【极简 Golang 入门】并发编程

作者：Allenxuxu
原文链接：https://ld246.com/article/1566219700322
来源网站：链滴

并发编程

并发与并行

并发与并行是不同的。一个并发程序可以在一个单核处理器使用多个线程来执行多个任务，就好像这任务同时执行一样。但是同一时间点只有一个任务在执行，是操作系统内核在调度不同的线程交叉执使得它们好像在同时执行一样。而并行是指在同一时间点程序同时执行多个任务，是物理上真正的同执行，而非看着像。

并行是一种利用多处理器提高运行速度的能力。所以并发程序可以是并行的，设计优秀的并发程序运在多核或者多处理器上也可以实现并行。

多线程程序可以编写出高并发应用，重复利用多核处理器性能，但是编写多线程程序非常容易出错，主要的问题是内存中的数据共享。多线程程序在多核处理器上的并行执行和操作系统对线程调度的随性，导致这多个线程中共享的数据会以无法预知的方式进行操作。

传统解决方案是同步不同的线程，即对数据加锁。这样在同一时间点就只有一个线程可以变更数据，是这使得原来可以在多核处理器上并行执行的程序串行化了，无法重复利用多核处理器的能力。

Go 提供的并发编程特性

Go 语言原生支持程序的并发执行。Go 语言提供 协程 (goroutine) 与通道 (channel) 来支持并发编。

Go 的协程和其他语言中的协程是不太一样。Go 的协程意味着并行，或是可以并行，而其他语言的协一般来说是单线程串形化执行的，需要程序主动让出当前CPU。

协程 goroutine

Go 的协程和操作系统线程不是一对一的关系，一个协程对应于一个或多个线程，映射（多路复用，行于）在它们之上。也就是说一个协程可能会在多个操作系统线程上都运行过，同一个操作系统线程运行多个 Go 协程，Go 语言的协程调度器负责完成调度。

操作系统线程上的协程时间片让我们可以使用少量的操作系统线程就能运行任意多个协程，而且 Go 行时可以聪明的意识到哪些协程被阻塞了，暂时搁置它们并处理其他协程。比如，当系统调用（比如待 I/O）阻塞协程时，当前协程会被挂起，其他协程会继续在其他线程上工作，当 I/O 事件到来，挂的协程会自动恢复执行。

Go 每个协程创建时占用4k栈内存，协程的栈会根据需要进行伸缩，不出现栈溢出，开发者不需要关栈的大小。当协程结束的时候，它会静默退出，用来启动这个协程的函数不会得到任何的返回值。

```
package main

import (
    "fmt"
    "time"
)

func GoRun(i int) int {
    fmt.Println("go ", i)
    return i
```

```
}

func main() {
    fmt.Println("Hello World")

    go func() {
        fmt.Println("go")
    }()

    go func(i int) {
        fmt.Println("go ", i)
    }(1)

    go GoRun(2)

    time.Sleep(1*time.Second)
}
```

输出：

```
Hello World
go  2
go
go  1
```

这个输出结果的顺序并不是固定的，因为 go 关键字启动的协程都是并发执行的。

Go 程序 main() 函数也可以看做是一个协程，尽管它并没有通过 go 来启动。如果 main() 函数退出，其他协程也会随之退出，这就是为什么上面的代码要在最后加上 time.Sleep(1*time.Second)。

在一个协程中，如果需要进行非常密集的运算，可以在运算循环中周期的使用 runtime.Gosched()。会让出处理器，允许运行其他协程；它并不会使当前协程挂起，所以它会自动恢复执行。使用 Gosched() 可以使计算均匀分布，使通信不至于迟迟得不到响应。

通道 channel

协程间可以使用共享内存来实现通信，Go 提供 sync 包来实现协程同步，不过 Go 中还提供一种更优的方式：使用 channels 来同步协程。

通道就像一个可以用于发送类型化数据的管道，Go 保障在任何给定时间内，通道内的一个数据只有个协程可以对其访问，所以不会发生数据竞争。也就是说，Go 语言保障通道的发送和接受的原子性。

```
package main

import "fmt"

func main() {
    var ch chan int
    fmt.Println(ch)    // <nil>

    ch = make(chan int, 1)
    fmt.Println(ch, len(ch), cap(ch)) // 0xc00008c000 0 1
}
```

通道是引用类型，未初始化的通道的值是nil，使用 make 分配内存 ch := make(chan int)。

通道只能传输一种类型的数据，比如 chan int 或者 chan string，所有的类型都可以用于通道，空接口 interface{} 也可以。通道在 Go 中同样是一等公民，可以存储在变量中，作为函数的参数传递，作为数返回值，甚至可以通过通道发送它们自身。

通道使用 <- 符号来发送或是接受数据，信息按照箭头的方向流动。

ch <- int1 表示用通道 ch 发送变量 int1。

int2 := <- ch 表示变量 int2 从通道 ch接收数据。如果 int2 已经声明过，则应该写成 int2 = <- ch 。

<- ch 表示获取通道的一个值，并且丢弃之，

```
package main

import (
    "fmt"
    "time"
)

func sendData(ch chan int) {
    ch <- 1
    ch <- 2
    ch <- 3
    ch <- 4
}

func getData(ch chan int) {
    var input int
    for {
        input = <-ch
        fmt.Println(input)
    }
}

func main() {
    ch := make(chan int)

    go sendData(ch)
    go getData(ch)

    time.Sleep(1*time.Second)
}
```

