

单例模式（懒汉模式）：

```
var m *single
type single struct{
    Name string
}
func GetInstance() *single{
    if m == nil{
        m = &single{}
    }
    return m
}
```

缺点：该方法在并发下容易出错，非线性安全。

单例模式（加锁）：

```
var m *single
var lock sync.Mutex
type single struct{
    Name string
}
func GetInstance() *single{
    lock.Lock()
    defer lock.Unlock()
    if m == nil{
        m = &single{}
    }
    return m
}
```

缺点：在高并发环境下，现在不管什么情况下都会上一把锁，而且加锁的代价是很大的

单例模式（双重判断）：

```
var m *single
var lock sync.Mutex
type single struct{
    Name string
}
func GetInstance() *single{
    if m == nil{
        lock.Lock()
        defer lock.Unlock()
        if m == nil{
            m = &single{}
        }
    }
    return m
}
```

这次我们用了两个判断，而且我们将同步锁放在了条件判断之后，这样做就避免了每次调用都加锁，提高了代码的执行效率

单例模式 (sync包里的Once.Do()方法) :

```
var m *single
var once sync.Once

type single struct{
    Name string
}

func GetInstance() *single{
    once.Do(func() {
        m = &single{}
    })
    return m
}
```

Once.Do方法的参数是一个函数，这里我们给的是一个匿名函数，在这个函数中我们做的工作很简单，就是去赋值m变量，而且go能保证这个函数中的代码仅仅执行一次！

创建者模式 或者 (建造者模式)

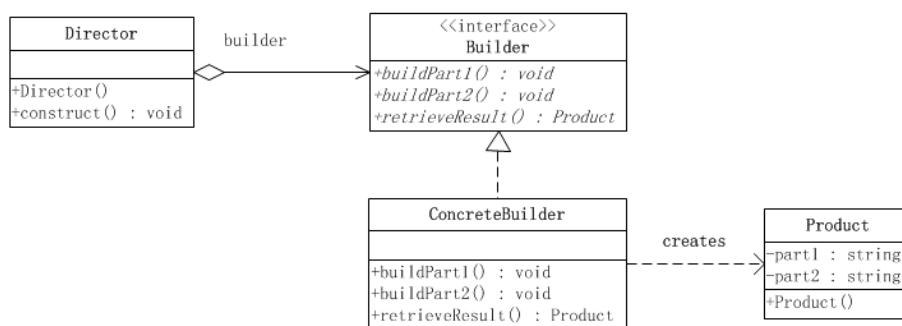
建造者模式中有一下几个角色需要我们熟悉：

Product 这是我们要创建的复杂对象(一般都是很复杂的对象才需要使用建造者模式)。

Builder 抽象的一个东西，主要是用来规范我们的建造者的。

ConcreteBuilder 具体的Builder实现，这是今天重点，主要用来根据不同的业务来完成要创建对象的组建的创建。

Director 这个的作用是规范复杂对象的创建流程。



```
// designBulider project main.go
```

```
//用游戏中的角色创建事例来展示创建者模式的实现，  
//这个游戏角色很简单，仅仅包含名称和使用的武器两个字段。  
package main
```

```
import (
    "fmt"
)

//定义游戏人物类和其方法(即产品)
type Character struct {
    Name string
    Arms string
}

func (p *Character) SetName(name string) {
    p.Name = name
}

func (p *Character) SetArms(arms string) {
    p.Arms = arms
}

func (p Character) GetName() string {
    return p.Name
}

func (p Character) GetArms() string {
    return p.Arms
}

//声明抽象建造者接口

type Builder interface {
    SetName(name string) Builder
    SetArms(arms string) Builder
    Build() *Character
}

//声明具体的建造者,要实现Builder接口

type CharacterBuilder struct {
    character *Character
}

func (p *CharacterBuilder) SetName(name string) Builder {
    if p.character == nil {

