结构体

Go语言的结构体有点像面向对象语言中的“类”，但不完全是，Go语言也没打算真正实现面向对象范式。

定义

使用type定义结构体，可以把结构体看做**类型**使用。必须指定结构体的字段（属性）名称和类型。

```
type User struct {
    id          int
    name, addr string // 多个字段类型相同可以合写
    score       float32
}
```

初始化

```
type User struct {
    id          int
    name, addr string
    score       float32
}

// 1 var声明，非常常用
var u1 User // 这种方式声明结构体变量很方便，所有字段都是零值
fmt.Println(u1)
fmt.Printf("%+v\n", u1) // 加上字段打印
fmt.Printf("%#v\n", u1) // 加上更多信息打印

// 2 字面量初始化，推荐
u2 := User{} // 字段为零值
fmt.Printf("%#v\n", u2)

// 3 字面量初始化，field: value为字段赋值
u3 := User{id: 100}
fmt.Printf("%+v\n", u3)

u4 := User{
    id: 102, score: 95.8,
    addr: "Nanjing", name: "Tom",
} // 名称对应无所谓顺序

u5 := User{103, "John", "Beijing", 98.5} // 无字段名称必须按照顺序给出全部字段值
fmt.Printf("%+v\n", u4)
fmt.Printf("%+v\n", u5)
```

可见性

- Go中大写字母开头标识符，跨package包可见，否则只能本包内可见
- 结构体名称以大写开头时，package内外皆可见，在此前提下，结构体中以大写开头的成员（属性、方法）在包外也可见

访问

可以使用字段名称访问

```
u1 := User{103, "John", "Beijing", 98.5} // 无字段名称必须按照顺序给出全部字段值
fmt.Println(u1.id, u1.name, u1.score)
```

修改

通过字段来修改

```
u1 := User{103, "John", "Beijing", 98.5} // 无字段名称必须按照顺序给出全部字段值
fmt.Println(u1)
u1.name = "Tom"
u1.score = 88
fmt.Println(u1)
```

成员方法

利用下面形式为结构体组合方法，这是很方便自由地扩展方式。

```
type User struct {
    id         int
    name, addr string
    score      float32
}

// u称为receiver
// 等价于 func (User) string
func (u User) getName() string {
    return u.name
}

func main() {
    u1 := User{103, "John", "Beijing", 98.5}
    fmt.Println(u1.getName())
}
```

指针

```
type Point struct {
    x, y int
}

```

```go
var p1 = Point{10, 20}          // 实例
fmt.Printf("%T, %[1]v\n", p1) // Point, {10, 20}

var p2 = &Point{5, 6}           // 指针
fmt.Printf("%T, %[1]v\n", p2) // *Point, &{5, 6}

var p3 = new(Point)             // new实例化一个结构体并返回
fmt.Printf("%T, %[1]v\n", p3) // *Point, &{0, 0}

// 通过实例修改属性
p1.x = 100
fmt.Printf("%T, %[1]v\n", p1) // Point, {100, 20}
// 通过指针修改属性
p2.x = 200
p3.x = 300
fmt.Printf("%T, %[1]v\n", p2) // *Point, &{200, 6}
fmt.Printf("%T, %[1]v\n", p3) // *Point, &{300, 0}
// p3.x中. 是 -> 的语法糖，更方便使用。等价于(*p3).x
fmt.Print(*p3, (*p3).x) // {300 0} 300
```

首先，看一个例子

```go
package main

import "fmt"

type Point struct {
    x, y int
}

func test(p Point) Point {
    fmt.Printf("4 %+v %p\n", p, &p)
    return p
}

func main() {
    var p1 = Point{10, 20} // 实例
    fmt.Printf("1 %+v %p\n", p1, &p1)
    p2 := p1
    fmt.Printf("2 %+v %p\n", p2, &p2)
    p3 := &p1
    fmt.Printf("3 %+v %p\n", p3, p3)
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~")

    p4 := test(p1)
    fmt.Printf("5 %+v %p\n", p4, &p4)
}
```

运行结果如下

```
1 {x:10 y:20}  0xc0000180a0
2 {x:10 y:20}  0xc0000180f0
3 &{x:10 y:20} 0xc0000180a0
~~~~~~~~~~~~~~~~~~~~~~~~~~
4 {x:10 y:20}  0xc000018150
5 {x:10 y:20}  0xc000018140
```

可以看出，结构体是**非引用类型**，使用的是值拷贝。传参或返回值如果使用结构体实例，将产生很多副本。如何避免过多副本，如何保证函数内外使用的是同一个结构体实例呢？使用指针。

