



下载APP

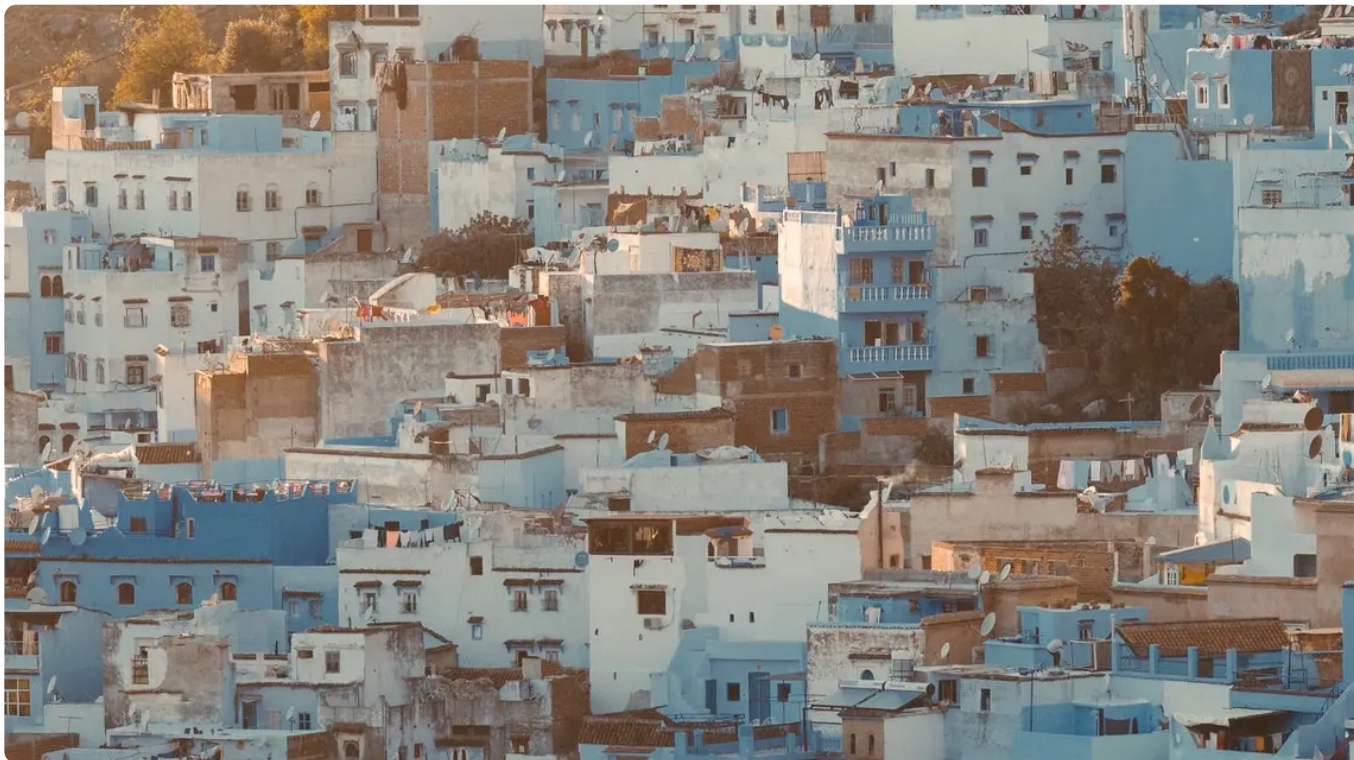


## 17 | 复合数据类型：用结构体建立对真实世界的抽象

2021-11-19 Tony Bai

《Tony Bai · Go语言第一课》

课程介绍 &gt;

**讲述：Tony Bai**

时长 29:10 大小 26.72M



你好，我是 Tony Bai。

在前面的几节课中，我们一直在讲数据类型，包括 Go 基本数据类型和三个复合数据类型。我们可以用这些数据类型来建立对真实世界的抽象。

那么什么是对真实世界的抽象呢？我们编写程序的目的就是与真实世界交互，解决真实世界的问题，帮助真实世界提高运行效率与改善运行质量。所以我们就需要对真实世界事物体的重要属性进行提炼，并映射到程序世界中，这就是所谓的对真实世界的抽象。

领资料



不同的数据类型具有不同的抽象能力，比如整数类型 `int` 可以用来抽象一个真实世界物体的长度，`string` 类型可以用来抽象真实世界物体的名字，等等。

但是光有这些类型的抽象能力还不够，我们还缺少一种通用的、对实体对象进行聚合抽象的能力。你可以回想一下，我们目前可以用学过的各种类型抽象出书名、书的页数以及书的索引，但

有没有一种类型，可以抽象出聚合了上述属性的“书”这个实体对象呢？

有的。在 Go 中，提供这种聚合抽象能力的类型是结构体类型，也就是 `struct`。这一节课，我们就围绕着结构体的使用和内存表示，由外及里来学习 Go 中的结构体类型。

不过，在学习如何定义一个结构体类型之前，我们首先要来看看如何在 Go 中自定义一个新类型。有了这个基础，我们再理解结构体类型的定义方法就十分自然了。

## 如何自定义一个新类型？

在 Go 中，我们自定义一个新类型一般有两种方法。**第一种是类型定义 (Type Definition)**，这也是我们最常用的类型定义方法。在这种方法中，我们会使用关键字 **`type`** 来定义一个新类型 `T`，具体形式是这样的：

```
1 type T S // 定义一个新类型T
```

[复制代码](#)