```

```

    p.character = &Character{}
}

p.characterSetName(name)
return p
}

func (p *CharacterBuilder) SetArms(arms string) Builder {
    if p.character == nil {
        p.character = &Character{}
    }
    p.character.SetArms(arms)
    return p
}

func (p *CharacterBuilder) Build() *Character {
    return p.character
}

//Director的实现,这个的作用是规范复杂对象的创建流程。例如先取名再配备武器
type Director struct {
    builder Builder
}

func (p Director) Create(name string, arms string) *Character {
    return p.builder.SetName(name).SetArms(arms).Build()
}

func main() {
    var builer Builder = &CharacterBuilder{}
    var director *Director = &Director{builder: builer}
    var ch *Character = director.Create("Ibb", "AK47")
    fmt.Println(ch.GetName() + "," + ch.GetArms())
}
*****

```

对象池模式：

初始化一个slice，初学者会用make([]int64, 0)

但是这样后续如果要append元素时，每次追加都需要进行扩容，参考下面的代码，追加了2个元素，slice 扩容了2次。

```
a := make([]int64, 0)
```

```
fmt.Println(cap(a), len(a))

for i := 0; i < 3; i++ {
    a = append(a, 1)
    fmt.Println(cap(a), len(a))
}
```

```
0 0
1 1
2 2
```

上面每一次扩容空间，都是会重新申请一块区域，把旧空间里面的元素复制进来，把新的追加进来。那旧空间里面的元素怎么办？等着垃圾回收呗。

程序在从操作系统申请一块内存时，这块内存会被分成堆和栈。栈可以简单得理解成一次函数调用内部申请到的内存，它们会随着函数的返回把内存还给系统。

```
func F() {
    temp := make([]int, 0, 20)
    ...
}
```

类似于上面代码里面的temp变量，只是内函数内部申请的临时变量，并不会作为返回值返回，它就是被编译器申请到栈里面。申请到栈内存好处：函数返回直接释放，不会引起垃圾回收，对性能没有影响。

```
func F() []int{
    a := make([]int, 0, 20)
    return a
}
```

而对于上面一段程序，因临时变量最终被返回，编译器会认为变量之后还会被使用，当函数返回之后并不会将其内存归还，那么它就会被申请到堆上面了。**申请到堆上面的内存才会引起垃圾回收。**

```
func F() {
    a := make([]int, 0, 20)
    b := make([]int, 0, 20000)

    l := 20
    c := make([]int, 0, l)
}
```

对上上述代码
a: 申请到栈上面

b:由于申请的内存较大，编译器会把这种申请内存较大的变量转移到堆上面。即使是临时变量，申请过大也会在堆上面申请。

d:编译器对于这种不定长度的申请方式，也会在堆上面申请，即使申请的长度很短。

(内存碎片化：简单得说，就是程序要从操作系统申请一块比较大的内存，内存分成小块，通过链表链接。每次程序申请内存，就从链表上面遍历每一小块，找到符合的就返回其地址，没有合适的就从操作系统再申请。如果申请内存次数较多，而且申请的大小不固定，就会引起内存碎片化的问题。申请的堆内存并没有用完，但是用户申请的内存的时候却没有合适的空间提供。这样会遍历整个链表，还会继续向操作系统申请内存。)

为了减少GX压力和减少不必要的内存分配，Golang在sync里面提供了对象池Pool。

```
// designBulider project main.go
//用游戏中的角色创建事例来展示创建者模式的实现，
//这个游戏角色很简单，仅仅包含名称和使用的武器两个字段。
package main

import (
    "fmt"
    "sync"
)

var bufPool sync.Pool

type buf struct {
    b []byte
}

func main() {

    for i := 0; i < 10; i++ {
        var bf *buf
        //从池子中去数据
        v := bufPool.Get()
        if v == nil {
            //若不存在buf，创建新的
            fmt.Println("no buf ,create!")
            bf = &buf{
                b: make([]byte, 10),
            }
        } else {
            bf = v.(*buf)
        }
        bf.b[i] = byte(i)
        fmt.Println(string(bf.b))
    }
}
```

```

    } else {
        // 池里存在buf,v这里是interface{}，需要做类型转换
        bf = v.(*buf)
    }
    fmt.Println("使用完毕", bf)
    bufPool.Put(bf)
}

}
*****

```

工厂模式：

简单工厂的实现思想，即创建一个工厂，将产品的实现逻辑集中在这个工厂中。

//定义food接口和实现类

```

type Food interface {
    Eat()
}

type Meat struct {}

type Hamberger struct {}

func (m Meat) Eat() {
    fmt.Println("Eat meat.")
}

func (h Hamberger) Eat() {
    fmt.Println("Eat Hamberger.")
}

