又吵起来了，Go 是传值还是传引用?

原创 陈煎鱼 脑子进煎鱼了 5月26日

收录于话题
#Go 98 #面试题 29

大家好，我是煎鱼。

前几天在咱们的 Go 交流群里，有一个小伙伴问了 “xxx 是不是引用类型？” 这个问题，引发了将近 5 小时的讨论：

interface 是不是引用类型

接口是引用类型

怎么证明呢

😳，golang不都是传值吗😳

引用类型也是传值

兜兜转转回到了日经的问题，几乎每个月都要有人因此吵一架。就是 **Go 语言到底是传值（值传递），还是传引用（引用传递）？**

Go 官方的定义

本部分引用 Go 官方 FAQ 的 “When are function parameters passed by value?”，内容如下。

如同 C 系列的所有语言一样，**Go 语言中的所有东西都是以值传递的**。也就是说，一个函数总是得到一个被传递的东西的副本，就像有一个赋值语句将值赋给参数一样。

When are function parameters passed by value?

As in all languages in the C family, everything in Go is passed by value. That is, a function always gets a copy of the thing being passed, as if there were an assignment statement assigning the value to the parameter. For instance, passing an `int` value to a function makes a copy of the `int`, and passing a pointer value makes a copy of the pointer, but not the data it points to. (See a later section for a discussion of how this affects method receivers.)

Map and slice values behave like pointers: they are descriptors that contain pointers to the underlying map or slice data. Copying a map or slice value doesn't copy the data it points to. Copying an interface value makes a copy of the thing stored in the interface value. If the interface value holds a struct, copying the interface value makes a copy of the struct. If the interface value holds a pointer, copying the interface value makes a copy of the pointer, but again not the data it points to.

Note that this discussion is about the semantics of the operations. Actual implementations may apply optimizations to avoid copying as long as the optimizations do not change the semantics.

例如：

- 向一个函数传递一个 int 值，就会得到 int 的副本。
  而传递一个指针值就会得到指针的副本，但不会得到它所指向的数据。
- map 和 slice 的行为类似于指针：它们是包含指向底层 map 或 slice 数据的指针的描述符。
  - 复制一个 map 或 slice 值并不会复制它所指向的数据。
  - 复制一个接口值会复制存储在接口值中的东西。
  - 如果接口值持有一个结构，复制接口值就会复制该结构。如果接口值持有一个指针，复制接口值会复制该指针，但同样不会复制它所指向的数据。

划重点，Go 语言中一切都是值传递，没有引用传递。不要直接把其他概念硬套上来，会犯先入为主的错误的。

传值和传引用

传值

传值，也叫做值传递（pass by value）。其**指的是在调用函数时将实际参数复制一份传递到函数中**，这样在函数中如果对参数进行修改，将不会影响到实际参数。

简单来讲，值传递，所传递的是该参数的副本，是复制了一份的，本质上不能认为是一个东西，指向的不是一个内存地址。

案例一如下：

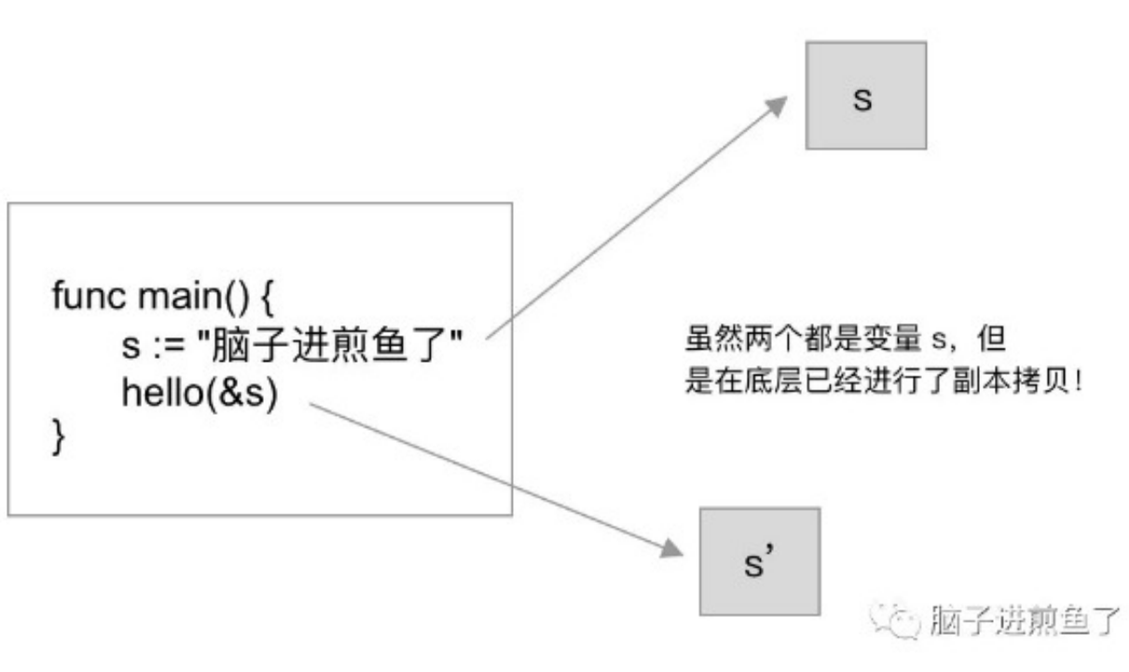
```
func main() {
 s := "脑子进煎鱼了"
 fmt.Printf("main 内存地址: %p\n", &s)
 hello(&s)
}

func hello(s *string) {
 fmt.Printf("hello 内存地址: %p\n", &s)
}
```

