

# Golang 并发编程

梁大光  
2020/12/16

## 目录

Golang 并发编程 .....	1
一，什么是并发 .....	2
1.1 并发的理解 .....	2
1.2 和并行的区别 .....	2
1.3 通过例子了解并发运行与普通串行运行的区别 .....	3
1.4 如何创建一个并发程序 .....	3
二，并发编程的作用与问题 .....	3
2.1 开启了多个并发之后，在主程序退出时，如何确保子程序已经执行完任务，主程序优雅的退出？ .....	3
2.2 资源的竞争 .....	4
2.3 线程间如何通讯 .....	8
select .....	11
三，通过并发编程实现一些小功能 .....	12
3.1 用 channel 来做一个简单的生产者和消费者模式 .....	12
3.2 其他并发原语 .....	13
四、Go 语言死锁、活锁和饥饿概述 .....	14
死锁 .....	14
活锁 .....	15
饥饿 .....	15
五、总结 .....	15
1.并发概述: .....	15
2.并发原语 .....	15

# 一，什么是并发

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 调度的最小单位。

## 1.1 并发的理解

并发 (concurrency): 同一时间段内，两个或则多个任务在执行，时间上有具有重叠性 (宏观上是同时，微观上任是顺序执行)。

## 1.2 并发和并行的区别

并行 (parallelism): 把每一个任务分配给每一个处理器独立完成。在同一时间点，任务一定是同时运行。

区别:

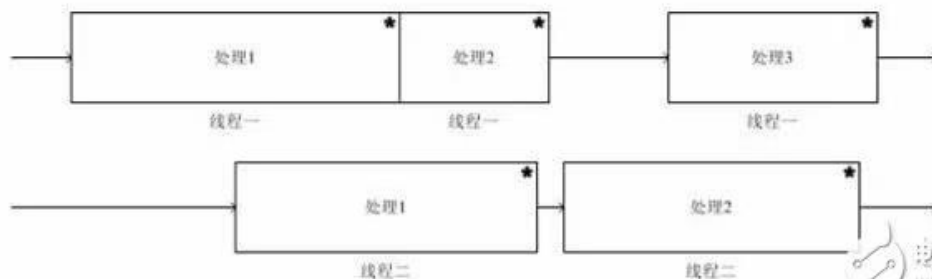
并行主要指，一个处理器同时处理多个任务，逻辑上是同时发生。

并发主要指，多个处理器同时处理多个任务，物理上是同时发生。

并发concurrent (单处理器环境)



并行parallel (多处理器环境)



## 1.3 通过例子了解并发运行与普通串行运行的区别

逐条执行示例: 串行调用 web 示例.go

并发执行示例: 并发调用 web 示例.go

## 1.4 如何创建一个并发程序

**goroutine** 是 Go 语言中的轻量级线程实现, 由 Go 运行时 (runtime) 管理。Go 程序会智能地将 goroutine 中的任务合理地分配给每个 CPU。

**使用 go 关键字创建 goroutine 时, 被调用函数的返回值会被忽略。**

### 1.4.1 为一个普通函数创建 goroutine 的写法如下

go 调用方法名

### 1.4.2 使用匿名函数创建 goroutine 的格式

```
go func( 参数列表 ){  
    函数体  
}( 调用参数列表 )
```

## 二, 并发编程的作用与问题

**作用: 尽可能的发挥操作系统多核的特性, 提升性能, 减少程序总运行时间**

2.1 开启了多个并发之后, 在主程序退出时, 如何确保子程序已经执行完任务, 主程序优雅的退出?

2.1.1 可以通过 Context 上下文 (不重点介绍)

2.1.2 通过 sync.WaitGroup (重点讲解)

2.1.3 通过 channel 返回 (后面会讲到)

Go 语言等待组 (sync.WaitGroup)

在 sync.WaitGroup (等待组) 类型中, 每个 sync.WaitGroup 值在内部维护着一个计数, 此计数的初始默认值为零

```

type WaitGroup struct {
    noCopy noCopy

    // ...

    statel [3]uint32
}

// state returns pointers to the state and sema fields stored within wg.statel.
func (wg *WaitGroup) state() (statep *uint64, semap *uint32) {...}

// ...

func (wg *WaitGroup) Add(delta int) {...}

// Done decrements the WaitGroup counter by one.
func (wg *WaitGroup) Done() {...}

// Wait blocks until the WaitGroup counter is zero.
func (wg *WaitGroup) Wait() {
    statep, semap := wg.state()
    // ...
}

