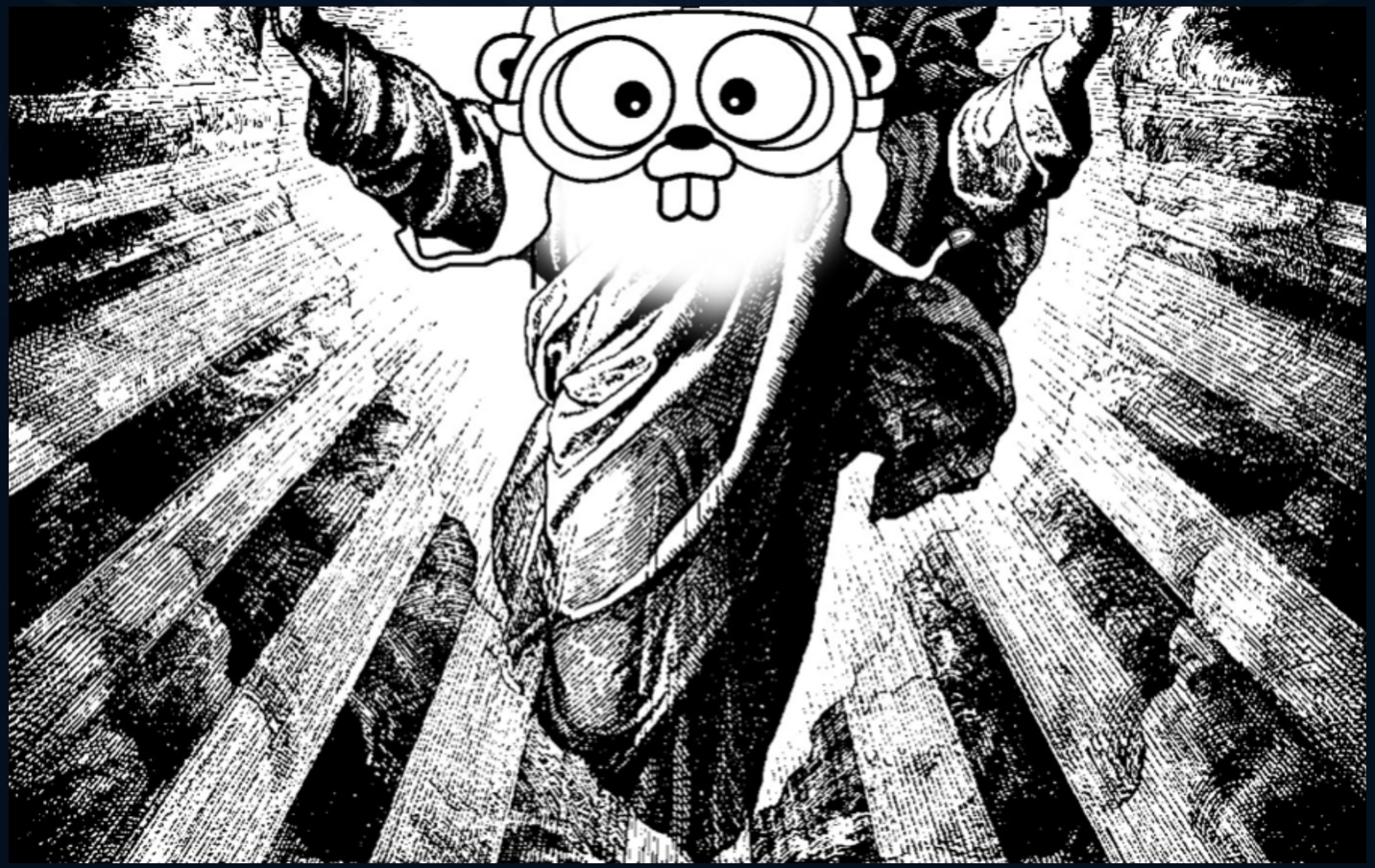
Let’s Go [2]

看看GO

Hiko Qiu / 2016-04-28

“Embrace Go, Embrace the future.”



目录

1 标准库概览

2 数据结构

3 并发/并行

4 goroutine

5 调度方式

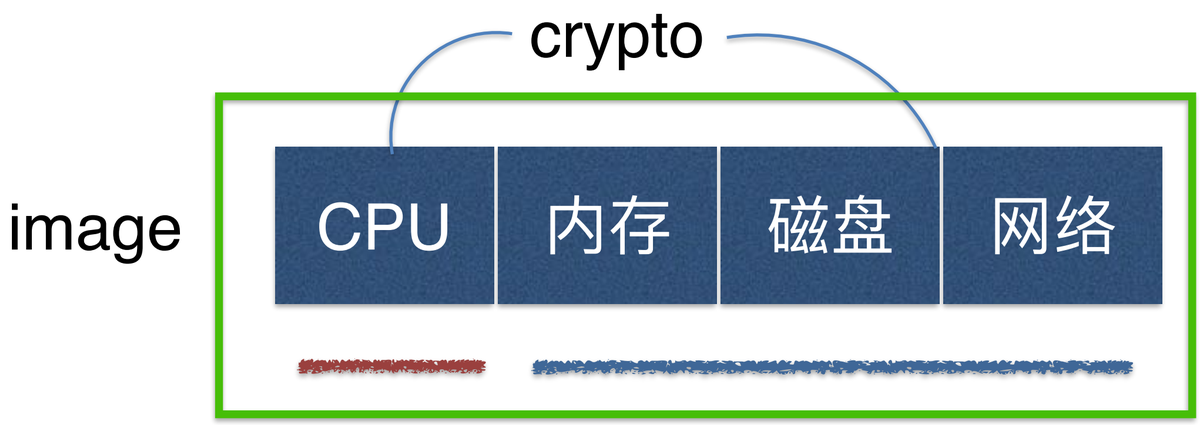
6 并发安全

1. 标准库概览

1. 标准库概览

```
/usr/local/go/src
➜  src git:(master) ls
1q                bootstrap.bash  clean.rc     encoding  html        make.bash     os        run.bat     syscall
Make.dist         bufio           cmd          errors    image       make.bat      path      run.rc      testing
all.bash          buildall.bash   compress     expvar    index       make.rc       race.bash runtime     text
all.bat           builtin         container    flag      internal    math          race.bat  sort        time
all.rc            bytes           crypto       fmt       io          mime          reflect   strconv     unicode
androidtest.bash  clean.bash      database     go        iostest.bash nacltest.bash regexp   strings     unsafe
archive           clean.bat       debug        hash      log         net           run.bash  sync        vendor
➜  src git:(master)
```

