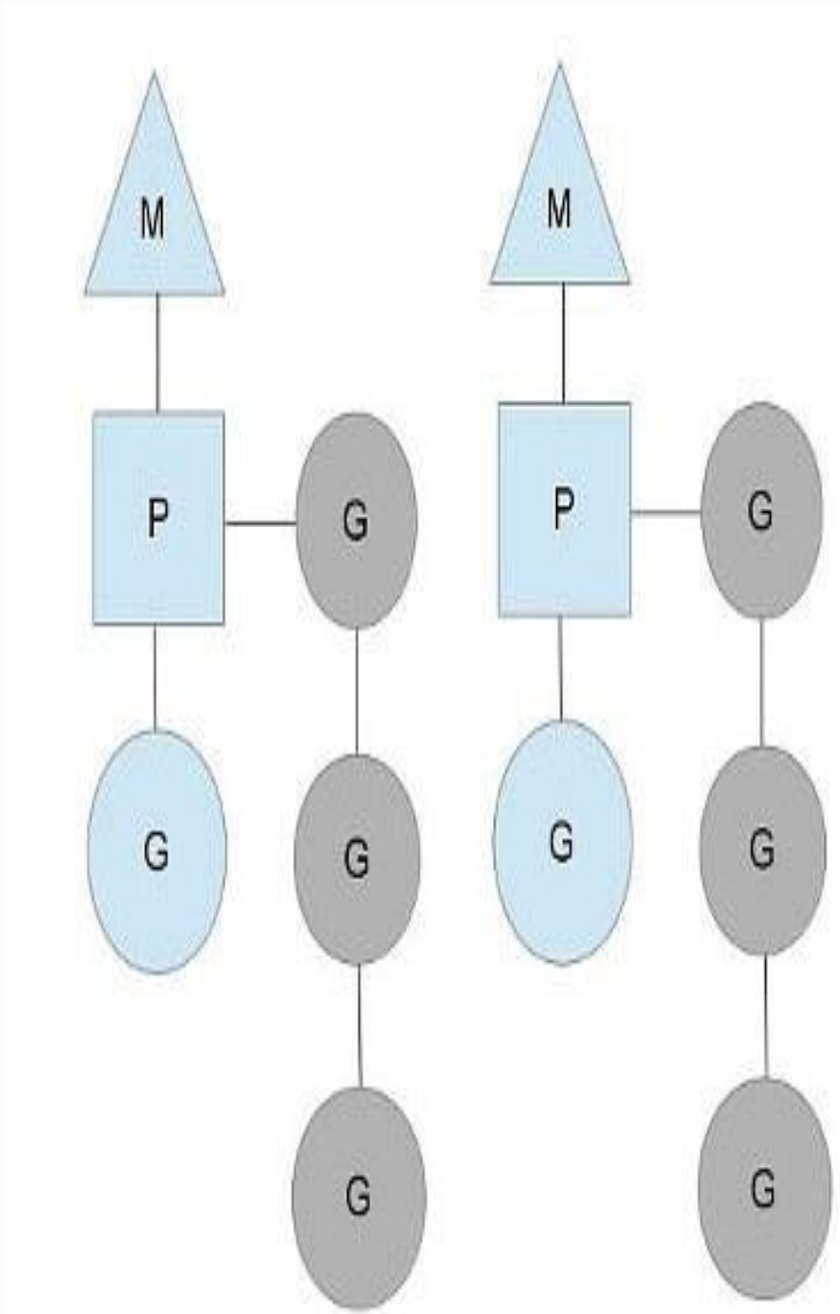
Golang语言进阶

Huayulei_2003@hotmail.com
2018/01/15

目录

理解goroutine：高级协程



- G: 表示goroutine，存储了goroutine的执行stack信息、goroutine状态以及goroutine的任务函数等；另外G对象是可以重用的。
- P: 表示逻辑processor，P的数量决定了系统内最大可并行的G的数量（前提：系统的物理cpu核数>=P的数量）；P的最大作用还是其拥有的各种G对象队列、链表、一些cache和状态。
- M: M代表着真正的执行计算资源。在绑定有效的p后，进入schedule循环；而schedule循环的机制大致是从各种队列、p的本地队列中获取G，切换到G的执行栈上并执行G的函数，调用goexit做清理工作并回到m，如此反复。M并不保留G状态，这是G可以跨M调度的基础。

理解goroutine：调度器(1/2)

