23 种设计模式详解（全23种）

设计模式的分类

总体来说设计模式分为三大类：

创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

A、创建模式（5种）

工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

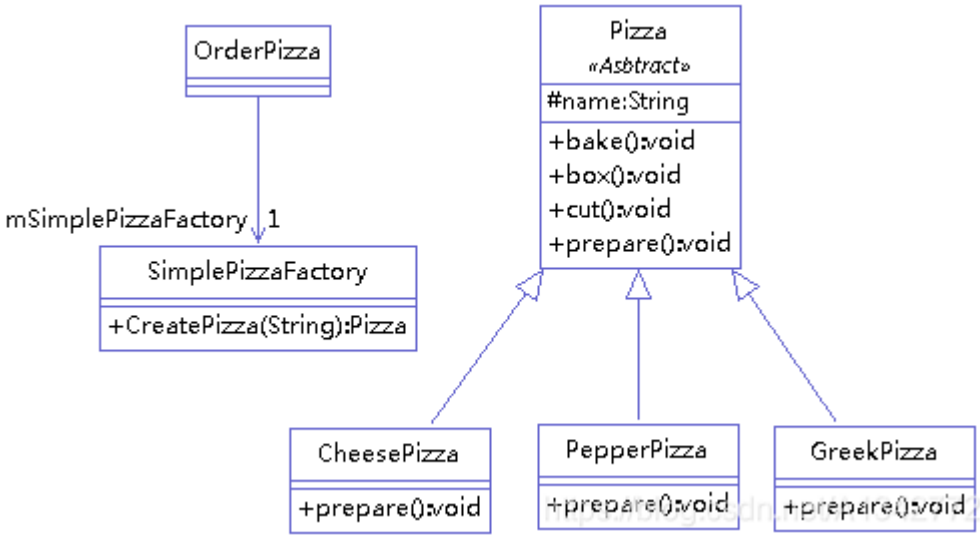
1 工厂模式

1.1 简单工厂模式

定义：定义了一个创建对象的类，由这个类来封装实例化对象的行为。

举例：（我们举一个pizza工厂的例子）

pizza工厂一共生产三种类型的pizza：chesse,pepper,greak。通过工厂类（SimplePizzaFactory）实例化这三种类型的对象。类图如下：



工厂类的代码：

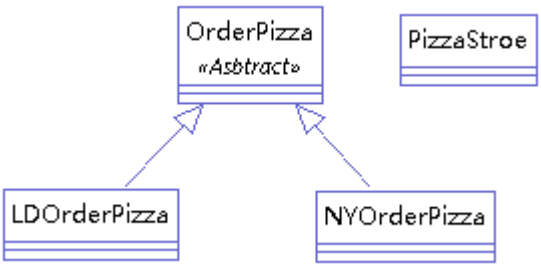
```
1. public class SimplePizzaFactory {
2.         public Pizza CreatePizza(String ordertype) {
3.                 Pizza pizza = null;
4.                 if (ordertype.equals("cheese")) {
5.                         pizza = new CheesePizza();
6.                 } else if (ordertype.equals("greek")) {
7.                         pizza = new GreekPizza();
8.                 } else if (ordertype.equals("pepper")) {
9.                         pizza = new PepperPizza();
10.                }
11.                return pizza;
12.        }
13. }
```

简单工厂存在的问题与解决方法： 简单工厂模式有一个问题就是，类的创建依赖工厂类，也就是说，如果想要拓展程序，必须对工厂类进行修改，这违背了开闭原则，所以，从设计角度考虑，有一定的问题，如何解决？我们可以定义一个创建对象的抽象方法并创建多个不同的工厂类实现该抽象方法，这样一旦需要增加新的功能，直接增加新的工厂类就可以了，不需要修改之前的代码。这种方法也就是我们接下来要说的工厂方法模式。

1.2 工厂方法模式

定义：定义了一个创建对象的抽象方法，由子类决定要实例化的类。工厂方法模式将对象的实例化推迟到子类。

举例：（我们依然举pizza工厂的例子，不过这个例子中，pizza产地有两个：伦敦和纽约）。添加了一个新的产地，如果用简单工厂模式的的话，我们要去修改工厂代码，并且会增加一堆的if else语句。而工厂方法模式克服了简单工厂要修改代码的缺点，它会直接创建两个工厂，纽约工厂和伦敦工厂。类图如下：



OrderPizza中有个抽象的方法：

```
abstract Pizza createPizza();
```

两个工厂类继承OrderPizza并实现抽象方法：

```
public class LDOrderPizza extends OrderPizza {
        Pizza createPizza(String ordertype) {
                Pizza pizza = null;
                if (ordertype.equals("cheese")) {
                        pizza = new LDCheesePizza();
                } else if (ordertype.equals("pepper")) {
                        pizza = new LDPepperPizza();
                }
                return pizza;
        }
}
public class NYOrderPizza extends OrderPizza {

         Pizza createPizza(String ordertype) {
                  Pizza pizza = null;

                  if (ordertype.equals("cheese")) {
                          pizza = new NYCheesePizza();
                  } else if (ordertype.equals("pepper")) {
                          pizza = new NYPepperPizza();
                  }
                  return pizza;

         }

}
```

、通过不同的工厂会得到不同的实例化的对象，PizzaStroe的代码如下：

```
1. public class PizzaStroe {
2.         public static void main(String[] args) {
3.                 OrderPizza mOrderPizza;
4.                 mOrderPizza = new NYOrderPizza();
5.         }
6. }
```

解决了简单工厂模式的问题：增加一个新的pizza产地（北京），只要增加一个BJOrderPizza类：

```
1. public class BJOrderPizza extends OrderPizza {
2.         Pizza createPizza(String ordertype) {
3.                 Pizza pizza = null;
4.                 if (ordertype.equals("cheese")) {
5.                         pizza = new LDCheesePizza();
6.                 } else if (ordertype.equals("pepper")) {
7.                         pizza = new LDPepperPizza();
8.                 }
9.                 return pizza;
10.         }
11. }
```

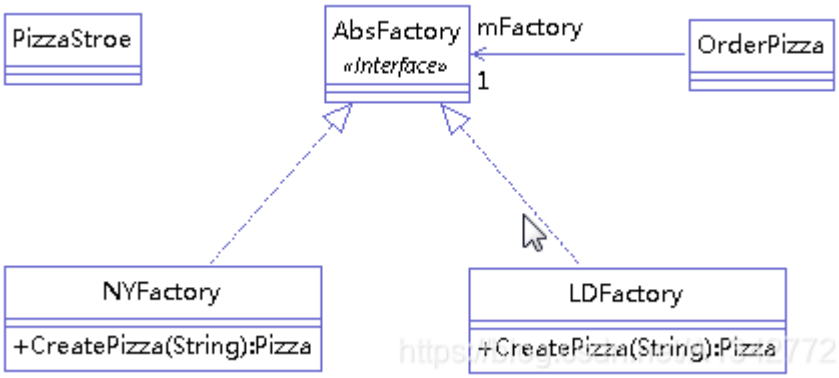
其实这个模式的好处就是，如果你现在想增加一个功能，只需做一个实现类就OK了，无需去改动现成的代码。这样做，拓展性较好！

工厂方法存在的问题与解决方法：客户端需要创建类的具体的实例。简单来说就是用户要订纽约工厂的披萨，他必须去纽约工厂，想订伦敦工厂的披萨，必须去伦敦工厂。 当伦敦工厂和纽约工厂发生变化了，用户也要跟着变化，这无疑就增加了用户的操作复杂性。为了解决这一问题，我们可以把工厂类抽象为接口，用户只需要去找默认的工厂提出自己的需求（传入参数），便能得到自己想要产品，而不用根据产品去寻找不同的工厂，方便用户操作。这也就是我们接下来要说的抽象工厂模式。

1.3 抽象工厂模式

定义：定义了一个接口用于创建相关或有依赖关系的对象族，而无需明确指定具体类。

举例：（我们依然举pizza工厂的例子，pizza工厂有两个：纽约工厂和伦敦工厂）。类图如下：



工厂的接口：

```
1. public interface AbsFactory {
2.         Pizza CreatePizza(String ordertype) ;
3. }
```

工厂的实现：

```
1. public class LDFactory implements AbsFactory {
2.         @Override
3.         public Pizza CreatePizza(String ordertype) {
4.                 Pizza pizza = null;
5.                 if ("cheese".equals(ordertype)) {
6.                         pizza = new LDCheesePizza();
7.                 } else if ("pepper".equals(ordertype)) {
8.                         pizza = new LDPepperPizza();
9.                 }
10.                return pizza;
11.        }
12. }
```

PizzaStroe的代码如下：

```
1. public class PizzaStroe {
2.         public static void main(String[] args) {
```

```
3.                 OrderPizza mOrderPizza;
4.                 mOrderPizza = new OrderPizza("London");
5.         }
6. }
```

解决了工厂方法模式的问题：在抽象工厂中PizzaStroe中只需要传入参数就可以实例化对象。

1.4 工厂模式适用的场合

大量的产品需要创建，并且这些产品具有共同的接口 。

1.5 三种工厂模式的使用选择

简单工厂： 用来生产同一等级结构中的任意产品。（不支持拓展增加产品）

工厂方法：用来生产同一等级结构中的固定产品。（支持拓展增加产品）

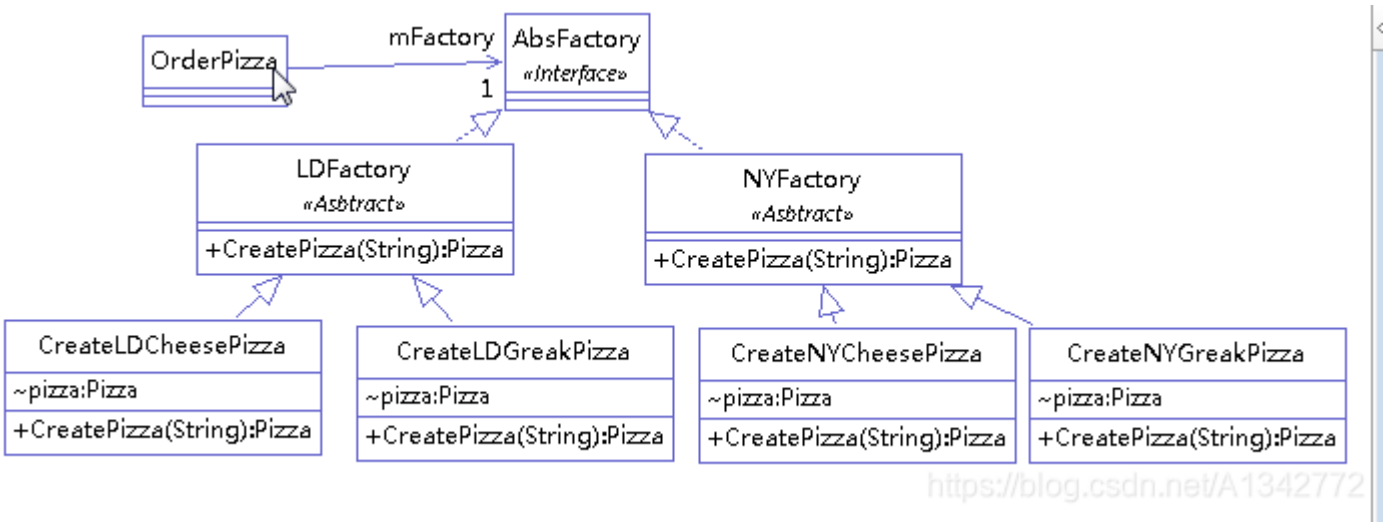
抽象工厂：用来生产不同产品族的全部产品。（支持拓展增加产品；支持增加产品族）

简单工厂的适用场合：只有伦敦工厂（只有这一个等级），并且这个工厂只生产三种类型的pizza：chesse,pepper,greak（固定产品）。

工厂方法的适用场合：现在不光有伦敦工厂，还增设了纽约工厂（仍然是同一等级结构，但是支持了产品的拓展），这两个工厂依然只生产三种类型的pizza：chesse,pepper,greak（固定产品）。

抽象工厂的适用场合：不光增设了纽约工厂（仍然是同一等级结构，但是支持了产品的拓展），这两个工厂还增加了一种新的类型的pizza：chinese pizza（增加产品族）。

所以说抽象工厂就像工厂，而工厂方法则像是工厂的一种产品生产线。因此，我们可以用抽象工厂模式创建工厂，而用工厂方法模式创建生产线。比如，我们可以使用抽象工厂模式创建伦敦工厂和纽约工厂，使用工厂方法实现cheese pizza和greak pizza的生产。类图如下：



总结一下三种模式：

简单工厂模式就是建立一个实例化对象的类，在该类中对多个对象实例化。工厂方法模式是定义了一个创建对象的抽象方法，由子类决定要实例化的类。这样做的好处是再有新的类型的对象需要实例化只要增加子类即可。抽象工厂模式定义了一个接口用于创建对象族，而无需明确指定具体类。抽象工厂也是把对象的实例化交给了子类，即支持拓展。同时提供给客户端接口，避免了用户直接操作子类工厂。

2 单例模式

定义：确保一个类最多只有一个实例，并提供一个全局访问点

单例模式可以分为两种：预加载和懒加载

2.1 预加载

顾名思义，就是预先加载。再进一步解释就是还没有使用该单例对象，但是，该单例对象就已经被加载到内存了。

```
1. public class PreloadSingleton {
2. 
3.         public static PreloadSingleton instance = new PreloadSingleton();
4. 
5.         //其他的类无法实例化单例类的对象
6.         private PreloadSingleton() {
7.         };
8. 
9.         public static PreloadSingleton getInstance() {
10.                return instance;
11.        }
12. }
```

很明显，没有使用该单例对象，该对象就被加载到了内存，会造成内存的浪费。

2.2 懒加载

为了避免内存的浪费，我们可以采用懒加载，即用到该单例对象的时候再创建。

```
1. public class Singleton {
2. 
```

```
3.         private static Singleton instance=null;
4.
5.         private Singleton(){
6.         };
7.
8.         public static Singleton getInstance()
9.         {
10.                if(instance==null)
11.                {
12.                        instance=new Singleton();
13.                }
14.                return instance;
15.
16.        }
17. }
```


2.3 单例模式和线程安全

（1）预加载只有一条语句return instance,这显然可以保证线程安全。但是，我们知道预加载会造成内存的浪费。

（2）懒加载不浪费内存，但是无法保证线程的安全。首先，if判断以及其内存执行代码是非原子性的。其次，new Singleton()无法保证执行的顺序性。

不满足原子性或者顺序性，线程肯定是不安全的，这是基本的常识，不再赘述。我主要讲一下为什么new Singleton()无法保证顺序性。我们知道创建一个对象分三步:

```
1. memory=allocate();//1:初始化内存空间
2.
3. ctorInstance(memory);//2:初始化对象
4.
5. instance=memory();//3:设置instance指向刚分配的内存地址
```

jvm为了提高程序执行性能，会对没有依赖关系的代码进行重排序，上面2和3行代码可能被重新排序。我们用两个线程来说明线程是不安全的。线程A和线程B都创建对象。其中，A2和A3的重排序，将导致线程B在B1处判断出instance不为空，线程B接下来将访问instance引用的对象。此时，线程B将会访问到一个还未初始化的对象（线程不安全）。

时间	线程A	线程B
t1	A1：分配对象的内存空间	
t2	A3：设置instance指向内存空间	
t3		B1：判断instance是否为空
t4		B2：由于instance不为null，线程B将访问instance引用的对象
t5	A2：初始化对象	
t6	A4：访问instance引用的对象	

2.4 保证懒加载的线程安全

我们首先想到的就是使用synchronized关键字。synchronized加载getInstace()函数上确实保证了线程的安全。但是，如果要经常的调用getInstance()方法，不管有没有初始化实例，都会唤醒和阻塞线程。为了避免线程的上下文切换消耗大量时间，如果对象已经实例化了，我们没有必要再使用synchronized加锁，直接返回对象。

```
1. public class Singleton {
2.         private static Singleton instance = null;
3.         private Singleton() {
4.         };
5.         public static synchronized Singleton getInstance() {
6.                 if (instance == null) {
7.                         instance = new Singleton();
8.                 }
9.                 return instance;
10.        }
```

```
}
```

我们把sychronized加在if(instance==null)判断语句里面，**保证instance未实例化的时候才加锁**

```
public class Singleton {
        private static Singleton instance = null;
        private Singleton() {
        };
        public static synchronized Singleton getInstance() {
                if (instance == null) {
                        synchronized (Singleton.class) {
                                if (instance == null) {
                                        instance = new Singleton();
                                }
                        }
                }
                return instance;
        }
}
```

我们经过2.3的讨论知道new一个对象的代码是无法保证顺序性的，因此，我们需要使用另一个关键字volatile保证对象实例化过程的顺序性。

```
public class Singleton {
        private static volatile Singleton instance = null;
        private Singleton() {
        };
        public static synchronized Singleton getInstance() {
                if (instance == null) {
                        synchronized (instance) {
                                if (instance == null) {
```

```
                                instance = new Singleton();
                            }
                        }
                }
                return instance;
        }
}
```

到此，我们就保证了懒加载的线程安全。

3 生成器模式

定义：封装一个复杂对象构造过程，并允许按步骤构造。

定义解释： 我们可以将生成器模式理解为，假设我们有一个对象需要建立，这个对象是由多个组件（Component）组合而成，每个组件的建立都比较复杂，但运用组件来建立所需的对象非常简单，所以我们就可以将构建复杂组件的步骤与运用组件构建对象分离，使用builder模式可以建立。

3.1 模式的结构和代码示例

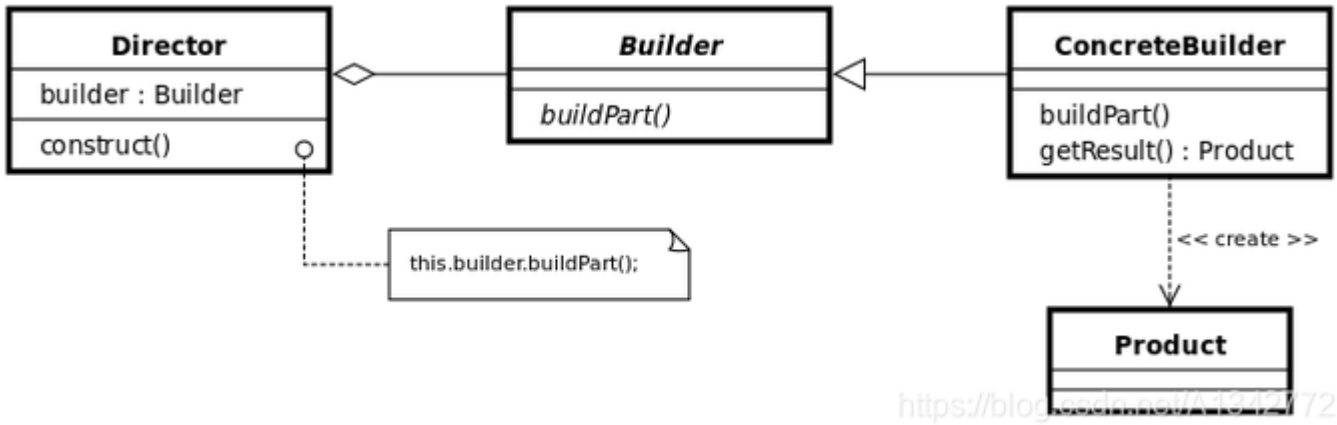
生成器模式结构中包括四种角色：

（1）产品(Product)：具体生产器要构造的复杂对象；

（2）抽象生成器(Bulider)：抽象生成器是一个接口，该接口除了为创建一个Product对象的各个组件定义了若干个方法之外，还要定义返回Product对象的方法（**定义构造步骤**）；

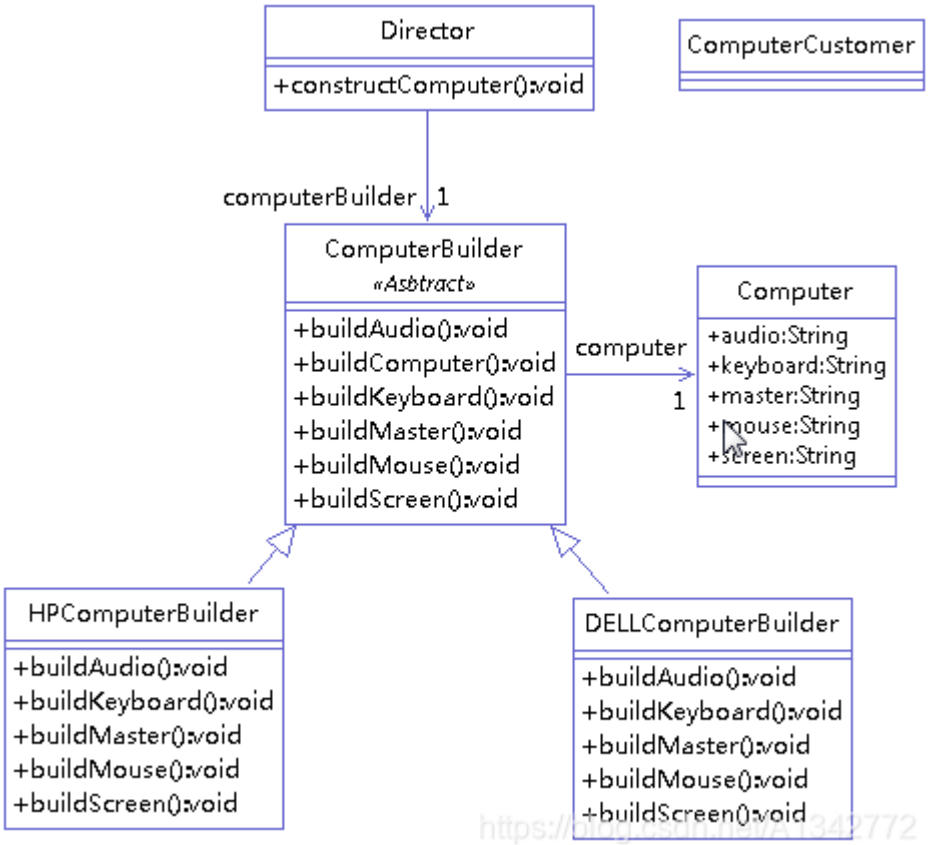
（3）具体生产器(ConcreteBuilder)：实现Builder接口的类，具体生成器将实现Builder接口所定义的方法（**生产各个组件**）；

