李文周的博客

JPG程序员/全栈开发 -- 专注互联网技术，相信代码改变世界。Go语言学习QQ群：645090316 公众号：李文周

Go语言基础之并发

2017年6月25日 | Golang |13031 阅读

并发是编程里面一个非常重要的概念，Go语言在语言层面天生支持并发，这也是Go语言流行的一个很重要的原因。

Go语言中的并发编程

并发与并行

并发：同一时间段内执行多个任务（你在用微信和两个女朋友聊天）。

并行：同一时刻执行多个任务（你和你朋友都在用微信和女朋友聊天）。

Go语言的并发通过 `goroutine` 实现。 `goroutine` 类似于线程，属于用户态的线程，我们可以根据需要创建成千上万个 `goroutine` 并发工作。 `goroutine` 是由Go语言的运行时（runtime）调度完成，而线程是由操作系统调度完成。

Go语言还提供 `channel` 在多个 `goroutine` 间进行通信。 `goroutine` 和 `channel` 是 Go 语言秉承的 CSP（Communicating Sequential Process）并发模式的重要实现基础。

goroutine

在java/c++中我们要实现并发编程的时候，我们通常需要自己维护一个线程池，并且需要自己去包装一个又一个的任务，同时需要自己去调度线程执行任务并维护上下文切换，这一切通常会耗费程序员大量的心智。那么能不能有一种机制，程序员只需要定义很多个任务，让系统去帮助我们把这些任务分配到CPU上实现并发执行呢？

Go语言中的 `goroutine` 就是这样一种机制， `goroutine` 的概念类似于线程，但 `goroutine` 是由Go的运行时（runtime）调度和管理的。Go程序会智能地将 goroutine 中的任务合理地分配给每个CPU。Go语言之所以被称为现代化的编程语言，就是因为它在语言层面已经内置了调度和上下文切换的机制。

在Go语言编程中你不需要去自己写进程、线程、协程，你的技能包里只有一个技能–`goroutine`，当你需要让某个任务并发执行的时候，你只需要把这个任务包装成一个函数，开启一个`goroutine`去执行这个函数就可以了，就是这么简单粗暴。

使用goroutine

Go语言中使用`goroutine`非常简单，只需要在调用函数的时候在前面加上`go`关键字，就可以为一个函数创建一个`goroutine`。

一个`goroutine`必定对应一个函数，可以创建多个`goroutine`去执行相同的函数。

启动单个goroutine

启动goroutine的方式非常简单，只需要在调用的函数（普通函数和匿名函数）前面加上一个`go`关键字。

举个例子如下：

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

这个示例中hello函数和下面的语句是串行的，执行的结果是打印完`Hello Goroutine!`后打印`main goroutine done!`。

接下来我们在调用hello函数前面加上关键字`go`，也就是启动一个goroutine去执行hello这个函数。

```
func main() {
    go hello() // 启动另外一个goroutine去执行hello函数
    fmt.Println("main goroutine done!")
}
```

这一次的执行结果只打印了`main goroutine done!`，并没有打印`Hello Goroutine!`。为什么呢?

在程序启动时，Go程序就会为`main()`函数创建一个默认的`goroutine`。

当main()函数返回的时候该`goroutine`就结束了，所有在`main()`函数中启动的`goroutine`会一同结束，`main`函数所在的`goroutine`就像是权利的游戏中的夜王，其他的`goroutine`都是异鬼，夜王一死它转化的那些异鬼也就全部GG了。

所以我们要想办法让main函数等一等hello函数，最简单粗暴的方式就是`time.Sleep`了。

```
func main() {
    go hello() // 启动另外一个goroutine去执行hello函数
    fmt.Println("main goroutine done!")
    time.Sleep(time.Second)
}
```

执行上面的代码你会发现，这一次先打印 `main goroutine done!` ，然后紧接着打印 `Hello Goroutine!` 。

首先为什么会先打印 `main goroutine done!` 是因为我们在创建新的goroutine的时候需要花费一些时间，而此时main函数所在的 `goroutine` 是继续执行的。

