Go并发编程——goroutine

目录：

一、并发与并行

（一）、Go是并发语言，而不是并行语言

1、在讨论如何在Go中进行并发处理之前，我们首先必须了解什么是并发，以及它与并行性有什么不同。(Go is a concurrent language and not a parallel one.)

2、并发性Concurrency

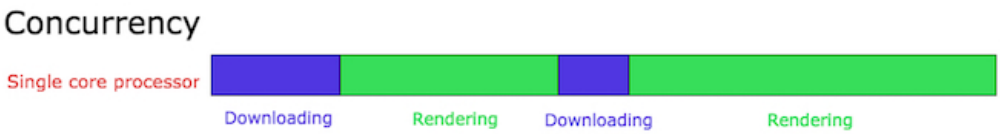
- 把任务在不同的时间点交给处理器进行处理。在同一个时间点，任务不会同时运行。
- 并发性是同时处理许多事情的能力。
- 举个例子，打电话和吃饭。吃饭时，电话来了，需要停止吃饭去接电话。电话接完后回来继续吃饭，这个过程是并发执行。

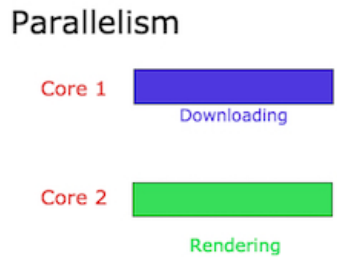
3、并行性parallelism

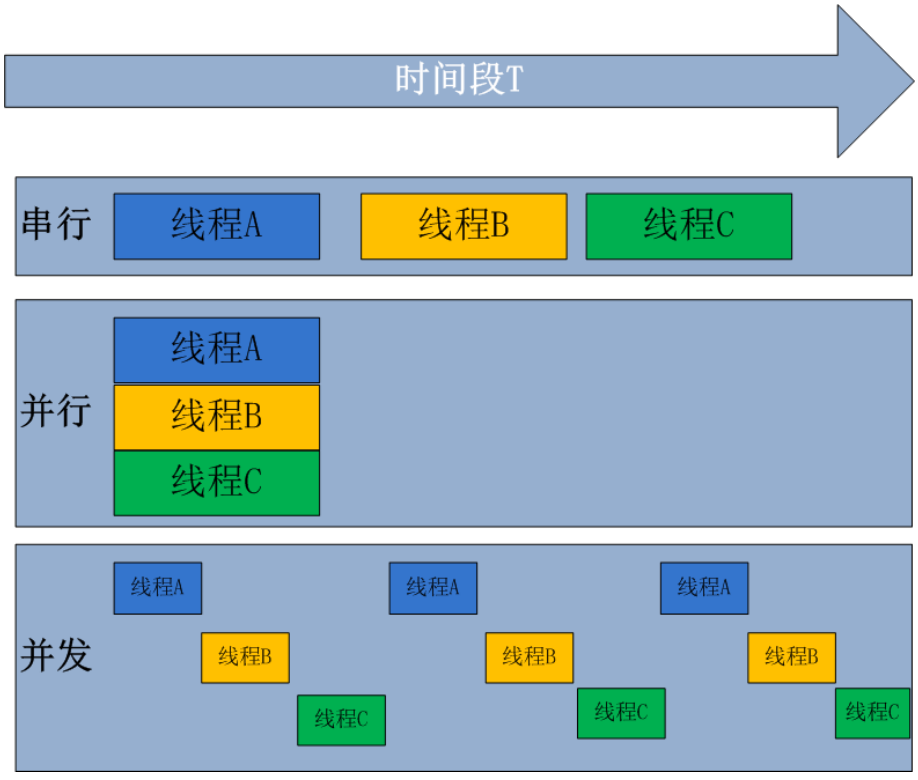
- 并行是把一个任务分配给每一个处理器独立完成。在同一个时间点，任务一定是同时运行。
- 并行就是同时做很多事情。这听起来可能与并发类似，但实际上是不同的。
- 用同样打电话和吃饭的例子去理解。吃饭时，电话来了，边吃饭边接电话，这个过程是并行执行。这就是所谓的并行性(parallelism)。
- Go在Gomaxprocs数量与任务数量相等时，可以做到并行执行，但是一般情况下都是并发执行。