```

//定义go

```

type FoodFactory struct {}

func (ff FoodFactory) CreateFood(name string) Food {
    var s Food;
    switch name {
        case "Meat":
            s = new(Meat);
        case "Hamberger":
            s = new(Hamberger)
    }
}

```

```
    }
    return s;
}
```

简单工厂存在的问题：

1. 当新增一种产品的时候，需要修改工厂逻辑
2. 简单工厂没有继承的结构
3. 一旦工厂瘫痪，整个系统都瘫痪

工厂方法:将产品的创建逻辑写在子类之中

```
type Factory interface {
    Create() Food
}

type MeatFactory struct {
}

func (mf MeatFactory) Create() Food {
    m := Meat{}
    return m
}

type HambergerFactory struct{
}

func (hf HambergerFactory) Create() Food {
    h := Hamberger{}
    return h
}
```

抽象工厂：

抽象工厂是针对一个产品族而言的.产品族就好像套餐.一个套餐包含了好几种食品，而每一种食品都是一种类型的食物。比如：

- 套餐A:肉和CoCo
- 套餐B:汉堡和茶

把套餐A外包给工厂A负责生产，套餐B外包给工厂B负责生产

//定义一个抽象的工厂及其实现类

```

type HJXFactory interface{
    CreateFood()Food
    CreateDrink()Drink
}

type FactoryA struct {

}

func (af FactoryA)CreateFood()Food{
    f := Meat{}
    return f
}

func (af FactoryA)CreateDrink()Drink{
    d := CoCo{}
    return d
}

type FactoryB struct {

}

func (bf FactoryB)CreateFood()Food{
    f := Hamberger{}
    return f
}

func (bf FactoryB)CreateDrink()Drink{
    d := Tea{}
    return d
}

```

享元模式：

实现对象的共享，即共享池，类型与对象池模式，当系统中对象多的时候可以减少内存的开销，通常与工厂模式一起使用。常见的即数据库连接池和 sync.Pool

```
package main
```

```

import (
    "fmt"
    "time"

```

```
)
```

```
// 数据库连接
type DbConnect struct {
}

func (*DbConnect) Do(name string) {
    fmt.Println("connect.....doing...." + name)
}
```

```
//数据库连接池
type DbConnnectPool struct {
    //使用一个DbConnect指针类型的通道作为容器
    ConnChan chan *DbConnect
}
```

```
func NewDbConnnectPool(chanLen int) *DbConnnectPool {
    return &DbConnnectPool{
        ConnChan: make(chan *DbConnect, chanLen),
    }
}
```

```
//定义数据库连接池上的get方法
func (dc *DbConnnectPool) Get() *DbConnect {
    select {
        case conn := <-dc.ConnChan:
            return conn
        default:
            //无则创建
            fmt.Println("creat a dbconnect")
            return new(DbConnect)
    }
}
```

```
//定义数据库连接池上的put方法
func (dc *DbConnnectPool) Put(conn *DbConnect) {
    select {
        case dc.ConnChan <- conn:
            return
        default:
            return
    }
}
```

```

    }

}

func main() {
    pool := NewDbConnectPool(3)
    conn := pool.Get()

    conn.Do("conn")
    //使用完将连接放置到池子里
    pool.Put(conn)

    time.Sleep(time.Second * 2)

    conn1 := pool.Get()
    conn1.Do("conn1")
    //使用完将连接放置到池子里
    pool.Put(conn1)

}
*****
```

