理解 Go make 和 new 的区别

Jul 26 2017 | golang

> new 和 make 都可以用来分配空间，初始化类型，但是它们确有不同。

new(T) 返回的是 T 的指针

new(T) 为一个 T 类型新值分配空间并将此空间初始化为 T 的零值，返回的是新值的地址，也就是 T 类型的指针 *T，该指针指向 T 的新分配的零值。

```
1  p1 := new(int)
2  fmt.Printf("p1 --> %#v \n ", p1) //(*int)(0xc42000e250)
3  fmt.Printf("p1 point to --> %#v \n ", *p1) //0
4
5  var p2 *int
6  i := 0
7  p2 = &i
8  fmt.Printf("p2 --> %#v \n ", p2) //(*int)(0xc42000e278)
9  fmt.Printf("p2 point to --> %#v \n ", *p2) //0
```

上面的代码是等价的，new(int) 将分配的空间初始化为 int 的零值，也就是 0，并返回 int 的指针，这和直接声明指针并初始化的效果是相同的。

make 只能用于 slice,map,channel

make 只能用于 slice，map，channel 三种类型，make(T, args) 返回的是初始化之后的 T 类型的值，这个新值并不是 T 类型的零值，也不是指针 *T，是经过初始化之后的 T 的引用。

```
var s1 []int
if s1 == nil {
    fmt.Printf("s1 is nil --> %#v \n ", s1) // []int(nil)
}

s2 := make([]int, 3)
if s2 == nil {
    fmt.Printf("s2 is nil --> %#v \n ", s2)
} else {
    fmt.Printf("s2 is not nill --> %#v \n ", s2)// []int{0, 0, 0}
}
```

slice 的零值是 nil，使用 make 之后 slice 是一个初始化的 slice，即 slice 的长度、容量、底层指向的 array 都被 make 完成初始化，此时 slice 内容被类型 int 的零值填充，形式是 [0 0 0]，map 和 channel 也是类似的。

```
var m1 map[int]string
if m1 == nil {
    fmt.Printf("m1 is nil --> %#v \n ", m1) //map[int]string(nil)
}

m2 := make(map[int]string)
if m2 == nil {
    fmt.Printf("m2 is nil --> %#v \n ", m2)
} else {
    fmt.Printf("m2 is not nill --> %#v \n ", m2) map[int]string{}
```

```
}


var c1 chan string
if c1 == nil {
    fmt.Printf("c1 is nil --> %#v \n ", c1) //(chan string)(nil)
}

c2 := make(chan string)
if c2 == nil {
    fmt.Printf("c2 is nil --> %#v \n ", c2)
} else {
    fmt.Printf("c2 is not nill --> %#v \n ", c2)//(chan string)(0xc420016120)
}
```

make(T, args) 返回的是 T 的 实例

```
func modifySlice(s []int) {
    s[0] = 1
}

s2 := make([]int, 3)
fmt.Printf("%#v", s2) //[]int{0, 0, 0}
modifySlice(s2)
fmt.Printf("%#v", s2) //[]int{1, 0, 0}
```

很少需要使用 new

以下代码演示了 **struct 初始化的过程**，可以说明不使用 new 一样可以完成 struct 的初始化工作。

```go

type Foo struct {
    name string
    age  int
}

//声明初始化
var foo1 Foo
fmt.Printf("foo1 --> %#v\n ", foo1) //main.Foo{age:0, name:""}
foo1.age = 1
fmt.Println(foo1.age)

//struct literal 初始化
foo2 := Foo{}
fmt.Printf("foo2 --> %#v\n ", foo2) //main.Foo{age:0, name:""}
foo2.age = 2
fmt.Println(foo2.age)

//指针初始化
foo3 := &Foo{}
fmt.Printf("foo3 --> %#v\n ", foo3) //&main.Foo{age:0, name:""}
foo3.age = 3
fmt.Println(foo3.age)

//new 初始化
foo4 := new(Foo)
fmt.Printf("foo4 --> %#v\n ", foo4) //&main.Foo{age:0, name:""}
foo4.age = 4
fmt.Println(foo4.age)

//声明指针并用 new 初始化
var foo5 *Foo = new(Foo)
fmt.Printf("foo5 --> %#v\n ", foo5) //&main.Foo{age:0, name:""}
```

```
34  foo5.age = 5
35  fmt.Println(foo5.age)
```

foo1 和 foo2 是同样的类型，都是 Foo 类型的值，foo1 是通过 var 声明，Foo 的 filed 自动初始化为每个类型的零值，foo2 是通过字面量的完成初始化。foo3，foo4 和 foo5 是一样的类型，都是 Foo 的指针 *Foo*。*但是所有 foo 都可以直接使用 Foo 的 filed，读取或修改，为什么？*

如果 x 是可寻址的，&x 的 filed 集合包含 m，x.m 和 (&x).m 是等同的，go 自动做转换，也就是 foo1.age 和 foo3.age 调用是等价的，go 在下面自动做了转换。

因而可以直接使用 struct literal 的方式创建对象，能达到和 new 创建是一样的情况而不需要使用 new。

小结

new(T) 返回 T 的指针 *T 并指向 T 的零值。

make(T) 返回的初始化的 T，只能用于 slice，map，channel。