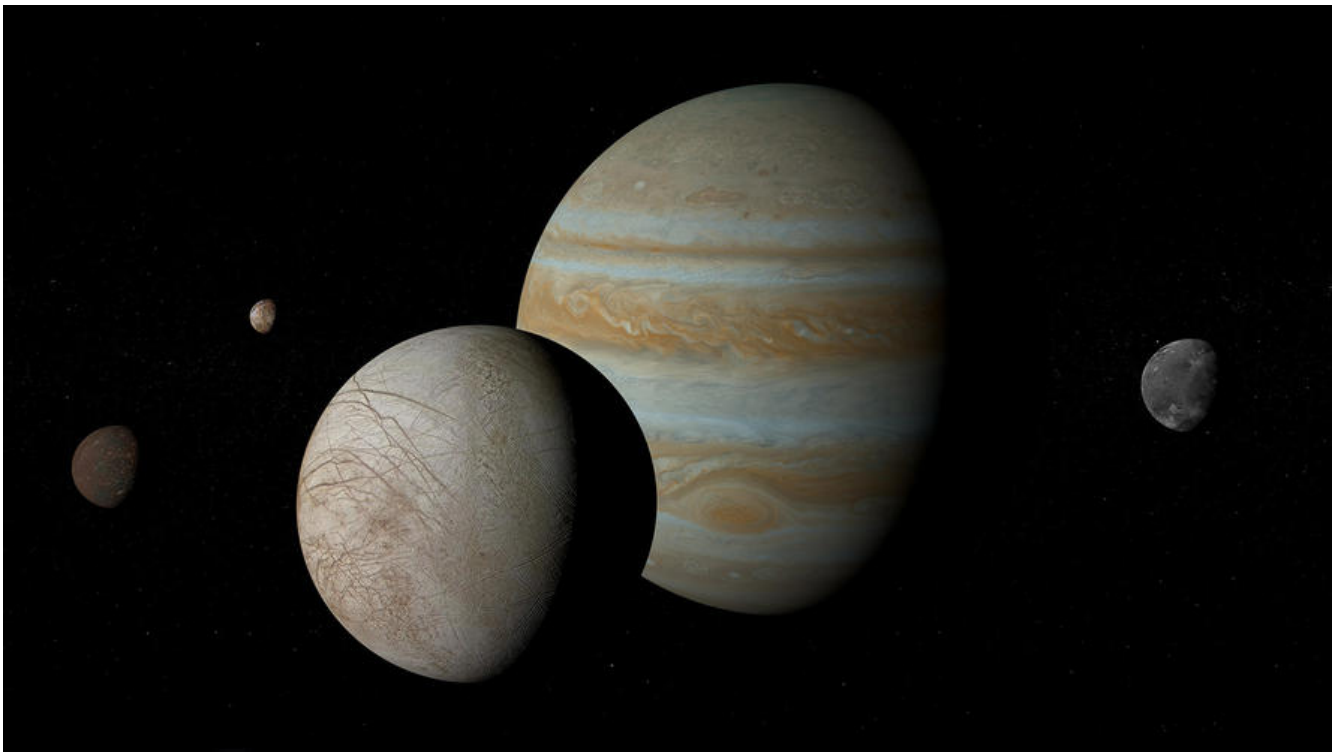
链滴

《Head First 设计模式》：命令模式

作者：jingqueyimu

原文链接：https://ld246.com/article/1597067792723

来源网站：链滴



正文

一、定义

命令模式将“请求”封装成对象（命令对象），以便使用不同的“请求”来参数化其他对象。

要点：

- 命令模式可将“动作的请求者”从“动作的执行者”对象中解耦。
- 被解耦的两者之间通过命令对象进行沟通。命令对象封装了接收者和一个或多个动作。
- 命令对象提供一个 execute() 方法，该方法封装了接收者的动作。当此方法被调用时，接收者就会行这些动作。
- 调用者持有一个或多个命令对象，通过调用命令对象的 execute() 方法发出请求，这会使得接收者动作被调用。

二、实现步骤

1、创建接收者类

接收者类中包含要执行的动作。

（1）接收者A

```
/**
 * 接收者A
 */
public class ReceiverA {
```

```
    /**
     * 动作1
     */
    public void action1() {
        System.out.println("ReceiverA do action1");
    }

    /**
     * 动作2
     */
    public void action2() {
        System.out.println("ReceiverA do action2");
    }
}
```

