# 策略模式解释

**策略模式的定义:**

     策略模式定义了一系列的算法，并将每一个算法**封装起**来，而且使它们还可以**相互替换**，策略模式让算法独立于使用它的客户而独立变化。

　　策略模式使这些算法在客户端调用它们的时候能够**互不影响**地变化。

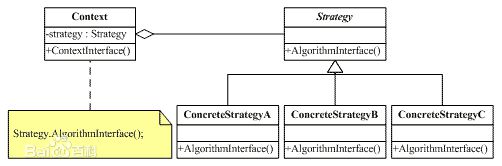
**策略模式的意义:**

　  策略模式使开发人员能够开发出由许多可替换的部分组成的软件，并且各个部分之间是低耦合的关系。

**低耦合**的特性使软件具有更强的可扩展性，易于维护；更重要的是，它大大提高了软件的可重用性。

**策略模式中有三个对象：**  
      环境对象(Context)：该类中实现了对抽象策略中定义的接口或者抽象类的引用。  
      抽象策略对象(Strategy)：它可由接口或抽象类来实现。  
      具体策略对象(ConcreteStrategy)：它封装了实现同不功能的不同算法。

**三者之间的关系可以用下图来表示**：



**策略模式的实现：**

　　1.对策略对象定义一个公共接口。

　　2.编写策略类，该类实现了上面的公共接口。

　　3.在使用策略对象的类中保存一个对策略对象的引用。

　　4.在使用策略对象的类中，实现对策略对象的set和get方法或者使用构造方法完成赋值。

**具体代码实现：**

定义一个接口( 抽象策略)，定义一个方法用于对两个整数进行运算

1 public interface Strategy {

2

3 public abstract int calculate(int a,int b);

4 }

定义具体的算法类，实现两个整数的加减乘除运算，但是外部调用形式需要符合接口的定义

**实现加法运算**

[复制代码](javascript:void(0);)

复制代码

1 public class AddStrategy implements Strategy{

2

3 @Override

4 public int calculate(int a, int b) {

5

6 return a+b;

7 }

8

9 }

复制代码

[复制代码](javascript:void(0);)

实现减法运算

[复制代码](javascript:void(0);)

复制代码

1 public class SubstractStrategy implements Strategy{

2

3 @Override

4 public int calculate(int a, int b) {

5

6 return a-b;

7 }

8

9 }

复制代码

[复制代码](javascript:void(0);)

实现乘法运算

[复制代码](javascript:void(0);)

复制代码

1 public class MultiplyStrategy implements Strategy {

2

3 @Override

4 public int calculate(int a, int b) {

5

6 return a\*b;

7 }

8

9 }

复制代码

[复制代码](javascript:void(0);)

实现除法运算

[复制代码](javascript:void(0);)

复制代码

1 public class DivisionStrategy implements Strategy{

2

3 @Override

4 public int calculate(int a, int b) {

5 if(b!=0){

6

7 return a/b;

8 }

9 else {

10 throw new RuntimeException("除数不能为零");

11 }

12 }

13

14 }

复制代码

[复制代码](javascript:void(0);)

定义具体的环境角色，持有Strategy接口的引用，并且有get和set方法可以完成策略更换。在环境角色中调用接口的方法完成动作。

[复制代码](javascript:void(0);)

复制代码

1 public class Context {

2

3 private Strategy strategy;

4

5 public Context(Strategy strategy) {

6 super();

7 this.strategy = strategy;

8 }

9

10 public Strategy getStrategy() {

11 return strategy;

12 }

13

14 public void setStrategy(Strategy strategy) {

15 this.strategy = strategy;

16 }

17 public int calculate(int a,int b){

18 return strategy.calculate(a, b);

19 }

20 }

复制代码

[复制代码](javascript:void(0);)

　这样在客户端在调用时，只需向环境角色设置相应的算法类，然后就可以得到相应的结果。

[复制代码](javascript:void(0);)

复制代码

1 public class StrategyTest {

2

3 /\*\*

4 \* @param args

5 \*/

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

7

8 //加法

9 Context context=new Context(new AddStrategy());

10 System.out.println(context.calculate(10, 5));

11 //减法

12 Context context2=new Context(new SubstractStrategy());

13 System.out.println(context2.calculate(3, 2));

14 //乘法

15 Context context3=new Context(new MultiplyStrategy());

16 System.out.println(context3.calculate(6, 8));

17 //除法

18 Context context4=new Context(new DivisionStrategy());

19 System.out.println(context4.calculate(90, 9));

20 }

21

22 }

复制代码

[复制代码](javascript:void(0);)

