进行解析完成。终结符表达式如下：

```
public class TerminalExpression extends Expression {

    //通常终结符表达式只有一个，但是有多实例
    public Object interpreter(Context ctx) {
```

```

        return null;
    }
}

```

通常终结符表达式比较简单，主要是处理场景元素和数据的转换。非终结表达式如下：

```

public class NonterminalExpression extends Expression {

    //每个非终结符表达式都会对其他表达式产生依赖
    public NonterminalExpression(Expression... expression){

    }

    public Object interpreter(Context ctx) {
        //进行文法处理
        return null;
    }

}

```

每个非终结符表达式都代表了一个文法规则，并且每个文法规则都只关心自己周边的文法规则的结果，注意是结果，因此这就产生了每个非终结符调用自己周边的非终结符，然后最终最小的文法规则就是终结符表达式，终结符表达式的概念就是如此，不能够再参与比自己更小的文法运算了。我们再来看 Client 类：

```

public class Client {

    public static void main(String[] args) {
        Context ctx = new Context();
        //通常定一个语法容器，容纳一个具体的表达式，通常为ListArray,LinkedList,Stack等类型
        Stack<Expression> stack = null;

        for(;;){
            //进行语法判断，并产生递归调用
        }

        //产生一个完整的语法树，由各各个具体的语法分析进行解析
        Expression exp = stack.pop();

        //具体元素进入场景
    }
}

```

```
        exp.interpreter(ctx);  
    }  
}
```

通常 Client 是一个封装类，封装的结果就是传递进来一个规范语法文件，解析器分析后产生结果并返回，避免了调用者与语法解析器的耦合关系。

解释器模式的应用

解释器模式的优点

解释器是一个简单语法分析工具，它最显著的优点就是扩展性，修改语法规则只要修改相应的非终结符表达式就可以了，若扩展语法，则只要增加非终结符类就可以了。

解释器模式的缺点

首先，解释器模式会引起类膨胀。每个语法都要产生一个非终结符表达式，语法规则比较复杂时，就可能产生大量的类文件，为维护带来了非常多的麻烦。

其次，解释器模式采用递归调用方法。每个非终结符表达式只关心与自己有关的表达式，每个表达式需要知道最终的结果，必须一层一层的剥茧，无论是面向过程的语言还是面向对象的语言，递归都是在必要条件下使用的，它导致调试非常复杂，想想看，如果我们要排查一个语法错误，我们是不是要一个一个断点的调试下去直到最小的语法单元。

最后，效率问题，解释器模式由于使用了大量的循环和递归，效率是个不容忽视的问题，特别是用于解析复杂、冗长的语法时，效率是难以忍受的。

解释器模式使用的场景

一些重复发生的问题可以使用解释器模式。例如，多个应用服务器，每天产生大量的日志，需要对日志文件进行分析处理，由于各个服务器的日志格式不同，但是数据元素都是相同的，按照解释器的说法就是终结符表达式都是相同的，但是非终结符表达式就需要制定了，在这种情况下，可以通过程序来一劳永逸的解决该问题。

一个简单语法需要解释的场景。为什么是简单？看看非终结表达式，文法规则越多，复杂度越高，而且类间还要进行递归调用（看看我们例子中的堆栈），不是一般的复杂，想想看多个类之间的调用你需要什么样的耐心和信心去排查问题。因此，解释器模式一般用来解析比较标准的字符，例如 SQL 语法分析，不过该部分逐渐被专用工具所取代。

在某些特用的商业环境下也会采用解释器模式，我们刚刚的例子就是一个商业环境，而且现在模型运

算的例子非常多，原因就是目前很多商业机构已经能够提供出大量的数据进行分析了。

解释器模式的注意事项

尽量不要在项目中使用解释器模式（那你还讲这么多！肃静肃静，学无止境，毕竟它也是一种设计模式），除非必要，那用什么来代替呢？可以使用 shell、JRuby、Groovy 等脚本语言来代替，完全可以满足一些商业的分析过程，我们在一个银行的分析型项目中就采用 JRuby 进行运算处理，代替了使用解释器模式的四则运算。

第 23 章 亨元模式【Flyweight Pattern】

下午，正在开会中，老大推门进来。

“三儿，出来一下。”

我刚出会议室门口，老大就发话了。

“郎当（姓朗，顺口就叫郎当）的那个报考系统又 crash 了一台机器，两天已经宕了 4 次了，你这边还有紧急的事情没有？……没有，那赶快过去顶一下，就运行三天的程序，两天 TMD 宕了 4 次，还怎么玩？！”

我马上收拾东西，无怪钱包、手机、笔记本，冲到马路上拦了出租车，同时打电话给郎当。

“三哥，厂商人员已经定位出了，OutOfMemory 内存溢出，没查到有 memory leak 的情况，现在还在跟踪，……是突然暴涨的，都是在繁忙期出现问题的……”

内存溢出对 Java 应用实在是太平常了，就两种可能：无意识的代码缺陷，导致内存泄露，JVM 不能获得连续的内存空间，还有就是代码写的很烂，产生的对象太多，内存被耗尽。现在的情况是没有内存泄露，那只有一种原因，代码太烂把内存耗尽。

到现场后，郎当给我介绍介绍了一下系统情况。该系统是一个报考系统，其中有一个模块是负责社会人员报名，该模块对全国的考试人员只开放三天，并且限制报考人员数量。第一天 9:00 开始报考，系统慢的像蜗牛，基本上都不能访问，后来设置了 HTTP Server 的并发数量，稍有缓解，9:30 宕了一台机器，10 分钟后，又挂了一台，下午 3:40 又挂了一台，看样子晚上要让郎当去寺庙烧烧香了。

该系统一共有 8 台应用服务器，基本上 CPU 繁忙程度都在 60% 以上，HTTP 的最大并发是 2000，平均分配到每台应用服务器上没有太大的压力呀，怀疑是代码问题，然后详细了解一下业务和数据流逻辑，基本的业务操作过程清楚了，先登录（没有账号的，则要先注册），登录后，需要填写以下信息：

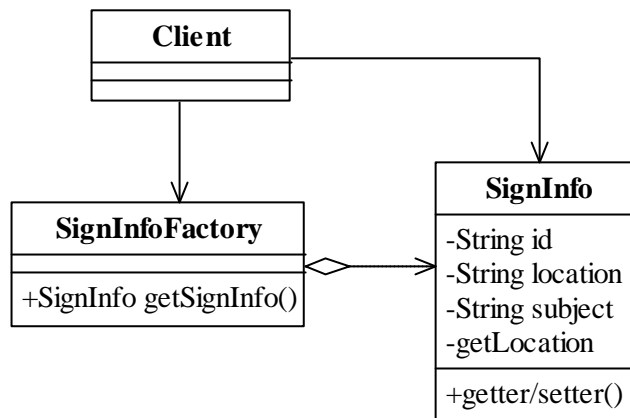
考试科目，选择框；

考试地点，选择框，根据科目不同，列表不同；

准考证邮寄地址，输入框

还有其他一堆信息，我们以这三者作为代表来讲解。信息填写完毕后，点击确认，报名就结束了，太简单的程序了，业务逻辑也确实是这样，为什么出现 Crash 情况呢？那肯定是和压力有关系！

我们先把这个过程的静态类图画出来，类图如下：



很简单的工厂方法模式，表现层通过工厂方法模式创建对象，然后传递给业务层和持久层，最终保存到数据库中，为什么要使用工厂方法模式而不用直接 new 一个对象呢？因为是在框架下编程，必须有一个对象工厂 (ObjectFactory, Spring 也有对象工厂，只是你没有注意而已)。我们先来看报考信息，代码如下：

```
public class SignInfo {

    //报名人员的ID
    private String id;
    //考试地点
    private String location;
    //考试科目
    private String subject;
    //邮寄地址
    private String postAddress;

    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getLocation() {
        return location;
    }
    public void setLocation(String location) {
        this.location = location;
    }
    public String getSubject() {
        return subject;
    }
}
```

```
    }  
    public void setSubject(String subject) {  
        this.subject = subject;  
    }  
    public String getPostAddress() {  
        return postAddress;  
    }  
    public void setPostAddress(String postAddress) {  
        this.postAddress = postAddress;  
    }  
}
```

很简单的一个 POJO 对象 (Plain Ordinary Java Objects, 简单 Java 对象)。我们再来看工厂类, 代码如下:

```
public class SignInfoFactory {  
    //报名信息的对象工厂  
    public static SignInfo getSignInfo(){  
        return new SignInfo();  
    }  
}
```

工厂类就这么简单? 非也, 这是我们的教学代码, 真实的 ObjectFactory 复杂的多, 主要是注入了部分 Handler 的管理。表现层是如何创建对象的, 如下:

```
public class Client {  
  
    public static void main(String[] args) {  
        //从工厂中获得一个对象  
        SignInfo signInfo = SignInfoFactory.getSignInfo();  
        //进行其他业务处理  
    }  
  
}
```

就这么简单, 但是简单为什么会出现问题呢? 而且这样写也没有问题呀, 很标准的工厂方法模式, 应

该不会有大问题,然后又看了看系统厂商提供的分析报告,报告中指出:内存是突然有800M直接飙升到1.4G,新的对象申请不到内存空间,于是乎出现 OutOfMemory,同时报告中还列出宕机时刻内存中的对象,其中 SignInfo 类的对象就有 400M,疯子,绝对是疯子!报告都没有看嘛!

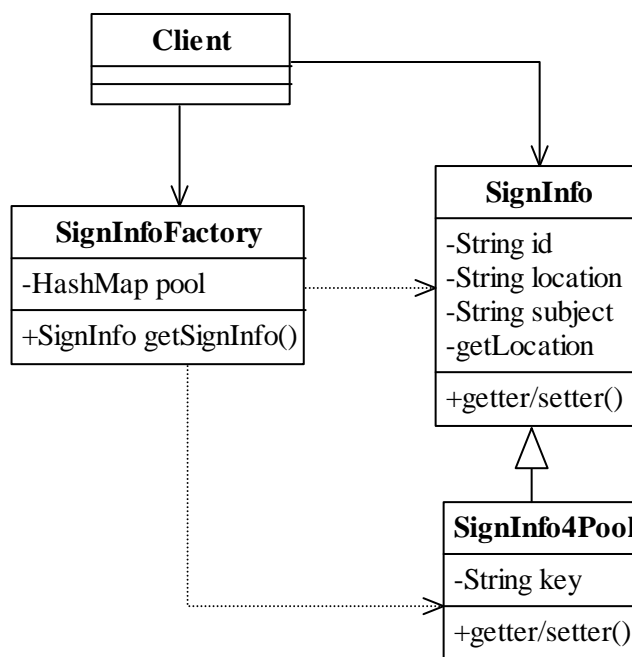
问题找到了,我拉郎当过来谈话,“厂商不是分析出原因了嘛,人家已经指出 SignInfo 类的对象占用了 400 多 M 的内存,这是怎么回事?”

“三哥,这是很正常的,这么大的访问量,产生出这么多的 SignInfo 对象也是应该的,内存中有这么多对象并不表示这些对象正在被使用呀,估计很大一部分还没有被回收而已,垃圾回收器什么时候回收内存中的对象这是不确定的。你看,并发 200 多个,这可是并发数量……”

我想了想,也确实是怎么回事。既然已经定位是内存中对象太多,那就应该想到使用缓存技术,更具体的说就是对象池(Object Pool)技术,使用共享的方法减少对象数量。

池的实现有很多开源工具,比如 apache 的 commons-pool 就是一个非常不错的池工具,我们暂时还用不到这么重量级的工具,我们自己来设计一个对象池。首先,我们要定义一个池容器,在这个容器中容纳哪些对象;其次,我们有提供一个接口供客户端访问,池中有可用对象时,可以直接从池中获得,否则建立一个新的对象,并放置到池中。

设计思路是有了,那我们池中对象的标准是什么呢?你想想看,如果你把所有的对象都放到池中,那还有什么意义?内存早就给你撑爆了!这么多对象,必然有一些相同的属性值,比如几十万 SignInfo 对象中,考试科目就 4 个,考试地点也就是 30 多个,其他的属性则是每个对象都不相同的,我们把对象的相同属性提取出来,不同的属性在系统内进行赋值处理,是不是就可以建立一个池了?话不需多说,我们以类图来表示:



很小的改动，增加了一个子类，实现带缓冲池的对象建立，同时在工厂类上增加了一个容器对象 HashMap，保存池中的所有对象。我们先来看产品子类：

```
public class SignInfo4Pool extends SignInfo {

    //定义一个对象池提取的KEY值
    private String key;

    //构造函数获得相同标志
    public SignInfo4Pool(String _key){
        this.key = _key;
    }

    public String getKey() {
        return key;
    }

    public void setKey(String key) {
        this.key = key;
    }

}
```

很简单，就是增加了一个 key 值，为什么要增加 key 值？为什么要使用子类，而不在 SignInfo 类上作修改？好，我来给你解释为什么要这样做，我们刚刚已经分析了所有的 SignInfo 对象都有一个些共同的属性：考试科目和考试地点，我们把这些共性提取出来作为所有对象的外部状态，在这个对象池中一个具体的外部状态只有一个对象，按照这个标注，我们定义的 key 值的标准为：考试科目+考试地点的复合字符串作为唯一的池对象标准，也就是说在对象池中，key 值唯一对象就唯一。

你可能马上就要提出了，为什么不建立一个新的类，包含 Subject 和 Location 两个属性作为外部状态呢？恩，是个办法，但是不是最好办法，有两个原因：一是修改的工作量太大，增加的这个类由谁来创建呢？同时，SignInfo 类是否也要修改呢？你不可能两段相同的 POJO 程序同时出现在同一模块中吧；二是性能问题，我们会在扩展模块中讲解。

说了这么多，我们还是继续来看程序，工厂类的代码如下：

```
public class SignInfoFactory {
    //池容器
    private static HashMap<String,SignInfo> pool = new
    HashMap<String,SignInfo>();

    //报名信息对象工厂
    @Deprecated
    public static SignInfo getSignInfo(){
        return new SignInfo();
    }

    //从池中获取对象
    public static SignInfo getSignInfo(String key){
        //设置返回对象
        SignInfo result = null;
        //池中不存在该对象，则建立，并放入池中
        if(!pool.containsKey(key)){
            System.out.println(key + "----建立对象，并放置到池中");
            result = new SignInfo4Pool(key);
            pool.put(key, result);
        }else{
            result = pool.get(key);
            System.out.println(key + "----直接从池中取得");
        }
        return result;
    }
}
```

```
}
```

方法都很简单，不多解释。读者注意一点的是 Deprecated 注解，不要有删除投产中代码的念头，如果方法或类确实不再使用了，增加该注解，表示已该方法或类经过时了，尽量不要再使用了，我们应该保持历史原貌，同时也有助于版本向下兼容，特别是在产品级研发中。

我们再来看看客户端是如何调用的，代码如下：

```
public class Client {

    public static void main(String[] args) {
        //初始化对象池
        for(int i=0;i<4;i++){
            String subject = "科目" + i;
            //初始化地址
            for(int j=0;j<30;j++){
                String key = subject + "考试地点"+j;
                SignInfoFactory.getSignInfo(key);
            }
        }
        SignInfo signInfo = SignInfoFactory.getSignInfo("科目1考试地点1");
    }

}
```

运行结果如下所示：

```
科目3考试地点25----建立对象，并放置到池中
科目3考试地点26----建立对象，并放置到池中
科目3考试地点27----建立对象，并放置到池中
科目3考试地点28----建立对象，并放置到池中
科目3考试地点29----建立对象，并放置到池中
科目1考试地点1---直接从直池中取得
```

前面还有很多对象创建提示语句，不再拷贝。通过这样的改造后，我们想想内存有多少个 SignInfo 对象？是的，最多 120 个对象，相比之前十几万个 SignInfo 对象优化了非常多。细心的读者可能注意到了

SignIn4Pool 类基本上没有跑出我们的视线范围, 仅仅在工厂方法中使用到了, 尽量缩小变更引起的风险, 想想看我们的改动是不是很小, 只要在展示层中拼一个字符传, 然后传递到工厂方法中就可以了。

通过这样的改造后, 第三天系统运行的非常稳定, CPU 占用率也下降了, 而且以后再也没有出现类似问题, 这就是享元模式的功劳。

享元模式的扩展

线程安全的问题

线程安全是一个老生常谈的话题, 只要使用 Java 开发都会遇到这个问题, 我们只所以要在今天的享元模式中提该问题, 是因为该模式有太多的几率发生线程不安全, 为什么呢?

我们还以报考系统为例, 来说明这个问题, 大家有没有想过, 为什么要以考试科目+考试地点做为外部状态呢? 为什么不能以考试科目或者考试地点作为外部状态呢? 这样池中的对象会更少的呀, 可以的, 完全可以的, 我们把程序以考试科目为外部状态, 享元工厂稍作修改, 代码如下:

```
public class SignInFactory {
    //池容器
    private static HashMap<String,SignIn> pool = new
    HashMap<String,SignIn>();

    //从池中获得对象
    public static SignIn getSignIn(String key){
        //设置返回对象
        SignIn result = null;
        //池中沒有该对象, 则建立, 并放入池中
        if(!pool.containsKey(key)){
            result = new SignIn();
            pool.put(key, result);
        }else{
            result = pool.get(key);
        }

        return result;
    }
}
```

很小的改动, 就修改了黑色字体部分。为了展示多线程的情况, 我们写一个多线程的类, 如下:

```
public class MultiThread extends Thread {
```

```
private SignInfo signInfo;
public MultiThread(SignInfo _signInfo){
    this.signInfo = _signInfo;
}

public void run(){
    if(!signInfo.getId().equals(signInfo.getLocation())){
        System.out.println("编号: "+signInfo.getId());
        System.out.println("考试地址: "+signInfo.getLocation());
        System.out.println("线程不安全了!");
    }
}
}
```

在 run 方法中判断特殊值，检查是否是线程安全，我们来看看场景类：

```
public class Client {

    public static void main(String[] args) {
        //在对象池中初始化四个对象
        SignInfoFactory.getSignInfo("科目1");
        SignInfoFactory.getSignInfo("科目2");
        SignInfoFactory.getSignInfo("科目3");
        SignInfoFactory.getSignInfo("科目4");

        //取得对象
        SignInfo signInfo = SignInfoFactory.getSignInfo("科目2");
        while(true){
            signInfo.setId("ZhangSan");
            signInfo.setLocation("ZhangSan");
            (new MultiThread(signInfo)).start();

            signInfo.setId("LiSi");
            signInfo.setLocation("LiSi");
            (new MultiThread(signInfo)).start();
        }
    }
}
```

```
}

```

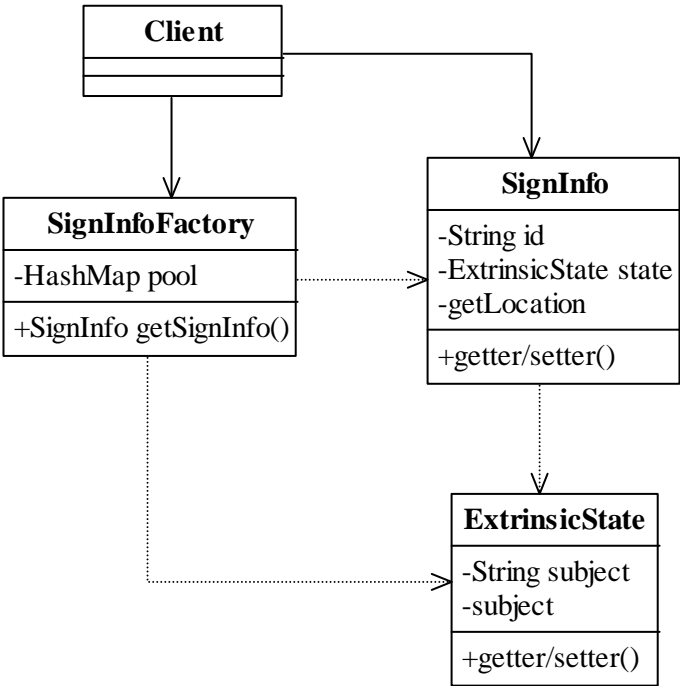
模拟实际的多线程情况，在对象池中我们保留四个对象，然后启动 N 多个线程来模拟，我们马上就看到如下的提示：

编号：LiSi
考试地址：ZhangSan
线程不安全了！

看看，线程不安全了吧，这是正常的，设置的享元对象数量太少，导致每个线程都到对象池中获得对象，然后都去修改其属性，于是就出现一些不和谐数据。只要使用 Java 开发，线程问题是不可避免的，那我们怎么去避免这个问题呢？享元模式是让我们使用共享技术，而 Java 的多线程又有如此问题，该如何设计呢？没什么可以参考的标准，只有依靠经验，在需要的地方考虑一下线程安全，在大部分的场景下不用考虑。我们在使用享元模式时，对象池中的享元对象尽量多的多，多到足够满足业务为止。

性能平衡

尽量使用 Java 基本类型作为外部状态。我们在报考系统中，不考虑系统的修改风险，完全可以重新新建一个类作为外部状态，因为这才完全符合面向对象编程的理念，好，我们实现处理，先看类图：



类作为外部状态

我们首先来看 ExtrinsicState 外部状态类，代码如下：

```
public class ExtrinsicState {
    //考试科目
    private String subject;
    //考试地点
    private String location;

    public String getSubject() {
        return subject;
    }
    public void setSubject(String subject) {
        this.subject = subject;
    }
    public String getLocation() {
        return location;
    }
    public void setLocation(String location) {
        this.location = location;
    }

    @Override
    public boolean equals(Object obj){
        if(obj instanceof ExtrinsicState){
            ExtrinsicState state = (ExtrinsicState)obj;
            return state.getLocation().equals(location) &&
state.getSubject().equals(subject);
        }
        return false;
    }

    @Override
    public int hashCode(){
        return subject.hashCode() + location.hashCode();
    }
}
```

注意注意，一定要覆写 equals 和 hashCode 方法，否则它作为 HashMap 中的 key 值是根本没有意义的，只有 hashCode 值相等，并且 equals 返回结果为 true，两个对象才相等，也只有在这种情况下才有可能从

对象池中查找获得对象。

SignInInfo 的修改较小，仅在 SignInInfo 中引入该 ExtrinsicState 外部状态对象，不再赘述。我们在来看享元工厂，它是以该 ExtrinsicState 类作为外部状态，代码如下：

```
public class SignInInfoFactory {
    //池容器
    private static HashMap<ExtrinsicState,SignInInfo> pool = new
    HashMap<ExtrinsicState,SignInInfo>();

    //从池中获得对象
    public static SignInInfo getSignInInfo(ExtrinsicState key){
        //设置返回对象
        SignInInfo result = null;
        //池中如果没有该对象，则建立，并放入池中
        if(!pool.containsKey(key)){
            result = new SignInInfo();
            pool.put(key, result);
        }else{
            result = pool.get(key);
        }
        return result;
    }
}
```

重点是看看我们的场景类，我们来测试一下性能差异，代码如下：

```
public class Client {

    public static void main(String[] args) {
        //初始化对象池
        ExtrinsicState statel = new ExtrinsicState();
        statel.setSubject("科目1");
        statel.setLocation("上海");
        SignInInfoFactory.getSignInInfo(statel);

        ExtrinsicState state2 = new ExtrinsicState();
        state2.setSubject("科目1");
        state2.setLocation("上海");

        //计算执行100万次需要的时间
        long currentTime = System.currentTimeMillis();
```

```
        for(int i=0;i<10000000;i++){
            SignInfoFactory.getSignInfo(state2);
        }
        long tailTime = System.currentTimeMillis();

        System.out.println("执行时间: "+(tailTime - currentTime) + " ms");
    }
}
```

运行结果为:

执行时间: 172 ms

同样我们看看以 String 类型作为外部状态的运行情况, 代码如下:

```
public class Client {

    public static void main(String[] args) {
        String key1 = "科目1上海";
        String key2 = "科目1上海";
        //初始化对象池
        SignInfoFactory.getSignInfo(key1);

        //计算执行10万次需要的时间
        long currentTime = System.currentTimeMillis();
        for(int i=0;i<10000000;i++){
            SignInfoFactory.getSignInfo(key2);
        }
        long tailTime = System.currentTimeMillis();

        System.out.println("执行时间: "+(tailTime - currentTime) + " ms");
    }
}
```

运行结果如下:

执行时间：78 ms

看到没？一半的效率，这还是非常简单的享元对象，看看我们重写的 equals 方法和 hashCode 方法，这段代码是必须实现的，如果比较复杂的话，这个时间差异会更大。

各位，想想看，使用自己编写的类作为外部状态，必须覆写 equals 方法和 hashCode 方法，而且执行效率还比较低，这种吃力不讨好的事情最好别做，外部状态最好以 Java 的基本类型作为标志，比如 String、int 等，可以大幅的提升效率。

最佳实践

Flyweight 是拳击比赛中的特用名词，意思是“特轻量级”，指的是 51 公斤级比赛，用到设计模式中是指我们的类要轻量级，粒度要小这才是它要表到的意思，粒度小了，带来的问题就是对象太多，那就是用共享技术来解决。

享元模式在 Java API 中也是随处可见，比如以下程序：

```
public class Test {  
  
    public static void main(String[] args) {  
        String str1 = "和谐";  
        String str2 = "社会";  
        String str3 = "和谐社会";  
        String str4;  
  
        str4 = str1 + str2;  
        System.out.println(str3 == str4);  
  
        str4 = (str1 + str2).intern();  
        System.out.println(str3 == str4);  
    }  
}
```

看看 String 对象的 intern 方法是什么意思吧，如果是 String 的对象池中有该类型的值，则直接返回对象池中的对象，那当然相等了，享元模式呀。

面向对象开发有时在设计和性能方面存在相悖的情况，例如从设计的观点来看，最好是将所有相关的信息和行为都封装在一个对象内，这样就可以统一的处理这些对象，复用率较高，扩展也容易。不幸的是，大对象在性能优化方面需要付出非常昂贵代价，所有拆拆分分无穷尽也！

第 24 章 备忘录模式【Memento Pattern】

大家有没有看过尼古拉斯·凯奇主演的《Next》(中文译名为《预见未来》)? 尼古拉斯·凯奇饰演一个可以预视并且扭转未来的人, 其中有一个情节很是让人心动: 男女主角见面的那段情节, Cris Johnson (尼古拉斯·凯奇饰演) 坐咖啡吧台前, 看着离自己近在咫尺的 Callie Ferris (朱莉安·摩尔饰演), 计划着怎么认识这个命中注定的女人, 看 Cris Johnson 如何利用自己的特异功能:

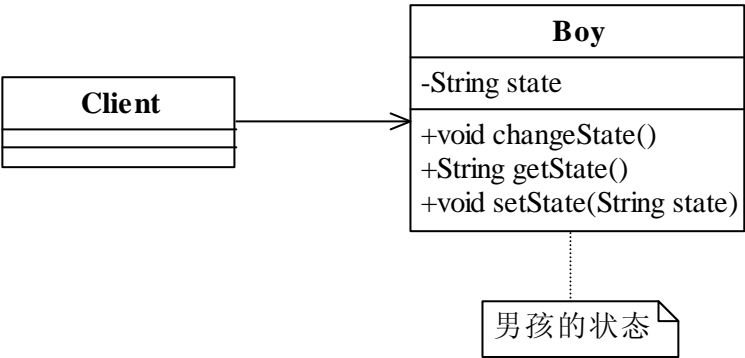
- 第一次, Cris Johnson 端着一杯咖啡走过去, 说“你好, 可以认识你吗?” 被拒绝, 恢复到坐在咖啡吧台前的状态;
- 第二次, 走过去询问是否可以搭车, 被拒绝, 恢复原状;
- 第三次, 帮助解决困境, 被拒绝, 恢复原状;
- 第四次, 采用嬉皮士的方式的解决困境, 被拒绝, 恢复原状;
- 第五次, 帮助解决困境, 被打伤, 装可怜, Callie Ferris 怜惜, 于是乎相识了。

看看这是一件多么幸福的事情, 追求一个女生可以多次反复的实验, 直到找到好的方法和途径为止, 这估计是大多数男生都希望获得特异功能。想想看, 看到一个心仪的女生, 我们反复的尝试, 终有一个方法打动她的, 多美好的一件事。继续想吧, 我们还得回到现实生活, 我们来分析一下类似事情的经过:

首先, 拷贝一个当前状态, 保留下来, 这个状态就是等会搭讪女孩子失败后要恢复的状态, 你不恢复原始状态, 这不就露馅儿了吗?

