



如何进行并发编程



张华

64174234@qq.com

内容

1.1 什么是进程

1.2 什么是线程

1.3 进程和线程的关系

1.4 CPU调度中会出现的那些问题

1.5 什么是并发

1.6 如何建立Goroutine——Go的轻量级线程

1.7 如何实现通信——channel

1.8 如何处理通信中的死锁问题

1.9 如何实现多核并行化

1.10 总结

1.11 思考

1.1 什么进程

■ 进程是正在执行的程序



Process Explorer - Sysinternals: www.sysinternals.com [THINKPAD\think]

Process	PID	CPU	Description	Company Name
chrome.exe	7696	< 0.01	Google Chrome	Google Inc.
chrome.exe	9484		Google Chrome	Google Inc.
chrome.exe	12524	0.49	Google Chrome	Google Inc.
cmd.exe	4104		Windows 命令处理程序	Microsoft Corporation
test1.exe	10804	1.77		
POWERPNT.EXE	5884	1.46	Microsoft PowerPoint	Microsoft Corporation
SogouSmartInfo.exe	13308		搜狗拼音输入法 搜索候选	Sogou.com Inc.
YoudaoDict.exe	11084	0.61	有道词典	网易公司
YoudaoIE.exe	10200	0.08	有道词典IE辅助进程	网易公司
YoudaoDictHelper.exe	1220			
WordBook.exe	9868	0.05	有道单词本	网易公司
YoudaoEH.exe	9212	0.11	有道词典64位取词辅助程序	网易公司
YoudaoWSH.exe	9748	< 0.01		
Acrobat.exe	6052	0.60	Adobe Acrobat 9.3	Adobe Systems Incor...
AdobeARM.exe	7740		Adobe Reader and Acro...	Adobe Systems Incor...
KwMusic.exe	11200	4.40		
IESandbox.exe	7700	13.60		

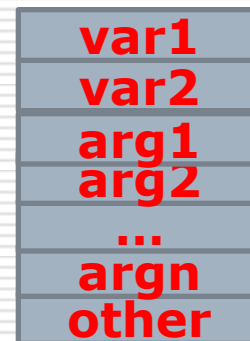
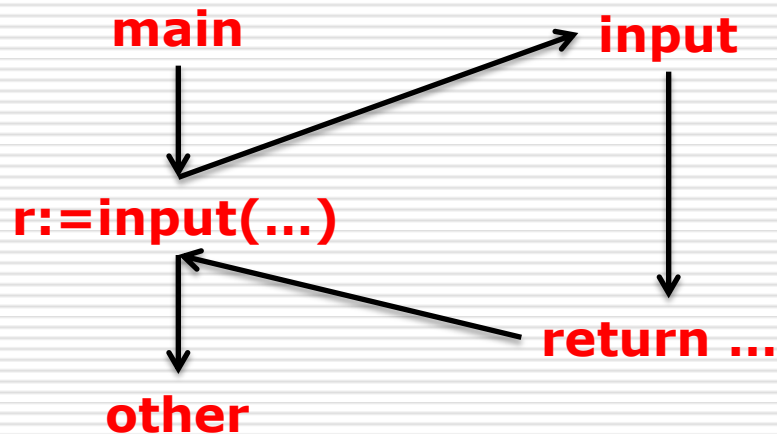
CPU Usage: 75.45% Commit Charge: 32.14% Processes: 153

进程查看

1.1什么是进程(续)

- 进程由操作系统进行调度,拥有自己独立的上下文环境(栈空间、堆空间),不和其他进程共享
 - 栈空间由编译器自动分配释放,在函数调用时,第一个进栈的是主函数中函数调用后的下一条指令的地址,然后是函数的各个参数,在大多数编译器中,参数是从右往左入栈的,当本次函数调用结束后,局部变量先出栈,然后是参数,最后栈顶指针指向最开始存的地址,即主函数中的下一条指令。
 - 堆空间由程序员分配释放(如C中的malloc, free)

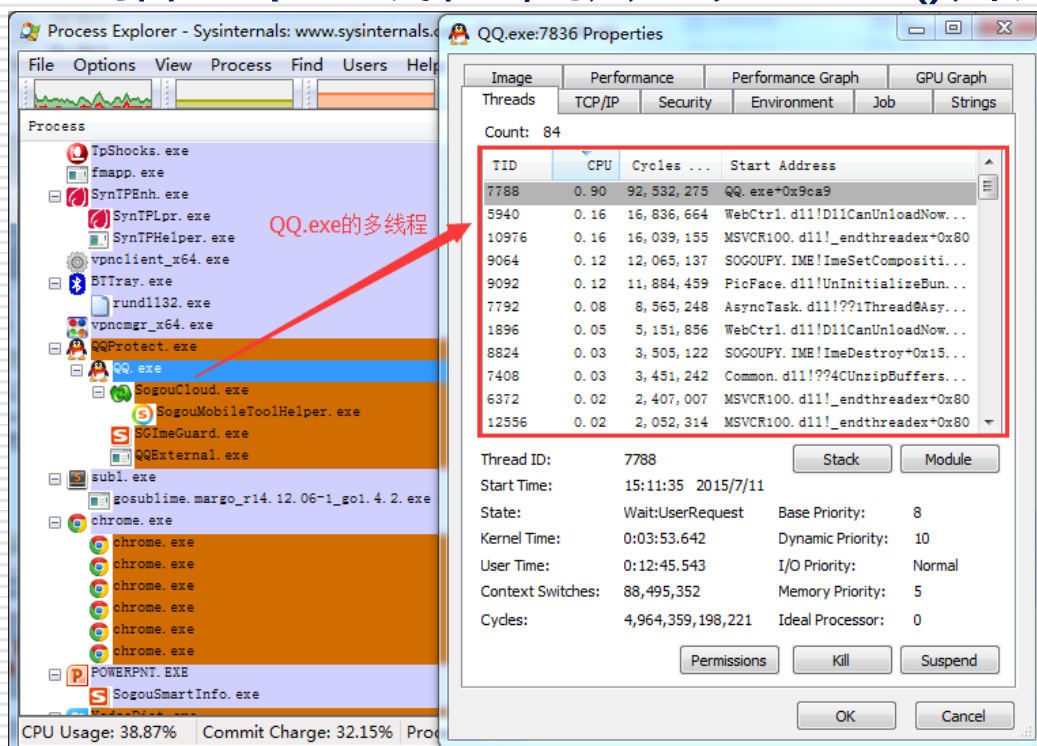
```
package main
import "fmt"
func main() {
    ...
    → r:=input(a1,a2,...an)
    other
}
func input(arg1,arg2,...argn) (int){
    var1
    var2
    ...
    return ...
}
```



执行name函数时的栈空间

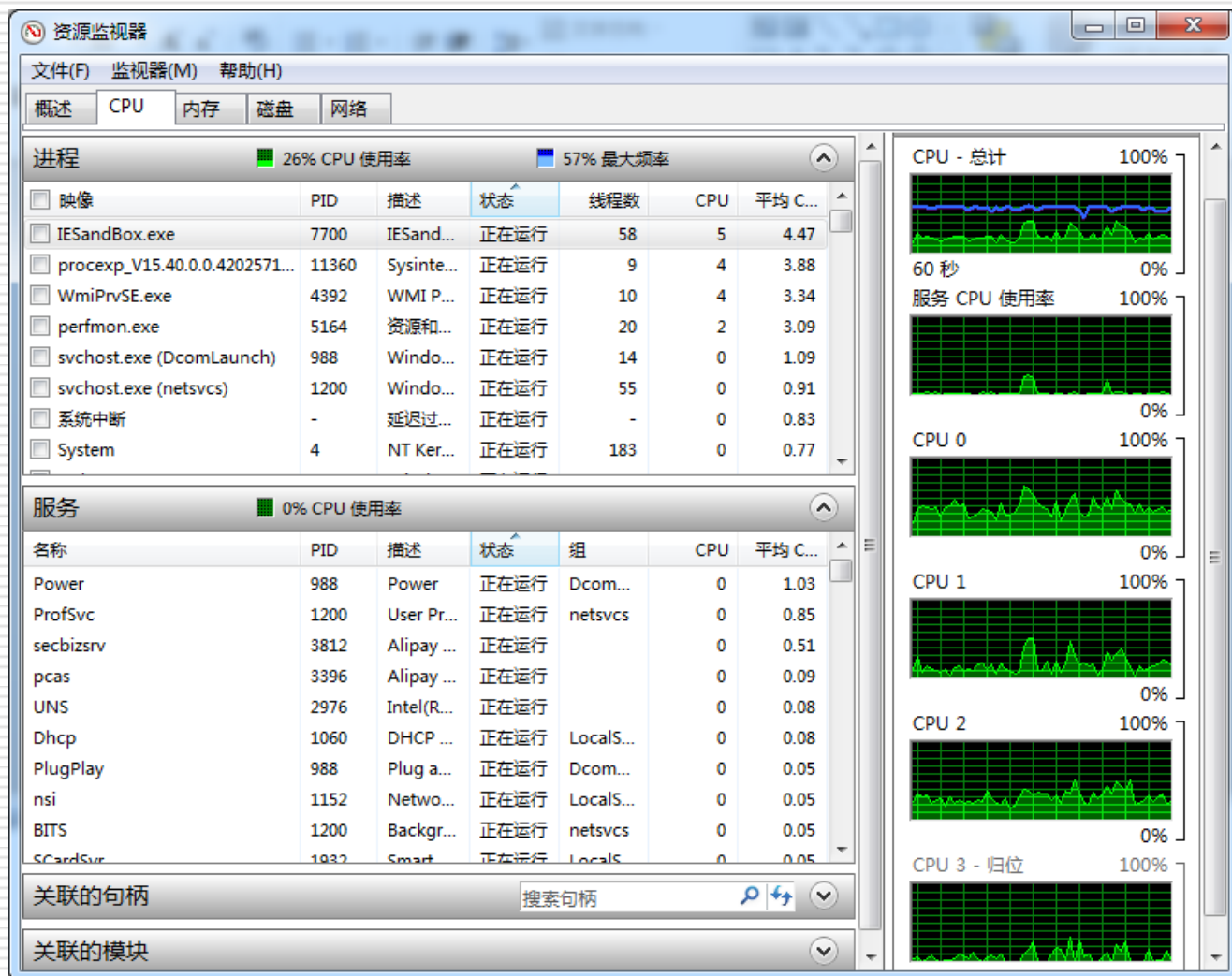
1.2什么是线程

- ❑ 线程是为了提高操系统的并发性而引入的，拥有自己独立的栈和共享的堆。
- 进程是系统进行资源分配和调度的一个独立单位，线程是CPU分配和调度的基本单位。
- 线程是进程的一个实体，一个进程可以有多个线程，至少包括一个主线程，就是以main()开始的函数



1.2什么是线程(续)

- 四个CPU处理，表示同一时刻可以处理四个线程，能够同时向CPU发送四个指令



1.3进程和线程的关系

- ❑ 进程就是整条道路
- ❑ 线程就是用白虚线分隔开来的各个车道
- ❑ 线程必须依赖于进程，如同车道离不开道路
- ❑ 多线程：线程之间可以并发执行，如同各个车道你走你的，我走我的