策略模式的缺点：

1、客户端必须知道所有的策略类，并自行决定使用哪一个策略类。

这就意味着客户端必须理解这些算法的区别，以便适时选择恰当的算法类。

换言之，策略模式  只适用于  **客户端知道**  **所有的**算法或行为的情况。

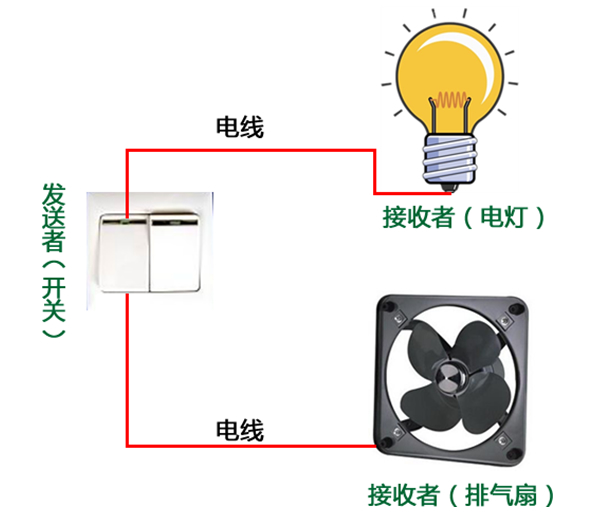
2、策略模式造成很多的策略类，每个具体策略类都会产生一个新类。

有时候可以通过把依赖于环境的状态保存到客户端里面，而将策略类设计成可共享的，这样策略类实例可以  **被不同** 客户端  使用。

换言之，可以使用  **享元模式**来减少对象的数量。

# 命令模式的解释

装修新房的最后几道工序之一是安装插座和开关，通过开关可以控制一些电器的打开和关闭，例如电灯或者排气扇。在购买开关时，我们并不知道它将来到底用于控制什么电器，也就是说，开关与电灯、排气扇并无直接关系，一个开关在安装之后可能用来控制电灯，也可能用来控制排气扇或者其他电器设备。开关与电器之间通过电线建立连接，如果开关打开，则电线通电，电器工作；反之，开关关闭，电线断电，电器停止工作。相同的开关可以通过不同的电线来控制不同的电器，如图1所示：



**图1 开关与电灯、排气扇示意图**

       在图1中，我们可以将开关理解成一个请求的发送者，用户通过它来发送一个“开灯”请求，而电灯是“开灯”请求的最终接收者和处理者，在图中，开关和电灯之间并不存在直接耦合关系，它们通过电线连接在一起，使用不同的电线可以连接不同的请求接收者，只需更换一根电线，相同的发送者（开关）即可对应不同的接收者（电器）。

       在软件开发中也存在很多与开关和电器类似的请求发送者和接收者对象，例如一个按钮，它可能是一个“关闭窗口”请求的发送者，而按钮点击事件处理类则是该请求的接收者。为了降低系统的耦合度，将请求的发送者和接收者解耦，我们可以使用一种被称之为命令模式的设计模式来设计系统，在命令模式中，发送者与接收者之间引入了新的命令对象（类似图1中的电线），将发送者的请求封装在命令对象中，再通过命令对象来调用接收者的方法。本章我们将学习用于将请求发送者和接收者解耦的命令模式。

## 1 自定义功能键

|  |
| --- |
| Sunny软件公司开发人员为公司内部OA系统开发了一个桌面版应用程序，该应用程序为用户提供了一系列自定义功能键，用户可以通过这些功能键来实现一些快捷操作。Sunny软件公司开发人员通过分析，发现不同的用户可能会有不同的使用习惯，在设置功能键的时候每个人都有自己的喜好，例如有的人喜欢将第一个功能键设置为“打开帮助文档”，有的人则喜欢将该功能键设置为“最小化至托盘”，为了让用户能够灵活地进行功能键的设置，开发人员提供了一个“功能键设置”窗口，该窗口界面如图2所示：  http://img.my.csdn.net/uploads/201304/15/1366033417_2468.jpg  **图2  “功能键设置”界面效果图**         通过如图2所示界面，用户可以将功能键和相应功能绑定在一起，还可以根据需要来修改功能键的设置，而且系统在未来可能还会增加一些新的功能或功能键。       Sunny软件公司某开发人员欲使用如下代码来实现功能键与功能处理类之间的调用关系： |

[复制代码](javascript:void(0);)

//FunctionButton：功能键类，请求发送者

class FunctionButton {

private HelpHandler help; //HelpHandler：帮助文档处理类，请求接收者

//在FunctionButton的onClick()方法中调用HelpHandler的display()方法

public void onClick() {

help = new HelpHandler();

help.display(); //显示帮助文档

}

}

[复制代码](javascript:void(0);)

