并发编程

作者：少林之巅

目录

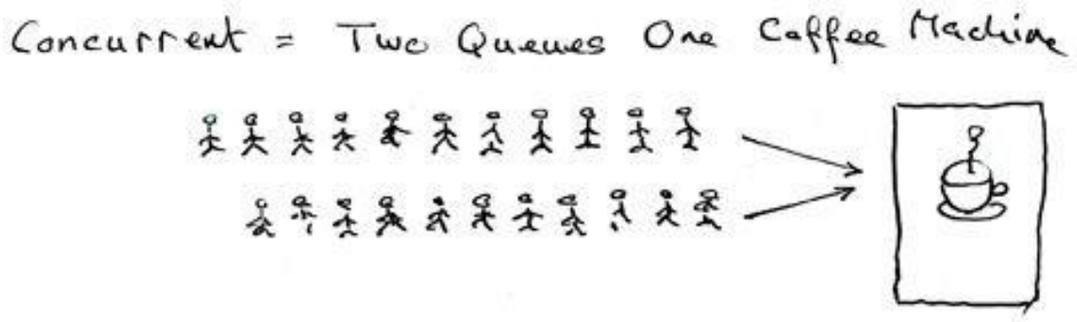
并发和并行

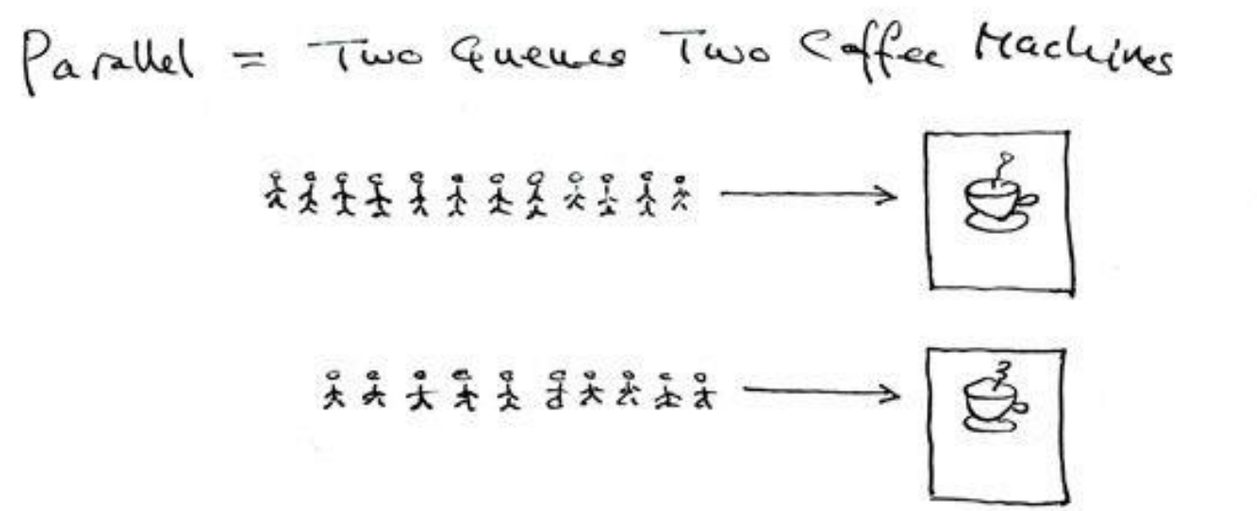
1. 概念

   A. 并发：同一时间段内执行多个操作。

   B. 并行：同一时刻执行多个操作。

并发和并行





3. 多线程

A. 线程是由操作系统进行管理，也就是处于内核态。

B. 线程之间进行切换，需要发生用户态到内核态的切换。

C. 当系统中运行大量线程，系统会变的非常慢。

D. 用户态的线程，支持大量线程创建。也叫协程或goroutine。

Goroutine使用介绍

4. 创建goroutine

```
package main

import (
    "fmt"
)

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

5. 修复代码

