

# Go并发编程

*GoLang: Let gopher run faster*

*v1.0.0*



# 关于 我



郝林，年龄33-，码龄9年+，Go语言非脑残粉  
(@特价萝卜)

- ✧ Java 软件工程师
- ✧ Clojure/Python 程序员
- ✧ Linux 爱好者
- ✧ NBA 观众



# 关于 《Go并发编程实战》



2013年7月受图灵公司之邀开始撰写书稿，2014年7月提交初稿，2014年11月出版。历经16个月。

- ✧ 分享Go语言编程知识
- ✧ 探究Go语言并发编程原理
- ✧ 为Go语言中文社区做一点贡献
- ✧ 找一个深入理解Go语言的理由

<http://www.ituring.com.cn/book/1525>



# 目录



1

- Go语言与并发编程

2

- Goroutine与go语句

3

- Channel

4

- 并发编程原理一窥

5

- 一些控制参数

6

- 推荐阅读列表

# Go语言是怎样的？



- ✓ 开源软件，社区活跃、迭代快速
- ✓ 通用、系统级、跨平台
- ✓ 静态类型、编译型，脚本化的语法
- ✓ 支持多种编程范式（函数式 + 面向对象）
- ✓ 提供了强大、易用的工具，涵盖软件的全生命周期
- ✓ 原生的并发编程支持
- ✓ 在大大提高开发效率的同时拥有极高的运行效率

# Go语言的哲学



- ✓ 崇尚简约
- ✓ 约定大于配置（但配置也不可或缺）
- ✓ 在编码规范上严谨，在软件设计上宽松
- ✓ 以通讯的方式共享内存，而不是将共享内存作为通讯手段
- ✓ 软件工程至上！

# Go语言与并发编程（1）



**Goroutine**是Go语言用于对并发编程提供原生支持的利器，也是其并发编程模型的核心概念之一。

Go语言的开发者们之所以专门创建了这样的一个名词，是因为他们认为现存的进程（*Process*）、线程（*Thread*）、协程（*Coroutine*）等概念都向应用程序员们传达了错误的信号。为了与它们有所区别，**Goroutine**这个词得以诞生。

# Go语言与并发编程（ 2 ）



多进程编程：

- 管道 ( Pipe )
- 信号 ( Signal )
- 消息队列 ( Message Queue )
- 信号灯 ( Semaphore )
- 共享内存区 ( Shared Memory )
- 套接字 ( Socket )



# Go语言与并发编程（3）



## 多线程编程：

### ➤ 共享内存区（Shared Memory）

- ✓ 互斥量（Mutex）
- ✓ 条件变量（Conditions）
- ✓ 原子操作（Atomic operation）
- ✓ 可重入函数（Reentrant function）

# Go语言与并发编程（4）



Go语言特有的并发编程方式：

*Goroutine & Channel*

# Goroutine与go语句（1）



Goroutine是怎样的？

轻量级线程？绿色线程？  
协程？超级协程？

笼统地讲，Goroutine基于两级线程模型，可以被视为“用户级线程”。但它背后的支撑体系比之更加复杂和高效。

# Goroutine与go语句（2）



与Goroutine有关的几个数字：

- ✧ 一个Goroutine栈的初始尺寸：2 KB  
（相比之下，一个线程栈默认需要 8 MB）
- ✧ 一个Goroutine栈的最大尺寸：
  - 在 32 位系统下为：250 MB
  - 在 64 位系统下为：1 GB
- ✧ 默认情况下，N（几十上百万）个Goroutine最多可以在 10000 个内核线程上调度运行。

# Goroutine与go语句 ( 3 )



怎样创建或启用一个Goroutine ?

```
go println("Go! Goroutine!")
```

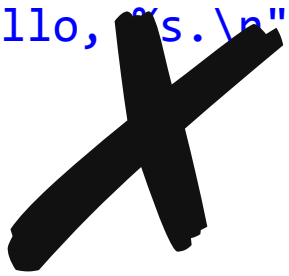
```
go func() {  
    println("Go! Goroutine!")  
}()
```

