课程安排，请关注**微信公众平台**或者**官方微博**

编程语言： Golang 与 html5

编程工具： Goland 和 HBuilder

预计平均一周左右更新一或二节课程

授人以鱼，不如授人以渔业。

大家好，

欢迎来到 **字节教育** 课程的学习

字节教育官网：www.ByteEdu.Com

腾讯课堂地址：Gopher.ke.qq.Com

技术交流群 ： 221 273 219

微信公众号 ：**Golang 语言社区**

微信服务号 ：**Golang 技术社区**

目录：

第一季　Golang 语言社区-综合面试题

初级面试题　channel 解释

一、公众账号：

回复关键字：客服　　获取课程助教的微信

二、核心内容讲解：

Channel 是 Go 中的一个核心类型，你可以把它看成一个管道，通过它并发核心单元就可以发送或者接收数据进行通讯(communication)。

它的操作符是箭头 <- 。

[plain] view plain copy print?

1. ch <- v // 发送值 v 到 Channel ch 中
2. v := <-ch // 从 Channel ch 中接收数据，并将数据赋值给 v

```
ch <- v    // 发送值 v 到 Channel ch 中
v := <-ch  // 从 Channel ch 中接收数据，并将数据赋值给 v
```

(箭头的指向就是数据的流向)

就像 map 和 slice 数据类型一样, channel 必须先创建再使用:

[plain] view plain copy print?

1. ch := make(chan int)

```
ch := make(chan int)
```

Channel 类型

Channel 类型的定义格式如下：

[plain] view plain copy print?

1. ChannelType = ("chan" | "chan" "<-" | "<-" "chan") ElementType .

```
ChannelType = ( "chan" | "chan" "<-" | "<-" "chan" ) ElementType .
```

它包括三种类型的定义。可选的<-代表 channel 的方向。如果没有指定方向，那么 Channel 就是双向的，既可以接收数据，也可以发送数据。

[plain] view plain copy print?

1. chan T // 可以接收和发送类型为 T 的数据
2. chan<- float64 // 只可以用来发送 float64 类型的数据
3. <-chan int // 只可以用来接收 int 类型的数据

```
chan T          // 可以接收和发送类型为 T 的数据
chan<- float64  // 只可以用来发送 float64 类型的数据
<-chan int      // 只可以用来接收 int 类型的数据
```

<-总是优先和最左边的类型结合。(The <- operator associates with the leftmost chan possible)

[plain] view plain copy print?

1. chan<- chan int // 等价 chan<- (chan int)
2. chan<- <-chan int // 等价 chan<- (<-chan int)

3. <-chan <-chan int // 等价 <-chan (<-chan int)
4. chan (<-chan int)

```
chan<- chan int    // 等价 chan<- (chan int)
chan<- <-chan int  // 等价 chan<- (<-chan int)
<-chan <-chan int  // 等价 <-chan (<-chan int)
chan (<-chan int)
```

使用 `make` 初始化 Channel,并且可以设置容量:

[plain] view plain copy print?

1. make(chan int, 100)

```
make(chan int, 100)
```

容量(capacity)代表 Channel 容纳的最多的元素的数量，代表 Channel 的缓存的大小。
如果没有设置容量，或者容量设置为 0, 说明 Channel 没有缓存，只有 sender 和 receiver 都准备好了后它们的通讯(communication)才会发生(Blocking)。如果设置了缓存，就有可能不发生阻塞， 只有 buffer 满了后 send 才会阻塞，而只有缓存空了后 receive 才会阻塞。一个 nil channel 不会通信。

可以通过内建的 `close` 方法可以关闭 Channel。

你可以在多个 goroutine 从/往 一个 channel 中 receive/send 数据, 不必考虑额外的同步措施。

Channel 可以作为一个先入先出(FIFO)的队列，接收的数据和发送的数据的顺序是一致的。

channel 的 receive 支持 multi-valued assignment，如

[plain] view plain copy print?

1. v, ok := <-ch

```
v, ok := <-ch
```

它可以用来检查 Channel 是否已经被关闭了。

1. send 语句
   send 语句用来往 Channel 中发送数据， 如 `ch <- 3`。
   它的定义如下:

[plain] view plain copy print?

1. SendStmt = Channel "<-" Expression .
2. Channel = Expression .

```
SendStmt = Channel "<-" Expression .
Channel  = Expression .
```

在通讯(communication)开始前 channel 和 expression 必选先求值出来(evaluated)，比如下面的(3+4)先计算出 7 然后再发送给 channel。

[plain] view plain copy print?