代理模式：

为其他对象提供一种代理以控制对这个对象的访问

适配器模式的区别：适配器模式主要改变所考虑对象的接口，
而代理模式不能改变所代理类的接口。
和装饰器模式的区别：装饰器模式为了增强功能，而代理模式
是为了加以控制。

```

package main

import (
    "errors"
    "fmt"
)

//声明读写设备接口
type Device interface {
    Read() ([]byte, error)
    Write(word []byte) error
}
```

```
//定义硬盘及其方法
type HardDisk struct {
    storage []byte
}

func (h *HardDisk) Read() ([]byte, error) {
    return h.storage, nil
}

func (h *HardDisk) Write(word []byte) error {
    h.storage = word
    return nil
}

//定义代理类及其方法,通过代理类控制器访问权限
type HardDiskPorxy struct {
    Opid string
    hd   *HardDisk
}

func (h *HardDiskPorxy) Read() ([]byte, error) {
    if !h.permission("read") {
        return nil, errors.New("你没有权限read")
    }
    return h.hd.storage, nil
}

func (h *HardDiskPorxy) Write(word []byte) error {
    if !h.permission("write") {
        return errors.New("你没有权限write")
    }
    return h.hd.Write(word)
}

//权限判断
func (h *HardDiskPorxy) permission(tag string) bool {
    if h.Opid == "admin" {
        return true
    }
    if h.Opid == "reader" && tag == "read" {
        return true
    }
}
```

```

if h.Opid == "writer" && tag == "write" {
    return true
}
return false
}

func main() {
    var dev Device
    dev = &HardDiskPorxy{Opid: "admin", hd: &HardDisk{}}
    dev.Write([]byte("Ibb wyt"))
    data, _ := dev.Read()
    fmt.Println(string(data))

    dev = &HardDiskPorxy{Opid: "reader", hd: &HardDisk{storage: []byte("only read")}}
    err := dev.Write([]byte("Hello world!"))
    fmt.Println(err.Error())
    data, _ = dev.Read()
    fmt.Println(string(data))
}
*****

```

适配器模式

将一个类的接口转换成客户希望的另外一个接口 或者作为两个不兼容的接口之间的桥梁

```
package main
```

```

import (
    "fmt"
)

//不可充电池使用接口及其实现类
type NonRechargeableBattery interface {
    Use()
}

//不可充电池A
type NonRechargeableA struct {
}

func (NonRechargeableA) Use() {
    fmt.Println("NonRechargeableA using ")
}
```

```
//可充电电池使用接口
type RechargeableBattery interface {
    Use()
    Charge()
}

//适配可充电电池使用接口
type AdapterNonToYes struct {
    NonRechargeableBattery
}

func (AdapterNonToYes) Charge() {
    fmt.Println("AdapterNonToYes Charging")
    //nothing to do, just adapt for RechargeableBattery's interface
}
```

```
func main() {
    var battary RechargeableBattery
    battary = &AdapterNonToYes{
        &NonRechargeableA{},
    }
    battary.Use()
    battary.Charge()
}
```

```
*****

```

```
package main
```

```
import (
    "fmt"
    "net/http"
)
```

```
//声明请求接口及其实现类
type Request interface {
    HttpRequest() (*http.Request, error)
}
```

```
type CdnRequest struct {
}
```

```

func (cdn *CdnRequest) HttpRequest() (*http.Request, error) {
    return http.NewRequest("GET", "/cdn", nil)
}

type LiveRequest struct {

}

func (cdn *LiveRequest) HttpRequest() (*http.Request, error) {
    return http.NewRequest("GET", "/live", nil)
}

//声明一个客户端及其方法

type Client struct {
    Client *http.Client
}

func (c *Client) Query(req Request) (response *http.Response, err error) {
    httpreq, _ := req.HttpRequest()
    response, err = c.Client.Do(httpreq)
    return
}

func main() {
    //客户端负责发送请求，他无需关注请求的类型
    client := &Client{http.DefaultClient}
    cndRequest := &CdnRequest{}
    fmt.Println(client.Query(cndRequest))

    liveReq := &LiveRequest{}
    fmt.Println(client.Query(liveReq))
}
*****

```

责任链模式：

责任链模式使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。
将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。

```

package main

import (

