

## 10 | 通道的基本操作

2018-09-03 郝林



作为Go语言最有特色的数据类型，通道（**channel**）完全可以与**goroutine**（也可称为go程）并驾齐驱，共同代表Go语言独有的并发编程模式和编程哲学。

Don't communicate by sharing memory; share memory by communicating. （不要通过共享内存来通信，而应该通过通信来共享内存。）

这是作为Go语言的主要创造者之一的**Rob Pike**的至理名言，这也充分体现了Go语言最重要的编程理念。而通道类型恰恰是后半句话的完美实现，我们可以利用通道在多个**goroutine**之间传递数据。

### 前导内容：通道的基础知识

通道类型的值本身就是并发安全的，这也是Go语言自带的、唯一一个可以满足并发安全性的类型。它使用起来十分简单，并不会徒增我们的心智负担。

在声明并初始化一个通道的时候，我们需要用到Go语言的内建函数**make**。就像用**make**初始化切片那样，我们传给这个函数的第一个参数应该是代表了通道的具体类型的类型字面量。

在声明一个通道类型变量的时候，我们首先要确定该通道类型的元素类型，这决定了我们可以通过这个通道传递什么类型的数据。

比如，类型字面量**chan int**，其中的**chan**是表示通道类型的关键字，而**int**则说明了该通道

类型的元素类型。又比如，`chan string`代表了一个元素类型为`string`的通道类型。

在初始化通道的时候，`make`函数除了必须接收这样的类型字面量作为参数，还可以接收一个`int`类型的参数。

后者是可选的，用于表示该通道的容量。所谓通道的容量，就是指通道最多可以缓存多少个元素值。由此，虽然这个参数是`int`类型的，但是它是不能小于0的。

当容量为0时，我们可以称通道为非缓冲通道，也就是不带缓冲的通道。而当容量大于0时，我们可以称为缓冲通道，也就是带有缓冲的通道。非缓冲通道和缓冲通道有着不同的数据传递方式，这个我在后面会讲到。

一个通道相当于一个先进先出（**FIFO**）的队列。也就是说，通道中的各个元素值都是严格地按照发送的顺序排列的，先被发送通道的元素值一定会先被接收。元素值的发送和接收都需要用到操作符`<-`。我们也可以叫它接送操作符。一个左尖括号紧接着一个减号形象地代表了元素值的传输方向。

```
package main

import "fmt"

func main() {
    ch1 := make(chan int, 3)
    ch1 <- 2
    ch1 <- 1
    ch1 <- 3
    elem1 := <-ch1
    fmt.Printf("The first element received from channel ch1: %v\n",
        elem1)
}
```

在`demo20.go`文件中，我声明并初始化了一个元素类型为`int`、容量为3的通道`ch1`，并用三条语句，向该通道先后发送了三个元素值2、1和3。

这里的语句需要这样写：依次敲入通道变量的名称（比如`ch1`）、接送操作符`<-`以及想要发送的元素值（比如2），并且这三者之间最好用空格进行分割。

这显然表达了“这个元素值将被发送该通道”这个语义。由于该通道的容量为3，所以，我可以在通道不包含任何元素值的时候，连续地向该通道发送三个值，此时这三个值都会被缓存在通道之

中。

当我们需要从通道接收元素值的时候，同样要用接送操作符`<-`，只不过，这时需要把它写在变量名的左边，用于表达“要从该通道接收一个元素值”的语义。

比如：`<-ch1`，这也可以被叫做接收表达式。在一般情况下，接收表达式的结果将会是通道中的一个元素值。

如果我们需要把如此得来的元素值存起来，那么在接收表达式的左边就需要依次添加赋值符号（`=`或`:=`）和用于存值的变量的名字。因此，语句`elem1 := <-ch1`会将最先进入`ch1`的元素接收来并存入变量`elem1`。

现在来看一道与此有关的题目。今天的问题是：**对通道的发送和接收操作都有哪些基本的特性？**

这个问题的背后隐藏着很多的知识点，我们来看一下典型回答。

它们的基本特性如下。

1. 对于同一个通道，发送操作之间是互斥的，接收操作之间也是互斥的。
2. 发送操作和接收操作中对元素值的处理都是不可分割的。
3. 发送操作在完全完成之前会被阻塞。接收操作也是如此。

## 问题解析

我们先来看**第一个基本特性**。在同一时刻，Go语言的运行时系统（以下简称运行时系统）只会执行对同一个通道的任意个发送操作中的某一个。

直到这个元素值被完全复制进该通道之后，其他针对该通道的发送操作才可能被执行。

类似的，在同一时刻，运行时系统也只会执行，对同一个通道的任意个接收操作中的某一个。

直到这个元素值完全被移出该通道之后，其他针对该通道的接收操作才可能被执行。即使这些操作是并发执行的也是如此。

这里所谓的并发执行，你可以这样认为，多个代码块分别在不同的goroutine之中，并有机会在同一个时间段内被执行。

另外，对于通道中的同一个元素值来说，发送操作和接收操作之间也是互斥的。例如，虽然会出现，正在被复制进通道但还未复制完成的元素值，但是这时它绝不会被想接收它的一方看到和取走。

这里要注意的一个细节是，元素值从外界进入通道时会被复制。更具体地说，进入通道的并不是在接收操作符右边的那个元素值，而是它的副本。

另一方面，元素值从通道进入外界时会被移动。这个移动操作实际上包含了两步，第一步是生成正在通道中的这个元素值的副本，并准备给到接收方，第二步是删除在通道中的这个元素值。