在这里，`S` 可以是任何一个已定义的类型，包括 Go 原生类型，或者是其他已定义的自定义类型，我们来演示一下这两种情况：

```
1 type T1 int
2 type T2 T1
```

[复制代码](#)

这段代码中，新类型 `T1` 是基于 Go 原生类型 `int` 定义的新自定义类型，而新类型 `T2` 则是基于刚刚定义的类型 `T1`，定义的新类型。

这里我们引入一个新概念，**底层类型**。如果一个新类型是基于某个 Go 原生类型定义的，那么我们就叫 Go 原生类型为新类型的**底层类型 (Underlying Type)**。比如这个例子中，类型 `int` 就是类型 `T1` 的底层类型。

那如果不是基于 Go 原生类型定义的新类型，比如 T2，它的底层类型是什么呢？这时我们就要看它定义时是基于什么类型了。这里，T2 是基于 T1 类型创建的，那么 T2 类型的底层类型就是 T1 的底层类型，而 T1 的底层类型我们已经知道了，是类型 int，那么 T2 的底层类型也是类型 int。

为什么我们要提到底层类型这个概念呢？因为底层类型在 Go 语言中有重要作用，**它被用来判断两个类型本质上是否相同 (Identical)**。

在上面例子中，虽然 T1 和 T2 是不同类型，但因为它们的底层类型都是类型 int，所以它们在本质上是相同的。**而本质上相同的两个类型，它们的变量可以通过显式转型进行相互赋值，相反，如果本质上是不同的两个类型，它们的变量间连显式转型都不可能，更不要说相互赋值了。**

比如你可以看看这个代码示例：

[复制代码](#)

```
1 type T1 int
2 type T2 T1
3 type T3 string
4
5 func main() {
6     var n1 T1
7     var n2 T2 = 5
8     n1 = T1(n2) // ok
9
10    var s T3 = "hello"
11    n1 = T1(s) // 错误：cannot convert s (type T3) to type T1
12 }
```

这段代码中，T1 和 T2 本质上是相同的类型，所以我们可以将 T2 变量 n2 的值，通过显式转型赋值给 T1 类型变量 n1。而类型 T3 的底层类型为类型 string，与 T1/T2 的底层类型不同，所以它们本质上就不是相同的类型。这个时候，如果我们把 T3 类型变量 s 赋值给 T1 类型变量 n1，编译器就会给出编译错误的提示。

除了基于已有类型定义新类型之外，我们还可以基于**类型字面值**来定义新类型，这种方式多用于自定义一个新的复合类型，比如：

```
1 type M map[int]string
2 type S []string
```

[复制代码](#)

和变量声明支持使用 var 块的方式类似，类型定义也支持通过 type 块的方式进行，比如我们可以把上面代码中的 T1、T2 和 T3 的定义放在同一个 type 块中：

```
1 type (
2     T1 int
3     T2 T1
4     T3 string
5 )
```

[复制代码](#)

**第二种自定义新类型的方式是使用类型别名（Type Alias）**，这种类型定义方式通常用在项目的渐进式重构，还有对已有包的二次封装方面，它的形式是这样的：

```
1 type T = S // type alias
```

[复制代码](#)

我们看到，与前面的第一种类型定义相比，类型别名的形式只是多了一个等号，但正是这个等号让新类型 T 与原类型 S 完全等价。完全等价的意思就是，类型别名并没有定义出新类型，类 T 与 S 实际上就是**同一种类型**，它们只是一种类型的两个名字罢了，就像一个人有一个大名、一个小名一样。我们看下面这个简单的例子：

```
1 type T = string
2
3 var s string = "hello"
4 var t T = s // ok
5 fmt.Printf("%T\n", t) // string
```

[复制代码](#)

因为类型 T 是通过类型别名的方式定义的，T 与 string 实际上是一个类型，所以这里，使用 string 类型变量 s 给 T 类型变量 t 赋值的动作，实质上就是同类型赋值。另外我们也可以看到，通过 Printf 输出的变量 t 的类型信息也是 string，这我们的预期也是一致的。

学习了两种新类型的自定义方法后，我们再来看一下如何定义一个结构体类型。

## 如何定义一个结构体类型？

我们前面说了，复合类型的定义一般都是通过类型字面值的方式来进行的，作为复合类型之一的结构体类型也不例外，下面就是一个典型的结构体类型的定义形式：

```
1 type T struct {  
2     Field1 T1  
3     Field2 T2  
4     ... ..  
5     FieldN Tn  
6 }
```

[复制代码](#)

根据这个定义，我们会得到一个名为 T 的结构体类型，定义中 struct 关键字后面的大括号包裹的内容就是一个**类型字面值**。我们看到这个类型字面值由若干个字段（field）聚合而成，每个字段有自己的名字与类型，并且在一个结构体中，每个字段的名称应该都是唯一的。

通过聚合其他类型字段，结构体类型展现出强大而灵活的抽象能力。我们直接上案例实操，来说明一下。

我们前面提到过对现实世界的书进行抽象的情况，其实用结构体类型就可以实现，比如这里，我就用前面的典型方法定义了一个结构体：


```
1 package book  
2  
3 type Book struct {  
4     Title string           // 书名  
5     Pages int              // 书的页数  
6     Indexes map[string]int // 书的索引  
7 }
```

[复制代码](#)

在这个结构体定义中，你会发现，我在类型 Book，还有它的各个字段中都用了首字母大写的名字。这是为什么呢？



你回忆一下，我们在第 11 讲中曾提到过，Go 用标识符名称的首字母大小写来判定这个标识符是否为导出标识符。所以，这里的类型 `Book` 以及它的各个字段都是导出标识符。这样，只要其他包导入了包 `book`，我们就可以在这些包中直接引用类型名 `Book`，也可以通过 `Book` 类型变量引用 `Name`、`Pages` 等字段，就像下面代码中这样：

