14 | 接口类型的合理运用

2018-09-12 郝林

你好，我是郝林，今天我们来聊聊接口的相关内容。

前导内容：正确使用接口的基础知识

在Go语言的语境中，当我们在谈论“接口”的时候，一定指的是接口类型。因为接口类型与其他数据类型不同，它是没法被实例化的。

更具体地说，我们既不能通过调用new函数或make函数创建出一个接口类型的值，也无法用字面量来表示一个接口类型的值。

对于某一个接口类型来说，如果没有任何数据类型可以作为它的实现，那么该接口的值就不可能存在。

我已经在前面展示过，通过关键字type和interface，我们可以声明出接口类型。

接口类型的类型字面量与结构体类型的看起来有些相似，它们都用花括号包裹一些核心信息。只不过，结构体类型包裹的是它的字段声明，而接口类型包裹的是它的方法定义。

这里你要注意的是：接口类型声明中的这些方法所代表的就是该接口的方法集合。一个接口的方法集合就是它的全部特征。

对于任何数据类型，只要它的方法集合中完全包含了一个接口的全部特征（即全部的方法），那么它就一定是这个接口的实现类型。比如下面这样：

```
type Pet interface {
 SetName(name string)
 Name() string
 Category() string
}
```

我声明了一个接口类型Pet，它包含了3个方法定义，方法名称分别为SetName、Name和Category。这3个方法共同组成了接口类型Pet的方法集合。

只要一个数据类型的方法集合中有这3个方法，那么它就一定是Pet接口的实现类型。这是一种无侵入式的接口实现方式。这种方式还有一个专有名词，叫“Duck typing”，中文常译作“鸭子类型”。你可以到百度的百科页面上去了解一下详情。

顺便说一句，**怎样判定一个数据类型的某一个方法实现的就是某个接口类型中的某个方法呢？**

这有两个充分必要条件，一个是“两个方法的签名需要完全一致”，另一个是“两个方法的名称要一模一样”。显然，这比判断一个函数是否实现了某个函数类型要更加严格一些。

如果你查阅了上篇文章附带的最后一个示例的话，那么就一定会知道，虽然结构体类型Cat不是Pet接口的实现类型，但它的指针类型*Cat却是这个的实现类型。

如果你还不知道原因，那么请跟着我一起来看。我已经把Cat类型的声明搬到了demo31.go文件中，并进行了一些简化，以便你看得更清楚。对了，由于Cat和Pet的发音过于相似，我还把Cat重命名为了Dog。

我声明的类型Dog附带了3个方法。其中有2个值方法，分别是Name和Category，另外还有一个指针方法SetName。

这就意味着，Dog类型本身的方法集合中只包含了2个方法，也就是所有的值方法。而它的指针类型*Dog方法集合却包含了3个方法，

也就是说，它拥有Dog类型附带的所有值方法和指针方法。又由于这3个方法恰恰分别是Pet接口中某个方法的实现，所以*Dog类型就成为了Pet接口的实现类型。

```
dog := Dog{"little pig"}
var pet Pet = &dog
```

正因为如此，我可以声明并初始化一个Dog类型的变量dog，然后把它的指针值赋给类型为Pet的变量pet。

这里有几个名词需要你先记住。对于一个接口类型的变量来说，例如上面的变量pet，我们赋给它的值可以被叫做它的实际值（也称**动态值**），而该值的类型可以被叫做这个变量的实际类型（也称**动态类型**）。

比如，我们把取址表达式&dog的结果值赋给了变量pet，这时这个结果值就是变量pet的动态值，而此结果值的类型*Dog就是该变量的动态类型。

动态类型这个叫法是相对于**静态类型**而言的。对于变量pet来讲，它的**静态类型**就是Pet，并且永远是Pet，但是它的动态类型却会随着我们赋给它的动态值而变化。

比如，只有我把一个*Dog类型的值赋给变量pet之后，该变量的动态类型才会是*Dog。如果还有一个Pet接口的实现类型*Fish，并且我又把一个此类型的值赋给了pet，那么它的动态类型就会变为*Fish。

还有，在我们给一个接口类型的变量赋予实际的值之前，它的动态类型是不存在的。

你需要想办法搞清楚接口类型的变量（以下简称接口变量）的动态值、动态类型和静态类型都是什么意思。因为我会在后面基于这些概念讲解更深层次的知识。

好了，我下面会就“怎样用好Go语言的接口”这个话题提出一系列问题，也请你跟着我一起思考这些问题。

那么今天的问题是：当我们为一个接口变量赋值时会发生什么？

为了突出问题，我把Pet接口的声明简化了一下。

```
type Pet interface {
 Name() string
 Category() string
}
```