**顺着这个细节再来看第二个基本特性。** 这里的“不可分割”的意思是，它们处理元素值时都是一气呵成的，绝不会被打断。

例如，发送操作要么还没复制元素值，要么已经复制完毕，绝不会出现只复制了一部分的情况。

又例如，接收操作在准备好元素值的副本之后，一定会删除掉通道中的原值，绝不会出现通道中仍有残留的情况。

这既是为了保证通道中元素值的完整性，也是为了保证通道操作的唯一性。对于通道中的同一个元素值来说，它只可能是某一个发送操作放入的，同时也只可能被某一个接收操作取出。

**再来说第三个基本特性。** 一般情况下，发送操作包括了“复制元素值”和“放置副本到通道内部”这两个步骤。

在这两个步骤完全完成之前，发起这个发送操作的那句代码会一直阻塞在那里。也就是说，在它之后的代码不会有执行的机会，直到这句代码的阻塞解除。

更细致地说，在通道完成发送操作之后，运行时系统会通知这句代码所在的goroutine，以使它去争取继续运行代码的机会。

另外，接收操作通常包含了“复制通道内的元素值”“放置副本到接收方”“删掉原值”三个步骤。

在所有这些步骤完全完成之前，发起该操作的代码也会一直阻塞，直到该代码所在的goroutine收到了运行时系统的通知并重新获得运行机会为止。

说到这里，你可能已经感觉到，**如此阻塞代码其实就是为了实现操作的互斥和元素值的完整。**

下面我来说一个关于通道操作阻塞的问题。

## 知识扩展

### 问题1：发送操作和接收操作在什么时候可能被长时间的阻塞？

先说针对**缓冲通道**的情况。如果通道已满，那么对它的所有发送操作都会被阻塞，直到通道中有元素值被接收走。

这时，通道会优先通知最早因此而等待的、那个发送操作所在的goroutine，后者会再次执行发送操作。

由于发送操作在这种情况下被阻塞后，它们所在的goroutine会顺序地进入通道内部的发送等待

队列，所以通知的顺序总是公平的。

相对的，如果通道已空，那么对它的所有接收操作都会被阻塞，直到通道中有新的元素值出现。这时，通道会通知最早等待的那个接收操作所在的`goroutine`，并使它再次执行接收操作。

因此而等待的、所有接收操作所在的`goroutine`，都会按照先后顺序被放入通道内部的接收等待队列。

对于**非缓冲通道**，情况要简单一些。无论是发送操作还是接收操作，一开始执行就会被阻塞，直到配对的操作也开始执行，才会继续传递。由此可见，非缓冲通道是在用同步的方式传递数据。也就是说，只有收发双方对接上了，数据才会被传递。

并且，数据是直接从发送方复制到接收方的，中间并不会用非缓冲通道做中转。相比之下，缓冲通道则在用异步的方式传递数据。

在大多数情况下，缓冲通道会作为收发双方的中间件。正如前文所述，元素值会先从发送方复制到缓冲通道，之后再由缓冲通道复制给接收方。

但是，当发送操作在执行的时候发现空的通道中，正好有等待的接收操作，那么它会直接把元素值复制给接收方。

以上说的都是在正确使用通道的前提下会发生的事情。下面我特别说明一下，由于错误使用通道而造成的阻塞。

对于值为`nil`的通道，不论它的具体类型是什么，对它的发送操作和接收操作都会永久地处于阻塞状态。它们所属的`goroutine`中的任何代码，都不再会被执行。

注意，由于通道类型是引用类型，所以它的零值就是`nil`。换句话说，当我们只声明该类型的变量但没有用`make`函数对它进行初始化时，该变量的值就会是`nil`。我们一定不要忘记初始化通道！

你可以去看一下[demo21.go](#)，我在里面用代码罗列了一下会造成阻塞的几种情况。

## 问题2：发送操作和接收操作在什么时候会引发panic？

对于一个已初始化，但并未关闭的通道来说，收发操作一定不会引发`panic`。但是通道一旦关闭，再对它进行发送操作，就会引发`panic`。

另外，如果我们试图关闭一个已经关闭了的通道，也会引发`panic`。注意，接收操作是可以感知到通道的关闭的，并能够安全退出。

更具体地说，当我们把接收表达式的结果同时赋给两个变量时，第二个变量的类型就是**一定**`bool`类型。它的值如果为`false`就说明通道已经关闭，并且再没有元素值可取了。

注意，如果通道关闭时，里面还有元素值未被取出，那么接收表达式的第一个结果，仍会是通道中的某一个元素值，而第二个结果值一定会是`true`。

因此，通过接收表达式的第二个结果值，来判断通道是否关闭是可能有延时的。

由于通道的收发操作有上述特性，所以除非有特殊的保障措施，我们千万不要让接收方关闭通道，而应当让发送方做这件事。这在`demo22.go`中有一个简单的模式可供参考。

## 总结

今天我们讲到了通道的一些常规操作，包括初始化、发送、接收和关闭。通道类型是Go语言特有的，所以你一开始肯定会感到陌生，其中的一些规则和奥妙还需要你铭记于心，并细心体会。

首先是在初始化通道时设定其容量的意义，这有时会让通道拥有不同的行为模式。对通道的发送操作和接收操作都有哪些基本特性，也是我们必须清楚的。

这涉及了它们什么时候会互斥，什么时候会造成阻塞，什么时候会引起`panic`，以及它们收发元素值的顺序是怎样的，它们是怎样保证元素值的完整性的，元素值通常会被复制几次，等等。

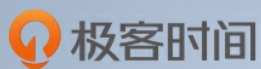
最后别忘了，通道也是Go语言的并发编程模式中重要的一员。

## 思考题

我希望能通过试验获得下述问题的答案。

1. 通道的长度代表着什么？它在什么时候会通道的容量相同？
2. 元素值在经过通道传递时会被复制，那么这个复制是浅表复制还是深层复制呢？

[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



