快收藏！最全GO语言实现设计模式

陈佼 腾讯云开发者 2022-11-21 19:22 发表于广东

收录于合集
#腾讯技术人原创集 32 #Go 32

点个关注👆跟腾讯工程师学技术

导语| 设计模式是针对软件设计中常见问题的工具箱，其中的工具就是各种经过实践验证的解决方案。即使你从未遇到过这些问题，了解模式仍然非常有用，因为它能指导你如何使用面向对象的设计原则来解决各种问题，提高开发效率，降低开发成本；本文囊括了GO语言实现的经典设计模式示例，每个示例都精心设计，力求符合模式结构，可作为日常编码参考，同时一些常用的设计模式融入了开发实践经验总结，帮助大家在平时工作中灵活运用。

责任链模式

（一）概念

责任链模式是一种行为设计模式， 允许你将请求沿着处理者链进行发送。收到请求后，每个处理者均可对请求进行处理，或将其传递给链上的下个处理者。

该模式允许多个对象来对请求进行处理，而无需让发送者类与具体接收者类相耦合。链可在运行时由遵循标准处理者接口的任意处理者动态生成。
一般意义上的责任链模式是说，请求在链上流转时任何一个满足条件的节点处理完请求后就会停止流转并返回，不过还可以根据不同的业务情况做一些改进：

- 请求可以流经处理链的所有节点，不同节点会对请求做不同职责的处理；
- 可以通过上下文参数保存请求对象及上游节点的处理结果，供下游节点依赖，并进一步处理；

- 处理链可支持节点的异步处理，通过实现特定接口判断，是否需要异步处理；

- 责任链对于请求处理节点可以设置停止标志位，不是异常，是一种满足业务流转的中断；

- 责任链的拼接方式存在两种，一种是节点遍历，一个节点一个节点顺序执行；另一种是节点嵌套，内层节点嵌入在外层节点执行逻辑中，类似递归，或者“回”行结构；

- 责任链的节点嵌套拼接方式多被称为拦截器链或者过滤器链，更易于实现业务流程的切面，比如监控业务执行时长，日志输出，权限校验等；

（二）示例

本示例模拟实现机场登机过程，第一步办理登机牌，第二步如果有行李，就办理托运，第三步核实身份，第四步安全检查，第五步完成登机；其中行李托运是可选的，其他步骤必选，必选步骤有任何不满足就终止登机；旅客对象作为请求参数上下文，每个步骤会根据旅客对象状态判断是否处理或流转下一个节点；

（三）登机过程

```go
package chainofresponsibility

import "fmt"

// BoardingProcessor 登机过程中，各节点统一处理接口
type BoardingProcessor interface {
  SetNextProcessor(processor BoardingProcessor)
  ProcessFor(passenger *Passenger)
}

// Passenger 旅客
type Passenger struct {
  name                string // 姓名
  hasBoardingPass     bool   // 是否办理登机牌
```

```
  hasLuggage            bool   // 是否有行李需要托运
  isPassIdentityCheck   bool   // 是否通过身份校验
  isPassSecurityCheck   bool   // 是否通过安检
  isCompleteForBoarding bool   // 是否完成登机
}

// baseBoardingProcessor 登机流程处理器基类
type baseBoardingProcessor struct {
  // nextProcessor 下一个登机处理流程
  nextProcessor BoardingProcessor
}

// SetNextProcessor 基类中统一实现设置下一个处理器方法
func (b *baseBoardingProcessor) SetNextProcessor(processor BoardingProcessor
  b.nextProcessor = processor
}

// ProcessFor 基类中统一实现下一个处理器流转
func (b *baseBoardingProcessor) ProcessFor(passenger *Passenger) {
  if b.nextProcessor != nil {
    b.nextProcessor.ProcessFor(passenger)
  }
}

// boardingPassProcessor 办理登机牌处理器
type boardingPassProcessor struct {
  baseBoardingProcessor // 引用基类
}

func (b *boardingPassProcessor) ProcessFor(passenger *Passenger) {
  if !passenger.hasBoardingPass {
    fmt.Printf("为旅客%s办理登机牌;\n", passenger.name)
    passenger.hasBoardingPass = true
  }
  // 成功办理登机牌后，进入下一个流程处理
  b.baseBoardingProcessor.ProcessFor(passenger)
}

// luggageCheckInProcessor 托运行李处理器
type luggageCheckInProcessor struct {
```

```
  baseBoardingProcessor
}

func (l *luggageCheckInProcessor) ProcessFor(passenger *Passenger) {
  if !passenger.hasBoardingPass {
    fmt.Printf("旅客%s未办理登机牌，不能托运行李;\n", passenger.name)
    return
  }
  if passenger.hasLuggage {
    fmt.Printf("为旅客%s办理行李托运;\n", passenger.name)
  }
  l.baseBoardingProcessor.ProcessFor(passenger)
}

// identityCheckProcessor 校验身份处理器
type identityCheckProcessor struct {
  baseBoardingProcessor
}

func (i *identityCheckProcessor) ProcessFor(passenger *Passenger) {
  if !passenger.hasBoardingPass {
    fmt.Printf("旅客%s未办理登机牌，不能办理身份校验;\n", passenger.name)
    return
  }
  if !passenger.isPassIdentityCheck {
    fmt.Printf("为旅客%s核实身份信息;\n", passenger.name)
    passenger.isPassIdentityCheck = true
  }
  i.baseBoardingProcessor.ProcessFor(passenger)
}

// securityCheckProcessor 安检处理器
type securityCheckProcessor struct {
  baseBoardingProcessor
}

func (s *securityCheckProcessor) ProcessFor(passenger *Passenger) {
  if !passenger.hasBoardingPass {
    fmt.Printf("旅客%s未办理登机牌，不能进行安检;\n", passenger.name)
    return
```

```
  }
  if !passenger.isPassSecurityCheck {
    fmt.Printf("为旅客%s进行安检;\n", passenger.name)
    passenger.isPassSecurityCheck = true
  }
  s.baseBoardingProcessor.ProcessFor(passenger)
}

// completeBoardingProcessor 完成登机处理器
type completeBoardingProcessor struct {
  baseBoardingProcessor
}

func (c *completeBoardingProcessor) ProcessFor(passenger *Passenger) {
  if !passenger.hasBoardingPass ||
    !passenger.isPassIdentityCheck ||
    !passenger.isPassSecurityCheck {
    fmt.Printf("旅客%s登机检查过程未完成，不能登机;\n", passenger.name)
    return
  }
  passenger.isCompleteForBoarding = true
  fmt.Printf("旅客%s成功登机;\n", passenger.name)
}
```

（四）测试程序

```
package chainofresponsibility

import "testing"

func TestChainOfResponsibility(t *testing.T) {
  boardingProcessor := BuildBoardingProcessorChain()
  passenger := &Passenger{
    name:                "李四",
    hasBoardingPass:     false,
    hasLuggage:          true,
```

```
    isPassIdentityCheck:   false,
    isPassSecurityCheck:   false,
    isCompleteForBoarding: false,
  }
  boardingProcessor.ProcessFor(passenger)
}

// BuildBoardingProcessorChain 构建登机流程处理链
func BuildBoardingProcessorChain() BoardingProcessor {
  completeBoardingNode := &completeBoardingProcessor{}

  securityCheckNode := &securityCheckProcessor{}
  securityCheckNode.SetNextProcessor(completeBoardingNode)

  identityCheckNode := &identityCheckProcessor{}
  identityCheckNode.SetNextProcessor(securityCheckNode)

  luggageCheckInNode := &luggageCheckInProcessor{}
  luggageCheckInNode.SetNextProcessor(identityCheckNode)

  boardingPassNode := &boardingPassProcessor{}
  boardingPassNode.SetNextProcessor(luggageCheckInNode)
  return boardingPassNode
}
```