在上述代码中，功能键类FunctionButton充当请求的发送者，帮助文档处理类HelpHandler充当请求的接收者，在发送者FunctionButton的onClick()方法中将调用接收者HelpHandler的display()方法。显然，如果使用上述代码，将给系统带来如下几个问题：

       (1) 由于请求发送者和请求接收者之间存在方法的直接调用，耦合度很高，**更换请求接收者必须修改发送者的源代码**，如果需要将请求接收者HelpHandler改为WindowHanlder（窗口处理类），则需要修改FunctionButton的源代码，违背了“开闭原则”。

       (2) FunctionButton类在设计和实现时功能已被固定，如果增加一个新的请求接收者，如果不修改原有的FunctionButton类，则必须增加一个新的与FunctionButton功能类似的类，这将**导致系统中类的个数急剧增加**。由于请求接收者HelpHandler、WindowHanlder等类之间可能不存在任何关系，它们没有共同的抽象层，因此也很难依据“依赖倒转原则”来设计FunctionButton。

       (3) **用户无法按照自己的需要来设置某个功能键的功能**，一个功能键类的功能一旦固定，在不修改源代码的情况下无法更换其功能，系统缺乏灵活性。

       不难得知，所有这些问题的产生都是因为请求发送者FunctionButton类和请求接收者HelpHandler、WindowHanlder等类之间存在直接耦合关系，如何降低请求发送者和接收者之间的耦合度，让相同的发送者可以对应不同的接收者？这是Sunny软件公司开发人员在设计“功能键设置”模块时不得不考虑的问题。命令模式正为解决这类问题而诞生，此时，如果我们使用命令模式，可以在一定程度上解决上述问题（**注：命令模式无法解决类的个数增加的问题**），下面就让我们正式进入命令模式的学习，看看命令模式到底如何实现请求发送者和接收者解耦。

## 2 命令模式概述

       在软件开发中，我们经常需要向某些对象发送请求（调用其中的某个或某些方法），但是并不知道请求的接收者是谁，也不知道被请求的操作是哪个，此时，我们特别希望能够以一种松耦合的方式来设计软件，使得请求发送者与请求接收者能够消除彼此之间的耦合，让对象之间的调用关系更加灵活，可以灵活地指定请求接收者以及被请求的操作。命令模式为此类问题提供了一个较为完美的解决方案。

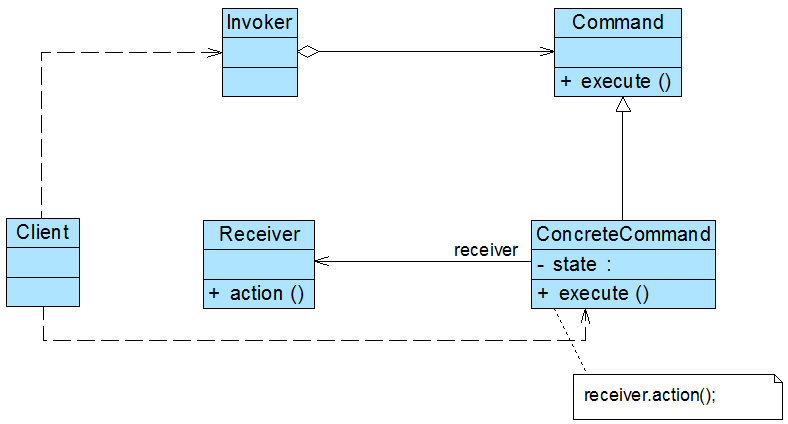
**命令模式可以将请求发送者和接收者完全解耦，发送者与接收者之间没有直接引用关系，发送请求的对象只需要知道如何发送请求，而不必知道如何完成请求**。

       命令模式定义如下：

|  |
| --- |
| **命令模式(Command Pattern)：将一个请求封装为一个对象，从而让我们可用不同的请求对客户进行参数化；对请求排队或者记录请求日志，以及支持可撤销的操作。命令模式是一种对象行为型模式，其别名为动作(Action)模式或事务(Transaction)模式。** |

       命令模式的定义比较复杂，提到了很多术语，例如“用不同的请求对客户进行参数化”、“对请求排队”，“记录请求日志”、“支持可撤销操作”等，在后面我们将对这些术语进行一一讲解。

       命令模式的核心在于引入了命令类，通过命令类来降低发送者和接收者的耦合度，请求发送者只需指定一个命令对象，再通过命令对象来调用请求接收者的处理方法，其结构如图3所示：



**图3 命令模式结构图**

       在命令模式结构图中包含如下几个角色：

       ● Command（抽象命令类）：抽象命令类一般是一个抽象类或接口，在其中声明了用于执行请求的execute()等方法，通过这些方法可以调用请求接收者的相关操作。

