更多类型:struts,lices,maps

学习如何基于现有类型(`strcut,array,slice,map`)定义更多的类型

指针

Go拥有，一个持有值的内存地址。

`*T` 类型是针对值 `T` 的一个。它的默认值或零值是 `nil` 。

```
var p *int // 声明*int类型的p，p指向底层int的值
```

```
package main

import "fmt"

func main() {
    var p *int
    fmt.Printf("%v, %T", p, p)
}
// <nil>, *int
```

`&` 操作符会 `生成` 一个指向其操作数的。

```
i := 34
p  = &i
```

```
package main

import "fmt"

func main() {
    i := 99
    p := &i
    fmt.Printf("%v, %T\n%v, %T", i, i, p, p)
}

// 99, int
// 0xc420070178, *int
```

```
package main

import "fmt"

func main() {
    var p *int
    i := 99
    //p = 99 注意这里不能直接赋值，类型不对应
    //p := &99 cannot take address of 99!!
    // &操作符会生成一个指向其操作数的
    p = &i
    fmt.Printf("%v, %T", p, p)
}
```

`*`操作符`表示`(建议理解为读取设置)指向的底层值

```
fmt.Println(*p) // 通过 p 读取 i
*p = 21         // 通过 p 设置 i
```

这也被作为`间接引用`或`重定向`。

和C语言不一样，Go没有运算。

几个案例

```
    var p *int
    //*p = 66 //panic: runtime error: invalid memory address or nil pointer dereference
    fmt.Printf("%v\n",*p) //也不能单独打印nil pointer,同上一样的错误
```

```
package main

import "fmt"

func main() {
    var p *int
    // *p = 66
    // fmt.Printf("%v\n",*p)
    i := 99
    p = &i
    fmt.Printf("%v, %T\n", p, p)
    *p = 100
```

```
    fmt.Printf("*p->%v", *p)
}

// 0xc42000e228, *int
// *p->100
```

```
var p *int
i := 100
// *p = 100 运行时错误 panic: runtime error: invalid memory address or nil poi
nter dereference
p = &i
fmt.Printf("*p=%v",*p)
```

```
package main

import "fmt"

func main() {
    i, j := 42, 2701

    p := &i
    fmt.Println(*p)
    *p = 21
    fmt.Println(i)

    p = &j
    *p = *p / 37
    fmt.Println(j)
}
```

小技巧:

- 变量通常缩写为 `ptr`

- 的格式化标识符为 `%p`

- 使用引用另一个值被称为间接引用 `*ptr`
- & 获取地址

- * 获取地址所指向的底层值

- 对于任何一个变量， `var = *(&ptr)` 都成立

- 不能得到常量的地址
- 经常导致内存泄漏的指针运算在go中不支持，比如*p++
- 空指针的反向引用是不合法的
```go
package main

func main() {
var p *int = nil
*p = 0
}
```

```go
package main

import (
    "fmt"
)

func main() {
    const i  = 5
    fmt.Printf("%p", &i) //can not take address of i
    x := 5
    fmt.Printf("%p", &x)
}
```

```go
package main

import (
    "fmt"
)

func main() {
    var a int = 5
    var ptr *int = &a
    fmt.Printf("%p\n", ptr)
    fmt.Printf("%v", *ptr) //间接引用(反引用)
}
```

go 面向对象

用户自定义类型

```
package main

import "fmt"

type foo int

func main() {
    var myAge foo
    myAge = 44
    fmt.Printf("%T %v \n", myAge, myAge)
}
// main.foo 44
```

别名类型与底层类型间转换

```
package main

import "fmt"

type foo int

func main() {
    var myAge foo
    myAge = 44
    fmt.Printf("%T %v \n", myAge, myAge)

    var age int = 23
    //可以与底层类型互相转换
    fmt.Println(foo(age))
    fmt.Println(int(myAge))

}
```