4、并发性和并行性——一种技术上的观点。

- 假设我们正在编写一个web浏览器。web浏览器有各种组件，其中两个是web页面呈现区域和从internet下载文件的下载器。
  - 当这个浏览器运行在单个核处理器中时，处理器将在浏览器的两个组件之间进行上下文切换。它可能会下载一个文件一段时间，然后它可能会切换到呈现用户请求的网页的html。这就是所谓的并发性。并发进程从不同的时间点开始，它们的执行周期重叠。在这种情况下，下载和呈现从不同的时间点开始，它们的执行重叠。
  - 假设同一浏览器运行在多核处理器上。在这种情况下，文件下载组件和HTML呈现组件可能同时在不同的内核中运行。这就是所谓的并行性。
  - 并行性Parallelism不会总是导致更快的执行时间。
    - 这是因为并行运行的组件可能需要相互通信。例如，在我们的浏览器中，当文件下载完成时，应该将其传递给用户，比如使用弹出窗口。这种通信发生在负责下载的组件和负责呈现用户界面的组件之间。当组件在多个内核中并行运行时，这种通信开销很高。因此，并行程序并不总是导致更快的执行时间!
    - 这种通信开销在并发concurrent 系统中很低。
- 在单CPU上，是os代码强制把一个进程或者线程挂起，换成另外一个，所以实际上是并发的，只是“概念上的并行”。在现在的多核的cpu上，线程可能是“真正并行的”。







（二）、什么是并发?

1、并发指在同一时间内可以执行多个任务。

- 并发编程含义广泛，可以是多线程编程，也可以是多进程编程，还可以是

编写分布式程序。

- 并发编程的思想来自于多任务操作系统，它允许同时运行多个程序。
- 早期单用户操作系统，任务是一个接一个运行，单个任务的执行完全是串行的。只有在一个任务运行完成之后，另一个任务才会被读取。
- 多任务操作系统则允许终端用户同时运行多个程序。当一个程序暂时不需要使用CPU的时候，系统会把该程序挂起或中断，以使其他程序可以使用CPU。
- 最早支持并发编程的计算机编程语言是汇编语言。
- 串行程序特指只能被顺序执行的指令列表，并发程序则是可以被并发执行的多个串行程序的综合体。因此并发程序内部会被划分为多个部分，每个部分都可以看做一个串行程序。
- 串行程序中的所有代码的先后顺序都是固定的，而并发程序中只有部分代码是有序的。这一特性被称为不确定性。这导致并发程序每次运行的代码执行路径都是不同的，即便输入数据相同也会如此。

2、Go语言通过编译器运行时runtime，从语言上支持并发的特性。

3、Go语言的并发通过Goroutine特性完成。

二、进程和线程

(一)、基本概念

1、进程（process）和线程（thread）

- 进程和线程是操作系统级别的两个基本概念。
- 计算机的核心是CPU，它承担了所有的计算任务。它就像一座工厂，时刻在运行。
- 进程就好比工厂的车间，它代表CPU所能处理的单个任务。进程是一个容器。
- 线程就好比车间里的工人。一个进程可以包括多个线程。线程是容器中的工作单位。
- 一个进程的内存空间是共享的，每个线程都可以使用这些共享内存。就如同车间的空间是工人们共享的，许多房间是每个工人都可以进出的。
- 车间的每个房间大小不同，有些房间最多只能容纳一人，比如厕所。里面有人的时候，其他人就不能进去了。这代表一个线程使用某些共享内存时，其他线程必须等它结束，才能使用这一块内存。防止他人进入的简单方法，

就是门口加一把锁。先到的人锁上门，后到的人看到上锁，就在门口排队，等锁打开再进去。这就叫"互斥锁"（Mutual exclusion，缩写 Mutex），防止多个线程同时读写某一块内存区域。

- 还有些房间，可以同时容纳n个人，比如厨房。如果人数大于n，多出来的人只能在外面等着。这好比某些内存区域，只能供给固定数目的线程使用。这时的解决方法，就是在门口挂n把钥匙。进去的人就取一把钥匙，出来时再把钥匙挂回原处。后到的人发现钥匙架空了，就知道必须在门口排队等着了。这种做法用来保证多个线程不会互相冲突。