```
package main

import (
    "fmt“
    “time”
)

func hello() {
    fmt.Println("Hello world goroutine")
}
func main() {
    go hello()
    time.Sleep(1*time.Second)
    fmt.Println("main function")
}
```

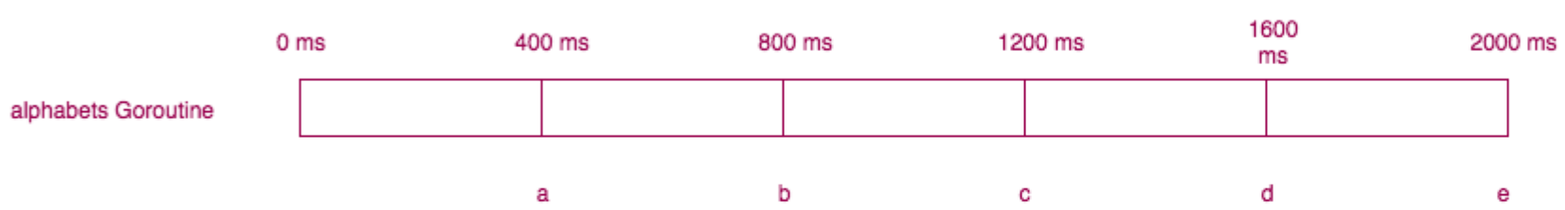
Goroutine使用介绍

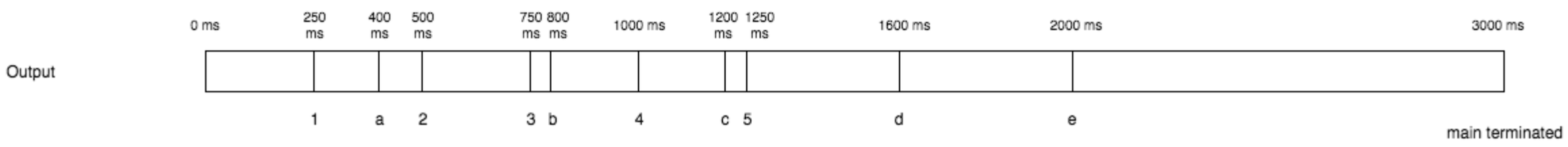
6. 启动多个goroutine

```
package main

import (
    "fmt"
    "time"
)