（4）指挥者(Director)：指挥者是一个类，该类需要含有Builder接口声明的变量。指挥者的职责是负责向用户提供具体生成器，即指挥者将请求具体生成器类来构造用户所需要的Product对象，如果所请求的具体生成器成功地构造出Product对象，指挥者就可以让该具体生产器返回所构造的Product对象。（**按照步骤组装部件，并返回Product**）



举例（我们如果构建生成一台电脑，那么我们可能需要这么几个步骤（1）需要一个主机（2）需要一个显示器（3）需要一个键盘（4）需要一个鼠标）

虽然我们具体在构建一台主机的时候，每个对象的实际步骤是不一样的，比如，有的对象构建了i7cpu的主机，有的对象构建了i5cpu的主机，有的对象构建了普通键盘，有的对象构建了机械键盘等。但不管怎样，你总是需要经过一个步骤就是构建一台主机，一台键盘。对于这个例子，我们就可以使用生成器模式来生成一台电脑，他需要通过多个步骤来生成。类图如下：



ComputerBuilder类定义构造步骤：

```
1. public abstract class ComputerBuilder {
2.
3.     protected Computer computer;
4.
5.     public Computer getComputer() {
6.         return computer;
7.     }
8.
9.     public void buildComputer() {
10.         computer = new Computer();
```

```
11.         System.out.println("生成了一台电脑！！！");
12.     }
13.     public abstract void buildMaster();
14.     public abstract void buildScreen();
15.     public abstract void buildKeyboard();
16.     public abstract void buildMouse();
17.     public abstract void buildAudio();
18. }
```

HPComputerBuilder定义各个组件：

```
1. public class HPComputerBuilder extends ComputerBuilder {
2.     @Override
3.     public void buildMaster() {
4.         // TODO Auto-generated method stub
5.         computer.setMaster("i7,16g,512SSD,1060");
6.         System.out.println("(i7,16g,512SSD,1060)的惠普主机");
7.     }
8.     @Override
9.     public void buildScreen() {
10.         // TODO Auto-generated method stub
11.         computer.setScreen("1080p");
12.         System.out.println("(1080p)的惠普显示屏");
13.     }
14.     @Override
15.     public void buildKeyboard() {
16.         // TODO Auto-generated method stub
17.         computer.setKeyboard("cherry 青轴机械键盘");
18.         System.out.println("(cherry 青轴机械键盘)的键盘");
```

```
    }
    @Override
    public void buildMouse() {
        // TODO Auto-generated method stub
        computer.setMouse("MI 鼠标");
        System.out.println("(MI 鼠标)的鼠标");
    }
    @Override
    public void buildAudio() {
        // TODO Auto-generated method stub
        computer.setAudio("飞利浦 音响");
        System.out.println("(飞利浦 音响)的音响");
    }
}
```

Director类对组件进行组装并生成产品

```
public class Director {

    private ComputerBuilder computerBuilder;
    public void setComputerBuilder(ComputerBuilder computerBuilder) {
        this.computerBuilder = computerBuilder;
    }

    public Computer getComputer() {
        return computerBuilder.getComputer();
    }

    public void constructComputer() {
```

```
13.         computerBuilder.buildComputer();
14.         computerBuilder.buildMaster();
15.         computerBuilder.buildScreen();
16.         computerBuilder.buildKeyboard();
17.         computerBuilder.buildMouse();
18.         computerBuilder.buildAudio();
19.     }
20. }
```

3.2 生成器模式的优缺点

优点

- 将一个对象分解为各个组件
- 将对象组件的构造封装起来
- 可以控制整个对象的生成过程

缺点

- 对不同类型的对象需要实现不同的具体构造器的类，这可能回答大大增加类的数量

3.3 生成器模式与工厂模式的不同

生成器模式构建对象的时候，对象通常构建的过程中需要多个步骤，就像我们例子中的先有主机，再有显示屏，再有鼠标等等，生成器模式的作用就是将这些复杂的构建过程封装起来。工厂模式构建对象的时候通常就只有一个步骤，调用一个工厂方法就可以生成一个对象。

4 原型模式

定义：通过复制现有实例来创建新的实例，无需知道相应类的信息。

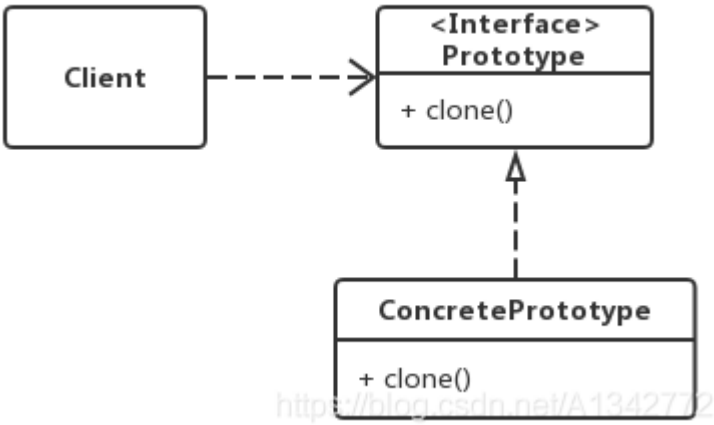
简单地理解，其实就是当需要创建一个指定的对象时，我们刚好有一个这样的对象，但是又不能直接使用，我会clone一个一毛一样的新对象来使用；基本上这就是原型模式。关键字：**Clone。**

4.1 深拷贝和浅拷贝

浅复制：将一个对象复制后，基本数据类型的变量都会重新创建，而引用类型，指向的还是原对象所指向的。

深复制：将一个对象复制后，不论是基本数据类型还有引用类型，都是重新创建的。简单来说，就是深复制进行了完全彻底的复制，而浅复制不彻底。clone明显是深复制，clone出来的对象是是不能去影响原型对象的

4.2 原型模式的结构和代码示例



Client：使用者

Prototype：接口（抽象类），声明具备clone能力，例如java中得Cloneable接口

ConcretePrototype：具体的原型类

可以看出设计模式还是比较简单的，重点在于Prototype接口和Prototype接口的实现类ConcretePrototype。原型模式的具体实现：一个原型类，只需要实现Cloneable接口，覆写clone方法，此处clone方法可以改成任意的名称，因为Cloneable接口是个空接口，你可以任意定义实现类的方法名，如cloneA或者cloneB，因为此处的重点是super.clone()这句话，super.clone()调用的是Object的clone()方法。

```
1. public class Prototype implements Cloneable {
2.       public Object clone() throws CloneNotSupportedException {
3.           Prototype proto = (Prototype) super.clone();
4.           return proto;
5.       }
6. }
```

举例（银行发送大量邮件，使用clone和不使用clone的时间对比**）：** 我们模拟创建一个对象需要耗费比较长的时间，因此，在构造函数中我们让当前线程sleep一会

```
1. public Mail(EventTemplate et) {
2.                this.tail = et.geteventContent();
3.                this.subject = et.geteventSubject();
4.                try {
5.                       Thread.sleep(1000);
6.                } catch (InterruptedException e) {
```

```
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }
```

不使用clone,发送十个邮件

```
public static void main(String[] args) {
                int i = 0;
                int MAX_COUNT = 10;
                EventTemplate et = new EventTemplate("9月份信用卡账单", "国庆抽奖活动...");
                long start = System.currentTimeMillis();
                while (i < MAX_COUNT) {
                        // 以下是每封邮件不同的地方
                        Mail mail = new Mail(et);
                        mail.setContent(getRandString(5) + ",先生（女士）:你的信用卡账单..." + mail.getTail());
                        mail.setReceiver(getRandString(5) + "@" + getRandString(8) + ".com");
                        // 然后发送邮件
                        sendMail(mail);
                        i++;
                }
                long end = System.currentTimeMillis();
                System.out.println("用时:" + (end - start));
        }
```

用时：10001

使用clone,发送十个邮件

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

```
        int i = 0;
        int MAX_COUNT = 10;
        EventTemplate et = new EventTemplate("9月份信用卡账单", "国庆抽奖活动...");
        long start=System.currentTimeMillis();
        Mail mail = new Mail(et);
        while (i < MAX_COUNT) {
            Mail cloneMail = mail.clone();
            mail.setContent(getRandString(5) + ",先生（女士）:你的信用卡账单..."
                    + mail.getTail());
            mail.setReceiver(getRandString(5) + "@" + getRandString(8) + ".com");
            sendMail(cloneMail);
            i++;
        }
        long end=System.currentTimeMillis();
        System.out.println("用时:"+(end-start));
    }
```

用时：1001

4.3 总结

原型模式的本质就是clone，可以解决构建复杂对象的资源消耗问题，能再某些场景中提升构建对象的效率；还有一个重要的用途就是保护性拷贝，可以通过返回一个拷贝对象的形式，实现只读的限制。

B、结构模式（7种）

适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

5 适配器模式

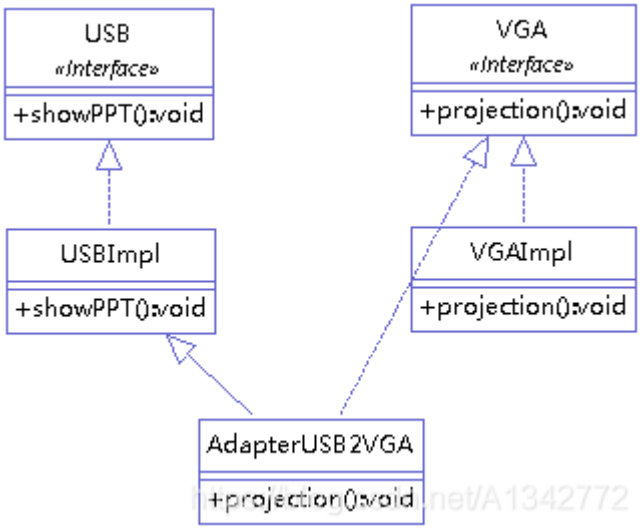
定义： 适配器模式将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。

主要分为三类：类的适配器模式、对象的适配器模式、接口的适配器模式。

5.1 类适配器模式

通过多重**继承**目标接口和被适配者类方式来实现适配

举例(将USB接口转为VGA接口)，类图如下：



USBImpl的代码：

```
1. public class USBImpl implements USB{
2.         @Override
3.         public void showPPT() {
4.                 // TODO Auto-generated method stub
5.                 System.out.println("PPT内容演示");
6.         }
7. }
```

AdatperUSB2VGA 首先继承USBImpl获取USB的功能，其次，实现VGA接口，表示该类的类型为VGA。

```
1. public class AdapterUSB2VGA extends USBImpl implements VGA {
2.         @Override
3.         public void projection() {
4.                 super.showPPT();
5.         }
6. }
```

Projector将USB映射为VGA，只有VGA接口才可以连接上投影仪进行投影

```
public class Projector<T> {
        public void projection(T t) {
                if (t instanceof VGA) {
                        System.out.println("开始投影");
                        VGA v = new VGAImpl();
                        v = (VGA) t;
                        v.projection();
                } else {
                        System.out.println("接口不匹配，无法投影");
                }
        }
}
```

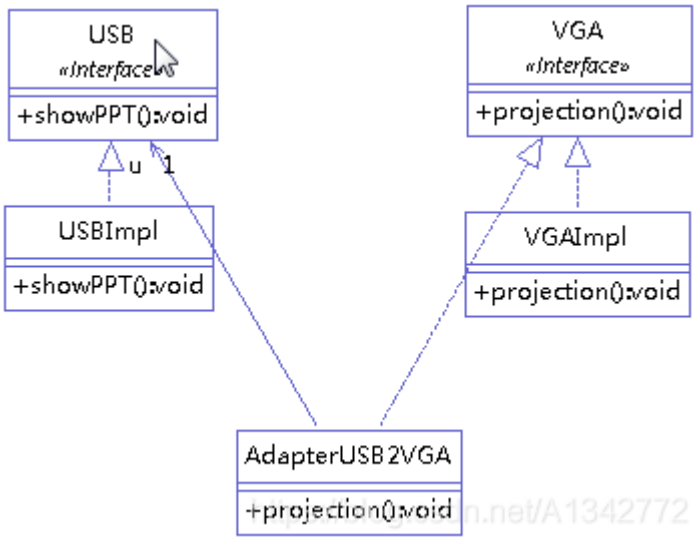
test代码

```
        @Test
        public void test2(){
                //通过适配器创建一个VGA对象，这个适配器实际是使用的是USB的showPPT（）方法
                VGA a=new AdapterUSB2VGA();
                //进行投影
                Projector p1=new Projector();
                p1.projection(a);
        }
```

5.2 对象适配器模式

对象适配器和类适配器使用了不同的方法实现适配，对象适配器使用组合，类适配器使用继承。

举例(将USB接口转为VGA接口)，类图如下:



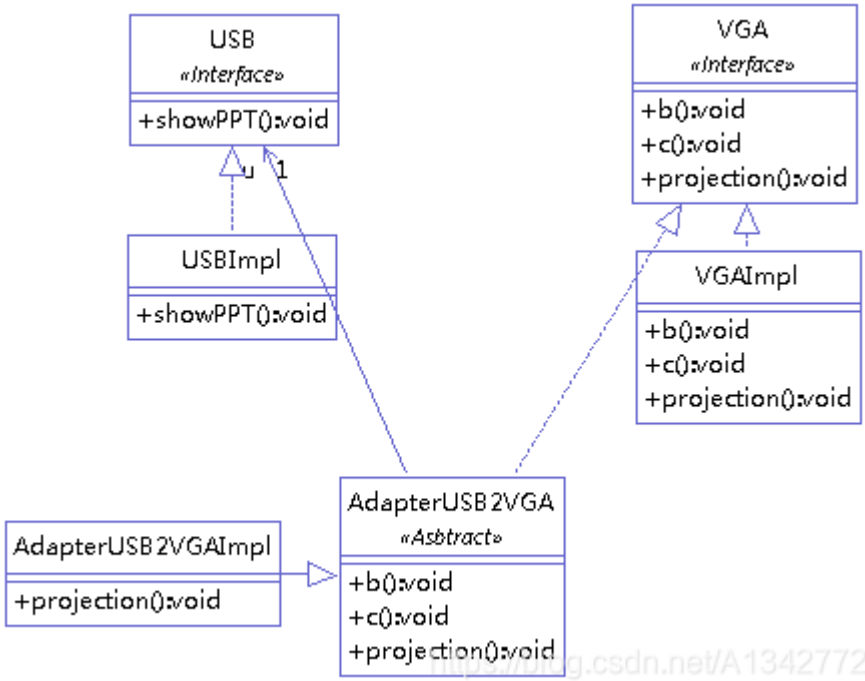

```
1. public class AdapterUSB2VGA implements VGA {
2.         USB u = new USBImpl();
3.         @Override
4.         public void projection() {
5.                 u.showPPT();
6.         }
7. }
```

实现VGA接口，表示适配器类是VGA类型的，适配器方法中直接使用USB对象。

5.3 接口适配器模式

当不需要全部实现接口提供的方法时，可先设计一个抽象类实现接口，并为该接口中每个方法提供一个默认实现（空方法），那么该抽象类的子类可有选择地覆盖父类的某些方法来实现需求，**它适用于一个接口不想使用其所有的方法的情况。**

举例(将USB接口转为VGA接口，VGA中的b()和c()不会被实现)，类图如下：



AdapterUSB2VGA抽象类

```
1. public abstract class AdapterUSB2VGA implements VGA {
2.         USB u = new USBImpl();
3.         @Override
4.         public void projection() {
5.                 u.showPPT();
6.         }
7.         @Override
8.         public void b() {
9.         };
10.        @Override
11.        public void c() {
12.        };
13. }
```

AdapterUSB2VGA实现，不用去实现b()和c()方法。

```
1. public class AdapterUSB2VGAImpl extends AdapterUSB2VGA {
2.         public void projection() {
```

```
3.                 super.projection();
4.         }
5. }
```

5.4 总结

总结一下三种适配器模式的应用场景：

类适配器模式：当希望将**一个类**转换成满足另一个新接口的类时，可以使用类的适配器模式，创建一个新类，继承原有的类，实现新的接口即可。

对象适配器模式：当希望将一个对象转换成满足另一个新接口的对象时，可以创建一个Wrapper类，持有原类的一个实例，在Wrapper类的方法中，调用实例的方法就行。

接口适配器模式：当不希望实现一个接口中所有的方法时，可以创建一个抽象类Wrapper，实现所有方法，我们写别的类的时候，继承抽象类即可。

命名规则：

我个人理解，三种命名方式，是根据 src是以怎样的形式给到Adapter（在Adapter里的形式）来命名的。

类适配器，以类给到，在Adapter里，就是将src当做类，继承，

对象适配器，以对象给到，在Adapter里，将src作为一个对象，持有。

接口适配器，以接口给到，在Adapter里，将src作为一个接口，实现。

使用选择：

根据合成复用原则，组合大于继承。因此，类的适配器模式应该少用。

6 装饰者模式

定义：动态的将新功能附加到对象上。在对象功能扩展方面，它比继承更有弹性。

6.1 装饰者模式结构图与代码示例

1.Component（被装饰对象的基类）

定义一个对象接口，可以给这些对象动态地添加职责。

2.ConcreteComponent（具体被装饰对象）

定义一个对象，可以给这个对象添加一些职责。

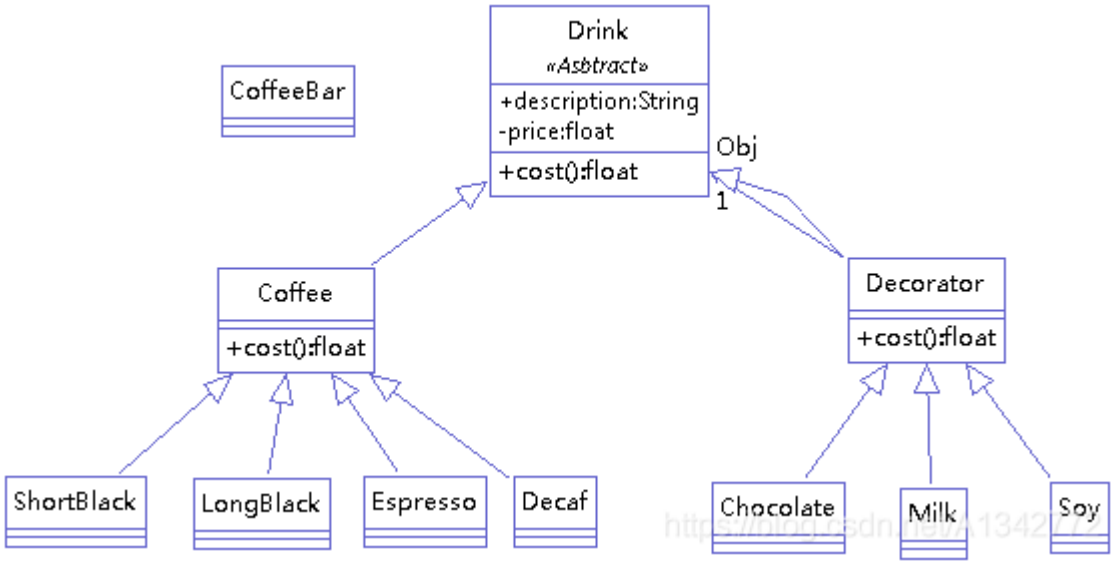
3.Decorator（装饰者抽象类）

维持一个指向Component实例的引用，并定义一个与Component接口一致的接口。

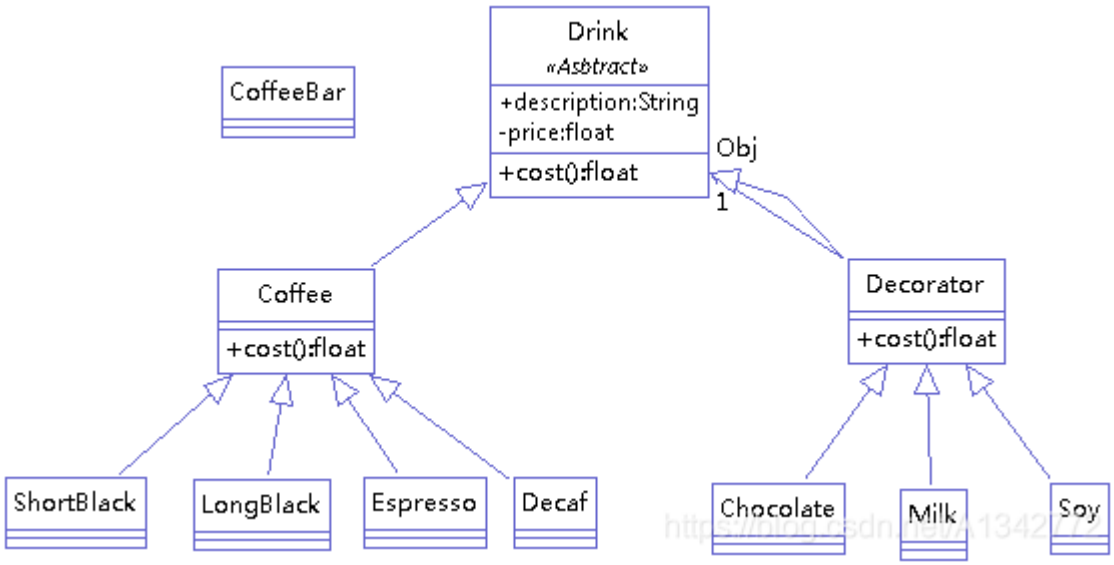
4.ConcreteDecorator（具体装饰者）

具体的装饰对象，给内部持有的具体被装饰对象，增加具体的职责。

被装饰对象和修饰者继承自同一个超类



举例(咖啡馆订单项目：1）、咖啡种类：Espresso、ShortBlack、LongBlack、Decaf2）、**调料（装饰者）**：Milk、Soy、Chocolate)，类图如下：



