15 | 关于指针的有限操作

2018-09-14 郝林

在前面的文章中，我们已经提到过很多次“指针”了，你应该已经比较熟悉了。不过，我们那时大多指的是指针类型及其对应的指针值，今天我们讲的则是更为深入的内容。

让我们先来复习一下。

```
type Dog struct {
 name string
}

func (dog *Dog) SetName(name string) {
 dog.name = name
}
```

对于基本类型Dog来说，*Dog就是它的指针类型。而对于一个Dog类型，值不为nil的变量dog，取址表达式&dog的结果就是该变量的值（也就是基本值）的指针值。

如果一个方法的接收者是*Dog类型的，那么该方法就是基本类型Dog的一个指针方法。

在这种情况下，这个方法的接收者实际上就是当前的基本值的指针值。我们可以通过指针值无缝地访问到基本值包含的任何字段，以及调用与之关联的任何方法。这应该就是我们在编写Go程

序的过程中，用得最频繁的“指针”了。

从传统意义上说，指针是一个指向某个确切的内存地址的值。这个内存地址可以是任何数据或代码的起始地址，比如，某个变量、某个字段或某个函数。

我们刚刚只提到了其中的一种情况，在Go语言中还有其他几样东西可以代表“指针”。其中最贴近传统意义的当属uintptr类型了。该类型实际上是一个数值类型，也是Go语言内建的数据类型之一。

根据当前计算机的计算架构的不同，它可以存储32位或64位的无符号整数，可以代表任何指针的位（bit）模式，也就是原始的内存地址。

再来看Go语言标准库中的unsafe包。unsafe包中有一个类型叫做Pointer，也代表了“指针”。

unsafe.Pointer可以表示任何指向可寻址的值的指针，同时它也是前面提到的指针值和uintptr值之间的桥梁。也就是说，通过它，我们可以在这两种值之上进行双向的转换。这里有一个很关键的词——可寻址的（addressable）。在我们继续说unsafe.Pointer之前，需要先要搞清楚这个词的确切含义。

今天的问题是：你能列举出Go语言中的哪些值是不可寻址的吗？

这道题的典型回答是以下列表中的值都是不可寻址的。

- 常量的值。
- 基本类型值的字面量。
- 算术操作的结果值。
- 对各种字面量的索引表达式和切片表达式的结果值。不过有一个例外，对切片字面量的索引结果值却是可寻址的。
- 对字符串变量的索引表达式和切片表达式的结果值。
- 对字典变量的索引表达式的结果值。
- 函数字面量和方法字面量，以及对它们的调用表达式的结果值。
- 结构体字面量的字段值，也就是对结构体字面量的选择表达式的结果值。
- 类型转换表达式的结果值。
- 类型断言表达式的结果值。
- 接收表达式的结果值。

问题解析

初看答案中的这些不可寻址的值好像并没有什么规律。不过别急，我们一起来梳理一下。你可以对照着demo35.go文件中的代码来看，这样应该会让你理解起来更容易一些。

常量的值总是会被存储到一个确切的内存区域中，并且这种值肯定是**不可变的**。基本类型值的字面量也是一样，其实它们本就可以被视为常量，只不过没有任何标识符可以代表它们罢了。

第一个关键词：不可变的。由于Go语言中的字符串值也是不可变的，所以对于一个字符串类型的变量来说，基于它的索引或切片的结果值也都是不可寻址的，因为即使拿到了这种值的内存地址也改变不了什么。

算术操作的结果值属于一种**临时结果**。在我们把这种结果值赋给任何变量或常量之前，即使能拿到它的内存地址也是没有任何意义的。

第二个关键词：临时结果。这个关键词能被用来解释很多现象。我们可以把各种对值字面量施加的表达式的求值结果都看做是临时结果。

我们都知道，Go语言中的表达式有很多种，其中常用的包括以下几种。

- 用于获得某个元素的索引表达式。
- 用于获得某个切片（片段）的切片表达式。
- 用于访问某个字段的选择表达式。
- 用于调用某个函数或方法的调用表达式。
- 用于转换值的类型的类型转换表达式。
- 用于判断值的类型的类型断言表达式。
- 向通道发送元素值或从通道那里接收元素值的接收表达式。