我从中去掉了Pet接口的那个名为SetName的方法。这样一来，Dog类型也就变成Pet接口的实现类型了。你可以在demo32.go文件中找到本问题的代码。

现在，我先声明并初始化了一个Dog类型的变量dog，这时它的name字段的值是"little pig"。然后，我把该变量赋给了一个Pet类型的变量pet。最后我通过调用dog的方法SetName把它的name字段的值改成了"monster"。

```
dog := Dog{"little pig"}
var pet Pet = dog
dog.SetName("monster")
```

所以，我要问的具体问题是：在以上代码执行后，pet变量的字段name的值会是什么？

这个题目的典型回答是：pet变量的字段name的值依然是"little pig"。

问题解析

首先，由于dog的SetName方法是指针方法，所以该方法持有的接收者就是指向dog的指针值的副本，因而其中对接收者的name字段的设置就是对变量dog的改动。那么当dog.SetName("monster")执行之后，dog的name字段的值就一定是"monster"。如果你理解到了这一层，那么请小心前方的陷阱。

为什么dog的name字段值变了，而pet的却没有呢？这里有一条通用的规则需要你知晓：如果我们使用一个变量给另外一个变量赋值，那么真正赋给后者的，并不是前者持有的那个值，而是该值的一个副本。

例如，我声明并初始化了一个Dog类型的变量dog1，这时它的name是"little pig"。然后，我在把dog1赋给变量dog2之后，修改了dog1的name字段的值。这时，dog2的name字段的值是什么？

```
dog1 := Dog{"little pig"}
dog2 := dog1
dog1.name = "monster"
```

这个问题与前面那道题几乎一样，只不过这里没有涉及接口类型。这时的dog2的name仍然会是"little pig"。这就是我刚刚告诉你的那条通用规则的又一个体现。

当你知道了这条通用规则之后，确实可以把前面那道题做对。不过，如果当我问你为什么的时候你只说出了这一个原因，那么，我只能说你仅仅答对了一半。

那么另一半是什么？这就需要从接口类型值的存储方式和结构说起了。我在前面说过，接口类型本身是无法被值化的。在我们赋予它实际的值之前，它的值一定会是nil，这也是它的零值。

反过来讲，一旦它被赋予了某个实现类型的值，它的值就不再是nil了。不过要注意，即使我们像前面那样把dog的值赋给了pet，pet的值与dog的值也是不同的。这不仅仅是副本与原值的那种不同。

当我们给一个接口变量赋值的时候，该变量的动态类型会与它的动态值一起被存储在一个专用的数据结构中。

严格来讲，这样一个变量的值其实是这个专用数据结构的一个实例，而不是我们赋给该变量的那个实际的值。所以我才说，pet的值与dog的值肯定是不同的，无论是从它们存储的内容，还是存储的结构上来看都是如此。不过，我们可以认为，这时pet的值中包含了dog值的副本。

我们就把这个专用的数据结构叫做iface吧，在Go语言的runtime包中它其实就叫这个名字。

iface的实例会包含两个指针，一个是指向类型信息的指针，另一个是指向动态值的指针。这里的类型信息是由另一个专用数据结构的实例承载的，其中包含了动态值的类型，以及使它实现了接口的方法和调用它们的途径，等等。

总之，接口变量被赋予动态值的时候，存储的是包含了这个动态值的副本的一个结构更加复杂的值。你明白了吗？

知识扩展

问题 1：接口变量的值在什么情况下才真正为nil？

这个问题初看起来就不是个问题。对于一个引用类型的变量，它的值是否为nil完全取决于我们赋给它了什么，是这样吗？我们先来看一段代码：

```
var dog1 *Dog
fmt.Println("The first dog is nil. [wrap1]")
dog2 := dog1
fmt.Println("The second dog is nil. [wrap1]")
var pet Pet = dog2
if pet == nil {
 fmt.Println("The pet is nil. [wrap1]")
} else {
 fmt.Println("The pet is not nil. [wrap1]")
}
```

在demo33.go文件的这段代码中，我先声明了一个*Dog类型的变量dog1，并且没有对它进行初始化。这时该变量的值是什么？显然是nil。然后我把该变量赋给了dog2，后者的值此时也必定是nil，对吗？

现在问题来了：当我把dog2赋给Pet类型的变量pet之后，变量pet的值会是什么？答案是nil吗？

如果你真正理解了我在上一个问题的解析中讲到的知识，尤其是接口变量赋值及其值的数据结构那部分，那么这道题就不难回答。你可以先思考一下，然后再接着往下看。

当我们把dog2的值赋给变量pet的时候，dog2的值会先被复制，不过由于在这里它的值是nil，所以就没必要复制了。

然后，Go语言会用我上面提到的那个专用数据结构iface的实例包装这个dog2的值的副本，这里是nil。

虽然被包装的动态值是nil，但是pet的值却不会是nil，因为这个动态值只是pet值的一部分而已。

顺便说一句，这时的pet的动态类型就存在了，是*Dog。我们可以通过fmt.Printf函数和占位符%T来验证这一点，另外reflect包的TypeOf函数也可以起到类似的作用。

