英文原文：Golang channels tutorial

标签： Go

愚安 推荐于 3年前 (共 11 段, 翻译完成于 12-12) (6评)

110人收藏此文章, 我要收藏

参与翻译(3人)： cmy00cmy, Garfielt, 愚安

仅中文 | 中英文对照 | 仅英文 | 打印此文章

Go语言内置了书写并发程序的工具。将go声明放到一个需调用的函数之前，在相同地址空间调用运行这个函数，这样该函数执行时便会作为一个独立的并发线程。这种线程在Go语言中称作goroutine。在这里我要提一下，并发并不总是意味着并行。Goroutines是指在硬件允许情况下创建能够并行执行程序的架构。这是这个主题的一次讨论：并发不是并行。

翻译的不错

Garfielt 顶 哦 3年前

2人顶

让我们从一个例子开始：

```
1  func main() {
2       // Start a goroutine and execute println concurrently
3       go println("goroutine message")
4       println("main function message")
5  }
```

这段程序将输出main function messageand **或者**goroutine message。我说“**或者**”是因为催生的goroutine有一些特点。当你运行一个goroutine时，调用的代码（在我们的例子里它是main函数）不等待goroutine完成，而是继续往下运行。在调用完println后main函数结束了它的执行，在Go语言里这意味着这个程序及所有催生的goroutines停止执行。但是，在这个发生之前，goroutine可能已经完成了其代码的执行并输出了goroutine message字符。

翻译的不错

Garfielt 顶 哦 3年前

2人顶

你明白这些后必须有方法来避免这种情况。这就是Go语言中**channels**的作用。

Channels 基础知识

翻译的不错

cmv00cmy 顶 哦 3年前

1人顶

Channels用来同步并发执行的函数并提供它们某种传值交流的机制。Channels的一些特性：通过channel传递的元素类型、容器（或缓冲区）和传递的方向由“<-”操作符指定。你可以使用内置函数 make分配一个channel:

```
1  i := make(chan int)       // by default the capacity is 0
2  s := make(chan string, 3) // non-zero capacity
3
4  r := make(<-chan bool)          // can only read from
5  w := make(chan<- []os.FileInfo) // can only write to
```

Channels是一个第一类值（一个对象在运行期间被创建，可以当做一个参数被传递，从子函数返回或者可以被赋给一个变量。）可以像其他值那样在任何地方使用：作为一个结构元素，函数参数、函数返回值甚至另一个channel的类型：

```
1  // a channel which:
2  //  - you can only write to
3  //  - holds another channel as its value
4  c := make(chan<- chan bool)
5
6  // function accepts a channel as a parameter
7  func readFromChannel(input <-chan string) {}
8
9  // function returns a channel
10 func getChannel() chan bool {
11      b := make(chan bool)
12      return b
13 }
```

其它翻译版本(1)

在读、写channel的时候要格外注意 <- 操作符。它的位置关乎到channel变量的读写操作。下面的例子标明了它的使用方法，但我还是要提醒你，这段代码 **并不会被完整地执行**，原因我们后面再讲：

翻译的不错

cmv00cmy 顶 哦 3年前

1人顶

```
1  func main() {
2       c := make(chan int)
3       c <- 42    // 写入channel
4       val := <-c // 从channel中读取
5       println(val)
6  }
```

现在我们知道了什么是channel，如何创建channel并且学了一些基础操作。现在让我们回到第一个示例，看看channel到底是如何帮助我们的。

```
1  func main() {
2       // 创建一个channel用以同步goroutine
3       done := make(chan bool)
4
5       // 在goroutine中执行输出操作
6       go func() {
7            println("goroutine message")
```

```
8
9             // 告诉main函数执行完毕.
10            // 这个channel在goroutine中是可见的
11            // 因为它是在相同的地址空间执行的.
12            done <- true
13      }()
14
15      println("main function message")
16      <-done // 等待goroutine结束
17  }
```

这个程序将顺溜地打印2条信息。为什么呢？因为channel没有缓冲（我们没有指定其容量）。所有基于未缓冲的channel的的操作会将操作锁死直到输出和接收全部准备就绪。这就是为什么未缓冲channel也被称作同步（synchronous）。在我们的例子中，主函数中的操作符<-将会把程序锁死直到goroutine在channel中写入数据。因此程序只有在读取操作成功结束后才会终止。

其它翻译版本(1)

