目　　录

前言

创建型模式

- 简单工厂模式
- 工厂方法模式
- 抽象工厂模式
- 创建者模式
- 原型模式
- 单例模式

结构型模式

- 外观模式
- 适配器模式
- 代理模式
- 组合模式
- 享元模式
- 装饰模式
- 桥模式

行为型模式

- 中介者模式
- 观察者模式
- 命令模式
- 迭代器模式
- 模板方法模式
- 策略模式
- 状态模式
- 备忘录模式
- 解释器模式
- 职责链模式
- 访问者模式

前言

设计模式（Design pattern）是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。
使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。 毫无疑问，设计模式于己于他人于系统都是多赢的，设计模式使代码编制真正工程化，设计模式是软件工程的基石，如同大厦的一块块砖石一样。
项目中合理的运用设计模式可以完美的解决很多问题，每种模式在现在中都有相应的原理来与之对应，每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的核心解决方案，这也是它能被广泛应用的原因。

一、设计模式的分类
创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代器模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

本文档转自：https://github.com/senghoo/golang-design-pattern

创建型模式

简单工厂模式

工厂方法模式

抽象工厂模式

创建者模式

原型模式

单例模式

简单工厂模式

go 语言没有构造函数一说，所以一般会定义NewXXX函数来初始化相关类。
NewXXX 函数返回接口时就是简单工厂模式，也就是说Golang的一般推荐做法就是简单工厂。

在这个simplefactory包中只有API 接口和NewAPI函数为包外可见，封装了实现细节。

simple.go代码

```go
package simplefactory

import "fmt"

//API is interface
type API interface {
    Say(name string) string
}

//NewAPI return Api instance by type
func NewAPI(t int) API {
    if t == 1 {
        return &hiAPI{}
    } else if t == 2 {
        return &helloAPI{}
    }
    return nil
}

//hiAPI is one of API implement
type hiAPI struct{}

//Say hi to name
func (*hiAPI) Say(name string) string {
    return fmt.Sprintf("Hi, %s", name)
}

//HelloAPI is another API implement
type helloAPI struct{}

//Say hello to name
func (*helloAPI) Say(name string) string {
    return fmt.Sprintf("Hello, %s", name)
}
```

simple_test.go代码

```
package simplefactory

import "testing"

//TestType1 test get hiapi with factory
func TestType1(t *testing.T) {
    api := NewAPI(1)
    s := api.Say("Tom")
    if s != "Hi, Tom" {
        t.Fatal("Type1 test fail")
    }
}

func TestType2(t *testing.T) {
    api := NewAPI(2)
    s := api.Say("Tom")
    if s != "Hello, Tom" {
        t.Fatal("Type2 test fail")
    }
}
```

工厂方法模式

工厂方法模式使用子类的方式延迟生成对象到子类中实现。

Go中不存在继承 所以使用匿名组合来实现

factorymethod.go

```go
package factorymethod

//Operator 是被封装的实际类接口
type Operator interface {
	SetA(int)
	SetB(int)
	Result() int
}

//OperatorFactory 是工厂接口
type OperatorFactory interface {
	Create() Operator
}

//OperatorBase 是Operator 接口实现的基类，封装公用方法
type OperatorBase struct {
	a, b int
}

//SetA 设置 A
func (o *OperatorBase) SetA(a int) {
	o.a = a
}

//SetB 设置 B
func (o *OperatorBase) SetB(b int) {
	o.b = b
}

//PlusOperatorFactory 是 PlusOperator 的工厂类
type PlusOperatorFactory struct{}

func (PlusOperatorFactory) Create() Operator {
	return &PlusOperator{
		OperatorBase: &OperatorBase{},
	}
}

//PlusOperator Operator 的实际加法实现
type PlusOperator struct {
    *OperatorBase
}

//Result 获取结果
func (o PlusOperator) Result() int {
    return o.a + o.b
}

//MinusOperatorFactory 是 MinusOperator 的工厂类
type MinusOperatorFactory struct{}

func (MinusOperatorFactory) Create() Operator {
    return &MinusOperator{
        OperatorBase: &OperatorBase{},
    }
}

//MinusOperator Operator 的实际减法实现
type MinusOperator struct {
    *OperatorBase
}

//Result 获取结果
func (o MinusOperator) Result() int {
    return o.a - o.b
}
```

factorymethod_test.go

```
package factorymethod

import "testing"

func compute(factory OperatorFactory, a, b int) int {
    op := factory.Create()
    op.SetA(a)
    op.SetB(b)
    return op.Result()
}

func TestOperator(t *testing.T) {
    var (
        factory OperatorFactory
    )

    factory = PlusOperatorFactory{}
    if compute(factory, 1, 2) != 3 {
        t.Fatal("error with factory method pattern")
    }

    factory = MinusOperatorFactory{}
    if compute(factory, 4, 2) != 2 {
        t.Fatal("error with factory method pattern")
    }
}
```

抽象工厂模式

抽象工厂模式用于生成产品族的工厂，所生成的对象是有关联的。

如果抽象工厂退化成生成的对象无关联则成为工厂函数模式。

比如本例子中使用RDB和XML存储订单信息，抽象工厂分别能生成相关的主订单信息和订单详情信息。
如果业务逻辑中需要替换使用的时候只需要改动工厂函数相关的类就能替换使用不同的存储方式了。

