

## 21 | panic函数、recover函数以及defer语句（上）

2018-09-28 郝林



## 21 | panic函数、recover函数以及defer语句（上）

朗读人：黄洲君 07'44" | 3.55M

我在上两篇文章中，详细地讲述了 Go 语言中的错误处理，并从两个视角为你总结了错误类型、错误值的处理技巧和设计方式。

在本篇，我要给你展示 Go 语言的另外一种错误处理方式。不过，严格来说，它处理的不是错误，而是异常，并且是一种在我们意料之外的程序异常。

这种程序异常被叫做 **panic**，我把它翻译为运行时恐慌。其中的“恐慌”二字是由 **panic** 直译过来的，而之所以前面又加上了“运行时”三个字，是因为这种异常只会在程序运行的时候被抛出来。

比如说，一个 Go 程序里有一个切片，它的长度是 5，也就是说该切片中的元素值的索引分别为0、1、2、3、4，但是，我在程序里却想通过索引5访问其中的元素值，显而易见，这样的访问是不正确的。

Go 程序，确切地说是程序内嵌的 Go 语言运行时系统，会在执行到这行代码的时候抛出一个“index out of range”的 **panic**，用以提示你索引越界了。

当然了，这不仅仅是个提示。当 **panic** 被抛出之后，如果我们没有在程序里添加任何保护措施的话，程序（或者说代表它的那个进程）就会在打印出 **panic** 的详细情况（以下简称 **panic** 详情）之后，终止运行。

现在，就让我们来看一下这样的 **panic** 详情中都有什么。

```
panic: runtime error: index out of range

goroutine 1 [running]:
main.main()
  /Users/haolin/GeekTime/Golang_Puzzlers/src/puzzlers/article19/q0/demo47.go:5
+0x3d
exit status 2
```

□复制代码

这份详情的第一行是“panic: runtime error: index out of range”。其中的“runtime error”的含义是，这是一个runtime代码包中抛出的 **panic**。在这个 **panic** 中，包含了一个runtime.Error接口类型的值。runtime.Error接口内嵌了error接口并做了一点点扩展，runtime包中有不少它的实现类型。

实际上，此详情中的“panic: ”右边的内容，正是这个 panic 包含的runtime.Error类型值的字符串表示形式。

此外，panic 详情中一般还会包含与它的引发原因有关的 goroutine 的代码执行信息。正如前述详情中的“goroutine 1 [running]”，它表示有一个 ID 为1的 goroutine 在此 panic 被引发的时候正在运行。

注意，这里的 ID 其实并不重要，因为它只是 Go 语言运行时系统内部给予的一个 goroutine 编号，我们在程序中是无法获取和更改的。

我们再看下一行，“main.main()”表明了这个 goroutine 包装的go函数就是命令源码文件中的那个main函数，也就是说这里的 goroutine 正是主 goroutine。再下面的一行，指出的就是这个 goroutine 中的哪一行代码在此 panic 被引发时正在执行。

这包含了此行代码在其所属的源码文件中的行数，以及这个源码文件的绝对路径。这一行最后的+0x3d代表的是：此行代码相对于其所属函数的入口程序计数偏移量。不过，一般情况下它的用处并不大。

最后，“exit status 2”表明我的这个程序是以退出状态码2结束运行的。在大多数操作系统中，只要退出状态码不是0，都意味着程序运行的非正常结束。在 Go 语言中，因 panic 导致程序结束运行的退出状态码一般都会是2。

综上所述，我们从上边的这个 panic 详情可以看出，作为此 panic 的引发根源的代码处于 demo47.go 文件中的第 5 行，同时被包含在main包（也就是命令源码文件所在的代码包）的main函数中。

那么，我的第一个问题也随之而来了。我今天的问题是：从 panic 被引发到程序终止运行的大致过程是什么？

这道题的典型回答是这样的。

我们先说一个大致的过程：某个函数中的某行代码有意或无意地引发了一个 panic。这时，初始的 panic 详情会被建立起来，并且该程序的控制权会立即从此行代码转移至调用其所属函数的那行代码上，也就是调用栈中的上一级。