```

```
"fmt"
)

//要处理的时间
type ScreenEvent struct {
    Type string
    Comment string
}

//事件处理器接口及其实现类
type IScreenEventHandler interface {
    Handle(*ScreenEvent) bool
    SetNextHandler(IScreenEventHandler)
}

//实现类(接受事件但并不处理,作为入口)
type AbsScreenEventHandler struct {
    NextHandler IScreenEventHandler
}

func (ase *AbsScreenEventHandler) Handle(se *ScreenEvent) bool {
    if ase.NextHandler != nil {
        return ase.NextHandler.Handle(se)
    }
    return false
}

func (ase *AbsScreenEventHandler) SetNextHandler(is IScreenEventHandler) {
    ase.NextHandler = is
}

//具体的时件处理器
type HomeScreenEventHandler struct {
    NextHandler IScreenEventHandler
}

func (hse *HomeScreenEventHandler) Handle(se *ScreenEvent) bool {
    fmt.Println("HomeScreenEventHandler.....")
    if se.Type == "HomeClick" {
        fmt.Println("HomeClick")
        return true
    }
}
```

```

}

return hse.NextHandler.Handle(se)
}

func (hse *HomeScreenEventHandler) SetNextHandler(is IScreenEventHandler) {
    hse.NextHandler = is
}

//具体的事件处理器
type UserScreenEventHandler struct {
    NextHandler IScreenEventHandler
}

func (use *UserScreenEventHandler) Handle(se *ScreenEvent) bool {
    fmt.Println("UserScreenEventHandler.....")
    if se.Type == " UserModelClick" {
        fmt.Println(" UserModelClick")
        return true
    }
    return use.NextHandler.Handle(se)
}

func (use *UserScreenEventHandler) SetNextHandler(is IScreenEventHandler) {
    use.NextHandler = is
}

func main() {
    var osd IScreenEventHandler
    osd = &AbsScreenEventHandler{}
    home := &HomeScreenEventHandler{}
    user := &UserScreenEventHandler{}
    home.SetNextHandler(user)
    osd.SetNextHandler(home)

    screenEvent := &ScreenEvent{Type: " UserModelClick"}
    osd.Handle(screenEvent)

    fmt.Println("-----\n")
    screenEvent = &ScreenEvent{Type: " Null"}
    osd.Handle(screenEvent)
}

```

```
}
```

以上程序存在的问题是，`screenEvent = &ScreenEvent{Type: "Null"}` 这里因为传的类型为null就不会有执行者去执行，传到最后一个时就会报错。

```
package main
```

```
import (  
    "fmt"  
)
```

```
//要处理的时间
```

```
type ScreenEvent struct {  
    Type string  
    Comment string  
}
```

```
//事件处理器接口及其实现类
```

```
type IScreenEventHandler interface {  
    Handle(*ScreenEvent) bool  
    SetNextHandler(IScreenEventHandler)  
}
```

```
//实现类(父类，接受事件但并不处理,作为入口)
```

```
type AbsScreenEventHandler struct {  
    NextHandler IScreenEventHandler  
}
```

```
func (ase *AbsScreenEventHandler) Handle(se *ScreenEvent) bool {  
    if ase.NextHandler != nil {  
        return ase.NextHandler.Handle(se)  
    }  
    return false  
}
```

```
func (ase *AbsScreenEventHandler) SetNextHandler(is IScreenEventHandler) {  
    ase.NextHandler = is  
}
```

```
//具体的时件处理器
```

```
type HomeScreenEventHandler struct {
    //定义HomeScreenEventHandler 为AbsScreenEventHandler的子类
    AbsScreenEventHandler
}

func (hse *HomeScreenEventHandler) Handle(se *ScreenEvent) bool {
    fmt.Println("HomeScreenEventHandler.....")
    if se.Type == "HomeClick" {
        fmt.Println("HomeClick")
        return true
    }
    //调用父类的方法将时间传递给下一位接受者
    return hse.AbsScreenEventHandler.Handle(se)
}

//func (hse *HomeScreenEventHandler) SetNextHandler(is IScreenEventHandler) {
//    hse.NextHandler = is
//}

//具体的事件处理器
type UserScreenEventHandler struct {
    //定义UserScreenEventHandler 为AbsScreenEventHandler的子类
    AbsScreenEventHandler
}

func (use *UserScreenEventHandler) Handle(se *ScreenEvent) bool {
    fmt.Println("UserScreenEventHandler.....")
    if se.Type == "UserModelClick" {
        fmt.Println("UserModelClick")
        return true
    }
    //调用父类的方法将时间传递给下一位接受者
    return use.AbsScreenEventHandler.Handle(se)
}

//func (use *UserScreenEventHandler) SetNextHandler(is IScreenEventHandler) {
//    use.NextHandler = is
//}

func main() {
    var osd IScreenEventHandler
```

```

osd = &AbsScreenEventHandler{}
home := &HomeScreenEventHandler{}
user := &UserScreenEventHandler{}
home.SetNextHandler(user)
osd.SetNextHandler(home)

screenEvent := &ScreenEvent{Type: "UserModelClick"}
osd.Handle(screenEvent)

fmt.Println("-----\n")
screenEvent = &ScreenEvent{Type: "Null"}
osd.Handle(screenEvent)

}
*****

