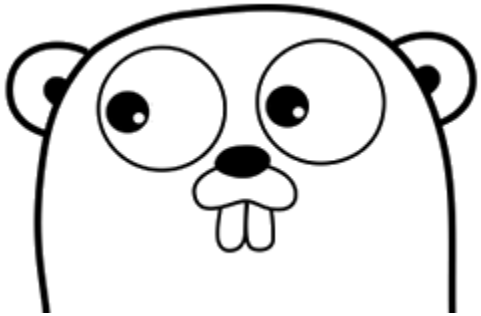
# Go语言编程实践

## 为软件工程而生；为Java程序员而写



撰写人：郝林（@特价萝卜）

# 目录

# Go语言基础——初看

- ✓ 通用编程语言，开源，跨平台
- ✓ 类C的、简介的语法，集多编程范式之大成者
- ✓ 静态类型、编译型语言
  （却看起来像动态类型、解释型语言）
- ✓ 自动垃圾回收，内置多核并发机制，强大的运行时反射
- ✓ 高生产力，高运行效率，体现优秀软件工程原则

# Go语言基础——再看

- ✓ 来自Google，2009年诞生，当前版本：1.1
- ✓ 主页：http://golang.org 和 https://code.google.com/p/go
- ✓ 语言规范： http://tip.golang.org/ref/spec
- ✓ API文档： http://godoc.org
- ✓ Go语言中文社区：
  http://www.golang.tc 和 http://bbs.mygolang.com

# Go语言基础——运算符

| 优先级 | 运算符 |
| --- | --- |
| 最高 | * / % << >> & &^ |
| | + - \| ^ |
| | == != < <= > >= |
| | <- |
| | && |
| 最低 | \|\| |

# Go语言基础——保留字

| | | | | |
|---|---|---|---|---|
| break | default | func | interface | select |
| case | defer | go | map | struct |
| chan | else | goto | package | switch |
| const | fallthrough | if | range | type |
| continue | for | import | return | var |

# Go语言基础——基本数据类型

| 类型 | 长度（字节） | 零值 | 说明 |
| --- | --- | --- | --- |
| bool | 1 | false | true，false。不能把非非零值当作 true。 |
| byte | 1 | 0 | 等同于uint8。 |
| rune | 4 | 0 | 等同于int32。存储 Unicode Code Point。 |
| int/uint | | 0 | 与平台有关，在 AMD64/X86-64 平台是 64 位整数。 |
| int8/uint8 | 1 | 0 | 范围：-128 ~ 127；0 ~ 255。 |
| int16/uint16 | | 0 | 范围：-32768 ~ 32767；0 ~ 65535。 |
| int32/uint32 | 4 | 0 | 范围：-21亿 ~ 21亿；0 ~ 42亿。 |
| int64/uint64 | 8 | 0 | |
| float32 | 4 | 0.0 | 精确到 7 个小小数位。 |
| float64 | 8 | 0.0 | 精确到 15 个小小数位。 |
| complex64 | 8 | 0.0 | |
| complex128 | 16 | 0.0 | |

# Go语言基础——基本数据类型（续）

| 类型 | 零值 | 说明 |
| --- | --- | --- |
| uintptr | nil | 足足够保存指针的 32 位或 64 位整数。 |
| array | nil | 值类型，如：[2]int。 |
| struct | | 结构体，值类型。无零值，自动实例化。 |
| string | “” | 值类型。多行时可用“`”包裹。 |
| slice | nil | 引用类型，如：[]int。 |
| map | nil | 引用类型。 |
| channel | nil | 引用类型。 |
| interface | nil | 接口类型。 |
| function | nil | 函数类型。 |

# Go语言基础——内置函数

- **close:** 关闭 channel。
- **len:** 获取string、array、slice的长度，map key的数量，以及缓冲channel的可用数据数量。
- **cap:** 获取 array的长度，slice的容量，以及缓冲channel的最大缓冲容量。
- **new:** 通常用于值类型，为指定类型分配初始化过的内存空间，返回指针。
- **make:** 仅用于slice、map、channel这些引用类型，除了初始化内存，还负责设置相关属性。
- **append:** 向slice追加 (在其尾部添加) 一个或多个元素。
- **copy:** 在不同slice间复制数据。
- **print/println:** 不支持 format，要格式化输出，要使用fmt包。
- **complex/real/imag:** 复数处理。
- **panic/recover:** 错误处理。

# Go语言基础——常量

声明：

```
const (
    LANG   = "Go"
    TOPIC  = "Practice"
    METHOD = "Coding"
)
```

# Go语言基础——变量

声明：

```
var i int64
var m map[string]int
var c chan
```

声明并赋值：

```
var i1, s1 = 123, "hello"
i2, s2 := 123, "hello"  //仅限函数内使用
array1 := [...]{1,2,3}  //仅限函数内使用
```

# Go语言基础——字符串

string :

```
s1 := "abcdefg"
fmt.Printf("s1: %s\n", s1[2:3])
// => s1 part: c
ba1 := []byte(s1)
ba1[2] = 'C'
s2 := string(ba1)
fmt.Printf("s2: %s\n", s2)
// => s2: abCdefg
```

# Go语言基础——字符串（续）

string :

```
// => s2: abCdefg
fmt.Printf("s2 (rune array):
%v\n",[]rune(s2))
// => s2 (rune array): [97 98 67 100
101 102 103]
fmt.Printf("Raw string:\n%s\n", `a\t
b`)
// => Raw string: a\t
// => b
```

这儿有一个回车

这儿也有一个回车

# Go语言基础——切片

slice :

```
var arr1 []int
fmt.Printf("arr1 (1):%v\n", arr1)
// => arr1 (1):[]
arr1 = append(arr1, 1)
arr1 = append(arr1, []int{2, 3, 4}...)
fmt.Printf("arr1 (2):%v\n", arr1)
// => arr1 (2):[1 2 3 4]
```

# Go语言基础——切片（续）

slice :

```
arr2 := make([]int, 5)
fmt.Printf(“arr2 (1):%v\n”, arr2)
// => arr2 (1):[0 0 0 0 0]
n := copy(arr2, arr1[1:4])
fmt.Printf("%d copied, arr2 (2):%v\n",
n, arr2)
// => 3 copied, arr2 (2):[2 3 4 0
0]
```

# Go语言基础——切片（续2）

slice :

```
// => arr2 (2):[2 3 4 0 0]
n = copy(arr2, arr1)
fmt.Printf("%d copied, arr2 (3):%v\n", n,
arr2)
// => 4 copied, arr2 (3):[1 2 3 4 0]
arr3 := []int{6, 5, 4, 3, 2, 1}
n = copy(arr2, arr3)
fmt.Printf("%d copied, arr2 (4):%v\n", n,
arr2)
// => 5 copied, arr2 (4):[6 5 4 3 2]
```

# Go语言基础——字典

map：

```
m1 := map[string]int{"A": 1, "B": 2}
fmt.Printf("m1 (1): %v\n", m1)
// => m1 (1): map[A:1 B:2]
delete(m1, "B")
fmt.Printf("m1 (2): %v\n", m1)
// => m1 (2): map[A:1]
v, ok := m1["a"]
fmt.Printf("v: %v, ok? %v\n", v, ok)
// => v: 0, ok? false
```

# Go语言基础——控制语句

if :

```
var i1 int
if i1 == 0 {
    fmt.Println("Zero value!")
} else {
    fmt.Println("Nonzero value")
}
if1 := interface{}(i1)
if i2, ok := if1.(int32); ok {
    fmt.Printf("i2: %d\n", i2) // 未被打印？
}
// => Zero value!
```

# Go语言基础——控制语句（续）

switch：

```
var n int8
switch n {
case 0:
    fallthrough  // 继续执行下面的case
case 1:
    n = (n + 1) * 2
default:
    n = -1
}
fmt.Printf("I: %d\n", n)  // => I: 2
```

# Go语言基础——控制语句（续2）

for :

```
var n uint8
for n < 100 {
    n++
}
fmt.Printf("N: %d\n", n)  // => N: 100

for i := 0; i < 100; i++ {
    n++
}
fmt.Printf("N: %d\n", n) )  // => N: 100
```

# Go语言基础——控制语句（续3）

for :

```
strings := []string{"A", "B", "C"}
for i, e := range strings {
    fmt.Printf("%d: %s\n", i, e)
}
// => 0: A
// => 1: B
// => 2: C
stringMap := map[int]string{1: "A", 2: "B", 3: "C"}
for k, v := range stringMap {
   fmt.Printf("%d: %s\n", k, v)
}
// 会打印出什么？
```

# Go语言基础——函数

函数声明（First Class Style）：

函数可作为参数

函数可作为返回值

```
func GenMyFunc(hash func(string) int64, content string) func() string {
        return func() string {
                return fmt.Sprintf("Content Hash: %v", hash(content))
        }
}
```

函数允许有多返回值。这里返回的是error实例，但是我们用占位符“_”扔掉了它。

调用代码：

```
myFunc := GenMyFunc(func(s string) int64 {
        result, _ := strconv.ParseInt(s, 0, 64)
        return result
}, "0x10")
fmt.Printf("%s\n", myFunc())
```

输出结果： Content Hash: 16

# Go语言基础——defer

defer的常用法：

```
func ReadFile(filePath string) error {
    file, err := os.Open(filePath)
    if err != nil {
        return err
    }
    defer file.Close()
    ……
    return nil
}
```

在退出函数ReadFile前，defer后的语句会被执行。

# Go语言基础——defer（续）

defer后也可以是一个匿名函数：

```
func ReadFile(filePath string) error {
    file, err := os.Open(filePath)
    if err != nil {
        return err
    }
    defer func() {
        file.Close()
    }()
    ……
    return nil
}
```

