let's GoLang(三): Goroutine&Channel

标记

Richard
full stack, gopher, rustacean

这是Let's GoLang系列的第三篇文章：Goroutine&Channel，上一篇是面向对象。

背景知识

Go最为开发者喜爱的特性就是goroutine。我觉得原因有三个:

1. goroutine的高性能。相比传统的多线程模型有很大的提升，能够轻松的利用多核CPU；并且创建协程的开销也很小，能够轻易的创建大量的协程
2. goroutine的易用性。不可否认，goroutine对开发者而言是很友好的，学习成本极低；同时结合channel，能够很方便的处理互斥，同步等场景
3. 官方库的支持。Go的很多官方库都是深度集合goroutine的，包括网络和io的库，使得goroutine能够深入人心；对于各种第三方库的开发者而言，也能够毫无负担的使用goroutine来开发。

在介绍协程之前，我们应该先知道线程、进程以及并行、并发：

- 进程是指一段程序在一定的数据集合上的一次动态执行，进程是运行在自己内存地址空间独立执行体，即内存分配的基本单位
- 进程内的线程共享同一个内存地址空间，是cpu调度的基本单位
- 并发是指基于CPU调度算法，多个线程轮流使用CPU时间片来执行任务，由于CPU速度非常快，就像有多个任务在后台执行一样
- 并行通常是指多个线程(任务)，在多核(多处理器)上并行执行，这才是真正的有多个任务同时运行

前面介绍了一些跟CPU执行的基本概念，我们来首先了解为什么要有goroutine这个概念。通常内核线程与用户态线程的比例模型有1:1，1:N，以及M:N，接下来我们简单的介绍一下这三种模型的优缺点：

- 1:1 即一个用户态线程对应着一个内核线程，优点是能够利用多核的特性，缺点是线程切换的开销比较大，每次线程的切换都要经历用户态和内核态的转换
- 1:N 即多个用户态线程对应着一个内核线程，优点是线程切换的开销很小，缺点是无法利用CPU多核
- M:N 即Go中的goroutine采用的形式，结合1:1和1:N模型的优点，能够充分利用多核，切换的开销又很小。当然，为了实现M:N的模型，Go在背后做了很多的事情，比如线程调度器，同时，为了了解协程的底层实现，还需要去了解MPG这三个分别代表着什么含义

初探

我们先来看一个复杂的例子，感受goroutine的强大:

```
package main

import (
        "fmt"
        "time"
)

func serial()  {
        sum, count, start := 0, 0, time.Now()
        for count < 4e8 {
                sum += count
                count += 1
        }
        end := time.Now()
        fmt.Println(fmt.Sprintf("result is %d, time is %s", sum, end.Sub(start)))
}

func parallel()  {
        sum, count, ch, start := 0, 0, make(chan int, 4), time.Now()
        for count < 4 {
                go func(count int) {
                        value := 0
                        for i := count * 1e8; i < (count + 1) * 1e8; i++ {
                                value += i
                        }
                        ch <- value
                }(count)
                count++
        }
        for count > 0 {
                sum += <- ch
                count--
        }
        end := time.Now()
        fmt.Println(fmt.Sprintf("result is %d, time is %s", sum, end.Sub(start)))
}

func main()  {
        serial()
        parallel()
}
```

我们可以看到上面的serial是单线程(协程)版的，parallel是多线程(协程)版，下面是运行结果：

提升还是十分明显的。

goroutine入门

上面例子算是比较复杂的了，但是也不难，接下来我们看一个简单的例子，来了解goroutine：

```
package main

import (
        "fmt"
        "time"
)

func main()  {
        go func() {
                fmt.Println("this is running in background")
        }()
        time.Sleep(1e9)
}
```

这里例子非常简单，在Go里面使用“**go**”关键词就能创建协程。细心的你会发现，我们使用了"time.Sleep(1e9)"让主协程 sleep了1s，这是因为在go中，main函数也运行在协程中，如果主协程执行完成，其他的协程也自动退出了。如果不让主协程 sleep，那么我们创建的这个协程就来不及执行，随着主协程退出也跟着退出了。试试注释掉这一句，你会发现控制台不会打印任何东西。

channel、互斥与同步

上面我们看到的这个简单例子不太有用，为什么呢？ 因为通常而言，我们会借助协程去做一些计算，IO等操作；但是操作完成之后，还需要把**结果返回给主协程**。因此，这里会涉及到协程之间的通信，那就是channel(后面会使用**管道**代指channel)了。

还是举个简单的例子吧：

```
package main

import "fmt"

func main()  {
        ch := make(chan int)
        go func() {
                sum := 0
                for i := 0; i < 1e8; i++ {
                        sum += i
                }
                ch <- sum
        }()
        result := <- ch
        fmt.Println(result)
}
```