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("main terminated")
}
```

Goroutine使用介绍

7. 程序分析





Goroutine使用介绍

8. 多核控制

   A. 通过runtime包进行多核设置

   B. GOMAXPROCS设置当前程序运行时占用的cpu核数

   C. NumCPU获取当前系统的cpu核数

9. Goroutine原理浅析

   A. 一个操作系统线程对应用户态多个goroutine

   B. 同时使用多个操作系统线程

   C. 操作系统线程对goroutine是多对多关系，即M:N

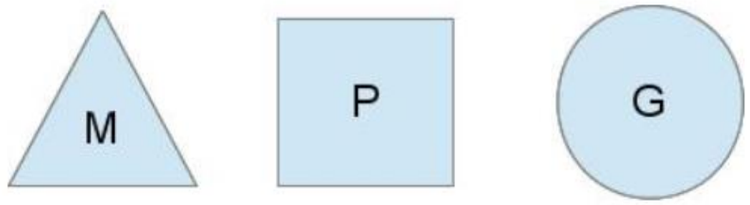
Goroutine原理浅析

10. 模型抽象

A. 操作系统线程： M

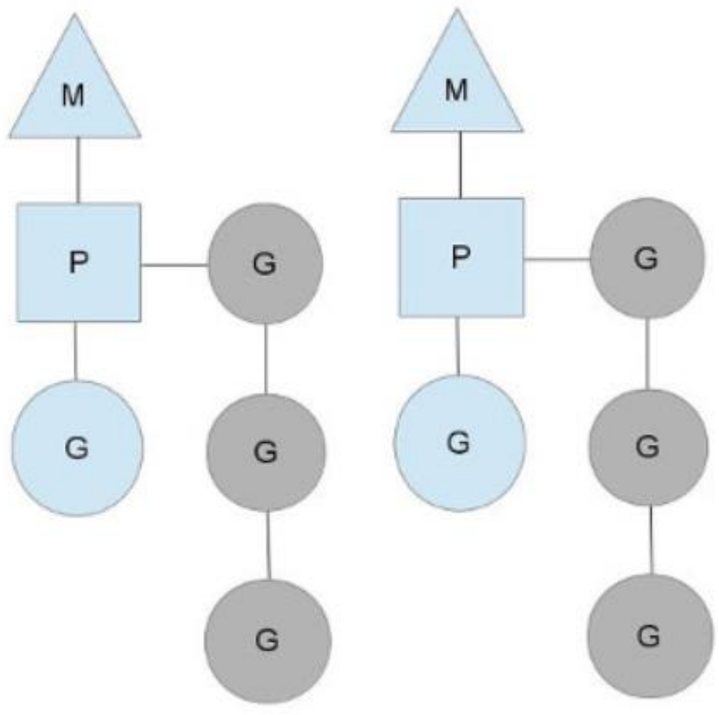
B. 用户态线程（goroutine）: G

C. 上下文对象：P

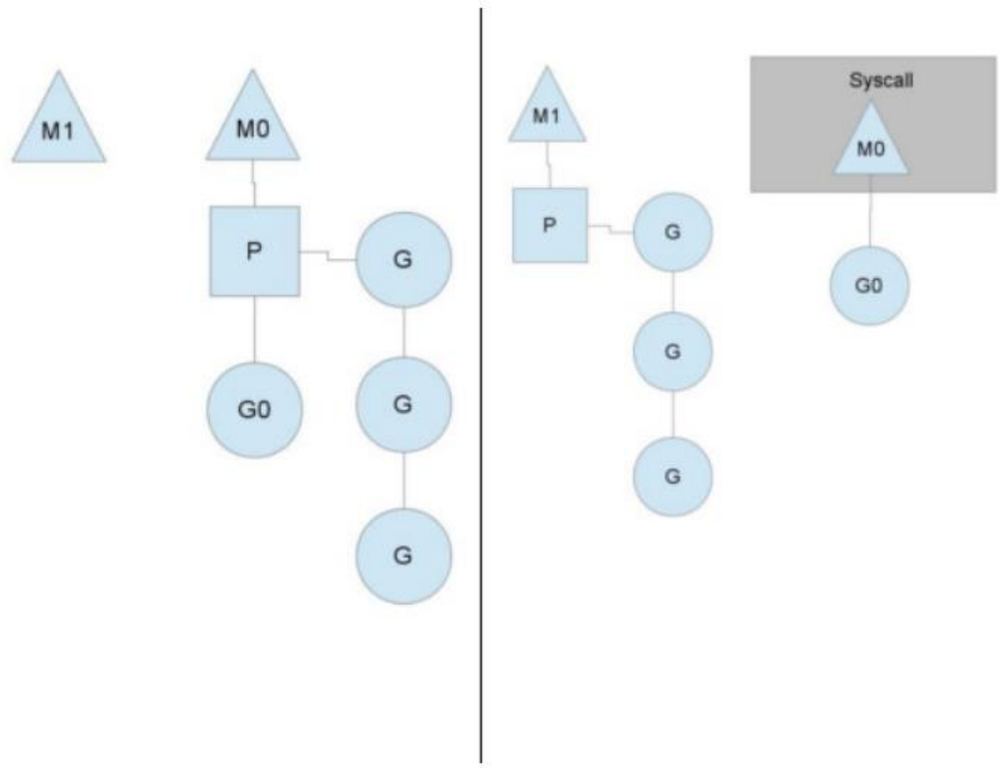


Goroutine原理浅析

11. goroutine调度



12. 系统调用怎么处理



Channel介绍

13. channel介绍

    A. 本质上就是一个队列，是一个容器

    B. 因此定义的时候，需要只定容器中元素的类型

    C. var 变量名 chan 数据类型

Channel介绍

```
package main

import "fmt"

func main() {
    var a chan int
    if a == nil {
        fmt.Println("channel a is nil, going to define it")
        a = make(chan int)
        fmt.Printf("Type of a is %T", a)
    }
}
```

14. 元素入队和出队

```
var a chan int
```

A. 入队操作，a <- 100

B. 出队操作：data := <- a

Channel介绍

15. 阻塞chan

```
package main

import "fmt"

func main() {
    var a chan int
    if a == nil {
        fmt.Println("channel a is nil, going to define it")
        a = make(chan int)
        a <- 10
        fmt.Printf("Type of a is %T", a)
    }
}
```

Channel介绍

16. 使用chan来进行goroutine同步

```
package main

import (
    "fmt"
)

func hello(done chan bool) {
    fmt.Println("Hello world goroutine")
    done <- true
}
func main() {
    done := make(chan bool)
    go hello(done)
    <-done
    fmt.Println("main function")
}
```

Channel介绍

17. 使用chan来进行goroutine同步