- 其次, 每次的试探性尝试失败后, 都必须恢复到这个原始状态;
- 最后, N 多次试探总有一次成功的吧, 成功以后即可走成功路线。

想想看, 我们这里的场景中最重要的是哪一块? 对的, 是原始状态的保留和恢复这块, 如何保留一个原始, 如何恢复一个原始状态这才是最重要的, 那想想看, 我们应该来怎么来实现呢? 很简单呀, 我们可以定义一个中间变量, 保留这个原始状态, 是的, 可以这样做, 我们先看看类图:



太简单的类图了，我们来解释一下上面的状态 state，在某一时刻的所有位置信息、心理信息、环境信息都属于状态，我们这里用了一个标示性的名词 state 代表这所有状态，比如在追女孩子前心情是期待、心理是焦躁不安等等。每一次去认识女孩子都是会发生状态变化的，我们使用 changeState 方法来代替，由于程序比较简单，就没有编写接口，我们来看实现：

```
public class Boy {

    //男孩的状态
    private String state = "";

    //认识女孩子后状态肯定改变，比如心情、手中的花朵等等
    public void changeState(){
        this.state = "心情可能很不好";
    }

    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
    }

}
```

程序是很简单，主要的业务逻辑是在场景类中，我们来看场景类是如何进行状态的保留、恢复的，看如下代码：

```
public class Client {

    public static void main(String[] args) {
        //声明出主角
        Boy boy = new Boy();
        //初始化当前状态
        boy.setState("心情很棒!");
        System.out.println("====男孩现在的状态====");
        System.out.println(boy.getState());
        //需要记录下当前状态呀
        Boy backup = new Boy();
    }
}
```

```

        backup.setState(boy.getState());
        //男孩去追女孩，状态改变
        boy.changeState();
        System.out.println("\n====男孩追女孩子后的状态====");
        System.out.println(boy.getState());
        //追女孩失败，恢复原状
        boy.setState(backup.getState());
        System.out.println("\n====男孩恢复后的状态====");
        System.out.println(boy.getState());
    }
}

```

运行结果如下：

```

====男孩现在的状态====
心情很棒！

```

```

====男孩追女孩子后的状态====
心情可能很不好

```

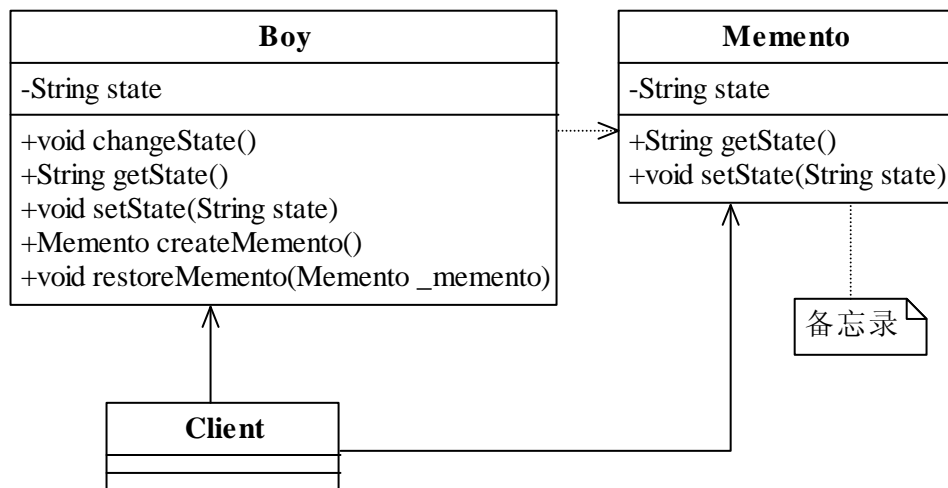
```

====男孩恢复后的状态====
心情很棒！

```

程序运行正确，输出结果也是我们期望的，但是，但是结果正确并不表示程序是最优的，我们来看看场景类 Client，它代表的是高层模块，或者说是非“近亲”模块的调用者，注意看 backup 变量的使用，它对于高层模块完全是多余的，为什么一个状态的保存和恢复要让高层模块来负责呢？这应该是 Boy 类的职责，不应该让高层模块来完成，也就是破坏了的 Boy 的封装，或者说 Boy 类没有封装好，它应该是把 backup 的定义容纳进来，而不应该让高层模块来定义。

问题我们已经知道了，就是 Boy 类封装不够，那我们怎么来修改呢？如果在 Boy 类中再增加一个方法或者其他的内部类来保存这个状态，则对单一职责原则是一种破坏，想想看单一职责原则是怎么说的？一个类的职责应该是单一的，Boy 类本身的职责是追求女孩子，而保留和恢复原始状态则应该由另外一个类来承担，那我们把这个类起个名字就叫做备忘录，这和你大家经常在桌面上贴的那个便签是一个概念，分析到这里我们已经思路已经非常清楚了，我们来修改一下类图：



改动很小，增加了一个新的类 Memento，负责状态的保存和备份；在 Boy 类中增加了 createMemento 创建一份备忘录和恢复一个备忘录 restoreMemento，我们先来看 Boy 类的变化：

```

public class Boy {

    //男孩的状态
    private String state = "";

    //认识女孩子后状态肯定改变，比如心情、手中的花朵等等
    public void changeState(){
        this.state = "心情可能很不好";
    }

    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
    }

    //保留一个备份
    public Memento createMemento(){
        return new Memento(this.state);
    }

    //恢复一个备份
    public void restoreMemento(Memento _memento){
        this.setState(_memento.getState());
    }
}
  
```

```
    }  
}
```

注意看，确实只增加了两个方法创建备份和恢复备份，至于在什么时候创建备份和恢复备份则是有高层模块决定的。我们再来看备忘录模块：

```
public class Memento {  
  
    //男孩的状态  
    private String state = "";  
  
    //通过构造函数传递状态信息  
    public Memento(String _state){  
        this.state = _state;  
    }  
  
    public String getState() {  
        return state;  
    }  
  
    public void setState(String state) {  
        this.state = state;  
    }  
  
}
```

就是一个简单 JavaBean，保留男孩当时的状态信息。我们再来看场景类，稍做修改，代码如下：

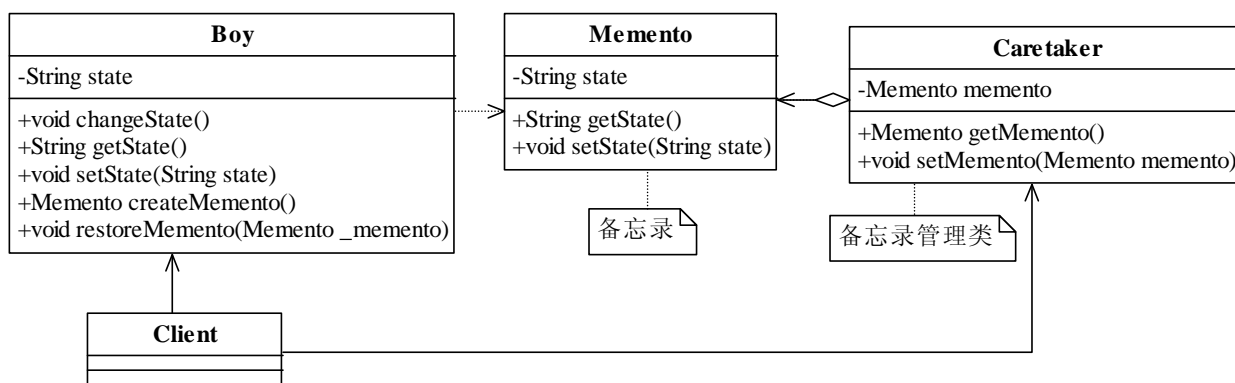
```
public class Client {  
  
    public static void main(String[] args) {  
        //声明出主角  
        Boy boy = new Boy();  
        //初始化当前状态  
        boy.setState("心情很棒！");  
        System.out.println("====男孩现在的状态====");  
        System.out.println(boy.getState());  
        //需要记录下当前状态呀  
        Memento mem = boy.createMemento();  
    }  
}
```

```

        //男孩去追女孩，状态改变
        boy.changeState();
        System.out.println("\n====男孩追女孩子后的状态====");
        System.out.println(boy.getState());
        //追女孩失败，恢复原状
        boy.restoreMemento(mem);
        System.out.println("\n====男孩恢复后的状态====");
        System.out.println(boy.getState());
    }
}

```

运行结果保持相同，虽然程序中不再重复定义 Boy 类的对象了，但是我们还是要关心备忘录，这对迪米特原则是一个亵渎，它告诉我们只和朋友类通讯，那这个备忘录对象是我们必须要通讯的朋友类吗？对高层模块来说，它最希望要做的就是创建一个备份点，然后在需要的时候再恢复到这个备份点就成了，它不用关心到底有没有备忘录这个类。那根据这一指示，我们就需要把备忘录类再包装一下，怎么包装呢？建立一个管理类，就是管理这个备忘录，类图如下：



又增加了一个 JavaBean，Boy 类和 Memento 没有任何改变，不再赘述。我们来看增加的备忘录管理类：

```

public class Caretaker {

    //备忘录对象
    private Memento memento;

    public Memento getMemento() {
        return memento;
    }

    public void setMemento(Memento memento) {

```

```

        this.memento = memento;
    }

}

```

太简单了这个，非常纯粹的一个 JavaBean，甭管它多简单只要有用就行，我们来看场景类如何调用：

```

public class Client {

    public static void main(String[] args) {
        //声明出主角
        Boy boy = new Boy();
        //声明出备忘录的管理者
        Caretaker caretaker = new Caretaker();
        //初始化当前状态
        boy.setState("心情很棒!");
        System.out.println("====男孩现在的状态====");
        System.out.println(boy.getState());
        //需要记录下当前状态呀
        caretaker.setMemento(boy.createMemento());
        //男孩去追女孩，状态改变
        boy.changeState();
        System.out.println("\n====男孩追女孩子后的状态====");
        System.out.println(boy.getState());
        //追女孩失败，恢复原状
        boy.restoreMemento(caretaker.getMemento());
        System.out.println("\n====男孩恢复后的状态====");
        System.out.println(boy.getState());
    }
}

```

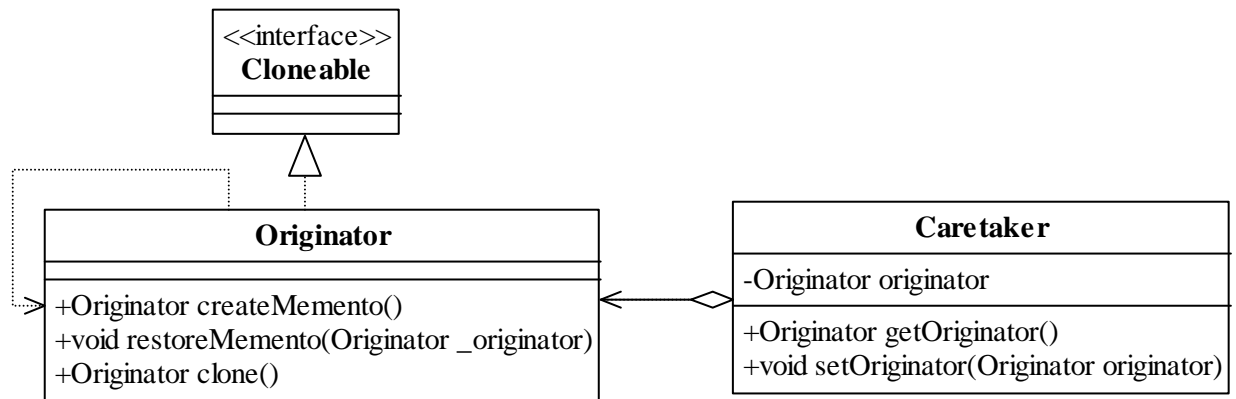
注意看黄色部分，就修改了这么多，看看程序的逻辑是不是清晰了很多，需要备份点的时候就创建一个备份，然后丢给备忘录管理者进行管理，需要的时候再从管理者手中拿到这个备份。这个备份者就类似与一个备份的仓库管理员，创建一个丢进去，需要的时候再拿出来。 这就是备忘录模式。

备忘录模式的扩展

Clone 方式的备忘录

大家还记得在 13 章中讲的原型模式吗？我们可以通过拷贝的方式产生一个对象的内部状态，很好的办

法，发起人角色只要实现 Cloneable 就成，比较简单，我们来看类图：



从类图上看，发起人角色融合了发起人角色、备忘录角色双重功效，我们来代码：

```
public class Originator implements Cloneable{

    //内部状态
    private String state = "";

    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
    }

    //创建一个备忘录
    public Originator createMemento(){
        return this.clone();
    }

    //恢复一个备忘录
    public void restoreMemento(Originator _originator){
        this.setState(_originator.getState());
    }

    //克隆当前对象
    @Override
    protected Originator clone(){

        try {
```



```
        return (Originator)super.clone();
    } catch (CloneNotSupportedException e) {
        e.printStackTrace();
    }
    return null;
}
}
```

增加了 clone 方法，产生一个备份对象，需要使用的时候再还原，我们再来看管理员角色：

```
public class Caretaker {

    //发起人对象
    private Originator originator;

    public Originator getOriginator() {
        return originator;
    }

    public void setOriginator(Originator originator) {
        this.originator = originator;
    }

}
```

没什么变化，还是一个简单的 JavaBean。我们来想想这种模式是不是还可以简化？要管理员角色干什么？就是为了管理备忘录角色，现在连备忘录角色都被合并了，留着你干啥？好，我们想办法把它也精简掉，看代码：

```
public class Originator implements Cloneable{
    private Originator backup;

    //内部状态
    private String state = "";

    public String getState() {
        return state;
    }

    public void setState(String state) {
```

```
        this.state = state;
    }

    //创建一个备忘录
    public void createMemento(){
        this.backup = this.clone();
    }

    //恢复一个备忘录
    public void restoreMemento(){
        //在进行恢复前应该进行断言，防止空指针
        this.setState(this.backup.getState());
    }

    //克隆当前对象
    @Override
    protected Originator clone(){

        try {
            return (Originator)super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

再看看 Client 是如何调用的，如下代码：

```
public class Client {

    public static void main(String[] args) {
        //定义发起人
        Originator originator = new Originator();
        //建立初始状态
        originator.setState("初始状态...");
        System.out.println("初始状态是: "+originator.getState());
        //建立备份
        originator.createMemento();
        //修改状态
        originator.setState("修改后的状态...");
        System.out.println("修改后状态是: "+originator.getState());
    }
}
```

```

        //恢复原有状态
        originator.restoreMemento();
        System.out.println("恢复后状态是: "+originator.getState());
    }
}

```

运行结果如下：

初始状态是：初始状态...

修改后状态是：修改后的状态...

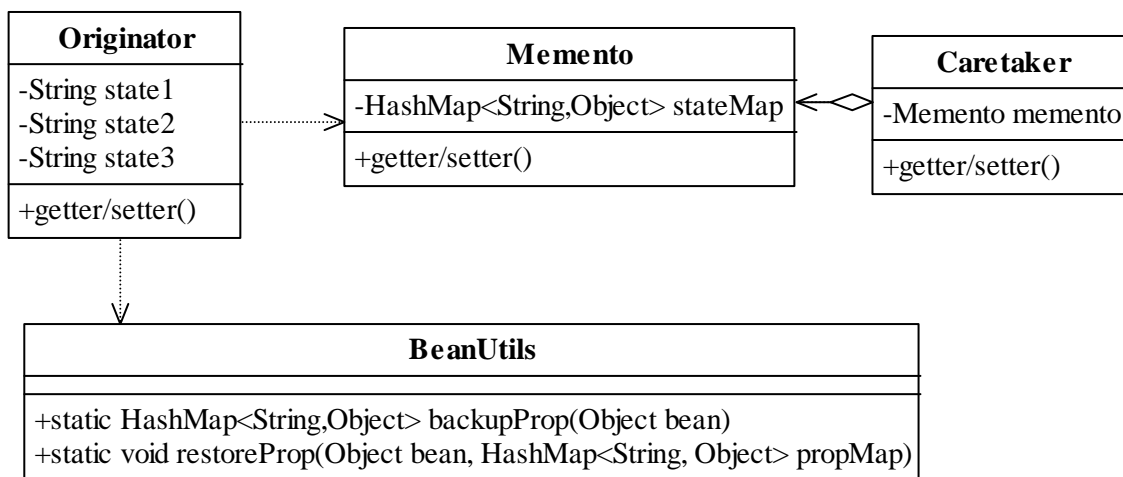
恢复后状态是：初始状态...

运行结果是我们希望的结果，正点呀，程序精简了非常多，而且高层模块的依赖也减少了，这正是我们期望的效果。缺点嘛，当然有了，想想我们原型模式深拷贝和浅拷贝的问题，在复杂的场景下它会让你的程序逻辑异常混乱，出现错误也很难跟踪。因此 Clone 方式的备忘录模式适用于较简单的场景。

多状态的备忘录模式

读者应该看到我们以上讲解都是单状态的情况，在实际的开发中一个对象不可能只有一个状态呀，这非常正常，一个 JavaBean 有 N 多个属性非常常见，这都是它的状态，如果照搬我们以上讲解的备忘录模式，是不是就要写一堆的状态备份、还原语句？这不是一个好办法，这种类似的非智力劳动越多，犯错误的几率越大，那我们有什么办法来处理多个状态的保存问题呢？

好的，我们来讲解一个对象全状态备份方案，它有多种处理方式，比如使用 Clone 的方式就可以解决，使用数据技术也可以解决（DTO 回写到临时表中）等等，我们就对备忘录模式继续扩展一下，实现一个 JavaBean 对象的所有状态的备份和还原，类图如下：



还是比较简单的类图，增加了一个 BeanUtils 类，其中 backupProp 是把发起人的所有属性值转换到 HashMap 中，方便备忘录角色存储；restoreProp 方法则是把 HashMap 中的值返回到发起人角色中。可能各位要说了，为什么要使用 HashMap，直接使用 Originator 对象的拷贝不是一个很好的方法吗？可以这样做，问题是你破坏了发起人的通用性，你在做恢复动作的时候需要对该对象继续多次的赋值操作，也容易产生错误。我们先来看发起人角色：

```
public class Originator {

    //内部状态
    private String statel = "";
    private String state2 = "";
    private String state3 = "";

    public String getState1() {
        return statel;
    }

    public void setState1(String statel) {
        this.statel = statel;
    }

    public String getState2() {
        return state2;
    }

    public void setState2(String state2) {
        this.state2 = state2;
    }

    public String getState3() {
        return state3;
    }

    public void setState3(String state3) {
        this.state3 = state3;
    }

    //创建一个备忘录
    public Memento createMemento(){
        return new Memento(BeanUtils.backupProp(this));
    }
}
```

```

//恢复一个备忘录
public void restoreMemento(Memento _memento){
    BeanUtils.restoreProp(this, _memento.getStateMap());
}

//增加一个toString方法
@Override
public String toString(){
    return "state1=" +state1+"\nstate2="+state2+"\nstate3="+state3;
}
}

```

覆写了 toString 方法是为了方便打印, 可以让展示的结果更清晰。我们再来看 BeanUtils 工具类, 如下代码:

```

public class BeanUtils {

    //把bean的所有属性及数值放入到HashMap中
    public static HashMap<String,Object> backupProp(Object bean){
        HashMap<String,Object> result = new HashMap<String,Object>();
        try {
            //获得Bean描述
            BeanInfo beanInfo = Introspector.getBeanInfo(bean.getClass());
            //获得属性描述
            PropertyDescriptor[] descriptors = beanInfo.getPropertyDescriptors();
            //遍历所有属性
            for(PropertyDescriptor des:descriptors){
                //属性名称
                String fieldName = des.getName();
                //读取属性的方法
                Method getter = des.getReadMethod();
                //读取属性值
                Object fieldValue = getter.invoke(bean, new Object[]{});
                if(!fieldName.equalsIgnoreCase("class")){
                    result.put(fieldName, fieldValue);
                }
            }
        } catch (Exception e) {
            //异常处理
        }
        return result;
    }
}

```

```

    }

    //把HashMap的值返回到bean中
    public static void restoreProp(Object bean, HashMap<String, Object> propMap) {
        try {
            //获得Bean描述
            BeanInfo beanInfo = Introspector.getBeanInfo(bean.getClass());
            //获得属性描述
            PropertyDescriptor[] descriptors = beanInfo.getPropertyDescriptors();
            //遍历所有属性
            for(PropertyDescriptor des:descriptors){
                //属性名称
                String fieldName = des.getName();
                //如果有这个属性
                if(propMap.containsKey(fieldName)){
                    //写属性的方法
                    Method setter = des.getWriteMethod();
                    setter.invoke(bean, new Object[]{propMap.get(fieldName)});
                }
            }
        } catch (Exception e) {
            //异常处理
            System.out.println("shit");
            e.printStackTrace();
        }
    }
}

```

该类大家在项目中会经常用到，可以作为参考使用。类似的功能有很多工具已经提供，比如 Spring、Apache 工具集 commons 等，大家也可以直接使用。我们再来看备忘录角色：

```

public class Memento {

    //接受HashMap作为状态
    private HashMap<String, Object> stateMap;

    //接受一个对象，建立一个备份
    public Memento(HashMap<String, Object> map){
        this.stateMap = map;
    }
}

```

```

    public HashMap<String,Object> getStateMap() {
        return stateMap;
    }

    public void setStateMap(HashMap<String,Object> stateMap) {
        this.stateMap = stateMap;
    }
}

```

备忘录发起人角色没有任何改变，不再赘述。我们再编写一个场景类，看看我们的成果是否正确，如下：

```

public class Client {

    public static void main(String[] args) {
        //定义出发起人
        Originator ori = new Originator();
        //定义出备忘录管理员
        Caretaker caretaker = new Caretaker();
        //初始化
        ori.setState1("中国");
        ori.setState2("强盛");
        ori.setState3("繁荣");
        System.out.println("===初始化状态===\n"+ori);
        //创建一个备忘录
        caretaker.setMemento(ori.createMemento());
        //修改状态值
        ori.setState1("软件");
        ori.setState2("架构");
        ori.setState3("优秀");
        System.out.println("\n===修改后状态===\n"+ori);
        //恢复一个备忘录
        ori.restoreMemento(caretaker.getMemento());
        System.out.println("\n===恢复后状态===\n"+ori);
    }
}

```

运行结果如下所示：

```
===初始化状态===
```

```
state1=中国
stat2=强盛
state3=繁荣

===修改后状态===
state1=软件
stat2=架构
state3=优秀

===恢复后状态===
state1=中国
stat2=强盛
state3=繁荣
```

通过这种方式的改造，甭管有多少状态都没有问题，直接把原有的对象所有属性都备份了一遍，想恢复当时的时点数据？那太容易了！

多备份的备忘录

各位读者中，不知道有没有做过系统级别的维护？比如 Backup Administrator(备份管理员)，每天负责查看系统的备份情况，所有的备份都是有自动化脚本产生的。有一天，突然有一个重要的系统说我数据库有点问题，请把上个月末的数据拉出来恢复，那怎么办？对备份管理员来说，这很好办，直接根据时间戳找到这个备份，还原回去就成了，但是对于我们的刚刚学习的备忘录模式却不行，为什么呢？它对于一个确定的发起人，永远只有一份备份，在这种情况下，单一的备份就不能满足要求了，我们需要设计一套多备份的架构。

我们先来说一个名词，检查点(Check Point)，就是你在备份的时候做的戳记，系统级的备份一般是时间戳，那我们程序的检查点该怎么设计呢？一般是一个有意义的字符串。

我们只要把通用代码中的 Caretaker 管理员稍作修改就可以了，代码如下：

```
public class Caretaker {
    //容纳备忘录的容器
    private HashMap<String,Memento> memMap = new HashMap<String,Memento>();

    public Memento getMemento(String idx) {
        return memMap.get(idx);
    }

    public void setMemento(String idx,Memento memento) {
        this.memMap.put(idx, memento);
    }
}
```



```
}  
  
}
```

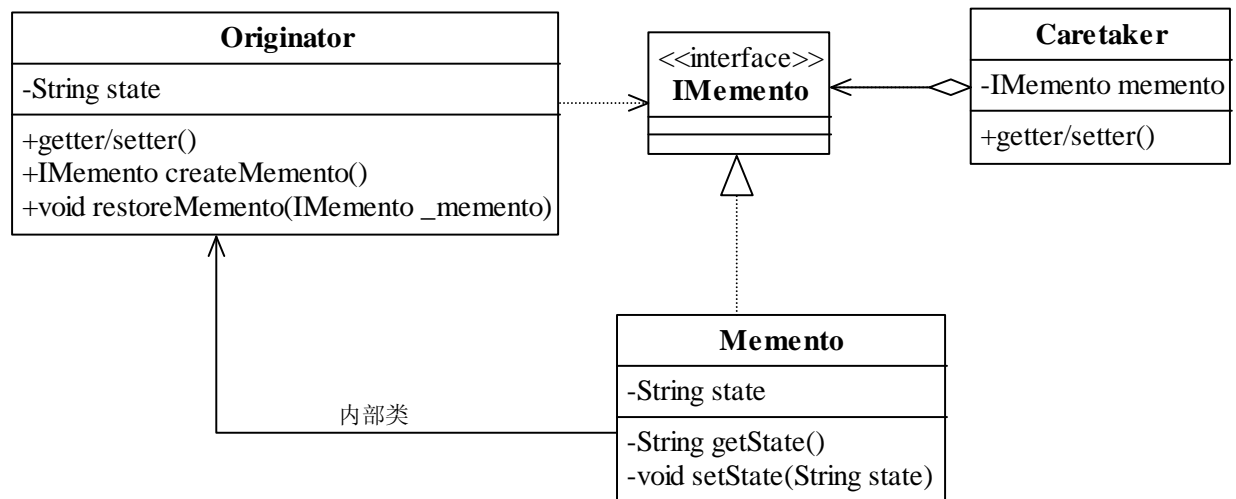
把容纳备忘录的容器修改为 Map 类型就可以了，场景类也稍作改动：