 复制代码

```
1 import ".../book"
2
3 var b book.Book
4 b.Title = "The Go Programming Language"
5 b.Pages = 800
```

如果结构体类型只在它定义的包内使用，那么我们可以将类型名的首字母小写；如果你不想将结构体类型中的某个字段暴露给其他包，那么我们同样可以把这个字段名字的首字母小写。

我们还可以用**空标识符 “\_” 作为结构体类型定义中的字段名称**。这样以空标识符为名称的字段，不能被外部包引用，甚至无法被结构体所在的包使用。那这么做有什么实际意义呢？这里先留个悬念，你可以自己先思考一下，我们在后面讲解结构体类型的内存布局时，会揭晓答案。

除了通过类型字面值来定义结构体这种典型操作外，我们还有另外几种特殊的情况。


### 第一种：定义一个空结构体。

我们可以定义一个空结构体，也就是没有包含任何字段的结构体类型，就像下面示例代码这样：

 复制代码

```
1 type Empty struct{} // Empty是一个不包含任何字段的空结构体类型
```

空结构体类型有什么用呢？我们继续看下面代码：

 复制代码

```
1 var s Empty
```

```
2 println(unsafe.Sizeof(s)) // 0
```

我们看到，输出的空结构体类型变量的大小为 0，也就是说，空结构体类型变量的内存占用为 0。基于空结构体类型内存零开销这样的特性，我们在日常 Go 开发中会经常使用空结构体类型元素，作为一种“事件”信息进行 Goroutine 之间的通信，就像下面示例代码这样：

[复制代码](#)

```
1 var c = make(chan Empty) // 声明一个元素类型为Empty的channel
2 c<-Empty{}              // 向channel写入一个“事件”
```

这种以空结构体为元素类建立的 channel，是目前能实现的、内存占用最小的 Goroutine 间通信方式。

## 第二种情况：使用其他结构体作为自定义结构体中字段的类型。

我们看这段代码，这里结构体类型 Book 的字段 Author 的类型，就是另外一个结构体类型 Person：

[复制代码](#)


```
1 type Person struct {
2     Name string
3     Phone string
4     Addr string
5 }
6
7 type Book struct {
8     Title string
9     Author Person
10    ... ..
11 }
```

如果我们要访问 Book 结构体字段 Author 中的 Phone 字段，我们可以这样操作：

[复制代码](#)


```
1 var book Book
2 println(book.Author.Phone)
```

不过，对于包含结构体类型字段的结构体类型来说，Go 还提供了一种更为简便的定义方法，**那就是我们可以无需提供字段的名字，只需要使用其类型就可以了**，以上面的 Book 结构体定义为例，我们可以用下面的方式提供一个等价的定义：

 复制代码

```
1 type Book struct {  
2     Title string  
3     Person  
4     ... ..  
5 }
```


以这种方式定义的结构体字段，我们叫做嵌入字段（Embedded Field）。我们也可以将这种字段称为匿名字段，或者把类型名看作是这个字段的名字。如果我们要访问 Person 中的 Phone 字段，我们可以通过下面两种方式进行：

 复制代码

```
1 var book Book  
2 println(book.Person.Phone) // 将类型名当作嵌入字段的名字  
3 println(book.Phone)        // 支持直接访问嵌入字段所属类型中字段
```

第一种方式显然是通过把类型名当作嵌入字段的名字来进行操作的，而第二种方式更像是一种“语法糖”，我们可以“绕过”Person 类型这一层，直接访问 Person 中的字段。关于这种“类型嵌入”特性，我们在以后的课程中还会详细说明，这里就先不深入了。

不过，看到这里，关于结构体定义，你可能还有一个疑问，**在结构体类型 T 的定义中是否可以包含类型为 T 的字段呢？**比如这样：


 复制代码

```
1 type T struct {  
2     t T  
3     ... ..  
4 }
```

答案是不可以的。Go 语言不支持这种在结构体类型定义中，递归地放入其自身类型字段的定义方式。面对上面的示例代码，编译器就会给出“invalid recursive type T”的错误信息。




同样，下面这两个结构体类型 T1 与 T2 的定义也存在递归的情况，所以这也是不合法的。

 复制代码

```
1 type T1 struct {  
2     t2 T2  
3 }  
4  
5 type T2 struct {  
6     t1 T1  
7 }
```

不过，虽然我们不能在结构体类型 T 定义中，拥有以自身类型 T 定义的字段，但我们却可以拥有自身类型的指针类型、以自身类型为元素类型的切片类型，以及以自身类型作为 value 类型的 map 类型的字段，比如这样：

 复制代码


```
1 type T struct {  
2     t *T           // ok  
3     st []T         // ok  
4     m map[string]T // ok  
5 }
```

你知道为什么这样的定义是合法的吗？我想把这个问题作为这节课的课后思考题留给你，你可以在留言区说一下你的想法。

关于结构体类型的知识我们已经学习得差不多了，接下来我们再来看看如何应用这些结构体类型来声明变量，并进行初始化。

## 结构体变量的声明与初始化

和其他所有变量的声明一样，我们也可以使用标准变量声明语句，或者是短变量声明语句声明一个结构体类型的变量：

 复制代码

```
1 type Book struct {  
2     ...  
3 }  
4  
5 var book Book
```

```
6 var book = Book{}  
7 book := Book{}
```