我们来简单分析下这段代码：

1. 通过make(chan int)来创建一个管道，这里还能传入第二个参数，作为管道的容量
2. 调用go func来创建一个协程，计算完成后，使用 "ch <- val "的形式将值写入管道中

3. 在主线程中，调用 "<- ch"，主协程将会阻塞，等待从管道中读取这个值，打印完成后，主协程退出

我们可以看到，这里的主协程并没有sleep，而是使用管道阻塞主协程，等待取出值再退出。这里简单的介绍channel是什么：

- *channel是类型化的消息队列，先进先出，用于值的交换、同步*
- *如果channel的容量大于0，这个管道是异步非阻塞的。在channel没有被消息值写满的时候，可以持续往里面写入值；如果channel不为空的时候，可以持续从channel中读取值。这就是非常经典的生产者消费者模型*
- *如果channel容量为0，那么这个管道是同步阻塞的，这个时候要进行通信，必须通信双方都准备好*

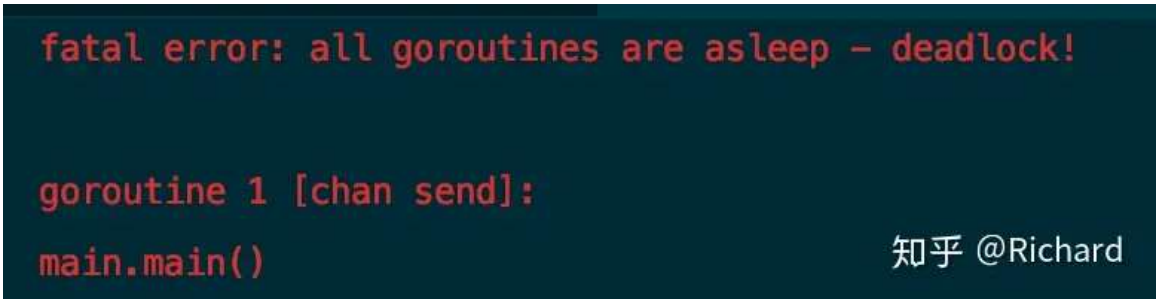
我们来看一个阻塞管道的例子：

```
package main

import "fmt"

func main()  {
        ch := make(chan int)
        ch <- 1
        val := <- ch
        fmt.Println(val)
}
```

执行上面的代码，你会看到这样的结果：



可以看到出现了死锁的情况，同时，go能够检测出大多数死锁的情况，开发体验很好。我们来简单的分析一下为什么会出现死锁，其实最开始我也是比较困惑的，但是仔细一想又是非常的简单：上面的ch <- 1 尝试着向管道中写入值，这个操作阻塞了主协程，导致后面的 val := <- ch无法执行。因此这里的情况就是，**生产者往管道中写入值，但是消费者还没准备好**，导致了死锁。那针对这种情况又该怎么解决了，大概有这两种：

第一种:

```
package main

import "fmt"

func main()  {
        ch := make(chan int, 1)
        ch <- 1
        val := <- ch
```

```
        fmt.Println(val)
}
```

第一种方法是让管道的容量大于0，还记得我们上面介绍管道时的第二点吗？如果管道的容量大于0，那么它就是异步非阻塞的，在管道容量没有满的时候，生产者可以持续的写入。因此，上面我们将管道的容量设为1，那么生产者就能正常地往管道中写入这个值，就不会阻塞主协程。

第二种：

上面的第一种方法是将管道的容量设置大于0；如果管道容量为0，那么只能让生产者和消费者都准备好再进行通信。所以这个时候我们该怎么办呢？那就需要借助前面提到的协程了，废话不多说，上代码：

```
package main

import "fmt"

func main()  {
        ch := make(chan int)
        go func() {
                ch <- 1
        }()
        val := <- ch
        fmt.Println(val)
}
```

这段代码仍然能够正常执行。首先，管道的容量依然为0；但是，通过go关键字来创建一个协程，这这个单独的协程中就可以做好准备，向管道中写入这个值，并且也不会阻塞主线程；在主线程中，消费者做好准备从管道中读取值；在某个时刻，生产者和消费者都准备好了，进行通信，这就不会导致死锁了。

channel的方向，close，for

通常情况下，我们那不需要手动去关闭关闭管道，但是也能使用close关键字来提前关闭，举个栗子：

```
package main

import "fmt"

func sender(ch chan<- int)  {
        for i := 0; i < 10; i++ {
                ch <- i
                if i == 5 {
                        close(ch)
                        break
                }
        }
}
func receiver(ch <-chan int)  {
        for val := range ch {
                fmt.Println(val)
```

```
        }
}

func main()  {
        ch := make(chan int)
        go sender(ch)
        receiver(ch)
}
```