```

等待组的方法

方法名	功能
(wg * WaitGroup) Add(delta int)	等待组的计数器 +1
(wg * WaitGroup) Done()	等待组的计数器 -1
(wg * WaitGroup) Wait()	当等待组计数器不等于 0 时阻塞直到变 0。

示例: waitGroup 请求远程服务实例.go

## 2.2 资源的竞争

先来看一个例子: 实现简单数据统计-没有加锁.go

如果统计的数据, 和我们预期的并不一样, 那是为什么呢?

有并发, 就有资源竞争, 如果两个或者多个 goroutine 在没有相互同步的情况下, 访问某个共享的资源, 比如同时对该资源进行读写时, 就会处于相互竞争的状态, 这就是并发中的资源竞争。

数据在多线程并发的环境下会存在安全问题的 3 个条件:

- 1) 多线程并发;
- 2) 多线程共享数据;
- 3) 共享数据有修改行为。

我们如何知道一个程序, 是否存在资源竞争, 导致线程不安全呢?

-race

Eg: go run -race XXXX ; go build -race XXXXX

```
E:\工作\优特云\内部后端分享\golang并发\示例\go run -race 实现简单数据统计-没有加锁.go
=====
WARNING: DATA RACE
Read at 0x0000010e2608 by goroutine 8:
    main.incCounter()
        E:/宸ヤ錄/浣樁搗淚?鍑囧儳錫鹿 錄嘒韩/golang奪蹂莢/紺袞緇/淪終幫綆 綈晨噉鋌 2坤璉?嫫G給鈞狎掌.go:41 +0x85

Previous write at 0x0000010e2608 by goroutine 7:
    main.incCounter()
```

那么我们需要通过什么手段，来确保数据的正确性呢？

## 2.2.1 锁住共享资源

### 2.2.1.1. atomic （原子性）

原子函数能够以很底层的加锁机制来同步访问整型变量和指针；

```
/* In particular, to decrement x, do AddUint32(&x, ^uint32(0)).
func AddUint32(addr *uint32, delta uint32) (new uint32)

// AddInt64 atomically adds delta to *addr and returns the new value.
func AddInt64(addr *int64, delta int64) (new int64)

// AddUint64 atomically adds delta to *addr and returns the new value.
// To subtract a signed positive constant value c from x, do AddUint64(&x, ^uint64(c-1)).
// In particular, to decrement x, do AddUint64(&x, ^uint64(0)).
func AddUint64(addr *uint64, delta uint64) (new uint64)

// AddUintptr atomically adds delta to *addr and returns the new value.
func AddUintptr(addr *uintptr, delta uintptr) (new uintptr)

// LoadInt32 atomically loads *addr.
func LoadInt32(addr *int32) (val int32)

// LoadInt64 atomically loads *addr.
func LoadInt64(addr *int64) (val int64)

// LoadUint32 atomically loads *addr.
func LoadUint32(addr *uint32) (val uint32)

// LoadUint64 atomically loads *addr.
func LoadUint64(addr *uint64) (val uint64)

// LoadUintptr atomically loads *addr.
func LoadUintptr(addr *uintptr) (val uintptr)
```

