一、sync包

1、基本介绍

sync包提供了基本的同步基元，如互斥锁。除了Once和WaitGroup类型，大部分都是适用于低水平程序线程，高水平的同步使用channel通信更好一些。本包的类型的值不应被拷贝。

2、Locker接口

Locker接口提供了加锁和解锁的方法，对于可以加锁和解锁的对象需要实现这个接口。在sync包中有Mutex互斥锁和RWMutex读写锁实现了这个接口。

type Locker

```
type Locker interface {
    Lock()
    Unlock()
}
```

Locker接口代表一个可以加锁和解锁的对象。

- type Locker
- type Once
  - func (o *Once) Do(f func())
- type Mutex
  - func (m *Mutex) Lock()
  - func (m *Mutex) Unlock()
- type RWMutex
  - func (rw *RWMutex) Lock()
  - func (rw *RWMutex) Unlock()
  - func (rw *RWMutex) RLock()
  - func (rw *RWMutex) RUnlock()
  - func (rw *RWMutex) RLocker() Locker
- type Cond
  - func NewCond(l Locker) *Cond
  - func (c *Cond) Broadcast()
  - func (c *Cond) Signal()
  - func (c *Cond) Wait()
- type WaitGroup
  - func (wg *WaitGroup) Add(delta int)
  - func (wg *WaitGroup) Done()
  - func (wg *WaitGroup) Wait()
- type Pool
  - func (p *Pool) Get() interface{}
  - func (p *Pool) Put(x interface{})

3、sync.Once

type Once ¶

```
type Once struct {
    // 包含隐藏或非导出字段
}
```

Once是只执行一次动作的对象。

func (*Once) Do

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

Do方法当且仅当第一次被调用时才执行函数f。换句话说，给定变量：

```
var once Once
```

也就是说如果有多次调用Do，即使每次Do的f函数都不一样，那也只会执行第一次的。

如果once.Do(f)被多次调用，只有第一次调用会执行f，即使f每次调用Do 提供的f值不同。需要给每个要执行仅一次的函数都建立一个Once类型的实例。

Do用于必须刚好运行一次的初始化。因为f是没有参数的，因此可能需要使用闭包来提供给Do方法调用：

```
config.once.Do(func() { config.init(filename) })
```

因为只有f返回后Do方法才会返回，f若引起了Do的调用，会导致死锁。


```go
package main

import (
    "fmt"
    "sync"
)

func main() {

    var once sync.Once      // Once对象

    // 准备好sync.Once对象要执行的函数
    onceBody := func() {
        fmt.Println("Only once")
    }

    done := make(chan bool)     // 信道

    // 开启10条协程
    for i := 0; i < 10; i++ {
        go func() {
            once.Do(onceBody)
            done <- true
        }()
    }

    // 主线程
    for i := 0; i < 10; i++ {
        <-done
```

```
        fmt.Printf("Main --- %d\n", i + 1)
    }
}
```

```
<4 go setup calls>
Only once
Main --- 1
Main --- 2
Main --- 3
Main --- 4
Main --- 5
Main --- 6
Main --- 7
Main --- 8
Main --- 9
Main --- 10
```


4、sync.Mutex

type Mutex

```
type Mutex struct {
    // 包含隐藏或非导出字段
}
```

Mutex是一个互斥锁，可以创建为其他结构体的字段；零值为解锁状态。Mutex类型的锁和线程无关，可以由不同的线程加锁和解锁。

func (*Mutex) Lock

```
func (m *Mutex) Lock()
```

Lock方法锁住m，如果m已经加锁，则阻塞直到m解锁。

func (*Mutex) Unlock

```
func (m *Mutex) Unlock()
```

Unlock方法解锁m，如果m未加锁会导致运行时错误。锁和线程无关，可以由不同的线程加锁和解锁。


```go
package main

import (
    "fmt"
    "sync"
)

// 互斥锁
var mutex sync.Mutex

func main() {

    // 使用互斥实现两个协程的交替打印
    ctrl := make(chan bool, 1)  // 信道

    // 开启协程
```

```
    go func(printChan <-chan bool) {
        for {
            if len(printChan) == 1 {
                mutex.Lock()    // 加锁
                <-printChan
                fmt.Println("gorountine...Print")
                mutex.Unlock()  // 解锁
            }
        }
    }(ctrl)

    // 主线程
    for {
        if len(ctrl) == 0 {
            mutex.Lock()    // 加锁
            ctrl <- true
            fmt.Println("Main...Print")
            mutex.Unlock()  // 解锁
        }
    }
}
```

```
gorountine...Print
Main...Print
gorountine...Print
Main...Print
gorountine...Print
Main...Print
gorountine...Print
Main...Print
gorountine...Print
Main...Print
```