# Goroutine与go语句 ( 4 )



有一点需要特别注意：

```
names := []string{"Eric", "Harry", "Robert", "Jim", "Mark"}
for _, name := range names {
    go func() {
        fmt.Printf("Hello, %s.\n", name)
    }()
}
```



输出：

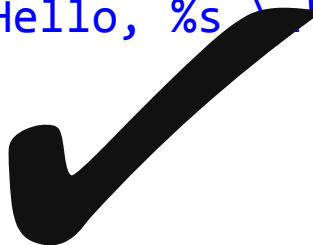
```
Hello, Mark.
Hello, Mark.
Hello, Mark.
Hello, Mark.
Hello, Mark.
```

# Goroutine与go语句 ( 5 )



正确的做法是：

```
names := []string{"Eric", "Harry", "Robert", "Jim", "Mark"}
for _, name := range names {
    go func(who string)
        { fmt.Printf("Hello, %s\n", who)
        }(name)
}
```



输出：

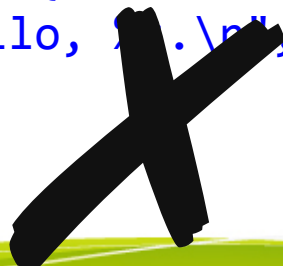
Hello, Eric.  
Hello, Harry.  
Hello, Robert.  
Hello, Jim.  
Hello, Mark.

# Goroutine与go语句 ( 6 )



让我们散焦一点：

```
package main
import (
    "fmt"
    "runtime"
)
func main() {
    names := []string{"Eric", "Harry", "Robert", "Jim", "Mark"}
    for _, name := range names {
        go func(who string) {
            fmt.Printf("Hello, %s.\n", who)
        }(name)
    }
}
```



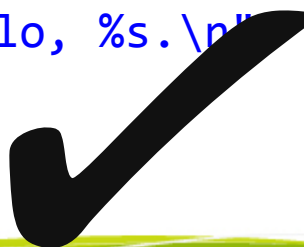


# Goroutine与go语句 ( 7 )



## 手动调度Goroutine :

```
package main
import (
    "fmt"
    "runtime"
)
func main() {
    names := []string{"Eric", "Harry", "Robert", "Jim", "Mark"}
    for _, name := range names {
        go func(who string) {
            fmt.Printf("Hello, %s.\n", who)
        }(name)
    }
    runtime.Gosched()
}
```



# Goroutine与go语句（8）



主Goroutine做了什么？

- ✓ 启动系统监测器
- ✓ 设定通用配置，检查运行环境
- ✓ 创建定时垃圾回收器
- ✓ 执行main包的init函数
- ✓ 执行main包的main函数
- ✓ 进行一些善后处理工作

# Channel ( 1 )



Channel是怎样的？

- ✓ 即通道类型，Go语言的**预定义类型**之一
- ✓ **类型化、并发安全的**通用型管道
- ✓ 用于在多个Goroutine之间传递数据
- ✓ 对于“以通讯的方式共享内存”的最直接体现

**Channel 与 Goroutine 堪称绝配！**

# Channel ( 2 )



怎样创建Channel ?

```
strChan := make(chan string, 3)
```

```
strChan := make(chan string)
```

```
// 相当于 strChan := make(chan string, 0)
```

# Channel ( 3 )



怎样向通道发送一个值？

```
strChan <- "first value"
```

怎样从通道接收一个值？

```
value := <- strChan  
// 或 value, ok := <- strChan
```

# Channel ( 4 )



怎样关闭通道？

```
close(strChan)
```

注意！

- (1) 试图从一个**未被初始化的通道接收**值，会造成当前Goroutine的**永久阻塞**！
- (2) 试图向一个**未被初始化的通道发送**值，也会造成当前Goroutine的**永久阻塞**！

