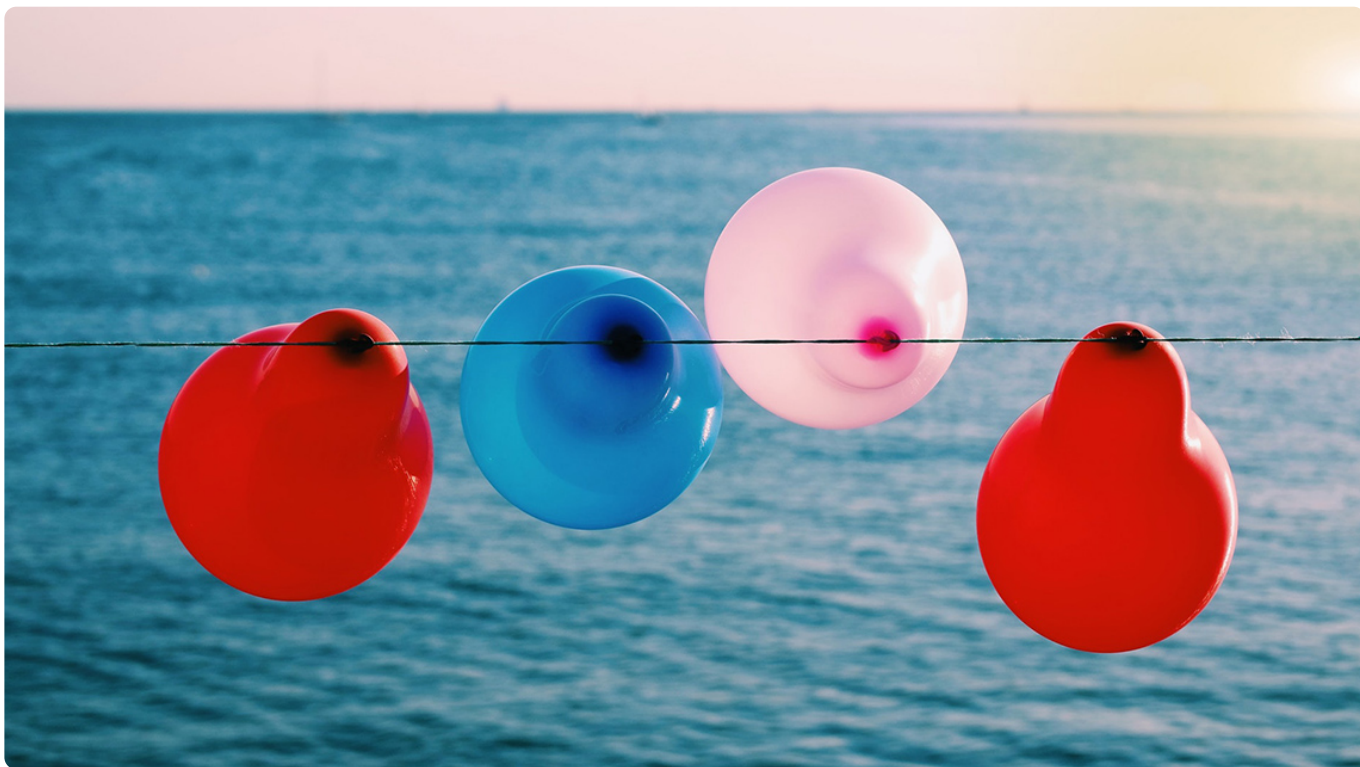


11 | 通道的高级玩法

2018-09-05 郝林

Go语言核心36讲

[进入课程 >](#)