       ● ConcreteCommand（具体命令类）：具体命令类是抽象命令类的子类，实现了在抽象命令类中声明的方法，它对应具体的接收者对象，将接收者对象的动作绑定其中。在实现execute()方法时，将调用接收者对象的相关操作(Action)。

       ● Invoker（调用者）：调用者即请求发送者，它通过命令对象来执行请求。一个调用者并不需要在设计时确定其接收者，因此它只与抽象命令类之间存在关联关系。在程序运行时可以将一个具体命令对象注入其中，再调用具体命令对象的execute()方法，从而实现间接调用请求接收者的相关操作。

       ● Receiver（接收者）：接收者执行与请求相关的操作，它具体实现对请求的业务处理。

**命令模式的本质是对请求进行封装，一个请求对应于一个命令，将发出命令的责任和执行命令的责任分割开**。每一个命令都是一个操作：请求的一方发出请求要求执行一个操作；接收的一方收到请求，并执行相应的操作。**命令模式允许请求的一方和接收的一方独立开来，使得请求的一方不必知道接收请求的一方的接口，更不必知道请求如何被接收、操作是否被执行、何时被执行，以及是怎么被执行的**。

**命令模式的关键在于引入了抽象命令类，请求发送者针对抽象命令类编程，只有实现了抽象命令类的具体命令才与请求接收者相关联。**在最简单的抽象命令类中只包含了一个抽象的execute()方法，每个具体命令类将一个Receiver类型的对象作为一个实例变量进行存储，从而具体指定一个请求的接收者，不同的具体命令类提供了execute()方法的不同实现，并调用不同接收者的请求处理方法。

        典型的抽象命令类代码如下所示：

abstract class Command {

public abstract void execute();

}

对于请求发送者即调用者而言，将针对抽象命令类进行编程，**可以通过构造注入或者设值注入的方式在运行时传入具体命令类对象**，**并在业务方法中调用命令对象的execute()方法**，其典型代码如下所示：

[复制代码](javascript:void(0);)

class Invoker {

private Command command;

//构造注入

public Invoker(Command command) {

this.command = command;

}

//设值注入

public void setCommand(Command command) {

this.command = command;

}

//业务方法，用于调用命令类的execute()方法

public void call() {

command.execute();

}

}

[复制代码](javascript:void(0);)

具体命令类继承了抽象命令类，它与请求接收者相关联，实现了在抽象命令类中声明的execute()方法，并在实现时调用接收者的请求响应方法action()，其典型代码如下所示：

[复制代码](javascript:void(0);)

class ConcreteCommand extends Command {

private Receiver receiver; //维持一个对请求接收者对象的引用

public void execute() {

receiver.action(); //调用请求接收者的业务处理方法action()

}

}

[复制代码](javascript:void(0);)

请求接收者Receiver类具体实现对请求的业务处理，它提供了action()方法，用于执行与请求相关的操作，其典型代码如下所示：

class Receiver {

public void action() {

//具体操作

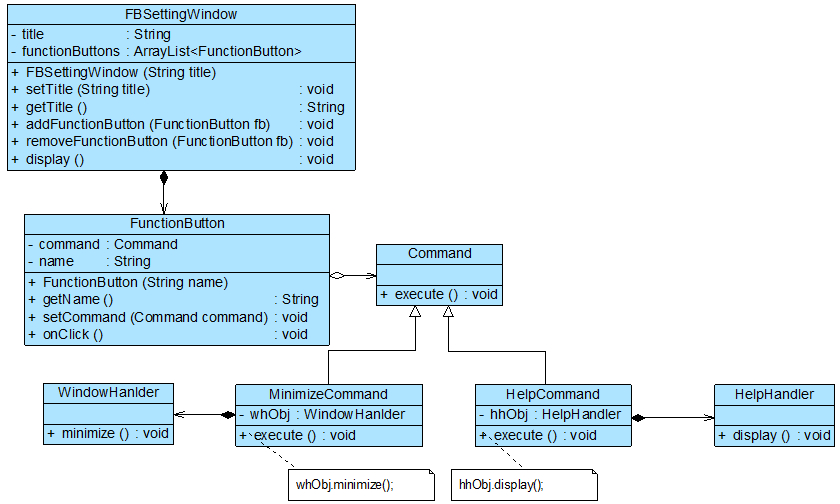
}

}

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | **疑问** | **思考**  一个请求发送者能否对应多个请求接收者？如何实现？ | |

**3 完整解决方案**

       为了降低功能键与功能处理类之间的耦合度，让用户可以自定义每一个功能键的功能，Sunny软件公司开发人员使用命令模式来设计“自定义功能键”模块，其核心结构如图4所示：



