Golang 从入门到不放弃

2017.2

目录

- 示例
  - 代码结构、包、单元测试、性能测试、文档，HTTP服务、调试
- 工具链
- 语法
  - 类型、语句、数据、函数方法集、接口、错误、包
- 并发 & 调度
- 垃圾回收
- 反射
- 测试
- 调试

适用于

- 熟悉服务端研发
- 有一定开发经验
- 对Golang有基础了解

示例

Hello

```go
package main

import "fmt"

func main() {
    fmt.Println("Hello, 世界")
}

```

Reset Format Run

```
Hello, 世界

Program exited.
```

https://tour.go-zh.org/welcome/1

格式

- 格式是语言规范一部分
  - 大括号
  - 分号
- 自动格式化工具
  - gofmt

```go
// Package user 实现了用户相关的逻辑
package user

import (
  "errors"
  "fmt"
)

//type UserID int

// ID 代表用户ID,
type ID int

// Name 返回id对应的名字
func (u ID) Name() (string, error) {
  // getter,  setter方法不需要get,  set前缀
  if u < 0 {
    return "", errors.New("invalid user id")
  }
  return fmt.Sprintf("[name-%d]", u), nil
}

// String 返回ID的说明
func (u ID) String() string {
  return fmt.Sprintf("userid:%d", u)
}
```

```go
package main

import (
  "fmt"
  "log"

  "haiziwang.com/godemo/user"
)

func logErr(v interface{}, msg string, err error) {
  log.Printf("ERROR: v=%v; msg=%s; err=%s \n", v, msg, err)
}

func printUserName(id user.ID) {
  if name, err := id.Name(); err != nil {
    logErr(id, "get user name", err)
  } else {
    fmt.Printf("user %d: %s \n", id, name)
  }
}

func main() {
  printUserName(-1)
  printUserName(101)
}
```

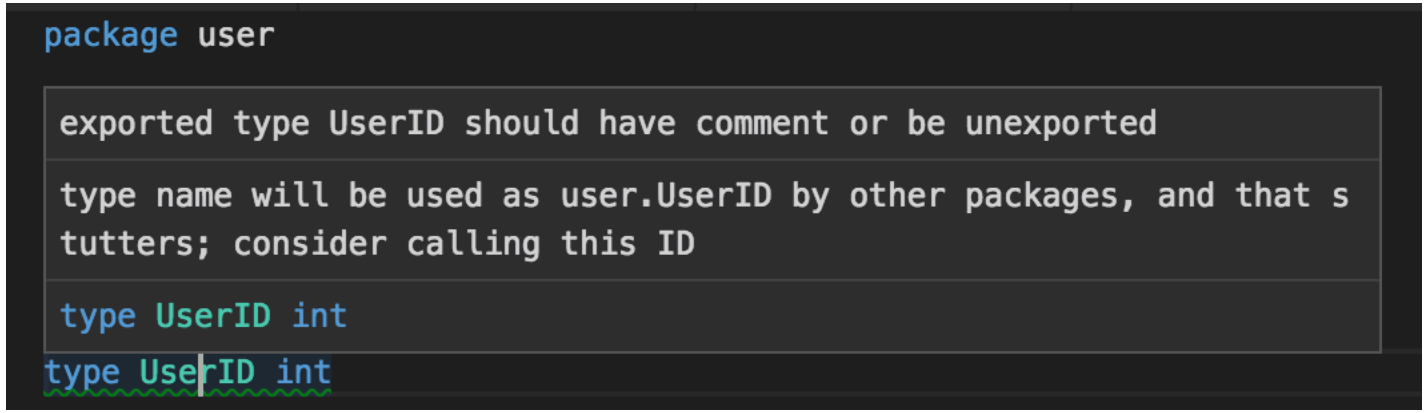
运行

```
GOPATH
|-- pkg
|-- bin
|-- src
    |-- haiziwang.com
        |-- godemo
            |-- demo
            |   |-- demo
            |   |-- main.go
            |-- user
                |-- user.go
                |-- user_test.go
```

```
➜  demo go build
➜  demo ./demo
2017/02/20 16:54:19 ERROR: v=userid:-1; msg=get user name; err=invalid user id
user 101: [name-101]
```

Golang风格

- 代码格式
  - gofmt
- 命名
- 注释
  - 注释规范
  - doc.go
- 文档
  - godoc生成文档，查看文档
  - https://godoc.org 查看文档



```go
import (
  "fmt"
  "testing"
)

func TestName(t *testing.T) {
  var id ID = -1
  if _, err := id.Name(); err == nil {
    t.Fail()
  }
}

func BenchmarkName(b *testing.B) {
  var id ID = 101
  for i := 0; i < b.N; i++ {
    name, err := id.Name()
    if err != nil {
      b.Fatalf("error: id=%d; name=%s; err=%v \n", id, name, err)
    }
  }
}

func ExampleID_Name() {
  var id ID = 101
  fmt.Println(id.Name())
  // Output: [name-101] <nil>
}
```

测试

```
➜ user go test -v
=== RUN   TestName
--- PASS: TestName (0.00s)
=== RUN   ExampleID_Name
--- PASS: ExampleID_Name (0.00s)
PASS
ok      haiziwang.com/godemo/user       0.005s
➜ user go test -bench .
BenchmarkName-8         10000000                195 ns/op
PASS
ok      haiziwang.com/godemo/user       2.170s
➜ user go test -coverprofile=coverage.out
PASS
coverage: 75.0% of statements
ok      haiziwang.com/godemo/user       0.007s
➜ user go tool cover -html=coverage.out
```

haiziwang.com/godemo/user/user.go (75.0%) not tracked not covered covered

```go
// Package user 实现了用户相关的逻辑
package user

import (
        "errors"
        "fmt"
)

//type UserID int

// ID 代表用户ID,
type ID int

// Name 返回id对应的名字
func (u ID) Name() (string, error) {
        // getter,  setter方法不需要get,  set前缀
        if u < 0 {
                return "", errors.New("invalid user id")
        }
        return fmt.Sprintf("[name-%d]", u), nil
}

// String 返回ID的说明
func (u ID) String() string {
        return fmt.Sprintf("userid:%d", u)
}
```

Package user

```
import "haiziwang.com/godemo/user"
```

Overview
Index
Examples

Overview

Package user 实现了用户相关的逻辑

Index

type ID
  func (u ID) Name() (string, error)
  func (u ID) String() string

Examples

ID.Name

Package files

user.go

http://localhost:6060/pkg/haiziwang.com/godemo/user/

```
➜  user godoc  haiziwang.com/godemo/user
use 'godoc cmd/haiziwang.com/godemo/user' for documentation on the haiziwang.com/godemo/user command

PACKAGE DOCUMENTATION

package user
    import "haiziwang.com/godemo/user"

    Package user 实现了用户相关的逻辑

TYPES

type ID int
    ID 代表用户ID,

func (u ID) Name() (string, error)
    Name 返回id对应的名字

func (u ID) String() string
    String 返回ID的说明
```

方法绑定

```go
type N int

func (n N) value() {
  fmt.Printf("value: p=%p, v=%v \n", &n, n)
}

func (n *N) pointer() {
  fmt.Printf("pointer: p=%p, v=%v \n", n, *n)
}

func receive() {
  var n N = 25
  p := &n
  fmt.Printf("v=%d, &p=%p \n", n, p)

  n.value()
  n.pointer()
  p.value()
  p.pointer()
  N.value(n)
  (*N).pointer(p)
}
```

```
v=25, &p=0xc42000a298
value: p=0xc42000a2e0, v=25
pointer: p=0xc42000a298, v=25
value: p=0xc42000a2f8, v=25
pointer: p=0xc42000a298, v=25
value: p=0xc42000a310, v=25
pointer: p=0xc42000a298, v=25
```

http服务

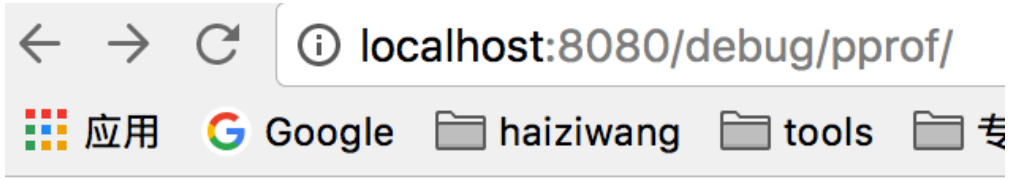
```
// http://localhost:8080/hello?name=go
http.HandleFunc("/hello", func(w http.ResponseWriter, r *http.Request) {
  fmt.Fprintf(w, "Hello, %s", r.FormValue("name"))
})

// http://localhost:8080/ping
http.HandleFunc("/ping", func(w http.ResponseWriter, r *http.Request) {
  w.Write([]byte("pong"))
})

log.Fatal(http.ListenAndServe(":8080", nil))
```

http pprof

```
import (
  "fmt"
  "log"
  "net/http"
  _ "net/http/pprof"
  "time"
)

var data = make(map[time.Time]string)
```

```
func c() {
  data[time.Now()] = fmt.Sprint(time.Now())
}

func b() {
  c()
}

func a() {
  b()
}

func runJob() {
  for {
    time.Sleep(10 * time.Millisecond)
    a()
  }
}
```

http pprof



/debug/pprof/

profiles:

0 block

5 goroutine

1 heap

7 threadcreate

full goroutine stack dump

```
# runtime.MemStats
# Alloc = 4049320
# TotalAlloc = 8249416
# Sys = 14113080
# Lookups = 15
# Mallocs = 75168
# Frees = 42790
# HeapAlloc = 4049320
# HeapSys = 6029312
# HeapIdle = 1409024
# HeapInuse = 4620288
# HeapReleased = 0
# HeapObjects = 32378
# Stack = 589824 / 589824
# MSpan = 54240 / 65536
# MCache = 9600 / 16384
# BuckHashSys = 1443798
# NextGC = 5843800
# PauseNs = [107055 132930 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
# NumGC = 2
# DebugGC = false
```

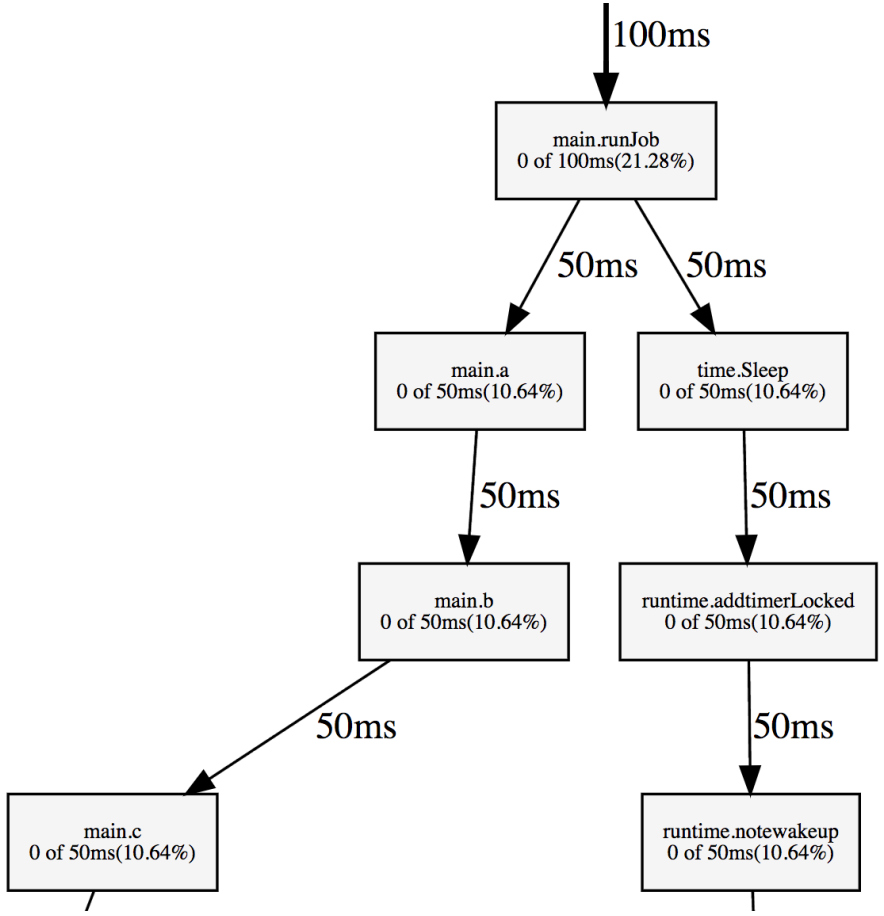
http://localhost:8080/debug/pprof/

go tool pprof

```
(pprof) top20
470ms of 470ms total (  100%)
Showing top 20 nodes out of 56 (cum >= 40ms)
      flat  flat%   sum%        cum   cum%
     150ms 31.91% 31.91%      150ms 31.91%  runtime.mach_semaphore_signal
     110ms 23.40% 55.32%      110ms 23.40%  runtime.mach_semaphore_wait
      70ms 14.89% 70.21%       70ms 14.89%  runtime.mach_semaphore_timedwait
      60ms 12.77% 82.98%       60ms 12.77%  runtime.usleep
      40ms  8.51% 91.49%       40ms  8.51%  runtime.memmove
      10ms  2.13% 93.62%       10ms  2.13%  runtime.duffcopy
      10ms  2.13% 95.74%       10ms  2.13%  runtime.getcallersp
      10ms  2.13% 97.87%       10ms  2.13%  runtime.heapBitsSetType
      10ms  2.13%   100%      190ms 40.43%  runtime.semasleep
         0     0%   100%       50ms 10.64%  main.a
         0     0%   100%       50ms 10.64%  main.b
         0     0%   100%       50ms 10.64%  main.c
         0     0%   100%      100ms 21.28%  main.runJob
         0     0%   100%       50ms 10.64%  runtime.addtimerLocked
```

go tool pprof http://localhost:8080/debug/pprof/profile

go tool pprof



(pprof) web runJob

特点

- 开源
- 跨平台，简单，开发效率高
- 静态链接，部署方便
- 支持cgo
- 垃圾回收
- 非侵入式接口
- 语言层面支持并发
- 实用的标准库
- 重视工程化，工具链完整
- 强制代码规范

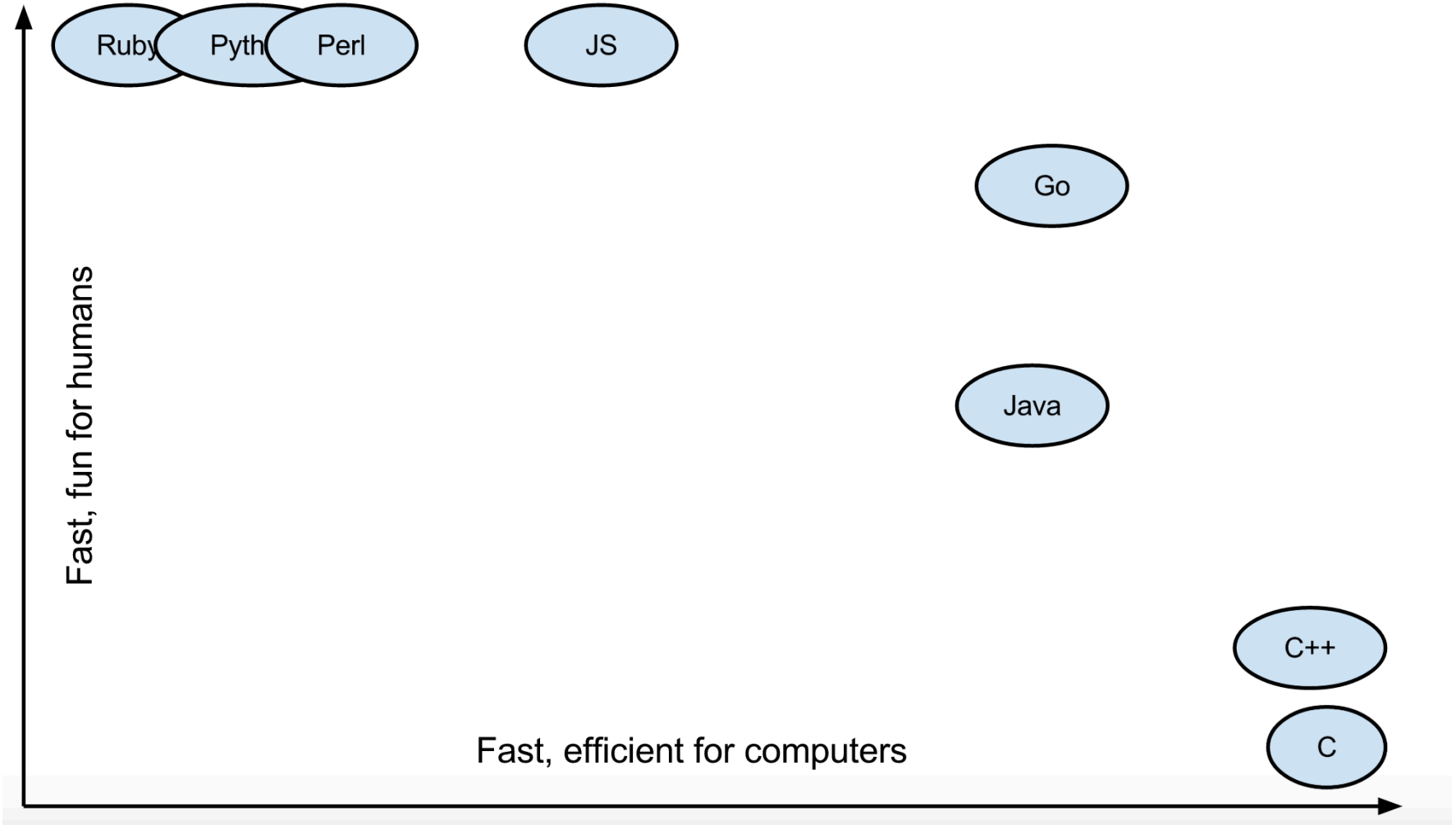
特点

- 没有类、继承、多态、重载，不是面向对象语言
- 没有构造、析构函数
- 没有注解
- 没有泛型
- 没有运算符重载
- 没有动态库
- 没有官方GUI库
- 没有类似字节码、Emit等动态代码生成方案