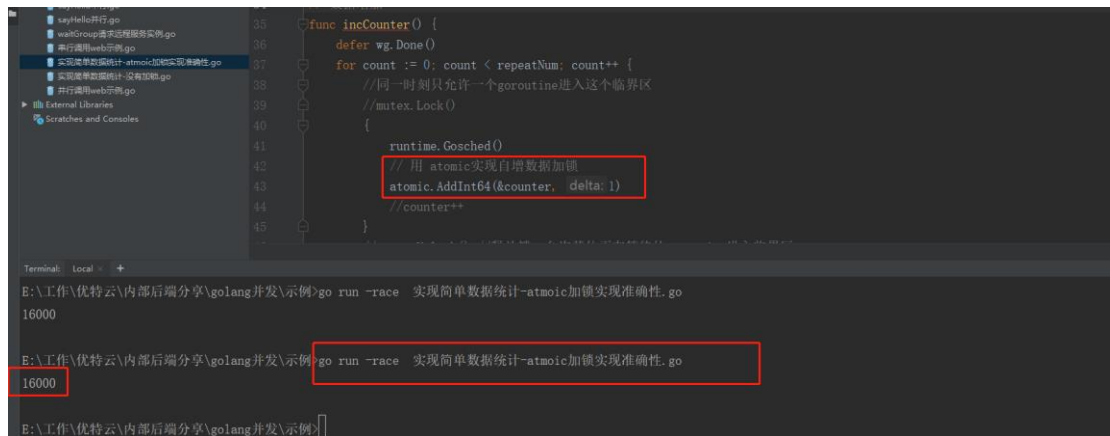
AddIntXX(): 函数会同步整型值的加法

LoadIntXX():安全地读

StoreInt(): 安全写

用 atomic 包实现对上个示例的数据统计加锁

**见示例：实现简单数据统计-atomic 加锁实现准确性.go**



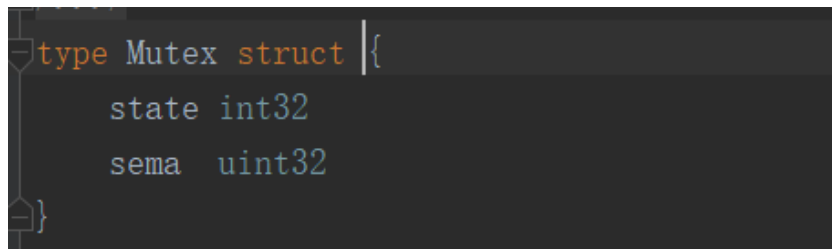
```
35 func incCounter() {
36     defer wg.Done()
37     for count := 0; count < repeatNum; count++ {
38         //同一时刻只允许一个goroutine进入这个临界区
39         //mutex.Lock()
40         {
41             runtime.Gosched()
42             // 用 atomic实现自增数据加锁
43             atomic.AddInt64(&counter, delta: 1)
44             //counter++
45         }
46     }
47 }
```

Terminal: Local

```
E:\工作\优特云\内部后端分享\golang开发\示例>go run -race 实现简单数据统计-atmoic加锁实现准确性.go
16000
E:\工作\优特云\内部后端分享\golang开发\示例>go run -race 实现简单数据统计-atmoic加锁实现准确性.go
16000
E:\工作\优特云\内部后端分享\golang开发\示例>
```

### 2.2.1.2 sync (互斥锁)

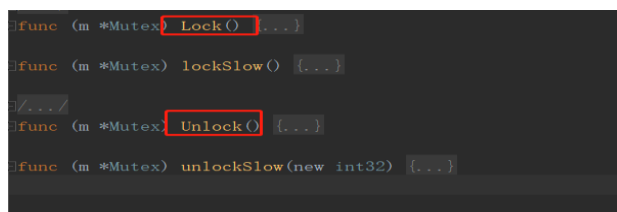
Mutex 是最简单的一种锁类型, 同时也比较暴力, 当一个 goroutine 获得了 Mutex 后, 其他 goroutine 就只能乖乖等到这个 goroutine 释放该 Mutex。先看下他的结构:



```
type Mutex struct {
    state int32
    sema  uint32
}
```

state=0 时是未上锁, state=1 时是锁定状态。

2 个方法:



```
func (m *Mutex) Lock() {...}
func (m *Mutex) lockSlow() {...}
// ...
func (m *Mutex) Unlock() {...}
func (m *Mutex) unlockSlow(new int32) {...}
```

通过 Lock 函数, 用于在代码上创建一个临界区, 保证同一时间只有一个 goroutine 可以执行这个临界代码, 当代码结束时, 可以通过 Unlock 方法来释放锁。当调用 runtime.Gosched 函数强制将当前 goroutine 退出当前线程后, 调度器会再次分配这个 goroutine 继续运行。

见示例: [实现简单数据统计-mutex 互斥锁实现准确性.go](#)

```
32 // 数据增加
33
34 func incCounter() {
35     defer wg.Done()
36     for count := 0; count < repeatNum; count++ {
37         // 同一时刻只允许一个goroutine进入这个临界区
38         mutex.Lock() 锁临界区
39         {
40             runtime.Gosched()
41             counter++
42         }
43         mutex.Unlock() // 释放锁，允许其他正在等待的goroutine进入临界区
44     }
45 }
46
47 incCounter()
48
49 done: 16000
50
51 E:\工作\优特云\内部后端分享\golang开发\示例>go run -race 实现简单数据统计-mutex互斥锁实现准确性.go
52 done: 16000
53
54 E:\工作\优特云\内部后端分享\golang开发\示例>
```