2、进程的定义

- 进程是程序的一次动态执行过程；
- 进程需要经历从代码加载、代码执行到执行完毕的一个完整过程，这个过程也就是进程本身从产生、发展到消亡的过程；
- 每个运行中的程序就是一个进程；
- 进程是一个实体。每一个进程都有它自己的地址空间，一般情况下，包括文本区域（text region）、数据区域（data region）和堆栈（stack region）；
- 进程在系统中独立存在，拥有自己独立的资源，多个进程可以在单个处理器上并发执行且互不影响；
- 操作系统可以同时执行多个任务，每个任务就是进程。
- 进程状态：进程有三个状态：就绪、运行和阻塞。
    - 就绪状态其实就是获取了除CPU之外的所有资源，只要处理器分配资源就可以马上执行。就绪状态有排队序列什么的，排队原则不再赘述；
    - 运行态就是获得了处理器分配的资源，程序开始执行；
    - 阻塞态，当程序条件不够时候，需要等待条件满足时候才能执行，如等待i/o操作时候，此刻的状态就叫阻塞态。

3、线程

- 线程是CPU的基本调度单位；
- 线程是在进程之后发展出来的概念，线程也叫轻量级进程；
- 通常一个进程包含若干个线程，当然一个进程中至少有一个线程，不然没有存在的意义；
- 线程可以利用进程所拥有的资源，在引入线程的操作系统中，通常都是把进程作为分配资源的基本单位，而把线程作为独立运行和独立调度的基本单

位；

- 由于线程比进程更小，基本上不拥有系统资源，故对它的调度所付出的开销就会小得多，能更高效的提高系统多个程序间并发执行的程度。
- 例如：杀毒软件程序是一个进程，该程序在为计算机进行检查的同时，可以清理垃圾文件，并进行日志记录。检查计算机、清理垃圾文件、日志记录这就是线程。
- 例如：启动word文档就是运行了一个进程，文字录入时，可以进行拼写错误检查，并使首字母大写。文字录入、拼写错误检查、首字母大写就属于word操作进程中的线程。

（二）、进程与线程的区别

1、进程和线程的主要差别在于它们是不同的操作系统资源管理方式。

进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程。

1) 简而言之，一个程序至少有一个进程，一个进程至少有一个线程；

2) 线程的划分尺度小于进程，使得多线程程序的并发性高；

3) 另外，进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。

4) 线程在执行过程中与进程还是有区别的。每个独立的进程有一个程序运行的入口、顺序执行程序。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

5) 从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。这就是进程和线程的重要区别。

6）线程执行开销小，但不利于资源的管理和保护；而进程正相反。同时，线程适合于在SMP(多核处理机)机器上运行，而进程则可以跨机器迁移。

（三）、相关背景知识

1、进程和线程都是一个时间段的描述，是CPU工作时间段的描述，不过是颗粒大小不同。

- CPU+RAM+各种资源（比如显卡、GPU）构成我们的电脑，但是电脑的运行，实际就是CPU和相关寄存器以及RAM之间的事情。
- 一个最最基础的事实：CPU太快，太快，太快了，寄存器仅仅能够追的上他的脚步，RAM和别的挂在各总线上的设备完全是望其项背。那当多个任务要执行的时候怎么办呢？轮流着来?或者谁优先级高谁来？不管怎么样的策略，一句话就是在CPU就是轮流着来。
- 一个必须知道的事实：执行一段程序代码，实现一个功能的过程介绍，当得到CPU的时候，相关的资源必须也已经就位，就是显卡啊，GPU什么的必须就位，然后CPU开始执行。这里除了CPU以外所有的就构成了这个程序的执行环境，也就是我们所定义的程序上下文。当这个程序执行完了，或者分配给他的CPU执行时间用完了，那它就要被切换出去，等待下一次CPU的临幸。在被切换出去的最后一步工作就是保存程序上下文，因为这个是下次他被CPU临幸的运行环境，必须保存。
- 串联起来的事实：前面讲过在CPU看来所有的任务都是一个一个的轮流执行的，具体的轮流方法就是：先加载程序A的上下文，然后开始执行A，保存程序A的上下文，调入下一个要执行的程序B的程序上下文，然后开始执行B,保存程序B的上下文。。。。
- 进程就是包换上下文切换的程序执行时间总和 = CPU加载上下文 + CPU执行 + CPU保存上下文
- 线程是什么呢?
    - 进程的颗粒度太大，每次都要有上下的调入，保存，调出。如果我们把进程比喻为一个运行在电脑上的软件，那么一个软件的执行不可能是一条逻辑执行的，必定有多个分支和多个程序段，就好比要实现程序A实际分成 a，b，c等多个块组合而成。那么这里具体的执行就可能变成：程序A得到CPU -> CPU加载上下文，开始执行程序A的a小段，然后执行A的b小段，然后再执行A的c小段，最后CPU保存A的上下文。
    - 这里a，b，c的执行是共享了A的上下文，CPU在执行的时候没有进行上下文切换的。这里的a，b，c就是线程，也就是说线程是共享了进程的上下文环境的更为细小的CPU时间段。