abstractfactory.go

```go
package abstractfactory

import "fmt"

//OrderMainDAO 为订单主记录
type OrderMainDAO interface {
    SaveOrderMain()
}

//OrderDetailDAO 为订单详情纪录
type OrderDetailDAO interface {
    SaveOrderDetail()
}

//DAOFactory DAO 抽象模式工厂接口
type DAOFactory interface {
    CreateOrderMainDAO() OrderMainDAO
    CreateOrderDetailDAO() OrderDetailDAO
}

//RDBMainDAP 为关系型数据库的OrderMainDAO实现
type RDBMainDAO struct{}

//SaveOrderMain ...
func (*RDBMainDAO) SaveOrderMain() {
    fmt.Print("rdb main save\n")
}

//RDBDetailDAO 为关系型数据库的OrderDetailDAO实现
type RDBDetailDAO struct{}
```

```go
// SaveOrderDetail ...
func (*RDBDetailDAO) SaveOrderDetail() {
    fmt.Print("rdb detail save\n")
}

//RDBDAOFactory 是RDB 抽象工厂实现
type RDBDAOFactory struct{}

func (*RDBDAOFactory) CreateOrderMainDAO() OrderMainDAO {
    return &RDBMainDAO{}
}

func (*RDBDAOFactory) CreateOrderDetailDAO() OrderDetailDAO {
    return &RDBDetailDAO{}
}

//XMLMainDAO XML存储
type XMLMainDAO struct{}

//SaveOrderMain ...
func (*XMLMainDAO) SaveOrderMain() {
    fmt.Print("xml main save\n")
}

//XMLDetailDAO XML存储
type XMLDetailDAO struct{}

// SaveOrderDetail ...
func (*XMLDetailDAO) SaveOrderDetail() {
    fmt.Print("xml detail save")
}

//XMLDAOFactory 是RDB 抽象工厂实现
type XMLDAOFactory struct{}

func (*XMLDAOFactory) CreateOrderMainDAO() OrderMainDAO {
    return &XMLMainDAO{}
}

func (*XMLDAOFactory) CreateOrderDetailDAO() OrderDetailDAO {
    return &XMLDetailDAO{}
}
```

abstractfactory_test.go

```go
package abstractfactory

func getMainAndDetail(factory DAOFactory) {
    factory.CreateOrderMainDAO().SaveOrderMain()
    factory.CreateOrderDetailDAO().SaveOrderDetail()
}

func ExampleRdbFactory() {
    var factory DAOFactory
    factory = &RDBDAOFactory{}
    getMainAndDetail(factory)
    // Output:
    // rdb main save
    // rdb detail save
}

func ExampleXmlFactory() {
    var factory DAOFactory
    factory = &XMLDAOFactory{}
    getMainAndDetail(factory)
    // Output:
    // xml main save
    // xml detail save
}
```

创建者模式

将一个复杂对象的构建分离成多个简单对象的构建组合

builder.go

```
package builder

//Builder 是生成器接口
type Builder interface {
    Part1()
    Part2()
    Part3()
}

type Director struct {
    builder Builder
}

// NewDirector ...
func NewDirector(builder Builder) *Director {
    return &Director{
        builder: builder,
    }
}

//Construct Product
func (d *Director) Construct() {
    d.builder.Part1()
    d.builder.Part2()
    d.builder.Part3()
}

type Builder1 struct {
    result string
}

func (b *Builder1) Part1() {
    b.result += "1"
}

func (b *Builder1) Part2() {
    b.result += "2"
}
```

```go
func (b *Builder1) Part3() {
	b.result += "3"
}

func (b *Builder1) GetResult() string {
	return b.result
}

type Builder2 struct {
	result int
}

func (b *Builder2) Part1() {
	b.result += 1
}

func (b *Builder2) Part2() {
	b.result += 2
}

func (b *Builder2) Part3() {
	b.result += 3
}

func (b *Builder2) GetResult() int {
	return b.result
}
```

builder_test.go

```go
package builder

import "testing"

func TestBuilder1(t *testing.T) {
	builder := &Builder1{}
	director := NewDirector(builder)
	director.Construct()
	res := builder.GetResult()
	if res != "123" {
		t.Fatalf("Builder1 fail expect 123 acture %s", res)
	}
}
```

```go
func TestBuilder2(t *testing.T) {
    builder := &Builder2{}
    director := NewDirector(builder)
    director.Construct()
    res := builder.GetResult()
    if res != 6 {
        t.Fatalf("Builder2 fail expect 6 acture %d", res)
    }
}
```

原型模式

原型模式使对象能复制自身，并且暴露到接口中，使客户端面向接口编程时，不知道接口实际对象的情况下生成新的对象。

原型模式配合原型管理器使用，使得客户端在不知道具体类的情况下，通过接口管理器得到新的实例，并且包含部分预设定配置。

prototype.go

```go
package prototype

//Cloneable 是原型对象需要实现的接口
type Cloneable interface {
    Clone() Cloneable
}

type PrototypeManager struct {
    prototypes map[string]Cloneable
}

func NewPrototypeManager() *PrototypeManager {
    return &PrototypeManager{
        prototypes: make(map[string]Cloneable),
    }
}

func (p *PrototypeManager) Get(name string) Cloneable {
    return p.prototypes[name]
}

func (p *PrototypeManager) Set(name string, prototype Cloneable) {
    p.prototypes[name] = prototype
}
```

prototype_test.go

```go
package prototype

import "testing"

var manager *PrototypeManager
```

```
type Type1 struct {
    name string
}

func (t *Type1) Clone() Cloneable {
    tc := *t
    return &tc
}

type Type2 struct {
    name string
}

func (t *Type2) Clone() Cloneable {
    tc := *t
    return &tc
}

func TestClone(t *testing.T) {
    t1 := manager.Get("t1")

    t2 := t1.Clone()

    if t1 == t2 {
        t.Fatal("error! get clone not working")
    }
}

func TestCloneFromManager(t *testing.T) {
    c := manager.Get("t1").Clone()

    t1 := c.(*Type1)
    if t1.name != "type1" {
        t.Fatal("error")
    }

}

func init() {
    manager = NewPrototypeManager()

    t1 := &Type1{
        name: "type1",
    }
    manager.Set("t1", t1)
}
```

单例模式

使用懒惰模式的单例模式，使用双重检查加锁保证线程安全

singleton.go

```go
package singleton

import "sync"

//Singleton 是单例模式类
type Singleton struct{}

var singleton *Singleton
var once sync.Once

//GetInstance 用于获取单例模式对象
func GetInstance() *Singleton {
	once.Do(func() {
		singleton = &Singleton{}
	})

	return singleton
}
```

singleton_test.go

```go
package singleton

import (
	"sync"
	"testing"
)

const parCount = 100

func TestSingleton(t *testing.T) {
	ins1 := GetInstance()
	ins2 := GetInstance()
	if ins1 != ins2 {
		t.Fatal("instance is not equal")
	}
}
```

```go
func TestParallelSingleton(t *testing.T) {
    wg := sync.WaitGroup{}
    wg.Add(parCount)
    instances := [parCount]*Singleton{}
    for i := 0; i < parCount; i++ {
        go func(index int) {
            instances[index] = GetInstance()
            wg.Done()
        }(i)
    }
    wg.Wait()
    for i := 1; i < parCount; i++ {
        if instances[i] != instances[i-1] {
            t.Fatal("instance is not equal")
        }
    }
}
```