为了避免存在一个channel的缓冲区所有读取操作都在没有锁定的情况下顺利完成（如果缓冲区是空的）并且写入操作也顺利结束（缓冲区不满），这样的channel被称作非同步的channel。下面是一个用来描述这两者区别的例子：

```
1   func main() {
2       message := make(chan string) // 无缓冲
3       count := 3
4
5       go func() {
6           for i := 1; i <= count; i++ {
7               fmt.Println("send message")
8               message <- fmt.Sprintf("message %d", i)
9           }
10      }()
11
12      time.Sleep(time.Second * 3)
13
14      for i := 1; i <= count; i++ {
15          fmt.Println(<-message)
16      }
17  }
```

在这个例子中，输出信息是一个同步的channel，程序输出结果为：

```
1   send message
2   // 等待3秒
3   message 1
4   send message
5   send message
6   message 2
7   message 3
```

正如你所看到的，在第一次goroutine中写入channel之后，其它在同一个channel中的写入操作都被锁住了，直到第一次读取操作执行完毕（大约3秒）。

现在我们提供一个缓冲区给输出信息的channel，例如：定义初始化行将被改为：message := make(chan string, 2)。这次程序输出将变为：

```
1   send message
2   send message
3   send message
4   // 等待3秒
5   message 1
6   message 2
7   message 3
```

这里我们看到所有的写操作的执行都不会等待第一次对缓冲区的读取操作结束，channel允许储存所有的三条信息。通过修改channel容器，我们通过可以控制处理信息的总数达到限制系统输出的目的。

翻译的不错

cmv00cmy 顶 哦！3年前

1人顶

死锁

现在让我们回到前面那个没有成功运行的读/写操作示例：

```
1   func main() {
2       c := make(chan int)
3       c <- 42    // 写入channel
4       val := <-c // 读取channel
5       println(val)
6   }
```

一旦运行此程序，你将得到以下错误：

```
1   fatal error: all goroutines are asleep - deadlock!
2
3   goroutine 1 [chan send]:
4   main.main()
5       /fullpathtofile/channelsio.go:5 +0x54
6   exit status 2
```

翻译的不错

cmv00cmy 顶 哦！3年前

2人顶

这个错误就是我们所知的**死锁**. 在这种情况下，两个goroutine互相等待对方释放资源，造成双方都无法继续运行。GO语言可以在运行时检测这种死锁并报错。这个错误是因为锁的自身特性产生的。

代码在次以单线程的方式运行，逐行运行。向channel写入的操作（c <- 42）会锁住整个程序的执行进程，因为在同步channel中的写操作只有在读取器准备就绪后才能成功执行。然而在这里，我们在写操作的下一行才创建了读取器。

为了使程序顺利执行，我们需要做如下改动：

```
func main() {
     c := make(chan int)

     // 使写操作在另一个goroutine中执行。
     go func() {
        c <- 42
     }()
     val := <-c
     println(val)
}
```

范围化的channels 和channel的关闭


翻译的不错

cmv00cmy
顶 哦!β年前

3人顶

在前面的一个例子中，我们向channel发送了多条信息并读取它们，读取器部分的代码如下：

```
for i := 1; i <= count; i++ {
     fmt.Println(<-message)
}
```

为了在执行读取操作的同时避免产生死锁，我们需要知道发送消息的确切数目，因为我们不能读取比写入条数还多的数据。但是这样很不方便，下面我们就提供了一个更为人性化的方法。

在Go语言中，存在一种称为范围表达式的代码，它允许程序反复声明数组、字符串、切片、图和channel，重复声明会一直持续到channel的关闭。请看下面的例子（虽然现在还不能执行）：

```
func main() {
     message := make(chan string)
     count := 3

     go func() {
          for i := 1; i <= count; i++ {
               message <- fmt.Sprintf("message %d", i)
          }
     }()

     for msg := range message {
          fmt.Println(msg)
     }
}
```


翻译的不错

cmv00cmy
顶 哦!β年前

1人顶

很不幸的是，这段代码现在还不能运行。正如我们之前提到的，范围（range）只有等到channel关闭后才会运行。因此我们需要使用 close 函数关闭channel，程序就会变成下面这个样子：

```
go func() {
     for i := 1; i <= count; i++ {
          message <- fmt.Sprintf("message %d", i)
     }
     close(message)
}()
```

关闭channel还有另外一个好处——被关闭的channel内的读取操作将不会引发锁，而是始终长生默认的对应channel类型的值：

```
done := make(chan bool)
close(done)

//不会产生锁，打印两次false
//因为false是bool类型的默认值
println(<-done)
println(<-done)
```

这个特性可以被用于控制goroutine的同步，让我们再回顾一下之前同步的例子：

```
func main() {
     done := make(chan bool)

     go func() {
          println("goroutine message")

          // 我们只关心被是否存在传送这个事实，而不是值的内容。
          done <- true
     }()

     println("main function message")
```

```
12      <-done
13  }
```

