1.5.1

Golang 源码剖析

《学习笔记》第五版 下册

前言

我是个安全感匮乏的人，对新鲜事物总会保持一定的警惕。总想知道为什么会这样，为什么会那样，渴望将一切都看得通透，而不仅仅是记住字里行间的规则条理。

知道 Golang 很早，但观望了相当长时间。究其原因，无非是一门新出的语言，自身和相关资源都不成熟，不值得立即投入精力。只是后来屡屡出现的“NextC”让我终究起了一探究竟的欲望，很想知道这个 goroutine 和 coroutine 究竟有什么区别。正好那段时间我在拆解 greenlet 和 lua 的源码，算是相互借鉴。

从 R60 到现在，历经好几年，一直跟着源码去学习。其间有各种故事，倒不值得在此絮叨，只能说欣喜苦恼掺杂，乐在其中罢了。虽说这是本写 Golang 的书，但我依然庆幸自己的 C、ASM 底子不错，让我多了种学习手段，能比多数人了解得更深入些。

尽管这已是本书第五版，但内容几乎全部重写，各种错漏在所难免，希望您能及时指正。

全书共分三册：上册《语言详解》，中册《标准库》（未定），下册《源码剖析》。

联系方式：

微博：weibo.com/qyuhen
开源：github.com/qyuhen/book
邮件：qyuhen@hotmail.com
社区：qyuhen.bearychat.com

雨痕

二〇一五年，冬

更新

2012-01-11　开始学习 Go。
2012-01-15　第一版，基于 R60。
2012-03-29　升级到 1.0。
2012-06-15　升级到 1.0.2。
2013-03-26　升级到 1.1。
2013-12-12　第二版，基于 1.2。
2014-05-22　第三版，基于 1.3。
2014-12-20　第四版，基于 1.4。
2015-06-15　第五版，基于 1.5 RC。
2015-11-01　新版《学习笔记》，升级到 1.5.1。
2015-12-09　下册《源码剖析》截稿。
2015-12-18　下册《源码剖析》校对结束，正式发布。

一. 准备

内容基于 Golang 1.5.1，测试环境 Linux AMD64，不包含 32 位内容。

> 我觉得是时候抛弃 32 位平台了。除了学习，日常开发和架构都不需要这个东西了。而且运行时内部对 32 位的处理看着就别扭。

本书重点剖析 Golang 运行时的内部执行机制，以便能深入了解程序运行期状态，这有助于深入理解语言规则，写出更好的代码，无论是规避 GC 潜在问题，还是为了节约内存，亦或提升运行性能。

> 为便于阅读，相关代码被删减，如有疑问请对照原始文件。如果 Golang 版本不同，示例代码行号可能会存在差异，请以您实际测试输出为准。

本书相关环境：

```
$ go version
go version go1.5.1 linux/amd64

$ lsb_release -d
Description:    Ubuntu 14.04.3 LTS

$ gdb --version
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
```

> 本书示例 go 安装包存放在 /usr/local/go 目录，可能与您的有所不同，不影响测试。

二. 引导

事实上，编译好的可执行文件真正执行入口并非我们所写的 main.main 函数，因为编译器总是会插入一段引导代码，完成诸如命令行参数、运行时初始化等工作，然后才会进入用户逻辑。

要从 src/runtime 目录下的一堆文件中找到真正的入口，其实很容易。随便准备一个编译好的目标文件，比如“Hello, World!”。

test.go

```
package main

func main() {
    println("hello, world!")
}
```

编译，然后用 GDB 查看。

> 建议：尽可能使用命令行编译，而不是某些 IDE 的菜单命令，这有助于我们熟悉各种编译开关参数的具体功能。其次，调试程序时，建议使用 -gcflags "-N -l" 参数关闭编译器代码优化和函数内联，避免断点和单步执行无法准确对应源码行，避免小函数和局部变量被优化掉。

```
$ go build -gcflags "-N -l" -o test test.go
```

> 如果在平台使用交叉编译（Cross Compile），需要设置 GOOS 环境变量。

```
$ gdb test

(gdb) info files
Local exec file:
    Entry point: 0x44dd00

(gdb) b *0x44dd00
Breakpoint 1 at 0x44dd00: file /usr/local/go/src/runtime/rt0_linux_amd64.s, line 8.
```

很简单，找到真正的入口地址，然后利用断点命令就可以轻松找到目标源文件信息。

在 src/runtime 目录下有很多不同平台的入口文件，都由汇编实现。

```
$ ls rt0_*
rt0_android_arm.s     rt0_dragonfly_amd64.s   rt0_linux_amd64.s     ...
rt0_darwin_386.s      rt0_freebsd_386.s       rt0_linux_arm.s       ...
rt0_darwin_amd64.s    rt0_freebsd_amd64.s     rt0_linux_arm64.s     ...
```

用你习惯的代码编辑器打开源文件，跳转到指定行，查看具体内容。

rt0_linux_amd64.s

```
TEXT _rt0_amd64_linux(SB),NOSPLIT,$-8
    LEAQ    8(SP), SI // argv
    MOVQ    0(SP), DI // argc
    MOVQ    $main(SB), AX
    JMP     AX

TEXT main(SB),NOSPLIT,$-8
    MOVQ    $runtime·rt0_go(SB), AX
    JMP     AX
```

用 GDB 设置断点命令看看这个 rt0_go 在哪。

> 注意：源码文件中的“ · ”符号编译后变成正常的“.”。

```
(gdb) b runtime.rt0_go
Breakpoint 2 at 0x44a780: file /usr/local/go/src/runtime/asm_amd64.s, line 12.
```