（五）运行结果

```
=== RUN   TestChainOfResponsibility
为旅客李四办理登机牌;
为旅客李四办理行李托运;
为旅客李四核实身份信息;
为旅客李四进行安检;
旅客李四成功登机;
--- PASS: TestChainOfResponsibility (0.00s)
PASS
```

命令模式

（一）概念

命令模式是一种行为设计模式，它可将请求转换为一个包含与请求相关的所有信息的独立对象。该转换让你能根据不同的请求将方法参数化、延迟请求执行或将其放入队列中，且能实现可撤销操作。

方法参数化是指将每个请求参数传入具体命令的工厂方法（go语言没有构造函数）创建命令，同时具体命令会默认设置好接受对象，这样做的好处是不管请求参数个数及类型，还是接受对象有几个，都会被封装到具体命令对象的成员字段上，并通过统一的Execute接口方法进行调用，屏蔽各个请求的差异，便于命令扩展，多命令组装，回滚等；

（二）示例

控制电饭煲做饭是一个典型的命令模式的场景，电饭煲的控制面板会提供设置煮粥、蒸饭模式，及开始和停止按钮，电饭煲控制系统会根据模式的不同设置相应的火力，压强及时间等参数；煮粥，蒸饭就相当于不同的命令，开始按钮就相当命令触发器，设置好做饭模式，点击开始按钮电饭煲就开始运行，同时还支持停止命令；

（三）电饭煲接收器

```
package command

import "fmt"

// ElectricCooker 电饭煲
type ElectricCooker struct {
  fire     string // 火力
  pressure string // 压力
}

```

```
// SetFire 设置火力
func (e *ElectricCooker) SetFire(fire string) {
  e.fire = fire
}

// SetPressure 设置压力
func (e *ElectricCooker) SetPressure(pressure string) {
  e.pressure = pressure
}

// Run 持续运行指定时间
func (e *ElectricCooker) Run(duration string) string {
  return fmt.Sprintf("电饭煲设置火力为%s,压力为%s,持续运行%s;", e.fire, e.pressu
}

// Shutdown 停止
func (e *ElectricCooker) Shutdown() string {
  return "电饭煲停止运行。"
}
```

（四）电饭煲命令

```
package command

// CookCommand 做饭指令接口
type CookCommand interface {
  Execute() string // 指令执行方法
}

// steamRiceCommand 蒸饭指令
type steamRiceCommand struct {
  electricCooker *ElectricCooker // 电饭煲
}

func NewSteamRiceCommand(electricCooker *ElectricCooker) *steamRiceCommand {
  return &steamRiceCommand{
```

```
    electricCooker: electricCooker,
  }
}

func (s *steamRiceCommand) Execute() string {
  s.electricCooker.SetFire("中")
  s.electricCooker.SetPressure("正常")
  return "蒸饭:" + s.electricCooker.Run("30分钟")
}

// cookCongeeCommand 煮粥指令
type cookCongeeCommand struct {
  electricCooker *ElectricCooker
}

func NewCookCongeeCommand(electricCooker *ElectricCooker) *cookCongeeCommand
  return &cookCongeeCommand{
    electricCooker: electricCooker,
  }
}

func (c *cookCongeeCommand) Execute() string {
  c.electricCooker.SetFire("大")
  c.electricCooker.SetPressure("强")
  return "煮粥:" + c.electricCooker.Run("45分钟")
}

// shutdownCommand 停止指令
type shutdownCommand struct {
  electricCooker *ElectricCooker
}

func NewShutdownCommand(electricCooker *ElectricCooker) *shutdownCommand {
  return &shutdownCommand{
    electricCooker: electricCooker,
  }
}

func (s *shutdownCommand) Execute() string {
  return s.electricCooker.Shutdown()
```

```go
}

// ElectricCookerInvoker 电饭煲指令触发器
type ElectricCookerInvoker struct {
  cookCommand CookCommand
}

// SetCookCommand 设置指令
func (e *ElectricCookerInvoker) SetCookCommand(cookCommand CookCommand) {
  e.cookCommand = cookCommand
}

// ExecuteCookCommand 执行指令
func (e *ElectricCookerInvoker) ExecuteCookCommand() string {
  return e.cookCommand.Execute()
}
```

（五）测试程序

```go
package command

import (
  "fmt"
  "testing"
)

func TestCommand(t *testing.T) {
  // 创建电饭煲，命令接受者
  electricCooker := new(ElectricCooker)
  // 创建电饭煲指令触发器
  electricCookerInvoker := new(ElectricCookerInvoker)

  // 蒸饭
  steamRiceCommand := NewSteamRiceCommand(electricCooker)
  electricCookerInvoker.SetCookCommand(steamRiceCommand)
  fmt.Println(electricCookerInvoker.ExecuteCookCommand())
```

```
18
19    // 煮粥
20    cookCongeeCommand := NewCookCongeeCommand(electricCooker)
21    electricCookerInvoker.SetCookCommand(cookCongeeCommand)
22    fmt.Println(electricCookerInvoker.ExecuteCookCommand())
23
24    // 停止
25    shutdownCommand := NewShutdownCommand(electricCooker)
26    electricCookerInvoker.SetCookCommand(shutdownCommand)
27    fmt.Println(electricCookerInvoker.ExecuteCookCommand())
28 }
```

（六）运行结果

```
1 === RUN   TestCommand
2 蒸饭:电饭煲设置火力为中,压力为正常,持续运行30分钟;
3 煮粥:电饭煲设置火力为大,压力为强,持续运行45分钟;
4 电饭煲停止运行。
5 --- PASS: TestCommand (0.00s)
6 PASS
```

迭代器模式

（一）概念

迭代器模式是一种行为设计模式，让你能在不暴露集合底层表现形式 （列表、 栈和树等）的情况下遍历集合中所有的元素。

在迭代器的帮助下， 客户端可以用一个迭代器接口以相似的方式遍历不同集合中的元素。

这里需要注意的是有两个典型的迭代器接口需要分清楚；一个是集合类实现的可以创建迭代器的工厂方法接口一般命名为Iterable，包含的方法类似CreateIterator；另一个是迭代器本身的接口，命名为Iterator，有Next及hasMore两个主要方法；

（二）示例

一个班级类中包括一个老师和若干个学生，我们要对班级所有成员进行遍历，班级中老师存储在单独的结构字段中，学生存储在另外一个slice字段中，通过迭代器，我们实现统一遍历处理；

（三）班级成员

```go
package iterator

import "fmt"

// Member 成员接口
type Member interface {
  Desc() string // 输出成员描述信息
}

// Teacher 老师
type Teacher struct {
  name    string // 名称
  subject string // 所教课程
}

// NewTeacher 根据姓名、课程创建老师对象
func NewTeacher(name, subject string) *Teacher {
  return &Teacher{
    name:    name,
    subject: subject,
  }
}

func (t *Teacher) Desc() string {
  return fmt.Sprintf("%s班主任老师负责教%s", t.name, t.subject)
```

```
}

// Student 学生
type Student struct {
  name     string // 姓名
  sumScore int    // 考试总分数
}

// NewStudent 创建学生对象
func NewStudent(name string, sumScore int) *Student {
  return &Student{
    name:     name,
    sumScore: sumScore,
  }
}

func (t *Student) Desc() string {
  return fmt.Sprintf("%s同学考试总分为%d", t.name, t.sumScore)
}
```