输出结果：

```
main 内存地址: 0xc000116220
hello 内存地址: 0xc000132020
```

我们可以看到在 main 函数中的变量 s 所指向的内存地址是 `0xc000116220` 。在经过 hello 函数的参数传递后，其在内部所输出的内存地址是 `0xc000132020` ，两者发生了改变。



据此我们可以得出结论，在 Go 语言确实都是值传递。那是不是在函数内修改值，就不会影响到 main 函数呢？

案例二如下：

```
func main() {
 s := "脑子进煎鱼了"
 fmt.Printf("main 内存地址: %p\n", &s)
 hello(&s)
 fmt.Println(s)
}

func hello(s *string) {
 fmt.Printf("hello 内存地址: %p\n", &s)
 *s = "煎鱼进脑子了"
}
```

我们在 hello 函数中修改了变量 s 的值，那么最后在 main 函数中我们所输出的变量 s 的值是什么呢。是 “脑子进煎鱼了”，还是 "煎鱼进脑子了"?

输出结果：

```
main 内存地址: 0xc000010240
hello 内存地址: 0xc00000e030
煎鱼进脑子了
```

输出的结果是 “煎鱼进脑子了”。这时候大家可能又犯嘀咕了，煎鱼前面明明说的是 Go 语言只有值传递，也验证了两者的内存地址，都是不一样的，怎么他这下他的值就改变了，这是为什么？

因为 “如果传过去的值是指向内存空间的地址，那么是可以对这块内存空间做修改的”。

也就是这两个内存地址，其实是指针的指针，其根源都指向着同一个指针，也就是指向着变量 s。因此我们进一步修改变量 s，得到输出 “煎鱼进脑子了” 的结果。

传引用

传引用，也叫做引用传递（pass by reference），**指在调用函数时将实际参数的地址直接传递到函数中**，那么在函数中对参数所进行的修改，将影响到实际参数。

在 Go 语言中，官方已经明确了没有传引用，也就是没有引用传递这一情况。

因此借用文字简单描述，像是例子中，即使你将参数传入，最终所输出的内存地址都是一样的。

争议最大的 map 和 slice

这时候又有小伙伴疑惑了，你看 Go 语言中的 map 和 slice 类型，能直接修改，难道不是同个内存地址，不是引用了？

其实在 FAQ 中有一句提醒很重要：“map 和 slice 的行为类似于指针，它们是包含指向底层 map 或 slice 数据的指针的描述符”。

map

针对 map 类型，进一步展开来看看例子：

```
func main() {
 m := make(map[string]string)
 m["脑子进煎鱼了"] = "这次一定！"
 fmt.Printf("main 内存地址: %p\n", &m)
 hello(m)

 fmt.Printf("%v", m)
}

func hello(p map[string]string) {
 fmt.Printf("hello 内存地址: %p\n", &p)
 p["脑子进煎鱼了"] = "记得点赞！"
}
```

输出结果：

```
main 内存地址: 0xc00000e028
hello 内存地址: 0xc00000e038
```

确实是值传递，那修改后的 map 的结果应该是什么。既然是值传递，那肯定就是 "这次一定！"，对吗？

输出结果：

```
map[脑子进煎鱼了:记得点赞！]
```

结果是修改成功，输出了 “记得点赞！”。这下就尴尬了，为什么是值传递，又还能做到类似引用的效果，能修改到源值呢？

这里的小窍门是：

```
func makemap(t *maptype, hint int, h *hmap) *hmap {}
```

这是创建 map 类型的底层 runtime 方法，注意其返回的是 `*hmap` 类型，是一个指针。也就是 Go 语言通过对 map 类型的相关方法进行封装，达到了用户需要关注指针传递的作用。