结构型模式

外观模式

适配器模式

代理模式

组合模式

享元模式

装饰模式

桥模式

外观模式

API 为facade 模块的外观接口，大部分代码使用此接口简化对facade类的访问。

facade模块同时暴露了a和b 两个Module 的NewXXX和interface，其它代码如果需要使用细节功能时可以直接调用。

facade.go

```
package facade

import "fmt"

func NewAPI() API {
    return &apiImpl{
        a: NewAModuleAPI(),
        b: NewBModuleAPI(),
    }
}

//API is facade interface of facade package
type API interface {
    Test() string
}

//facade implement
type apiImpl struct {
    a AModuleAPI
    b BModuleAPI
}

func (a *apiImpl) Test() string {
    aRet := a.a.TestA()
    bRet := a.b.TestB()
    return fmt.Sprintf("%s\n%s", aRet, bRet)
}

//NewAModuleAPI return new AModuleAPI
func NewAModuleAPI() AModuleAPI {
    return &aModuleImpl{}
}

//AModuleAPI ...
type AModuleAPI interface {
    TestA() string
}

type aModuleImpl struct{}

func (*aModuleImpl) TestA() string {
    return "A module running"
}

//NewBModuleAPI return new BModuleAPI
func NewBModuleAPI() BModuleAPI {
    return &bModuleImpl{}
}

//BModuleAPI ...
type BModuleAPI interface {
    TestB() string
}

type bModuleImpl struct{}

func (*bModuleImpl) TestB() string {
    return "B module running"
}
```

facade_test.go

```
package facade

import "testing"

var expect = "A module running\nB module running"

// TestFacadeAPI ...
func TestFacadeAPI(t *testing.T) {
    api := NewAPI()
    ret := api.Test()
    if ret != expect {
        t.Fatalf("expect %s, return %s", expect, ret)
    }
}
```

适配器模式

适配器模式用于转换一种接口适配另一种接口。

实际使用中Adaptee一般为接口，并且使用工厂函数生成实例。

在Adapter中匿名组合Adaptee接口，所以Adapter类也拥有SpecificRequest实例方法，又因为Go语言中非入侵式接口特征，其实Adapter也适配Adaptee接口。

adapter.go

```go
package adapter

//Target 是适配的目标接口
type Target interface {
    Request() string
}

//Adaptee 是被适配的目标接口
type Adaptee interface {
    SpecificRequest() string
}

//NewAdaptee 是被适配接口的工厂函数
func NewAdaptee() Adaptee {
    return &adapteeImpl{}
}

//AdapteeImpl 是被适配的目标类
type adapteeImpl struct{}

//SpecificRequest 是目标类的一个方法
func (*adapteeImpl) SpecificRequest() string {
    return "adaptee method"
}

//NewAdapter 是Adapter的工厂函数
func NewAdapter(adaptee Adaptee) Target {
    return &adapter{
        Adaptee: adaptee,
    }
}

//Adapter 是转换Adaptee为Target接口的适配器
```

```go
type adapter struct {
    Adaptee
}

//Request 实现Target接口
func (a *adapter) Request() string {
    return a.SpecificRequest()
}
```

adapter_test.go

```go
package adapter

import "testing"

var expect = "adaptee method"

func TestAdapter(t *testing.T) {
    adaptee := NewAdaptee()
    target := NewAdapter(adaptee)
    res := target.Request()
    if res != expect {
        t.Fatalf("expect: %s, actual: %s", expect, res)
    }
}
```

代理模式

代理模式用于延迟处理操作或者在进行实际操作前后进行其它处理。

代理模式的常见用法有

- 虚代理
- COW代理
- 远程代理
- 保护代理
- Cache 代理
- 防火墙代理
- 同步代理
- 智能指引

等。。。

proxy.go

```go
package proxy

type Subject interface {
    Do() string
}

type RealSubject struct{}

func (RealSubject) Do() string {
    return "real"
}

type Proxy struct {
    real RealSubject
}

func (p Proxy) Do() string {
    var res string

    // 在调用真实对象之前的工作，检查缓存，判断权限，实例化真实对象等。。
    res += "pre:"

    // 调用真实对象
    res += p.real.Do()

    // 调用之后的操作，如缓存结果，对结果进行处理等。。
    res += ":after"

    return res
}
```

proxy_test.go

```
package proxy

import "testing"

func TestProxy(t *testing.T) {
    var sub Subject
    sub = &Proxy{}

    res := sub.Do()

    if res != "pre:real:after" {
        t.Fail()
    }
}
```

组合模式

组合模式统一对象和对象集，使得使用相同接口使用对象和对象集。

组合模式常用于树状结构，用于统一叶子节点和树节点的访问，并且可以用于应用某一操作到所有子节点。

composite.go

```go
package composite

import "fmt"

type Component interface {
	Parent() Component
	SetParent(Component)
	Name() string
	SetName(string)
	AddChild(Component)
	Print(string)
}

const (
	LeafNode = iota
	CompositeNode
)

func NewComponent(kind int, name string) Component {
	var c Component
	switch kind {
	case LeafNode:
		c = NewLeaf()
	case CompositeNode:
		c = NewComposite()
	}

	c.SetName(name)
	return c
}

type component struct {
	parent Component
	name   string
}
```

```go
func (c *component) Parent() Component {
	return c.parent
}

func (c *component) SetParent(parent Component) {
	c.parent = parent
}

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

func (c *component) SetName(name string) {
	c.name = name
}

func (c *component) AddChild(Component) {}

func (c *component) Print(string) {}

type Leaf struct {
	component
}

func NewLeaf() *Leaf {
	return &Leaf{}
}

func (c *Leaf) Print(pre string) {
	fmt.Printf("%s-%s\n", pre, c.Name())
}

type Composite struct {
	component
	childs []Component
}

func NewComposite() *Composite {
	return &Composite{
		childs: make([]Component, 0),
	}
}

func (c *Composite) AddChild(child Component) {
	child.SetParent(c)
```

```
    c.childs = append(c.childs, child)
}

func (c *Composite) Print(pre string) {
    fmt.Printf("%s+%s\n", pre, c.Name())
    pre += " "
    for _, comp := range c.childs {
        comp.Print(pre)
    }
}
```

