22 | panic函数、recover函数以及defer语句（下）

2018-10-01 郝林

你好，我是郝林，今天我们继续来聊聊panic函数、recover函数以及defer语句的内容。

我在前一篇文章提到过这样一个说法，panic之中可以包含一个值，用于简要解释引发此panic的原因。

如果一个panic是我们在无意间引发的，那么其中的值只能由Go语言运行时系统给定。但是，当我们使用panic函数有意地引发一个panic的时候，却可以自行指定其包含的值。我们今天的第一个问题就是针对后一种情况提出的。

知识扩展

问题 1：怎样让panic包含一个值，以及应该让它包含什么样的值？

这其实很简单，在调用panic函数时，把某个值作为参数传给该函数就可以了。由于panic函数的唯一一个参数是空接口（也就是interface{}）类型的，所以从语法上讲，它可以接受任何类型的值。

但是，我们最好传入error类型的错误值，或者其他的可以被有效序列化的值。这里的“有效序列化”指的是，可以更易读地去表示形式转换。

还记得吗？对于fmt包下的各种打印函数来说，error类型值的Error方法与其他类型值的String方法是等价的，它们的唯一结果都是string类型的。

我们在通过占位符%s打印这些值的时候，它们的字符串表示形式分别都是这两种方法产出的。

一旦程序异常了，我们就一定要把异常的相关信息记录下来，这通常都是记到程序日志里。

我们在为程序排查错误的时候，首先要做的就是查看和解读程序日志；而最常用也是最方便的日志记录方式，就是记下相关值的字符串表示形式。

所以，如果你觉得某个值有可能会被记到日志里，那么就应该为它关联String方法。如果这个值是error类型的，那么让它的Error方法返回你为它定制的字符串表示形式就可以了。

对于此，你可能会想到fmt.Sprintf，以及fmt.Fprintf这类可以格式化并输出参数的函数。

是的，它们本身就可以被用来输出值的某种表示形式。不过，它们在功能上，肯定远不如我们自己定义的Error方法或者String方法。因此，为不同的数据类型分别编写这两种方法总是首选。

可是，这与传给panic函数的参数值又有什么关系呢？其实道理是相同的。至少在程序崩溃的时候，panic包含的那个值字符串表示形式会被打印出来。

另外，我们还可以施加某种保护措施，避免程序的崩溃。这个时候，panic包含的值会被取出，而在取出之后，它一般都会被打印出来或者记录到日志里。

既然说到了应对panic的保护措施，我们再来看下面一个问题。

问题 2：怎样施加应对panic的保护措施，从而避免程序崩溃？

Go语言的内建函数recover专用于恢复panic，或者说平息运行时恐慌。recover函数无需任何参数，并且会返回一个空接口类型的值。

如果用法正确，这个值实际上就是即将恢复的panic包含的值。并且，如果这个panic是因我们调用panic函数而引发的，那么该值同时也会是我们此次调用panic函数时，传入的参数值副本。请注意，这里强调用法的正确。我们先来看看什么是不正确的用法。

```
package main

import (
 "fmt"
 "errors"
)

func main() {
 fmt.Println("Enter function main.")
 // 引发panic。
 panic(errors.New("something wrong"))
 p := recover()
 fmt.Printf("panic: %s\n", p)
 fmt.Println("Exit function main.")
}
```

在上面这个main函数中，我先通过调用panic函数引发了一个panic，紧接着想通过调用recover函数恢复这个panic。可结果呢？你一试便知，程序依然会崩溃，这个recover函数调用并不会起到任何作用，甚至都没有机会执行。

还记得吗？我提到过panic一旦发生，控制权就会讯速地沿着调用栈的反方向传播。所以，在panic函数调用之后的代码，根本就没有执行的机会。

那如果我把调用recover函数的代码提前呢？也就是说，先调用recover函数，再调用panic函数会怎么样呢？

这显然也是不行的，因为，如果在我们调用recover函数时未发生panic，那么该函数就不会做任何事情，并且只会返回一个nil。

换句话说，这样做毫无意义。那么，到底什么才是正确的recover函数用法呢？这就不得不提到defer语句了。

顾名思义，defer语句就是被用来延迟执行代码的。延迟到什么时候呢？这要延迟到该语句所在的函数即将执行结束的那一刻，无论结束执行的原因是什么。

这与go语句有些类似，一个defer语句总是由一个defer关键字和一个调用表达式组成。

这里存在一些限制，有一些调用表达式是不能出现在这里的，包括：针对Go语言内建函数的调用表达式，以及针对unsafe包中的函数的调用表达式。

顺便说一下，对于go语句中的调用表达式，限制也是一样的。另外，在这里被调用的函数可以是有名称的，也可以是匿名的。我们可以把这里的函数叫做defer函数或者延迟函数。注意，被延迟执行的是defer函数，而不是defer语句。