# Go语言基础——异常处理

panic：

```
func ReadInputs(exitMark string, buffer bytes.Buffer) {
    end := false
    reader := bufio.NewReader(os.Stdin)
    fmt.Println("Please input:\n")
    for !end {
        line, err := reader.ReadString('\n')
        if err != nil {
            panic(err)
        }
        if (exitMark + "\n") == line {
            break
        }
        buffer.WriteString(line)
    }
}
```

普通错误常常被作为返回值，而不是被抛出。

如果你认为某类错误或异常是不可容忍的，甚至需要终止程序，那么你可以用panic制造一个“恐慌”并附上错误信息！

# Go语言基础——异常处理（续）

recover：

当然，你可以在调用处添加“保护层”。这是defer的另一个常用法。

```
func main() {
    defer func() {
        if err := recover(); err != nil {
            debug.PrintStack()
            fmt.Printf("Fatal Error: %s\n", err)
        }
    }()
    var buffer bytes.Buffer
    ReadInput("exit", buffer)
    fmt.Printf("Inputs: %s\n", buffer.String())
}
```

如果“恐慌”发生了，我们可以“平息”它，以防程序终止。

我们为致命的异常打印一下调用栈吧。

panic不一定会被调用方recover，只有在确认有必要的时候才应该这么做！

# Go语言基础——结构体与方法

一个简单的struct以及它的一个方法：

```
type Dept struct {
    name string
    building string
    floor uint8
}

func (self Dept) Name() string {
    return self.name
}
```

是不是有些眼熟？（如果你写过Python代码的话）。这里也相当于Java中的“this”。

# Go语言基础——结构体与方法（续）

再看看这个struct的其他几个方法：

```
func (self Dept) SetName(name string) {
    self.name = name
}

func (self *Dept) Relocate(building string,
floor uint8) {
    self.building = building
    self.floor = floor
}
```

注意这个星号！这意味将Dept实例的指针赋值给了“self”，后面我们将会看到它们的不同之处。

# Go语言基础——结构体与方法（续2）

struct的使用方法：

```
dept1 :=
    Dept{
        name: "MySohu",
        building: "Internet",
        floor: 7,
    }
fmt.Printf("dept (1): %v\n", dept1)
// => dept (1): {MySohu Internet 7}
dept1.Relocate("Media", 12)
fmt.Printf("dept (3): %v\n", dept1)
// => dept (2): {MySohu Media 12}
```

# Go语言基础——结构体与方法（续3）

struct方法中的传值与传引用（指针）：

```
dept1.SetName("Other")
fmt.Printf("dept (3): %v\n", dept1)
// => dept (3): {MySohu Media 12}
```

看这里，说明SetName方法没起作用，为什么？

回顾一下两个设置方法的签名：

```
func (self Dept) SetName…

func (self *Dept) Relocate…
```

说明：

- “(self Dept)”相当于把本Dept实例的副本赋值给了“self”。
- “(self *Dept)”相当于把本Dept实例的指针的副本赋值给了“self”。

# Go语言基础——接口


interface：

```
type DeptModeFull interface { //包含了结构Dept及其指针上的所有方法
    Name() string
    SetName(name string)
    Relocate(building string, floor uint8)
}

type DeptModeA interface { //仅包含了结构Dept上的方法
    Name() string
    SetName(name string)
}

type DeptModeB interface { //仅包含了结构Dept的指针上的方法
    Relocate(building string, floor uint8)
}
```

# Go语言基础——接口（续）

结构Dept实例实现了哪个接口：

```
dept1 :=
    Dept{
        name: "MySohu",
        building: “Media",
        floor: 7}
switch v := interface{}(dept1).(type) {
case DeptModeFull:
    fmt.Printf("The dept1 is a DeptModeFull.\n")
case DeptModeB:
    fmt.Printf("The dept1 is a DeptModeB.\n")
case DeptModeA:
    fmt.Printf("The dept1 is a DeptModeA.\n")
default:
    fmt.Printf("The type of dept1 is %v\n", v)
} // => The dept1 is a DeptModeA.
```

# Go语言基础——接口（续2）

结构Dept实例的指针实现了哪些接口：

```
deptPtr1 := &dept1
if _, ok := interface{}(deptPtr1).(DeptModeFull); ok {
    fmt.Printf("The deptPtr1 is a DeptModeFull.\n")
}
if _, ok := interface{}(deptPtr1).(DeptModeA); ok {
    fmt.Printf("The deptPtr1 is a DeptModeA.\n")
}
if _, ok := interface{}(deptPtr1).(DeptModeB); ok {
    fmt.Printf("The deptPtr1 is a DeptModeB.\n")
}
// => The deptPtr1 is a DeptModeFull.
// => The deptPtr1 is a DeptModeA.
// => The deptPtr1 is a DeptModeB.
```

# Go语言基础——接口（续3）

为什么deptPtr1被判定为全部三个接口DeptModeFull、DeptModeA和DeptModeB的实现？而dept1只实现了接口DeptModeA？

依据Go语言规范：

- 结构Dept的方法集中仅包含方法接收者为Dept的方法，即：Name()和SetName()。所以，结构Dept的实例仅为DeptModeA的实现。
- 结构的指针*Dept的方法集包含了方法接受者为Dept和*Dept的方法，即：Name()、SetName()和Relocate()。所以，接口Dept的实例的指针为全部三个接口—DeptModeFull、DeptModeA和DeptModeB的实现。

# Go语言基础——接口（续4）

继续延伸：调用方法时发生的隐形转换：

```
dept1.Relocate("Media", 12)
fmt.Printf("Dept: %v\n", dept1)
fmt.Printf("Dept name: %v\n", deptPtr1.Name())

// => Dept: {MySohu Media 12 }
// => Dept name: MySohu
```

？ 那为什么结构Dept的实例却可以调用其指针方法集中的方法？

依据Go语言规范：

- 如果结构的实例x是“可被寻址的”，且&x的方法集中包含方法m，则x.m()为(&x).m()的速记（快捷方式）。

即：dept1是可被寻址的，且&dept1的方法集中包含方法Relocate()，则dept1.Relocate()为&dept1.Relocate()的快捷方式。

# Go语言基础——程序结构

目录结构：

<GOPATH>：自定义Go程序代码包的根目录
|__ src：Go程序源码文件的存放目录，一般每个项目会有一个子目录
|__ pkg：通过“go install”命令编译安装的二进制静态包文件（.a）的存放目录
|__ bin：通过“go install”命令编译安装的可执行文件的存放目录

比如：我有个项目的名字是“go_lib”，那么这个项目的源码就应该存放在这个目录下：

```
<GOPATH>
|__ src
    |__ go_lib
```

# Go语言基础——程序结构（续）

源代码文件与包：

- Go语言的源码是以UTF-8的形式存储的。
- Go语言以package来组织代码，所有的代码都必须在package中。
- 同一包中可以有多个源码文件（.go），且这些文件的包声明必须一致。
- 源码文件中包声明可以与目录不同，但编译后的静态文件（.a）会与该目录同名。
- 包内部的所有成员是共享的，即包内源码文件之间可以无障碍访问。而包外程序仅可访问名字首字母大写（相当于public）的成员。
- 生产代码和测试代码需要分别放在单独的文件中，测试代码文件以“_test.go”结尾，且这些文件需要在同一个目录中。

# Go语言基础——程序结构（续2）

源代码文件与包：

- 可执行程序（或者说程序入口）文件需要声明为main包，并有无参数和无返回值的入口函数，像这样：

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("Here we go!")
}
```

- 可以使用os.Exit(0)返回终止止进程，也可以用os.Args获取命令行行启动参数。

# Go语言基础——程序结构（续3）

## 源代码文件与包：

- 如果要在代码中使用另一个包的程序，首先需要导入（import）：

```
package main

import (
    "fmt" // 要使用fmt包
    "go_lib/logging" //要使用go_lib项目中的logging包
)

func main() {
    fmt.Println("Here we go!")
    logger := logging.GetSimpleLogger()
    logger.Infoln("Here we go!")
}
```

- 包路径 = 包根目录下路径/静态包主文文件名（不包含.a）

# Go语言基础——程序结构（续4）

多样的包导入方式：

```
import "go_lib/logging" // 正常模式：导入后即可通过logging.xxx访问包中成员。

import L "go_lib/logging" // 别名模式：导入包并为其设定一个别名，之后可以通过L.xxx访问包中成员。
import . "go_lib/logging" // 简便模式：导入包并可像访问当钱包成员那样访问该包的成员，即：直接通过xxx访问该包成员。
import _ "go_lib/logging" // 丢垃圾桶：仅执行该包中的初始化函数，然后将其扔掉（不真正使用）。
```

# Go语言基础——程序结构（续5）

包的初始化：

- 每个源码文件都可以定义一一个或多个包初始化函数func init() {}，同一包中可以在多个源码文件内定义任意个包初始化函数。
- 所有包初始化函数都会在main()之前、在单一线程上被调用，且仅执行一次。
- 编译器不能保证多个包初始化函数的执行次序。
- 包初始化函数在当前包所有全局变量初始化（零或初始化表达式值）完成后执行。
- 不能在程序代码中直接或间接调用用初始化函数。

# Go语言基础——程序结构（续6）

包的初始化：

```
package main

import (
    "go_lib/logging"
)

var logger logging.Logger

func init() {
    logger = logging.GetSimpleLogger()
}

func main() {
    logger.Infoln("Here we go!")
}
```

# Go语言基础——安装和环境设置

安装与设置：

1. 从 https://code.google.com/p/go/downloads/list 下载相应版本，portable版本即可。
2. 解压压缩包，并将go文件夹拷贝到适当目录下，如：“/usr/local”。
3. 设置Go根目录：export GOROOT=/usr/local/go。
4. 设置Go项目根目录，也即是前文所说的GOPATH，如：
   export GOPATH=$HOME/go-projects:$HOME/go-demo
5. 按照惯例，也为了让其他项目使用，用Go写的项目都应该放在<GOPATH>/src目录下，且子目录与项目同名。

# Go语言基础——构建工具

Go自带的命令行工具一览：

如果已经正确的安装和设置Go的话，在命令行上敲“go”后会出现：

```
Go is a tool for managing Go source code.