1. 标准库概览



os
path
syscall

sync
atomic/mutex/waitgroup
channel
runtime
race/pprof/trace/signal/stack/…

bytes
bufio
strings
strconv
encoding

archive
container
reflecting
hash
sort
regexp
cmd/flag
debug

1. 标准库概览 - 应用

基础/公共服务

net (HTTP/TCP/UDP server)
e.g : 推送服务/HTTP 服务器/RPC调用

io/path/os — 文件/系统
e.g : 分布式文件系统/监控系统

image - 图片处理
e.g : 验证码公共服务

... 等等

服务端业务

net
e.g : Mysql/Redis/Mongo... 客户端

其他工具

e.g : log ...等等

PHP就不能实现相同的功能吗？


```c
struct _zval_struct {
        zend_value        value;                        /* value */
        union {
                struct {
                        ZEND_ENDIAN_LOHI_4(
                                zend_uchar    type,                     /* active type */
                                zend_uchar    type_flags,
                                zend_uchar    const_flags,
                                zend_uchar    reserved)     /* call info for EX(This) */
                } v;
                uint32_t type_info;
        } u1;
        union {
                uint32_t     next;                /* hash collision chain */
                uint32_t     cache_slot;          /* literal cache slot */
                uint32_t     lineno;              /* line number (for ast nodes) */
                uint32_t     num_args;            /* arguments number for EX(This) */
                uint32_t     fe_pos;              /* foreach position */
                uint32_t     fe_iter_idx;         /* foreach iterator index */
                uint32_t     access_flags;        /* class constant access flags */
        } u2;
};
```

Ref - PHP _zval_struct

2. 数据结构

2. 数据结构

意义

了解实现、值传递、放心使用

例子

slice / channel

string # 只读的[]byte

map # 下次介绍

slice

A slice is not an array. A slice describes a piece of an array.

By Rob Pike

2.1 数据结构 - slice用法

初始化变量

```go
// 方法一: make (常用)
ages := make([]int, 5, 20)
fmt.Println(ages, len(ages), cap(ages))

ages = make([]int, 5)
fmt.Println(ages, len(ages), cap(ages))

// 方法二: 初始化
names := []string{"Name1", "Name2"}
fmt.Println(names, len(names), len(names))
```

常用操作

```go
// copy
users := []int{5, 6}
newUsers := make([]int, 10)
copy(newUsers[4:], users)
fmt.Println(newUsers, len(newUsers), cap(newUsers))
```

```go
// append
users = append(users, 9)
fmt.Println(users, len(users), cap(users))
```

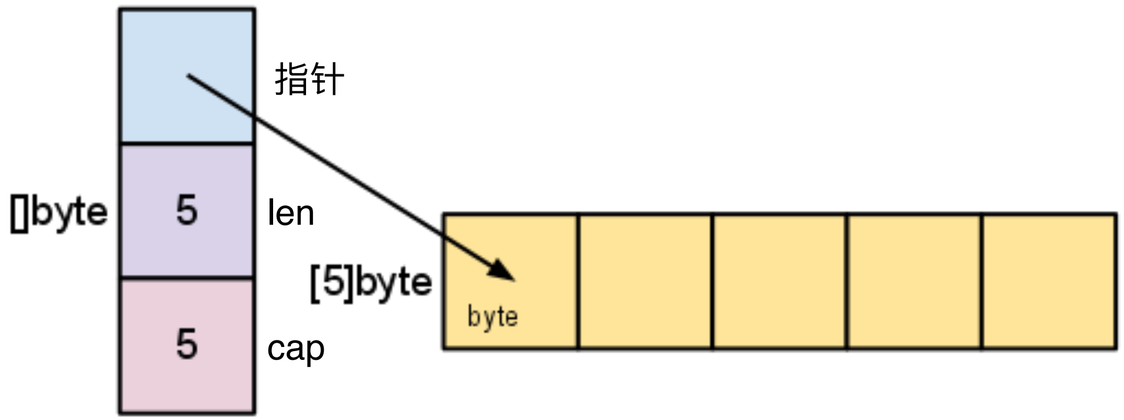
2.1 数据结构 - slice结构

结构

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

SliceHeader

样子



0 <= len(slice) <= cap(slice)

2.1 数据结构 - slice结构

make 实际上干的事

```
// TODO: take uintptrs instead of int64s?
func makeslice(t *slicetype, len64, cap64 int64) slice {
    // NOTE: The len > MaxMem/elemsize check here is not strict
    // but it produces a 'len out of range' error instead of a
    // when someone does make([]T, bignumber). 'cap out of rang
    // but since the cap is only being supplied implicitly, say
    // See issue 4085.
    len := int(len64)
    if len64 < 0 || int64(len) != len64 || t.elem.size > 0 && u
        panic(errorString("makeslice: len out of range"))
    }
    cap := int(cap64)
    if cap < len || int64(cap) != cap64 || t.elem.size > 0 && u
        panic(errorString("makeslice: cap out of range"))
    }
    p := newarray(t.elem, uintptr(cap))
    return slice{p, len, cap}
}
```

len / cap 最大值?

```
bigs := make([]byte, 1 << 64 - 1)
fmt.Println(bigs)
```

// On other 64-bit platforms, we limit the arena to 512GB, or 39 bits.
// On 32-bit, we don't bother limiting anything, so we use the full 32-bit address.

2.1 数据结构 - slice len & cap

例子

```
ints := make([]int, 10, 10) // len = 0, diff ?
ints = append(ints, 1)
ints = append(ints, 2)
fmt.Println(len(ints), cap(ints))
```

- The length of a slice identifies the number of elements of the underlying array we are using from the starting index position.
- The capacity identifies the number of elements we have available for use.

// 使用 append 需知

- Remember, since the slice header is always updated by a call to append, you need to save the returned slice after the call. In fact, the compiler won't let you call append without saving the result.

2.2 数据结构 - new / make 区别

- `new(T)`
  - Allocates memory for item with type `T`
  - Zeroes it

```
ret = runtime·mallocgc(typ->size, flag, 1, 1);
```

zeroed

- `make(T)`
  - Allocates memory for item with type `T`
  - Initializes it
  - Needed for channels, maps and slices

- Memory comes from internal heap

w and make?

, map, and channel types.