1. c := make(chan int)
2. defer close(c)
3. go func() { c <- 3 + 4 }()

4. i := <-c
5. fmt.Println(i)

```
c := make(chan int)
defer close(c)
go func() { c <- 3 + 4 }()
i := <-c
fmt.Println(i)
```

send 被执行前(proceed)通讯(communication)一直被阻塞着。如前所言，无缓存的 channel 只有在 receiver 准备好后 send 才被执行。如果有缓存，并且缓存未满，则 send 会被执行。

往一个已经被 close 的 channel 中继续发送数据会导致 run-time panic。

往 nil channel 中发送数据会一致被阻塞着。

1. receive 操作符
   `<-ch` 用来从 channel ch 中接收数据，这个表达式会一直被 block,直到有数据可以接收。

从一个 nil channel 中接收数据会一直被 block。

从一个被 close 的 channel 中接收数据不会被阻塞，而是立即返回，接收完已发送的数据后会返回元素类型的零值(zero value)。

如前所述，你可以使用一个额外的返回参数来检查 channel 是否关闭。

[plain] **view plain copy print?**

1. x, ok := <-ch
2. x, ok = <-ch
3. var x, ok = <-ch

```
x, ok := <-ch
x, ok = <-ch
var x, ok = <-ch
```

如果 OK 是 false，表明接收的 x 是产生的零值，这个 channel 被关闭了或者为空。

blocking

缺省情况下，发送和接收会一直阻塞着，直到另一方准备好。这种方式可以用来在 gororutine 中进行同步，而不必使用显示的锁或者条件变量。

如官方的例子中 `x, y := <-c, <-c` 这句会一直等待计算结果发送到 channel 中。

[plain] **view plain copy print?**

```
1.  import "fmt"
2.  func sum(s []int, c chan int) {
3.      sum := 0
4.      for _, v := range s {
5.          sum += v
6.      }
7.      c <- sum // send sum to c
8.  }
9.  func main() {
10.     s := []int{7, 2, 8, -9, 4, 0}
11.     c := make(chan int)
```

```
12.     go sum(s[:len(s)/2], c)
13.     go sum(s[len(s)/2:], c)
14.     x, y := <-c, <-c // receive from c
15.     fmt.Println(x, y, x+y)
16. }
```

```
import "fmt"
func sum(s []int, c chan int) {
          sum := 0
          for _, v := range s {
                    sum += v
          }
          c <- sum // send sum to c
}
func main() {
          s := []int{7, 2, 8, -9, 4, 0}
          c := make(chan int)
          go sum(s[:len(s)/2], c)
          go sum(s[len(s)/2:], c)
          x, y := <-c, <-c // receive from c
          fmt.Println(x, y, x+y)
}
```

Buffered Channels

make 的第二个参数指定缓存的大小：`ch := make(chan int, 100)`。

通过缓存的使用，可以尽量避免阻塞，提供应用的性能。

Range

`for …… range` 语句可以处理 Channel。

[plain] **view plain copy print?**

```
1.  func main() {
2.      go func() {
3.          time.Sleep(1 * time.Hour)
4.      }()
5.      c := make(chan int)
6.      go func() {
7.          for i := 0; i < 10; i = i + 1 {
8.              c <- i
9.          }
10.         close(c)
11.     }()
12.     for i := range c {
13.         fmt.Println(i)
14.     }
```

```
15.     fmt.Println("Finished")
16. }
```

```
func main() {
        go func() {
                time.Sleep(1 * time.Hour)
        }()
        c := make(chan int)
        go func() {
                for i := 0; i < 10; i = i + 1 {
                        c <- i
                }
                close(c)
        }()
        for i := range c {
                fmt.Println(i)
        }
        fmt.Println("Finished")
}
```

`range c` 产生的迭代值为 Channel 中发送的值，它会一直迭代直到 channel 被关闭。上面的例子中如果把 `close(c)`注释掉，程序会一直阻塞在 `for …… range` 那一行。

select

`select` 语句选择一组可能的 send 操作和 receive 操作去处理。它类似 `switch`,但是只是用来处理通讯(communication)操作。
它的 `case` 可以是 send 语句，也可以是 receive 语句，亦或者 `default`。