不过，这里要注意，我们在前面说过，结构体类型通常是对真实世界复杂事物的抽象，这和简单的数值、字符串、数组 / 切片等类型有所不同，**结构体类型的变量通常都要被赋予适当的初始值后，才会有合理的意义。**

接下来，我把结构体类型变量的初始化大致分为三种情况，我们逐一看一下。

## 零值初始化

零值初始化说的是使用结构体的零值作为它的初始值。在前面的课程中，“零值”这个术语反复出现过多次，它指的是一个类型的默认值。对于 Go 原生类型来说，这个默认值也称为零值。Go 结构体类型由若干个字段组成，当这个结构体类型变量的各个字段的值都是零值时，我们就说这个结构体类型变量处于零值状态。

前面提到过，结构体类型的零值变量，通常不具有或者很难具有合理的意义，比如通过下面代码得到的零值 book 变量就是这样：

```
1 var book Book // book为零值结构体变量
```

[复制代码](#)

你想象一下，一本书既没有书名，也没有作者、页数、索引等信息，那么通过 Book 类型对这本书的抽象就失去了实际价值。所以对于像 Book 这样的结构体类型，使用零值初始化并不是正确的选择。

那么采用零值初始化的零值结构体变量就真的没有任何价值了吗？恰恰相反。如果一种类型采用零值初始化得到的零值变量，是有意义的，而且是直接可用的，我称这种类型为“**零值可用**”类型。可以说，定义零值可用类型是简化代码、改善开发者使用体验的一种重要的手段。

在 Go 语言标准库和运行时的代码中，有很多践行“零值可用”理念的好例子，最典型的莫过于 sync 包的 Mutex 类型了。Mutex 是 Go 标准库中提供的、用于多个并发 Goroutine 之间进行同步的互斥锁。

运用“零值可用”类型，给 Go 语言中的线程互斥锁带来了什么好处呢？我们横向对比一下 C 语言中的做法你就知道了。如果我们要在 C 语言中使用线程互斥锁，我们通常需要这么做：

[复制代码](#)

```
1 pthread_mutex_t mutex;  
2 pthread_mutex_init(&mutex, NULL);  
3  
4 pthread_mutex_lock(&mutex);  
5 ...  
6 pthread_mutex_unlock(&mutex);
```

我们可以看到，在 C 中使用互斥锁，我们需要首先声明一个 mutex 变量。但这个时候，我们不能直接使用声明过的变量，因为它的零值状态是不可用的，我们必须使用 pthread\_mutex\_init 函数对其进行专门的初始化操作后，它才能处于可用状态。再之后，我们才能进行 lock 与 unlock 操作。

但是在 Go 语言中，我们只需要这几行代码就可以了：

[复制代码](#)

```
1 var mu sync.Mutex  
2 mu.Lock()  
3 mu.Unlock()
```

Go 标准库的设计者很贴心地将 sync.Mutex 结构体的零值状态，设计为可用状态，这样开发者便可直接基于零值状态下的 Mutex 进行 lock 与 unlock 操作，而且不需要额外显式地对它进行初始化操作了。

Go 标准库中的 bytes.Buffer 结构体类型，也是一个零值可用类型的典型例子，这里我演示了 bytes.Buffer 类型的常规用法：

[复制代码](#)

```
1 var b bytes.Buffer  
2 b.Write([]byte("Hello, Go"))  
3 fmt.Println(b.String()) // 输出：Hello, Go
```

你可以看到，我们不需要对 `bytes.Buffer` 类型的变量 `b` 进行任何显式初始化，就可以直接通过处于零值状态的变量 `b`，调用它的方法进行写入和读取操作。

不过有些类型确实不能设计为零值可用类型，就比如我们前面的 `Book` 类型，它们的零值并非有效值。对于这类类型，我们需要对它的变量进行显式的初始化后，才能正确使用。在日常开发中，对结构体类型变量进行显式初始化的最常用方法就是使用复合字面值，下面我们就来看看这种方法。

## 使用复合字面值

其实我们已经不是第一次接触复合字面值了，之前我们讲解数组 / 切片、`map` 类型变量的变量初始化的时候，都提到过用复合字面值的方法。

最简单的对结构体变量进行显式初始化的方式，就是**按顺序依次给每个结构体字段进行赋值**，比如下面的代码：

[复制代码](#)


```
1 type Book struct {  
2     Title string           // 书名  
3     Pages int              // 书的页数  
4     Indexes map[string]int // 书的索引  
5 }  
6  
7 var book = Book{"The Go Programming Language", 700, make(map[string]int)}
```

我们依然可以用这种方法给结构体的每一个字段依次赋值，但这种方法也有很多问题：

首先，当结构体类型定义中的字段顺序发生变化，或者字段出现增删操作时，我们就需要手动调整该结构体类型变量的显式初始化代码，让赋值顺序与调整后的字段顺序一致。

其次，当一个结构体的字段较多时，这种逐一字段赋值的方式实施起来就会比较困难，而且容易出错，开发人员需要来回对照结构体类型中字段的类型与顺序，谨慎编写字面值表达式。

最后，一旦结构体中包含非导出字段，那么这种逐一字段赋值的方式就不再被支持了，编译器会报错：


 复制代码

```
1 type T struct {
2     F1 int
3     F2 string
4     f3 int
5     F4 int
6     F5 int
7 }
8
9 var t = T{11, "hello", 13} // 错误: implicit assignment of unexported field 'f3'
10 或
11 var t = T{11, "hello", 13, 14, 15} // 错误: implicit assignment of unexported f
```