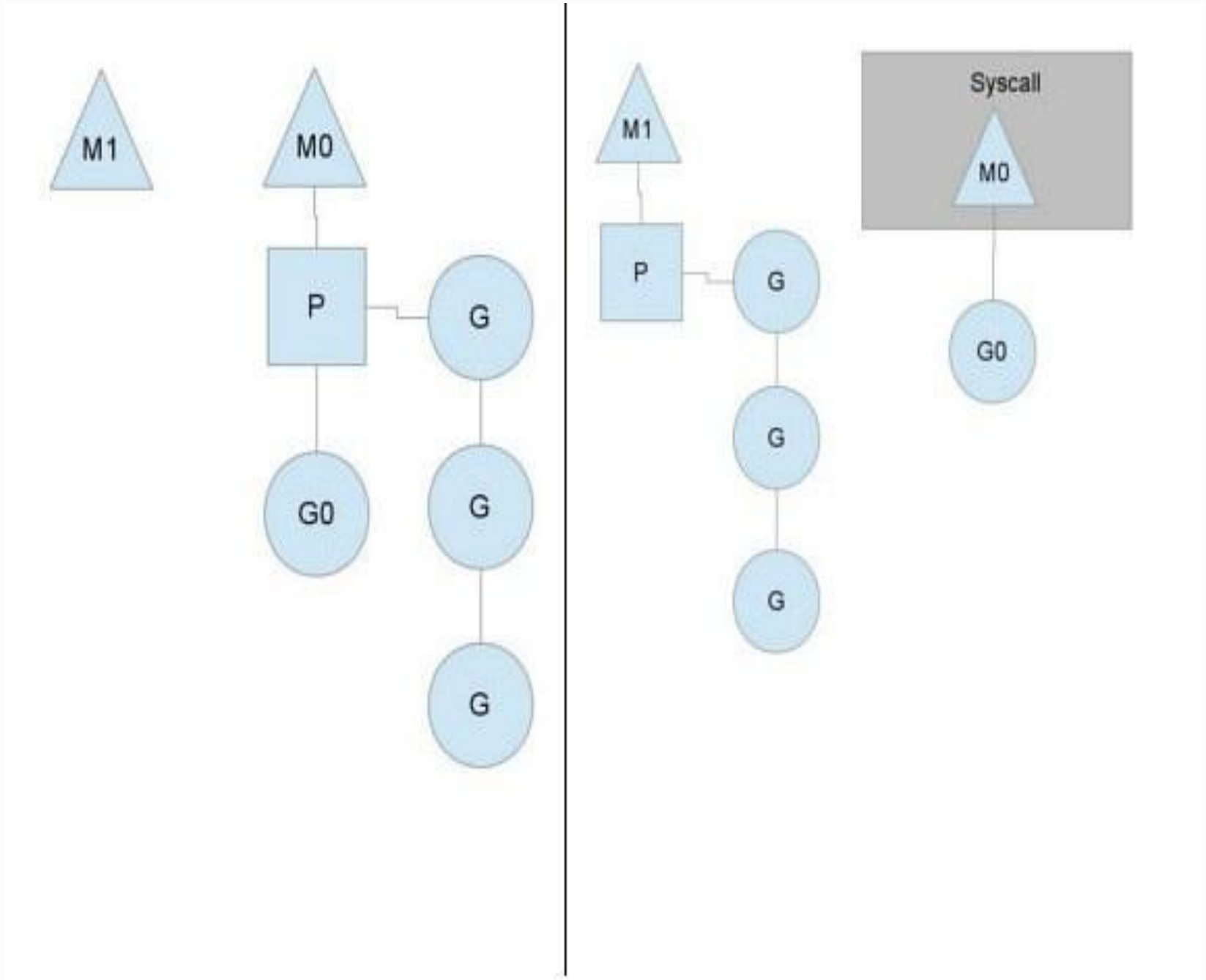
1.抢占式调度的原理是在每个函数或方法的入口，加上一段额外的代码，让runtime有机会检查是否需要执行抢占调度。实现：如果一个G任务运行10ms，sysmon就会认为其运行时间太久而发出抢占式调度的请求。一旦G的抢占标志位被设为true，那么当这个G下一次调用函数或方法时，runtime便可以将G抢占，并移出运行状态，放入P的local runq中，等待下一次被调度

2. Go runtime实现了netpoller，这使得G发起网络I/O操作也不会导致M被阻塞（仅阻塞G），从而不会导致大量M被创建出来。但是对于regular file的I/O操作一旦阻塞，那么M将进入sleep状态，等待I/O返回后被唤醒；这种情况下P将与sleep的M分离，再选择一个idle的M。如果此时没有idle的M，则会新创建一个M，这就是大量I/O操作导致大量Thread被创建的原因。

3. channel阻塞或network I/O情况下的调度：如果G被阻塞在某个channel操作或network I/O操作上时，G会被放置到某个wait队列中，而M会尝试运行下一个runnable的G；如果此时没有runnable的G供m运行，那么m将解绑P，并进入sleep状态。当I/O available或channel操作完成，在wait队列中的G会被唤醒，标记为runnable，放入到某P的队列中，绑定一个M继续执行。

4. system call阻塞情况下的调度:如果G被阻塞在某个system call操作上，那么不光G会阻塞，执行该G的M也会解绑P，与G一起进入阻塞状态。如果此时有idle的M，则P与其绑定继续执行其他G；如果没有idle M，但仍然有其他G要去执行，那么就会创建一个新M。当阻塞在syscall上的G完成syscall调用后，G会去尝试获取一个可用的P，如果没有可用的P，那么G会被标记为runnable，之前的那个sleep的M将再次进入sleep。

