协程Coroutine本质

先看Python的一个函数

```
1  def count():
2      c = 1
3      for i in range(5):
4          print(c)
5          c += 1
6
7  count()
8  print("@@@")
9
10 运行结果
11 1
12 2
13 3
14 4
15 5
16 @@@
```

上例中，第7行是函数调用，必须等其调用结束后返回了，才能执行第8行代码，否则要一直等count函数执行。

在count函数中增加一个yield语句

```
1  def count():
2      c = 1
3      for i in range(5):
4          print(c)
5          yield c
6          c += 1
7
8  count()
9  print("@@@")
10
11 运行结果
12 @@@
```

发现count()没有了输出，能打印@@@，说明count()确实执行过了。这是因为在Python中含有yield关键字的函数是一种特殊函数，称为生成器函数。count()调用返回的将不再是执行到函数return的结果，而是返回一个生成器对象即迭代器对象。

生成器对象

- 就是迭代器对象，不过是特殊语法构造出的迭代器对象
- 也可以使用next函数驱动它执行，但执行到yield就**暂停**函数执行
- 可以使用for循环迭代它，相当于连续的next，直到不可迭代为止
- 只能单向向后迭代，不可以重头开始

```
1  def count():
2      c = 1
```

```
    for i in range(5):
        print(c)
        yield c
        print("###")
        c += 1

t = count() # 迭代器对象
next(t)
print("@@@")

输出结果
1
@@@
```

执行第10行输出结果为1，说明函数在第5行处**暂停**执行了（实际上这个函数**没有执行完**），且能继续向下执行到11行，打印了3个@。

如果有2个生成器函数，试着分析一下，代码如下

```
import string


def count():
    c = 1
    for i in range(5):
        print(c)
        yield c
        print("###")
        c += 1

def char():
    s = string.ascii_lowercase
    for c in s:
        print(c)
        yield c

t1 = count() # 迭代器对象
t2 = char() # 迭代器对象
next(t1)
next(t1)
next(t1)
next(t2)
print("@@@")
```


可以看出代码在yield出暂停，通过next来驱动各个函数执行，可以由程序员在合适的地方通过yield来暂停一个函数执行，让另外一个函数执行。

问题：

1. 请问目前代码中有几个线程?
2. 有没有实现和线程切换导致函数切换执行的效果?

暂停是一种非常重要的能力，以前函数正常要执行到return后，现在可以由开发者控制暂停执行的时机。而线程时间片用完导致的函数切换对开发人员来说是不可控的，而且线程控制能力是内核的功能，是在内核态完成的，而上例（协程）的控制是在用户态完成的。

如何才能让上例中所有任务反复交替执行呢?

1. 构建一个循环
2. 构建一个任务列表，循环执行其中的任务们

```
import string
import time

def count():
    c = 1
    for i in range(5):
        print(c)
        yield c
        print("###")
        c += 1

def char():
    s = string.ascii_lowercase
    for c in s:
        print(c)
        yield c

t1 = count()  # 迭代器对象
t2 = char()  # 迭代器对象

tasks = [t1, t2]
while True:
    pops = [] # 待移除的已经完成的任务
    for i, task in enumerate(tasks):
        if next(task, None) is None: # 如果迭代到头了，返回给定的缺省值
            print("task {} finished.".format(task))
            pops.append(i) # 记住索引
    for i in reversed(pops):
        tasks.pop(i)
    print(len(tasks), tasks)
    if len(tasks) == 0:
        time.sleep(1) # 如果任务列表为0，就等待

print("@@@")
```

可以通过上面的代码看到2个任务交替进行，而这个函数的交替，完全是靠程序员的代码实现的，而不是靠多线程的时间片用完操作系统强行切换，而且这种切换是在同一个线程中完成的。

最重要的是，协程的切换是在用户态完成，而不是像线程那样在内核态完成。所以，Coroutine是可以在用户态通过控制在适当的时机让出执行权的多任务切换技术。

上例中，交替执行任务是可以由程序员在一个线程内完成，这个任务如果再被按照Python语法封装后就是Python的协程。核心点是，在适当的时候要暂停一个正在运行的任务，让出来去执行另外一个任务。

