Go培训第八天

tony

Outline

1. Goroute

2. Channel

3. 单元测试

4. 课后作业

Goroutine

1. 进程和线程

A. 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。

B. 线程是进程的一个执行实体,是CPU调度和分派的基本单位,它是比进程更小的能独立运行的基本单位。

C.一个进程可以创建和撤销多个线程;同一个进程中的多个线程之间可以并发执行.

Goroutine

2. 进程和线程

Goroutine

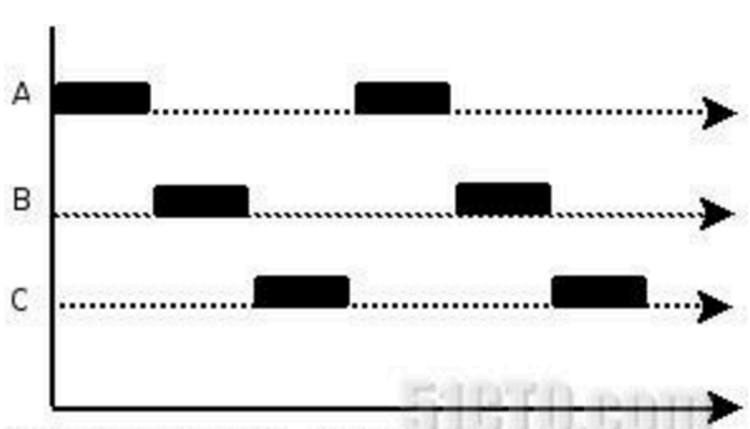
3. 并发和并行

A. 多线程程序在一个核的cpu上运行，就是并发

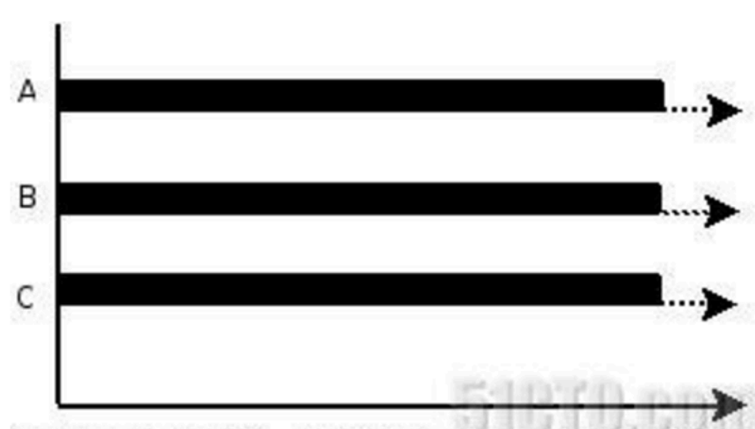
B. 多线程程序在多个核的cpu上运行，就是并行

Goroutine

4. 并发和并行



Concurrency : 1. Single Processor



Parallelism : 1. Multiprocessores, Multicore

Goroutine

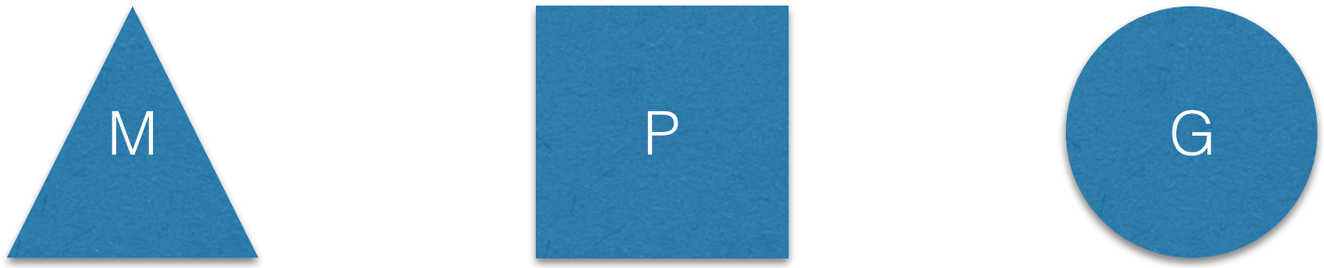
5. 协程和线程