composite_test.go

```
package composite

func ExampleComposite() {
    root := NewComponent(CompositeNode, "root")
    c1 := NewComponent(CompositeNode, "c1")
    c2 := NewComponent(CompositeNode, "c2")
    c3 := NewComponent(CompositeNode, "c3")

    l1 := NewComponent(LeafNode, "l1")
    l2 := NewComponent(LeafNode, "l2")
    l3 := NewComponent(LeafNode, "l3")

    root.AddChild(c1)
    root.AddChild(c2)
    c1.AddChild(c3)
    c1.AddChild(l1)
    c2.AddChild(l2)
    c2.AddChild(l3)

    root.Print("")
    // Output:
    // +root
    //  +c1
    //   +c3
    //   -l1
    //  +c2
    //   -l2
    //   -l3
}
```

享元模式

享元模式从对象中剥离出不发生改变且多个实例需要的重复数据，独立出一个享元，使多个对象共享，从而节省内存以及减少对象数量。

flyweight.go

```go
package flyweight

import "fmt"

type ImageFlyweightFactory struct {
    maps map[string]*ImageFlyweight
}

var imageFactory *ImageFlyweightFactory

func GetImageFlyweightFactory() *ImageFlyweightFactory {
    if imageFactory == nil {
        imageFactory = &ImageFlyweightFactory{
            maps: make(map[string]*ImageFlyweight),
        }
    }
    return imageFactory
}

func (f *ImageFlyweightFactory) Get(filename string) *ImageFlyweight {
    image := f.maps[filename]
    if image == nil {
        image = NewImageFlyweight(filename)
        f.maps[filename] = image
    }

    return image
}

type ImageFlyweight struct {
    data string
}

func NewImageFlyweight(filename string) *ImageFlyweight {
    // Load image file
    data := fmt.Sprintf("image data %s", filename)
    return &ImageFlyweight{
        data: data,
    }
}

func (i *ImageFlyweight) Data() string {
    return i.data
}

type ImageViewer struct {
    *ImageFlyweight
}

func NewImageViewer(filename string) *ImageViewer {
    image := GetImageFlyweightFactory().Get(filename)
    return &ImageViewer{
        ImageFlyweight: image,
    }
}

func (i *ImageViewer) Display() {
    fmt.Printf("Display: %s\n", i.Data())
}
```

flyweight_test.go

```
package flyweight

import "testing"

func ExampleFlyweight() {
    viewer := NewImageViewer("image1.png")
    viewer.Display()
    // Output:
    // Display: image data image1.png
}

func TestFlyweight(t *testing.T) {
    viewer1 := NewImageViewer("image1.png")
    viewer2 := NewImageViewer("image1.png")

    if viewer1.ImageFlyweight != viewer2.ImageFlyweight {
        t.Fail()
    }
}
```

装饰模式

装饰模式使用对象组合的方式动态改变或增加对象行为。

Go语言借助于匿名组合和非入侵式接口可以很方便实现装饰模式。

使用匿名组合，在装饰器中不必显式定义转调原对象方法。

decorator.go

```go
package decorator

type Component interface {
	Calc() int
}

type ConcreteComponent struct{}

func (*ConcreteComponent) Calc() int {
	return 0
}

type MulDecorator struct {
	Component
	num int
}

func WarpMulDecorator(c Component, num int) Component {
	return &MulDecorator{
		Component: c,
		num:       num,
	}
}

func (d *MulDecorator) Calc() int {
	return d.Component.Calc() * d.num
}

type AddDecorator struct {
	Component
	num int
}

func WarpAddDecorator(c Component, num int) Component {
    return &AddDecorator{
        Component: c,
        num:       num,
    }
}

func (d *AddDecorator) Calc() int {
    return d.Component.Calc() + d.num
}
```

decorator_test.go

```
package decorator

import "fmt"

func ExampleDecorator() {
    var c Component = &ConcreteComponent{}
    c = WarpAddDecorator(c, 10)
    c = WarpMulDecorator(c, 8)
    res := c.Calc()

    fmt.Printf("res %d\n", res)
    // Output:
    // res 80
}
```

桥模式

桥接模式分离抽象部分和实现部分。使得两部分独立扩展。

桥接模式类似于策略模式，区别在于策略模式封装一系列算法使得算法可以互相替换。

策略模式使抽象部分和实现部分分离，可以独立变化。

bridge.go

```go
package bridge

import "fmt"

type AbstractMessage interface {
    SendMessage(text, to string)
}

type MessageImplementer interface {
    Send(text, to string)
}

type MessageSMS struct{}

func ViaSMS() MessageImplementer {
    return &MessageSMS{}
}

func (*MessageSMS) Send(text, to string) {
    fmt.Printf("send %s to %s via SMS", text, to)
}

type MessageEmail struct{}

func ViaEmail() MessageImplementer {
    return &MessageEmail{}
}

func (*MessageEmail) Send(text, to string) {
    fmt.Printf("send %s to %s via Email", text, to)
}

type CommonMessage struct {
    method MessageImplementer
}

func NewCommonMessage(method MessageImplementer) *CommonMessage {
    return &CommonMessage{
        method: method,
    }
}

func (m *CommonMessage) SendMessage(text, to string) {
    m.method.Send(text, to)
}

type UrgencyMessage struct {
    method MessageImplementer
}

func NewUrgencyMessage(method MessageImplementer) *UrgencyMessage {
    return &UrgencyMessage{
        method: method,
    }
}

func (m *UrgencyMessage) SendMessage(text, to string) {
    m.method.Send(fmt.Sprintf("[Urgency] %s", text), to)
}
```

bridge_test.go

```
package bridge

func ExampleCommonSMS() {
    m := NewCommonMessage(ViaSMS())
    m.SendMessage("have a drink?", "bob")
    // Output:
    // send have a drink? to bob via SMS
}

func ExampleCommonEmail() {
    m := NewCommonMessage(ViaEmail())
    m.SendMessage("have a drink?", "bob")
    // Output:
    // send have a drink? to bob via Email
}

func ExampleUrgencySMS() {
```

```go
    m := NewUrgencyMessage(ViaSMS())
    m.SendMessage("have a drink?", "bob")
    // Output:
    // send [Urgency] have a drink? to bob via SMS
}

func ExampleUrgencyEmail() {
    m := NewUrgencyMessage(ViaEmail())
    m.SendMessage("have a drink?", "bob")
    // Output:
    // send [Urgency] have a drink? to bob via Email
}
```