5、sync.RWMutex

type RWMutex

```
type RWMutex struct {
    // 包含隐藏或非导出字段
}
```

RWMutex是读写互斥锁。该锁可以被同时多个读取者持有或唯一个写入者持有。RWMutex可以创建为其他结构体的字段；零值为解锁状态。RWMutex类型的锁也和线程无关，可以由不同的线程加读取锁/写入和解读取锁/写入锁。

func (*RWMutex) Lock

```
func (rw *RWMutex) Lock()
```

Lock方法将rw锁定为写入状态，禁止其他线程读取或者写入。

func (*RWMutex) Unlock

```
func (rw *RWMutex) Unlock()
```

Unlock方法解除rw的写入锁状态，如果m未加写入锁会导致运行时错误。

func (*RWMutex) RLock

```
func (rw *RWMutex) RLock()
```

RLock方法将rw锁定为读取状态，禁止其他线程写入，但不禁止读取。

func (*RWMutex) RUnlock

```
func (rw *RWMutex) RUnlock()
```

Runlock方法解除rw的读取锁状态，如果m未加读取锁会导致运行时错误。

func (*RWMutex) RLocker

```
func (rw *RWMutex) RLocker() Locker
```

Rlocker方法返回一个互斥锁，通过调用rw.Rlock和rw.Runlock实现了Locker接口。


```go
package main

import (
    "fmt"
    "sync"
    "time"
)

// 读写锁
var rwMutex sync.RWMutex
// 全局资源
var SHARE int = 100

func main() {

    // 协程
```

```
    go func() {
        for i := 1; i < 99999999999; i++ {
            rwMutex.Lock()  // 这种加锁方式将会禁止其他线程进入
            fmt.Printf("goruntine Log --> %d\n", SHARE)
            time.Sleep(time.Second * 2) // 睡眠时间到了之后才会去释放锁，其他线程才能拿到
锁
            rwMutex.Unlock()
        }
    }()

    // 主线程
    for i := 1; i < 99999999999; i++ {
        rwMutex.Lock()
        fmt.Printf("Main Log --> %d\n", SHARE)
        rwMutex.Unlock()
    }
}
```

```
package main

import (
    "fmt"
    "sync"
    "time"
)

// 读写锁
var rwMutex sync.RWMutex
// 全局资源
var SHARE int = 100

func main() {

    // 协程
    go func() {
        for i := 1; i < 99999999999; i++ {
            rwMutex.RLock() // 这种加锁方式可以让其他线程拿到读锁
            fmt.Printf("goruntine Log --> %d\n", SHARE)
            time.Sleep(time.Second * 2) // 加锁之后未解锁，此处睡眠两秒，其他线程仍然可以
拿到这个锁，继续执行
            rwMutex.RUnlock()
        }
    }()

    // 主线程
    for i := 1; i < 99999999999; i++ {
        rwMutex.RLock()
        fmt.Printf("Main Log --> %d\n", SHARE)
        rwMutex.RUnlock()
    }
}
```

6、sync.Cond

type Cond

```
type Cond struct {
    // 在观测或更改条件时L会冻结
    L Locker
    // 包含隐藏或非导出字段
}
```