```

命令模式：

```

package main

import (
    "errors"
    "fmt"
)

// 命令模式 ( Command Pattern ) 是一种数据驱动的设计模式，它属于行为型模式。
// 请求以命令的形式包裹在对象中，并传给调用对象。调用对象寻找可以处理该命令
// 的合适的对象，并把该命令传给相应的对象，该对象执行命令。

//命令接口及其实现类
type Command interface {
    Do(args interface{}) (interface{}, error)
}

//get命令
type GetCommand struct {
}

func (gc *GetCommand) Do(args interface{}) (interface{}, error) {
    fmt.Println("get 命令")
    return args, nil
}

```

```
type PostCommand struct {
}

func (gc *PostCommand) Do(args interface{}) (interface{}, error) {
    fmt.Println("post 命令")
    return args, nil
}

//上下文，用于封装命令
type CmdContext struct {
    ComType string    //用于识别命令
    Args   interface{} //命令参数
}

//命令管理者。用于调用命令
type CommandHandler struct {
    //调用者维护一个注册表
    CmdMap map[string]Command
}

// 注册命令
func (ch *CommandHandler) Register(cmdType string, cmd Command) {
    ch.CmdMap[cmdType] = cmd
}

//处理命令
func (ch *CommandHandler) Handle(ctex *CmdContext) (interface{}, error) {
    if ctex == nil {
        return nil, errors.New("上下文为空")
    }
    //确认该命令是否已被注册
    cmd, ok := ch.CmdMap[ctex.ComType]
    if ok {
        return cmd.Do(ctex.Args)
    }

    return nil, errors.New("无效的参数")
}

func main() {
```

```

cmdHandler := &CommandHandler{CmdMap: make(map[string]Command)}
postCtx := &CmdContext{ComType: "post", Args: "post args"}
cmdHandler.Register("post", &PostCommand{})
result, error := cmdHandler.Handle(postCtx)
if error == nil {
    fmt.Println(result)
}

nullCtx := &CmdContext{ComType: "null", Args: " Get"}
fmt.Println(cmdHandler.Handle(nullCtx))
}
*****

```

装饰模式：

```

package main

import (
    "fmt"
    "log"
    "strings"
)

// 装饰模式
// 动态地给一个对象添加一些额外的职责，同时又不改变其结构
//

// 接口
type MessageBuilder interface {
    Build(messages ...string) string
}

// 基本信息构造器
type BaseMessageBuilder struct {
}

func (b *BaseMessageBuilder) Build(messages ...string) string {
    return strings.Join(messages, ",")
}

// 引号装饰器

```

```
type QuoteMessageBuilderDecorator struct {
    Builder MessageBuilder
}

func (q *QuoteMessageBuilderDecorator) Build(messages ...string) string {
    return `"` + q.Builder.Build(messages...) + `"`
}

// 大括号装饰器
type BraceMessageBuilderDecorator struct {
    Builder MessageBuilder
}

func (b *BraceMessageBuilderDecorator) Build(messages ...string) string {
    return `{"` + b.Builder.Build(messages...) + `"}`
}

// 或者

type Object func(int) int

func LogDecorate(fn Object) Object {
    return func(n int) int {
        log.Println("Starting the execution with the integer", n)

        result := fn(n)

        log.Println("Execution is completed with the result", result)

        return result
    }
}

func Double(n int) int {
    return n * 2
}

func main() {
    var MB MessageBuilder

    MB = &BaseMessageBuilder{}
```

```
fmt.Println(MB.Build("hello world"))

MB = &QuoteMessageBuilderDecorator{MB}
```

```
fmt.Println(MB.Build("hello world"))

MB = &BraceMessageBuilderDecorator{MB}
```

```
fmt.Println(MB.Build("hello world"))
```

```
//
f := LogDecorate(Double)
f(5)
}

*****
```