协程：独立的栈空间，共享堆空间，调度由用户自己控制，本质上有点类似于用户级线程，这些用户级线程的调度也是自己实现的

线程：一个线程上可以跑多个协程，协程是轻量级的线程。

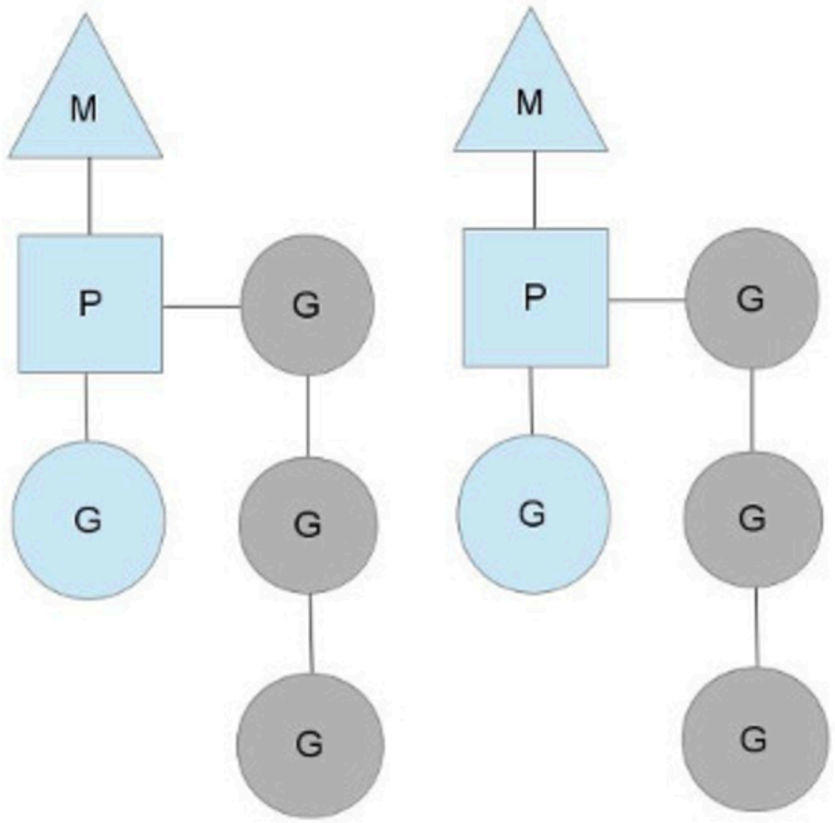
Goroutine

6. goroutine调度模型



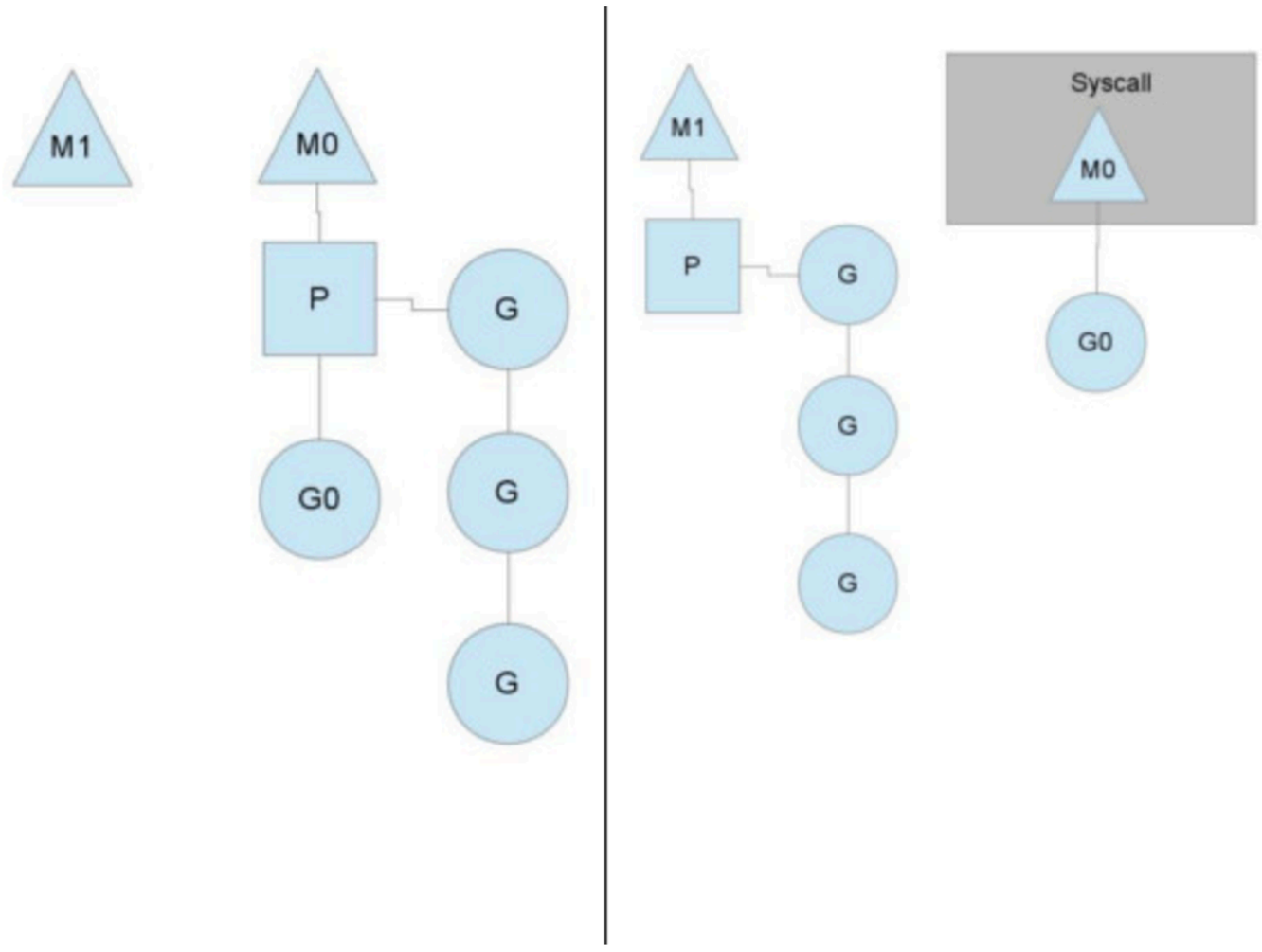
Goroutine

7. goroutine调度模型



Goroutine

8. goroutine调度模型



Goroutine

9. 如何设置golang运行的cpu核数

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

func main() {
	num := runtime.NumCPU()
	runtime.GOMAXPROCS(num)
	fmt.Println(num)
}
```

Goroutine

11. goroute练习

Channel

1. 不同goroutine之间如何进行通讯?

    a. 全局变量和锁同步

    b. Channel

Channel

2. channel概念

a. 类似unix中管道（pipe）

b. 先进先出

c. 线程安全，多个goroutine同时访问，不需要加锁

d. channel是有类型的，一个整数的channel只能存放整数

Channel

3. channel声明

var 变量名 chan 类型

var test chan int

var test chan string

var test chan map[string]string

var test chan stu

var test chan *stu

4. channel初始化

使用make进行初始化，比如：

```
var test chan int
test = make(chan int, 10)

var test chan string
test = make(chan string, 10)
```

Channel

5. channel基本操作

1. 从channel读取数据:

```
var testChan chan int
testChan = make(chan int, 10)
var a int
a = <- testChan
```

2. 从channel写入数据:

```
var testChan chan int
testChan = make(chan int, 10)
var a int  = 10
testChan <- a
```

6. goroutine和channel相结合