但上述例子中的转换并不是好的实践方式。在官方API中的`Duration`为了给该类型添加方法，需要根据返回值进行转换。

```
type Duration int64

// Hours returns the duration as a floating point number of hours.
func (d Duration) Hours() float64 {
    hour := d / Hour
```

```
    nsec := d % Hour
    return float64(hour) + float64(nsec)/(60*60*1e9)
}
```

struct

`struct`结构体是字段的集合。 结构体是也是自定义类型底层类型的一种。结构体是组合类型。

```
package main

import "fmt"

type Vertex struct { // 结构为{a int b bool}的x类型 落点是:xxx结构的yyy类型。但一般大家都称xxx结构体，具体内部结构类型忽略
    X int
    Y int
}

func main() {
    fmt.Println(Vertex{1,2})
    fmt.Printf("%q\n", Vertex{1,2})
    fmt.Printf("%v", Vertex{1,2})
}

// {1 2}
// {'\x01' '\x02'}
// {1 2}
```

- 空结构体，无字段

```
package main

import "fmt"

type test struct {

}
func main() {
    t := test{}
    fmt.Println(t)
}
// {}
```

```go
package main

import "fmt"

type test struct {
}

func main() {
    var t test
    fmt.Println(t)
}
```

```go
package main

import "fmt"

type test struct {
    x,y int
}

func main() {
    var t test
    fmt.Println(t)
}
```

通过上面案例的测试，结构体的默认值为字段的零值

```go
package main

import "fmt"

type Person struct {
    Name string
    age int
}

func main() {
    var p Person
    fmt.Println(p)
}
// { 0}
```

```go
package main
```

```go
import "fmt"

type person struct {
    first string
    last  string
    age   int
}

func main() {
    p1 := person{"James", "Bond", 20}
    p2 := person{"Miss", "Moneypenny", 18}
    fmt.Println(p1.first, p1.last, p1.age)
    fmt.Println(p2.first, p2.last, p2.age)

    p3 := person{}
    fmt.Println(p3)

    var p4 person
    fmt.Println(p4)

}

//James Bond 20
//Miss Moneypenny 18
//{  0}
//{  0}
```

结构体字段

使用点号访问结构体字段(struct field)

```go
package main

import "fmt"

type Vertex struct {
    X int
    Y int
}

func main() {
    v := Vertex{3,4}
    x := v.X
    fmt.Println(x)
```

```
    v.X = 5
    fmt.Println(x)
    fmt.Println(v.X)
}

//3
//3
//5
```

```
package main

import "fmt"

type Person struct {
    Name string
    age int
}

func main() {
    var p Person
    p.Name="jordan"
    p.age=18
    fmt.Println(p)
}
// {jordan 18}
```

```
var p Person
p.Name = "jordan"
p.age = 18
fmt.Println(p)

p1 := Person{Name: "iversion", age: 20}
fmt.Println(p1)
p2 := Person{"kobe", 12}
fmt.Println(p2)
```

结构也是值类型

```
package main

import "fmt"
```

```go
type Person struct {
    Name string
    age  int
}

func main() {
    var p Person
    p.Name = "jordan"
    p.age = 18
    fmt.Println(p)

    p1 := Person{Name: "iversion", age: 20}
    fmt.Println(p1)
    p2 := Person{"kobe", 12}
    fmt.Println(p2)

    p3 := Person{"zhangsan",25}
    fmt.Println(p3)
    Change(p3)
    fmt.Println(p3)
}

func Change(p Person) {
    p.age = 24
    fmt.Println(p)
}

// {zhangsan 25}
// {zhangsan 24}
// {zhangsan 25}
```

- 修改为指针可以拷贝引用

```go
package main

import "fmt"

type Person struct {
    Name string
    age  int
}

func main() {
    p3 := Person{"zhangsan",25}
    fmt.Println(p3)
    Change(&p3)
    fmt.Println(p3)
```

```
}

func Change(p *Person) {
    p.age = 24
    fmt.Println(p)
}
// {zhangsan 25}
// &{zhangsan 24}
// {zhangsan 24}
```