这段汇编代码就是要找的真正目标，正是它完成了初始化和运行时启动。

asm_amd64.s

```
TEXT runtime·rt0_go(SB),NOSPLIT,$0

    ...

    // 调用初始化函数。
    CALL    runtime·args(SB)
    CALL    runtime·osinit(SB)
    CALL    runtime·schedinit(SB)

    // 创建 main goroutine 用于执行 runtime.main。
    MOVQ    $runtime·mainPC(SB), AX
```

```
    PUSHQ   AX
    PUSHQ   $0
    CALL    runtime·newproc(SB)
    POPQ    AX
    POPQ    AX

    // 让当前线程开始执行 main goroutine。
    CALL    runtime·mstart(SB)

    RET

DATA    runtime·mainPC+0(SB)/8,$runtime·main(SB)
GLOBL   runtime·mainPC(SB),RODATA,$8
```

至此，由汇编针对特定平台实现的引导过程就全部完成。后续内容基本上都是由 Golang 代码实现。

```
(gdb) b runtime.main
Breakpoint 3 at 0x423250: file /usr/local/go/src/runtime/proc.go, line 28.
```

三. 初始化

整个初始化过程相当繁琐，要完成诸如命令行参数整理，环境变量设置，以及内存分配器、垃圾回收器和并发调度器的工作现场准备。

依照前一章找出的线索，先依次看看几个初始化函数的内容。依旧用设置断点命令确定函数所在源文件名和代码行号。

```
(gdb) b runtime.args
Breakpoint 7 at 0x42ebf0: file /usr/local/go/src/runtime/runtime1.go, line 48.

(gdb) b runtime.osinit
Breakpoint 8 at 0x41e9d0: file /usr/local/go/src/runtime/os1_linux.go, line 172.

(gdb) b runtime.schedinit
Breakpoint 9 at 0x424590: file /usr/local/go/src/runtime/proc1.go, line 40.
```

函数 args 整理命令行参数，这个没什么需要深究的。

runtime1.go

```
func args(c int32, v **byte) {
    argc = c
    argv = v
    sysargs(c, v)
}
```

函数 osinit 确定 CPU Core 数量。

os1_linux.go

```
func osinit() {
    ncpu = getproccount()
}
```

最关键的就是 schedinit 这里，几乎我们要关注的所有运行时环境初始化构造都在这里被调用。函数头部的注释列举了启动过程，也就是前一章的内容，不过信息太过简洁了点。

proc1.go

```
// The bootstrap sequence is:
//
```

```
//   call osinit
//   call schedinit
//   make & queue new G
//   call runtime·mstart
func schedinit() {
    // 最大系统线程数量限制，参考标准库 runtime/debug.SetMaxThreads。
    sched.maxmcount = 10000

    // 栈、内存分配器、调度器相关初始化。
    stackinit()
    mallocinit()
    mcommoninit(_g_.m)

    // 处理命令行参数和环境变量。
    goargs()
    goenvs()

    // 处理 GODEBUG、GOTRACEBACK 调试相关的环境变量设置。
    parsedebugvars()

    // 垃圾回收器初始化。
    gcinit()

    // 通过 CPU Core 和 GOMAXPROCS 环境变量确定 P 数量。
    procs := int(ncpu)
    if n := atoi(gogetenv("GOMAXPROCS")); n > 0 {
        if n > _MaxGomaxprocs {
            n = _MaxGomaxprocs
        }
        procs = n
    }

    // 调整 P 数量。
    if procresize(int32(procs)) != nil {
        throw("unknown runnable goroutine during bootstrap")
    }
}
```

内存分配器、垃圾回收器、并发调度器的初始化细节需要涉及很多专属特征，先不去理会，留待后续章节再做详解。

事实上，初始化操作到此并未结束，因为接下来要执行的是 runtime.main，而不是用户逻辑入口函数 main.main。

```
(gdb) b runtime.main
Breakpoint 10 at 0x423250: file /usr/local/go/src/runtime/proc.go, line 28.
```

在这里我们关注的焦点是：包初始化函数 init 的执行。

proc.go

```
// The main goroutine.
func main() {
    // 执行栈最大限制：1 GB on 64-bit, 250 MB on 32-bit.
    if ptrSize == 8 {
        maxstacksize = 1000000000
    } else {
        maxstacksize = 250000000
    }

    ...

    // 启动系统后台监控（定期垃圾回收，以及并发任务调度相关）。
    systemstack(func() {
        newm(sysmon, nil)
    })

    ...

    // 执行 runtime 包内所有初始化函数 init。
    runtime_init()

    ...

    // 启动垃圾回收器后台操作。
    gcenable()

    // 执行所有的用户包（包括标准库）初始化函数 init。
    main_init()

    ...

    // 执行用户逻辑入口 main.main 函数。
    main_main()

    // 执行结束，返回退出状态码。
    exit(0)
}
```

与之相关的就是 runtime_init 和 main_init 这两个函数，它们都是由编译器动态生成。

proc.go

```
//go:linkname runtime_init runtime.init
func runtime_init()

//go:linkname main_init main.init
func main_init()
```

```
//go:linkname main_main main.main
func main_main()
```

注意链接后符号名的变化：runtime_init > runtime.init。

我们准备一个稍微复杂点的示例，看看编译器究竟干了什么。

```
+ <src>
    |
    +- main.go, test.go
    |
    +- <lib>
         |
         +- sum.go
```

lib/sum.go

```
package lib

func init() {
    println("sum.init")
}

func Sum(x ...int) int {
    n := 0
    for _, i := range x {
        n += i
    }

    return n
}
```

test.go

```
package main

import (
    "lib"
)

func init() {
    println("test.init")
}

func test() {
    println(lib.Sum(1, 2, 3))
}
```

main.go

```
package main

import (
    _ "net/http"   // 引入一个标准库里的包。
)

func init() {
    println("main.init.2")
}

func main() {
    test()
}

func init() {
    println("main.init.1")
}
```