```
public class Client {  
  
    public static void main(String[] args) {  
        //定义出发起人  
        Originator originator = new Originator();  
        //定义出备忘录管理员  
        Caretaker caretaker = new Caretaker();  
        //创建两个备忘录  
        caretaker.setMemento("001",originator.createMemento());  
        caretaker.setMemento("002",originator.createMemento());  
        //恢复一个指定标记的备忘录  
        originator.restoreMemento(caretaker.getMemento("001"));  
    }  
}
```

需要注意的是，内存溢出问题，这个备份一旦产生就装入内存，没有任何销毁的意向，这是非常危险的，因此在系统设计时，要严格限定备忘录的创建，建议是增加Map的上限，否则系统很容易产生OutOfMemory内存溢出情况。

封装的更好一点

在系统管理上，一个备份的数据是完全的、绝对的不能修改，它保证数据的洁净，避免数据污染而使备份失去意义。在我们的设计领域中，也存在着同样的问题，备份是不能被篡改的，也就说是需要缩小备份出的备忘录的阅读权限，保证只能是发起人可读就成了，那怎么才能做到这一点呢？使用内置类，类图如下：



也是比较简单的，建立一个空接口 `IMemento`，什么方法属性都没有的接口，然后在发起人 `Originator` 类中建立一个内置类（也叫做类中类）`Memento` 实现 `IMemento` 接口，同时也实现自己的业务逻辑，代码如下：

```

public class Originator {

    //内部状态
    private String state = "";

    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
    }

    //创建一个备忘录
    public IMemento createMemento(){
        return new Memento(this.state);
    }

    //恢复一个备忘录
    public void restoreMemento(IMemento _memento){
        this.setState(((Memento)_memento).getState());
    }
}
  
```

```
//内置类
private class Memento implements IMemento{

    //发起人的内部状态
    private String state = "";

    //构造函数传递参数
    private Memento(String _state){
        this.state = _state;
    }

    private String getState() {
        return state;
    }

    private void setState(String state) {
        this.state = state;
    }

}

}
```

看到没，内置类 Memento 全部是 private 的访问权限，也就是说除了发起人外，别人休想访问的到，那如果要产生关联关系如何处理呢？通过接口，别忘记了我们还有一个空接口是公共的访问权限，代码如下：

```
public interface IMemento {

}
```

我们再来看管理者，代码如下：

```
public class Caretaker {

    //备忘录对象
    private IMemento memento;

    public IMemento getMemento() {
        return memento;
    }

}
```

```
    }  
  
    public void setMemento(IMemento memento) {  
        this.memento = memento;  
    }  
  
}
```

全部通过接口访问，这当然没有问题，如果你想访问它的属性那是肯定不行的。但是安全是相对的，没有绝对的安全，Memento 的数据真的就不能被修改了吗？不是，你使用 reflect 反射的话照样能修改，这个不再多说了。

我们在这里使用了一个新的设计方法：双接口设计，我们的一个类可以实现多个接口，在系统设计时，如果考虑对象的安全问题，则可以提供两个接口，一个是业务的正常接口，实现必要的业务逻辑，叫做宽接口，另外一个接口是一个空接口，什么方法都没有，其目的是提供给子系统外的模块访问，比如容器对象，这个叫做窄接口，由于窄接口中没有提供任何操纵数据的方法，因此相对来说比较安全。

最佳实践

备忘录模式是我们设计上“月光宝盒”，可以“biu~”的一下让我们回到需要的年代，是程序数据的“后悔药”，吃了它就可以返回上一个状态，是设计人员的定心丸，确保即使在最坏的情况下也能获得最近的对象状态。如果大家看懂了的话，请各位在设计的时候就不要使用数据库的临时表作为缓存备份数据了，虽然是一个简单的办法，但是它加大了数据库操作的频繁性，把压力下放到数据库。

第 25 章 模式大 PK

计划完成时间：2009 年 8 月 31 日

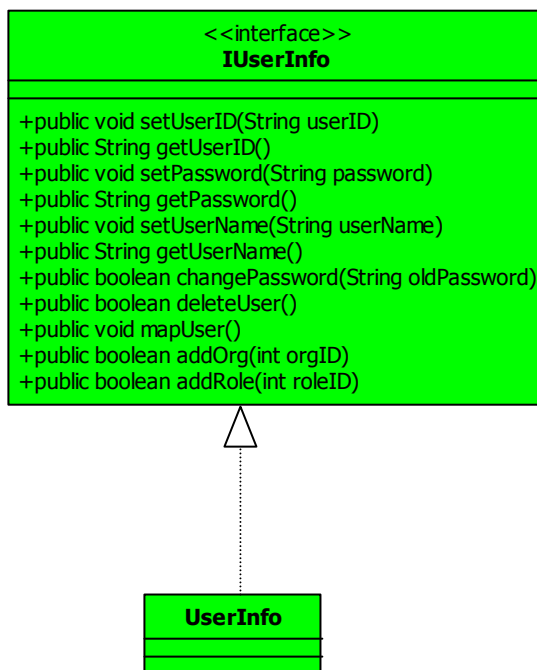
第 26 章 六大设计原则

26.1 单一职责原则【Single Responsibility Principle】

◇ 我是“牛”类，我可以担任多职吗？

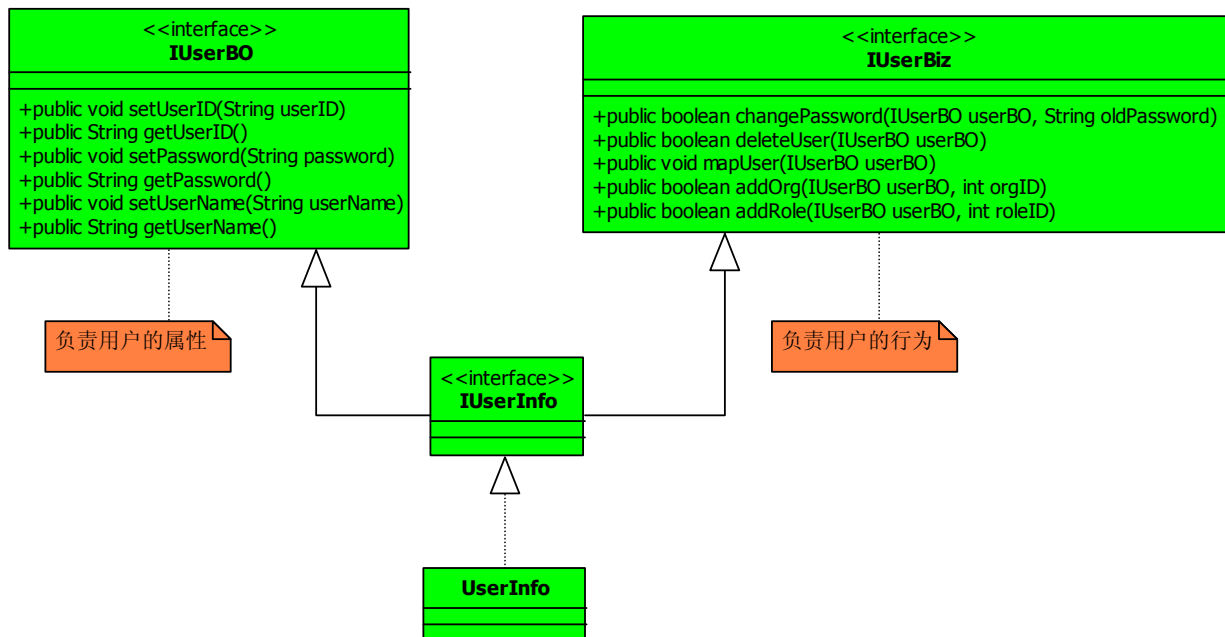
单一职责原则简称是 SRP，就是三个开通字母的缩写，这个设计原则备受争议的一个原则，只要你想和人争执、怄气或者是吵架，这个原则是屡试不爽的，如果你是老大，看到一个接口或类是这样…那样…设计的，你就问一句“你设计的类符合 SRP 原则吗？”，保准立马萎缩掉，而且还一脸崇拜的看着你“老大确实英明”，这个原则争论在什么地方呢？就是职责的定义，什么是类的职责，怎么划分类的职责。我们先讲个例子来说明什么是单一职责原则。

只要做过项目，肯定要接触到用户、机构、角色管理这个模块，基本使用都是 RBAC 这个模型，确实是很好的一个解决办法，我们今天要讲的是用户管理，管理用户的信息，增加机构（一个人属于多个机构），增加角色等，用户有这么的信息和行为要维护，我们就把这些写到一个接口中，都是用户管理类嘛，我们先来看类图：



太 easy 的类图了，我相信即使一个初级的程序员也可以看出这个接口设计的有问题，用户的属性（Properties）和用户的行为（Behavior）没有分开，这是一个严重的错误！非常正确，确实是这个接口设计的一团糟，应该把用户的信息抽取成一个业务对象(Bussiness Object, 简称 BO)，把行为抽取成另外一

到另外一个接口中，我们把这个类图重新画一下：



重新拆封成两个接口，IUserBO 负责用户的属性，简单的说就是 IUserBO 的职责是收集和反馈用户的属性信息；IUserBiz 负责用户的行为，完成用户的信息的维护和变更。各位可能要问了，这个和我实际的工作中用到的 User 类还是有差别的呀，别着急，我们先来想一想分拆成两个接口怎么使用，想清楚了，我们看是面向的接口编程，所有呀你产生了这个 UserInfo 对象之后，你当然可以把它当 IUserBO 接口使用，当然也可以当 IUserBiz 接口使用，这要看你在怎么地方使用了，你要获得用户想信息，你就当时 IUserBO 的实现类；你要是想维护用户的信息，就当是 IUserBiz 的实现类就成了，类似于如下代码：

```

.....

IUserBiz userInfo = new UserInfo();

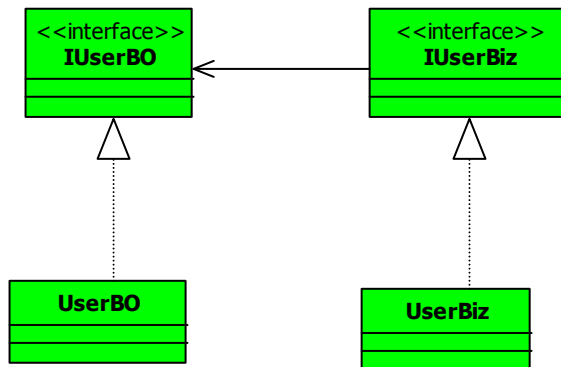
//我要赋值了，我就认为它是一个纯粹的BO
IUserBO userBO = (IUserBO)userInfo;
userBO.setPassword("abc");

//我要执行动作了，我就认为是一个业务逻辑类
IUserBiz userBiz = (IUserBiz)userInfo;
userBiz.deleteUser();

.....

```

确实可以如此，问题也解决掉了，但是我们来回想一下我们刚刚的动作，为什么要把一个接口拆分成两个？其实在实际的使用中，我们更倾向于使用两个不同的类或接口，一个就是 IUserBO，一个是 IUserBiz，类图应该如如下图：

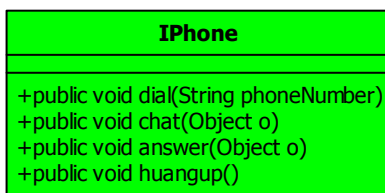


以上我们把一个接口拆分成两个接口的动作，就是依赖了单一职责原则，那什么是单一职责原则呢？

单一职责原则：应该有且仅有一个原因引起类的变更

✧ 绝杀技，打破你的传统思维

我解释到这里估计你现在很不屑了，“切！这么简单的东西还要讲？！弱智！”，好，我们来讲点复杂的。SRP 的原话解释是：There should never be more than one reason for a class to change。这句话初中生都能看懂，不多说，但是看懂是一回事，实施就是另外一回事了，上面我说的例子很好理解，大家已经都是这么做了，那我再举个例子来看看是否好理解。举个电话的例子，电话通话的时候有四过程发生：拨号、通话、回应、挂机，那我们写一个接口应该如下这个类图：



我不是有意要冒犯 iPhone 的，同名纯属巧合，我们来看一个接口程序：

```
package com.cbf4life.advance;
```

```
/**
```

```
 * @author cbf4Life cbf4life@126.com
```

```
 * I'm glad to share my knowledge with you all.
```



```
* 电话的接口
*/
public interface IPhone {

    //拨通电话
    public void dial(String phoneNumber);

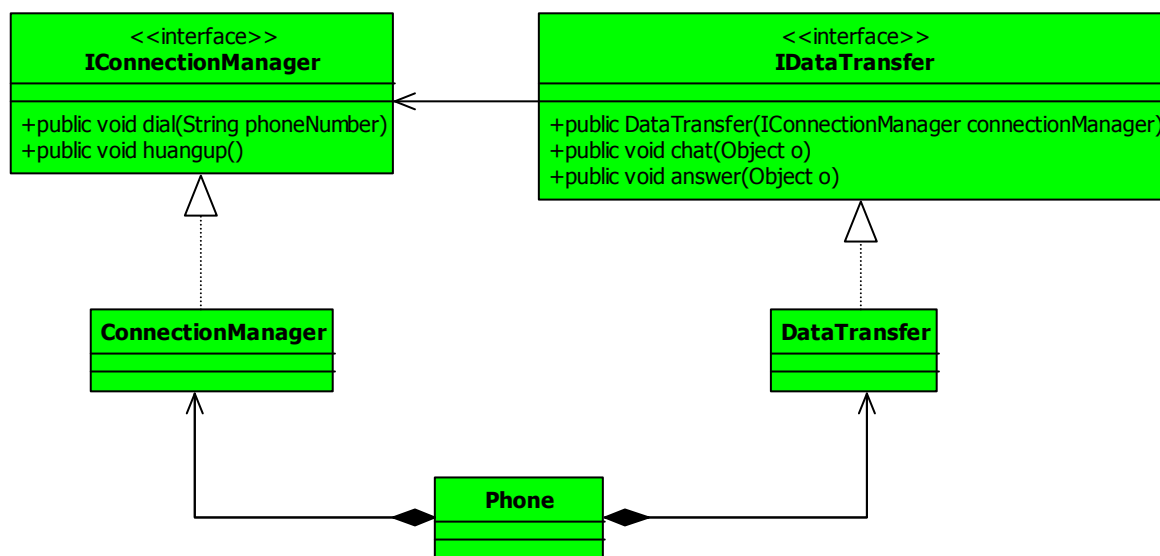
    //通话
    public void chat(Object o);

    //回应，只有自己说话而没有回应，那算啥？！
    public void answer(Object o);

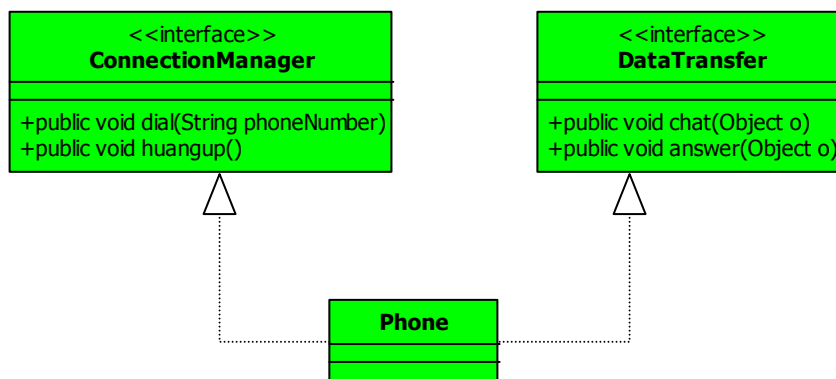
    //通话完毕，挂电话
    public void huangup();
}
```

实现类我就不再写了，大家看看这个接口有没有问题？我相信大部分的读者都会说这个问题没有，以前我就是这么做的呀，XXX 这本书上也是这么写的呀，还有什么什么的源码也是这么写的，是的，这个接口接近于完美，看清楚了是“接近于”。单一职责要求一个接口或类只有一个原因引起变化，也就是一个接口或类只有一个职责，它就负责一件事情，看看上面的接口只负责一件事情吗？是只有一个原因引起变化吗？好像不是！

IPhone 这个接口可不是只有一个职责，它是由两个职责：一个是协议管理，一个是数据传送，`dial()` 和 `huangup()` 两个方法实现的是协议管理，拨号接通和关闭；`chat()` 和 `answer()` 是数据的传送，把我们说的话转换成模拟信号或者是数字信号传递到对方，然后再把对方传递过来的信号还原成我们听的懂人话。我们可以这样考虑这个问题，协议接通的变化会引起这个接口或实现类的变化吗？会的！那数据传送（想想看，电话不仅仅可以通电话，还可以上网，Modem 拨号上网呀）的变化会引起这个接口或者实现类的变化吗？会的！那就很简单了，这里有两个原因都引起了类的变化，而且这两个职责会相互影响吗？电话通话，我只要能接通就行，甭管是 2G 还是 3G 的协议；电话连接后还关心传递的是什么数据吗？不关心，你要是乐意使用 56K 的小猫传递一个高清的片子，那也没有问题（顶多有人说你 13 了），也就说我们现在提供的这个 IPhone 接口**包含了两个职责，而且这两个职责的变化不相互影响**，那就考虑拆开成两个接口：



这个类图看这有点复杂了，完全满足了类和接口的单一职责要求，非常符合标准，但是我相信你在设计的时候肯定不会采用这种方式，一个手机类要把把两个 **ConnectionManager** 和 **DataTransfer** 组合在一块才能使用，组合是一种强耦合关系，你和我都是有共同的生命期，这样的强耦合关系还不如使用接口实现的方式呢，而且还增加了类的复杂性，多了两个类呀，好，我们修改一下类图：



这样的设计才是完美的，一个手机实现了两个接口，把两个职责融合一个类中，你会觉得这个 **Phone** 有两个原因引起变化了呀，是的是的，但是别忘记了我们是面向接口编程，我们对外公布的是接口而不是实现类；而且如果真要实现类的单一职责的话，这个就必须使用了上面组合的方式了，那这个会引起类间耦合过重的问题（我们等会再说明这个问题，继续往下看）。那使用了单一职责原则有什么好处呢？

类的复杂性降低，实现什么职责都有清晰明确的定义；

可读性提高，复杂性降低，那当然可读性提高了；

可维护性提高，那当然了，可读性提高，那当然更容易维护了；

变更引起的风险降低，变更是必不可少的，接口的单一职责做的好的话，一个接口修改只对相应的实现类有影响，与其他的接口无影响，这个是对项目有非常大的帮助。

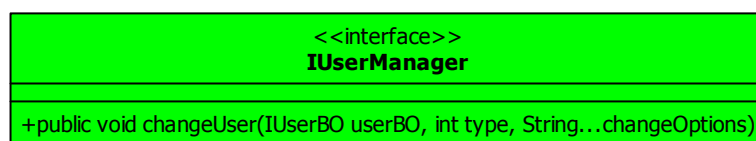
讲过这个例子后是不是有点凡反思了，我以前的设计是不是有点的问题了？是的，单一职责原则最难划分的就是职责，一个职责一个接口，但是问题是“职责”是一个没有量化的标准，一个类到底要负责那些职责？这些职责怎么细化？细化后是否都要有一个接口或类？这个都是需要从实际的项目区考虑的，从功能上来说，定义一个 iPhone 接口也没有错，实现了电话的功能呀，而且设计还很简单，就一个接口一个实现类，真正的项目我想大家一般都是会这么设计的，从设计原则上来看就有问题了，有两个可以变化的原因放到了一个接口中了，这就为以后的变化带来了风险，我从 2G 通讯协议修改到 3G 通讯，你看看你提供出的接口 iPhone 是不是要修改了？接口修改对其他的 Invoker 是不是有很大影响？！

◇ 我单纯，所有我快乐

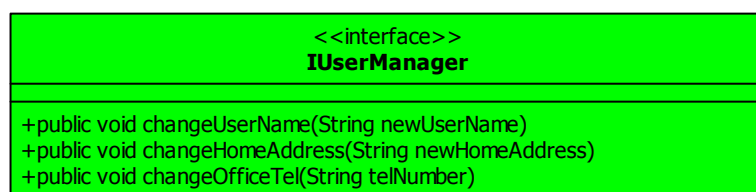
对于接口，我们在设计的时候一定要做到单一，但是对于实现类就需要多方面考虑了，生搬硬套单一职责原则会引起类的剧增，给维护带来非常多的麻烦；而且过分的细分类的职责也为人造的制造系统的负责性，本来一个类可以实现的行为非要拆成两个，然后使用聚合或组合的方式再耦合在一起，这个是人为了制造了系统的复杂性，所以原则是死的，人是活的，这句话是非常好的。

单一职责原则很难在项目中得到体现，非常难，为什么？考虑项目环境，考虑工作量，考虑人员的技术水平，考虑硬件的资源情况等等，最终融合的结果是经常违背这一单一原则。而且我们中华文明就有很多属于混合型的产物，比如筷子，我们使用筷子可以当做刀来使用，分割食物；还可以当叉使用，把食物从盘子中移动到口中；而在西方的文化中，刀就是刀，叉就是叉，你去吃西餐的时候这两样肯定都是有的，分工很明晰，刀就是切割食物，叉就是固定食物或者移动食物，这个中国的文化有非常大的关系，中国文化就是一个综合性的文化，要求一个人或是一个事物能够满足多个用途，一个人既要求你会技术还要会管理，还要会财务，要计算成本呀，这个是从小兵就开始要求了，哎，比较悲哀！我相信如果电脑是中国发明的话，肯定是这个样子：CPU、内存、主板、电源全部融合在一起，什么逻辑运算，数据处理，图像显示全部放一块，呵呵，有点愤青的成分在里面了。但是我相信随着技术的深入，单一职责原则必然会深入到项目的设计中去，而且这个原则是那么的简单，简单的我都不知道该怎么更加详细的去描述，单从字面上大家都应该知道是什么意思，单一职责嘛！

单一职责使用于接口、类，同时也使用方法，什么意思呢？一个方法尽可能做一件事情，比如一个方法修改用户密码，别把这个方法放到“修改用户信息”方法中，这个方法的颗粒度很粗，比如这个一个方法：



在 IUserManager 中定义了一个方法叫 changeUser，根据传递的 type 不同，把可变长度参数 changeOptions 修改到 userBo 这个对象上，并调用持久层的方法保存到数据库中。在我的项目组中如果有人写了这样一个方法，我不管他写了多上程序化了多少工夫，一律重写！原因是：方法职责不清晰，不单一，一般方法设计成这样的：



你要修改用户名称，就调用 changeUserName 方法，你要修改家庭地址就调用 changeHomeAddress，你要修改单位单户就调用 changeOfficeTel 方法，每个方法的职责就非常清晰，这也是一个良好的设计习惯。

所以，不管是对接口、类、方法使用了单一规则原则，那么快乐的就不仅仅是你了，还有你项目的成员，你的板，减少了因为变更引起的工作量呀，加官进爵等着你么！

你看到这里，就会问我，你写是类的设计原则吗？你通篇都在说接口的单一职责，类的单一职责你都违背了呀，呵呵，这个还真的是，我的本意是想把这个原则讲清楚，类的单一职责嘛，这个很简单，但当我回头写的时候，发觉才不是这么回事，翻看了以前一些设计和代码，基本上拿的出手的类设计都是和单一职责向违背的，静下心来回忆，发觉每一个类这样设计都是有原因的。这几天我查阅了 wikipedia、oodesign 等几个网站，专家和我也有类似的经验，基本上类的单一职责都用了类似的一句话来说 “This is sometimes hard to see”，这句话翻译过来就是 “这个有时候很难说”，是的，类的单一职责确实受非常多的因素制约，纯理论的来讲，这个原则是非常优秀的，但是现实有现实难处，你必须去考虑项目工期、成本、人员技术水平、硬件情况、网络情况甚至有时候还要考虑政府政策、垄断协议等等原因，比如 04 年就做过一个项目，做加密处理的，甲方就甩过来一句话，你什么都不用管，就调用这个 API 就可以了，不用考虑什么传输协议、异常处理、安全连接等等，所以我们就直接使用了 JNI 与加密厂商提供的 API 通信，什么单一职责原则，根本就不用考虑，因为人家不公布通信接口、异常处理。

对于单一职责原则，我的建议是接口一定要做到单一职责，类设计尽量只有一个原因引起变化。

26.2 里氏替换原则【Liskov Substitution Principle】

里氏替换法则是非常简单的，而且你在系统设计的时候肯定已经用到了，而且现在已经延伸到了 Web Service 的开发领域。里氏替换法则有两种定义：

第一个定义，最正宗的定义：If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.

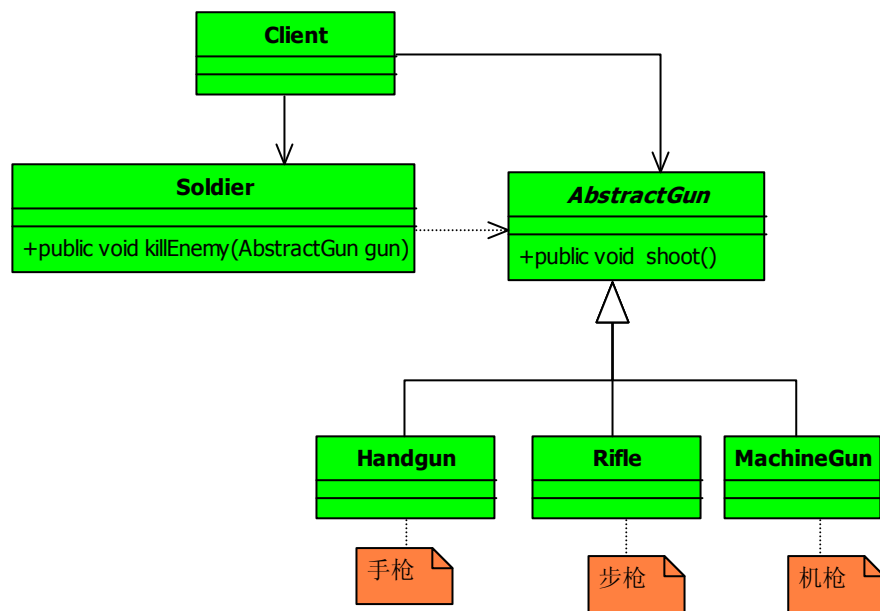
如果对每一个类型为 T1 的对象 o1，都有类型为 T2 的对象 o2，使得以 T1 定义的所有程序 P 在所有的对象 o1 都代换成 o2 时，程序 P 的行为没有发生变化，那么类型 T2 是类型 T1 的子类型。

第二个定义，functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

所有引用基类的地方必须能透明地使用其子类的对象。

第二个定义是最清晰明确的，通俗点讲只要父类能出现的地方我子类就可以出现，而且调用子类还不产生任何的错误或异常，调用者可能根本就不需要知道是父类还是子类。但是反过来就不成了，有子类出现的地方，父类未必就能适应，里氏替换法则包含了四层意思：

子类必须完全的实现父类的方法。我们在做系统设计时，经常会定义一个接口或者抽象类，然后编写实现，调用类则直接传入接口或抽象类，其实这里已经使用了里氏替换法则。我们举个例子还说明这个法则，大家都打过 CS 吧，非常经典的 FPS 类游戏，我们描述一下里面用到的枪，先看类图：



枪的主要职责就是射击，怎么射击就是在各个具体的子类中定义了，手枪是单发射程比较近，步枪威力大射程远，机枪用于扫射，然后在士兵类中定义了一个方法 `killEnemy` 杀敌人，使用枪来杀，具体使用什么枪来杀敌人，调用的时候才知道，我先看 `AbstractGun` 类的程序：