被装饰的对象和装饰者都继承自同一个超类

```
1. public abstract class Drink {
2.         public String description="";
3.         private float price=0f;;
4.
5.
6.         public void setDescription(String description)
```

```
7.          {
8.                  this.description=description;
9.          }
10.
11.         public String getDescription()
12.         {
13.                 return description+"-"+this.getPrice();
14.         }
15.         public float getPrice()
16.         {
17.                 return price;
18.         }
19.         public void setPrice(float price)
20.         {
21.                 this.price=price;
22.         }
23.         public abstract float cost();
24.
25. }
```

被装饰的对象，不用去改造。原来怎么样写，现在还是怎么写。

```
1. public  class Coffee extends Drink {
2.         @Override
3.         public float cost() {
4.                 // TODO Auto-generated method stub
5.                 return super.getPrice();
6.         }
7.
```

```
8. }
```

coffee类的实现

```
1. public class Decaf extends Coffee {
2.         public Decaf()
3.         {
4.                 super.setDescription("Decaf");
5.                 super.setPrice(3.0f);
6.         }
7. }
```

装饰者

装饰者不仅要考虑自身，还要考虑被它修饰的对象，它是在被修饰的对象上继续添加修饰。例如，咖啡里面加牛奶，再加巧克力。加糖后价格为coffee+milk。再加牛奶价格为coffee+milk+chocolate。

```
1. public class Decorator extends Drink {
2.         private Drink Obj;
3.         public Decorator(Drink Obj) {
4.                 this.Obj = Obj;
5.         };
6.         @Override
7.         public float cost() {
8.                 // TODO Auto-generated method stub
9.                 return super.getPrice() + Obj.cost();
10.        }
11.        @Override
12.        public String getDescription() {
13.                return super.description + "-" + super.getPrice() + "&&" +
    Obj.getDescription();
14.        }
```

```
}
```

装饰者实例化（加牛奶）。这里面要对被修饰的对象进行实例化。

```
public class Milk extends Decorator {
        public Milk(Drink Obj) {
                super(Obj);
                // TODO Auto-generated constructor stub
                super.setDescription("Milk");
                super.setPrice(2.0f);
        }
}
```

coffee店：初始化一个被修饰对象，修饰者实例需要对被修改者实例化，才能对具体的被修饰者进行修饰。

```
public class CoffeeBar {
        public static void main(String[] args) {
                Drink order;
                order = new Decaf();
                System.out.println("order1 price:" + order.cost());
                System.out.println("order1 desc:" + order.getDescription());
                System.out.println("***************");
                order = new LongBlack();
                order = new Milk(order);
                order = new Chocolate(order);
                order = new Chocolate(order);
                System.out.println("order2 price:" + order.cost());
                System.out.println("order2 desc:" + order.getDescription());
        }
}
```

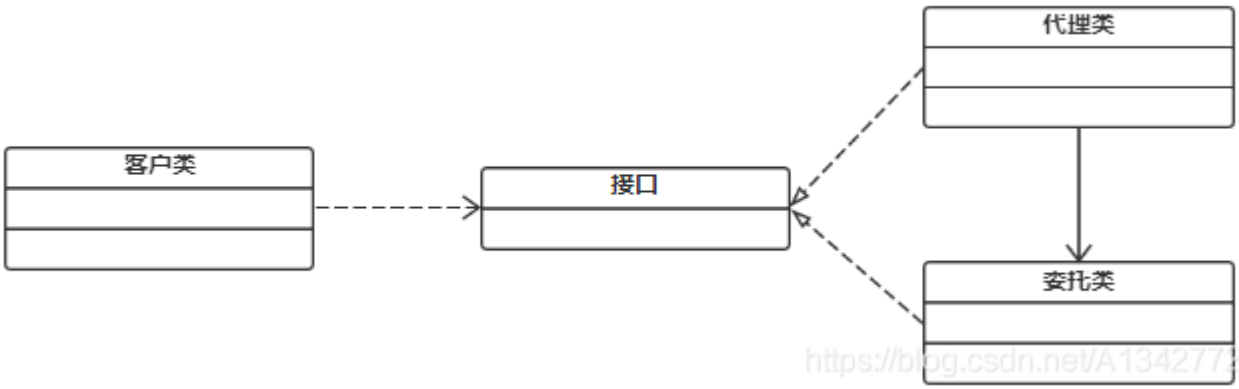
6.2 总结

装饰者和被装饰者之间必须是一样的类型,也就是要有共同的超类。在这里应用继承并不是实现方法的复制,而是实现类型的匹配。因为装饰者和被装饰者是同一个类型,因此装饰者可以取代被装饰者,这样就使被装饰者拥有了装饰者独有的行为。根据装饰者模式的理念,我们可以在任何时候,实现新的装饰者增加新的行为。如果是用继承,每当需要增加新的行为时,就要修改原程序了。

7 代理模式

定义：代理模式给某一个对象提供一个代理对象，并由代理对象控制对原对象的引用。通俗的来讲代理模式就是我们生活中常见的中介。

举个例子来说明：假如说我现在想买一辆二手车，虽然我可以自己去找车源，做质量检测等一系列的车辆过户流程，但是这确实太浪费我得时间和精力了。我只是想买一辆车而已为什么我还要额外做这么多事呢？于是我就通过中介公司来买车，他们来给我找车源，帮我办理车辆过户流程，我只是负责选择自己喜欢的车，然后付钱就可以了。用图表示如下：



7.1 为什么要用代理模式?

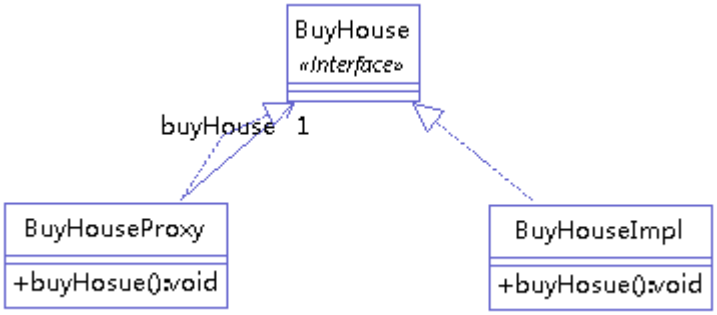
中介隔离作用：在某些情况下，一个客户类不想或者不能直接引用一个委托对象，而代理类对象可以在客户类和委托对象之间起到中介的作用，其特征是代理类和委托类实现相同的接口。

开闭原则，增加功能：代理类除了是客户类和委托类的中介之外，我们还可以通过给代理类增加额外的功能来扩展委托类的功能，这样做我们只需要修改代理类而不需要再修改委托类，符合代码设计的开闭原则。代理类主要负责为委托类预处理消息、过滤消息、把消息转发给委托类，以及事后对返回结果的处理等。代理类本身并不真正实现服务，而是同过调用委托类的相关方法，来提供特定的服务。真正的业务功能还是由委托类来实现，但是可以在业务功能执行的前后加入一些公共的服务。例如我们想给项目加入缓存、日志这些功能，我们就可以使用代理类来完成，而没必要打开已经封装好的委托类。

代理模式分为三类：1. 静态代理 2. 动态代理 3. CGLIB代理

7.2 静态代理

举例(买房），类图如下：



第一步：创建服务类接口

```
1. public interface BuyHouse {
2.     void buyHosue();
3. }
```

第二步：实现服务接口

```
1. public class BuyHouseImpl implements BuyHouse {
2.         @Override
3.         public void buyHosue() {
4.                 System.out.println("我要买房");
5.         }
6. }
```

第三步：创建代理类

```
1. public class BuyHouseProxy implements BuyHouse {
2.         private BuyHouse buyHouse;
3.         public BuyHouseProxy(final BuyHouse buyHouse) {
4.                 this.buyHouse = buyHouse;
5.         }
6.         @Override
7.         public void buyHosue() {
8.                 System.out.println("买房前准备");
```

```
            buyHouse.buyHosue();
            System.out.println("买房后装修");
        }
}
```

总结：

优点：可以做到在符合开闭原则的情况下对目标对象进行功能扩展。

缺点： **代理对象与目标对象要实现相同的接口，我们得为每一个服务都得创建代理类，工作量太大**，不易管理。同时接口一旦发生改变，代理类也得相应修改。

7.3 动态代理

动态代理有以下特点:

1.代理对象,不需要实现接口

2.代理对象的生成,是利用JDK的API,动态的在内存中构建代理对象(需要我们指定创建代理对象/目标对象实现的接口的类型)

代理类不用再实现接口了。但是，要求被代理对象必须有接口。

动态代理实现：

Java.lang.reflect.Proxy类可以直接生成一个代理对象

- Proxy.newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)生成一个代理对象
    - 参数1:ClassLoader loader 代理对象的类加载器 一般使用被代理对象的类加载器
    - 参数2:Class<?>[] interfaces 代理对象的要实现的接口 一般使用的被代理对象实现的接口
    - 参数3:InvocationHandler h (接口)执行处理类
- InvocationHandler中的invoke(Object proxy, Method method, Object[] args)方法：调用代理类的任何方法，此方法都会执行
    - 参数3.1:代理对象(慎用)
    - 参数3.2:当前执行的方法
    - 参数3.3:当前执行的方法运行时传递过来的参数

第一步：编写动态处理器

```
public class DynamicProxyHandler implements InvocationHandler {
        private Object object;
```

```
3.         public DynamicProxyHandler(final Object object) {
4.                 this.object = object;
5.         }
6.         @Override
7.         public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
8.                 System.out.println("买房前准备");
9.                 Object result = method.invoke(object, args);
10.                System.out.println("买房后装修");
11.                return result;
12.        }
13. }
```

第二步：编写测试类

```
1. public class DynamicProxyTest {
2.     public static void main(String[] args) {
3.         BuyHouse buyHouse = new BuyHouseImpl();
4.         BuyHouse proxyBuyHouse = (BuyHouse)
   Proxy.newProxyInstance(BuyHouse.class.getClassLoader(), new
5.                 Class[]{BuyHouse.class}, new DynamicProxyHandler(buyHouse));
6.         proxyBuyHouse.buyHosue();
7.     }
8. }
```

动态代理总结：虽然相对于静态代理，动态代理大大减少了我们的开发任务，同时减少了对业务接口的依赖，降低了耦合度。但是还是有一点点小小的遗憾之处，那就是它始终无法摆脱仅支持interface代理的桎梏（我们要使用被代理的对象的接口），因为它的设计注定了这个遗憾。

7.4 CGLIB代理

CGLIB 原理：动态生成一个要代理类的子类，子类重写要代理的类的所有不是final的方法。在子类中采用方法拦截的技术拦截所有父类方法的调用，顺势织入横切逻辑。它比使用java反射的JDK动态代理要快。

CGLIB 底层：使用字节码处理框架ASM，来转换字节码并生成新的类。不鼓励直接使用ASM，因为它要求你必须对JVM内部结构包括class文件的格式和指令集都很熟悉。

CGLIB缺点：对于final方法，无法进行代理。

CGLIB的实现步骤：

第一步：建立拦截器

```
1. public Object intercept(Object object, Method method, Object[] args, MethodProxy
   methodProxy) throws Throwable {
2.
3.         System.out.println("买房前准备");
4.
5.         Object result = methodProxy.invoke(object, args);
6.
7.         System.out.println("买房后装修");
8.
9.         return result;
10.
11.     }
```

参数：Object为由CGLib动态生成的代理类实例，Method为上文中实体类所调用的被代理的方法引用，Object[]为参数值列表，MethodProxy为生成的代理类对方法的代理引用。

返回：从代理实例的方法调用返回的值。

其中，**proxy.invokeSuper(obj,arg)** 调用代理类实例上的proxy方法的父类方法（即实体类TargetObject中对应的方法）

第二步： 生成动态代理类

```
1. public class CglibProxy implements MethodInterceptor {
2.     private Object target;
3.     public Object getInstance(final Object target) {
4.         this.target = target;
5.         Enhancer enhancer = new Enhancer();
6.         enhancer.setSuperclass(this.target.getClass());
7.         enhancer.setCallback(this);
```

```
        return enhancer.create();
    }
    public Object intercept(Object object, Method method, Object[] args, MethodProxy methodProxy) throws Throwable {
        System.out.println("买房前准备");
        Object result = methodProxy.invoke(object, args);
        System.out.println("买房后装修");
        return result;
    }
}
```

这里Enhancer类是CGLib中的一个字节码增强器，它可以方便的对你想要处理的类进行扩展，以后会经常看到它。

首先将被代理类TargetObject设置成父类，然后设置拦截器TargetInterceptor，最后执行enhancer.create()动态生成一个代理类，并从Object强制转型成父类型TargetObject。

第三步：测试

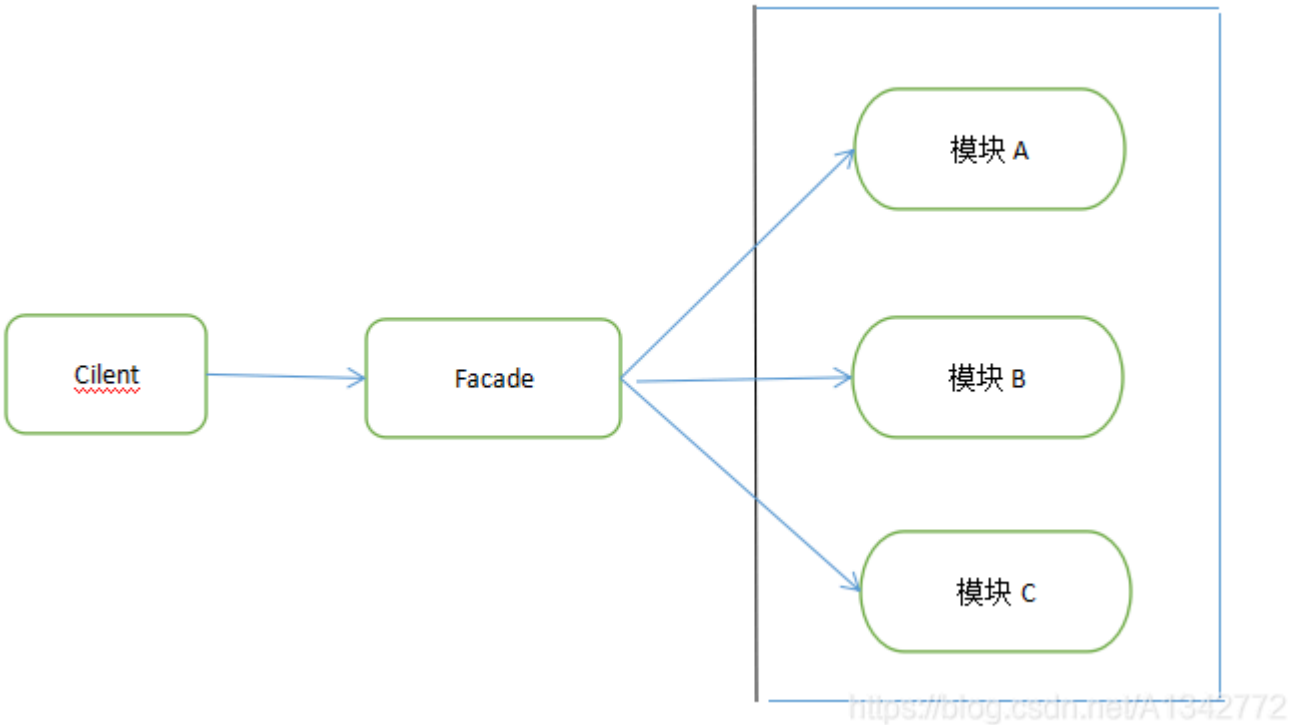
```
public class CglibProxyTest {
    public static void main(String[] args){
        BuyHouse buyHouse = new BuyHouseImpl();
        CglibProxy cglibProxy = new CglibProxy();
        BuyHouseImpl buyHouseCglibProxy = (BuyHouseImpl) cglibProxy.getInstance(buyHouse);
        buyHouseCglibProxy.buyHosue();
    }
}
```

CGLIB代理总结： **CGLIB创建的动态代理对象比JDK创建的动态代理对象的性能更高，但是CGLIB创建代理对象时所花费的时间却比JDK多得多。所以对于单例的对象，因为无需频繁创建对象，用CGLIB合适，反之使用JDK方式要更为合适一些。同时由于CGLib由于是采用动态创建子类的方法，对于final修饰的方法无法进行代理。**

8 外观模式

定义： 隐藏了系统的复杂性，并向客户端提供了一个可以访问系统的接口。

8.1 模式结构和代码示例



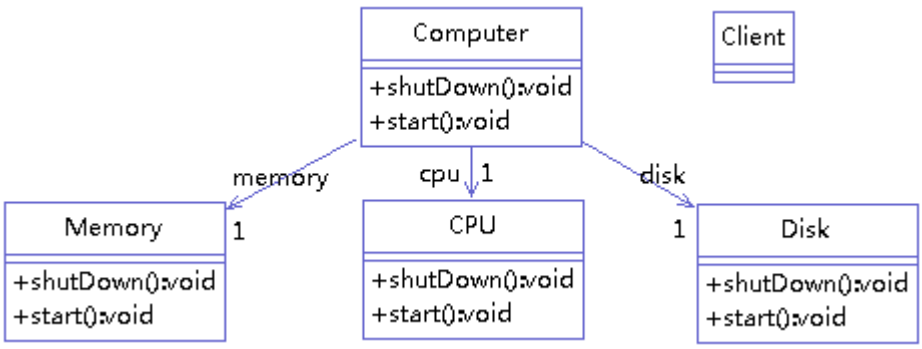
简单来说，该模式就是把一些复杂的流程封装成一个接口供给外部用户更简单的使用。这个模式中，设计到3个角色。

1）.门面角色：外观模式的核心。它被客户角色调用，它熟悉子系统的功能。内部根据客户角色的需求预定了几种功能的组合。（**客户调用，同时自身调用子系统功能**）

2）.子系统角色:实现了子系统的功能。它对客户角色和Facade时未知的。它内部可以有系统内的相互交互，也可以由供外界调用的接口。（实现具体功能）

3）.客户角色:通过调用Facede来完成要实现的功能（**调用门面角色**）。

举例（每个Computer都有CPU、Memory、Disk。在Computer开启和关闭的时候，相应的部件也会开启和关闭），类图如下：



首先是子系统类：

```
1. public class CPU {
```

```
2.
3.          public void start() {
4.                  System.out.println("cpu is start...");
5.          }
6.
7.          public void shutDown() {
8.                  System.out.println("CPU is shutDown...");
9.          }
10. }
11.
12. public class Disk {
13.          public void start() {
14.                  System.out.println("Disk is start...");
15.          }
16.
17.          public void shutDown() {
18.                  System.out.println("Disk is shutDown...");
19.          }
20. }
21.
22. public class Memory {
23.          public void start() {
24.                  System.out.println("Memory is start...");
25.          }
26.
27.          public void shutDown() {
28.                  System.out.println("Memory is shutDown...");
29.          }
30. }
```

然后是，门面类Facade

```
public class Computer {

        private CPU cpu;
        private Memory memory;
        private Disk disk;

        public Computer() {
                cpu = new CPU();
                memory = new Memory();
                disk = new Disk();
        }

        public void start() {
                System.out.println("Computer start begin");
                cpu.start();
                disk.start();
                memory.start();
                System.out.println("Computer start end");
        }

        public void shutDown() {
                System.out.println("Computer shutDown begin");
                cpu.shutDown();
                disk.shutDown();
                memory.shutDown();
                System.out.println("Computer shutDown end...");
```

```
27.         }
28. }
```

最后为，客户角色

```
1. public class Client {
2.
3.         public static void main(String[] args) {
4.                 Computer computer = new Computer();
5.                 computer.start();
6.                 System.out.println("=================");
7.                 computer.shutDown();
8.         }
9.
10. }
```

