

14 | 接口类型的合理运用

2018-09-12 郝林

Go语言核心36讲

[进入课程 >](#)



讲述：黄洲君

时长 15:09 大小 5.21M



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：

 复制代码

```
1 type Pet interface {  
2     SetName(name string)  
3     Name() string  
4     Category() string  
5 }
```

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。

 复制代码

```
1 dog := Dog{"little pig"}
2 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。

 复制代码

```
1 type Pet interface {  
2     Name() string  
3     Category() string  
4 }
```

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`"。

 复制代码

```
1 dog := Dog{"little pig"}  
2 var pet Pet = dog  
3 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？

 复制代码

```
1 dog1 := Dog{"little pig"}
2 dog2 := dog1
3 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：

 复制代码

```
1 var dog1 *Dog
2 fmt.Println("The first dog is nil. [wrap1]")
3 dog2 := dog1
4 fmt.Println("The second dog is nil. [wrap1]")
5 var pet Pet = dog2
6 if pet == nil {
7     fmt.Println("The pet is nil. [wrap1]")
8 } else {
9     fmt.Println("The pet is not nil. [wrap1]")
10 }
```


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价值放在一起考虑。

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：

 复制代码

```
1 type Animal interface {
2     ScientificName() string
3     Category() string
4 }
5
6 type Pet interface {
7     Animal
8     Name() string
9 }
```

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。](#)

 极客时间

GO语言核心36讲

3个月带你通关GO语言

郝林

《Go 并发编程实战》作者
GoHackers 技术社群发起人
前轻松筹大数据负责人



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 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 13 | 结构体及其方法的使用法门

下一篇 15 | 关于指针的有限操作

精选留言 (33)

 写留言



hiyanxu 置顶

2018-12-19

 2

老师，您好：

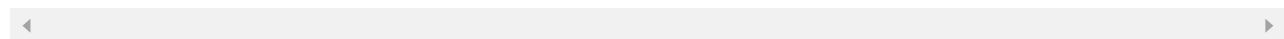
我在这篇文章中看到您说，给接口类型变量赋值时传递的都是副本，我测试了，确实是不会改变被赋值后的接口类型变量。

后面，我重新给Pet接口加上了SetName()方法，然后让*Dog类型实现了该Pet接口，然后声明并初始化了一个d，将d的地址&d赋值给Pet类型的接口变量：...

展开 ▾

作者回复: 没错，虽然是副本，但却是指针的副本，SetName又是指针方法。所以综合起来这种修改就生效了。

另外这类副本都是浅表复制。也没错。



xlh

2018-09-13

👍 51

大神，每篇文章前能先解答上次留的问题吗?思考过后有个答案，有错思之，无错加勉



extraterre...

2018-09-16

👍 24

有个疑问，go里面一个类型实现了接口所有的方法，才算该接口类型，但并没有语法显式说明这个类型实现了哪个接口(例如java中有implements), 这样看别人代码的时候，碰到一个类型，无法知道这个类型是不是实现了一个接口，除非类型和接口写在一个文件，然后还要自己一个一个方法去对比。有比较快的方法可以知道当前类型实现了哪些接口么？

展开 ▾



追梦

2018-09-17

👍 7

关于思考题，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，在这个接口变量上仍然可以访问其方法，但无法访问其属性。使用时需要注意：如果涉及到变量属性，这些属性值均为默认值。



asdf100

2018-09-12

👍 6

golang中，结构体类型struct包裹的是它的字段声明，而接口类型interface包裹的是它的方法定义。



colben

👍 1



2018-09-13



package main

```
import (  
    "fmt"  
)...  
展开
```

展开



undifined

2018-09-12

4

因为将 nil 实际类型的变量赋值给接口变量，会包装为 iface 实例，这个实例不为空，所以依然可以调用接口的方法，但是通过方法访问变量的属性，则会返回空

展开



Aaron

2018-09-20

3

文章中demo32.go demo31.go可不可以直接贴出来

展开



枫林火山

2019-03-28

1

老师，您好，在demo34.go 中我本意是想尝试用interface做内嵌字段来显式表明一个struct的接口能力的。但是这过程中发现，如果我在Dog中内嵌了Animal接口，然后注释掉Dog的ScientificName实现，line37 - 45 如下

