

Go 并发

Go 语言支持并发，我们只需要通过 go 关键字来开启 goroutine 即可。

goroutine 是轻量级线程，goroutine 的调度是由 Golang 运行时进行管理的。

goroutine 语法格式：

```
go 函数名( 参数列表 )
```

例如：

```
go f(x, y, z)
```

开启一个新的 goroutine:

```
f(x, y, z)
```

Go 允许使用 go 语句开启一个新的运行期线程，即 goroutine，以一个不同的、新创建的 goroutine 来执行一个函数。同一个程序中的所有 goroutine 共享同一个地址空间。

实例

```
package main

import (
    "fmt"
    "time"
)

func say(s string) {
    for i := 0; i < 5; i++ {
        time.Sleep(100 * time.Millisecond)
        fmt.Println(s)
    }
}

func main() {
    go say("world")
    say("hello")
}
```

执行以上代码，你会看到输出的 hello 和 world 是没有固定先后顺序。因为它们是两个 goroutine 在执行：

```
world
hello
```

```
hello
world
world
hello
hello
world
world
hello
```

通道 (channel)

通道 (channel) 是用来传递数据的一个数据结构。

通道可用于两个 goroutine 之间通过传递一个指定类型的值来同步运行和通讯。操作符 <- 用于指定通道的方向，发送或接收。如果未指定方向，则为双向通道。

```
ch <- v    // 把 v 发送到通道 ch
v := <-ch  // 从 ch 接收数据
           // 并把值赋给 v
```

声明一个通道很简单，我们使用chan关键字即可，通道在使用前必须先创建：

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

注意：默认情况下，通道是不带缓冲区的。发送端发送数据，同时必须又接收端相应的接收数据。

以下实例通过两个 goroutine 来计算数字之和，在 goroutine 完成计算后，它会计算两个结果的和：

实例

```
package main

import "fmt"

func sum(s []int, c chan int) {
    sum := 0
    for _, v := range s {
        sum += v
    }
    c <- sum // 把 sum 发送到通道 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 // 从通道 c 中接收
```

```
    fmt.Println(x, y, x+y)
}
```

输出结果为：

```
-5 17 12
```

通道缓冲区

通道可以设置缓冲区，通过 make 的第二个参数指定缓冲区大小：

```
ch := make(chan int, 100)
```

带缓冲区的通道允许发送端的数据发送和接收端的数据获取处于异步状态，就是说发送端发送的数据可以放在缓冲区里面，可以等待接收端去获取数据，而不是立刻需要接收端去获取数据。

不过由于缓冲区的大小是有限的，所以还是必须有接收端来接收数据的，否则缓冲区一满，数据发送端就无法再发送数据了。

注意：如果通道不带缓冲，发送方会阻塞直到接收方从通道中接收了值。如果通道带缓冲，发送方则会阻塞直到发送的值被拷贝到缓冲区内；如果缓冲区已满，则意味着需要等待直到某个接收方获取到一个值。接收方在有值可以接收之前会一直阻塞。

实例

```
package main

import "fmt"

func main() {
    // 这里我们定义了一个可以存储整数类型的带缓冲通道
    // 缓冲区大小为2
    ch := make(chan int, 2)

    // 因为 ch 是带缓冲的通道，我们可以同时发送两个数据
    // 而不用立刻需要去同步读取数据
    ch <- 1
    ch <- 2

    // 获取这两个数据
    fmt.Println(<-ch)
    fmt.Println(<-ch)
}
```

执行输出结果为：

```
1
2
```

Go 遍历通道与关闭通道

Go 通过 range 关键字来实现遍历读取到的数据，类似于与数组或切片。格式如下：

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

如果通道接收不到数据后 ok 就为 false，这时通道就可以使用 `close()` 函数来关闭。

实例

```
package main


import (
    "fmt"
)

func fibonacci(n int, c chan int) {
    x, y := 0, 1
    for i := 0; i < n; i++ {
        c <- x
        x, y = y, x+y
    }
    close(c)
}


func main() {
    c := make(chan int, 10)
    go fibonacci(cap(c), c)
    // range 函数遍历每个从通道接收到的数据，因为 c 在发送完 10 个
    // 数据之后就关闭了通道，所以这里我们 range 函数在接收到 10 个数据
    // 之后就结束了。如果上面的 c 通道不关闭，那么 range 函数就不
    // 会结束，从而在接收第 11 个数据的时候就阻塞了。
    for i := range c {
        fmt.Println(i)
    }
}
```

执行输出结果为：

```
0
1
1
2
3
5
8
13
21
34
```



3 篇笔记

 写笔记