输出：

```
1
2
3
4
```

通道是可以带缓冲的，ch := make(chan int, 5) 即通道里可以容纳 5 个 int 类型的值。ch := make(chan int) 默认是没有缓冲区的，即容量大小为1 。当通道数据满时，往通道中发送操作会阻塞，直到通中有空闲的空间。当通知中没有数据时，从通道中接受数据的操作会被阻塞，直到通道缓冲区中有数

。

将上面的例子稍作修改：

```
package main

import (
    "fmt"
    "time"
)

func sendData(ch chan int) {
    fmt.Println("sendData")
    ch <- 1
    fmt.Println("ch <- 1")
    ch <- 2
    fmt.Println("ch <- 2")
    ch <- 3
    fmt.Println("ch <- 3")
    ch <- 4
    fmt.Println("ch <- 4")
}

func main() {
    ch := make(chan int)

    go sendData(ch)

    time.Sleep(1 * time.Second)
}
```

输出：

```
sendData
```

因为没有接收通道 ch 数据，所以协程 sendData 一直阻塞在 ch <- 1，直到 main 函数 time.Sleep 束后程序退出。

将通道设为有缓冲区的，设置容量为2: ch := make(chan int, 2), 重新执行，输出如下：

```
sendData
ch <- 1
ch <- 2
```

下面验证一下接收数据阻塞的情况

```
package main

import (
    "fmt"
    "time"
)

func getData(ch chan int) {
    var input int
```

```
    for {
        fmt.Println("getData")
        input = <-ch
        fmt.Println(input)
    }
}

func main() {
    ch := make(chan int, 2)

    go getData(ch)

    time.Sleep(1 * time.Second)
}
```

输出：

```
getData
```

程序启动了一个协程来接收通道 ch 中的数据，但是没有操作来往通道中发送数据，所以协程 getData 一直阻塞在 input = <-ch，直到程序退出。

通道创建的时候都是双向的，但是通道类型可以用注解来表示它只发送或者只接收，从而来限制协程通道的操作。

```
package main

import (
    "fmt"
    "time"
)

func sendData(ch chan<- int) {
    ch <- 1
    ch <- 2
    ch <- 3
    ch <- 4
}

func getData(ch <-chan int) {
    var input int
    for {
        input = <-ch
        fmt.Println(input)
    }
}

func main() {
    ch := make(chan int)

    go sendData(ch)
    go getData(ch)

    time.Sleep(1 * time.Second)
}
```

通道可以通过 close 显式关闭，如果通道类型被注解，只有发送类型的通道可以被关闭。对已经 close 过的通过再次 close 会导致运行时的 panic 。读取已经关闭的通道，会立即返回通道数据类型的零值。

```
package main

import (
    "fmt"
    "time"
)

func sendData(ch chan<- int) {
    ch <- 1
    ch <- 2
    ch <- 3
    ch <- 4
    close(ch)
}

func getData(ch <-chan int) {
    var input int
    for {
        input = <-ch
        fmt.Println(input)
    }
}

func main() {
    ch := make(chan int)

    go sendData(ch)
    go getData(ch)

    time.Sleep(1 * time.Second)
}
```

输出：

```
1
2
3
4
0
0
...
```

上面的输出，会继续一直打印 0 ， 直到程序退出。

Go 提供方法来检测通道是否已经关闭：

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

当通道已经关闭的时候，ok 为 false；通道打开时，ok 为 true 。

还可以使用 for-range 来读取通道，这会自动检测通道是否关闭。

```
package main

import (
    "fmt"
    "time"
)

func sendData(ch chan<- int) {
    ch <- 1
    ch <- 2
    ch <- 3
    ch <- 4
    close(ch)
}

func getData(ch <-chan int) {
    var input int

    for input = range ch {
        fmt.Println(input)
    }

    fmt.Println("getData exit")
}

func main() {
    ch := make(chan int)

    go sendData(ch)
    go getData(ch)

    time.Sleep(1 * time.Second)
}
```

输出：

```
1
2
3
4
getData exit
```

从上面的例子可以看出，当通道被关闭时， for-range 循环会自动跳出，结束循环。

现实的开发中，会运行很多的协程，可能需要从多个通道中接收或者发送数据，Go 可以使用 select 键字来处理多个通道的问题。

select 监听进入通道的数据，如果所有的通道的都没有数据则会一直阻塞，直到有一个通道有数据；果有多个可以处理，select 会随机选择一个处理；特别需要注意的是，如果所有的通道都没有数据，且写了 default 语句，则会执行 default 。

```
package main
```

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

func sendData1(ch chan<- int) {
    ch <- 1
    ch <- 2
    ch <- 3
    ch <- 4
    // close(ch)
}

func sendData2(ch chan<- string) {
    ch <- "a"
    ch <- "b"
    ch <- "c"
    ch <- "d"
    // close(ch)
}

func getData(ch1 <-chan int, ch2 <-chan string) {
    for {
        select {
        case v := <-ch1:
            fmt.Println(v)
        case v := <-ch2:
            fmt.Println(v)
        // default:
        //     fmt.Println("default")
        }
    }
}

func main() {
    ch1 := make(chan int)
    ch2 := make(chan string)

    go sendData1(ch1)
    go sendData2(ch2)
    go getData(ch1, ch2)

    time.Sleep(1 * time.Second)
}
```

输出：

```
1
2
a
b
3
c
4
```

d

如果将上面注释掉的 default 语句处的代码打开，则在正确接收所有通道的所有数据后会一直打印 default ，直到程序退出。

select 不会自动处理通道关闭的情况，如果将代码中关于 close 的代码注释打开，select 正确接收所通道的所有数据后会只一直打印 0 和 "" (int 和 string 的零值)。case v,ok := <-ch1: 可以判断通道开关情况。