```
public abstract class AbstractGun {

    //枪用来干什么的？射击杀戮！
    public abstract void shoot();
}
```

以下是三个具体的枪械的实现类：

```
public class Handgun extends AbstractGun {

    //手枪的特点是携带方便，射程短
    @Override
    public void shoot() {
        System.out.println("手枪射击...");
    }

}

public class Rifle extends AbstractGun{
```

```
//步枪的特点是射程远，威力大
public void shoot(){
    System.out.println("步枪射击...");
}

public class MachineGun extends AbstractGun{

    public void shoot(){
        System.out.println("机枪扫射...");
    }
}
```

再来看士兵类的源码：

```
public class Soldier {

    public void killEnemy(AbstractGun gun){
        System.out.println("士兵开始杀人...");
        gun.shoot();
    }
}
```

注意看这里黄色标记的那部分，我们要求传入进来的是一个抽象的枪，具体是手枪还是步枪需要在调用的时候传入，我们来看 Client 类：

```
public class Client {

    public static void main(String[] args) {
        //产生三毛这个士兵
        Soldier sanMao = new Soldier();
        sanMao.killEnemy(new Rifle());
    }
}
```

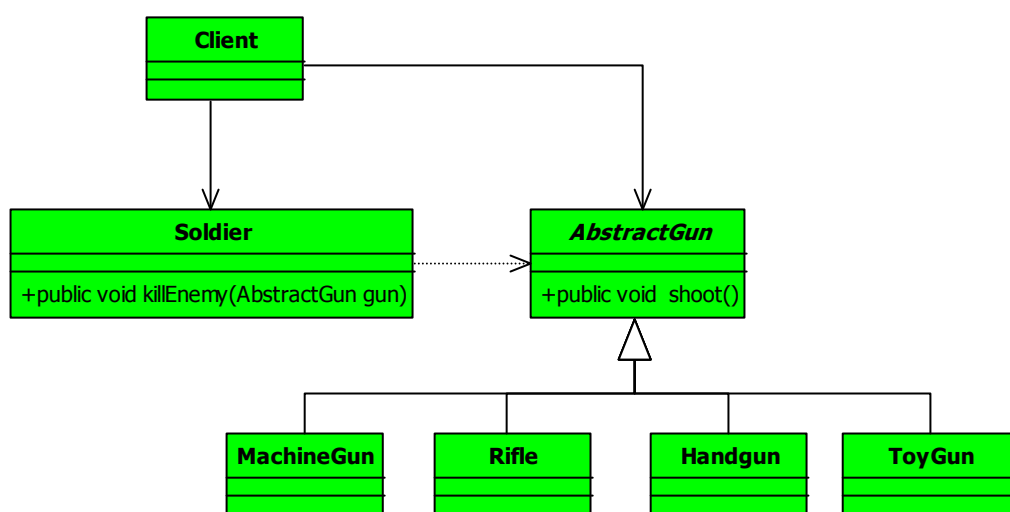
运行结果：

```
士兵开始杀人...
```

步枪射击...

在这个程序中，我们给三毛这个士兵一把步枪，然后就开始杀敌了，如果三毛要使用机枪当然也可以，直接把 `sanMao.killEnemy(new Rifle())` 修改为 `sanMao.killEnemy(new MachineGun())` 就可以了，Soldier 根本就不用知道是哪个子类。我们在类中调用其他类是务必要使用父类或接口，如果不能使用父类或接口，则说明类的设计已经违背了 LSP 原则。

我们再来想想，如果我们有一个玩具手枪，该怎么去定义呢？我们先在类图上增加一个类：



增加了一个 ToyGun 这个类，继承于 AbstractGun 抽象类。首先我们想，玩具枪是不能用来射击的，杀不死人的，当然你要把玩具枪往让头上砸也能砸死人，这个不算是在 shoot 方法中的功能。我们来看 ToyGun 类：

```

public class ToyGun extends AbstractGun {

    //玩具枪式不能射击的，但是编译器又要求实现这个方法，怎么办？虚假一个呗！
    @Override
    public void shoot() {
        //玩具枪不能射击，这个方法就不能实现了
    }

}
  
```

然后我们来看这个场景：

```

public class Client {
  
```



```
public static void main(String[] args) {  
    //产生三毛这个士兵  
    Soldier sanMao = new Soldier();  
    sanMao.killEnemy(new ToyGun());  
}
```

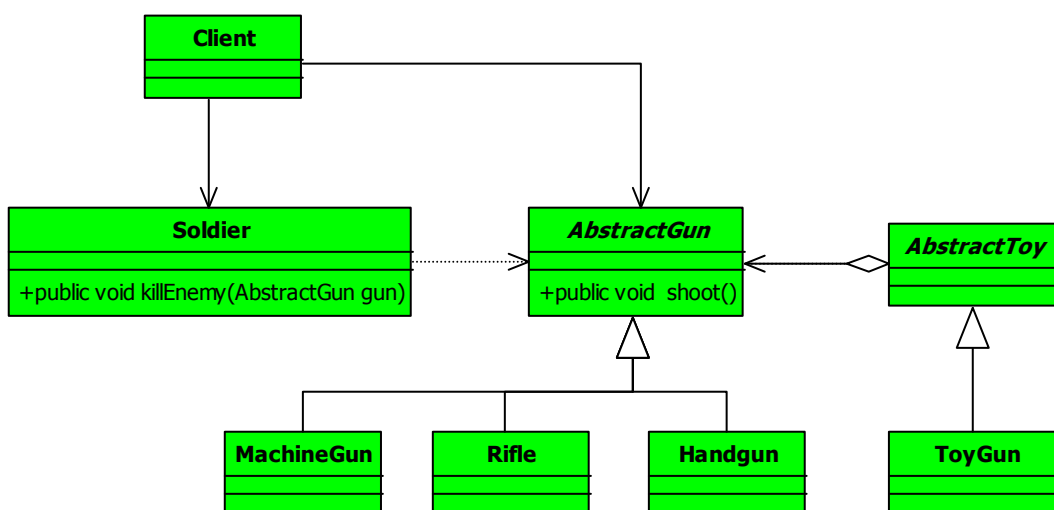
士兵使用玩具枪来杀人了，看运行结果：

士兵开始杀人...

坏了，士兵拿着玩具枪来杀人，射击不出子弹呀！如果在 CS 游戏中有这种事情发生，那你就等着被人爆头吧，然后看着自己凄凉的倒地。在这种情况下，我们已经发现业务调用类已经出现了问题，这个是业务已经不能运行了，那怎么办？有两种解决办法：

Number one:在 Soldier 中增加 instanceof 的判断，如果是玩具枪，就不用来杀敌人。这个方法可以解决问题，但是你要知道在项目中，特别是产品，增加一个类，我要让所有与这个父类有关系的类都需要修改，你觉的可行吗？你要是在产品中出现这个问题，因为修正了这样一个 BUG，就要求所有与这个父类有关系的类都增加一个判断？客户非跳起来跟你干！你还想要你的客户忠诚你吗？这个方案否定。

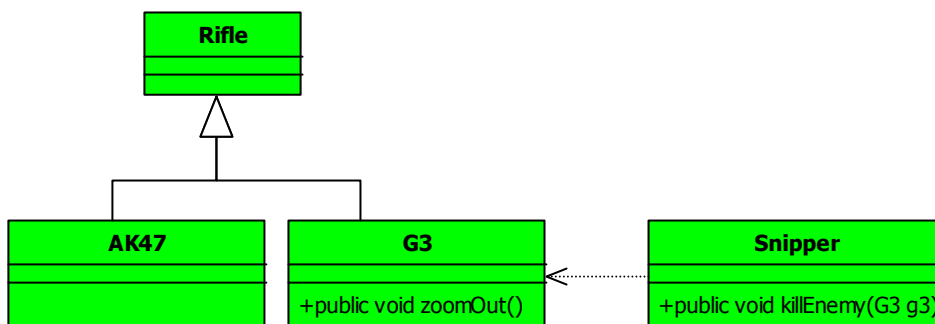
Number two:ToyGun 脱离继承，建立独立的父类，为了做到代码可以服用，可以与 AbstractGun 建立关联委托关系，如下图：



比如可以在 **AbstractToy** 中定义声音、形状都委托给 **AbstractGun**，仿真枪嘛当然要让形状、声音和真实的枪都一样了，然后两个父类下的子类各自发展，互不影响。

在 Java 的基础知识中，每位老师都会讲继承，Java 的三大特征嘛，继承、封装、多态，继承就是告诉你拥有父类的方法和属性，然后你就可以重写父类的方法。按照类的继承原则，我们上面的玩具枪继承 AbstractGun 也是没有问题的呀，毕竟也是枪嘛，但是到我们的具体项目中就要考虑这个问题了：子类是否完整的实现了父类的业务，否则就会出现像上面的拿枪杀敌人时却发现是把玩具枪的笑话。

子类可以有自己个性。子类当然可以有自己的行为和外貌了，也就是方法和属性，那这里为什么要再提呢？是因为里氏替换法则可以正着用，但是不能反过来用。在子类出现的地方，父类未必就可以胜任。还是以刚才的那个枪械的例子为来说明，步枪有几个比较响亮的型号比如 AK47、G3 狙击步枪等，我们来看类图：



很简单，G3 继承了 Rifle 类，狙击手(Sniper)则直接使用 G3 狙击步枪，我们来看一下程序：

```
public class G3 extends Rifle {

    //狙击枪都是携带一个精准的望远镜
    public void zoomOut(){
        System.out.println("通过望远镜观看敌人...");
    }

    public void shoot(){
        System.out.println("G3射击...");
    }
}
```

然后我们再声明一个狙击手类：

```
public class Sniper {
```

```
public void killEnemy(G3 g3){
    //首先看看敌人的情况，别杀死敌人，自己也被人干掉
    g3.zoomOut();
    //开始射击
    g3.shoot();
}
}
```

狙击手，为什么叫 Sniper? snipe 翻译过来就是鹬，就是鹬蚌相争，渔翁得利中的那个动物，英国贵族到印度打猎，发现这个鹬很聪明，人一靠近就飞走了，没办法就开始伪装、远程精准射击，于是乎 sniper 就诞生了。

我们来看一下业务业务场景是怎么调用的：

```
public class Client {

    public static void main(String[] args) {
        //产生三毛这个狙击手
        Sniper sanMao = new Sniper();
        sanMao.killEnemy(new G3());
    }
}
```

运行结果如下：

```
通过望远镜观看敌人...
G3射击...
```

在这里我们直接调用了子类，一个狙击手是很依赖枪支的，别说换一个型号的枪了，就是换一个同型号的枪也会影响射击的，所以这里就直接传递进来了子类。那这个时候，我们能不能直接使用父类传递进来呢？修改一下 Client 类：

```
public class Client {

    public static void main(String[] args) {
        //产生三毛这个狙击手
        Sniper sanMao = new Sniper();
        Rifle rifle = new Rifle();
    }
}
```

```
        sanMao.killEnemy((G3)rifle);
    }
}
```

显示是不行的，会在运行期报 [java.lang.ClassCastException](#) 异常，这也是大家经常说的向下转型（downcast）是不安全的，从里氏替换法则来看，就是有子类出现的地方父类未必就可以出现。

覆盖或实现父类的方法时输入参数可以被放大。方法中的输入参数叫做前置条件，这是什么呢？大家做过 Web Service 开发就应该知道有一个“契约优先”的原则，也就是先定义出 WSDL 接口，制定好双方的开发协议，然后再各自实现。里氏替换法则也要求制定了一个契约，就是父类或接口，这种设计方法也叫做 Design by Contract，契约优先设计，是和里氏替换法则融合在一起的。契约制定了，但是契约有前置条件和后置条件，前置条件就是你要让我执行，就必须满足我的条件；后置条件就是我执行完了，必须符合规定的契约。这个比较难理解，我们来看一个例子，我们先定义个 Father 类：

```
public class Father {
    public Collection doSomething(HashMap map){
        System.out.println("父类被执行...");
        return map.values();
    }
}
```

这个类非常简单，就是把 HashMap 转换为 Collection 集合类型，然后我们来看子类：

```
public class Son extends Father {

    //放大输入参数类型
    public Collection doSomething(Map map){
        System.out.println("子类被执行...");
        return map.values();
    }
}
```

大家注意看黄色明显标记部分，和父类同样的一个方法名称，但是又不是重写（Override）父类的方法，你加个@Override 试试看，报错的，为什么呢？是输入参数类型不同，编译器就不认为是重写父类的方法了，那这是什么？是重载（Overload）！不用大惊小怪的，不在一个类就不能是重载了？继承是什么意思，

子类拥有父类的所有属性和方法，那方法名重复输入参数类型又不相同当然是重载了。我们再来看业务调用类：

```
public class Client {  
    public static void invoker(){  
        //父类存在的地方，子类就应该能够存在  
        Father f = new Father();  
        HashMap map = new HashMap();  
        f.doSomething(map);  
    }  
  
    public static void main(String[] args) {  
        invoker();  
    }  
}
```

运行结果如下：

父类被执行...

里氏替换法则说是父类出现的地方子类就能出现，我们把上面的黄色部分修改为子类，程序如下：

```
public class Client {  
    public static void invoker(){  
        //父类存在的地方，子类就应该能够存在  
        Son f = new Son();  
        HashMap map = new HashMap();  
        f.doSomething(map);  
    }  
  
    public static void main(String[] args) {  
        invoker();  
    }  
}
```

运行结果还是一样，看明白是怎么回事了吗？父类方法的输入参数是 HashMap 类型，子类的输入参数是 Map 类型，也就是说子类的输入参数类型的范围扩大了，子类代替父类传递到调用类用，子类的方法永

远都不回被执行，这是正确的，如果你想让子类的方法运行，你就必须重写父类的方法。大家可以这样想想看，在一个 Invoker 类中关联了一个父类，调用了父类的方法，子类可以重写这个方法，也可以重载这个方法，前提是要扩大这个前置条件，就是输入参数的类型大于父类的类型覆盖范围。可能比较难理解，那我们再反过来想一下，如果 Father 类的输入参数类型大于子类的输入参数类型，会出现什么问题？就会出现父类存在的地方，子类就未必可以存在，因为一旦把子类作为参数传入进去，调用者就很可能进入子类的方法范畴。我们把上面的例子修改一下，先看父类：

```
public class Father {  
  
    public Collection doSomething(Map map){  
        System.out.println("Map 转Collection被执行");  
        return map.values();  
    }  
}
```

把父类的前置条件修改为 Map 类型，我们再修改一下子类方法的输入参数，相对父类缩小输入参数的类型范围，也就是缩小前置条件：

```
public class Son extends Father {  
  
    //缩小输入参数范围  
    public Collection doSomething(HashMap map){  
        System.out.println("HashMap转Collection被执行...");  
        return map.values();  
    }  
}
```

再来看业务场景类：

```
public class Client {  
    public static void invoker(){  
        //有父类的地方就有子类  
        Father f= new Father();  
        HashMap map = new HashMap();  
        f.doSomething(map);  
    }  
}
```

```
public static void main(String[] args) {  
    invoker();  
}  
}
```

运行结果如下：

父类被执行...

那我们再把里氏替换法则引入进来会有什么问题？有父类的地方子类就可以使用，好，我们把这个 Client 类修改一下，程序如下：

```
public class Client {  
    public static void invoker(){  
        //有父类的地方就有子类  
        Son f = new Son();  
        HashMap map = new HashMap();  
        f.doSomething(map);  
    }  
  
    public static void main(String[] args) {  
        invoker();  
    }  
}
```

运行结果如下：

子类被执行...

完蛋了吧？！子类在没有重写父类的方法的前提下，子类方法被执行了，这个绝对会引起以后的业务逻辑混乱，因为在项目的应用中父类一般都是抽象类，子类是实现类，你传递一个这样的实现类就会引起一堆意想不到的业务逻辑混乱，所以子类中方法的前置条件必须与超类中被覆盖的方法的前置条件相同或者更宽松。

覆盖或实现父类的方法是输出结果可以被缩小。这个是什么意思呢，父类的一个方法返回值是一个类

型 T，子类相同方法(重载或重写)返回值为 S，那么里氏替换法则就要求 S 必须小于等于 T，也就是说要么 S 和 T 是同一个类型，要么 S 是 T 的子类，为什么呢？分两种情况，如果是重写，方法的输入参数父类子类是相同的，两个方法的返回值 S 小于等于 T，这个是重写的要求，这个才是重中之重，子类重写父类的方法，天经地义；如果是重载，则要求方法的输入参数不相同，在里氏替换法则要求下就是子类的输入参数大于等于父类的输入参数，那就是说你写的这个方法是不会被调用到的，参考上面讲的前置条件。

里氏替换法则诞生的目的就是加强程序的健壮性，同时版本升级也可以做到非常好的兼容性，增加子类，原有的子类还可以继续运行。在我们项目实施中就是每个子类对应了不同的业务含义，使用父类作为参数，传递不同的子类完成不同的业务逻辑，非常完美！

26.3 依赖倒置原则【Dependence Inversion Principle】



26.4 接口隔离原则【Interface Segregation Principle】

在讲接口隔离原则之前，我们先明确一下我们的主角，什么是接口，接口分为两种：一种是实例接口（Object Interface），在 Java 中声明一个类，然后用 new 关键字产生的一个实例，它是对一个类型的事物描述，这是一种接口，比如你定义个 Person 这个类，然后使用 `Person zhangSan = new Person()` 产生了一个实例，这个实例要遵从的标准就是 Person 这个类，Person 类就是 zhangSan 的接口，看不懂？不要紧，那是让 Java 语言浸染的时间太长了。主角已经出场了，那我们看它的原则是什么，它有两种定义：

第一种定义：Clients should not be forced to depend upon interfaces that they don't use.
客户端不应该依赖它不需用的接口。

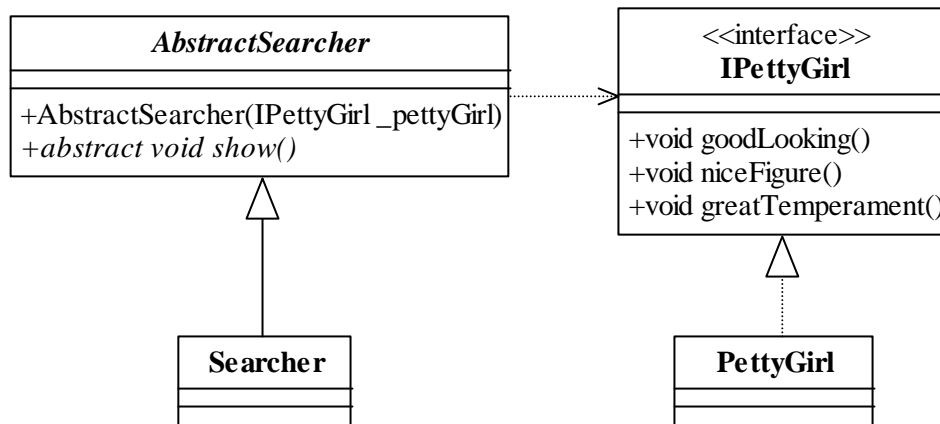
第二种定义：The dependency of one class to another one should depend on the smallest possible interface。类间的依赖关系应该建立在最小的接口上。

一个新事物的定义一般都是比较难理解的，晦涩难懂是正常的，否则那会让人家觉的你没有水平，这也是一些国际厂商在国内忽悠的基础，不整些名词怎么能让你崇拜我呢？我们把这个定义剖析一下，先说第一种定义客户端不应该依赖它不需要接口，那依赖什么？依赖它需要的接口，客户端需要什么接口就提供什么借口，把不需要的接口剔除掉，那就需要对接口进行细化，保证其纯洁性；再看第二个定义，类间的依赖关系应该建立在最小的接口上，它要求是最小的接口，也是要求接口细化，接口纯洁，与第一个定义如出一辙，只是一个事物的两种不同描述。

我们可以把这两个定义概括为一句话：建立单一接口，不要建立臃肿庞大的接口。再通俗的一点讲：接口尽量细化，同时接口中的方法尽量少的。看到这里大家有可能要疑惑了，这与单一职责原则不是相同的吗？错，接口隔离原则与单一职责的定义的规则是不相同的，单一职责要求的是类和接口职责单一，注重的是职责，没有要求接口的方法减少，例如一个职责可能包含 10 个方法，这 10 个方法都放在一个接口中，并且提供给多个模块访问，各个模块按照规定的权限来访问，在系统外通过文档约束不使用的方法不要访问，按照单一职责原则是允许的，按照接口隔离原则是不允许的，因为它要求“尽量使用多个专门的接口”，专门的接口指什么？就是指提供给多个模块的接口，提供给几个模块就应该有几个接口，而不是建立一个庞大的臃肿的接口，所有的模块可以来访问。

我们来举个例子来说明接口隔离原则到底对我们提出了什么要求。现在男生对小姑娘的称呼使用频率最高的应该是“美女”了吧，我们今天来定义一下什么是美女：首先要面貌好看，其次是身材要窈窕，然后要有气质，当然了，这三者各人的排列顺序不一样，总之要成为一名美女就必须具备：面貌、身材和气

质，我们用类图来体现一下星探（当然，你也可以把你想想成星探）找美女的过程，看类图：



定义了一个 **IPettyGirl** 接口，声明所有的美女都应该有 `goodLooking`、`niceFigure` 和 `greatTemperament`，然后又定义了一个抽象类 **AbstractSearcher**，其作用就是搜索美女然后展示信息，只要美女都是按照这个规范定义，**Searcher** (星探) 就轻松的多了，我们先来看美女的定义：

```

public interface IPettyGirl {

    //要有姣好的面孔
    public void goodLooking();

    //要有好身材
    public void niceFigure();

    //要有气质
    public void greatTemperament();
}
  
```

美女就是这样的一个定义，各位色狼别把口水流到了键盘上，然后我们看美女的实现类：

```

public class PettyGirl implements IPettyGirl {
    private String name;
    //美女都有名字
    public PettyGirl(String _name){
        this.name=_name;
    }

    //脸蛋漂亮
  
```

```
public void goodLooking() {
    System.out.println(this.name + "---脸蛋很漂亮!");
}

//气质要好
public void greatTemperament() {
    System.out.println(this.name + "---气质非常好!");
}

//身材要好
public void niceFigure() {
    System.out.println(this.name + "---身材非常棒!");
}
}
```

然后我们来看 AbstractSearcher 类，这个类一般就是指星探这个行业了，源代码如下：

```
public abstract class AbstractSearcher {
    protected IPettyGirl pettyGirl;
    public AbstractSearcher(IPettyGirl _pettyGirl){
        this.pettyGirl = _pettyGirl;
    }

    //搜索美女，列出美女信息
    public abstract void show();
}
```

星探查找到美女，打印出美女的信息，源码如下：

```
public class Searcher extends AbstractSearcher{
    public Searcher(IPettyGirl _pettyGirl){
        super(_pettyGirl);
    }

    //展示美女的信息
    public void show(){
        System.out.println("-----美女的信息如下：-----");
        //展示面容
        super.pettyGirl.goodLooking();
        //展示身材
    }
}
```

```
        super.pettyGirl.niceFigure();  
        //展示气质  
        super.pettyGirl.greatTemperament();  
    }  
}
```

场景中的两个角色美女和星探都已经完成了，我们再来写个场景类，展示一下我们的这个过程：

```
public class Client {  
  
    //搜索并展示美女信息  
    public static void main(String[] args) {  
        //定义一个美女  
        IPettyGirl yanYan = new PettyGirl("嫣嫣");  
        AbstractSearcher searcher = new Searcher(yanYan);  
        searcher.show();  
    }  
}
```

运行结果如下：

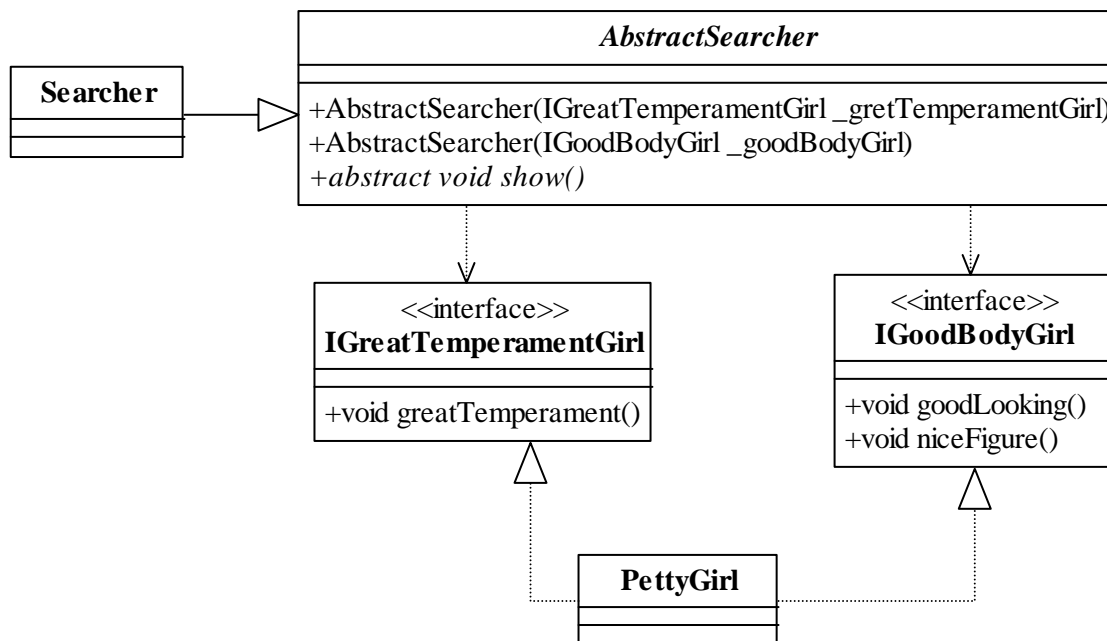
```
-----美女的信息如下：-----  
嫣嫣---脸蛋很漂亮！  
嫣嫣---身材非常棒！  
嫣嫣---气质非常好！
```

星探寻找美女的程序我们就开发完毕了，我们来想想这个程序有没有问题，思考一下 IPettyGirl 这个接口，这个接口是否做到了最优秀的设计。

我们的审美观点都在改变，美女的定义也在变化。一千多年前的唐朝杨贵妃如果活在现代这个年代非羞愧死不行，为什么？胖呀！但是胖不影响她入选中国的四大美女行列，说明当时的审美和现在是有差异地，当然随着时代的发展我们的审美观也在变化，突然有一天，也不用突然有一天，就现在，你发现有后一个女孩，脸蛋不怎么样，身材也一般般，但是气质非常好，我相信大部分人都会把这样的女孩叫美女，审美素质提升了，但是我们接口却定义了美女必须是三者都具备呀，可能你要说了，我重新扩展一个美女类，只实现 greatTemperament 方法其他两个方法置空，什么都不写，不就可以了吗？聪明，但是行不通！为什么呢？星探 AbstractSearcher 依赖的是 IPettyGirl 接口，它有三个方法，你只实现了两个方法，星

探的方法是不是要修改？我们上面的程序打印出来的信息少了两条，还让星探怎么去辨别是不是美女呢？

好了，我们发现我们的接口 IPettyGirl 接口设计是有缺陷地，过于庞大了，容纳了一些可变的因素，根据接口隔离原则，星探 AbstractSearcher 应该依赖与具有部分特质的女孩子，而我们却把这些特质都封装了起来，放到了一个接口中了，封装过渡了！问题查找到了，我们重新修改一下类图：



把原 IPettyGirl 接口拆分为两个接口，一种是外形美的美女 IGoodBodyGirl，这类美女的特点就是脸蛋和身材极棒，超一流，但是没有审美素质，比如随地吐痰，出口就是 KAO，CAO 之类的，文化程度比较低；另外一种气质美的美女 IGreatTemperamentGirl，谈吐和修养都非常高。我们从一个比较臃肿的接口拆分成了两个专门的接口，灵活性提高了，可维护性也增加了，不管以后是要外形美的美女还是气质美的美女都可以轻松的通过 PettyGirl 定义。我们先看两种类型的美女接口：