争论

- 错误处理
- 接口机制
- 包管理

定位


Ruby
Pyth
Perl
JS
Go
Java
C++
C
Fast, fun for humans
Fast, efficient for computers

编码规范

- https://github.com/golang/go/wiki/CodeReviewComments
- https://github.com/alecthomas/gometalinter

tips

- 值，指针，可寻址(addressable)
- goroutine，channel，select，sync
- interface，method set

文档

- https://tour.go-zh.org
- https://golang.org/doc/effective_go.html
- https://golang.org/ref/spec
- https://golang.org/pkg/runtime/#hdr-Environment_Variables

Tools

Go was designed with tools in mind. (Rob Pike)

tools

```
build       compile packages and dependencies
clean       remove object files
doc         show documentation for package or symbol
env         print Go environment information
fix         run go tool fix on packages
fmt         run gofmt on package sources
generate    generate Go files by processing source
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
```

https://golang.org/cmd/

go get

- $GPPATH
  - GOPATH/src/<import-path>
- 常用参数
  - -d
  - -u
  - -v
- go help get

go build

- -x 显示执行的命令
- -gcflags 编译器参数
- -ldflags 链接器参数
- -race

go build – 动态修改内容

```go
package main

import (
  "fmt"
)

var version = "1.0"

func main() {
  fmt.Println("version:", version)
}
```

```
➜  build go build -ldflags "-X main.version=1.0.1"
➜  build ./build
version: 1.0.1
```

go build – 条件编译

- 编译指令
  - // +build linux darwin
  - // +build 386
  - // +build !linux
  - // +build go1.5
- 文件名后缀
  - mypkg_linux.go
  - mypkg_windows_amd64.go
- 自定义tag
  - go build –tags "xxx”

go build – 交叉编译

```go
import (
  "fmt"
)

func main() {
  fmt.Println("hello")
}
```

```
➜  crosscompile go build
➜  crosscompile file crosscompile
crosscompile: Mach-O 64-bit executable x86_64
➜  crosscompile env GOOS=linux GOARCH=amd64 go build
➜  crosscompile file crosscompile
crosscompile: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
```

go doc

- go doc sync
- go doc sync.Once.Do

```
➜  godemo go doc sync
package sync // import "sync"

Package sync provides basic synchronization primitives such as mutual
exclusion locks. Other than the Once and WaitGroup types, most are intended
for use by low-level library routines. Higher-level synchronization is
better done via channels and communication.

Values containing the types defined in this package should not be copied.

type Cond struct { ... }
    func NewCond(l Locker) *Cond
type Locker interface { ... }
type Mutex struct { ... }
type Once struct { ... }
type Pool struct { ... }
type RWMutex struct { ... }
type WaitGroup struct { ... }
```

godoc

- godoc -http=:6060
- godoc sync
- godoc sync Once
- godoc -src sync Once
- godoc -q sync.Once

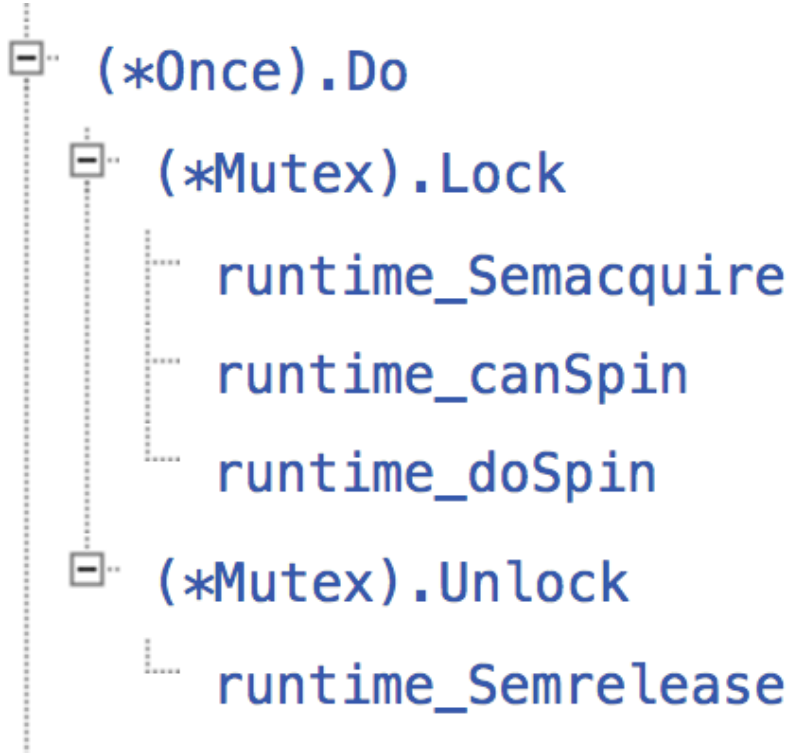
```
➜  godemo godoc -src sync Once
use 'godoc cmd/sync' for documentation on the sync command

// Once is an object that will perform exactly one action.
type Once struct {
    m    Mutex
    done uint32
}
```

```
//
func (o *Once) Do(f func()) {
        if atomic.LoadUint32(&o.done) == 1 {
                return
        }
        // Slow-path.
        o.m.Lock()
        defer o.m.Unlock()
        if o.done == 0 {
                defer atomic.StoreUint32(&o.done, 1)
                f()
        }
}
```

godoc -analysis=type,pointer

```
35  func (o *Once) Do(f func()) {
36          if atomic.LoadUint32(&o.done) == 1 {
37                  return
38          }
39          // Slow-path.
40          o.m.Lock()
41          defer o.m.Unlock()
42          if o.done == 0 {
43                  defer atomic.StoreUint32(&o.done, 1)
44                  f()
45          }
46  }
```

Callers of (*sync.Once).Do:
(*time.Location).get at line 76
syscall.Unsetenv at line 58
syscall.Getenv at line 72
syscall.Setenv at line 94
syscall.Environ at line 139
net.systemConf at line 41
(*net.resolverConfig).tryUpdate at line 275
(*net.pollDesc).init at line 34



https://golang.org/lib/godoc/analysis/help.html

godoc -play

godoc -play -http=:6060

gofmt

from

```
for{
fmt.Println(      "I feel pretty." );
       }
```

to

```
for {
    fmt.Println("I feel pretty.")
}
```

go vet

- go vet package/path/name
- go tool vet source/directory

go vet

```go
func doWithLock(l sync.Mutex, fn func()) {
  l.Lock()
  fn()
  l.Unlock()
}

func main() {
  var l sync.Mutex
  var value int

  for i := 0; i < 1000; i++ {
    go doWithLock(l, func() {
      value++
    })
  }
  time.Sleep(time.Second)
  fmt.Println("value=", value)
}
```

```
➜  vet go tool vet -all *.go
main.go:9: doWithLock passes lock by value: sync.Mutex
main.go:20: function call copies lock value: sync.Mutex
➜  vet
```

race detector

```go
m := map[string]int{}

go func() {
  m["race"] = 1
}()

go func() {
  m["race"] = 2
}()

<-time.After(time.Millisecond)
fmt.Println("m race:", m["race"])
```

```
==================
WARNING: DATA RACE
Write at 0x00c420016150 by goroutine 7:
  runtime.mapassign1()
      /usr/local/go/src/runtime/hashmap.go:442 +0x0
  main.readConcurrency.func2()
      /Users/songdianming/hzw/svn/front/golang/src/ha

Previous write at 0x00c420016150 by goroutine 6:
  runtime.mapassign1()
      /usr/local/go/src/runtime/hashmap.go:442 +0x0
  main.readConcurrency.func1()
      /Users/songdianming/hzw/svn/front/golang/src/ha
```

https://golang.org/doc/articles/race_detector.html

语法

关键词

```
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
```

代码格式

- 代码格式是规范一部分
  - 大括号位置
  - 自动插入分号
- 官方格式化工具
  - gofmt

变量

```
package main

import "fmt"

func main() {
    var i, j int = 1, 2
    k := 3
    c, python, java := true, false, "no!"

    fmt.Println(i, j, k, c, python, java)
}
```

变量

- 变量会自动初始化为二进制零值(zero value)
  - 数值：0 ；布尔：false；字符串： ""
- 支持类型推导
- :=简洁声明方式只能用在函数内
- 支持多变量赋值
- 常量不能使用 := 方式定义
- 常量不可寻址
- "_"代表支持空标识符

变量 - 坑

- 简短模式并不总是重新定义变量
  - 最少有一个新变量
- 定义但未使用的变量
- shadow

```
x := 1
fmt.Println(&x)
x, y := 2, "y"
fmt.Println(&x, x, y)
/*
  0xc42000a298
  0xc42000a298 2 y
*/
```

变量 - shadow

```go
func do() (int, error) {
  return 100, nil
}

func main() {
  i := 10
  if i, err := do(); err == nil {
    fmt.Println("done:", i)
  }
  fmt.Println("i:", i)
}
```

```
➜  var go tool vet -shadow main.go

main.go:11: declaration of "i" shadows declaration at main.go:10
```

自增 - iota

```
// A Weekday specifies a day of the week (Sunday = 0, ...).
type Weekday int

const (
        Sunday Weekday = iota
        Monday
        Tuesday
        Wednesday
        Thursday
        Friday
        Saturday
)

var days = [...]string{
        "Sunday",
        "Monday",
        "Tuesday",
        "Wednesday",
        "Thursday",
        "Friday",
        "Saturday",
}

// String returns the English name of the day ("Sunday", "Monday", ...).
func (d Weekday) String() string { return days[d] }
```

https://golang.org/src/time/time.go

类型 – 基础类型

- bool
- string
- int int8 int16 int32 int64
- uint uint8 uint16 uint32 uint64
- uintptr
- byte // uint8 的别名
- rune // int32 的别名，代表一个Unicode码
- float32 float64
- complex64 complex128

类型 – 基础类型

- array
- slice(引用类型，make)
- struct
- function
- interface
- map (引用类型，make)
- channel (引用类型，make)

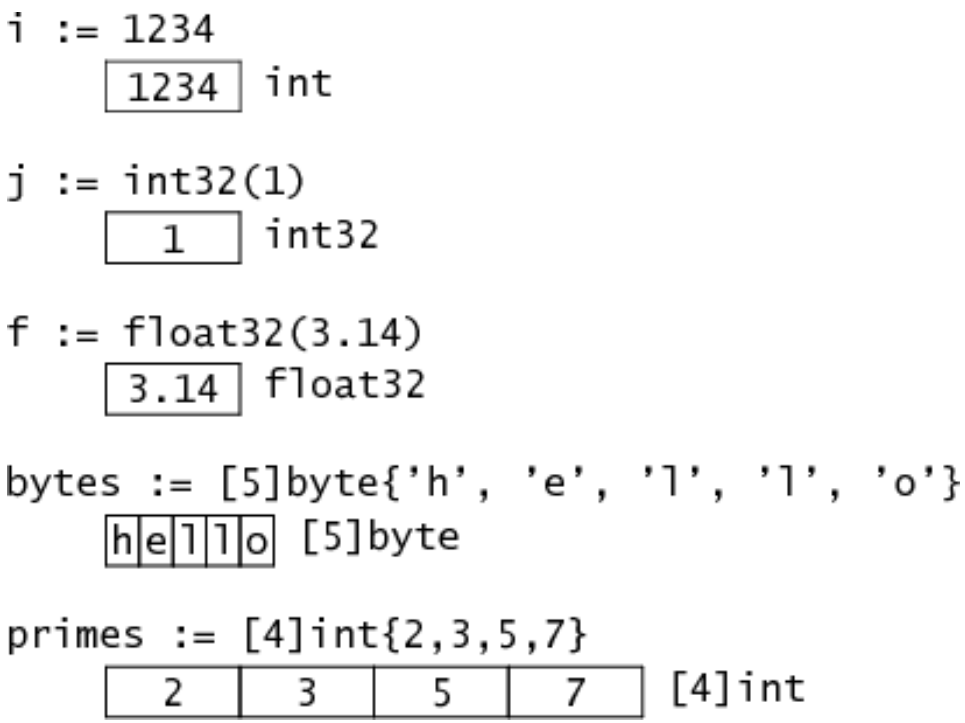
自定义类型

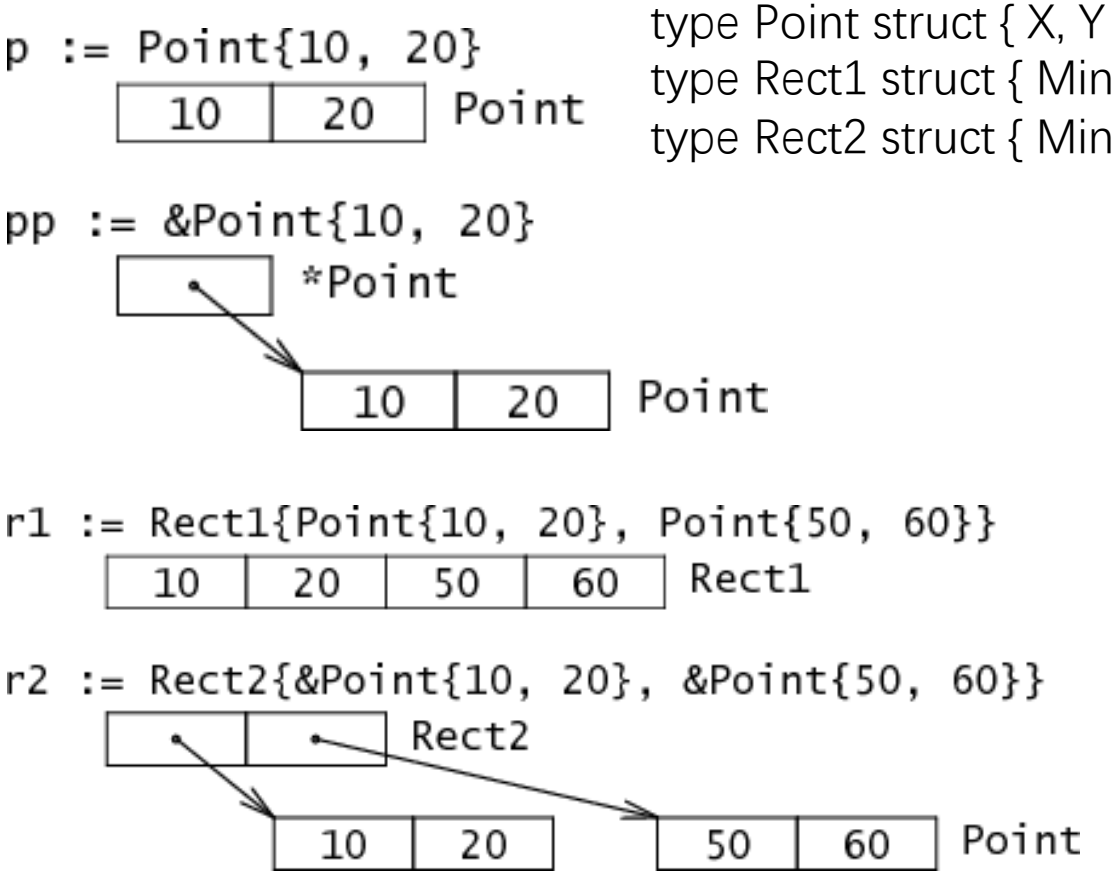
- 自定义
  - 不继承基础类型信息
  - 不是别名
  - 不能隐式转换

```
// A Weekday specifies a day of the week (Sunday = 0, ...).
type Weekday int
```

- 未命名类型
  - 相同元素类型的指针、数组、slice、map、channel
  - struct：相同的字段名、字段类型、顺序、标签
  - 函数：相同的参数和返回值（不包含参数名）
  - 接口：相同方法集（不包含顺序）

Go Data Structures





type Point struct { X, Y int }
type Rect1 struct { Min, Max Point }
type Rect2 struct { Min, Max *Point }

https://research.swtch.com/godata

类型 – 转换

- 强制显式转换
- T(v) 将值 v 转换为类型 T
- strconv库处理字符串转换

```
var i int = 42
var f float64 = float64(i)
var u uint = uint(f)
```