Ref - [Allocation with new]

2.2 数据结构 - new / make 区别

```
sliceMake := make([]int, 0)
sliceNew := new([]int)
fmt.Printf("make slice : %v \n", sliceMake == nil)
fmt.Printf("new slice : %v \n", *sliceNew == nil)
```

2.3 数据结构 - channel用法

1. 无缓冲 channel

```
// 1. 无缓冲 channel # CSP
wait := make(chan bool)
go func() {
    fmt.Println("goroutine waits 2 seconds.")
    time.Sleep(time.Second * 2)
    wait <- true
}()

<-wait
fmt.Println("go main finish.")
```

2. 有缓冲 channel

```
// 2. 带缓冲 channel
bufc := make(chan bool, 3)
bufc <- true
bufc <- true
bufc <- true
//bufc <- true
fmt.Println("over.")
```

channel

Don't communicate by sharing memory, share memory by communicating.

— Go Proverbs

2.3 数据结构 - channel用法

问题：无缓冲 和 缓冲为 1 一样？

```
cNoBuf := make(chan bool)
cBuf := make(chan bool, 1)

cNoBuf <- true
cBuf <- true
```

```
// 无缓冲
c := make(chan bool)

go func() {
    fmt.Println("before c <- true")
    c <- true
    fmt.Println("after c <- true")
}()

time.Sleep(time.Second)
fmt.Println("get c : ",  <- c)

time.Sleep(time.Second)
fmt.Println("finish")
```

```
// 缓冲为1
c := make(chan bool, 1)

go func() {
    fmt.Println("before c <- true")
    c <- true
    fmt.Println("after c <- true")
}()

time.Sleep(time.Second)
fmt.Println("get c : ",  <- c)

time.Sleep(time.Second)
fmt.Println("finish")
```

// 1. 看看len(chan) 区别
// 2. make(chan int, 0)

2.3 数据结构 - why channel

IPC 方法

- shm (共享内存) / shm_open()
- semaphore (信号量) / sem_open()
- pipe (管道) / pipe()
- socket (套接字) / socket
- condwait (条件等待) / pthread_cond_wait()
- signal (信号) / kill()

…

以上都是内核提供

2.3 数据结构 - why channel

CSP

通信顺序进程

CSP := Communication Sequential Processing

is a formal language for describing patterns of interaction
in concurrent systems.

一种用于定义并发交互的形式语言。

CSP was first described in a 1978 paper by C. A. R. Hoare

2.3 数据结构 - why channel

CSP定义的交互

1. 通过Send/Receive实现通信
2. 保证收发同时成功

```go
// 无缓冲
c := make(chan bool)

go func() {
    fmt.Println("before c <- true")
    c <- true
    fmt.Println("after c <- true")
}()

time.Sleep(time.Second)
fmt.Println("get c : ",  <- c)

time.Sleep(time.Second)
fmt.Println("finish")
```

```go
// 无缓冲
c := make(chan bool)

c <- true
<-c

fmt.Println("finish")
```

deadlock !

2.3 数据结构 - channel数据结构

channel结构体

```go
type hchan struct {
    qcount   uint           // total data in the queue
    dataqsiz uint           // size of the circular queue
    buf      unsafe.Pointer // points to an array of dataqsiz elements
    elemsize uint16
    closed   uint32
    elemtype *_type // element type
    sendx    uint   // send index
    recvx    uint   // receive index
    recvq    waitq  // list of recv waiter
    sendq    waitq  // list of send waiter
    lock     mutex
}

type waitq struct {
    first *sudog
    last  *sudog
}
```

```go
// Known to compiler.
// Changes here must also be made in src/cmd/inte
type sudog struct {
    g           *g
    selectdone  *uint32
    next        *sudog
    prev        *sudog
    elem        unsafe.Pointer // data element
    releasetime int64
    nrelease    int32  // -1 for acquire
    waitlink    *sudog // g.waiting list
}
```

2.3 数据结构 - channel数据结构

带锁的**FIFO**队列

```
lock(&c.lock)

if c.closed != 0 {
    unlock(&c.lock)
    panic("send on closed channel")
}

if sg := c.recvq.dequeue(); sg != nil {
    // Found a waiting receiver. We pass the value we want t
    // directly to the receiver, bypassing the channel buffe
    send(c, sg, ep, func() { unlock(&c.lock) })
    return true
}

if c.qcount < c.dataqsiz {
    // Space is available in the channel buffer.  Enqueue th
    qp := chanbuf(c, c.sendx)
    if raceenabled {
        raceacquire(qp)
        racerelease(qp)
    }
```

问题：channel效率一定高么？

mutex/RWMutex

3. 并发/并行

Concurrency is not parallelism.

— Go Proverbs

3. 并发/并行 - 概念

并发是指同时有很多事要做，你可以串行处理也可以并行处理。

并行是指同时做多件事。

图 12-30　顺序、并发和并行程序集合之间的关系

Ref -
- 并发和并行的区别
- Concurrency is not parallelism
- 并发不是并行

3. 并发/并行 - 举个例子

KowloonZh 喂小孩奶

1. 一只手 + 一个奶瓶 + 一个小孩

2. 一只手 + 一个奶瓶 + 两个小孩

3. 两只手 + 两个奶瓶 + 两个小孩

4. goroutine

4.1 goroutine - 一个并发例子①

i. Go1.5 / Go1.6 goroutine实现

```go
for i:=0; i < 100; i++ {
    go func() {
        fmt.Println("当前 goroutine id : ", GoID())
    }()
}

for {}
```

```
...

当前 goroutine id :  22
当前 goroutine id :  23
当前 goroutine id :  24
当前 goroutine id :  25
当前 goroutine id :  26
当前 goroutine id :  77
当前 goroutine id :  51
当前 goroutine id :  52

...
```

备注：Go1.5 / Go1.6执行结果不一样！为什么？ 类似例子1 类似的例子2 Go1.6 解决 (Preemptible)