1.4CPU调度中会出现的那些问题

□ 同步

- 多个线程并发执行时的一种通信规则，以保证有依赖关系的线程有序执行。如同十字路口内，某些车道在交通灯亮时，禁止前行或转向，必须等其他车道的车辆通行完毕。前一个线程的输出作为后一个线程的输入，当前一个线程没有输出时，第二个必须等待。同步需要等待，协调运行。



□ 异步

- 异步就是线程间彼此独立，异步是让调用方法的主线程不需要同步等待另一个线程的完成，从而让主线程可以干其他的事情。多个线程的异步执行实现就是并发

同步 (Synchronous)



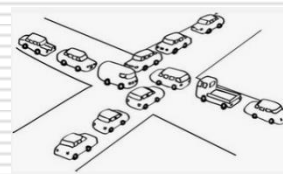
异步 (Asynchronous)



1.4CPU调度中会出现的那些问题(续)

□ 死锁

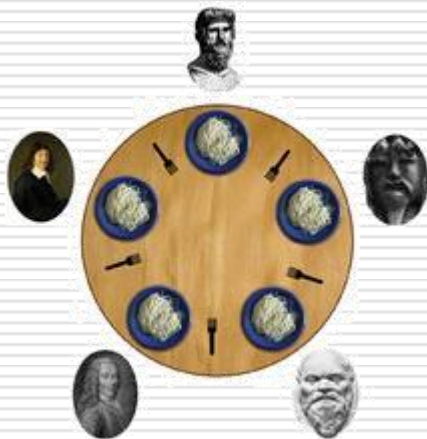
- 死锁是同步的，可以认为是两个线程同时请求占有对方资源，如同十字路口信号灯出现问题一样(同步控制器)，当都是绿灯的时候，大家就都堵在了路中央，谁也无法通过。
- 死锁的四种情况
 - 互相排斥：指的是资源在任意时刻只能由一个线程使用。如果此时还有其它任务请求该资源，则请求者只能等待，直至占有资源的线程释放资源。
 - 一个线程占有共享资源时，其他线程不能访问
 - 循环等待：A等待B，B等待C，C等待A
 - 部分分配：A和B同时需要打印机和访问一个文件，此时A得到了文件资源，B得到了打印机资源，但都不能全部得到
 - 不可抢占：指的是当一线程拥有某种资源时，除非它主动释放它，否则无法让该线程失去该资源的拥有权。



1.4CPU调度中会出现的那些问题(续)

□ 饥饿

- 饥饿是异步的，是一个线程在无限的等待另外两个或多个线程占有的但不会往外释放的资源
- 三个进程P1,P2,P3,每个进程都周期性的访问资源，P1拥有资源，P2和P3都被延迟，等待这个资源，当P1退出时，P2和P3都被允许访问R，在P3访问完成R之前P1又需要访问R，这样将访问权轮流授予P1和P3，P2就会因无限制的等待而处于饥饿状态



1.4CPU调度中会出现的那些问题(续)

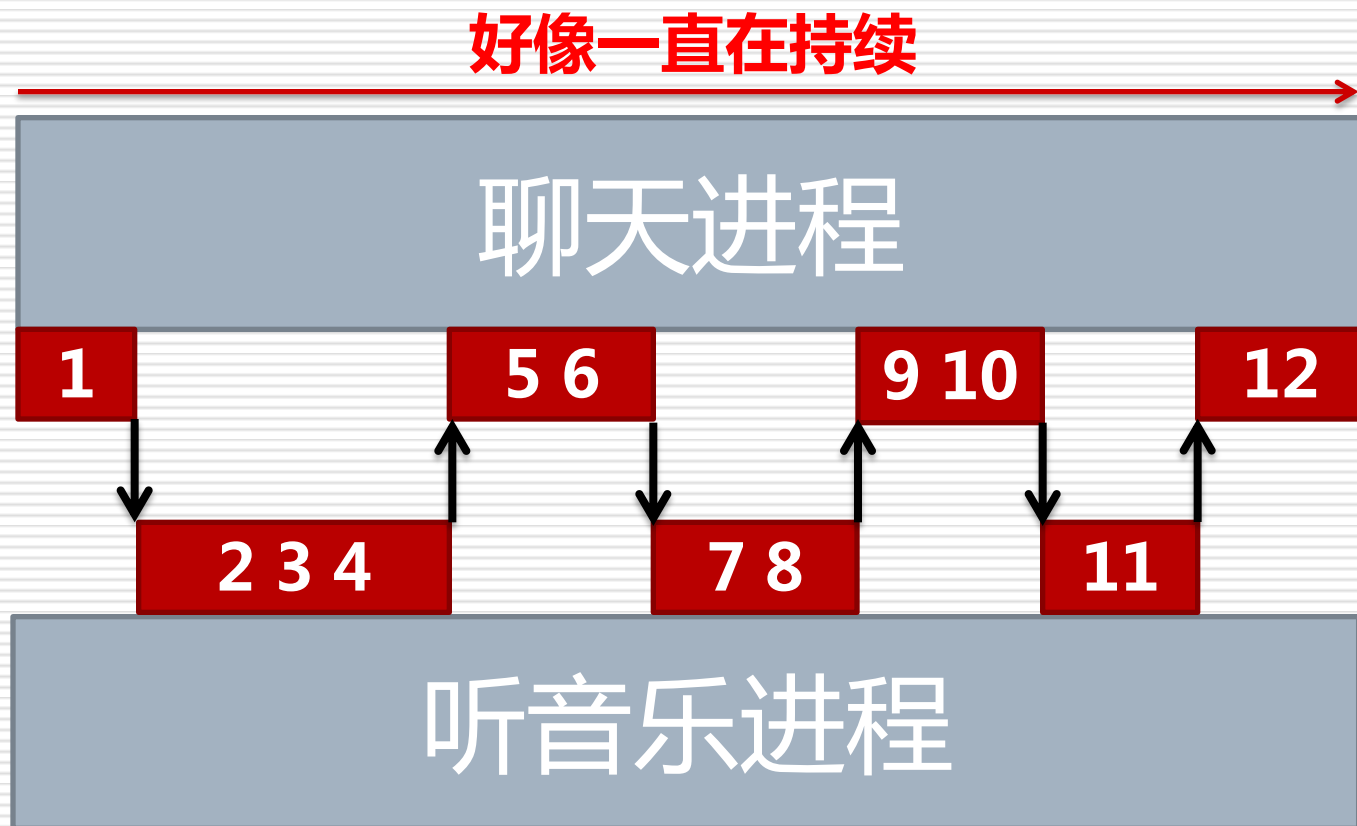
□ 进程状态

- **运行**：当一个进程在处理机上运行时，则称该进程处于运行状态。处于此状态的进程的数目小于等于处理器的数目，对于单处理机系统，处于运行状态的进程只有一个。
- **就绪**：当一个进程获得了除处理机以外的一切所需资源，一旦得到处理机即可运行，则称此进程处于就绪状态。就绪进程可以按多个优先级来划分队列。例如，当一个进程由于时间片用完而进入就绪状态时，排入低优先级队列；当进程由I / O操作完成而进入就绪状态时，排入高优先级队列。
- **阻塞**：也称为等待或睡眠状态，一个进程正在等待某一事件发生（例如请求I/O并等待I/O完成等）而暂时停止运行，这时即使把处理机分配给进程也无法运行，故称该进程处于阻塞状态。



1.5什么是并发

- 我们为什么既能听音乐又能和朋友聊天呢？
- 因为CPU在飞快的切换(时间片快速轮转)



1.5什么是并发(续)

□ 分工与合作问题

- 时间片解决了分工问题，却没有解决合作问题
- 分工：大部分情况下分工就是顺序执行，一个任务可以分成几步，前一步是后一步的输入
- 合作：合作就是模块间的通讯，如果是跨计算机，合作就是网络通讯，如果是再同一台主机，合作可以是网络协议，可以是进程间通讯，也可以是线程间共享数据，也可以是基于消息的事件触发。目的是让工作完成的更快，使同一件事的软件模块更加独立和简单，以至于能够并发或并行执行

1.5什么是并发(续)

□ 进程→线程(轻量级进程)

- Unix发明之初没有线程这个概念，分工合作通过进程间的通讯来完成，如果通讯是基于socket，那么进程在本机内的通讯和进程在网络间的通讯无差别。进程间不共享数据，交换信息只能通过socket互相发送。
- 后来为了提高本机内的进程间通讯效率发明了线程，被称为轻量级进程
 - 创建线程占用的内存更少。
 - 线程间传输数据时，因为是在同一个进程的地址空间，可以通过引用来访问共享的数据块，节省了调用内核发送/接收数据的开销
- 线程的最大优势，是可以充分利用单台服务器的多核cpu计算资源，并发地处理任务。虽然使用进程能达到同样的目的，但使用了线程，速度可以提高一个数量级，由于支持线程带来的性能提升是如此明显，大约在10年前，继Windows之后，Linux内核支持了多线程。

1.5什么是并发(续)

□ 线程的不足

- 即便被称为“轻量级进程”，线程消耗的内存资源仍然不少，通常情况下，操作系统创建一个线程需要消耗1Mb的内存。
- 设想我们开发一个有大量用户并发访问的网站，我们为每个访问请求创建一个线程（这样编写软件是有理由的，提供了较好的并发性，编写代码简单），当并发访问量急剧上升时，就会把系统内存耗尽。
- 同时，创建过多的线程对整体性能提高也没有帮助，同时执行的线程数受cpu内核数目的限制，比如服务器的cpu是16核，那么最多可有16个线程同时（并发）执行，其它的线程也只能等待下一个cpu时间片。

1.5什么是并发(续)

□ 线程→轻量级线程

- 如何用有限的线程，来创建无数个并发任务？轻量级线程因此而生
- 它的内存占用比线程更少，它可以在一个线程内分“时间片”，在一个操作系统的线程上，创建多个“轻量级线程”，来实现多任务切换，而只占用一个线程的内存
- 进一步，可以创建上万个“轻量级线程”，共享一个线程池，线程池里面的线程数目只需要和cpu核数相当，就可以充分利用计算资源，而上万个“轻量级线程”在线程池内并发执行，既保证了并发性，又严格控制了整体资源的消耗。
- 假设有一支持5万在线玩家的单台服务器,则可创建5万个“轻量级线程”来服务每一个玩家,实际上却只要消耗操作系统30个线程,“轻量级线程”的代码编写一定是简单的,因为这一段代码只需要服务一个玩家,功能非常单一。