```
package main

import (
	"fmt"
	"time"
)

func main() {
	ch := make(chan string)
	go sendData(ch)
	go getData(ch)
	time.Sleep(100 * time.Second)
}

func sendData(ch chan string) {
	ch <- "Washington"
	ch <- "Tripoli"
	ch <- "London"
	ch <- "Beijing"
	ch <- "Tokio"
}

func getData(ch chan string) {
	var input string
	for {
		input = <-ch
		fmt.Println(input)
	}
}
```

7. channel阻塞

```
package main

import (
	"fmt"
	"time"
)

func main() {
	ch := make(chan string)
	go sendData(ch)
	time.Sleep(100 * time.Second)
}

func sendData(ch chan string) {
	var i int
	for {
		var str string
		str = fmt.Sprintf("stu %d", i)
		fmt.Println("write:", str)
		ch <- str
		i++
	}
}
```

Channel

8. 带缓冲区的channel

1. 如下所示，testChan只能放一个元素：

```
var testChan chan int
testChan = make(chan int)
var a int
a = <- testChan
```

2. 如下所示，testChan是带缓冲区的chan，一次可以放10个元素:

```
var testChan chan int
testChan = make(chan int, 10)
var a int  = 10
testChan <- a
```

9. chan之间的同步

```
package main

import (
	"fmt"
	"time"
)

func main() {
	ch := make(chan string)
	go sendData(ch)
	go getData(ch)
	time.Sleep(100 * time.Second)
}

func sendData(ch chan string) {
	ch <- "Washington"
	ch <- "Tripoli"
	ch <- "London"
	ch <- "Beijing"
	ch <- "Tokio"
}

func getData(ch chan string) {
	var input string
	for {
		input = <-ch
		fmt.Println(input)
	}
}
```

10. for range遍历chan

```
package main

import (
	"fmt"
	"time"
)

func main() {
	ch := make(chan string)
	go sendData(ch)
	go getData(ch)
	time.Sleep(100 * time.Second)
}

func sendData(ch chan string) {
	ch <- "Washington"
	ch <- "Tripoli"
	ch <- "London"
	ch <- "Beijing"
	ch <- "Tokio"
}

func getData(ch chan string) {

	for input  := range ch {
		fmt.Println(input)
	}
}
```

11. chan的关闭

    1. 使用内置函数close进行关闭，chan关闭之后，for range遍历chan中已经存在的元素后结束

    2. 使用内置函数close进行关闭，chan关闭之后，没有使用for range的写法需要使用，v, ok := <- ch进行判断chan是否关闭

12. chan的只读和只写

a. 只读chan的声明

Var 变量的名字 <-chan int

Var readChan <- chan int

b. 只写chan的声明

Var 变量的名字 chan<- int

Var writeChan chan<- int

12. 对chan进行select操作

```
Select {
    case u := <- ch1:
    case e := <- ch2:
    default:
}
```

13. 练习

```
Select {
    case u := <- ch1:
    case e := <- ch2:
    default:
}
```

14. 定时器的使用

```
package main

import (
	"fmt"
	"time"
)

func main() {
	t := time.NewTicker(time.Second)
	for v := range t.C {
		fmt.Println("hello, ", v)
	}
}
```

15. 一次定时器

```
package main

import (
        "fmt"
        "time"
)

func main() {
        select {
         Case <- time.After(time.Second):
          fmt.Println(“after”)
         }
}
```

16. 超时控制

```
package main

import (
	"fmt"
	"time"
)
func queryDb(ch chan int) {

	time.Sleep(time.Second)
	ch <- 100
}
func main() {
	ch := make(chan int)
	go queryDb(ch)
	t := time.NewTicker(time.Second)

	select {
	case v := <-ch:
		fmt.Println("result", v)
	case <-t.C:
		fmt.Println("timeout")
	}
}
```

17. goroutine中使用recover

应用场景，如果某个goroutine panic了，而且这个goroutine里面没有捕获(recover)，那么整个进程就会挂掉。所以，好的习惯是每当go产生一个goroutine，就需要写下recover

单元测试

1. 文件名必须以_test.go结尾

2. 使用go test执行单元测试

3. 练习，编写单元测试用例：

课后工作

1. 完善之前讲的图书管理系统，使各个功能正确无误.