Cond实现了一个条件变量，一个线程集合地，供线程等待或者宣布某事件的发生。

每个Cond实例都有一个相关的锁（一般是*Mutex或*RWMutex类型的值），它必须在改变条件时或者调用Wait方法时保持锁定。Cond可以创建为其他结构体的字段，Cond在开始使用后不能被拷贝。

func NewCond

```
func NewCond(l Locker) *Cond
```

使用锁l创建一个*Cond。

func (*Cond) Broadcast

```
func (c *Cond) Broadcast()
```

Broadcast唤醒所有等待c的线程。调用者在调用本方法时，建议（但并非必须）保持c.L的锁定。

func (*Cond) Signal

```
func (c *Cond) Signal()
```

Signal唤醒等待c的一个线程（如果存在）。调用者在调用本方法时，建议（但并非必须）保持c.L的锁定。

func (*Cond) Wait

```
func (c *Cond) Wait()
```

Wait自行解锁c.L并阻塞当前线程，在之后线程恢复执行时，Wait方法会在返回前锁定c.L。和其他系统不同，Wait除非被Broadcast或者Signal唤醒，不会主动返回。

因为线程中Wait方法是第一个恢复执行的，而此时c.L未加锁。调用者不应假设Wait恢复时条件已满足，相反，调用者应在循环中等待：

```
c.L.Lock()
for !condition() {
    c.Wait()
}
... make use of condition ...
c.L.Unlock()
```



```
package main

import (
    "fmt"
    "sync"
    "time"
)

func main() {
```

```
    signal := 1
    mutex := sync.Mutex{}
    cond := sync.NewCond(&mutex)

    for i := 0; i < 10; i++ {

        go func(int1 int) {

            cond.L.Lock()
            for signal == 1 {
                fmt.Printf("线程%d开始等待\n",int1 + 1)
                cond.Wait() // 等待通知
                fmt.Printf("线程%d结束\n",int1 + 1)
            }
            cond.L.Unlock()
        }(i)
    }

    // 睡眠300毫秒，先让所有协程都进入for循环
    time.Sleep(300 * time.Millisecond)

    //cond.Broadcast()  //通知所有的线程，所有等待中的协程都会继续执行，不再等待
    cond.Signal()   // 只会通知一个线程，其他线程还会处于等待中
    signal = 0  // 控制协程执行一次循环

    // 等待一段时间，防止主线程执行完导致协程提前结束
    time.Sleep(300 * time.Millisecond)
}
```

7、sync.WaitGroup

type WaitGroup

```
type WaitGroup struct {
    // 包含隐藏或非导出字段
}
```

WaitGroup用于等待一组线程的结束。父线程调用Add方法来设定应等待的线程的数量。每个被等待的线程在结束时应调用Done方法。同时，主线程里可以调用Wait方法阻塞至所有线程结束。

Example

func (*WaitGroup) Add

```
func (wg *WaitGroup) Add(delta int)
```

Add方法向内部计数加上delta，delta可以是负数；如果内部计数器变为0，Wait方法阻塞等待的所有线程都会释放，如果计数器小于0，方法panic。注意Add加上正数的调用应在Wait之前，否则Wait可能只会等待很少的线程。一般来说本方法应在创建新的线程或者其他应等待的事件之前调用。

func (*WaitGroup) Done

```
func (wg *WaitGroup) Done()
```

Done方法减少WaitGroup计数器的值，应在线程的最后执行。

func (*WaitGroup) Wait

```
func (wg *WaitGroup) Wait()
```

Wait方法阻塞直到WaitGroup计数器减为0。


```go
package main

import (
    "fmt"
    //"fmt"
    //"net/http"
    "sync"
    //"time"
)

func main() {

    var wg sync.WaitGroup

    var urls = []string{
        "http://www.golang.org/",
        "http://www.google.com/",
        "http://www.somestupidname.com/",
    }

    for _, url := range urls {

        // Increment the WaitGroup counter.
        wg.Add(1)
```

```
        // Launch a goroutine to fetch the URL.
        go func(url string) {

            // Decrement the counter when the goroutine completes.
            defer wg.Done()
            // Print the URL.
            fmt.Println(url)
        }(url)
    }

    // Wait for all HTTP fetches to complete.
    wg.Wait()
}
```

8、sync.Pool

type Pool

```
type Pool struct {
    // 可选参数New指定一个函数在Get方法可能返回nil时来生成一个值
    // 该参数不能在调用Get方法时被修改
    New func() interface{}
    // 包含隐藏或非导出字段
}
```