这也意味着，此行代码所属函数的执行随即终止。紧接着，控制权并不会在此有片刻停留，它又会立即转移至再上一级的调用代码处。控制权如此一级一级地沿着调用栈的反方向传播至顶端，也就是我们编写的最外层函数那里。

这里的最外层函数指的是go函数，对于主 goroutine 来说就是main函数。但是控制权也不会停留在那里，而是被 Go 语言运行时系统收回。

随后，程序崩溃并终止运行，承载程序这次运行的进程也会随之死亡并消失。与此同时，在这个控制权传播的过程中，**panic** 详情会被逐渐地积累和完善，并会在程序终止之前被打印出来。

## 问题解析

**panic** 可能是我们在无意间（或者说一不小心）引发的，如前文所述的索引越界。这类 **panic** 是真正的、在我们意料之外的程序异常。不过，除此之外，我们还是可以有意地引发 **panic**。

**Go** 语言的内建函数 `panic` 是专门用于引发 **panic** 的。`panic` 函数使程序开发者可以在程序运行期间报告异常。

注意，这与从函数返回错误值的意义是完全不同的。当我们的函数返回一个非 `nil` 的错误值时，函数的调用方有权选择不处理，并且不处理的后果往往是不致命的。

这里的“不致命”的意思是，不至于使程序无法提供任何功能（也可以说僵死）或者直接崩溃并终止运行（也就是真死）。

但是，当一个 **panic** 发生时，如果我们不施加任何保护措施，那么导致的直接后果就是程序崩溃，就像前面描述的那样，这显然是致命的。

为了更清楚地展示答案中描述的过程，我编写了 **demo48.go** 文件。你可以先查看一下其中的代码，再试着运行它，并体会它打印的内容所代表的含义。

我在这里再提示一点。**panic** 详情会在控制权传播的过程中，被逐渐地积累和完善，并且，控制权会一级一级地沿着调用栈的反方向传播至顶端。

因此，在针对某个 **goroutine** 的代码执行信息中，调用栈底端的信息会先出现，然后是上一级调用的信息，以此类推，最后才是此调用栈顶端的信息。

比如，`main` 函数调用了 `caller1` 函数，而 `caller1` 函数又调用了 `caller2` 函数，那么 `caller2` 函数中代码的执行信息会先出现，然后是 `caller1` 函数中代码的执行信息，最后才是 `main` 函数的信息。

```
goroutine 1 [running]:
main.caller2()

/Users/haolin/GeekTime/Golang_Puzzlers/src/puzzlers/article19/q1/demo48.go:22
+0x91
main.caller1()
```

```
/Users/haolin/GeekTime/Golang_Puzzlers/src/puzzlers/article19/q1/demo48.go:15
+0x66
main.main()
  /Users/haolin/GeekTime/Golang_Puzzlers/src/puzzlers/article19/q1/demo48.go:9
+0x66
exit status 2
```

□复制代码

好了，到这里，我相信你已经对 **panic** 被引发后的程序终止过程有一定的了解了。深入地了解此过程，以及正确地解读 **panic** 详情应该是我们的必备技能，这在调试 **Go** 程序或者为 **Go** 程序排查错误的时候非常重要。

## 总结

最近的两篇文章，我们是围绕着 **panic** 函数、**recover** 函数以及 **defer** 语句进行的。今天我主要讲了 **panic** 函数。这个函数是专门被用来引发 **panic** 的。**panic** 也可以被称为运行时恐慌，它是一种只能在程序运行期间抛出的程序异常。

**Go** 语言的运行时系统可能会在程序出现严重错误时自动地抛出 **panic**，我们在需要时也可以通过调用 **panic** 函数引发 **panic**。但不论怎样，如果不加以处理，**panic** 就会导致程序崩溃并终止运行。

## 思考题

一个函数怎样才能把 **panic** 转化为 **error** 类型值，并将其作为函数的结果值返回给调用方？