指针

```
func main() {
    i, j := 42, 2701

    p := &i          // point to i
    fmt.Println(*p) // read i through the pointer
    *p = 21          // set i through the pointer
    fmt.Println(i)  // see the new value of i

    p = &j          // point to j
    *p = *p / 37    // divide j through the pointer
    fmt.Println(j) // see the new value of j
}
```

字符串

- 只读的byte序列
  - 每次复制要分配新内存
  - 动态拼接字符串的性能风险
- 默认以utf8编码存储Unicode字符
- len()返回byte数量，不是字符数量
- rune存储Unicode码点 (code point)

```
type stringStruct struct {
        str unsafe.Pointer
        len int
}
```

https://golang.org/src/runtime/string.go

string – byte

```go
func print(s string) {
  fmt.Println("print:", s)
  fmt.Printf("% x\n", s)
  for i := 0; i < len(s); i++ {
    fmt.Printf("%x ", s[i])
  }
  fmt.Printf("\n%+q\n", s)
}
```

```
print: hello
68 65 6c 6c 6f
68 65 6c 6c 6f
"hello"
print: 语言
e8 af ad e8 a8 80
e8 af ad e8 a8 80
"\u8bed\u8a00"
```

string - rune

```go
func rangLoop(s string) {
  fmt.Println("rang loop:", s)
  for index, runeValue := range s {
    fmt.Printf("%#U position %d\n", runeValue, index)
  }
}
```

```
rang loop: 语言
U+8BED '语' position 0
U+8A00 '言' position 3
```

string - unicode/utf8

```go
func decode(s string) {
  b := []byte(s)
  fmt.Println("utf8 decode:", s, "bytes=", len(b), "runes=", utf8.RuneCount(b))
  for len(b) > 0 {
    r, size := utf8.DecodeRune(b)
    fmt.Printf("%c %v\n", r, size)
    b = b[size:]
  }
}
```

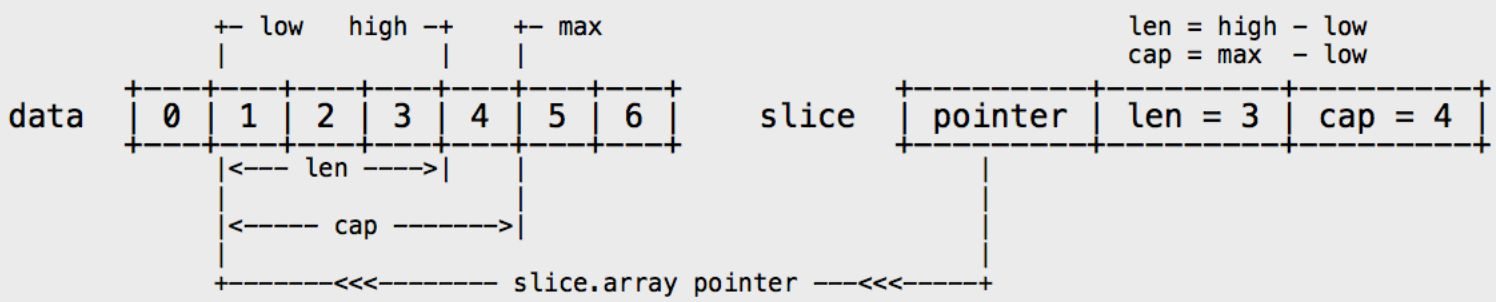
```
utf8 decode: go语言 bytes= 8 runes= 4
g 1
o 1
语 3
言 3
```

array

- 长度是数组类型一部分
  - 不能改变大小
  - 元素类型相同长度不同，不是同一类型
- 数组是值类型
  - 数组变量不是指向第一个元素的指针
  - 赋值、传参都会复制整个数组数据
- 如果元素支持==、!=，数组也支持

slice(切片)

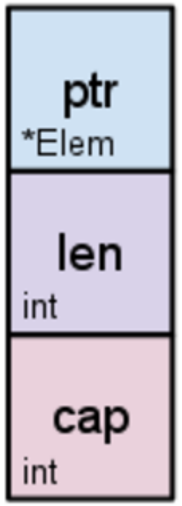
```
data := [...]int{0, 1, 2, 3, 4, 5, 6}
slice := data[1:4:5]                          // [low : high : max]
```



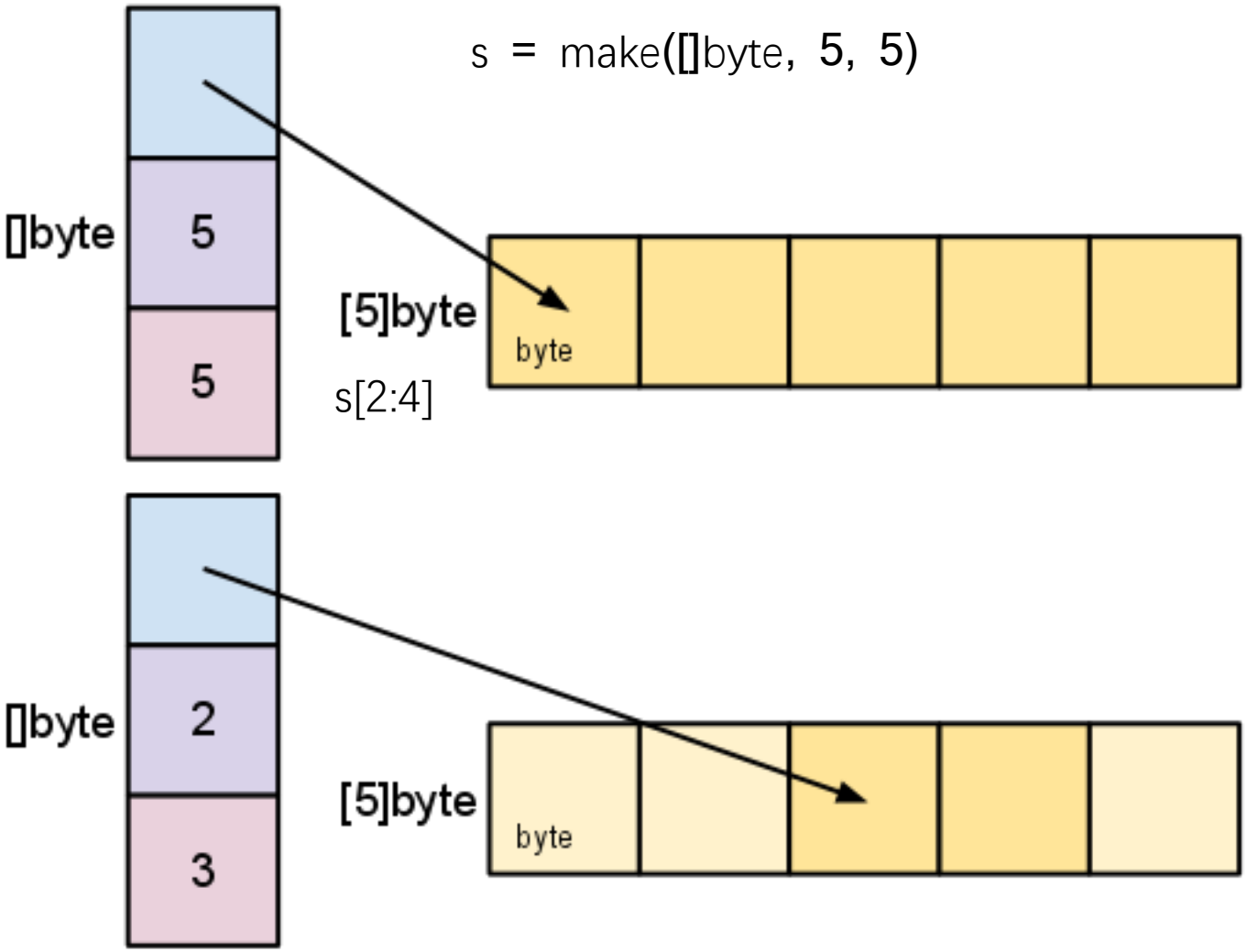

```
data := [...]int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

expression   slice                  len    cap     comment
-----------+--------------------+------+-------+--------------------
data[:6:8]   [0 1 2 3 4 5]          6      8       省略 low.
data[5:]     [5 6 7 8 9]            5      5       省略 high、max。
data[:3]     [0 1 2]                3      10      省略 low、max。
data[:]      [0 1 2 3 4 5 6 7 8 9]  10     10      全部省略。
```

slice

```
type slice struct {
        array unsafe.Pointer
        len   int
        cap   int
}
```



https://golang.org/src/runtime/slice.go



slice

- 自身是结构体，值拷贝传递
- 需要初始化才分配内存 (make、初始化表达式)
- 属性
  - len 表示可用元素数量，读写操作不能超过该限制
  - cap 表示最大扩张容量
  - nil 的 slice 的长度和容量是 0
- 半开区间
- 不支持比较操作

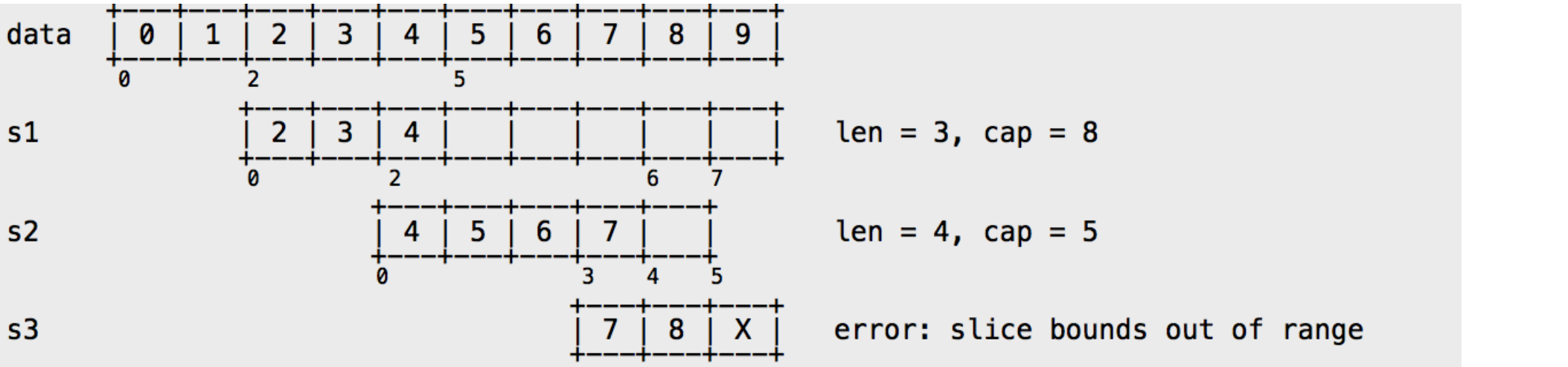
slice

- reslice
  - 不能超出cap，但不受len限制
  - 指向相同底层数组
- append
  - 返回新slice对象
  - 如超出cap限制则重新分配数组

```go
s1 := make([]int, 1, 5)
s2 := append(s1, 1)

fmt.Println(s1, len(s1), cap(s1)) // [0] 1 5
fmt.Println(s2, len(s2), cap(s2)) // [0 1] 2 5
```

reslice

```
s := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

s1 := s[2:5]          // [2 3 4]
s2 := s1[2:6:7]       // [4 5 6 7]
s3 := s2[3:6]         // Error
```



《Go语言学习笔记》

append 扩容

- 分配内存
- 复制数据
- 不是链表

```go
s := []int{0}
fmt.Println("0: cap=", cap(s), "ptr(s)=", &s[0])

s = append(s, 1)
fmt.Println("1: cap=", cap(s), "ptr(s)=", &s[0])

s = append(s, 2)
fmt.Println("2: cap=", cap(s), "ptr(s)=", &s[0])

x := append(s, 3)
fmt.Println("3: cap=", cap(s), "ptr(s)=", &s[0], "ptr(x) =", &x[0])

y := append(s, 4)
fmt.Println("4: cap=", cap(s), "ptr(s)=", &s[0], "ptr(y) =", &y[0])
```

```
0: cap= 1 ptr(s)= 0xc42000a298
1: cap= 2 ptr(s)= 0xc42000a300
2: cap= 4 ptr(s)= 0xc420012280
3: cap= 4 ptr(s)= 0xc420012280 ptr(x) = 0xc420012280
4: cap= 4 ptr(s)= 0xc420012280 ptr(y) = 0xc420012280
```

map

- 访问不存在的key不会报错
  - 返回零值
  - 使用ok-idiom模式
- 迭代顺序是随机的
- 并发操作会crash
- map元素是无法取址的

```
type data struct {
  name string
}

func updateMap() {
  m := map[string]data{"x": {"a"}}
  m["x"].name = "b"
}
```

len & cap

Call	Argument type	Result
len(s)	string type	string length in bytes
	[n]T, *[n]T	array length (== n)
	[]T	slice length
	map[K]T	map length (number of defined keys)
	chan T	number of elements queued in channel buffer
cap(s)	[n]T, *[n]T	array length (== n)
	[]T	slice capacity
	chan T	channel buffer capacity

0 <= len(s) <= cap(s)

内置函数

	len	cap	close	delete	make
string	Yes				
array (and array pointer)	Yes	Yes			
slice	Yes	Yes			Yes
map	Yes			Yes	Yes
channel	Yes	Yes	Yes		Yes

struct

```
type Vertex struct {
    X, Y int
}

var (
    v1 = Vertex{1, 2}  // 类型为 Vertex
    v2 = Vertex{X: 1}  // Y:0 被省略
    v3 = Vertex{}      // X:0 和 Y:0
    p  = &Vertex{1, 2} // 类型为 *Vertex
)

func main() {
    fmt.Println(v1, p, v2, v3)
}
```

struct

- 支持匿名字段
  - 以类型名作为字段名
  - 可以直接引用匿名字段的成员
  - 不是继承
- 支持空结构
  - 没有字段
  - 不分配内存
  - 作为数组元素，数组不分配内存

```
type Seq struct {
  sync.Mutex
  Value int
}

func incr(seq *Seq) {
  seq.Lock()
  defer seq.Unlock()

  seq.Value++
}
```

类型初始化

- 初始化表达式
- new
- make
  - slice
  - map
  - channel
- 初始化函数
  - 按惯例名字为NewXXX()

```go
type person struct {
  Name string
}

func newPerson(name string) person {
  return person{
    Name: name,
  }
}

func declare() {
  var a person
  b := person{}
  c := &person{}
  d := new(person)
  e := newPerson("e")

  fmt.Println(a, b, c, d, e)
}
```

struct赋值

```
var a struct {
  Name string
}
a.Name = "a"
var p person
p = a
p.Name = "p"

fmt.Printf("a=%#v, &a=%p \n", a, &a)
fmt.Printf("p=%#v, &p=%p \n", p, &p)
```

```
a=struct { Name string }{Name:"a"}, &a=0xc420076030
p=main.person{Name:"p"}, &p=0xc420076040
```

标签

- 标签是类型一部分

```
type StringTag struct {
        BoolStr bool   `json:",string"`
        IntStr  int64  `json:",string"`
        StrStr  string `json:",string"`
}
```

运算符

- ++\--不是运算符
  - 能用于语句(statement)
  - 不能用于表达式(expression)
  - var j = i++ //错误
- 没有条件运算符
  - condition ? x : y

```
55      a := 1
56      j := a++
57      fmt.Println("a++", a++)
58      p:= &a
59      *p++
60      fmt.Println(a)
```

```
./main.go:56: syntax error: unexpected ++ at end of statement
./main.go:57: syntax error: unexpected ++, expecting comma or )
```

if...else

```
func pow(x, n, lim float64) float64 {
    if v := math.Pow(x, n); v < lim {
        return v
    }
    return lim
}
```

switch

- 表达式switch
- type switch

```
func main() {
    fmt.Print("Go runs on ")
    switch os := runtime.GOOS; os {
    case "darwin":
        fmt.Println("OS X.")
    case "linux":
        fmt.Println("Linux.")
    default:
        fmt.Printf("%s.", os)
    }
}
```

switch

- 条件从上到下执行
  - 匹配成功的时候停止
- 分支会自动终止
  - 除非以 fallthrough 语句结束
- 支持动态分支条件
  - 代替 if-then-else 链

```
func main() {
    t := time.Now()
    switch {
    case t.Hour() < 12:
        fmt.Println("Good morning!")
    case t.Hour() < 17:
        fmt.Println("Good afternoon.")
    default:
        fmt.Println("Good evening.")
    }
}
```

for

- 只有一种循环结构
- 初始化语句只执行一次
- for...range
  - rang是值复制

range

Range expression			1st value			2nd value	
array or slice	a	[n]E, *[n]E, or []E	index	i	int	a[i]	E
string	s	string type	index	i	int	see below	rune
map	m	map[K]V	key	k	K	m[k]	V
channel	c	chan E, <-chan E	element	e	E		

range – 值复制

```go
var a = [5]int{0, 1, 2, 3, 4}
for i, v := range a {
  if i == 0 {
    a[0] = 10
    a[1] = 11
    a[2] = 12
    a[3] = 13
    a[4] = 14
  }
  fmt.Println(i, "=", v)
}
fmt.Println("a=", a)
```

```
➜  range go run main.go
0 = 0
1 = 1
2 = 2
3 = 3
4 = 4
a= [10 11 12 13 14]
```

```go
var a = [5]int{0, 1, 2, 3, 4}
for i, v := range a[:] {
  if i == 0 {
    a[0] = 10
    a[1] = 11
    a[2] = 12
    a[3] = 13
    a[4] = 14
  }
  fmt.Println(i, "=", v)
}
fmt.Println("a=", a)
```

```
➜  range go run main.go
0 = 0
1 = 11
2 = 12
3 = 13
4 = 14
a= [10 11 12 13 14]
```