**图4 自定义功能键核心结构图**

       在图4中，FBSettingWindow是“功能键设置”界面类，FunctionButton充当请求调用者，Command充当抽象命令类，MinimizeCommand和HelpCommand充当具体命令类，WindowHanlder和HelpHandler充当请求接收者。完整代码如下所示：

[复制代码](javascript:void(0);)

import java.util.\*;

//功能键设置窗口类

class FBSettingWindow {

private String title; //窗口标题

//定义一个ArrayList来存储所有功能键

private ArrayList<FunctionButton> functionButtons = new ArrayList<FunctionButton>();

public FBSettingWindow(String title) {

this.title = title;

}

public void setTitle(String title) {

this.title = title;

}

public String getTitle() {

return this.title;

}

public void addFunctionButton(FunctionButton fb) {

functionButtons.add(fb);

}

public void removeFunctionButton(FunctionButton fb) {

functionButtons.remove(fb);

}

//显示窗口及功能键

public void display() {

System.out.println("显示窗口：" + this.title);

System.out.println("显示功能键：");

for (Object obj : functionButtons) {

System.out.println(((FunctionButton)obj).getName());

}

System.out.println("------------------------------");

}

}

//功能键类：请求发送者

class FunctionButton {

private String name; //功能键名称

private Command command; //维持一个抽象命令对象的引用

public FunctionButton(String name) {

this.name = name;

}

public String getName() {

return this.name;

}

//为功能键注入命令

public void setCommand(Command command) {

this.command = command;

}

//发送请求的方法

public void onClick() {

System.out.print("点击功能键：");

command.execute();

}

}

//抽象命令类

abstract class Command {

public abstract void execute();

}

//帮助命令类：具体命令类

class HelpCommand extends Command {

private HelpHandler hhObj; //维持对请求接收者的引用

public HelpCommand() {

hhObj = new HelpHandler();

}

//命令执行方法，将调用请求接收者的业务方法

public void execute() {

hhObj.display();

}

}

//最小化命令类：具体命令类

class MinimizeCommand extends Command {

private WindowHanlder whObj; //维持对请求接收者的引用

public MinimizeCommand() {

whObj = new WindowHanlder();

}

//命令执行方法，将调用请求接收者的业务方法

public void execute() {

whObj.minimize();

}

}

//窗口处理类：请求接收者

class WindowHanlder {

public void minimize() {

System.out.println("将窗口最小化至托盘！");

}

}

//帮助文档处理类：请求接收者

class HelpHandler {

public void display() {

System.out.println("显示帮助文档！");

}

}

[复制代码](javascript:void(0);)

 为了提高系统的灵活性和可扩展性，我们将具体命令类的类名存储在配置文件中，并通过工具类XMLUtil来读取配置文件并反射生成对象，XMLUtil类的代码如下所示：

[复制代码](javascript:void(0);)

import javax.xml.parsers.\*;

import org.w3c.dom.\*;

import org.xml.sax.SAXException;

import java.io.\*;

public class XMLUtil {

//该方法用于从XML配置文件中提取具体类类名，并返回一个实例对象，可以通过参数的不同返回不同类名节点所对应的实例

public static Object getBean(int i) {

try {

//创建文档对象

DocumentBuilderFactory dFactory = DocumentBuilderFactory.newInstance();

DocumentBuilder builder = dFactory.newDocumentBuilder();

Document doc;

doc = builder.parse(new File("config.xml"));

//获取包含类名的文本节点

NodeList nl = doc.getElementsByTagName("className");

Node classNode = null;

if (0 == i) {

classNode = nl.item(0).getFirstChild();

}

else {

classNode = nl.item(1).getFirstChild();

}

String cName = classNode.getNodeValue();

//通过类名生成实例对象并将其返回

Class c = Class.forName(cName);

Object obj = c.newInstance();

return obj;

}

catch(Exception e){

e.printStackTrace();

return null;

}

}

}

[复制代码](javascript:void(0);)

    配置文件config.xml中存储了具体建造者类的类名，代码如下所示：

**[java]** [view plain](http://blog.csdn.net/lovelion/article/details/8806049) [copy](http://blog.csdn.net/lovelion/article/details/8806049)

1. <?xml version="1.0"?>
2. <config>
3. <className>HelpCommand</className>
4. <className>MinimizeCommand</className>
5. </config>

编写如下客户端[测试](http://lib.csdn.net/base/softwaretest)代码：

[复制代码](javascript:void(0);)

class Client {

public static void main(String args[]) {

FBSettingWindow fbsw = new FBSettingWindow("功能键设置");

FunctionButton fb1,fb2;

fb1 = new FunctionButton("功能键1");

fb2 = new FunctionButton("功能键1");