编译，执行输出。

```
$ go build -gcflags "-N -l" -o test

$ ./test
sum.init
main.init.2
main.init.1
test.init
6
```

接下来我们用反汇编工具，看看最终动态生成代码的真实面目。

```
$ go tool objdump -s "runtime\.init\b" test

TEXT runtime.init.1(SB) /usr/local/go/src/runtime/alg.go
    alg.go:322      ...

TEXT runtime.init.2(SB) /usr/local/go/src/runtime/mstats.go
    mstats.go:148   ...

TEXT runtime.init.3(SB) /usr/local/go/src/runtime/panic.go
    panic.go:154    ...

TEXT runtime.init.4(SB) /usr/local/go/src/runtime/proc.go
    proc.go:140     ...
```

```
TEXT runtime.init(SB) /usr/local/go/src/runtime/zversion.go
    zversion.go:9   ...
    panic.go:9      ...
    select.go:45    ...

    zversion.go:9   CALL runtime.init.1(SB)
    zversion.go:9   CALL runtime.init.2(SB)
    zversion.go:9   CALL runtime.init.3(SB)
    zversion.go:9   CALL runtime.init.4(SB)
    zversion.go:9   MOVL $0x2, 0x43f436(IP)
    zversion.go:9   ADDQ $0x58, SP
    zversion.go:9   RET
```

命令行工具 go tool objdump 可用来查看实际生成的汇编代码，参数使用正则表达式。当然如果习惯 Intel 格式，那么还是用 GDB 吧。

很显然，runtime 内相关的多个 init 函数被赋予唯一符号名，然后再由 runtime.init 进行统一调用。注意，zversion.go 也是动态生成的。

zversion.go

```
// auto generated by go tool dist

package runtime

const defaultGoroot = `/usr/local/go`
const theVersion = `go1.5.1`
const goexperiment = ``
const stackGuardMultiplier = 1
var buildVersion = theVersion
```

至于 main.init，情况基本一致。区别在于它负责调用非 runtime 包的初始化函数。

```
$ go tool objdump -s "main\.init\b" test

TEXT main.init.1(SB) src/main.go
    main.go:7   ...

TEXT main.init.2(SB) src/main.go
    main.go:15  ...

TEXT main.init.3(SB) src/test.go
    test.go:7   ...

TEXT main.init(SB) src/test.go
    test.go:13  ...
    test.go:13  CALL net/http.init(SB)
```

```
test.go:13  CALL test/lib.init(SB)
test.go:13  CALL main.init.1(SB)
test.go:13  CALL main.init.2(SB)
test.go:13  CALL main.init.3(SB)
test.go:13  MOVL $0x2, 0x48d543(IP)
test.go:13  RET
```

被引用的包，包括 lib 和标准库 net/http 里的 init 函数都被 main.init 调用。

> 虽然从当前版本的编译器角度来说，init 的执行顺序和依赖关系、文件名以及定义顺序有关。但这种次序非常不便于维护和理解，极易造成潜在错误，所以强烈要求让 init 只做该做的事情：局部初始化。

最后需要记住：

- 所有 init 函数都在同一个 goroutine 内执行。
- 所有 init 函数结束后才会执行 main.main 函数。

四. 内存分配

内置运行时的编程语言通常会抛弃传统的内存分配方式，改由自主管理。这样可以完成类似预分配、内存池等操作，以避开系统调用带来的性能问题。当然，还有一个重要原因是为了更好地配合垃圾回收。

1. 概述

在深入内存分配算法细节前，我们需要了解一些基本概念，这有助于建立宏观认识。

基本策略：

1. 每次从操作系统申请一大块内存（比如 1MB），以减少系统调用。
2. 将申请到的大块内存按照特定大小预先切分成小块，构成链表。
3. 为对象分配内存时，只需从大小合适的链表提取一个小块即可。
4. 回收对象内存时，将该小块内存重新归还到原链表，以便复用。
5. 如闲置内存过多，则尝试归还部分内存给操作系统，降低整体开销。

内存分配器只管理内存块，并不关心对象状态。且不会主动回收内存，由垃圾回收器在完成清理操作后，触发内存分配器回收操作。

内存块

分配器将其管理的内存块分为两种：

- **span**: 由多个地址连续的页（page）组成的大块内存。
- **object**: 将 span 按特定大小切分成多个小块，每个小块可存储一个对象。

按照其用途，span 面向内部管理，object 面向对象分配。

分配器按页数来区分不同大小的 span。比如，以页数为单位将 span 存放到管理数组中，需要时就以页数为索引进行查找。当然，span 大小并非固定不变。在获取闲置 span 时，如果没找到大小合适的，那就返回页数更多的，此时会引发裁剪操作，多余部分将构成新

的 span 被放回管理数组。分配器还会尝试将地址相邻的空闲 span 合并，以构建更大的内存块，减少碎片，提供更灵活的分配策略。

malloc.go

```
_PageShift = 13
_PageSize  = 1 << _PageShift   // 8KB
```

mheap.go

```
type mspan struct {
    next     *mspan      // 双向链表。
    prev     *mspan
    start    pageID      // 起始序号 = (address >> _PageShift)
    npages   uintptr     // 页数
    freelist gclinkptr   // 待分配的 object 链表。
}
```