就是说当我们在调用 `hello` 方法时，其相当于是在传入一个指针参数 `hello(*hmap)` ，与前面的值类型的案例二类似。

这类情况我们称其为 “引用类型”，但 “引用类型” 不等同于就是传引用，又或是引用传递了，还是有比较明确的区别的。

在 Go 语言中与 map 类型类似的还有 chan 类型：

```
func makechan(t *chantype, size int) *hchan {}
```

一样的效果。

slice

针对 slice 类型，进一步展开来看看例子：

```
func main() {
 s := []string{"烤鱼", "咸鱼", "摸鱼"}
 fmt.Printf("main 内存地址: %p\n", s)
 hello(s)
 fmt.Println(s)
}

func hello(s []string) {
 fmt.Printf("hello 内存地址: %p\n", s)
 s[0] = "煎鱼"
}
```

输出结果：

```
main 内存地址: 0xc000098180
hello 内存地址: 0xc000098180
[煎鱼 咸鱼 摸鱼]
```

从结果来看，两者的内存地址一样，也成功的变更到了变量 s 的值。这难道不是引用传递吗，煎鱼翻车了？

关注两个细节：

- 没有用 `&` 来取地址。
- 可以直接用 `%p` 来打印。

之所以可以同时做到上面这两件事，是因为标准库 `fmt` 针对在这一块做了优化：

```
func (p *pp) fmtPointer(value reflect.Value, verb rune) {
 var u uintptr
 switch value.Kind() {
 case reflect.Chan, reflect.Func, reflect.Map, reflect.Ptr, reflect.Slice, reflect.UnsafePointer:
  u = value.Pointer()
 default:
  p.badVerb(verb)
  return
 }
```

留意到代码 `value.Pointer` ，标准库进行了特殊处理，直接对应的值的指针地址，当然就不需要取地址符了。

标准库 `fmt` 能够输出 slice 类型对应的值的原因也在此：

```
func (v Value) Pointer() uintptr {
 ...
 case Slice:
  return (*SliceHeader)(v.ptr).Data
 }
}

type SliceHeader struct {
 Data uintptr
 Len  int
 Cap  int
}
```

其在内部转换的 `Data` 属性，正正是 Go 语言中 slice 类型的运行时表现 SliceHeader。我们在调用 `%p` 输出时，是在输出 slice 的底层存储数组元素的地址。

下一个问题是：为什么 slice 类型可以直接修改源数据的值呢。

其实和输出的原理是一样的，在 Go 语言运行时，传递的也是相应 slice 类型的底层数组的指针，但需要注意，其使用的是指针的副本。严格意义是引用类型，依旧是值传递。

妙不妙？

总结

在今天这篇文章中，我们针对 Go 语言的日经问题：“Go 语言到底是传值（值传递），还是传引用（引用传递）” 进行了基本的讲解和分析。

另外在业内中，最多人犯迷糊的就是 slice、map、chan 等类型，都会认为是 “引用传递”，从而认为 Go 语言的 xxx 就是引用传递，我们对此也进行了案例演示。

这实则是不大对的认知，因为：“如果传过去的值是指向内存空间的地址，是可以对这块内存空间做修改的”。

其确实复制了一个副本，但他也借由各手段（其实就是传指针），达到了能修改源数据的效果，是引用类型。

石锤，Go 语言只有值传递，

参考

- Go 读者交流群
- When are function parameters passed by value?
- Java 到底是值传递还是引用传递？
- Go语言参数传递是传值还是传引用

脑子进煎鱼了
分享计算机基础、Go 语言、微服务架构和系统设计；著有图书《Go 语言编程之旅》。
143篇原创内容
公众号

关注煎鱼，吸取他的知识 👆

··························>

你好，我是煎鱼。高一折腾过前端，参加过国赛拿了奖，大学搞过 PHP。现在整 Go，在公司负责微服务架构等相关工作推进和研发。

从大学开始靠自己赚生活费和学费，到出版 Go 畅销书《Go 语言编程之旅》，再到获得 GOP（Go 领域最有观点专家）荣誉，**点击蓝字查看我的出书之路**。

日常分享高质量文章，输出 Go 面试、工作经验、架构设计，加微信拉读者交流群，记得点赞！

收录于话题 #Go·98个 >

< 上一篇
我这样升级 Go 版本，你呢?

下一篇 >
Go 编程怎么也有踩内存?