Command command1,command2;

//通过读取配置文件和反射生成具体命令对象

command1 = (Command)XMLUtil.getBean(0);

command2 = (Command)XMLUtil.getBean(1);

//将命令对象注入功能键

fb1.setCommand(command1);

fb2.setCommand(command2);

fbsw.addFunctionButton(fb1);

fbsw.addFunctionButton(fb2);

fbsw.display();

//调用功能键的业务方法

fb1.onClick();

fb2.onClick();

}

}

[复制代码](javascript:void(0);)

    编译并运行程序，输出结果如下：

|  |
| --- |
| 显示窗口：功能键设置  显示功能键：  功能键1  功能键1  ------------------------------  点击功能键：显示帮助文档！  点击功能键：将窗口最小化至托盘！ |

       如果需要修改功能键的功能，例如某个功能键可以实现“自动截屏”，只需要对应增加一个新的具体命令类，在该命令类与屏幕处理者(ScreenHandler)之间创建一个关联关系，然后将该具体命令类的对象通过配置文件注入到某个功能键即可，原有代码无须修改，符合“开闭原则”。在此过程中，**每一个具体命令类对应一个请求的处理者（接收者），通过向请求发送者注入不同的具体命令对象可以使得相同的发送者对应不同的接收者，从而实现“将一个请求封装为一个对象，用不同的请求对客户进行参数化”**，客户端只需要将具体命令对象作为参数注入请求发送者，无须直接操作请求的接收者。

**4 命令队列的实现**

       有时候我们需要**将多个请求排队**，**当一个请求发送者发送一个请求时，将不止一个请求接收者产生响应，这些请求接收者将逐个执行业务方法，完成对请求的处理**。此时，我们可以通过**命令队列**来实现。

       命令队列的实现方法有多种形式，其中最常用、灵活性最好的一种方式是增加一个CommandQueue类，由该类来负责存储多个命令对象，而不同的命令对象可以对应不同的请求接收者，CommandQueue类的典型代码如下所示：

[复制代码](javascript:void(0);)

import java.util.\*;

class CommandQueue {

//定义一个ArrayList来存储命令队列

private ArrayList<Command> commands = new ArrayList<Command>();

public void addCommand(Command command) {

commands.add(command);

}

public void removeCommand(Command command) {

commands.remove(command);

}

//循环调用每一个命令对象的execute()方法

public void execute() {

for (Object command : commands) {

((Command)command).execute();

}

}

}

[复制代码](javascript:void(0);)

       在增加了命令队列类CommandQueue以后，请求发送者类Invoker将针对CommandQueue编程，代码修改如下：

[复制代码](javascript:void(0);)

class Invoker {

private CommandQueue commandQueue; //维持一个CommandQueue对象的引用

//构造注入

public Invoker(CommandQueue commandQueue) {

this. commandQueue = commandQueue;

}

//设值注入

public void setCommandQueue(CommandQueue commandQueue) {

this.commandQueue = commandQueue;

}

//调用CommandQueue类的execute()方法

public void call() {

commandQueue.execute();

}

}

[复制代码](javascript:void(0);)

  命令队列与我们常说的“批处理”有点类似。批处理，顾名思义，可以对一组对象（命令）进行批量处理，当一个发送者发送请求后，将有一系列接收者对请求作出响应，命令队列可以用于设计批处理应用程序，如果请求接收者的接收次序没有严格的先后次序，我们还可以使用多线程技术来并发调用命令对象的execute()方法，从而提高程序的执行效率。

## 5 撤销操作的实现

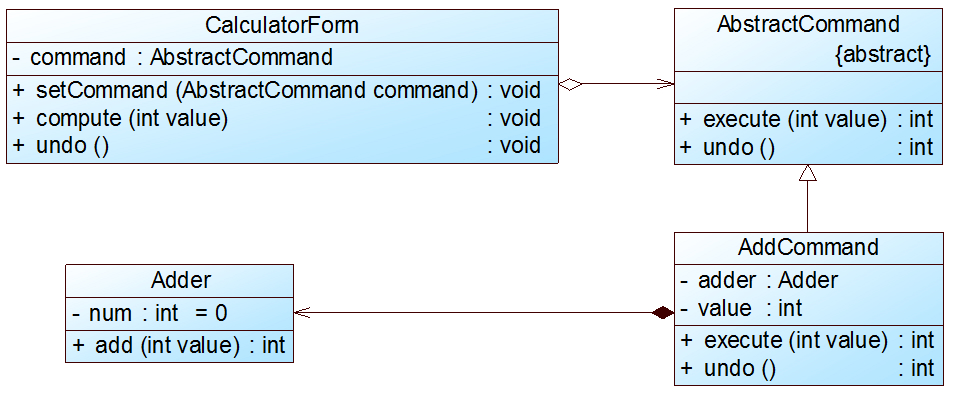
       在命令模式中，我们可以通过调用一个命令对象的execute()方法来实现对请求的处理，如果需要撤销(Undo)请求，可通过在命令类中增加一个逆向操作来实现。

