命令模式

命令模式的定义：将一个请求封装成一个对象，从而让用户使用不同的请求把客户端参数化；对请求排队或记录日志，以及支持可撤销的操作。

1. 场景的设计

   利用命令模式描述顾客在餐馆进行点餐吃饭的处理过程

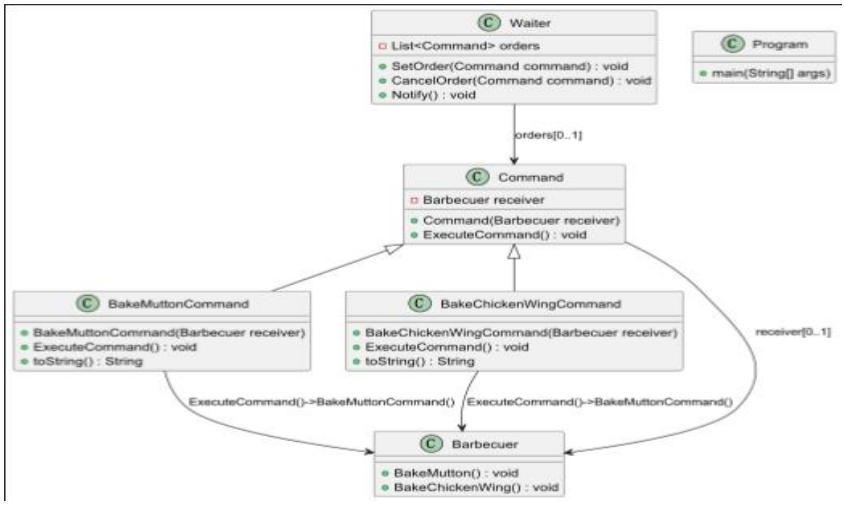
   在点餐的整个流程中，包括顾客、服务员以及厨师三类基本角色，其中：

   - 顾客按单点菜
   - 服务员负责处理顾客的菜单
   - 厨师根据菜单进行炒菜

2. 引入命令模式的设计思想，构建实现上述过程的类图结构以及角色定义，利用面向对象程序设计语言实现对顾客点餐过程的描述。

3. 通过对命令模式类图与结构功能的理解，编写实现命令模式的一般化程序。

类图



```java
//烤肉串者
public class Barbecuer  6 个用法
{
    public void BakeMutton() { System.out.println("kebabs of mutton!"); }

    public void BakeChickenWing() { System.out.println("Roast chicken wings!"); }
}
```

先定义了一个烤肉串者类 Barbecuer，拥有两个方法 BakeMutton 和 BakeChickenWing 用于指示烤羊肉串和烤鸡翅

```java
abstract class Command  10 个用法  2 个继承者
{
    protected Barbecuer receiver;  3 个用法

    @Contract(pure = true)
    public Command(Barbecuer receiver) { this.receiver = receiver; }

    abstract public void ExecuteCommand();  1 个用法  2 个实现
}
```

接着定义一个抽象类 Command 作为命令，内有一个受保护的属性 receiver 用于存储烤肉串者，一个 Command 方法接收烧烤命令，还有一个抽象方法用于执行烧烤命令

```java
class BakeMuttonCommand extends Command  2 个用法
{
    public BakeMuttonCommand(Barbecuer receiver) { super(receiver); }

    public void ExecuteCommand() { receiver.BakeMutton(); }

    public String toString() { return "BakeMuttonCommand"; }
}
class BakeChickenWingCommand extends Command  1 个用法
{
    public BakeChickenWingCommand(Barbecuer receiver) { super(receiver); }

    public void ExecuteCommand() { receiver.BakeChickenWing(); }


    public String toString() { return "BakeChickenWingCommand"; }
}
```