（2）接收者B

```
/**
 * 接收者B
 */
public class ReceiverB {

    /**
     * 动作1
     */
    public void action1() {
        System.out.println("ReceiverB do action1");
    }

    /**
     * 动作2
     */
    public void action2() {
        System.out.println("ReceiverB do action2");
    }
}
```

2、创建命令接口

```
/**
 * 命令接口
 */
public interface Command {

    /**
     * 执行命令
     */
    public void execute();
}
```

3、创建具体的命令，并实现命令接口

命令对象中，封装了命令接收者和相关动作。

（1）命令A1

```
/**
 * 命令A1
 */
public class ConcreteCommandA1 implements Command{

    /**
     * 接收者A
     */
    ReceiverA receive;

    public ConcreteCommandA1(ReceiverA receive) {
        this.receive = receive;
    }

    @Override
    public void execute() {
        // 接收者A执行动作1
        receive.action1();
    }
}
```

（2）命令A2

```
/**
 * 命令A2
 */
public class ConcreteCommandA2 implements Command{

    /**
     * 接收者A
     */
    ReceiverA receive;

    public ConcreteCommandA2(ReceiverA receive) {
        this.receive = receive;
    }

    @Override
    public void execute() {
        // 接收者A执行动作2
        receive.action2();
    }
}
```

（3）命令B1

```
/**
 * 命令B1
```

```
 */
public class ConcreteCommandB1 implements Command{

    /**
     * 接收者B
     */
    ReceiverB receive;

    public ConcreteCommandB1(ReceiverB receive) {
        this.receive = receive;
    }

    @Override
    public void execute() {
        // 接收者B执行动作1
        receive.action1();
    }
}
```

（4）命令B2

```
/**
 * 命令B2
 */
public class ConcreteCommandB2 implements Command{

    /**
     * 接收者B
     */
    ReceiverB receive;

    public ConcreteCommandB2(ReceiverB receive) {
        this.receive = receive;
    }

    @Override
    public void execute() {
        // 接收者B执行动作2
        receive.action2();
    }
}
```

4、创建调用者

调用者通过持有的命令对象，来调用接收者的动作。

```
/**
 * 调用者
 */
public class Invoker {

    /**
     * 命令对象
```

```
     */
    Command command;

    /**
     * 设置命令
     */
    public void setCommand(Command command) {
        this.command = command;
    }

    /**
     * 调用动作
     */
    public void invoke() {
        command.execute();
    }
}
```

5、调用者执行命令

```
public class Test {

    public static void main(String[] args) {
        // 命令调用者
        Invoker invoker = new Invoker();

        // 命令接收者
        ReceiverA receiverA = new ReceiverA();
        ReceiverB receiverB = new ReceiverB();

        // 创建命令，并指定接收者
        Command commandA1 = new ConcreteCommandA1(receiverA);
        Command commandA2 = new ConcreteCommandA2(receiverA);
        Command commandB1 = new ConcreteCommandB1(receiverB);
        Command commandB2 = new ConcreteCommandB2(receiverB);

        // 调用者执行命令
        invoker.setCommand(commandA1);
        invoker.invoke();
        invoker.setCommand(commandA2);
        invoker.invoke();
        invoker.setCommand(commandB1);
        invoker.invoke();
        invoker.setCommand(commandB2);
        invoker.invoke();
    }
}
```

三、举个栗子

1、背景

假设你要设计一个家电自动化遥控器的 API。这个遥控器具有七个可编程的插槽（每个都可以指定到个不同的家电装置），每个插槽都有对应的“开”、“关”按钮。

你希望每个插槽都能够控制一个或一组装置。并且，这个遥控器适用于目前的装置和任何未来可能出的装置。

2、实现

（1）创建家电类

```
/**
 * 电灯（接收者）
 */
public class Light {

    /**
     * 电灯位置
     */
    String location;

    public Light(String location) {
        this.location = location;
    }

    /**
     * 开灯
     */
    public void on() {
        System.out.println(location + " light is on");
    }

    /**
     * 关灯
     */
    public void off() {
        System.out.println(location + " light is off");
    }
}

/**
 * 音响（接收者）
 */
public class Stereo {

    /**
     * 打开音响
     */
    public void on() {
        System.out.println("Stereo is on");
    }

    /**
     * 关闭音响
```

```
     */
    public void off() {
        System.out.println("Stereo is off");
    }

    /**
     * 设置为播放CD
     */
    public void setCd() {
        System.out.println("Stereo is set for CD input");
    }

    /**
     * 设置为播放DVD
     */
    public void setDvd() {
        System.out.println("Stereo is set for DVD input");
    }

    /**
     * 设置音量
     */
    public void setVolume(int volume) {
        System.out.println("Stereo volume set to " + volume);
    }
}
```