函数

```
package main

import "fmt"

func swap(x, y string) (string, string) {
    return y, x
}

func main() {
    a, b := swap("hello", "world")
    fmt.Println(a, b)
}
```

函数

- 支持多返回值，返回值可命名
- 支持变长参数
- 支持匿名函数
- 支持闭包
- 支持内部延迟调用(defer)
- 不支持重载
- 不支持默认参数

函数

- 函数是第一类对象
  - 可以作为参数或返回值
  - 相同签名为同一类型
- 支持闭包
  - 同样有其他语言闭包的坑
  - 可以安全返回局部变量地址
- 逃逸分析
  - 编译器决定在堆还是栈分配内存

```
func adder() func(int) int {
    sum := 0
    return func(x int) int {
        sum += x
        return sum
    }
}

func main() {
    pos, neg := adder(), adder()
    for i := 0; i < 10; i++ {
        fmt.Println(
            pos(i),
            neg(-2*i),
        )
    }
}
```

函数 – 参数

- 参数总是传值 (值copy)
  - 容易有坑
  - 没有ref、out之类的概念
  - 要分析应该copy值还是copy指针

函数 – 可变参数

- 变参是slice
- slice可以展开为变参

```go
func print(a ...interface{}) {
  printLen(a)
  for _, i := range a {
    fmt.Print(" ", i)
  }
  fmt.Println("")
}

func printLen(a []interface{}) {
  fmt.Print("total:", len(a))
}

func main() {
  values := []interface{}{1, 2, 3, 4, 5}
  print(1, 2, 3, 4, 5)
  print(values)
  print(values...)
}
```

函数 – 规范

- 返回值
  - 如果返回error，error作为最后一个返回值
  - 返回值是否命名
    - 命名会自动生成一个局部变量
    - 尽量不命名
- 函数名
  - MixedCaps或者mixedCaps格式，不要下划线
  - 简单，清晰
- 注释
  - 符合golang风格：函数名 xxxx

defer

```go
package main

import "fmt"

func main() {
    defer fmt.Println("world")

    fmt.Println("hello")
}

```

Reset Format Run

```
hello
world
```

defer - 坑

- defer函数的参数会在defer声明时求值
- defer在函数结束时才执行
- 后进先出
- 性能损耗

```go
func a() {
  i := 0
  defer fmt.Println(i)
  i++
  return
}
```

```go
func b() {
  for i := 0; i < 4; i++ {
    defer fmt.Println(i)
  }
}
```

```go
func c() (i int) {
  defer func() { i++ }()
  return 1
}
```

defer - 坑

```go
func f() (result int) {
    defer func() {
        result++
    }()
    return 0
}
```

```
➜  defer go run main.go
1
```

```go
func f() (result int) {
    // return xxx 是赋值+ret指令
    result = 0

    //defer被插入到return之前执行，也就是赋返回值和ret指令之间
    func() {
        result++
    }()
    return
}
```

先给返回值赋值，然后调用defer表达式，最后返回

初始化函数

- 不保证执行顺序
  - 先初始化全局变量
  - 初始化函数之间不应该有依赖关系
- 只执行一次
- 初始化函数不能直接调用
- 所有init函数执行后才执行main函数

方法

- 与实例绑定的特殊函数
  - 接受参数(receiver)
- 编译器会处理值和指针之间的切换
- nil同样可以执行

方法

```go
type N int

func (n N) value() {
  fmt.Printf("value: p=%p, v=%v \n", &n, n)
}

func (n *N) pointer() {
  fmt.Printf("pointer: p=%p, v=%v \n", n, *n)
}

func receive() {
  var n N = 25
  p := &n

  n.value()
  n.pointer()
  p.value()
  p.pointer()
  N.value(n)
  (*N).pointer(p)
}
```

```
value: p=0xc42000a2c0, v=25
pointer: p=0xc42000a298, v=25
value: p=0xc42000a2e8, v=25
pointer: p=0xc42000a298, v=25
value: p=0xc42000a300, v=25
pointer: p=0xc42000a298, v=25
```

receiver 值 &指针

- 对象大小
- 是否修改状态
- 复制是否对逻辑有影响

匿名字段

```go
type animal struct{}

func (animal) hi() {
  fmt.Println("hi")
}

func (animal) name() string {
  return "animal"
}

type dog struct {
  animal
}

func (dog) name() string {
  return "dog"
}

func override() {
  d := dog{}

  d.hi()                          // hi
  fmt.Println("I am:", d.name()) // I am: dog
}
```

method set (方法集)

- 类型 T 方法集包含全部 receiver T 方法
- 类型 *T 方法集包含全部 receiver T + *T 方法
- 如类型 S 包含匿名字段 T，则 S 方法集包含 T 方法
- 如类型 S 包含匿名字段 *T，则 S 方法集包含 T + *T 方法
- 不管嵌入 T 或 *T，*S 方法集总是包含 T + *T 方法

method value

- 立即计算该方法执行所需的receiver对象
  - 为了执行时能传入receiver参数
- 作为参数时复制receiver在内的整个method value

method value

```go
type N int

func (n N) test() {
  fmt.Printf("test.n: p=%p, v=%v \n", &n, n)
}

func methodValue() {
  n := N(25)
  p := &n

  n++
  f1 := n.test

  n++
  f2 := p.test

  n++
  fmt.Printf("main.n: p=%p, v=%v \n", p, n)
  f1()
  f2()
}
```

```
main.n: p=0xc42000a298, v=28
test.n: p=0xc42000a2c8, v=26
test.n: p=0xc42000a2e8, v=27
```

方法规范

- getter函数不需要带get前缀
- setter函数不需要带set前缀
- 名称简短，清晰，无歧义
- 如果不确定用值还是用指针，优先用指针
- 修改状态时，不要用值作为receiver

接口

```
type Stringer interface {
        String() string
}
```

https://golang.org/src/fmt/print.go

```
var days = [...]string{
        "Sunday",
        "Monday",
        "Tuesday",
        "Wednesday",
        "Thursday",
        "Friday",
        "Saturday",
}

// String returns the English name of the day ("Sunday", "Monday", ...).
func (d Weekday) String() string { return days[d] }
```

https://golang.org/src/time/time.go

接口

- 非侵入设计
  - 可以先开发后抽象
- 接口组合
  - 嵌入接口相当于将方法集导入
  - 超集接口可以隐式转换为子集接口
- 数据
  - 对象赋值给接口变量时会复制该对象
- 空接口interface{}

```
type N struct {
  x int
}

func main() {
  n := N{x: 100}
  var i interface{} = n
  fmt.Println("i:", i, "n:", n)
  // i: {100} n: {100}

  n.x++
  fmt.Println("i:", i, "n:", n)
  // i: {100} n: {101}

  // i.(N).x = 101 //cannot assign to i.(N).x
}
```

接口组合

```
type Reader interface {
  Read(p []byte) (n int, err error)
}

type Writer interface {
  Write(p []byte) (n int, err error)
}

type Closer interface {
  Close() error
}

type ReadWriter interface {
  Reader
  Writer
}

type ReadWriteCloser interface {
  Reader
  Writer
  Closer
}
```

type Closer
type ReadCloser
type ReadSeeker
type ReadWriteCloser
type ReadWriteSeeker
type ReadWriter
type Reader
type ReaderAt
type ReaderFrom
type Seeker
type WriteCloser
type WriteSeeker
type Writer
type WriterAt
type WriterTo

接口 – 结构

```
type iface struct {
        tab  *itab
        data unsafe.Pointer
}
```

```
type itab struct {
        inter  *interfacetype
        _type  *_type
        link   *itab
        bad    int32
        inhash int32      // has this itab been added to hash?
        fun    [1]uintptr // variable sized
}
```

https://golang.org/src/runtime/runtime2.go

接口 - nil

```
var n *N
var i interface{}

fmt.Println(n, n == nil) // <nil> true
fmt.Println(i, i == nil) // <nil> true

i = n
// 数据(data)和类型(tab)都为nil时接口才是nil
fmt.Println(i, i == nil) // <nil> false
```

类型转换

```go
type N struct {
  x int
}

func (n N) String() string {
  return fmt.Sprintf("N:x=%d", n.x)
}
```

```go
n := N{100}
var i interface{} = n

// 转化为接口
if x, ok := i.(fmt.Stringer); ok {
  fmt.Println("fmt.Stringer", x.String())
  // fmt.Stringer N:x=100
}

// 转换为原始类型
if x, ok := i.(N); ok {
  fmt.Println("N:x", x.x)
  // N:x 100
}
```

类型判断(Type switch)

```
// Some types can be done without reflection.
switch f := arg.(type) {
case bool:
        p.fmtBool(f, verb)
case float32:
        p.fmtFloat(float64(f), 32, verb)
case float64:
        p.fmtFloat(f, 64, verb)
case complex64:
        p.fmtComplex(complex128(f), 64, verb)
case complex128:
        p.fmtComplex(f, 128, verb)
case int:
        p.fmtInteger(uint64(f), signed, verb)
```

https://golang.org/src/fmt/print.go

函数实现接口

```
type Handler interface {
        ServeHTTP(ResponseWriter, *Request)
}
```

```
// The HandlerFunc type is an adapter to allow the use of
// ordinary functions as HTTP handlers. If f is a function
// with the appropriate signature, HandlerFunc(f) is a
// Handler that calls f.
type HandlerFunc func(ResponseWriter, *Request)

// ServeHTTP calls f(w, r).
func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
        f(w, r)
}
```

接口规范

- 单函数接口，用函数名加er后缀命名
- 优先用接口组合
- 面向抽象设计

error

```
type error interface {
    Error() string
}

if f, err = os.Open("filename.ext"); err != nil{
    log.Fatal(err)
}
```

error – 争论

Most used words in go ▾ files

0	err	17,669,596
1	if	15,885,829
2	return	14,278,450
3	nil	13,254,150
4	the	8,535,644
5	string	7,762,689
6	s	6,615,007
7	func	6,459,627
8	c	5,649,769
9	t	5,088,720
10	v	4,640,586

Most used words in go ▾ files

all → err - 17,669,596

return err	1,518,678
if err != nil {	3,731,570
return nil, err	870,029
t.Fatal(err)	320,011
panic(err)	172,161
if err == nil {	169,101
var err error	136,062
c.Assert(err, jc.ErrorIsNil)	135,421
c.Assert(err, IsNil)	104,368
return "", err	83,980

https://anvaka.github.io/common-words/#?lang=go

error – 争论

- 倒退
- exception
  - 性能
  - 抛出和处理分在不同地方
  - 用好try-catch的难度
  - 异常安全，事务回滚

error – 自定义错误类型

```go
type appError struct {
  Message string
  Code    int
}

func (e *appError) Error() string {
  return fmt.Sprintf("app error: code: %d; message: %s.", e.Code, e.Message)
}

func parseInt(s string) (int64, error) {
  if s == "-1" {
    return 0, &appError{Code: 101, Message: "-1 is invalid"}
  }
  return strconv.ParseInt(s, 10, 64)
}

func main() {
  if i, err := parseInt("-1"); err != nil {
    fmt.Println("err:", err)
    if appErr, ok := err.(*appError); ok {
      fmt.Println("app err code:", appErr.Code)
    }
  } else {
    fmt.Println("parseInt:", i)
  }
}
```

error – 原则

- 异常(panic)和错误分开
- 检查所有返回的error
- error尽可能包含更多的信息
- 通过error变量而不是文本内容来判断错误类型
- error变量名以err为前缀，自定义错误类型名以Error为后缀
- 错误内容全小写，没有结束标点

error - 坑

```go
func formatInt(i int) (string, error) {
  var err *appError
  if i < 0 {
    return "", &appError{Code: 201, Message: "i less then 0"}
  }
  return strconv.Itoa(i), err
}

func main() {
  if s, err := formatInt(1); err != nil {
    fmt.Println("err:", err)
  } else {
    fmt.Println("formatInt:", s)
  }
}
```

值和类型都没有值，才是nil

panic && recover

- 不要用panic来代替正常的错误处理流程
- recover要在defer里调用才有效

```
func do() {
  defer func() {
    if r := recover(); r != nil {
      fmt.Println("recover:", r)
    }
  }()
  fmt.Println("start")
  panic("some thing wrong")
  fmt.Println("done")
}
```

stack

```go
func stack() {
  defer func() {
    err := recover()
    if err != nil {
      var buf bytes.Buffer
      fmt.Fprintln(&buf, "error:", err)
      fmt.Fprintln(&buf, "stack")
      buf.Write(debug.Stack())
      fmt.Println(buf.String())
    }
  }()

  panic("some thing wrong")
}
```

```
error: some thing wrong
stack
goroutine 1 [running]:
runtime/debug.Stack(0x100100, 0xc42004e070, 0xc42003ddc8)
        /usr/local/go/src/runtime/debug/stack.go:24 +0x79
main.stack.func1()
        /Users/songdianming/projects/defer/main.go:49 +0x21f
panic(0x90480, 0xc42000a2d0)
        /usr/local/go/src/runtime/panic.go:458 +0x243
main.stack()
        /Users/songdianming/projects/defer/main.go:54 +0x8d
main.main()
        /Users/songdianming/projects/defer/main.go:28 +0x14
```

workspace

- 目录结构
  - src
  - bin
  - pkg

import

- import "xxx/xxx”
- import 别名 "xxx/xxx”
- import . "xxx/xxx”
- import _ "xxx/xxx”

目录组织

- import导入的是路径
  - 包名一般与导入路径最后一级目录名一致
  - 包名简短，小写，单个词，不要下划线
  - 报名用单数形式
- internal
- vendor
  - 版本
- 团队成员目录结构一致

可访问性

- 包内
  - 所有成员均可访问
- 包外
  - 首字母大写（可导出）

测试

测试

```go
func repeat(s string, count int) string {
  var result string
  for i := 0; i < count; i++ {
    result = result + s
  }
  return result
}
```

```go
func TestRepeat(t *testing.T) {
  s := "abc"
  count := 3
  expect := "abcabcabc"
  actual := repeat(s, count)
  if actual != expect {
    t.FailNow()
  }
}
```

```
➜  test go test -v
=== RUN   TestRepeat
--- PASS: TestRepeat (0.00s)
PASS
ok      haiziwang.com/godemo/test       0.006s
```

测试

- 文件
  - xxx_test.go
- 用例
  - func TestXxx(*testing.T)
  - func BenchmarkXxx(*testing.B)
  - func ExampleXxx()
- table driver
- cover
- sub-test

testing.T

- Skip、SkipNow、Skipf、Skipped
- Fail、FailNow、Failed
- Parallel、Run
- Error、Errorf、Fatal、Fatalf、Log、Logf

Example

- 包
  - func Example() { ... }
- 函数
  - func ExampleF() { ... }
- 类型
  - func ExampleT() { ... }
- 类型的方法
  - func ExampleT_M() { ... }

```go
func ExampleRepeat() {
    fmt.Println(repeat("Abc", 3))

    // Output:
    // AbcAbcAbc
}
```

```
➜  test go test -v
=== RUN   TestRepeat
--- PASS: TestRepeat (0.00s)
=== RUN   ExampleRepeat
--- PASS: ExampleRepeat (0.00s)
PASS
ok      haiziwang.com/godemo/test       0.007s
```

table driver

```
var RepeatTests = []struct {
        in, out string
        count   int
}{
        {"", "", 0},
        {"", "", 1},
        {"", "", 2},
        {"-", "", 0},
        {"-", "-", 1},
        {"-", "----------", 10},
        {"abc ", "abc abc abc ", 3},
}

func TestRepeat(t *testing.T) {
        for _, tt := range RepeatTests {
                a := Repeat(tt.in, tt.count)
                if !equal("Repeat(s)", a, tt.out, t) {
                        t.Errorf("Repeat(%v, %d) = %v; want %v", tt.in, tt.count, a, tt.out)
                        continue
                }
        }
}
```

https://golang.org/src/strings/strings_test.go

性能

```go
func BenchmarkRepeat(b *testing.B) {
  s := "abc"
  count := 3
  for i := 0; i < b.N; i++ {
    _ = repeat(s, count)
  }
}
```

```
➜  test go test -run=NONE -bench .
BenchmarkRepeat-8       20000000               111 ns/op
PASS
ok      haiziwang.com/godemo/test       2.351s
➜  test go test -run=NONE -bench . -benchmem
BenchmarkRepeat-8       20000000               102 ns/op              16 B/op          2 allocs/op
PASS
ok      haiziwang.com/godemo/test       2.173s
```