```go
package main

import "fmt"

type Point struct {
    x, y int
}

func test(p *Point) *Point {
    p.x += 100
    fmt.Printf("4 %+v %p\n", p, p)
    return p
}

func main() {
    var p1 = Point{10, 20} // 实例
    fmt.Printf("1 %+v %p\n", p1, &p1)
    p2 := p1
    fmt.Printf("2 %+v %p\n", p2, &p2)
    p3 := &p1
    fmt.Printf("3 %+v %p\n", p3, p3)
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~")

    p4 := test(p3)
    fmt.Printf("5 %+v %p\n", p1, &p1)
    fmt.Printf("6 %+v %p\n", p4, p4)
    p4.x += 200
    fmt.Printf("7 %+v %p\n", p1, &p1)
    fmt.Printf("8 %+v %p\n", p4, p4)

    p5 := p3
    p5.y = 400 // 会发生什么？
}
```

运行结果如下

```
1 {x:10 y:20}  0xc0000180a0
2 {x:10 y:20}  0xc0000180f0
3 &{x:10 y:20} 0xc0000180a0
~~~~~~~~~~~~~~~~~~~~~~~~~~
4 &{x:110 y:20} 0xc0000180a0
5 {x:110 y:20}  0xc0000180a0
6 &{x:110 y:20} 0xc0000180a0
7 {x:310 y:20}  0xc0000180a0
8 &{x:310 y:20} 0xc0000180a0
&{310 400} {310 400}
```

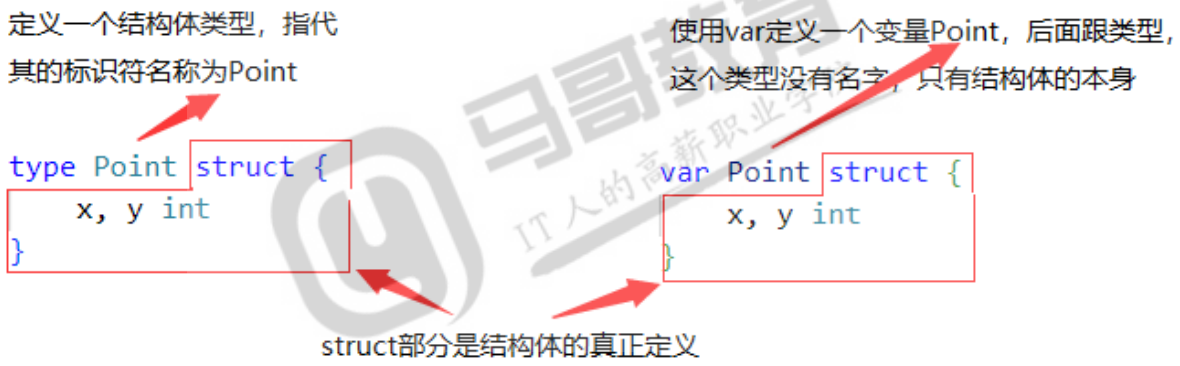
说明，使用了同一个内存区域中的结构体实例，减少了拷贝。

匿名结构体

匿名结构体：标识符直接使用struct部分结构体本身来作为类型，而不是使用type定义的有名字的结构体的标识符。

可以使用`var`、`const`、`:=`来定义匿名结构体。

type定义结构体的标识符，可以反复定义其结构体实例，但是匿名结构体是一次性的。




```go
var Point struct {
    x, y int
} // 定义Point是后面匿名结构体类型的，用零值
fmt.Printf("%#v\n", Point) // 得到的是一个结构体实例

var message = struct {
    id   int
    data string
}{1, "OK"} // 不用零值，初始化
fmt.Printf("%#v\n", message)

student := struct {
    id   int
    name string
}{1, "Tom"} // 短格式定义并初始化
fmt.Printf("%#v\n", student)
```

匿名结构体，只是为了快速方便地得到**一个**结构体实例，而不是使用结构体创建N个实例。

匿名成员

有时候属性名可以省略

```go
type Point struct {
    x    int
    int  // 字段，匿名成员变量
    bool // 匿名，必须类型不一样才能区分
}

var p1 = Point{1, 2, false}
fmt.Println(p1)

var p2 = Point{x: 20, int: 5, bool: false} // 使用类型名作为字段名
fmt.Println(p2, p1.x, p2.int, p2.bool)
```

个人不建议这么做，一般情况下，字段名还是应该见名知义，匿名不便于阅读。

构造函数

Go语言并没有从语言层面为结构体提供什么构造器，但是有时候可以通过一个函数为结构体初始化提供属性值，从而方便得到一个结构体实例。习惯上，函数命名为 NewXxx 的形式。


```go
package main

import "fmt"

type Animal struct {
    name string
    age  int
}

func NewAnimal(name string, age int) Animal {
    a := Animal{name, age}
    fmt.Printf("%+v, %p\n", a, &a)
    return a
}

func main() {
    a := NewAnimal("Tom", 20)
    fmt.Printf("%+v, %p\n", a, &a)
}
```

上例中，NewAnimal的返回值使用了值拷贝，增加了内存开销，**习惯上返回值会采用指针类型**，避免实例的拷贝。

```go
func NewAnimal(name string, age int) *Animal {
    a := Animal{name, age}
    fmt.Printf("%+v, %p\n", a, &a)
    return &a
}
```