注意：只要是代码就要在线程中执行，协程也不例外。

问题：有了协程，还会不会出现线程的切换？

协程弊端

- 一旦一个协程阻塞，阻塞了什么？阻塞当前所在线程？那么该线程代码被阻塞不能向下继续执行了
- 协程必须主动让出，才能轮到该线程中另外一个协程运行

能否让协程自由的在不同线程中移动，这样就不会阻塞某一个线程到导致线程中其他协程得不到执行？

Go语言对Coroutine做了非常多的优化，提出了Goroutine。

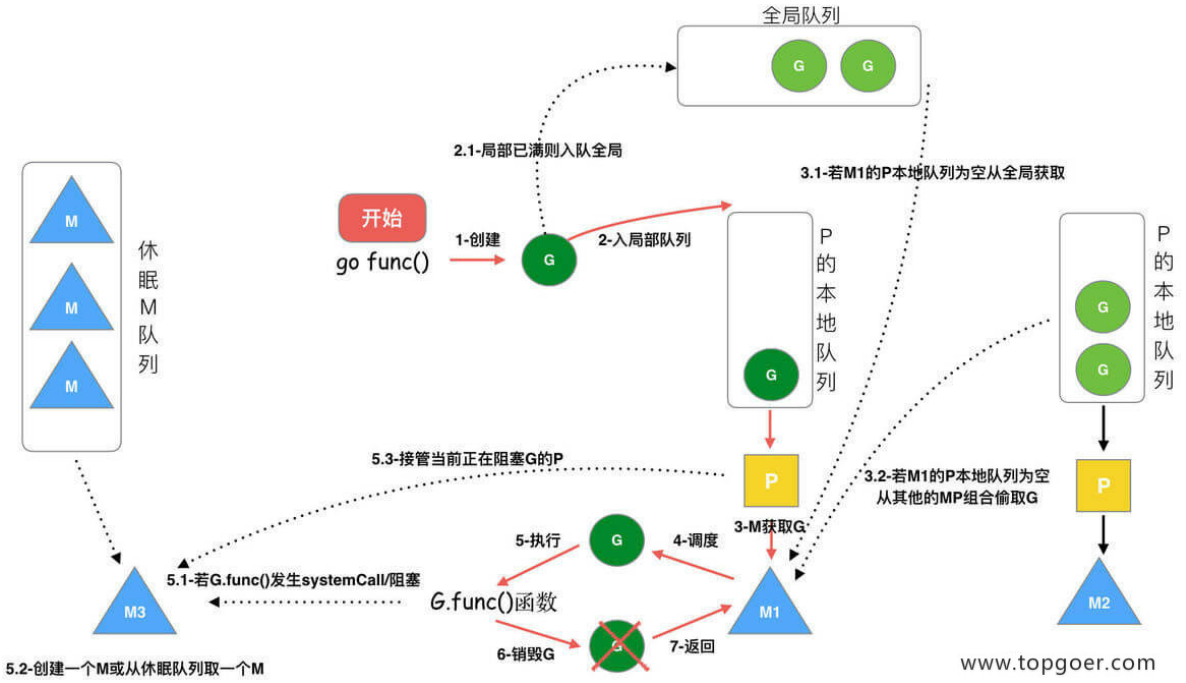
GMP模型

Robert Griesemer、Rob Pike、Ken Thompson三位Go语言创始人，对新语言商在讨论时，就决定了要让Go语言成为面向未来的语言。当时多核CPU已经开始普及，但是众多“古老”编程语言却不能很好的适应新的硬件进步，Go语言诞生之初就为多核CPU并行而设计。

Go语言协程中，非常重要的就是协程调度器scheduler和网络轮询器netpoller。

Go协程调度中，有三个重要角色：

- M：Machine Thread，对系统线程抽象、封装。所有代码最终都要在系统线程上运行，协程最终也是代码，也不例外
- G：Goroutine，Go协程。存储了协程的执行栈信息、状态和任务函数等。初始栈大小约为2~4k，理论上开启百万个Goroutine不是问题
- P：Go1.1版本引入，Processor，虚拟处理器
    - 可以通过环境变量GOMAXPROCS或runtime.GOMAXPROCS()设置，默认为CPU核心数
    - P的数量决定着最大可并行的G的数量
    - P有自己的队列（长度256），里面放着待执行的G
    - M和P需要绑定在一起，这样P队列中的G才能真正在线程上执行