# Channel ( 5 )



注意！（续）

- (3) 试图向一个已被关闭的通道发送值，会立即引发一个运行时恐慌！
- (4) 试图关闭一个已被关闭的通道，也会立即引发一个运行时恐慌！
- (5) 被通道传递的是值的副本，而不是值本身。
- (6) 关闭通道不会使针对于此的接收操作立即结束。

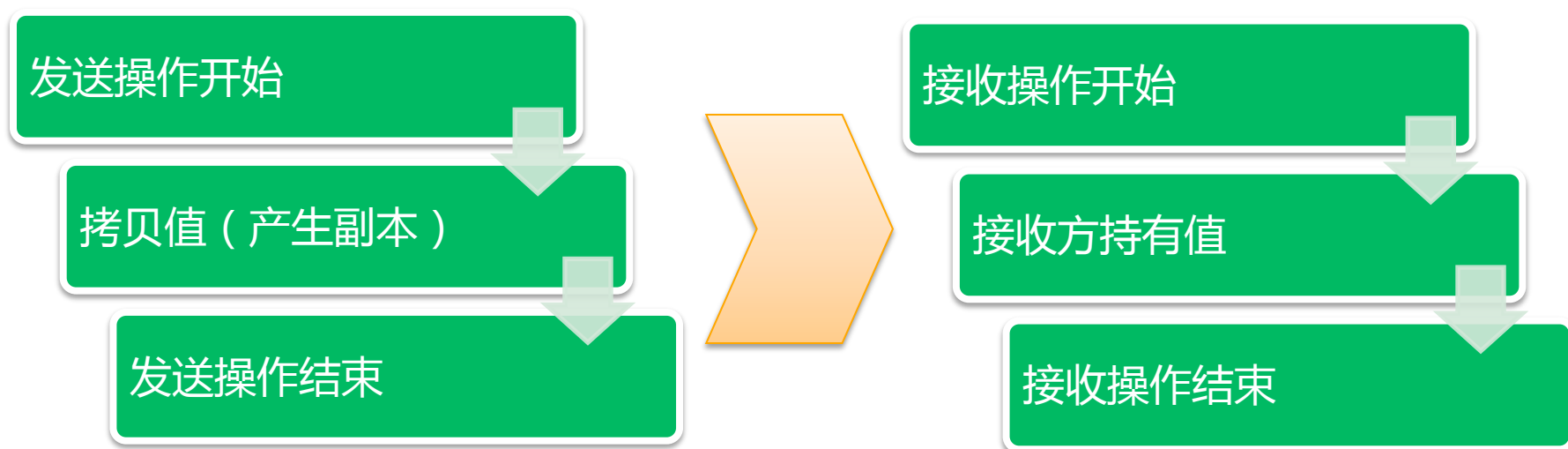
还记得 `value, ok := <- strChan` 吗？

# Channel ( 6 )



## Happens before 原则

【针对缓冲（长度大于零的）通道：`make(chan string, 3)`】



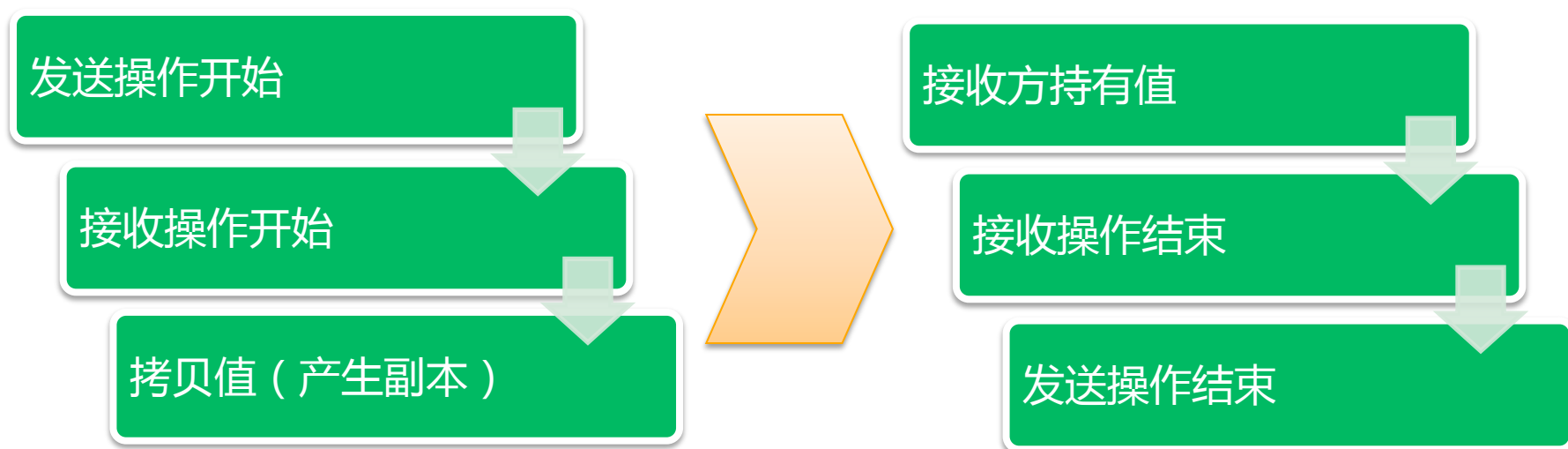


# Channel ( 7 )



## Happens before 原则

【针对非缓冲（长度等于零的）通道：`make(chan string)`】



# Channel ( 8 )



通道可以协调多个Goroutine的运行：

```
package main
import (
    "fmt"
)
func main() {
    name := "Robert"
    ch := make(chan byte, 1)
    go func() {
        fmt.Printf("Hello, %s.\n", name)
        ch <- 1
    }()
    <-ch
}
```

另一个可选方案是：  
使用 `sync.WaitGroup`