启动多个goroutine

在Go语言中实现并发就是这样简单，我们还可以启动多个 `goroutine` 。让我们再来一个例子：（这里使用了 `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
}
```

多次执行上面的代码，会发现每次打印的数字的顺序都不一致。这是因为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` 的栈在其生命周期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， `goroutine` 的栈大小限制可以达到1GB，虽然极少会用到这个大。所以在Go语言中一次创建十万左右的 `goroutine` 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。区别于操作系统调度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的绑定等信息。
- 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，堆栈地址及地址边界），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goroutine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，如果全局队列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长久阻塞在一个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G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。 在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` 则是由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， 不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。 另一方面充分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的超轻量，以上种种保证了go调度方面的性能。

点我了解更多

GOMAXPROCS

Go运行时的调度器使用 `GOMAXPROCS` 参数来确定需要使用多少个OS线程来同时执行Go代码。默认值是机器上的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（GOMAXPROCS是m:n调度中的n）。

Go语言中可以通过 `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，这里举个例子：

```
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，此时两个任务并行执行，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

单纯地将函数并发执行是没有意义的。函数与函数间需要交换数据才能体现并发执行函数的意义。

虽然可以使用共享内存进行数据交换，但是共享内存在不同的 `goroutine` 中容易发生竞态问题。为了保证数据交换的正确性，必须使用互斥量对内存进行加锁，这种做法势必造成性能问题。

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**通过共享内存而实现通信**。

如果说 `goroutine` 是Go程序并发的执行体， `channel` 就是它们之间的连接。 `channel` 是可以让一个 `goroutine` 发送特定值到另一个 `goroutine` 的通信机制。

Go 语言中的通道（channel）是一种特殊的类型。通道像一个传送带或者队列，总是遵循先入先出（First In First Out）的规则，保证收发数据的顺序。每一个通道都是一个具体类型的导管，也就是声明channel的时候需要为其指定元素类型。

channel类型

`channel` 是一种类型，一种引用类型。声明通道类型的格式如下：

```
var 变量 chan 元素类型
```

举几个例子：

```
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，通道类型的空值是 `nil` 。

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

声明的通道后需要使用 `make` 函数初始化之后才能使用。

创建channel的格式如下：

```
make(chan 元素类型, [缓冲大小])
```

channel的缓冲大小是可选的。

举几个例子：

```
ch4 := make(chan int)
ch5 := make(chan bool)
ch6 := 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。

现在我们先使用以下语句定义一个通道：

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

发送

将一个值发送到通道中。

```
ch <- 10 // 把10发送到ch中
```

接收

从一个通道中接收值。

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

关闭

我们通过调用内置的 `close` 函数来关闭通道。

```
close(ch)
```

关于关闭通道需要注意的事情是，只有在通知接收方goroutine所有的数据都发送完毕的时候才需要关闭通道。通道是可以被垃圾回收机制回收的，它和关闭文件是不一样的，在结束操作之后关闭文件是必须要做的，但关闭通道不是必须的。

关闭后的通道有以下特点：

1. 对一个关闭的通道再发送值就会导致panic。
2. 对一个关闭的通道进行接收会一直获取值直到通道为空。
3. 对一个关闭的并且没有值的通道执行接收操作会得到对应类型的零值。

4. 关闭一个已经关闭的通道会导致panic。

无缓冲的通道

无缓冲的通道又称为阻塞的通道。我们来看一下下面的代码：

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

上面这段代码能够通过编译，但是执行的时候会出现以下错误：

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

goroutine 1 [chan send]:
main.main()
        .../src/github.com/Q1mi/studygo/day06/channel02/main.go:8 +0x54
```

为什么会出现 `deadlock` 错误呢?

因为我们使用 `ch := make(chan int)` 创建的是无缓冲的通道，无缓冲的通道只有在有人接收值的时候才能发送值。就像你住的小区没有快递柜和代收点，快递员给你打电话必须要把这个物品送到你的手中，简单来说就是无缓冲的通道必须有接收才能发送。

上面的代码会阻塞在 `ch <- 10` 这一行代码形成死锁，那如何解决这个问题呢?

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，这时值才能发送成功，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，直到另一个 `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 `同步通道` 。

有缓冲的通道

解决上面问题的方法还有一种就是使用有缓冲区的通道。我们可以在使用make函数初始化通道的时候为其指定通道的容量，例如：