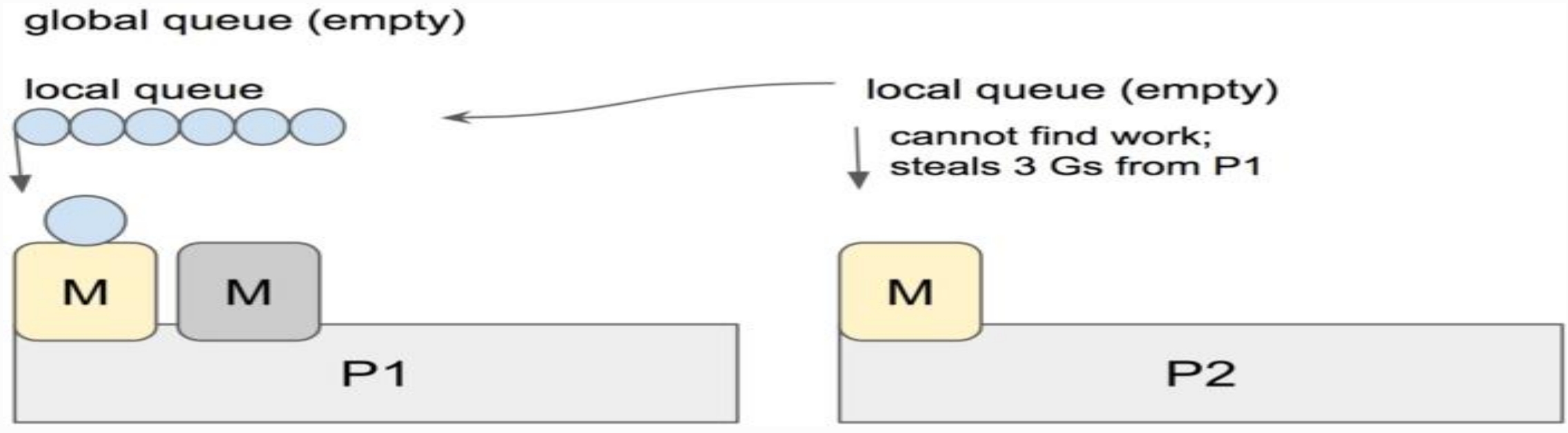
理解goroutine：调度器(2/2)



1.当一个OS线程M0陷入阻塞时，P转而在OS线程M1上运行。调度器保证有足够的线程来运行所有的context P。图中的M1可能是被创建，也可能从线程缓存中取出。

2.当M0返回时，它必须尝试取得一个context P来运行goroutine，一般情况下，它会从其他的OS线程那里steal偷一个过来，如果没有偷到，它就把goroutine放在global runqueue里。然后把自己放到线程缓存中或者转入睡眠状态。

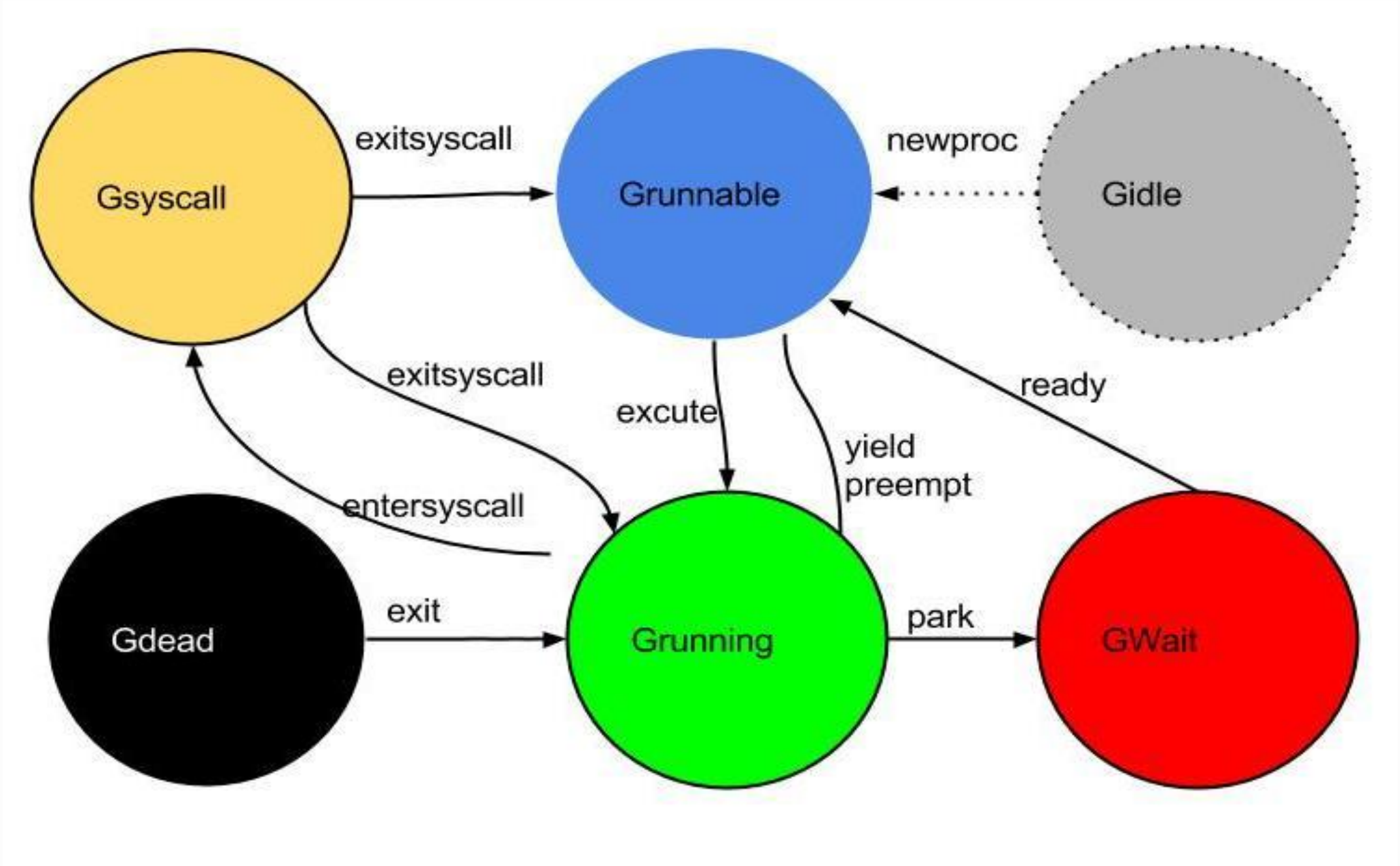
理解goroutine: G的偷取




```
runtime.schedule() {
// only 1/61 of the time, check the global runnable queue for a G.
 // if not found, check the local queue. // if not found,
// try to steal from other Ps.
// if not, check the global runnable queue.
// if not found, poll network.
}
```