行为型模式

中介者模式

观察者模式

命令模式

迭代器模式

模板方法模式

策略模式

状态模式

备忘录模式

解释器模式

职责链模式

访问者模式

中介者模式

中介者模式封装对象之间互交，使依赖变的简单，并且使复杂互交简单化，封装在中介者中。

例子中的中介者使用单例模式生成中介者。

中介者的change使用switch判断类型。

mediator.go

```go
package mediator

import (
    "fmt"
    "strings"
)

type CDDriver struct {
    Data string
}

func (c *CDDriver) ReadData() {
    c.Data = "music,image"

    fmt.Printf("CDDriver: reading data %s\n", c.Data)
    GetMediatorInstance().changed(c)
}

type CPU struct {
    Video string
    Sound string
}

func (c *CPU) Process(data string) {
    sp := strings.Split(data, ",")
    c.Sound = sp[0]
    c.Video = sp[1]

    fmt.Printf("CPU: split data with Sound %s, Video %s\n", c.Sound, c.Video)
    GetMediatorInstance().changed(c)
}

type VideoCard struct {
    Data string
}

func (v *VideoCard) Display(data string) {
	v.Data = data
	fmt.Printf("VideoCard: display %s\n", v.Data)
	GetMediatorInstance().changed(v)
}

type SoundCard struct {
	Data string
}

func (s *SoundCard) Play(data string) {
	s.Data = data
	fmt.Printf("SoundCard: play %s\n", s.Data)
	GetMediatorInstance().changed(s)
}

type Mediator struct {
	CD    *CDDriver
	CPU   *CPU
	Video *VideoCard
	Sound *SoundCard
}

var mediator *Mediator

func GetMediatorInstance() *Mediator {
	if mediator == nil {
		mediator = &Mediator{}
	}
	return mediator
}

func (m *Mediator) changed(i interface{}) {
	switch inst := i.(type) {
	case *CDDriver:
		m.CPU.Process(inst.Data)
	case *CPU:
		m.Sound.Play(inst.Sound)
		m.Video.Display(inst.Video)
	}
}
```

mediator_test.go

```go
package mediator

import "testing"

func TestMediator(t *testing.T) {
	mediator := GetMediatorInstance()
	mediator.CD = &CDDriver{}
	mediator.CPU = &CPU{}
	mediator.Video = &VideoCard{}
	mediator.Sound = &SoundCard{}

	//Tiggle
	mediator.CD.ReadData()

	if mediator.CD.Data != "music,image" {
		t.Fatalf("CD unexpect data %s", mediator.CD.Data)
	}

	if mediator.CPU.Sound != "music" {
		t.Fatalf("CPU unexpect sound data %s", mediator.CPU.Sound)
	}

	if mediator.CPU.Video != "image" {
		t.Fatalf("CPU unexpect video data %s", mediator.CPU.Video)
	}

	if mediator.Video.Data != "image" {
		t.Fatalf("VidoeCard unexpect data %s", mediator.Video.Data)
	}

	if mediator.Sound.Data != "music" {
		t.Fatalf("SoundCard unexpect data %s", mediator.Sound.Data)
	}
}
```

观察者模式

观察者模式用于触发联动。

一个对象的改变会触发其它观察者的相关动作，而此对象无需关心连动对象的具体实现。

obserser.go

```go
package observer

import "fmt"

type Subject struct {
    observers []Observer
    context   string
}

func NewSubject() *Subject {
    return &Subject{
        observers: make([]Observer, 0),
    }
}

func (s *Subject) Attach(o Observer) {
    s.observers = append(s.observers, o)
}

func (s *Subject) notify() {
    for _, o := range s.observers {
        o.Update(s)
    }
}

func (s *Subject) UpdateContext(context string) {
    s.context = context
    s.notify()
}

type Observer interface {
    Update(*Subject)
}

type Reader struct {
    name string
}

func NewReader(name string) *Reader {
    return &Reader{
        name: name,
    }
}

func (r *Reader) Update(s *Subject) {
    fmt.Printf("%s receive %s\n", r.name, s.context)
}
```

obserser_test.go

```
package observer

func ExampleObserver() {
    subject := NewSubject()
    reader1 := NewReader("reader1")
    reader2 := NewReader("reader2")
    reader3 := NewReader("reader3")
    subject.Attach(reader1)
    subject.Attach(reader2)
    subject.Attach(reader3)

    subject.UpdateContext("observer mode")
    // Output:
    // reader1 receive observer mode
    // reader2 receive observer mode
    // reader3 receive observer mode
}
```

命令模式

命令模式本质是把某个对象的方法调用封装到对象中，方便传递、存储、调用。

示例中把主板单中的启动(start)方法和重启(reboot)方法封装为命令对象，再传递到主机(box)对象中。于两个按钮进行绑定：

- 第一个机箱(box1)设置按钮1(button1) 为开机按钮2(button2)为重启。
- 第二个机箱(box1)设置按钮2(button2) 为开机按钮1(button1)为重启。