- 使用指针拷贝，进行性能优化

```
package main

import (
    "fmt"
)

type Person struct {
    Name string
    age  int
}

func main() {

    var p Person
    p.Name = "jordan"
    p.age = 18
    fmt.Println(p)

    p1 := Person{Name: "iversion", age: 20}
    fmt.Println(p1)
    p2 := Person{"kobe", 12}
    fmt.Println(p2)

    p3 := Person{"zhangsan", 25}
    fmt.Println(p3)
    Change(&p3)
    fmt.Println(p3)

    Change2(&p3)
    fmt.Println(p3)

    // 每次都需要单独&获取地址比较麻烦
    //可以采取在初始化的时候就将地址去除
    pp := &Person{"lisi",30}
```

```
    fmt.Println(pp)
    Change3(pp)
    fmt.Println(pp)

}

func Change(p *Person) {
    p.age = 24
    fmt.Println(p)
}

func Change2(p *Person) {
    p.age = 100
    fmt.Println(p)
}

func Change3(p *Person) {
    p.age = 66
    fmt.Println(p)
}
```

结构体

- 结构体字段可以使用结构体访问。
- 当拥有结构体(的) `p` 时，我们可以通过 `(*p).X` 访问结构体字段 `X` 。
- 不过 `(*p).X` 这样写有些笨重啰嗦，所以Go语言允许我们使用 `隐式间接引用` ，直接写 `p.X` 即可。
- 注意不要直接写成 `*p.X` ，而是 `(*p).X` 。比如 `fmt.Printf("%v", (*p).X)`

```
package main

import "fmt"

type Vertex struct {
    X int
    Y int
}

func main() {
    v := Vertex{1,2}
    p := &v
    fmt.Println(v.X)
    p.X = 1e9
```

```
    fmt.Println(p.X) // 隐式间接引用
    // fmt.Println(*p.X) // compile error! ! ! !
    fmt.Println((*p).X) // 间接引用
    fmt.Println(v)
}

// 1
// 1000000000
// 1000000000
// {1000000000 2}
```

结构体字面值

- 结构体字面值通过**直接列出字段值**来分配一个结构体。
- 使用`Name:value`语法可以仅列出部分字段。(与字段顺序无关)
- 特殊的前缀`&`返回一个指向结构体的。

```
package main

import "fmt"

type Vertex struct {
    X, Y int
}

var (
    v1 = Vertex{1, 2}
    v2 = Vertex{X: 22}
    v3 = Vertex{}
    p  = &Vertex{1, 2}
)

func main() {
    fmt.Println(v1,p,v2,v3)
    fmt.Println(*p)
    fmt.Println(p.X)
}

// {1 2} &{1 2} {22 0} {0 0}
// {1 2}
// 1
```

```go
package main

import (
    "fmt"
)

type Person struct {
    Name string
    age  int
}

func main() {
    // 每次都需要单独&获取地址比较麻烦
    //可以采取在初始化的时候就将地址去除
    pp := &Person{"lisi",30}
    fmt.Println(pp)
    Change3(pp)
    fmt.Println(pp)
    pp.Name = "zhangs" // 这种方式叫做隐式重定向
    fmt.Println(pp)
    (*pp).Name= "wangwu" // 这种方式较为麻烦，*pp 不如 隐式 pp go编译器的功劳
    fmt.Println(pp)
}


func Change3(p *Person) {
    p.age = 66
    fmt.Println(p)
}
```

总结:在使用结构的时候，总是习惯性的在初始化结构体的时候获取地址，以后再每次使用结构时，就不需要
麻烦的使用&，这样就很方便，go在修改字段的时候也兼容*p和p的使用，只是在传参的时候加上 `*p` 取地址的符号。

```go
type Person struct {
    Name string
    Age int
}
func main() {
    p := &Person{"张三",20}
    F(p)
}
```

```
func F(p *Person) {
    fmt.Println(p)
}
```