我们把以上这些表达式施加在某个值字面量上一般都会得到一个临时结果。比如，对数组字面量和字典字面量的索引结果值，又比如，对数组字面量和切片字面量的切片结果值。它们都属于临时结果，都是不可寻址的。

一个需要特别注意的例外是，对切片字面量的索引结果值是可寻址的。因为不论怎样，每个切片值都会持有一个底层数组，而这个底层数组中的每个元素值都是有一个确切的内存地址的。

你可能会问，那么对切片字面量的切片结果值为什么却是不可寻址的？这是因为切片表达式总会返回一个新的切片值，而这个新的切片值在被赋给变量之前属于临时结果。

你可能已经注意到了，我一直在说针对数组值、切片值或字典值的**字面量**的表达式会产生临时结果。如果针对的是数组类型或切片类型的**变量**，那么索引或切片的结果值就都不属于临时结果了，是可寻址的。

这主要因为变量的值本身就不是“临时的”。对比而言，值字面量在还没有与任何变量（或者说任何标识符）绑定之前是没有落脚点的，我们无法以任何方式引用到它们。这样的值就是“临时的”。

再说一个例外。我们通过对字典类型的变量施加索引表达式，得到的结果值不属于临时结果，可

是，这样的值却是不可寻址的。原因是，字典中的每个键-元素对的存储位置都可能会变化，而且这种变化外界是无法感知的。

我们都知道，字典中总会有若干个哈希桶用于均匀地储存键-元素对。当满足一定条件时，字典可能会改变哈希桶的数量，并适时地把其中的键-元素对搬运到对应的新的哈希桶中。

在这种情况下，获取字典中任何元素值的指针都是无意义的，也是**不安全的**。我们不知道什么时候那个元素值会被搬运到何处，也不知道原先的那个内存地址上还会被存放什么别的东西。所以，这样的值就应该是不可寻址的。

第三个关键词：不安全的。“不安全的”操作很可能会破坏程序的一致性，引发不可预知的错误，从而严重影响程序的功能和稳定性。

再来看函数。函数在Go语言中是一等公民，所以我们可以把代表函数或方法的字面量或标识符赋给某个变量、传给某个函数或者从某个函数传出。但是，这样的函数和方法都是不可寻址的。一个原因是函数就是代码，是不可变的。

另一个原因是，拿到指向一段代码的指针是不安全的。此外，对函数或方法的调用结果值也是不可寻址的，这是因为它们都属于临时结果。

至于典型回答中最后列出的那几种值，由于都是针对值字面量的某种表达式的结果值，所以都属于临时结果，都不可寻址。

好了，说了这么多，希望你已经有所领悟了。我来总结一下。

1. **不可变的**值不可寻址。常量、基本类型的值字面量、字符串变量的值、函数以及方法的字面量都是如此。其实这样规定也有安全性方面的考虑。
2. 绝大多数被视为**临时结果**的值都是不可寻址的。算术操作的结果值属于临时结果，针对值字面量的表达式结果值也属于临时结果。但有一个例外，对切片字面量的索引结果值虽然也属于临时结果，但却是可寻址的。
3. 若拿到某值的指针可能会破坏程序的一致性，那么就是**不安全的**，该值就不可寻址。由于字典的内部机制，对字典的索引结果值的取址操作都是不安全的。另外，获取由字面量或标识符代表的函数或方法的地址显然也是不安全的。

最后说一句，如果我们把临时结果赋给一个变量，那么它就是可寻址的了。如此一来，取得的指针指向的就是这个变量持有的那个值了。

知识扩展

问题1：不可寻址的值在使用上有哪些限制？

首当其冲的当然是无法使用取址操作符&获取它们的指针了。不过，对不可寻址的值施加取址操作都会使编译器报错，所以倒是不用太担心，你只要记住我在前面讲述的那几条规律，并在编码