在上图中，P2 这个处理器无法找到任何可执行的 goroutines。因此，它随机选择另一个处理器 P1，并将 3 个 goroutines 偷到自己的局部队列中。P2 将执行这些 goroutines，而调度器也将在多个处理器之间更均衡的调度。

理解goroutine：状态迁移



理解Channel：概述

- Channel是goroutine-safe的

hchan 中的 lock mutex

- 可存储、传递值，FIFO(先入先出)

通过 hchan 中的环形缓冲区来实现

- 导致 goroutine 的阻塞和恢复

hchan 中的 sendq和recvq，也就是 sudog 结构的链表队列

调用运行时调度器 (gopark(), goready())

本质是指向hchan 结构的堆内存空间的指针

理解Channel：基础回顾

```
// G1
func main() {
  ...
  for _, task := range tasks {
    ch <- task
  }
  ...
}
// G2
func worker(ch chan Task) {
  for {
    task :=<-ch
    process(task)
  }
}
```

G1 中的 ch <- taskN具体流程：

① 获取锁
② enqueue(taskN)（这里是内存复制taskN）
③ 释放锁

G2 的 t:= <- ch读取数据流程：

① 获取锁
② t = dequeue()（同样，这里也是内存复制）
③ 释放锁

所有通讯的数据都是内存拷贝，核心思想：

Do not communicate by sharing memory;instead, share memory by communicating.

理解Channel：写入满阻塞和恢复

goroutine 被阻塞的具体过程：

当 ch <- task4 执行的时候，channel 中已经满了，需要 pause G1。这个时候，：

① G1 会调用运行时的 gopark，

② 然后 Go 的运行时调度器就会接管

③ 将 G1 的状态设置为 waiting

④ 断开 G1 和 M 之间的关系（switch out），因此 G1 脱离 M，换句话说，M 空闲了，可以安排别的任务了。

⑤ 从 P 的运行队列中，取得一个可运行的 goroutine G

⑥ 建立新的 G 和 M 的关系（Switch in），因此 G 就准备好运行了。

⑦ 当调度器返回的时候，新的 G 就开始运行了，而 G1 则不会运行，也就是 block 了。

对于 goroutine 来说，G1 被阻塞了，新的 G 开始运行了；而对于操作系统线程 M 来说，则根本没有被阻塞。

当 G2 调用 t := <- ch 的时候，channel 的缓冲是满的，而且还有一个 G1 在等候发送队列里，然后 G2 执行下面的操作：

① G2 先执行 dequeue() 从缓冲队列中取得 task1 给 t

② G2 从 sendq 中弹出一个等候发送的 sudog

③ 将弹出的 sudog 中的 elem 的值 enqueue() 到 buf 中

④ 将弹出的 sudog 中的 goroutine，也就是 G1，状态从 waiting 改为 runnable

⑤ 然后，G2 需要通知调度器 G1 已经可以进行调度了，因此调用 goready(G1)。

⑥ 调度器将 G1 的状态改为 runnable

⑦ 调度器将 G1 压入 P 的运行队列，因此在将来的某个时刻调度的时候，G1 就会开始恢复运行。

⑧ 返回到 G2

由 G2 来负责将 G1 的 elem 压入 buf 的，这是一个优化。这样将来 G1 恢复运行后，就不必再次获取锁、enqueue()、释放锁了。这样就避免了多次锁的开销