用于存储对象的 object，按 8 字节倍数分为 n 种。比如说，大小为 24 的 object 可用来存储范围在 17 ~ 24 字节的对象。这种方式虽然会造成一些内存浪费，但分配器只需面对有限的几种规格（size class）小块内存，优化了分配和复用管理策略。

分配器会尝试将多个微小对象组合到一个 object 块内，以节约内存。

---

malloc.go

```
_NumSizeClasses = 67
```

分配器初始化时，会构建对照表存储大小和规格的对应关系，包括用来切分的 span 页数。

msize.go

```
// Size classes.  Computed and initialized by InitSizes.
//
// SizeToClass(0 <= n <= MaxSmallSize) returns the size class,
//   1 <= sizeclass < NumSizeClasses, for n.
//   Size class 0 is reserved to mean "not small".
//
// class_to_size[i] = largest size in class i
// class_to_allocnpages[i] = number of pages to allocate when
//   making new objects in class i

var class_to_size [_NumSizeClasses]int32
```

```
var class_to_allocnpages [_NumSizeClasses]int32

var size_to_class8 [1024/8 + 1]int8
var size_to_class128 [(_MaxSmallSize-1024)/128 + 1]int8
```

若对象大小超出特定阈值限制，会被当做大对象（large object）特别对待。

malloc.go

```
_MaxSmallSize = 32 << 10    // 32KB
```

管理组件

优秀的内存分配器必须要在性能和内存利用率之间做到平衡。好在，Golang 的起点很高，直接采用了 tcmalloc 的成熟架构。

malloc.go

```
// Memory allocator, based on tcmalloc.
// http://goog-perftools.sourceforge.net/doc/tcmalloc.html
```

分配器由三种组件组成。

- **cache**: 每个运行期工作线程都会绑定一个 cache，用于无锁 object 分配。
- **central**: 为所有 cache 提供切分好的后备 span 资源。
- **heap**: 管理闲置 span，需要时向操作系统申请新内存。

mheap.go

```
type mheap struct {
    free      [_MaxMHeapList]mspan   // 页数在 127 以内的闲置 span 链表数组。
    freelarge mspan                 // 页数大于 127 (>= 1MB) 大 span 链表。

    // 每个 central 对应一种 sizeclass。
    central [_NumSizeClasses]struct {
            mcentral mcentral
    }
}
```

mcentral.go

```
type mcentral struct {
    sizeclass int32   // 规格。
    nonempty  mspan   // 链表：尚有空闲 object 的 span。
    empty     mspan   // 链表：没有空闲 object，或已被 cache 取走的 span。
}
```

mcache.go

```
type mcache struct {
    alloc [_NumSizeClasses]*mspan   // 以 sizeclass 为索引管理多个用于分配的 span。
}
```

分配流程：

1. 计算待分配对象对应规格（size class）。
2. 从 cache.alloc 数组找到规格相同的 span。
3. 从 span.freelist 链表提取可用 object。
4. 如 span.freelist 为空，从 central 获取新 span。
5. 如 central.nonempty 为空，从 heap.free/freelarge 获取，并切分成 object 链表。
6. 如 heap 没有大小合适的闲置 span，向操作系统申请新内存块。

释放流程：

1. 将标记为可回收 object 交还给所属 span.freelist。
2. 该 span 被放回 central，可供任意 cache 重新获取使用。
3. 如 span 已收回全部 object，则将其交还给 heap，以便重新切分复用。
4. 定期扫描 heap 里长时间闲置的 span，释放其占用内存。

注：以上不包括大对象，它直接从 heap 分配和回收。

作为工作线程私有且不被共享的 cache 是实现高性能无锁分配的核心，而 central 的作用是在多个 cache 间提高 object 利用率，避免内存浪费。

假如 cache1 获取一个 span 后，仅使用了一部分 object，那么剩余空间就可能会被浪费。而回收操作将该 span 交还给 central 后，该 span 完全可以被 cache2、cacheN 获取使用。此时，cache1 已不再持有该 span，完全不会造成问题。

将 span 归还给 heap，是为了在不同规格 object 需求间平衡。

> 某时段某种规格的 object 需求量可能激增，那么当需求过后，大量被切分成该规格的 span 就会被闲置浪费。将归还给 heap，就可被其他需求获取，重新切分。

2. 初始化

因为内存分配器和垃圾回收算法都依赖连续地址，所以在初始化阶段，预先保留了很大的一段虚拟地址空间。

> 注意：保留地址空间，并不会分配内存。

该段空间被划分成三个区域：

> 可分配区域从 Golang 1.4 的 128GB 提高到 512GB。

简单点说，就是用三个数组组成一个高性能内存管理结构。

1. 使用 arena 地址向操作系统申请内存，其大小决定了可分配用户内存上限。
2. 位图 bitmap 为每个对象提供 4bit 标记位，用以保存指针、GC 标记等信息。
3. 创建 span 时，按页填充对应 spans 空间。在回收 object 时，只需将其地址按页对齐后就可找到所属 span。分配器还用此访问相邻 span，做合并操作。

> 任何 arena 区域的地址，只要将其偏移量配以不同步幅和起始位置，就可快速访问与之对应的 spans、bitmap 数据。最关键的是，这三个数组可以按需同步线性扩张，无须预先分配内存。

这些区域相关属性被保存在 heap 里，其中包括递进的分配位置 mapped/used。

mheap.go

```
type mheap struct {
    spans          **mspan
    spans_mapped   uintptr

    bitmap         uintptr
    bitmap_mapped  uintptr

    arena_start    uintptr
    arena_used     uintptr
    arena_end      uintptr
    arena_reserved bool
}
```