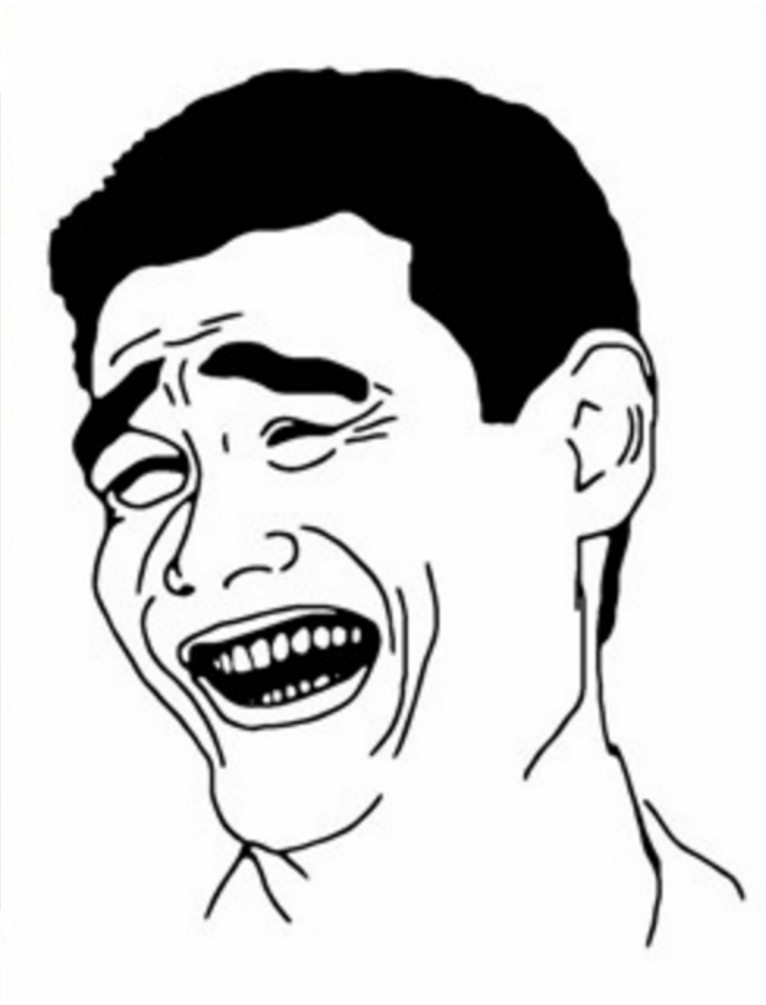
4.1 goroutine - 一个并发例子②

ii. C 语言 多进程

```php
// 1.2 fork $procNum 个子进程
for($i = 0; $i < $procNum; $i++) {
    $offset = $i * $step;
    $cpid = pcntl_fork();

    // fork 失败
    if($cpid == -1) {
        Loger::error(['current_pid' => $curPid], 'fork.'.$logFile);
        continue;
    }

    // 多进程并发操作的时候，不可以使用相同一个Redis链接的句柄，否则会报：Fail
    // 所以这里需要为每个子进程分配新的Redis链接
    $r = $this->cloneRedis();
```



大PHP也可以并发操作 →.→

4.1 goroutine - 一个并发例子③

iii. Java 多线程

```
for(int i = 0; i < 100; i++) {
    PrintNumThread tr = new PrintNumThread();
    tr.start();
}

while (true) {}
```

```
@Override
public void run() {
    super.run();

    try {
        Thread.currentThread().sleep(5);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("current thread id : "+ Thread.currentThread().getId());
}
```

```
current thread id : 63
current thread id : 94
current thread id : 66
current thread id : 67
current thread id : 95
current thread id : 69
current thread id : 93
current thread id : 98
current thread id : 96
current thread id : 103
current thread id : 92
```

goroutine, 要你何用？

4.2 goroutine - Why ①

① 内存开销

线程：8M

进程：> 8M

```
➜  ~ ulimit -s
8192
```

64位：8M，32位：2M

Linux进程栈和线程栈

参考资料：

http://blog.csdn.net/xhhjin/article/details/7579145

总结：

1、进程的栈大小是在进程执行的时刻才能指定的，即不是在编译的时候决定的，也不是在链接的时候决定的

2、进程的栈大小是随机确定的至少比线程栈要大，但是不到线程栈大小的2倍

3、线程栈大小是固定的，也就是ulimit -a 显示的值

goroutine：2K　　备注：Linux, >= Go1.4

Ref -

虚拟地址空间，堆栈、堆、数据段、代码段

Linux进程栈和线程栈

pthread_create

Changes to the runtime (1.4)