1、使用go func创建一个Goroutine g1

2、当前P为p1，将g1加入当前P的本地队列LRQ(Local Run Queue)。如果LRQ满了，就加入到GRQ（Global Run Queue）

3、p1和m1绑定，m1先尝试从p1的LRQ中请求G。如果没有，就从GRQ中请求G。如果还没有，就随机从别的P的LRQ中偷（work stealing）一部分G到本地LRQ中。

4、假设m1最终拿到了g1

5、执行，让g1的代码在m1线程上运行

5.1、g1正常执行完了（函数调用完成了），g1和m1解绑，执行第3步的获取下一个可执行的g

5.2、g1中代码主动让出控制权，g1和m1解绑，将g1加入到GRQ中，执行第3步的获取下一个可执行的g

5.3、g1中进行channel、互斥锁等操作进入阻塞态，g1和m1解绑，执行第3步的获取下一个可执行的g。如果阻塞态的g1被其他协程g唤醒后，就尝试加入到唤醒者的LRQ中，如果LRQ满了，就连同g和LRQ中一半转移到GRQ中。

5.4、系统调用

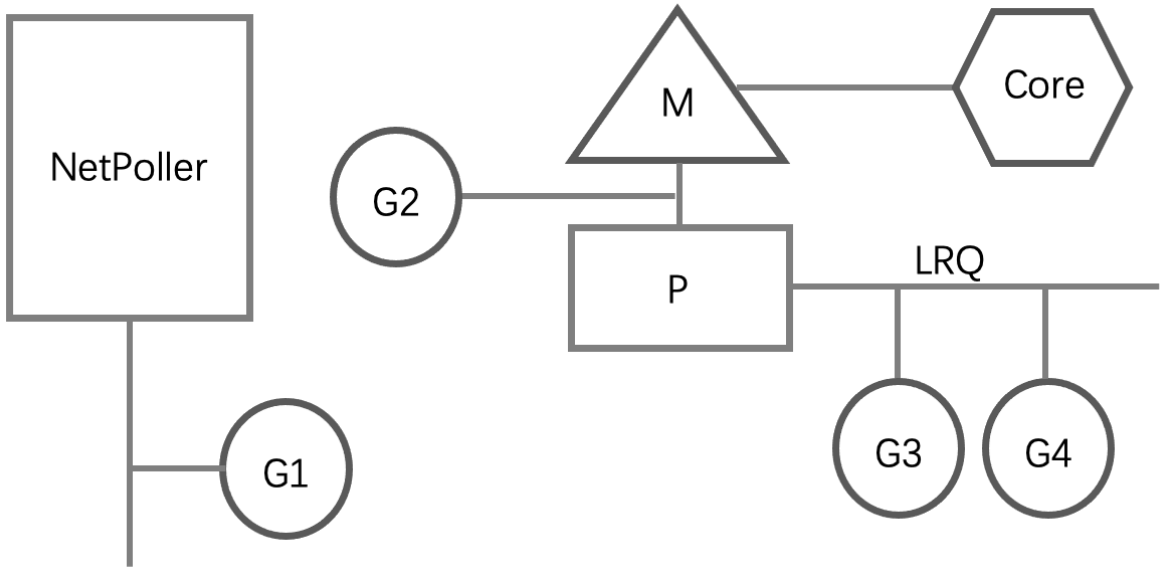
① **同步**系统调用时，执行如下

如果遇到了同步阻塞系统调用，g1阻塞，m1也被阻塞了，m1和p1解绑。

从休眠线程队列中获取一个空闲线程，和p1绑定，并从p1队列中获取下一个可执行的g来执行；如果休眠队列中无空闲线程，就创建一个线程提供给p1。

如果m1阻塞结束，需要和一个空闲的p绑定，优先和原来的p1绑定。如果没有空闲的P，g1会放到GRQ中，m1加入到休眠线程队列中。

② **异步**网络IO调用时，如下



网络IO代码会被Go在底层变成非阻塞IO，这样就可以使用IO多路复用了。

m1执行g1，执行过程中发生了非阻塞IO调用（读/写）时，g1和m1解绑，g1会被网络轮询器Netpoller接手。m1再从p1的LRQ中获取下一个Goroutine g2执行。注意，m1和p1不解绑。