初始化工作很简单：

1. 创建对象规格大小对照表。
2. 计算相关区域大小，并尝试从某个指定位置开始保留地址空间。
3. 在 heap 里保存区域信息，包括起始位置和大小。
4. 初始化 heap 其他属性。

malloc.go

```
func mallocinit() {
    // 初始化规格对照表。
    initSizes()

    ...

    // 64 位系统。
    if ptrSize == 8 && (limit == 0 || limit > 1<<30) {
            // 计算相关区域大小。
            arenaSize := round(_MaxMem, _PageSize)
            bitmapSize = arenaSize / (ptrSize * 8 / 4)
            spansSize = arenaSize / _PageSize * ptrSize
            spansSize = round(spansSize, _PageSize)

            // 尝试从 0xc000000000 开始设置保留地址。
            // 如果失败，则尝试 0x1c000000000 ~ 0x7fc000000000。
            for i := 0; i <= 0x7f; i++ {
                    switch {
                    case GOARCH == "arm64" && GOOS == "darwin":
                            p = uintptr(i)<<40 | uintptrMask&(0x0013<<28)
                    case GOARCH == "arm64":
                            p = uintptr(i)<<40 | uintptrMask&(0x0040<<32)
                    default:
                            p = uintptr(i)<<40 | uintptrMask&(0x00c0<<32)
                    }

                    // 计算整个区域大小，并从指定位置开始保留地址空间。
                    pSize = bitmapSize + spansSize + arenaSize + _PageSize
```

```
                    p = uintptr(sysReserve(unsafe.Pointer(p), pSize, &reserved))
                    if p != 0 {
                            break
                    }
            }
    }

    // 按页对齐。
    p1 := round(p, _PageSize)

    // 保存相关属性。
    mheap_.spans = (**mspan)(unsafe.Pointer(p1))
    mheap_.bitmap = p1 + spansSize
    mheap_.arena_start = p1 + (spansSize + bitmapSize)
    mheap_.arena_used = mheap_.arena_start
    mheap_.arena_end = p + pSize
    mheap_.arena_reserved = reserved  // 非指定起始地址，备用地址标记。

    ...

    // 初始化 heap。
    mHeap_Init(&mheap_, spansSize)

    // 为当前线程绑定 cache 对象。
    _g_ := getg()
    _g_.m.mcache = allocmcache()
}
```

区域所指定的起始位置，在不同平台会有一些差异。这个无关紧要，实际上我们关心的是保留地址操作细节。

mem_linux.go

```
func sysReserve(v unsafe.Pointer, n uintptr, reserved *bool) unsafe.Pointer {
    if ptrSize == 8 && uint64(n) > 1<<32 {
            p := mmap_fixed(v, 64<<10, _PROT_NONE, _MAP_ANON|_MAP_PRIVATE, -1, 0)
            if p != v {
                    if uintptr(p) >= 4096 {
                            munmap(p, 64<<10)
                    }
                    return nil
            }
            munmap(p, 64<<10)
            *reserved = false
            return v
    }
}

func mmap_fixed(v unsafe.Pointer, n uintptr, prot, flags, fd int32, offset uint32) ... {
    p := mmap(v, n, prot, flags, fd, offset)
    if p != v && addrspace_free(v, n) {
            if uintptr(p) > 4096 {
```

```
                    munmap(p, n)
            }
            p = mmap(v, n, prot, flags|_MAP_FIXED, fd, offset)
    }
    return p
}
```

对系统编程稍有了解的都知道 mmap 的用途。

函数 mmap 要求操作系统内核创建新的虚拟存储器区域，可指定起始地址和长度。Windows 没有此函数，对应 API 是 VirtualAlloc。

PORT_NONE: 页面无法访问。
MAP_FIXED: 必须使用指定起始地址。

另外，作为内存管理的全局根对象 heap， 其相关属性也必须初始化。

mheap.go

```
func mHeap_Init(h *mheap, spans_size uintptr) {
    // 初始化几个用于管理用途的固定分配器（参见本章后续）。

    // 初始化相关属性。
    for i := range h.free {
            mSpanList_Init(&h.free[i])
            mSpanList_Init(&h.busy[i])
    }

    mSpanList_Init(&h.freelarge)
    mSpanList_Init(&h.busylarge)

    // 创建 central。
    for i := range h.central {
            mCentral_Init(&h.central[i].mcentral, int32(i))
    }

    // 将全局变量 h_spans 指向 heap.spans。
    sp := (*slice)(unsafe.Pointer(&h_spans))
    sp.array = unsafe.Pointer(h.spans)
    sp.len = int(spans_size / ptrSize)
    sp.cap = int(spans_size / ptrSize)
}
```

强烈建议所有程序员都学习一下虚拟存储器的相关知识（推荐《深入理解计算机系统》），很多误解都源自对系统层面的认知匮乏。下面，我们用一个简单示例来澄清有关内存分配的几个常见误解。

test.go

```
package main

import (
    "fmt"
    "os"
    "github.com/shirou/gopsutil/process"
)

var ps *process.Process

// 输出内存状态信息。
func mem(n int) {
    if ps == nil {
        p, err := process.NewProcess(int32(os.Getpid()))
        if err != nil {
            panic(err)
        }

        ps = p
    }

    mem, _ := ps.MemoryInfoEx()
    fmt.Printf("%d. VMS: %d MB, RSS: %d MB\n", n, mem.VMS>>20, mem.RSS>>20)
}

func main() {
    // 1. 初始化结束后的内存状态。
    mem(1)

    // 2. 创建一个 10 * 1MB 数组后的内存状态。
    data := new([10][1024 * 1024]byte)
    mem(2)

    // 3. 填充该数组过程中的内存状态。
    for i := range data {
        for x, n := 0, len(data[i]); x < n; x++ {
            data[i][x] = 1
        }

        mem(3)
    }
}
```