（四）班级成员迭代器

```
package iterator

// Iterator 迭代器接口
type Iterator interface {
  Next() Member  // 迭代下一个成员
  HasMore() bool // 是否还有
}

// memberIterator 班级成员迭代器实现
type memberIterator struct {
  class *Class // 需迭代的班级
  index int    // 迭代索引
}

```

```go
func (m *memberIterator) Next() Member {
  // 迭代索引为-1时，返回老师成员，否则遍历学生slice
  if m.index == -1 {
    m.index++
    return m.class.teacher
  }
  student := m.class.students[m.index]
  m.index++
  return student
}

func (m *memberIterator) HasMore() bool {
  return m.index < len(m.class.students)
}

// Iterable 可迭代集合接口，实现此接口返回迭代器
type Iterable interface {
  CreateIterator() Iterator
}

// Class 班级，包括老师和同学
type Class struct {
  name     string
  teacher  *Teacher
  students []*Student
}

// NewClass 根据班主任老师名称，授课创建班级
func NewClass(name, teacherName, teacherSubject string) *Class {
  return &Class{
    name:    name,
    teacher: NewTeacher(teacherName, teacherSubject),
  }
}

// CreateIterator 创建班级迭代器
func (c *Class) CreateIterator() Iterator {
  return &memberIterator{
    class: c,
    index: -1, // 迭代索引初始化为-1，从老师开始迭代
```

```
  }
}

func (c *Class) Name() string {
  return c.name
}

// AddStudent 班级添加同学
func (c *Class) AddStudent(students ...*Student) {
  c.students = append(c.students, students...)
}
```

（五）测试程序

```
package iterator

import (
  "fmt"
  "testing"
)

func TestIterator(t *testing.T) {
  class := NewClass("三年级一班", "王明", "数学课")
  class.AddStudent(NewStudent("张三", 389),
    NewStudent("李四", 378),
    NewStudent("王五", 347))

  fmt.Printf("%s成员如下:\n", class.Name())
  classIterator := class.CreateIterator()
  for classIterator.HasMore() {
    member := classIterator.Next()
    fmt.Println(member.Desc())
  }
}
```

（六）运行结果

```
=== RUN   TestIterator
三年级一班成员如下:
王明班主任老师负责教数学课
张三同学考试总分为389
李四同学考试总分为378
王五同学考试总分为347
--- PASS: TestIterator (0.00s)
PASS
```

中介者模式

（一）概念

中介者模式是一种行为设计模式，能让你减少对象之间混乱无序的依赖关系。该模式会限制对象之间的直接交互，迫使它们通过一个中介者对象进行合作，将网状依赖变为星状依赖。

中介者能使得程序更易于修改和扩展，而且能更方便地对独立的组件进行复用，因为它们不再依赖于很多其他的类。

中介者模式与观察者模式之间的区别是，中介者模式解决的是同类或者不同类的多个对象之间多对多的依赖关系，观察者模式解决的是多个对象与一个对象之间的多对一的依赖关系。

（二）示例

机场塔台调度系统是一个体现中介者模式的典型示例，假设是一个小机场，每次只能同时允许一架飞机起降，每架靠近机场的飞机需要先与塔台沟通是否可以降落，如果没有空闲的跑道，需要在天空盘旋等待，如果有飞机离港，等待的飞机会收到塔台的通知，按先后

顺序降落；这种方式，免去多架飞机同时到达机场需要相互沟通降落顺序的复杂性，减少多个飞机间的依赖关系，简化业务逻辑，从而降低系统出问题的风险。

（三）飞机对象

```go
package mediator

import "fmt"

// Aircraft 飞机接口
type Aircraft interface {
  ApproachAirport() // 抵达机场空域
  DepartAirport()   // 飞离机场
}

// airliner 客机
type airliner struct {
  name            string          // 客机型号
  airportMediator AirportMediator // 机场调度
}

// NewAirliner 根据指定型号及机场调度创建客机
func NewAirliner(name string, mediator AirportMediator) *airliner {
  return &airliner{
    name:            name,
    airportMediator: mediator,
  }
}

func (a *airliner) ApproachAirport() {
  if !a.airportMediator.CanLandAirport(a) { // 请求塔台是否可以降落
    fmt.Printf("机场繁忙，客机%s继续等待降落;\n", a.name)
    return
  }
  fmt.Printf("客机%s成功滑翔降落机场;\n", a.name)
}

func (a *airliner) DepartAirport() {
```

```
  fmt.Printf("客机%s成功滑翔起飞，离开机场;\n", a.name)
  a.airportMediator.NotifyWaitingAircraft() // 通知等待的其他飞机
}

// helicopter 直升机
type helicopter struct {
  name            string
  airportMediator AirportMediator
}

// NewHelicopter 根据指定型号及机场调度创建直升机
func NewHelicopter(name string, mediator AirportMediator) *helicopter {
  return &helicopter{
    name:            name,
    airportMediator: mediator,
  }
}

func (h *helicopter) ApproachAirport() {
  if !h.airportMediator.CanLandAirport(h) { // 请求塔台是否可以降落
    fmt.Printf("机场繁忙，直升机%s继续等待降落;\n", h.name)
    return
  }
  fmt.Printf("直升机%s成功垂直降落机场;\n", h.name)
}

func (h *helicopter) DepartAirport() {
  fmt.Printf("直升机%s成功垂直起飞，离开机场;\n", h.name)
  h.airportMediator.NotifyWaitingAircraft() // 通知其他等待降落的飞机
}
```

（四）机场塔台

```
package mediator

// AirportMediator 机场调度中介者
type AirportMediator interface {
```

```
    CanLandAirport(aircraft Aircraft) bool // 确认是否可以降落
    NotifyWaitingAircraft()                // 通知等待降落的其他飞机
}

// ApproachTower 机场塔台
type ApproachTower struct {
    hasFreeAirstrip bool
    waitingQueue    []Aircraft // 等待降落的飞机队列
}

func (a *ApproachTower) CanLandAirport(aircraft Aircraft) bool {
    if a.hasFreeAirstrip {
        a.hasFreeAirstrip = false
        return true
    }
    // 没有空余的跑道，加入等待队列
    a.waitingQueue = append(a.waitingQueue, aircraft)
    return false
}

func (a *ApproachTower) NotifyWaitingAircraft() {
    if !a.hasFreeAirstrip {
        a.hasFreeAirstrip = true
    }
    if len(a.waitingQueue) > 0 {
        // 如果存在等待降落的飞机，通知第一个降落
        first := a.waitingQueue[0]
        a.waitingQueue = a.waitingQueue[1:]
        first.ApproachAirport()
    }
}
```

（五）测试程序

```
package mediator

import "testing"
```

```go

func TestMediator(t *testing.T) {
    // 创建机场调度塔台
    airportMediator := &ApproachTower{hasFreeAirstrip: true}
    // 创建C919客机
    c919Airliner := NewAirliner("C919", airportMediator)
    // 创建米-26重型运输直升机
    m26Helicopter := NewHelicopter("米-26", airportMediator)

    c919Airliner.ApproachAirport()  // c919进港降落
    m26Helicopter.ApproachAirport() // 米-26进港等待

    c919Airliner.DepartAirport()  // c919飞离，等待的米-26进港降落
    m26Helicopter.DepartAirport() // 最后米-26飞离
}
```