的时候提前注意一下就好了。

我们来看下面这个小问题。我们依然以那个结构体类型Dog为例。

```
func New(name string) Dog {
 return Dog{name}
}
```

我们再为它编写一个函数New。这个函数会接受一个名为name的string类型的参数，并会用这个参数初始化一个Dog类型的值，最后返回该值。我现在要问的是：如果我调用该函数，并直接以链式的手法调用其结果值的指针方法SetName，那么可以达到预期的效果吗？

```
New("little pig").SetName("monster")
```

如果你还记得我在前面讲述的内容，那么肯定会知道调用New函数所得到的结果值属于临时结果，是不可寻址的。

可是，那又怎样呢？别忘了，我在讲结构体类型及其方法的时候还说过，我们可以在一个基本类型的值上调用它的指针方法，这是因为Go语言会自动地帮我们转译。

更具体地说，对于一个Dog类型的变量dog来说，调用表达式dog.SetName("monster")会被自动地转译为(&dog).SetName("monster")，即：先取dog的指针值，再在该指针值上调用SetName方法。

发现问题了吗？由于New函数的调用结果值是不可寻址的，所以无法对它进行取址操作。因此，上边这行链式调用会让编译器报告两个错误，一个是果，即：不能在New("little pig")的结果值上调用指针方法。一个是因，即：不能取得New("little pig")的地址。

除此之外，我们都知道，Go语言中的++和--并不属于操作符，而分别是自增语句和自减语句的重要组成部分。

虽然Go语言规范中的语法定义是，只要在++或--的左边添加一个表达式，就可以组成一个自增语句或自减语句，但是，它还明确了一个很重要的限制，那就是这个表达式的结果值必须是可寻址的。这就使得针对值字面量的表达式几乎都无法被用在这里。

不过这有一个例外，虽然对字典字面量和字典变量索引表达式的结果值都是不可寻址的，但是这样的表达式却可以被用在自增语句和自减语句中。

与之类似的规则还有两个。一个是，在赋值语句中，赋值操作符左边的表达式的结果值必须可寻

址的，但是对字典的索引结果值也是可以的。

另一个是，在带有range子句的for语句中，在range关键字左边的表达式的结果值也都必须是可寻址的，不过对字典的索引结果值同样可以被用在这里。以上这三条规则我们合并起来记忆就可以了。

与这些定死的规则相比，我刚刚讲到的那个与指针方法有关的问题，你需要好好理解一下，它涉及了两个知识点的联合运用。起码在我面试的时候，它是一个可选择的考点。

问题 2：怎样通过unsafe.Pointer操纵可寻址的值？

前边的基础知识很重要。不过现在让我们再次关注指针的用法。我说过，unsafe.Pointer是像*Dog类型的值这样的指针值和uintptr值之间的桥梁，那么我们怎样利用unsafe.Pointer的中转和uintptr的底层操作来操纵像dog这样的值呢？

首先说明，这是一项黑科技。它可以绕过Go语言的编译器和其他工具的重重检查，并达到潜入内存修改数据的目的。这并不是一种正常的编程手段，使用它会很危险，很有可能造成安全隐患。

我们总是应该优先使用常规代码包中提供的API去编写程序，当然也可以把像reflect以及go/ast这样的代码包作为备选项。作为上层应用的开发者，请谨慎地使用unsafe包中的任何程序实体。

不过既然说到这里了，我们还是要来一探究竟的。请看下面的代码：

```
dog := Dog{"little pig"}
dogP := &dog
dogPtr := uintptr(unsafe.Pointer(dogP))
```

我先声明了一个Dog类型的变量dog，然后用取址操作符&，取出了它的指针值，并把它赋给了变量dogP。

最后，我使用了两个类型转换，先把dogP转换成了一个unsafe.Pointer类型的值，然后紧接着又把后者转换成了一个uintptr的值，并把它赋给了变量dogPtr。这背后隐藏着一些转换规则，如下：