讲述：黄洲君

时长 13:53 大小 6.36M



我们已经讨论过了通道的基本操作以及背后的规则。今天，我再来讲讲通道的高级玩法。

首先来说说单向通道。我们在说“通道”的时候指的都是双向通道，即：既可以发也可以收的通道。

所谓单向通道就是，只能发不能收，或者只能收不能发的通道。一个通道是双向的，还是单向的是由它的类型字面量体现的。

还记得我们在上篇文章中说过的接收操作符`<-`吗？如果我们把它用在通道的类型字面量中，那么它代表的就不是“发送”或“接收”的动作了，而是表示通道的方向。

比如：

```
1 var uselessChan = make(chan<- int, 1)
```

我声明并初始化了一个名叫uselessChan的变量。这个变量的类型是chan<- int，容量是1。

请注意紧挨在关键字chan右边的那个<-，这表示了这个通道是单向的，并且只能发而不能收。

类似的，如果这个操作符紧挨在chan的左边，那么就说明该通道只能收不能发。所以，前者可以被简称为发送通道，后者可以被简称为接收通道。

注意，与发送操作和接收操作对应，这里的“发”和“收”都是站在操作通道的代码的角度上说的。

从上述变量的名字上你也能猜到，这样的通道是没用的。通道就是为了传递数据而存在的，声明一个只有一端（发送端或者接收端）能用的通道没有任何意义。那么，单向通道的用途究竟在哪儿呢？

问题：单向通道有什么应用价值？

你可以先自己想想，然后再接着往下看。

典型回答

概括地说，单向通道最主要的用途就是约束其他代码的行为。

问题解析


这需要从两个方面讲，都跟函数的声明有些关系。先来看下面的代码：

```
1 func SendInt(ch chan<- int) {  
2     ch <- rand.Intn(1000)
```

```
3 }
```

我用`func`关键字声明了一个叫做`SendInt`的函数。这个函数只接受一个`chan<- int`类型的参数。在这个函数中的代码只能向参数`ch`发送元素值，而不能从它那里接收元素值。这就起到了约束函数行为的作用。

你可能会问，我自己写的函数自己肯定能确定操作通道的方式，为什么还要再约束？好吧，这个例子可能过于简单了。在实际场景中，这种约束一般会出现在接口类型声明中的某个方法定义上。请看这个叫`Notifier`的接口类型声明：

 复制代码


```
1 type Notifier interface {  
2     SendInt(ch chan<- int)  
3 }
```

在接口类型声明的花括号中，每一行都代表着一个方法的定义。接口中的方法定义与函数声明很类似，但是只包含了方法名称、参数列表和结果列表。

一个类型如果想成为一个接口类型的实现类型，那么就必须实现这个接口中定义的所有方法。因此，如果我们在某个方法的定义中使用了单向通道类型，那么就相当于在对它的所有实现做出约束。


在这里，`Notifier`接口中的`SendInt`方法只会接受一个发送通道作为参数，所以，在该接口的所有实现类型中的`SendInt`方法都会受到限制。这种约束方式还是很有用的，尤其是在我们编写模板代码或者可扩展的程序库的时候。

顺便说一下，我们在调用`SendInt`函数的时候，只需要把一个元素类型匹配的双向通道传给它就行了，没必要用发送通道，因为 Go 语言在这种情况下会自动地把双向通道转换为函数所需的单向通道。

 复制代码

```
1 intChan1 := make(chan int, 3)  
2 SendInt(intChan1)
```

在另一个方面，我们还可以在函数声明的结果列表中使用单向通道。如下所示：


 复制代码

```
1 func getIntChan() <-chan int {
2     num := 5
3     ch := make(chan int, num)
4     for i := 0; i < num; i++ {
5         ch <- i
6     }
7     close(ch)
8     return ch
9 }
```

函数`getIntChan`会返回一个`<-chan int`类型的通道，这就意味着得到该通道的程序，只能从通道中接收元素值。这实际上就是对函数调用方的一种约束了。

另外，我们在 Go 语言中还可以声明函数类型，如果我们在函数类型中使用了单向通道，那么就相等于在约束所有实现了这个函数类型的函数。

我们再顺便看一下调用`getIntChan`的代码：

 复制代码

```
1 intChan2 := getIntChan()
2 for elem := range intChan2 {
3     fmt.Printf("The element in intChan2: %v\n", elem)
4 }
```