性能

- -bench xxx
- -run=NONE
- -cpu
- -benchtime
- -benchmem
- -memprofile
- -cpuprofile

timer

- StartTimer
- StopTimer
- ResetTimer

setup & teardown

```go
func TestMain(m *testing.M) {

  // setup
  fmt.Println("test main setup")
  code := m.Run()
  //testing.MainStart(xxx)

  // teardown
  fmt.Println("test main teardown:", code)
  os.Exit(code)
}
```

```
➜  test go test -v
test main setup
=== RUN   TestRepeat
--- PASS: TestRepeat (0.00s)
PASS
test main teardown: 0
ok      haiziwang.com/godemo/test       0.005s
```

fixtures

- 编译出的执行文件在临时目录
- working目录会被设置为测试代码所在目录
- 目录名以"."或"_"，目录名为"testdata”会被go tool忽略
  - 测试数据存放在testdata目录
  - f, err := os.Open("testdata/somefixture.json")

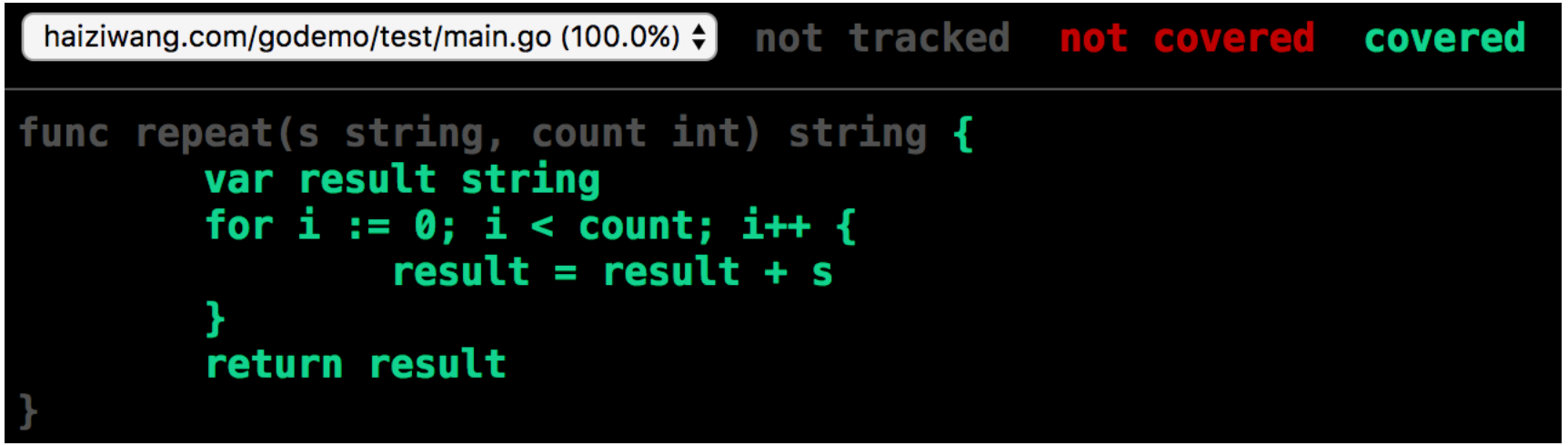
cover

- -cover
- -covermode
  - set
  - count
  - atomic
- -coverprofile
- go tool cover
  - -func
  - -html

```
➜  test go test -cover
PASS
coverage: 100.0% of statements
ok      haiziwang.com/godemo/test       0.006s
```

cover

```
➜  test go test -coverprofile=coverage.out
PASS
coverage: 100.0% of statements
ok      haiziwang.com/godemo/test       0.006s
➜  test go tool cover -func=coverage.out
haiziwang.com/godemo/test/main.go:7:    repeat          100.0%
total:                                  (statements)    100.0%
➜  test go tool cover -html=coverage.out
```



Subtests

- 控制粒度细化到subtest的级别
- setup和teardown不再局限尽在TestMain级别
- 简化编写一组相似测试的工作量

Subtests

```go
func BenchmarkRepeatSubtest(b *testing.B) {
  benchmarks := []struct {
    name  string
    s     string
    count int
  }{
    {"small-s;small-count", "abc", 3},
    {"small-s;large-count", "abc", 30},
    {"large-s;small-count", "abcdefghijklmn", 3},
    {"large-s;large-count", "abcdefghilklmn", 30},
  }
  for _, bm := range benchmarks {
    b.Run(bm.name, func(b *testing.B) {
      for i := 0; i < b.N; i++ {
        repeat(bm.s, bm.count)
      }
    })
  }
}
```

Subtests

```
➜  test go test -run=NONE -bench Subtest
BenchmarkRepeatSubtest/small-s;small-count-8          20000000               107 ns/op
BenchmarkRepeatSubtest/small-s;large-count-8           1000000              1673 ns/op
BenchmarkRepeatSubtest/large-s;small-count-8          10000000               119 ns/op
BenchmarkRepeatSubtest/large-s;large-count-8            500000              2712 ns/op
PASS
ok      haiziwang.com/godemo/test       6.674s
➜  test go test -run=NONE -bench Subtest/small-s
BenchmarkRepeatSubtest/small-s;small-count-8          20000000               103 ns/op
BenchmarkRepeatSubtest/small-s;large-count-8           1000000              1660 ns/op
PASS
ok      haiziwang.com/godemo/test       3.862s
```

规则

- 编写可测试的代码
  - 全局变量
  - 依赖
- 测试既文档
- mock
- table driver
- net/http/httptest
- -race

反射

reflect

- 反射所有信息来自接口
  - func TypeOf(i interface{}) Type
  - func ValueOf(i interface{}) Value
- 可寻址

```
x := 2                   // value   type    variable?
a := reflect.ValueOf(2)  // 2       int     no
b := reflect.ValueOf(x)  // 2       int     no
c := reflect.ValueOf(&x) // &x      *int    no
d := c.Elem()            // 2       int     yes (x)
```

《The Go Programming Language》

reflect

- 值 reflect.Value
- 类型
  - 真实类型 reflect.Type
  - 底层类型 reflect.Kind
- 操作
  - 遍历成员
  - 创建(new、make)
  - get
  - set
  - call

reflect.Type

- 值类型和指针类型属于不同类型
- TypeOf返回具体类型
- Elem()返回基类型
  - （Ptr、Array、Slice、Map、Chan）
- Type支持比较
- 能读取非导出类型
  - 不能设置

```go
var w io.Writer = os.Stdout
fmt.Println(reflect.TypeOf(w))
// Output: *os.File

w = &bytes.Buffer{}
fmt.Println(reflect.TypeOf(w))
// Output: *bytes.Buffer
```

reflect.Type

- TypeOf ()
- XxxOf
  - ArrayOf ()
  - ChanOf ()
  - MapOf ()
  - 。 。 。

```
w := struct {
  w io.Writer
}{w: os.Stdout}

f, _ := reflect.TypeOf(w).FieldByName("w")
fmt.Println(f.Type)
// io.Writer

wv := reflect.ValueOf(w)
fmt.Println(wv.FieldByName("w").Type())
fmt.Println(wv.FieldByName("w").Elem().Type())
// io.Write
// *os.File
```

reflect.Kind

- Invalid
- Bool
- Int
- Int8
- ◦ ◦ ◦

Type & Kind

```
type N int

func demo() {
  var i = 100
  var n N = 100
  ti, tn, tip, tnp :=
    reflect.TypeOf(i), reflect.TypeOf(n), reflect.TypeOf(&i), reflect.TypeOf(&n)

  fmt.Printf("type i:%v; type n:%v; type &i:%v; type &n:%v  \n",
    ti, tn, tip, tnp)
  fmt.Printf("kind i:%v; kind n:%v; kind &i:%v; kind &n:%v \n",
    ti.Kind(), tn.Kind(), tip.Kind(), tnp.Kind())
  fmt.Printf("elem type &i:%v; elem kind &i:%v  \n",
    tip.Elem(), tnp.Elem().Kind())
}
```

```
type i:int; type n:main.N; type &i:*int; type &n:*main.N
kind i:int; kind n:int; kind &i:ptr; kind &n:ptr
elem type &i:int; elem kind &i:int
```

reflect.Value

- ValueOf(i interface{}) Value
  - 接口复制变量,
  - 不可寻址

```
a := 1
va, vp := reflect.ValueOf(a), reflect.ValueOf(&a)
elem := vp.Elem()

fmt.Println(va.CanAddr(), va.CanSet())     // false false
fmt.Println(vp.CanAddr(), vp.CanSet())     // false false
fmt.Println(elem.CanAddr(), elem.CanSet()) // true true
```

reflect.Value

- ==比较的是value struct
- IsNil 不等同于v== nill
- Zero
- IsValid
- 多数方法非类型安全

```
v1, v2 := reflect.ValueOf(1), reflect.ValueOf(1)
fmt.Println(v1 == v2)                          // false
fmt.Println(v1.Type() == v2.Type())            // true
fmt.Println(v1.Interface() == v2.Interface()) // true
```

```
var a interface{}
v := reflect.ValueOf(a)

fmt.Println(a == nil)    // true
fmt.Println(v.Kind())    // invalid
fmt.Println(v.IsValid()) // false
fmt.Println(v.IsNil())   // panic
```

nil

```
var a interface{}
var b interface{} = (*int)(nil)

fmt.Println(a == nil)                    // true
fmt.Println(b == nil)                    // false
fmt.Println(reflect.ValueOf(b).IsNil()) // true
```

判断

- Type
  - Implements(u Type) bool
  - AssignableTo(u Type) bool
  - ConvertibleTo(u Type) bool
  - Comparable() bool
- Value
  - CanAddr() bool
  - CanInterface() bool
  - CanSet() bool

AssignableTo & ConvertibleTo

```
type N int

func convert() {
  i := 1
  var n N = 1

  ti, tn := reflect.TypeOf(i), reflect.TypeOf(n)
  fmt.Println(ti.AssignableTo(tn), ti.ConvertibleTo(tn))
  // false, true
}
```

CanAddr & CanSet

```go
type foo struct {
  i int
  X int
}

func can() {
  f := foo{1, 2}
  p := &foo{1, 2}
  fi, fx, pi, px := reflect.ValueOf(f).Field(0), reflect.ValueOf(f).Field(1),
    reflect.ValueOf(p).Elem().Field(0), reflect.ValueOf(p).Elem().Field(1)

  fmt.Println(fi.CanAddr(), fi.CanSet()) // false false
  fmt.Println(fx.CanAddr(), fx.CanSet()) // false false
  fmt.Println(pi.CanAddr(), pi.CanSet()) // true false
  fmt.Println(px.CanAddr(), px.CanSet()) // true true
}
```

遍历

```go
type base struct {
  a int
  B string
}

type child struct {
  base
  B []byte
  C interface{}
}
```

```go
var x child
t := reflect.TypeOf(x)

for i := 0; i < t.NumField(); i++ {
  f := t.Field(i)
  fmt.Println(f.Name, f.Type, f.Offset)

  if f.Anonymous {
    for y := 0; y < f.Type.NumField(); y++ {
      af := f.Type.Field(y)
      fmt.Println("+--", af.Name, af.Type, af.Offset)
    }
  }
}
```

创建

- MakeXxx
  - MakeChan
  - MakeFunc
  - MakeMap
  - MakeSlice
- New
- NewAt

MakeFunc

```
swap := func(in []reflect.Value) []reflect.Value {
  return []reflect.Value{in[1], in[0]}
}

makeSwap := func(fptr interface{}) {
  fn := reflect.ValueOf(fptr).Elem()
  v := reflect.MakeFunc(fn.Type(), swap)
  fn.Set(v)
}

var intSwap func(int, int) (int, int)
makeSwap(&intSwap)
fmt.Println(intSwap(0, 1)) // 1 0

var floatSwap func(float64, float64) (float64, float64)
makeSwap(&floatSwap)
fmt.Println(floatSwap(2.72, 3.14)) // 3.14 2.72
```

get

- Interface() interface{}
  - 接口值到反射对象再转回来
- 类型
  - Bool() bool
  - Bytes() []byte
  - Float() float64
  - 。。。

set

- Set
  - Set(x Value)
- SetXxx
  - SetBool()
  - SetBytes()
  - SetInt()
  - 。 。 。

```
x := 2
b := reflect.ValueOf(x)
b.Set(reflect.ValueOf(3))          // panic
d := reflect.ValueOf(&x).Elem()    // d refers to the variable x
px := d.Addr().Interface().(*int) // px := &x
*px = 3                            // x = 3
fmt.Println(x)                     // "3"
d.Set(reflect.ValueOf(4))
fmt.Println(x) // "4"
```

《The Go Programming Language》

call

- Call
  - func (v Value) Call(in []Value) []Value
- CallSlice
  - func (v Value) CallSlice(in []Value) []Value

内置函数

- Copy(dst, src Value) int
- Append(s Value, x ...Value) Value
- func (v Value) Cap() int
- func (v Value) Len() int
- func (v Value) Convert(t Type) Value

chan

- func (v Value) Close()
- func (v Value) Recv() (x Value, ok bool)
- func (v Value) Send(x Value)
- func (v Value) TryRecv() (x Value, ok bool)
- func (v Value) TrySend(x Value) bool
- Select(cases []SelectCase) (chosen int, recv Value, recvOK bool)

反射

- 类型安全
- 静态类型检查
- 机制比较难用
- 性能风险
  - type switch配合
  - 缓存
  - 代码生成

profile

pprof

- runtime/pprof
- net/http/pprof
  - import _ "net/http/pprof"
- go tool pprof
  - go tool pprof /path/to/your/binary /path/to/your/profile
  - https://github.com/google/pprof
- go test
  - -cpuprofile
  - -memprofile
  - -blockprofile

pprof.Profile

- goroutine
- heap
- threadcreate
- block
- mutex

package

- expvar
- runtime
- runtime/debug

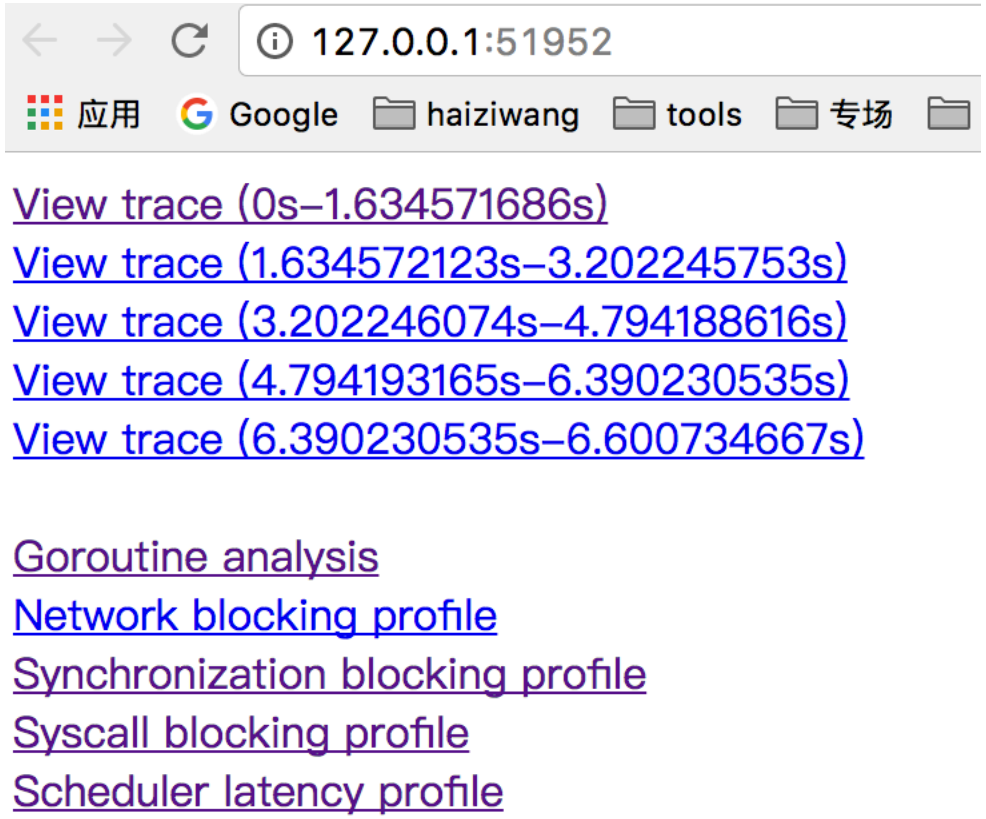
env

- GODEBUG
  - gctrace=xxx
  - allocfreetrace=xxx
  - schedtrace=xxx
- GOMAXPROCS
- GOTRACEBACK

https://dave.cheney.net/tag/godebug

trace

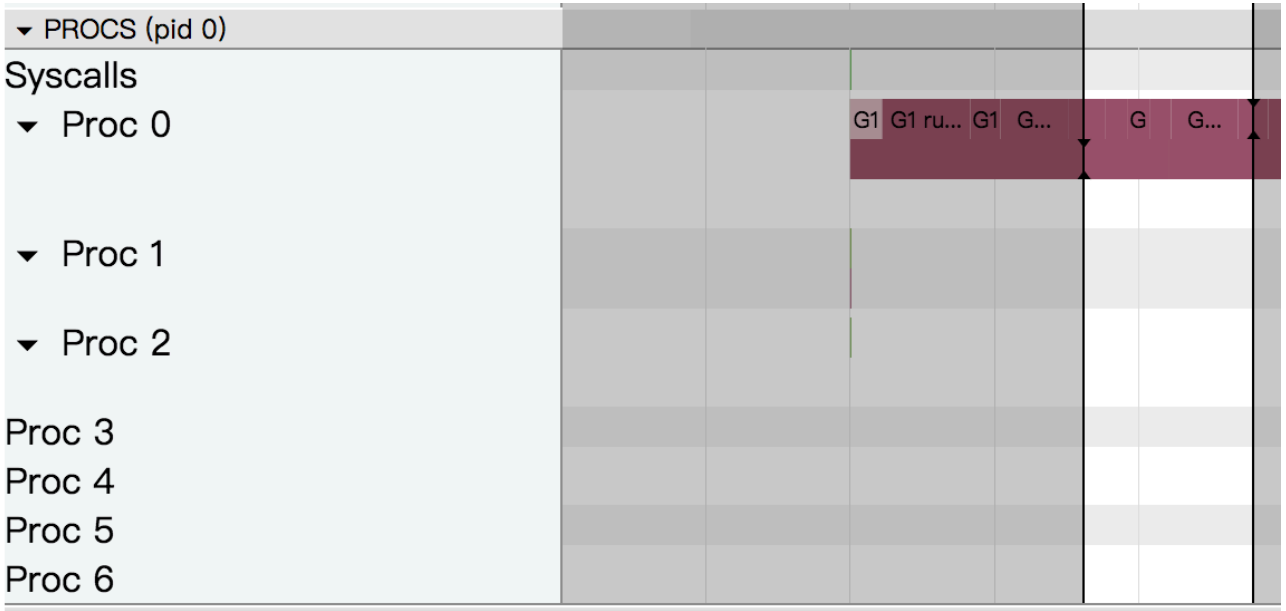
- runtime/trace
- go test -trace=trace.out
- go tool trace [flags] trace.out



