创建型模式（Creational Patterns）

这些设计模式提供了一种在创建对象的同时隐藏创建逻辑的方式，而不是使用new运算符直接实例化对象。这使得程序在判断针对某个给定实例需要创建哪些对象时更加灵活。

简单工厂模式（Simple Factory）

说明

定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，工厂模式使其创建过程延迟到子类进行。

何时使用

我们明确地计划不同条件下创建不同实例时。

使用场景

1. 日志记录器：记录可能记录到本地硬盘、系统事件、远程服务器等，用户可以选择记录日志到什么地方。
2. 数据库访问，当用户不知道最后系统采用哪一类数据库，以及数据库可能有变化时。
3. 设计一个连接服务器的框架，需要三个协议，"POP3"、"IMAP"、"HTTP"，可以把这三个作为产品类，共同实现一个接口。

Code

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。

```
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)
}
```

```
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（Factory Method）

目的

当对象的创建逻辑比较复杂，不只是简单的new一下就可以，而是要组合其他类对象，做各种初始化操作的时候，推荐使用工厂方法模式，将复杂的创建逻辑拆分到多个工厂类中，让每个工厂类都不至于过于复杂。

https://lailin.xyz/post/factory.html

Code

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。

```
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
}
```

```
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（Abstract Factory）

说明

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。

何时使用

系统的产品有多于一个的产品族，而系统只消费其中某一族的产品。

应用实例

工作了，为了参加一些聚会，肯定有两套或多套衣服吧，比如说有商务装（成套，一系列具体产品）、时尚装（成套，一系列具体产品）。

假设一种情况，在您的家中，某一个衣柜（具体工厂）只能存放某一种这样的衣服（成套，一系列具体产品），每次拿这种成套的衣服时也自然要从这个衣柜中取出了。用OOP的思想去理解，所有的衣柜（具体工厂）都是衣柜类的（抽象工厂）某一个，而每一件成套的衣服又包括具体的上衣（某一具体产品），裤子（某一具体产品），这些具体的上衣其实也都是上衣（抽象产品），具体的裤子也都是裤子（另一个抽象产品）。

Code

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。

```
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{}

// 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XML 抽象工厂实现
type XMLDAOFactory struct{}

func (*XMLDAOFactory) CreateOrderMainDAO() OrderMainDAO {
```

```
  return &XMLMainDAO{}
}

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

```
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（Builder）

说明

创建者模式（Builder）使用多个简单的对象一步一步构建成一个复杂的对象。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

一些基本部件不会变，而其组合经常变化的时候。

何时使用

一些基本部件不会变，而其组合经常变化的时候。

使用场景

1. 需要生成的对象具有复杂的内部结构。
2. 需要生成的对象内部属性本身相互依赖。

Code

```
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"
}

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
}
```

```
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)
  }
}

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（Prototype）

说明

原型模式（Prototype Pattern）是用于创建重复的对象，同时又能保证性能。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

这种模式是实现了一个原型接口，该接口用于创建当前对象的克隆。当直接创建对象的代价比较大时，则采用这种模式。例如，一个对象需要在一个高代价的数据库操作之后被创建。我们可以缓存该对象，在下一个请求时返回它的克隆，在需要的时候更新数据库，以此来减少数据库调用。

何时使用

1. 当一个系统应该独立于它的产品创建，构成和表示时。
2. 当要实例化的类是在运行时刻指定时，例如，通过动态装载。
3. 为了避免创建一个与产品类层次平行的工厂类层次时。
4. 当一个类的实例只能有几个不同状态组合中的一种时。建立相应数目的原型并克隆它们可能比每次用合适的状态手工实例化该类更方便一些。

使用场景

1. 资源优化场景。
2. 类初始化需要消化非常多的资源，这个资源包括数据、硬件资源等。
3. 性能和安全要求的场景。
4. 通过new产生一个对象需要非常繁琐的数据准备或访问权限，则可以使用原型模式。
5. 一个对象多个修改者的场景。
6. 一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多个对象供调用者使用。
7. 在实际项目中，原型模式很少单独出现，一般是和工厂方法模式一起出现，通过clone的方法创建一个对象，然后由工厂方法提供给调用者。原型模式已经与Java融为浑然一体，大家可以随手拿来使用。

Code

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。