我把调用`getIntChan`得到的结果值赋给了变量`intChan2`，然后用`for`语句循环地取出了该通道中的所有元素值，并打印出来。

这里的`for`语句也可以被称为带有`range`子句的`for`语句。它的用法我在后面讲`for`语句的时候专门说明。现在你只需要知道关于它的三件事。

- 一、这样一条for语句会不断地尝试从intChan2中取出元素值，即使intChan2被关闭，它也会在取出所有剩余的元素值之后再结束执行。
- 二、当intChan2中没有元素值时，它会被阻塞在有for关键字的那一行，直到有新的元素值可取。
- 三、假设intChan2的值为nil，那么它会被永远阻塞在有for关键字的那一行。

这就是带range子句的for语句与通道的联用方式。不过，它是一种用途比较广泛的语句，还可以被用来从其他一些类型的值中获取元素。除此之外，Go 语言还有一种专门为了操作通道而存在的语句：select语句。

知识扩展

问题 1：select语句与通道怎样联用，应该注意些什么？


select语句只能与通道联用，它一般由若干个分支组成。每次执行这种语句的时候，一般只有一个分支中的代码会被运行。

select语句的分支分为两种，一种叫做候选分支，另一种叫做默认分支。候选分支总是以关键字case开头，后跟一个case表达式和一个冒号，然后我们可以从下一行开始写入当分支被选中时需要执行的语句。

默认分支其实就是 default case，因为，当且仅当没有候选分支被选中时它才会被执行，所以它以关键字default开头并直接后跟一个冒号。同样的，我们可以在default:的下一行写入要执行的语句。

由于select语句是专为通道而设计的，所以每个case表达式中都只能包含操作通道的表达式，比如接收表达式。

当然，如果我们需要把接收表达式的结果赋给变量的话，还可以把这里写成赋值语句或者短变量声明。下面展示一个简单的例子。

 复制代码

```
1 // 准备好几个通道。
2 intChannels := [3]chan int{
3     make(chan int, 1),
```

```
4         make(chan int, 1),
5         make(chan int, 1),
6     }
7     // 随机选择一个通道，并向它发送元素值。
8     index := rand.Intn(3)
9     fmt.Printf("The index: %d\n", index)
10    intChannels[index] <- index
11    // 哪一个通道中有可取的元素值，哪个对应的分支就会被执行。
12    select {
13    case <-intChannels[0]:
14        fmt.Println("The first candidate case is selected.")
15    case <-intChannels[1]:
16        fmt.Println("The second candidate case is selected.")
17    case elem := <-intChannels[2]:
18        fmt.Printf("The third candidate case is selected, the element is %d.\n", elem)
19    default:
20        fmt.Println("No candidate case is selected!")
21    }
```

我先准备好了三个类型为`chan int`、容量为1的通道，并把它们存入了一个叫做`intChannels`的数组。

然后，我随机选择一个范围在 $[0, 2]$ 的整数，把它作为索引在上述数组中选择一个通道，并向其中发送一个元素值。

最后，我用一个包含了三个候选分支的`select`语句，分别尝试从上述三个通道中接收元素值，哪一个通道中有值，哪一个对应的候选分支就会被执行。后面还有一个默认分支，不过在这里它是不可能被选中的。

在使用`select`语句的时候，我们首先需要注意下面几个事情。

1. 如果像上述示例那样加入了默认分支，那么无论涉及通道操作的表达式是否有阻塞，`select`语句都不会被阻塞。如果那几个表达式都阻塞了，或者说都没有满足求值的条件，那么默认分支就会被选中并执行。
2. 如果没有加入默认分支，那么一旦所有的`case`表达式都没有满足求值条件，那么`select`语句就会被阻塞。直到至少有一个`case`表达式满足条件为止。
3. 还记得吗？我们可能会因为通道关闭了，而直接从通道接收到一个其元素类型的零值。所以，在很多时候，我们需要通过接收表达式的第二个结果值来判断通道是否已经关