2、操作系统的设计，可归结为三点：

- （1）以多进程形式，允许多个任务同时运行；
- （2）以多线程形式，允许单个任务分成不同的部分运行；
- （3）提供协调机制，一方面防止进程之间和线程之间产生冲突，另一方

面允许进程之间和线程之间共享资源。

三、轻量级线程——协程coroutine

（一）、什么是协程?

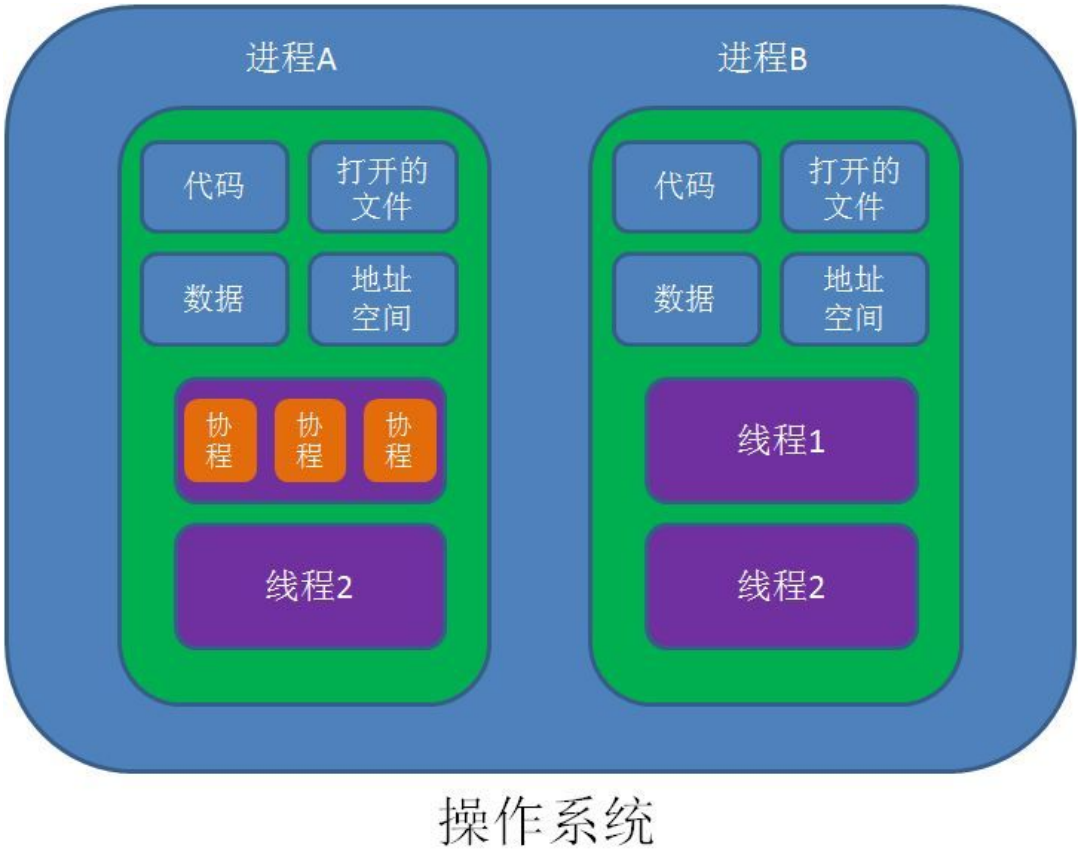
1、协程，英文Coroutine，有称为微线程。是一种比线程更加轻量级的存在。正如一个进程可以拥有多个线程一样，一个线程也可以拥有多个协程。协程最初在1963年被提出。