编译后执行：

```
1. VMS:  5 MB, RSS: 1 MB
2. VMS: 15 MB, RSS: 1 MB
3. VMS: 15 MB, RSS: 2 MB
3. VMS: 15 MB, RSS: 3 MB
3. VMS: 15 MB, RSS: 4 MB
3. VMS: 15 MB, RSS: 5 MB
3. VMS: 15 MB, RSS: 6 MB
3. VMS: 15 MB, RSS: 7 MB
3. VMS: 15 MB, RSS: 8 MB
3. VMS: 15 MB, RSS: 9 MB
3. VMS: 15 MB, RSS: 10 MB
3. VMS: 15 MB, RSS: 11 MB
```

按序号对照输出结果：

1. 尽管初始化时预留了 544GB 的虚拟地址空间，但并没有分配内存。
2. 操作系统大多采取机会主义分配策略。申请内存时，仅承诺但不立即分配物理内存。
3. 物理内存分配发生在写操作导致缺页异常调度时，且按页提供。

> 注意：不同操作系统，可能会存在一些差异。

3. 分配

为对象分配内存需区分在栈还是堆上完成。通常情况下，编译器有责任尽可能使用寄存器和栈来存储对象，这有助于提升性能，减少垃圾回收器压力。

但千万不要以为用了 new 函数就一定会分配在堆上，相同的源码也有不同的结果。

test.go

```
package main

import ()

func test() *int {
    x := new(int)
    *x = 0xAABB
    return x
}

func main() {
    println(*test())
```

```
}
```

当编译器禁用内联优化时，所生成代码和我们源码表面预期一致。

```
$ go build -gcflags "-l" -o test test.go     // 关闭内联优化。

$ go tool objdump -s "main\.test" test

TEXT main.test(SB) test.go
    test.go:5   SUBQ $0x10, SP
    test.go:6   LEAQ 0x56d46(IP), BX
    test.go:6   MOVQ BX, 0(SP)
    test.go:6   CALL runtime.newobject(SB)   // 在堆上分配。
    test.go:6   MOVQ 0x8(SP), AX
    test.go:7   MOVQ $0xaabb, 0(AX)
    test.go:8   MOVQ AX, 0x18(SP)
    test.go:8   ADDQ $0x10, SP
    test.go:8   RET
```

但当使用默认参数时，函数 test 会被 main 内联，此时结果就变得不同了。

```
$ go build -o test test.go                  // 默认优化。

$ go tool objdump -s "main\.main" test

TEXT main.main(SB) test.go
    test.go:11  SUBQ $0x18, SP
    test.go:12  MOVQ $0x0, 0x10(SP)
    test.go:12  LEAQ 0x10(SP), BX
    test.go:12  MOVQ $0xaabb, 0(BX)
    test.go:12  MOVQ 0(BX), BP
    test.go:12  MOVQ BP, 0x8(SP)
    test.go:12  CALL runtime.printlock(SB)
    test.go:12  MOVQ 0x8(SP), BX
    test.go:12  MOVQ BX, 0(SP)
    test.go:12  CALL runtime.printint(SB)
    test.go:12  CALL runtime.printnl(SB)
    test.go:12  CALL runtime.printunlock(SB)
    test.go:13  ADDQ $0x18, SP
    test.go:13  RET
```

看不懂汇编没关系，但显然内联优化后的代码没有调用 newobject 在堆上分配内存。

编译器这么做，道理很简单。没有内联时，需要在两个栈帧间传递对象，因此在堆上分配而不是返回一个失效栈帧里的数据。而当内联后，实际上就成了 main 栈帧内的局部变量，无需去堆上操作。

Golang 编译器支持逃逸分析（escape analysis），它会在编译期通过构建调用图来分析局部变量是否会被外部引用，从而决定是否可直接分配在栈上。

编译参数 -gcflags "-m" 可输出编译优化信息，其中包括内联和逃逸分析。

你或许见过“Zero Garbage”这个说法，其目的就是避免在堆上的分配行为，从而减小垃圾回收压力，提升性能。另外，做性能测试时使用 go test -benchmem 参数可以输出堆分配次数统计。

好了，本章要关注的是内存分配器，而非编译器。借着上面这个例子，我们开始深入挖掘 newobject 具体是如何为对象分配内存的。

mcache.go

```
type mcache struct {
    // Allocator cache for tiny objects w/o pointers.
    tiny             unsafe.Pointer
    tinyoffset       uintptr

    alloc [_NumSizeClasses]*mspan
}
```