```
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].Clone()
}

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

```
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（Singleton）

说明

这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。

注意：

1. 单例类只能有一个实例。
2. 单例类必须自己创建自己的唯一实例。
3. 单例类必须给所有其他对象提供这一实例。

何时使用

当您想控制实例数目，节省系统资源的时候。

使用场景

1. 要求生产唯一序列号。
2. WEB中的计数器，不用每次刷新都在数据库里加一次，用单例先缓存起来。
3. 创建的一个对象需要消耗的资源过多，比如I/O与数据库的连接等。

Code

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。

```
package singleton

import "sync"
```

```go
// Singleton 是单例模式接口，导出的
// 通过该接口可以避免 GetInstance 返回一个包私有类型的指针
type Singleton interface {
  foo()
}

// singleton 是单例模式类，包私有的
type singleton struct{}

func (s singleton) foo() {}

var (
  instance *singleton
  once     sync.Once
)

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

  return instance
}
```

```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")
  }
}

func TestParallelSingleton(t *testing.T) {
  start := make(chan struct{})
  wg := sync.WaitGroup{}
  wg.Add(parCount)
  instances := [parCount]Singleton{}
  for i := 0; i < parCount; i++ {
    go func(index int) {
```

```
        //协程阻塞，等待channel被关闭才能继续运行
        <-start
        instances[index] = GetInstance()
        wg.Done()
      }(i)
    }
    //关闭channel，所有协程同时开始运行，实现并行(parallel)
    close(start)
    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（Structural Patterns）

这些设计模式关注类和对象的组合。继承的概念被用来组合接口和定义组合对象获得新功能的方式。

外观模式（Facade）

说明

外观模式（Facade Pattern）隐藏系统的复杂性，并向客户端提供了一个客户端可以访问系统的接口。这种类型的设计模式属于结构型模式，它向现有的系统添加一个接口，来隐藏系统的复杂性。

何时使用

1. 客户端不需要知道系统内部的复杂联系，整个系统只需提供一个"接待员"即可。
2. 定义系统的入口。

应用实例

1. 去医院看病，可能要去挂号、门诊、划价、取药，让患者或患者家属觉得很复杂，如果有提供接待人员，只让接待人员来处理，就很方便。
2. JAVA的三层开发模式。

Code

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。

```
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"
}
```

```
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（Adapter）

说明

适配器模式（Adapter Pattern）是作为两个不兼容的接口之间的桥梁。这种类型的设计模式属于结构型模式，它结合了两个独立接口的功能。

这种模式涉及到一个单一的类，该类负责加入独立的或不兼容的接口功能。举个真实的例子，读卡器是作为内存卡和笔记本之间的适配器。您将内存卡插入读卡器，再将读卡器插入笔记本，这样就可以通过笔记本来读取内存卡。

何时使用

1. 系统需要使用现有的类，而此类的接口不符合系统的需要。
2. 想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作，这些源类不一定有一致的接口。
3. 通过接口转换，将一个类插入另一个类系中。（比如老虎和飞禽，现在多了一个飞虎，在不增加实体的需求下，增加一个适配器，在里面包容一个虎对象，实现飞的接口。）

Code

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。

```
package adapter
```

```go
//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
type adapter struct {
  Adaptee
}

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

```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（Proxy）

说明

在代理模式（Proxy Pattern）中，一个类代表另一个类的功能。这种类型的设计模式属于结构型模式。

在代理模式中，我们创建具有现有对象的对象，以便向外界提供功能接口。

何时使用

想在访问一个类时做一些控制。

Code

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。

```
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
}
```

```
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（Composite）

说明

组合模式（Composite Pattern），又叫部分整体模式，是用于把一组相似的对象当作一个单一的对象。组合模式依据树形结构来组合对象，用来表示部分以及整体层次。这种类型的设计模式属于结构型模式，它创建了对象组的树形结构。