```
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零，那么该通道就是有缓冲的通道，通道的容量表示通道中能存放元素的数量。就像你小区的快递柜只有那么个多格子，格子满了就装不下了，就阻塞了，等到别人取走一个快递员就能往里面放一个。

我们可以使用内置的 `len` 函数获取通道内元素的数量，使用 `cap` 函数获取通道的容量，虽然我们很少会这么做。

for range从通道循环取值

当向通道中发送完数据时，我们可以通过 `close` 函数来关闭通道。

当通道被关闭时，再往该通道发送值会引发 `panic` ，从该通道取值的操作会先取完通道中的值，再然后取到的值一直都是对应类型的零值。那如何判断一个通道是否被关闭了呢?

我们来看下面这个例子：

```
// channel 练习
func main() {
    ch1 := make(chan int)
    ch2 := make(chan int)
    // 开启goroutine将0~100的数发送到ch1中
    go func() {
        for i := 0; i < 100; 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)
    }
}
```

从上面的例子中我们看到有两种方式在接收值的时候判断该通道是否被关闭，不过我们通常使用的是`for range`的方式。使用`for range`遍历通道，当通道被关闭的时候就会退出`for range`。

单向通道

有的时候我们会将通道作为参数在多个任务函数间传递，很多时候我们在不同的任务函数中使用通道都会对其进行限制，比如限制通道在函数中只能发送或只能接收。

Go语言中提供了**单向通道**来处理这种情况。例如，我们把上面的例子改造如下：

```
func counter(out chan<- int) {
    for i := 0; i < 100; 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)
}
```

其中，

- `chan<- int`是一个只写单向通道（只能对其写入int类型值），可以对其执行发送操作但是不能执行接收操作；
- `<-chan int`是一个只读单向通道（只能从其读取int类型值），可以对其执行接收操作但是不能执行发送操作。

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，但反过来是不可以的。

通道总结

`channel` 常见的异常总结，如下图：

channel异常情况总结					
channel	nil	非空	空的	满了	没满
接收	阻塞	接收值	阻塞	接收值	接收值
发送	阻塞	发送值	发送值	阻塞	发送值
关闭	panic	关闭成功，读完数据后返回零值	关闭成功，返回零值	关闭成功，读完数据后返回零值	关闭成功，读完数据后返回零值

关闭已经关闭的 `channel` 也会引发 `panic` 。

worker pool（goroutine池）

在工作中我们通常会使用可以指定启动的goroutine数量–`worker pool` 模式，控制 `goroutine` 的数量，防止 `goroutine` 泄漏和暴涨。

一个简易的 `work pool` 示例代码如下：

```
func worker(id int, jobs <-chan int, results chan<- int) {
    for j := range jobs {
        fmt.Printf("worker:%d start job:%d\n", id, j)
        time.Sleep(time.Second)
        fmt.Printf("worker:%d end job:%d\n", id, j)
        results <- j * 2
    }
}


func main() {
    jobs := make(chan int, 100)
    results := make(chan int, 100)
    // 开启3个goroutine
    for w := 1; w <= 3; w++ {
        go worker(w, jobs, results)
    }
    // 5个任务
    for j := 1; j <= 5; j++ {
        jobs <- j
    }
    close(jobs)
    // 输出结果
    for a := 1; a <= 5; a++ {
        <-results
    }
}
```

select多路复用

在某些场景下我们需要同时从多个通道接收数据。通道在接收数据时，如果没有数据可以接收将会发生阻塞。你也许会写出如下代码使用遍历的方式来实现：

```
for{
    // 尝试从ch1接收值
    data, ok := <-ch1
    // 尝试从ch2接收值
    data, ok := <-ch2
    …
}
```

