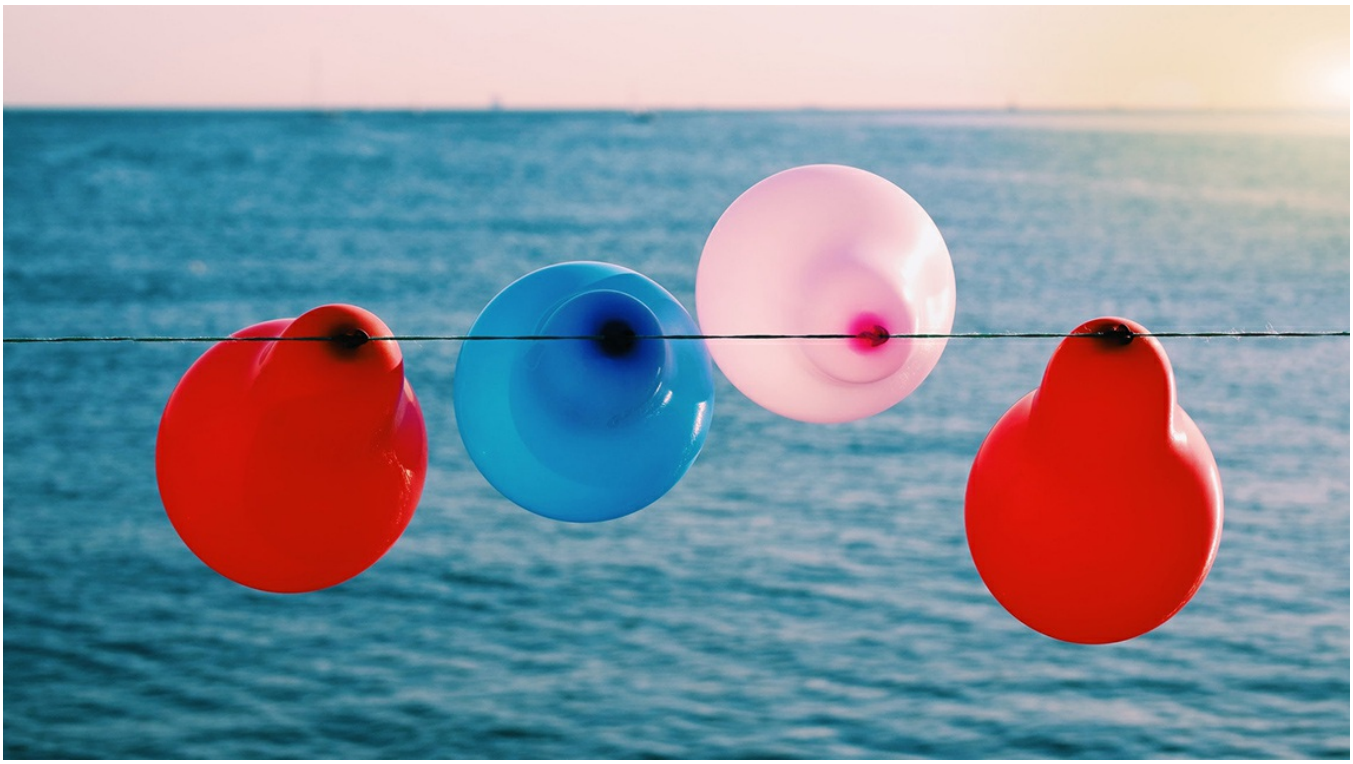
11 | 通道的高级玩法

2018-09-05 郝林



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：

```
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：

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

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：

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

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。

```
intChan1 := make(chan int, 3)
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：

```
func getIntChan() <-chan int {
 num := 5
 ch := make(chan int, num)
 for i := 0; i < num; i++ {
  ch <- i
 }
 close(ch)
 return ch
}
```

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：

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

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种取出元素值，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。

```
// 准备好几个通道。
intChannels := [3]chan int{
 make(chan int, 1),
 make(chan int, 1),
 make(chan int, 1),
}
// 随机选择一个通道，并向它发送元素值。
index := rand.Intn(3)
fmt.Printf("The index: %d\n", index)
intChannels[index] <- index
// 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。
select {
case <-intChannels[0]:
 fmt.Println("The first candidate case is selected.")
case <-intChannels[1]:
 fmt.Println("The second candidate case is selected.")
case elem := <-intChannels[2]:
 fmt.Printf("The third candidate case is selected, the element is %d.\n", elem)
default:
 fmt.Println("No candidate case is selected!")
}
```

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。

```
intChan := make(chan int, 1)
// 一秒后关闭通道。
time.AfterFunc(time.Second, func() {
 close(intChan)
})
select {
case _, ok := <-intChan:
 if !ok {
  fmt.Println("The candidate case is closed.")
  break
 }
 fmt.Println("The candidate case is selected.")
}
```

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。