匿名结构

```
package main

import "fmt"

func main() {
    p := struct {
        Name string
        Age int
    }{
        "kobe",24}

    fmt.Println(p)
}

// {kobe 24}
```

匿名结构嵌套

类似于java中的引用类型字段，go则使用匿名结构字段

```
package main

import (
    "fmt"
)

type Person struct {
    Name string
    Age  int
    Address struct{
        Province string
        City    string
    }
}

func main() {
```

```
    //p := Person{"kobe",24} //too few fields 编译不通过
    p := Person{Name:"kobe",Age:12}
    p.Address.Province="shandong"
    p.Address.City="weihai"
    fmt.Println(p)
}
```

注意：结构体可以省略字段名，但不要混合使用。

```
package main

import (
    "fmt"
)

type Person struct {
    Name string
    Age  int
}

func main() {
    p := Person{"kobe",24}
    p2:= Person{Name:"kobe",Age:34}
    fmt.Println(p,p2)
}
```

内嵌类型

也是复合结构体字段，某类型中嵌套另个一结构体字段

```
package main

import (
    "fmt"
)

type Address struct {
    Province string
    City     string
}
type Person struct {
    Name    string
    Age     int
```

```
    Address
}

func main() {
    //p := Person{"kobe",24} //too few fields 编译不通过
    p := Person{Name: "kobe", Age: 12}
    p.Address.Province = "shandong"
    p.Address.City = "weihai"
    fmt.Println(p)

    p2 := Person{"kobe", 39, Address{"shandong", "weihai"}} // 不能省略Addres
s struct类型
    fmt.Println(p2)
}
```

stackoverflow

匿名结构字段

```
package main

import "fmt"

type Person struct {
    string
    int
}

func main() {
    p := Person{"kobe",22} // 一定要按照类型的顺序来，否则将异常
    fmt.Println(p)
}
```

不建议使用匿名结构体字段。

结构体指针

```
package main

import "fmt"
```

```
type person struct {
    name string
    age  int
}

func main() {
    p1 := &person{"James", 20}
    fmt.Println(p1)
    fmt.Printf("%T\n", p1)
    fmt.Println(p1.name)
    fmt.Println(p1.age)

   //fmt.Println(*p1.name)
    fmt.Println((*p1).name)
}
```

注意对于地址重定向的写法。 另外golang语言自身对于地址重定向做了语法糖优化，开发者编码时无需再做这步操作。但如果手动操作，则需注意格式，(*p)是主体，而`.`操作符的优先级较高，所以需要`()`提升`*`操作符的优先级。

结构体比较

```
package main

import "fmt"

type Person struct {
    string
    int
}

func main() {
    p := Person{"kobe",22}
    p2 := p
    p2 = Person{"12",12}
    p3 := p
    fmt.Println(p==p2)
    fmt.Println(p==p3) //值(内容)比较 ,必须在相同类型下比较
    fmt.Println(p)
    fmt.Println(p2)
}

// false
// true
// {kobe 22}
```

```
// {12 12}
```

注意结构体不能和nil比较
```go
package main

import (
"fmt"
)

func main() {
var stu student
fmt.Printf("%v",stu,stu==nil) // compile error cannot convert nil to type student
}

type student struct {
Id int
Name string
}
func Reset(x student){
fmt.Println(x.Id,x.Name)
}
```
注意和nil进行比较的还真不多...
```go
package main

import (
"fmt"
"os"
)

func main() {
var x os.File
fmt.Printf("%v %t\n",x, x == nil)

}

func Reset(x os.File){
fmt.Println(x.Name())
}
```

```

# go中没有Class也没有继承，如何面向对象

go与其他面向对象语言非常突兀的不同点就是没有Class和继承，但是并不代表go没有面向对象的特性，恰恰相反，go使用struct组合模式进行面向对象开发一样很优雅。