这种方式虽然可以实现从多个通道接收值的需求，但是运行性能会差很多。为了应对这种场景，Go内置了`select`关键字，可以同时响应多个通道的操作。

`select`的使用类似于switch语句，它有一系列case分支和一个默认的分支。每个case会对应一个通道的通信（接收或发送）过程。`select`会一直等待，直到某个`case`的通信操作完成时，就会执行`case`分支对应的语句。具体格式如下：

```
select{
    case <-ch1:
        ...
    case data := <-ch2:
        ...
    case ch3<-data:
        ...
    default:
        默认操作
}
```

举个小例子来演示下 `select` 的使用：

```
func main() {
    ch := make(chan int, 1)
    for i := 0; i < 10; i++ {
        select {
        case x := <-ch:
            fmt.Println(x)
        case ch <- i:
        }
    }
}
```

使用 `select` 语句能提高代码的可读性。

- 可处理一个或多个channel的发送/接收操作。
- 如果多个 case同时满足， select会随机选择一个。
- 对于没有 case的 select{}会一直等待，可用于阻塞main函数。

并发安全和锁

有时候在Go代码中可能会存在多个 `goroutine` 同时操作一个资源（临界区），这种情况会发生 `竞态问题` （数据竞态）。类比现实生活中的例子有十字路口被各个方向的的汽车竞争；还有火车上的卫生间被车厢里的人竞争。

举个例子：

```
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)
}
```

上面的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` 变量的时候就会存在数据竞争，导致最后的结果与期待的不符。

互斥锁

互斥锁是一种常用的控制共享资源访问的方法，它能够保证同时只有一个 `goroutine` 可以访问共享资源。Go语言中使用 `sync` 包的 `Mutex` 类型来实现互斥锁。 使用互斥锁来修复上面代码的问题：

```
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)
}
```

使用互斥锁能够保证同一时间有且只有一个 `goroutine` 进入临界区，其他的 `goroutine` 则在等待锁；当互斥锁释放后，等待的 `goroutine` 才可以获取锁进入临界区，多个 `goroutine` 同时等待一个锁时，唤醒的策略是随机的。

读写互斥锁

互斥锁是完全互斥的，但是有很多实际的场景下是读多写少的，当我们并发的去读取一个资源不涉及资源修改的时候是没有必要加锁的，这种场景下使用读写锁是更好的一种选择。读写锁在Go语言中使用`sync`包中的`RWMutex`类型。

读写锁分为两种：读锁和写锁。当一个goroutine获取读锁之后，其他的`goroutine`如果是获取读锁会继续获得锁，如果是获取写锁就会等待；当一个`goroutine`获取写锁之后，其他的`goroutine`无论是获取读锁还是写锁都会等待。

读写锁示例：

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

func write() {
    // 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))
}
```

需要注意的是读写锁非常适合读多写少的场景，如果读和写的操作差别不大，读写锁的优势就发挥不出来。

sync.WaitGroup

在代码中生硬的使用 `time.Sleep` 肯定是不合适的，Go语言中可以使用 `sync.WaitGroup` 来实现并发任务的同步。 `sync.WaitGroup` 有以下几个方法：

方法名	功能

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，计数器的值可以增加和减少。例如当我们启动了N 个并发任务时，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()来等待并发任务执行完，当计数器值为0时，表示所有并发任务已经完成。

我们利用 `sync.WaitGroup` 将上面的代码优化一下：

