课程安排，请关注**微信公众平台**或者**官方微博**

编程语言： Golang 与 html5

编程工具： Goland 和 HBuilder

预计平均一周左右更新一或二节课程

授人以鱼，不如授人以渔业。

大家好，

欢迎来到 **字节教育** 课程的学习

字节教育官网：www.ByteEdu.Com

腾讯课堂地址：Gopher.ke.qq.Com

技术交流群 ： 221 273 219

微信公众号 ：Golang 语言社区

微信服务号 ：Golang 技术社区

目录：

第一季　Golang 语言社区-设计模式

第五节 命令模式

一、公众账号：

回复关键字：客服　　获取课程助教的微信，申请加入课程微信群

二、命令模式讲解：

命令模式是将一个请求封装为一个对象,从而使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,其别名为动作模式或事务模式.

在命令模式中有如下几个角色:
Command: 命令
Invoker: 调用者
Receiver: 接受者
Client: 客户端

他们的关系可以这么来描述:

客户端通过调用者发送命令,命令调用接收者执行相应操作.

其实命令模式也很简单,不过不知道大家发现没有,在上述描述中调用者和接收者并不知道对方的存在,也就是说他们之间是解耦合的.

还是用遥控器的例子来解释一下吧,遥控器对应上面的角色就是调用者,电视就是接收者,命令呢?对应的就是遥控器上的按键,最后客户端就是我们人啦,当我们想打开电视的时候,就会通过遥控器(调用者)的电源按钮(命令)来打开电视(接收者),在这个过程中遥控器是不知道电视的,但是电源按钮是知道他要控制谁的什么操作.

... 哎呀, 语文不太好,突然发现越描述越不容易让人理解了呢? 还是用代码来实现一下上面遥控器的例子吧. 我们对照着上面的角色一个个的去实现. 接收者,也就是一台大大的电视,

```
// receiver

type TV struct{}

func (p TV) Open() {

    fmt.Println("play...")

}

func (p TV) Close() {

    fmt.Println("stop...")

}
```

这台电视弱爆了,只有打开和关闭两个功能,对应的就是上面代码中的 `Open` 和 `Close` 两个方法,虽然很简单,不过我们确实造出了一台电视(估计还是彩色的). 下面,我们再来实现命令,也就是遥控器上的按键,因为电视只有打开和关闭功能,所以按键我们也只提供两个,多了用不上.

```
// command

type Command interface {

    Press()

}
```

```
type OpenCommand struct {

    tv TV

}

func (p OpenCommand) Press() {

    p.tv.Open()

}

type CloseCommand struct {

    tv TV

}

func (p CloseCommand) Press() {

    p.tv.Close()

}
```

首先我们定义了一个命令接口,只有一个方法就是 `Press`,当我们按下按键时会去调用这个方法,然后我们果然只实现了两个按键,分别是 `OpenCommand` 和 `CloseCommand`,这两个实现中都保存着一台电视的句柄,并且在 `Press` 方法中根据功能去调用了这个 tv 的相应方法来完成正确的操作.

还有什么我们没有实现? 调用者,也就是遥控器了,来看看遥控器怎么实现吧.

```
// sender

type Invoker struct {

    cmd Command

}

func (p *Invoker) SetCommand(cmd Command) {

    p.cmd = cmd

}

func (p Invoker) Do() {

    p.cmd.Press()

}
```

在遥控器中我们有一个 `Command` 类型的变量,并且提供了 `SetCommand` 方法来设置命令,那我们按下遥控器上的键对应哪个方法呢?看看 `Do` 方法,我们直接调用了命令的 `Press` 方法,这个时候客户端(就是我们人)拿起遥控器,瞅准了打开按钮(SetCommand),并且按钮该键(Do),按键被按下(Press),电视打开了(Open).

那么最后,来看看客户端怎么去调用吧,

```
func main() {

    var tv TV

    openCommand := OpenCommand{tv}

    invoker := Invoker{openCommand}

    invoker.Do()

    invoker.SetCommand(CloseCommand{tv})

    invoker.Do()

}
```

命令模式实现起来也很简单,它降低了我们这个系统的耦合度,并且我们还可以任意拓展命令使用而不需要修改代码,**开闭原则**体现的淋漓尽致.

现在大家应该对命令模式有了一个直观的认识吧, 不过大家有没有发现一个问题,我们的遥控器在使用某个命令的时候只有 set 后才能用,也就是说每次只能使用一个命令,那有没有更好的办法来预装命令呢?肯定是有的,这就是我们接下来要介绍的**复合命令**,也叫做**宏命令**,复合命令其实说白了就是将多个命令保存起来,通过遍历这个集合来分别调用各个命令.

```
func NewOpenCloseCommand() *OpenCloseCommand {

    var openClose *OpenCloseCommand = &OpenCloseCommand{}

    openClose.cmds = make([]Command, 2)

    return openClose

}

type OpenCloseCommand struct {

    index int

    cmds  []Command

}



func (p *OpenCloseCommand) AddCommand(cmd Command) {
```

```
    p.cmds[p.index] = cmd

    p.index++

}

func (p OpenCloseCommand) Press() {

    for _, item := range p.cmds {

        item.Press()

    }

}

func main() {

  var openClose *OpenCloseCommand = NewOpenCloseCommand()

  openClose.AddCommand(OpenCommand{tv})

  openClose.AddCommand(CloseCommand{tv})

  invoker.SetCommand(openClose)

  invoker.Do()

}
```

我们定义了一个 OpenCloseCommand,这里面用一个切片来保存各个命令,并通过 AddCommand 方法来王里面添加命令,OpenCloseCommand 实现了 Press 方法,所以本质上他也是一个命令,我们调用他的 Press 方法来遍历保存的 Command,并调用每一个的 Press 方法.

到这里,命令模式我们就介绍完了,命令模式具有很高的扩展性,遵循了开闭原则,所以减少了修改代码引入 bug 的可能性,快来想一想你的代码中哪些地方可以用命令模式来解决吧.