```

public interface IGoodBodyGirl {
    //要有姣好的面孔
    public void goodLooking();

    //要有好身材
    public void niceFigure();
}

public interface IGreatTemperamentGirl {
    //要有气质

```

```
    public void greatTemperament();  
}
```

实现类没有改变，只是实现类两个接口，源码如下：

```
public class PettyGirl implements IGoodBodyGirl,IGreatTemperamentGirl {  
    private String name;  
    //美女都有名字  
    public PettyGirl(String _name){  
        this.name=_name;  
    }  
  
    //脸蛋漂亮  
    public void goodLooking() {  
        System.out.println(this.name + "---脸蛋很漂亮!");  
    }  
  
    //气质要好  
    public void greatTemperament() {  
        System.out.println(this.name + "---气质非常好!");  
    }  
  
    //身材要好  
    public void niceFigure() {  
        System.out.println(this.name + "---身材非常棒!");  
    }  
}
```

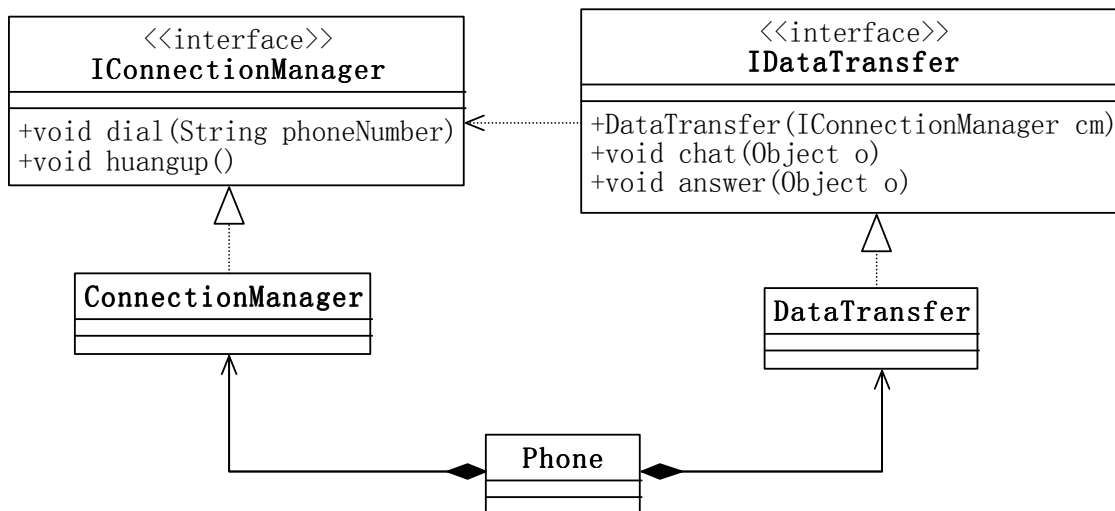
通过这样的改造以后，不管以后是要气质美女还是要外形美女，都可以保持接口的稳定。当然你可能要说了，以后可能审美观点再发生改变，只有脸蛋好看就是美女，那这个 IGoodBody 接口还是要修改的呀，确实是，但是设计时有限度的，不能无限的考虑未来的变更情况，否则就会陷入设计的泥潭中而不能自拔。

以上把一个臃肿的接口变更为两个独立的接口依赖的原则就是接口隔离原则，让 AbstractSearcher 依赖两个专用的接口比依赖一个综合的接口要灵活。接口是我们设计时对外提供的契约，通过分散定义多个接口，可以预防未来变更的扩散，提高系统的灵活性和可维护性。

接口隔离原则是对接口进行规范约束，其包含以下四层含义：

接口尽量要小。这是接口隔离原则的核心定义，不出现臃肿的接口（Fat Interface），但是“小”是有限度的，首先就是不能违反单一职责原则，什么意思呢？我们在单一职责原则中提到一个 iPhone 的例子，

在这里例子中我们使用单一职责原则把在一个接口中的方法分解到两个接口中，类图如下：

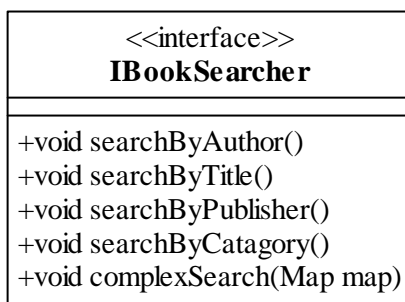


我们想想 **IConnectManager** 接口是否还可以再继续拆分下去，挂电话有两种方式：一种是正常的电话挂掉，一种是手机突然没电了，通讯当然就断了，这两种方式的处理应该是不同的，为什么呢？正常挂电话，对方接受到挂机信号，计费系统也就停止计费了，那手机没电了这种方式就不同了，它是信号丢失了，中继服务器检查到了，然后通知计费系统停止计费，否则你的费用不是要疯狂的增长了吗？！思考到这里，我们是不是就要动手把 **IConnectManager** 接口拆封成两个，一个是负责连接的，一个接口是负责挂电话的？是要这样做吗？且慢，让我们再思考一下，如果拆分了，那就不符合单一职责原则了，因为从业务上来讲通讯的建立和关闭已经是最小的业务单位了，再细分下去就是对业务或是协议(其他业务逻辑)的拆解了，想想看一个电话要关心 3G 协议，要考虑中继服务器等等，这个电话还这么做的出来呢？从业务层次来看这样的设计就是一个失败的设计。一个原则要拆，你原则又不要拆，那怎么办？好办，**根据接口隔离原则拆分接口时，必须首先满足单一职责原则。**

接口要高内聚。什么是高内聚？高内聚就是提高接口、类、模块的处理能力，减少对外的交互，就如一个人，你告诉下属“到奥巴马的办公室偷一个 XX 文件”，然后就听到下属就坚定的口吻回答你“好的，保证完成！”，然后一个月后还真的把 XX 文件放到你的办公桌了，这种不讲任何条件、立刻完成任务的行为就是高内聚的表现。具体到接口隔离原则就是要求在接口中尽量少公布 public 方法，接口是对外的承诺，承诺越少对系统的开发越有利，变更的风险也就越少，同时也有利于降低成本。

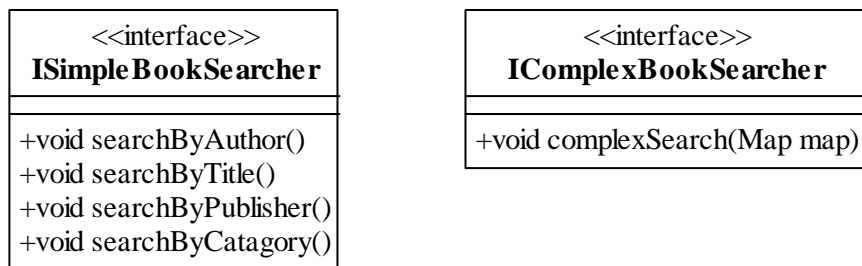
定制服务。一个系统或系统内的模块之间必然会有耦合，有耦合就要相互访问的接口（并不不一定是 Java 中定义的 Interface，也可能是一个类或者是单纯的数据交换），我们设计时就需要给各个访问者（也

就客户端)定制服务,什么是定制服务?你到商场买衣服,找到符合自己身体的尺码的衣服就成了,基本上就不会差别太大,可能是前松后紧,晚上睡不着觉之类的不太合适,但是好歹也是个衣服,能穿。如果你到裁缝店里做衣服会是什么样子呢?裁缝会帮你量腰围,胸围,肩宽等等,然后做出一件衣服,这件衣服肯定非常符合你的身体,那这就是定制服务,单独为一个个体提供优良优良的服务。我们在做系统设计时也需要考虑对系统之间或模块之间的定义要采用定制服务,采用定制服务就必然有一个要求就是:只提供访问者需要的方法,这是什么意思?我们举个例子来说明,比如我们做了一个图书管理系统,其中有一个查询接口的,方便管理员查询图书,其类图如下:



在接口中定义了多个方法,分别可以按照作者,标题,出版社,分类查询,最后还提供了混合查询方式,程序也写好了,投产上线了,突然有一天发现系统速度非常慢,然后就开始痛苦的分析,最终发现是通过这个接口的 `complexSearch(Map map)` 方法并发量太大,导致应用服务器性能下降,然后继续跟踪下去才发现这些查询都是从公网上发起的,进一步分析,发现问题了:提供给公网(公网项目是另外一个项目组开发的)的查询接口和提供给系统内管理人员的接口是相同的,都是 `IBookSearcher` 接口,但是权限不同,系统管理人员可以通过这个接口查询到所有的书籍,而公网的这个方法是被限制的,不返回任何值的,在设计时通过口头约束,这个方法是不可被调用的,但是由于公网项目组的疏忽,这个方法还是公布了出去,虽然不能返回结果,但是还是引起了应用服务器的性能巨慢的情况发生,这就是活生生的一起接口臃肿引起性能部长的案例。

问题找到了,我们就把这个接口进行重构:把 `IBookSearcher` 拆分为两个接口,如下图:



提供给管理人员的实现类同时实现 `ISimpleBookSearcher` 和 `IComplexBookSearcher` 两个接口，原有程序不用任何改变，而提供给公网的接口变为 `ISimpleBookSearcher`，只允许进行简单的查询，单独为它定制服务。

接口设计是有限度的。接口的设计粒度是越小系统越灵活，这是不争的事实，但是这就带来的结构的复杂化，开发难度增加，维护性降低，这不是一个项目或产品所期望看到的，所有接口设计一定要注意适度，适度的“度”怎么来判断的呢？根据经验和常识判断！

接口隔离原则是对接口的定义也同时是对类的定义，接口和都尽量使用原子接口或原子类来组装，但是这个原子该怎么划分是这个模式也是设计中的一大难题，在实践中应用时可以根据以下几个规则来衡量：

一个接口只服务于一个子模块或者业务逻辑。

通过业务逻辑压缩接口中的 `public` 方法。接口时常去回顾，尽量做让接口达到“满身筋骨肉”，而不是“肥嘟嘟”的一大堆方法。

已经被污染了的接口，尽量去修改，若变更的风险较大，则采用适配器模式进行转化处理。

了解环境，拒绝盲从。每个项目或产品都有特定的环境因素，别看到大师是这样做的你就照抄，千万别，环境不同的，接口拆分的标准就不同。深入了解的业务逻辑，最好的接口设计就出自的你的手！

接口隔离原则和其他的设计原则一样，都是需要花费较多的时间和精力来进行设计和筹划，但是它带来了设计的灵活性，让你在业务人员在提出“无理”要求的时候可以轻松应付。贯彻使用接口隔离原则最好的方法就是一个接口一个方法，保证绝对符合接口隔离原则（有可能不符合单一职责原则），但你会采用吗？！不会，除非你是疯子！那怎么才能正确的使用接口隔离原则呢？答案是根据经验和常识决定接口的粒度大小，接口粒度太小，导致接口数据剧增，开发人员呛死在接口的海洋里；接口粒度太大，灵活性降低，无法提供定制服务，给整体项目带来无法预计的风险。

怎么准确的实践接口隔离原则？一句话：实践，经验和领悟！

✧

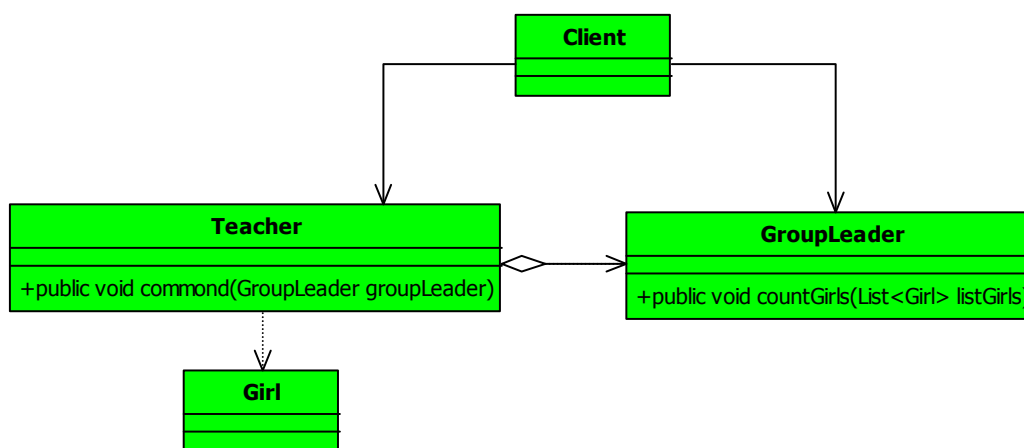
26.5 迪米特法则【Low Of Demeter】

迪米特法则的简写为 LoD, 看清楚中间的那个 o 是小写。迪米特法则也叫做最少知识原则 (Least Knowledge Principle, 简称 LKP) 说的都是一会事, 一个对象应该对其他对象有最少的了解, 通俗的讲一个类对自己需要耦合或者调用的类应该知道的最少, 你类内部是怎么复杂、怎么的纠缠不清都和我没关系, 那是你的类内部的事情, 我就知道你提供的这么多 public 方法, 我就调用这个; 迪米特法则包含以下四层意思:

只和朋友交流。迪米特还有一个英文解释叫做 “Only talk to your immedate friends”, 只和直接的朋友通信, 什么叫做直接的朋友呢? 每个对象都必然会和其他对象有耦合关系, 两个对象之间的耦合就成为朋友关系, 这种关系有很多比如组合、聚合、依赖等等。我们来说个例子说明怎么做到只和朋友交流。

说是有这么一个故事, 老师想让体育委员确认一下全班女生来齐没有, 就对他说: “你去把全班女生清一下。” 体育委员没听清楚, 或者是当时脑子正在回忆什么东西, 就问道: “亲哪个?” 老师 ¥#……¥%。

我们来看这个笑话怎么用程序来实现, 先看类图:



Teacher.java 的源程序如下:

```
package com.cbf4life.common;
```

```
import java.util.ArrayList;
```

```
import java.util.List;
```

```
/**
```

```

* @author cbf4Life cbf4life@126.com
* I'm glad to share my knowledge with you all.
* 老师类
*/
public class Teacher {

    //老师对学生发布命令,清一下女生
    public void commond(GroupLeader groupLeader){
        List<Girl> listGirls = new ArrayList();
        //初始化女生
        for(int i=0;i<20;i++){
            listGirls.add(new Girl());
        }

        //告诉体育委员开始执行清查任务
        groupLeader.countGirls(listGirls);
    }
}

```

老师就有一个方法，发布命令给体育委员，去清查一下女生的数量。下面是体育委员 GroupLeader.java 的源程序：

```

package com.cbf4life.common;

import java.util.List;

/**
* @author cbf4Life cbf4life@126.com
* I'm glad to share my knowledge with you all.
* 体育委员，这个太难翻译了都是中国的特色词汇
*/
public class GroupLeader {

    //有清查女生的工作
    public void countGirls(List<Girl> listGirls){
        System.out.println("女生数量是: "+listGirls.size());
    }
}

```

下面是 Girl.java，就声明一个类，没有任何的代码：

```
package com.cbf4life.common;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 女生
 */
public class Girl {

}
```

我们来看这个业务调用类 Client:

```
package com.cbf4life.common;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 我们使用Client来描绘一下这个场景
 */
public class Client {

    public static void main(String[] args) {
        Teacher teacher= new Teacher();

        //老师发布命令
        teacher.commond(new GroupLeader());
    }
}
```

运行的结果如下:

```
女生数量是: 20
```

我们回过头来看这个程序有什么问题, 首先来看 Teacher 有几个朋友, 就一个 GroupLeader 类, 这个就是朋友类, 朋友类是怎么定义的呢? 出现在成员变量、方法的输入输出参数中的类被称为成员朋友类, 迪米特法则说是一个类只和朋友类交流, 但是 commond 方法中我们与 Girl 类有了交流, 声明了一个 List<Girls>动态数组, 也就是与一个陌生的类 Girl 有了交流, 这个不好, 那我们再来修改一下, 类图还

是不变，先修改一下 GroupLeader 类，看源码：

```
package com.cbf4life.common2;

import java.util.ArrayList;
import java.util.List;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 体育委员，这个太难翻译了都是中国的特色词汇
 */
public class GroupLeader {

    //有清查女生的工作
    public void countGirls(){
        List<Girl> listGirls = new ArrayList<Girl>();
        //初始化女生
        for(int i=0;i<20;i++){
            listGirls.add(new Girl());
        }

        System.out.println("女生数量是: "+listGirls.size());
    }
}
```

下面是 Teacher.java 程序：

```
package com.cbf4life.common2;

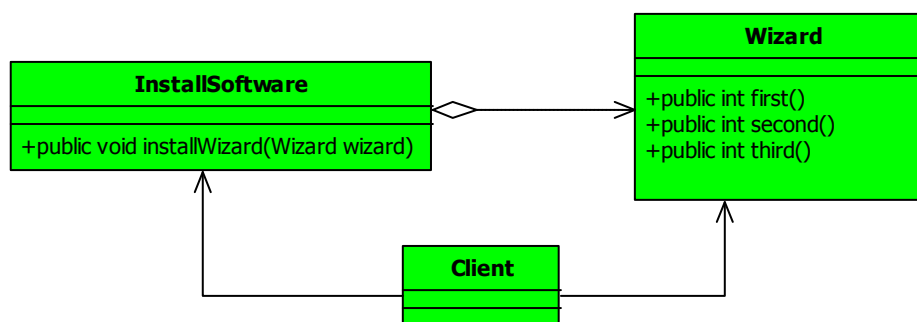
/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 老师类
 */
public class Teacher {

    //老师对学生发布命令,清一下女生
    public void command(GroupLeader groupLeader){
        //告诉体育委员开始执行清查任务
        groupLeader.countGirls();
    }
}
```

```
}
```

程序做了一个简单的修改，就是把 Teacher 中的对 List<Girl>初始化（这个是有业务意义的，产生出全班的所有人员）移动到了 GroupLeader 的 countGrils 方法中，避开了 Teacher 类对陌生类 Girl 的访问，减少系统间的耦合。记住了，一个类只和朋友交流，不与陌生类交流，不要出现 getA().getB().getC().getD() 这种情况（在一种极端的情况下是允许出现这种访问：每一个点号后面的返回类型都相同），那当然还要和 JDK API 提供的类交流，否则你想脱离编译器存在呀！

朋友间也是有距离的。人和人之间是有距离的，太远就不是朋友了，太近就浑身不自在，这和类间关系也是一样，即使朋友类也不能无话不说，无所不知。大家在项目中应该都碰到过这样的需求：调用一个类，然后必须是先执行第一个方法，然后是第二个方法，根据返回结果再来看是否可以调用第三个方法，或者第四个方法等等，我们用类图表示一下：



很简单的类图，实现软件安装过程的第一步做什么、第二步做什么、第三步做什么这样一个过程，我们来看三个类的源代码，先看 Wizard 的源代码：

```

package com.cbf4life.common3;

import java.util.Random;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 按照步骤执行的业务逻辑类
 */
public class Wizard {
    private Random rand = new Random(System.currentTimeMillis());
    //第一步
  
```



```
public int first(){
    System.out.println("执行第一个方法...");
    return rand.nextInt(100);
}

//第二步
public int second(){
    System.out.println("执行第二个方法...");
    return rand.nextInt(100);
}

//第三个方法
public int third(){
    System.out.println("执行第三个方法...");
    return rand.nextInt(100);
}
}
```

分别定义了三个步骤方法，每个步骤中都有相关的业务逻辑完成指定的任务，我们使用一个随机函数来代替业务执行的返回值。再来看软件安装过程 InstallSoftware 源码：

```
package com.cbf4life.common3;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 业务组装类，负责调用各个步骤
 */
public class InstallSoftware {

    public void installWizard(Wizard wizard){
        int first = wizard.first();
        //根据first返回的结果，看是否需要执行second
        if(first>50){
            int second = wizard.second();
            if(second>50){
                int third = wizard.third();
                if(third >50){
                    wizard.first();
                }
            }
        }
    }
}
```

```

    }

    }
}

```

其中 installWizard 就是一个向导式的安装步骤，我们看场景是怎么调用的：

```

package com.cbf4life.common3;

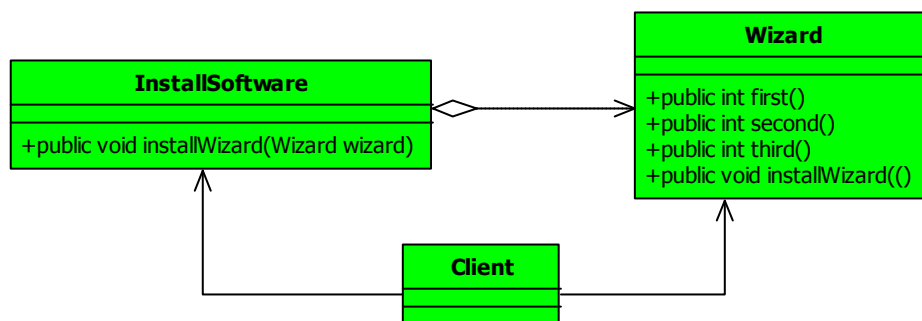
/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 业务场景
 */
public class Client {

    public static void main(String[] args) {
        InstallSoftware invoker = new InstallSoftware();
        invoker.installWizard(new Wizard());
    }
}

```

这个程序很简单，运行结果和随机数有关，我就不粘贴上来了。我们想想这个程序有什么问题吗？

Wizard 类把太多的方法暴露给 InstallSoftware 类了，这样耦合关系就非常紧了，我想修改一个方法的返回值，本来是 int 的，现在修改为 boolean，你看就需要修改其他的类，这样的耦合是极度不合适的，**迪米特法则**就要求类“小气”一点，尽量不要对外公布太多的 public 方法和非静态的 public 变量，尽量内敛，多使用 private、package-private、protected 等访问权限。我们来修改一下类图：



我们再来看一下程序的变更，先看 Wizard 程序：

```

package com.cbf4life.common4;

import java.util.Random;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 按照步骤执行的业务逻辑类
 */
public class Wizard {
    private Random rand = new Random(System.currentTimeMillis());

    //第一步
    private int first(){
        System.out.println("执行第一个方法...");
        return rand.nextInt(100);
    }

    //第二步
    private int second(){
        System.out.println("执行第二个方法...");
        return rand.nextInt(100);
    }

    //第三个方法
    private int third(){
        System.out.println("执行第三个方法...");
        return rand.nextInt(100);
    }

    //软件安装过程
    public void installWizard(){
        int first = this.first();
        //根据first返回的结果，看是否需要执行second
        if(first>50){
            int second = this.second();
            if(second>50){
                int third = this.third();
                if(third >50){
                    this.first();
                }
            }
        }
    }
}

```

```
}
```

三个步骤的访问权限修改为 private，同时把 installWizard 移动到 Wizard 方法中，这样 Wizard 类就对外只公布了一个 public 方法，类的高内聚特定显示出来了。我们再来看 InstallSoftware 源码：

```
package com.cbf4life.common4;

