并发

一、Go语言并发简述（并发的优势）

有人把Go语言比作 21 世纪的C语言，第一是因为Go语言设计简单，第二则是因为 21 世纪最重要的就是并发程序设计，而 Go 从语言层面就支持并发。同时实现了自动垃圾回收机制。

Go语言的并发机制运用起来非常简便，在启动并发的方式上直接添加了语言级的关键字就可以实现，和其他编程语言相比更加轻量。

下面来介绍几个概念：

进程/线程

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基本单位。

一个进程可以创建和撤销多个线程，同一个进程中的多个线程之间可以并发执行。

并发/并行

多线程程序在单核心的 cpu 上运行，称为并发；多线程程序在多核心的 cpu 上运行，称为并行。

并发与并行并不相同，并发主要由切换时间片来实现“同时”运行，并行则是直接利用多核实现多线程的运行，Go程序可以设置使用核心数，以发挥多核计算机的能力。

协程/线程

协程：独立的栈空间，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，这些用户级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。

优雅的并发编程范式，完善的并发支持，出色的并发性能是Go语言区别于其他语言的一大特色。使用Go语言开发服务器程序时，就需要对它的并发机制有深入的了解。

并发模型

业界将如何实现并发编程总结归纳为各式各样的并发模型，常见的并发模型有以下几种：

- 线程&锁模型
- Actor模型
- CSP模型
- Fork&Join模型

Go语言中的并发程序主要是通过基于CSP（communicating sequential processes）的goroutine和channel来实现，当然也支持使用传统的多线程共享内存的并发方式。

Goroutine 介绍

Goroutine 是 Go 语言支持并发的核心，在一个Go程序中同时创建成百上千个goroutine是非常普遍的，一个goroutine会以一个很小的栈开始其生命周期，一般只需要2KB。区别于操作系统线程由系统内核进行调度， goroutine 是由Go运行时（runtime）负责调度。例如Go运行时会智能地将 m个 goroutine 合理地分配给n个操作系统线程，实现类似m:n的调度机制，不再需要Go开发者自行在代码层面维护一个线程池。

Goroutine 是 Go 程序中最基本的并发执行单元。每一个 Go 程序都至少包含一个 goroutine——main goroutine，当 Go 程序启动时它会自动创建。

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。

go关键字

Go语言中使用 goroutine 非常简单，只需要在函数或方法调用前加上`go`关键字就可以创建一个 goroutine ，从而让该函数或方法在新创建的 goroutine 中执行。

goroutine 的用法如下：

```
//go 关键字放在方法调用前新建一个 goroutine 并执行方法体
go GetThingDone(param1, param2);
//新建一个匿名方法并执行
go func(param1, param2) {
}(val1, val2)
//直接新建一个 goroutine 并在 goroutine 中执行代码块
go {
    //do someting...
}
```

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

启动单个goroutine

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

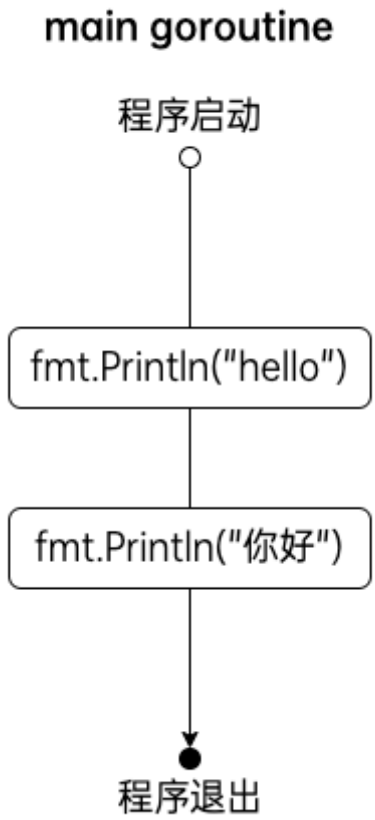
我们先来看一个在 main 函数中执行普通函数调用的示例。

```
package main

import (
    "fmt"
)

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

func main() {
    hello()
    fmt.Println("你好")
}
```

将上面的代码编译后执行，得到的结果如下：

```
hello
你好
```

代码中 hello 函数和其后面的打印语句是串行的。



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。

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

将上述代码重新编译后执行，得到输出结果如下。

```
你好
```

这一次的执行结果只在终端打印了”你好”，并没有打印`hello`。这是为什么呢?