事实上，Go 语言并不推荐我们按字段顺序对一个结构体类型变量进行显式初始化，甚至 Go 官方还在提供的 go vet 工具中专门内置了一条检查规则：[🔗 "composites"](#)，用来静态检查代码中结构体变量初始化是否使用了这种方法，一旦发现，就会给出警告。

### 那么我们应该用哪种形式的复合字面值给结构体变量赋初值呢？


Go 推荐我们用 **“field:value” 形式的复合字面值**，对结构体类型变量进行显式初始化，这种方式可以降低结构体类型使用者和结构体类型设计者之间的耦合，这也是 Go 语言的惯用法。这里，我们用 “field:value” 形式复合字面值，对上面的类型 T 的变量进行初始化看看：

 复制代码

```
1 var t = T{
2     F2: "hello",
3     F1: 11,
4     F4: 14,
5 }
```


我们看到，使用这种 “field:value” 形式的复合字面值对结构体类型变量进行初始化，非常灵活。和之前的顺序复合字面值形式相比，“field:value” 形式字面值中的字段可以以任意次序出现。未显式出现在字面值中的结构体字段（比如上面例子中的 F5）将采用它对应类型的零值。

复合字面值作为结构体类型变量初值被广泛使用，即便结构体采用类型零值时，我们也会使用复合字面值的形式：

 复制代码

```
1 t := T{}
```

而比较少使用 `new` 这一个 Go 预定义的函数来创建结构体变量实例：

 复制代码


```
1 tp := new(T)
```

这里值得我们注意的是，我们不能用从其他包导入的结构体中的未导出字段，来作为复合字面值中的 field。这会导致编译错误，因为未导出字段是不可见的。

那么，如果一个结构体类型中包含未导出字段，并且这个字段的零值还不可用时，我们要如何初始化这个结构体类型的变量呢？又或是一个结构体类型中的某些字段，需要一个复杂的初始化逻辑，我们又该怎么做呢？这时我们就需要使用一个特定的构造函数，来创建并初始化结构体变量了。

## 使用特定的构造函数


其实，使用特定的构造函数创建并初始化结构体变量的例子，并不罕见。在 Go 标准库中就有很多，其中 `time.Timer` 这个结构体就是一个典型的例子，它的定义如下：

 复制代码

```
1 // $GOROOT/src/time/sleep.go
2 type runtimeTimer struct {
3     pp      uintptr
4     when    int64
5     period  int64
6     f       func(interface{}, uintptr)
7     arg     interface{}
8     seq     uintptr
9     nextwhen int64
10    status  uint32
11 }
12
13 type Timer struct {
14     C <-chan Time
15     r runtimeTimer
16 }
```




我们看到，Timer 结构体中包含了一个非导出字段 `r`，`r` 的类型为另外一个结构体类型 `runtimeTimer`。这个结构体更为复杂，而且我们一眼就可以看出来，这个 `runtimeTimer` 结构体不是零值可用的，那我们在创建一个 Timer 类型变量时就没法使用显式复合字面值的方式了。这个时候，Go 标准库提供了一个 Timer 结构体专用的构造函数 `NewTimer`，它的实现如下：

 复制代码

```
1 // $GOROOT/src/time/sleep.go
2 func NewTimer(d Duration) *Timer {
3     c := make(chan Time, 1)
4     t := &Timer{
5         C: c,
6         r: runtimeTimer{
7             when: when(d),
8             f:     sendTime,
9             arg:  c,
10        },
11    }
12    startTimer(&t.r)
13    return t
14 }
```

我们看到，`NewTimer` 这个函数只接受一个表示定时时间的参数 `d`，在经过一个复杂的初始化过程后，它返回了一个处于可用状态的 Timer 类型指针实例。

像这类通过专用构造函数进行结构体类型变量创建、初始化的例子还有很多，我们可以总结一下，它们的专用构造函数大多都符合这种模式：

 复制代码

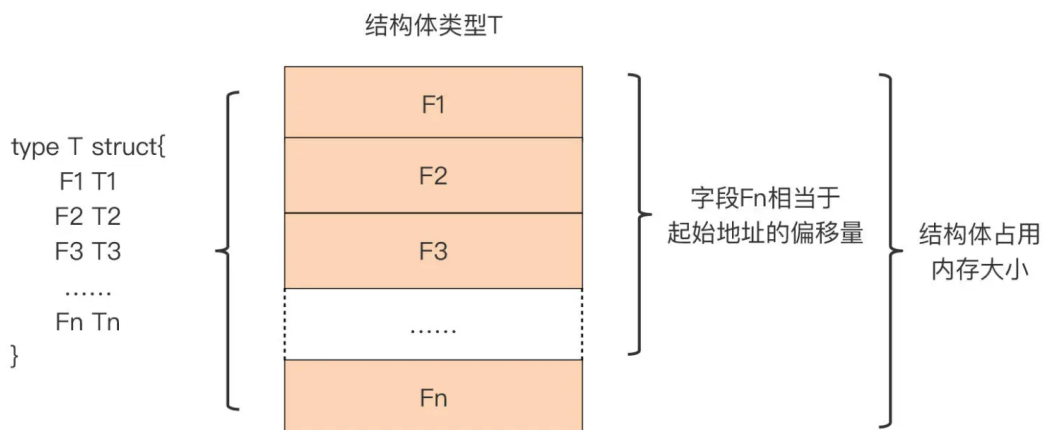
```
1 func NewT(field1, field2, ...) *T {
2     ... ..
3 }
```