小思考？

1.如果在一个地方 Lock(), 在另一个地方不 Lock()而是直接修改或访问共享数据，是否可行？

**见示例：mutex 互斥锁实现简单数据统计&一个地方 Lock&在另一个地方不**

## Lock.go

2. 刚才提到了，一个线程获得互斥锁之后，其他线程要获取锁，就只能乖乖的等待，如果我们将 CPU 的调度权主动让出给其他线程，那么当前的线程锁会不会主动被释放？

### 2.2.1.3sync.RWMutex（读写互斥锁）

RWMutex，是经典的单写多读模型。在读锁占用的情况下，会阻止写，但不阻止读，也就是多个 goroutine 可同时获取读锁，（调用 RLock() 方法；而写锁（调用 Lock() 方法）会阻止任何其他 goroutine（无论读和写）进来，整个锁相当于由该 goroutine 独占。

看下读写锁的结构：

```
type RWMutex struct {
    w      Mutex // held if there are pending writers
    writerSem uint32 // semaphore for writers to wait for completing readers
    readerSem uint32 // semaphore for readers to wait for completing writers
    readerCount int32 // number of pending readers
    readerWait int32 // number of departing readers
}

type RWMutex struct {
    w      Mutex // held if there are pending writers
    writerSem uint32 // 写锁需要等待读锁释放的信号量
    readerSem uint32 // 读锁需要等待写锁释放的信号量
    readerCount int32 // 读锁后面挂起了多少个写锁申请
    readerWait int32 // 已释放了多少个读锁
}
```

可导出的方法：

```

.../
type RWMutex struct {...}

const rwmutexMaxReaders = 1 << 30

.../
func (rw *RWMutex) RLock() {...}

.../
func (rw *RWMutex) RUnlock() {...}

func (rw *RWMutex) rUnlockSlow(r int32) {...}

.../
func (rw *RWMutex) Lock() {...}

.../
func (rw *RWMutex) Unlock() {...}

.../
func (rw *RWMutex) RLocker() Locker {...}

```

### 见示例：RWMutex 实现单写多读.go

观察读取锁的时候，会不会因为另一个 goroutine 获得锁，而另一个 goroutine 需要等待；RWMutex 的特点：

可以同时申请多个读锁

有读锁时申请写锁将阻塞，有写锁时申请读锁将阻塞

只要有写锁，后续申请读锁和写锁都将阻塞

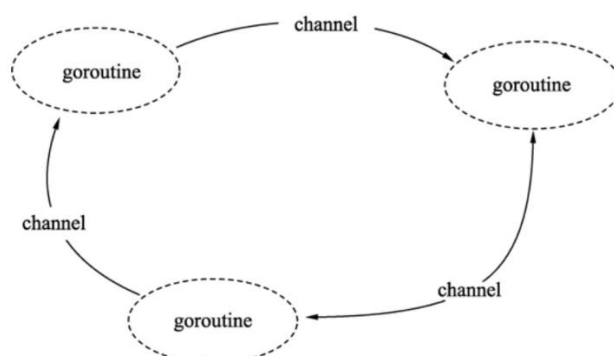
## 2.3 线程间如何通讯

上面在创建 goroutine 的时候，提到过；通过 go 关键词启动的线程，会忽略返回结果，那么需要不同线程间返回结果，该怎么处理？

### 2.3.1 goroutine 之间通信的管道-channel

Go 语言提倡使用通信的方法代替共享内存，当一个资源需要在 goroutine 之间共享时，通道在 goroutine 之间架起了一个管道，并提供了**确保同步交换数据的机制（在任何时候，同时只能有一个 goroutine 访问通道进行发送和获取数据）**。

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。



图：goroutine 与 channel 的通信



### 2.3.1.1 channel 的定义

```
chan T           // 可以接收和发送类型为 T 的数据
chan<- float64   // 只可以用来发送 float64 类型的数据
<-chan int       // 只可以用来接收 int 类型的数据
```

<-总是优先和最左边的类型结合。

```
chan<- chan int   // 等价 chan<- (chan int)
chan<- <-chan int // 等价 chan<- (<-chan int)
<-chan <-chan int // 等价 <-chan (<-chan int)
chan (<-chan int)
```

