链滴

Golang 入门笔记 -12- 并发

作者：zyk

原文链接：https://ld246.com/article/1611287209320

来源网站：链滴

基本概念

线程和进程

- **进程**： 是程序在操作系统中的一次执行过程，系统进行资源分配和调度的一个**独立单位** 。
- **线程**： 是进程的一个执行实体，是 CPU 调度和分派的**基本单位** ，它是比进程更小的能独立运行的**本单位** 。

一个进程可以创建或撤销多个线程；同一个进程中的多个线程之间可以并发执行。

线程和协程

- **协程**： 独立栈空间，共享堆空间，调度由用户自己控制，本质上有点类似于用户级线程，这些用户线程的调度也是自己实现的。
- **线程**： 一个线程上可以跑多个协程，协程是轻量级的线程。

并发和并行

并发 和 **并行** 虽然相似，但是有所不同：

- **并发**： 逻辑上具备同时处理多个任务的能力。
- **并行**： 物理上在同一时刻执行多个并发任务。

我们一般会说程序是并发的，意思是说它允许多个任务同时进行，但不一定在同一时刻发生。对于单处理器，它们能以间隔方式切换运行；而并行则依赖多核心处理器，是让多个任务真正能在同一时刻行，是一种程序运行的状态。

多线程和多进程是并行的基本条件，但单线程也可用协程（coroutine）做到并发。虽然单线程只能过主动切换来实现并发，但也有优点，可以避免由于切换线程导致的资源消耗。协程上运行多个任务质上是串行的，且可控，无需做同步处理。

采用多进程也不一定能并行，Python 由于 GIL 的限制，默认只能并发而不能并行，转而使用“**多进 + 协程**”来实现并行。

多进程、多线程和协程各有利弊，一般用多进程来实现分布式和负载均衡，减轻单进程垃圾回收压力用多线程抢夺更多的处理器资源；用协程来提高处理器时间片利用率。

goroutine

在 Java 中要实现并发编程，需要自己维护一个线程池，处理线程调度和上下文切换，非常麻烦。Go 言的 goroutine 很好地解决了这一问题，goroutine 类似于线程，但 goroutine 由 Go 的运行时（ru time）来调度和管理，Go 程序会智能地将 goroutine 中的任务合理分配给每个 CPU。

在 Go 语言中使用 goroutine 很简单，只要在调用函数前加一个 go 关键字即可。

一个 goroutine 必须对应一个函数，可以创建多个 goroutine 去执行相同函数。

启动单个 goroutine

启动单个 Goroutine 只需在调用函数时加一个 go 关键字即可，例如：

```
package main

import "fmt"

func hello() {
    fmt.Println("hello goroutine!")
}

func main() {
    go hello()
    fmt.Println("main goroutine done!")
}
```

上述代码的运行结果为：

```
main goroutine done!
```

运行的结果是不是有点出乎意料，只打印了 main goroutine done!，因为在程序运行时，main 函默认创建了一个 goroutine，在 main 函数返回时由 main 函数创建的 goroutine 结束了，而在 main 函数中用 go 关键字创建的 goroutine 需要一定时间来生成，因此也一并结束了（main 函数默认创的 goroutine 就是老大哥，在 main 函数内部用 go 关键字创建的其他 goroutine 是小弟，只要老哥一阵亡，小弟全部阵亡）。

如果我们想让 main 执行之后，hello 函数也能执行，可以用 time.Sleep 来实现：

```
package main

import (
    "fmt"
    "time"
```

```
)

func hello() {
    fmt.Println("hello goroutine!")
}

func main() {
    go hello()
    fmt.Println("main goroutine done!")
    time.Sleep(time.Second)
}
```

上述代码运行结果为：

创建新的 goroutine 需要一定时间，而此时 main 函数中的 goroutine 是继续执行的。

```
main goroutine done!
hello goroutine!
```

启动多个 goroutine

启动多个 goroutine 只需使用多个 go 关键字：

使用 sync.WaitGroup 来实现 goroutine 的同步

```
var wg sync.WaitGroup

func hello(i int) {
    defer wg.Done() // goroutine 结束就登记 -1
    fmt.Println("Hello Goroutine!", i)
}

func main() {
    for i := 0; i < 10; i++ {
        wg.Add(1) // 启动一个 goroutine 就登记 +1
        go hello(i)
    }
    wg.Wait() // 等待所有登记的 goroutine 都结束
}
```