Usage:

        go command [arguments]

The commands are:

    build       compile packages and dependencies
    clean       remove object files
    doc         run godoc on package sources
    env         print Go environment information
    fix         run go tool fix on packages
    fmt         run gofmt on package sources
    get         download and install packages and dependencies
    install     compile and install packages and dependencies
    list        list packages
    run         compile and run Go program
    test        test packages
    tool        run specified go tool
    version     print Go version
    vet         run go tool vet on packages

Use "go help [command]" for more information about a command.
```

# Go语言基础——构建工具（续）

<GOPATH>/src/demo1.go：

```
package main

import (
    "flag"
    "fmt"
)

var name string

func init() {
    flag.StringVar(&name, "vn", " visitor ", "The name of the visitor.")
}

func main() {
    flag.Parse()
    fmt.Printf("Here we go, %s!\n", name)
}
```

# Go语言基础——构建工具（续2）

go run:

```
<GOPATH>/src$ go run demo1.go -vn "Harry"
Here we go, Harry!
```

go build:

```
<GOPATH>/src$ go build demo1.go
<GOPATH>/src$ ls
demo1  demo1.go
<GOPATH>/src$ ./demo1 -vn "Harry"
Here we go, Harry!
```

# 基础编程实战——Stack

<GOPATH>/src/part1/stack/stack.go :

```
package stack

type Stack interface {
    Clear()
    Len() uint
    Cap() uint
    Peek() interface{}
    Pop() interface{}
    Push(value interface{})
}
```

# 基础编程实战——Stack（续）

<GOPATH>/src/part1/stack/simple_stack.go :

```
package stack

type SimpleStack struct {
    capacity uint
    cursor uint
    container []interface{}
}

…… // 实现了Stack接口中定义的所有方法

func NewSimpleStack(myCapacity uint) Stack {
     return &SimpleStack{capacity: myCapacity}
}
```

# 基础编程实战——Stack（续2）

<GOPATH>/src/part1/stack/simple_stack_test.go :

```
package stack

import (
    "fmt"
    "runtime/debug"
    "testing"
)
……

func TestOps(t *testing.T) {
    ……
}
```

# 基础编程实战——Stack（续3）

```
<GOPATH>/src/part1/stack$ go test
PASS
ok      stack   0.115s
```

Go语言内置了测试工具和标准库。

这也充分体现了Go语言是“为软件工程而生”的。

编程和测试是密不可分的。

一定要用测试来为你的程序保驾护航！

# 基础编程实战——Stack（续4）

```
<GOPATH>/src/part1/stack$ go test -v
```

```
=== RUN TestInterface
--- PASS: TestInterface (0.00 seconds)
=== RUN TestClear
--- PASS: TestClear (0.00 seconds)
        simple_test.go:14: Ignore the cap error.

=== RUN TestLen
--- PASS: TestLen (0.00 seconds)
        simple_test.go:14: Ignore the cap error.

=== RUN TestCap
--- PASS: TestCap (0.00 seconds)
        simple_test.go:14: Ignore the cap error.

=== RUN TestOps
--- PASS: TestOps (0.00 seconds)
        simple_test.go:14: Ignore the cap error.