理解Channel：读取空阻塞和恢复

接收方先阻塞的流程：

如果 G2 先执行了 t := <- ch，此时 buf 是空的，因此 G2 会被阻塞，他的流程是：

① G2 给自己创建一个 sudog 结构变量。其中 g 是自己，也就是 G2，而 elem 则指向 t

② 将这个 sudog 变量压入 recvq 等候接收队列

③ G2 需要告诉 goroutine，自己需要 pause 了，于是调用 gopark(G2)

④ 和之前一样，调度器将其 G2 的状态改为 waiting

⑤ 断开 G2 和 M 的关系

⑥ 从 P 的运行队列中取出其他 goroutine

建立新的 goroutine 和 M 的关系

① 返回，开始继续运行新的 goroutine

当 G1 开始发送数据的流程：

G1 可以将 enqueue(task)，然后调用 goready(G2)。

根据 hchan 结构的状态，已经知道 task 进入 buf 后，G2 恢复运行后，会读取其值，复制到 t 中。G1 不走 buf，G1 直接把数据给 G2。

Goroutine 通常都有自己的栈，互相之间不会访问对方的栈内数据，除了 channel。由于已经知道了 t 的地址（通过 elem指针），而且由于 G2 不在运行，所以可以很安全的直接赋值。当 G2 恢复运行的时候，既不需要再次获取锁，也不需要对 buf 进行操作。从而节约了内存复制、以及锁操作的开销。

理解Channel:无缓冲和select

无缓冲 channel

无缓冲的 channel 行为与前面说的直接发送的一样:

① 接收方阻塞 → 发送方直接写入接收方的栈

② 发送方阻塞 → 接收方直接从发送方的 sudog 中读取

select

① 先把所有需要操作的 channel 上锁

② 给自己创建一个 sudog，然后添加到所有 channel 的 sendq或recvq（取决于是发送还是接收）

③ 把所有的 channel 解锁，然后 pause 当前调用 select 的 goroutine（gopark()）

④ 然后当有任意一个 channel 可用时，select 的这个 goroutine 就会被调度执行。

⑤ resuming mirrors the pause sequence

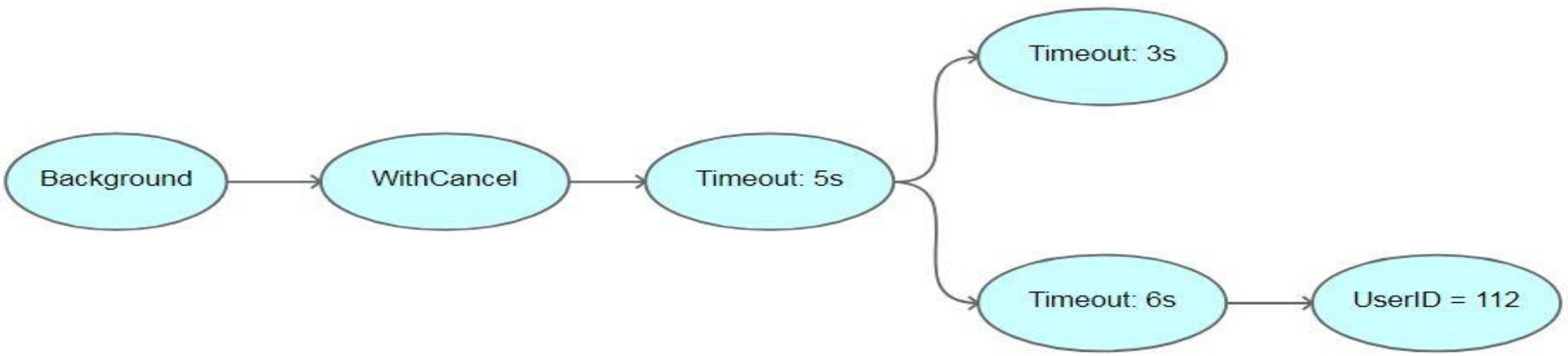
设计上:更倾向于带锁的队列，而不是无锁的实现。

“性能提升不是凭空而来的，是随着复杂度增加而增加的。” - dvyokov

simplicity vs performance 的权衡后的结果

理解Context:调用链

```
func tree() {
  ctx1 := context.Background()
  ctx2, _ := context.WithCancel(ctx1)
  ctx3, _ := context.WithTimeout(ctx2, time.Second * 5)
  ctx4, _ := context.WithTimeout(ctx3, time.Second * 3)
  ctx5, _ := context.WithTimeout(ctx3, time.Second * 6)
  ctx6 := context.WithValue(ctx5, "userID", 12)
}
```



理解Context:概述

context.Context API：

```
type Context interface {
  Deadline() (deadline time.Time, ok bool)
  Done() <-chan struct{}
  Err() error
  Value(key interface{}) interface{}
}
```