上述代码运行结果为：

```
Hello Goroutine! 1
Hello Goroutine! 2
Hello Goroutine! 0
Hello Goroutine! 9
Hello Goroutine! 4
Hello Goroutine! 5
Hello Goroutine! 6
Hello Goroutine! 7
Hello Goroutine! 8
```

多运行几次会发现每次打印的数字顺序都不一样，这是因为这 10 个 goroutine 是并发执行的，而 goroutine 的调度是 **随机** 的。

goroutine 与线程

可增长的栈

OS 线程（操作系统线程）一般都有固定的栈内存（通常为 2MB）,一个 goroutine 的栈在其生命周开始时只有很小的栈（典型情况下 2KB），goroutine 的栈不是固定的，他可以按需增大和缩小，goutine 的栈大小限制可以达到 1GB，虽然极少会用到这么大。所以在 Go 语言中一次创建十万左右的oroutine 也是可以的。

goroutine 调度

GPM 是 Go 语言运行时（runtime）层面的实现，是 go 语言自己实现的一套调度系统。区别于操作统调度 OS 线程。

- G 很好理解，就是个 goroutine 的，里面除了存放本 goroutine 信息外还有与所在 P 的绑定等信。
- P 管理着一组 goroutine 队列，P 里面会存储当前 goroutine 运行的上下文环境（函数指针，堆地址及地址边界），P 会对自己管理的 goroutine 队列做一些调度（比如把占用 CPU 时间较长的 goutine 暂停、运行后续的 goroutine 等等）当自己的队列消费完了就去全局队列里取，如果全局队里也消费完了会去其他 P 的队列里抢任务。
- M（machine） 是 Go 运行时（runtime）对操作系统内核线程的虚拟， M 与内核线程一般是一映射的关系， 一个 groutine 最终是要放到 M 上执行的；

P 与 M 一般也是一一对应的。他们关系是： P 管理着一组 G 挂载在 M 上运行。当一个 G 长久阻塞一个 M 上时，runtime 会新建一个 M，阻塞 G 所在的 P 会把其他的 G 挂载在新建的 M 上。当旧的 阻塞完成或者认为其已经死掉时 回收旧的 M。

P 的个数是通过 runtime.GOMAXPROCS 设定（最大 256），Go1.5 版本之后默认为物理线程数。并发量大的时候会增加一些 P 和 M，但不会太多，切换太频繁的话得不偿失。

单从线程调度讲，Go 语言相比起其他语言的优势在于 OS 线程是由 OS 内核来调度的，goroutine 是由 Go 运行时（runtime）自己的调度器调度的，这个调度器使用一个称为 m:n 调度的技术（复用调度 m 个 goroutine 到 n 个 OS 线程）。 其一大特点是 goroutine 的调度是在用户态下完成的，涉及内核态与用户态之间的频繁切换，包括内存的分配与释放，都是在用户态维护着一块大的内存池，不直接调用系统的 malloc 函数（除非内存池需要改变），成本比调度 OS 线程低很多。 另一方面充利用了多核的硬件资源，近似的把若干 goroutine 均分在物理线程上， 再加上本身 goroutine 的超量，以上种种保证了 go 调度方面的性能。

GOMAXPROCS

GOMAXPROCS 是一个参数，它可以确定 Go 调度器同时执行 Go 代码的 OS 线程个数，默认值是器的 CPU 核心数。例如在一个 8 核心的机器上，调度器会把 Go 代码同时调度到 8 个 OS 线程上（OMAXPROCS 是 m:n 调度中的 n）。

可以通过 runtime.GOMAXPROCS() 函数设置当前程序并发时占用的 CPU 逻辑核心数。

Go1.5 版本之前，默认使用的是单核心执行。Go1.5 版本之后，默认使用全部的 CPU 逻辑核心数。

我们可以通过将任务分配到不同的 CPU 逻辑核心上实现并行的效果，例如：

```
func a() {
```

```
    for i := 1; i < 10; i++ {
        fmt.Println("A:", i)
    }
}

func b() {
    for i := 1; i < 10; i++ {
        fmt.Println("B:", i)
    }
}

func main() {
    runtime.GOMAXPROCS(1)
    go a()
    go b()
    time.Sleep(time.Second)
}
```