```
package main

import "fmt"

type People struct {
    Name string
    Age int
    Sex byte
}

type Boss struct {
    People
    salary float32
}

type Employee struct {
    People
    salary float32
}

func main() {
    // b := Boss{"kobe",24,1,100}
    // e := Employee{"james",23,0,50}

    b := Boss{People{"kobe",24,1},100}
    e := Employee{People{"james",23,0},50}
    b.People.Age = 39
    b.Age = 40
    fmt.Println(b,e)
}
```

在组合之间优先级是怎样的呢?

也就是说在Boss和People中拥有相同的字段名，是报异常呢还是优先取其一呢?

go组合在命名冲突的情况下会采取就近原则，并向下传递，找不到或级别相同都会异常。
```

```go
package main

import "fmt"

type People struct {
    Name string
    Age int
    Sex byte
}

type Boss struct {
    People
    Age int
    salary float32
}

type Employee struct {
    People
    salary float32
}

func main() {
    b := Boss{People{"kobe",24,1},39,100}
    e := Employee{People{"james",23,0},50}
    fmt.Println(b.Age) // 就近原则
    fmt.Println(b.People.Age) // 向下传递
    fmt.Println(b,e)
}
```

一种特殊情况，被组合的多个字段名相同，编译器将无法选择。

1. 直接就编译不通过

```go
package main

import "fmt"

type A struct {
    B
    C
    Name string
}

type B struct {
    Age int
}
```

```
type C struct {
    Age int
}

func main() {
    a := A{"zhangsan", B{12}, C{12}}
    fmt.Println(a)
}

// # command-line-arguments
// src/run.go:20: cannot use "zhangsan" (type string) as type B in field val
ue
// src/run.go:20: cannot use B literal (type B) as type C in field value
// src/run.go:20: cannot use C literal (type C) as type string in field valu
e
```

1. 获取字段时编译不通过

```
package main

import "fmt"

type A struct {
    Name string
    B
    C
}

type B struct {
    Age int
}

type C struct {
    Age int
}

func main() {
    a := A{"zhangsan", B{12}, C{12}}
    a.Age = 20 //编译无法通过 Ambiguous reference 'age' 引起歧义的变量引用
    fmt.Println(a)
}
```

1. 正确的方案 使用组合嵌套，而不是重复定义

```
package main

import "fmt"

type A struct {
    Name string
    B
    C
}

type B struct {
    C
}

type C struct {
    Age int
}

func main() {
    a := A{"zhangsan", B{}, C{12}}
    a.Age = 20
    fmt.Println(a)
}
```

方法

前面我们讲到struct在go中充当了经典面向对象模型中的Class，那么go中是如何实现方法的呢?

通常来说面向对象方法的调用就是向调用对象发送消息，从这个角度而言，对象(struct)是消息的接收者。

Go对方法的实现就是基于上述原理基于函数增加了接收者(receiver)，很巧妙地链接到了struct对象上。

```
package main

import "fmt"

type Dog struct {
    Name string
}

func (dog Dog) Say(name string) {
```

```
    fmt.Println("my name is:",name)
}

func main() {
    odie := Dog{"Odie"}
    odie.Say(odie.Name)
}
// my name is: Odie
```

go中没有方法重载，那么当garfield也说话时，我们同样可以绑定一个say方法到Cat上。

```
package main

import "fmt"

type Cat struct {
    Name string
}

func (cat Cat) Say(name string) {
    fmt.Println("my name is:",name)
}

func main() {
    garfield := Cat{"Garfield"}
    garfield.Say(garfield.Name)
}
```