Pool是一个可以分别存取的临时对象的集合。

Pool中保存的任何item都可能随时不做通告的释放掉。如果Pool持有该对象的唯一引用，这个item就可能被回收。

Pool可以安全的被多个线程同时使用。

Pool的目的是缓存申请但未使用的item用于之后的重用，以减轻GC的压力。也就是说，让创建高效而线程安全的空闲列表更容易。但Pool并不适用于所有空闲列表。

Pool的合理用法是用于管理一组静静的被多个独立并发线程共享并可能重用的临时item。Pool提供了让多个线程分摊内存申请消耗的方法。

Pool的一个好例子在fmt包里。该Pool维护一个动态大小的临时输出缓存仓库。该仓库会在过载（许多线程活跃的打印时）增大，在沉寂时缩小。

另一方面，管理着短寿命对象的空闲列表不适合使用Pool，因为这种情况下内存申请消耗不能很好的分配。这时应该由这些对象自己实现空闲列表。

func (*Pool) Get

```
func (p *Pool) Get() interface{}
```

Get方法从池中选择任意一个item，删除其在池中的引用计数，并提供给调用者。Get方法也可能选择无视内存池，将其当作空的。调用者不应认为Get的返回这和传递给Put的值之间有任何关系。

假使Get方法没有取得item：如p.New非nil，Get返回调用p.New的结果；否则返回nil。

func (*Pool) Put

```
func (p *Pool) Put(x interface{})
```

Put方法将x放入池中。

二、并发编程

1、基本概念

串行、并发与并行 概念

串行：我们都是先读小学，小学毕业后再读初中，读完初中再读高中。

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）。

进程、线程和协程 概念

进程（process）：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（thread）：操作系统基于进程开启的轻量级进程，是操作系统调度执行的最小单位。

协程（coroutine）：非操作系统提供而是由用户自行创建和控制的用户态'线程'，比线程更轻量级。

并发模型 概念

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。

2、goroutine

2.1 基本介绍

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。

2.2 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。

```
func run() {
    XXX
}

go run()
```

匿名函数也支持使用go关键字创建 goroutine 去执行。

```
go func() {

}
```

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。

2.3 启动单个线程

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”`。

```
package main

import (
    "fmt"
    "time"
)

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

func main() {

    go hello()
    fmt.Println("你好")

    // 防止主线程执行太快而结束导致协程未能执行。
    time.Sleep(time.Second)
}
```

上面使用的睡眠方式来处理显然是不合理的，可以使用sync包中的`WaitGroup`来处理，改写如下：

```
package main

import (
    "fmt"
    "sync"
)

// 声明全局等待组变量
var wg sync.WaitGroup

func hello() {

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

```go
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
}
```

2.4 启动多个goroutine