2、协程不是进程或线程，其执行过程更类似于子程序，或者说不带返回值的函数调用。

- 最重要的是，协程不是被操作系统内核所管理，而完全是由程序所控制。这样带来的好处就是性能得到了很大的提升，不会像线程切换那样消耗资源。
- Coroutine是编译器级的，Process和Thread是操作系统级的。

3、协程的优势

- 与传统的系统级线程和进程相比，协程的最大优势在于其轻量级，可以轻松创建上百万个，而不会导致系统资源衰竭；线程和进程通常最多也不能超过一万个。这也是协程叫做轻量级线程的原因。
- 协程的执行效率极高。因为子程序切换不是线程切换，而是由程序自身控制，因此，没有线程切换的开销。和多线程比，线程数量越多，协程的性能优势就越明显。



（二）、协程的应用

- 有哪些编程语言应用到了协程呢?

1、Lua语言

Lua从5.0版本开始使用协程，通过扩展库coroutine来实现。

2、Python语言

python可以通过 yield/send 的方式实现协程。在python 3.5以后，async/await 成为了更好的替代方案。

3、Go语言

Go语言对协程的实现非常强大而简洁，可以轻松创建成百上千个协程并发执行。

4、Java语言

Java语言并没有对协程的原生支持，但是某些开源框架模拟出了协程的功

能，例如：Kilim框架。

四、Go语言中协程——goroutine

（一）、什么是Goroutine?

1、在执行多任务时，为了保证每个任务能及时被分配到CPU上运行处理，同时避免多个任务频繁地在线程间切换执行而损失效率，常使用线程池；但是如果面对随时随地可能发生的并发和线程处理需求，线程池不是非常直观和方便；

2、使用者分配足够多的任务，系统能自动帮助使用者把任务分配到CPU上，让这些任务尽量并发运行。这种机制在GO语言中被称为Goroutine。Goroutine类似于线程，Goroutine由GO程序运行时runtime调度和管理，GO程序会智能地将Goroutine中的任务合理地分配给每个CPU。

3、Go程序从main包的main()函数开始，在程序启动时，Go程序会为main()函数创建一个默认的Goroutine。

（二）、Goroutine在线程上的优势

1、Go中使用Goroutine来实现并发concurrently。

2、Goroutine是与其他函数或方法同时运行的函数或方法。

3、Goroutine在线程上的优势

- 与线程相比，Goroutine非常便宜。它们只是堆栈大小的几个kb，堆栈可以根据应用程序的需要增长和收缩，而在线程的情况下，堆栈大小必须指定并且是固定的。由于创建Goroutine的成本很小。因此，Go应用程序可以并发运行数千个Goroutine。
- 当使用Goroutine访问共享内存时，通过设计的通道可以防止竞态条件发生。通道可以被认为是Goroutine通信的管道。

（三）、Coroutine与Goroutine

1、Coroutine协程

- C#、Python都支持Coroutine。

2、Coroutine与Goroutine比较：

都可以将函数或者语句在独立的环境中运行，但是他们之间存在两点不同：

（1）、goroutine可能发生并行执行；但是coroutine始终顺序执行；

- goroutine可能发生在多线程环境下，coroutine始终发生在单线程，coroutine程序需要主动交出控制权，宿主才能获得控制权并将控制权交给其他coroutine。
- coroutine的运行机制属于协作式任务处理，应用程序在不需要使用CPU时，需要主动交出CPU使用权。如果开发者无意间让应用程序长时间占用CPU，操作系统也无能为力，计算机很容易失去响应或者死机。
- goroutine属于抢占式任务处理，已经和现有的多线程和多进程任务处理非常类似。应用程序对CPU的控制最终需要有操作系统来管理，操作系统如何发现一个应用程序常时间占用CPU，那么用户有权终止这个任务。

（2）、goroutine间使用channel通信；coroutine使用yield和resume操作。

（四）、使用普通函数创建goroutine

1、在函数或方法调用前面加上关键字go，将会同时运行一个新的Goroutine。

- 使用go关键字创建Goroutine时，被调用的函数往往没有返回值，如果有返回值也会被忽略。
- 如果需要在Goroutine中返回数据，必须使用通道channel，通过通道把数据从Goroutine中作为返回值传出。