# Channel ( 9 )



## 单向 Channel :

```
type IntChan <-chan int

func asyncReceive(ch <-IntChan) {
    //
}
```

```
type IntChan chan<- int

func asyncSend() <-chan int {
    //
}
```

# Channel ( 10 )



## Channel 与 for 语句：

```
var ch = make(chan int, 10)
// 省略若干条语句
if ch != nil {
    for e := range ch {
        fmt.Printf("Element: %v\n", e)
    }
}
```

如此从通道中接收值的优势是在通道关闭后for循环会自动退出。

# Channel ( 11 )



## Channel 与 select语句：

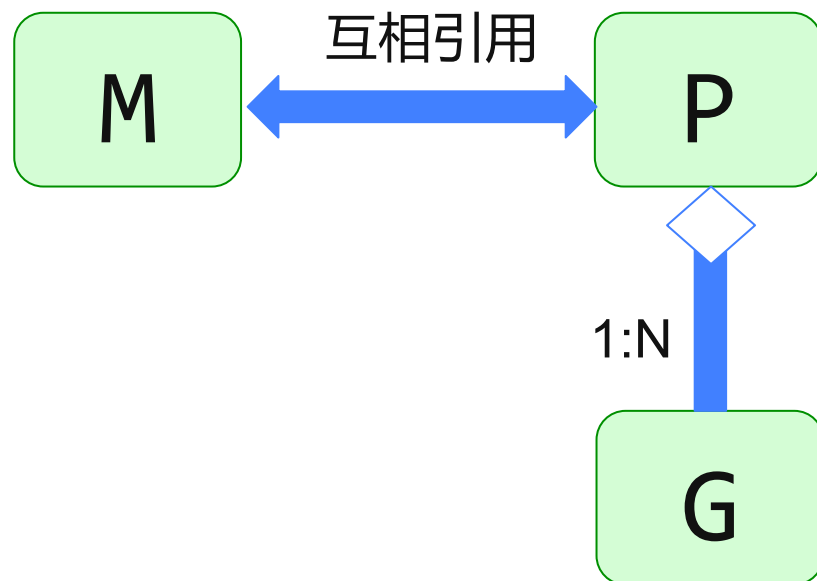
```
var ch1 = make(chan int, 10)
var ch2 = make(chan string, 10)
// 省略若干条语句
select {
case e1 := <-ch1:
    fmt.Printf("1th case is selected. e1=%v.\n", e1)
case e2 := <-ch2:
    fmt.Printf("2th case is selected. e2=%v.\n", e2)
default:
    fmt.Println("default!")
}
```

