14 内存分配：new 还是 make？什么情况下该用谁？

程序的运行都需要内存，比如像变量的创建、函数的调用、数据的计算等。所以在需要内存的时候就要申请内存，进行内存分配。在 C/C++ 这类语言中，内存是由开发者自己管理的，需要主动申请和释放，而在 Go 语言中则是由该语言自己管理的，开发者不用做太多干涉，只需要声明变量，Go 语言就会根据变量的类型自动分配相应的内存。

Go 语言程序所管理的虚拟内存空间会被分为两部分：堆内存和栈内存。栈内存主要由 Go 语言来管理，开发者无法干涉太多，堆内存才是我们开发者发挥能力的舞台，因为程序的数据大部分分配在堆内存上，一个程序的大部分内存占用也是在堆内存上。

> 小提示：我们常说的 Go 语言的内存垃圾回收是针对堆内存的垃圾回收。

变量的声明、初始化就涉及内存的分配，比如声明变量会用到 var 关键字，如果要对变量初始化，就会用到 = 赋值运算符。除此之外还可以使用内置函数 new 和 make，这两个函数你在前面的课程中已经见过，它们的功能非常相似，但你可能还是比较迷惑，所以这节课我会基于内存分配，进而引出内置函数 new 和 make，为你讲解他们的不同，以及使用场景。

变量

一个数据类型，在声明初始化后都会赋值给一个变量，变量存储了程序运行所需的数据。

变量的声明

和前面课程讲的一样，如果要单纯声明一个变量，可以通过 var 关键字，如下所示：

```
var s string
```

该示例只是声明了一个变量 s，类型为 string，并没有对它进行初始化，所以它的值为 string 的零值，也就是 ""（空字符串）。

上节课你已经知道 string 其实是个值类型，现在我们来声明一个指针类型的变量试试，如下所示：

```
var sp *string
```

发现也是可以的，但是它同样没有被初始化，所以它的值是 *string 类型的零值，也就是 nil。

变量的赋值

变量可以通过 = 运算符赋值，也就是修改变量的值。如果在声明一个变量的时候就给这个变量赋值，这种操作就称为变量的初始化。如果要对一个变量初始化，可以有三种办法。

1. 声明时直接初始化，比如 var s string = "飞雪无情"。
2. 声明后再进行初始化，比如 s="飞雪无情"（假设已经声明变量 s）。
3. 使用 := 简单声明，比如 s:="飞雪无情"。

> 小提示：变量的初始化也是一种赋值，只不过它发生在变量声明的时候，时机最靠前。也就是说，当你获得这个变量时，它就已经被赋值了。

现在我们就对上面示例中的变量 s 进行赋值，示例代码如下：

ch14/main.go

```
func main() {
   var s string
   s = "张三"
   fmt.Println(s)
}
```

运行以上代码，可以正常打印出张三，说明值类型的变量没有初始化时，直接赋值是没有问题的。那么对于指针类型的变量呢？

在下面的示例代码中，我声明了一个指针类型的变量 sp，然后把该变量的值修改为“飞雪无情”。

ch14/main.go

```
func main() {
   var sp *string
   *sp = "飞雪无情"
   fmt.Println(*sp)
}
```

运行这些代码，你会看到如下错误信息：

```
panic: runtime error: invalid memory address or nil pointer dereference
```

这是因为指针类型的变量如果没有分配内存，就默认是零值 nil，它没有指向的内存，所以无法使用，强行使用就会得到以上 nil 指针错误。

而对于值类型来说，即使只声明一个变量，没有对其初始化，该变量也会有分配好的内存。

在下面的示例中，我声明了一个变量 s，并没有对其初始化，但是可以通过 &s 获取它的内存地址。这其实是 Go 语言帮我们做的，可以直接使用。

```
func main() {
   var s string
   fmt.Printf("%p\n",&s)
}
```

还记得我们在讲并发的时候，使用 var wg sync.WaitGroup 声明的变量 wg 吗？现在你应该知道为什么不进行初始化也可以直接使用了吧？因为 sync.WaitGroup 是一个 struct 结构体，是一个值类型，Go 语言自动分配了内存，所以可以直接使用，不会报 nil 异常。

于是可以得到结论：**如果要对一个变量赋值，这个变量必须有对应的分配好的内存，这样才可以对这块内存操作，完成赋值的目的**。

> 小提示：其实不止赋值操作，对于指针变量，如果没有分配内存，取值操作一样会报 nil 异常，因为没有可以操作的内存。

所以一个变量必须要经过声明、内存分配才能赋值，才可以在声明的时候进行初始化。指针类型在声明的时候，Go 语言并没有自动分配内存，所以不能对其进行赋值操作，这和值类型不一样。

> 小提示：map 和 chan 也一样，因为它们本质上也是指针类型。

new 函数

既然我们已经知道了声明的指针变量默认是没有分配内存的，那么给它分配一块就可以了。于是就需要今天的主角之一 new 函数出场了，对于上面的例子，可以使用 new 函数进行如下改造：

ch14/main.go

```
func main() {
    var sp *string
    sp = new(string)//关键点
    *sp = "飞雪无情"
    fmt.Println(*sp)
}
```

以上代码的关键点在于通过内置的 new 函数生成了一个 *string，并赋值给了变量 sp。现在再运行程序就正常了。

内置函数 new 的作用是什么呢？可以通过它的源代码定义分析，如下所示：

```
// The new built-in function allocates memory. The first argument is a type
// not a value, and the value returned is a pointer to a newly
// allocated zero value of that type.
func new(Type) *Type
```