再加一层父类抽象

```
package main

import "fmt"

type Animal struct {
    Name string
}

type Cat struct {
    Animal
}

type Dog struct {
    Animal
}
```

```
func (cat Cat) Say(name string) {
    fmt.Println("my name is:", name)
}

func (dog Dog) Say(name string) {
    fmt.Println("my name is:", name)
}

func main() {
    odie := Dog{Animal{"Odie"}}
    odie.Say(odie.Name)
    garfield := Cat{Animal{"Garfield"}}
    garfield.Say(garfield.Name)
}
```

注意我们认为的常规的方法重载，在Go中是不支持的。

```
func (cat Cat) Say(name string) {
    fmt.Println("my name is:", name)
}

func (cat Cat) Say()  {
    fmt.Println(cat.Name)
}
```

receiver 是参数而非self或this

> (p person) is the "receiver"
> it is another parameter
> not idiomatic to use "this" or "self"
> "Not many people know this, but method notation, i.e. v.Method() is actually syntactic sugar and Go also understands the de-sugared version of it: (T).Method(v). You can see an example here. Naming the receiver like any other parameter reflects that it is, in fact, just another parameter quite well.
> This also implies that the receiver-argument inside a method may be nil. This is not the case with this in e.g. Java."
> SOURCE:
> https://www.reddit.com/r/golang/comments/3qoo36/question_why_is_self_or_this_not_considered_a/?utm_source=golangweekly&utm_medium=email

```go
package main

type T struct{}

func (t T) Method() {}

func main() {
    t := T{}
    t.Method()    // this is valid
    (T).Method(t) // this too
}
```

注意下面的p5也是有初始值的
```go
package main

import "fmt"

type person struct {
first string
last string
age int
}

func (p person) fullName() string {
return p.first + p.last
}

func (p *person) String() {
fmt.Printf("%s - %d\n", p.fullName(), p.age)
}

func main() {
p1 := person{"James", "Bond", 20}
p2 := person{"Miss", "Moneypenny", 18}
fmt.Println(p1.fullName())
fmt.Println(p2.fullName())
```

```go
p3 := person{"kobe", "brant", 24}
p3.String()

p4 := person{}
```

```
    p4.String()

    var p5 person
    p5.String()
```

}
```

# receiver的值传递与指针传递

```
package main

import "fmt"

type Animal struct {
    Name string
}
type Cat struct {
    Animal
}

type Dog struct {
    Animal
}

func (cat Cat) ChangeName() {
    cat.Name = "Garfield_change"
    fmt.Println("改变中:",cat.Name)
}
func main() {
    garfield := Cat{Animal{"Garfield"}}
    fmt.Println("改变前:",garfield.Name)
    garfield.ChangeName()
    fmt.Println("改变后:",garfield.Name)
}

// 改变前: Garfield
// 改变中: Garfield_change
// 改变后: Garfield
```

通过打印结果我们可以看到receiver默认为值传递，我们希望能够方法修改起到作用，

将receiver的声明修改为指针方式`func (cat *Cat) ChangeName()`
```

再看打印结果

```
// 改变前: Garfield
// 改变中: Garfield_change
// 改变后: Garfield_change
```

总结:传递都是拷贝，数值型是值拷贝，指针则是指针地址的拷贝，换而言之，使用指针则是修改原始的对象，不使用指针则不会修改原值。

值类型得到的是值的副本，修改也是针对副本而言。而指针的副本，即使是副本，也是指向底层的原始值，因此修改是会保留的。

另外在调用上，我们只需要在方法上声明好为指针类型，调用的时候可以不必进行`&,*`的引用，Go编译器这点做得很人性化。注意这里针对的是receiver。
函数的指针参数推荐在初始化的时候进行`&`引用，再进行传参，否则要么编译不通过，要么不方便(前面章节有讲到原因)。

面向对象:
使用方法表达对属性和对应行为的操作，无需直接去操作对象，而是使用方法来操作对象。

```
const day = 24 * time.Hour
fmt.Println(day.Seconds()) // 86400

func (c Celsius) String() string {
    return fmt.Sprintf("%g°C", c)
}
```

go的面向对象两大特点: 封装 组合