这里，`NewT` 是结构体类型 `T` 的专用构造函数，它的参数列表中的参数通常与 `T` 定义中的导出字段相对应，返回值则是一个 `T` 指针类型的变量。`T` 的非导出字段在 `NewT` 内部进行初始化，一些需要复杂初始化逻辑的字段也会在 `NewT` 内部完成初始化。这样，我们只要调用 `NewT` 函数就可以得到一个可用的 `T` 指针类型变量了。

和之前学习复合数据类型的套路一样，接下来，我们再回到结构体类型的定义，看看结构体类型在内存中的表示，也就是内存布局。

## 结构体类型的内存布局

Go 结构体类型是既数组类型之后，第二个将它的元素（结构体字段）一个接着一个以“平铺”形式，存放在一个连续内存块中的。下图是一个结构体类型 T 的内存布局：

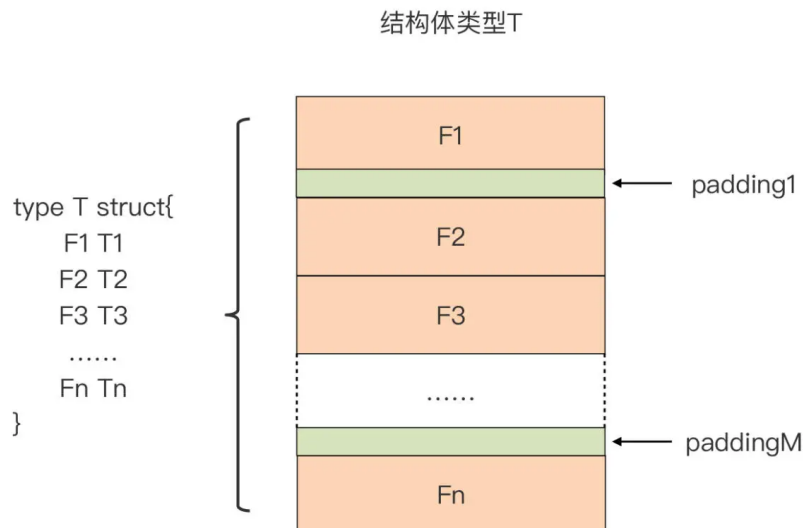


我们看到，结构体类型 T 在内存中布局是非常紧凑的，Go 为它分配的内存都用来存储字段了，没有被 Go 编译器插入的额外字段。我们可以借助标准库 `unsafe` 包提供的函数，获得结构体类型变量占用的内存大小，以及它每个字段在内存中相对于结构体变量起始地址的偏移量：

复制代码

```
1 var t T
2 unsafe.Sizeof(t)           // 结构体类型变量占用的内存大小
3 unsafe.Offsetof(t.Fn)     // 字段Fn在内存中相对于变量t起始地址的偏移量
```

不过，上面这张示意图是比较理想的状态，真实的情况可能就没那么好了：



在真实情况下，虽然 Go 编译器没有在结构体变量占用的内存空间中插入额外字段，但结构体字段实际上可能并不是紧密相连的，中间可能存在“缝隙”。这些“缝隙”同样是结构体变量占用的内存空间的一部分，它们是 Go 编译器插入的“填充物（Padding）”。


那么，Go 编译器为什么要在结构体的字段间插入“填充物”呢？这其实是**内存对齐**的要求。所谓内存对齐，指的就是各种内存对象的内存地址不是随意确定的，必须满足特定要求。

对于各种基本数据类型来说，它的变量的内存地址值必须是其类型本身大小的整数倍，比如，一个 int64 类型的变量的内存地址，应该能被 int64 类型自身的大小，也就是 8 整除；一个 uint16 类型的变量的内存地址，应该能被 uint16 类型自身的大小，也就是 2 整除。

这些基本数据类型的对齐要求很好理解，那么像结构体类型这样的复合数据类型，内存对齐又是怎么要求的呢？是不是它的内存地址也必须是它类型大小的整数倍呢？

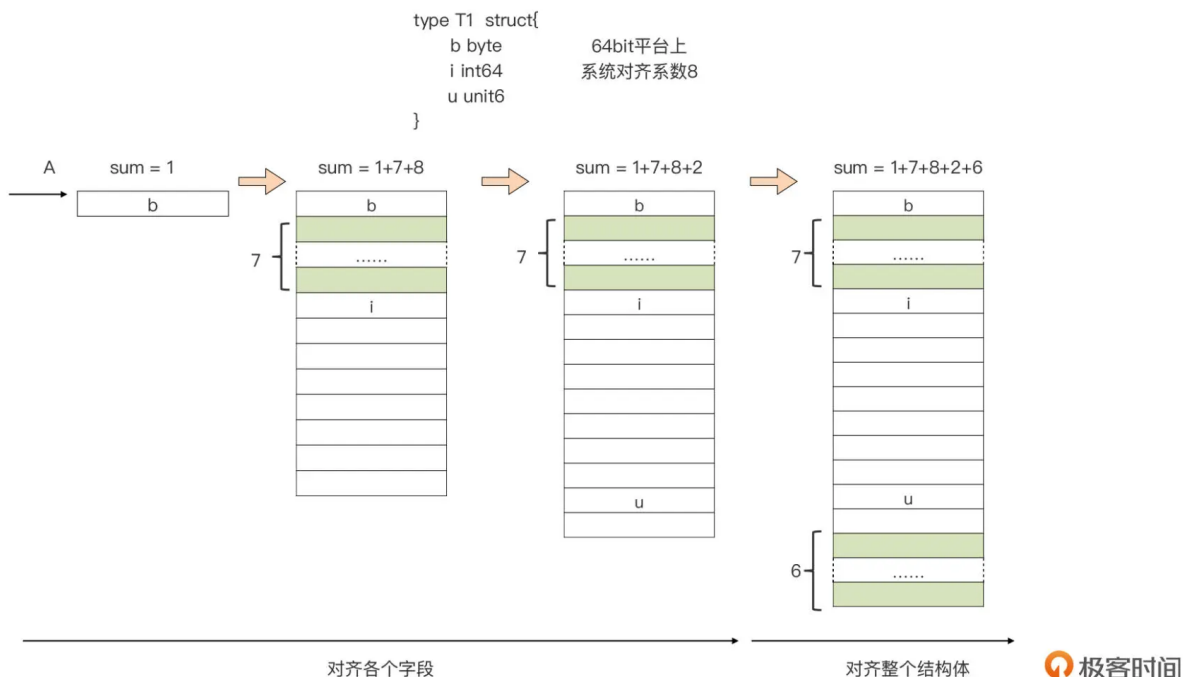
实际上没有这么严格。对于结构体而言，它的变量的内存地址，只要是它最长字段长度与系统对齐系数两者之间较小的那个的整数倍就可以了。但对于结构体类型来说，我们还要让它每个字段的内存地址都严格满足内存对齐要求。