（六）运行结果

```
=== RUN   TestMediator
客机C919成功滑翔降落机场;
机场繁忙，直升机米-26继续等待降落;
客机C919成功滑翔起飞，离开机场;
直升机米-26成功垂直降落机场;
直升机米-26成功垂直起飞，离开机场;
--- PASS: TestMediator (0.00s)
PASS
```

备忘录模式

（一）概念

备忘录模式是一种行为设计模式， 允许在不暴露对象实现细节的情况下保存和恢复对象之前的状态。

备忘录不会影响它所处理的对象的内部结构， 也不会影响快照中保存的数据。

一般情况由原发对象保存生成的备忘录对象的状态不能被除原发对象之外的对象访问，所以通过内部类定义具体的备忘录对象是比较安全的，但是go语言不支持内部类定义的方式，因此go语言实现备忘录对象时，首先将备忘录保存的状态设为非导出字段，避免外部对象访问，其次将原发对象的引用保存到备忘录对象中，当通过备忘录对象恢复时，直接操作备忘录的恢复方法，将备份数据状态设置到原发对象中，完成恢复。

（二）示例

大家平时玩的角色扮演闯关游戏的存档机制就可以通过备忘录模式实现，每到一个关键关卡，玩家经常会先保存游戏存档，用于闯关失败后重置，存档会把角色状态及场景状态保存到备忘录中，同时将需要恢复游戏的引用存入备忘录，用于关卡重置；

（三）闯关游戏

```
package memento

import "fmt"

// Originator 备忘录模式原发器接口
type Originator interface {
  Save(tag string) Memento // 当前状态保存备忘录
}

// RolesPlayGame 支持存档的RPG游戏
type RolesPlayGame struct {
  name          string   // 游戏名称
  rolesState    []string // 游戏角色状态
  scenarioState string   // 游戏场景状态
}

// NewRolesPlayGame 根据游戏名称和角色名，创建RPG游戏
```

```go
func NewRolesPlayGame(name string, roleName string) *RolesPlayGame {
  return &RolesPlayGame{
    name:          name,
    rolesState:    []string{roleName, "血量100"}, // 默认满血
    scenarioState: "开始通过第一关",                    // 默认第一关开始
  }
}

// Save 保存RPG游戏角色状态及场景状态到指定标签归档
func (r *RolesPlayGame) Save(tag string) Memento {
  return newRPGArchive(tag, r.rolesState, r.scenarioState, r)
}

func (r *RolesPlayGame) SetRolesState(rolesState []string) {
  r.rolesState = rolesState
}

func (r *RolesPlayGame) SetScenarioState(scenarioState string) {
  r.scenarioState = scenarioState
}

// String 输出RPG游戏简要信息
func (r *RolesPlayGame) String() string {
  return fmt.Sprintf("在%s游戏中，玩家使用%s,%s,%s;", r.name, r.rolesState[0],
}
```

（四）游戏存档

```go
package memento

import "fmt"

// Memento 备忘录接口
type Memento interface {
  Tag() string // 备忘录标签
  Restore()    // 根据备忘录存储数据状态恢复原对象
```

```
}

// rpgArchive rpg游戏存档，
type rpgArchive struct {
  tag           string         // 存档标签
  rolesState    []string       // 存档的角色状态
  scenarioState string         // 存档游戏场景状态
  rpg           *RolesPlayGame // rpg游戏引用
}

// newRPGArchive 根据标签，角色状态，场景状态，rpg游戏引用，创建游戏归档备忘录
func newRPGArchive(tag string, rolesState []string, scenarioState string, rpg
  return &rpgArchive{
    tag:           tag,
    rolesState:    rolesState,
    scenarioState: scenarioState,
    rpg:           rpg,
  }
}

func (r *rpgArchive) Tag() string {
  return r.tag
}

// Restore 根据归档数据恢复游戏状态
func (r *rpgArchive) Restore() {
  r.rpg.SetRolesState(r.rolesState)
  r.rpg.SetScenarioState(r.scenarioState)
}

// RPGArchiveManager RPG游戏归档管理器
type RPGArchiveManager struct {
  archives map[string]Memento // 存储归档标签对应归档
}

func NewRPGArchiveManager() *RPGArchiveManager {
  return &RPGArchiveManager{
    archives: make(map[string]Memento),
  }
}
```

```
49
50 // Reload 根据标签重新加载归档数据
51 func (r *RPGArchiveManager) Reload(tag string) {
52   if archive, ok := r.archives[tag]; ok {
53     fmt.Printf("重新加载%s;\n", tag)
54     archive.Restore()
55   }
56 }
57
58 // Put 保存归档数据
59 func (r *RPGArchiveManager) Put(memento Memento) {
60   r.archives[memento.Tag()] = memento
61 }
```

（五）测试程序

```
1  package memento
2
3  import (
4    "fmt"
5    "testing"
6  )
7
8  func TestMemento(t *testing.T) {
9    // 创建RPG游戏存档管理器
10   rpgManager := NewRPGArchiveManager()
11   // 创建RPG游戏
12   rpg := NewRolesPlayGame("暗黑破坏神2", "野蛮人战士")
13   fmt.Println(rpg)                    // 输出游戏当前状态
14   rpgManager.Put(rpg.Save("第一关存档")) // 游戏存档
15
16   // 第一关闯关失败
17   rpg.SetRolesState([]string{"野蛮人战士", "死亡"})
18   rpg.SetScenarioState("第一关闯关失败")
19   fmt.Println(rpg)
20
```

```
    // 恢复存档，重新闯关
    rpgManager.Reload("第一关存档")
    fmt.Println(rpg)
}
```

（六）运行结果

```
=== RUN   TestMemento
在暗黑破坏神2游戏中，玩家使用野蛮人战士,血量100,开始通过第一关;
在暗黑破坏神2游戏中，玩家使用野蛮人战士,死亡,第一关闯关失败;
重新加载第一关存档;
在暗黑破坏神2游戏中，玩家使用野蛮人战士,血量100,开始通过第一关;
--- PASS: TestMemento (0.00s)
PASS
```

观察者模式

（一）概念

观察者模式是一种行为设计模式，允许你定义一种订阅机制，可在对象事件发生时通知多个“观察”该对象的其他对象。

观察者模式提供了一种作用于任何实现了订阅者接口的对象的机制，可对其事件进行订阅和取消订阅。

观察者模式是最常用的模式之一，是事件总线，分布式消息中间件等各种事件机制的原始理论基础，常用于解耦多对一的对象依赖关系；

增强的实现功能包括：

- 当被观察者通过异步实现通知多个观察者时就相当于单进程实例的消息总线；

- 同时还可以根据业务需要，将被观察者所有数据状态变更进行分类为不同的主题，观察者通过不同主题进行订阅；

- 同一个主题又可分为增加，删除，修改事件行为；

- 每个主题可以实现一个线程池，多个主题通过不同的线程池进行处理隔离，线程池可以设置并发线程大小、缓冲区大小及调度策略，比如先进先出，优先级等策略；

- 观察者处理事件时有可能出现异常，所以也可以注册异常处理函数，异常处理也可以通过异常类型进行分类；

- 根据业务需求也可以实现通知异常重试，延迟通知等功能；

（二）示例

信用卡业务消息提醒可通过观察者模式实现，业务消息包括日常消费，出账单，账单逾期，消息提醒包括短信、邮件及电话，根据不同业务的场景会采用不同的消息提醒方式或者多种消息提醒方式，这里信用卡相当于被观察者，观察者相当于不同的通知方式；日常消费通过短信通知，出账单通过邮件通知，账单逾期三种方式都会进行通知；