How Stacks are Handled in Go

4.2 goroutine - Why ①

查看进程的堆 sudo cat /proc/{pid}/maps

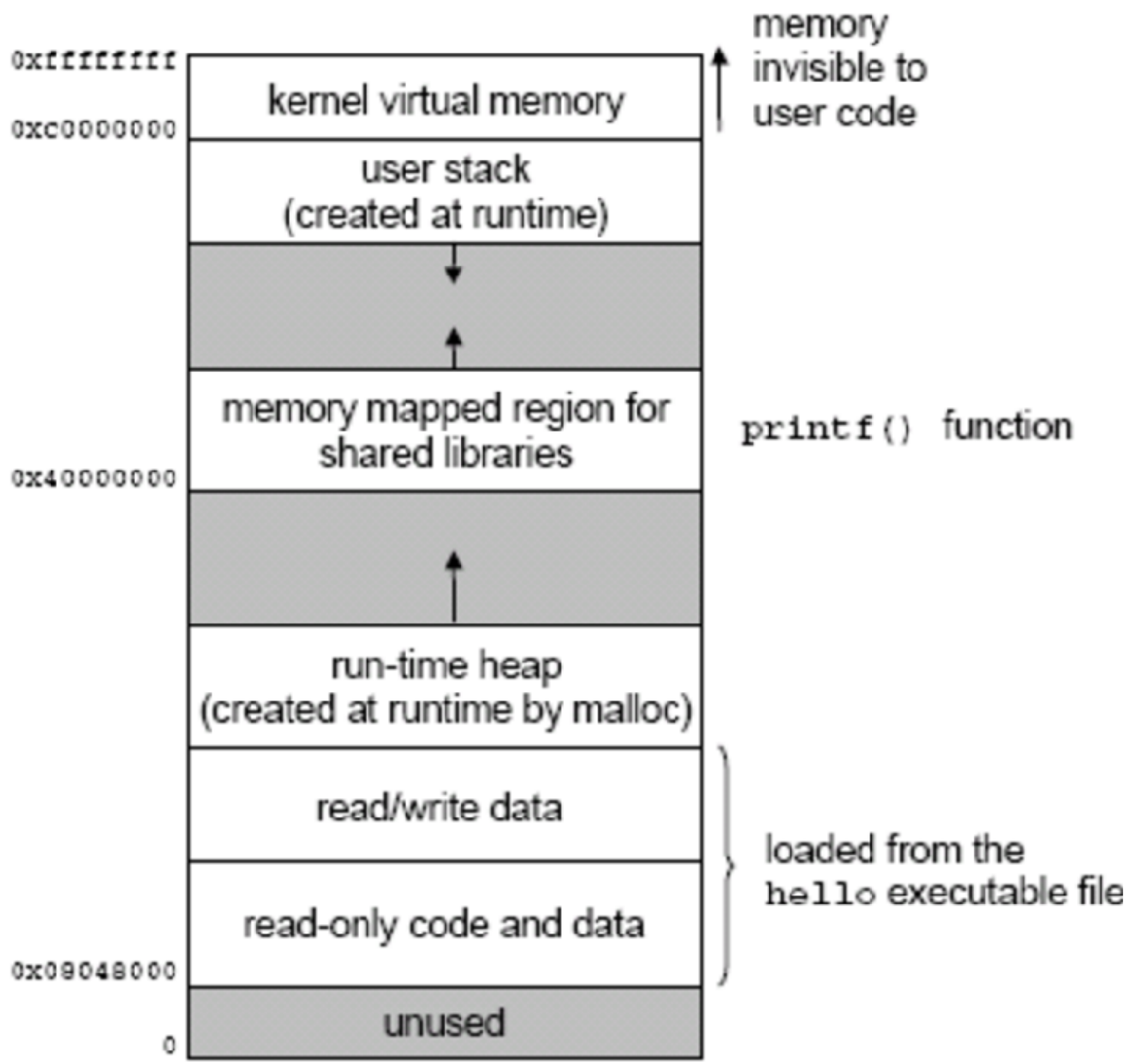


Figure 1.13: **Linux process virtual address space.**

Ref - 线程栈

4.2 goroutine - Why ①

goroutine Stack 大小

```
newg := gfget(_p_)
if newg == nil {
    newg = malg(_StackMin)
    casgstatus(newg, _Gidle, _Gdead)
    allgadd(newg) // publishes with a
}
```

```
// Allocate a new g, with a stack big enough for stacksize bytes.
func malg(stacksize int32) *g {
    newg := new(g)
    if stacksize >= 0 {
        stacksize = round2(_StackSystem + stacksize)
        systemstack(func() {
            newg.stack, newg.stkbar = stackalloc(uint32(stacksize))
        })
        newg.stackguard0 = newg.stack.lo + _StackGuard
        newg.stackguard1 = ^uintptr(0)
        newg.stackAlloc = uintptr(stacksize)
    }
    return newg
}
```

# 备注:
_StackMin = 2048 Bytes
_StackSystem = 0 Bytes // Linux

go func() ->newproc -> malg

① 内存开销 - AND THEN ?

- 10k个线程 (thread) 内存消耗

10k * 8M = 80G

C10K - connection

1w thread * 8M

- 10k个 goroutine 内存消耗

10k * 2K = 20M

1w , goroutine , 1w线程

300thread * 8M

备注:

以上计算只是理论值。

1. 实际应用中可以调整线程默认分配栈的大小。
2. goroutine 是寄托于线程，并且计算理论值时，未考虑goroutine遇到系统调用时， runtime 重新分配线程等处理。

4.2.1 进程/线程/goroutine 切换

If you do asynchronous preemption you need save/restore ALL registers, that is, 16 general purpose registers, PC, SP, segment registers, 16 XMM registers, FP coprocessor state, X AVX registers, all MSRs and so on.
If you cooperatively preempt at known points, then you need to safe/restore only what is known to be alive at these points. In Go that's just 3 registers -- PC, SP and DX.

Ref - Dmitry Vyukou 的解释

Vyukou优化了 runtime 调度器

他的博客

4.2 goroutine - Why ②

4.2.2 进程切换

表 3.10　UNIX 进程映像

用户级上下文	
进程正文	程序中可执行的机器指令
进程数据	由这个进程的程序可访问的数据
用户栈	包含参数、局部变量和在用户态下运行的函数指针
共享内存区	与其他进程共享的内存区，用于进程间的通信
寄存器上下文环境	
程序计数器	将要执行的下一条指令地址，该地址是内核中或用户内存空间中的内存地址
处理器状态寄存器	包含在抢占时的硬件状态，其内容和格式取决于硬件
栈指针	指向内核栈或用户栈的栈顶，取决于当前的运行模式
通用寄存器	与硬件相关
系统级上下文环境	
进程表项	定义了进程的状态，操作系统总是可以取到这个信息
U（用户）区	含有进程控制信息，这些信息只需要在该进程的上下文环境中存取
本进程区表	定义了从虚地址到物理地址的映射，还包含一个权限域，用于指明进程允许的访问类型：只读、读写或读-执行
内核栈	当进程在内核态下执行时，它含有内核过程的栈帧

用户级上下文包括用户程序的基本成分，可以由已编译的目标文件直接产生。用户程序被分

4.2 goroutine - Why②

4.2.3 线程切换

内核级线程

在一个纯粹的内核级线程软件中，有关线程管理的所有工作都是由内核完成的，应用程序部分没有进行线程管理的代码，只有一个到内核线程设施的应用程序编程接口（API）。Windows是这种方法的一个例子。

图 4.6b 显示了纯粹的内核级线程的方法。内核为进程及其内部的每个线程维护上下文信息。调度是由内核基于线程完成的。该方法克服了用户级线程方法的两个基本缺陷。首先，内核可以同时把同一个进程中的多个线程调度到多个处理器中；其次，如果进程中的一个线程被阻塞，内核可以调度同一个进程中的另一个线程。内核级线程方法的另一个优点是内核例程自身也是可以使用多线程的。

相对于用户级线程方法，内核级线程方法的主要缺点是：在把控制从一个线程传送到同一个进程内的另一个线程时，需要到内核的状态切换。为说明它们的区别，表 4.1 给出了在单处理器 VAX 机上运行类 UNIX 操作系统的测量结果。这里进行了两种测试：Null Fork 和 Signal-Wait，前者测试创建、调度、执行和完成一个调用空过程的进程/线程的时间（也就是派生一个进程/线程的开销），后者测量进程/线程给正在等待的进程/线程发信号，然后在某个条件上等待所需要的时间（也就是两个进程/线程的同步时间）。可以看出用户级线程和内核级线程之间、内核级线程和进程之间都有一个数量级以上的性能差距。

表 4.1　线程和进程操作执行时间（μs）

操　作	用户级线程	内核级线程	进　程
Null Fork	34	948	11300
Signal-Wait	37	441	1840

4.3 goroutine - 一种情况

4.3 每个 goroutine 都发生阻塞的系统调用？

```
// SetMaxThreads returns the previous setting.
// The initial setting is 10,000 threads.
//
// The limit controls the number of operating system threads, not the number
// of goroutines. A Go program creates a new thread only when a goroutine
// is ready to run but all the existing threads are blocked in system calls, cgo calls,
// or are locked to other goroutines due to use of runtime.LockOSThread.
//
// SetMaxThreads is useful mainly for limiting the damage done by
// programs that create an unbounded number of threads. The idea is
// to take down the program before it takes down the operating system.
func SetMaxThreads(threads int) int {
    return setMaxThreads(threads)
}
```

5. 调度方式

runtime 对 goroutine 的调度方式

5.1 调度方式 - 概念

- Cooperative multitasking/time-sharing

协作式：正在执行的代码片段主动让出CPU，并切换到另一条执行路径。

4.1 例子 Go1.5 / Go1.6 区别

- Preemptive multitasking/time-sharing

抢占式：对正在执行的代码，被操作系统强行中断执行路径（例如通过时钟中断），并切换到另一个执行路径。

More - 用户态、内核态及进程切换和系统调用原理

5.1 调度方式 - 概念

Go 协作式

为什么代码中没主动 runtime.GoSched()，其他 goroutine 也被调度进行执行?