两个任务只有一个逻辑核心，此时是做完一个任务再做另一个任务。 将逻辑核心数设为 2，此时两个务并行执行，代码如下。

```
func a() {
    for i := 1; i < 10; i++ {
        fmt.Println("A:", i)
    }
}

func b() {
    for i := 1; i < 10; i++ {
        fmt.Println("B:", i)
    }
}

func main() {
    runtime.GOMAXPROCS(2)
    go a()
    go b()
    time.Sleep(time.Second)
}
```

Go 语言中的操作系统线程和 goroutine 的关系：

1. 一个操作系统线程对应用户态多个 goroutine。
2. go 程序可以同时使用多个操作系统线程。
3. goroutine 和 OS 线程是多对多的关系，即 m:n。

channel

单纯编写并发函数并没有太大意义，函数之间的数据交换才是并发执行函数的关键。

Go 语言的并发模型是 CSP(Communicating Sequential Processes)，提倡 **通过通信共享内存** 而不是 **过内存共享通信** 。

如果说 goroutine 是 Go 程序并发的 **执行体** ，那么 channel 就是他们之间的 **连接** 。channel 是可让一个 goroutine 发送特定值到另一个 goroutine 的特定机制。

Go 语言中，通道（channel）是一种特殊的数据类型。通道是队列，遵循 **先进先出** 的原则，以保证发数据的顺序。每一个通道都是具体类型的载体，在声明通道（channel）的时候需要指定元素类型类似于数组和切片）。

channel 类型

channel 是一种数据类型，属于引用类型，声明格式如下：

```
var variable chan type
```

例如：

```
var ch1 chan int   // 声明一个传递整型的通道
var ch2 chan bool  // 声明一个传递布尔型的通道
var ch3 chan []int // 声明一个传递 int 切片的通道
```

创建 channel

通道是引用类型，因此空值是 nil：

```
var ch chan int
fmt.Println(ch) // <nil>
```

声明的通道需要使用 make 函数初始化后才能使用，格式如下：

缓冲区大小是可选的。

```
make(chan 元素类型, [缓冲大小])
```

举几个例子：

```
ch1 := make(chan int)
ch2 := make(chan bool)
ch3 := make(chan []int)
```

channel 操作

通道有发送（send），接收（receive）和关闭（close）三种操作。

发送和接收都使用 <- 符号。

先定义一个通道：

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

发送

将一个值发送到通道：

```
ch <- 10 // 把 10 发送到 ch 中
```

接收

从通道中接收值：

```
x := <- ch // 从 ch 中接收值，并赋值给 x
<- ch      // 从 ch 接收值，忽略结果
```

关闭

通过调用 close 函数来关闭通道：

```
close(ch)
```

只有在通知接收方 goroutine 所有数据都发送完毕时才需要关闭通道，因为通道可被垃圾回收器回收
这与关闭文件不同，在结束操作之后必须关闭文件，但关闭通道非必须。

注意：

- 对一个关闭的通道再发送值会引发 panic；
- 对一个关闭的通道继续进行接收会一直获取值直到通道为空；
- 对一个关闭且无值的通道执行接收操作得到的是对应的零值；
- 关闭一个已关闭的通道会引发 panic。

无缓冲的通道

无缓冲通道也称阻塞通道，我们来看一段代码：

```
func main() {
    ch := make(chan int)
    ch <- 10
    fmt.Println("发送成功")
}
```

上述代码能够成功编译，但是运行时会报错：

```
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send]:
main.main()
    D:/projects/go_projects/goBasic/routine/single_goroutine.go:9 +0x5f

Process finished with exit code 2
```

为什么会出现上述这种死锁（deadlock）情况呢？因为我们使用 ch := make(chan int) 创建的是无冲通道，无缓冲通道只有在有接收者情况下才能发送值。

如何才能解决该问题呢？

一种方法是启用一个 goroutine 去接收值，例如：

```
func recv(c chan int) {
    ret := <-c
```

```
    fmt.Println("接收成功", ret)
}

func main() {
    ch := make(chan int)
    go recv(ch) // 启用 goroutine 从通道接收值
    ch <- 10
    fmt.Println("发送成功")
}
```