（三）通知方式

```
package observer

import "fmt"

// Subscriber 订阅者接口
type Subscriber interface {
  Name() string          //订阅者名称
  Update(message string) //订阅更新方法
}

```

```go
// shortMessage 信用卡消息短信订阅者
type shortMessage struct{}

func (s *shortMessage) Name() string {
  return "手机短息"
}

func (s *shortMessage) Update(message string) {
  fmt.Printf("通过【%s】发送消息:%s\n", s.Name(), message)
}

// email 信用卡消息邮箱订阅者
type email struct{}

func (e *email) Name() string {
  return "电子邮件"
}

func (e *email) Update(message string) {
  fmt.Printf("通过【%s】发送消息:%s\n", e.Name(), message)
}

// telephone 信用卡消息电话订阅者
type telephone struct{}

func (t *telephone) Name() string {
  return "电话"
}

func (t *telephone) Update(message string) {
  fmt.Printf("通过【%s】告知:%s\n", t.Name(), message)
}
```

（四）信用卡业务

```go
package observer
```

```
2
3  import "fmt"
4
5  // MsgType 信用卡消息类型
6  type MsgType int
7
8  const (
9    ConsumeType MsgType = iota // 消费消息类型
10   BillType                   // 账单消息类型
11   ExpireType                 // 逾期消息类型
12 )
13
14 // CreditCard 信用卡
15 type CreditCard struct {
16   holder          string                    // 持卡人
17   consumeSum      float32                   // 消费总金额
18   subscriberGroup map[MsgType][]Subscriber // 根据消息类型分组订阅者
19 }
20
21 // NewCreditCard 指定持卡人创建信用卡
22 func NewCreditCard(holder string) *CreditCard {
23   return &CreditCard{
24     holder:          holder,
25     subscriberGroup: make(map[MsgType][]Subscriber),
26   }
27 }
28
29 // Subscribe 支持订阅多种消息类型
30 func (c *CreditCard) Subscribe(subscriber Subscriber, msgTypes ...MsgType)
31   for _, msgType := range msgTypes {
32     c.subscriberGroup[msgType] = append(c.subscriberGroup[msgType], subscr
33   }
34 }
35
36 // Unsubscribe 解除订阅多种消息类型
37 func (c *CreditCard) Unsubscribe(subscriber Subscriber, msgTypes ...MsgType
38   for _, msgType := range msgTypes {
39     if subs, ok := c.subscriberGroup[msgType]; ok {
40       c.subscriberGroup[msgType] = removeSubscriber(subs, subscriber)
41     }
```

```
  }
}

func removeSubscriber(subscribers []Subscriber, toRemove Subscriber) []Subs
  length := len(subscribers)
  for i, subscriber := range subscribers {
    if toRemove.Name() == subscriber.Name() {
      subscribers[length-1], subscribers[i] = subscribers[i], subscribers[
      return subscribers[:length-1]
    }
  }
  return subscribers
}

// Consume 信用卡消费
func (c *CreditCard) Consume(money float32) {
  c.consumeSum += money
  c.notify(ConsumeType, fmt.Sprintf("尊敬的持卡人%s,您当前消费%.2f元;", c.hol
}

// SendBill 发送信用卡账单
func (c *CreditCard) SendBill() {
  c.notify(BillType, fmt.Sprintf("尊敬的持卡人%s,您本月账单已出，消费总额%.2f元
}

// Expire 逾期通知
func (c *CreditCard) Expire() {
  c.notify(ExpireType, fmt.Sprintf("尊敬的持卡人%s,您本月账单已逾期，请及时还款
}

// notify 根据消息类型通知订阅者
func (c *CreditCard) notify(msgType MsgType, message string) {
  if subs, ok := c.subscriberGroup[msgType]; ok {
    for _, sub := range subs {
      sub.Update(message)
    }
  }
}
```

（五）测试程序

```go
package observer

import "testing"

func TestObserver(t *testing.T) {
  // 创建张三的信用卡
  creditCard := NewCreditCard("张三")
  // 短信通知订阅信用卡消费及逾期消息
  creditCard.Subscribe(new(shortMessage), ConsumeType, ExpireType)
  // 电子邮件通知订阅信用卡账单及逾期消息
  creditCard.Subscribe(new(email), BillType, ExpireType)
  // 电话通知订阅信用卡逾期消息，同时逾期消息通过三种方式通知
  creditCard.Subscribe(new(telephone), ExpireType)

  creditCard.Consume(500.00) // 信用卡消费
  creditCard.Consume(800.00) // 信用卡消费
  creditCard.SendBill()      // 信用卡发送账单
  creditCard.Expire()        // 信用卡逾期

  // 信用卡逾期消息取消电子邮件及短信通知订阅
  creditCard.Unsubscribe(new(email), ExpireType)
  creditCard.Unsubscribe(new(shortMessage), ExpireType)
  creditCard.Consume(300.00) // 信用卡消费
  creditCard.Expire()        // 信用卡逾期
}
```

（六）运行结果

```
=== RUN   TestObserver
通过【手机短息】发送消息:尊敬的持卡人张三,您当前消费500.00元;
通过【手机短息】发送消息:尊敬的持卡人张三,您当前消费800.00元;
通过【电子邮件】发送消息:尊敬的持卡人张三,您本月账单已出，消费总额1300.00元;
```

```
通过【手机短息】发送消息:尊敬的持卡人张三,您本月账单已逾期，请及时还款，总额1300.00元;
通过【电子邮件】发送消息:尊敬的持卡人张三,您本月账单已逾期，请及时还款，总额1300.00元;
通过【电话】告知:尊敬的持卡人张三,您本月账单已逾期，请及时还款，总额1300.00元;
通过【手机短息】发送消息:尊敬的持卡人张三,您当前消费300.00元;
通过【电话】告知:尊敬的持卡人张三,您本月账单已逾期，请及时还款，总额1600.00元;
--- PASS: TestObserver (0.00s)
PASS
```

状态模式

（一）概念

状态模式是一种行为设计模式，让你能在一个对象的内部状态变化时改变其行为，使其看上去就像改变了自身所属的类一样。

该模式将与状态相关的行为抽取到独立的状态类中，让原对象将工作委派给这些类的实例，而不是自行进行处理。

状态迁移有四个元素组成，起始状态、触发迁移的事件，终止状态以及要执行的动作，每个具体的状态包含触发状态迁移的执行方法，迁移方法的实现是执行持有状态对象的动作方法，同时设置状态为下一个流转状态；持有状态的业务对象包含有触发状态迁移方法，这些迁移方法将请求委托给当前具体状态对象的迁移方法。

（二）示例

IPhone手机充电就是一个手机电池状态的流转，一开始手机处于有电状态，插入充电插头后，继续充电到满电状态，并进入断电保护，拔出充电插头后使用手机，由满电逐渐变为没电，最终关机；