Go封装了系统调用，并且在每个可能发生阻塞的地方，主动让出CPU (P)，同时在长时间运行的 goroutine 也会让出 CPU (P)。

More - [Why 1000 goroutine generats 1000 os threads?]

5.1 调度方式 - 让出CPU (P)

```
// Hands off P from syscall or locked M.
// Always runs without a P, so write barriers are not allowed.
//go:nowritebarrier
func handoffp(_p_ *p) {
    // handoffp must start an M in any situation where
    // findrunnable would return a G to run on _p_.

    // if it has local work, start it straight away
    if !runqempty(_p_) || sched.runqsize != 0 {
        startm(_p_, false)
        return
    }
```

省略...

```
    // If this is the last running P and nobody is polling network,
    // need to wakeup another M to poll network.
    if sched.npidle == uint32(gomaxprocs-1) && atomic.Load64(&sched.
        unlock(&sched.lock)
        startm(_p_, false)
        return
    }
    pidleput(_p_)
    unlock(&sched.lock)
}
```

5.2 调度方式 - runtime

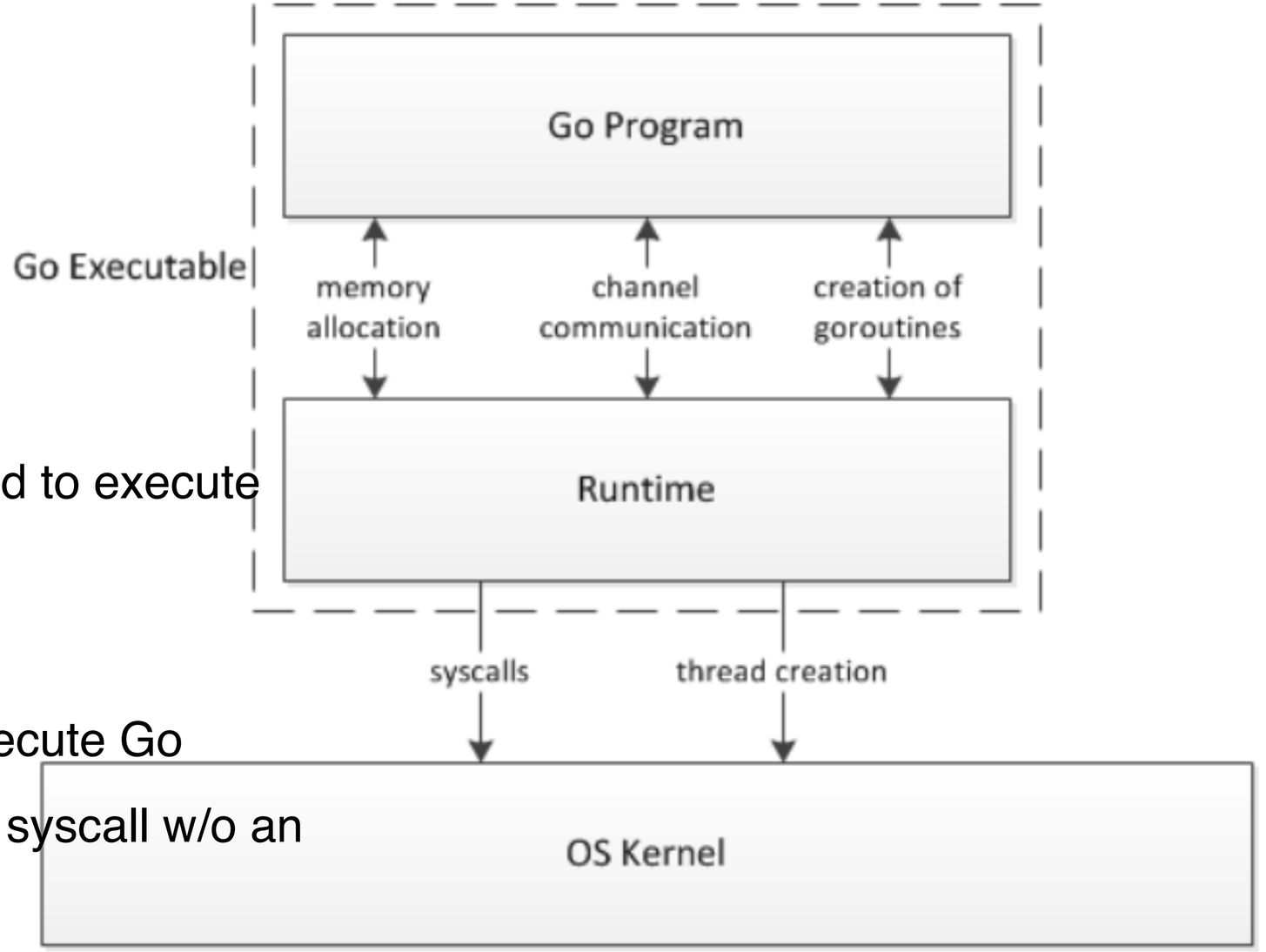
M / P / G

G - goroutine.

M - worker thread, or machine.

P - processor, a resource that is required to execute Go code.

Ps: M must have an associated P to execute Go code, however it can be blocked or in a syscall w/o an associated P.