这么说依然比较绕，我们来看一个具体例子，计算一下这个结构体类型 T 的对齐系数：

 复制代码

```
1 type T struct {  
2     b byte  
3     u uint16  
4     i int64  
5 }
```

计算过程是这样的：



我们简单分析一下，整个计算过程分为两个阶段。**第一个阶段是对齐结构体的各个字段。**

首先，我们看第一个字段 `b` 是长度 1 个字节的 `byte` 类型变量，这样字段 `b` 放在任意地址上都可以被 1 整除，所以我们说它是天生对齐的。我们用一个 `sum` 来表示当前已经对齐的内存空间的大小，这个时候 `sum=1`；

接下来，我们看第二个字段 `i`，它是一个长度为 8 个字节的 `int64` 类型变量。按照内存对齐要求，它应该被放在可以被 8 整除的地址上。但是，如果把 `i` 紧邻 `b` 进行分配，当 `i` 的地址可以被 8 整除时，`b` 的地址就无法被 8 整除。这个时候，我们需要在 `b` 与 `i` 之间做一些填充，使得 `i` 的地址可以被 8 整除时，`b` 的地址也始终可以被 8 整除，于是我们在 `i` 与 `b` 之间填充了 7 个字节，此时此刻 `sum=1+7+8`；

再下来，我们看第三个字段 `u`，它是一个长度为 2 个字节的 `uint16` 类型变量，按照内存对其要求，它应该被放在可以被 2 整除的地址上。有了对其的 `i` 作为基础，我们现在知道将 `u` 与 `i` 相邻而放，是可以满足其地址的对齐要求的。`i` 之后的那个字节的地址肯定可以被 8 整除，也一定可以被 2 整除。于是我们把 `u` 直接放在 `i` 的后面，中间不需要填充，此时此刻， $\text{sum} = 1 + 7 + 8 + 2$ 。

**现在结构体 `T` 的所有字段都已经对齐里，我们开始第二个阶段，也就是对齐整个结构体。**

我们前面提到过，结构体的内存地址为  $\min(\text{结构体最长字段的长度}, \text{系统内存对齐系数})$  的整数倍，那么这里结构体 `T` 最长字段为 `i`，它的长度为 8，而 64bit 系统上的系统内存对齐系数一般为 8，两者相同，我们取 8 就可以了。那么整个结构体的对齐系数就是 8。

这个时候问题就来了！为什么上面的示意图还要在结构体的尾部填充了 6 个字节呢？

我们说过结构体 `T` 的对齐系数是 8，那么我们就要保证每个结构体 `T` 的变量的内存地址，都能被 8 整除。如果我们只分配一个 `T` 类型变量，不再继续填充，也可能保证其内存地址为 8 的倍数。但如果考虑我们分配的是一个元素为 `T` 类型的数组，比如下面这行代码，我们虽然可以保证 `T[0]` 这个元素地址可以被 8 整除，但能保证 `T[1]` 的地址也可以被 8 整除吗？

 复制代码

```
1 var array [10]T
```

我们知道，数组是元素连续存储的一种类型，元素 `T[1]` 的地址为 `T[0]` 地址 + `T` 的大小 (18)，显然无法被 8 整除，这将导致 `T[1]` 及后续元素的地址都无法对齐，这显然不能满足内存对齐的要求。

问题的根源在哪里呢？问题就在于 `T` 的当前大小为 18，这是一个不能被 8 整除的数值，如果 `T` 的大小可以被 8 整除，那问题就解决了。于是我们才有了最后一个步骤，我们从 18 开始向后找到第一个可以被 8 整除的数字，也就是将 18 圆整到 8 的倍数上，我们得到 24，我们将 24 作为类型 `T` 最终的大小就可以了。

为什么会出现内存对齐的要求呢？这是出于对处理器存取数据效率的考虑。在早期的一些处理器中，比如 Sun 公司的 Sparc 处理器仅支持内存对齐的地址，如果它遇到没有对齐的内存地址，会引发段错误，导致程序崩溃。我们常见的 x86-64 架构处理器虽然处理未对齐的内存地址不会出现段错误，但数据的存取性能也会受到影响。