状态迁移表：

起始状态	触发事件	终止状态	执行动作
有电	插入充电线	满电	充电

有电	拔出充电线	没电	耗电
满电	插入充电线	满电	停止充电
满电	拔出充电线	有电	耗电
没电	插入充电线	有电	充电
没电	拔出充电线	没电	关机

（三）电池状态

```go
package state

import "fmt"

// BatteryState 电池状态接口，支持手机充电线插拔事件
type BatteryState interface {
  ConnectPlug(iPhone *IPhone) string
  DisconnectPlug(iPhone *IPhone) string
}

// fullBatteryState 满电状态
type fullBatteryState struct{}

func (s *fullBatteryState) String() string {
  return "满电状态"
}

func (s *fullBatteryState) ConnectPlug(iPhone *IPhone) string {
  return iPhone.pauseCharge()
}

func (s *fullBatteryState) DisconnectPlug(iPhone *IPhone) string {
  iPhone.SetBatteryState(PartBatteryState)
  return fmt.Sprintf("%s,%s转为%s", iPhone.consume(), s, PartBatteryState)
}

// emptyBatteryState 空电状态
type emptyBatteryState struct{}
```

```go

func (s *emptyBatteryState) String() string {
  return "没电状态"
}

func (s *emptyBatteryState) ConnectPlug(iPhone *IPhone) string {
  iPhone.SetBatteryState(PartBatteryState)
  return fmt.Sprintf("%s,%s转为%s", iPhone.charge(), s, PartBatteryState)
}

func (s *emptyBatteryState) DisconnectPlug(iPhone *IPhone) string {
  return iPhone.shutdown()
}

// partBatteryState 部分电状态
type partBatteryState struct{}

func (s *partBatteryState) String() string {
  return "有电状态"
}

func (s *partBatteryState) ConnectPlug(iPhone *IPhone) string {
  iPhone.SetBatteryState(FullBatteryState)
  return fmt.Sprintf("%s,%s转为%s", iPhone.charge(), s, FullBatteryState)
}

func (s *partBatteryState) DisconnectPlug(iPhone *IPhone) string {
  iPhone.SetBatteryState(EmptyBatteryState)
  return fmt.Sprintf("%s,%s转为%s", iPhone.consume(), s, EmptyBatteryState)
}
```

（四）IPhone手机

```go
package state

import "fmt"
```

```go

// 电池状态单例，全局统一使用三个状态的单例，不需要重复创建
var (
  FullBatteryState  = new(fullBatteryState)  // 满电
  EmptyBatteryState = new(emptyBatteryState) // 空电
  PartBatteryState  = new(partBatteryState)  // 部分电
)

// IPhone 已手机充电为例，实现状态模式
type IPhone struct {
  model        string       // 手机型号
  batteryState BatteryState // 电池状态
}

// NewIPhone 创建指定型号手机
func NewIPhone(model string) *IPhone {
  return &IPhone{
    model:        model,
    batteryState: PartBatteryState,
  }
}

// BatteryState 输出电池当前状态
func (i *IPhone) BatteryState() string {
  return fmt.Sprintf("iPhone %s 当前为%s", i.model, i.batteryState)
}

// ConnectPlug 连接充电线
func (i *IPhone) ConnectPlug() string {
  return fmt.Sprintf("iPhone %s 连接电源线,%s", i.model, i.batteryState.Conn
}

// DisconnectPlug 断开充电线
func (i *IPhone) DisconnectPlug() string {
  return fmt.Sprintf("iPhone %s 断开电源线,%s", i.model, i.batteryState.Disc
}

// SetBatteryState 设置电池状态
func (i *IPhone) SetBatteryState(state BatteryState) {
  i.batteryState = state
```

```
}

func (i *IPhone) charge() string {
  return "正在充电"
}

func (i *IPhone) pauseCharge() string {
  return "电已满,暂停充电"
}

func (i *IPhone) shutdown() string {
  return "手机关闭"
}

func (i *IPhone) consume() string {
  return "使用中,消耗电量"
}
```

（五）测试程序

```
package state

import (
  "fmt"
  "testing"
)

func TestState(t *testing.T) {
  iPhone13Pro := NewIPhone("13 pro") // 刚创建的手机有部分电

  fmt.Println(iPhone13Pro.BatteryState()) // 打印部分电状态
  fmt.Println(iPhone13Pro.ConnectPlug())  // 插上电源插头，继续充满电
  fmt.Println(iPhone13Pro.ConnectPlug())  // 满电后再充电，会触发满电保护

  fmt.Println(iPhone13Pro.DisconnectPlug()) // 拔掉电源，使用手机消耗电量，变为有
  fmt.Println(iPhone13Pro.DisconnectPlug()) // 一直使用手机，直到没电
```

```
  fmt.Println(iPhone13Pro.DisconnectPlug()) // 没电后会关机

  fmt.Println(iPhone13Pro.ConnectPlug()) // 再次插上电源一会，变为有电状态
}
```

（六）运行结果

```
=== RUN   TestState
iPhone 13 pro 当前为有电状态
iPhone 13 pro 连接电源线,正在充电,有电状态转为满电状态
iPhone 13 pro 连接电源线,电已满,暂停充电
iPhone 13 pro 断开电源线,使用中,消耗电量,满电状态转为有电状态
iPhone 13 pro 断开电源线,使用中,消耗电量,有电状态转为没电状态
iPhone 13 pro 断开电源线,手机关闭
iPhone 13 pro 连接电源线,正在充电,没电状态转为有电状态
--- PASS: TestState (0.00s)
PASS
```

策略模式

（一）概念

策略模式是一种行为设计模式，它能让你定义一系列算法，并将每种算法分别放入独立的类中，以使算法的对象能够相互替换。

原始对象被称为上下文，它包含指向策略对象的引用并将执行行为的任务分派给策略对象。为了改变上下文完成其工作的方式，其他对象可以使用另一个对象来替换当前链接的策略对象。

策略模式是最常用的设计模式，也是比较简单的设计模式，是以多态替换条件表达式重构方法的具体实现，是面向接口编程原则的最直接体现；

（二）示例

北京是一个四季分明的城市，每个季节天气情况都有明显特点；我们定义一个显示天气情况的季节接口，具体的四季实现，都会保存一个城市和天气情况的映射表，城市对象会包含季节接口，随着四季的变化，天气情况也随之变化；

（三）四季天气

```
package strategy

import "fmt"

// Season 季节的策略接口，不同季节表现得天气不同
type Season interface {
  ShowWeather(city string) string // 显示指定城市的天气情况
}

type spring struct {
  weathers map[string]string // 存储不同城市春天气候
}

func NewSpring() *spring {
  return &spring{
    weathers: map[string]string{"北京": "干燥多风", "昆明": "清凉舒适"},
  }
}

func (s *spring) ShowWeather(city string) string {
  return fmt.Sprintf("%s的春天，%s;", city, s.weathers[city])
}

type summer struct {
  weathers map[string]string // 存储不同城市夏天气候
}

func NewSummer() *summer {
```

```
  return &summer{
    weathers: map[string]string{"北京": "高温多雨", "昆明": "清凉舒适"},
  }
}

func (s *summer) ShowWeather(city string) string {
  return fmt.Sprintf("%s的夏天，%s;", city, s.weathers[city])
}

type autumn struct {
  weathers map[string]string // 存储不同城市秋天气候
}

func NewAutumn() *autumn {
  return &autumn{
    weathers: map[string]string{"北京": "凉爽舒适", "昆明": "清凉舒适"},
  }
}

func (a *autumn) ShowWeather(city string) string {
  return fmt.Sprintf("%s的秋天，%s;", city, a.weathers[city])
}

type winter struct {
  weathers map[string]string // 存储不同城市冬天气候
}

func NewWinter() *winter {
  return &winter{
    weathers: map[string]string{"北京": "干燥寒冷", "昆明": "清凉舒适"},
  }
}

func (w *winter) ShowWeather(city string) string {
  return fmt.Sprintf("%s的冬天，%s;", city, w.weathers[city])
}
```

