Go中error类型的nil值和nil

先看C语言中的类似问题：空字符串。

|  |  |
| --- | --- |
| 1 | const char\* empty\_str0 = ""; |
| 2 | const char\* empty\_str1 = "\0empty"; | |

|  |  |
| --- | --- |
| 3 | const char\* empty\_str2 = NULL; |

以上3个字符串并不相等，但是从某种角度看，它们都是对应空的字符串。

* empty\_str0 指向一个空的字符串，但是empty\_str0本身的值是有效的。
* empty\_str1 指向一个非空的字符串，但是字符串的第一个字符是'\0'。
* empty\_str2 本身是一个空的指针。

Go的error是一个interface类型，error的nil问题和C语言的字符串类似。

参考官方的error文档说明：

* <http://golang.org/doc/go_faq.html#nil_error>

在底层，interface作为两个成员实现：一个类型和一个值。该值被称为接口的动态值， 它是一个任意的具体值，而该接口的类型则为该值的类型。对于 int 值3， 一个接口值示意性地包含(int, 3)。

只有在内部值和类型都未设置时(nil, nil)，一个接口的值才为 nil。特别是，一个 nil 接口将总是拥有一个 nil 类型。若我们在一个接口值中存储一个 *int 类型的指针，则内部类型将为*int，无论该指针的值是什么：(\*int, nil)。 因此，这样的接口值会是非 nil 的，即使在该指针的内部为 nil。

下面是一个错误的错误返回方式：

|  |  |
| --- | --- |
| 1 | func returnsError() error { |
| 2 | var p \*MyError = nil | |

|  |  |
| --- | --- |
| 3 | if bad() { |
| 4 | p = ErrBad | |

|  |  |
| --- | --- |
| 5 | } |
| 6 | return p // Will always return a non-nil error. | |

|  |  |
| --- | --- |
| 7 | } |

这里 p 返回的是一个有效值（非nil），值为 nil。   
类似上面的 empty\_str0。

因此，下面判断错误的代码会有问题：

|  |  |
| --- | --- |
| 1 | func main() { |
| 2 | if err := returnsError(); err != nil { | |

|  |  |  |
| --- | --- | --- |
| 3 | panic(nil) | |
| 4 | } |

|  |  |
| --- | --- |
| 5 | } |

针对 returnsError 的问题，可以这样处理（不建议的方式）：

|  |  |
| --- | --- |
| 1 | func main() { |
| 2 | if err := returnsError(); err.(\*MyError) != nil { | |

|  |  |  |
| --- | --- | --- |
| 3 | panic(nil) | |
| 4 | } |

|  |  |
| --- | --- |
| 5 | } |

在判断前先将err转型为\*MyError，然后再判断err的值。   
类似的C语言空字符串可以这样判断：

|  |  |
| --- | --- |
| 1 | bool IsEmptyStr(const char\* str) { |
| 2 | return !(str && str[0] != '\0'); | |

|  |  |
| --- | --- |
| 3 | } |

但是Go语言中标准的错误返回方式不是returnsError这样。   
下面是改进的returnsError：

|  |  |
| --- | --- |
| 1 | func returnsError() error { |
| 2 | var p \*MyError = nil | |

|  |  |
| --- | --- |
| 3 | if bad() { |
| 4 | return nil | |

|  |  |
| --- | --- |
| 5 | } |
| 6 | return p // Will always return a non-nil error. | |

|  |  |
| --- | --- |
| 7 | } |

因此，在处理错误返回值的时候，一定要将正常的错误值转换为 nil。

比如，syscall中就有一个bug是由于没有处理好error导致的：

|  |  |
| --- | --- |
| 01 | // syscall: (\*Proc).Call always returns non-nil err |
| 02 | // http://code.google.com/p/go/issues/detail?id=4686 | |

|  |  |  |
| --- | --- | --- |
| 03 | package main | |
| 04 |  |

|  |  |  |
| --- | --- | --- |
| 05 | import "syscall" | |
| 06 |  |

|  |  |
| --- | --- |
| 07 | func main() { |
| 08 | h := syscall.MustLoadDLL("kernel32.dll") | |

|  |  |  |
| --- | --- | --- |
| 09 | proc := h.MustFindProc("GetVersion") | |
| 10 | r, \_, err := proc.Call() |

|  |  |
| --- | --- |
| 11 | major := byte(r) |
| 12 | minor := uint8(r >> 8) | |

|  |  |
| --- | --- |
| 13 | build := uint16(r >> 16) |
| 14 | print("windows version ", major, ".", minor, " (Build ", build, ")\n") | |

|  |  |
| --- | --- |
| 15 | if err != nil { |
| 16 | e := err.(syscall.Errno) | |

|  |  |  |
| --- | --- | --- |
| 17 | println(err.Error(), "errno =", e) | |
| 18 | } |

|  |  |
| --- | --- |
| 19 | } |

目前issues4686这个bug已经在修复中。

作为用户，临时可以用前面的方法回避这个bug：

|  |  |  |
| --- | --- | --- |
| 01 | // Issue 4686: syscall: (\*Proc).Call always returns non-nil err | |
| 02 | // https://code.google.com/p/go/issues/detail?id=4686 |

|  |  |
| --- | --- |
| 03 | func call(h \*syscall.LazyDLL, name string, |
| 04 | a ...uintptr) (r1, r2 uintptr, err error) { | |

|  |  |  |
| --- | --- | --- |
| 05 | r1, r2, err = h.NewProc(name).Call(a...) | |
| 06 | if err.(syscall.Errno) == 0 { |

|  |  |  |
| --- | --- | --- |
| 07 | return r1, r2, nil | |
| 08 | } |

|  |  |  |
| --- | --- | --- |
| 09 | return | |
| 10 | } |

Go作为一个强类型语言，不同类型之前必须要显示的转换（而且必须是基础类型相同）。   
这样可以回避很多类似C语言中因为隐式类型转换引入的bug。

但是，Go中interface是一个例外：type到interface和interface之间可能是隐式转换的。   
或许，这是Go做得不太好的地方吧。