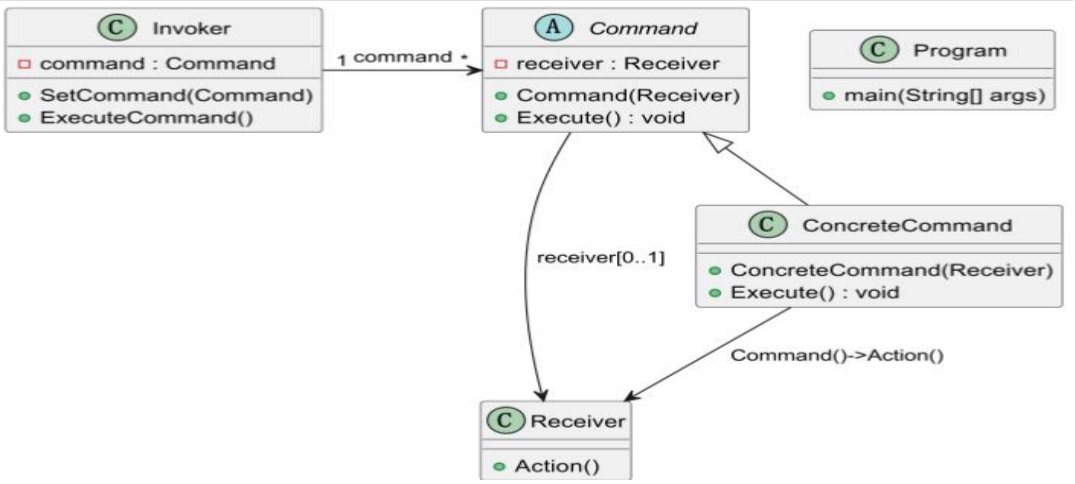
定义烤羊肉命令 BakeMuttonCommand 和烤鸡翅命令 BakeChickenWingCommand 继承 Command,调用父类的构造器来进行构造，实现 ExecuteCommand 方法

```java
public class Waiter  2 个用法
{
    private final List<Command> orders = new LinkedList<>();  3 个用法

    //设置订单
    public void SetOrder( @NotNull Command command)  3 个用法
    {
        if (Objects.equals( a: command.toString(),  b: "BakeChickenWingCommand"))
        {
            System.out.println("Waiter: We're out of chicken wings. Please order another barbecue.");
        } else
        {
            orders.add(command);
            System.out.println("Add order:" + command + "  time:" + (new Date()));
        }
    }
```

```
//取消订单
public void CancelOrder(Command command)  0 个用法
{
    orders.remove( o: command);
    System.out.println("cancel order:: " + command.toString() + "  time:" + (new Date()));
}

//通知全部执行
public void Notify()  1 个用法
{
    for (Command cmd : orders)
    {
        cmd.ExecuteCommand();
    }
}
```

定义服务生类，WaiterSetOrder(Command command)用于设置订单。如果订单是烤鸡翅而店里没有鸡翅了，就会提示客户；否则，将命令添加到订单列表中。CancelOrder(Command command)用于取消订单。Notify()通知所有订单执行。

命令模式一般性代码




```
class Invoker  2 个用法
{
    private Command command;  2 个用法

    public void SetCommand(Command command) { this.command = command; }

    public void ExecuteCommand() { command.Execute(); }
}
```

Invoker 类：这个类代表调用者，它持有一个命令对象，并可以执行这个命令。拥有一个设置命令的方法 SetCommand 和一个执行命令的方法 ExecuteCommand

```
class Receiver  5 个用法
{
    public void Action() { System.out.println("The receiver performs the request"); }
}
```

Receiver 类：这个类代表接收者，它执行具体的命令。在这个例子中，它有一个 Action 方法，当调用这个方法时，它会输出一条消息。

```
abstract class Command  4 个用法  1 个继承者
{
    protected Receiver receiver;  2 个用法

    @Contract(pure = true)
    public Command(Receiver receiver) { this.receiver = receiver; }

    abstract public void Execute();  1 个用法  1 个实现
}

class ConcreteCommand extends Command  1 个用法
{
    public ConcreteCommand(Receiver receiver) { super(receiver); }

    public void Execute() { receiver.Action(); }
}
```