父子关系构造

动物类包括猫类，猫属于猫类，猫也属于动物类，某动物一定是动物类，但不能说某动物一定是猫类。

将上例中的Animal结构体，使用匿名成员的方式，嵌入到Cat结构体中，看看效果

```go
package main

import "fmt"

type Animal struct {
    name string
    age  int
}

type Cat struct {
    Animal // 匿名成员，可以使用类型名作为访问的属性名
    color  string
}

func main() {
    var cat = new(Cat) // Cat实例化，Animal同时被实例化
    fmt.Printf("%#v\n", cat)
    cat.color = "black"      // 子结构体属性
    cat.Animal.name = "Tom" // 完整属性访问
    cat.age = 20             // 简化写法，只有匿名成员才有这种效果
    fmt.Printf("%#v\n", cat)
}
```

- 使用结构体嵌套实现类似面向对象父类子类继承（派生）的效果
- 子结构体使用匿名成员能简化调用父结构体成员

指针类型receiver

Go语言中，可以为任意类型包括结构体增加方法，形式是`func Receiver 方法名 签名 {函数体}`，这个receiver类似其他语言中的this或self。

receiver必须是一个类型T实例或者类型T的指针，T不能是指针或接口。

```go
package main

import "fmt"

type Point struct {
    x, y int
}

func (p Point) getX() int { // getX方法绑定到结构体类型Point
    fmt.Println("instance")
    return p.x
}
```

```
13
14 func (p *Point) getY() int {
15     fmt.Println("pointer")
16     return p.y
17 }
18
19 func main() {
20     p := Point{4, 5}
21     fmt.Println(p)
22     fmt.Println(p.getX(), (&p).getX())
23     fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~")
24     fmt.Println(p.getY(), (&p).getY())
25 }
```

运行结果如下

```
1 {4 5}
2 instance
3 instance
4 4 4
5 ~~~~~~~~~~~~~~~~~~~~~~~~~~
6 pointer
7 pointer
8 5 5
```

如果方法中不使用receiver，其标识符可以省略

```
1 func (Point) Comment() {
2     fmt.Println("这是个点")
3 }
```

接收器receiver可以是类型T也可以是指针*T，定义的方法有什么区别?

```
1  package main
2
3  import "fmt"
4
5  type Point struct {
6      x, y int
7  }
8
9  func (p Point) getX() int {
10     return p.x
11 }
12
13 func (p *Point) getY() int {
14     return p.y
15 }
16
17 func (p Point) setX(v int) {
```

```
    fmt.Printf("%+v, %p\n", p, &p)
    p.x = v
    fmt.Printf("%+v, %p\n", p, &p)
}

func (p *Point) setY(v int) {
    fmt.Printf("%+v, %p\n", p, p)
    p.y = v
    fmt.Printf("%+v, %p\n", p, p)
}

func main() {
    p := Point{4, 5}
    fmt.Printf("%+v, %p\n", p, &p)
    p.setX(11) // 实例调用是值拷贝
    p.setY(22) // 看似实例调用，实则是指针，操作同一处内存
    fmt.Printf("%+v, %p\n", p, &p)
}
```

```
{x:4 y:5}, 0xc000128070
{x:4 y:5}, 0xc0001280c0
{x:11 y:5}, 0xc0001280c0
&{x:4 y:5}, 0xc000128070
&{x:4 y:22}, 0xc000128070
{x:4 y:22}, 0xc000128070
```

非常明显，如果是非指针接收器方法调用有值拷贝，操作的是副本，而指针接收器方法调用操作的是同一个内存的同一个实例。

如果是操作大内存对象时，且操作同一个实例时，一定要采用指针接收器的方法。

深浅拷贝

- shadow copy
    - 影子拷贝，也叫浅拷贝。遇到引用类型数据，仅仅复制一个引用而已
- deep copy
    - 深拷贝，往往会递归复制一定深度

注意，深浅拷贝说的是拷贝过程中是否发生递归拷贝，也就是说如果某个值是一个地址，是只复制这个地址，还是复制地址指向的内容。

值拷贝是深拷贝，地址拷贝是浅拷贝，这种说法是**错误**的。因为地址拷贝只是拷贝了地址，因此本质上来讲也是值拷贝。

Go语言中，引用类型实际上拷贝的是标头值，这也是值拷贝，并没有通过标头值中对底层数据结构的指针指向的内容进行复制，这就是浅拷贝。非引用类型的复制就是值拷贝，也就是再造一个副本，这也是浅拷贝。因为你不能说对一个整数值在内存中复制出一个副本，就是深的拷贝。像整数类型这样的基本类型就是一个单独的值，没法深入拷贝，根本没法去讲深入的事儿。

简单讲，大家可以用拷贝文件是否对软链接跟进来理解。直接复制软链接就是浅拷贝，钻进软链接里面复制其内容就是深拷贝。

复杂数据结构，往往会有嵌套，有时嵌套很深，如果都采用深拷贝，那代价很高，所以，浅拷贝才是语言普遍采用的方案。

马哥教育
IT人的高薪职业学院