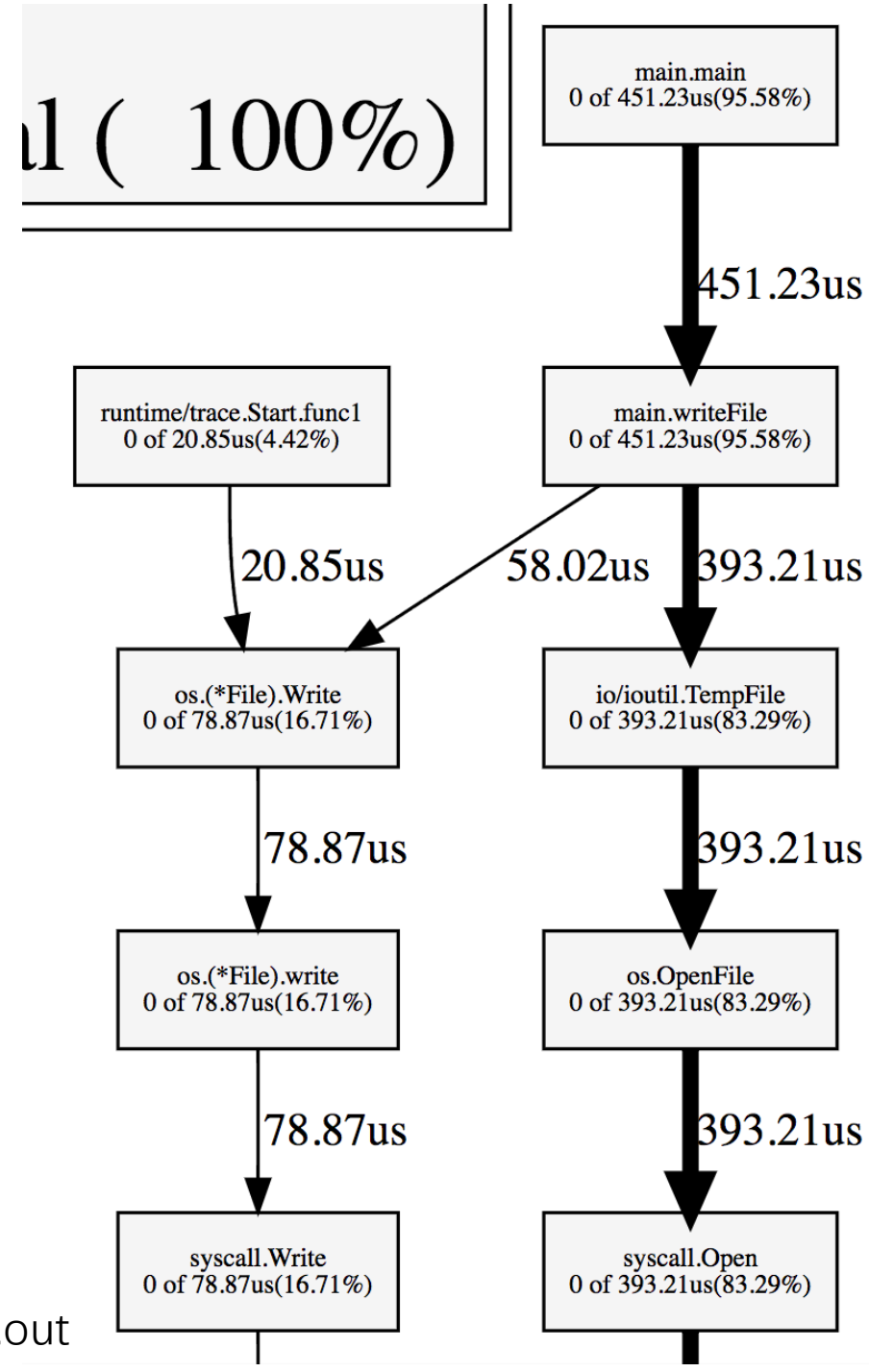
trace

```go
func writeFile(data []byte) {
  for i := 0; i < 10000; i++ {
    f, _ := ioutil.TempFile("", "test")
    defer os.Remove(f.Name())
    if _, err := f.Write(data); err != nil {
      log.Fatal(err)
    }
    f.Close()
  }
}
```

```go
func main() {
  f, err := os.Create("trace.out")
  if err != nil {
    log.Fatal(err)
  }
  if err := trace.Start(f); err != nil {
    log.Fatal(err)
  }

  data := []byte(strings.Repeat("abcde", 100))
  writeFile(data)

  trace.Stop()
}
```



1 item selected:	Slice (1)
Title	G1 runtime.main
Start	0.019 ms
Wall Duration	44.357 ms
Self Time	44.357 ms
Start Stack Trace	**Title**
	runtime.main:106
End Stack Trace	**Title**
	path/filepath.Clean:154
	path/filepath.join:42
	path/filepath.Join:205
	io/ioutil.TempFile:54
	main.writeFile:14
	main.main:66



go tool trace trace.out

第三方工具

- https://github.com/uber/go-torch
- https://github.com/pkg/profile
- https://github.com/google/gops

优化原则

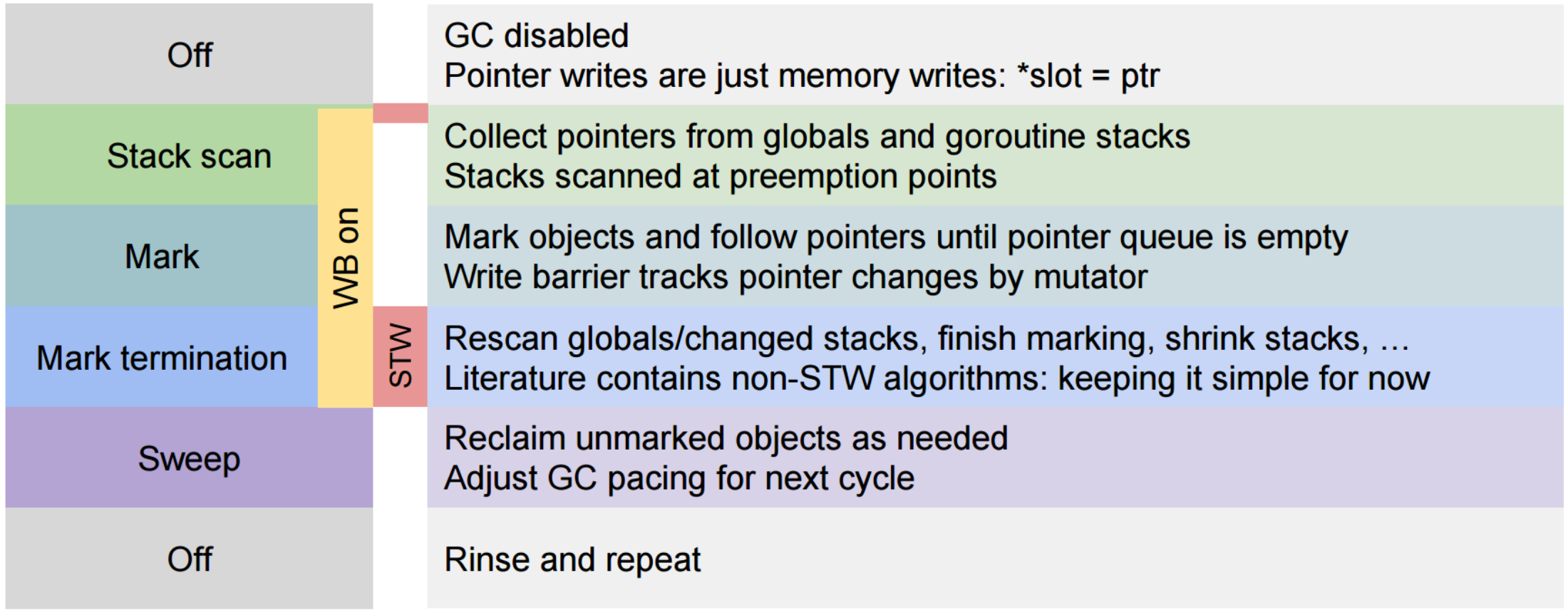
- 不要过早优化
- 一次关注一个指标
- 一切以数据为准
- 合理比性能更重要
- 平衡稳定性、可扩展、延迟、吞吐量
- 平衡RPC、IO、网络、DB、Cache、MQ
- 从系统的整体看性能瓶颈
- 架构设计优于算法，算法优于语言层面优化

GC

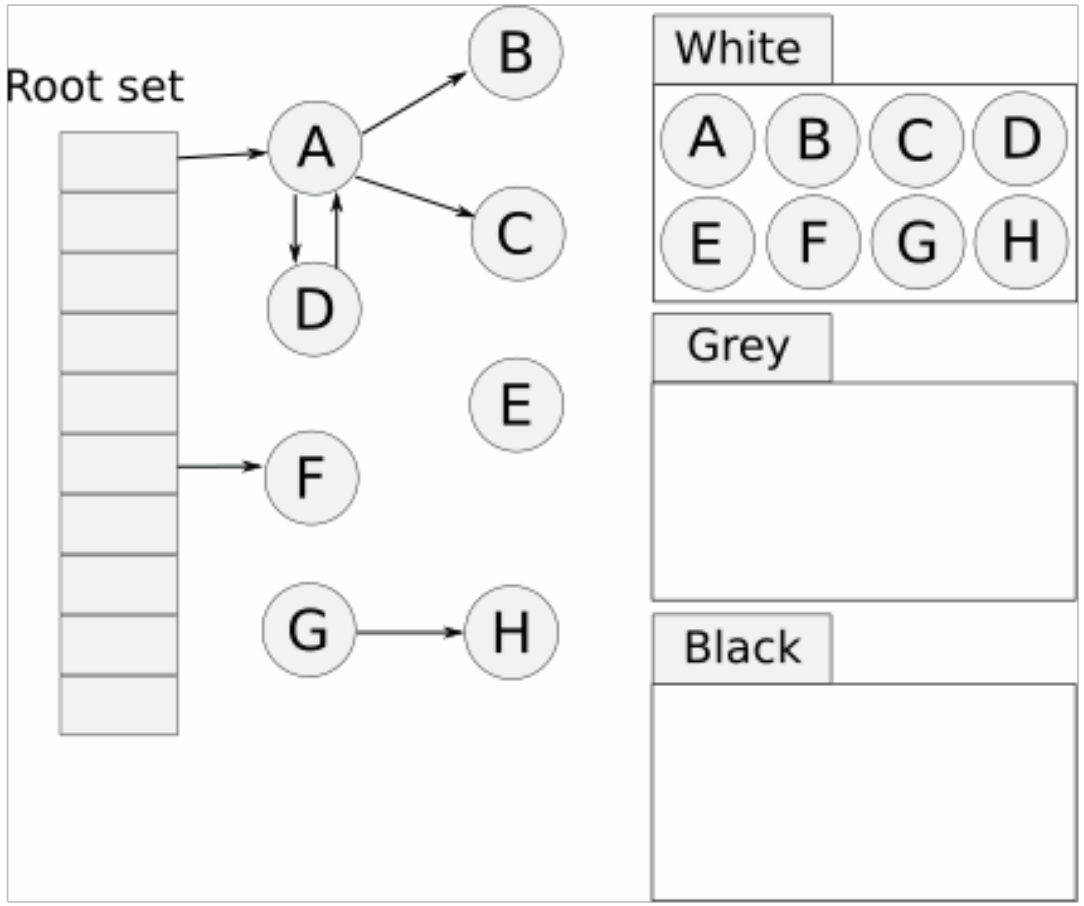
- 非分代
- 写屏障(WB)
- 并行三色标记
  - 扫描
  - 标记
  - 清理