从而得到配置灵活性。

除了配置灵活外，使用命令模式还可以用作：

- 批处理
- 任务队列
- undo, redo

等把具体命令封装到对象中使用的场合

command.go

```go
package command

import "fmt"

type Command interface {
    Execute()
}

type StartCommand struct {
    mb *MotherBoard
}

func NewStartCommand(mb *MotherBoard) *StartCommand {
    return &StartCommand{
        mb: mb,
    }
}

func (c *StartCommand) Execute() {
    c.mb.Start()
}
```

```go
type RebootCommand struct {
    mb *MotherBoard
}

func NewRebootCommand(mb *MotherBoard) *RebootCommand {
    return &RebootCommand{
        mb: mb,
    }
}

func (c *RebootCommand) Execute() {
    c.mb.Reboot()
}

type MotherBoard struct{}

func (*MotherBoard) Start() {
    fmt.Print("system starting\n")
}

func (*MotherBoard) Reboot() {
    fmt.Print("system rebooting\n")
}

type Box struct {
    button1 Command
    button2 Command
}

func NewBox(button1, button2 Command) *Box {
    return &Box{
        button1: button1,
        button2: button2,
    }
}

func (b *Box) PressButton1() {
    b.button1.Execute()
}

func (b *Box) PressButton2() {
    b.button2.Execute()
}
```

command_test.go

```go
package command

func ExampleCommand() {
	mb := &MotherBoard{}
	startCommand := NewStartCommand(mb)
	rebootCommand := NewRebootCommand(mb)

	box1 := NewBox(startCommand, rebootCommand)
	box1.PressButton1()
	box1.PressButton2()

	box2 := NewBox(rebootCommand, startCommand)
	box2.PressButton1()
	box2.PressButton2()
	// Output:
	// system starting
	// system rebooting
	// system rebooting
	// system starting
}
```

迭代器模式

送代器模式用于使用相同方式送代不同类型集合或者隐藏集合类型的具体实现。

可以使用送代器模式使遍历同时应用送代策略，如请求新对象、过滤、处理对象等。

iterator.go

```go
package iterator

import "fmt"

type Aggregate interface {
    Iterator() Iterator
}

type Iterator interface {
    First()
    IsDone() bool
    Next() interface{}
}

type Numbers struct {
    start, end int
}

func NewNumbers(start, end int) *Numbers {
    return &Numbers{
        start: start,
        end:   end,
    }
}

func (n *Numbers) Iterator() Iterator {
    return &NumbersIterator{
        numbers: n,
        next:    n.start,
    }
}

type NumbersIterator struct {
    numbers *Numbers
    next    int
}
```

```go
func (i *NumbersIterator) First() {
    i.next = i.numbers.start
}

func (i *NumbersIterator) IsDone() bool {
    return i.next > i.numbers.end
}

func (i *NumbersIterator) Next() interface{} {
    if !i.IsDone() {
        next := i.next
        i.next++
        return next
    }
    return nil
}

func IteratorPrint(i Iterator) {
    for i.First(); !i.IsDone(); {
        c := i.Next()
        fmt.Printf("%#v\n", c)
    }
}
```

iterator_test.go

```go
package iterator

func ExampleIterator() {
    var aggregate Aggregate
    aggregate = NewNumbers(1, 10)

    IteratorPrint(aggregate.Iterator())
    // Output:
    // 1
    // 2
    // 3
    // 4
    // 5
    // 6
    // 7
    // 8
    // 9
    // 10
}
```

模板方法模式

模版方法模式使用继承机制，把通用步骤和通用方法放到父类中，把具体实现延迟到子类中实现。使得实现符合开闭原则。

如实例代码中通用步骤在父类中实现（`准备`、`下载`、`保存`、`收尾`）下载和保存的具体实现留到子类中，并且提供 `保存` 方法的默认实现。

因为Golang不提供继承机制，需要使用匿名组合模拟实现继承。

此处需要注意：因为父类需要调用子类方法，所以子类需要匿名组合父类的同时，父类需要持有子类的引用。

templatemethod.go

```go
package templatemethod

import "fmt"

type Downloader interface {
    Download(uri string)
}

type template struct {
    implement
    uri string
}

type implement interface {
    download()
    save()
}

func newTemplate(impl implement) *template {
    return &template{
        implement: impl,
    }
}

func (t *template) Download(uri string) {
    t.uri = uri
    fmt.Print("prepare downloading\n")
    t.implement.download()
    t.implement.save()
    fmt.Print("finish downloading\n")
}

func (t *template) save() {
    fmt.Print("default save\n")
}

type HTTPDownloader struct {
    *template
}

func NewHTTPDownloader() Downloader {
    downloader := &HTTPDownloader{}
    template := newTemplate(downloader)
    downloader.template = template
    return downloader
}

func (d *HTTPDownloader) download() {
    fmt.Printf("download %s via http\n", d.uri)
}

func (*HTTPDownloader) save() {
    fmt.Printf("http save\n")
}

type FTPDownloader struct {
    *template
}

func NewFTPDownloader() Downloader {
    downloader := &FTPDownloader{}
    template := newTemplate(downloader)
    downloader.template = template
    return downloader
}

func (d *FTPDownloader) download() {
    fmt.Printf("download %s via ftp\n", d.uri)
}
```

templatemethod_test.go

```
package templatemethod
```

```go
func ExampleHTTPDownloader() {
	var downloader Downloader = NewHTTPDownloader()

	downloader.Download("http://example.com/abc.zip")
	// Output:
	// prepare downloading
	// download http://example.com/abc.zip via http
	// http save
	// finish downloading
}

func ExampleFTPDownloader() {
	var downloader Downloader = NewFTPDownloader()

	downloader.Download("ftp://example.com/abc.zip")
	// Output:
	// prepare downloading
	// download ftp://example.com/abc.zip via ftp
	// default save
	// finish downloading
}
```

策略模式

定义一系列算法，让这些算法在运行时可以互换，使得分离算法，符合开闭原则。

strategy.go

```
package strategy

import "fmt"

type Payment struct {
    context  *PaymentContext
    strategy PaymentStrategy
}

type PaymentContext struct {
    Name, CardID string
    Money        int
}

func NewPayment(name, cardid string, money int, strategy PaymentStrategy) *Payme
nt {
    return &Payment{
        context: &PaymentContext{
            Name:   name,
            CardID: cardid,
            Money:  money,
        },
        strategy: strategy,
    }
}

func (p *Payment) Pay() {
    p.strategy.Pay(p.context)
}

type PaymentStrategy interface {
    Pay(*PaymentContext)
}

type Cash struct{}

func (*Cash) Pay(ctx *PaymentContext) {
    fmt.Printf("Pay $%d to %s by cash", ctx.Money, ctx.Name)
}

type Bank struct{}

func (*Bank) Pay(ctx *PaymentContext) {
    fmt.Printf("Pay $%d to %s by bank account %s", ctx.Money, ctx.Name, ctx.Card
ID)

}
```

strategy_test.go

```
package strategy

func ExamplePayByCash() {
    payment := NewPayment("Ada", "", 123, &Cash{})
    payment.Pay()
    // Output:
    // Pay $123 to Ada by cash
}

func ExamplePayByBank() {
    payment := NewPayment("Bob", "0002", 888, &Bank{})
    payment.Pay()
    // Output:
    // Pay $888 to Bob by bank account 0002
}
```