迭代器模式，这种模式用于顺序访问集合的元素，无需关心集合对象的底层表示。

```
package main
```

```
import (
    "fmt"
)
```

```
//迭代器接口及其实现
```

```
type Iterator interface {
    HasNext() bool
    Next() interface{}
}
```

```
type ArrayIterator struct {
    currentIndex int
    ac           *ArrayContainer
}
```

```
func (ai *ArrayIterator) HasNext() bool {
    if ai.ac.arrayData != nil && ai.currentIndex < len(ai.ac.arrayData) {
        return true
    }
    return false
}
```

```
func (ai *ArrayIterator) Next() interface{} {
    if ai.hasNext() {
        defer func() {
            ai.currentIndex++
        }()
        return ai.ac.arrayData[ai.currentIndex]
    }
    return nil
}
```

//集合接口及其实现

```
type Container interface {
    GetIterator() Iterator
}

type ArrayContainer struct {
    arrayData []interface{}
}

func (ac *ArrayContainer) GetIterator() Iterator {
    return &ArrayIterator{
        currentIndex: 0,
        ac:           ac,
    }
}

func main() {
    arr := []interface{}{"a", "b", "c", "d"}
    arrContianer := &ArrayContainer{
        arrayData: arr,
    }
    iterator := arrContianer.GetIterator()
    for iterator.hasNext() {
        fmt.Println(iterator.Next().(string))
    }
}
*****
```

中介者模式：

```
package main

import (
    "fmt"
)

// 中介者模式 ( Mediator Pattern )
//   是用来降低多个对象和类之间的通信复杂性。
// 这种模式提供了一个中介类，该类通常处理不同类之间的通信，并支持松耦合，使代码易于维护
//中介类，
type ChatRoom struct {
    name string
}

func (cr *ChatRoom) SendMsg(msg string) {
    fmt.Println(cr.name + " :" + msg)
}

func (cr *ChatRoom) RegisterUser(u *User) {
    u.cr = cr
}

type User struct {
    name string
    //每一个用户都有一个指定的聊天室
    cr *ChatRoom
}

func (u *User) SendMsg(msg string) {
    if u.cr != nil {
        u.cr.SendMsg(u.name + ":" + msg)
    }
}

func main() {

AUser := &User{name: "AUser"}
BUser := &User{name: "BUser"}

chatroom := &ChatRoom{"room123"}
```

```
chatroom.RegisterUser(AUser)
chatroom.RegisterUser(BUser)
AUser.SendMsg("hello im auser")
BUser.SendMsg("hello im Buser")

}

*****
```