```
var wg sync.WaitGroup

func hello() {
    defer wg.Done()
    fmt.Println("Hello Goroutine!")
}
func main() {
    wg.Add(1)
    go hello() // 启动另外一个goroutine去执行hello函数
    fmt.Println("main goroutine done!")
    wg.Wait()
}
```

需要注意 `sync.WaitGroup` 是一个结构体，传递的时候要传递指针。

sync.Once

说在前面的话：这是一个进阶知识点。

在编程的很多场景下我们需要确保某些操作在高并发的场景下只执行一次，例如只加载一次配置文件、只关闭一次通道等。

Go语言中的 `sync` 包中提供了一个针对只执行一次场景的解决方案– `sync.Once` 。

`sync.Once` 只有一个 `Do` 方法，其签名如下：

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

备注：如果要执行的函数 `f` 需要传递参数就需要搭配闭包来使用。

加载配置文件示例

延迟一个开销很大的初始化操作到真正用到它的时候再执行是一个很好的实践。因为预先初始化一个变量（比如在init函数中完成初始化）会增加程序的启动耗时，而且有可能实际执行过程中这个变量没有用上，那么这个初始化

操作就不是必须要做的。我们来看一个例子：

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

func loadIcons() {
    icons = map[string]image.Image{
        "left":  loadIcon("left.png"),
        "up":    loadIcon("up.png"),
        "right": loadIcon("right.png"),
        "down":  loadIcon("down.png"),
    }
}

// Icon 被多个goroutine调用时不是并发安全的
func Icon(name string) image.Image {
    if icons == nil {
        loadIcons()
    }
    return icons[name]
}
```

多个 `goroutine` 并发调用Icon函数时不是并发安全的，现代的编译器和CPU可能会在保证每个 `goroutine` 都满足串行一致的基础上自由地重排访问内存的顺序。loadIcons函数可能会被重排为以下结果：

```
func loadIcons() {
    icons = make(map[string]image.Image)
    icons["left"] = loadIcon("left.png")
    icons["up"] = loadIcon("up.png")
    icons["right"] = loadIcon("right.png")
    icons["down"] = loadIcon("down.png")
}
```

在这种情况下就会出现即使判断了 `icons` 不是nil也不意味着变量初始化完成了。考虑到这种情况，我们能想到的办法就是添加互斥锁，保证初始化 `icons` 的时候不会被其他的 `goroutine` 操作，但是这样做又会引发性能问题。

使用 `sync.Once` 改造的示例代码如下：

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

var loadIconsOnce sync.Once

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)
    return icons[name]
}
```

并发安全的单例模式

下面是借助 `sync.Once` 实现的并发安全的单例模式：

```
package singleton

import (
    "sync"
)

type singleton struct {}

var instance *singleton
var once sync.Once

func GetInstance() *singleton {
    once.Do(func() {
        instance = &singleton{}
    })
    return instance
}
```

`sync.Once` 其实内部包含一个互斥锁和一个布尔值，互斥锁保证布尔值和数据的安全，而布尔值用来记录初始化是否完成。这样设计就能保证初始化操作的时候是并发安全的并且初始化操作也不会被执行多次。

sync.Map

Go语言中内置的map不是并发安全的。请看下面的示例：

```
var m = make(map[string]int)

func get(key string) int {
    return m[key]
}

func set(key string, value int) {
    m[key] = value
}

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

上面的代码开启少量几个`goroutine`的时候可能没什么问题，当并发多了之后执行上面的代码就会报`fatal error: concurrent map writes`错误。

像这种场景下就需要为map加锁来保证并发的安全性了，Go语言的`sync`包中提供了一个开箱即用的并发安全版map–`sync.Map`。开箱即用表示不用像内置的map一样使用make函数初始化就能直接使用。同时`sync.Map`内置了诸如`Store`、`Load`、`LoadOrStore`、`Delete`、`Range`等操作方法。

```
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()
}
```

原子操作

代码中的加锁操作因为涉及内核态的上下文切换会比较耗时、代价比较高。针对基本数据类型我们还可以使用原子操作来保证并发安全，因为原子操作是Go语言提供的方法它在用户态就可以完成，因此性能比加锁操作更好。Go语言中原子操作由内置的标准库 `sync/atomic` 提供。