这种模式创建了一个包含自己对象组的类。该类提供了修改相同对象组的方式。

何时使用

1. 您想表示对象的部分-整体层次结构（树形结构）。
2. 您希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象。

Code

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。

```
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
}

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
}
```

```go
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)
  }
}
```

```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（Flyweight）

说明

享元模式（Flyweight Pattern）主要用于减少创建对象的数量，以减少内存占用和提高性能。这种类型的设计模式属于结构型模式，它提供了减少对象数量从而改善应用所需的对象结构的方式。

享元模式尝试重用现有的同类对象，如果未找到匹配的对象，则创建新对象。

何时使用

1. 系统中有大量对象。
2. 这些对象消耗大量内存。
3. 这些对象的状态大部分可以外部化。
4. 这些对象可以按照内蕴状态分为很多组，当把外蕴对象从对象中剔除出来时，每一组对象都可以用一个对象来代替。
5. 系统不依赖于这些对象身份，这些对象是不可分辨的。

Code

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。

```
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())
}
```

```
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（Decorator）

说明

装饰器模式（Decorator Pattern）允许向一个现有的对象添加新的功能，同时又不改变其结构。这种类型的设计模式属于结构型模式，它是作为现有的类的一个包装。

这种模式创建了一个装饰类，用来包装原有的类，并在保持类方法签名完整性的前提下，提供了额外的功能。

何时使用

在不想增加很多子类的情况下扩展类。

Code

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。

```
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
}
```

```
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（Bridge）

说明

桥接（Bridge）是用于把抽象化与实现化解耦，使得二者可以独立变化。这种类型的设计模式属于结构型模式，它通过提供抽象化和实现化之间的桥接结构，来实现二者的解耦。

这种模式涉及到一个作为桥接的接口，使得实体类的功能独立于接口实现类。这两种类型的类可被结构化改变而互不影响。

何时使用

实现系统可能有多个角度分类，每一种角度都可能变化。

Code

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。

```
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)
}
```

```
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() {
  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
}
```

过滤器模式（FilterPattern）

说明

过滤器模式（Filter Pattern）或标准模式（Criteria Pattern）是一种设计模式，这种模式允许开发人员使用不同的标准来过滤一组对象，通过逻辑运算以解耦的方式把它们连接起来。这种类型的设计模式属于结构型模式，它结合多个标准来获得单一标准。

行为型模式（Behavioral Patterns）

这些设计模式特别关注对象之间的通信。

中介者模式（Mediator）

说明

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。中介者模式属于行为型模式。

何时使用

多个类相互耦合，形成了网状结构。

应用实例

MVC框架，其中C（控制器）就是M（模型）和V（视图）的中介者。

Code

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。

```
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)
  }
}
```

```
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（Observer）

说明

当对象间存在一对多关系时，则使用观察者模式（Observer Pattern）。比如，当一个对象被修改时，则会自动通知依赖它的对象。观察者模式属于行为型模式。

何时使用

一个对象（目标对象）的状态发生改变，所有的依赖对象（观察者对象）都将得到通知，进行广播通知。

Code

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。

```
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)
}
```

```
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（Command）

说明

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。

何时使用

在某些场合，比如要对行为进行"记录、撤销/重做、事务"等处理，这种无法抵御变化的紧耦合是不合适的。在这种情况下，如何将"行为请求者"与"行为实现者"解耦？将一组行为抽象为对象，可以实现二者之间的松耦合。

Code

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。

```
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()
}

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()
}
```

```
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（Iterator）

说明

迭代器模式（Iterator Pattern）是Java和.Net编程环境中非常常用的设计模式。这种模式用于顺序访问集合对象的元素，不需要知道集合对象的底层表示。