/**
 * @author cbf4Life cbf4life@126.com
 * I'm glad to share my knowledge with you all.
 * 业务组装类，负责调用各个步骤
 */
public class InstallSoftware {

    public void installWizard(Wizard wizard){
        //不废话，直接调用
        wizard.installWizard();
    }
}
```

Client 类没有任何改变，就不在拷贝了，这样我们的程序就做到了弱耦合，一个类公布越多的 public 属性或方法，修改的涉及面也就越大，也就是变更引起的风险就越大。因此为了保持朋友类间的距离，你需要做的是：减少 public 方法，多使用 private、package-private（这个就是包类型，在类、方法、变量前不加访问权限，则默认为包类型）protected 等访问权限，减少非 static 的 public 属性，如果成员变量或方法能加上 final 关键字就加上，不要让外部去改变它。

是自己的就是自己的。在项目中有一些方法，放在本类中也可以，放在其他类中也没有错误，那怎么去衡量呢？你可以坚持这样一个原则：**如果一个方法放在本类中，即不增加类间关系，也对本类不产生负面影响，就放置在本类中。**

谨慎使用 Serializable。实话说，这个问题会很少出现的，即使出现也会马上发现问题。是怎么回事呢？举个例子来说，如果你使用 RMI 的方式传递一个对象 VO（Value Object），这个对象就必须使用 Serializable 接口，也就是把你的这个对象进行序列化，然后进行网络传输。突然有一天，客户端的 VO 对象修改了一个属性的访问权限，从 private 变更为 public 了，如果服务器上没有做出响应的变更的话，就会报序列化失败。这个应该属于项目管理范畴，一个类或接口客户端变更了，而服务端没有变更，那像话吗？！

迪米特法则的核心观念就是类间解耦，弱耦合，只有弱耦合了以后，类的复用率才可以提高，其要求的结果就是产生了大量的中转或跳转类，类只能和朋友交流，朋友少了你业务跑不起来，朋友多了，你项目管理就复杂，大家在使用的时候做相互权衡吧。

不知道大家有没有听过这样一个理论：“任何 2 个素不相识的人中间最多只隔着 6 个人，即只用 6 个人就可以将他们联系在一起”，这个理论的学名叫做“六度分离”，应用到我们项目中就是说我和我要调用的类之间最多有 6 次传递，呵呵，这只能让大家当个乐子来看，在实际项目中你跳两次才访问到一个类估计你就会想办法了，这也是合理的，迪米特法则要求我们类间解耦，但是解耦是有限度的，除非是计算机的最小符号二进制的 0 和 1，那才是完全解耦，我们在实际的项目中时，需要适度的考虑这个法则，别为了套用法则而做项目，法则只是一个参考，你跳出了这个法则，也不会有人判你刑，项目也未必会失败，这就需要大家使用的是考虑如何度量法则了。

26.6 开闭原则【Open Close Principle】

在哲学上矛盾法则，即对立统一的法则，是唯物辩证法的最根本的法则，我们伟大的毛泽东同志在 1937 就出版了《矛盾论》来阐述矛盾的普遍性、特殊性、及主要矛盾和矛盾的主要方面等等，我们今天要讲的开闭原则是不是也有同样的重要性和普遍性呢？确实是，开闭原则是 Java 世界里一个最基础的设计原则，它指导我们如何建立一个稳定、灵活的系统，先来看开闭原则的定义：

Software entities like classes, modules and functions should be open for extension but closed for modifications. 一个软件实体应该对扩展开放，对修改关闭。

初看到这个定义，会很迷惑，对扩展开放？开放什么？对修改关闭，怎么关闭？我会一步一带领大家解释这些疑惑。

我们做一件事情，或者选择一个方向，一般需要经历三个步骤：What——是什么，Why——为什么，How——怎么做（简称 3W 原则，How 取最后一个 w），对于开闭原则，我们也采用这三步骤来分析，即什么是开闭原则，为什么要使用开闭原则，怎么使用开闭原则。

什么是开闭原则

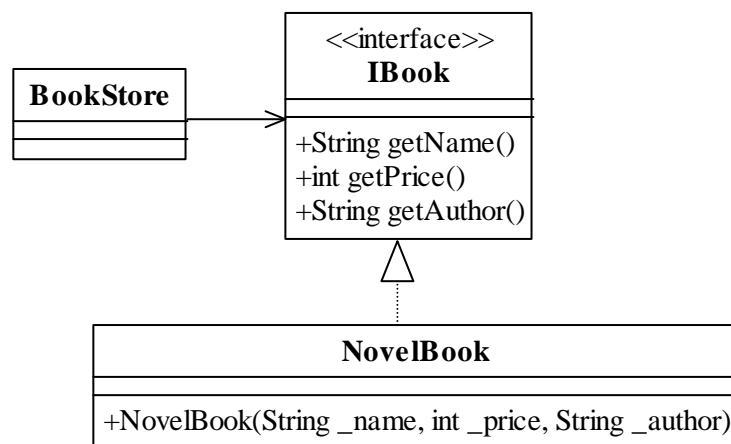
开闭原则的定义已经非常明确告诉我们：软件实体应该对扩展开放，对修改关闭，其含义是说一个软件实体应该通过扩展来实现变化，而不是通过修改已有的代码来实现变化。那什么又是软件实体呢？软实体包括以下部分：

项目或软件产品中按照一个的逻辑规则划分的模块

抽象或类

方法

我们思考这样一个问题：一个软件产品只要在生命期内，都会发生变化，变化既然是一个既定的事实，我们就应该在设计时候尽量适应这些变化，以提高项目的稳定性和灵活性，真正实现“拥抱变化”，开闭原则告诉我们通过尽量通过扩展软件实体的行为来实现变化，而不通过修改来已有的代码来完成变化，它是为软件实体的未来事件而制定的对现行开发设计进行约束的一个原则，我们举例什么是开闭原则，以书店销售书籍为例，类图如下：



IBook 是定义了数据的三个属性：名称、价格和作者，小说类 NovelBook 是一个具体的实现类，所有小说书籍的总称，BookStore 指的是书店，我们先来看 IBook 接口：

```
public interface IBook {

    //书籍有名称
    public String getName();

    //书籍有售价
    public int getPrice();

    //书籍有作者
    public String getAuthor();
}
```

小说书籍的源代码如下：

```
public class NovelBook implements IBook {
    //书籍名称
    private String name;

    //书籍的价格
    private int price;

    //书籍的作者
    private String author;

    //通过构造函数传递书籍数据
```

```

public NovelBook(String _name,int _price,String _author){
    this.name = _name;
    this.price = _price;
    this.author = _author;
}

//获得作者是谁
public String getAuthor() {
    return this.author;
}

//书籍叫什么名字
public String getName() {
    return this.name;
}

//获得书籍的价格
public int getPrice() {
    return this.price;
}
}

```

然后我们看书店是怎么销售书籍的：

```

public class BookStore {
    private final static ArrayList<IBook> bookList = new ArrayList<IBook>();

    //静态模块初始化，项目中一般是从持久层初始化产生
    static{
        bookList.add(new NovelBook("天龙八部",3200,"金庸"));
        bookList.add(new NovelBook("巴黎圣母院",5600,"雨果"));
        bookList.add(new NovelBook("悲惨世界",3500,"雨果"));
        bookList.add(new NovelBook("金瓶梅",4300,"兰陵笑笑生"));
    }

    //模拟书店买书
    public static void main(String[] args) {
        NumberFormat formatter = NumberFormat.getCurrencyInstance();
        formatter.setMaximumFractionDigits(2);
        System.out.println("-----书店买出去的书籍记录如下：
-----");
        for(IBook book:bookList){

```



```
        System.out.println("书籍名称: " + book.getName()+"\t书籍作者: " +  
book.getAuthor()+ "\t书籍价格: " + formatter.format(book.getPrice()/100.0)+"元");  
    }  
}  
}
```

注意，我们在 BookStore 中声明了一个静态模块，实现了数据的初始化，这部分应该是从持久层产生的，由持久层工具进行管理。运行结果如下：

-----书店买出去的书籍记录如下：-----

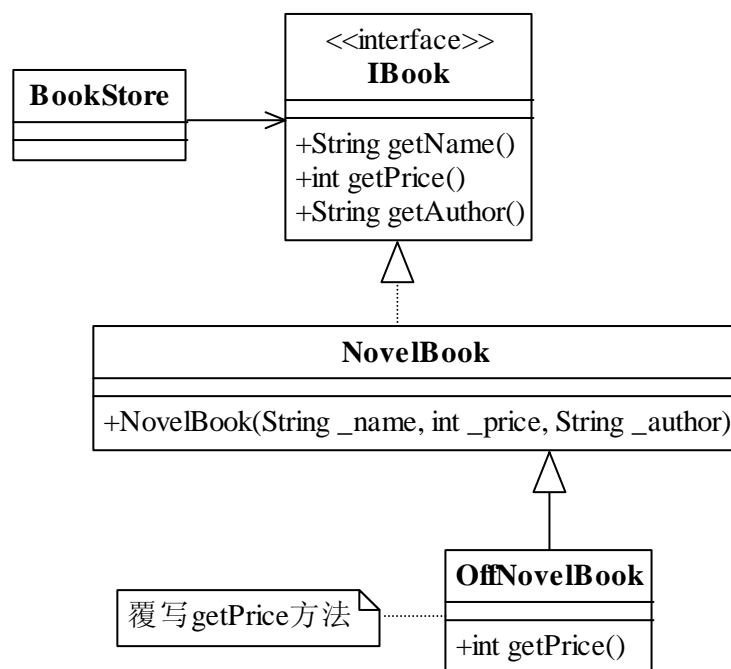
书籍名称：天龙八部 书籍作者：金庸 书籍价格：¥32.00元
书籍名称：巴黎圣母院 书籍作者：雨果 书籍价格：¥56.00元
书籍名称：悲惨世界 书籍作者：雨果 书籍价格：¥35.00元
书籍名称：金瓶梅 书籍作者：兰陵笑笑生 书籍价格：¥43.00元

项目投产了，书籍正常销售出去，书店也盈利了。从 2008 年开始，全球经济都开始下滑，对零售业影响还是比较大，书店为了生存开始打折销售：所有 40 元以上的书籍 9 折销售，其他的 8 折销售。对已经投产的项目来说，这就是一个变化，我们来看看这样的一个需求变化，我们该怎么去应对，有三种方法可以解决这个问题：

修改接口。在 IBook 上新增加一个方法 getOffPrice()，专门进行打折处理，所有的实现类实现该方法。但是这样修改的后果就是实现类 NovelBook 要修改，BookStore 中的 main 方法也修改，同时 IBook 作为接口应该是稳定且可靠的，不应该经常发生变化，否则接口做为契约的作用就失去了效能，——因此，该方案否定。

修改实现类。修改 NovelBook 类中的方法，直接在 getPrice() 中实现打折处理，好办法，我相信大家在项目中经常使用的就是这样办法，通过 class 文件替换的方式可以完成部分业务（或是缺陷修复）变化，该方法在项目有明确的章程（团队内约束）或优良的架构设计时，是一个非常优秀的方法，但是该方法还是有缺陷的，例如采购书籍人员也是要看价格的，由于该方法已经实现了打折处理价格，因此采购人员看到的也是打折后的价格，这就产生了信息的蒙蔽效果，导致信息不对称而出现决策失误的情况。——因此，该方案也不是一个最优的方案。

通过扩展实现变化。增加一个子类 OffNovelBook，覆写 getPrice 方法，高层次的模块（也就是 static 静态模块区）通过 OffNovelBook 类产生新的对象，完成对业务变化开发任务。——好办法，修改也少，风险也小，我们来看类图：



OffNovelBook 类继承了 NovelBook，并覆写了 getPrice 方法，不修改原有的代码。我们来看新增加
子类 OffNovelBook:

```

public class OffNovelBook extends NovelBook {
    public OffNovelBook(String _name, int _price, String _author) {
        super(_name, _price, _author);
    }

    //覆写销售价格
    @Override
    public int getPrice() {
        //原价
        int selfPrice = super.getPrice();
        int offPrice = 0;
        if (selfPrice > 4000) { //原价大于40元，则打9折
            offPrice = selfPrice * 90 / 100;
        } else {
            offPrice = selfPrice * 80 / 100;
        }

        return offPrice;
    }
}

```

很简单，仅仅覆写了 getPrice 方法，通过扩展完成了新增加的业务。然后我们来看 BookStore 类的修改：

```
public class BookStore {
    private final static ArrayList<IBook> bookList = new ArrayList<IBook>();

    //静态模块初始化，项目中一般是从持久层初始化产生
    static{
        bookList.add(new OffNovelBook("天龙八部",3200,"金庸"));
        bookList.add(new OffNovelBook("巴黎圣母院",5600,"雨果"));
        bookList.add(new OffNovelBook("悲惨世界",3500,"雨果"));
        bookList.add(new OffNovelBook("金瓶梅",4300,"兰陵笑笑生"));
    }

    //模拟书店买书
    public static void main(String[] args) {
        NumberFormat formatter = NumberFormat.getCurrencyInstance();
        formatter.setMaximumFractionDigits(2);
        System.out.println("-----书店买出去的书籍记录如下：
        -----");
        for(IBook book:bookList){
            System.out.println("书籍名称：" + book.getName()+"\t书籍作者：" +
            book.getAuthor()+"\t书籍价格：" + formatter.format(book.getPrice()/100.0)+"元");
        }
    }
}
```

我们只修改了黄色部分，其他的部分没有任何改动，看运行结果：

```
-----书店买出去的书籍记录如下：-----
书籍名称：天龙八部   书籍作者：金庸   书籍价格：¥25.60元
书籍名称：巴黎圣母院 书籍作者：雨果   书籍价格：¥50.40元
书籍名称：悲惨世界   书籍作者：雨果   书籍价格：¥28.00元
书籍名称：金瓶梅 书籍作者：兰陵笑笑生 书籍价格：¥38.70元
```

OK，搞定，打折销售开发完成了。看到这里，各位可能有想法了：增加了一个 OffNoveBook 类后，你的业务逻辑还是修改了，你修改了 static 静态模块区域，这部分确实修改了，该部分属于高层次的模块，是由持久层产生的，在业务规则改变的情况下高层模块必须有部分改变以适应新业务，改变时尽量少的，压制变化风险的扩散。注意：开闭原则说是对扩展开放，对修改关闭，并不意味着不做任何的修改，我们

可以把变化归纳为以下几个类型：

逻辑变化。只变化一个逻辑，而不涉及到其他模块，比如原有的一个算法是 $a*b+c$ ，现在需要修改为 $a*b*c$ ，可以通过修改原有的类中的方法方式来完成，前提条件是所有依赖或关联类都按照相同的逻辑处理。

子模块变化。一个模块变化，会对其他的模块产生影响，特别是一个低层次的模块变化必然引起高层模块的变化，因此在通过扩展完成变化时，高层次的模块修改是必然的，刚刚的书籍打折处理就是类似的处理模块，该部分的变化甚至会引起界面的变化。

可见视图变化。可见视图是提供给客户使用的界面，如 jsp 程序，swing 界面等，该部分的变化一般会引起连锁反应（特别是在国内做项目，做欧美的外包项目一般不会影响太大），如果仅仅是界面上按钮、文字的重新排布倒是简单，最司空见惯的是业务耦合变化，什么意思呢？一个展示数据的列表，按照原有的需求是六列，突然有一天要增加一列，而且这一列要跨度 N 张表，处理 M 个逻辑才能展现出来，这样的变化是比较恐怖的，但是我们还是可以通过扩展来完成变化，这就依赖我们原有的设计是否灵活。

我们再来回顾一下书店销售书籍的程序，首先是我们有一个还算灵活的设计（不灵活是什么样子？BookStore 中所有使用到 IBook 的地方全部修改为实现类，然后再扩展一个 ComputerBook 书籍，你就知道什么是不灵活了），然后有一个需求变化，然后通过扩展一个子类拥抱了变化，然后把子类投入运行环境中，新逻辑正式投产。通过分析，我们发现我们并没有修改原有的模块代码，IBook 接口没有改变，NovelBook 类没有改变，这属于已有的业务代码，我们保持了历史的纯洁性。放弃修改历史的想法吧，一个项目的基本路径应该是这样的项目开发、重构、测试、投产、运维，其中的重构可以对原有的设计和代码进行修改，运维尽量减少对原有代码的修改，保持历史代码的纯洁性，提高系统得到稳定性。

为什么要使用开闭原则

每个事物的诞生都有它存在的必要性，存在即合理，那我们的开闭原则的存在也是合理的，为什么这么说呢？

首先，开闭原则是那么的著名，只要是做面向对象编程的，甭管是什么语言，Java 也好 C++ 也好或者是 Smalltalk，在做开发时都会提及开闭原则，如果你是个架构师，没有听说过开闭原则，那你绝对可以闭口气把自己憋死，如果你是一个 Java 程序员，没有听说过开闭原则，那你最好喝口水把自己呛死。

其次，开闭原则是最基础的一个原则，前边五个章节介绍的原则（单一职责原则、里氏替换原则、依赖倒置原则、接口隔离原则、迪米特法则）都是开闭原则的具体形态，也就是说前五个原则就是指导设计的工具和方法，而开闭原则才是其精神领袖，换一个角度来理解，依照 Java 语言的称谓，开闭原则是抽象类，其他五大原则则是具体的实现类，开闭原则在面向对象设计领域中的地位就类似于的牛顿第一定律在力学界、勾股定律在几何学、质能方程在狭义相对论中的地位，其地位无人能及。。

最后，开闭原则是非常重要的，通过以下几个方面来理解其重要性：

开闭原则对测试的影响。每个已经投产了的代码都是有意义的，并且都受系统的规则约束着，这样的代码都是经过“千锤百炼”的测试过程，不仅保证逻辑是正确的，还要保证苛刻条件（高压力、异常、错误）下的不产生“有毒代码”（Poisonous Code），因此有变化提出时，我们就需要考虑一下，原有健壮的代码是否可以不修改，仅仅通过扩展开实现变化呢？否则，就需要把原有的测试过程回笼一遍，需要进行单元测试、功能测试、集成测试，甚至是验收测试，现在虽然在大力提倡自动化测试工具，但是仍然代替不了人工的测试工作。

以上面我们提到的书店售书为例，IBook 接口写完了，实现类 NovelBook 也写好，我们需要写一个测试类进行测试，测试类源代码如下：

```
public class NovelBookTest extends TestCase {
    private String name = "平凡的世界";
    private int price = 6000;
    private String author = "路遥";

    private IBook novelBook = new NovelBook(name, price, author);

    //测试getPrice方法
    public void testGetPrice() {
        //原价销售，判断输入和输出的值是否相等进行断言
        super.assertEquals(this.price, this.novelBook.getPrice());
    }
}
```

单元测试通过，显示绿条，当然是绿条了，这么简单的逻辑再不是绿条，那就找块豆腐把自己撞死。在单元测试中，有一句非常有名的话，叫做“Keep the bar green to keep the code clean”，保持绿条有利于代码整洁，这是什么意思呢？绿条就是 Junit 运行的两种结果中的一种：要么是红条，单元测试失败；要么是绿条，单元测试通过。一个方法的测试方法一般不少于 3 个，为什么呢？首先是正常的业务逻辑要保证测试到，其次是边界条件要测试到，然后是异常要测试到，比较重要的方法的测试方法甚至有十多个，而且单元测试是对类的测试，类内的方法耦合是允许的，在这这样的条件下，如果再想着通过修改一个方法或多个方法代码来完成变化，基本上就是痴人说梦，该类的所有测试方法都要重构，想象一下你在一堆你并不熟悉的代码中进行重构时的感觉吧！

在书店售书的例子中，增加了一个打折销售的需求，如果我们直接修改 getPrice 方法，来实现业务需

求的变化，那我们就要修改单元测试类，想想看我们举这个例子是非常简单的，如果是一个复杂的逻辑，你的测试类就要修改的面目全非，还有，在实际的项目中一般一个类只有一个测试类，其中可以有很多的测试方法，在一堆本来就很复杂的断言中进行大量修改，难免就会出现测试遗漏情况，这是一个项目经理很难容忍的事情。

所以，我们需要通过扩展来实现业务逻辑的变化，而不是修改。上面的例子中通过增加一个子类 OffNovelBook 来完成了业务需求的变化，对测试有什么好处呢？我们重新生成一个测试文件 OffNovelBookTest，然后对 getPrice 进行测试，单元测试是孤立测试，我只要保证我提供的方法正确就成了，其他的我不管，这不是单元测试的范畴。OK，源代码如下：

```
public class OffNovelBookTest extends TestCase {

    private IBook below40NovelBook = new OffNovelBook("平凡的世界", 3000, "路遥");
    private IBook above40NovelBook = new OffNovelBook("平凡的世界", 6000, "路遥");

    //测试低于40元的数据是否是打8折
    public void testGetPriceBelow40() {
        super.assertEquals(2400, this.below40NovelBook.getPrice());
    }

    //测试大于40的书籍是否是打9折
    public void testGetPriceAbove40(){
        super.assertEquals(5400, this.above40NovelBook.getPrice());
    }

}
```

新增加的类，新增加的测试方法，只要保证新增加类就是正确的就可以了。

开闭原则可以提高复用性。在面向对象的设计中，我们所有的逻辑都是从原子逻辑组合而来的，而不是在一个类中独立实现一个业务逻辑，只要这样代码才可以复用，粒度越小，被复用的可能性就越大。那为什么要复用呢？较少代码量，避免相同的逻辑分散在多个角落，避免日后的维护人员为了修改一个微小的缺陷或新增加新功能，而要在整个项目中到处找相关的代码，然后发出对开发人员“极度失望”的感慨。那怎么才能提高复用率呢？缩小逻辑粒度，直到一个逻辑不可再拆分为止。

开闭原则可以提高可维护性。一个软件投产后，维护人员的工作不仅仅是对数据进行维护，还可能对程序进行扩展，那维护人员最乐意做的事情，就是扩展一个类，而不是修改一个类，甭管原有的代码写的

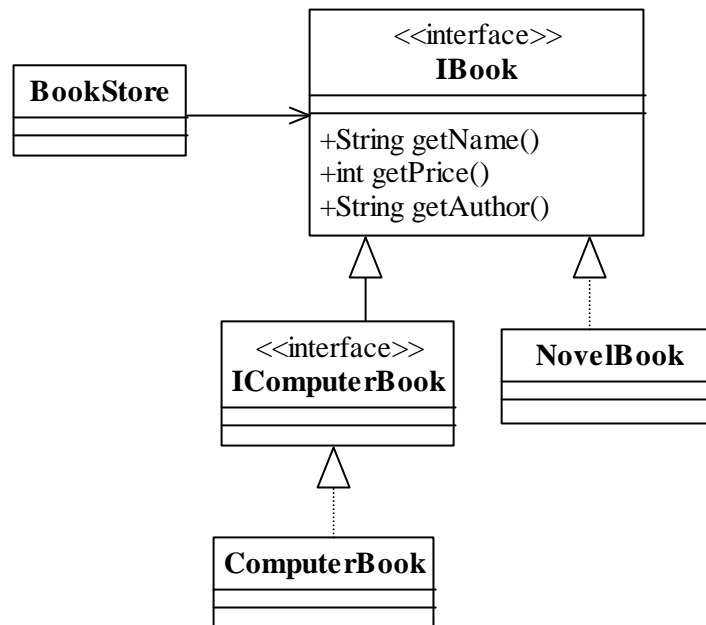
多么优秀还是写的多么糟糕，让维护人员读懂原有的代码，然后再修改是一件很痛苦的事情，不要让他原有的代码海洋里徜徉完毕后再修改，这是对维护人员的一种折磨和摧残。

面向对象开发的要求。万物皆对象，我们需要把所有的事物都抽象成对象，然后针对对象进行操作，但是万物皆运动，有运动就有变化，有变化就要有策略去应对，怎么快速的应对？就需要在设计之初考虑到所有可能变化的因素，然后留下接口，等待可变因素转变真正的变化时轻松应对。

怎么使用开闭原则

开闭原则是一个非常虚的原则，前边五个原则是对开闭原则的具体解释，但是开闭原则并不局限于这么多，它“虚”的没有边界，就像“好好学习，天天向上”的口号一样，告诉我们要好好学习，但是学什么，怎么学并没有告诉我们，需要去体会去掌握，开闭原则也是一个口号，那我们怎么把这个口号应用到我们实际的工作中呢？

抽象约束。抽象是对一组事物的通用描述，没有具体的实现，也就表示它可以有非常多可能性，可以跟随需求的变化而变化，因此通过接口或抽象类可以约束一组行为，并且能够实现扩展开放，其包含三层含义：一是通过接口或抽象类约束扩展，对扩展进行边界限定，不允许出现在接口或抽象类中不存在的 public 方法；二是在参数类型定义、输入输出参数尽量使用接口或者抽象类，而不是实现类，三是抽象层尽量保持稳定，一旦确定即不允许修改。还是以书店为例，目前只是销售小说书籍，单一经营毕竟是有风险的，于是书店新增加了一种书籍：计算机书籍，它不仅仅包含书籍名称、作者、价格等信息，还有一个独特的属性：面向的是什么领域，也就是它的范围(scope)，比如是讲语言的，数据库，还是硬件等等，我们先来看类图的修改：



增加了一个接口 IComputerBook 和实现类 ComputerBook，而 BookStore 不用做任何修改就可以完成书店销售计算机书籍的业务，我们来看源代码：

```
public interface IComputerBook extends IBook{

    //计算机书籍是有一个范围
    public String getScope();
}
```

很简单，计算机数据增加了一个方法，就是获得该书籍的范围，同时继承 IBook 接口，毕竟计算机书籍也是书籍。其实现类如下：

```
public class ComputerBook implements IComputerBook {
    private String name;
    private String scope;
    private String author;
    private int price;

    public ComputerBook(String _name,int _price,String _author,String _scope){
        this.name=_name;
        this.price = _price;
        this.author = _author;
        this.scope = _scope;
    }
}
```



```

    public String getScope() {
        return this.scope;
    }

    public String getAuthor() {
        return this.author;
    }

    public String getName() {
        return this.name;
    }

    public int getPrice() {
        return this.price;
    }
}

```

也很简单，实现 IComputerBook 就可以，而 BookStore 类没有做任何修改，只是在 static 静态模块中增加一条数据，代码如下：

```

public class BookStore {
    private final static ArrayList<IBook> bookList = new ArrayList<IBook>();

    //静态模块初始化，项目中一般是从持久层初始化产生
    static{
        bookList.add(new OffNovelBook("天龙八部",3200,"金庸"));
        bookList.add(new OffNovelBook("巴黎圣母院",5600,"雨果"));
        bookList.add(new OffNovelBook("悲惨世界",3500,"雨果"));
        bookList.add(new OffNovelBook("金瓶梅",4300,"兰陵笑笑生"));
        //增加计算机书籍
        bookList.add(new ComputerBook("Think in Java",4300,"Bruce Eckel","编程语
言"));
    }

    //模拟书店买书
    public static void main(String[] args) {
        NumberFormat formatter = NumberFormat.getCurrencyInstance();
        formatter.setMaximumFractionDigits(2);
        System.out.println("-----书店买出去的书籍记录如下：
-----");
    }
}

```

```

        for (IBook book: bookList) {
            System.out.println("书籍名称: " + book.getName() + "\t书籍作者: " +
                book.getAuthor() + "\t书籍价格: " + formatter.format(book.getPrice()/100.0) + "元");
        }
    }
}

```

运行结果如下：

-----书店买出去的书籍记录如下：-----

书籍名称：天龙八部 书籍作者：金庸 书籍价格：¥32.00元

书籍名称：巴黎圣母院 书籍作者：雨果 书籍价格：¥56.00元

书籍名称：悲惨世界 书籍作者：雨果 书籍价格：¥35.00元

书籍名称：金瓶梅 书籍作者：兰陵笑笑生 书籍价格：¥43.00元

书籍名称：Think in Java 书籍作者：Bruce Eckel 书籍价格：¥43.00元

如果我是做维护的，我就非常乐意做这样的事情，简单而且不需要与其他的业务做耦合，我只需要做的事情就是在原有的代码上进行添砖加瓦，就可以实现业务的变化。我们来看看这段代码给我们带来了几层意思。

首先，ComputerBook 类必须实现 IBook 的三个方法，是通过 IComputerBook 接口传递进来的约束，也就是我们制定的 IBook 接口对扩展类 ComputerBook 产生了约束力，这个约束力同时对我们扩展提供非常好的帮助，否则 BookStore 类就需要进行大量的修改了。

其次，如果我们原有的程序设计时采用的不是接口，而是实现类，那会出现什么问题呢？我们把 BookStore 类中的私有变量 bookList 修改为：

```
private final static ArrayList<NovelBook> bookList = new ArrayList<NovelBook>();
```

把原有 IBook 的依赖修改为对 NovelBook 实现类的依赖，想想看我们这次的扩展是否还能继续下去呢？一旦这样设计，我们就根本没有办法扩展，需要修改原有的业务逻辑（也就是 main 方法），这样的扩展基本上就是形同虚设。

最后，如果我们在 IBook 上增加一个方法 getScope 是否可以呢？答案是不可以，因为原有的实现类 NovelBook 已经在投产运行中，它不需要改方法，而且接口是与其他模块交流的契约，修改契约就等于让其他模块修改。因此，我们的接口或抽象类一旦定义，就应该立即执行，不能有修改接口的思想，除非是彻底的大返工。

所以，要实现对扩展开放，首要的前提条件就是：抽象约束。

参数控制模块行为。程序员是一个很苦很累的活，那怎么才能减轻我们的压力呢？答案是尽量是参数

来控制我们的程序的行为，减少重复开发。参数可以从文件中获得，也可以从数据库中获得，举个非常简单的例子，login 方法中提供了这样的逻辑：先检查 IP 地址是否在允许访问的列表中，然后再决定是否需要到数据库中验证密码（如果采用 SSH 架构，则可以通过 Struts 的拦截器来实现），该行为就是一个典型的参数控制模块行为的例子，其中达到极致的就是控制翻转（Inversion of Control），使用最多的就是 Spring 容器，在 SpringContext 配置文件中，有这样一段配置：