Command 抽象类：这个类代表命令的抽象，它定义了一个 Execute 抽象方法，一个受保护的属性 receiver 保存命令执行者

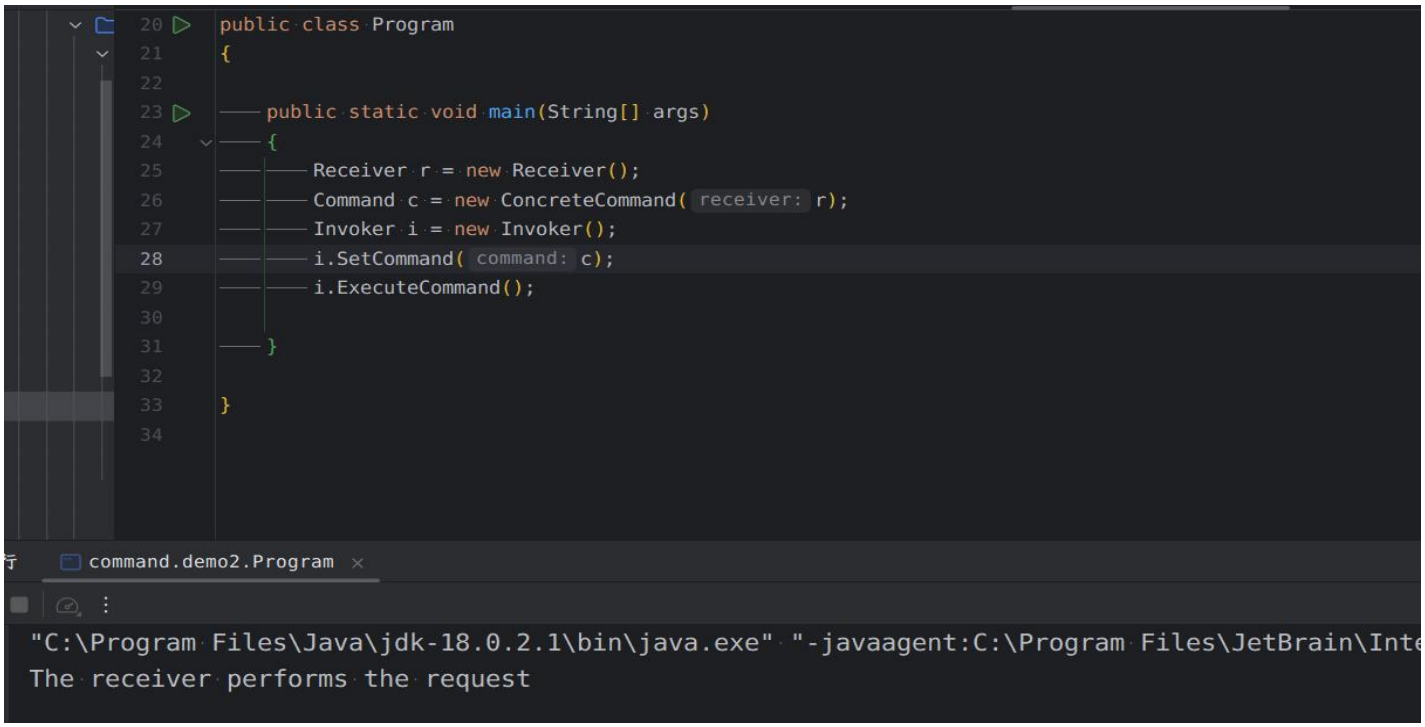
ConcreteCommand 类：作为具体命令类，这个类继承自 Command 抽象类，它实现了 Execute 方法，并调用了接收者的 Action 方法。

四. 调试和运行结果

```
22
23      public static void main(String[] args)
24      {
25
26          //开店前的准备
27          Barbecuer boy = new Barbecuer();
28          Command bakeMuttonCommand1 = new BakeMuttonCommand( receiver: boy);
29          Command bakeMuttonCommand2 = new BakeMuttonCommand( receiver: boy);
30          Command bakeChickenWingCommand1 = new BakeChickenWingCommand( receiver: boy);
31          Waiter girl = new Waiter();
32
33          //开门营业 顾客点菜
34          girl.SetOrder(bakeMuttonCommand1);
35          girl.SetOrder(bakeMuttonCommand2);
36          girl.SetOrder(bakeChickenWingCommand1);
37
38          //点菜完闭，通知厨房
39          girl.Notify();
40
41      }
42
43  }
```

```
command.demo1.Program
"C:\Program Files\Java\jdk-18.0.2.1\bin\java.exe" "-javaagent:C:\Program Files\JetBrain
Add order:BakeMuttonCommand  time:Mon Apr 29 00:44:48 GMT+08:00 2024
Add order:BakeMuttonCommand  time:Mon Apr 29 00:44:48 GMT+08:00 2024
Waiter: We're out of chicken wings. Please order another barbecue.
kebabs of mutton!
kebabs of mutton!
```

创建了一个 Barbecuer 对象（烧烤者）和几个具体的命令对象（这些命令对象包含了指向 Barbecuer 对象的引用），然后创建了一个 Waiter 对象（服务员）。通过调用 Waiter 对象的 SetOrder 方法添加命令，然后调用 Waiter 对象的 Notify 方法执行命令。当添加命令时，Waiter 会根据命令的类型做出不同的反应。最后输出命令执行结果



当 Invoke 类调用 ExecuteCommand()方法时，实际上调用了 ConcreteCommand 的 Execute()方法。而在 ConcreteCommand 的 Execute()方法中，调用了 Receiver 的 Action()方法，表示接收者已经接收到了请求并执行了相应的行为。

命令模式实例总结

案例中顾客（Invoker）通过服务员（Command）传达烹饪指令给厨师（Receiver）。由于点餐请求被封装在服务员这一角色中，顾客无需了解厨房的复杂操作流程，只需关注最终的食物。这种解耦使得系统更加灵活，易于扩展和维护。命令模式的核心优势在于它能够将请求的发送者与请求的接收者解耦。这意味着发送请求的对象不需要知道如何执行该请求，而执行请求的对象也不需要知道请求来自何处。这种解耦提高了系统的灵活性和可维护性，使得两者可以独立地变化而不影响彼此。由于命令对象本身可以被存储和传递，因此可以很容易地实现撤销和重做功能。只需将执行过的命令保存到历史记录中，然后在需要时重新执行或撤销这些命令即可。命令模式可以将命令对象放入队列中，然后在合适的时候执行这些命令，例如支持批量处理和延迟执行。这种能力使得系统能够更灵活地处理用户请求，提高响应速度和效率。