# 并发编程原理一窥（1）



## 三个核心元素：

- ✧ M：Machine的缩写。一个M代表了一个内核线程。
- ✧ P：Processor的缩写。一个P代表了M所需的上下文环境。
- ✧ G：Goroutine的缩写。一个G代表了对一段需要被并发执行的Go语言代码的封装。

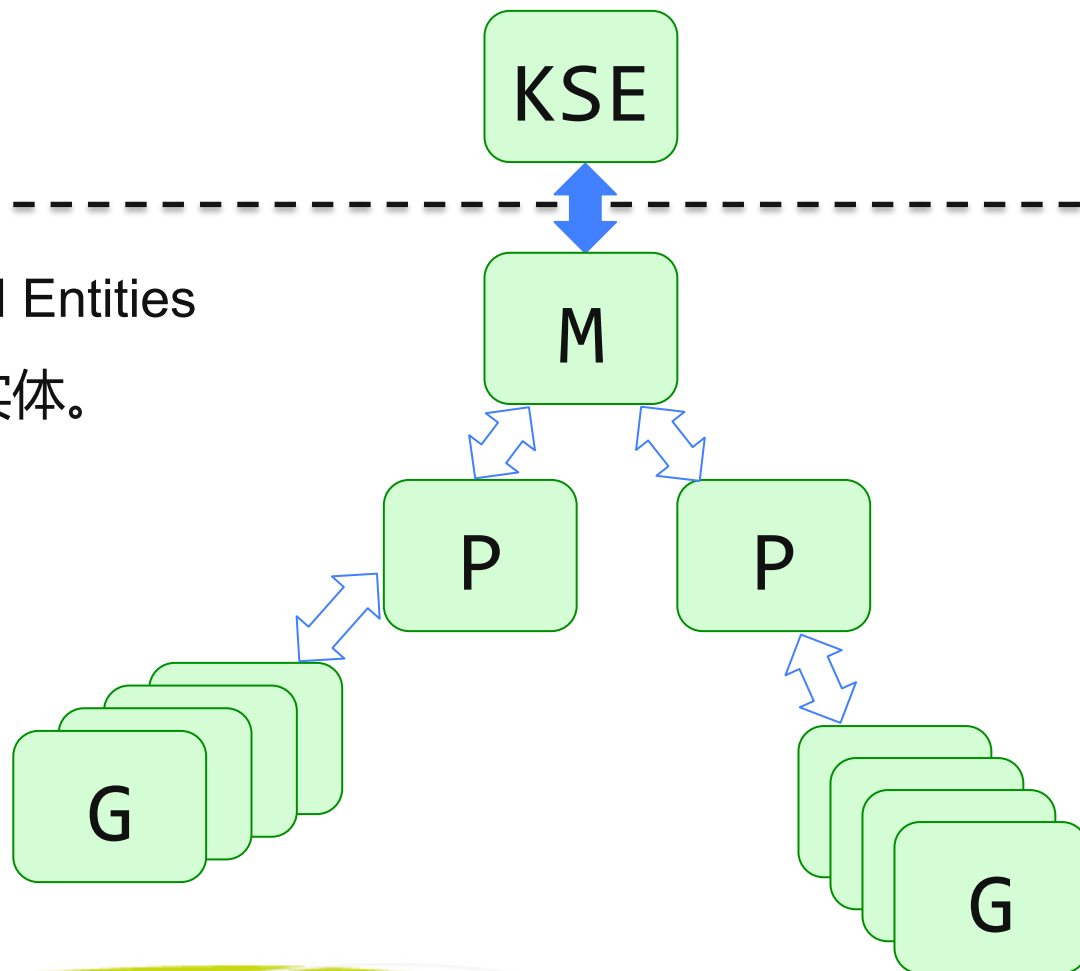


# 并发编程原理一窥（2）



M、P、G 与 KSE：

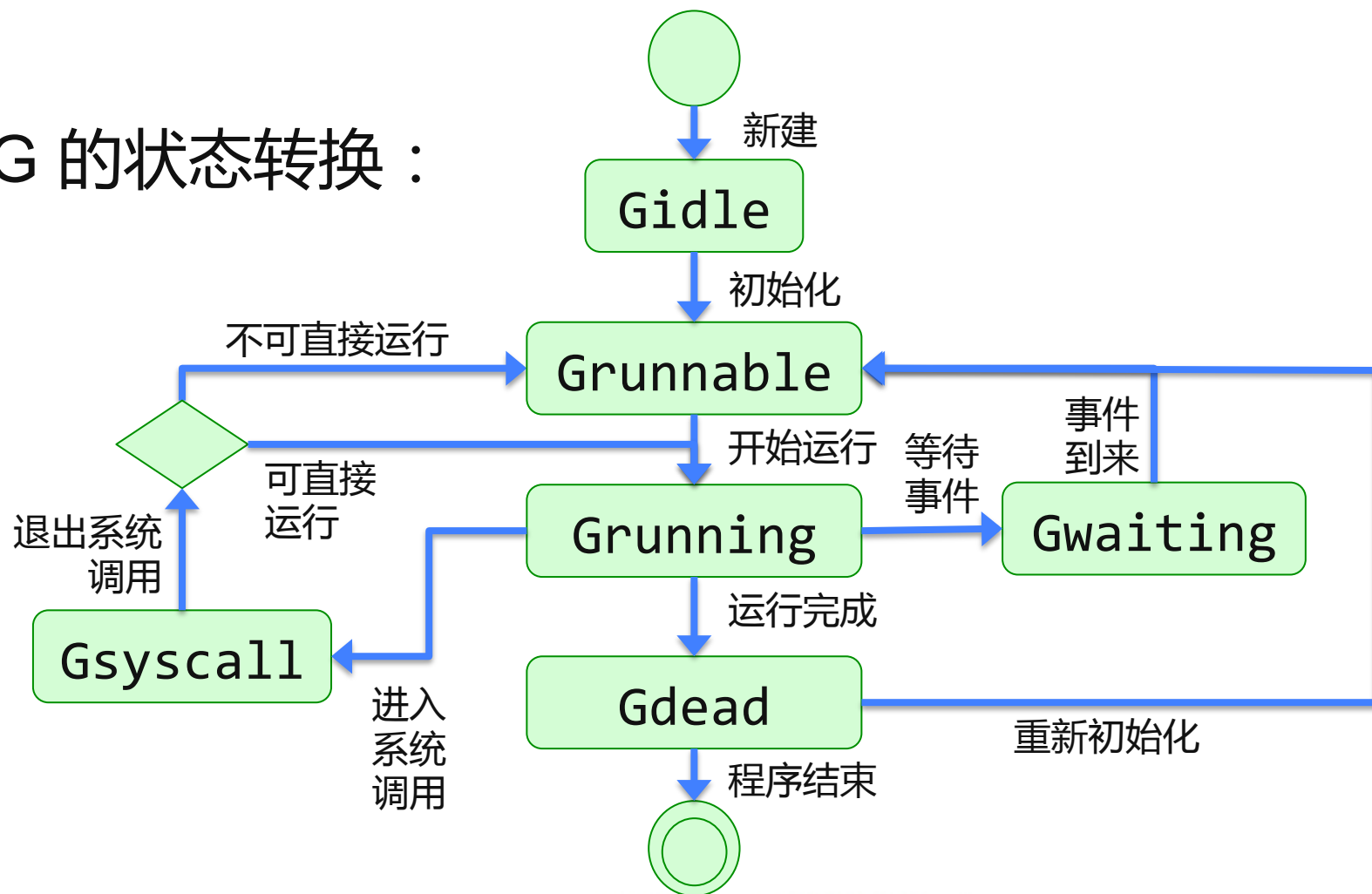
✧ KSE：Kernel Scheduled Entities  
的缩写，即：内核调度实体。