PASS
ok      stack   0.088s
```

# 第一部分小结

Go基础编程及实战的源码可以到下列网址找到：

```
https://github.com/hyper-carrot/my_slides/tree/master/go_programming_practice/src/part1
```

大家可以依据前面所讲的内容，把代码运行起来...

大家对此slide和源码有任何意见和建议都可以给我发issue。

现在，欢迎进入Go的世界！

# Go并发编程——概览

Go在并发编程方面给我们带来了什么？

- ✓ goroutines
  一种比线程更轻量，比协程更灵活的语言级并发机制。

- ✓ channel
  通道，可以在普通和Goruntine场景下作为数据甚至代码的载体和通讯手段。

- ✓ sync
  提供了一些经典的并发同步机制，比如锁和原子操作。

# Go并发编程——goroutines

*goroutines之所以被叫做goroutines，是因为现存的一些术语——线程、协程和进程等等——都传达了错误的涵义。*

——摘自官方文档《Effective Go》

那……什么是正确的？

*Do not communicate by sharing memory. Instead, share memory by communicating.*

# Go并发编程——goroutines（续）

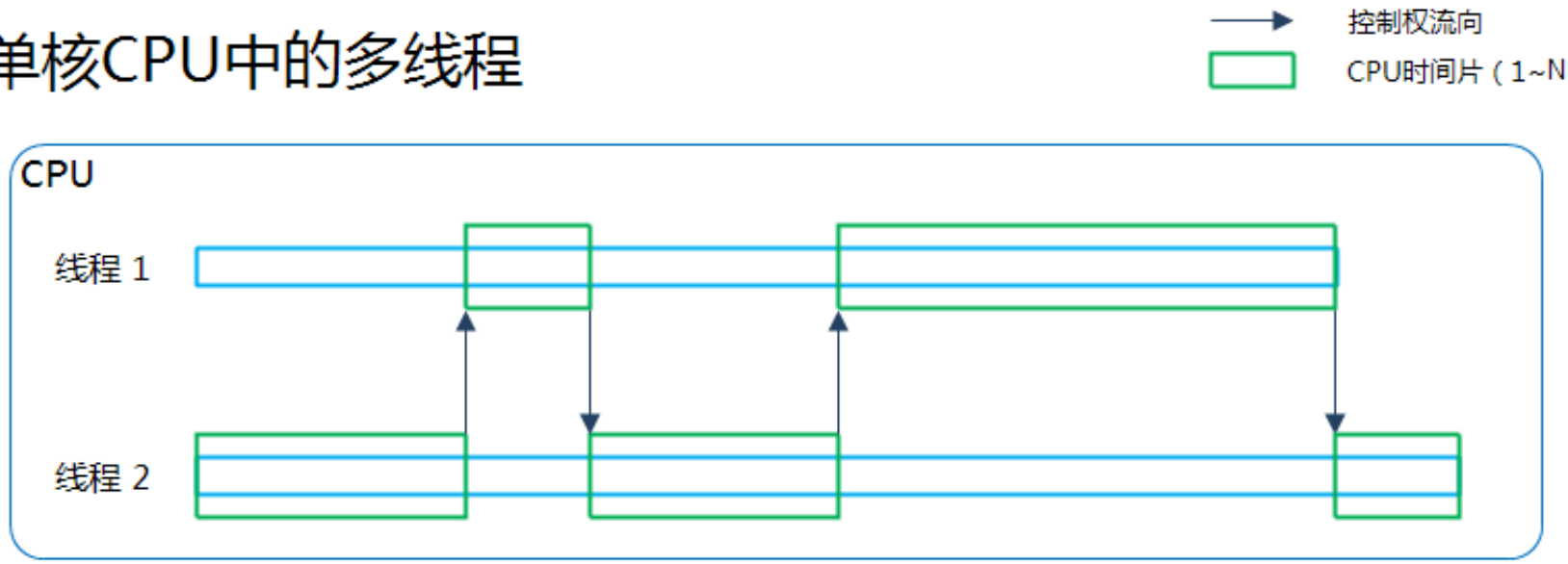
goroutine是这样的：

- 相当于一个函数入口，它与在同一地址空间中的其他goroutines并行的执行。（注意：是“并行”！）
- 由Go语言特有的关键字“go”来启动。
- 一个goroutine仅需4~5KB的栈内存（按需增减），与线程栈相比也少得多。这让“同时运行”成千上万个并发任务成为可能。（注意：不要滥用！）