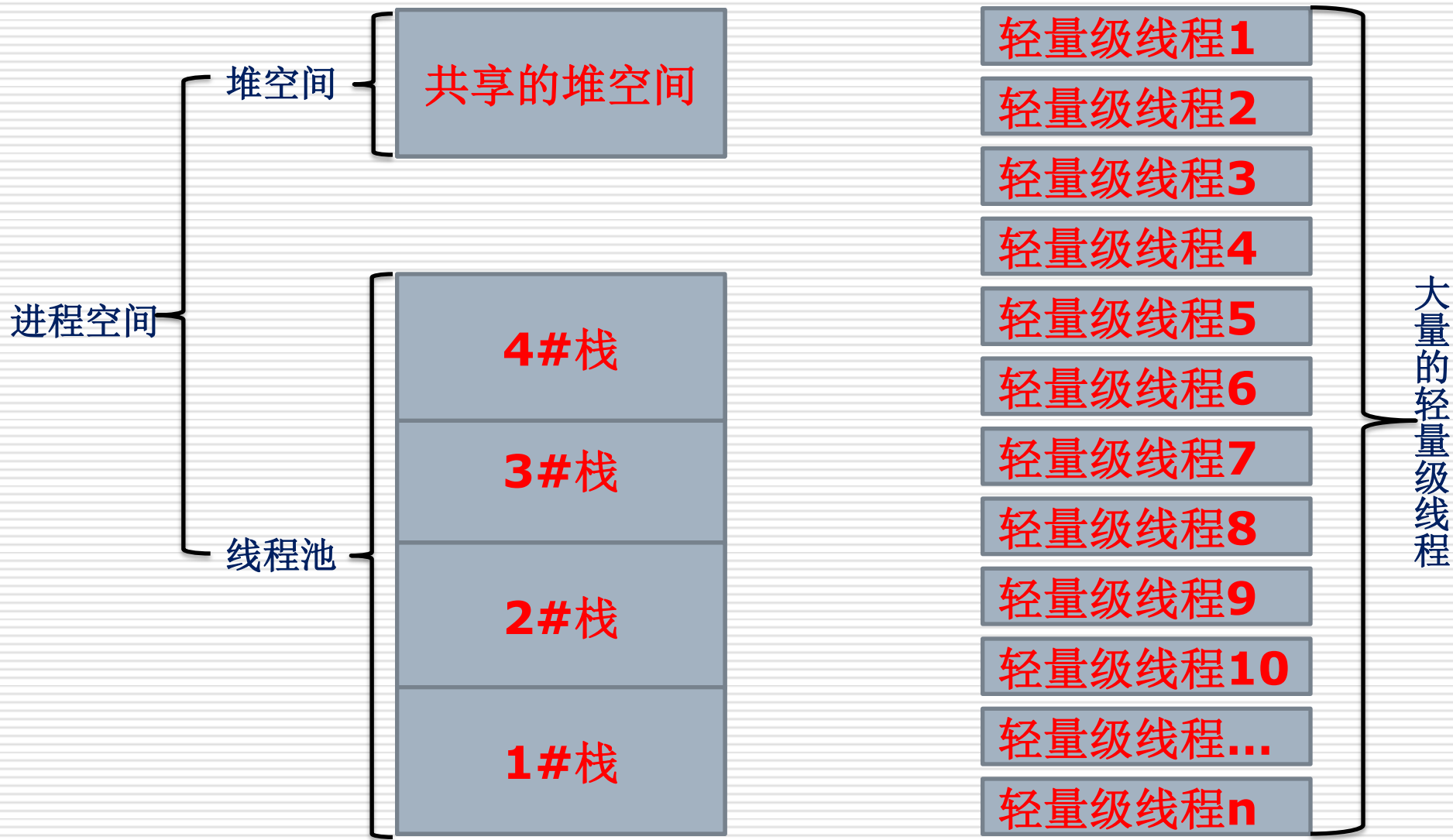
1.5什么是并发(续)

□ 并发编程

- 其实时间片，是由操作系统来分配，只能分配给进程和线程，所谓线程内的“时间片”是不存在的，那么“轻量级线程”如何实现在一个线程内的交替执行，这就是并发编程。
- 并发编程是基于事件触发的，简单来说，是由消息触发“轻量级线程”的一段代码执行，执行完毕后，“轻量级线程”交出线程控制权，进入等待状态，等待下一个消息触发，这时线程可以去服务另一个等待执行的“轻量级线程”。有一个全局的线程，是用来监视所有的消息，来决定哪个“轻量级线程”被触发执行。

1.5什么是并发(续)

□ 4核CPU下的并发效果示意



1.5什么是并发(续)

- 现在有哪些编程语言和框架支持“轻量级线程”？
 - Erlang——纯函数式语言
 - Go语言——混合式(面向对象、函数式)
 - F# with MailboxProcessor——纯函数式语言
 - Scala with Akka——纯函数式语言

1.6如何建立Goroutine——Go的轻量级线程

- 在一个函数前加上go关键字就可以创建一个Goroutine并调用该函数。当该函数执行结束，Goroutine也随之自动退出。

```
1 package main
```

```
2
```

```
3 import "fmt"
```

```
4
```

```
5 func main() {
```

```
6     go fmt.Println("Go Goroutine!")
```

```
7 }
```

运行时，系统会并发地执行go函数。而go语句之后没有任何语句，main函数至此执行完毕这样意味着go程序结束，而那个并发go函数还没来得及执行，即封装这个go函数的这个Goroutine还没来得及被调度并运行。所以没有输出结果。

那么如何才能让go函数执行呢？

```
[ `go run go2.go` | done: 1.2570719s ]
```

```
[ D:/go-dev/src/ ] #
```

1.6如何建立Goroutine——Go的轻量级线程

- Go语言为我们提供了多种手段，最简陋的是使用time包中的Sleep函数

```
1 package main
2
3 import "fmt"
4 import "time"
5
6 func main() {
7     go fmt.Println("Go Goroutine!")
8     time.Sleep(time.Millisecond)
9 }
```

time.Sleep的作用是让调用它的Goroutine暂停（进入Gwaiting状态）一段时间，这里main函数所在的Goroutine暂停了1毫秒。理想情况下，运行该源码会输出GO Goroutine！。但，请注意，情况不总是这样。调度器的实时调度是我们无法控制的，所以这个Sleep手段是不保险的。那什么是更保险的方式呢？

```
[ `go run go3.go` | done: 978.0559ms ]
Go Goroutine!
```

1.6如何建立Goroutine——Go的轻量级线程

- 用runtime.Gosched替换Sleep是一种保险的手段。Gosched作用是让其他Goroutine有机会被运行

```
1 package main
2
3 import "fmt"
4 import "runtime"
5
6 func main() {
7     go fmt.Println("Go Goroutine!")
8     runtime.Gosched()
9 }
```

```
[ `go run go4.go` | done: 917.0525ms ]
Go Goroutine!
```


1.6如何建立Goroutine——Go的轻量级线程

- 实际情况往往更复杂，那时runtime.Gosched就会变得不适用

```
package main
import "fmt"
import "runtime"
func main() {
```

```
    names := []string{"Eric", "Harry", "Robert", "Jim", "Mark"}
```

```
    for _, name := range names {
```

```
        go func() {
```

```
            fmt.Printf("Hello,%s.\n", name)
```

```
        }()
```

```
    }
```

```
    runtime.Gosched()
```

```
}
```

```
D:\go-dev\src>go run go7.go
```

```
Hello,Mark.
```

```
Hello,Mark.
```

```
Hello,Mark.
```

```
Hello,Mark.
```

```
Hello,Mark.
```

总之不要对go的执行时机做任何假设，除非你有事实为证来保证。

迭代了5次，最后迭代后name的值为Mark，这里被执行的5个go函数，name的值都是mark。这是因为它们都是在for语句被执行完毕后才被执行的，此时name=Mark的原因之一是因为for语句简单，执行快。即使for复杂，也可能发生这种情况。

go后面是匿名函数的调用表达式，不要忘了最后的那对小括号，代表了对函数的调用行为这就是函数闭包(closure):保证了在这类函数中被引用的变量在函数结束之前不会被释放

```
go func(){
```

```
...
}()
```

1.6如何建立Goroutine——Go的轻量级线程

□ 让5个go函数在每次迭代完成之前执行完毕

```
package main
import "fmt"
import "runtime"
func main() {
    names := []string{"Eric", "Harry", "Robert", "Jim", "Mark"}
    for _, name := range names {
        go func() {
            fmt.Printf("Hello,%s.\n", name)
        }()
        runtime.Gosched()
    }
}
```

D:\go-dev\src>go run go8.go
Hello, Eric.
Hello, Harry.
Hello, Robert.
Hello, Jim.
Hello, Mark.

这个思路看上去有点画蛇添足了，因为不用并发，去掉go 和runtime.Gosched()就可以达到目的，而且会很简单得多。但这样实在算不上“同时问候”这几个人了（并发）

1.6如何建立Goroutine——Go的轻量级线程

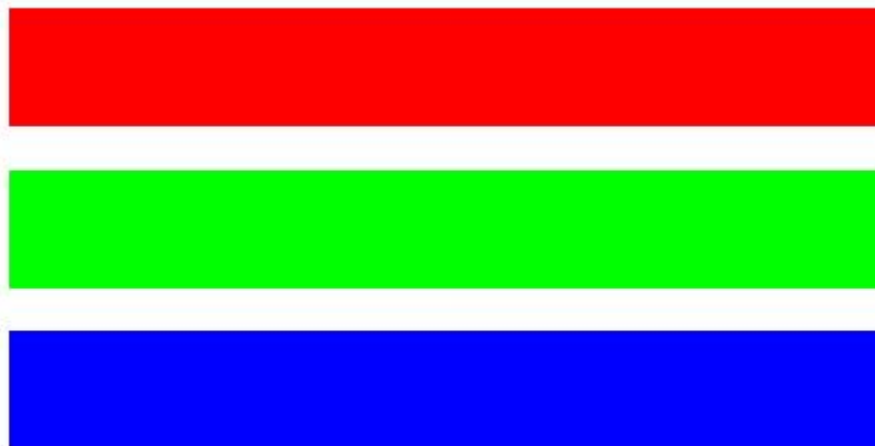
□ 建立5个带参匿名函数的Goroutine来实现同时问候

```
package main                                     D:\go-dev\src>go run go10.go
                                                    Hello, Eric.
import "fmt"                                       Hello, Harry.
import "runtime"                                   Hello, Robert.
                                                    Hello, Jim.
                                                    Hello, Mark.
func main() {
    names := []string{"Eric", "Harry", "Robert", "Jim", "Mark"}
    for _, name := range names {
        go func(who string) {
            fmt.Printf("Hello,%s.\n", who)
        }(name)
    }
    runtime.Gosched()
}
```

每次迭代的时候，name的值都作为参数传给形参who，此后name的改变与本次的go函数完全无关，新的name会传给下一个go函数的who参数中，这样go并发的时候就能得到正确的结果。name和who属于传值，name的改变不影响who

1.7如何实现通信——channel

- ❑ runtime.Gosched虽然能保证其他Goroutine有被执行的机会，但不能发布Goroutine完成信号，以及传递Goroutine的执行结果。
- ❑ 因此我们还需要一种结构，实现在两个或多个Goroutine之间传递消息，通过通信的方式来共享内存
- ❑ Go语言提供了channel这一结构来实现通信



1.7如何实现通信——channel(续)

- ❑ 通道是类型相关的，一个通道只能传递一种类型的值，这个值类型需要在声明channel时指定。
- ❑ channel的一般声明方式为：
 - `var channname chan ElementType`
 - `ElementType`指定了这个channel所能传递的元素类型
 - `var ch chan int` 声明ch是一个channel，传递int值
 - `var m = map[string] chan bool` 声明m是一个map，key是string类型，值是bool型的channel
 - `ch:=make(chan int)` 声明并初始化了一个int型channel
 - `ch:=make(chan int,10)` 带缓冲区的通道，容量是10
 - 内部函数len和cap也可以获得通道的当前实际的缓冲元素的数量，和可容纳的最大数量，cap值为0表示无缓冲，因此，可用cap来判断通道是否带缓冲

1.7如何实现通信——channel(续)