g1等待的IO就绪后，g1从网络轮询器移回P的LRQ（本地运行队列）或全局GRQ中，重新进入可执行状态。

就大致相当于网络轮询器Netpoller内部就是使用了IO多路复用和非阻塞IO，类似我们课件代码中的select的循环。GO对不同操作系统MAC（kqueue）、Linux（epoll）、Windows（iocp）提供了支持。

问题：如果GOMAXPROCS为1，说明什么?

Go TCP编程

和第一天我们讲的Python TCP编程一样，只是函数略有不同。

需要用到net库。

```go
package main

import (
    "log"
    "net"
)

func main() {
    laddr, err := net.ResolveTCPAddr("tcp4", "0.0.0.0:9999") // 解析地址
    if err != nil {
        log.Panicln(err) // Panicln会打印异常，程序退出
    }
    server, err := net.ListenTCP("tcp4", laddr)
    if err != nil {
        log.Panicln(err)
```

```
    }
    defer server.Close() // 保证一定关闭

    conn, err := server.Accept() // 接收连接，分配socket
    if err != nil {
        log.Panicln(err)
    }

    defer conn.Close() // 保证一定关闭

    buffer := make([]byte, 4096) // 设置缓冲区
    n, err := conn.Read(buffer)  // 成功返回接收了多少字节
    if err != nil {
        log.Panicln(err)
    }
    data := buffer[:n]
    conn.Write(data) // 原样写回客户端
}
```

Goroutine

协程创建

使用go关键字就可以把一个函数定义为一个协程，非常方便。

先看下面的代码

```
package main

import "fmt"

func add(x, y int) int {
    var c int
    defer fmt.Printf("1 return %d\n", c)              // 打印的c是什么？
    defer func() { fmt.Printf("2 return %d\n", c) }() // 打印的c是什么？
    fmt.Printf("add called: x=%d, y=%d\n", x, y)
    c = x + y
    return c
}

func main() {
    fmt.Println("main start")
    add(4, 5)
    fmt.Println("main end")
}

执行结果如下
main start
add called: x=4, y=5
2 return 9
1 return 0
main end
```

将`add(4, 5)`改为`go add(4, 5)`，运行结果会怎么样呢？

```go
package main

import (
    "fmt"
    "runtime"
    "time"
)

func add(x, y int) int {
    var c int
    defer fmt.Printf("1 return %d\n", c)              // 打印的c是什么?
    defer func() { fmt.Printf("2 return %d\n", c) }() // 打印的c是什么?
    fmt.Printf("add called: x=%d, y=%d\n", x, y)
    c = x + y
    return c
}

func main() {
    fmt.Println(runtime.NumGoroutine())
    fmt.Println("main start")
    go add(4, 5) // 协程
    fmt.Println(runtime.NumGoroutine())
    // time.Sleep(2 * time.Second) // 放开这一句，看看效果
    fmt.Println("main end")
    fmt.Println(runtime.NumGoroutine())
}
```

如果没有`time.Sleep(2)`，结果如下

```
1
main start
2
main end
2
```

放开了`time.Sleep(2)`，结果如下

```
1
main start
2
add called: x=4, y=5
2 return 9
1 return 0
main end
1 注意这里是1了
```

为什么?