# 并发编程原理一窥 ( 3 )



G 的状态转换：

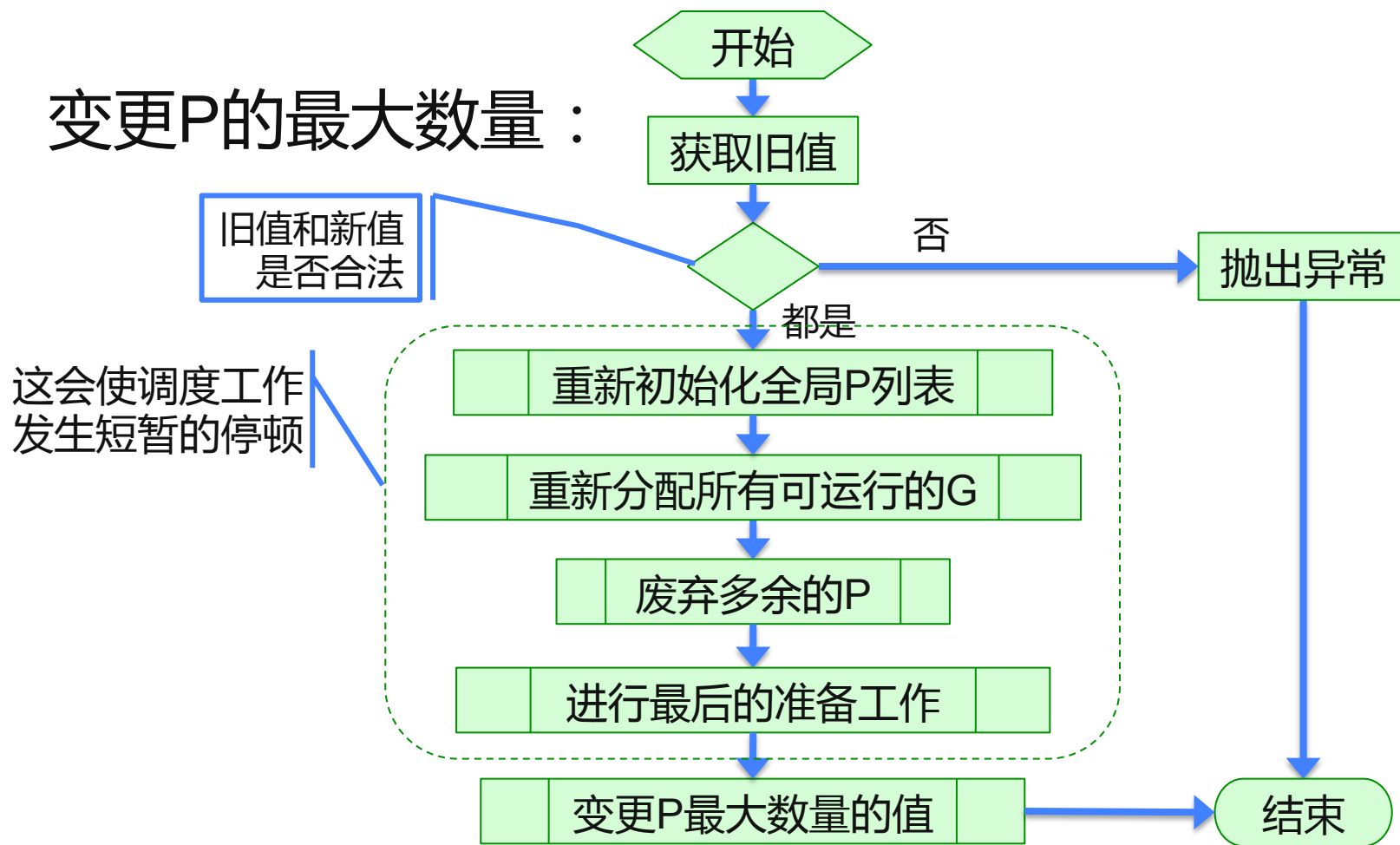




# 并发编程原理一窥（4）



变更P的最大数量：



# 并发编程原理一窥（ 5 ）



## 调度器跟踪：

- 方法：设置环境变量GODEBUG
- 值的选项：
  - ✓ 打印简要信息：**schedtrace=N**  
（ 每N毫秒打印一次简要信息 ）
  - ✓ 打印详细信息：**schedtrace=N,scheddetail=1**  
（ 每N毫秒打印一次详细信息 ）

# 一些控制参数 ( 1 )



P的最大数量：

➤ 设定方法：

- ✓ 调用函数 `runtime.GOMAXPROCS`
- ✓ 设置环境变量 `GOMAXPROCS`

➤ 备注：

- ✧ 默认数量是1，认可的数量最大为256
- ✧ 尽早设置，以避免对程序性能带来影响

# 一些控制参数 ( 2 )



M的最大数量：

➤ 设定方法：

✓ 调用函数 `runtime/debug.SetMaxThreads`

➤ 备注：

✧ 默认数量是10000

✧ 过小的参数值会引发运行时恐慌！

✧ 当M的实际数量大于设定值时也会引发运行时恐慌！

# 一些控制参数 ( 3 )



G栈空间的最大字节数量：

➤ 设定方法：

✓ 调用函数 `runtime/debug.SetMaxStack`

➤ 备注：

✧ 当此数量大于设定值时会引发运行时恐慌！

✧ 正因为如此，要避免设置过小的数值

✧ 同时也要避免设置过大的数值（你懂的）