### 2.3.1.2 接受数据

#### 1) 阻塞接收数据

```
data := <-ch
```

#### 2) 非阻塞接收数据

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

这个用法与 map 中的按键获取 value 的过程比较类似，只需要看第二个 bool 返回值即可，如果返回值是 false 则表示 ch 已经被关闭。

#### 3) 接收任意数据，忽略接收的数据

```
<-ch
```

该方式会阻塞接收数据后，忽略从通道返回的数据

#### 4) 循环接收（常用的方式）

```
for data := range ch {
}
```

思考题：

1.通过并发的方式，将 5 组数据的总和加起来？

见示例: channel 实现并发计算 5 组数据.go

通过示例，观察返回结果。

2.既然说在同一时刻，只能有一个 goroutine 获取到通道，那么之前的锁程序，可不可以通过通道的方式来实现？

见示例: 实现简单数据统计-channel.go

通道的关闭:

Close(ch)

思考题?

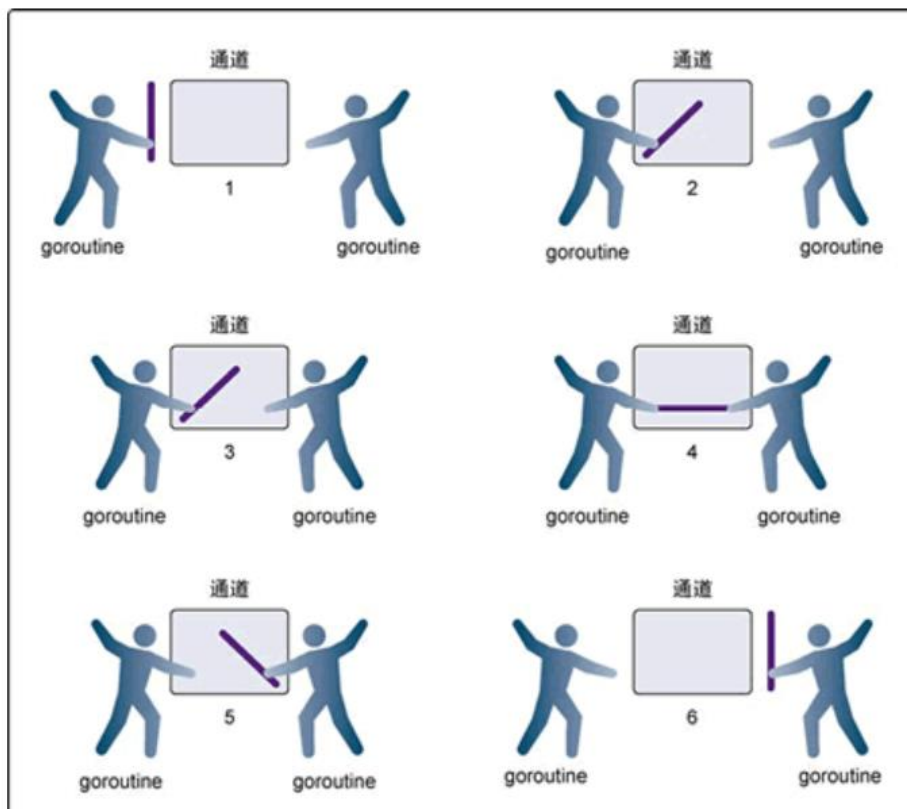
1.如何判断一个 channel 是否已经被关闭?

2.3.2 单向通道 (留给大家下去思考。。。。, 后面有个类似的列子)

2.3.3 有缓冲通道(buffered channel)

### 1). 无缓冲通道

无缓冲的通道 (unbuffered channel) 是指在接收前没有能力保存任何值的通道。这种类型的通道要求发送 goroutine 和接收 goroutine 同时准备好, 才能完成发送和接收操作。下图展示两个 goroutine 如何利用无缓冲的通道来共享一个值。



思考?

1.对于无缓冲通道, 我们可不可以在同一个 goroutine 中自己发送数据, 自己接受数据呢?

