Go语言 runtime 包

Go 语言的 goroutine 是由 运行时（runtime）调度和管理的。它负责管理包括内存分配、垃圾回收、栈处理、goroutine、channel、切片（slice）、map 和反射（reflection）等等。

runtime包里面定义了一些协程管理相关的方法

runtime.Gosched()

让出当前协程的 CPU 时间片给其他协程。当前协程等待时间片未来继续执行。

释放时间片，先让别的协程执行，它执行完，再回来执行此协程。

```
package main

import (
    "fmt"
    "runtime"
)

func main() {
    go func(s string) {
        for i := 0; i < 2; i++ {
            fmt.Println(s)
        }
    }("world")

    // 主协程
    for i := 0; i < 2; i++ {
        runtime.Gosched()     //主协程释放CPU时间片，此时上面的协程得以执行
        fmt.Println("hello") //CPU时间片回来后继续执行
    }
}
#结果
world
world
hello
hello
```

进入主协程的第一轮 for 循环，主协程让出CPU时间片时，上面的协程已经创建好，并打印了两个world，然后主协程继续执行，打印了一个hello。进入主协程的第二轮 for 循环，主协程让出CPU时间片时，已经没有协程正在等待执行，所以主协程继续打印了一个hello，然后结束。

runtime.Goexit()

退出当前协程，但是 defer 语句会照常执行。

```
package main

import (
    "fmt"
    "runtime"
```

```
    "time"
)

func main() {
    go func() {
        defer fmt.Println("A.defer")
        func() {
            defer fmt.Println("B.defer")
            runtime.Goexit() // 结束当前协程
            defer fmt.Println("C.defer")
            fmt.Println("B")
        }()
        fmt.Println("A")
    }()

    time.Sleep(time.Second) // 睡一会儿，不让主协程很快结束
    fmt.Println("main")
}
#结果
B.defer
A.defer
main
```

在我们自己的协程结束之前，是会打印已定义的 B.defer 和 A.defer 的，这说明：

如果我们用 runtime.Goexit() 结束协程，仍然会执行 defer 语句。结束之后的不会被执行

runtime.GOMAXPROCS()

Golang 默认所有任务都运行在一个 cpu 核里，如果要在 goroutine 中使用多核，可以使用 runtime.GOMAXPROCS 函数修改，当参数小于 1 时使用默认值。

Go运行时的调度器使用 GOMAXPROCS 参数来指定需要使用多少个 OS 线程来同时执行 Go 代码。默认值是机器上的 CPU 核心数。例如在一个 8 核心的机器上，调度器会把 Go 代码同时调度到 8 个 OS 线程上（GOMAXPROCS 是m:n调度中的n）。

Go语言中可以通过 runtime.GOMAXPROCS() 函数设置当前程序并发时占用的 CPU 逻辑核心数。

Go1.5版本之前，默认使用的是单核心执行。Go1.5版本之后，默认使用全部的 CPU 逻辑核心数。

```
package main

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

func a() {
    for i := 1; i < 10; i++ {
        fmt.Println("A:", i)
    }
}

func b() {
    for i := 1; i < 10; i++ {
```

```
        fmt.Println("B:", i)
    }
}

func main() {
    runtime.GOMAXPROCS(2) //可以通过调整核心数看看执行效果
    go a()
    go b()
    time.Sleep(time.Second) //睡一会儿，不让主协程结束
    fmt.Println("over")
}
```

Go语言中的操作系统线程和 goroutine 的关系：

1. 一个操作系统线程对应用户态多个 goroutine。
2. go 程序可以同时使用多个操作系统线程。
3. goroutine 和 OS 线程是多对多的关系，即 m:n。

runtime.NumCPU()

返回当前系统的 CPU 核数量。

runtime.GOROOT()

获取 goroot 目录。

runtime.GOOS

查看目标操作系统。很多时候，我们会根据平台的不同实现不同的操作，就可以用GOOS来查看自己所在的操作系统。

runtime.GC

会让运行时系统进行一次强制性的垃圾收集。

强制的垃圾回收：不管怎样，都要进行的垃圾回收。非强制的垃圾回收：只会在一定条件下进行的垃圾回收（即运行时，系统自上次垃圾回收之后新申请的堆内存的单元（也成为单元增量）达到指定的数值）。

runtime.LockOSThread 和 runtime.UnlockOSThread

runtime.LockOSThread调用会使调用他的 Goroutine 与当前运行它的M锁定到一起。

runtime.UnlockOSThread调用会解除这样的锁定。

runtime.NumGoroutine

返回正在执行和排队的任务总数。

runtime.NumGoroutine函数在被调用后，会返回系统中的处于特定状态的 Goroutine 的数量。这里的特定状态是指Grunnable\Gruning\Gsyscall\Gwaition。处于这些状态的Groutine即被看做是活跃的或者说正在被调度。
注意：垃圾回收所在Groutine的状态也处于这个范围内的话，也会被纳入该计数器。