我刚才说了，无论函数结束执行的原因是什么，其中的defer函数调用都会在它即将结束执行的那一刻执行。即使导致它执行结束的原因是一个panic也会是这样。正因为如此，我们需要联用defer语句和recover函数调用，才能够恢复一个已经发生的panic。

我们来看一下经过修正的代码。

```
package main

import (
 "fmt"
 "errors"
)

func main() {
 fmt.Println("Enter function main.")
 defer func(){
  fmt.Println("Enter defer function.")
  if p := recover(); p != nil {
   fmt.Printf("panic: %s\n", p)
  }
  fmt.Println("Exit defer function.")
 }()
 // 引发panic。
 panic(errors.New("something wrong"))
 fmt.Println("Exit function main.")
}
```

在这个main函数中，我先编写了一条defer语句，并在defer函数中调用了recover函数。仅当调用的结果值不为nil时，也就是说只有panic确实已发生时，我才会打印一行以“panic:”为前缀的内容。

紧接着，我调用了panic函数，并传入了一个error类型值。这里一定要注意，我们要尽量把defer语句写在函数体的开始处，因为在引发panic的语句之后的所有语句，都不会有任何执行机会。

也只有这样，defer函数中的recover函数调用才会拦截，并恢复defer语句所属的函数，及其调用的代码中发生的所有panic。

至此，我向你展示了两个很典型的recover函数的错误用法，以及一个基本的正确用法。

我希望你能够记住错误用法背后的缘由，同时也希望你能真正地理解联用defer语句和recover函数调用的真谛。

在命令源码文件demo50.go中，我把上述三种用法合并在了一段代码中。你可以运行该文件，并体会各种用法所产生的不同效果。

下面我再来多说一点关于defer语句的事情。

问题 3：如果一个函数中有多条defer语句，那么那几个defer函数调用的执行顺序是怎样的？

如果只用一句话回答的话，那就是：在同一个函数中，defer函数调用的执行顺序与它们分别所属的defer语句的出现顺序（更严谨地说，是执行顺序）完全相反。

当一个函数即将结束执行时，其中的写在最下边的defer函数调用会最先执行，其次是写在它上边、与它的距离最近的那个defer函数调用，以此类推，最上边的defer函数调用会最后一个执行。

如果函数中有一条for语句，并且这条for语句中包含了一条defer语句，那么，显然这条defer语句的执行次数，就取决于for语句的迭代次数。

并且，同一条defer语句每被执行一次，其中的defer函数调用就会产生一次，而且，这些函数调用同样不会被立即执行。

那么问题来了，这条for语句中产生的多个defer函数调用，会以怎样的顺序执行呢？

为了彻底搞清楚，我们需要弄明白defer语句执行时发生的事情。

其实也并不复杂，在defer语句每次执行的时候，Go语言会把它携带的defer函数及其参数值另行存储到一个队列中。

这个队列与该defer语句所属的函数是对应的，并且，它是先进后出（FILO）的，相当于一个栈。

在需要执行某个函数中的defer函数调用的时候，Go语言会先拿到对应的队列，然后从该队列中一个一个地取出defer函数及其参数值，并逐个执行调用。

这正是我说“defer函数调用与其所属的defer语句的执行顺序完全相反”的原因了。

下面该你出场了，我在demo51.go文件中编写了一个与本问题有关的示例，其中的核心代码很简单，只有几行而已。

我希望你先查看代码，然后思考并写下该示例被运行时，会打印出哪些内容。

如果你实在想不出来，那么也可以先运行示例，再试着解释打印出的内容。总之，你需要完全搞明白那几行内容为什么会以那样的顺序出现的确切原因。

总结

我们这两期的内容主要讲了两个函数和一条语句。recover函数专用于恢复panic，并且调用即恢复。

它在被调用时会返回一个空接口类型的结果值。如果在调用它时并没有panic发生，那么这个结果值就会是nil。

而如果被恢复的panic是我们通过调用panic函数引发的，那么它返回的结果值就会是我们传给panic函数参数值的副本。

对recover函数的调用只有在defer语句中才能真正起作用。defer语句是被用来延迟执行代码的。

更确切地说，它会让其携带的defer函数的调用延迟执行，并且会延迟到该defer语句所属的函数即将结束执行的那一刻。

在同一个函数中，延迟执行的defer函数调用，会与它们分别所属的defer语句的执行顺序完全相反。还要注意，同一条defer语句每被执行一次，就会产生一个延迟执行的defer函数调用。

这种情况在defer语句与for语句联用时经常出现。这时更要关注for语句中，同一条defer语句产生的多个defer函数调用的实际执行顺序。

以上这些，就是关于Go语言中特殊的程序异常，及其处理方式的核心知识。这里边可以衍生出很多面试题目。

思考题

我们可以在defer函数中恢复panic，那么可以在其中引发panic吗？

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