2、示例代码1：

```
package main
import (
  "fmt"
)

func hello() {
  fmt.Println("Hello world goroutine")
}

func main() {
  go hello()
  //time.Sleep(50 * time.Microsecond)
  fmt.Println("main function")
}
```

运行结果：

main function

3、代码分析

- Go程序的执行过程是：创建和启动主goroutine，初始化操作，执行main()函数，当main()函数结束，主goroutine随之结束，程序结束。
- 被启动的goroutine叫做子goroutine。
- 如果main的goroutine终止了，程序将被终止，而其他Goroutine将不再运行。换句话说，所有goroutine在main()函数结束时会一同结束。

4、修改后示例代码

```
func hello() {
  fmt.Println("Hello world goroutine")
}

func main() {
  go hello()
  time.Sleep(50 * time.Microsecond)
  fmt.Println("main function")
}
```

运行结果：

Hello world goroutine

main function

- 在上面的程序中，我们已经调用了时间包的Sleep方法，它会在执行过程中睡觉。在这种情况下，main的goroutine被用来睡觉50毫秒。现在调用go hello()有足够的时间在main Goroutine终止之前执行。这个程序首先打印Hello world goroutine，等待50毫秒，然后打印main函数。

5、示例代码2：

```
func main() {
  go running()
  var input string
  fmt.Scanln(&input)
}
```

```
func running() {
  var times int
  for {
    times++
    fmt.Println("tick", times)
    time.Sleep(time.Second)
  }
}
```

- 程序运行效果。控制台不断输出tick，同时还可以接收用户输入。两个环节同时运行。
- 该案例中，Go程序在启动时，运行时runtime默认为main()函数创建一个goroutine。在main()函数的goroutine执行到go running()语句时，归属于running()函数的goroutine被创建，running()函数开始在自己的goroutine中执行。
- 此时，main()继续执行，两个goroutine通过GO程序的调度机制同时运行。

（五）、使用匿名函数创建goroutine

1、go关键字后也可以是匿名函数或闭包。

2、示例代码：

```
func main() {
  go func() {
    var times int
    for  {
      times++
      fmt.Println("tick" , times)
      time.Sleep(time.Second)
    }
  }()
  var input string
  fmt.Scanln(&input)
}
```

（六）、启动多个Goroutines

1、示例代码：

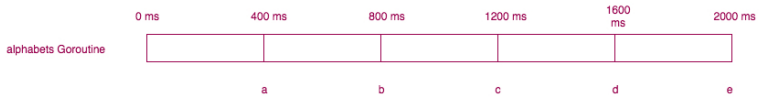
```
func numbers() {
  for i := 1; i <= 5; i++ {
    time.Sleep(250 * time.Millisecond)
    fmt.Printf("%d ", i)
  }
}
func alphabets() {
  for i := 'a'; i <= 'e'; i++ {
    time.Sleep(400 * time.Millisecond)
    fmt.Printf("%c ", i)
  }
}
func main() {
  go numbers()
  go alphabets()
  time.Sleep(3000 * time.Millisecond)
  fmt.Println("\n main terminated")
}
```

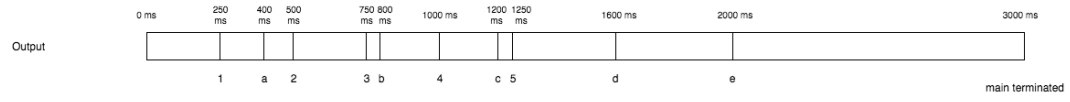
运行结果：

1 a 2 3 b 4 c 5 d e

main terminated

时间轴分析：





（七）、调整并发的运行性能Gomaxprocs

1、在Go程序运行时runtime实现了一个小型的任务调度器。这套调度器的工作原理类似于操作系统调度线程，Go程序调度器可以高效地将CPU资源分配给每一个任务。

传统逻辑中，开发者需要维护线程池中的线程与CPU核心数量的对应关系。在Go语言中可以通过runtime.GOMAXPROCS()函数做到。

语法为：runtime.GOMAXPROCS(逻辑CPU数量)

2、逻辑CPU数量有如下几种数值：

- <1 ，不修改任何数值；
- =1，单核执行；
- >1，多核并发执行。
- Go1.5版本之前，默认使用单核执行。Go1.5版本开始，默认执行：runtime.GOMAXPROCS(逻辑CPU数量)，让代码并发执行，最大效率地利用CPU。