8.2 优点

- 松散耦合

使得客户端和子系统之间解耦，让子系统内部的模块功能更容易扩展和维护；

- 简单易用

客户端根本不需要知道子系统内部的实现，或者根本不需要知道子系统内部的构成，它只需要跟Facade类交互即可。

- 更好的划分访问层次

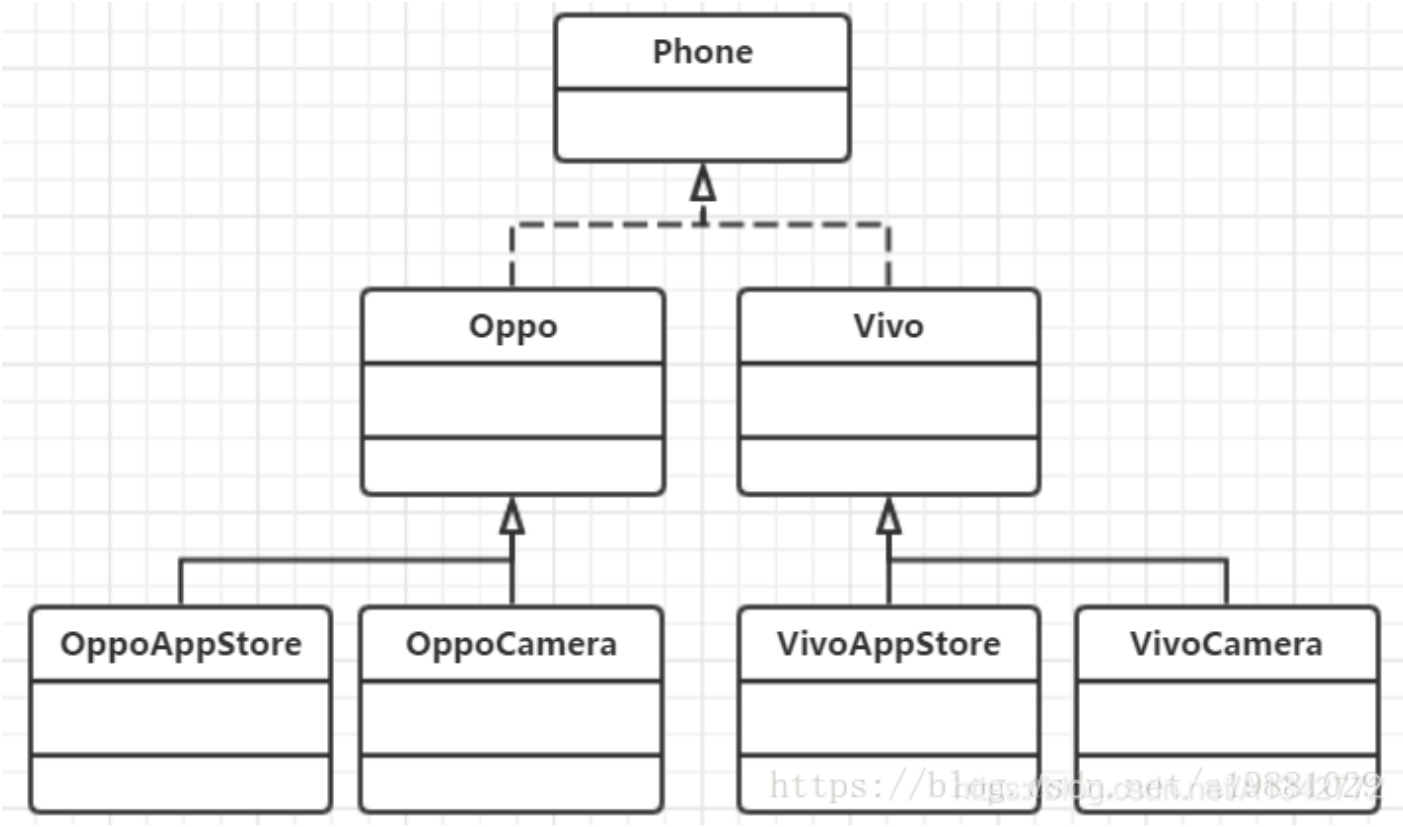
有些方法是对系统外的，有些方法是系统内部相互交互的使用的。子系统把那些暴露给外部的功能集中到门面中，这样就可以实现客户端的使用，很好的隐藏了子系统内部的细节。

9 桥接模式

定义： 将抽象部分与它的实现部分分离，使它们都可以独立地变化。

9.1 案例

看下图手机与手机软件的类图



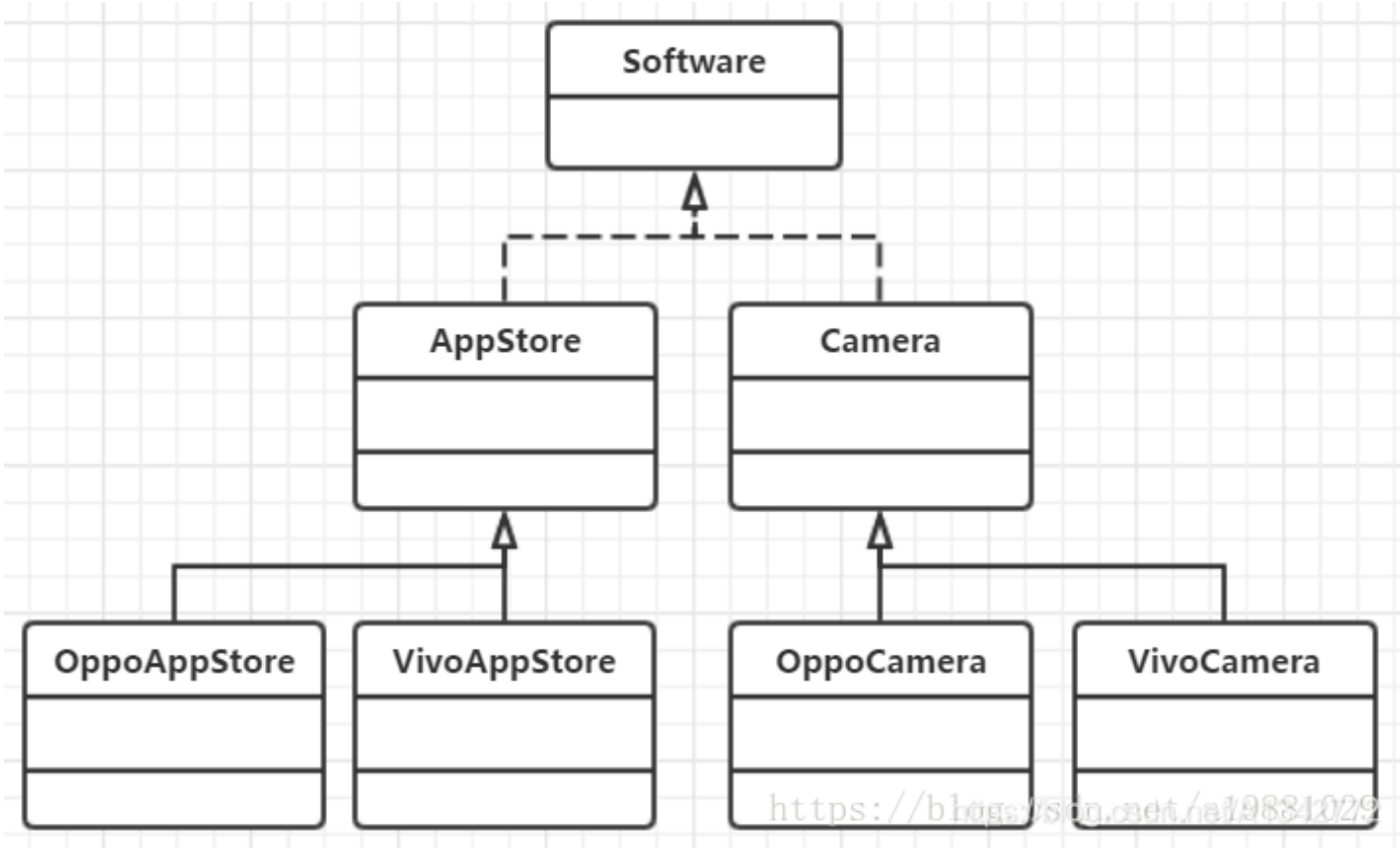
增加一款新的手机软件，需要在所有手机品牌类下添加对应的手机软件类，当手机软件种类较多时，将导致类的个数急剧膨胀，难以维护

手机和手机中的软件是什么关系?

手机中的软件从本质上来说并不是一种手机，手机软件运行在手机中，是一种包含与被包含关系，而不是一种父与子或者说一般与特殊的关系，通过继承手机类实现手机软件类的设计是违反一般规律的。

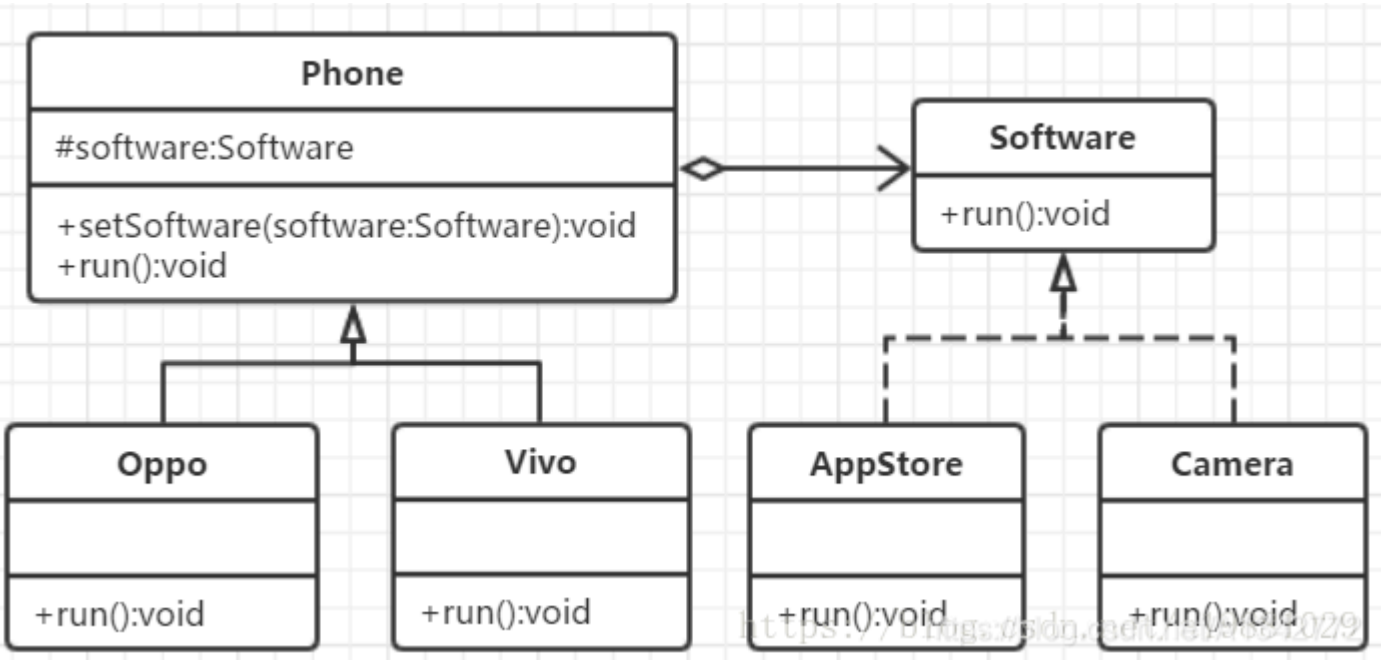
如果Oppo手机实现了wifi功能，继承它的Oppo应用商城也会继承wifi功能，并且Oppo手机类的任何变动，都会影响其子类

换一种解决思路

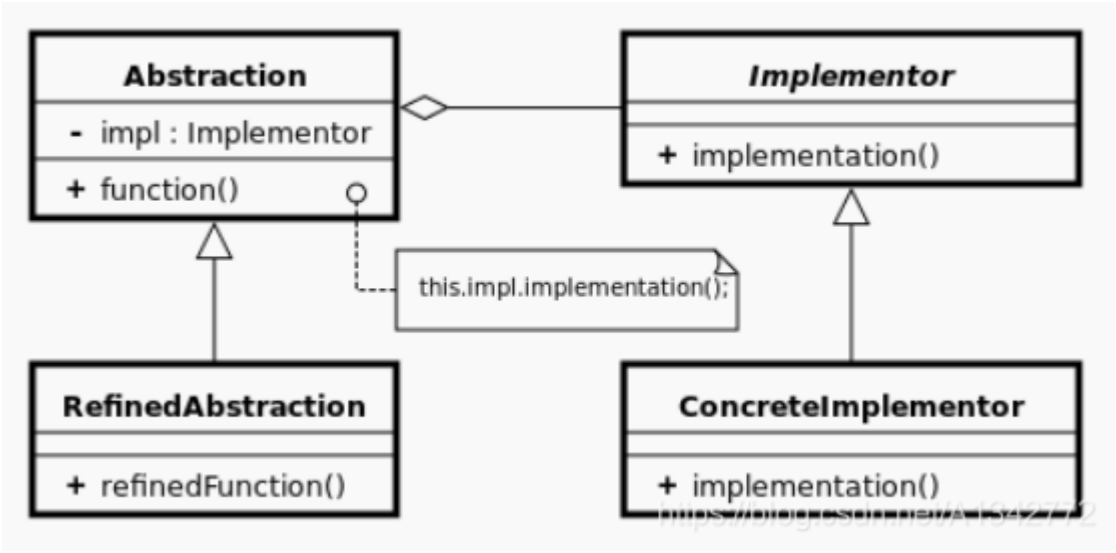


从类图上看起来更像是手机软件类图，涉及到手机本身相关的功能，比如说：wifi功能，放到哪个类中实现呢？放到OppoAppStore中实现显然是不合适的

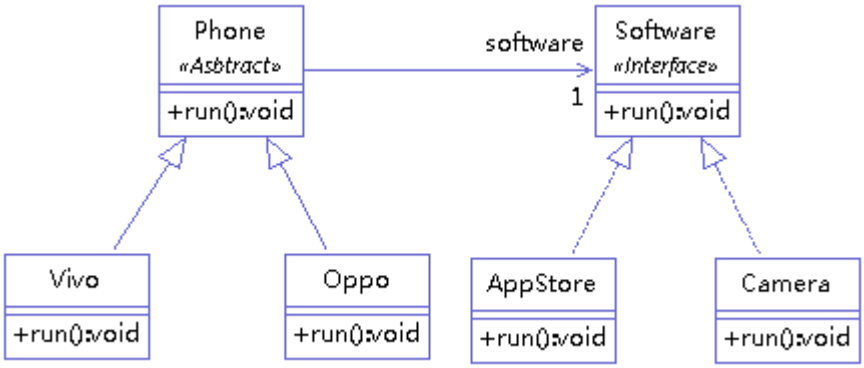
引起整个结构变化的元素有两个，一个是手机品牌，一个是手机软件，所以我们将这两个点抽出来，分别进行封装



9.2 桥接模式结构和代码示例



类图：



实现：

```
1. public interface Software {
2.         public void run();
3.
4. }
5. public class AppStore implements Software {
6.
7.     @Override
8.     public void run() {
9.         System.out.println("run app store");
10.     }
11. }
12. public class Camera implements Software {
```

```
13.
14.     @Override
15.     public void run() {
16.         System.out.println("run camera");
17.     }
18. }
```

抽象:

```
1. public abstract class Phone {
2.
3.         protected Software software;
4.
5.         public void setSoftware(Software software) {
6.                 this.software = software;
7.         }
8.
9.         public abstract void run();
10.
11. }
12. public class Oppo extends Phone {
13.
14.     @Override
15.     public void run() {
16.         software.run();
17.     }
18. }
19. public class Vivo extends Phone {
20.
```

```
21.     @Override
22.     public void run() {
23.         software.run();
24.     }
25. }
```

对比最初的设计，将抽象部分（手机）与它的实现部分（手机软件类）分离，将实现部分抽象成单独的类，使它们都可以独立地变化。整个类图看起来像一座桥，所以称为桥接模式

继承是一种强耦合关系，子类的实现与它的父类有非常紧密的依赖关系，父类的任何变化 都会导致子类发生变化，因此继承或者说强耦合关系严重影响了类的灵活性，并最终限制了可复用性

从桥接模式的设计上我们可以看出聚合是一种比继承要弱的关联关系，手机类和软件类都可独立的进行变化，不会互相影响

9.3 适用场景

桥接模式通常适用于以下场景。

1. 当一个类存在两个独立变化的维度，且这两个维度都需要进行扩展时。
2. 当一个系统不希望使用继承或因为多层次继承导致系统类的个数急剧增加时。
3. 当一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性时。

9.4 优缺点

优点：

(1)在很多情况下，桥接模式可以取代多层继承方案，多层继承方案违背了“单一职责原则”，复用性较差，且类的个数非常多，桥接模式是比多层继承方案更好的解决方法，它极大减少了子类的个数。

(2)桥接模式提高了系统的可扩展性，在两个变化维度中任意扩展一个维度，都不需要修改原有系统，符合“开闭原则”。

缺点：

桥接模式的使用会增加系统的理解与设计难度，由于关联关系建立在抽象层，要求开发者一开始就针对抽象层进行设计与编程。

10 组合模式

定义：有时又叫作部分-整体模式，它是一种将对象组合成树状的层次结构的模式，用来表示“部分-整体”的关系，使用户对单个对象和组合对象具有一致的访问性。

意图：将对象组合成树形结构以表示"部分-整体"的层次结构。组合模式使得用户对单个对象和组合对象的使用具有一致性。

主要解决：它在我们树型结构的问题中，模糊了简单元素和复杂元素的概念，客户程序可以向处理简单元素一样来处理复杂元素，从而使得客户程序与复杂元素的内部结构解耦。

何时使用： 1、您想表示对象的部分-整体层次结构（树形结构）。 2、您希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象。

如何解决：树枝和叶子实现统一接口，树枝内部组合该接口。

关键代码：树枝内部组合该接口，并且含有内部属性 List，里面放 Component。

组合模式的主要优点有：

1. 组合模式使得客户端代码可以一致地处理单个对象和组合对象，无须关心自己处理的是单个对象，还是组合对象，这简化了客户端代码；

2. 更容易在组合体内加入新的对象，客户端不会因为加入了新的对象而更改源代码，满足“开闭原则”；

其主要缺点是：

1. 设计较复杂，客户端需要花更多时间理清类之间的层次关系；

2. 不容易限制容器中的构件；

3. 不容易用继承的方法来增加构件的新功能；

10.1 模式结构和代码示例

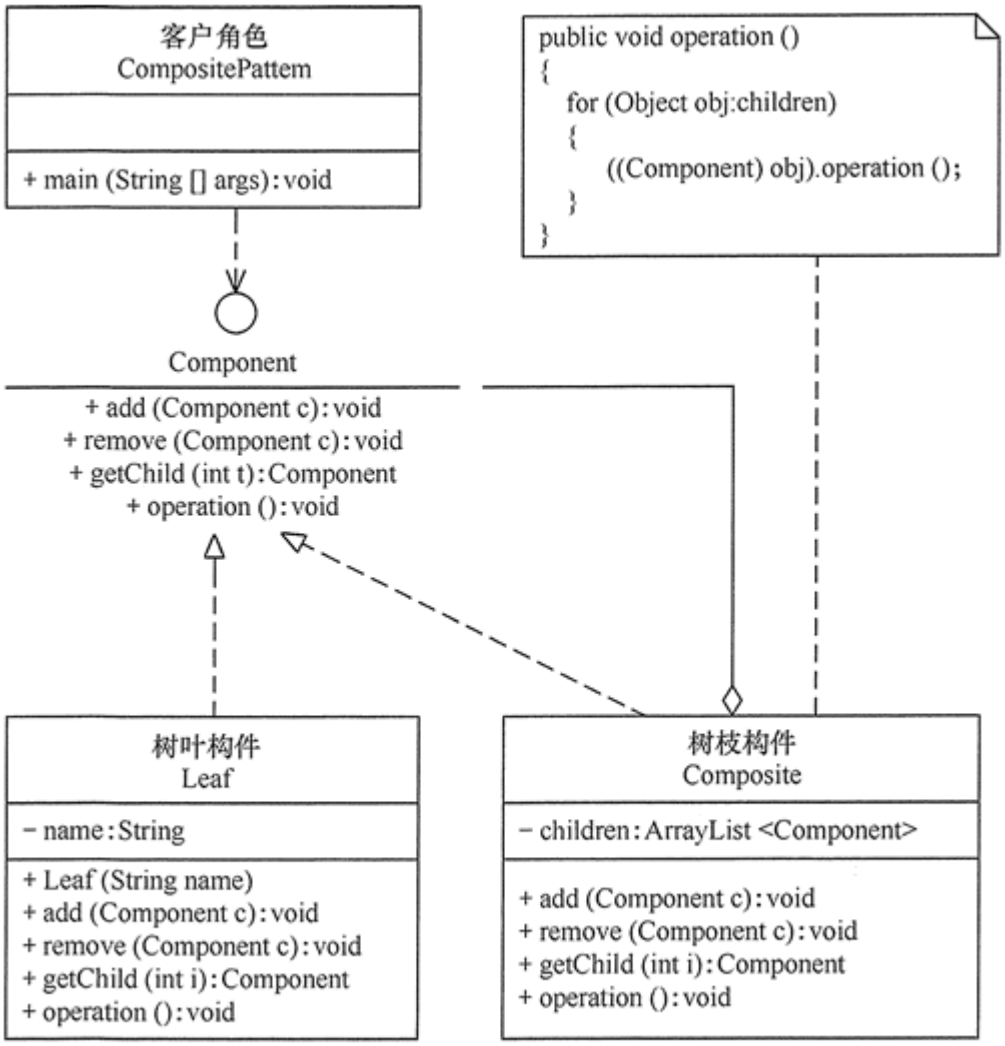
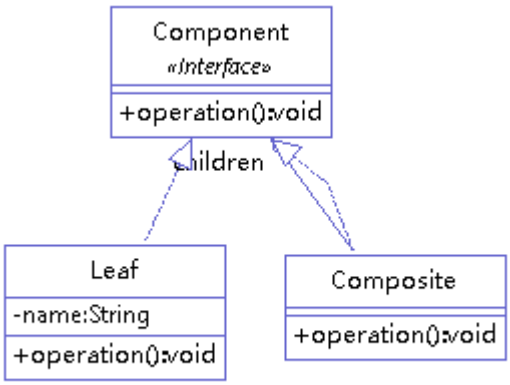


图1 透明式的组合模式的结构图

- 抽象构件（Component）角色：它的主要作用是为树叶构件和树枝构件声明公共接口，并实现它们的默认行为。在透明式的组合模式中抽象构件还声明访问和管理子类的接口；在安全式的组合模式中不声明访问和管理子类的接口，管理工作由树枝构件完成。
- 树叶构件（Leaf）角色：是组合中的叶节点对象，它没有子节点，用于实现抽象构件角色中 声明的公共接口。
- 树枝构件（Composite）角色：是组合中的分支节点对象，它有子节点。它实现了抽象构件角色中声明的接口，它的主要作用是存储和管理子部件，通常包含 Add()、Remove()、GetChild() 等方法

举例（访问一颗树），类图如下：



1 组件

```
1. public interface Component {
2.     public void add(Component c);
3.     public void remove(Component c);
4.     public Component getChild(int i);
5.     public void operation();
6.
7. }
```

2 叶子

```
1. public class Leaf implements Component{
2.
3.         private String name;
4.
5.
6.         public Leaf(String name) {
7.                 this.name = name;
8.         }
9.
10.         @Override
11.         public void add(Component c) {}
```

```
12.
13.         @Override
14.         public void remove(Component c) {}
15.
16.         @Override
17.         public Component getChild(int i) {
18.                 // TODO Auto-generated method stub
19.                 return null;
20.         }
21.
22.         @Override
23.         public void operation() {
24.                 // TODO Auto-generated method stub
25.                  System.out.println("树叶"+name+"：被访问！");
26.         }
27.
28. }
```