我们来分析一下这段代码

1. sender函数中，持续地往管道中写入int类型的消息，并且在i为5的时候调用close手动关闭了管道，并且跳出了循环。这里需要注意的是，不能再向已经关闭的管道中写入值，因此如果没有上面的break，会触发panic
2. receiver函数中，使用for-range的形式从管道中读取值，在管道被关闭之后，会自动的结束循环
3. 同时，我们还注意到，sender函数的形参类型是 chan<- int，receiver函数的形参类型是 <-chan int，这代表着管道是单向的，分别只能向管道中写入消息、读取消息

channel&select

Go语言里面还有一个关键词，用于从选择channel，举个栗子：

```
package main

import (
        "fmt"
        "strconv"
        "time"
)

func channelOdd(ch chan<- int)  {
        for i := 0; ; i++ {
                if i % 2 == 1 {
                        ch <- i
                }
        }
}

func channelEven(ch chan<- int)  {
        for i := 0; ; i++ {
                if i % 2 == 0 {
                        ch <- i
                }
        }
}

func selectTwoChannel(oddCh <-chan int, evenCh <-chan int)  {
        for {
                select {
                case a := <-oddCh:
                        fmt.Println("odd ch " + strconv.Itoa(a))
                case b := <- evenCh:
                        fmt.Println("even ch " + strconv.Itoa(b))
```

```
                }
        }
}
```

```
func main() {
        chOne := make(chan int)
        chTwo := make(chan int)
        go channelOdd(chOne)
        go channelEven(chTwo)
        go selectTwoChannel(chOne, chTwo)
        time.Sleep(1e9)
}
```

select的作用是：

1. select会选择多个管道中的一个，如果都阻塞了，会等待其中一个可以处理
2. 如果多个管道可以处理，随机选择一个
3. 如果没有通道操作可以处理，并且包含default语句，则会执行default语句中的代码

具体到这里，我们来解释一下上面那段代码。channelOdd和channelEven函数会持续的往管道中写入消息，selectTwoChannel包含for循环和select语句，因为select语句执行一次就结束了，所以需要使用for的死循环来持续执行select，从两个管道中读取值并进行打印。select集合timer和ticker在执行定时和延时任务的时候非常有用。

ticker

我们先来看看time.NewTicker()，其返回的是一个time.Ticker的struct 实例，其中包含一个名为C的管道，这个管道每隔一段时间自动的写入一个消息，类似于JavaScript中的setInterval，我们使用time.NewTicker来模拟一下setInterval：

```
package main

import (
        "fmt"
        "reflect"
        "time"
)

func setInterval(callback interface{}, interval int)  {
        callbackType := reflect.TypeOf(callback)
        callbackValue:= reflect.ValueOf(callback)
        if callbackType.Kind() == reflect.Func {
                ticker := time.NewTicker(time.Duration(interval))
                for {
                        select {
                                case <- ticker.C:
                                        args := make([]reflect.Value, 0)
                                        callbackValue.Call(args)
```

```
                        }
                }
        }
}
func sayHelloWorld()  {
        fmt.Println("hello world")
}

func main()  {
        setInterval(sayHelloWorld, 1e9)
}
```

我们可以看到，使用timer.Ticker，可以非常轻松的模拟JavaScript中的setInterval，在上面这段代码中，还使用了反射 来判断setInterval的第一个参数是否为函数。同时，还能使用ticker.Stop()来停止计时器，就跟clearInterval一样。

还有一个函数是time.Tick()，它的函数签名是：Tick(d Duration) <- Time，跟time.NewTicker的区别是不用手动去关闭管道，非常方便。

After

上面使用它timer.newTicker()来模仿了JavaScript中的，下面我们使用timer.After来模拟一下JavaScript的setTimeout：

```
package main

import (
        "fmt"
        "reflect"
        "time"
)

func setTimeout(callback interface{}, interval int)  {
        callbackType := reflect.TypeOf(callback)
        callbackValue:= reflect.ValueOf(callback)
        if callbackType.Kind() == reflect.Func {
                ticker := time.After(time.Duration(interval))
                <- ticker
                args := make([]reflect.Value, 0)
                callbackValue.Call(args)
        }
}
func printHelloWorld()  {
        fmt.Println("hello world")
}

func main()  {
        setTimeout(printHelloWorld, 1e9)
}
```

上面的setTimeout跟JavaScript中的setTimeout一模一样，能够在固定的时间点执行一次，并且会自动关闭管道，time里面还有time.AfterFunc，可以直接执行延时任务。

上面我们提到的After和Ticker非常有用，可以来处理很超时，竞态的任务，感兴趣的小伙伴也可以去研究一下更多的用法。

发布于 2019-01-20 16:43

Go 语言

后端技术

编程语言