无缓冲通道上的发送操作会阻塞，直到另一个 goroutine 在该通道上执行接收操作，此时才能发送成，两个 goroutine 将继续执行。相反，如果接收操作先执行，接收方的 goroutine 将阻塞，直到另个 goroutine 在该通道上发送一个值。

使用无缓冲通道进行通信将导致发送和接收的 goroutine 同步化，因此， **无缓冲通道** 也被称为 **同步道** 。

有缓冲的通道

解决上述问题还有一个办法，就是使用有缓冲的通道，可以在使用 make() 函数初始化通道的时候指通道的容量，如下：

```
func main() {
    ch := make(chan int, 1) // 创建一个容量为 1 的有缓冲区通道
    ch <- 10
    fmt.Println("发送成功")
}
```

只要通道的容量 **大于 0**，那么该通道就是有缓冲的通道，通道容量表示通道能存放元素的数量。

我们可以通过 len 函数获取通道内元素数量，通过 cap 函数获取通道的容量。

for range 从通道循环取值

我们可以通过 close() 函数来关闭通道，那么如何判断一个通道已经被关闭了呢？（对于已关闭的通，我们不能再关闭，不然会引发 panic）。

我们来看一个例子：

```
package main

import (
    "fmt"
)

// channel 练习
func main() {
    ch1 := make(chan int)
    ch2 := make(chan int)
    // 开启 goroutine 将 1~10 的数发送到 ch1 中
    go func() {
        for i := 1; i <= 10; i++ {
            ch1 <- i
```

```
        }
        close(ch1)
    }()
    // 开启 goroutine 从 ch1 中接收值，并将该值的平方发送到 ch2 中
    go func() {
        for {
        i, ok := <-ch1 // 通道关闭后再取值 ok=false
        if !ok {
            break
        }
        ch2 <- i * i
        }
        close(ch2)
    }()
    // 在主 goroutine 中从 ch2 中接收值打印
    for i := range ch2 { // 通道关闭后会退出 for range 循环
        fmt.Println(i)
    }
}
```

单向通道

有时我们会将通道作为参数在函数之间传递，而我们需要对通道进行限制，比如限制通道只能发送或只能接收。Go 语言中可以通过 **单向通道** 来解决该问题，例如：

```
func counter(out chan<- int) {
    for i := 1; i <= 10; i++ {
        out <- i
    }
    close(out)
}

func squarer(out chan<- int, in <-chan int) {
    for i := range in {
        out <- i * i
    }
    close(out)
}

func printer(in <-chan int) {
    for i := range in {
        fmt.Println(i)
    }
}

func main() {
    ch1 := make(chan int)
    ch2 := make(chan int)
    go counter(ch1)
    go squarer(ch2, ch1)
    printer(ch2)
}
```

注意

- chan<- int 是一个 **只写单向通道** （只能对其写入 int 类型值），可对其执行发送操作但是不能执接收操作；
- <-chan int 是一个 **只读单向通道** （只能从其读取 int 类型值），可对其执行接收操作但是不能执发送操作。
- 在函数传参及任何赋值操作中可以将 **双向通道** 转换为 **单向通道**，反之则不行。

并发安全和锁

多个 goroutine 同时操作一个资源时，会导致数据竞争。

我们来看一个例子：

```
var x int64
var wg sync.WaitGroup

func add() {
    for i := 0; i < 5000; i++ {
        x = x + 1
    }
    wg.Done()
}

func main() {
    wg.Add(2)
    go add()
    go add()
    wg.Wait()
    fmt.Println(x)
}
```

上述代码中，我们开启了两个 goroutine 去累加变量 x 的值，这两个 goroutine 在访问和修改 x 的时会存在数据竞争，导致最后的结果与预期不符合。

互斥锁

互斥锁可以保证在同一时刻，只有一个 goroutine 能访问共同资源。Go 语言中可以使用 sync 包的 utex 来实现互斥锁，例如：

```
var x int64
var wg sync.WaitGroup
var lock sync.Mutex

func add() {
    for i := 0; i < 5000; i++ {
    lock.Lock() // 加锁
    x = x + 1
    lock.Unlock() // 解锁
    }
    wg.Done()
}
func main() {
    wg.Add(2)
```

```
    go add()
    go add()
    wg.Wait()
    fmt.Println(x)
}
```