3 树枝

```
1. public class Composite implements Component {
2.
3.         private ArrayList<Component> children = new ArrayList<Component>();
4.
5.         public void add(Component c) {
6.                 children.add(c);
7.         }
8.
9.         public void remove(Component c) {
```

```
10.                 children.remove(c);
11.         }
12.
13.         public Component getChild(int i) {
14.                 return children.get(i);
15.         }
16.
17.         public void operation() {
18.                 for (Object obj : children) {
19.                         ((Component) obj).operation();
20.                 }
21.         }
22. }
```


11 享元模式

定义：通过共享的方式高效的支持大量细粒度的对象。

主要解决：在有大量对象时，有可能会造成内存溢出，我们把其中共同的部分抽象出来，如果有相同的业务请求，直接返回在内存中已有的对象，避免重新创建。

何时使用： 1、系统中有大量对象。 2、这些对象消耗大量内存。 3、这些对象的状态大部分可以外部化。 4、这些对象可以按照内蕴状态分为很多组，当把外蕴对象从对象中剔除出来时，每一组对象都可以用一个对象来代替。 5、系统不依赖于这些对象身份，这些对象是不可分辨的。

如何解决：用唯一标识码判断，如果在内存中有，则返回这个唯一标识码所标识的对象。

关键代码：用 HashMap 存储这些对象。

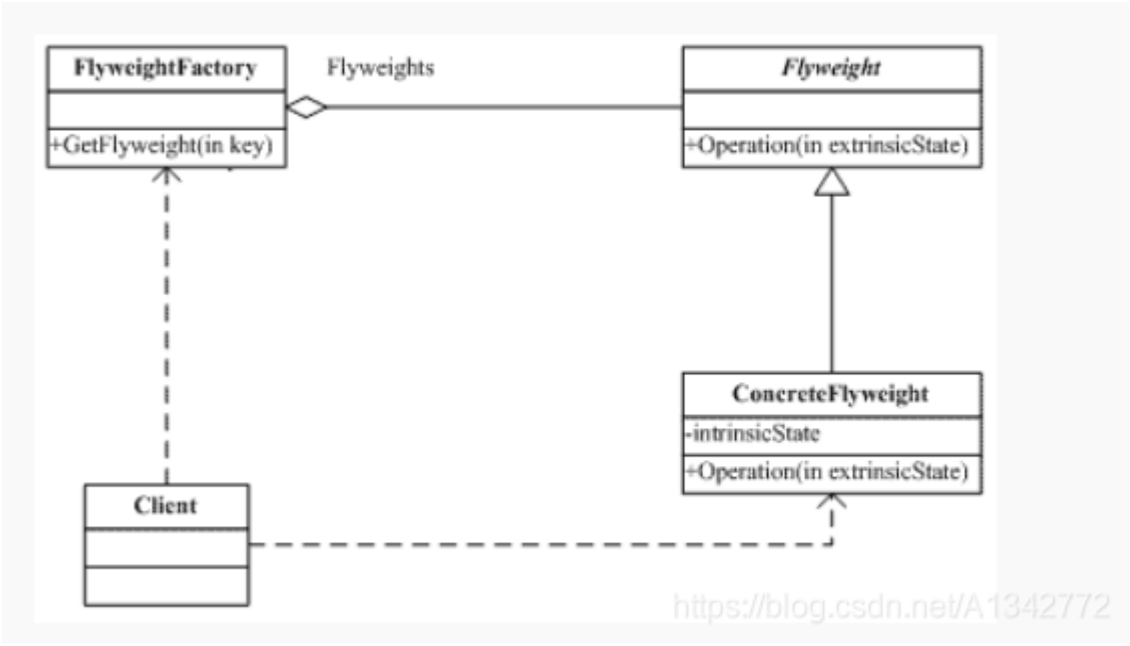
应用实例： 1、JAVA 中的 String，如果有则返回，如果没有则创建一个字符串保存在字符串缓存池里面。

优点：大大减少对象的创建，降低系统的内存，使效率提高。

缺点：提高了系统的复杂度，需要分离出外部状态和内部状态，而且外部状态具有固有化的性质，不应该随着内部状态的变化而变化，否则会造成系统的混乱。

简单来说，我们抽取出一个对象的外部状态（不能共享）和内部状态（可以共享）。然后根据外部状态的决定是否创建内部状态对象。内部状态对象是通过哈希表保存的，当外部状态相同的时候，不再重复的创建内部状态对象，从而减少要创建对象的数量。

11.1 享元模式的结构图和代码示例

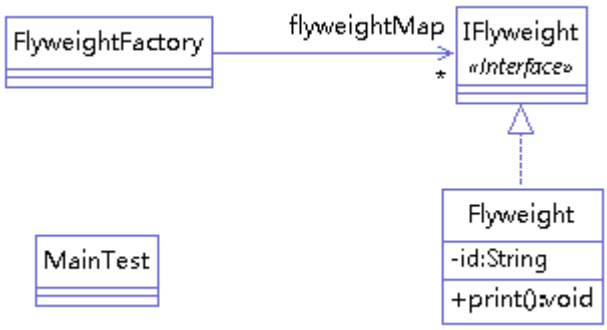


1、Flyweight (享元抽象类)：一般是接口或者抽象类，定义了享元类的公共方法。这些方法可以分享内部状态的数据，也可以调用这些方法修改外部状态。

2、ConcreteFlyweight(具体享元类)：具体享元类实现了抽象享元类的方法，为享元对象开辟了内存空间来保存享元对象的内部数据，同时可以通过和单例模式结合只创建一个享元对象。

3、FlyweightFactory(享元工厂类)：享元工厂类创建并且管理享元类，享元工厂类针对享元类来进行编程，通过提供一个享元池来进行享元对象的管理。一般享元池设计成**键值对**，或者其他的存储结构来存储。当客户端进行享元对象的请求时，如果享元池中有对应的享元对象则直接返回对应的对象，否则工厂类创建对应的享元对象并保存到享元池。

举例（JAVA 中的 String，如果有则返回，如果没有则创建一个字符串保存在字符串缓存池里面）。类图如下：



（1）创建享元对象接口

```
1. public interface IFlyweight {
2.     void print();
3. }
```

(2) 创建具体享元对象

```
1. public class Flyweight implements IFlyweight {
2.     private String id;
3.     public Flyweight(String id){
4.         this.id = id;
5.     }
6.     @Override
7.     public void print() {
8.         System.out.println("Flyweight.id = " + getId() + " ...");
9.     }
10.     public String getId() {
11.         return id;
12.     }
13. }
```

(3) 创建工厂，这里要特别注意，为了避免享元对象被重复创建，我们使用HashMap中的key值保证其唯一。

```
1. public class FlyweightFactory {
2.     private Map<String, IFlyweight> flyweightMap = new HashMap();
3.     public IFlyweight getFlyweight(String str){
4.         IFlyweight flyweight = flyweightMap.get(str);
5.         if(flyweight == null){
6.             flyweight = new Flyweight(str);
7.             flyweightMap.put(str, flyweight);
8.         }
9.         return  flyweight;
10.     }
11.     public int getFlyweightMapSize(){
12.         return flyweightMap.size();
```

```
    }
}
```

（4）测试，我们创建三个字符串，但是只会产生两个享元对象

```
public class MainTest {
        public static void main(String[] args) {
        FlyweightFactory flyweightFactory = new FlyweightFactory();
        IFlyweight flyweight1 = flyweightFactory.getFlyweight("A");
        IFlyweight flyweight2 = flyweightFactory.getFlyweight("B");
        IFlyweight flyweight3 = flyweightFactory.getFlyweight("A");
        flyweight1.print();
        flyweight2.print();
        flyweight3.print();
        System.out.println(flyweightFactory.getFlyweightMapSize());
    }

}
```

```
<terminated> MainTest (14) [Java Application] D:\jdk1.8\bin\javaw.exe (2019年6
Flyweight.id = A ...
Flyweight.id = B ...
Flyweight.id = A ...
2
```

C、关系模式（11种）

先来张图，看看这11中模式的关系：

第一类：通过父类与子类的关系进行实现。

第二类：两个类之间。

第三类：类的状态。

第四类：通过中间类

12 策略模式

定义： 策略模式定义了一系列算法，并将每个算法封装起来，使他们可以相互替换，且算法的变化不会影响到使用算法的客户。

意图：定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。

主要解决：在有多种算法相似的情况下，使用 if...else 所带来的复杂和难以维护。

何时使用：一个系统有许多许多类，而区分它们的只是他们直接的行为。

如何解决：将这些算法封装成一个一个的类，任意地替换。

关键代码：实现同一个接口。

优点： 1、算法可以自由切换。 2、避免使用多重条件判断。 3、扩展性良好。

缺点： 1、策略类会增多。 2、所有策略类都需要对外暴露。

12.1 策略模式结构和示例代码

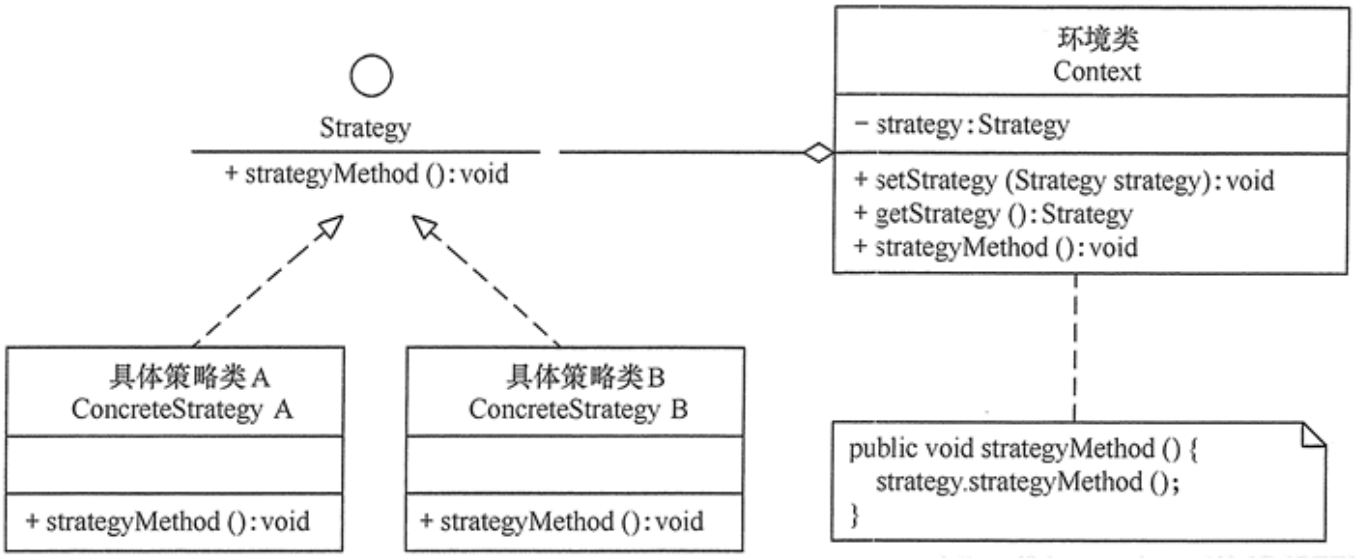


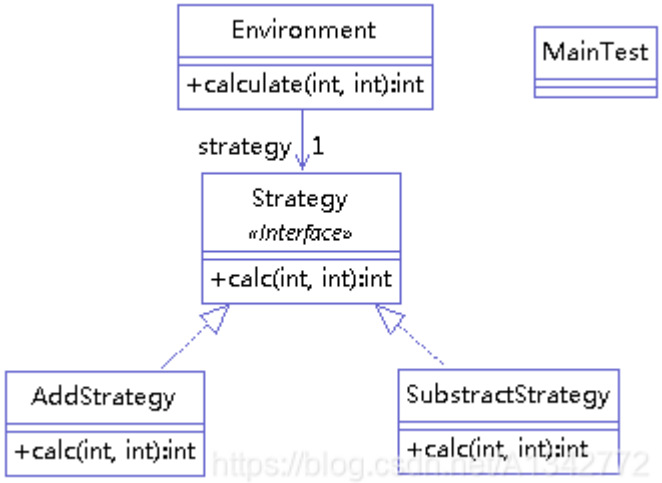
图1 策略模式的结构图

抽象策略角色: 这个是一个抽象的角色，通常情况下使用接口或者抽象类去实现。对比来说，就是我们的Comparator接口。

具体策略角色: 包装了具体的算法和行为。对比来说，就是实现了Comparator接口的实现一组实现类。

环境角色: 内部会持有一个抽象角色的引用，给客户端调用。

举例如下（实现一个加减的功能），类图如下：



1、定义抽象策略角色

```
1. public interface Strategy {
2.
3.         public int calc(int num1,int num2);
4. }
```

2、定义具体策略角色

```
1. public class AddStrategy implements Strategy {
2. 
3.         @Override
4.         public int calc(int num1, int num2) {
5.                 // TODO Auto-generated method stub
6.                 return num1 + num2;
7.         }
8. 
9. }
10. public class SubstractStrategy implements Strategy {
11. 
12.         @Override
13.         public int calc(int num1, int num2) {
14.                 // TODO Auto-generated method stub
15.                 return num1 - num2;
16.         }
17. 
18. }
```


3、环境角色

```
1. public class Environment {
2.         private Strategy strategy;
3. 
4.         public Environment(Strategy strategy) {
5.                 this.strategy = strategy;
6.         }
7. 
8.         public int calculate(int a, int b) {
```

```
9.                  return strategy.calc(a, b);
10.         }
11.
12. }
```

4、测试

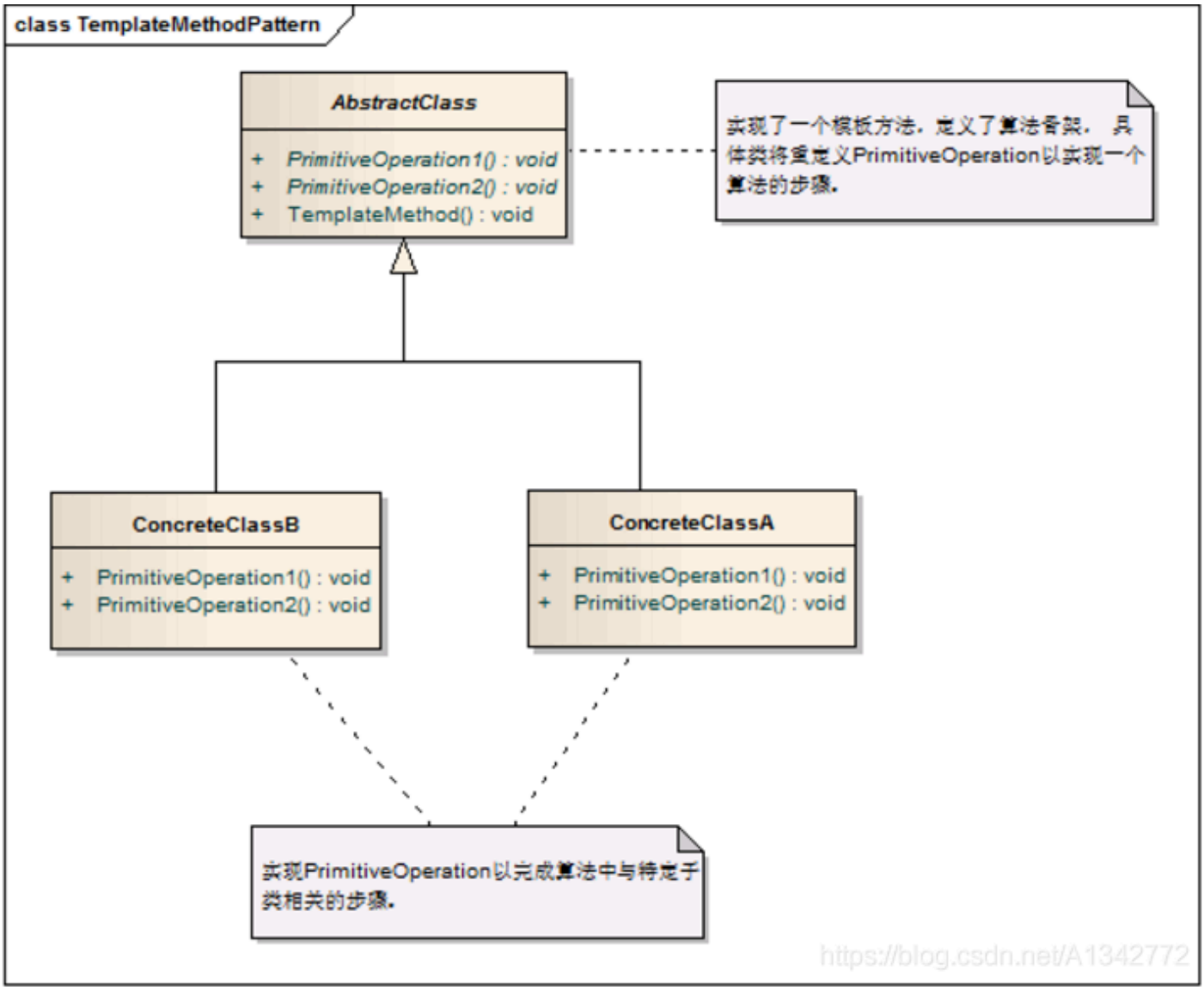
```
1. public class MainTest {
2.         public static void main(String[] args) {
3.
4.                 Environment environment=new Environment(new AddStrategy());
5.                 int result=environment.calculate(20, 5);
6.                 System.out.println(result);
7.
8.                 Environment environment1=new Environment(new SubstractStrategy());
9.                 int result1=environment1.calculate(20, 5);
10.                System.out.println(result1);
11.        }
12.
13. }
```

13 模板模式

定义：定义一个操作中算法的骨架，而将一些步骤延迟到子类中，模板方法使得子类可以不改变算法的结构即可重定义该算法的某些特定步骤。

通俗点的理解就是：完成一件事情，有固定的数个步骤，但是每个步骤根据对象的不同，而实现细节不同；就可以在父类中定义一个完成该事情的总方法，按照完成事件需要的步骤去调用其每个步骤的实现方法。每个步骤的具体实现，由子类完成。

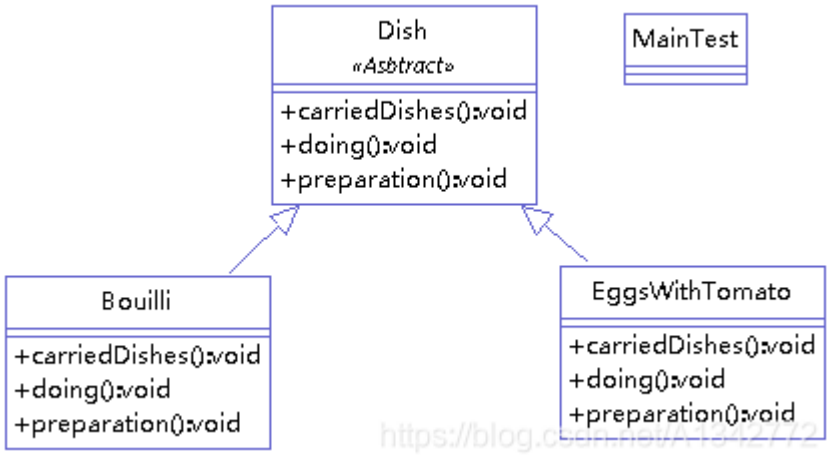
13.1 模式结构和代码示例



抽象父类（AbstractClass）：实现了模板方法，定义了算法的骨架。

具体类（ConcreteClass)：实现抽象类中的抽象方法，即不同的对象的具体实现细节。

举例（我们做菜可以分为三个步骤 （1）备料 （2）具体做菜 （3）盛菜端给客人享用，这三部就是算法的骨架；然而做不同菜需要的料，做的方法，以及如何盛装给客人享用都是不同的这个就是不同的实现细节。）。类图如下：



a. 先来写一个抽象的做菜父类：

```
public abstract class Dish {
    /**
     * 具体的整个过程
     */
    protected void dodish(){
        this.preparation();
        this.doing();
        this.carriedDishes();
    }
    /**
     * 备料
     */
    public abstract void preparation();
    /**
     * 做菜
     */
    public abstract void doing();
    /**
     * 上菜
     */
    public abstract void carriedDishes ();
}
```

b. 下来做两个番茄炒蛋（EggsWithTomato）和红烧肉（Bouilli）实现父类中的抽象方法

```
public class EggsWithTomato extends Dish {

        @Override
```

```
4.          public void preparation() {
5.                  System.out.println("洗并切西红柿，打鸡蛋。");
6.          }
7.
8.          @Override
9.          public void doing() {
10.                 System.out.println("鸡蛋倒入锅里，然后倒入西红柿一起炒。");
11.         }
12.
13.         @Override
14.         public void carriedDishes() {
15.                 System.out.println("将炒好的西红寺鸡蛋装入碟子里，端给客人吃。");
16.         }
17.
18. }
19. public class Bouilli extends Dish{
20.
21.     @Override
22.     public void preparation() {
23.         System.out.println("切猪肉和土豆。");
24.     }
25.
26.     @Override
27.     public void doing() {
28.         System.out.println("将切好的猪肉倒入锅中炒一会然后倒入土豆连炒带炖。");
29.     }
30.
31.     @Override
32.     public void carriedDishes() {
```

```
33.         System.out.println("将做好的红烧肉盛进碗里端给客人吃。");
34.     }
35.
36. }
```

c. 在测试类中我们来做菜：

```
1. public class MainTest {
2.         public static void main(String[] args) {
3.                 Dish eggsWithTomato = new EggsWithTomato();
4.                 eggsWithTomato.dodish();
5.
6.                 System.out.println("--------------------------------");
7.
8.                 Dish bouilli = new Bouilli();
9.                 bouilli.dodish();
10.         }
11.
12. }
```