`receive` 语句可以将值赋值给一个或者两个变量。它必须是一个 receive 操作。

最多允许有一个 `default case`,它可以放在 case 列表的任何位置，尽管我们大部分会将它放在最后。

[plain] view plain copy print?

```
1.  import "fmt"
2.  func fibonacci(c, quit chan int) {
3.      x, y := 0, 1
4.      for {
5.          select {
6.          case c <- x:
7.              x, y = y, x+y
8.          case <-quit:
9.              fmt.Println("quit")
10.             return
11.         }
12.     }
13. }
14. func main() {
15.     c := make(chan int)
16.     quit := make(chan int)
17.     go func() {
18.         for i := 0; i < 10; i++ {
19.             fmt.Println(<-c)
20.         }
```

```
21.         quit <- 0
22.     }()
23.     fibonacci(c, quit)
24. }
```

```
import "fmt"
func fibonacci(c, quit chan int) {
        x, y := 0, 1
        for {
                select {
                case c <- x:
                        x, y = y, x+y
                case <-quit:
                        fmt.Println("quit")
                        return
                }
        }
}
func main() {
        c := make(chan int)
        quit := make(chan int)
        go func() {
                for i := 0; i < 10; i++ {
                        fmt.Println(<-c)
                }
                quit <- 0
        }()
        fibonacci(c, quit)
}
```

如果有同时多个 case 去处理,比如同时有多个 channel 可以接收数据，那么 Go 会伪随机的选择一个 case 处理(pseudo-random)。如果没有 case 需要处理，则会选择 `default` 去处理，如果 `default case` 存在的情况下。如果没有 `default case`，则 `select` 语句会阻塞，直到某个 case 需要处理。

需要注意的是，nil channel 上的操作会一直被阻塞，如果没有 default case,只有 nil channel 的 select 会一直被阻塞。

`select` 语句和 `switch` 语句一样，它不是循环，它只会选择一个 case 来处理，如果想一直处理 channel，你可以在外面加一个无限的 for 循环：

[plain] view plain copy print?

```
1.  for {
2.      select {
3.      case c <- x:
4.          x, y = y, x+y
5.      case <-quit:
6.          fmt.Println("quit")
7.          return
8.      }
9.  }
```

```
for {
```

```
        select {
        case c <- x:
                x, y = y, x+y
        case <-quit:
                fmt.Println("quit")
                return
        }
}
```

timeout

`select` 有很重要的一个应用就是超时处理。 因为上面我们提到，如果没有 case 需要处理，select 语句就会一直阻塞着。这时候我们可能就需要一个超时操作，用来处理超时的情况。
下面这个例子我们会在 2 秒后往 channel c1 中发送一个数据，但是 `select` 设置为 1 秒超时,因此我们会打印出 `timeout 1`,而不是 `result 1`。

[plain] view plain copy print?

```
1.  import "time"
2.  import "fmt"
3.  func main() {
4.      c1 := make(chan string, 1)
5.      go func() {
6.          time.Sleep(time.Second * 2)
7.          c1 <- "result 1"
8.      }()
9.      select {
10.     case res := <-c1:
11.         fmt.Println(res)
12.     case <-time.After(time.Second * 1):
13.         fmt.Println("timeout 1")
14.     }
15. }
```

```
import "time"
import "fmt"
func main() {
    c1 := make(chan string, 1)
    go func() {
        time.Sleep(time.Second * 2)
        c1 <- "result 1"
    }()
    select {
    case res := <-c1:
        fmt.Println(res)
    case <-time.After(time.Second * 1):
        fmt.Println("timeout 1")
    }
```

```
}
```

其实它利用的是 `time.After` 方法，它返回一个类型为`<-chan Time` 的单向的 channel，在指定的时间发送一个当前时间给返回的 channel 中。

Timer 和 Ticker

我们看一下关于时间的两个 Channel。
timer 是一个定时器，代表未来的一个单一事件，你可以告诉 timer 你要等待多长时间，它提供一个 Channel，在将来的那个时间那个 Channel 提供了一个时间值。下面的例子中第二行会阻塞 2 秒钟左右的时间，直到时间到了才会继续执行。

[plain] **view plain** **copy** **print?**

```
1.  timer1 := time.NewTimer(time.Second * 2)
2.  <-timer1.C
3.  fmt.Println("Timer 1 expired")
```

```
timer1 := time.NewTimer(time.Second * 2)
<-timer1.C
fmt.Println("Timer 1 expired")
```