## 2). 有缓冲通道

有缓冲的通道（buffered channel）是一种通道在被接收前有能力存储一个或者多个值的通道。

思考？：

生活中，哪些场景可以类似于有缓冲通道？

### 创建带缓冲通道

```
Ch := make(chan type, length int)
```

length: 缓冲通道的大小

有缓冲通道示例: [channel-有缓冲通道示例.go](#)

有缓冲通道阻塞条件：

- 带缓冲通道被填满时，尝试再次发送数据时发生阻塞；
- 带缓冲通道为空时，尝试接收数据时发生阻塞；

## Goroutine 的超时机制

如果我们请求某个方法地址的时候，所用时间超过了我们设定的限制时间，该如何终止该请求继续处理？

示例：[channel 超时机制 select.go](#)

## select

select 是 Go 中的一个控制结构，类似于用于通信的 switch 语句；

但是 select 的 case 必须是一个 IO 操作，要么是发送要么是接收。select 随机执行一个可运行的 case。如果没有 case 可运行，它将阻塞，直到有 case 可运行。

## 结构

```
select{  
    case 操作 1（必须是一个通信操作）：  
        响应操作 1  
    case 操作 2（必须是一个通信操作）：  
        响应操作 2  
    default:  
        没有操作情况
```

```
}
```

思考？

1. 如果我们同时需要接受或者对多个通道进行数据的传递和写入，我们怎实现？

## Go 语言通道的多路复用

多路复用通常表示在一个信道上传输多路信号或数据流的过程和技术。

示例 1: channel 随机发送数据.go

示例 2: channel 多路复用-接收多个通道的值.go

## 三，通过并发编程实现一些小功能

### 3.1 用 channel 来做一个简单的生产者和消费者模式

示例: channel 实现简单的生产者和消费者.go

## 3.2 其他并发原语



### 1) sync.Map

示例 1: 并发环境下使用 map-不做处理.go

示例 2: 并发环境下使用 map-加以处理.go

### 2) Cond

Cond 条件变量 Cond 实现一个条件变量，即等待或宣布事件发生的 goroutines 的会合点，它会保存一个通知列表。

Cond 需要指定一个 Locker，通常是一个 \*Mutex 或 \*RWMutex

3 个核心方法:

```

+.../
+type Cond struct {...}

// NewCond returns a new Cond with Locker l.
+func NewCond(l Locker) *Cond {...}

+.../
+func (c *Cond) Wait() {...}

+.../
+func (c *Cond) Signal() {...}

+.../
+func (c *Cond) Broadcast() {...}

```

**Wait:** 必须获取该锁之后才能调用 Wait()方法, Wait 方法在调用时会释放底层锁 Locker, 并且将当前 goroutine 挂起, 直到另一个 goroutine 执行 Signal 或者 Broadcast, 该 goroutine 才有机会重新唤醒, 并尝试获取 Locker, 完成后续逻辑。

**Signal:** 释放一个

**Broadcast:** 释放所有

示例: Cond 示例.go

### 3) Pool

### 4) Once

```

// Once is an object that will perform exactly one action.
+type Once struct {...}

+.../
+func (o *Once) Do(f func()) {...}
|
+func (o *Once) doSlow(f func()) {...}

```

示例: Once 示例.go

## 四、Go 语言死锁、活锁和饥饿概述

### 死锁

死锁是指两个或两个以上的进程（或线程）在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产

生了死锁，这些永远在互相等待的进程称为死锁进程。

## 活锁

活锁是另一种形式的活跃性问题，该问题尽管不会阻塞线程，但也不能继续执行，因为线程将不断重复同样的操作，而且总会失败。

## 饥饿

饥饿是指一个可运行的进程尽管能继续执行，但被调度器无限期地忽视，而不能被调度执行的情况。

与死锁不同的是，饥饿锁在一段时间内，优先级低的线程最终还是会执行的，比如高优先级的线程执行完之后释放了资源。

活锁与饥饿是无关的，因为在活锁中，所有并发进程都是相同的，并且没有完成工作。更广泛地说，饥饿通常意味着有一个或多个贪婪的并发进程，它们不公平地阻止一个或多个并发进程，以尽可能有效地完成工作，或者阻止全部并发进程。

## 五、总结

### 1.并发概述:

- a.并发，并行；
- b. golang 如何创建并发线程 (go)

### 2.并发原语

Contex  
Runtime  
Sync.WaitGroup  
Atomic  
Sync.Mutex  
Sync.RWMutex  
Select  
Channel(无缓冲，有缓冲)  
Sync.Map  
Sync.Cond  
Sync.Pool  
Sync.Once