其实在 Go 程序启动时，Go 程序就会为 main 函数创建一个默认的 goroutine 。在上面的代码中我们在 main 函数中使用 go 关键字创建了另外一个 goroutine 去执行 hello 函数，而此时 main goroutine 还在继续往下执行，我们的程序中此时存在两个并发执行的 goroutine。当 main 函数结束时整个程序也就结束了，同时 main goroutine 也结束了，所有由 main goroutine 创建的 goroutine 也会一同退出。也就是说我们的 main 函数退出太快，另外一个 goroutine 中的函数还未执行完程序就退出了，导致未打印出“hello”。

main goroutine 就像是《权利的游戏》中的夜王，其他的 goroutine 都是夜王转化出的异鬼，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”将在另一个 goroutine 中运行的 hello 函数。其中最简单粗暴的方式就是在 main 函数中“time.Sleep”一秒钟了（这里的1秒钟只是我们为了保证新的 goroutine 能够被正常创建和执行而设置的一个值）。

按如下方式修改我们的示例代码。

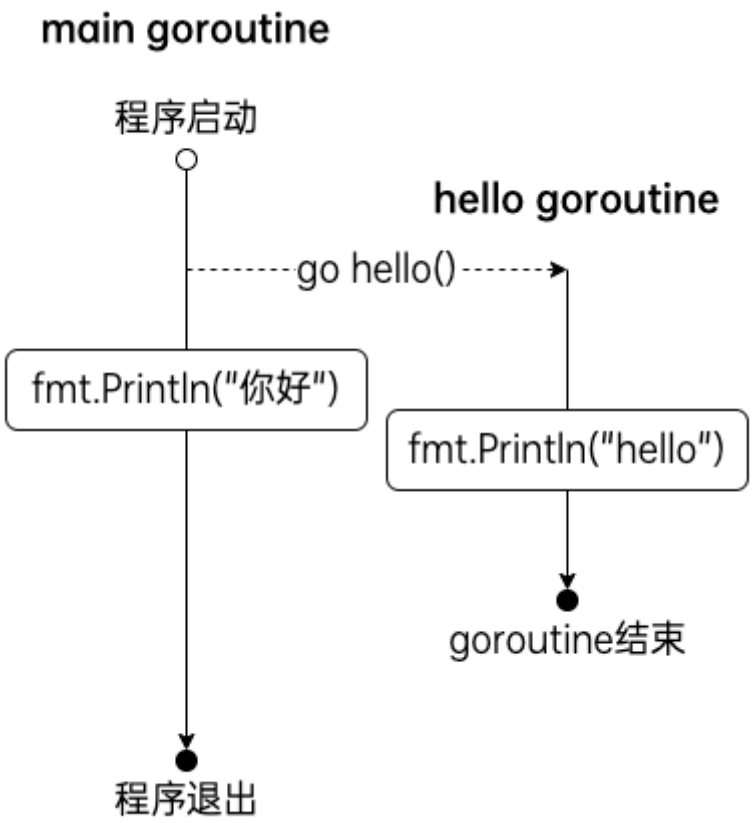
```
package main

import (
    "fmt"
    "time"
)

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

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

将我们的程序重新编译后再次执行，程序会在终端输出如下结果，并且会短暂停顿一会儿。

```
你好
hello
```

为什么会先打印 `你好` 呢?

这是因为在程序中创建 goroutine 执行函数需要一定的开销，而与此同时 main 函数所在的 goroutine 是继续执行的。



在上面的程序中使用`time.Sleep`让 main goroutine 等待 hello goroutine执行结束是不优雅的，当然也是不准确的。

Go 语言中通过`sync`包为我们提供了一些常用的并发原语，我们会在后面的小节单独介绍`sync`包中的内容。在这一小节，我们会先介绍一下 sync 包中的`WaitGroup`。当你并不关心并发操作的结果或者有其它方式收集并发操作的结果时，`WaitGroup`是实现等待一组并发操作完成的好方法。

下面的示例代码中我们在 main goroutine 中使用`sync.WaitGroup`来等待 hello goroutine 完成后再退出。

```
package main

import (
    "fmt"
    "sync"
)

// 声明全局等待组变量
var wg sync.WaitGroup

func hello() {
    fmt.Println("hello")
    wg.Done() // 告知当前goroutine完成
}

func main() {
    wg.Add(1) // 登记1个goroutine
    go hello()
    fmt.Println("你好")
    wg.Wait() // 阻塞等待登记的goroutine完成
```

```
}
```

将代码编译后再执行，得到的输出结果和之前一致，但是这一次程序不再会有多余的停顿，hello goroutine 执行完毕后程序直接退出。

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 。让我们再来看一个新的代码示例。这里同样使用了`sync.WaitGroup`来实现 goroutine 的同步。