何时使用

遍历一个聚合对象。

Code

迭代器模式用于使用相同方式迭代不同类型集合或者隐藏集合类型的具体实现。

可以使用迭代器模式使遍历同时应用迭代策略，如请求新对象、过滤、处理对象等。

```
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
}

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)
  }
}
```

```
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（Template Method）

说明

在模板模式（Template Pattern）中，一个抽象类公开定义了执行它的方法的方式/模板。它的子类可以按需要重写方法实现，但调用将以抽象类中定义的方式进行。

何时使用

有一些通用的方法。

Code

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。

```
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)
}
```

```
package templatemethod

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（Strategy）

说明

在策略模式（Strategy Pattern）中，一个类的行为或其算法可以在运行时更改。这种类型的设计模式属于行为型模式。

在策略模式中，我们创建表示各种策略的对象和一个行为随着策略对象改变而改变的context对象。策略对象改变context对象的执行算法。

何时使用

一个系统有许多许多类，而区分它们的只是他们直接的行为。

Code

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。

```
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) *Payment {
  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.CardID)

}
```

```
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（State）

说明

在状态模式（State Pattern）中，类的行为是基于它的状态改变的。

在状态模式中，我们创建表示各种状态的对象和一个行为随着状态对象改变而改变的context对象。

何时使用

代码中包含大量与对象状态有关的条件语句。

Code

状态模式用于分离状态和行为。

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

type Saturday struct{}

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

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

```
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（Memento）

说明

备忘录模式（Memento Pattern）保存一个对象的某个状态，以便在适当的时候恢复对象。

何时使用

很多时候我们总是需要记录一个对象的内部状态，这样做的目的就是为了允许用户取消不确定或者错误的操作，能够恢复到他原先的状态，使得他有"后悔药"可吃。

Code

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传递给外部进行存储，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。

```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)
}
```

```
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（Interpreter）

说明

解释器模式（Interpreter Pattern）提供了评估语言的语法或表达式的方式，这种模式实现了一个表达式接口，该接口解释一个特定的上下文。这种模式被用在SQL解析、符号处理引擎等。

何时使用

如果一种特定类型的问题发生的频率足够高，那么可能就值得将该问题的各个实例表述为一个简单语言中的句子。这样就可以构建一个解释器，该解释器通过解释这些句子来解决该问题。

Code

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。

```
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{
    val: v,
  }
}

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

```
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（Chain of Responsibility）

说明

顾名思义，责任链模式（Chain of Responsibility Pattern）为请求创建了一个接收者对象的链。这种模式给予请求的类型，对请求的发送者和接收者进行解耦。

在这种模式中，通常每个接收者都包含对另一个接收者的引用。如果一个对象不能处理该请求，那么它会把相同的请求传给下一个接收者，依此类推。

何时使用

在处理消息的时候以过滤很多道。

Code

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。

```
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{}

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
}
```

```
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（Visitor）

说明

在访问者模式（Visitor Pattern）中，我们使用了一个访问者类，它改变了元素类的执行算法。通过这种方式，元素的执行算法可以随着访问者改变而改变。根据模式，元素对象已接受访问者对象，这样访问者对象就可以处理元素对象上的操作。

何时使用

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，使用访问者模式将这些封装到类中。

Code

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。

```
package visitor

import "fmt"

type Customer interface {
  Accept(Visitor)
}

type Visitor interface {
  Visit(Customer)
}

type EnterpriseCustomer struct {
  name string
}

type CustomerCol struct {
  customers []Customer
}
```

```go
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,
  }
}

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)
  }
}
```

```
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
}
```

空对象模式（Null Object Pattern）

说明

在空对象模式（Null Object Pattern）中，一个空对象取代NULL对象实例的检查。Null对象不是检查空值，而是反应一个不做任何动作的关系。这样的Null对象也可以在数据不可用的时候提供默认的行为。

在空对象模式中，我们创建一个指定各种要执行的操作的抽象类和扩展该类的实体类，还创建一个未对该类做任何实现的空对象类，该空对象类将无缝地使用在需要检查空值的地方。

参考链接：

- https://www.runoob.com/design-pattern
- https://github.com/senghoo/golang-design-pattern
- https://github.com/mohuishou/go-design-pattern