□ 写入通道/向通道发送信息

- `ch<-value`//将一个值value发送给通道ch称为写入
- 向channel写入数据通常会导致程序阻塞，直到有其他Goroutine从这个通道中读取数据

□ 读取通道/接收通道的信息

- `value := <-ch`//定义变量value并读取ch中的值
- 如果通道之前没有写入数据，那么从通道中读取数据也会导致程序阻塞，直到channel中被写入数据

1.7如何实现通信——channel(续)

- ❑ 已知：`strchan:=make(chan string,3)`
- ❑ 如果我们向`strchan`发送一个值“a”应该：
- ❑ `strchan<-" a"`
- ❑ 再向通道发送两个值
- ❑ `strchan<-" b"`
- ❑ `strchan<-" c"`
- ❑ 现在通道的缓冲区已经满了，当某个Goroutine在向`strchan`发送数据的时候，该Goroutine会被阻塞在那里，直到该通道中有足够的空间容纳该元素为止。
- ❑ 我们再从`strchan`接收一个元素值之后，那个Goroutine就会被唤醒，并完成那个发送操作

1.7如何实现通信——channel(续)

- 注意
- 当我们向一个值为nil的通道类型的通道变量发送元素值的时候，当前Goroutine会被永久地阻塞。
- 如果我们试图向一个已被关闭的通道发送元素值，那么会立即引起一个运行时恐慌。
- 即使发送操作正在因通道缓冲已满而被阻塞，这个通道的关闭同样会使该操作引发一个运行时恐慌
- 为了避免这样的流程中断，我们可以在select代码块中进行发送操作

1.7如何实现通信——channel(续)

- ❑ 关闭通道 `close`(通道变量)
- ❑ 关闭通道应该在保证安全的情况下进行
- ❑ 无论怎样都不应该在接收端关闭通道。因为那里无法判断发送端是否还会向该通道发送值，如果非要这样，就需要用辅助手段来避免发送端引发的运行时恐慌。
- ❑ 我们在发送端调用`close`关闭通道不会对接收端接收该通道中已有的元素值产生任何影响。

1.7如何实现通信——channel(续)

□ 举例

```
package main
import "fmt"
import "time"
func main() {
    ch := make(chan int, 5)
    sign := make(chan byte, 2)
    go func() {
        for i := 0; i < 5; i++ { //执行了5次发送操作, 每次间隔1s
            ch <- i
            time.Sleep(1 * time.Second)
        }
        close(ch) //关闭通道
        fmt.Println("The channel is closed.")
        sign <- 0 //
    }()
    go func() {
        for {
            //接收操作, 这里使用了接收的多重返回值方式接收
            //e代表值, ok是布尔值, 如果是false表示关闭, true表示未关闭
            e, ok := <-ch
            fmt.Printf("%d(%v)\n", e, ok)
            if !ok { //如果ok为false表示通道已经关闭, 退出循环
                break
            }
            time.Sleep(2 * time.Second) //每次接收操作间隔2s
        }
        fmt.Println("Done.") //输出
        sign <- 1
    }()
    //sign通道的作用是推迟主Goroutine被运行完成的时间
    //是两个go 匿名函数对应的 Goroutine顺利执行的关键
    //读取通道sign的值, 前提是通道 sign被写入了一个值, 否则该读取操作就出于阻塞等待状态
    //因为有两个go 匿名 goroutine, 他们执行完毕后都要向通道sign写入数据, 因此要接收两次
    <-sign
    <-sign
}
```

```
[ `go run go12.go` | done: 11.0786336s ]
0(true)
1(true)
2(true)
The channel is closed.
3(true)
4(true)
0(false)
Done.
```

不同的时间间隔的作用：接收端在没有全部接收完发送端的数据之前，通道就会被关闭

结果表明运行时系统并没有在通道被关闭后立即把false作为相应的接收操作的第二个结果,而是等到把已在通道中的所有元素接收之后才这样做,这确保了发送端关闭通道的安全性

1.8如何处理通信中的死锁问题

- 在并发编程的通信过程中，最需要处理的就是死锁问题
 - 向channel写数据时，发现channel已满
 - 试图从channel读数据时，发现channel为空
- 解决方案是引入超时限制
 - 如果一个Goroutine，超过设定的时间，仍然没有完成处理的任务（如因为不能向channel读写数据而被阻塞的情况），则该方法会立即终止并返回对应的超时信息
 - 超时机制可能带来一些问题，如在高速机器或网络上运行的程序，到了慢速机器或网络上就会出问题，从而出现结果不一致的现象
 - 从根本上来说引入超时机制解决通信死锁这一问题的价值要大于所带来的问题

1.8如何处理通信中的死锁问题(续)

□ 实现码段

//首先，实现并执行一个匿名的超时等待函数

```
timeout:=make(chan bool,1)
```

```
go func(){
```

```
    time.Sleep(1*time.Second)//等待1s钟
```

```
    timeout=true
```

```
}()
```

//然后，把这个timeout利用起来

```
select{
```

```
    case <-ch
```

```
    //从ch中读到数据
```

```
    case <-timeout
```

```
    //一直没有从ch中读到数据，但从timeout中读到了数据
```

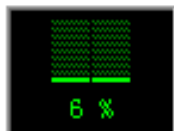
```
}
```

- select用于处理异步IO问题，用法与switch类似，要求所有的case必须是一个IO操作

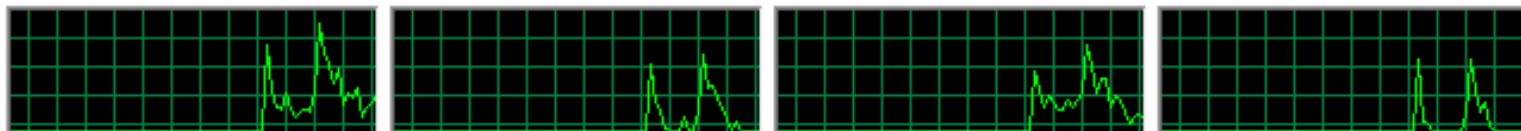
1.9如何实现多核并行化

- ❑ 我们需要了解CPU核心的数量，并有针对性地分解计算任务到多个Goroutine中去并行运行
- ❑ 如何设置最大可用CPU核数呢？
 - `runtime.GOMAXPROCS(4)`
 - 在启动Goroutine之前来调用这个方法。
- ❑ 到底应该设置多少个CPU核心呢
 - `runtime.NumCPU()` //获取CPU的逻辑核数
 - `runtime.GOMAXPROCS(runtime.NumCPU())`

CPU 使用率



CPU 使用记录



1.9如何实现多核并行化(续)

□ 并行实现计算 $1+2+\dots+1000000000=?$

□ 量化

```
1 package main
2 import "fmt"
3 import "runtime"
4 //定义vector类型 实际为[]int64类型的切片,存放计算的数据
5 type vector []int64
6 var result int64 //全局变量result存放计算结果
```


1.9如何实现多核并行化(续)

- 并行实现计算 $1+2+\dots+1000000000=?$
- 计算总控模块

```
17 func (v vector) DoAll() { //计算总控模块
18     var NCPU int = runtime.NumCPU() //获取CPU的逻辑核数
19     //用于接收每个CPU任务完成的信号, 缓冲数量为CPU的逻辑核数
20     c := make(chan int64, NCPU)
21     //按核数分解计算任务, 然后开启独立Goroutine去并发计算
22     for i := 0; i < NCPU; i++ {
23         go v.Dosome(i*len(v)/NCPU, (i+1)*len(v)/NCPU, c)
24     }
25     for i := 0; i < NCPU; i++ { //等待所有的CPU任务的完成
26         //获取c中的计算结果, 同时也表示一个CPU计算完成了
27         value := <-c
28         result += value //将得到的值汇总到result
29     }
30 }
```

1.9如何实现多核并行化(续)

- 并行实现计算 $1+2+\dots+1000000000=?$
- 计算模块

```
var result int64 //全局变量result存放计算结果
//计算从v[i]到v[n]的各元素和
func (v vector) Dosome(i, n int, c chan int64) {
    for ; i < n-1; i++ {
        v[i+1] += v[i] //累加求和
    }
    //发送信号告诉主Goroutine已经计算完成，c的值为计算结果
    c <- v[i]
}
```

1.9如何实现多核并行化(续)

- 并行实现计算 $1+2+\dots+1000000000=?$
- 主程序模块

```
32 func main() {  
33     //设置最大使用的核心数  
34     runtime.GOMAXPROCS(runtime.NumCPU())  
35     //建立切片  
36     v := make(vector, 1000000000, 1000000000)  
37     for i := 0; i < 1000000000; i++ { //初始化  
38         v[i] = int64(i + 1) //每个元素的值设为i+1  
39     }  
40     v.DoAll() //调用总控模块  
41     fmt.Println("The result is:", result)  
42 }
```

1.10总结

- ❑ 如何通过go建立Goroutine(掌握)
- ❑ 如何通过channel实现通信(掌握)
- ❑ 如何进行多核化编程(掌握)
- ❑ 补充基于最小权限原则的单向通道
 - 所谓的单向通道只是对channel的一种使用限制，我们在将一个channel变量传递到一个函数时，可以限制该函数对此channel只能进行读或写的操作，防止滥用产生程序失控，单向通道就是起到一种契约作用

```
var ch1 chan int // ch1是一个正常的channel，不是单向的
var ch2 chan<- float64 // ch2是单向channel，只用于写float64数据
var ch3 <-chan int // ch3是单向channel，只用于读取int数据
ch4 := make(chan int)
ch5 := <-chan int(ch4) // ch5就是一个单向的读取channel
ch6 := chan<- int(ch4) // ch6 是一个单向的写入channel
```

1.10总结(续)

□ 单向通道举例

```
1 package main
2 import "fmt"
3 import "time"
4 //接受一个参数,是只允许读取通道,除非直接强制转换
5 //要么只能从channel中读取数据
6 func sCh(ch <-chan int) {
7     for val := range ch {
8         fmt.Println(val)
9     }
10 }
11 func main() {
12     //创建一个带100缓冲的通道 可以直接写入而不会导致主线程堵塞
13     dch := make(chan int, 100)
14     for i := 0; i < 100; i++ {
15         dch <- i
16     }
17     //传递进去 只读通道
18     go sCh(dch)
19     time.Sleep(1e9)
20 }
```

1.10总结(续)

□ 补充——如何通过共享内存来通信？

- Go除了为开发者提供了自己特有的并发编程模型和工具之外，还提供了传统的同步工具。它们都在Go语言的sync包和sync/atomic中。

□ 互斥锁

- 互斥锁是传统并发程序对资源进行访问控制的主要手段。它由包sync中的Mutex结构体类型代表。sync.Mutex只有两个公开的方法Lock和Unlock，前者用于锁定当前的互斥量，后者可以解锁当前的互斥量。
- 通过简单的声明就可以使用了

```
var mutex sync.Mutex
mutex.Lock()
...
mutex.Unlock()
```

1.10总结(续)

□ 互斥锁实际有两种

□ sync.Mutex

- sync.Mutex比较简单，但也比较暴力，当一个Goroutine获得了Mutex后，其他的Goroutine就只能乖乖的等待这个Goroutine释放该Mutex