在这里，done channel仅仅被用于同步程序执行，而不是发送数据。再举一个类似的例子：

```
1   func main() {
2       // 与数据内容无关
3       done := make(chan struct{})
4
5       go func() {
6           println("goroutine message")
7
8           // 发送信号"I'm done"
9           close(done)
10      }()
11
12      println("main function message")
13      <-done
14  }
```

我们关闭了goroutine中的channel，读取操作不会产生锁，因此主函数可以继续执行下去。

多channel模式和channel的选择

在真正的项目开发中，你可能需要多个goroutine和channel。当各部分的独立性越强，他们之间也就越需要高效的同步措施。让我们看个略微复杂的例子：

```
1   func getMessagesChannel(msg string, delay time.Duration) <-chan string {
2       c := make(chan string)
3       go func() {
4           for i := 1; i <= 3; i++ {
5               c <- fmt.Sprintf("%s %d", msg, i)
6               // 在发送信息前等待
7               time.Sleep(time.Millisecond * delay)
8           }
9       }()
10      return c
11  }
12
13  func main() {
14      c1 := getMessagesChannel("first", 300)
15      c2 := getMessagesChannel("second", 150)
16      c3 := getMessagesChannel("third", 10)
17
18      for i := 1; i <= 3; i++ {
19          println(<-c1)
20          println(<-c2)
21          println(<-c3)
22      }
23  }
```

这里我们创建了一个方法，用来创建channel并定义了一个goroutine使之在一此调用中向channel发送三条信息。我们看到，c3理应是最后一次channel调用，所以它的输出信息应该在其它信息之前。但是我们得到的却是如下输出：

```
1   first 1
2   second 1
3   third 1
4   first 2
5   second 2
6   third 2
7   first 3
8   second 3
9   third 3
```

翻译的不错

cmv00cmy
顶 哦!3年前

1人顶

显然我们成功输出了所有的信息，这是因为第一个channel中的读取操作在每个循环声明中被锁住300毫秒，其它操作必须随之进入等待状态。而我们期望的却是从所有channel中尽快读取信息。

我们可以使用select 在多个channel之间进行选择。这种选择类似于普通的switch，但是所有的情况在这里都是数值传递操作（读/写）。即使操作数增加，程序也不会在更多的锁下运行。因此，如果想要达到我们之前的目的，我们可以这么改写程序：

```
1   for i := 1; i <= 9; i++ {
2       select {
3       case msg := <-c1:
4           println(msg)
5       case msg := <-c2:
6           println(msg)
7       case msg := <-c3:
8           println(msg)
9       }
10  }
```

翻译的不错

cmv00cmy
顶 哦!3年前

1人顶

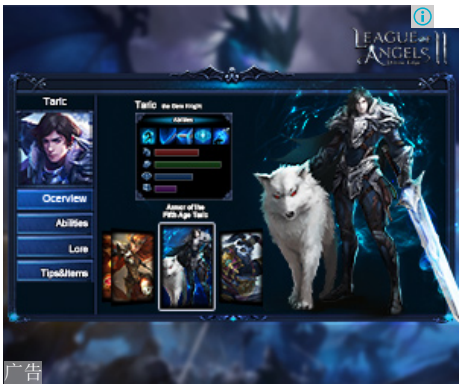
注意循环中的9这个数：每个channel存在三个写操作，这就是为什么这里需要9次循环的原因。在一般的守护进程中，我们可以使用无限循环执行选择操作，但如果我在这里那么做了，那我们将得到一个死锁：

```
first 1
second 1
third 1 // 这个channel将不会等待其他channel
third 2
third 3
second 2
first 2
second 3
first 3
```

总结.

channel是Go语言中颇为有趣的一个机制。但是在高效地使用它们之前你必须搞清楚他们是如何工作的。我试图在本文中对channel做出最基础的解释，如果你想要更深入地学习这个机制，我建议你阅读以下文章：

- 并发不是并行 - Rob Pike 之前我们有提到过
- Go语言并发模式——初级篇
- Go语言并发模式——高级篇



翻译的不错

cmv00cmy
顶 哦!8年前

1人顶

网友评论 共6条 发表评论 回页面顶部

- 高达 发表于 2013-12-21 10:36 回复
  不错
- 壬癸甲乙 发表于 2013-12-21 12:17 回复
  可以看
  看，有些点记一下。
- xym 发表于 2013-12-21 13:36 回复
  很好
- yearnfar 发表于 2013-12-23 00:49 回复
  内容很
  多，mark下
- max佩恩 发表于 2014-04-09 17:51 回复
  写得真
  好，有几个小错字，如：
  >使之在一此调用中向channel发送三条信息

  应该是
  使之在一次调用中向channel发送三条信息