Ref - Analyse of Go runtime scheduler

5.2 调度方式 - M / P / G

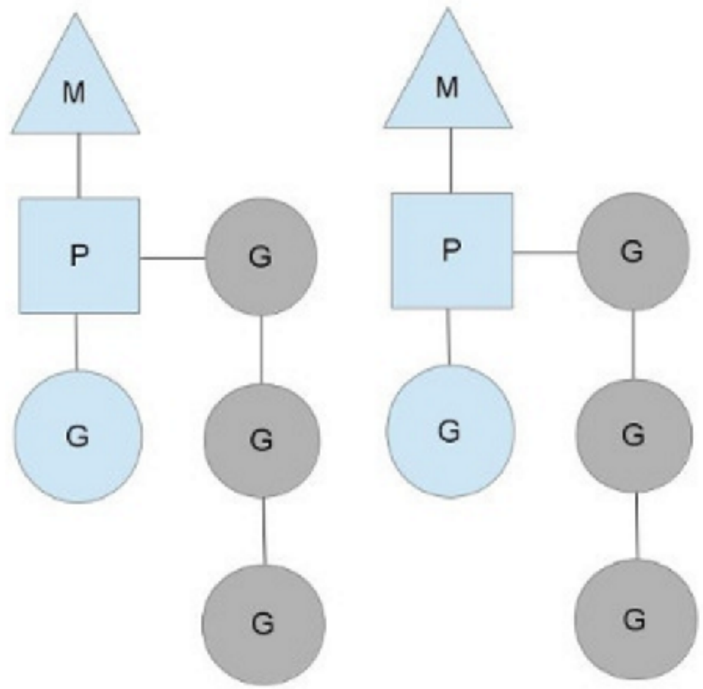
简约版 M / P / G

```go
type m struct {
    curg *g // current running goroutine
    p    *p // attached p for executing go code
}
```

```go
type p struct {
    m    *m // back-link to associated m (nil if idle)
    runq [256]*g // runnable goroutine queue
}
```

```go
type g struct {
    m *m // current m;
}
```

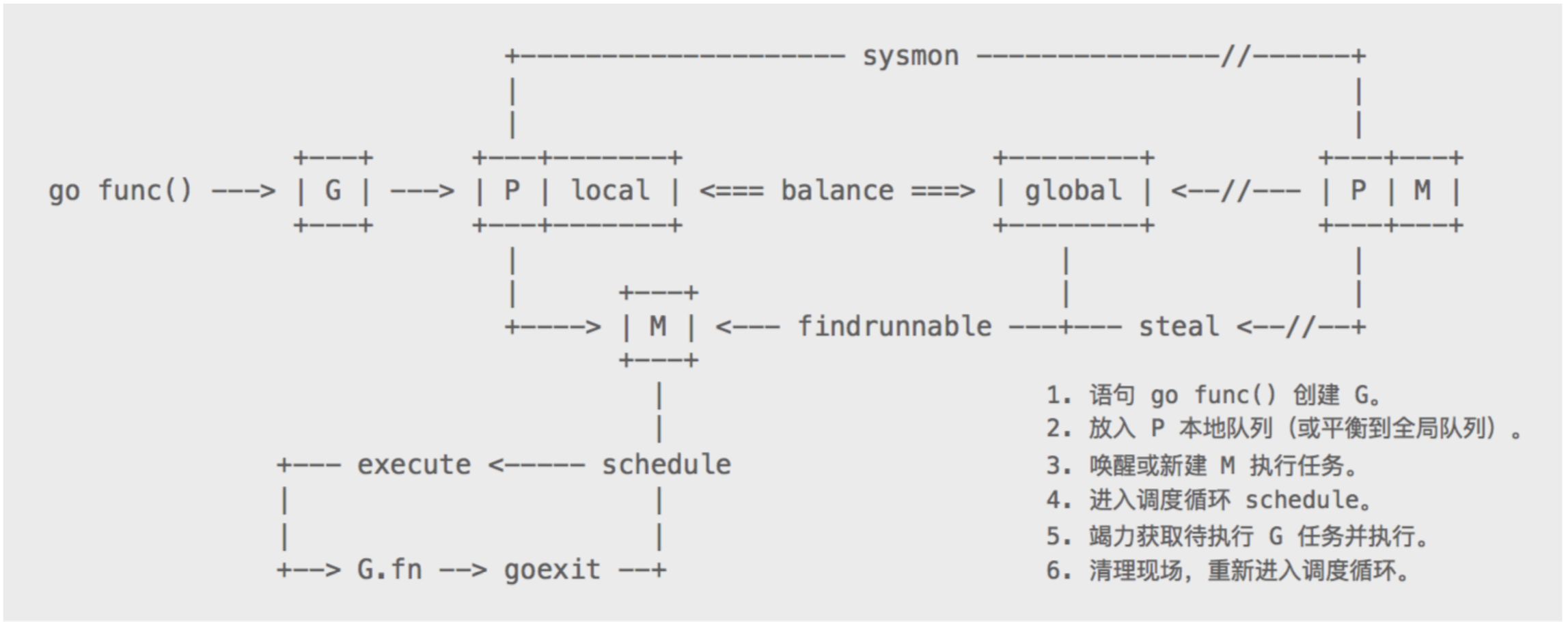
看上去的样子



Ref - [The Go Scheduler]