# 一些控制参数（4）



## 手动调度G：

### ➤ 使当前G让出CPU：

- ✓ 调用函数 `runtime.Gosched`

- ✓ 备注：只能使其暂停运行，且无法精确预知恢复时间

### ➤ 结束当前G的运行：

- ✓ 调用函数 `runtime.Goexit`

- ✓ 备注：仍会保证相关defer语句的执行完成

# 一些控制参数 ( 5 )



锁定当前G和当前M :

➤ 锁定方法 :

✓ 调用函数 `runtime.LockOSThread`

➤ 备注 :

✧ 多次调用不会有副作用 , 但仅有最后一次调用会生效

✧ 解锁方法 : 调用函数 `runtime.UnlockOSThread`

# 一些控制参数 ( 6 )



与GC相关：

➤ 设定垃圾收集比率：

- ✓ 调用函数 `runtime/debug.SetGCPercent`
- ✓ 设置环境变量 `GOGC`

➤ 手动操作：

- ✧ `runtime.GC`：手动垃圾收集
- ✧ `runtime/debug.FreeOSMemory`：手动垃圾收集和清扫



# 推荐阅读列表



## 【仅限Go并发编程方面】

### ● 英文资料

- ✓ Go语言源码
- ✓ Golang Nuts ( Google Groups )
- ✓ Share Memory By Communicating - The Go Blog
- ✓ Go Concurrency Patterns
- ✓ Advanced Go Concurrency Patterns

### ● 中文资料

- ✓ 《Go并发编程实战》



# Q & A

## Let's Go!