13.2 模板模式的优点和缺点

优点：

（1）具体细节步骤实现定义在子类中，子类定义详细处理算法是不会改变算法整体结构。

（2）代码复用的基本技术，在数据库设计中尤为重要。

（3）存在一种反向的控制结构，通过一个父类调用其子类的操作，通过子类对父类进行扩展增加新的行为，符合“开闭原则”。

缺点：

每个不同的实现都需要定义一个子类，会导致类的个数增加，系统更加庞大。

14 观察者模式

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

主要解决：一个对象状态改变给其他对象通知的问题，而且要考虑到易用和低耦合，保证高度的协作。

何时使用：一个对象（目标对象）的状态发生改变，所有的依赖对象（观察者对象）都将得到通知，进行广播通知。

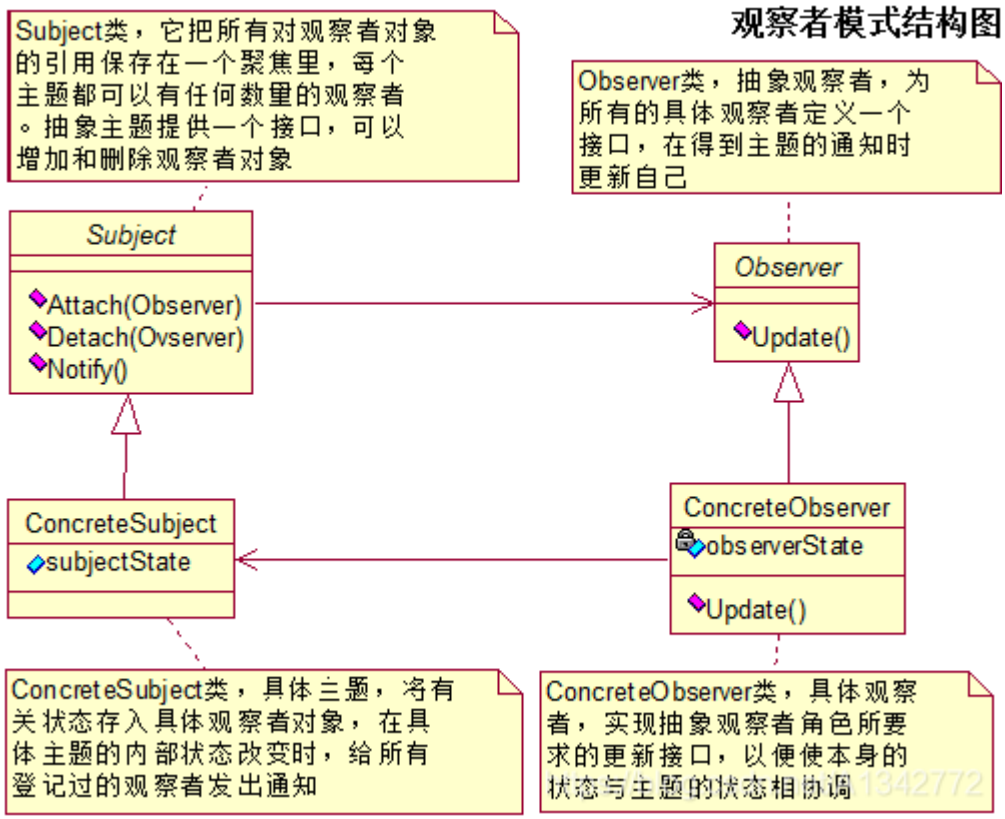
如何解决：使用面向对象技术，可以将这种依赖关系弱化。

关键代码：在抽象类里有一个 ArrayList 存放观察者们。

优点： 1、观察者和被观察者是抽象耦合的。 2、建立一套触发机制。

缺点： 1、如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。2、如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃。3、观察者模式没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而仅仅只是知道观察目标发生了变化。

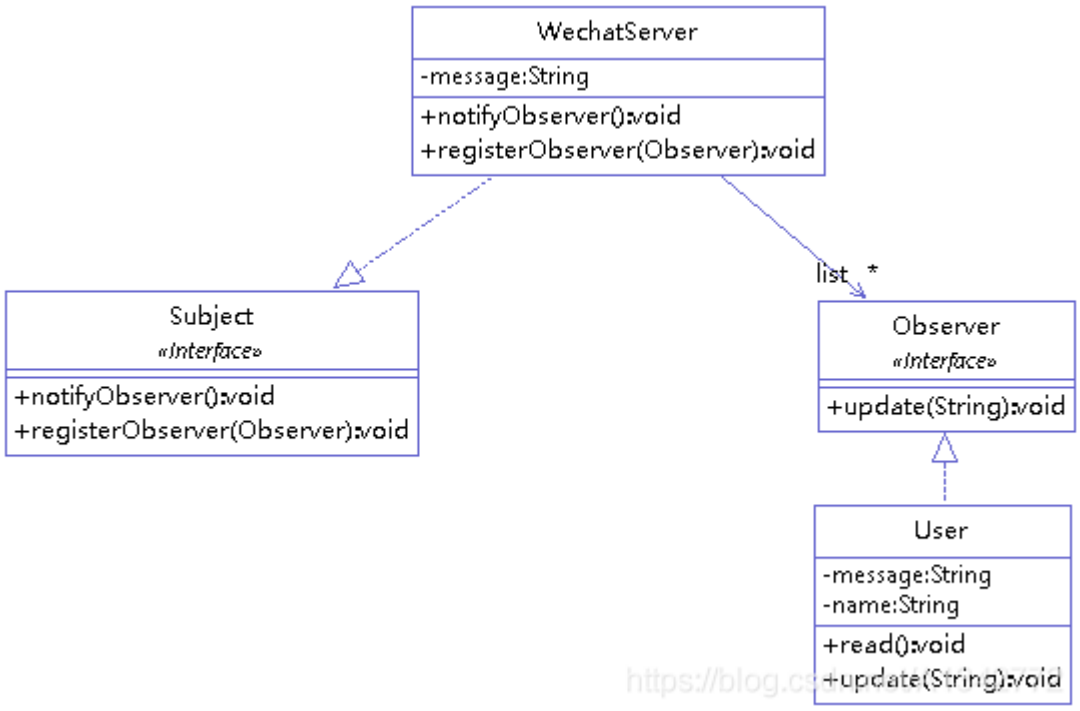
14.1 模式结构图和代码示例



- **抽象被观察者角色**：也就是一个抽象主题，它把所有对观察者对象的引用保存在一个集合中，每个主题都可以有任意数量的观察者。抽象主题提供一个接口，可以增加和删除观察者角色。一般用一个抽象类和接口来实现。
- **抽象观察者角色**：为所有的具体观察者定义一个接口，在得到主题通知时更新自己。

- **具体被观察者角色**：也就是一个具体的主题，在集体主题的内部状态改变时，所有登记过的观察者发出通知。
- **具体观察者角色**：实现抽象观察者角色所需要的更新接口，一边使本身的状态与制图的状态相协调。

举例（有一个微信公众号服务，不定时发布一些消息，关注公众号就可以收到推送消息，取消关注就收不到推送消息。）类图如下：



1、定义一个抽象被观察者接口

```
1. public interface Subject {
2.
3.          public void registerObserver(Observer o);
4.          public void removeObserver(Observer o);
5.          public void notifyObserver();
6.
7. }
```

2、定义一个抽象观察者接口

```
1. public interface Observer {
2.
3.         public void update(String message);
```

```
4.
5. }
```

3、定义被观察者，实现了Observerable接口，对Observerable接口的三个方法进行了具体实现，同时有一个List集合，用以保存注册的观察者，等需要通知观察者时，遍历该集合即可。

```
1. public class WechatServer implements Subject {
2.
3.         private List<Observer> list;
4.         private String message;
5.
6.         public WechatServer() {
7.                 list = new ArrayList<Observer>();
8.         }
9.
10.         @Override
11.         public void registerObserver(Observer o) {
12.                 // TODO Auto-generated method stub
13.                 list.add(o);
14.         }
15.
16.         @Override
17.         public void removeObserver(Observer o) {
18.                 // TODO Auto-generated method stub
19.                 if (!list.isEmpty()) {
20.                         list.remove(o);
21.                 }
22.         }
23.
24.         @Override
```

```
25.         public void notifyObserver() {
26.                 // TODO Auto-generated method stub
27.                 for (Observer o : list) {
28.                         o.update(message);
29.                 }
30.         }
31.
32.         public void setInfomation(String s) {
33.                 this.message = s;
34.                 System.out.println("微信服务更新消息： " + s);
35.                 // 消息更新，通知所有观察者
36.                 notifyObserver();
37.         }
38.
39. }
```

4、定义具体观察者，微信公众号的具体观察者为用户User

```
1. public class User implements Observer {
2.
3.         private String name;
4.         private String message;
5.
6.         public User(String name) {
7.                 this.name = name;
8.         }
9.
10.         @Override
11.         public void update(String message) {
```

```
                this.message = message;
                read();
        }

        public void read() {
                System.out.println(name + " 收到推送消息： " + message);
        }

}
```

5、编写一个测试类

```
public class MainTest {

         public static void main(String[] args) {

                WechatServer server = new WechatServer();

                Observer userZhang = new User("ZhangSan");
                Observer userLi = new User("LiSi");
                Observer userWang = new User("WangWu");

                server.registerObserver(userZhang);
                server.registerObserver(userLi);
                server.registerObserver(userWang);
                server.setInfomation("PHP是世界上最好用的语言！");

                System.out.println("----------------------------------------------");
                server.removeObserver(userZhang);
```

```
18.                 server.setInfomation("JAVA是世界上最好用的语言！");
19.
20.             }
21.
22. }
```


15 迭代器模式

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

简单来说，不同种类的对象可能需要不同的遍历方式，我们对每一种类型的对象配一个迭代器，最后多个迭代器合成一个。

主要解决：不同的方式来遍历整个整合对象。

何时使用：遍历一个聚合对象。

如何解决：把在元素之间游走的责任交给迭代器，而不是聚合对象。

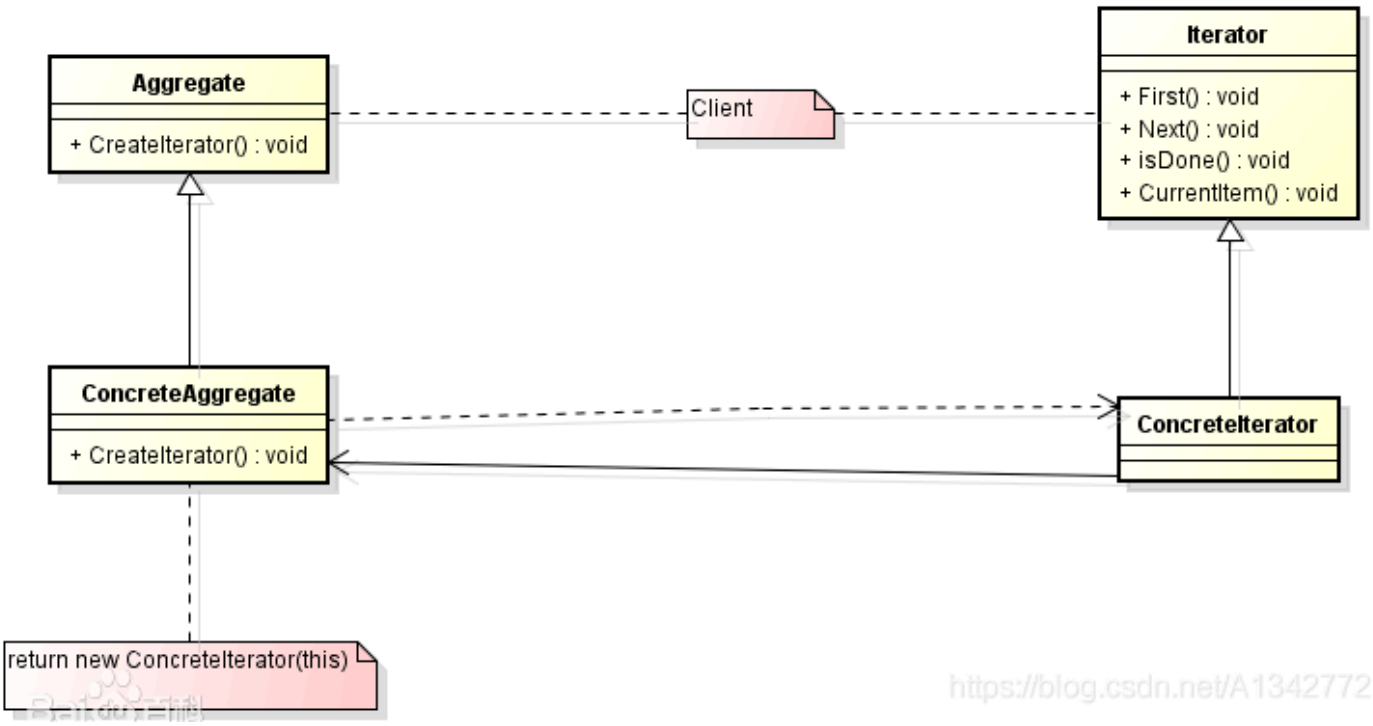
关键代码：定义接口：hasNext, next。

应用实例：JAVA 中的 iterator。

优点： 1、它支持以不同的方式遍历一个聚合对象。 2、迭代器简化了聚合类。 3、在同一个聚合上可以有多个遍历。 4、在迭代器模式中，增加新的聚合类和迭代器类都很方便，无须修改原有代码。

缺点：由于迭代器模式将存储数据和遍历数据的职责分离，增加新的聚合类需要对应增加新的迭代器类，类的个数成对增加，这在一定程度上增加了系统的复杂性。

15.1 模式结构和代码示例



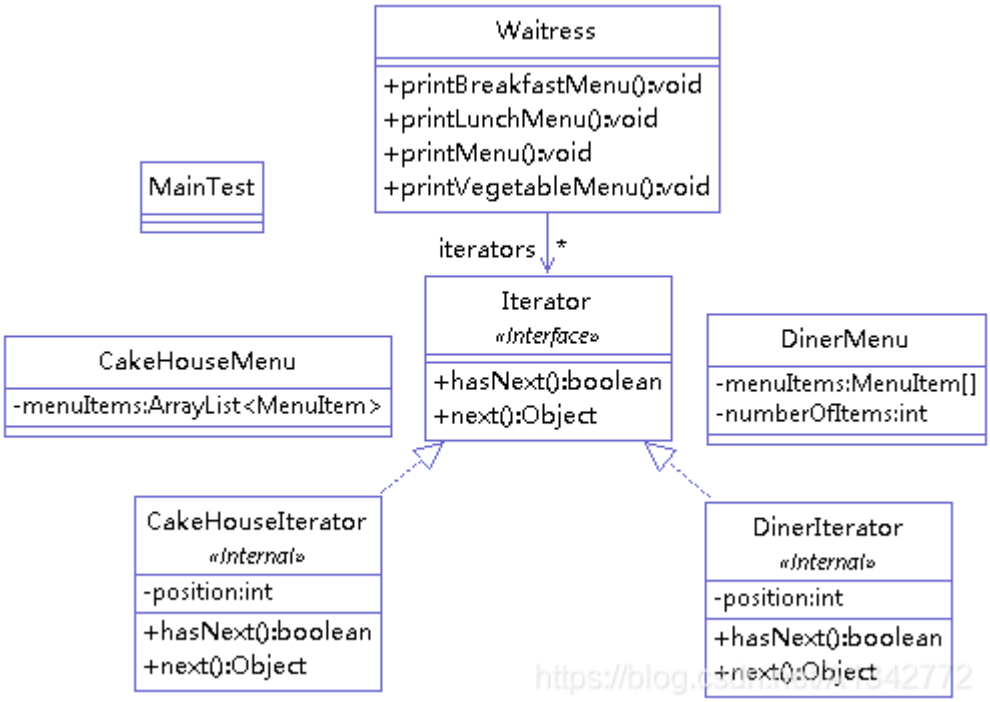
(1)迭代器角色（Iterator）:定义遍历元素所需要的方法，一般来说会有这么三个方法：取得下一个元素的方法next()，判断是否遍历结束的方法hasNext()），移出当前对象的方法remove(),

(2)具体迭代器角色（Concrete Iterator）：实现迭代器接口中定义的方法，完成集合的迭代。

(3)容器角色(Aggregate)：一般是一个接口，提供一个iterator()方法，例如java中的Collection接口，List接口，Set接口等

(4)具体容器角色（ConcreteAggregate）：就是抽象容器的具体实现类，比如List接口的有序列表实现ArrayList，List接口的链表实现LinkList，Set接口的哈希列表的实现HashSet等。

举例（咖啡厅和中餐厅合并，他们两个餐厅的菜单一个是数组保存的，一个是ArrayList保存的。遍历方式不一样，使用迭代器聚合访问，只需要一种方式）



1 迭代器接口

```
1. public interface Iterator {
2.
3.         public boolean hasNext();
4.         public Object next();
5.
6. }
```

2 咖啡店菜单和咖啡店菜单遍历器

```
1. public class CakeHouseMenu {
2.         private ArrayList<MenuItem> menuItems;
3.
4.
5.         public CakeHouseMenu() {
6.                 menuItems = new ArrayList<MenuItem>();
7.
8.                 addItem("KFC Cake Breakfast","boiled eggs&toast&cabbage",true,3.99f);
```

```
                addItem("MDL Cake Breakfast","fried eggs&toast",false,3.59f);
                addItem("Stawberry Cake","fresh stawberry",true,3.29f);
                addItem("Regular Cake Breakfast","toast&sausage",true,2.59f);
        }

        private void addItem(String name, String description, boolean vegetable,
                        float price) {
                MenuItem menuItem = new MenuItem(name, description, vegetable, price);
                menuItems.add(menuItem);
        }



        public Iterator getIterator()
        {
                return new CakeHouseIterator() ;
        }

        class CakeHouseIterator implements  Iterator
         {
                private int position=0;
                public CakeHouseIterator()
                {
                          position=0;
                }

                        @Override
                        public boolean hasNext() {
                        // TODO Auto-generated method stub
```

```
38.                         if(position<menuItems.size())
39.                         {
40.                                 return true;
41.                         }
42.
43.                         return false;
44.                 }
45.
46.                 @Override
47.                 public Object next() {
48.                         // TODO Auto-generated method stub
49.                         MenuItem menuItem =menuItems.get(position);
50.                         position++;
51.                         return menuItem;
52.                 }};
53.         //鍏朵粬鍔熻兘 g 爨
54.
55. }
```

3 中餐厅菜单和中餐厅菜单遍历器

```
1. public class DinerMenu {
2.         private final static int Max_Items = 5;
3.         private int numberOfItems = 0;
4.         private MenuItem[] menuItems;
5.
6.         public DinerMenu() {
7.                 menuItems = new MenuItem[Max_Items];
8.                 addItem("vegetable Blt", "bacon&lettuce&tomato&cabbage", true, 3.58f);
```

```
9.                 addItem("Blt", "bacon&lettuce&tomato", false, 3.00f);
10.                addItem("bean soup", "bean&potato salad", true, 3.28f);
11.                addItem("hotdog", "onions&cheese&bread", false, 3.05f);
12.
13.        }
14.
15.        private void addItem(String name, String description, boolean vegetable,
16.                        float price) {
17.                MenuItem menuItem = new MenuItem(name, description, vegetable, price);
18.                if (numberOfItems >= Max_Items) {
19.                        System.err.println("sorry,menu is full!can not add another item");
20.                } else {
21.                        menuItems[numberOfItems] = menuItem;
22.                        numberOfItems++;
23.                }
24.
25.        }
26.
27.        public Iterator getIterator() {
28.                return new DinerIterator();
29.        }
30.
31.        class DinerIterator implements Iterator {
32.                private int position;
33.
34.                public DinerIterator() {
35.                        position = 0;
36.                }
37.
```

```
38.                 @Override
39.                 public boolean hasNext() {
40.                         // TODO Auto-generated method stub
41.                         if (position < numberOfItems) {
42.                                 return true;
43.                         }
44.
45.                         return false;
46.                 }
47.
48.                 @Override
49.                 public Object next() {
50.                         // TODO Auto-generated method stub
51.                         MenuItem menuItem = menuItems[position];
52.                         position++;
53.                         return menuItem;
54.                 }
55.         };
56. }
```

4 女服务员

```
1. public class Waitress {
2.         private ArrayList<Iterator> iterators = new ArrayList<Iterator>();
3.
4.         public Waitress() {
5.
6.         }
7.
```

```
8.          public void addIterator(Iterator iterator) {
9.                  iterators.add(iterator);
10.
11.         }
12.
13.         public void printMenu() {
14.                 Iterator iterator;
15.                 MenuItem menuItem;
16.                 for (int i = 0, len = iterators.size(); i < len; i++) {
17.                         iterator = iterators.get(i);
18.
19.                         while (iterator.hasNext()) {
20.                                 menuItem = (MenuItem) iterator.next();
21.                                 System.out
22.                                                 .println(menuItem.getName() + "***" +
    menuItem.getPrice() + "***" + menuItem.getDescription());
23.
24.                         }
25.
26.                 }
27.
28.         }
29.
30.         public void printBreakfastMenu() {
31.
32.         }
33.
34.         public void printLunchMenu() {
35.
```

```
36.         }
37.
38.         public void printVegetableMenu() {
39.
40.         }
41. }
```