```go
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
}
```

2.5 动态栈

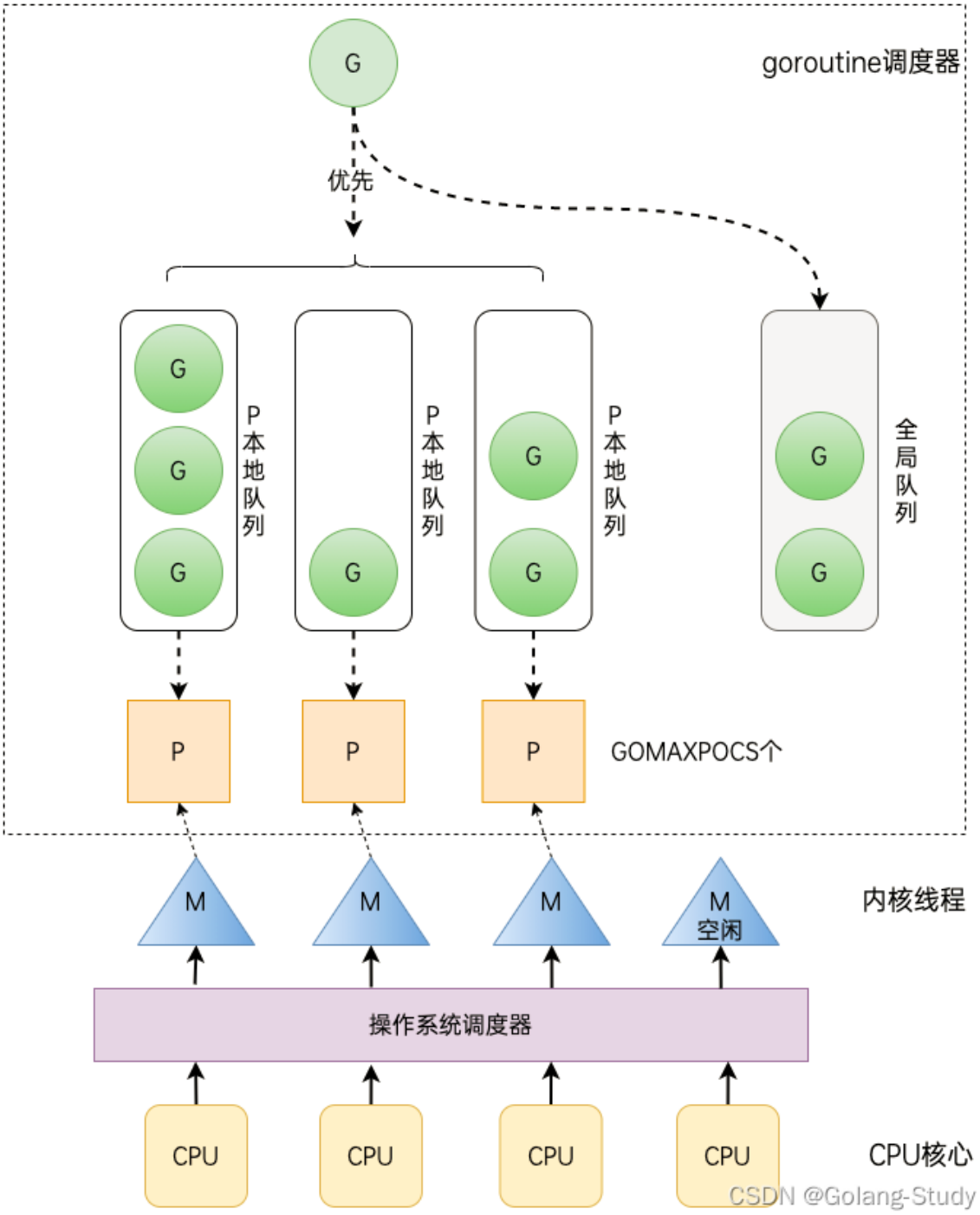
操作系统的线程一般都有固定的栈内存（通常为2MB）,而 Go 语言中的 goroutine 非常轻量级，一个 goroutine 的初始栈空间很小（一般为 2KB ），所以在 Go 语言中一次创建数万个 goroutine 也是可能的。并且 goroutine 的栈不是固定的，`可以根据需要动态地增大或缩小`， Go 的 runtime 会自动为 goroutine 分配合适的栈空间。

2.6 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。

2.7 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`。）

3、channel信道

3.1 基本介绍

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，很多并发模型中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语言采用的并发模型是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。

3.2 channel类型

`channel` 是 Go 语言中一种特有的类型。声明通道类型变量的格式如下：

```
var 变量名称 chan 元素类型
```

其中：

- chan：是关键字
- 元素类型：是指通道中传递元素的类型

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
```


channel是一种引用类型，未初始化的通道类型变量其`默认零值是nil`。

`初始化channel`：

声明的通道类型变量需要使用内置的 `make` 函数初始化之后才能使用。具体格式如下：

```
make(chan 元素类型, [缓冲大小])
```

其中：

- channel的缓冲大小是可选的。

举几个例子：

```
ch4 := make(chan int)
ch5 := make(chan bool, 1)  // 声明一个缓冲区大小为1的通道
```

3.3 channel的操作

通道共有发送（send）、接收(receive）和关闭（close）三种操作。而发送和接收操作都使用 `<-` 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)
```

注意：一个通道值是可以被垃圾回收掉的。通道通常由发送方执行关闭操作，并且只有在接收方明确等待通道关闭的信号时才需要执行关闭操作。它和关闭文件不一样，通常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。

3.4 无缓冲通道

无缓冲的通道又称为阻塞的通道。如下代码片段。