它的作用就是根据传入的类型申请一块内存，然后返回指向这块内存的指针，指针指向的数据就是该类型的零值。

比如传入的类型是 string，那么返回的就是 string 指针，这个 string 指针指向的数据就是空字符串，如下所示：

```
    sp1 = new(string)
    fmt.Println(*sp1)//打印空字符串,也就是string的零值。
```

通过 new 函数分配内存并返回指向该内存的指针后，就可以通过该指针对这块内存进行赋值、取值等操作。

变量初始化

当声明了一些类型的变量时，这些变量的零值并不能满足我们的要求，这时就需要在变量声明的同时进行赋值（修改变量的值），这个过程称为变量的初始化。

下面的示例就是 string 类型的变量初始化，因为它的零值（空字符串）不能满足需要，所以需要在声明的时候就初始化为“飞雪无情”。

```
var s string = "飞雪无情"
s1:="飞雪无情"
```

不止基础类型可以通过以上这种字面量的方式进行初始化，复合类型也可以，比如之前课程示例中的 person 结构体，如下所示：

```
type person struct {
   name string
   age int
}
func main() {
   //字面量初始化
   p:=person{name: "张三",age: 18}
}
```

该示例代码就是在声明这个 p 变量的时候，把它的 name 初始化为张三，age 初始化为 18。

指针变量初始化

在上个小节中，你已经知道了 new 函数可以申请内存并返回一个指向该内存的指针，但是这块内存中数据的值默认是该类型的零值，在一些情况下并不满足业务需求。比如我想得到一个 *person 类型的指针，并且它的 name 是飞雪无情、age 是 20，但是 new 函数只有一个类型参数，并没有初始化值的参数，此时该怎么办呢？

要达到这个目的，你可以自定义一个函数，对指针变量进行初始化，如下所示：

ch14/main.go

```
func NewPerson() *person{
   p:=new(person)
   p.name = "飞雪无情"
   p.age = 20
   return p
}
```

还记得前面课程讲的工厂函数吗？这个代码示例中的 NewPerson 函数就是工厂函数，除了使用 new 函数创建一个 person 指针外，还对它进行了赋值，也就是初始化。这样

NewPerson 函数的使用者就会得到一个 name 为飞雪无情、age 为 20 的 *person 类型的指针，通过 NewPerson 函数做一层包装，把内存分配（new 函数）和初始化（赋值）都完成了。

下面的代码就是使用 NewPerson 函数的示例，它通过打印 *pp 指向的数据值，来验证 name 是否是飞雪无情，age 是否是 20。

```
pp:=NewPerson()
fmt.Println("name为",pp.name,",age为",pp.age)
```

为了让自定义的工厂函数 NewPerson 更加通用，我把它改造一下，让它可以接受 name 和 age 参数，如下所示：

ch14/main.go

```
pp:=NewPerson("飞雪无情",20)
func NewPerson(name string,age int) *person{
   p:=new(person)
   p.name = name
   p.age = age
   return p
}
```

这些代码的效果和刚刚的示例一样，但是 NewPerson 函数更通用，因为你可以传递不同的参数，构建出不同的 *person 变量。

make 函数

铺垫了这么多，终于讲到今天的第二个主角 make 函数了。在上节课中你已经知道，在使用 make 函数创建 map 的时候，其实调用的是 makemap 函数，如下所示：

src/runtime/map.go

```
// makemap implements Go map creation for make(map[k]v, hint).
func makemap(t *maptype, hint int, h *hmap) *hmap{
  //省略无关代码
}
```

makemap 函数返回的是 *hmap 类型，而 hmap 是一个结构体，它的定义如下面的代码所示：

src/runtime/map.go

```
// A header for a Go map.
type hmap struct {
   // Note: the format of the hmap is also encoded in cmd/compile/internal/
   // Make sure this stays in sync with the compiler's definition.
   count     int // # live cells == size of map.  Must be first (used by le
   flags     uint8
   B         uint8  // log_2 of # of buckets (can hold up to loadFactor * 2
   noverflow uint16 // approximate number of overflow buckets; see incrnove
   hash0     uint32 // hash seed
   buckets    unsafe.Pointer // array of 2^B Buckets. may be nil if count==
   oldbuckets unsafe.Pointer // previous bucket array of half the size, non
   nevacuate  uintptr        // progress counter for evacuation (buckets le
   extra *mapextra // optional fields
}
```

可以看到，我们平时使用的 map 关键字其实非常复杂，它包含 map 的大小 count、存储桶 buckets 等。要想使用这样的 hmap，不是简单地通过 new 函数返回一个 *hmap 就可以，还需要对其进行初始化，这就是 make 函数要做的事情，如下所示：

```
m:=make(map[string]int,10)
```

是不是发现 make 函数和上一小节中自定义的 NewPerson 函数很像？其实 make 函数就是 map 类型的工厂函数，它可以根据传递它的 K-V 键值对类型，创建不同类型的 map，同时可以初始化 map 的大小。

> 小提示：make 函数不只是 map 类型的工厂函数，还是 chan、slice 的工厂函数。它同时可以用于 slice、chan 和 map 这三种类型的初始化。

总结

通过这节课的讲解，相信你已经理解了函数 new 和 make 的区别，现在我再来总结一下。

new 函数只用于分配内存，并且把内存清零，也就是返回一个指向对应类型零值的指针。new 函数一般用于需要显式地返回指针的情况，不是太常用。

make 函数只用于 slice、chan 和 map 这三种内置类型的创建和初始化，因为这三种类型的结构比较复杂，比如 slice 要提前初始化好内部元素的类型，slice 的长度和容量等，这样才可以更好地使用它们。

在这节课的最后，给你留一个练习题：使用 make 函数创建 slice，并且使用不同的长度和容量作为参数，看看它们的效果。

下节课我将介绍“运行时反射：字符串和结构体之间如何转换？”记得来听课！

上一页 下一页