□ sync.RWMutex

- sync.RWMutex相对好一些，是经典的单写多读模式。读锁占用的情况下，会阻止写，但不会阻止别的Goroutine也读(调用Rlock方法)。而写锁(调用Lock方法)占用的情况下会阻止任何其他Goroutine进来，无论读写都不可以，整个锁相当于由该Goroutine独占

1.10总结(续)

- 我们可能犯的低级错误就是忘记及时解开被锁住的锁，从而导致流程异常、停止、死锁等问题

- Go中可通过defer语句使得这个错误发生率变低

```
var mutex sync.Mutex
func write(){
    mutex.Lock()
    defer mutex.Unlock()
    ...
}
```

- defer会延迟 函数的执行直到上层函数返回

- write中的这条defer语句会延迟到write函数返回前执行
- 写在mutex.Lock()后面就是配对使用，防止忘记。也很优雅

1.10总结(续)

□ 互斥锁举例

- 10个Goroutine共享了变量counter，每个Goroutine执行完毕后，counter的值加1，因为是并发执行，每次counter++前都要锁住，保证当前Goroutine的控制权，counter++后解锁，释放控制权给其他Goroutine。
- 主函数的第二个for循环也要时刻检查counter的值，这个操作也要加锁和解锁，保证控制权和释放控制权

```
1 package main
2 import (
3     "fmt"
4     "runtime"
5     "sync"
6 )
7 var counter int = 0
8 func Count(lock *sync.Mutex) {
9     lock.Lock()
10    counter++
11    lock.Unlock()
12 }
13 func main() {
14     lock := &sync.Mutex{}
15     for i := 0; i < 10; i++ {
16         go Count(lock)
17     }
18     for {
19         lock.Lock()
20         c := counter
21         lock.Unlock()
22         runtime.Gosched()
23         if c == 10 {
24             break
25         }
26     }
27     fmt.Println(counter)
28 }
```

1.10总结(续)

```
1 package main
2 import (
3     "fmt"
4     "runtime"
5     "sync"
6 )
7 var counter int = 0
8 func Count(lock *sync.Mutex) {
9     lock.Lock()
10    counter++
11    lock.Unlock()
12 }
13 func main() {
14     lock := &sync.Mutex{}
15     for i := 0; i < 10; i++ {
16         go Count(lock)
17     }
18     for {
19         lock.Lock()
20         c := counter
21         lock.Unlock()
22         runtime.Gosched()
23         if c == 10 {
24             break
25         }
26     }
27     fmt.Println(counter)
28 }
```

优雅的defer

```
1 package main
2 import (
3     "fmt"
4     "runtime"
5     "sync"
6 )
7 var counter int = 0
8 func Count(lock *sync.Mutex) {
9     lock.Lock()
10    defer lock.Unlock()
11    counter++
12 }
13 func main() {
14     lock := &sync.Mutex{}
15     for i := 0; i < 10; i++ {
16         go Count(lock)
17     }
18     for {
19         lock.Lock()
20         c := counter
21         lock.Unlock()
22         runtime.Gosched()
23         if c == 10 {
24             break
25         }
26     }
27     fmt.Println(counter)
28 }
```

1.10总结(续)

□ 补充：全局唯一性操作

- 这段代码如果没有引用Once，setup()将会被每个Goroutine调用1次，实际上执行一次就够了。

- Go标准库为我们引入了Once类型，once的Do()方法可以保证在全局范围内只调用指定的函数一次，而且其他Goroutine在调用到此语句时候，将会先被阻塞，直到全局唯一操作once.Do()调用结束后才继续

- 更方便了并发开发

```
1 var a string
2 var once sync.Once
3 func setup(){
4     a="hello,world"
5 }
6 func doprint(){
7     once.Do(setup)
8     fmt.Println(a)
9 }
10 func twoprint(){
11     go doprint()
12     go doprint()
13 }
```

1.11思考

```
1 package main
2 import (
3     "fmt"
4     "time"
5 )
6 func say(s string) {
7     for i := 0; i < 5; i++ {
8         time.Sleep(100 * time.Millisecond)
9         fmt.Println(s)
10    }
11 }
12 func main() {
13     go say("world")
14     say("hello")
15 }
```

[`go run go1.go` | done: 907.0519ms]

hello
world
hello
world
hello
world
hello
world
hello

为什么只输出了4个
world?

1.11思考(续)

```
1 package main          [ `go run go1.go` | done: 934.0534ms ]
2 import (               hello
3     "fmt"              hello
4     // "time"          hello
5 )                      hello
6 func say(s string) {   hello
7     for i := 0; i < 5; i++ {
8         //time.Sleep(100 * time.Millisecond)
9         fmt.Println(s)
10    }
11 }
12 func main() {
13     go say("world")
14     say("hello")
15 }
```

为什么结果只输出了5个hello ?

1.11思考(续)

□ 给出下面程序的输出结果

```
1  package main
2
3  import "fmt"
4  import "runtime"
5
6  func main() {
7      name := "Eric"
8      go func() {
9          fmt.Printf("Hello,%s", name)
10
11      }()
12      name = "Harry"
13      runtime.Gosched()
14 }
```

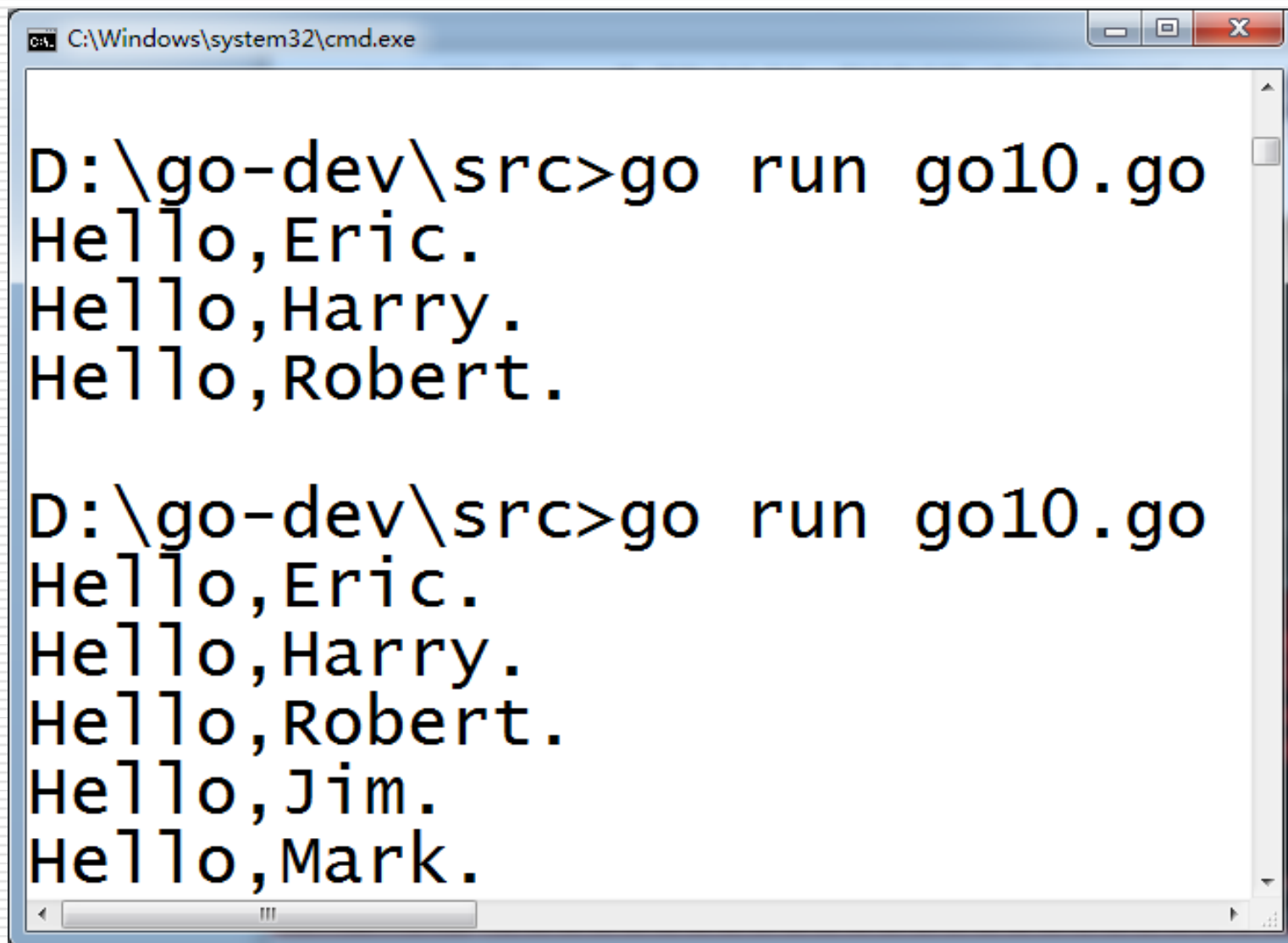
1.11思考(续)

□ 给出下面程序的输出结果

```
1  package main
2
3  import "fmt"
4  import "runtime"
5
6  func main() {
7      name := "Eric"
8      go func() {
9          fmt.Printf("Hello,%s", name)
10
11      }()
12      runtime.Gosched()
13      name = "Harry"
14 }
```

1.11思考(续)

□ 两次执行出现不同的结果，为什么？



The screenshot shows a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The prompt is at "D:\go-dev\src>". The first execution of "go run go10.go" produces the output: "Hello, Eric.", "Hello, Harry.", and "Hello, Robert.". The second execution of the same command produces the output: "Hello, Eric.", "Hello, Harry.", "Hello, Robert.", "Hello, Jim.", and "Hello, Mark.". The difference in output between the two runs illustrates a non-deterministic behavior, likely due to a race condition in the Go program.

```
C:\Windows\system32\cmd.exe

D:\go-dev\src>go run go10.go
Hello, Eric.
Hello, Harry.
Hello, Robert.

D:\go-dev\src>go run go10.go
Hello, Eric.
Hello, Harry.
Hello, Robert.
Hello, Jim.
Hello, Mark.
```


1.11思考(续)

□ 分析下面代码的执行逻辑及执行结果

```
1  package main
2  import "fmt"
3  func sum(a []int, c chan int) {
4      sum := 0
5      for _, v := range a {
6          sum += v
7      }
8      c <- sum // 将和送入 c
9  }
10 func main() {
11     a := []int{7, 2, 8, -9, 4, 0}
12
13     c := make(chan int)
14     go sum(a[:len(a)/2], c)
15     go sum(a[len(a)/2:], c)
16     x, y := <-c, <-c // 从 c 中获取
17     fmt.Println(x, y, x+y)
18 }
```

1.11思考(续)

□ 分析下面代码的执行逻辑及执行结果

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      c := make(chan int, 2)
7      c <- 1
8      c <- 2
9      fmt.Println(<-c)
10     fmt.Println(<-c)
11 }
```

1.11思考(续)

□ 分析下面代码的执行逻辑及执行结果

```
1 package main
2 import (
3     "fmt"
4 )
5 func fibonacci(n int, c chan int) {
6     x, y := 0, 1
7     for i := 0; i < n; i++ {
8         c <- x
9         x, y = y, x+y
10    }
11    close(c)
12 }
13 func main() {
14     c := make(chan int, 10)
15     go fibonacci(cap(c), c)
16     for i := range c {
17         fmt.Println(i)
18     }
19 }
```

1.11思考(续)

□ 分析下面代码的执行逻辑及执行结果