```
package main

import (
    "fmt"
    "time"
)

func hello(done chan bool) {
    fmt.Println("hello go routine is going to sleep")
    time.Sleep(4 * time.Second)
    fmt.Println("hello go routine awake and going to write to done")
    done <- true
}
func main() {
    done := make(chan bool)
    fmt.Println("Main going to call hello go goroutine")
    go hello(done)
    <-done
    fmt.Println("Main received data")
}
```

Channel介绍

18. 单向chan

```
package main

import "fmt"

func sendData(sendch chan<- int) {
    sendch <- 10
}


func readData(sendch <-chan int) {
    sendch <- 10
}

func main() {
    chnl := make(chan int)
    go sendData(chnl)
    readData(chn1)
}
```

Channel介绍

19. chan关闭

```
package main

import (
    "fmt"
)

func producer(chnl chan int) {
    for i := 0; i < 10; i++ {
        chnl <- i
    }
    close(chnl)
}
func main() {
    ch := make(chan int)
    go producer(ch)
    for {
        v, ok := <-ch
        if ok == false {
            break
        }
        fmt.Println("Received ", v, ok)
    }
}
```

Channel介绍

20. for range操作

```
package main

import (
    "fmt"
)

func producer(chnl chan int) {
    for i := 0; i < 10; i++ {
        chnl <- i
    }
    close(chnl)
}
func main() {
    ch := make(chan int)
    go producer(ch)
    for v := range ch {
        fmt.Println("Received ",v)
    }
}
```

Channel介绍

21. 带缓冲区的chanel

A. Ch := make(chan type, capacity)

```
package main

import (
    "fmt"
)


func main() {
    ch := make(chan string, 2)
    ch <- “hello"
    ch <- “world"
    fmt.Println(<- ch)
    fmt.Println(<- ch)
}
```

Channel介绍

```
package main

import (
    "fmt"
    "time"
)

func write(ch chan int) {
    for i := 0; i < 5; i++ {
        ch <- i
        fmt.Println("successfully wrote", i, "to ch")
    }
    close(ch)
}
func main() {
    ch := make(chan int, 2)
    go write(ch)
    time.Sleep(2 * time.Second)
    for v := range ch {
        fmt.Println("read value", v,"from ch")
        time.Sleep(2 * time.Second)

    }
}
```

Channel介绍

22. channel的长度和容量

A. Ch := make(chan type, capacity)

```
package main

import (
    "fmt"
)

func main() {
    ch := make(chan string, 3)
    ch <- "naveen"
    ch <- "paul"
    fmt.Println("capacity is", cap(ch))
    fmt.Println("length is", len(ch))
    fmt.Println("read value", <-ch)
    fmt.Println("new length is", len(ch))
}
```

Waitgroup介绍

23. 如何等待一组goroutine结束?

A. 方法一，使用**不带缓冲区**的channel实现

```
package main
import (
    "fmt"
    "time"
)
func process(i int, ch chan bool) {
    fmt.Println("started Goroutine ", i)
    time.Sleep(2 * time.Second)
    fmt.Printf("Goroutine %d ended\n", i)
    ch <- true
}
func main() {
    no := 3
    exitChan := make(chan bool, no)
    for i := 0; i < no; i++ {
        go process(i, exitChan)
    }
    for i := 0; I < no;i++{
        <-exitChan
    }
    fmt.Println("All go routines finished executing")
}
```

Waitgroup介绍

24. 如何等待一组goroutine结束?

B. 方法二，使用sync.WaitGroup实现

```
package main

import (
    "fmt"
    "sync"
    "time"
)
func process(i int, wg *sync.WaitGroup) {
    fmt.Println("started Goroutine ", i)
    time.Sleep(2 * time.Second)
    fmt.Printf("Goroutine %d ended\n", i)
    wg.Done()
}
func main() {
    no := 3
    var wg sync.WaitGroup
    for i := 0; i < no; i++ {
        wg.Add(1)
        go process(i, &wg)
    }
    wg.Wait()
    fmt.Println("All go routines finished executing")
}
```

Workerpool的实现

25. worker池的实现

    A. 生产者、消费者模型，简单有效

    B. 控制goroutine的数量，防止goroutine泄露和暴涨

    C. 基于goroutine和chan，构建workerpool非常简单

Workerpool的实现

26. 项目需求分析

A. 计算一个数字的各个位数之和，比如123，和等于1+2+3=6

B. 需要计算的数字使用随机算法生成

Workerpool的实现

27. 方案介绍

A. 任务抽象成一个个job

B. 使用job队列和result队列

C. 开一组goroutine进行实际任务计算，并把结果放回result队列