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | **微笑** | **扩展**  除了通过一个**逆向操作**来实现撤销(Undo)外，还可以通过**保存对象的历史状态**来实现撤销，后者可使用备忘录模式(Memento Pattern)来实现。 | |

       下面通过一个简单的实例来学习如何使用命令模式实现撤销操作：

|  |
| --- |
| Sunny软件公司欲开发一个简易计算器，该计算器可以实现简单的数学运算，还可以对运算实施撤销操作。 |

       Sunny软件公司开发人员使用命令模式设计了如图5所示结构图，其中计算器界面类CalculatorForm充当请求发送者，实现了数据求和功能的加法类Adder充当请求接收者，界面类可间接调用加法类中的add()方法实现加法运算，并且提供了可撤销加法运算的undo()方法。



**图5  简易计算器结构图**

       本实例完整代码如下所示：

[复制代码](javascript:void(0);)

//加法类：请求接收者

class Adder {

private int num=0; //定义初始值为0

//加法操作，每次将传入的值与num作加法运算，再将结果返回

public int add(int value) {

num += value;

return num;

}

}

//抽象命令类

abstract class AbstractCommand {

public abstract int execute(int value); //声明命令执行方法execute()

public abstract int undo(); //声明撤销方法undo()

}

//具体命令类

class ConcreteCommand extends AbstractCommand {

private Adder adder = new Adder();

private int value;

//实现抽象命令类中声明的execute()方法，调用加法类的加法操作

public int execute(int value) {

this.value=value;

return adder.add(value);

}

//实现抽象命令类中声明的undo()方法，通过加一个相反数来实现加法的逆向操作

public int undo() {

return adder.add(-value);

}

}

//计算器界面类：请求发送者

class CalculatorForm {

private AbstractCommand command;

public void setCommand(AbstractCommand command) {

this.command = command;

}

//调用命令对象的execute()方法执行运算

public void compute(int value) {

int i = command.execute(value);

System.out.println("执行运算，运算结果为：" + i);

}

//调用命令对象的undo()方法执行撤销

public void undo() {

int i = command.undo();

System.out.println("执行撤销，运算结果为：" + i);

}

}

//编写如下客户端测试代码：class Client {

public static void main(String args[]) {

CalculatorForm form = new CalculatorForm();

AbstractCommand command;

command = new ConcreteCommand();

form.setCommand(command); //向发送者注入命令对象

form.compute(10);

form.compute(5);

form.compute(10);

form.undo();

}

}

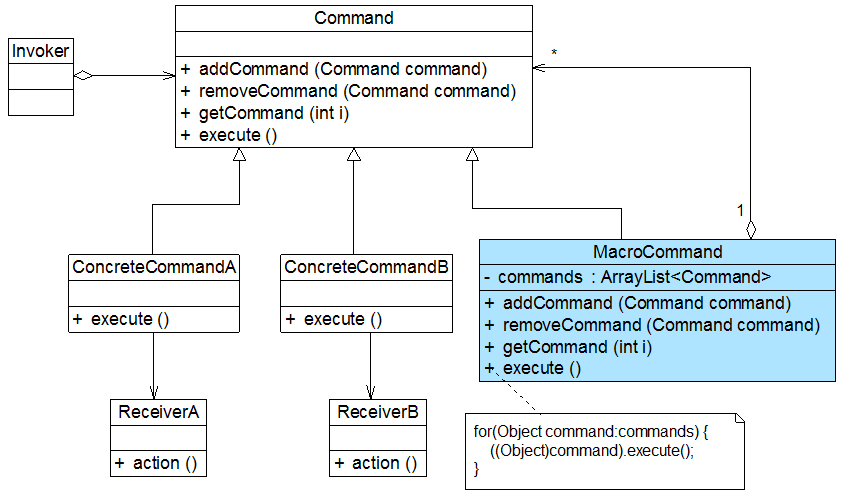
[复制代码](javascript:void(0);)

编译并运行程序，输出结果如下：

|  |
| --- |
| 增加新节点：网站首页  增加新节点：端口号  修改节点：网站首页  修改节点：端口号  ----------------------------  保存配置  ----------------------------  恢复配置  ----------------------------  **增加新节点：网站首页**  **增加新节点：端口号**  **修改节点：网站首页**  **修改节点：端口号** |