- 通过基于OS线程的多路复用技术来实现更灵活的调度和管理，这也为并行执行提供了底层支持。而协程（coroutine），通常只是通过在同一线程上的切换调度（yield/resume）来实现并发执行。

# Go并发编程——goroutines（续2）

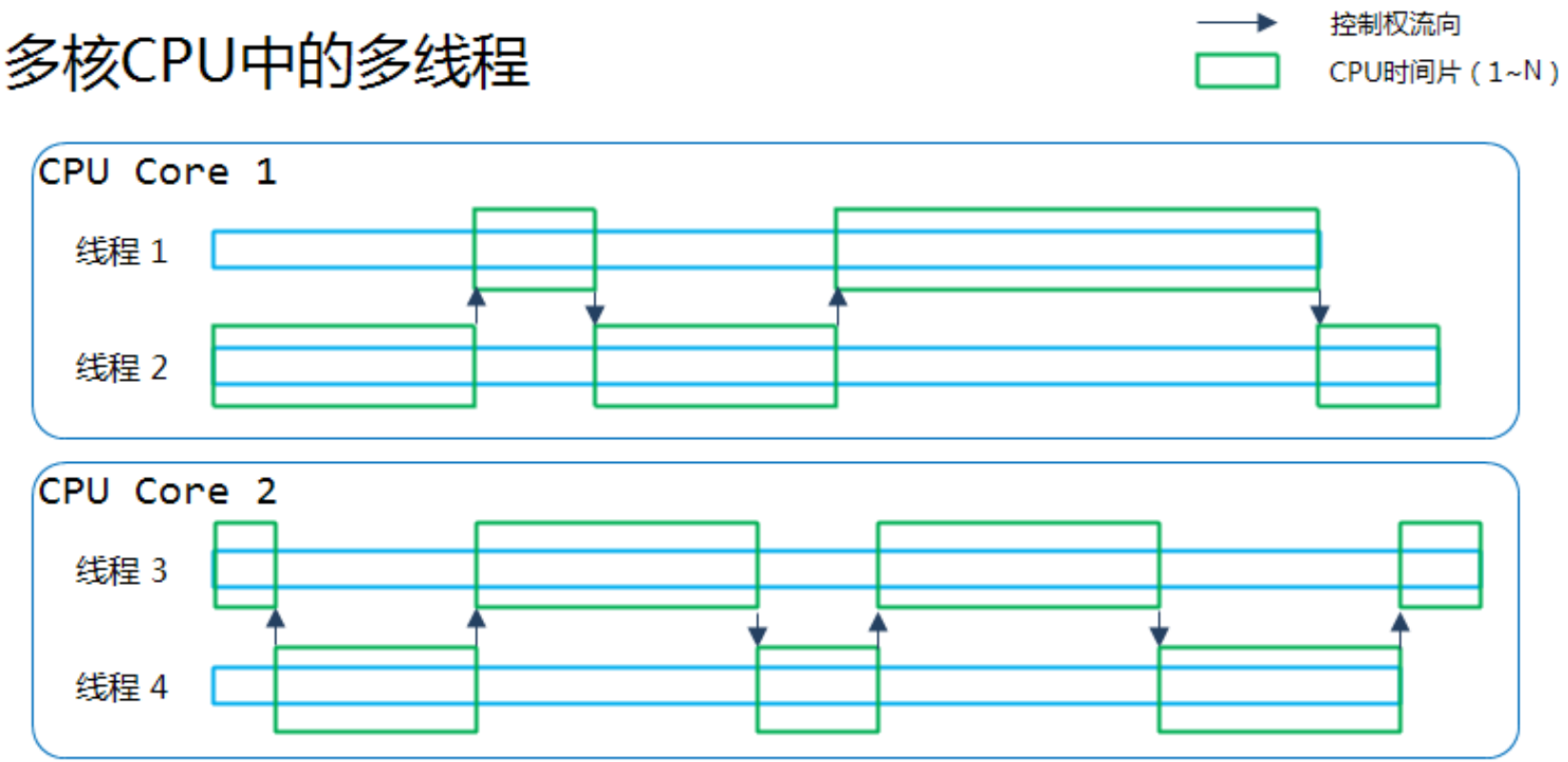
什么是并发：

单核CPU中的多线程



# Go并发编程——goroutines（续3）

多（核）CPU中的并发造就了并行：

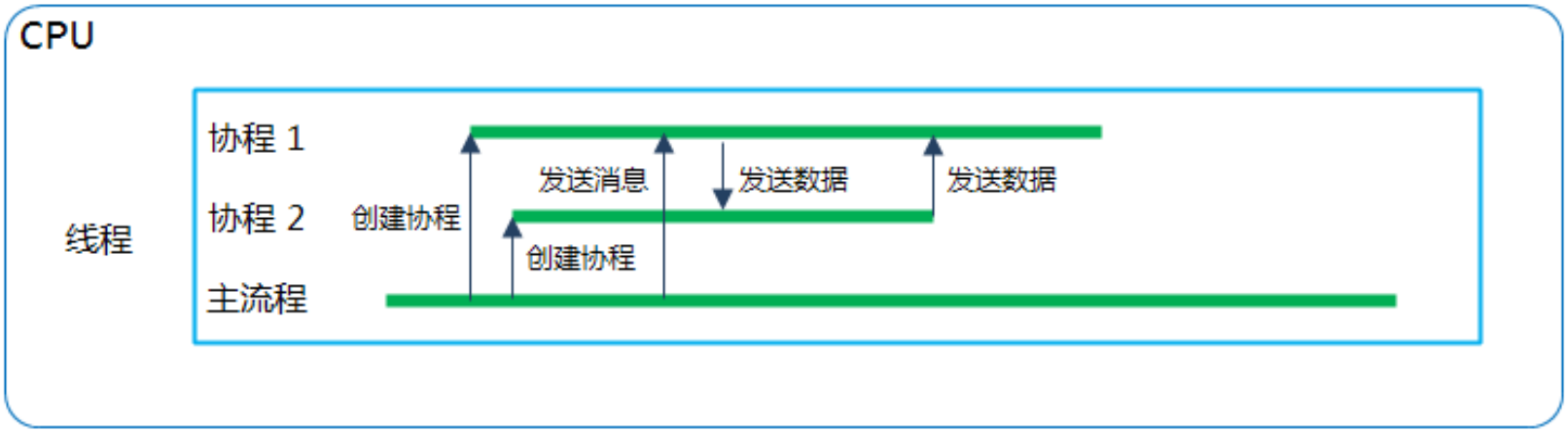


# Go并发编程——goroutines（续4）

协程（coroutine）：

旨在不额外创建线程/进程的前提下实现异步和并发。

协程的运作方式



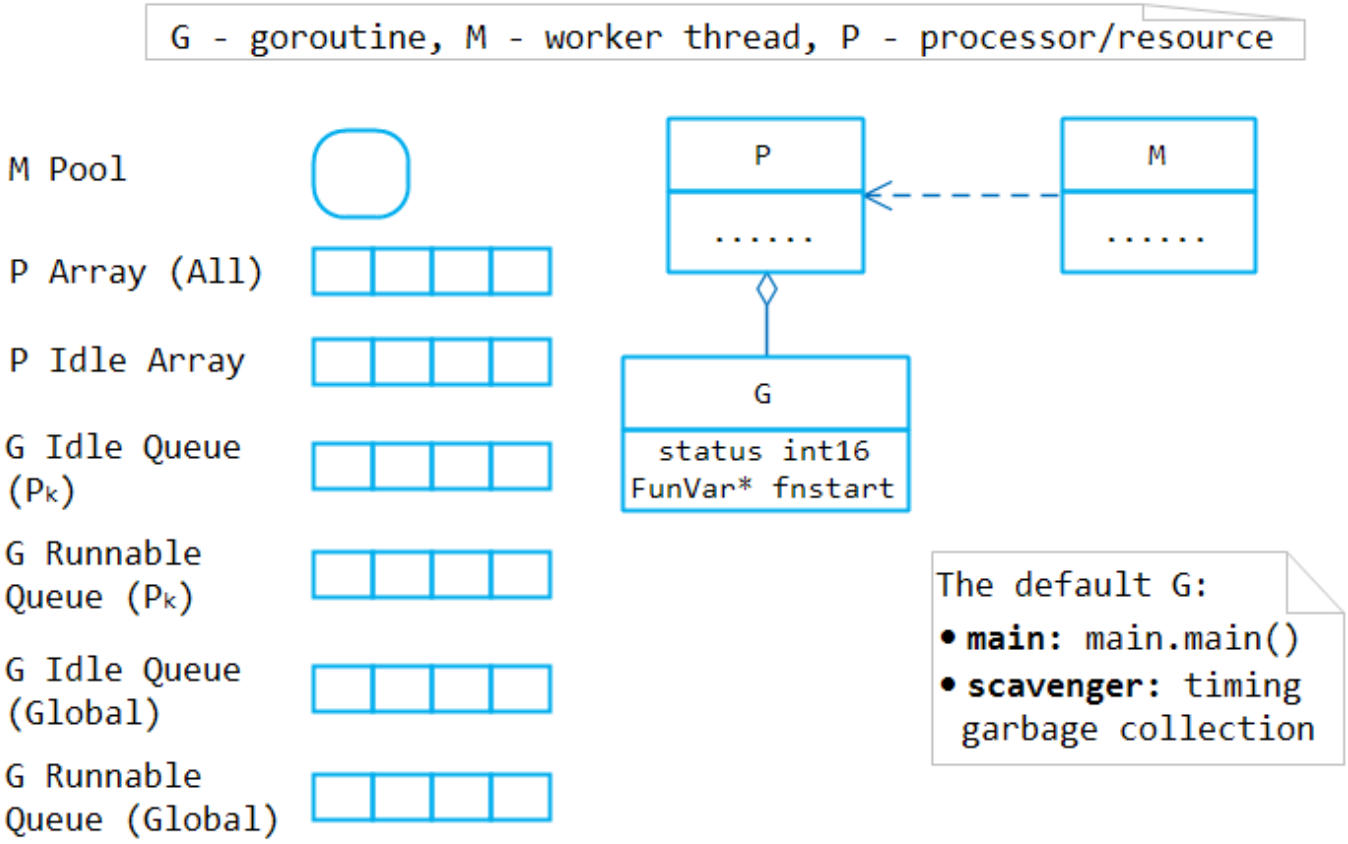
# Go并发编程——goroutines（续5）

协程（coroutine）在语言中：

- Lua 的 coroutine
- Python 的迭代器和生成器（yield），以及Greenlet
- Ruby & C# 的 Fiber（微线程、纤程）
- Erlang 的 Green Process
- Scala 的 Actor

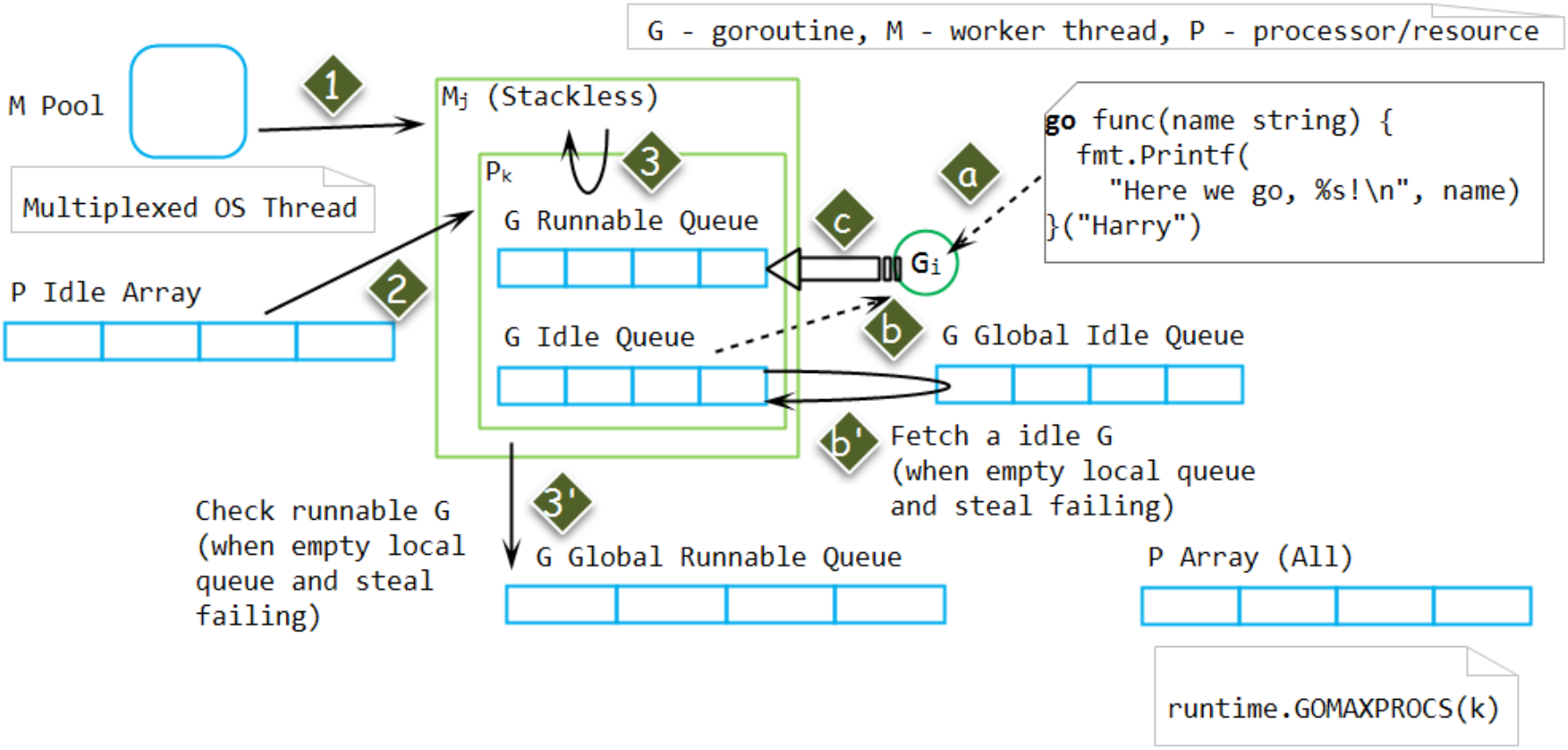
# Go并发编程——goroutines（续6）

goroutine — 更高级的协程（coroutine）：



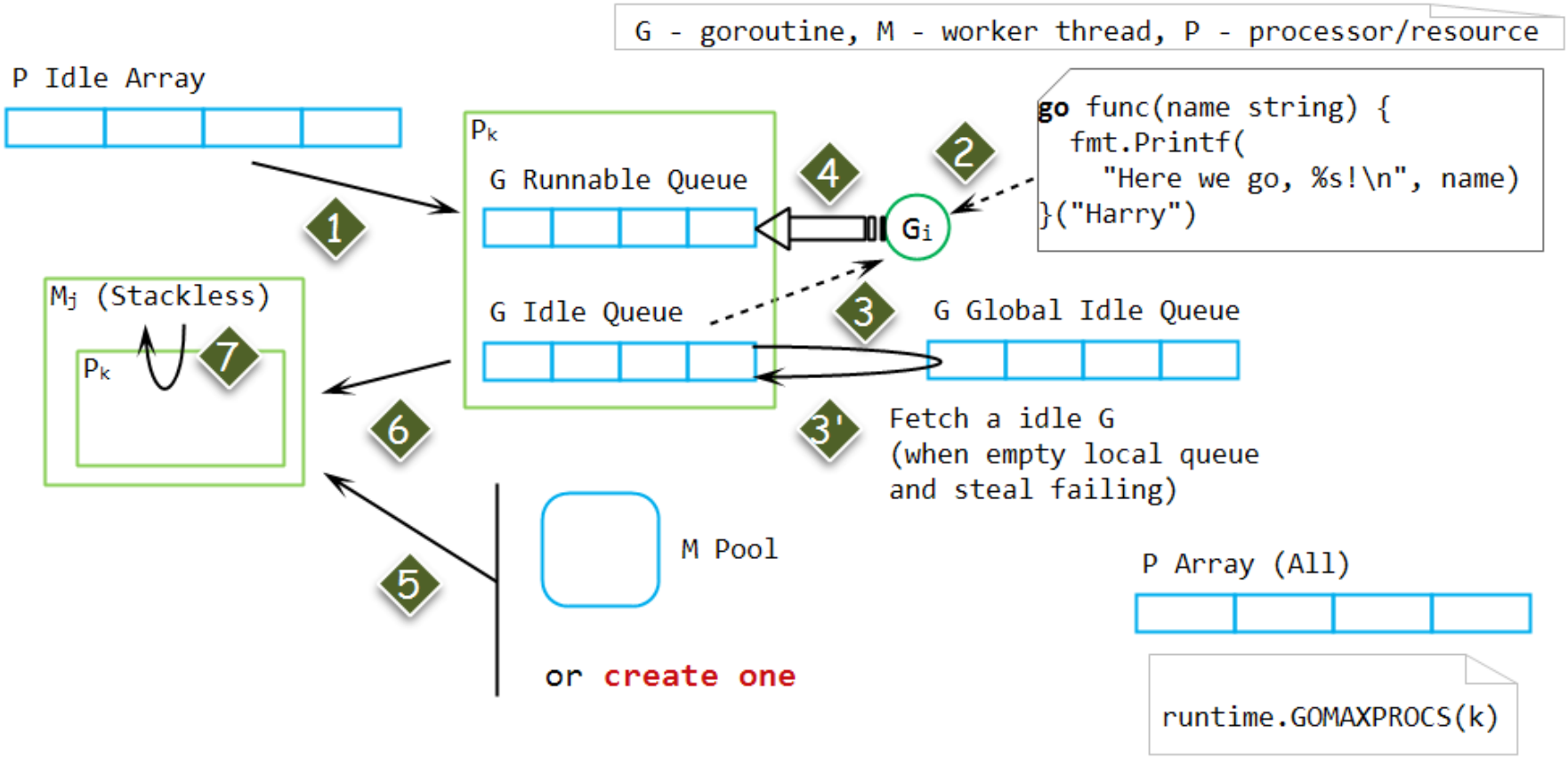
# Go并发编程——goroutines（续7）

准备执行goroutine：



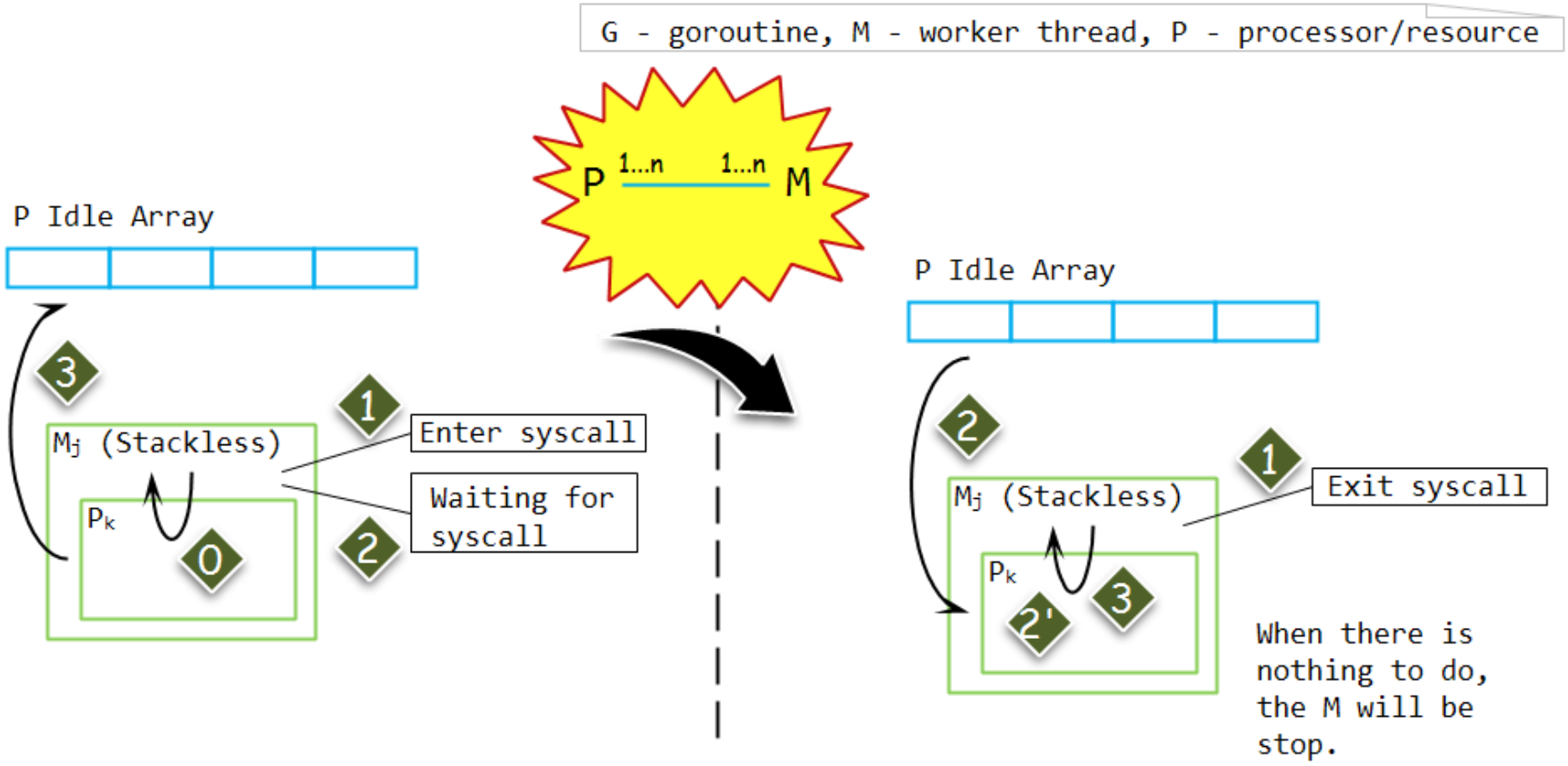
# Go并发编程——goroutines（续8）

准备执行goroutine（另一种情况）：



# Go并发编程——goroutines（续9）

goroutines调度：



# Go并发编程——goroutines（续10

<GOPATH>/src/demo2.go：