（四）城市气候

```go
package strategy

import (
  "fmt"
)

// City 城市
type City struct {
  name    string
  feature string
  season  Season
}

// NewCity 根据名称及季候特征创建城市
func NewCity(name, feature string) *City {
  return &City{
    name:    name,
    feature: feature,
  }
}

// SetSeason 设置不同季节，类似天气在不同季节的不同策略
func (c *City) SetSeason(season Season) {
  c.season = season
}

// String 显示城市的气候信息
func (c *City) String() string {
  return fmt.Sprintf("%s%s，%s", c.name, c.feature, c.season.ShowWeather(c.na
}
```

（五）测试程序

```go
package strategy

import (
  "fmt"
  "testing"
)

func TestStrategy(t *testing.T) {
  Beijing := NewCity("北京", "四季分明")

  Beijing.SetSeason(NewSpring())
  fmt.Println(Beijing)

  Beijing.SetSeason(NewSummer())
  fmt.Println(Beijing)

  Beijing.SetSeason(NewAutumn())
  fmt.Println(Beijing)

  Beijing.SetSeason(NewWinter())
  fmt.Println(Beijing)
}
```

（六）运行结果

```
=== RUN   TestStrategy
北京四季分明，北京的春天，干燥多风;
北京四季分明，北京的夏天，高温多雨;
北京四季分明，北京的秋天，凉爽舒适;
北京四季分明，北京的冬天，干燥寒冷;
--- PASS: TestStrategy (0.00s)
PASS
```

模板方法模式

（一）概念

模板方法模式是一种行为设计模式，它在超类中定义了一个算法的框架，允许子类在不修改结构的情况下重写算法的特定步骤。

由于GO语言没有继承的语法，模板方法又是依赖继承实现的设计模式，因此GO语言实现模板方法比较困难， GO语言支持隐式内嵌字段“继承”其他结构体的字段与方法，但是这个并不是真正意义上的继承语法，外层结构重写隐式字段中的算法特定步骤后，无法动态绑定到“继承”过来的算法的框架方法调用中，因此不能实现模板方法模式的语义。

（二）示例

本示例给出一种间接实现模板方法的方式，也比较符合模板方法模式的定义：

- 将多个算法特定步骤组合成一个接口；
- 基类隐式内嵌算法步骤接口，同时调用算法步骤接口的各方法，实现算法的模板方法，此时基类内嵌的算法步骤接口并没有真正的处理行为；
- 子类隐式内嵌基类，并覆写算法步骤接口的方法；
- 通过工厂方法创建具体子类，并将自己的引用赋值给基类中算法步骤接口字段；

以演员装扮为例，演员的装扮是分为化妆，穿衣，配饰三步骤，三个步骤又根据不同角色的演员有所差别，因此演员基类实现装扮的模板方法，对于化妆，穿衣，配饰的三个步骤，在子类演员中具体实现，子类具体演员分为，男演员、女演员和儿童演员；

（三）演员基类

```
package templatemethod

```

```go
import (
  "bytes"
  "fmt"
)

// IActor 演员接口
type IActor interface {
  DressUp() string // 装扮
}

// dressBehavior 装扮的多个行为，这里多个行为是私有的，通过DressUp模版方法调用
type dressBehavior interface {
  makeUp() string // 化妆
  clothe() string // 穿衣
  wear() string   // 配饰
}

// BaseActor 演员基类
type BaseActor struct {
  roleName      string // 扮演角色
  dressBehavior        // 装扮行为
}

// DressUp 统一实现演员接口的DressUp模版方法，装扮过程通过不同装扮行为进行扩展
func (b *BaseActor) DressUp() string {
  buf := bytes.Buffer{}
  buf.WriteString(fmt.Sprintf("扮演%s的", b.roleName))
  buf.WriteString(b.makeUp())
  buf.WriteString(b.clothe())
  buf.WriteString(b.wear())
  return buf.String()
}
```

（四）具体演员

```go
package templatemethod
```

```
2
3  // womanActor 扩展装扮行为的女演员
4  type womanActor struct {
5    BaseActor
6  }
7
8  // NewWomanActor 指定角色创建女演员
9  func NewWomanActor(roleName string) *womanActor {
10   actor := new(womanActor)    // 创建女演员
11   actor.roleName = roleName   // 设置角色
12   actor.dressBehavior = actor // 将女演员实现的扩展装扮行为，设置给自己的装扮行为
13   return actor
14 }
15
16 // 化妆
17 func (w *womanActor) makeUp() string {
18   return "女演员涂着口红，画着眉毛；"
19 }
20
21 // 穿衣
22 func (w *womanActor) clothe() string {
23   return "穿着连衣裙；"
24 }
25
26 // 配饰
27 func (w *womanActor) wear() string {
28   return "带着耳环，手拎着包；"
29 }
30
31 // manActor 扩展装扮行为的男演员
32 type manActor struct {
33   BaseActor
34 }
35
36 func NewManActor(roleName string) *manActor {
37   actor := new(manActor)
38   actor.roleName = roleName
39   actor.dressBehavior = actor // 将男演员实现的扩展装扮行为，设置给自己的装扮行为
40   return actor
41 }
```

```
42
43 func (m *manActor) makeUp() string {
44   return "男演员刮净胡子，抹上发胶；"
45 }
46
47 func (m *manActor) clothe() string {
48   return "穿着一身西装；"
49 }
50
51 func (m *manActor) wear() string {
52   return "带上手表，抽着烟；"
53 }
54
55 // NewChildActor 扩展装扮行为的儿童演员
56 type childActor struct {
57   BaseActor
58 }
59
60 func NewChildActor(roleName string) *childActor {
61   actor := new(childActor)
62   actor.roleName = roleName
63   actor.dressBehavior = actor // 将儿童演员实现的扩展装扮行为，设置给自己的装扮行
64   return actor
65 }
66
67 func (c *childActor) makeUp() string {
68   return "儿童演员抹上红脸蛋；"
69 }
70
71 func (c *childActor) clothe() string {
72   return "穿着一身童装；"
73 }
74
75 func (c *childActor) wear() string {
76   return "手里拿着一串糖葫芦；"
77 }
78
```

（五）测试程序

```
package templatemethod

import (
  "fmt"
  "testing"
)

func TestTemplateMethod(t *testing.T) {
  showActors(NewWomanActor("妈妈"), NewManActor("爸爸"), NewChildActor("儿子"))
}

// showActors 显示演员的装扮信息
func showActors(actors ...IActor) {
  for _, actor := range actors {
    fmt.Println(actor.DressUp())
  }
}
```

（六）运行结果

```
=== RUN   TestTemplateMethod
扮演妈妈的女演员涂着口红，画着眉毛；穿着连衣裙；带着耳环，手拎着包；
扮演爸爸的男演员刮净胡子，抹上发胶；穿着一身西装；带上手表，抽着烟；
扮演儿子的儿童演员抹上红脸蛋；穿着一身童装；手里拿着一串糖葫芦；
--- PASS: TestTemplateMethod (0.00s)
PASS
```

访问者模式

（一）概念

访问者模式是一种行为设计模式，它能将算法与其所作用的对象隔离开来。允许你在不修改已有代码的情况下向已有类层次结构中增加新的行为。

访问者接口需要根据被访问者具体类，定义多个相似的访问方法，每个具体类对应一个访问方法；每个被访问者需要实现一个接受访问者对象的方法，方法的实现就是去调用访问者接口对应该类的访问方法；这个接受方法可以传入不同目的访问者接口的具体实现，从而在不修改被访问对象的前提下，增加新的功能；