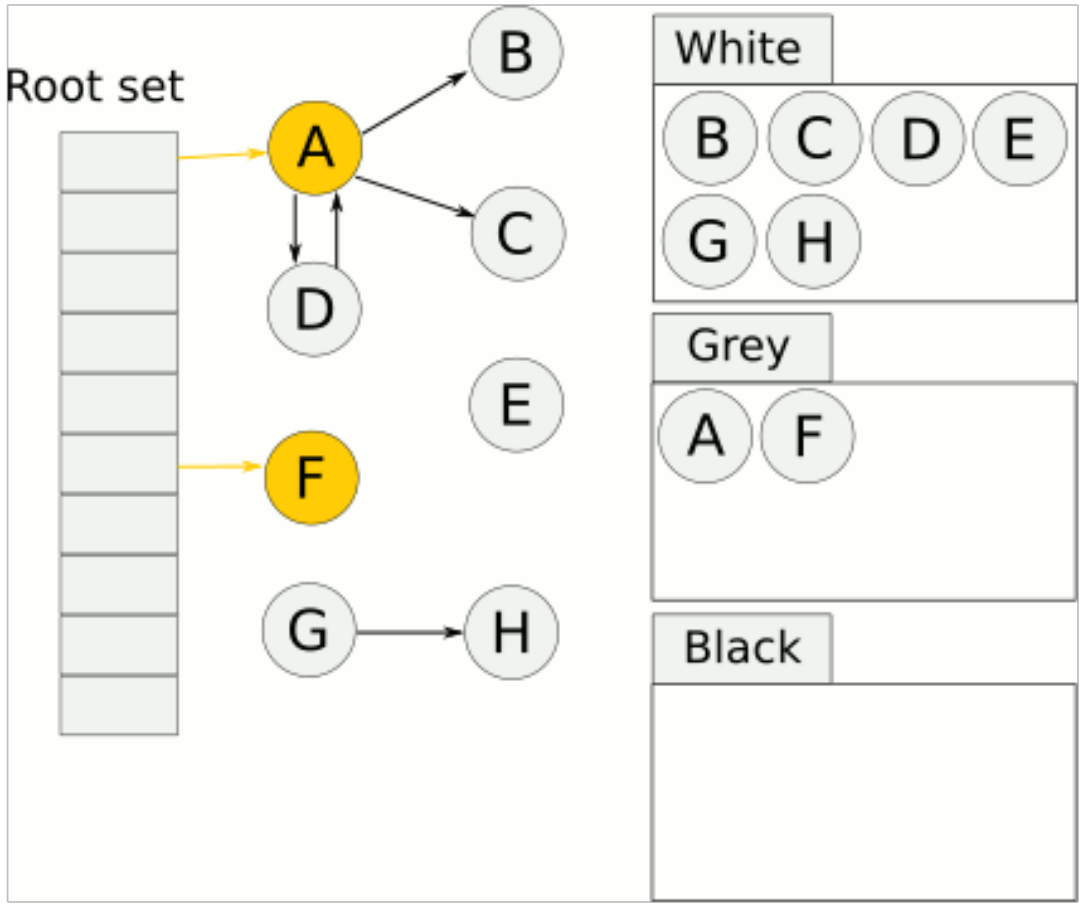
GC

GC Algorithm Phases



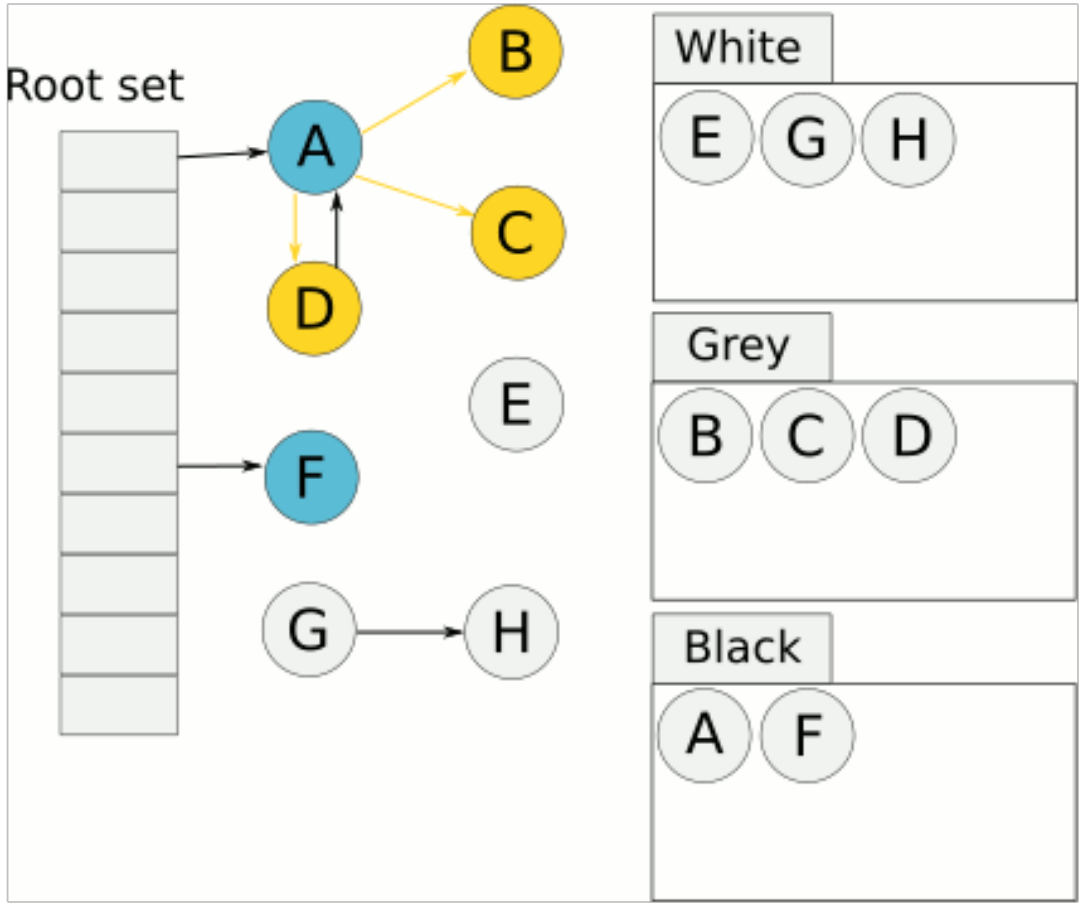
mark & sweep

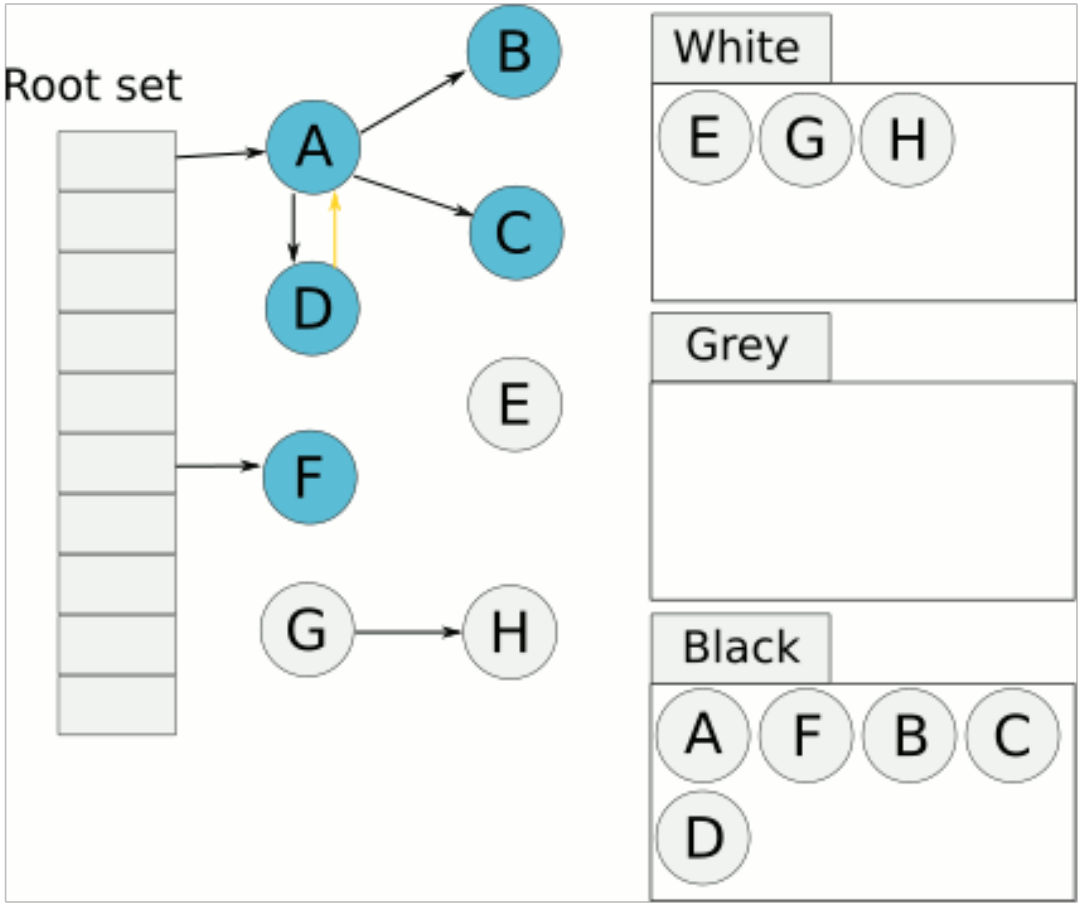




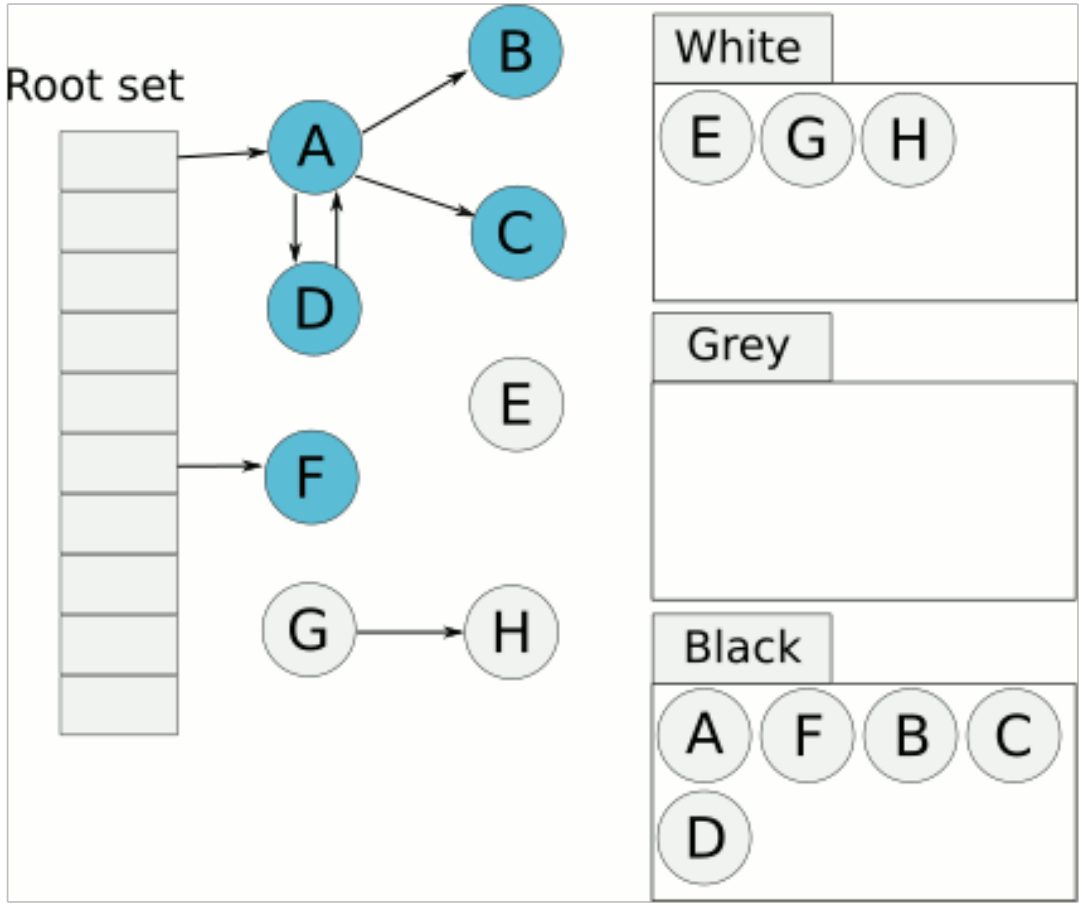
https://en.wikipedia.org/wiki/Tracing_garbage_collection

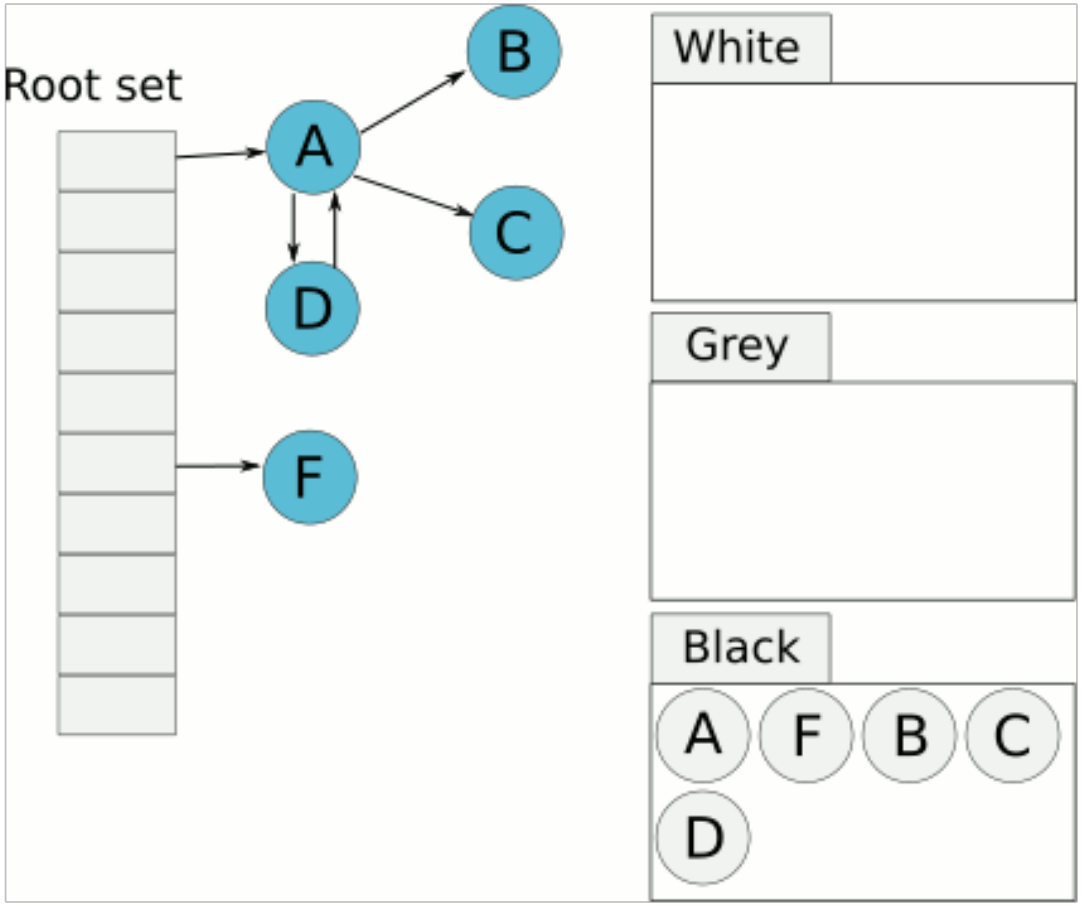
mark & sweep


Root set
A
B
C
D
E
F
G
H
White
E G H
Grey
B C D
Black
A F


Root set
A
B
C
D
E
F
G
H
White
E G H
Grey
Black
A F B C
D

mark & sweep


Root set
A
B
C
D
E
F
G
H
White
E G H
Grey
Black
A F B C
D


Root set
A
B
C
D
F
White
Grey
Black
A F B C
D

编译器优化

- 每个版本都有优化
- 编译器尽可能使用寄存器和栈来存储对象
- 编译器用逃逸分析来判断局部变量是否会被外部引用
  - gcflags -m
- string和[]byte的优化
- interface{}的优化
  - 没有泛型
- 大对象 (32k)

分析

- profile、trace、runtime/debug
- runtime.ReadMemStats (STW)
- GODEBUG
  - gctrace
  - allocfreetrace
  - schedtrace
- 编译参数gcflags
  - -N：关闭编译器优化
  - -l：关闭函数内联
  - -m：输出编译器优化分析，如逃逸分析等

析构

- runtime.SetFinalizer
- 加入待执行队列，下次垃圾回收时清理
- 无法预计执行时间，无法保证一定会执行

GC触发

- 环境变量GOGC阈值
  - 当新分配的内存大小/上次GC存活的内存大小
  - 默认100
  - runtime/debug/#SetGCPercent
- 如果2分钟没执行清理，则强制执行一次

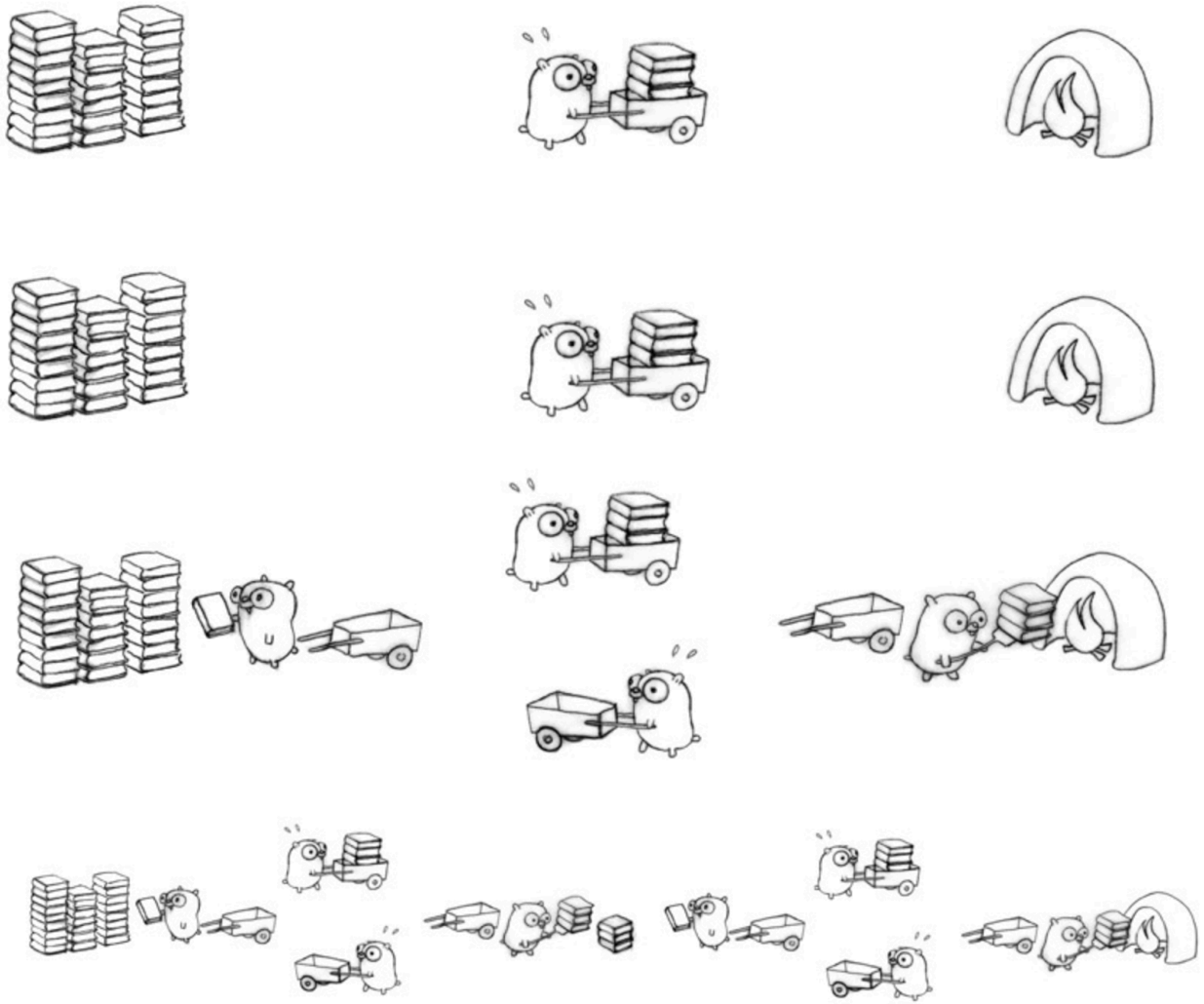
GC原则

- 使用pool、buffer，复用内存
- 减少堆内存分配, 降低GC负担
- 避免string连接，避免[]byte和string频繁转换
- 容器类对象创建时预估大小，减少动态扩容
- 避免频繁创建临时对象
- 函数参数类型避免需要创建中间变量的实现