```
1 package main
2 import "fmt"
3 func fibonacci(c, quit chan int) {
4     x, y := 0, 1
5     for {
6         select {
7             case c <- x:
8                 x, y = y, x+y
9             case <-quit:
10                 fmt.Println("quit")
11                 return
12         }
13     }
14 }
15 func main() {
16     c := make(chan int)
17     quit := make(chan int)
18     go func() {
19         for i := 0; i < 10; i++ {
20             fmt.Println(<-c)
21         }
22         quit <- 0
23     }()
24     fibonacci(c, quit)
25 }
```

1.11思考(续)

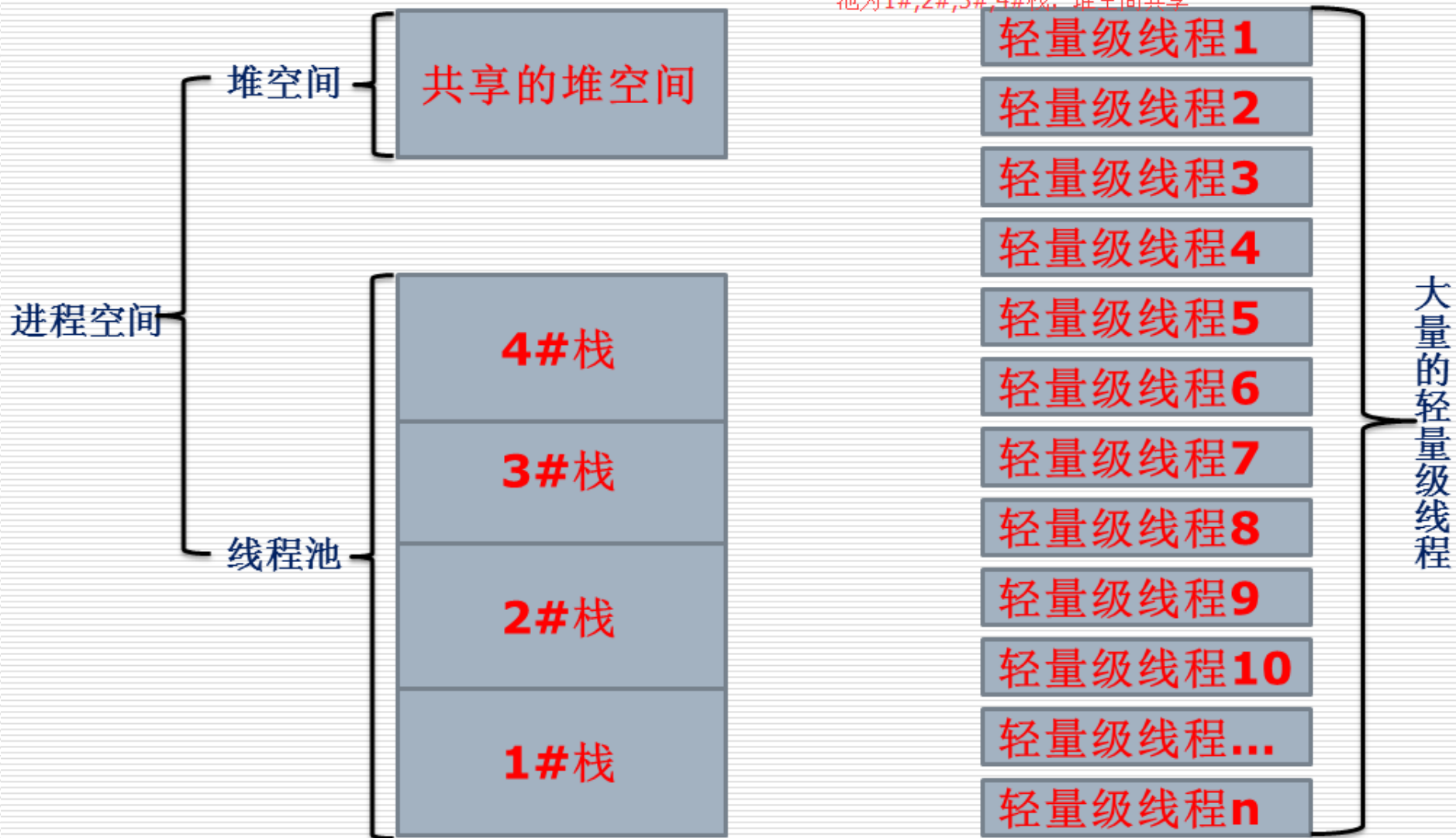
- 什么是并行、什么是并发？二者关系与区别
- 对1.9中并行实现计算 $1+2+\dots+1000000000=?$ 的代码进行性能测试，并生成web方式下的CPU使用性能图，观察使用了几核CPU？

附：4核CPU下的并发示意动画截图

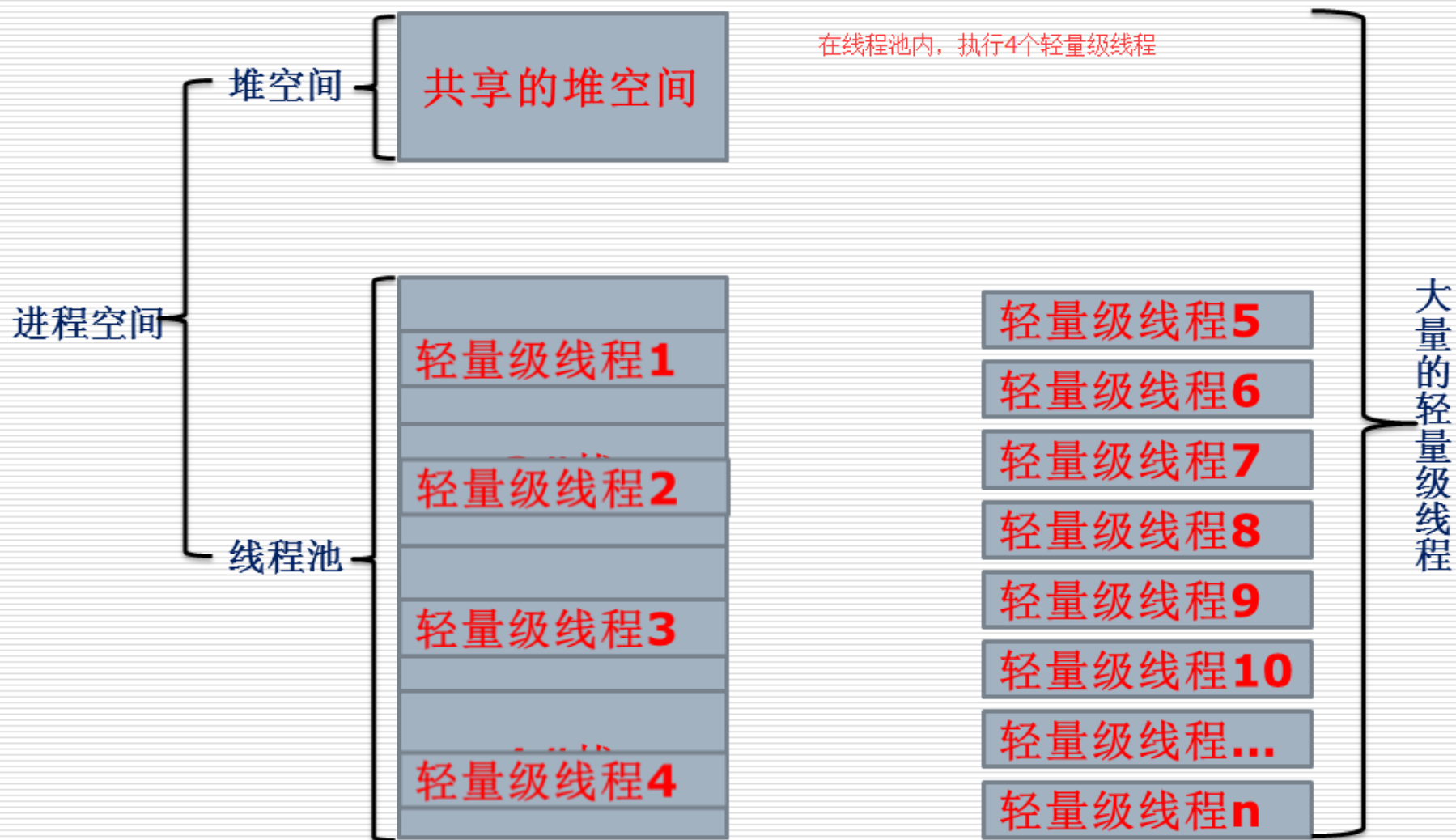
□ 4核CPU下的并发效果示意

假设4核心CPU可以并行处理4个线程，我们为一个进程分为n个轻量级线程来执行。

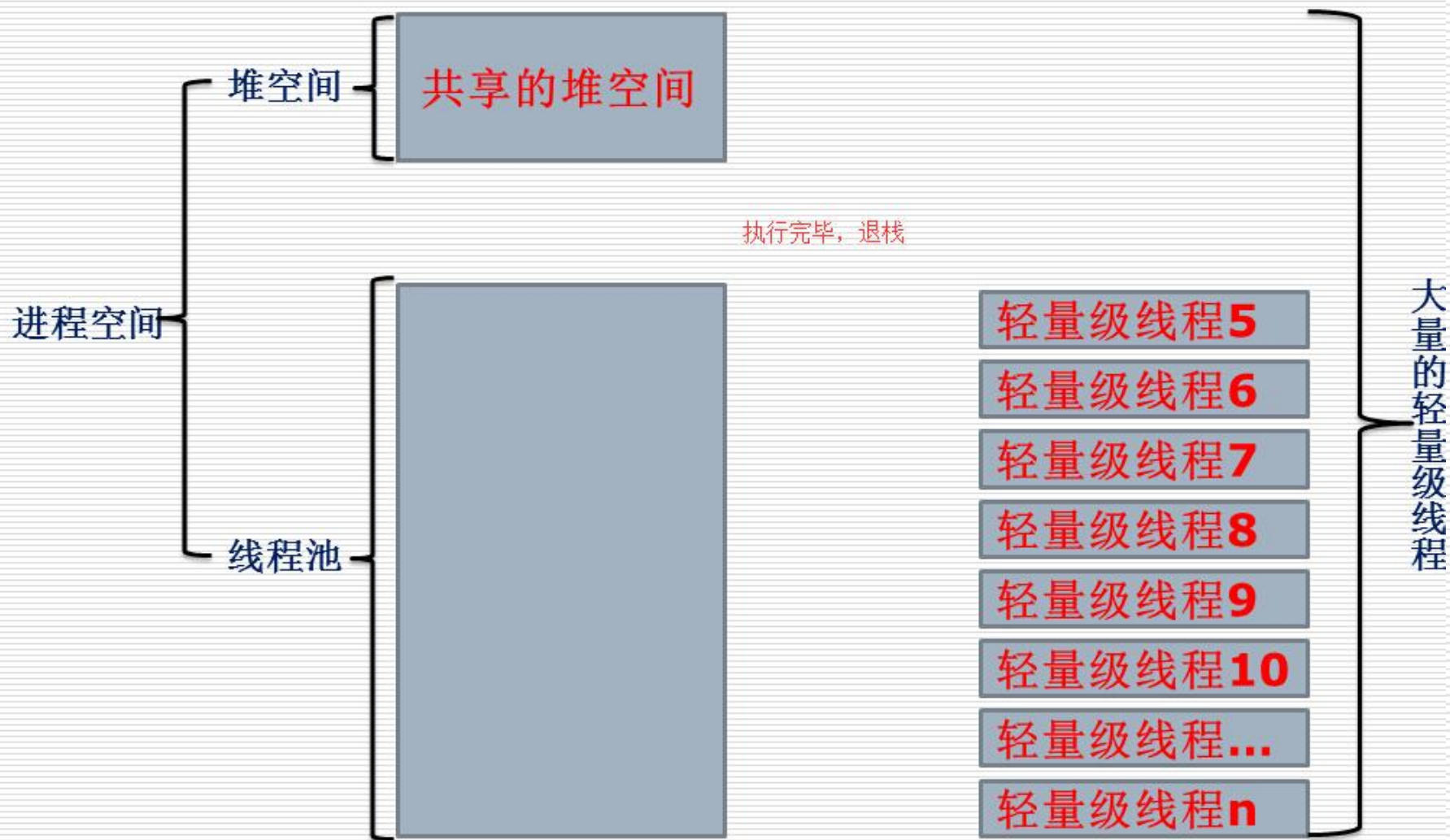
首先为将要执行的4个就绪状态下的4个轻量级线程建立线程池为1#,2#,3#,4#栈，堆空间共享



附：4核CPU下的并发示意动画截图(续)

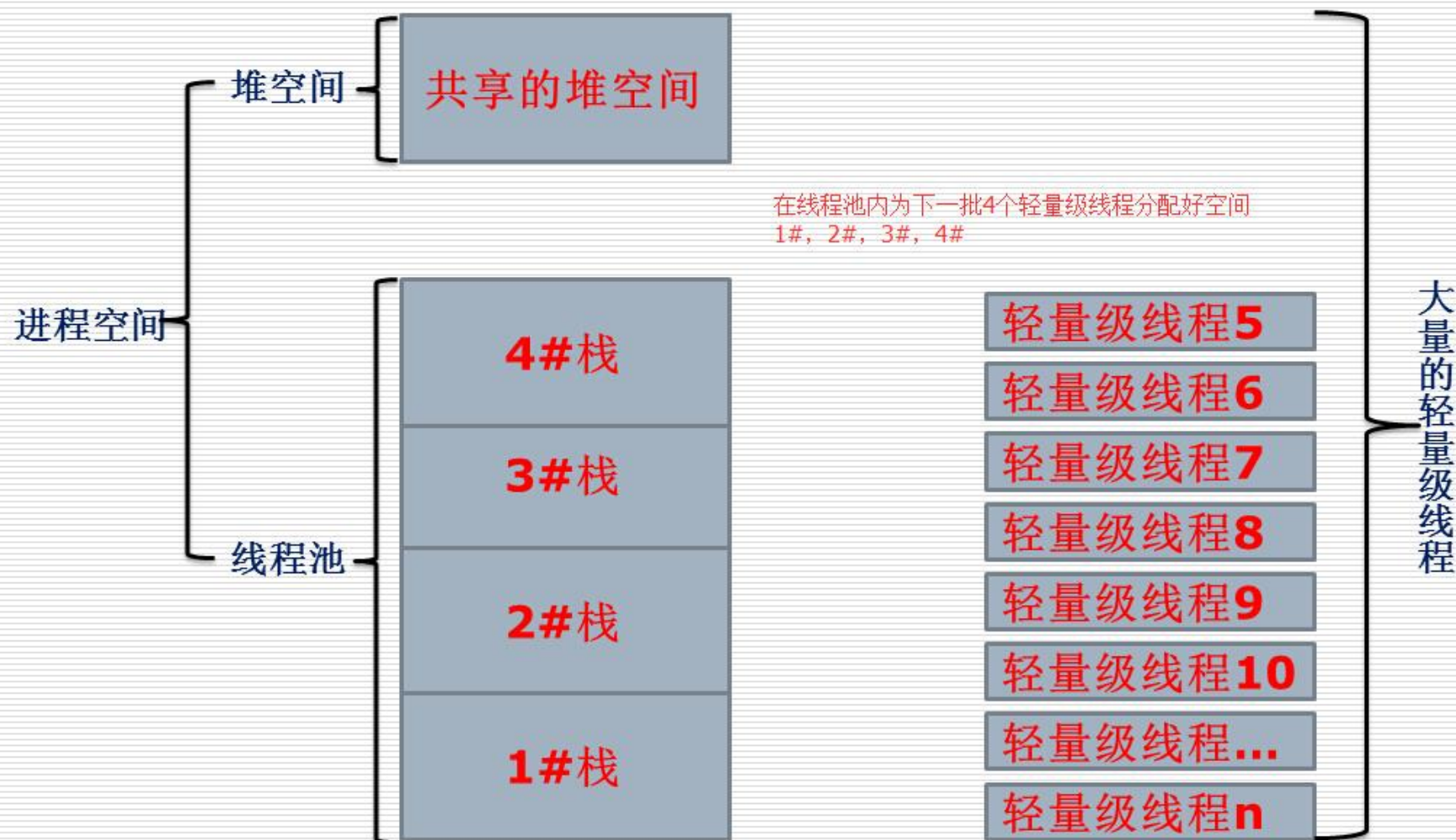


附：4核CPU下的并发示意动画截图(续)