（二）示例

公司中存在多种类型的员工，包括产品经理、软件工程师、人力资源等，他们的KPI指标不尽相同，产品经理为上线产品数量及满意度，软件工程师为实现的需求数及修改bug数，人力资源为招聘员工的数量；公司要根据员工完成的KPI进行表彰公示，同时根据KPI完成情况定薪酬，这些功能都是员工类职责之外的，不能修改员工本身的类，我们通过访问者模式，实现KPI表彰排名及薪酬发放；

（三）员工结构

```go
package visitor

import "fmt"

// Employee 员工接口
type Employee interface {
  KPI() string                      // 完成kpi信息
  Accept(visitor EmployeeVisitor) // 接受访问者对象
}

// productManager 产品经理
type productManager struct {
  name         string // 名称
  productNum   int    // 上线产品数
```

```go
  satisfaction int    // 平均满意度
}

func NewProductManager(name string, productNum int, satisfaction int) *produc
  return &productManager{
    name:         name,
    productNum:   productNum,
    satisfaction: satisfaction,
  }
}

func (p *productManager) KPI() string {
  return fmt.Sprintf("产品经理%s，上线%d个产品，平均满意度为%d", p.name, p.produc
}

func (p *productManager) Accept(visitor EmployeeVisitor) {
  visitor.VisitProductManager(p)
}

// softwareEngineer 软件工程师
type softwareEngineer struct {
  name           string // 姓名
  requirementNum int    // 完成需求数
  bugNum         int    // 修复问题数
}

func NewSoftwareEngineer(name string, requirementNum int, bugNum int) *softwa
  return &softwareEngineer{
    name:           name,
    requirementNum: requirementNum,
    bugNum:         bugNum,
  }
}

func (s *softwareEngineer) KPI() string {
  return fmt.Sprintf("软件工程师%s，完成%d个需求，修复%d个问题", s.name, s.requir
}

func (s *softwareEngineer) Accept(visitor EmployeeVisitor) {
  visitor.VisitSoftwareEngineer(s)
```

```go
}

// hr 人力资源
type hr struct {
  name       string // 姓名
  recruitNum int    // 招聘人数
}

func NewHR(name string, recruitNum int) *hr {
  return &hr{
    name:       name,
    recruitNum: recruitNum,
  }
}

func (h *hr) KPI() string {
  return fmt.Sprintf("人力资源%s，招聘%d名员工", h.name, h.recruitNum)
}

func (h *hr) Accept(visitor EmployeeVisitor) {
  visitor.VisitHR(h)
}
```

（四）员工访问者

```go
package visitor

import (
  "fmt"
  "sort"
)

// EmployeeVisitor 员工访问者接口
type EmployeeVisitor interface {
  VisitProductManager(pm *productManager)     // 访问产品经理
  VisitSoftwareEngineer(se *softwareEngineer) // 访问软件工程师
```

```
  VisitHR(hr *hr)                                  // 访问人力资源
}

// kpi kpi对象
type kpi struct {
  name string // 完成kpi姓名
  sum  int    // 完成kpi总数量
}

// kpiTopVisitor 员工kpi排名访问者
type kpiTopVisitor struct {
  top []*kpi
}

func (k *kpiTopVisitor) VisitProductManager(pm *productManager) {
  k.top = append(k.top, &kpi{
    name: pm.name,
    sum:  pm.productNum + pm.satisfaction,
  })
}

func (k *kpiTopVisitor) VisitSoftwareEngineer(se *softwareEngineer) {
  k.top = append(k.top, &kpi{
    name: se.name,
    sum:  se.requirementNum + se.bugNum,
  })
}

func (k *kpiTopVisitor) VisitHR(hr *hr) {
  k.top = append(k.top, &kpi{
    name: hr.name,
    sum:  hr.recruitNum,
  })
}

// Publish 发布KPI排行榜
func (k *kpiTopVisitor) Publish() {
  sort.Slice(k.top, func(i, j int) bool {
    return k.top[i].sum > k.top[j].sum
  })
```

```
  for i, curKPI := range k.top {
    fmt.Printf("第%d名%s：完成KPI总数%d\n", i+1, curKPI.name, curKPI.sum)
  }
}

// salaryVisitor 薪酬访问者
type salaryVisitor struct{}

func (s *salaryVisitor) VisitProductManager(pm *productManager) {
  fmt.Printf("产品经理基本薪资：1000元，KPI单位薪资：100元，")
  fmt.Printf("%s，总工资为%d元\n", pm.KPI(), (pm.productNum+pm.satisfaction)*1
}

func (s *salaryVisitor) VisitSoftwareEngineer(se *softwareEngineer) {
  fmt.Printf("软件工程师基本薪资：1500元，KPI单位薪资：80元，")
  fmt.Printf("%s，总工资为%d元\n", se.KPI(), (se.requirementNum+se.bugNum)*80+
}

func (s *salaryVisitor) VisitHR(hr *hr) {
  fmt.Printf("人力资源基本薪资：800元，KPI单位薪资：120元，")
  fmt.Printf("%s，总工资为%d元\n", hr.KPI(), hr.recruitNum*120+800)
}
```

（五）测试程序

```
package visitor

import "testing"

func TestVisitor(t *testing.T) {
  allEmployees := AllEmployees() // 获取所有员工
  kpiTop := new(kpiTopVisitor)   // 创建KPI排行访问者
  VisitAllEmployees(kpiTop, allEmployees)
  kpiTop.Publish() // 发布排行榜

  salary := new(salaryVisitor) // 创建薪酬访问者
```

```
  VisitAllEmployees(salary, allEmployees)
}

// VisitAllEmployees 遍历所有员工调用访问者
func VisitAllEmployees(visitor EmployeeVisitor, allEmployees []Employee) {
  for _, employee := range allEmployees {
    employee.Accept(visitor)
  }
}

// AllEmployees 获得所有公司员工
func AllEmployees() []Employee {
  var employees []Employee
  employees = append(employees, NewHR("小明", 10))
  employees = append(employees, NewProductManager("小红", 4, 7))
  employees = append(employees, NewSoftwareEngineer("张三", 10, 5))
  employees = append(employees, NewSoftwareEngineer("李四", 3, 6))
  employees = append(employees, NewSoftwareEngineer("王五", 7, 1))
  return employees
}
```

（六）运行结果

```
=== RUN   TestVisitor
第1名张三：完成KPI总数15
第2名小红：完成KPI总数11
第3名小明：完成KPI总数10
第4名李四：完成KPI总数9
第5名王五：完成KPI总数8
人力资源基本薪资：800元，KPI单位薪资：120元，人力资源小明，招聘10名员工，总工资为2000
产品经理基本薪资：1000元，KPI单位薪资：100元，产品经理小红，上线4个产品，平均满意度为
软件工程师基本薪资：1500元，KPI单位薪资：80元，软件工程师张三，完成10个需求，修复5个问
软件工程师基本薪资：1500元，KPI单位薪资：80元，软件工程师李四，完成3个需求，修复6个问
软件工程师基本薪资：1500元，KPI单位薪资：80元，软件工程师王五，完成7个需求，修复1个问
--- PASS: TestVisitor (0.00s)
```

收录于合集 #腾讯技术人原创集 32

‹上一篇
万字图文讲透数据库缓存一致性问题

下一篇›
这项技术，让虚拟世界走进现实！

文章已于2022-11-21修改

喜欢此内容的人还喜欢

Python老手也会犯的20个新手级错误
51CTO技术栈

Jenkins + Docker 一键自动化部署 Spring Boot 项目
码猿技术专栏

go语言sync.map源码阅读
海生的go花园