因为会启动协程来运行add，那么go add(4, 5)这一句没有必要等到函数返回才结束，所以程序执行下一行打印Main Exit。这时main函数无事可做，Go程序启动时也创建了一个协程，main函数运行其中，可以称为main goroutine（主协程）。但是主协程一旦执行结束，则进程结束，根本不会等待未执行完的其它协程。

那么，除了像`time.Sleep(2)`这样一直等，如何才能让主线程优雅等待协程执行结束呢？等待组

等待组

使用参考 https://pkg.go.dev/sync#WaitGroup

使用等待组修改上例

```go
package main

import (
    "fmt"
    "runtime"
    "sync"
)

func add(x, y int, wg *sync.WaitGroup) int {
    defer wg.Done() // add执行完后计数器减1
    var c int
    defer fmt.Printf("1 return %d\n", c)              // 打印的c是什么？
    defer func() { fmt.Printf("2 return %d\n", c) }() // 打印的c是什么？
    fmt.Printf("add called: x=%d, y=%d\n", x, y)
    c = x + y
    fmt.Printf("add called: c=%d\n", c)
    return c
}

func main() {
    var wg sync.WaitGroup // 定义等待组
    fmt.Println(runtime.NumGoroutine())
    fmt.Println("main start")
    wg.Add(1)             // 计数加1
    go add(4, 5, &wg) // 协程
    fmt.Println(runtime.NumGoroutine())
    // time.Sleep(2 * time.Second) // 这一句不需要了
    wg.Wait() // 阻塞到wg的计数为0
    fmt.Println("main end")
    fmt.Println(runtime.NumGoroutine())
}
```

父子协程

一个协程A中创建了另外一个协程B，A称作父协程，B称为子协程。

```go
package main

import (
    "fmt"
    "sync"
    "time"
)

func main() {
    var wg sync.WaitGroup // 定义等待组
    fmt.Println("main start")
    count := 6
```

```
    wg.Add(count)

    go func() {
        fmt.Println("父协程开始，准备启动子协程")
        defer func() {
            wg.Done() // 注意wg的作用域
            fmt.Println("父协程结束了~~~~")
        }()
        for i := 0; i < count-1; i++ {
            go func(id int) {
                defer wg.Done()
                fmt.Printf("子协程 %d 运行中\n", id)
                time.Sleep(5 * time.Second)
                fmt.Printf("子协程 %d 结束\n", id)
            }(i)
        }
    }()

    wg.Wait() // 阻塞到wg的计数为0
    fmt.Println("main end")
}
// 注：上例协程最好协程独立的函数，而不是这样嵌套，只是为了演示。
```

父协程结束执行，子协程不会有任何影响。当然子协程结束执行，也不会对父协程有什么影响。父子协程没有什么特别的依赖关系，各自独立运行。

只有主协程特殊，它结束程序结束。

实战：实现WEB服务器——Goroutine版

```
package main

import (
    "fmt"
    "log"
    "net"
)

var html = `<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>magedu</title>
</head>
<body>
    <h1>马哥教育www.magedu.com -- Goroutine</h1>
</body>
</html>`

var head = `HTTP/1.1 200 OK
Date: Mon, 24 Oct 2022 20:04:23 GMT
Content-Type: text/html
Content-Length: %d
```

```
Connection: keep-alive
Server: wayne.magedu.com

%s`

var response = fmt.Sprintf(head, len(html), html)

func main() {
    laddr, err := net.ResolveTCPAddr("tcp4", "0.0.0.0:9999") // 解析地址
    if err != nil {
        log.Panicln(err) // Panicln会打印异常，程序退出
    }
    server, err := net.ListenTCP("tcp4", laddr)
    if err != nil {
        log.Panicln(err)
    }
    defer server.Close() // 保证一定关闭

    for {
        conn, err := server.Accept() // 接收连接，分配socket
        if err != nil {
            log.Panicln(err)
        }

        go func() {
            defer conn.Close() // 保证一定关闭

            buffer := make([]byte, 4096) // 设置缓冲区
            n, err := conn.Read(buffer)  // 成功返回接收了多少字节
            if n == 0 {
                log.Printf("客户端%s主动断开", conn.RemoteAddr().String())
                return
            }
            if err != nil {
                log.Println(err)
                return
            }
            conn.Write([]byte(response))
        }()
    }
}
// 大家可以自行抽取成协程函数
```

上述代码是goroutine per connection模式，看似使用的同步方式开发，这大大减少了开发人员的心智负担。