两类操作：

3个函数用于限定什么时候子节点退出；

1个函数用于设置请求范畴的变量

什么时候应该使用 Context：

1. 每一个 调用都应该有超时退出的场景
2. 不仅仅是超时，还需要去结束那些不再需要操作的行为
3. 任何函数可能被阻塞，或者需要很长时间来完成的，都应该有个 context.Context

如何创建 Context:

1. 在 调用 开始的时候，使用 context.Background()
2. 请求来了后从创建context, 每个请求都有自己的 context.Background() 。
3. 如果你没有 context，却需要调用一个 context 的函数的话，用 context.TODO()
4. 如果某步操作需要自己的超时设置的话，给它一个独立的 sub-context

理解Context:使用方法

Context的使用:

1. 如果有 Context，将其作为第一个变量。如 func (d* Dialer) DialContext(ctx context.Context, network, address string) (Conn, error)
2. 将其作为可选的方式，用 request 结构体方式。如：func (r *Request) WithContext(ctx context.Context) *Request
3. 不要把 Context 存储到一个 struct 里，除非你使用的是像 http.Request 中的 request 结构体的方式，并且request 结构体应该以 Request 结束为生命终止
4. 要养成关闭 Context 的习惯，特别是超时的Context

   *ctx, cancel := context.WithTimeout(parentCtx, time.Second * 2)*

   defer cancel()

   使用 Timeout 会导致内部使用 time.AfterFunc，从而会导致 context 在计时器到时之前都不会被垃圾回收。在建立之后，应立即 defer cancel() 。
5. 可以用errgroup 类似的实现，使用在以下场景：同时有很多并发请求，并需要集中处理超时、出错终止其它并发任务，如下面例子:

理解Context:errgroup例子

例子:

```
Import   eg “golang.org/x/sync/errgroup”
func DoTwoRequestsAtOnce(ctx context.Context) error {
  eg, egCtx := errgroup.WithContext(ctx)
  var resp1, resp2 *http.Response
  f := func(loc string, respIn **http.Response) func() error {
    return func() error {
      reqCtx, cancel := context.WithTimeout(egCtx, time.Second)
      defer cancel()
      req, _ := http.NewRequest("GET", loc, nil)
      var err error
      *respIn, err = http.DefaultClient.Do(req.WithContext(reqCtx))
      if err == nil && (*respIn).StatusCode >= 500 {
        return errors.New("unexpected!")
      }
      return err
    }
  }
  eg.Go(f("http://localhost:8080/fast_request", &resp1))
  eg.Go(f("http://localhost:8080/slow_request", &resp2))
  return eg.Wait()
}
```

在这个例子中，同时发起了两个调用，当任何一个调用超时或者出错后，会终止另一个调用。这里就是利用前面讲到的 errgroup 来实现的，应对有很多并非请求，并需要集中处理超时、出错终止其它并发任务的时候，这个 pattern 使用起来很方便。

理解Context:Value

例子:

```
package context
type valueCtx struct {
  Context
  key, val interface{}
}
func WithValue(parent Context, key, val interface{})
Context {
  //  ...
  return &valueCtx{parent, key, val}
}
func (c *valueCtx) Value(key interface{}) interface{} {
  if c.key == key {
    return c.val
  }
  return c.Context.Value(key)
}
```

1. WithValue() 实际上就是在 Context 树形结构中，增加一个节点.

2. Context 是 immutable 的,不要试图在 Context.Value 里存某个可变更的值，然后改变，期望别的 Context 可以看到这个改变

3.为了防止树形结构中出现重复的键，建议约束键的空间。比如使用私有类型，然后用 GetXxx() 和 WithXxxx() 来操作私有实体。

4.应该保存 Request 范畴的值:

① 任何关于 Context 自身的都是 Request 范畴的

② 从 Request 数据衍生出来，并且随着 Request 的结束而终结

比如:Request ID, Incoming Request ID

5.应该被用于告知性质的事物，而不是控制性质的事物

6.尽量不要用 Context.Value

Golang编译： 概述

常用的编译命令： *go build [-o output] [-i] [build flags] [packages]*
重要的*build flags*选项：

1. **-p n** ： the number of programs, such as build commands or test binaries, that can be run in parallel. The default is the number of CPUs available.
2. **-buildmode mode** ： build mode to use. (-buildmode=archive ； -buildmode=shared ; -buildmode=exe ; -buildmode=pie ; -buildmode=default)
3. **-compiler name** ： name of compiler to use, as in runtime.Compiler (gccgo or gc).
4. **-gccgoflags ‘[pattern=]arg list’ :** arguments to pass on each gccgo compiler/linker invocation.
5. **-ldflags ’[pattern=]arg list’** : arguments to pass on each go tool link invocation.
6. **-gcflags ‘[pattern=]arg list’ :** arguments to pass on each go tool compile invocation.
7. **-linkshared** : link against shared libraries previously created with -buildmode=shared.
8. **-pkgdir dir :** install and load all packages from dir instead of the usual locations.
9. **-v :** print the names of packages as they are compiled.
10. **-x :** print the commands.
11. **-a** : force rebuilding of packages that are already up-to-date.