```
package main

import (
    "fmt"
    "sync"
)

var wg sync.WaitGroup

func hello(i int) {
    defer wg.Done() // goroutine结束就登记-1
    fmt.Println("hello", 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终端上打印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。

动态栈

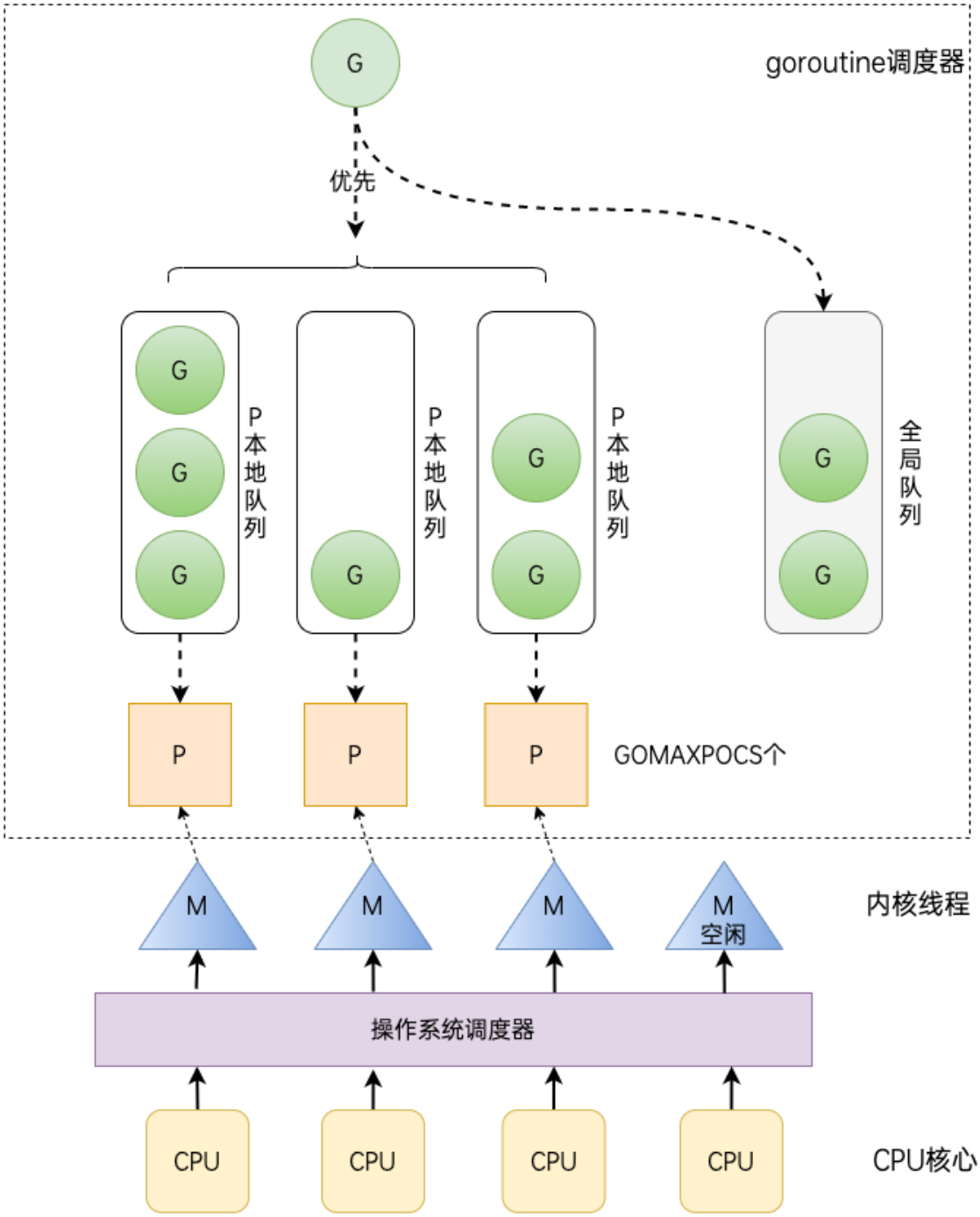
操作系统的线程一般都有固定的栈内存（通常为2MB）,而 Go 语言中的 goroutine 非常轻量级，一个 goroutine 的初始栈空间很小（一般为2KB），所以在 Go 语言中一次创建数万个 goroutine 也是可能的。并且 goroutine 的栈不是固定的，可以根据需要动态地增大或缩小， Go 的 runtime 会自动为 goroutine 分配合适的栈空间。

goroutine调度

操作系统内核在调度时会挂起当前正在执行的线程并将寄存器中的内容保存到内存中，然后选出接下来要执行的线程并从内存中恢复该线程的寄存器信息，然后恢复执行该线程的现场并开始执行线程。从一个线程切换到另一个线程需要完整的上下文切换。因为可能需要多次内存访问，索引这个切换上下文的操作开销较大，会增加运行的cpu周期。

