2016年04月14日 • GO by smallnest

Go Channel 详解

目录 [+]

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 **<-** 。

```
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:

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

Channel类型

Channel类型的定义格式如下:

```
ChannelType = ( "chan" | "chan" "<-" | "<-" "chan" ) Element
```

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。

```
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)

```
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:

```
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

```
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:

```
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。

```
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。

```
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。

```
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
```

```
18
19          fmt.Println(x, y, x+y)
20  }
```

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。

```
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
13          for i := range c {
14                  fmt.Println(i)
15          }
16
17          fmt.Println("Finished")
18  }
```

`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。

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

```
21                          fmt.Println(<-c)
22                  }
23                  quit <- 0
24          }()
25          fibonacci(c, quit)
26  }
```

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：

```
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  }
```

1/ 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`。

```
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。

```
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。

```
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。

```
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`:

```
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:

```
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:

```
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。

```
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。

```
import (
        "fmt"
        "time"
)

func worker(done chan bool) {
        time.Sleep(time.Second)

        // 通知任务已完成
        done <- true
}

func main() {
        done := make(chan bool, 1)
        go worker(done)

        // 等待任务完成
        <-done
}
```

参考资料

1. https://gobyexample.com/channels
2. https://tour.golang.org/concurrency/2
3. https://golang.org/ref/spec#Select_statements
4. https://github.com/a8m/go-lang-cheat-sheet
5. http://devs.cloudimmunity.com/gotchas-and-common-mistakes-in-go-golang/
6. http://guzalexander.com/2013/12/06/golang-channels-tutorial.html

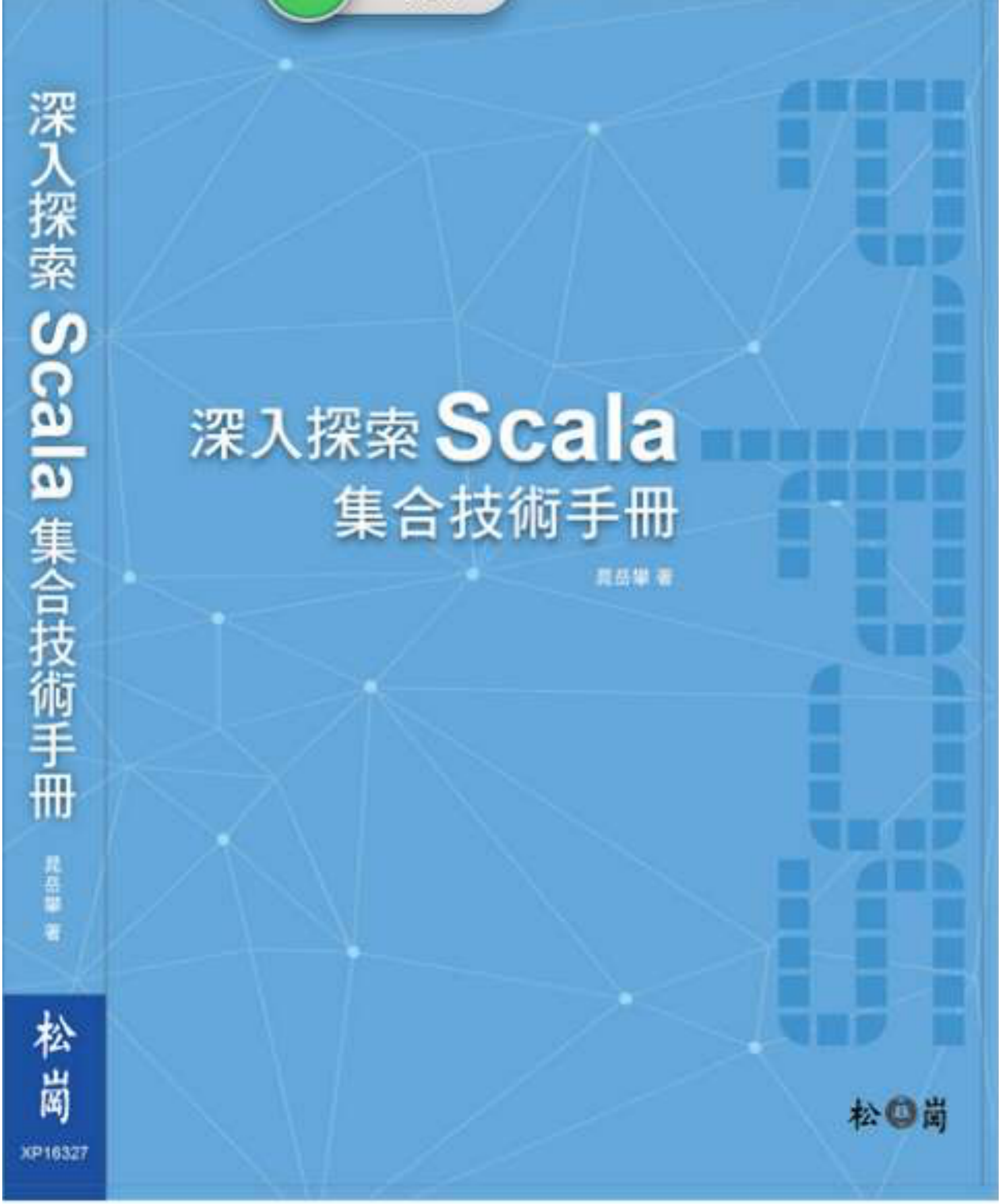
NEWER

[译]Go Stack Trace

OLDER

Go泛型提案


Broadview
www.broadview.com.cn
Scala集合技术手册
晁岳攀 著
Scala
中国工信出版集团
電子工業出版社



分类

- Android (12)
- DOTNET (1)
- Docker (2)

标签云

Android ApacheBench Bower C# CDN CQRS CRC CSS CompletableFuture Comsat Curator DSL Disruptor Docker Ember FastJson Fiber GAE GC Gnuplot Go Gradle Grunt Gulp Hadoop Hazelcast Ignite JVM Java Kafka Lambda Linux LongAdder MathJax Maven Memcached Metrics Mongo Netty Nginx

归档

近期文章

友情链接

以前的博客

技术栈

码农周刊
编程狂人周刊
importnew
并发编程网

github
stackoverflow
javacodegeeks
infoq
dzone
leetcode
jenkov