Golang编译： Build Mode(1/2)

模式	编译选项	编译示例	备注
exe	-buildmode=exe	go build -a -x -v -buildmode=exe main.go	编译应用程序
pie	-buildmode=pie	go build -a -x -v -buildmode=pie main.go	构建运行地址无关应用程序
default	-buildmode=default	go build -a -x -v -buildmode=default main.go	编译应用程序
archive(Go的静态库)	-buildmode=archive	go build -buildmode=archive test.go	编译package生成.a文件
shared (Go 的动态链接库)	-buildmode=shared		Go 语言实现动态链接库，供go程序用
plugin (Go 的插件)	-buildmode=plugin	go build -buildmode=plugin plug.go go build main.go	Go语言实现动态库和动态调用实现
c-archive (C 的静态链接库)	-buildmode=c-archive	go build -buildmode=c-archive hello.go	构建 C 所支持的静态链接库.a文件
c-shared (C 的动态链接库)	-buildmode=c-shared	go build -buildmode=c-shared -o hello.so hello.go	构建 C 所支持的动态链接库.so文件

Golang编译： Build Mode(2/2)

模式	优点	缺点	备注
exe	全部集成，不需要过多依赖；非常适合容器环境；适用不同 Linux 发行版		
pie	安全性高	包会大一些，性能有0%~1%损耗	
default	全部集成，不需要过多依赖；非常适合容器环境；适用不同 Linux 发行版		
archive(Go的静态库)	不需要过多依赖	文件size相对较大	
shared（Go 的动态链接库）	多个应用程序可以共享动态链接库，可以减少应用程序的size	依赖管理、以及部署发布可能有问题的	一般不使用
plugin（Go 的插件）	方便Go运行时加载；方便热更新	构建和部署相对复杂	
c-archive（C 的静态链接库）	方便Go集成到现有的 C 程序中	跨语言调用可能会带来一些问题；性能会有损耗；调试和跟踪问题复杂	
c-shared（C 的动态链接库）	方便Go集成到现有的 C 程序中；可以运行时动态加载	跨语言调用可能会带来一些问题；性能会有损耗；调试和跟踪问题复杂	

Golang测试：概述

1. 基础测试-自带使用testing包：

- 规则1：test文件以file_test.go命名
- 规则2：函数名以TestXxx命名，使用(t *testing.T)参数,性能测试函数以BenchmarkXxx(*testing.B)命名.
- 规则3：测试框架执行测试函数时调用t.Error、t.Fail等函数认为测试失败。
- 规则4: The go tool will ignore a directory named "testdata", making it available to hold ancillary data needed by the tests.

例子1: go语言常见测试写法

例子2: go使用goroutine做复杂场景测试

例子3: 使用Go 的httptest包，简单创建一个测试的http server

进阶测试: subtest sub-benchmarks ,参考:net/http/httptest/server_test.go

2. 使用gomock测试:

go-mock是专门为go语言开发的mock库，该库使用方式简单，支持自动生成代码 go-mock包地址：github.com/golang/mock

使用方法参考：https://github.com/golang/mock/tree/master/sample

Golang测试：例子1

例子： hex_test.go

```
var encDecTests = []encDecTest{
		{"", []byte{}},
		{"0001020304050607", []byte{0, 1, 2, 3, 4, 5, 6, 7}},… …
}
func TestEncode(t *testing.T) {
	for i, test := range encDecTests {
		dst := make([]byte, EncodedLen(len(test.dec)))
		n := Encode(dst, test.dec)
		if n != len(dst) {
			t.Errorf("#%d: bad return value: got: %d want: %d", i, n, len(dst))
		}
		if string(dst) != test.enc {
			t.Errorf("#%d: got: %#v want: %#v", i, dst, test.enc)
		}
	}
}
```

Golang测试：例子2

例子： sendfile_test.go

```
func TestSendfile(t *testing.T) {
        ln, err := newLocalListener("tcp") ……
        errc := make(chan error, 1)
        go func(ln Listener) {
                // Wait for a connection.
                conn, err := ln.Accept()
                … …
                }()
        }(ln)……
        c, err := Dial("tcp", ln.Addr().String())
        rbytes, err := io.Copy(h, c)
        ……
}
```

1.使用goroutine来处理server端逻辑

2.使用errc channel来传递错误信息

Golang测试：例子3

```
http.HandleFunc("/ping", HealthCheckHandler)
func HealthCheckHandler(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusOK)
    w.Header().Set("Content-Type", "application/json")
    io.WriteString(w, `{ “ping": “pong” }`)
}
```

使用httptest包创建测试的web server, 测试代码:

```
req, err := http.NewRequest("GET", "/ping", nil)
    if err != nil {
        t.Fatal(err)
    }
    res := httptest.NewRecorder()
    HealthCheckHandler(res, req)
    if status := res.Code; status != http.StatusOK {
        t.Errorf("error status code: res %v:%v",
            status, http.StatusOK)
    }
```