状态模式

状态模式用于分离状态和行为。

state.go

```
package state

import "fmt"

type Week interface {
    Today()
    Next(*DayContext)
}

type DayContext struct {
    today Week
}

func NewDayContext() *DayContext {
    return &DayContext{
        today: &Sunday{},
    }
}

func (d *DayContext) Today() {
    d.today.Today()
}

func (d *DayContext) Next() {
    d.today.Next(d)
}

type Sunday struct{}

func (*Sunday) Today() {
    fmt.Printf("Sunday\n")
}

func (*Sunday) Next(ctx *DayContext) {
    ctx.today = &Monday{}
}

type Monday struct{}
```

```
func (*Monday) Today() {
    fmt.Printf("Monday\n")
}

func (*Monday) Next(ctx *DayContext) {
    ctx.today = &Tuesday{}
}

type Tuesday struct{}

func (*Tuesday) Today() {
    fmt.Printf("Tuesday\n")
}

func (*Tuesday) Next(ctx *DayContext) {
    ctx.today = &Wednesday{}
}

type Wednesday struct{}

func (*Wednesday) Today() {
    fmt.Printf("Wednesday\n")
}

func (*Wednesday) Next(ctx *DayContext) {
    ctx.today = &Thursday{}
}

type Thursday struct{}

func (*Thursday) Today() {
    fmt.Printf("Thursday\n")
}

func (*Thursday) Next(ctx *DayContext) {
    ctx.today = &Friday{}
}

type Friday struct{}

func (*Friday) Today() {
    fmt.Printf("Friday\n")
}

func (*Friday) Next(ctx *DayContext) {
```

```go
    ctx.today = &Saturday{}
}

type Saturday struct{}

func (*Saturday) Today() {
    fmt.Printf("Saturday\n")
}

func (*Saturday) Next(ctx *DayContext) {
    ctx.today = &Sunday{}
}
```

state_test.go

```go
package state

func ExampleWeek() {
    ctx := NewDayContext()
    todayAndNext := func() {
        ctx.Today()
        ctx.Next()
    }

    for i := 0; i < 8; i++ {
        todayAndNext()
    }
    // Output:
    // Sunday
    // Monday
    // Tuesday
    // Wednesday
    // Thursday
    // Friday
    // Saturday
    // Sunday
}
```

备忘录模式

备忘录模式用于保存程序内部状态到外部，又不希望暴露内部状态的情形。

程序内部状态使用窄接口船体给外部进行存储，从而不暴露程序实现细节。

备忘录模式同时可以离线保存内部状态，如保存到数据库，文件等。

memento.go

```go
package memento

import "fmt"

type Memento interface{}

type Game struct {
    hp, mp int
}

type gameMemento struct {
    hp, mp int
}

func (g *Game) Play(mpDelta, hpDelta int) {
    g.mp += mpDelta
    g.hp += hpDelta
}

func (g *Game) Save() Memento {
    return &gameMemento{
        hp: g.hp,
        mp: g.mp,
    }
}

func (g *Game) Load(m Memento) {
    gm := m.(*gameMemento)
    g.mp = gm.mp
    g.hp = gm.hp
}

func (g *Game) Status() {
    fmt.Printf("Current HP:%d, MP:%d\n", g.hp, g.mp)
}
```

memento_test.go

```
package memento

func ExampleGame() {
    game := &Game{
        hp: 10,
        mp: 10,
    }

    game.Status()
    progress := game.Save()

    game.Play(-2, -3)
    game.Status()

    game.Load(progress)
    game.Status()

    // Output:
    // Current HP:10, MP:10
    // Current HP:7, MP:8
    // Current HP:10, MP:10
}
```

解释器模式

解释器模式定义一套语言文法，并设计该语言解释器，使用户能使用特定文法控制解释器行为。

解释器模式的意义在于，它分离多种复杂功能的实现，每个功能只需关注自身的解释。

对于调用者不用关心内部的解释器的工作，只需要用简单的方式组合命令就可以。

interpreter.go

```go
package interpreter

import (
    "strconv"
    "strings"
)

type Node interface {
    Interpret() int
}

type ValNode struct {
    val int
}

func (n *ValNode) Interpret() int {
    return n.val
}

type AddNode struct {
    left, right Node
}

func (n *AddNode) Interpret() int {
    return n.left.Interpret() + n.right.Interpret()
}

type MinNode struct {
    left, right Node
}

func (n *MinNode) Interpret() int {
    return n.left.Interpret() - n.right.Interpret()
```

```
}

type Parser struct {
    exp   []string
    index int
    prev  Node
}

func (p *Parser) Parse(exp string) {
    p.exp = strings.Split(exp, " ")

    for {
        if p.index >= len(p.exp) {
            return
        }
        switch p.exp[p.index] {
        case "+":
            p.prev = p.newAddNode()
        case "-":
            p.prev = p.newMinNode()
        default:
            p.prev = p.newValNode()
        }
    }
}

func (p *Parser) newAddNode() Node {
    p.index++
    return &AddNode{
        left:  p.prev,
        right: p.newValNode(),
    }
}

func (p *Parser) newMinNode() Node {
    p.index++
    return &MinNode{
        left:  p.prev,
        right: p.newValNode(),
    }
}

func (p *Parser) newValNode() Node {
    v, _ := strconv.Atoi(p.exp[p.index])
    p.index++
    return &ValNode{
```

```go
		val: v,
	}
}

func (p *Parser) Result() Node {
	return p.prev
}
```

interpreter_test.go

```go
package interpreter

import "testing"

func TestInterpreter(t *testing.T) {
	p := &Parser{}
	p.Parse("1 + 2 + 3 - 4 + 5 - 6")
	res := p.Result().Interpret()
	expect := 1
	if res != expect {
		t.Fatalf("expect %d got %d", expect, res)
	}
}
```