```
<bean id="father" class="xxx.xxx.xxx.Father" />
<bean id="xx" class="xxx.xxx.xxx.xxx">
    <property name="biz" ref="father"></property>
</bean>
```

然后，通过建立一个 Father 类的子类 Son，完成一个新的业务，修改一下配置文件：

```
<bean id="son" class="xxx.xxx.xxx.Son" />
<bean id="xx" class="xxx.xxx.xxx.xxx">
    <property name="biz" ref="son"></property>
</bean>
```

通过扩展一个子类，修改配置文件，完成了业务变化，这也是采用框架的好处。

制定项目章程。在一个团队中，项目章程的建立是非常重要的，因为在章程中指定了所有人员都必须遵守的约束，而对项目来说约定是优于配置。相信大家都做过项目，会发现一个项目会产生非常多的配置文件，举个简单的例子，以 SSH 项目开发为例，一个项目中的 Bean 配置文件是非常多的，管理非常麻烦，如果需要扩展就需要增加子类，并修改 SpringContext 文件，而如果你在项目中指定这样一个章程：所有的 Bean 都自动注入，使用 Annotation 进行装配，进行扩展时，甚至只用写一个子类，然后由持久层生成对象，其他的都不需要修改，这就需要项目内约束，每个项目成员都必须遵守，该方法需要一个团队有较高的自觉性，需要一个较长时间的磨合，一旦项目成员都熟悉这样的规则，比通过接口或抽象类进行约束效率更高，而且扩展性一点也没有减少。

封装变化。对变化的封装包含两层含义：一是对相同的变化封装到一个接口或抽象类中，二是对不同的变化封装到不同的接口或抽象类中，不应该出现两个不同的变化出现同一个接口或抽象类中。封装变化，准确的讲就是封装可能发生的变化，一旦预测到或“第六感”发觉有变化，就可以进行封装，23 个设计模式都是从各个不同的角度对变化进行封装，我们会在各个模式中逐步讲解。

最佳实践

软件设计最大的难题就是如何应对需求的变化，但是缤纷复杂的需求变化又是不可预料的，我们要为

不可预料的事情在当前做好准备，这本身就是一个非常痛苦的事情，但是大师们还是给我们提出非常好的六大设计原则以及 23 个设计模式来封装未来的变化，我们在前五章中讲过如下设计原则：

Single Responsibility Principle 单一职责原则

Open Closed Principle 开闭原则

Liskov Substitution Principle 里氏替换原则

Low Of Demete 迪米特法则

Interface Segregation Principle 接口隔离原则

Dependence Inversion Principle 依赖倒置原则

把这六个原则的首字母联合起来就是 S.O.L.I.D (solid, 稳定的)，其代表的含义也就是把这六个原则结合使用的好处：建立稳定、灵活、健壮的设计，而开闭原则又是重中之重，是最基础的原则，是其他五大原则的精神领袖。我们在使用开闭原则要注意的几个问题：

开闭原则也只是一个原则。开闭原则只是精神口号，实现拥抱变化的方法非常多，并不局限于这六大设计原则，但是遵循这六大设计原则可以基本上应对大部分变化。因此，我们在项目中尽量采用这六大原则，适当时候可以进行扩充，例如通过类文件替换的方式完全可以解决系统中的一些缺陷，大家如果做过软件产品开发，比较常用的修复缺陷方法就是类替换，比如一个软件产品已经在运行中，发现了一个缺陷，需要修正怎么办？如果有自动更新(Auto Upgrade)功能，则可以下载一个.class 文件直接覆盖原有的 class，重新启动应用（也不一定非要重新启动）就可以解决问题，也就是通过类文件的替换方式修正了一个缺陷，当然这种方式也可以应用到项目中，正在运行中的项目开发需要增加一个新功能，通过修改原有实现类的方式就可以解决这个问题，前提条件是：类必须做到高内聚低耦合，否则类文件的替换会引起意想不到的故障。

项目规章非常重要。如果你是一个项目经理或者架构师，尽量能让自己的项目成员稳定，稳定后才能建立高效的团队文化，章程是一个团队所有成员共同的知识结晶，也是所有成员必须遵守的规章制度。一个优秀的章程能带给项目带来非常多的优点，如提高开发效率，降低缺陷率，提高团队士气，提高技术成员水平等等。

在实践中实施时，架构师或项目经理一旦发现有变化的可能，或者变化曾经演绎过，则需要考虑现有的架构中是否可以轻松的实现这一变化，架构师设计一套系统不是仅仅符合现有的需求，还要适应可能发生的变化，这才是一个优良的架构。

开闭原则是一个终极目标，任何人包括大师级人物都是无法百分之百的做到的，但朝这个方向努力，可以非常显著改善一个系统的架构，做到“拥抱变化”。

第 27 章 混编模式讲解

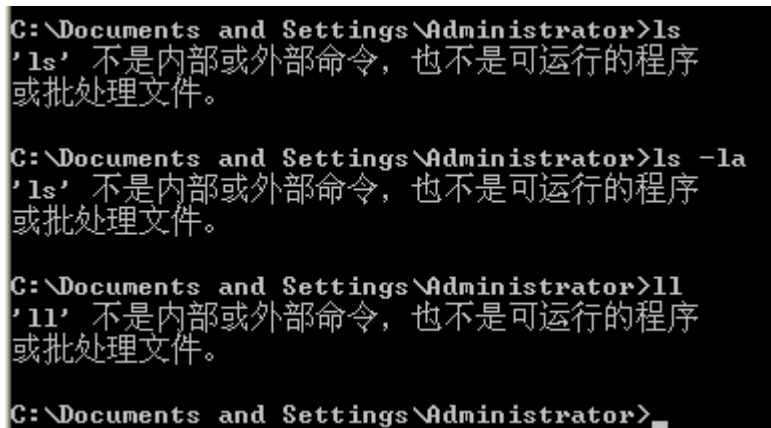
混编模式这一章节我也是思考了很久，模式的混编必须和实际的项目相关联，采用混合模式可以扬长避短，提高项目的稳定性、扩展性、可靠性等等，就是一句话：少折腾！在客户变更了需求，你不因为工作量暴增而沮丧；在业务临时爆发时，你不用心惊胆战的盯着看服务器的资源情况；在系统发现 BUG 的时候，你不用皱着眉头思考怎么处理淆乱的数据；这些我相信你只要做个中型项目就肯定遇到过，而且有些问题还是灾难性的。

我这一章节的讲解尽量通过引入设计模式来完善我们的系统，本来思考了有两个讲解思路（实话说，以前讲课，混编模式我是不讲的，这个不是一节两节课能够讲明白，你让听众稀里糊涂的还不如不讲）：一是根据我自己的经验，把我自己做过的项目引入进来，通过亲身体验来讲解，这样效果可能会，但是也是有缺点的，因为每个项目都有背景信息，为什么要这样设计必须把背景信息（业务需求）讲清楚，这可能要花比较长的篇幅；另外一种是根据现在一些开源项目来讲解，这些开源项目都是大师的杰作，比如 Struts，Ibatis 等这些源码就用到了很多模式，而且有些代码非常经典，经典的会让你拍案叫绝，但是这样讲解也是有缺陷的，代码量太大，而且由于是产品，会考虑的边缘问题非常多，需要把源码摘出来分析才可以，需要花大量的时间，这个我已经在整理了。

好，我们就先言归正传，像根据第一种思路来讲一个案例。

27.1 命令模式+责任链模式

在操作系统的世界里，有两大阵营一直在 PK 着：*nix（包括 Unix 和 Linux）和 Windows，这两个阵营从一开始就互掐，各说各的好，从目前的统计数据来看*nix 在应用服务器领域占据相对优势，不过也正在被 Windows 蚕食，国内某些小型银行已经在使用 PC Server（安装 Windows 操作系统的服务器）集群来进行银行业务运算，而且稳定性、性能各方面的效果还不错；而在个人桌面方面，Windows 是绝对占优势的，我相信大家基本上都在用这个操作系统，它的诸多优点就不多说了，我们今天就来解决一个习惯问题，如果你负责过 Unix 系统维护话，你自己的笔记本又是 windows 操作系统，我想你肯定有这样的经验，如下图：



```
C:\Documents and Settings\Administrator>ls
'ls' 不是内部或外部命令，也不是可运行的程序
或批处理文件。

C:\Documents and Settings\Administrator>ls -la
'ls' 不是内部或外部命令，也不是可运行的程序
或批处理文件。

C:\Documents and Settings\Administrator>ll
'll' 不是内部或外部命令，也不是可运行的程序
或批处理文件。

C:\Documents and Settings\Administrator>
```

哈哈，是不是经常发生这种事情，把 Unix 上的命令敲写到 windows 系统了？我是经常发生，然后就会惊奇的说“嗯~，怎么不支持这个命令了”（如果你还没有发生过这种情况，赶快找个 unix 机器去联系），为了避免这种情况发生，我就想了个招儿，把 Unix 上的命令移植到 windows 上，也就是 windows 下的 shell 工具，有很多类似的工具，比如 cygwin、GUN Bash 等等，这些都是非常完美的工具，我们今天的任务就是自己写一个这样的工具，怎么写呢？我们学了这么多的模式，当然要融会贯通了，使用命令模式、责任链模式、模版方法模式设计一个方便扩展、稳定的工具，我们按照这个思路一步一步来设计。

我们先说说 unix 下的命令，一条命令分为命令名、选项和操作数，例如这样一条命令 `ls -l /usr`，其中 `ls` 是命令名，`-l` 是选项，`/usr` 是操作数，后两项都是可选项，根据实际情况而定。Unix 命令一定遵守以下几个规则：

命令名为小写

命令名、选项、操作数之间以空格为分割符号，空格数量不限制；

选项之间可以组合使用，也可以单独拆分使用；

选项以-（横杠）开通；

在 unix 世界中，我们最常用的就是 `ls` 这个命令，它是显示目录或文件信息，我们就先来看看这个命

令，常用的有以下几条组合命令：

ls:简单列出一个目录下的文件

ls -l:详细列表目录下的文件；

ls -a:列出目录下包含隐藏文件，主要是带.(点号)的文件；

ls -s 列出文件的大小；

还有非常常用的组合命令，如 ls -la、ls -ls 等等，那我们来思考这个问题，ls 命令名是确定了，但是其后连接的选项和操作数是不确定的，操作数我们不用管它不跟操作数就是当前的目录，关键是选项，用哪个选项，什么时候使用都是由用户决定的，也就是我们要完全的解析所有的参数，那我们应该有 N 多个类来处理如此多的选项，客户输入一个参数，就返回一个结果，针对一个 ls 命令族其要求如下：

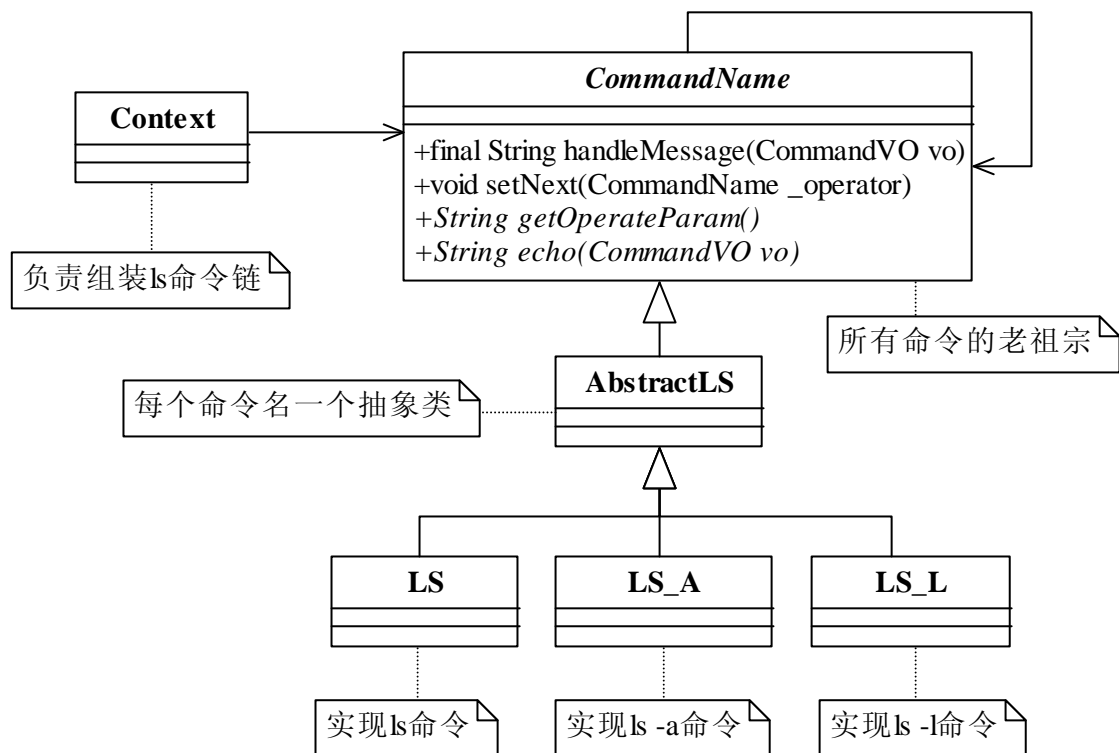
每一个 ls 命令都操作数，默认操作数为当前目录；

选项不可重复，例如 ls -l -l -s，解析出的选项应该只有 2 个：l 选项和 s 选项；

每个选项返回不同的结果，也就是说每个选项应该由不同的业务逻辑来处理；

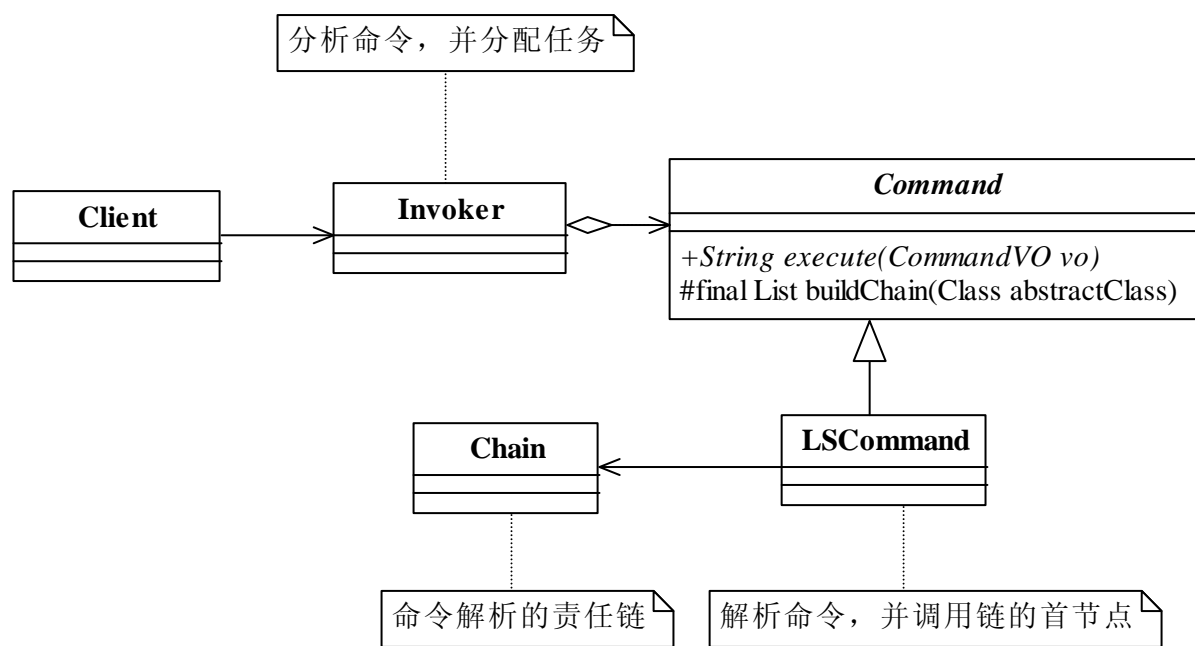
为提高扩展性，ls 命令族内的运算对外应该是封闭的，减少外界访问 ls 命令族访问的可能性；

针对一个命令族的分析结束，想想看看，我们可以使用什么模式？责任链模式！对，只要把一个参数传递到一个链首，就可以立刻获得一个结果，中间是如何传递、由哪个逻辑解析的都不是外界（高层）模块关心的，该模块的类图的如下：

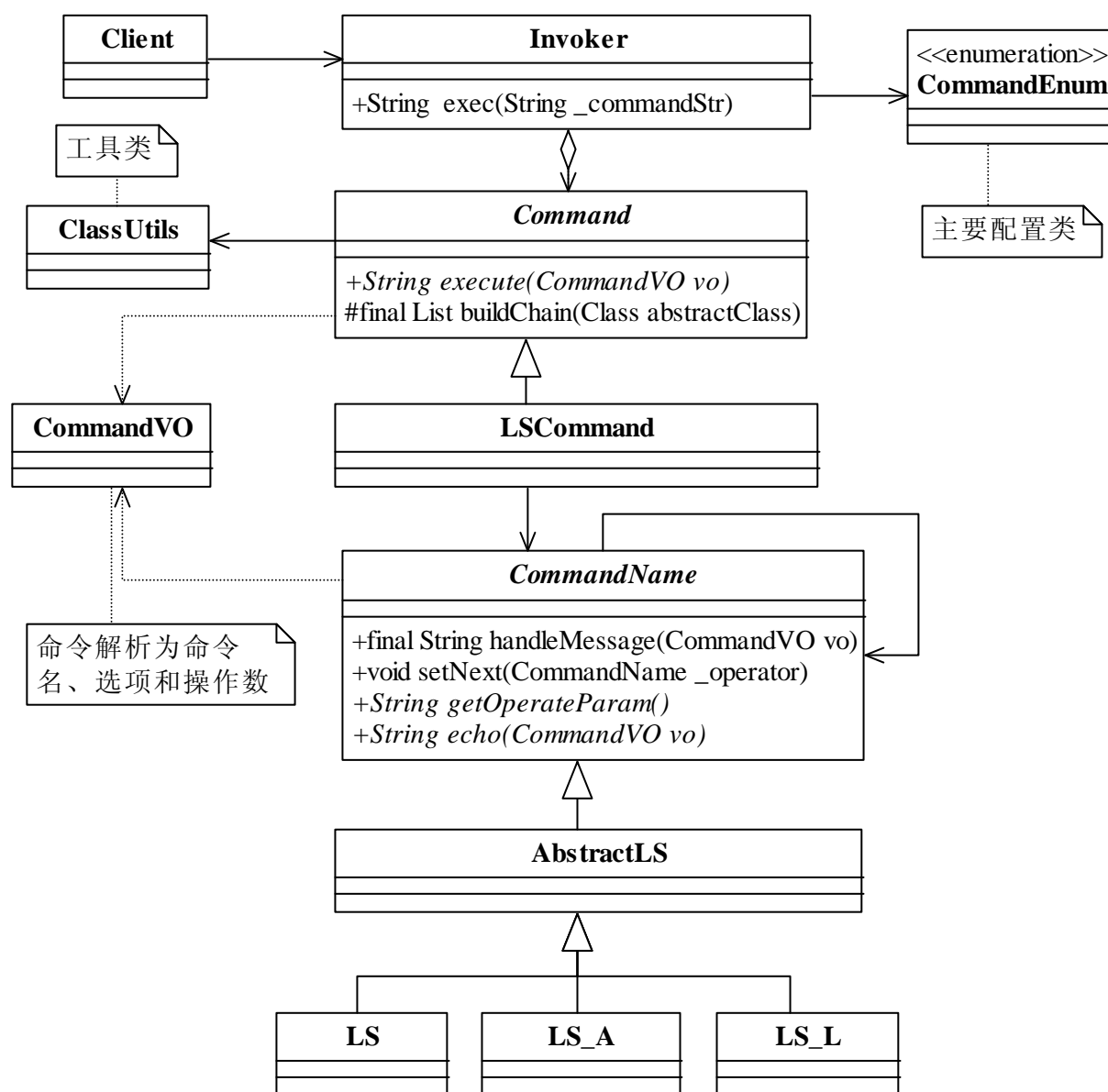


类图还是比较清晰的，unix 的命令有上百个，我们定义一个 `CommandName` 抽象类，所有的命令都继承于该类，它就是责任链模式的 handler 类，负责链表控制；每个命令族都有一个抽象类，因为每个命令族都有其独特的个性，比如 `ls` 命令和 `df` 命令，其后可加的参数是不一样的，这就可以在抽象类 `AbstractLS` 中定义，而且它还有标示作用，标示其下的实现类都是实现 `ls` 命令的，只是命令的选项不同；`Context` 负责建立一条命令的链表，比如 `ls` 命令族，`df` 命令族等等，它组装出一个 `ls` 命令链，并返回首节点供高层模块调用，非常典型的责任链模式。

具体的一个命令族分析完毕，可以采用责任链模式，我们继续往下分析，unix 命令有很多，非常多，敲一个命令返回一个结果，每个具体的命令可以由相关的命令族（也就是责任链）来解析了，但是这么多命令还是需要有一个派发的角色，我输入一个命令，不管后台谁来解析，就是你了，你给我返回一个结果就成，其他的我都不关心，什么模式？命令模式！命令模式负责协调各个命令正确的被传递到了各个责任链的首节点，这就是它的任务，任务及其重要！看类图：



是不是典型的命令模式类图？那就好，老熟人，好办事，其中 `Chain` 是一个标示符，表示的就是我们上面分析的责任链，每一个具体的命令负责调用责任链的首节点，获得返回值，结束命令的执行。两个核心模块都分析完毕了，完美就可以把类图融合在一起，就有了如下类图：



不要发怵，这个类图还是比较简单的，我来解释一下各个类的职责：

ClassUtils：工具类，主要职责是根据一个接口、父类查找到所有的子类；这个有用，超级有用，在不考虑效率的应用中，使用该类可以带来非常好的扩展性。

CommandVO：命令的值对象，把一个命令解析为命令名、选项、操作数，例如 `ls -l /usr` 命令，分别解析为 `getCommandName`、`getParam`、`getData` 三个方法的返回值，面向对象操作嘛。

CommandEnm：枚举类型，主要的命令配置文件，为什么要 Enum 类型？这是 JDK 1.5 提供的一个非常好的功能，我们在程序中再讲解如何使用。

所有的分析都已经完成了，我们来看看我们的程序，程序不复杂，还是很简单的，看看类图，如果你，做为程序员拿到这样的一个类图，先开始写一个类？Of Course，命令的解释了，这是一我们这个项目

的核心，好，我们先来看 CommandName 抽象类：

```
public abstract class CommandName {

    private CommandName nextOperator;

    public final String handleMessage(CommandVO vo){
        //处理结果
        String result = "";

        //判断是否是自己处理的参数
        if(vo.getParam().size() == 0 ||
vo.getParam().contains(this.getOperateParam())){
            result = this.echo(vo);
        }else{
            if(this.nextOperator !=null){
                result = this.nextOperator.handleMessage(vo);
            }else{
                result = "命令无法执行";
            }
        }
    }

    return result;
}

//设置剩余参数谁来处理
public void setNext(CommandName _operator){
    this.nextOperator = _operator;
}

//每个处理者都要处理一个后缀参数
protected abstract String getOperateParam();

//每个处理者都必须实现处理任务
protected abstract String echo(CommandVO vo);
}
```

很简单，就是责任链模式中的 handler，也就是中控程序，控制一个链应该如何建立。我们再来看三个 ls 命令家族，先看 AbstractLS 抽象类：

```

public abstract class AbstractLS extends CommandName{
    //默认参数
    public final static String DEFAULT_PARAM = "";
    //参数a
    public final static String A_PARAM = "a";
    //参数l
    public final static String L_PARAM = "l";
}

```

很惊讶，是吗？怎么是个空的抽象类，是的，确实是一个空类，就定义了三个参数名称，它有两个职责：一是标记 ls 命令家族，所有的 ls 命令都是我的子类，别到处再找了；二是个性化处理，因为现在还没有思考清楚 ls 有什么个性（可以把命令的选项也认为是其个性化数据），先写个空类放这里，以后想清楚了再填写上去，世界是不完美的，设计总有缺陷，该留下一些可扩展的类就留下，对未来会产生不可估量的好处。我们再来看 ls 不带任何参数的命令，如下代码：

```

public class LS extends AbstractLS {

    //最简单的ls命令
    protected String echo(CommandVO vo) {
        return FileManager.ls(vo.formatData());
    }

    //参数为空
    protected String getOperateParam() {
        return super.DEFAULT_PARAM;
    }

}

```

太简单了，首先定义了自己能处理什么样的参数，我只能处理不带参数的 ls 命令，那 getOperateParam 就返回一个长度为零的字符串，就是说该类作为链上的一个节点，只处理没有参数的 ls 命令。Echo 方法是执行执行 ls 命令，通过调用操作系统相关的命令返回结果。我们再来看 ls -l 命令，如下所示：

```

public class LS_L extends AbstractLS {

    protected String echo(CommandVO vo) {
        return FileManager.ls_l(vo.formatData());
    }

}

```

```

    }

    //1参数
    protected String getOperateParam() {
        return super.L_PARAM;
    }
}

```

该类只处理选项为“l”的命令，不多说，也非常简单。ls -a 命令处理与此类似，如下所示：

```

public class LS_A extends AbstractLS {
    //ls -a命令
    protected String echo(CommandVO vo) {
        return FileManager.ls_a(vo.formatData());
    }

    protected String getOperateParam() {
        return super.A_PARAM;
    }
}

```

在三个实现类中，都是用了 FileManager，这个类是做什么用的呢？是负责与操作系统交互的，要把 unix 的命令迁移到 windows 上运行，那就需要调用 windows 的低层函数，实现较复杂，而且和我们本章要讲的内容没有太大的关系，我们就采用示例性代码代替，如下所示：

```

public class FileManager {

    //ls命令
    public static String ls(String path){
        return "file1\nfile2\nfile3\nfile4";
    }

    //ls -l命令
    public static String ls_l(String path){
        String str = "drw-rw-rw    root    system    1024    2009-8-20 10:23
file1\n";
    }
}

```

```

        str = str + "drw-rw-rw  root    system    1024    2009-8-20 10:23
file2\n";
        str = str + "drw-rw-rw  root    system    1024    2009-8-20 10:23  file3";
        return str;
    }