```
package main

import (
    "fmt"
    "time"
)

func greet(name string) {
    fmt.Printf("Hello, %s!\n", name)
}

func main() {
    name := "Harry"
    go greet(name)
    time.Sleep(10 * time.Millisecond)
    fmt.Printf("Goodbye, %s!\n", name)
}
```

# Go并发编程——goroutines（续11

运行：

```
<GOPATH>/src$ go run demo2.go
Hello, Harry!
Goodbye, Harry!
```

```
    // "time"
    ……
    // time.Sleep(10 * time.Millisecond)
```

```
<GOPATH>/src$ go run demo2.go
Goodbye, Harry!
```

# Go并发编程——goroutines（续12

<GOPATH>/src/demo3.go：

```
package main

import (
    "fmt"
    "runtime"
)

func main() {
    runtime.GOMAXPROCS(runtime.NumCPU() - 1)
    go func() {
        fmt.Printf("GN: %d.\n", runtime.NumGoroutine())
        runtime.Goexit()
        fmt.Println("Exited.")
    }()
    runtime.Gosched()
    fmt.Println("Ended.")
}
```

# Go并发编程——goroutines（续13

运行：

```
<GOPATH>/src$ go run demo3.go
GN: 3.
Ended.
```

```
    ……
    // runtime.Gosched()
    fmt.Println("Ended.")
```

```
<GOPATH>/src$ go run demo3.go
Ended.
```

# Go并发编程——channel

*许多Goroutine会跑在同一个地址空间里，所以对共享内容的访问必须是同步的。同步可以使用sync包来实现（后面会讲到），但是在Go中这是严重不被推荐的。我们应该用Go的方式—channel—来处理goroutine之间的同步问题。*

——摘自《The Way To Go》

```
var ch1 chan = make(chan int)  // int类型的非缓冲通道
ch2 := make(chan int, 5) // int类型的缓冲通道
```

# Go并发编程——channel（续）

与channel有关的Happens Before：

- 对于非缓冲通道，“从通道接收数据”的操作一定会在“向通道发送数据”的操作完成之前发生。
- 对于缓冲通道，“向通道发送数据”的操作一定会在“从通道接收数据”的操作完成之前发生。

补：

用于启动goroutine的go语句一定会在这个goroutine开始执行之前执行。

# Go并发编程——channel（续2）

<GOPATH>/src/demo_chan.go：

```
package main

import (
    "fmt"
)

var ch = make(chan int)
var content string

func set() {
    content = "It's a unbuffered channel."
    <-ch
}

func main() {
    go set()
    ch <- 0
    fmt.Println(content)
}
```

“ch <- 0”语句会阻塞，直至“<- ch”执行完毕。

# Go并发编程——channel（续3）

运行：

```
<GOPATH>/src$ go run demo_chan.go
It's a unbuffered channel.
```

```
//  var ch = make(chan int)
    ……
    // <-ch
    ……
    // ch <- 0
```

```
<GOPATH>/src$ go run demo_chan.go
```

# Go并发编程——channel（续4）

<GOPATH>/src/demo_chan2.go （1）：

```
package main

import (
    "fmt"
    "time"
)

func main() {
    tick := make(chan int, 1)
    go func() {
        time.Sleep(2 * time.Second)
        count := 1
        for {
            time.Sleep(1 * time.Second)
            tick <- count
            count++
        }
    }()
```

# Go并发编程——channel（续5）

<GOPATH>/src/demo_chan2.go （2）：

```
    for v := range tick {
        fmt.Printf("%d\n", v)
        if v == 5 {
            break
        }
}
```

可以用迭代器从通道中取数据。tick是非缓冲通道，所以只有当通道中有数据时，这里才会继续执行。

运行：

```
<GOPATH>/src$ go run demo_chan2.go
```

```
1
2
3
4
5
```

# Go并发编程——channel（续6）

让我们换一种写法：

```
func main() {
    tick := make(chan int, 1)
    go func() {
        time.Sleep(2 * time.Second)
        count := 1
        for {
            time.Sleep(1 * time.Second)
            tick <- count
            if count == 5 {
                close(tick)
            }
            count++
        }
    }()
    for v := range tick {
        fmt.Printf("%d\n", v)
    }
}
```

当通道被关闭时，迭代器的执行会自动结束。

# Go并发编程——channel（续7）

运行：

```
<GOPATH>/src$ go run demo_chan2.go
```

```
1
2
3
4
5
```

# Go并发编程——channel（续8）

单向channel：

可将 channel 指定为单向通道。比如：“<-chan string”仅能从通道接收字符串，而“chan<- string”仅能向通道发送字符串。

<GOPATH>/src/demo_chan3.go ：

```
package main

func receive(over chan<- bool) {
    <-over
}

func main() {
    o := make(chan bool)
    go receive(o)
    <-o
}
```

# Go并发编程——channel（续9）

运行：

```
<GOPATH>/src$ go run demo_chan3.go
# command-line-arguments
./demo_chan3.go:4: invalid operation: <-over (receive from
   send-only type chan<- bool)
```

修正：

```
func receive(over chan<- bool) {
    over <- true
}
```

再运行：

```
<GOPATH>/src$ go run demo_chan3.go
```

# Go并发编程——channel（续10）

<GOPATH>/src/demo_chan4.go（1）：

```
package main

func receive(c <-chan int, over chan<- bool) {
    for v := range c {
        println(v)
    }
    over <- true
}

func send(c chan<- int) {
    for i := 0; i < 3; i++ {
        c <- i
    }
    close(c)
}
```

# Go并发编程——channel（续11）

<GOPATH>/src/demo_chan4.go（2）：

```
func main() {
    c := make(chan int)
    o := make(chan bool)
    go receive(c, o)
    go send(c)
    <-o
}
```

等待结束信号。

运行：

```
<GOPATH>/src$ go run demo_chan4.go
0
1
2
```

# Go并发编程——channel（续12）

Channel上的switch——select：

- 如果需要从多个不同的并发执行的goroutines获取值，则可以用select来协助完成。
- select可以监听多个channel的输入数据，一个channel对应一个case。*当任何被监听的channel中都没有的数据的时候，select语句块会阻塞。*
- select可以有一个default子句。*当任何被监听的channel中都没有的数据的时候，default子句将会被执行。*
- 与switch不同的是，select语句块中不能出现fallthrough。

# Go并发编程——channel（续13）

<GOPATH>/src/demo_chan5.go（1）：

```
package main

import (
    "fmt"
)

func send(ch chan<- int, number int) {
    ch <- number
    close(ch)
}

func main() {
    ch1 := make(chan int)
    ch2 := make(chan int)
    go send(ch1, 2)
    go send(ch2, 1)
```

# Go并发编程——channel（续14）

<GOPATH>/src/demo_chan5.go（2）：

```
    select {
    case v1 := <-ch1:
        fmt.Printf("%s: %d\n", "CH1", v1)
    case v2 := <-ch2:
        fmt.Printf("%s: %d\n", "CH2", v2)
    }
    fmt.Println("End.")
}
```

运行：

```
<GOPATH>/src$ go run demo_chan5.go
CH1: 1
End.
```

# Go并发编程——channel（续15）

<GOPATH>/src/demo_chan6.go（2'）：

```
    select {
    case v1 := <-ch1:
        fmt.Printf("%s: %d\n", "CH1", v1)
    case v2 := <-ch2:
        fmt.Printf("%s: %d\n", "CH2", v2)
    default:
        fmt.Println("None of the channel operations can proceed!")
    }
    fmt.Println("End.")
}
```

运行：

```
<GOPATH>/src$ go run demo_chan5.go
None of the channel operations can proceed!
End.
```

# Go并发编程——channel（续16）

select的行为：

1. 当所有的被监听channel中都无数据时，则select会一直等到其中一个有数据为止。
2. 当多个被监听channel中都有数据时，则select会随机选择一个case执行。
3. 当所有的被监听channel中都无数据，且default子句存在时，则default子句会被执行。
4. 如果想持续的监听多个channel的话需要用for语句协助。

# Go并发编程——channel（续17）

<GOPATH>/src/demo_chan6.go（1）：

```
package main

import (
    "fmt"
)

func send(ch chan<- int) {
    for i := 0; i < 5; i++ {
        ch <- i
    }
    close(ch)
}

func main() {
    ch1 := make(chan int)
    ch2 := make(chan int)
    go send(ch1)
    go send(ch2)
```

# Go并发编程——channel（续18）

<GOPATH>/src/demo_chan6.go（2）：