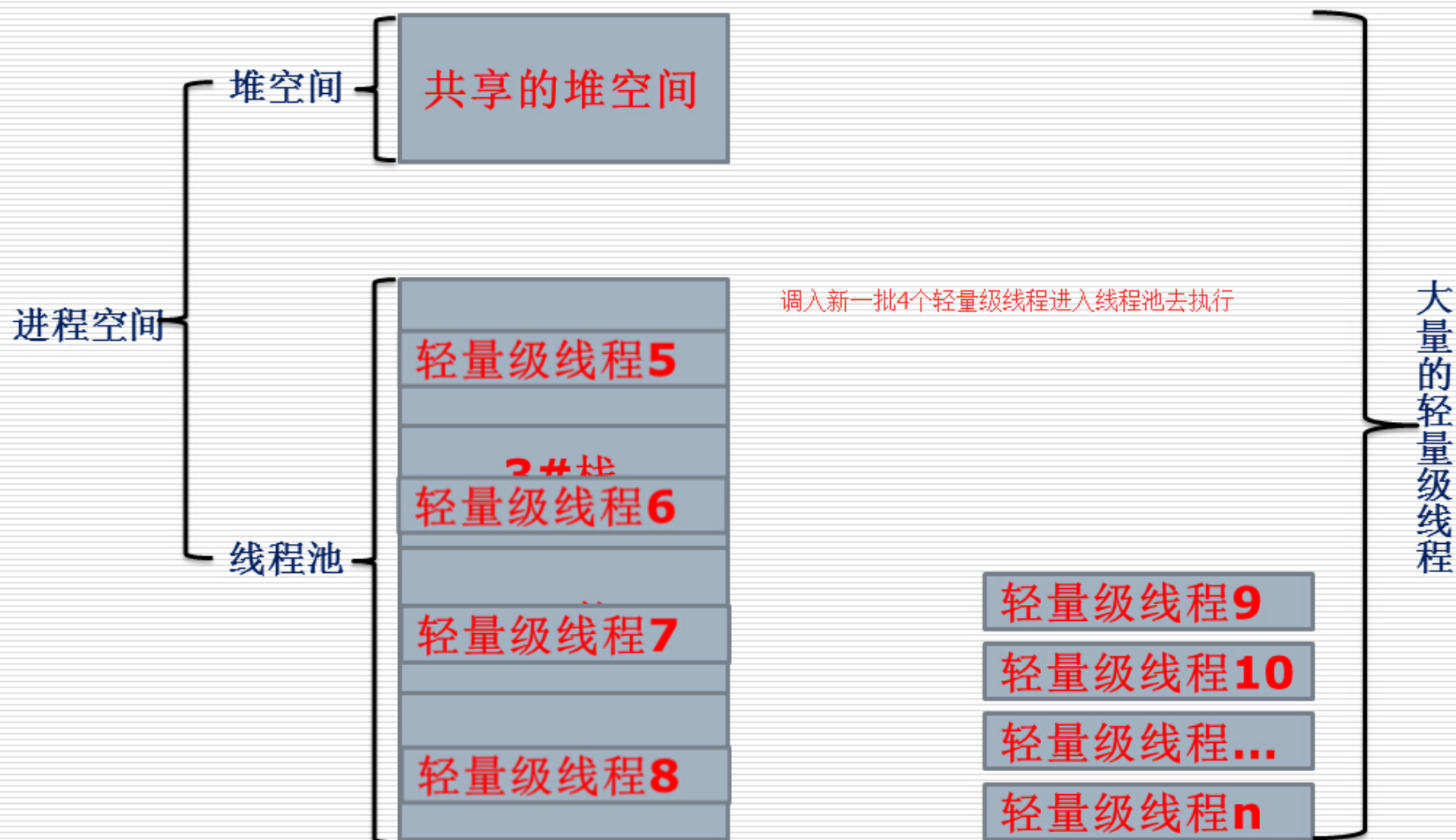


附：4核CPU下的并发示意动画截图(续)

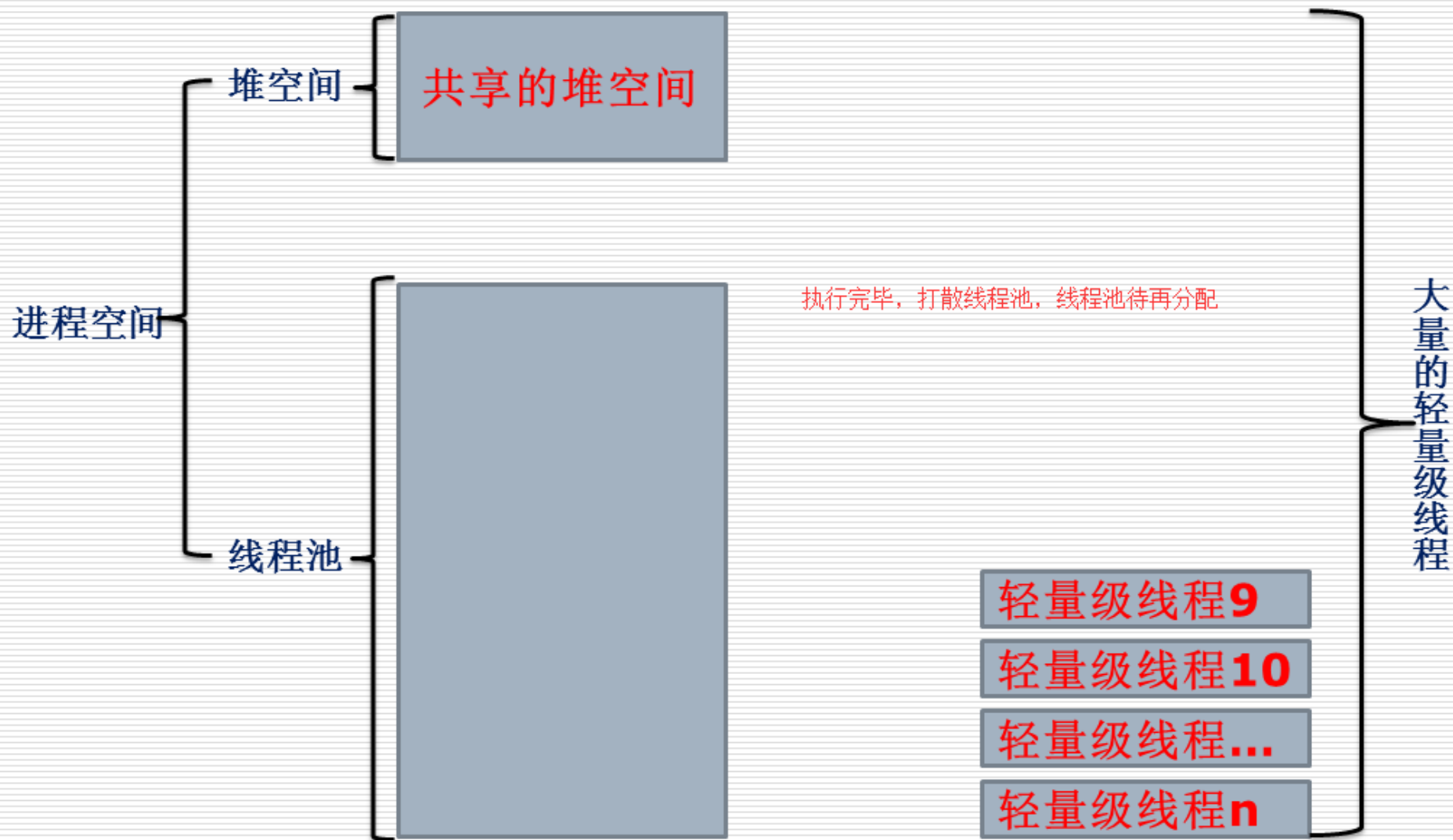
□ 4核CPU下的并发效果示意



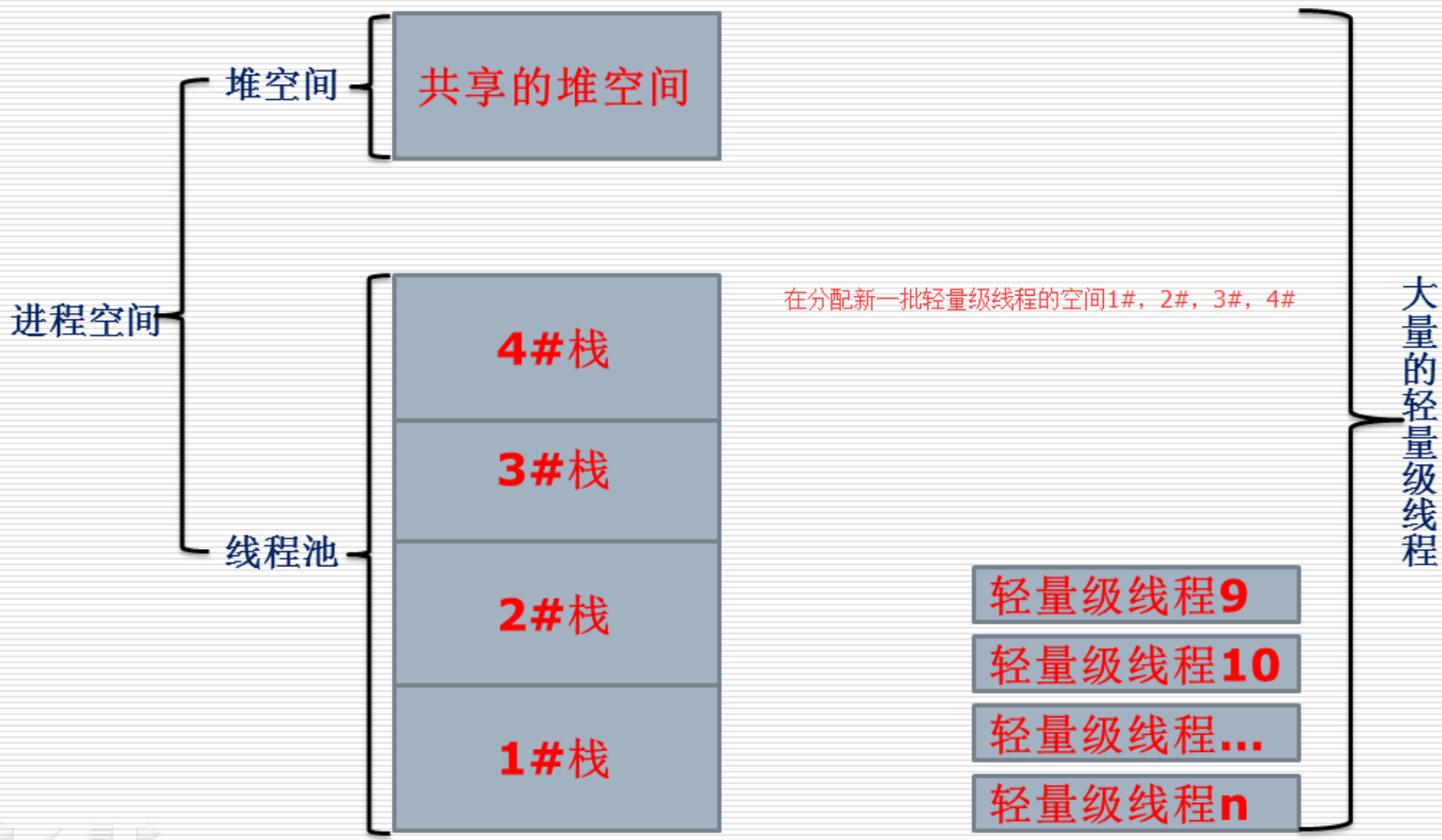
附：4核CPU下的并发示意动画截图(续)



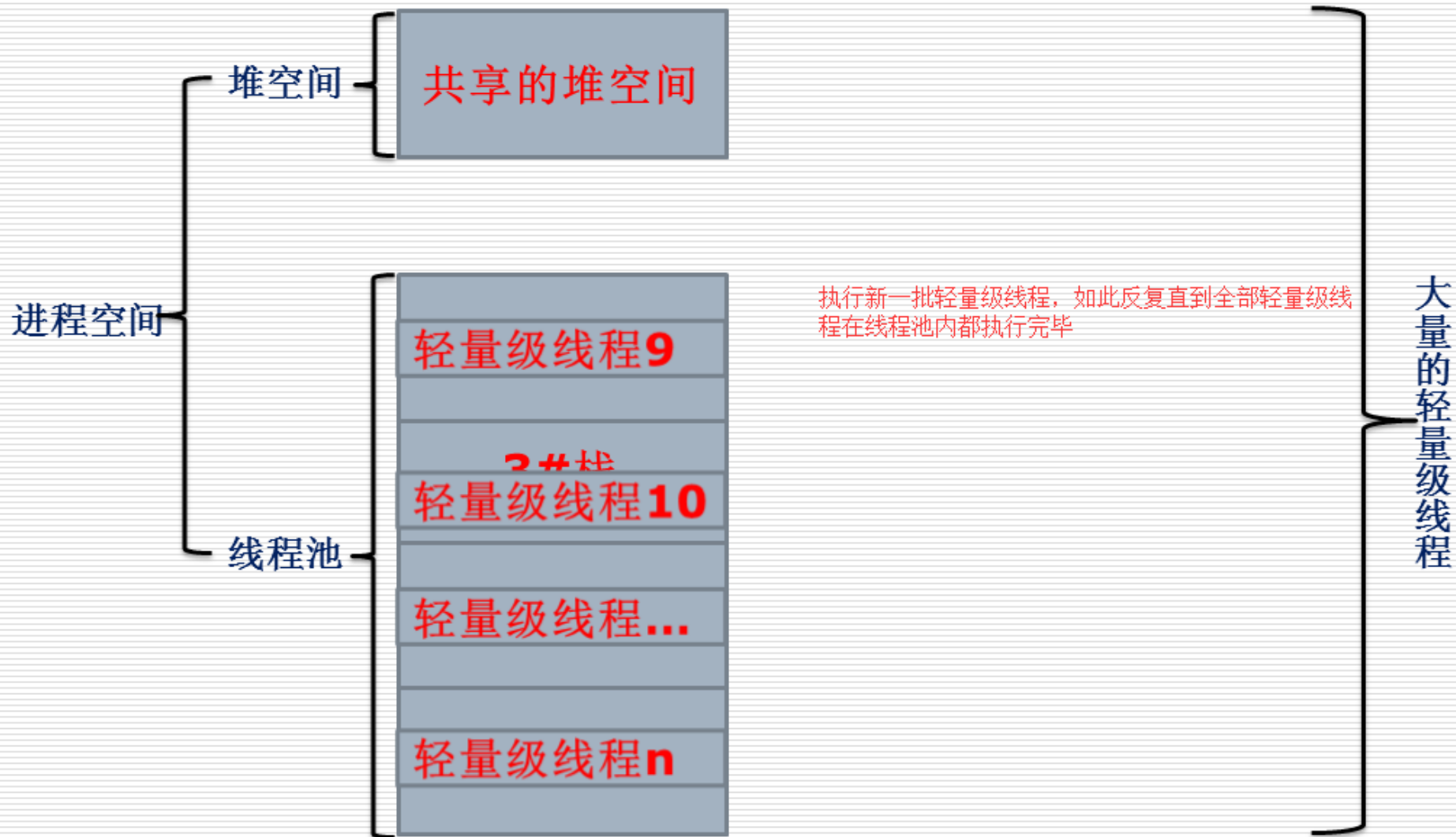
附：4核CPU下的并发示意动画截图(续)



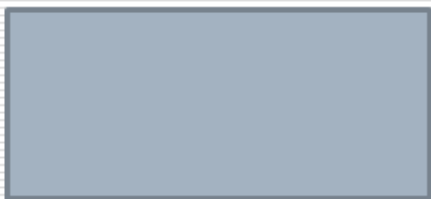
附：4核CPU下的并发示意动画截图(续)



附：4核CPU下的并发示意动画截图(续)

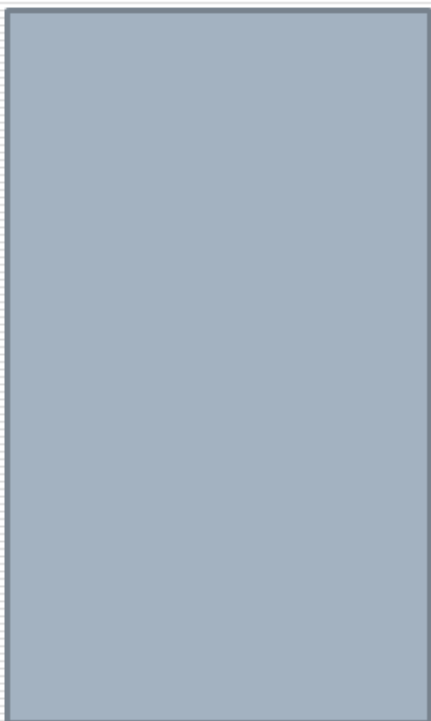


附：4核CPU下的并发示意动画截图(续)



进程执行完毕，即若干轻量级线程在线程池中全部执行完毕。

空间重新归操作系统调用



Thank you very much

*Any comments and suggestions
are beyond welcome*