闭。一旦发现某个通道关闭了，我们就应该及时地屏蔽掉对应的分支或者采取其他措施。这对于程序逻辑和程序性能都是有好处的。

4. `select`语句只能对其中的每一个`case`表达式各求值一次。所以，如果我们想连续或定时地操作其中的通道的话，就往往需要通过在`for`语句中嵌入`select`语句的方式实现。但这时要注意，简单地在`select`语句的分支中使用`break`语句，只能结束当前的`select`语句的执行，而并不会对外层的`for`语句产生作用。这种错误的用法可能会让这个`for`语句无休止地运行下去。

下面是一个简单的示例。

 复制代码

```
1 intChan := make(chan int, 1)
2 // 一秒后关闭通道。
3 time.AfterFunc(time.Second, func() {
4     close(intChan)
5 })
6 select {
7 case _, ok := <-intChan:
8     if !ok {
9         fmt.Println("The candidate case is closed.")
10        break
11    }
12    fmt.Println("The candidate case is selected.")
13 }
```

我先声明并初始化了一个叫做`intChan`的通道，然后通过`time`包中的`AfterFunc`函数约定在一秒钟之后关闭该通道。

后面的`select`语句只有一个候选分支，我在其中利用接收表达式的第二个结果值对`intChan`通道是否已关闭做了判断，并在得到肯定结果后，通过`break`语句立即结束当前`select`语句的执行。

这个例子以及前面那个例子都可以在 `demo24.go` 文件中被找到。你应该运行下，看看结果如何。

上面这些注意事项中的一部分涉及到了`select`语句的分支选择规则。我觉得很有必要再专门整理和总结一下这些规则。

问题 2：select 语句的分支选择规则都有哪些？

规则如下面所示。

1. 对于每一个case表达式，都至少会包含一个代表发送操作的发送表达式或者一个代表接收操作的接收表达式，同时也可能会包含其他的表达式。比如，如果case表达式是包含了接收表达式的短变量声明时，那么在赋值符号左边的就可以是一个或两个表达式，不过此处的表达式的结果必须是可以被赋值的。当这样的case表达式被求值时，它包含的多个表达式总会以从左到右的顺序被求值。
2. select语句包含的候选分支中的case表达式都会在该语句执行开始时先被求值，并且求值的顺序是依从代码编写的顺序从上到下的。结合上一条规则，在select语句开始执行时，排在最上边的候选分支中最左边的表达式会最先被求值，然后是它右边的表达式。仅当最上边的候选分支中的所有表达式都被求值完毕后，从上边数第二个候选分支中的表达式才会被求值，顺序同样是从左到右，然后是第三个候选分支、第四个候选分支，以此类推。
3. 对于每一个case表达式，如果其中的发送表达式或者接收表达式在被求值时，相应的操作正处于阻塞状态，那么对该case表达式的求值就是不成功的。在这种情况下，我们可以说，这个case表达式所在的候选分支是不满足选择条件的。
4. 仅当select语句中的所有case表达式都被求值完毕后，它才会开始选择候选分支。这时候，它只会挑选满足选择条件的候选分支执行。如果所有的候选分支都不满足选择条件，那么默认分支就会被执行。如果这时没有默认分支，那么select语句就会立即进入阻塞状态，直到至少有一个候选分支满足选择条件为止。一旦有一个候选分支满足选择条件，select语句（或者说它所在的goroutine）就会被唤醒，这个候选分支就会被执行。
5. 如果select语句发现同时有多个候选分支满足选择条件，那么它就会用一种伪随机的算法在这些分支中选择一个并执行。注意，即使select语句是在被唤醒时发现的这种情况，也会这样做。
6. 一条select语句中只能够有一个默认分支。并且，默认分支只在无候选分支可选时才会被执行，这与它的编写位置无关。

7. `select` 语句的每次执行，包括 `case` 表达式求值和分支选择，都是独立的。不过，至于它的执行是否是并发安全的，就要看其中的 `case` 表达式以及分支中，是否包含并发不安全的代码了。