malloc.go

```
// 内置函数 new 实现。
func newobject(typ *_type) unsafe.Pointer {
    return mallocgc(uintptr(typ.size), typ, flags)
}

func mallocgc(size uintptr, typ *_type, flags uint32) unsafe.Pointer {
    // 当前线程所绑定的 cache。
    c := gomcache()

    // 小对象。
    if size <= maxSmallSize {
            // 无需扫描非指针微小对象（小于16）。
            if flags&flagNoScan != 0 && size < maxTinySize {
                    off := c.tinyoffset

                    // 对齐，调整偏移量。
                    if size&7 == 0 {
                            off = round(off, 8)
```

```
                } else if size&3 == 0 {
                        off = round(off, 4)
                } else if size&1 == 0 {
                        off = round(off, 2)
                }

                // 如果剩余空间足够...
                if off+size <= maxTinySize && c.tiny != nil {
                        // 返回指针，调整偏移量为下次分配做好准备。
                        x = add(c.tiny, off)
                        c.tinyoffset = off + size
                        return x
                }

                // 获取新的 tiny 块。
                // 就是从 sizeclass = 2 的 span.freelist 获取一个 16 字节 object。
                s = c.alloc[tinySizeClass]
                v := s.freelist

                // 如果没有可用 object，那么需要从 central 获取新的 span。
                if v.ptr() == nil {
                        systemstack(func() {
                                mCache_Refill(c, tinySizeClass)
                        })

                        // 重新提取 tiny 块。
                        s = c.alloc[tinySizeClass]
                        v = s.freelist
                }

                // 提取 object 后，调整 span.freelist 链表，增加使用计数。
                s.freelist = v.ptr().next
                s.ref++

                // 初始化（零值）tiny 块。
                x = unsafe.Pointer(v)
                (*[2]uint64)(x)[0] = 0
                (*[2]uint64)(x)[1] = 0

                // 对比新旧两个 tiny 块剩余空间。
                // 新块分配后，其 tinyoffset = size，因此比对偏移量即可。
                if size < c.tinyoffset {
                        // 用新块替换。
                        c.tiny = x
                        c.tinyoffset = size
                }

                // 消费一个新的完整 tiny 块。
                size = maxTinySize
        } else {
                // 普通小对象。

                // 查表，以确定 sizeclass。
                var sizeclass int8
```

```
                    if size <= 1024-8 {
                            sizeclass = size_to_class8[(size+7)>>3]
                    } else {
                            sizeclass = size_to_class128[(size-1024+127)>>7]
                    }
                    size = uintptr(class_to_size[sizeclass])

                    // 从对应规格的 span.freelist 提取 object。
                    s = c.alloc[sizeclass]
                    v := s.freelist

                    // 没有可用 object，从 central 获取新的 span。
                    if v.ptr() == nil {
                            systemstack(func() {
                                    mCache_Refill(c, int32(sizeclass))
                            })

                            // 重新提取 object。
                            s = c.alloc[sizeclass]
                            v = s.freelist
                    }

                    // 调整 span.freelist 链表，增加使用计数。
                    s.freelist = v.ptr().next
                    s.ref++

                    // 清零（变量默认总是初始化为零值）。
                    x = unsafe.Pointer(v)
                    if flags&flagNoZero == 0 {
                            v.ptr().next = 0
                            if size > 2*ptrSize && ((*[2]uintptr)(x))[1] != 0 {
                                    memclr(unsafe.Pointer(v), size)
                            }
                    }
            }
    } else {
            // 大对象直接从 heap 分配 span。
            var s *mspan
            systemstack(func() {
                    s = largeAlloc(size, uint32(flags))
            })

            // span.start 实际由 address >> pageShift 生成。
            x = unsafe.Pointer(uintptr(s.start << pageShift))
            size = uintptr(s.elemsize)
    }

    // 在 bitmap 做标记 ...
    // 检查触发条件，启动垃圾回收 ...

    return x
}
```

整理一下这段代码的基本思路：

- 大对象直接从 heap 获取 span。
- 小对象从 cache.alloc[sizeclass].freelist 获取 object。
- 微小对象组合使用 cache.tiny object。

对微小对象的处理很有意思。首先，它不能是指针，因为多个小对象被组合到一个 object 里，显然无法应对垃圾扫描。其次，它从 span.freelist 获取一个 16 字节的 object，然后利用偏移量来记录下一次分配位置。

> 这里有个小细节，体现了作者的细心。当 tiny 因剩余空间不足而使用新 object 时，会比较新旧两个 tiny object 的剩余空间，而非粗暴地喜新厌旧。

分配算法本身并不复杂，没什么好说的，接下来要关注的自然是资源不足时如何扩张。考虑到大对象分配过程没有 central 这个中间环节，所以先跳 largeAlloc 这个坑。

malloc.go

```
func largeAlloc(size uintptr, flag uint32) *mspan {
    // 计算所需页数。
    npages := size >> _PageShift
    if size&_PageMask != 0 {
            npages++
    }

    // 清理 (sweep) 垃圾 ...

    // 从 heap 获取 span，并重置在 bitmap 里的标记。
    s := mHeap_Alloc(&mheap_, npages, 0, true, flag&_FlagNoZero == 0)
    heapBitsForSpan(s.base()).initSpan(s.layout())

    return s
}
```

先不忙跟过去看 mHeap_Alloc，因为小对象扩张函数 mCache_Refill 最终也会调用它。

mcache.go

```
func mCache_Refill(c *mcache, sizeclass int32) *mspan {
    // 放弃当前正在使用的 span (尚在 central.empty 里) 。
    s := c.alloc[sizeclass]
    if s != &emptymspan {
            s.incache = false  // 取消正在使用标志。
    }
```

```
    // 从 central 获取 span 进行替换。
    s = mCentral_CacheSpan(&mheap_.central[sizeclass].mcentral)
    c.alloc[sizeclass] = s
    return s
}
```

在跳转到 central 之前，先得了解 sweepgen 这个概念。垃圾回收每次都会累加这个类似代龄的计数值，而每个等待处理的 span 也有该属性。

mheap.go

```
type mheap struct {
    sweepgen  uint32   // sweep generation, see comment in mspan
    sweepdone uint32   // all spans are swept
}

type mspan struct {
    // if sweepgen == h->sweepgen - 2, the span needs sweeping
    // if sweepgen == h->sweepgen - 1, the span is currently being swept
    // if sweepgen == h->sweepgen,     the span is swept and ready to use
    // h->sweepgen is incremented by 2 after every GC
    sweepgen    uint32
}
```