16 责任链模式

定义：如果有多个对象有机会处理请求，责任链可使请求的发送者和接受者解耦，请求沿着责任链传递，直到有一个对象处理了它为止。

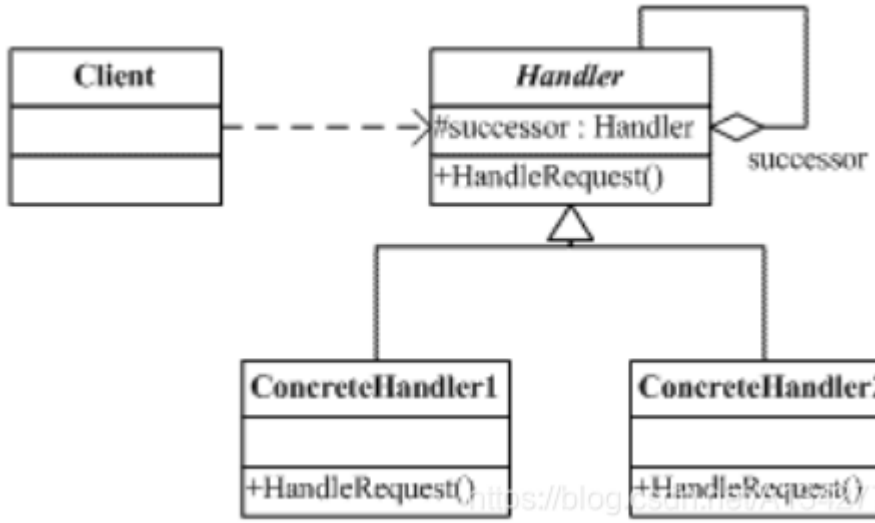
主要解决：职责链上的处理者负责处理请求，客户只需要将请求发送到职责链上即可，无须关心请求的处理细节和请求的传递，所以职责链将请求的发送者和请求的处理者解耦了。

何时使用：在处理消息的时候以过滤很多道。

如何解决：拦截的类都实现统一接口。

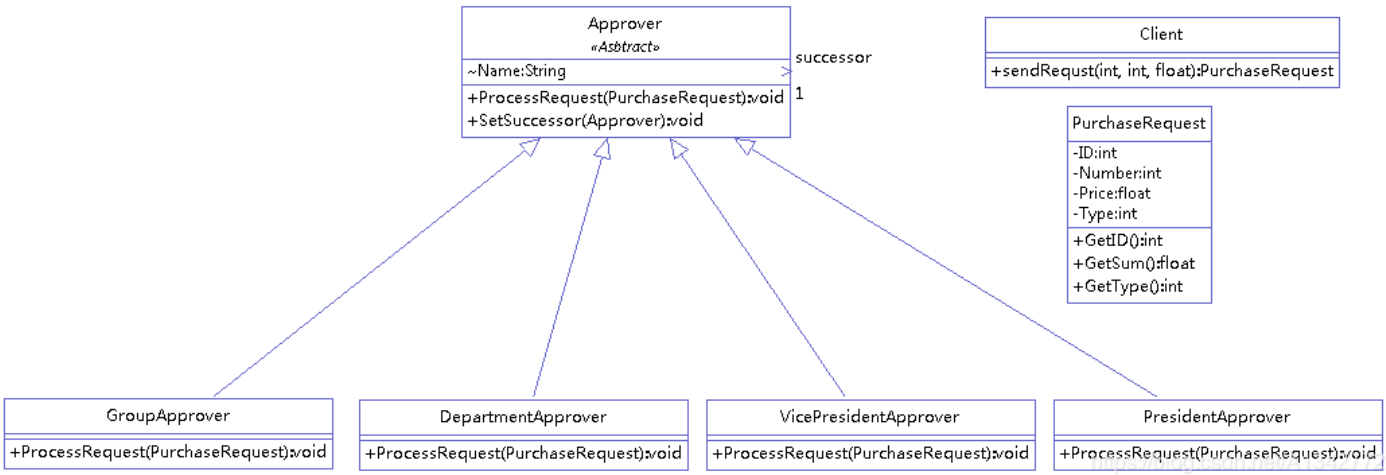
关键代码：Handler 里面聚合它自己，在 HandlerRequest 里判断是否合适，如果没达到条件则向下传递，向谁传递之前 set 进去。

16.1 模式的结构和代码示例



1. 抽象处理者（Handler）角色：定义一个处理请求的接口，包含抽象处理方法和一个后继连接。
2. 具体处理者（Concrete Handler）角色：实现抽象处理者的处理方法，判断能否处理本次请求，如果可以处理请求则处理，否则将该请求转给它的后继者。
3. 客户类（Client）角色：创建处理链，并向链头的具体处理者对象提交请求，它不关心处理细节和请求的传递过程。

举例（购买请求决策，价格不同要由不同的级别决定：组长、部长、副部、总裁）。类图如下：



1 决策者抽象类，包含对请求处理的函数，同时还包含指定下一个决策者的函数

```
1. public abstract class Approver {
2.          Approver successor;
3.          String Name;
4.         public Approver(String Name)
5.         {
6.                 this.Name=Name;
7.         }
8.         public abstract void ProcessRequest( PurchaseRequest request);
9.         public void SetSuccessor(Approver successor) {
10.                // TODO Auto-generated method stub
11.                this.successor=successor;
12.        }
13. }
```

2 客户端以及请求

```
1. public class PurchaseRequest {
2.         private int Type = 0;
3.         private int Number = 0;
4.         private float Price = 0;
```

```
5.          private int ID = 0;
6.
7.          public PurchaseRequest(int Type, int Number, float Price) {
8.                  this.Type = Type;
9.                  this.Number = Number;
10.                 this.Price = Price;
11.         }
12.
13.         public int GetType() {
14.                 return Type;
15.         }
16.
17.         public float GetSum() {
18.                 return Number * Price;
19.         }
20.
21.         public int GetID() {
22.                 return (int) (Math.random() * 1000);
23.         }
24. }
25. public class Client {
26.
27.         public Client() {
28.
29.         }
30.
31.         public PurchaseRequest sendRequst(int Type, int Number, float Price) {
32.                 return new PurchaseRequest(Type, Number, Price);
33.         }
```

```
34.
35. }
```

3 组长、部长。。。继承决策者抽象类

```
1. public class GroupApprover extends Approver {
2.
3.         public GroupApprover(String Name) {
4.                 super(Name + " GroupLeader");
5.                 // TODO Auto-generated constructor stub
6.
7.         }
8.
9.         @Override
10.         public void ProcessRequest(PurchaseRequest request) {
11.                 // TODO Auto-generated method stub
12.
13.                 if (request.GetSum() < 5000) {
14.                         System.out.println("**This request " + request.GetID() + " will be handled by " + this.Name + " **");
15.                 } else {
16.                         successor.ProcessRequest(request);
17.                 }
18.         }
19.
20. }
21. public class DepartmentApprover extends Approver {
22.
23.         public DepartmentApprover(String Name) {
```

```
24.                 super(Name + " DepartmentLeader");
25.
26.         }
27.
28.         @Override
29.         public void ProcessRequest(PurchaseRequest request) {
30.                 // TODO Auto-generated method stub
31.
32.                 if ((5000 <= request.GetSum()) && (request.GetSum() < 10000)) {
33.                         System.out.println("**This request " + request.GetID()
34.                                         + " will be handled by " + this.Name + " **");
35.                 } else {
36.                         successor.ProcessRequest(request);
37.                 }
38.
39.         }
40.
41. }
```

4测试

```
1. public class MainTest {
2.
3.         public static void main(String[] args) {
4.
5.                 Client mClient = new Client();
6.                 Approver GroupLeader = new GroupApprover("Tom");
7.                 Approver DepartmentLeader = new DepartmentApprover("Jerry");
8.                 Approver VicePresident = new VicePresidentApprover("Kate");
```

```
                Approver President = new PresidentApprover("Bush");

                GroupLeader.SetSuccessor(VicePresident);
                DepartmentLeader.SetSuccessor(President);
                VicePresident.SetSuccessor(DepartmentLeader);
                President.SetSuccessor(GroupLeader);

                GroupLeader.ProcessRequest(mClient.sendRequst(1, 10000, 40));

        }

}
```

17 命令模式

定义：将一个请求封装为一个对象，使发出请求的责任和执行请求的责任分割开。这样两者之间通过命令对象进行沟通，这样方便将命令对象进行储存、传递、调用、增加与管理。

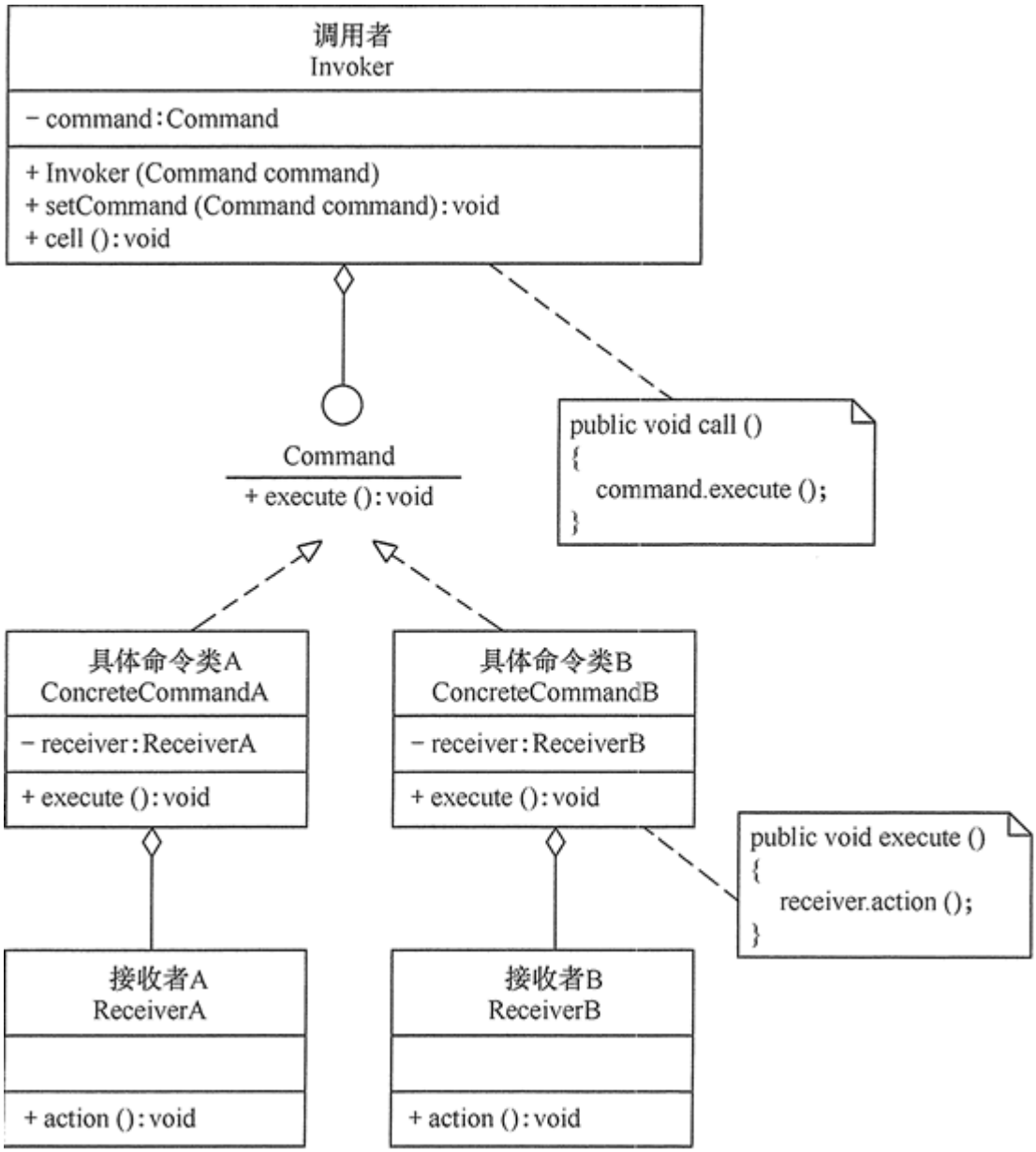
意图：将一个请求封装成一个对象，从而使您可以用不同的请求对客户进行参数化。

主要解决：在软件系统中，行为请求者与行为实现者通常是一种紧耦合的关系，但某些场合，比如需要对行为进行记录、撤销或重做、事务等处理时，这种无法抵御变化的紧耦合的设计就不太合适。

何时使用：在某些场合，比如要对行为进行"记录、撤销/重做、事务"等处理，这种无法抵御变化的紧耦合是不合适的。在这种情况下，如何将"行为请求者"与"行为实现者"解耦？将一组行为抽象为对象，可以实现二者之间的松耦合。

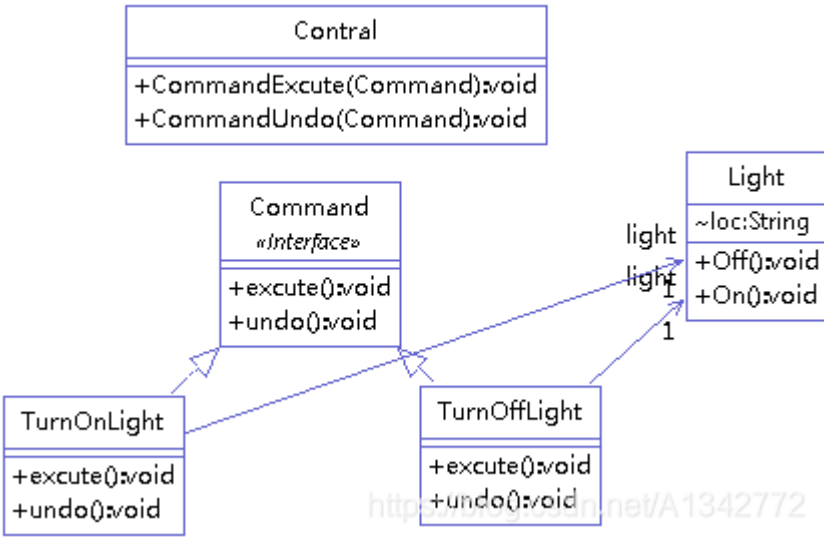
如何解决：通过调用者调用接受者执行命令，顺序：调用者→接受者→命令。

17.1模式结构和代码示例



1. 抽象命令类（Command）角色：声明执行命令的接口，拥有执行命令的抽象方法 execute()。
2. 具体命令角色（Concrete　Command）角色：是抽象命令类的具体实现类，它拥有接收者对象，并通过调用接收者的功能来完成命令要执行的操作。
3. 实现者/接收者（Receiver）角色：执行命令功能的相关操作，是具体命令对象业务的真正实现者。
4. 调用者/请求者（Invoker）角色：是请求的发送者，它通常拥有很多的命令对象，并通过访问命令对象来执行相关请求，它不直接访问接收者。

代码举例（开灯和关灯），类图如下：



1 命令抽象类

```
1. public interface Command {
2.
3.         public void excute();
4.         public void undo();
5.
6. }
```

2 具体命令对象

```
1. public class TurnOffLight implements Command {
2.
3.         private Light light;
4.
5.         public TurnOffLight(Light light) {
6.                 this.light = light;
7.         }
8.
9.         @Override
10.         public void excute() {
11.                 // TODO Auto-generated method stub
```

```
12.                 light.Off();
13.         }
14.
15.         @Override
16.         public void undo() {
17.                 // TODO Auto-generated method stub
18.                 light.On();
19.         }
20.
21. }
```

3 实现者

```
1. public class Light {
2.
3.         String loc = "";
4.
5.         public Light(String loc) {
6.                 this.loc = loc;
7.         }
8.
9.         public void On() {
10.
11.                 System.out.println(loc + " On");
12.         }
13.
14.         public void Off() {
15.
16.                 System.out.println(loc + " Off");
```

```
17.         }
18.
19. }
```

4 请求者

```
1. public class Contral{
2.
3.         public void CommandExcute(Command command) {
4.                 // TODO Auto-generated method stub
5.                 command.excute();
6.         }
7.
8.         public void CommandUndo(Command command) {
9.                 // TODO Auto-generated method stub
10.                command.undo();
11.        }
12.
13. }
```

18 状态模式

定义： 在状态模式中，我们创建表示各种状态的对象和一个行为随着状态对象改变而改变的 context 对象。

简单理解，一个拥有状态的context对象，在不同的状态下，其行为会发生改变。

意图：允许对象在内部状态发生改变时改变它的行为，对象看起来好像修改了它的类。

主要解决：对象的行为依赖于它的状态（属性），并且可以根据它的状态改变而改变它的相关行为。

何时使用：代码中包含大量与对象状态有关的条件语句。

如何解决：将各种具体的状态类抽象出来。

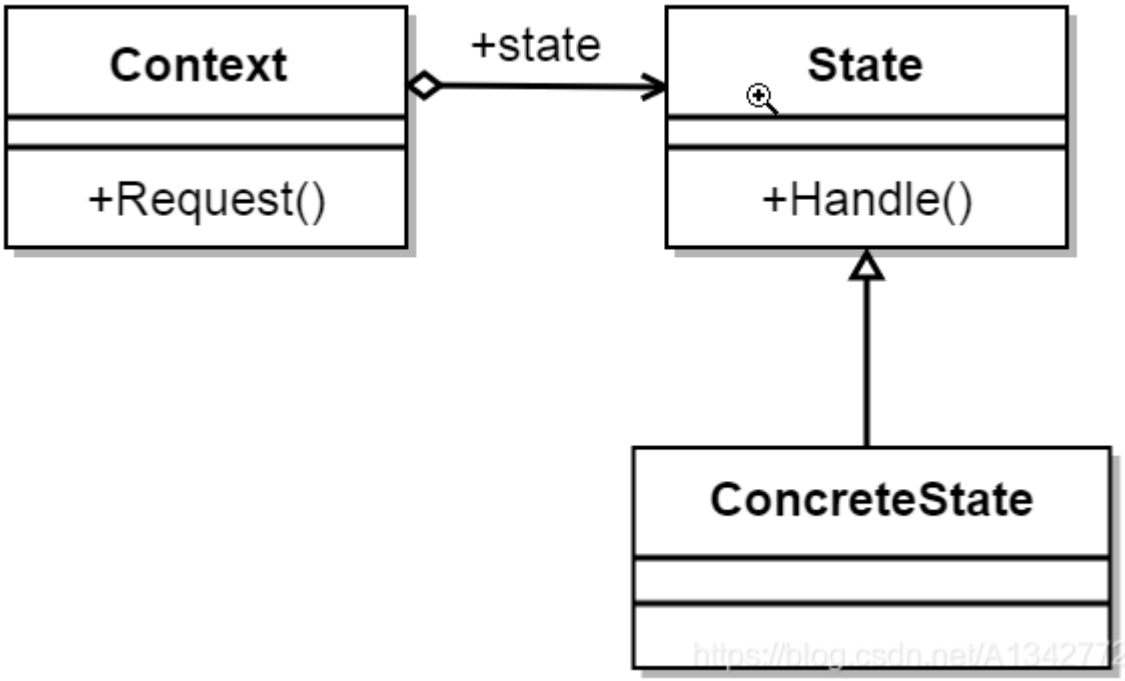
关键代码：通常命令模式的接口中只有一个方法。而状态模式的接口中有一个或者多个方法。而且，状态模式的实现类的方法，一般返回值，或者是改变实例变量的值。也就是说，状态模式一般和对象的状态有关。实现类的方法

有不同的功能，覆盖接口中的方法。状态模式和命令模式一样，也可以用于消除 if...else 等条件选择语句。

优点： 1、封装了转换规则。 2、枚举可能的状态，在枚举状态之前需要确定状态种类。 3、将所有与某个状态有关的行为放到一个类中，并且可以方便地增加新的状态，只需要改变对象状态即可改变对象的行为。 4、允许状态转换逻辑与状态对象合成一体，而不是某一个巨大的条件语句块。 5、可以让多个环境对象共享一个状态对象，从而减少系统中对象的个数。

缺点： 1、状态模式的使用必然会增加系统类和对象的个数。 2、状态模式的结构与实现都较为复杂，如果使用不当将导致程序结构和代码的混乱。 3、状态模式对"开闭原则"的支持并不太好，对于可以切换状态的状态模式，增加新的状态类需要修改那些负责状态转换的源代码，否则无法切换到新增状态，而且修改某个状态类的行为也需修改对应类的源代码。

18.1 模式结构和代码示例



- **State抽象状态角色**

  接口或抽象类，负责对象状态定义，并且封装环境角色以实现状态切换。

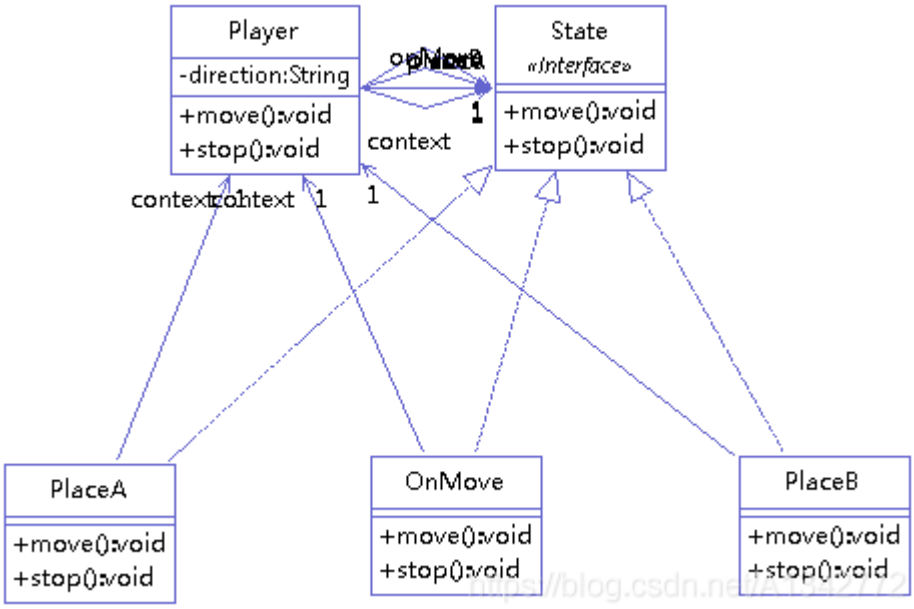
- **ConcreteState具体状态角色**

  具体状态主要有两个职责：一是处理本状态下的事情，二是从本状态如何过渡到其他状态。

- **Context环境角色**

  定义客户端需要的接口，并且负责具体状态的切换。

举例（人物在地点A向地点B移动，在地点B向地点A移动）。类图如下：



1 state接口

```
1. public interface State {
2.         public void stop();
3.         public void move();
4.
5. }
```

