详解 Go 空结构体的 3 种使用场景

在 Go 语言中，有一个比较特殊的类型，经常会有刚接触 Go 的小伙伴问到，又或是不理解。

他就是 Go 里的空结构体（struct）的使用，常常会有看到有人使用：

```Go
ch := make(chan struct{})
```

还清一色的使用结构体，也不用其他类型。高度常见，也就不是一个偶发现象了，肯定是背后必然有什么原因。

今天煎鱼这篇文章带大家了解一下为什么要这么用，知其然知其所以然。

一起愉快地开始吸鱼之路。

为什么使用

说白了，就是希望节省空间。但，新问题又来了，为什么不能用其他的类型来做？

这就涉及到在 Go 语言中 ”宽度“ 的概念，宽度描述了一个类型的实例所占用的存储空间的字节数。

宽度是一个类型的属性。在 Go 语言中的每个值都有一个类型，值的宽度由其类型定义，并且总是 8 bits 的倍数。

在 Go 语言中我们可以借助 `unsafe.Sizeof` 方法，来获取：

```Go
// Sizeof takes an expression x of any type and returns the size in bytes
// of a hypothetical variable v as if v was declared via var v = x.
// The size does not include any memory possibly referenced by x.
// For instance, if x is a slice, Sizeof returns the size of the slice
// descriptor, not the size of the memory referenced by the slice.
// The return value of Sizeof is a Go constant.
func Sizeof(x ArbitraryType) uintptr
```

该方法能够得到值的宽度，自然而然也就能知道其类型对应的宽度是多少了。

我们对应看看 Go 语言中几种常见的类型宽度大小：

Go

```go
func main() {
    var a int
    var b string
    var c bool
    var d [3]int32
    var e []string
    var f map[string]bool
    fmt.Println(
        unsafe.Sizeof(a),
        unsafe.Sizeof(b),
        unsafe.Sizeof(c),
        unsafe.Sizeof(d),
        unsafe.Sizeof(e),
        unsafe.Sizeof(f),
    )
}
```

输出结果：

Apache

```
8 16 1 12 24 8
```

你可以发现我们列举的几种类型，只是单纯声明，我们也啥没干，依然占据一定的宽度。

如果我们的场景，只是占位符，那怎么办，系统里的开销就这么白白浪费了？

空结构体的特殊性

空结构体在各类系统中频繁出现的原因之一，就是需要一个占位符。而恰恰好，Go 空结构体的宽度是特殊的。

如下：

Go

```go
func main() {
    var s struct{}
    fmt.Println(unsafe.Sizeof(s))
}
```

输出结果：

Go

```
0
```

空结构体的宽度是很直接了当的 0，即便是变形处理：

Go

```
type S struct {
    A struct{}
    B struct{}
}
func main() {
    var s S
    fmt.Println(unsafe.Sizeof(s))
}
```

其最终输出结果也是 0，完美切合人们对占位符的基本诉求，就是占着坑位，满足基本输入输出就好。

但这时候问题又出现了，为什么只有空结构会有这种特殊待遇，其他类型又不行?

这是 Go 编译器在内存分配时做的优化项

Go

```
// base address for all 0-byte allocations
var zerobase uintptr
func mallocgc(size uintptr, typ *_type, needzero bool) unsafe.Pointer {
    ...
    if size == 0 {
        return unsafe.Pointer(&zerobase)
    }
}
```

当发现 size 为 0 时，会直接返回变量 `zerobase` 的引用，该变量是所有 0 字节的基准地址，不占据任何宽度。

因此空结构体的广泛使用，是 Go 开发者们借助了这个小优化，达到了占位符的目的。

使用场景

了解清楚为什么空结构作为占位符使用的原因后，我们更进一步了解其真实的使用场景有哪些。

主要分为三块：

- 实现方法接收者。

- 实现集合类型。
- 实现空通道。

实现方法接收者

在业务场景下，我们需要将方法组合起来，代表其是一个 ” 分组 “ 的，便于后续拓展和维护。

但是如果我们使用：

Go

```go
type T string

func (s *T) Call()
```

又似乎有点不大友好，因为作为一个字符串类型，其本身会占据定的空间。

这种时候我们会采用空结构体的方式，这样也便于未来针对该类型进行公共字段等的增加。如下：

Go

```go
type T struct{}

func (s *T) Call() {
    fmt.Println("脑子进煎鱼了")
}
func main() {
    var s T
    s.Call()
}
```

在该场景下，使用空结构体从多维度来考量是最合适的，易拓展，省空间，最结构化。

另外你会发现，其实你在日常开发中下意识就已经这么做了，你可以理解为设计模式和日常生活相结合的另类案例。

实现集合类型

在 Go 语言的标准库中并没有提供集合（Set）的相关实现，因此一般在代码中我们图方便，会直接用 map 来替代。

但有个问题，就是集合类型的使用，只需要用到 key（键），不需要 value（值）。

这就是空结构体大战身手的场景了：

Go

```go
type Set map[string]struct{}

func (s Set) Append(k string) {
    s[k] = struct{}{}
}
func (s Set) Remove(k string) {
    delete(s, k)
}

func (s Set) Exist(k string) bool {
    _, ok := s[k]
    return ok
}
func main() {
    set := Set{}
    set.Append("煎鱼")
    set.Append("咸鱼")
    set.Append("蒸鱼")
    set.Remove("煎鱼")
    fmt.Println(set.Exist("煎鱼"))
}
```

空结构体作为占位符，不会额外增加不必要的内存开销，很方便的就是解决了。

实现空通道

在 Go channel 的使用场景中，常常会遇到通知型 channel，其不需要发送任何数据，只是用于协调 Goroutine 的运行，用于流转各类状态或是控制并发情况。

如下：

CSS

```
func main() {
    ch := make(chan struct{})
    go func() {
        time.Sleep(1 * time.Second)
        close(ch)
    }()
    fmt.Println("脑子好像进...")
    <-ch fmt.Println("煎鱼了！")
}
```

输出结果：

Go

```
1  脑子好像进...煎鱼了!
```

该程序会先输出 ” 脑子好像进...“ 后，再睡眠一段时间再输出 "煎鱼了！"，达到间断控制 channel 的效果。

由于该 channel 使用的是空结构体，因此也不会带来额外的内存开销。