    //ls -a命令
    public static String ls_a(String path){
        String str = ".\n..\nfile1\nfile2\nfile3";
        return str;
    }
}

```

都是比较简单的方法，大家有兴趣可以自己实现一下，给大家提供三种思路：一种是通过 java.io.File 类自己封装出类似 unix 的返回格式；一种是通过 java.lang.Runtime 类的 exec 方法执行 dos 的 dir 命令，产生类似的 ls 结果，但是格式和 unix 不同，需要读者自己处理一下；还有一种就是通过 JNI (Java Native Interface) 来调用与操作系统有关的动态链接库，当然前提是需要自己写一个动态链接库文件。

三个具体的命令都已经解析完毕，我们再来看看如何建立一条处理链，由于建链的任务已经移植到具抽象命令类，我们就先来看抽象类 Command：

```

public abstract class Command {
    public abstract String execute(CommandVO vo);

    //建立链表
    protected final List<? extends CommandName> buildChain(Class<? extends
CommandName> abstractClass){
        //取出所有的命令名下的子类
        List<Class> classes = ClassUtils.getSonClass(abstractClass);
        //存放命令的实例，并建立链表关系
        List<CommandName> commandNameList = new ArrayList<CommandName>();
        for(Class c:classes){
            CommandName commandName =null;
            try {
                //产生实例
                commandName =
                (CommandName)Class.forName(c.getName()).newInstance();
            } catch (Exception e){
                // TODO 异常处理
            }
        }
    }
}

```

```

        //建立链表
        if(commandNameList.size()>0){

            commandNameList.get(commandNameList.size()-1).setNext(commandName);
        }
        commandNameList.add(commandName);
    }
    return commandNameList;
}
}

```

再看一遍上面的程序，它让你思绪万千，Command 抽象类有两个作用：一是定义命令的执行方法，二是负责命令族（责任链）的建立，其中 buildChain 方法负责建立一个责任链，它的通过接收一个抽象的命令家族类就可以建立一条命令解析链，如传递 AbstractLS 类就可以建立一条解析 ls 命令家族的责任链，非常完美，请读者注意这句话：

```
commandName = (CommandName)Class.forName(c.getName()).newInstance();
```

在一个遍历中，classes 中的每个元素都是一个类名，然后根据类名产生一个实例，它会抛出异常，例如类文件不存在、初始化失败等，读者在自己设计是要出来该部分的异常。我们再来想，每个实现类的类名是如何取得的呢？问的好，看这句话：

```
List<Class> classes = ClassUtils.getSonClass(abstractClass);
```

根据一个父类取得所有子类，非常好的一个工具类，其实现如下：

```

public class ClassUtils {

    //根据父类查找到所有的子类，默认情况是子类 and 父类都在同一个包名下
    public static List<Class> getSonClass(Class fatherClass){
        //定义一个返回值
        List<Class> returnClassList = new ArrayList<Class>();
        //获得包名称
        String packageName = fatherClass.getPackage().getName();
        //获得包中的所有类
        List<Class> packClasses = getClasses(packageName);
        //判断是否是子类
        for(Class c:packClasses){
            if(fatherClass.isAssignableFrom(c) && !fatherClass.equals(c)){
                returnClassList.add(c);
            }
        }
    }
}

```



```
    }  
    return returnClassList;  
}
```

//从一个包中查找出所有的类，在jar包中不能查找

```
private static List<Class> getClasses(String packageName) {  
    ClassLoader classLoader = Thread.currentThread()  
        .getContextClassLoader();  
    String path = packageName.replace('.', '/');  
    Enumeration<URL> resources = null;  
    try {  
        resources = classLoader.getResources(path);  
    } catch (IOException e) {  
        // TODO Auto-generated catch block  
        e.printStackTrace();  
    }  
    List<File> dirs = new ArrayList<File>();  
    while (resources.hasMoreElements()) {  
        URL resource = resources.nextElement();  
        dirs.add(new File(resource.getFile()));  
    }  
    ArrayList<Class> classes = new ArrayList<Class>();  
    for (File directory : dirs) {  
        classes.addAll(findClasses(directory, packageName));  
    }  
    return classes;  
}  
  
private static List<Class> findClasses(File directory, String packageName) {  
    List<Class> classes = new ArrayList<Class>();  
    if (!directory.exists()) {  
        return classes;  
    }  
    File[] files = directory.listFiles();  
    for (File file : files) {  
        if (file.isDirectory()) {  
            assert !file.getName().contains(".");  
            classes.addAll(findClasses(file, packageName + "." +  
file.getName()));  
        } else if (file.getName().endsWith(".class")) {  
            try {  

```

```

        classes.add(Class.forName(packageName + '.' +
file.getName().substring(0, file.getName().length() - 6)));
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
}
}
return classes;
}
}

```

这个类请大家谨慎使用，工具是好工具，但是在核心的应用中不要使用该工具，它会严重影响你的性能，而且它是一条“旁门左道”，逼不得已的时候才使用，根据父类查找子类呀，想想看，这是一件多么危险的事情呀，真是世界是先有父才有子，而我们却采用了扭转时空的做法，子类的实例对象是根据父类查找到的，危险危险！

再来看实现类 LSCommand 类，如下所示：

```

public class LSCommand extends Command{

    public String execute(CommandVO vo){
        //返回链表的首节点
        return super.buildChain(AbstractLS.class).get(0).handleMessage(vo);
    }
}

```

很简单的方法，建立一个命令族的责任链，然后找到首节点调用，完成，非常简单。在该父子类中我们使用 CommandVO 类，它是一个封装对象，这也比较简单，其代码如下：

```

public class CommandVO {
    //定义参数名与参数的分割符号,一般是空格
    public final static String DIVIDE_FLAG = " ";

    //定义参数前的符号,UNIX一般是-,如ls -la
    public final static String PREFIX="-";

    //命令名,如ls,du
    private String commandName = "";

    //参数列表
}

```

```

private ArrayList<String> paramList = new ArrayList<String>();

//操作数列表
private ArrayList<String> dataList = new ArrayList<String>();

//通过构造函数传递进来命令
public CommandVO(String commandStr){
    //常规判断
    if(commandStr != null && commandStr.length() !=0){
        //根据分割符号拆分出执行符号
        String[] complexStr = commandStr.split(CommandVO.DIVIDE_FLAG);
        //第一个参数是执行符号
        this.commandName = complexStr[0];
        //把参数放到List中
        for(int i=1;i<complexStr.length;i++){
            String str = complexStr[i];
            //包含前缀符号，认为是参数
            if(str.indexOf(CommandVO.PREFIX)==0){
                this.paramList.add(str.replace(CommandVO.PREFIX,
"" ).trim());
            }else{
                this.dataList.add(str.trim());
            }
        }
    }else{
        //传递的命令错误
        System.out.println("命令解析失败，必须传递一个命令才能执行！");
    }
}

//得到命令名
public String getCommandName(){
    return this.commandName;
}

//获得参数
public ArrayList<String> getParam(){
    //为了方便处理空参数
    if(this.paramList.size() ==0){
        this.paramList.add("");
    }
    return new ArrayList(new HashSet(this.paramList));
}

```

```

//获得操作数
public ArrayList<String> getData(){
    return this.dataList;
}

}

```

CommandVO 解析一个命令，规定一个命令必须有三项：命令名、选项、操作数，如果没有呢？那就以长度为零的字符串代替，通过这样的一个约定可以大大降低命令解析的开发工作。注意 getParam 参数中的返回值：

```
new ArrayList(new HashSet(this.paramList));
```

为什么要这么处理？想想看看，HashSet 有什么优点？值唯一，对，我们这样处理就是为了避免出现两个相同的参数，比如 `ls -l -l -s` 这样一个命令，通过 getParam 返回的参数是几个呢？两个：l 选项和 s 选项，其中就是这句话去掉了重复的 l 选项。

我们再来看 Invoker 类，负责命令分发的类，代码如下：

```

public class Invoker {

    //执行命令
    public String exec(String _commandStr){
        //定义返回值
        String result = "";
        //首先解析命令
        CommandVO vo = new CommandVO(_commandStr);
        //检查是否支持支持该命令
        if(CommandEnum.getNames().contains(vo.getCommandName())){
            //产生命令对象
            String className =
                CommandEnum.valueOf(vo.getCommandName()).getValue();
            Command command;
            try {
                command = (Command)Class.forName(className).newInstance();
                result = command.execute(vo);
            }catch(Exception e){
                // TODO 异常处理
            }
        }else{
            result = "无法执行命令，请检查命令格式";
        }
    }
}

```

```
        return result;
    }

}
```

实现也是比较简单的，从 CommandEnum 中获得命令与命令类的配置信息，然后建立一个命令实例，调用其 execute 方法，完成命令的执行操作，该方法是考虑以后的系统扩展要求。刚刚我们提到了 CommandEnum 类，这是一个枚举类型，其代码如下：

```
public enum CommandEnum {
    ls("com.cbf4life.common.command.LSCommand");

    private String value = "";

    //定义构造函数，目的是Data(value)类型的相匹配
    private CommandEnum(String value){
        this.value = value;
    }

    public String getValue(){
        return this.value;
    }

    //返回所有的enum对象
    public static List<String> getNames(){
        CommandEnum[] commandEnum = CommandEnum.values();
        List<String> names = new ArrayList<String>();
        for(CommandEnum c:commandEnum){
            names.add(c.name());
        }

        return names;
    }

    /*
     * java enum类型尽量简单使用，尽量不要使用多态、继承等方法
     * 毕竟用Class完全可以代替enum
     */
}
```

跳到你脑海中的第一印象是：为什么要用 Enum？这用一个类来管理很容易来实现的。没错，别整天使

用那些 10 年前的老技术了，换新的吧，你会发现有惊喜等待着你。顺便说下，注意 CommandEnum 中的构造函数 CommandEnum(String value) 和 getValue 类，有意思吧，你没有 new 一个 Enum 对象，但是你可以直接使用 CommandEnum.ls.getValue 方法获得值，这就是 Enum 类型的独特地方，而且看到这个：

```
ls("com.cbf4life.common.command.LSCommand");
```

你是不是很不舒服？别就抓住 Enum 的基本功能不妨，是的，枚举的基本功能就是定义默认可选值，但是 Java 世界中则枚举它又增强了很多，可以实现接口、可以继承，还可以添加方法和属性，若要详细了解 Enum，读者可以翻阅一下相关语法书。

现在剩下的工作就是写一个 Client 类，然后看运行情况如何，其代码如下所示：

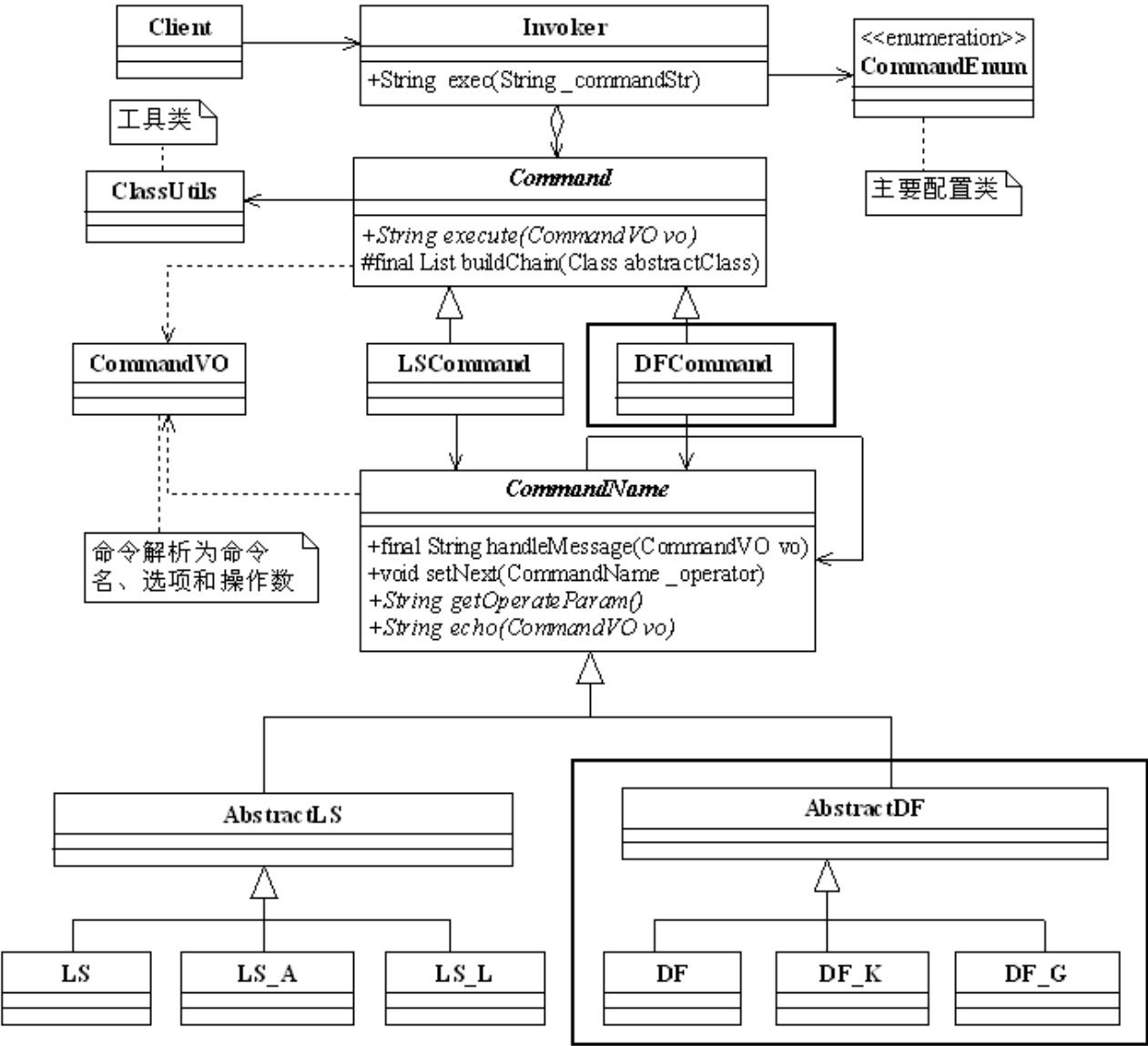
```
public class Client {
    public static void main(String[] args) throws IOException {
        Invoker invoker = new Invoker();
        while(true){
            //unix下的默认提示符号
            System.out.print("#");
            //捕获输出
            String input = (new BufferedReader(new
InputStreamReader(System.in))).readLine();
            //输入quit或exit则退出
            if(input.equals("quit") || input.equals("exit")){
                return;
            }
            System.out.println(invoker.exec(input));
        }
    }
}
```

Client 也很简单，通过一个 while 循环允许使用者持续输入，然后把打印出返回值，运行结果如下：

```
#ls
file1
file2
file3
file4
#ls -l
drw-rw-rw  root  system  1024  2009-8-20 10:23  file1
drw-rw-rw  root  system  1024  2009-8-20 10:23  file2
```

```
drw-rw-rw  root  system  1024  2009-8-20 10:23  file3
#ls -a
.
..
file1
file2
file3
#quit
```

哈哈，我们已经实现在 windows 下操作 unix 命令的功能，但是就仅仅一个 ls 命令族是不够的，我们要扩展，把一百多个命令都扩展出来，那怎么扩展呢？好，现在增加一个命令 df 命令，显示磁盘的大小，如何处理呢？很好办，类图只要增加就成，如下所示：



仅仅增加了黑框的部分，也就是 DFCommand、AbstractDF 以及实现类就可以完成其扩展功能，不信？

Look，先看 AbstractDF 代码：

```
public abstract class AbstractDF extends CommandName {
    //默认参数
    public final static String DEFAULT_PARAM = "";
    //参数k
    public final static String K_PARAM = "k";
    //参数g
    public final static String G_PARAM = "g";
}
```

一样的功能，定义选项名称，接下来是三个实现类，都非常简单，如下所示：

```
public class DF extends AbstractDF{
    //定义一下自己能处理什么参数
    protected String getOperateParam() {
        return super.DEFAULT_PARAM;
    }

    //命令处理
    protected String echo(CommandVO vo) {
        return DiskManager.df();
    }
}
```

```
public class DF_K extends AbstractDF{
    //定义一下自己能处理什么参数
    protected String getOperateParam() {
        return super.K_PARAM;
    }

    //命令处理
    protected String echo(CommandVO vo) {
        return DiskManager.df_k();
    }
}
```



```

public class DF_G extends AbstractDF{
    //定义一下自己能处理什么参数
    protected String getOperateParam() {
        return super.G_PARAM;
    }

    //命令处理
    protected String echo(CommandVO vo) {
        return DiskManager.df_g();
    }
}

```

每个选项的实现类都定义了自己能解析什么命令，然后通过 echo 方法返回执行结果。在三个实现类中都与 DiskManager 类有关联关系，该类负责与操作系统有关的功能，必须要实现的，其示例代码如下：

```

public class DiskManager {

    //默认的计算大小
    public static String df(){
        return "/\t10485760\n/usr\t104857600\n/home\t1048576000\n";
    }

    //按照kb来计算
    public static String df_k(){
        return "/\t10240\n/usr\t102400\n/home\t10240000\n";
    }

    //按照gb计算
    public static String df_g(){
        return "/\t10\n/usr\t100\n/home\t10000\n";
    }
}

```

以上为示例代码，若要实际计算磁盘大小，可以使用 JNI 的方式或者执行操作系统的命令方式获得，特别是 JDK 1.6 提供了获得一个 Root 目录大小的方法，非常简单就可以实现。

然后再增加一个 DFCommand 命令，负责执行命令，如下代码：

```

public class DFCommand extends Command {

```

```

    public String execute(CommandVO vo) {
        return super.buildChain(AbstractDF.class).get(0).handleMessage(vo);
    }
}

```

最后一步，修改一下 CommandEnum 配置，增加一个枚举项，如下所示：

```

public enum CommandEnum {
    ls("com.cbf4life.common.command.LSCommand"),
    df("com.cbf4life.common.command.DFCommand");

    private String value = "";

    //定义构造函数，目的是Data(value)类型的相匹配
    private CommandEnum(String value){
        this.value = value;
    }

    public String getValue(){
        return this.value;
    }

    //返回所有的enum对象
    public static List<String> getNames(){
        CommandEnum[] commandEnum = CommandEnum.values();
        List<String> names = new ArrayList<String>();
        for(CommandEnum c:commandEnum){
            names.add(c.name());
        }

        return names;
    }
}

```

好了，完工，运行，结果如下所示：

```

#ls
file1

```

```
file2
file3
file4
#df
/      10485760
/usr 104857600
/home      1048576000

#df -k
/      10240
/usr 102400
/home      t10240000

#df -g
/      10
/usr 100
/home      t10000

#
```

如何，是不是仅仅增加类就完成了变更？这才是我们要的结果：对修改关闭，对扩展开放。我们回想一下我们在这里例子中使用到了什么模式：

责任链模式：负责对命令的参数进行解析，而且所有的扩展都是增加链数量和节点，涉及不到原有的代码变更；

命令模式：负责命令的分发，把适当的命令分发到指定的链上；

模版方法模式：在 Command 类以及子类中，buildChain 方法是模版方法，只是没有基本方法而已；在责任链中的 CommandName 类中，一个典型的模版方法 handlerMessage，调用了基本方法，基本方法由各个实现类实现，非常有利于扩展。

迭代器模式：在 for 循环中我们多次使用到 for(Class c:classes)类似的结构，是谁来支撑该方法可以运行？Iterator 模式，只是 JDK 已经把它融入了 API 中，更方便使用了。

可能读者已经注意到了，ls -l -a 这样的组合选项还没有处理呢，确实没有处理，可以为大家提供两个思路来处理：ls -l -a 等同于 ls -la，也等同于 ls -al 命令，可以把 ls -la 中的选项 la 作为一个参数来进行处理，扩展一个类就可以了，该方法的缺点就是类膨胀的太大，但是简单；第二种方法，是修正命令族处理链，每个命令处理节点运行完毕后，继续由后续节点处理，最终由 Command 类组装结果，根据每个节点的处理结果，组合后生成完整的返回信息，如 ls -l -a 就应该是 LS_L 类与 LS_A 类两者返

回值组装的结果，那当然链上的节点返回值就要放 Collection 类型中了。

该框架大家可以继续扩展下去，当然了，上面的程序还是可以优化的，优化的结果就是 Command 类萎缩为一个类，通过 CommandEnum 配置文件类传递命令，比较容易实现，读者可以自行进行扩展设计。

第 28 章 更新记录:

2009 年 4 月 22 日 完成策略模式和代理模式的编写；并发布到论坛上；

2009 年 4 月 24 日 完成单例模式和多例模式的编写；

2009 年 4 月 29 日 完成工厂方法编写；

2009 年 5 月 2 日 完成抽象工厂模式的编写；

2009 年 5 月 2 日 完成了门面模式的编写；增加封面、编写计划、后序以及部分类图

2009 年 5 月 10 日 完成适配器模式的编写；

2009 年 5 月 17 日 完成模板方法模式和建造者模式；

2009 年 5 月 24 日 完成桥梁模式；

2009 年 5 月 30 日 完成命令模式和装饰模式；

2009 年 6 月 6 日 完成迭代器模式；

2009 年 6 月 7 日 完成组合模式；

2009 年 6 月 18 日 完成观察者模式；

2009 年 6 月 21 日 完成责任链模式；

2009 年 6 月 28 日 完成状态模式

2009 年 6 月 30 日 完成单一职责原则；

2009 年 7 月 4 日 完成访问者模式；

2009 年 7 月 5 日 完成迪米特法则；

2009 年 7 月 5 日 完成里氏替换法则；

2009 年 7 月 12 日 完成原型模式；

2009 年 7 月 26 日 完成中介者模式、迪米特原则以及接口隔离原则。

2009 年 8 月 5 日 完成开闭原则

2009 年 8 月 13 日 完成解释器模式

2009 年 9 月 16 日 完成第一个混编模式

相关说明

软件环境: JDK 1.5 MyEclipse 7.0

类图设计软件: Rational Rose 和 StartUML

博客地址:

<http://hi.baidu.com/cbf4life/blog/item/e1ff58f849a8ea51242df219.html>

源代码及类图下载地址:

[http:// cbf4life.free.66ip.com /source-code.rar](http://cbf4life.free.66ip.com/source-code.rar)

论坛:

<http://www.javaeye.com/topic/372233>

本文的最新下载地址, 请博客上查找:

<http://hi.baidu.com/cbf4life/blog/item/e1ff58f849a8ea51242df219.html>

第 29 章 后序

首先，要说的是非常感谢大家的支持，让我有勇气续写下去，我为什么要写这本书？从 2000 年毕业，到现在 9 年的光影，在 IT 技术这块基本上都做过了，从程序员起步，高级程序员，系统分析师，项目经理，测试经理，质量经理，项目维护，IT 老师，呵呵，接触的语言也比较多，什么 C 了汇编了 Ruby 了这些边角的東西都做過項目，更別說 Java 了，Java 項目做了 6 年，金融交易類的，OA 管理類的，政府類的等都做過，這幾年基本上都是項目經理和技术經理一塊兼職，搞的很累，帶過最少 3 個人的團隊，也帶過 40 多人的研發團隊，可以說自己比較失敗，9 年了，始終還是個做技術的，不過技術還是我最熱衷的，我喜歡看簡單清晰明了易懂的代碼，一看代碼一團糟，那這人肯定不怎麼樣。

9 年了，整天忙的跟瘋子一樣，不知道這樣忙下去什麼時候是個頭，而且自己的年齡也不小了，不能跟着這批 80 后甚至是 90 后一起瘋狂加班了，並且最近在工作上也發生了一些事情，讓我根本看不清楚未來的路，單位是好，福利應該是同行業的中上等吧，可我們這小兵的路在何方？40 歲了還寫代碼，中國可行嗎？跟 90 后甚至是 00 后一起加班？這不是我想要的工作模式，很失落，所以想找个精神激勵，於是就想把自己的這幾年的工作經驗整理成一本書，讓大家盡量能看懂的書，大家的每一個回帖、郵件、評價都給了我莫大的鼓勵！由於是第一次寫書，可能確實有部分考慮不周，請大家指正。這本書能出版更好，不能出版也無所謂，我只是想找个精神鼓勵。

大家一看目錄可能就發怵了，怎麼是 24 個模式呀，一般書上都是 23 個模式，呵呵，確實是，我增加了多例模式，這個一般都是融合在單例模式中講的，我是拆出來了。

設計模式這塊，我是從 04 年開始系統學習的，《J2EE 核心模式》、《Java 與模式》、《Head First 設計模式》等幾本經典的書都拜讀過，也確實學到了不少東西，但是始終覺的這些書有缺陷，前兩本都是一副孔老夫子假正經的摸樣，板著臉一本正經的在講技術，真的是研究，實話說我是忍著看下去的，技術是為了使用服務的，你說那么多用不到的优缺点、场景干什么，干嘛不说个大家接触到的看的懂例子？！《Head First 设计模式》实话说，我不喜欢，西式的幽默不合我的胃口，看过一遍就不想看第二遍了。

这个序是我想到那里就写到那里，没有个重点，也没个理个头绪出来，大家将就吧。

2009 年 5 月 2 日

还有一个小时就又到明天了，这段时间写这本书，确实感触良多，已经一个多月过去了，进展还算可以，至少还是比自己预期的计划提前了一些。

我看书有个坏习惯，看完书后书本就找不着了！就图个新鲜，基本上一本书看很多遍的情况比较少（也有，金庸的《鹿鼎记》、《笑傲江湖》、《倚天屠龙记》等看过应该不下十遍，大学的时候，同学借书了

就蹭着看一遍)，而且自己也不是很聪明的那种人，一目十行，过目不忘统统和我不沾边，我基本上属于那种看的多记得少的人，我从中学就开始看什么四大名著了、《初刻拍案惊奇》、《古文观止》还有什么《老残游记》等等一些让人看着比较反胃的东西，基本上看过就忘，反正老爸也不管，他就看我在看书就行，记住多少老爸也不管不问，要看书就给你买，在苦也要给我买书，这也是天下父母的一片苦心哪！做了技术这一行以后，看书还是这个习惯，书是看了不少，看过就看过了，你要问过我这本书到底写什么，我会大致给你描述是写 Struts 的，或是写项目管理的，或是写领域模型的，再详细的就没了，再去找那本书，也不找不到了，不知道是借给谁了，或者当垃圾扔掉了。而且我读书也没有记笔记的习惯，看到哪里是哪里，怎么看自己都像个懒人！

前段时间和一个师兄在争执一个问题，最后我接受了他的说法：技术上没有绝对的好与坏，一个方案或者提议，如果你要是按照理论值来争论，那就没有个结果，就像以前有人争论过使用 delegate 的方式隔离各个分层模块一样，有人说好，有人说不好，各说各的，确实不是很好，但是在项目中用了那就是好，分层清晰了，开发不相互污染了，进度加快了，有什么不好的，所以大家有时候为了一个问题争的面红耳赤的，何必呢！技术上的事，特别是一应用到项目上，懂点技术的人都想套用理论知识，特别是所谓的“砖家”，吓不死你不算完，好像不按他们的套路来走，项目就必定玩儿完似的！记住了那是理论，不是实际！呵呵，这也是有感而发。

“纸上得来终觉浅，绝知此事要躬行”，大家看书的也体会了一下这句话。

又是不知所云，记录一下心情吧！大家凑合着理解吧。

2009 年 6 月 7 日