当然如果你只是想单纯的等待的话，可以使用 `time.Sleep` 来实现。

你还可以使用 `timer.Stop` 来停止计时器。

[plain] **view plain** **copy** **print?**

```
1.  timer2 := time.NewTimer(time.Second)
2.  go func() {
3.      <-timer2.C
4.      fmt.Println("Timer 2 expired")
5.  }()
6.  stop2 := timer2.Stop()
7.  if stop2 {
8.      fmt.Println("Timer 2 stopped")
9.  }
```

```
timer2 := time.NewTimer(time.Second)
go func() {
          <-timer2.C
          fmt.Println("Timer 2 expired")
}()
stop2 := timer2.Stop()
if stop2 {
          fmt.Println("Timer 2 stopped")
}
```

`ticker` 是一个定时触发的计时器，它会以一个间隔(interval)往 Channel 发送一个事件(当前时间)，而 Channel 的接收者可以以固定的时间间隔从 Channel 中读取事件。下面的例子中 ticker 每 500 毫秒触发一次，你可以观察输出的时间。

[plain] **view plain** **copy** **print?**

```
1.  ticker := time.NewTicker(time.Millisecond * 500)
2.  go func() {
```

```
3.      for t := range ticker.C {
4.          fmt.Println("Tick at", t)
5.      }
6.  }()
```

```
ticker := time.NewTicker(time.Millisecond * 500)
go func() {
          for t := range ticker.C {
                    fmt.Println("Tick at", t)
          }
}()
```

类似 timer, ticker 也可以通过 `Stop` 方法来停止。一旦它停止，接收者不再会从 channel 中接收数据了。

close

内建的 close 方法可以用来关闭 channel。

总结一下 channel 关闭后 sender 的 receiver 操作。
如果 channel c 已经被关闭,继续往它发送数据会导致 `panic: send on closed channel`:

[plain] view plain copy print?

```
1.  import "time"
2.  func main() {
3.      go func() {
4.          time.Sleep(time.Hour)
5.      }()
6.      c := make(chan int, 10)
7.      c <- 1
8.      c <- 2
9.      close(c)
10.     c <- 3
11. }
```

```
import "time"
func main() {
          go func() {
                    time.Sleep(time.Hour)
          }()
          c := make(chan int, 10)
          c <- 1
          c <- 2
          close(c)
          c <- 3
}
```

但是从这个关闭的 channel 中不但可以读取出已发送的数据，还可以不断的读取零值:

[plain] view plain copy print?

```
1.  c := make(chan int, 10)
2.  c <- 1
3.  c <- 2
```

```
4. close(c)
5. fmt.Println(<-c) //1
6. fmt.Println(<-c) //2
7. fmt.Println(<-c) //0
8. fmt.Println(<-c) //0
```

```
c := make(chan int, 10)
c <- 1
c <- 2
close(c)
fmt.Println(<-c) //1
fmt.Println(<-c) //2
fmt.Println(<-c) //0
fmt.Println(<-c) //0
```

但是如果通过 `range` 读取，channel 关闭后 for 循环会跳出：

[plain] view plain copy print?

```
1. c := make(chan int, 10)
2. c <- 1
3. c <- 2
4. close(c)
5. for i := range c {
6.     fmt.Println(i)
7. }
```

```
c := make(chan int, 10)
c <- 1
c <- 2
close(c)
for i := range c {
          fmt.Println(i)
}
```

通过 `i, ok := <-c` 可以查看 Channel 的状态，判断值是零值还是正常读取的值。

```
1. c := make(chan int, 10)
2. close(c)
3. i, ok := <-c
4. fmt.Printf("%d, %t", i, ok) //0, false
```

```
c := make(chan int, 10)
close(c)
i, ok := <-c
fmt.Printf("%d, %t", i, ok) //0, false
```

同步

channel 可以用在 goroutine 之间的同步。
下面的例子中 main goroutine 通过 done channel 等待 worker 完成任务。 worker 做完任务后只需往 channel 发送一个数据就可以通知 main goroutine 任务完成。

```
1.  import (
2.      "fmt"
3.      "time"
4.  )
5.  func worker(done chan bool) {
6.      time.Sleep(time.Second)
7.      // 通知任务已完成
8.      done <- true
9.  }
10. func main() {
11.     done := make(chan bool, 1)
12.     go worker(done)
13.     // 等待任务完成
14.     <-done
15. }
```