我把与以上规则相关的示例放在 `demo25.go` 文件中了。你一定要去试运行一下，然后尝试用上面的规则去解释它的输出内容。

总结

今天，我们先讲了单向通道的表示方法，操作符 “`<-`” 仍然是关键。如果只用一个词来概括单向通道存在的意义的话，那就是“约束”，也就是对代码的约束。

我们可以使用带 `range` 子句的 `for` 语句从通道中获取数据，也可以通过 `select` 语句操纵通道。

`select` 语句是专门为通道而设计的，它可以包含若干个候选分支，每个分支中的 `case` 表达式都会包含针对某个通道的发送或接收操作。

当 `select` 语句被执行时，它会根据一套**分支选择规则**选中某一个分支并执行其中的代码。如果所有的候选分支都没有被选中，那么默认分支（如果有的话）就会被执行。注意，发送和接收操作的阻塞是分支选择规则的一个很重要的依据。

思考题

今天的思考题都由上述内容中的线索延伸而来。

1. 如果在 `select` 语句中发现某个通道已关闭，那么应该怎样屏蔽掉它所在的分支？
2. 在 `select` 语句与 `for` 语句联用时，怎样直接退出外层的 `for` 语句？

[戳此查看 Go 语言专栏文章配套详细代码。](#)

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 10 | 通道的基本操作

下一篇 12 | 使用函数的正确姿势

精选留言 (36)

写留言



江山如画

2018-09-05

20

感觉方法应该挺多，就看解决的是不是优雅

第一个问题：发现某个channel被关闭后，为了防止再次进入这个分支，可以把这个channel重新赋值成为一个长度为0的非缓冲通道，这样这个case就一直被阻塞了：

for {...

展开



任性😁

2018-09-25

12

demo24里边少了rand.Seed(time.Now().Unix())，不然每次随机数都是固定的顺序



笨笨

2018-09-05

👍 7

谢谢赫老师今日分享，回答问题如下

- 1.对于select中被close的channel判断其第二个boolean参数，如果是false则被关闭，那么赋值此channel为nil，那么每次到这个nil的channel就会阻塞，select会忽略阻塞的通道，如果再搭配上default就一定能保证不会被阻塞了。
 - 2.通过定义标签，配合goto或者break能实现在同一个函数内任意跳转，故可以跳出多层...
- 展开 ▾

作者回复: 你的问题是什么？



癫狂的小兵

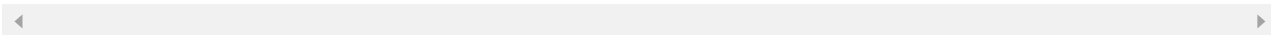
2018-09-05

👍 4

请问当select语句发现多个分支满足条件时随机选择一个分支执行 那怎样让其他满足条件的分支执行呢？ for 循环 等待下一次循环时再执行？

展开 ▾

作者回复: 放在for循环里每次也是随机的，不过可以用for循环，或者再次执行select语句。



zhaopan

2019-02-19

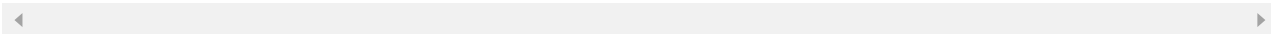
👍 3

老师好:

仅当select语句中的所有case表达式都被求值完毕后，它才会开始选择候选分支。
当接收通道操作有多个满足条件时, 这里的所有case表达式都被求值完毕, 应该怎么理解?
是多个case表达式都能接收到通道的数据么?
如果都接收了, 随机选择一个分支去处理接收的通道数据. 那其他满足条件的case分支怎...

展开 ▾

作者回复: 这里只会检查一下接收操作或发送操作是否可以进行（是否不会被阻塞）。有兴趣的话可以看一下 runtime/select.go 中的 selectgo 函数的源码。





破破

2018-09-11

👍 3

第一个问题，如果判断到chan关闭，即取到的第二个值为false。则将该chan赋值为nil。
第二个问题，根据情况使用goto或者return。或者加一个是否结束的标识，goto然后用两个break。



Kennedy

2018-09-10

👍 3

通道的类型如果是方法，性能会差吗？

展开 ▾



阿海

2019-01-05

👍 2

运行了demo25.go, 发现结果是No candidate case is selected，原因跟
var channels = [3]chan int{
 nil,
 make(chan int, 1),
 nil,...