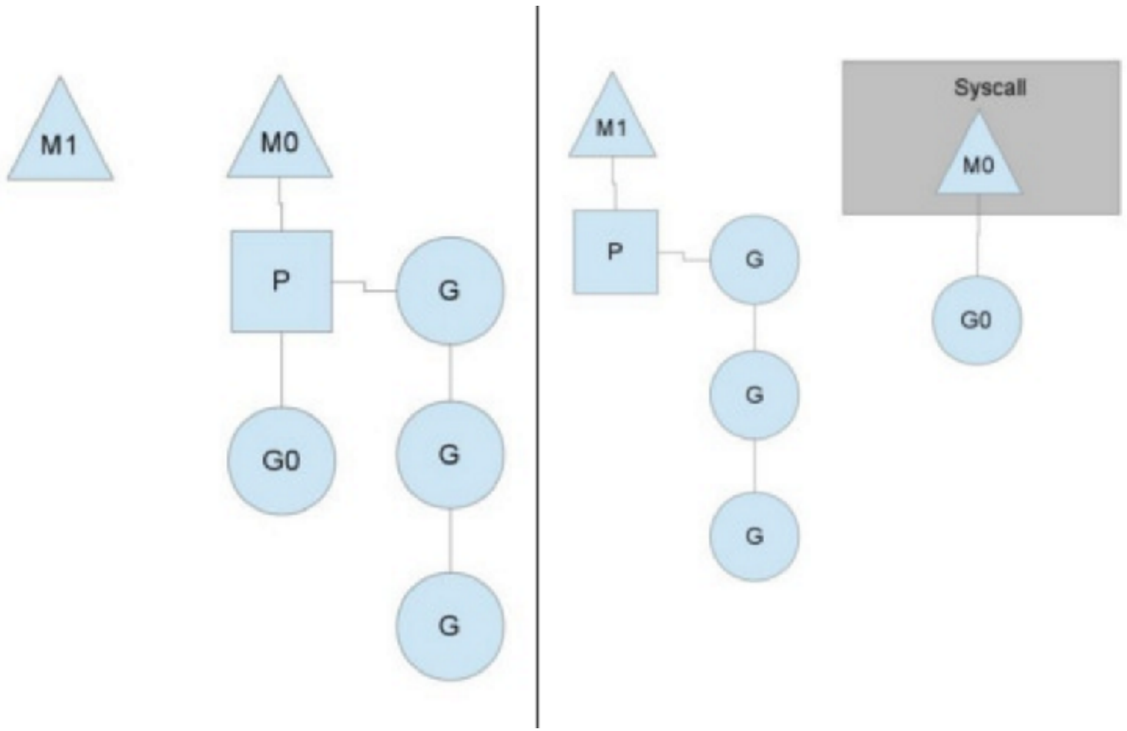
5.3 调度方式 -runtime scheduler

Go 1.5 runtime scheduler

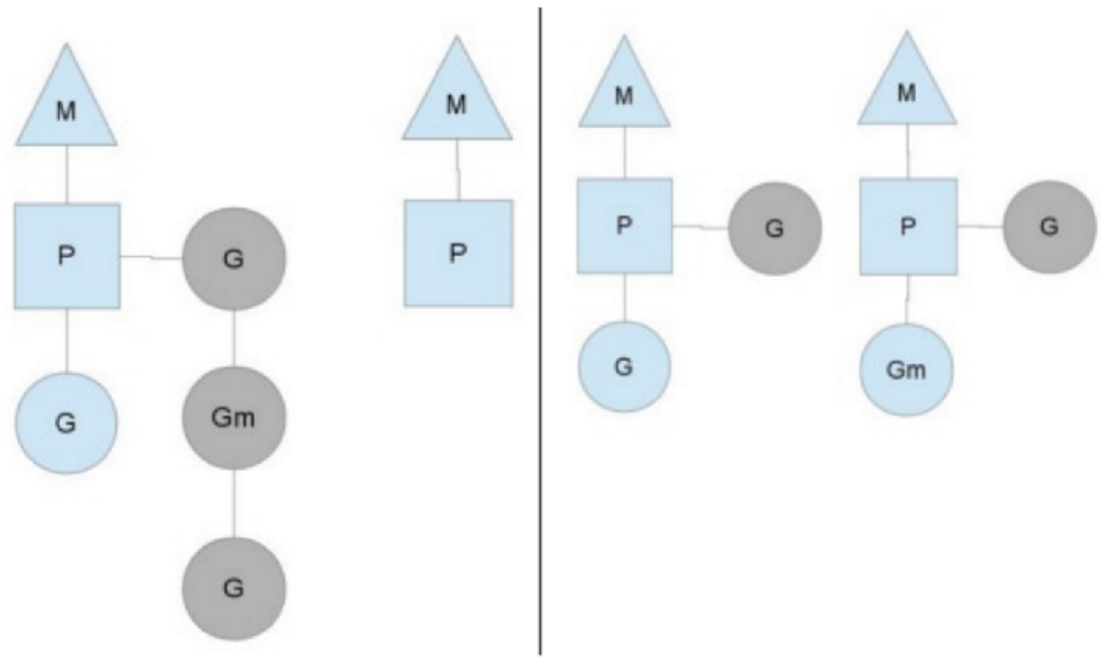


5.3 调度方式 -runtime Scheduler

1. goroutine 中发生系统调用时



2. P 完成本地（G）队列时



e.g 1 : os.Write -> runtime.entersyscall -> stackguard0 <- sysmon(run when program begins)

// sysmon : retake P's blocked in sys calls and preempt long running G's

5.4 调度方式 - 综述

综述

- 协作式

- P 本地 G 队列 / 全局队列

- 复用 M

6. 并发安全

6.1 并发安全 - 官方建议

Advice

Programs that modify data being simultaneously accessed by multiple goroutines must serialize such access.

To serialize access, protect the data with channel operations or other synchronization primitives such as those in the `sync` and `sync/atomic` packages.

If you must read the rest of this document to understand the behavior of your program, you are being too clever.

Don't be clever.

- channel / mutex/ RWMutex / atomic

6.2.1 并发安全 - channel

无缓冲channel

```
counter := 0
//lockc := make(chan bool)
for i := 0; i < 100; i++ {
    go func() {
        //lockc <- true
        counter++
    }()

    //<-lockc
}

fmt.Println("counter: ", counter)
```

6.2.1 并发安全 - mutex

```go
//lock := sync.Mutex{}
counter := 0

for i := 0; i < 100; i++ {
    go func() {
        //lock.Lock()
        counter++
        //lock.Unlock()
    }()
}

time.Sleep(time.Second)
fmt.Println("counter: ", counter)
```

6.2.1 并发安全 - atomic

原子操作

```go
var counter int32 = 0
for i := 0; i < 100; i++ {
    go func() {
        atomic.AddInt32(&counter, 1)
    }()
}

time.Sleep(time.Second)
fmt.Println("counter: ", counter)
```

版本特性

- https://golang.org/doc/go1
- https://golang.org/doc/go1.1
- https://golang.org/doc/go1.2
- https://golang.org/doc/go1.3
- https://golang.org/doc/go1.4
- https://golang.org/doc/go1.5
- https://golang.org/doc/go1.6

参考

- http://go-proverbs.github.io
- http://dave.cheney.net/2013/01/19/what-is-the-zero-value-and-why-is-it-useful
- https://bitbucket.org/golang-china/gopl-zh/wiki/Home
- http://dave.cheney.net/2015/10/09/padding-is-hard
- https://blog.golang.org/slices
- http://blog.csdn.net/pingnanlee/article/details/8351990
- http://www.infoq.com/cn/articles/knowledge-behind-goroutine
- https://golang.org/doc/effective_go.html#allocation_new
- http://concur.rspace.googlecode.com/hg/talk/concur.html
- http://blog.csdn.net/coolmeme/article/details/9997609
- https://blog.cloudflare.com/how-stacks-are-handled-in-go/
- http://man7.org/linux/man-pages/man3/pthread_create.3.html

参考

- http://www.cnblogs.com/luosongchao/p/3680312.html
- https://golang.org/doc/go1.4
- http://www.cs.columbia.edu/~aho/cs6998/reports/12-12-11_DeshpandeSponslerWeiss_GO.pdf
- https://groups.google.com/d/msg/golang-nuts/j51G7ieoKh4/aDoTNvIpXhQJ
- http://www.2cto.com/os/201412/359272.html
- https://groups.google.com/forum/?fromgroups#!searchin/golang-nuts/thread/golang-nuts/2IdA34yR8gQ/nOwLL2FqVIIJ
- http://morsmachine.dk/go-scheduler
- https://groups.google.com/forum/?fromgroups#!searchin/golang-nuts/thread/golang-nuts/2IdA34yR8gQ/nOwLL2FqVIIJ

谢谢

Q & A