区别于操作系统内核调度操作系统线程，goroutine 的调度是Go语言运行时（runtime）层面的实现，是完全由 Go 语言本身实现的一套调度系统——go scheduler。它的作用是按照一定的规则将所有的 goroutine 调度到操作系统线程上执行。

在经历数个版本的迭代之后，目前 Go 语言的调度器采用的是`GPM`调度模型。



其中：

- G：表示 goroutine，每执行一次 `go f()` 就创建一个 G，包含要执行的函数和上下文信息。
- 全局队列（Global Queue）：存放等待运行的 G。
- P：表示 goroutine 执行所需的资源，最多有 GOMAXPROCS 个。
- P 的本地队列：同全局队列类似，存放的也是等待运行的G，存的数量有限，不超过256个。新建 G 时，G 优先加入到 P 的本地队列，如果本地队列满了会批量移动部分 G 到全局队列。
- M：线程想运行任务就得获取 P，从 P 的本地队列获取 G，当 P 的本地队列为空时，M 也会尝试从全局队列或其他 P 的本地队列获取 G。M 运行 G，G 执行之后，M 会从 P 获取下一个 G，不断重复下去。
- Goroutine 调度器和操作系统调度器是通过 M 结合起来的，每个 M 都代表了1个内核线程，操作系统调度器负责把内核线程分配到 CPU 的核上执行。

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，完全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 的超轻量级，以上种种特性保证了 goroutine 调度方面的性能。

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，GOMAXPROCS 默认为 8。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。）

并发安全和锁

有时候我们的代码中可能会存在多个 goroutine 同时操作一个资源（临界区）的情况，这种情况下就会发生`竞态问题`（数据竞态）。这就好比现实生活中十字路口被各个方向的汽车竞争，还有火车上的卫生间被车厢里的人竞争。

我们用下面的代码演示一个数据竞争的示例。

```
package main

import (
    "fmt"
    "sync"
)

var (
    x int64

    wg sync.WaitGroup // 等待组
)

// add 对全局变量x执行5000次加1操作
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)
}
```

我们将上面的代码编译后执行，不出意外每次执行都会输出诸如9537、5865、6527等不同的结果。这是为什么呢?

在上面的示例代码片中，我们开启了两个 goroutine 分别执行 add 函数，这两个 goroutine 在访问和修改全局的`x`变量时就会存在数据竞争，某个 goroutine 中对全局变量`x`的修改可能会覆盖掉另一个 goroutine 中的操作，所以导致最后的结果与预期不符。

互斥锁

互斥锁是一种常用的控制共享资源访问的方法，它能够保证同一时间只有一个 goroutine 可以访问共享资源。Go 语言中使用`sync`包中提供的`Mutex`类型来实现互斥锁。

`sync.Mutex`提供了两个方法供我们使用。

方法名	功能
func (m *Mutex) Lock()	获取互斥锁
func (m *Mutex) Unlock()	释放互斥锁

我们在下面的示例代码中使用互斥锁限制每次只有一个 goroutine 才能修改全局变量`x`，从而修复上面代码中的问题。

```
package main

import (
    "fmt"
    "sync"
)

// sync.Mutex

var (
    x int64

    wg sync.WaitGroup // 等待组

    m sync.Mutex // 互斥锁
)

// add 对全局变量x执行5000次加1操作
func add() {
    for i := 0; i < 5000; i++ {
        m.Lock() // 修改x前加锁
        x = x + 1
        m.Unlock() // 改完解锁
    }
    wg.Done()
}

func main() {
    wg.Add(2)

    go add()
    go add()
```

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

将上面的代码编译后多次执行，每一次都会得到预期中的结果——10000。

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，但是实际上有很多场景是读多写少的，当我们并发的去读取一个资源而不涉及资源修改的时候是没有必要加互斥锁的，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。

`sync.RWMutex` 提供了以下5个方法。

方法名	功能
func (rw *RWMutex) Lock()	获取写锁
func (rw *RWMutex) Unlock()	释放写锁
func (rw *RWMutex) RLock()	获取读锁
func (rw *RWMutex) RUnlock()	释放读锁
func (rw *RWMutex) RLocker() Locker	返回一个实现Locker接口的读写锁

读写锁分为两种：读锁和写锁。当一个 goroutine 获取到读锁之后，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；而当一个 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。