atomic包

方法	解释
func LoadInt32(addr *int32) (val int32) func LoadInt64(addr *int64) (val int64) func LoadUint32(addr *uint32) (val uint32) func LoadUint64(addr *uint64) (val uint64) func LoadUintptr(addr *uintptr) (val uintptr) func LoadPointer(addr *unsafe.Pointer) (val unsafe.Pointer)	读取操作
func StoreInt32(addr *int32, val int32) func StoreInt64(addr *int64, val int64) func StoreUint32(addr *uint32, val uint32) func StoreUint64(addr *uint64, val uint64) func StoreUintptr(addr *uintptr, val uintptr) func StorePointer(addr *unsafe.Pointer, val unsafe.Pointer)	写入操作
func AddInt32(addr *int32, delta int32) (new int32) func AddInt64(addr *int64, delta int64) (new int64) func AddUint32(addr *uint32, delta uint32) (new uint32) func AddUint64(addr *uint64, delta uint64) (new uint64) func AddUintptr(addr *uintptr, delta uintptr) (new uintptr)	修改操作
func SwapInt32(addr *int32, new int32) (old int32) func SwapInt64(addr *int64, new int64) (old int64) func SwapUint32(addr *uint32, new uint32) (old uint32) func SwapUint64(addr *uint64, new uint64) (old uint64) func SwapUintptr(addr *uintptr, new uintptr) (old uintptr) func SwapPointer(addr *unsafe.Pointer, new unsafe.Pointer) (old unsafe.Pointer)	交换操作
func CompareAndSwapInt32(addr *int32, old, new int32) (swapped bool) func CompareAndSwapInt64(addr *int64, old, new int64) (swapped bool) func CompareAndSwapUint32(addr *uint32, old, new uint32) (swapped bool) func CompareAndSwapUint64(addr *uint64, old, new uint64) (swapped bool) func CompareAndSwapUintptr(addr *uintptr, old, new uintptr) (swapped bool) func CompareAndSwapPointer(addr *unsafe.Pointer, old, new unsafe.Pointer) (swapped bool)	比较并交换操作

示例

我们填写一个示例来比较下互斥锁和原子操作的性能。

```
package main
```

```
import (
    "fmt"
    "sync"
    "sync/atomic"
    "time"
)

type Counter interface {
    Inc()
    Load() int64
}

// 普通版
type CommonCounter struct {
    counter int64
}

func (c CommonCounter) Inc() {
    c.counter++
}

func (c CommonCounter) Load() int64 {
    return c.counter
}

// 互斥锁版
type MutexCounter struct {
    counter int64
    lock    sync.Mutex
}

func (m *MutexCounter) Inc() {
    m.lock.Lock()
    defer m.lock.Unlock()
    m.counter++
}

func (m *MutexCounter) Load() int64 {
    m.lock.Lock()
    defer m.lock.Unlock()
    return m.counter
}

// 原子操作版
type AtomicCounter struct {
    counter int64
}

func (a *AtomicCounter) Inc() {
    atomic.AddInt64(&a.counter, 1)
}

func (a *AtomicCounter) Load() int64 {
    return atomic.LoadInt64(&a.counter)
```

```
}

func test(c Counter) {
    var wg sync.WaitGroup
    start := time.Now()
    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go func() {
            c.Inc()
            wg.Done()
        }()
    }
    wg.Wait()
    end := time.Now()
    fmt.Println(c.Load(), end.Sub(start))
}

func main() {
    c1 := CommonCounter{} // 非并发安全
    test(c1)
    c2 := MutexCounter{} // 使用互斥锁实现并发安全
    test(&c2)
    c3 := AtomicCounter{} // 并发安全且比互斥锁效率更高
    test(&c3)
}
```

`atomic` 包提供了底层的原子级内存操作，对于同步算法的实现很有用。这些函数必须谨慎地保证正确使用。除了某些特殊的底层应用，使用通道或者sync包的函数/类型实现同步更好。

练习题

1. 使用goroutine和channel实现一个计算int64随机数各位数和的程序。
    1. 开启一个goroutine循环生成int64类型的随机数，发送到jobChan
    2. 开启24个goroutine从jobChan中取出随机数计算各位数的和，将结果发送到resultChan
    3. 主goroutine从resultChan取出结果并打印到终端输出
2. 为了保证业务代码的执行性能将之前写的日志库改写为异步记录日志方式。