职责链模式

职责链模式用于分离不同职责，并且动态组合相关职责。

Golang实现职责链模式时候，因为没有继承的支持，使用链对象包涵职责的方式，即：

- 链对象包含当前职责对象以及下一个职责链。
- 职责对象提供接口表示是否能处理对应请求。
- 职责对象提供处理函数处理相关职责。

同时可在职责链类中实现职责接口相关函数，使职责链对象可以当做一般职责对象是用。

chain.go

```go
package chain

import "fmt"

type Manager interface {
    HaveRight(money int) bool
    HandleFeeRequest(name string, money int) bool
}

type RequestChain struct {
    Manager
    successor *RequestChain
}

func (r *RequestChain) SetSuccessor(m *RequestChain) {
    r.successor = m
}

func (r *RequestChain) HandleFeeRequest(name string, money int) bool {
    if r.Manager.HaveRight(money) {
        return r.Manager.HandleFeeRequest(name, money)
    }
    if r.successor != nil {
        return r.successor.HandleFeeRequest(name, money)
    }
    return false
}

func (r *RequestChain) HaveRight(money int) bool {
    return true
}

type ProjectManager struct{}

func NewProjectManagerChain() *RequestChain {
    return &RequestChain{
        Manager: &ProjectManager{},
    }
}

func (*ProjectManager) HaveRight(money int) bool {
    return money < 500
}

func (*ProjectManager) HandleFeeRequest(name string, money int) bool {
    if name == "bob" {
        fmt.Printf("Project manager permit %s %d fee request\n", name, money)
        return true
    }
    fmt.Printf("Project manager don't permit %s %d fee request\n", name, money)
    return false
}

type DepManager struct{}

func NewDepManagerChain() *RequestChain {
    return &RequestChain{
        Manager: &DepManager{},
    }
}

func (*DepManager) HaveRight(money int) bool {
    return money < 5000
}

func (*DepManager) HandleFeeRequest(name string, money int) bool {
    if name == "tom" {
        fmt.Printf("Dep manager permit %s %d fee request\n", name, money)
        return true
    }
    fmt.Printf("Dep manager don't permit %s %d fee request\n", name, money)
    return false
}

type GeneralManager struct{}
```

```go
func NewGeneralManagerChain() *RequestChain {
    return &RequestChain{
        Manager: &GeneralManager{},
    }
}

func (*GeneralManager) HaveRight(money int) bool {
    return true
}

func (*GeneralManager) HandleFeeRequest(name string, money int) bool {
    if name == "ada" {
        fmt.Printf("General manager permit %s %d fee request\n", name, money)
        return true
    }
    fmt.Printf("General manager don't permit %s %d fee request\n", name, money)
    return false
}
```

chain_test.go

```go
package chain

func ExampleChain() {
    c1 := NewProjectManagerChain()
    c2 := NewDepManagerChain()
    c3 := NewGeneralManagerChain()

    c1.SetSuccessor(c2)
    c2.SetSuccessor(c3)

    var c Manager = c1

    c.HandleFeeRequest("bob", 400)
    c.HandleFeeRequest("tom", 1400)
    c.HandleFeeRequest("ada", 10000)
    c.HandleFeeRequest("floar", 400)
    // Output:
    // Project manager permit bob 400 fee request
    // Dep manager permit tom 1400 fee request
    // General manager permit ada 10000 fee request
    // Project manager don't permit floar 400 fee request

}
```

访问者模式

访问者模式可以给一系列对象透明的添加功能，并且把相关代码封装到一个类中。

对象只要预留访问者接口 `Accept` 则后期为对象添加功能的时候就不需要改动对象。

visitor.go

```
package visitor

import "fmt"

type Customer interface {
	Accept(Visitor)
}

type Visitor interface {
	Visit(Customer)
}

type EnterpriseCustomer struct {
	name string
}

type CustomerCol struct {
	customers []Customer
}

func (c *CustomerCol) Add(customer Customer) {
	c.customers = append(c.customers, customer)
}

func (c *CustomerCol) Accept(visitor Visitor) {
	for _, customer := range c.customers {
		customer.Accept(visitor)
	}
}

func NewEnterpriseCustomer(name string) *EnterpriseCustomer {
	return &EnterpriseCustomer{
		name: name,
	}
}
```

```go
func (c *EnterpriseCustomer) Accept(visitor Visitor) {
	visitor.Visit(c)
}

type IndividualCustomer struct {
	name string
}

func NewIndividualCustomer(name string) *IndividualCustomer {
	return &IndividualCustomer{
		name: name,
	}
}

func (c *IndividualCustomer) Accept(visitor Visitor) {
	visitor.Visit(c)
}

type ServiceRequestVisitor struct{}

func (*ServiceRequestVisitor) Visit(customer Customer) {
	switch c := customer.(type) {
	case *EnterpriseCustomer:
		fmt.Printf("serving enterprise customer %s\n", c.name)
	case *IndividualCustomer:
		fmt.Printf("serving individual customer %s\n", c.name)
	}
}

// only for enterprise
type AnalysisVisitor struct{}

func (*AnalysisVisitor) Visit(customer Customer) {
	switch c := customer.(type) {
	case *EnterpriseCustomer:
		fmt.Printf("analysis enterprise customer %s\n", c.name)
	}
}
```

visitor_test.go

```go
package visitor

func ExampleRequestVisitor() {
	c := &CustomerCol{}
```

```
    c.Add(NewEnterpriseCustomer("A company"))
    c.Add(NewEnterpriseCustomer("B company"))
    c.Add(NewIndividualCustomer("bob"))
    c.Accept(&ServiceRequestVisitor{})
    // Output:
    // serving enterprise customer A company
    // serving enterprise customer B company
    // serving individual customer bob
}

func ExampleAnalysis() {
    c := &CustomerCol{}
    c.Add(NewEnterpriseCustomer("A company"))
    c.Add(NewIndividualCustomer("bob"))
    c.Add(NewEnterpriseCustomer("B company"))
    c.Accept(&AnalysisVisitor{})
    // Output:
    // analysis enterprise customer A company
    // analysis enterprise customer B company
}
```