使用互斥锁能保证同一时刻，只有一个 goroutine 进入临界区，其他 goroutine 在等待锁；当互斥释放后，等待的 goroutine 才能获取锁并进入临界区，而多个 goroutine 等待一个锁的唤醒策略是机的。

读写互斥锁

读写锁分为两种，**读锁**和**写锁**。当一个 goroutine 获取**读锁**后，其他的 goroutine 可以继续获取**读锁**但无法获取**写锁**（需要进行等待）；当一个 goroutine 获取**写锁**后，其他的 goroutine 既不能获取**锁**也不能获取**写锁**。

```
var (
    x      int64
    wg     sync.WaitGroup
    rwlock sync.RWMutex
)

func write() {go
    // lock.Lock() // 加互斥锁
    rwlock.Lock() // 加写锁
    x = x + 1
    time.Sleep(10 * time.Millisecond) // 假设读操作耗时10毫秒
    rwlock.Unlock()                   // 解写锁
    // lock.Unlock()                  // 解互斥锁
    wg.Done()
}

func read() {
    // lock.Lock()               // 加互斥锁
    rwlock.RLock()               // 加读锁
    time.Sleep(time.Millisecond) // 假设读操作耗时1毫秒
    rwlock.RUnlock()             // 解读锁
    // lock.Unlock()             // 解互斥锁
    wg.Done()
}

func main() {
    start := time.Now()

    for i := 0; i < 10; i++ {
        wg.Add(1)
        go write()
    }

    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go read()
    }
```

```
    wg.Wait()
    end := time.Now()
    fmt.Println(end.Sub(start))
}
```

sync.WaitGroup

Go 语言中可以使用 sync.WaitGroup 来实现并发任务的同步，它有如下几个方法：

方法名	功能
(wg * WaitGroup) Add(delta int)	计数器 + delta
(wg *WaitGroup) Done()	计数器 - 1
(wg *WaitGroup) Wait()	阻塞直到计数器变为 0

sync.WaitGroup 内部维护着一个计数器，计数器的值可以发生变化。例如当我们启动了 N 个并发任时，就将计数器值增加 N。每个任务完成时通过调用 Done() 方法将计数器减 1。通过调用 Wait() 来待并发任务执行完，当计数器值为 0 时，表示已经完成所有并发任务。

sync.Once

在高并发的场景下，我们有时需要保证某些操作**只执行一次**，例如读取配置文件，连接数据库等。Go 语言中 sync 包中的 Once 可以帮我们来解决这个问题，sync.Once 只有一个 Do 方法：

```
func (o *Once) Do(f func()) {}
```

当有多个 goroutine 对一个变量（资源）进行**写操作**时，我们很难保证这个变量在**数据竞争**中能被正初始化，因此需要用**互斥锁**或 sync.Once 来保证该变量不被别的 goroutine 修改。

使用**互斥锁**会存在性能问题，因此面对这种情况，优先使用 sync.Once 来解决：

```
var icons map[string]image.Image

var loadIconsOnce sync.Once

// 对 icons 变量进行初始化
func loadIcons() {
    icons = map[string]image.Image{
        "left":  loadIcon("left.png"),
    "up":    loadIcon("up.png"),
    "right": loadIcon("right.png"),
    "down":  loadIcon("down.png"),
    }
}

// Icon 是并发安全的
func Icon(name string) image.Image {
    loadIconsOnce.Do(loadIcons) // 调用 Do 方法确保 loadIcons 函数只执行一次
    return icons[name]
}
```

sync.Map

Go 语言内置的 Map 并不是并发安全的，Go 语言为我们提供了内置的并发安全的 Map—— sync.Map，它无需初始化即可使用，内置了 Store、Load、LoadOrStore、Delete 和 Range 等方法：

```
var m = sync.Map{}

func main() {
    wg := sync.WaitGroup{}
    for i := 0; i < 20; i++ {
        wg.Add(1)
        go func(n int) {
            key := strconv.Itoa(n)
            m.Store(key, n)
            value, _ := m.Load(key)
            fmt.Printf("k=:%v,v:=%v\n", key, value)
            wg.Done()
        }(i)
    }
    wg.Wait()
}
```