## 7 宏命令

**宏命令(Macro Command)又称为组合命令，它是组合模式和命令模式联用的产物**。宏命令是一个具体命令类，它拥有一个集合属性，在该集合中包含了对其他命令对象的引用。**通常宏命令不直接与请求接收者交互，而是通过它的成员来调用接收者的方法**。当调用宏命令的execute()方法时，将递归调用它所包含的每个成员命令的execute()方法，一个宏命令的成员可以是简单命令，还可以继续是宏命令。**执行一个宏命令将触发多个具体命令的执行，从而实现对命令的批处理**，其结构如图7所示：



**图7  宏命令结构图**

## 8 命令模式总结

       命令模式是一种使用频率非常高的设计模式，它可以将请求发送者与接收者解耦，请求发送者通过命令对象来间接引用请求接收者，使得系统具有更好的灵活性和可扩展性。在基于GUI的软件开发，无论是在电脑桌面应用还是在移动应用中，命令模式都得到了广泛的应用。

**1. 主要优点**

       命令模式的主要优点如下：

       (1) **降低系统的耦合度**。由于请求者与接收者之间不存在直接引用，因此请求者与接收者之间实现完全解耦，相同的请求者可以对应不同的接收者，同样，相同的接收者也可以供不同的请求者使用，两者之间具有良好的独立性。

       (2) **新的命令可以很容易地加入到系统中**。由于增加新的具体命令类不会影响到其他类，因此增加新的具体命令类很容易，无须修改原有系统源代码，甚至客户类代码，满足“开闭原则”的要求。

       (3) **可以比较容易地设计一个命令队列或宏命令（组合命令）**。

       (4) **为请求的撤销(Undo)和恢复(Redo)操作提供了一种设计和实现方案**。

**2. 主要缺点**

       命令模式的主要缺点如下：

**使用命令模式可能会导致某些系统有过多的具体命令类**。因为针对每一个对请求接收者的调用操作都需要设计一个具体命令类，因此在某些系统中可能需要提供大量的具体命令类，这将影响命令模式的使用。

**3. 适用场景**

      在以下情况下可以考虑使用命令模式：

       (1) 系统需要将请求调用者和请求接收者解耦，使得调用者和接收者不直接交互。请求调用者无须知道接收者的存在，也无须知道接收者是谁，接收者也无须关心何时被调用。

       (2) 系统需要在不同的时间指定请求、将请求排队和执行请求。一个命令对象和请求的初始调用者可以有不同的生命期，换言之，最初的请求发出者可能已经不在了，而命令对象本身仍然是活动的，可以通过该命令对象去调用请求接收者，而无须关心请求调用者的存在性，可以通过请求日志文件等机制来具体实现。

       (3) 系统需要支持命令的撤销(Undo)操作和恢复(Redo)操作。

       (4) 系统需要将一组操作组合在一起形成宏命令。

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | **微笑** | **练习**  Sunny软件公司欲开发一个基于Windows平台的公告板系统。该系统提供了一个主菜单(Menu)，在主菜单中包含了一些菜单项(MenuItem)，可以通过Menu类的addMenuItem()方法增加菜单项。菜单项的主要方法是click()，每一个菜单项包含一个抽象命令类，具体命令类包括OpenCommand(打开命令)，CreateCommand(新建命令)，EditCommand(编辑命令)等，命令类具有一个execute()方法，用于调用公告板系统界面类(BoardScreen)的open()、create()、edit()等方法。试使用命令模式设计该系统，以便降低MenuItem类与BoardScreen类之间的耦合度。 | |

# 区别

策略模式 把易于变化的行为分别封装起来，让它们之间可以互相替换， 让这些行为的变化独立于拥有这些行为的客户。  
  GoF《设计模式》中说道：定义一系列算法，把它们一个个封装起来，并且使它们可以相互替换。该模式使得算法可独立于它们的客户变化。  
  
Command命令模式是一种对象行为型模式，它主要解决的问题是：在软件构建过程中，“行为请求者”与“行为实现者”通常呈现一种“紧耦合”的问题。  
  GoF《设计模式》中说道：将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可撤销的操作。  
  
从这点看：

策略模式是通过不同的算法做同一件事情：例如排序

而命令模式则是通过不同的命令做不同的事情，常含有（关联）接收者。

目标不同！

**命令模式是含有不同的命令（含有接收者的请求）：做不同的事情；隐藏接收者执行细节。常见菜单事件，**

**而策略模式含有不同的算法，做相同的事情；**

**区别在于是否含有接收者。命令模式含有，策略模式不含有。**

命令模式中的命令可以单独运行

打个比喻就是：

**命令模式等于菜单中的复制，移动，压缩等，而策略模式是其中一个菜单的例如复制到不同算法实现。**