（2）创建命令接口

```
/**
 * 命令接口
 */
public interface Command {

    /**
     * 执行命令
     */
    public void execute();
}
```

（3）创建具体的命令，并继承命令接口

```
/**
 * 无命令
 */
public class NoCommand implements Command {

    @Override
    public void execute() {}
}

/**
 * 开灯命令
```

```
 */
public class LightOnCommand implements Command {

    Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

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

/**
 * 关灯命令
 */
public class LightOffCommand implements Command {

    Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.off();
    }
}

/**
 * 打开音响播放CD命令
 */
public class StereoOnWithCDCommand implements Command {

    Stereo stereo;

    public StereoOnWithCDCommand(Stereo stereo) {
        this.stereo = stereo;
    }

    @Override
    public void execute() {
        stereo.on();
        stereo.setCd();
        stereo.setVolume(11);
    }
}

/**
 * 关闭音响命令
 */
```

```
public class StereoOffCommand implements Command {

    Stereo stereo;

    public StereoOffCommand(Stereo stereo) {
        this.stereo = stereo;
    }

    @Override
    public void execute() {
        stereo.off();
    }
}
```

（4）创建遥控器类

```
/**
 * 遥控器（调用者）
 */
public class RemoteControl {

    Command[] onCommands;
    Command[] offCommands;

    public RemoteControl() {
        onCommands = new Command[7];
        offCommands = new Command[7];
        // 初始化插槽命令为无命令状态
        Command noCommand = new NoCommand();
        for (int i = 0; i < 7; i++) {
            onCommands[i] = noCommand;
            offCommands[i] = noCommand;
        }
    }

    /**
     * 设置插槽命令
     */
    public void setCommand(int slot, Command onCommand, Command offCommand) {
        onCommands[slot] = onCommand;
        offCommands[slot] = offCommand;
    }

    /**
     *  “开”按钮被按下
     */
    public void onButtonWasPushed(int slot) {
        // 执行开命令
        onCommands[slot].execute();
    }

    /**
     *  “关”按钮被按下
     */
```

```
    public void offButtonWasPushed(int slot) {
        // 执行关命令
        offCommands[slot].execute();
    }
}
```

（5）使用遥控器控制家电

```
public class Test {

    public static void main(String[] args) {
        // 遥控器
        RemoteControl remoteControl = new RemoteControl();

        // 家电
        Light livingRoomLight = new Light("Living Room");
        Light kitchenLight = new Light("Kitchen");
        Stereo stereo = new Stereo();

        // 创建命令，并指定家电
        Command livingRoomLightOn = new LightOnCommand(livingRoomLight);
        Command livingRoomLightOff = new LightOffCommand(livingRoomLight);
        Command kitchenLightOn = new LightOnCommand(kitchenLight);
        Command kitchenLightOff = new LightOffCommand(kitchenLight);
        Command stereoOnWithCD = new StereoOnWithCDCommand(stereo);
        Command stereoOff = new StereoOffCommand(stereo);

        // 设置插槽命令
        remoteControl.setCommand(0, livingRoomLightOn, livingRoomLightOff);
        remoteControl.setCommand(1, kitchenLightOn, kitchenLightOff);
        remoteControl.setCommand(2, stereoOnWithCD, stereoOff);

        // 遥控器执行命令
        remoteControl.onButtonWasPushed(0);
        remoteControl.offButtonWasPushed(0);
        remoteControl.onButtonWasPushed(1);
        remoteControl.offButtonWasPushed(1);
        remoteControl.onButtonWasPushed(2);
        remoteControl.offButtonWasPushed(2);
    }
}
```

四、命令模式的更多用途

1、队列请求

命令可以将运算块打包（一个接收者和一组动作），然后将它传来传去，就像是一般的对象一样。即在命令对象被创建许久之后，运算依然可以被调用。事实上，它甚至可以在不同的线程中被调用。

因此，命令模式可以用来实现队列请求。具体做法是：在工作队列的一端添加命令，另一端通过线程出命令，调用它的 execute() 方法，等调用完成后，将此命令对象丢弃，再取出下一个命令……

2、日志请求

某些应用需要我们将所有的动作都记录在日志中，并能在系统死机后，重新调用这些动作恢复到之前状态。

命令模式能够支持这一点。具体做法是：当我们执行命令时，利用序列化将命令对象存储在磁盘中。旦系统死机，再利用反序列化将命令对象重新加载，并依次调用这些对象的 execute() 方法。