展开 ▾



heha37

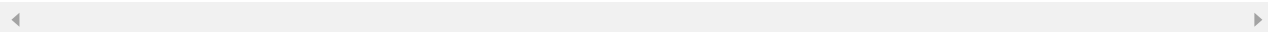
2018-11-05

👍 2

当第二个boolean参数为false的时候，在相应的case中设置chan为nil零值，再次case求值的时候会遭遇阻塞，会屏蔽该case。

展开 ▾

作者回复: 是的。



许森森

2018-09-14

👍 2

1 发现channel是closed之后，重新make，使得为nil，保证一直阻塞。

```
i := 0  
for {
```

select {...

展开 ▾



左氧佛沙星...

2019-03-25

👍 1

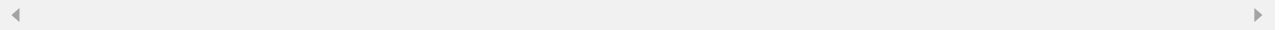
老师好，demo25中的这段代码我没看懂，不是这个匹配上了吗？为啥没有执行呢？我理解应该打印The second candidate case is selected。。。。

```
case getChan(1) <- getNumber(1):
```

```
`` ...
```

展开 ▾

作者回复: 因为 make(chan int) 初始化的是不带缓冲的通道。非缓冲通道只有在收发双方都就绪的情况下才能传递元素值，否则就阻塞。



Cxb

2019-03-07

👍 1

@王小勃

demo25 不输出second candidate 原因应该是channels[1]是非缓冲通道，select语句检测到阻塞状态，所以case语句不成立。把channels[1]设置为缓冲通道，或者写个协程接受channels[1]就可以输出second candidate

```
func main() {...
```

展开 ▾



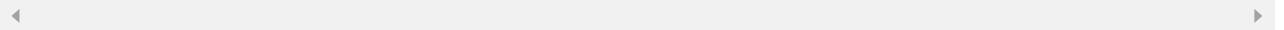
一只傻哈皮

2019-02-22

👍 1

请问select伪随机执行的目的是什么呢？没太理解这样做的目的。

作者回复: 这属于语言特性之一，没有什么特殊原因。你可以理解为避免case书写顺序影响到执行顺序。



王小勃

2018-12-21

👍 1

有个问题不明：demo25中 select为什么选择的不是第二个case输出？

第二个case实际上在往channels[1] <- 2 ,那为什么不输出：The second candidate case is selected.

package main

...

展开 ▾



到不了的塔

2018-10-24

👍 1

郝老师，请问第一题的答案是啥，不知道怎么屏蔽分支呢

作者回复: 设置为nil就可以了。



一步

2018-09-14

👍 1

老师你上面的发送通道<-是代表，可以向通道中发送数据呢？那对于通道而言是不是就是，通道接收数据，叫做接收通道？

展开 ▾

作者回复: 只能向它发送数据的我简称为发送通道。



Neo

2018-09-12

👍 1

请问老师：

select 分支选择规则中第5个："如果select语句发现同时有多个候选分支满足选择条件，那么它就会用一种伪随机算法在这些分支中选择一个执行" 随机选一个执行 那我们是不是就不能确定程序会执行哪一条与语句了？

展开 ▾

作者回复: 这种情况下是这样。





michael

2018-09-09

👍 1

第二个问题，可以将for select写到一个函数里面，用return的方式来跳出for循环



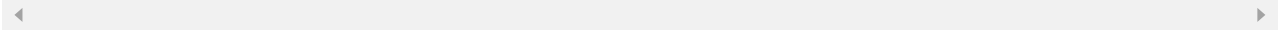
王腾飞

2018-09-08

👍 1

如何动态的调整select的channel，比如我能否select一个channel切片，然后根据需要调整切片的内容？

作者回复: 切片不是并发安全的，在这种场景下最好不要用。



冰激凌的眼...

2018-09-07

👍 1

单向通道乍看没什么作用，但是看了后面文章，看到在函数传参、返回时可以通过双向通道的退化形成单向通道，就明白作者说的约束行为的意义了。

展开 ▾