Golang测试：辅助package

使用外部包帮助快速写test文件:

建议使用：github.com/stretchr/testify/assert 官方例子：

```
func TestSomething(t *testing.T) {
  // assert equality
  assert.Equal(t, 123, 123, "they should be equal")
  // assert inequality
 assert.NotEqual(t, 123, 456, "they should not be equal")
  // assert for nil (good for errors)
  assert.Nil(t, object)
  // assert for not nil (good when you expect something)
  if assert.NotNil(t, object) {
    // now we know that object isn't nil, we are safe to make
    // further assertions without causing any errors
    assert.Equal(t, "Something", object.Value)
  }
}
```

Golang测试：常用命令总结

常用的命令参数： *go test* ，重要的*flags*选项：

1. **-v ：** Verbose output: log all tests as they are run. Also print all text from Log and Logf calls even if the test succeeds.
2. **-cover ：** Enable coverage analysis.
3. **-run regexp** : Run only those tests and examples matching the regular expression.
4. **-count n** : Run each test and benchmark n times (default 1).
5. **-parallel n** : Allow parallel execution of test functions that call t.Parallel. The value of this flag is the maximum number of tests to run simultaneously;
6. **-bench regexp** : Run only those benchmarks matching a regular expression. To run all benchmarks, use '-bench .' or '-bench=.'.
7. **-coverprofile cover.out** : Write a coverage profile to the file after all tests have passed.
8. **-cpuprofile cpu.out** : Write a CPU profile to the specified file before exiting.
9. **-memprofile mem.out** : Write a memory profile to the file after all tests have passed.
10. **-mutexprofile mutex.out** : Write a mutex contention profile to the specified file
11. **-blockprofile block.out** : Write a goroutine blocking profile to the specified file
12. **-memprofilerate n** : Enable more precise (and expensive) memory profiles by setting runtime.MemProfileRate.

Golang性能调优：

1. GODEBUG:

Go 的 runtime 可以收集程序运行周期内的很多数据。默认都是不启用的，但可手动启用。

- 如果关心垃圾收集，则可以启用 gctrace=1 标志。如：env GODEBUG=gctrace=1 godoc -http=:8080
- 内存分配器跟踪:GODEBUG=allocfreetrace=1
- 启用调度器追踪:GODEBUG=schedtrace=1000 （周期，单位 ms，本次设置是每秒一次）
- 可以随意组合追踪器，如： env GODEBUG=gctrace=1,allocfreetrace=1,schedtrace=1000 godoc -http=:8080
- 详细的调度器追踪，可以这样：GODEBUG = schedtrace = 1000,scheddetail = 1。它将会输出每一个 goroutine、工作线程和处理器的详细信息。详细信息在src/pkg/runtime/proc.c中 。

Golang性能调优：

2. pprof：

Go runtime 中内置了 pprof 的性能分析功能.

- CPU 性能分析:当 CPU 性能分析启用后，Go runtime 会每 10ms 就暂停一下，记录当前运行的 Go routine 的调用堆栈及相关数据.
- 内存性能分析则是在堆(Heap)分配的时候，记录一下调用堆栈。默认情况下，是每 1000 次分配，取样一次，这个数值可以改变。栈(Stack)分配 由于会随时释放，因此不会被内存分析所记录.
- 阻塞性能分析:它所记录的是 goroutine 等待资源所花的时间.阻塞包括：
  ① 发送、接受无缓冲的 channel；
  ② 发送给一个满缓冲的 channel，或者从一个空缓冲的 channel 接收；
  ③ 试图获取已被另一个 go routine 锁定的 sync.Mutex 的锁；
- 性能分析不是没有开销的,记住,一次只分析一种
- 对函数分析,最简单的办法就是使用 testing 包。testing 包内置支持生成 CPU、内存、阻塞的性能分析数据.
- 分析整个应用，则可以使用 runtime/pprof 包,也可以使用: https://github.com/pkg/profile
- http的web服务可使用net/http/pprof分析

Golang性能调优：

3. perf：

对于 Linux，perf 是一个非常好用的工具。由于现在 Go1.7后 已经支持了 Frame Pointer，所以可以和 -toolexec= 配合来对 Go 应用进行性能分析。.

- go build -toolexec="perf stat" cmd/compile/internal/gc
- 用 perf record 来记录一段性能分析数据,如go build -toolexec="perf record -g -o /tmp/p" cmd/compile/internal/gc ； perf report -i /tmp/p

4. 火焰图 (Flame Graph)

- X 轴显示的是在该性能指标分析中所占用的资源量，也就是横向越宽，则意味着在该指标中占用的资源越多，Y 轴则是调用栈的深度。
- 注意：
  ① 左右顺序不重要，X 轴不是按时间顺序发生的，而是按字母顺序排序的
  ② 虽然很好看，但是颜色深浅没关系，这是随机选取的。
- 火焰图可以来自于很多数据源，包括 pprof 和 perf,在提供了 /debug/pprof 的情况下，可以自动进行分析处理生成火焰图。

Q&A

Thank you very much !