```
    for {
        select {
        case v1 := <-ch1:
            fmt.Printf("%s: %d\n", "CH1", v1)
        case v2 := <-ch2:
            fmt.Printf("%s: %d\n", "CH2", v2)
        }
    }
    fmt.Println("End.")
}
```

运行：

```
<GOPATH>/src$ go run demo_chan6.go
CH1: 0
CH1: 1
```

# Go并发编程——channel（续19）

<GOPATH>/src/demo_chan6.go 运行（2）：

```
CH2: 0
CH1: 2
CH2: 1
CH2: 2
CH2: 3
CH1: 3
CH2: 4
CH1: 4
CH1: 0
CH1: 0
CH1: 0
CH2: 0
CH2: 0
……
```

# Go并发编程——channel（续20）

<GOPATH>/src/demo_chan6.go（2'）：

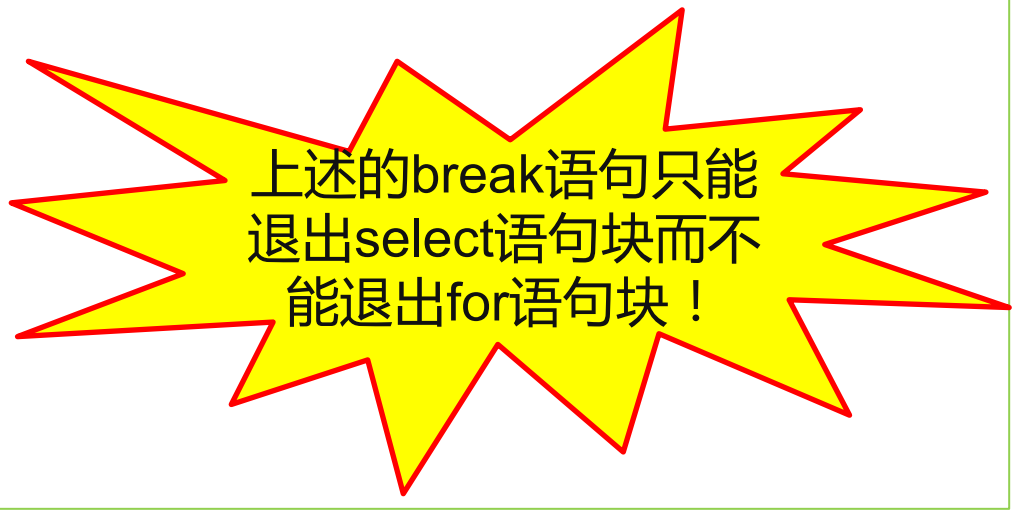
```
    for {
        select {
        case v1, ok := <-ch1:
            if !ok {
                break
            }
            fmt.Printf("%s: %d\n", "CH1", v1)
        case v2, ok := <-ch2:
            if !ok {
                break
            }
            fmt.Printf("%s: %d\n", "CH2", v2)
        }
    }
   fmt.Println("End.")
}
```

# Go并发编程——channel（续22）

运行：

```
<GOPATH>/src$ go run demo_chan6.go
CH1: 0
CH1: 1
CH2: 0
CH1: 2
CH2: 1
CH2: 2
CH2: 3
CH1: 3
CH2: 4
CH1: 4

// 程序还是会一直运行下去...
```



# Go并发编程——channel（续23）

<GOPATH>/src/demo_chan6.go（2''）：

```
    overTag := make(chan int)
    go func() {
        for {
            select {
            case v1, ok := <-ch1:
                if !ok {
                    overTag <- 0
                    break
                }
                fmt.Printf("%s: %d\n", "CH1", v1)
            case v2, ok := <-ch2:
                if !ok {
                    overTag <- 1
                    break
                }
                fmt.Printf("%s: %d\n", "CH2", v2)
            }
        }
    }()
```

# Go并发编程——channel（续24）

<GOPATH>/src/demo_chan6.go（3）：

```
    overTags := make([]int, 2)
    for v := range overTag {
        overTags[v] = 1
        if overTags[0] == 1 && overTags[1] == 1 {
            break
        }
    }
    fmt.Println("End.")
}
```

# Go并发编程——channel（续25）

运行：

```
<GOPATH>/src$ go run demo_chan6.go
CH1: 0
CH2: 0
CH1: 1
CH2: 1
CH1: 2
CH2: 2
CH2: 3
CH1: 3
CH2: 4
CH1: 4
End.
```

接收到了所有数据，并正常退出。

# Go并发编程——channel（续26）

channel与time包：

time包里提供了一些有意思的功能，这些功能是与channel相结合的。

- After函数：起到定时器的作用，指定的纳秒后会向返回的channel中放入一个当前时间（time.Time）的实例。
- Tick函数：起到循环定时器的作用，每过指定的纳秒后都会向返回的channel中放入一个当前时间（time.Time）的实例。
- Ticker结构：循环定时器。Tick函数就是包装它来完成功能的。该定时器可以被中止。

# Go并发编程——channel（续27）

<GOPATH>/src/demo_chan7.go（1）：

```
package main

import (
    "fmt"
    "time"
)

func main() {
    ch1 := make(chan int)
    go func() {
        for i := 0; i < 10; i++ {
            ch1 <- i
            time.Sleep(1 * time.Second)
        }
    }()
    timeout := time.After(5 * time.Second)
    overTag := make(chan bool)
```

从此刻算起，
5秒后超时。

# Go并发编程——channel（续28）

<GOPATH>/src/demo_chan7.go（2）：

```
    go func() {
        for {
            select {
            case v1, ok := <-ch1:
                if !ok {
                    overTag <- true
                    break
                }
                fmt.Printf("%s: %d\n", "CH1", v1)
            case <-timeout:
                fmt.Println("Timeout.")
                overTag <- true
            }
        }
    }()
    <-overTag
    fmt.Println("End.")
}
```

# Go并发编程——channel（续29）

运行：

```
<GOPATH>/src$ go run demo_chan7.go
CH1: 0
CH1: 1
CH1: 2
CH1: 3
CH1: 4
Timeout.
End.
```

# Go并发编程——channel（续30）

<GOPATH>/src/demo_chan8.go（1）：

```
package main

import (
    "fmt"
    "time"
)

func main() {
    ch1 := make(chan int)
    go func() {
        for i := 0; i < 5; i++ {
            ch1 <- i
            time.Sleep(1 * time.Second)
        }
        close(ch1)
    }()
    tick := time.Tick(1 * time.Second)
    overTag := make(chan bool)
```

每1秒会将一个“当前时间”数据放入通道tick中。

# Go并发编程——channel（续31）

<GOPATH>/src/demo_chan8.go（2）：

```
    go func() {
        for {
            select {
            case <-tick:
                v, ok := <-ch1
                if !ok {
                    overTag <- true
                    fmt.Println("Closed channel.")
                    break
                }
                fmt.Printf("%s: %d\n", "CH1", v)
            }
        }
    }()
    <-overTag
    fmt.Println("End.")
}
```

# Go并发编程——channel（续32）

运行：

```
<GOPATH>/src$ go run demo_chan8.go
CH1: 0
CH1: 1
CH1: 2
CH1: 3
CH1: 4
Closed channel.
End.
```

# Go并发编程——channel（续33）

<GOPATH>/src/demo_chan9.go（1）：

```
package main

import (
    "fmt"
    "time"
)

func main() {
    ch1 := make(chan int)
    ticker := time.NewTicker(1 * time.Second)
```

NewXxx(…)相当于Java中的工厂方法，在Go中属于实例化struct的惯用法，使用相当广泛。

# Go并发编程——channel（续34）

<GOPATH>/src/demo_chan9.go（2）：

```
go func() {
    for {
        select {
        case <-ticker.C:
            select {
            case ch1 <- 1:
            case ch1 <- 2:
            case ch1 <- 3:
            }
        case <-time.After(2 * time.Second):
            fmt.Println("Time out. Stopped ticker.")
            close(ch1)
            return
        }
    }
}()
```

select语句块也可以用在发送端，这相当于每次随机选择一个case执行。

每次等待ticker的“事件”到来，限时2秒，若超时则关闭ch1并结束该goroutine。

# Go并发编程——channel（续35）

<GOPATH>/src/demo_chan9.go（3）：

```
    overTag := make(chan bool)
    go func() {
        for {
            select {
            case v, ok := <-ch1:
                if !ok {
                    fmt.Println("Closed channel.")
                    overTag <- true
                    break
                }
                fmt.Printf("%s: %d\n", "CH1", v)
            }
        }
    }()
    time.Sleep(5 * time.Second)
    fmt.Println("Stop ticker.")
    ticker.Stop()
    <-overTag
    fmt.Println("End.")
}
```

5秒后停止ticker。

# Go并发编程——channel（续36）

运行：

```
<GOPATH>/src$ go run demo_chan9.go
CH1: 3
CH1: 2
CH1: 1
CH1: 2
Stop ticker.
CH1: 1
Time out. Stopped ticker.
Closed channel.
Closed channel.
End.
```

# 并发编程实战——NIO

<GOPATH>/src/part2/nio/base.go :

```
package nio

import (
    "net"
    "time"
)

type PubSubListener interface {
    Init(addr string) error
    Listen(handler func(conn net.Conn)) error
    Close() bool
    Addr() net.Addr
}
```

需传入一个用于处理请求的函数。

# 并发编程实战——NIO（续）

<GOPATH>/src/part2/nio/base.go（2）：

```
type PubSubSender interface {
    Init(remoteAddr string, timeout time.Duration) error
    Send(content string) error
    Receive(delim byte) <-chan PubSubContent
    Close() bool
    Addr() net.Addr
    RemoteAddr() net.Addr
}
```

注意这里，含义：
1. 这个方法被调用后会立即返回。（非阻塞的）
2. 这个被返回的通道仅能接收数据而不能发送数据。（单向的）

# 并发编程实战——NIO（续2）

<GOPATH>/src/part2/nio/base.go（3）：