备忘录模式：

```
package main

import (
    "container/list"
    "fmt"
)

// 备忘录模式 ( Memento pattern )
// 在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。
// 这样以后就可将该对象恢复到原先保存的状态。

// 相关模式：原型模式
```

```
// 文本编辑
type Text struct {
    Value string
}
```

```
// 写
func (t *Text) Write(value string) {
    t.Value = value
}
```

```
// 读取
func (t *Text) Read() string {
    return t.Value
}
```

```
// 备忘
func (t *Text) SaveToMemento() *Memento {
    return &Memento{Value: t.Value}
}
```

```
// 从备忘恢复
func (t *Text) RestoreFromMemento(m *Memento) {
    if m != nil {
        t.Value = m.Value
    }
    return
}
```

```
// 备忘结构
type Memento struct {
    Value string
}
```

```
// 管理备忘记录, Storage为List的子类
```

```
type Storage struct {
    *list.List
}
```

```
// Back returns the last element of list l or nil.
```

```
// and remove form list
```

```
func (s *Storage) RPop() *list.Element {
    ele := s.Back()
    if ele != nil {
        s.Remove(ele)
    }
    return ele
}
```

```
func main()
```

```
storage := &Storage{list.New()}
text := &Text{"hello world"}
fmt.Println(text.Read())
storage.PushBack(text.SaveToMemento())
text.Write("nihao")
fmt.Println(text.Read())
storage.PushBack(text.SaveToMemento())
text.Write("i know")
fmt.Println(text.Read())
fmt.Println(storage.List.Len())
```

```
//后退回滚
```

```
mediator := storage.RPop()
if mediator != nil {
    text.RestoreFromMemento(mediator.Value.(*Memento))
}
fmt.Println(text.Read())
fmt.Println(storage.List.Len())

//后退回滚
mediator = storage.RPop()
if mediator != nil {
    text.RestoreFromMemento(mediator.Value.(*Memento))
}
fmt.Println(text.Read())

//后退 已没有
mediator = storage.RPop()
if mediator != nil {
    text.RestoreFromMemento(mediator.Value.(*Memento))
}
fmt.Println(text.Read())

}
```

```
*****
观察者模式 : :
package main

import (
    "fmt"
)

// 观察者模式 ( Observer Pattern )
// 定义对象间的一种一对多的依赖关系,
// 当一个对象的状态发生改变时, 所有依赖于它的对象都得到通知并被自动更新。
//
//定义观察者及其实现
type Observer interface {
    Notify(interface{})
}

type AObserver struct {
    Id string
}
```

```

}

func (ao *AObserver) Notify(sub interface{}) {
    fmt.Println(ao.Id, " receive ", sub.(*Subject).state)
}

//定义subject及其方法
type Subject struct {
    observers []Observer
    state    string
}

//状态发生变化后要通知所有的观察者
func (s *Subject) SetState(state string) {
    s.state = state
    s.NotifyAllObserver()
}

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

func (s *Subject) NotifyAllObserver() {
    for _, observer := range s.observers {
        observer.Notify(s)
    }
}

func main() {
    sub := &Subject{}
    oba := &AObserver{
        Id: "a123",
    }
    obb := &AObserver{
        Id: "b123",
    }
    sub.Attach(oba, obb)
    sub.SetState("修改状态")
}
*****
```

模板方法：

```
package main

import (
    "fmt"
)

// 模板模式 ( Template Pattern )
// 定义一个操作中算法的框架，而将一些步骤延迟到子类中。
// 模板方法模式使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。
// 定义游戏及其实现类
type Game interface {
    Start()
    Playing()
    End()
}

// 父类
type AbsGame struct {
}

func (ag *AbsGame) Start() {
    fmt.Println("start game")
}

func (ag *AbsGame) Playing() {
    fmt.Println("playing game")
}

func (ag *AbsGame) End() {
    fmt.Println("end game")
}

// 继承父类AbsGame
type FootBall struct {
    *AbsGame
}

// 重新定义父类上的Playing方法
func (fb *FootBall) Playing() {
    fmt.Println("playing FootBall game")
}

// 继承父类AbsGame
```

```

type Cricket struct {
    *AbsGame
}

//重新定义父类上的playing方法
func (fb *Cricket) Playing() {
    fmt.Println("playing Cricket game ")
}

func RunGame(g Game) {
    g.Start()
    g.Playing()
    g.End()
}
func main() {
    RunGame(&FootBall{})
    RunGame(&Cricket{})
}

```

访问者模式：

package pattern

```

import (
    "fmt"
    "math"
)

// 访问者模式 ( Visitor Pattern ) 主要将数据结构与数据操作分离
//
// 1、对象结构中对象对应的类很少改变，但经常需要在此对象结构上定义新的操作。
// 2、需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而需要避免让这些操作"污染"这些
// 对象的类，也不希望在增加新操作时修改这些类。

```

```

type Visitor interface {
    Visit(DataStruct)
}

```

```

type DataStruct interface {
    Accept(Visitor)
}

```

```
}

type ABData struct {
    A int
    B int
}

func (as *ABData)Accept(vi Visitor){
    vi.Visit(as)
}
```

```
type AddVisitor struct {

}

func (av *AddVisitor)Visit(dataS DataStruct){
    data:=dataS.(*ABData)
    sum:=data.A+data.B
    fmt.Println("A+B=",sum)
}
```

```
type SubVisitor struct {

}

func (sv *SubVisitor)Visit(dataS DataStruct){
    data:=dataS.(*ABData)
    sum:=data.A-data.B
    fmt.Println("abs(A-B)=",math.Abs(float64(sum)))
}
```

```
func VisitorTest(){
    Data:=&ABData{A:8,B:10}
    add:=&AddVisitor{}
    sub:=&SubVisitor{}

    Data.Accept(add)
    Data.Accept(sub)
}
```