```
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()
        .../main.go:8 +0x54
```

`deadlock` 表示我们程序中的 goroutine 都被挂起导致程序死锁了。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接收方能够接收值的时候才能发送成功，否则会一直处于等待发送的阶段。同理，如果对一个无缓冲通道执行接收操作时，没有任何向通道中发送值的操作那么也会导致接收操作阻塞。就像田径比赛中的4x100接力赛，想要完成交棒必须有一个能够接棒的运动员，否则只能等待。简单来说就是无缓冲的通道必须有至少一个接收方才能发送成功。

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？

其中一种可行的方法是创建一个 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) // 创建一个 goroutine 从通道接收值
    ch <- 10
    fmt.Println("发送成功")
}
```

创建一个接收方，接收这个无缓冲的信道数据

首先无缓冲通道 `ch` 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，这时数字10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，接收方所在的 goroutine 将阻塞，直到 main goroutine 中向该通道发送数字10。

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`。

3.5 有缓冲通道

还有另外一种解决上面死锁问题的方法，那就是使用有缓冲区的通道。我们可以在使用 make 函数初始化通道时，可以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")
}
```

如果有1万个数据，此时创建了一个缓冲容量为1000的信道，一个协程作为发送方，不断的将1万个数据写入信道。

如果没有接收方，那么就会出现deadlock错误，如果有接收方，go底层会判断出来，不会出现deadlock错误。除非接收方无法合理的接收所有数据

只要通道的容量大于零，那么该通道就属于有缓冲的通道，通道的容量表示通道中最大能存放的元素数量。当通道内已有元素数达到最大容量后，再向通道执行发送操作就会阻塞，除非有从通道执行接收操作。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。

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，虽然很少会这么做。

3.6 多返回值模式

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。当一个通道被关闭后，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。通道内的值被接收完后再对通道执行接收操作得到的值会一直都是对应元素类型的零值。那我们如何判断一个通道是否被关闭了呢?

对一个通道执行接收操作时支持使用如下多返回值模式。

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

其中:

- value：从通道中取出的值，如果通道被关闭则返回对应类型的零值。
- ok：通道ch关闭时返回 false，否则返回 true。

下面代码片段中的 `f2` 函数会循环从通道 `ch` 中接收所有值，直到通道被关闭后退出。

```
func f2(ch chan int) {
    for {
        v, ok := <-ch
        if !ok {
            fmt.Println("通道已关闭")
            break
        }
        fmt.Printf("v:%#v ok:%#v\n", v, ok)
    }
}

func main() {
    ch := make(chan int, 2)
    ch <- 1
    ch <- 2
    close(ch)
    f2(ch)
}
```

在读到信道中的数据的时候会返回实际的元素，以及是否读取到，只有当发送方close(channel)，接收方读取才会返回false

for range接收值

通常我们会选择使用 `for range` 循环从通道中接收值，当通道被关闭后，会在通道内的所有值被接收完毕后会自动退出循环。上面那个示例我们使用 `for range` 改写后会很简洁。

```
func f3(ch chan int) {
    for v := range ch {
        fmt.Println(v)
    }
}
```

注意：目前Go语言中并没有提供一个不对通道进行读取操作就能判断通道是否被关闭的方法。不能简单的通过 `len(ch)` 操作来判断通道是否被关闭。

通道操作结果表				
状态 / 操作	nil	没值	有值	满
发送	阻塞	发送成功	发送成功	阻塞
接收	阻塞	阻塞	接收成功	接收成功
关闭	panic	关闭成功	关闭成功	关闭成功

3.7 select多路复用

select 语句具有以下`特点`。

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 会随机选择一个执行`。
对于没有 case 的 select 会一直阻塞，可用于阻塞 main 函数，防止退出。

4、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。

用下面的代码演示一个数据竞争的示例。

```go
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)
}
```

代码中会出现争夺共享数据而出现问题的情况

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。

4.1 互斥锁sync.Mutex

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。

```go
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。

4.2 读写锁sync.RWMutex

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。

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。

4.3 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：

```go
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。

4.4 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：

```go
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。

4.5 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。

```go
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` 。

```go
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()
}
```