```
type Dog struct {  
    Animal...
```

展开

作者回复: 郝林回复：首先，你要明白在Dog里内嵌Animal意味着什么。这意味着Dog类型中包含了一个Animal类型的匿名字段。正是由于这个匿名字段，Dog类型就包含了实现Animal接口所需的所有方法，只不过你并没有为这个匿名字段赋值而已。但是这个字段就摆在那儿了，匿名字段Animal的方法照样会融入Dog的方法集合。这是一个不争的事实，所以类型判断的时候照样会返回true。实际上，单从方法集合融入的这个方面讲，这与Dog内嵌PetTag是一样的。

这样的嵌入方式是可以的。当你要做部件动态装配的时候可以这么做，不论是嵌入接口类型还是嵌入非接口类型。当然了，内嵌接口类型会更加灵活一些，因为你可以为这个匿名字段赋予不同的实现值。



Arthur.Li

2018-09-21

👍 1

方法签名在这里是指什么呀？我看定义说是方法名称和一个参数列表（方法的参数的顺序和类型）组成。

文章里面写的两个条件是方法签名一致和方法名一致，所以有些疑惑了

作者回复: 方法签名严格来说不包含方法名。



Michael

2018-09-17

👍 1

这结课终于让我明白了 reflect 包中，reflect.Type 和 reflect.Value 存在的意义，茅塞顿开啊！

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，一个指向动态值，一个指向动态类型。...

展开 ▾



兔子高

2018-09-12

👍 1

你好，请问分别在什么情况下使用值方法和什么情况下使用引用方法呢

作者回复: 需要改动值内部数据的时候必须使用指针方法。其他时候就要看接口实现、被嵌入要求等方面了。我觉得一般情况下用指针方法就好了。



志鑫

2019-05-10

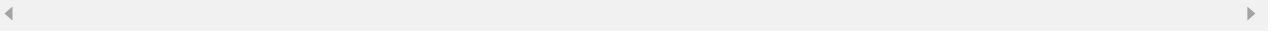
👍

思考题，要看实现类型是值类型还是指针类型；

```
var d2 Dog
var p2 Pet = d2
if p2 != nil {
    fmt.Println("p2.Name()", p2.Name())...
```

展开 ▾

作者回复: 这主要是因为 Dog 和 *Dog 的零值是完全不同的, 前者是 Dog{} , 而后者是 nil 。



张sir

2019-04-19

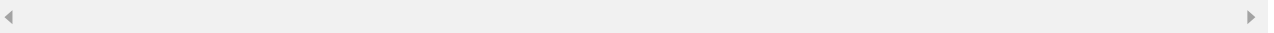


```
var pet Pet = &dog
```

Pet是个指针类型, 不是不能通过字面量来表示一个值吗, 这种赋值操作代表什么

作者回复: 像 &dog 这种式子要先分开来看, “dog” 是一个变量并代表了一个 Dog 类型的值, “&” 是一个操作符并代表了取址操作。合在一起就是一个表达式, 含义是获取一个指针值, 这个指针值指向了变量 dog 代表的那个值。

又因为 *Dog 类型实现了 Pet 接口, 所以才能把这样一个指针值赋给变量 pet 。



枫林火山

2019-03-30



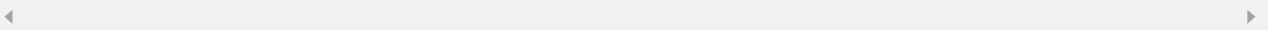
谢谢老师, 有点理解了。内嵌接口不是声明了该类型实现了接口描述的能力, 而是声明该类型有一个实现该接口的内嵌类型。

如果取消注释部分, 直接打印dog.Animal.ScientificName()还是会报错, 因为这个内嵌接口表示的类型一直没初始化😓。

之前的理解还是下意识的往继承靠拢了

展开 ▾

作者回复: 嗯, 确切地说: 这个内嵌的匿名字段还没有被赋予具体的值。



小豆角

2019-03-25



讲的非常棒

展开 ▾



俊杰

2019-03-20



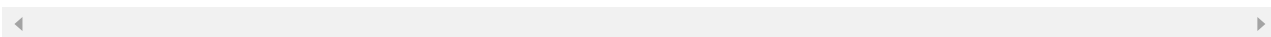
老师您好，有个地方不理解，对象赋值给接口后，为什么判等操作返回的是true呢？比如上面的例子：pet = dog之后紧接着判断pet == dog，返回的是true，按上面的说法，赋值后不是应该被包装成了一个iface吗？这里的判等操作到底是依据什么来判断的呢？麻烦老师解释一下，谢谢~

展开 ∨

作者回复: 你可以参照Go语言规范中的说明：

https://golang.google.cn/ref/spec#Comparison_operators，请注意下面这句：

A value x of non-interface type X and a value t of interface type T are comparable when values of type X are comparable and X implements T. They are equal if t's dynamic type is identical to X and t's dynamic value is equal to x.



思维

2019-03-14



这一课收获很多，里面包括了go语言的规则：类型 *T 的可调用方法集包含接受者为 *T 或 T 的所有方法集

这条规则说的是如果我们用来调用特定接口方法的接口变量是一个指针类型，那么方法的接受者可以是值类型也可以是指针类型.

...

展开 ∨



zhaopan

2019-02-20



```
var dog1 *Dog
fmt.Println("The first dog is nil.")
dog2 := dog1
fmt.Println("The second dog is nil.")
var pet Pet = dog2...
```

展开 ∨



w

2019-01-17



go语言里面有值方法和指针方法，请问这样设计的目的和好处有哪些呢？

作者回复: 这种设计实际上依从的是基本值和指针值的设计思路。在设计灵活性上会有一些好处。