从这个推演过程中，你应该已经知道了，Go 语言中结构体类型的大小受内存对齐约束的影响。这样一来，不同的字段排列顺序也会影响到“填充字节”的多少，从而影响到整个结构体大小。比如下面两个结构体类型表示的抽象是相同的，但正是因为字段排列顺序不同，导致它们的大小也不同：

 复制代码

```
1 type T struct {
2     b byte
3     i int64
4     u uint16
5 }
6
7 type S struct {
8     b byte
9     u uint16
10    i int64
11 }
12
13 func main() {
14     var t T
15     println(unsafe.Sizeof(t)) // 24
16     var s S
17     println(unsafe.Sizeof(s)) // 16
18 }
```

所以，你在日常定义结构体时，一定要注意结构体中字段顺序，尽量合理排序，降低结构体对内存空间的占用。

另外，前面例子中的内存填充部分，是由编译器自动完成的。不过，有些时候，为了保证某个字段的内存地址有更为严格的约束，我们也会做主动填充。比如 runtime 包中的 mstats 结构体定义就采用了主动填充：

 复制代码

```
1 // $GOROOT/src/runtime/mstats.go
2 type mstats struct {
```



```
3     ... ..
4     // Add an uint32 for even number of size classes to align below fields
5     // to 64 bits for atomic operations on 32 bit platforms.
6     _ [1 - _NumSizeClasses%2]uint32 // 这里做了主动填充
7
8     last_gc_nanotime uint64 // last gc (monotonic time)
9     last_heap_inuse  uint64 // heap_inuse at mark termination of the previous
10    ... ..
11 }
```

通常我们会通过空标识符来进行主动填充，因为填充的这部分内容我们并不关心。关于主动填充的话题不是我们这节课的重点，我就介绍到这里了。如果你对这个话题感兴趣，你也可以自行阅读相关资料进行扩展学习，并在留言区和我们分享。

## 小结

好了，今天的课讲到这里就结束了，现在我们一起来回顾一下吧。

通过前面的学习我们知道，Go 语言不是一门面向对象范式的编程语言，它没有 C++ 或 Java 中的那种 class 类型。如果非要在 Go 中选出一个与 class 接近的语法元素，那非结构体类型莫属。Go 中的结构体类型提供了一种聚合抽象能力，开发者可以使用它建立对真实世界的事物的抽象。

在讲解结构体相关知识前，我们在先介绍了如何自定义一个新类型，通常会使用类型定义这种标准方式定义新类型另外，我们还可以用类型别名的方式自定义类型，你要多注意着这两种方式的区别。

对于结构体这类复合类型，我们通过类型字面值方式来定义，它包含若干个字段，每个字段都有自己的名字与类型。如果不包含任何字段，我们称这个结构体类型为“空结构体类型”，空结构体类型的变量不占用内存空间，十分适合作为一种“事件”在并发的 Goroutine 间传递。

当我们使用结构体类型作为字段类型时，Go 还提供了“嵌入字段”的语法糖，关于这种嵌入方式，我们在后续的课程中还会有更详细的讲解。另外，Go 的结构体定义不支持递归，这点你一定要注意。

结构体类型变量的初始化有几种方式：零值初始化、复合字面值初始化，以及使用特定构造函数进行初始化，日常编码中最常见的是第二种。支持零值可用的结构体类型对于简化

代码，改善体验具有很好的作用。另外，当复合字面值初始化无法满足要求的情况下，我们需要为结构体类型定义专门的构造函数，这种方式同样有广泛的应用。

结构体类型是既数组类型之后，又一个以平铺形式存放在连续内存块中的类型。不过与数组类型不同，由于内存对齐的要求，结构体类型各个相邻字段间可能存在“填充物”，结构体的尾部同样可能被 Go 编译器填充额外的字节，满足结构体整体对齐的约束。正是因为这点，我们在定义结构体时，一定要合理安排字段顺序，要让结构体类型对内存空间的占用最小。

关于结构体类型的知识点比较多，你先消化一下。在后面讲解方法的时候，我们还会继续讲解与结构体类型有关的内容。

## 思考题

Go 语言不支持在结构体类型定义中，递归地放入其自身类型字段，但却可以拥有自身类型的指针类型、以自身类型为元素类型的切片类型，以及以自身类型作为 value 类型的 map 类型的字段，你能思考一下其中的原因吗？期待在留言区看到你的想法。

欢迎你把这节课分享给更多对 Go 复合数据类型感兴趣的朋友。我是 Tony Bai，我们下节课见。

分享给需要的人，Ta 订阅后你可得 **20 元现金奖励**

 生成海报并分享

 赞 0  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 16 | 复合数据类型：原生map类型的实现机制是怎样的？

更多学习推荐

# 2021 最新大厂 Go 工程师面试真题

大厂面试真题 + 金九银十全新整理 + 核心知识全覆盖

免费领取 

## 精选留言 (5)

 写留言**功夫熊猫**

2021-11-19

因为指针的值是变量的地址，而变量的地址是一种新的数据类型。

 2**liaomars**

2021-11-19

type T struct { t T ... ... } 这种方式，t T是一个新的自定义数据类型了，而可以接受 指针，切片这些，因为本质上还是指向底层数据是一样的，不知道这样理解对不对。

**dollboy**

2021-11-19

因为指针、map、切片的变量元数据的内存占用大小是固定的。

**小豆子** 

2021-11-19

声明 结构体变量时 需要分配内存，指针/切片/map类型 针对特定架构 占用的内存是已知

的，与底层具体类型无关。

展开 ▾



罗杰

2021-11-19

内存对齐要好好理解

展开 ▾