```
type PubSubContent struct {
    content string
    err error
}

func (self PubSubContent) Content() string {
    return self.content
}

func (self PubSubContent) Err() error {
    return self.err
}

func NewPubSubContent(content string, err error) PubSubContent {
    return PubSubContent{content: content, err: err}
}
```

# 并发编程实战——NIO（续3）

<GOPATH>/src/part2/nio/tcp.go:

```
package nio

import (
    "errors"
    "net"
    "sync"
    "time"
)

type TcpListener struct {
    listener net.Listener
    active bool
    lock *sync.Mutex
}
```

还记得之前说过的sync包吗？

这里是锁的一个官方实现。

# 并发编程实战——NIO（续4）

<GOPATH>/src/part2/nio/tcp.go（2）:

```
func (self *TcpListener) Init(addr string) error {
    self.lock.Lock()
    defer self.lock.Unlock()
    if self.active {
        return nil
    }
    ln, err := net.Listen("tcp", addr)
    if err != nil {
        return err
    }
    self.listener = ln
    self.active = true
    return nil
}
```

锁一般是这么用的…

# 并发编程实战——NIO（续5）

## <GOPATH>/src/part2/nio/tcp.go（3）：

```
func (self *TcpListener) Listen(handler func(conn net.Conn)) error {
    if !self.active {
        return errors.New("Send Error: Uninitialized listener!")
    }
    go func() {
        for {
            conn, err := self.listener.Accept()
            if err != nil {
                Logger().Errorf(
                    "Listener: Accept Error: %s\n", err)
                continue
            }
            go handler(conn)
        }
    }()
    return nil
}
```

创建一个goroutine来监听端口。

当请求到来时，再创建新的goroutine来处理请求。

# 并发编程实战——NIO（续6）

<GOPATH>/src/part2/nio/tcp.go（4）：

```
func (self *TcpListener) Close() bool {
    self.lock.Lock()
    defer self.lock.Unlock()
    if self.active {
        self.listener.Close()
        self.active = false
        return true
    } else {
        return false
    }
}

func (self *TcpListener) Addr() net.Addr {……}

func NewTcpListener() PubSubListener {
    return &TcpListener{lock: new(sync.Mutex)}
}
```

新建（new）一个锁来初始化TcpListener结构的实例。

# 并发编程实战——NIO（续7）

<GOPATH>/src/part2/nio/tcp.go（5）：

```
type TcpSender struct {
    active bool
    lock *sync.Mutex
    conn net.Conn
}

func (self *TcpSender) Init(
    remoteAddr string, timeout time.Duration) error {
    self.lock.Lock()
    defer self.lock.Unlock()
    if !self.active {
        conn, err := net.DialTimeout("tcp", remoteAddr, timeout)
        ……
    }
    return nil
}
```

# 并发编程实战——NIO（续8）

## <GOPATH>/src/part2/nio/tcp.go（6）：

```
func (self *TcpSender) Send(content string) error {
    self.lock.Lock()
    defer self.lock.Unlock()
    if !self.active {
        return errors.New("Send Error: Uninitialized sender!")
    }
    _, err := WriteToTcp(self.conn, content)
    return err
}

func (self *TcpSender) Receive(delim byte) <-chan PubSubContent {
    respChan := make(chan PubSubContent, 1)
    go func(conn net.Conn, ch chan<- PubSubContent) {
        content, err := ReadFromTcp(conn, DELIM)
        ch <- NewPubSubContent(content, err)
    }(self.conn, respChan)
    return respChan
}
```

注意这里通道的用法！

# 并发编程实战——NIO（续9）

## <GOPATH>/src/part2/nio/tcp.go（7）:

```
func (self *TcpSender) Addr() net.Addr {……}
func (self *TcpSender) RemoteAddr() net.Addr {……}

func (self *TcpSender) Close() bool {
    self.lock.Lock()
    defer self.lock.Unlock()
    if self.active {
        self.conn.Close()
        self.active = false
        return true
    } else {
        return false
    }
}

func NewTcpSender() PubSubSender {
    return &TcpSender{lock: new(sync.Mutex)}
}
```

# 并发编程实战——NIO（续10）

<GOPATH>/src/part2/nio/tcp_test.go:

```
package nio

import (
    "bytes"
    "fmt"
    "net"
    "strings"
    "sync"
    "testing"
    "time"
)
……
func TestMainFuncs(t *testing.T) {
    serverAddr := "127.0.0.1:8080"
    var listener PubSubListener = NewTcpListener()
    ……
    err := listener.Init(serverAddr)
    ……
```

# 并发编程实战——NIO（续11）

<GOPATH>/src/part2/nio/tcp_test.go（2）：

```
requestHandler := func(conn net.Conn) {
    for {
        content, err := ReadFromTcp(conn, DELIM)
        if err != nil {
            ……
        } else {
            ……
            if strings.HasSuffix(content, "q!") {
                t.Log("Listener: Quit!")
                break
            }
            ……
            n, err := WriteToTcp(conn, resp)
            ……
        }
    }
}
```

按照Listen方法的定义，需要定义一个用来处理请求的函数。

如果接收到了约定好的带有特殊标记的数据，就结束监听。

# 并发编程实战——NIO（续12）

<GOPATH>/src/part2/nio/tcp_test.go（3）：

```
    err = listener.Listen(requestHandler)
    if err != nil {
        t.Errorf("Listener: Error: %s\n", err)
        t.FailNow()
    }
    var wg sync.WaitGroup
    wg.Add(2)
    go func() {
        defer wg.Done()
        multiSend("127.0.0.1:8080", "S1", 2, (2 * time.Second), t)
    }()
    go func() {
        defer wg.Done()
        multiSend("127.0.0.1:8080", "S2", 1, (2 * time.Second), t)
    }()
    wg.Wait()
    listener.Close()
}
```

组团进行同步！

当调用与传入Add方法的数相同的Done方法时，Wait结束阻塞。

# 并发编程实战——NIO（续13）

<GOPATH>/src/part2/nio/tcp_test.go（4）：

```
func multiSend(
    remoteAddr string,
    clientName string,
    number int,
    timeout time.Duration,
    t *testing.T) {
   sender := NewTcpSender()
   ……
   err := sender.Init(remoteAddr, timeout)
   ……
   for i := 0; i < number; i++ {
        ……
        err := sender.Send(content)
        ……
        respChan := sender.Receive(DELIM)
        var resp PubSubContent
```

# 并发编程实战——NIO（续14）

<GOPATH>/src/part2/nio/tcp_test.go（5）：

```
        timeoutChan := time.After(1 * time.Second)
        select {
        case resp = <-respChan:
        case <-timeoutChan:
                break
        }
        if err = resp.Err(); err == nil {
            respContent := resp.Content()
        } else {
            ……
        }
    }
    content := generateTestContent(fmt.Sprintf("%s-q!", clientName))
    t.Logf("%s: Send content: '%s'\n", clientName, content)
    err = sender.Send(content)
    ……
    sender.Close()
}
```

还记得select以及超时设置和处理吗？

需要用特定标记告诉监听器，这次通讯可以结束了。

# 并发编程实战——NIO（续15）

运行测试：

```
<GOPATH>/src/part2/nio$ go test -v
```

```
=== RUN TestMainFuncs
--- PASS: TestMainFuncs (0.04 seconds)
        tcp_test.go:20: Start Listening at address 127.0.0.1:8080 ...
        tcp_test.go:69: Initializing sender (Sender1) (remote address: 127.0.0.1:8080, timeout: 2000000000) ...
        tcp_test.go:69: Initializing sender (Sender2) (remote address: 127.0.0.1:8080, timeout: 2000000000) ...
        tcp_test.go:80: Sender2: Send content: 'Sender2-0       '
        tcp_test.go:80: Sender1: Send content: 'Sender1-0       '
        tcp_test.go:32: Listener: Received content: 'Sender2-0  '
        tcp_test.go:43: Listener: Send response: 'Resp: Sender2-0       ' (n=16)
        tcp_test.go:32: Listener: Received content: 'Sender1-0  '
        tcp_test.go:97: Sender2: Received response: 'Resp: Sender2-0    '
        tcp_test.go:101: Sender2: Send content: 'Sender2-q!     '
        tcp_test.go:43: Listener: Send response: 'Resp: Sender1-0       ' (n=16)
        tcp_test.go:32: Listener: Received content: 'Sender2-q! '
        tcp_test.go:35: Listener: Quit!
        tcp_test.go:97: Sender1: Received response: 'Resp: Sender1-0    '
        tcp_test.go:80: Sender1: Send content: 'Sender1-1       '
        tcp_test.go:32: Listener: Received content: 'Sender1-1  '
        tcp_test.go:43: Listener: Send response: 'Resp: Sender1-1       ' (n=16)
        tcp_test.go:97: Sender1: Received response: 'Resp: Sender1-1    '
        tcp_test.go:101: Sender1: Send content: 'Sender1-q!     '
        tcp_test.go:32: Listener: Received content: 'Sender1-q! '
        tcp_test.go:35: Listener: Quit!
PASS
ok      part2/nio       0.213s
```

# 第二部分小结

Go基础编程及实战的源码可以到下列网址找到：

```
https://github.com/hyper-carrot/my_slides/tree/master/go_programming_practice/src/part2
```

大家可以依据前面所讲的内容，把代码运行起来...

大家对此slide和源码有任何意见和建议都可以给我发issue。

并发编程是Go语言的特性中最靓和最给力的部分！

# To be continue...

Talk is cheap, show me the code!