1. 一个指针值（比如*Dog类型的值）可以被转换为一个unsafe.Pointer类型的值，反之亦然。
2. 一个uintptr类型的值也可以被转换为一个unsafe.Pointer类型的值，反之亦然。
3. 一个指针值无法被直接转换成一个uintptr类型的值，反过来也是如此。

所以，对于指针值和uintptr类型值之间的转换，必须使用unsafe.Pointer类型的值作为中转。那么，我们把指针值转换成uintptr类型的值有什么意义吗？

```
namePtr := dogPtr + unsafe.Offsetof(dogP.name)
nameP := (*string)(unsafe.Pointer(namePtr))
```

这里需要与unsafe.Offsetof函数搭配使用才能看出端倪。unsafe.Offsetof函数用于获取两个值在内存中的起始存储地址之间的偏移量，以字节为单位。

这两个值一个是某个字段的值，另一个是该字段值所属的那个结构体值。我们在调用这个函数的时候，需要把针对字段的选择表达式传给它，比如dogP.name。

有了这个偏移量，又有了结构体值在内存中的起始存储地址（这里由dogPtr变量代表），把它们相加我们就可以得到dogP的name字段值的起始存储地址了。这个地址由变量namePtr代表。

此后，我们可以再通过两次类型转换把namePtr的值转换成一个*string类型的值，这样就得到了指向dogP的name字段值的指针值。

你可能会问，我直接用取址表达式&(dogP.name)不就能拿到这个指针值了吗？干嘛绕这么大一圈呢？你可以想象一下，如果我们根本就不知道这个结构体类型是什么，也拿不到dogP这个变量，那么还能去访问它的name字段吗？

答案是，只要有namePtr就可以。它就是一个无符号整数，但同时也是一个指向了程序内部数据的内存地址。它可能会给我们带来一些好处，比如可以直接修改埋藏得很深的内部数据。

但是，一旦我们有意或无意地把这个内存地址泄露出去，那么其他人就能够肆意地改动dogP.name的值，以及周围的内存地址上存储的任何数据了。

即使他们不知道这些数据的结构也无所谓啊，改不好还改不坏吗？不正确地改动一定会给程序带来不可预知的问题，甚至造成程序崩溃。这可能还是最好的灾难性后果；所以我才说，使用这种非正常的编程手段会很危险。

好了，现在你知道了这种手段，也知道了它的危险性，那就谨慎对待，防患于未然吧。

总结

我们今天集中说了说与指针有关的问题。基于基本类型的指针值应该是我们最常用到的，也是我们最需要关注的，比如*Dog类型的值。怎样得到一个这样的指针值呢？这需要用到取址操作和操作符&。

不过这里还有个前提，那就是取址操作的操作对象必须是可寻址的。关于这方面你需要记住三个关键词：不可变的、临时结果和不安全的。只要一个值符合了这三个关键词中的任何一个，它就是不可寻址的。

但有一个例外，对切片字面量的索引结果值是可寻址的。那么不可寻址的值在使用上有哪些限制呢？一个最重要的限制是关于指针方法的，即：无法调用一个不可寻址值的指针方法。这涉及了两个知识点的联合运用。

相比于刚说到的这些，`unsafe.Pointer`类型和`uintptr`类型的重要性好像就没那么高了。它们的值同样可以代表指针，并且比前面说的指针值更贴近于底层和内存。

虽然我们可以利用它们去访问或修改一些内部数据，而且就灵活性而言，这种要比通用的方式高很多，但是这往往也会带来不容小觑的安全隐患。

因此，在很多时候，使用它们操纵数据是弊大于利的。不过，对于硬币的背面，我们也总是有必要去了解的。

思考题

今天的思考题是：引用类型的值的指针值是有意义的吗？如果没有意义，为什么？如果有意义，意义在哪里？

戳此查看Go语言专栏文章配套详细代码。