并发

并发 &并行

- 并发(concurrency)
  - 逻辑，设计
- 并行(parallesim)
  - 物理，执行



并发

- 多进程
- 多线程
- 协程(coroutine)
- goroutine
  - 多线程+协程

并发

- go：创建并发任务 (runtime.newproc)
- chan：消息通讯
- select：控制
- sync：同步
- atomic：原子操作

goroutine

```go
func task(done chan struct{}) {
  fmt.Println("task start")
  time.Sleep(time.Second)
  fmt.Println("task done")
  close(done)
}

func main() {
  done := make(chan struct{})
  go task(done)

  fmt.Println("main waiting ...")
  <-done
  fmt.Println("main done")
}
```

goroutine

- 初始栈2KB
- 连续栈 (continuous stack)
  - 旧栈数据复制到新栈
  - 继续执行
- 对比线程
  - 堆栈大小
  - 优先级
  - 线程ID
  - Thead Local Storage
  - 切换成本

chan

- Share Memory By Communicating
  - Do not communicate by sharing memory
- CSP： Communicating Sequential Processes
  - https://en.wikipedia.org/wiki/Communicating_sequential_processes

chan

- 同步/异步 (buffer)
  - c:= make(chan int)
  - c := make(chan int, 10)
- 单向/双向
  - chan int 双向
  - chan<- int 发送
  - <-chan int 接收
- nil的chan永远阻塞
- close的chan永远不会阻塞
  - 通过v, ok := <-ch模式判断是否关闭

chan

```go
func produce(data chan int) {
  defer close(data)

  for i := 0; i < 10; i++ {
    time.Sleep(time.Duration(rand.Int63n(10)) * time.Millisecond)
    data <- i
    fmt.Println("producer:", i)
  }
}

func main() {
  rand.Seed(time.Now().UnixNano())

  data := make(chan int, 3)
  go produce(data)

  for x := range data {
    time.Sleep(time.Duration(rand.Int63n(10)) * time.Millisecond)
    fmt.Println("consumer:", x)
  }
}
```

select

- 规则
  - 随机选择一个可用通道
  - 都不可用时执行default语句

select

```
func any() {
  a := make(chan time.Duration)
  b := make(chan time.Duration)
  var ok bool
  var x time.Duration

  go wait(a)
  go wait(b)

  for !ok {
    select {
    case x, ok = <-a:
      fmt.Println("a\t", x, ok)
    case x, ok = <-b:
      fmt.Println("b\t", x, ok)
    }
  }
}
```

```
func sleep(name string, c chan int) chan int {
  d := rand.Intn(100)
  time.Sleep(time.Duration(d) * time.Millisecond)
  c <- d
  fmt.Println("sleep done", name, d)
  return c
}
```

```
func both() {
  a := make(chan time.Duration)
  b := make(chan time.Duration)

  go wait(a)
  go wait(b)

  for a != nil || b != nil {
    select {
    case x, ok := <-a:
      fmt.Println("a\t", x, ok)
      a = nil
    case x, ok := <-b:
      fmt.Println("b\t", x, ok)
      b = nil
    }
  }
}
```

default

`select` for non-blocking receive

A buffered channel makes a simple queue

```
idle:= make(chan []byte, 5)

select {
case b = <-idle:

default:
    makes += 1
    b = make([]byte, size)
}
```

Try to get from the idle queue

Idle queue empty? Make a new buffer

`select` for non-blocking send

A buffered channel makes a simple queue

```
idle:= make(chan []byte, 5)

select {
case idle <- b:

default:
}
```

Try to return buffer to the idle queue

Idle queue full? GC will have to deal with the buffer

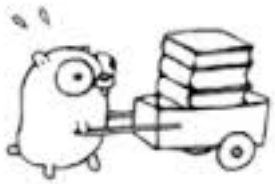
《A Channel Compendium》

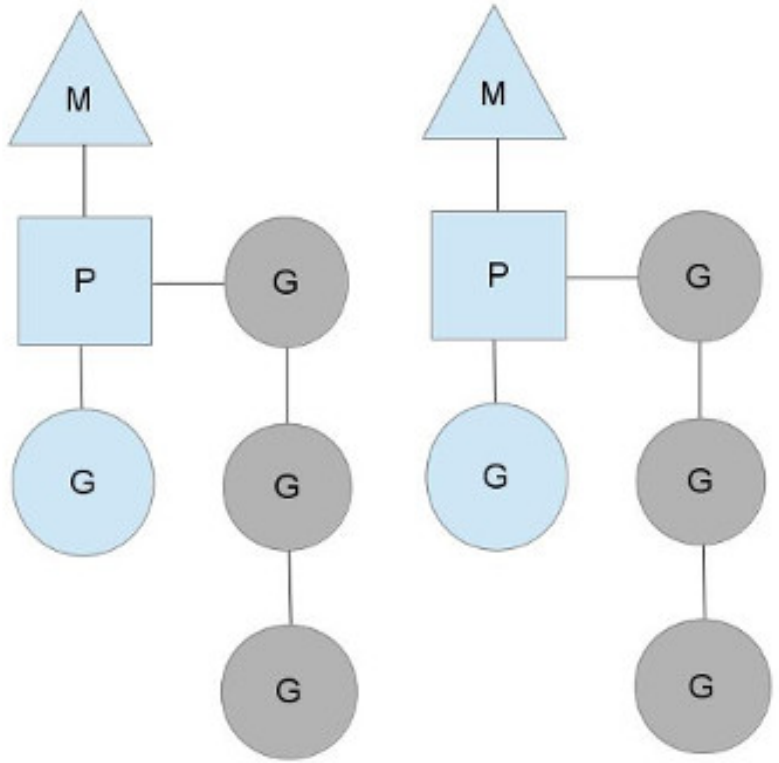
Patterns

- Pipeline
- 等待通知开始/结束
- 任意一个完成
- 等待全部完成
- 超时
- 传递channel的channel
- Context https://blog.golang.org/context
- Cancellation

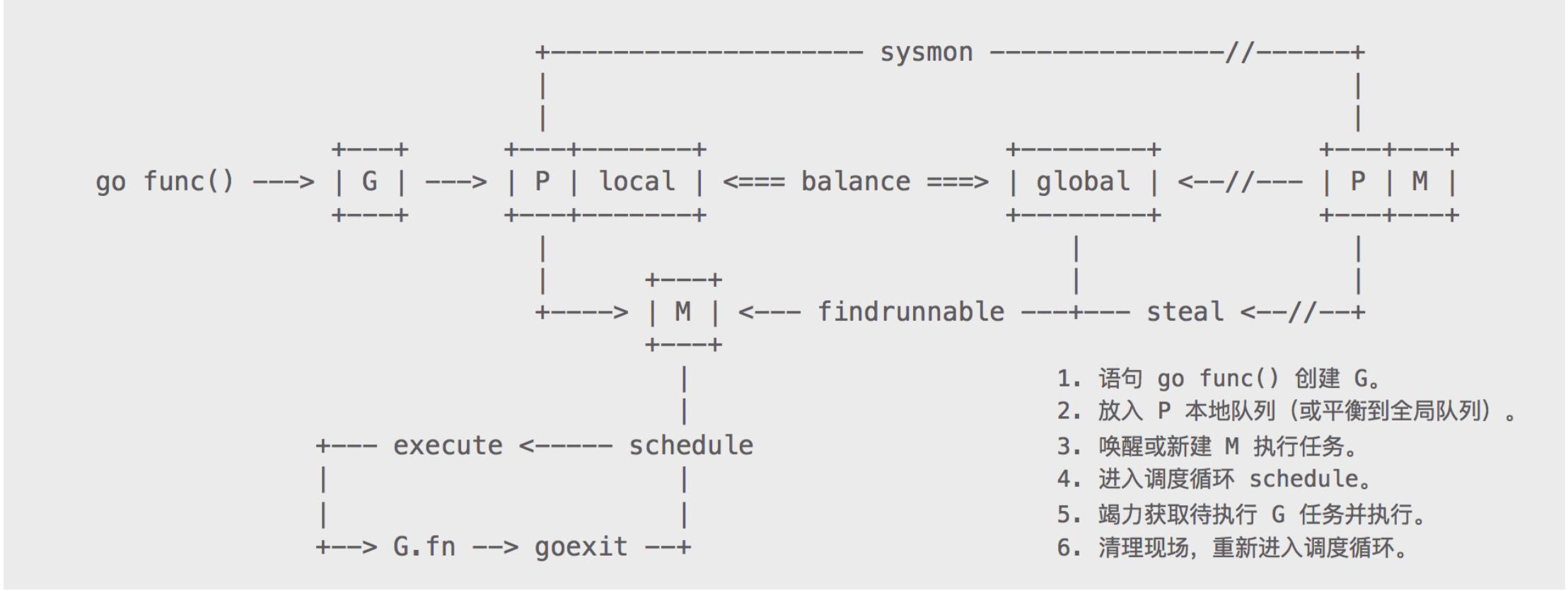
scheduler

- P：processor
  - 类似CPU
  - 默认等于CPU核数
- G：goroutine
  - 任务状态
  - 初始栈2KB
- M：
  - 系统线程，需要绑定一个P
  - 默认最多10000(debug.SetMaxThreads)
- Sched





scheduler



《Go 1.5 源码剖析》

调度

- GODEBUG=schedtrace=xxx
- runtime.GOMAXPROCS
  - 默认为cpu核数 (runtime.NumCPU)
  - 同名环境变量
- runtime.Gosched
- runtime.Goexit
  - 终止所在的goroutine
  - main goroutine调用不终止其他goroutine运行

调度

```go
func work(wg *sync.WaitGroup) {
  time.Sleep(time.Second)
  var counter int
  for i := 0; i < 1e10; i++ {
    counter++
  }
  wg.Done()
}

func main() {
  var wg sync.WaitGroup
  wg.Add(10)

  for i := 0; i < 10; i++ {
    go work(&wg)
  }
  wg.Wait()
  time.Sleep(2 * time.Second)
}
```

GODEBUG=schedtrace=1000

调度

```
➜  scheduler GODEBUG=schedtrace=1000 GOMAXPROCS=2 go run main.go
SCHED 0ms: gomaxprocs=2 idleprocs=0 threads=4 spinningthreads=1 idlethreads=0 runqueue=0 [0 0]
# command-line-arguments
SCHED 0ms: gomaxprocs=2 idleprocs=0 threads=3 spinningthreads=1 idlethreads=0 runqueue=0 [0 0]
# command-line-arguments
SCHED 0ms: gomaxprocs=2 idleprocs=0 threads=3 spinningthreads=1 idlethreads=0 runqueue=0 [0 0]
SCHED 0ms: gomaxprocs=2 idleprocs=0 threads=3 spinningthreads=1 idlethreads=0 runqueue=0 [0 0]
SCHED 1007ms: gomaxprocs=2 idleprocs=2 threads=7 spinningthreads=0 idlethreads=2 runqueue=0 [0 0]
SCHED 1007ms: gomaxprocs=2 idleprocs=0 threads=4 spinningthreads=0 idlethreads=1 runqueue=0 [4 4]
SCHED 2016ms: gomaxprocs=2 idleprocs=2 threads=7 spinningthreads=0 idlethreads=2 runqueue=0 [0 0]
SCHED 2016ms: gomaxprocs=2 idleprocs=0 threads=4 spinningthreads=0 idlethreads=1 runqueue=0 [4 4]
SCHED 3022ms: gomaxprocs=2 idleprocs=2 threads=7 spinningthreads=0 idlethreads=2 runqueue=0 [0 0]
SCHED 3021ms: gomaxprocs=2 idleprocs=0 threads=4 spinningthreads=0 idlethreads=1 runqueue=0 [4 4]
SCHED 4025ms: gomaxprocs=2 idleprocs=2 threads=7 spinningthreads=0 idlethreads=2 runqueue=0 [0 0]
SCHED 4030ms: gomaxprocs=2 idleprocs=0 threads=4 spinningthreads=0 idlethreads=1 runqueue=0 [4 4]
SCHED 5028ms: gomaxprocs=2 idleprocs=2 threads=7 spinningthreads=0 idlethreads=2 runqueue=0 [0 0]
```

sync

- type Cond
  - func (c *Cond) Broadcast()
  - func (c *Cond) Signal()
  - func (c *Cond) Wait()
- type Once
  - func (o *Once) Do(f func())
- type WaitGroup
  - func (wg *WaitGroup) Add(delta int)
  - func (wg *WaitGroup) Done()
  - func (wg *WaitGroup) Wait()

sync.Locker

- type Mutex
  - func (m *Mutex) Lock()
  - func (m *Mutex) Unlock()
- type RWMutex
  - func (rw *RWMutex) Lock()
  - func (rw *RWMutex) RLock()
  - func (rw *RWMutex) RUnlock()
  - func (rw *RWMutex) Unlock()

atomic

- AddXxx
  - func AddInt32(addr *int32, delta int32) (new int32)
- LoadXxx
  - func LoadInt32(addr *int32) (val int32)
- StoreXxx
  - func StoreInt32(addr *int32, val int32)
- SwapXxx
  - func SwapInt32(addr *int32, new int32) (old int32)
- CompareAndSwapXxx
  - func CompareAndSwapInt32(addr *int32, old, new int32) (swapped bool)

happens-before

- A happens-before B
  - A对内存的影响将对执行B的线程(且执行B之前)可见
  - 可传递
- 非时间先后概念
  - A happens-before B并不意味着A在B之前发生
  - A在B之前发生并不意味着A happens-before B
- 处理
  - 多级缓存
  - 编译优化，指令重排
  - CPU乱序执行

happens-before

```
var a string
var done bool

func setup() {
        a = "hello, world"
        done = true
}

func doprint() {
        if !done {
                once.Do(setup)
        }
        print(a)
}

func twoprint() {
        go doprint()
        go doprint()
}
```

```
var a string
var done bool

func setup() {
        a = "hello, world"
        done = true
}

func main() {
        go setup()
        for !done {
        }
        print(a)
}
```

https://golang.org/ref/mem

go synchronization

- go启动一个新goroutine **happens-before** 该goroutine开始执行
- channel的发送 **happens-before** 该channel的接收
- channel的关闭 **happens-before** 该channel接收到最后的返回值
- 无缓冲channel的接收 **happens-before** 该channel的发送

https://golang.org/ref/mem

channel happens-before

```
var c = make(chan int)
var a string

func f() {
  a = "hello, world" // (1)
  <-c                // (2)
}
func main() {
  go f()
  c <- 0          // (3)
  fmt.Println(a) // (4)
  <-time.After(time.Second)
}
```

```
var c = make(chan int, 1)
var a string

func f() {
  a = "hello, world" // (1)
  <-c                // (2)
}
func main() {
  go f()
  c <- 0          // (3)
  fmt.Println(a) // (4)
  <-time.After(time.Second)
}
```

timer

```go
func do1() {
  c := make(chan int)
  var timeout int
  var total int

  fmt.Println("start do1 ...")

  go func() {
    for {
      select {
      case _, ok := <-c:
        if !ok {
          return
        }
        total++
      case <-time.After(time.Millisecond):
        timeout++
      }
    }
  }()

  for i := 0; i < 10; i++ {
    c <- i
    time.Sleep(time.Second)
  }
  close(c)
  fmt.Println("total:", total, "timeout count", timeout)
}
```

```go
func do2() {
  c := make(chan int)
  var timeout int
  var total int

  fmt.Println("start do2 ...")

  go func() {
    t := time.NewTimer(time.Millisecond)
    for {
      select {
      case _, ok := <-c:
        if !ok {
          return
        }
        total++
        t.Stop()
        t.Reset(time.Millisecond)
      case <-t.C:
        timeout++
        t.Reset(time.Millisecond)
      }
    }
  }()

  for i := 0; i < 10; i++ {
    c <- i
    time.Sleep(time.Second)
  }
  close(c)
  fmt.Println("total:", total, "timeout count", timeout)
}
```

timer

```
➜  time go run main.go
start do1 ...
total: 10 timeout count 7960
start do2 ...
total: 10 timeout count 7949
➜  time go test -bench . -benchmem
start do1 ...
total: 10 timeout count 7954
BenchmarkDo1-8                 1        10001640175 ns/op         1535440 B/op      23938 allocs/op
start do2 ...
total: 10 timeout count 8101
BenchmarkDo2-8                 1        10001992678 ns/op            3464 B/op         32 allocs/op
PASS
ok      haiziwang.com/godemo/time       20.012s
```

并发原则

- 要理论保证，错误不能重现不代表逻辑正确
- 确保知道goroutine如何结束，避免goroutine泄漏
- 并发访问变量用锁
- race检测
- 清楚并发的成本和风险

Q / A