换个角度来看。我们把nil赋给了pet，但是pet的值却不是nil。

这很奇怪对吗？其实不然。在Go语言中，我们把由字面量nil表示的值叫做无类型的nil。这是真正的nil，因为它的类型也是nil的。虽然dog2的值是真正的nil，但是当我们把这个变量赋给pet的时候，Go语言会把它的类型和值放在一起考虑。

也就是说，这时Go语言会识别出赋予pet的值是一个*Dog类型的nil。然后，Go语言就会用一个iface的实例包装它，包装后的产物肯定就不是nil了。

只要我们把一个有类型的nil赋给接口变量，那么这个变量的值就一定不会是那个真正的nil。因此，当我们使用判等符号==判断pet是否与字面量nil相等的时候，答案一定会是false。

那么，怎样才能让一个接口变量的值真正为nil呢？要么只声明它但不做初始化，要么直接把字面量nil赋给它。

问题 2：怎样实现接口之间的组合？

接口类型间的嵌入也被称为接口的组合。我在前面讲过结构体类型的嵌入字段，这其实就是在说结构体类型间的嵌入。

接口类型间的嵌入要更简单一些，因为它不会涉及方法间的“屏蔽”。只要组合的接口之间有同名的方法就会产生冲突，从而无法通过编译，即使同名方法的签名彼此不同也会是如此。因此，接口的组合根本不可能导致“屏蔽”现象的出现。

与结构体类型间的嵌入很相似，我们只要把一个接口类型的名称直接写到另一个接口类型的成员列表中就可以了。比如：

```
type Animal interface {
 ScientificName() string
 Category() string
}

type Pet interface {
 Animal
 Name() string
}
```

接口类型Pet包含了两个成员，一个是代表了另一个接口类型的Animal，一个是方法Name的定义。它们都被包含在Pet的类型声明的花括号中，并且都各自独占一行。此时，Animal接口包含的所有方法也就成为了Pet接口的方法。

Go语言团队鼓励我们声明体量较小的接口，并建议我们通过这种接口间的组合来扩展程序、增加程序的灵活性。

这是因为相比于包含很多方法的大接口而言，小接口可以更加专注地表达某一种能力或某一类特征，同时也更容易被组合在一起。

Go语言标准库代码包io中的ReadWriteCloser接口和ReadWriter接口就是这样的例子，它们都是由若干个小接口组合而成的。以io.ReadWriteCloser接口为例，它是由io.Reader、io.Writer和io.Closer这三个接口组成的。

这三个接口都只包含了一个方法，是典型的小接口。它们中的每一个都只代表了一种能力，分别是读出、写入和关闭。我们编写这几个小接口的实现类型通常都会很容易。并且，一旦我们同时实现了它们，就等于实现了它们的组合接口io.ReadWriteCloser。

即使我们只实现了io.Reader和io.Writer，那么也等同于实现了io.ReadWriter接口，因为后者就是前两个接口组成的。可以看到，这几个io包中的接口共同组成了一个接口矩阵。它们既相互关联又独立存在。

我在demo34.go文件中写了一个能够体现接口组合优势的小例子，你可以去参看一下。总之，善用接口组合和小接口可以让你的程序框架更加稳定和灵活。

总结

好了，我们来简要总结一下。

Go语言的接口常用于代表某种能力或某类特征。首先，我们要弄清楚的是，接口变量的动态

值、动态类型和静态类型都代表了什么。这些都是正确使用接口变量的基础。当我们给接口变量赋值时，接口变量会持有被赋予值的副本，而不是它本身。

更重要的是，接口变量的值并不等同于这个可被称为动态值的副本。它会包含两个指针，一个指针指向动态值，一个指针指向类型信息。

基于此，即使我们把一个值为nil的某个实现类型的变量赋给了接口变量，后者的值也不可能是真正的nil。虽然这时它的动态值会为nil，但它的动态类型确是存在的。

请记住，除非我们只声明而不初始化，或者显式地赋给它nil，否则接口变量的值就不会为nil。

后面的一个问题相对轻松一些，它是关于程序设计方面的。用好小接口和接口组合总是有益的，我们可以以此形成接口矩阵，进而搭起灵活的程序框架。如果在实现接口时再配合运用结构体类型间的嵌入手法，那么接口组合就可以发挥更大的效用。

思考题

如果我们把一个值为nil的某个实现类型的变量赋给了接口变量，那么在这个接口变量上仍然可以调用该接口的方法吗？如果可以，有哪些注意事项？如果不可以，原因是什么？

戳此查看Go语言专栏文章配套详细代码。