下面我们使用代码构造一个读多写少的场景，然后分别使用互斥锁和读写锁查看它们的性能差异。

```
var (
    x       int64
    wg      sync.WaitGroup
    mutex   sync.Mutex
    rwMutex sync.RWMutex
)

// writeWithLock 使用互斥锁的写操作
func writeWithLock() {
    mutex.Lock() // 加互斥锁
    x = x + 1
    time.Sleep(10 * time.Millisecond) // 假设读操作耗时10毫秒
    mutex.Unlock()                    // 解互斥锁
    wg.Done()
}

// readWithLock 使用互斥锁的读操作
func readWithLock() {
```

```
    mutex.Lock()                  // 加互斥锁
    time.Sleep(time.Millisecond) // 假设读操作耗时1毫秒
    mutex.Unlock()                // 释放互斥锁
    wg.Done()
}

// writeWithLock 使用读写互斥锁的写操作
func writeWithRWLock() {
    rwMutex.Lock() // 加写锁
    x = x + 1
    time.Sleep(10 * time.Millisecond) // 假设读操作耗时10毫秒
    rwMutex.Unlock()                  // 释放写锁
    wg.Done()
}

// readWithRWLock 使用读写互斥锁的读操作
func readWithRWLock() {
    rwMutex.RLock()               // 加读锁
    time.Sleep(time.Millisecond) // 假设读操作耗时1毫秒
    rwMutex.RUnlock()             // 释放读锁
    wg.Done()
}

func do(wf, rf func(), wc, rc int) {
    start := time.Now()
    // wc个并发写操作
    for i := 0; i < wc; i++ {
        wg.Add(1)
        go wf()
    }

    //  rc个并发读操作
    for i := 0; i < rc; i++ {
        wg.Add(1)
        go rf()
    }

    wg.Wait()
    cost := time.Since(start)
    fmt.Printf("x:%v cost:%v\n", x, cost)

}
```

我们假设每一次读操作都会耗时1ms，而每一次写操作会耗时10ms，我们分别测试使用互斥锁和读写互斥锁执行10次并发写和1000次并发读的耗时数据。

```
// 使用互斥锁，10并发写，1000并发读
do(writeWithLock, readWithLock, 10, 1000) // x:10 cost:1.466500951s

// 使用读写互斥锁，10并发写，1000并发读
do(writeWithRWLock, readWithRWLock, 10, 1000) // x:10 cost:117.207592ms
```

从最终的执行结果可以看出，使用读写互斥锁在读多写少的场景下能够极大地提高程序的性能。不过需要注意的是如果一个程序中的读操作和写操作数量级差别不大，那么读写互斥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
func (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，进行参数传递的时候要传递指针。

sync.Once

在某些场景下我们需要确保某些操作即使在高并发的场景下也只会被执行一次，例如只加载一次配置文件等。

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())
```

注意：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，请看下面这段示例代码。

```
package main

import (
    "fmt"
    "strconv"
    "sync"
)

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 < 10; 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()
}
```

将上面的代码编译后执行，会报出`fatal error: concurrent map writes`错误。我们不能在多个 goroutine 中并发对内置的 map 进行读写操作，否则会存在数据竞争问题。

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`。开箱即用表示其不用像内置的 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。

方法名	功能
func (m *Map) Store(key, value interface{})	存储key-value数据
func (m *Map) Load(key interface{}) (value interface{}, ok bool)	查询key对应的value
func (m *Map) LoadOrStore(key, value interface{}) (actual interface{}, loaded bool)	查询或存储key对应的value
func (m *Map) LoadAndDelete(key interface{}) (value interface{}, loaded bool)	查询并删除key
func (m *Map) Delete(key interface{})	删除key
func (m *Map) Range(f func(key, value interface{}) bool)	对map中的每个key-value依次调用f

下面的代码示例演示了并发读写`sync.Map`。

```
package main

import (
    "fmt"
    "strconv"
    "sync"
)

// 并发安全的map
var m = sync.Map{}

func main() {
    wg := sync.WaitGroup{}
    // 对m执行20个并发的读写操作
    for i := 0; i < 20; i++ {
        wg.Add(1)
        go func(n int) {
            key := strconv.Itoa(n)
            m.Store(key, n)          // 存储key-value
            value, _ := m.Load(key) // 根据key取值
            fmt.Printf("k=:%v,v:=%v\n", key, value)
```

```
            wg.Done()
        }(i)
    }
    wg.Wait()
}
```

原子操作

针对整数数据类型（int32、uint32、int64、uint64）我们还可以使用原子操作来保证并发安全，通常直接使用原子操作比使用锁操作效率更高。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。