2 状态实例

```
1. public class PlaceA implements State {
2.
3.         private Player context;
4.
5.         public PlaceA(Player context) {
6.                 this.context = context;
7.         }
8.
9.         @Override
10.         public void move() {
```

```
                System.out.println("处于地点A,开始向B移动");
                System.out.println("---------");
                context.setDirection("AB");
                context.setState(context.onMove);

        }

        @Override
        public void stop() {
                // TODO Auto-generated method stub
                System.out.println("正处在地点A，不用停止移动");
                System.out.println("---------");
        }

}
```


3 context(player)拥有状态的对象

```
public class Player {

        State placeA;
        State placeB;
        State onMove;
        private State state;
        private String direction;

        public Player() {
                direction = "AB";
                placeA = new PlaceA(this);
```

```
12.                 placeB = new PlaceB(this);
13.                 onMove = new OnMove(this);
14.                 this.state = placeA;
15.         }
16.
17.         public void move() {
18.                 System.out.println("指令:开始移动");
19.                 state.move();
20.         }
21.
22.         public void stop() {
23.                 System.out.println("指令:停止移动");
24.                 state.stop();
25.         }
26.
27.         public State getState() {
28.                 return state;
29.         }
30.
31.         public void setState(State state) {
32.                 this.state = state;
33.         }
34.
35.         public void setDirection(String direction) {
36.                 this.direction = direction;
37.         }
38.
39.         public String getDirection() {
40.                 return direction;
```

```
41.         }
42.
43. }
```

19 备忘录模式

定义： 在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，以便以后当需要时能将该对象恢复到原先保存的状态。该模式又叫快照模式。

备忘录模式是一种对象行为型模式，其主要优点如下。

- 提供了一种可以恢复状态的机制。当用户需要时能够比较方便地将数据恢复到某个历史的状态。
- 实现了内部状态的封装。除了创建它的发起人之外，其他对象都不能够访问这些状态信息。
- 简化了发起人类。发起人不需要管理和保存其内部状态的各个备份，所有状态信息都保存在备忘录中，并由管理者进行管理，这符合单一职责原则。

其主要缺点是：资源消耗大。如果要保存的内部状态信息过多或者特别频繁，将会占用比较大的内存资源。

19.1 模式结构图和代码示例

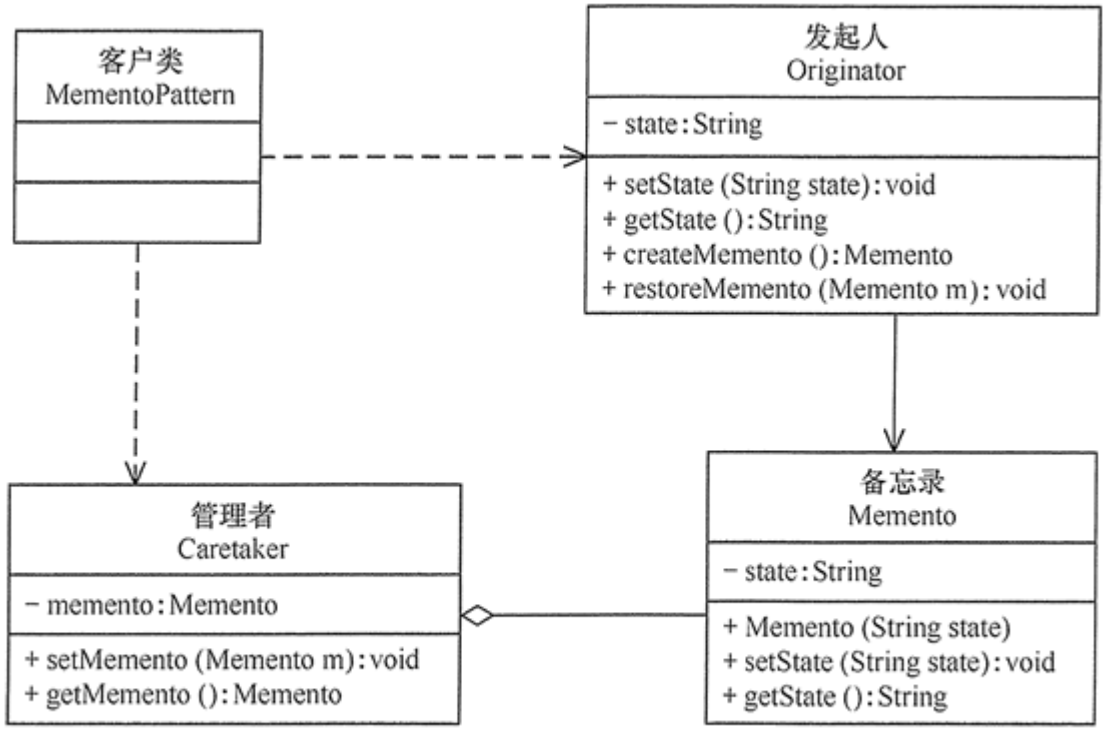


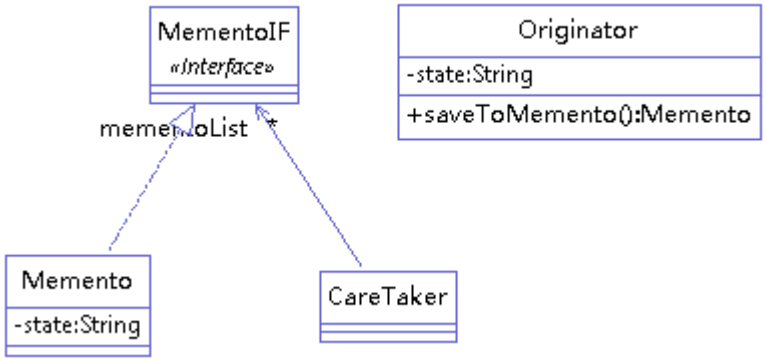
图1 备忘录模式的结构图

1. 发起人（Originator）角色：记录当前时刻的内部状态信息，提供创建备忘录和恢复备忘录数据的功能，实现其他业务功能，它可以访问备忘录里的所有信息。

2. 备忘录（Memento）角色：负责存储发起人的内部状态，在需要的时候提供这些内部状态给发起人。

3. 管理者（Caretaker）角色：对备忘录进行管理，提供保存与获取备忘录的功能，但其不能对备忘录的内容进行访问与修改。

举例（发起者通过备忘录存储信息和获取信息），类图如下：



1 备忘录接口

```
1. public interface MementoIF {
2.
3. }
```

2 备忘录

```
1. public class Memento implements MementoIF{
2.
3.         private String state;
4.
5.         public Memento(String state) {
6.                 this.state = state;
7.         }
8.
9.         public String getState(){
10.                return state;
11.        }
12.
```

```
13.
14. }
```

3 发起者

```
1. public class Originator {
2.
3.         private String state;
4.
5.         public String getState() {
6.                 return state;
7.         }
8.
9.         public void setState(String state) {
10.                 this.state = state;
11.         }
12.
13.         public Memento saveToMemento() {
14.                 return new Memento(state);
15.         }
16.
17.         public String getStateFromMemento(MementoIF memento) {
18.                 return ((Memento) memento).getState();
19.         }
20.
21. }
```

4 管理者

```
1. public class CareTaker {
```

```
2.
3.          private List<MementoIF> mementoList = new ArrayList<MementoIF>();
4.
5.          public void add(MementoIF memento) {
6.                  mementoList.add(memento);
7.          }
8.
9.          public MementoIF get(int index) {
10.                 return mementoList.get(index);
11.         }
12.
13. }
```

20 访问者模式

定义：将作用于某种数据结构中的各元素的操作分离出来封装成独立的类，使其在不改变数据结构的前提下可以添加作用于这些元素的新的操作，为数据结构中的每个元素提供多种访问方式。它将对数据的操作与数据结构进行分离。

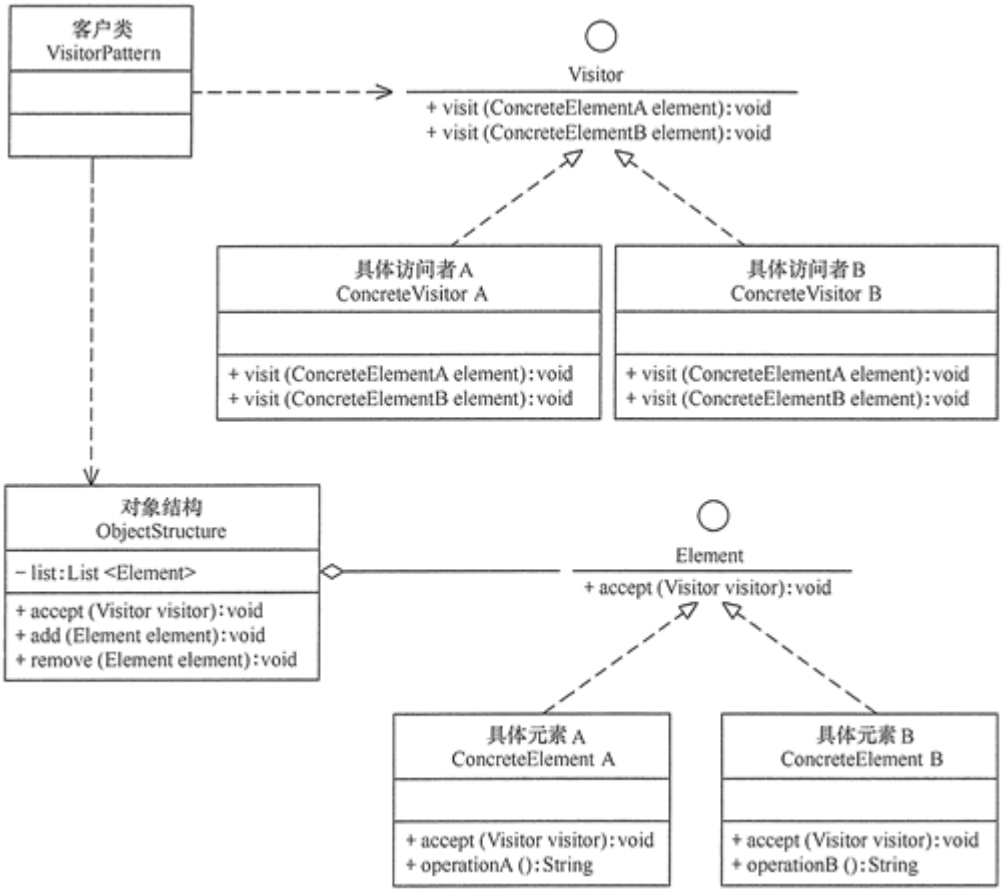
访问者（Visitor）模式是一种对象行为型模式，其主要优点如下。

1. 扩展性好。能够在不修改对象结构中的元素的情况下，为对象结构中的元素添加新的功能。
2. 复用性好。可以通过访问者来定义整个对象结构通用的功能，从而提高系统的复用程度。
3. 灵活性好。访问者模式将数据结构与作用于结构上的操作解耦，使得操作集合可相对自由地演化而不影响系统的数据结构。
4. 符合单一职责原则。访问者模式把相关的行为封装在一起，构成一个访问者，使每一个访问者的功能都比较单一。

访问者（Visitor）模式的主要缺点如下。

1. 增加新的元素类很困难。在访问者模式中，每增加一个新的元素类，都要在每一个具体访问者类中增加相应的具体操作，这违背了“开闭原则”。
2. 破坏封装。访问者模式中具体元素对访问者公布细节，这破坏了对象的封装性。
3. 违反了依赖倒置原则。访问者模式依赖了具体类，而没有依赖抽象类。

20.1 模式结构和代码示例



访问者模式包含以下主要角色。

1. 抽象访问者（Visitor）角色：定义一个访问具体元素的接口，为每个具体元素类对应一个访问操作 visit()，该操作中的参数类型标识了被访问的具体元素。
2. 具体访问者（ConcreteVisitor）角色：实现抽象访问者角色中声明的各个访问操作，确定访问者访问一个元素时该做什么。
3. 抽象元素（Element）角色：声明一个包含接受操作 accept() 的接口，被接受的访问者对象作为 accept() 方法的参数。
4. 具体元素（ConcreteElement）角色：实现抽象元素角色提供的 accept() 操作，其方法体通常都是 visitor.visit(this)，另外具体元素中可能还包含本身业务逻辑的相关操作。
5. 对象结构（Object Structure）角色：是一个包含元素角色的容器，提供让访问者对象遍历容器中的所有元素的方法，通常由 List、Set、Map 等聚合类实现。

1 抽象访问者

```
1. public interface Visitor {
2.
3.         abstract public void Visit(Element element);
4. }
```

2 具体访问者

```
1. public class CompensationVisitor implements Visitor {
2.
3.         @Override
4.         public void Visit(Element element) {
5.                 // TODO Auto-generated method stub
6.                 Employee employee = ((Employee) element);
7.
8.                 System.out.println(
9.                                 employee.getName() + "'s Compensation is " +
   (employee.getDegree() * employee.getVacationDays() * 10));
10.         }
11.
12. }
```

3 抽象元素

```
1. public interface Element {
2.         abstract public void Accept(Visitor visitor);
3.
4. }
```

4 具体元素

```
1. public class CompensationVisitor implements Visitor {
2.
3.         @Override
4.         public void Visit(Element element) {
5.                 // TODO Auto-generated method stub
6.                 Employee employee = ((Employee) element);
```

```
7.
8.                 System.out.println(
9.                                 employee.getName() + "'s Compensation is " +
    (employee.getDegree() * employee.getVacationDays() * 10));
10.         }
11.
12. }
```

5 对象结构

```
1. public class ObjectStructure {
2.         private HashMap<String, Employee> employees;
3.
4.         public ObjectStructure() {
5.                 employees = new HashMap();
6.         }
7.
8.         public void Attach(Employee employee) {
9.                 employees.put(employee.getName(), employee);
10.         }
11.
12.         public void Detach(Employee employee) {
13.                 employees.remove(employee);
14.         }
15.
16.         public Employee getEmployee(String name) {
17.                 return employees.get(name);
18.         }
19.
20.         public void Accept(Visitor visitor) {
```

```
21.                 for (Employee e : employees.values()) {
22.                         e.Accept(visitor);
23.                 }
24.         }
25.
26. }
```

21 中介者模式

定义：定义一个中介对象来封装一系列对象之间的交互，使原有对象之间的耦合松散，且可以独立地改变它们之间的交互。中介者模式又叫调停模式，它是迪米特法则的典型应用。

中介者模式是一种对象行为型模式，其主要优点如下。

1. 降低了对象之间的耦合性，使得对象易于独立地被复用。
2. 将对象间的一对多关联转变为一对一的关联，提高系统的灵活性，使得系统易于维护和扩展。

其主要缺点是：当同事类太多时，中介者的职责将很大，它会变得复杂而庞大，以至于系统难以维护。

21.1 模式结构和代码示例

中介者模式的结构图如图 1 所示。

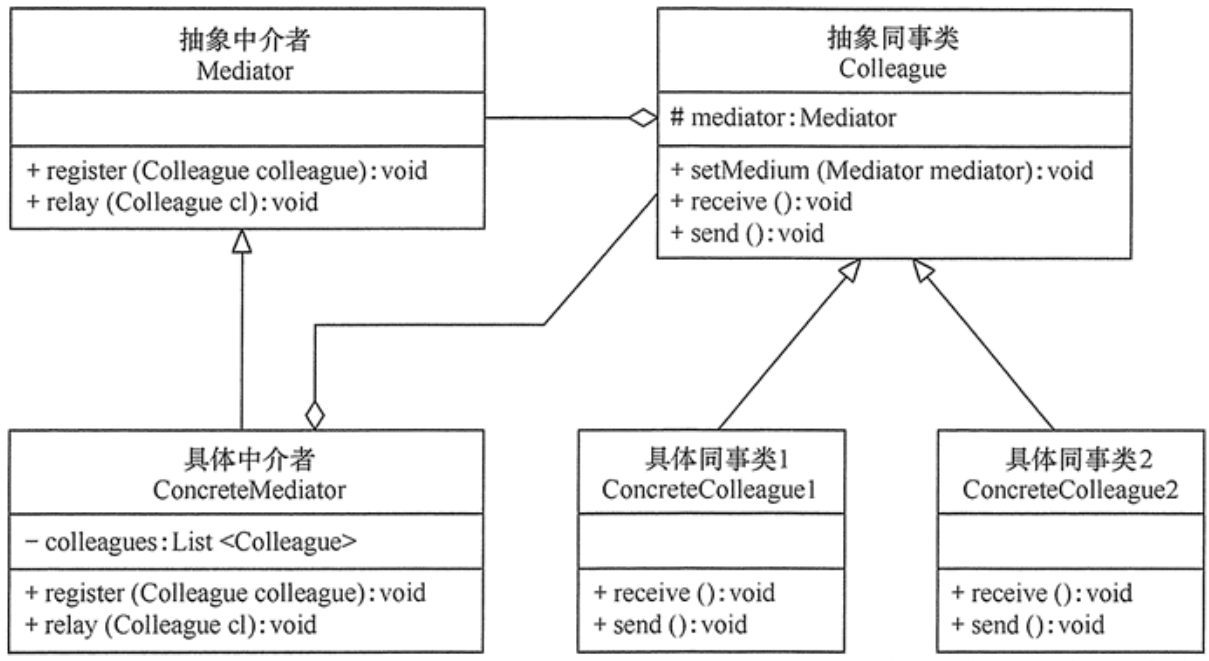
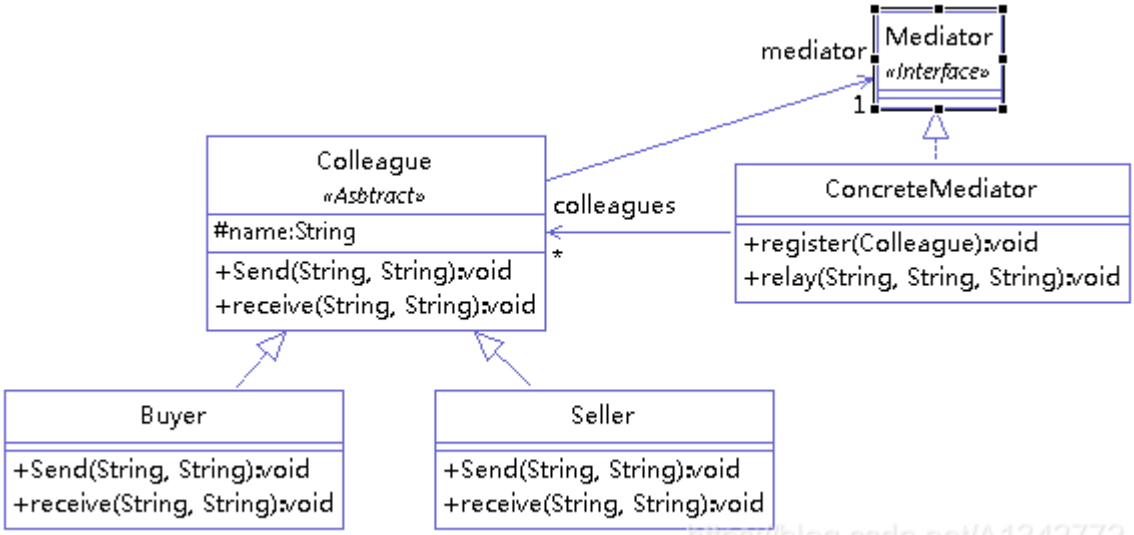


图1 中介者模式的结构图

1. 抽象中介者（Mediator）角色：它是中介者的接口，提供了同事对象注册与转发同事对象信息的抽象方法。
2. 具体中介者（ConcreteMediator）角色：实现中介者接口，定义一个 List 来管理同事对象，协调各个同事角色之间的交互关系，因此它依赖于同事角色。
3. 抽象同事类（Colleague）角色：定义同事类的接口，保存中介者对象，提供同事对象交互的抽象方法，实现所有相互影响的同事类的公共功能。
4. 具体同事类（Concrete Colleague）角色：是抽象同事类的实现者，当需要与其他同事对象交互时，由中介者对象负责后续的交互。

举例（通过中介卖方），类图如下：



1 抽象中介者

```
1. public interface Mediator {
2.
3.         void register(Colleague colleague); // 客户注册
4.
5.         void relay(String from, String to,String// 转发
6.
7. }
```

2 具体中介者

```
1. public class ConcreteMediator implements Mediator {
2.
3.         private List<Colleague> colleagues = new ArrayList<Colleague>();
```

```
4.
5.          @Override
6.          public void register(Colleague colleague) {
7.                  // TODO Auto-generated method stub
8.                  if (!colleagues.contains(colleague)) {
9.                          colleagues.add(colleague);
10.                         colleague.setMedium(this);
11.                 }
12.         }
13.
14.         @Override
15.         public void relay(String from, String to, String
16.                 // TODO Auto-generated method stub
17.                 for (Colleague cl : colleagues) {
18.
19.                         String name = cl.getName();
20.                         if (name.equals(to)) {
21.                                 cl.receive(from
22.                         }
23.
24.                 }
25.
26.         }
27.
28. }
```

3 抽象同事类

```
1. public abstract class Colleague {
```

```
2.
3.          protected Mediator mediator;
4.          protected String name;
5.
6.          public Colleague(String name) {
7.                  this.name = name;
8.          }
9.
10.         public void setMedium(Mediator mediator) {
11.
12.                 this.mediator = mediator;
13.
14.         }
15.
16.         public String getName() {
17.                 return name;
18.         }
19.
20.         public abstract void Send(String to, String
21.
22.         public abstract void receive(String from, String
23.
24. }
```

4 具体同事类

```
1. public class Buyer extends Colleague {
2.
3.          public Buyer(String name) {
```

```
4.
5.                  super(name);
6.
7.          }
8.
9.          @Override
10.         public void Send(String to, String
11.                 // TODO Auto-generated method stub
12.                 mediator.relay(name, to
13.         }
14.
15.         @Override
16.         public void receive(String from, String
17.                 // TODO Auto-generated method stub
18.                 System.out.println(name + "接收到来自" + from + "的消息:" +
19.         }
20.
21. }
```