在 heap 里闲置的 span 不会被垃圾回收器关注，但 central 里的 span 却有可能正在被清理。所以当 cache 从 central 提取 span 时，该属性值就非常重要。

mcentral.go

```
type mcentral struct {
    nonempty  mspan   // 链表：span 尚有空闲 object 可用。
    empty     mspan   // 链表：span 没有空闲 object 可用，或已被 cache 取走。
}

func mCentral_CacheSpan(c *mcentral) *mspan {
    // 清理 (sweep) 垃圾 ...

    sg := mheap_.sweepgen

retry:
    // 遍历 nonempty 链表。
    for s = c.nonempty.next; s != &c.nonempty; s = s.next {
            // 需要清理的 span。
            if s.sweepgen == sg-2 && cas(&s.sweepgen, sg-2, sg-1) {
                    // 因为要交给 cache 使用，所以转移到 empty 链表。
                    mSpanList_Remove(s)
                    mSpanList_InsertBack(&c.empty, s)
```

```
                        // 垃圾清理。
                        mSpan_Sweep(s, true)
                        goto havespan
                }

                // 忽略正在清理的 span。
                if s.sweepgen == sg-1 {
                        continue
                }

                // 已清理过的 span。
                mSpanList_Remove(s)
                mSpanList_InsertBack(&c.empty, s)

                goto havespan
        }

        // 遍历 empty 链表。
        for s = c.empty.next; s != &c.empty; s = s.next {
                // 需要清理的 span。
                if s.sweepgen == sg-2 && cas(&s.sweepgen, sg-2, sg-1) {
                        mSpanList_Remove(s)
                        mSpanList_InsertBack(&c.empty, s)
                        mSpan_Sweep(s, true)

                        // 清理后有可用 object。
                        if s.freelist.ptr() != nil {
                                goto havespan
                        }

                        // 清理后依然没可用 object，重试。
                        goto retry
                }

                // 忽略正在清理的 span。
                if s.sweepgen == sg-1 {
                        continue
                }

                // 已清理过，且不为空的 span 都被转移到 noempty 链表。
                // 这里剩下的自然都是全空或正在被 cache 使用的，继续循环已没有意义。
                break
        }

        // 如果两个链表里都没 span 可用，扩张。
        s = mCentral_Grow(c)

        // 新 span 将被 cache 使用，所以放到 empty 链表尾部。
        mSpanList_InsertBack(&c.empty, s)

havespan:
        // 设置被 cache 使用标志。
        s.incache = true
```

```
    return s
}
```

可以看出，从 central 里获取 span 时，优先取用已有资源，哪怕是要执行清理操作。只有当现有资源都无法满足时，才去 heap 获取 span，并重新切分成 object 链表。

mcentral.go

```
func mCentral_Grow(c *mcentral) *mspan {
    // 查表获取所需页数。
    npages := uintptr(class_to_allocnpages[c.sizeclass])
    size := uintptr(class_to_size[c.sizeclass])

    // 计算切分 object 数量。
    n := (npages << _PageShift) / size

    // 从 heap 获取 span。
    s := mHeap_Alloc(&mheap_, npages, c.sizeclass, false, true)

    // 切分成 object 链表。
    p := uintptr(s.start << _PageShift)  // 内存地址
    head := gclinkptr(p)
    tail := gclinkptr(p)
    for i := uintptr(1); i < n; i++ {
            p += size
            tail.ptr().next = gclinkptr(p)
            tail = gclinkptr(p)
    }
    tail.ptr().next = 0
    s.freelist = head

    // 重置在 bitmap 里的标记。
    heapBitsForSpan(s.base()).initSpan(s.layout())

    return s
}
```

好了，现在大小对象殊途同归，都到了 mHeap_Alloc 这里。

mheap.go

```
type mheap struct {
    busy      [_MaxMHeapList]mspan   // 链表数组：已分配大对象 span，127 页以内。
    busylarge mspan                 // 链表：已分配超过 127 页大对象 span。
}

func mHeap_Alloc(h *mheap, npage uintptr, sizeclass int32, large bool, needzero bool) *mspan {
    systemstack(func() {
```

```
                s = mHeap_Alloc_m(h, npage, sizeclass, large)
        })

        // 对 span 清零。

        return s
}

func mHeap_Alloc_m(h *mheap, npage uintptr, sizeclass int32, large bool) *mspan {

        // 清理 (sweep) 垃圾 ...

        // 从 heap 获取指定页数的 span。
        s := mHeap_AllocSpanLocked(h, npage)
        if s != nil {
                // 重置 span 状态。
                atomicstore(&s.sweepgen, h.sweepgen)
                s.state = _MSpanInUse
                s.freelist = 0
                s.ref = 0
                s.sizeclass = uint8(sizeclass)
                ...

                // 小对象取用的 span 被存放到 central.empty 链表。
                // 而大对象所取用的 span 则放在 heap.busy 链表。
                if large {
                        // 根据页数来判断将其放到 busy 还是 busylarge 链表。
                        // 数组 free 使用页数作为索引，那么 len(free) 就是最大页数边界。
                        if s.npages < uintptr(len(h.free)) {
                                mSpanList_InsertBack(&h.busy[s.npages], s)
                        } else {
                                mSpanList_InsertBack(&h.busylarge, s)
                        }
                }
        }

        return s
}
```

从 heap 获取 span 的算法核心是找到大小最合适的块。首先从页数相同的链表查找，如没有结果，再从页数更多的链表提取，直至超大块或申请新块。

如返回更大的 span，为避免浪费，会将多余部分切出来重新放回 heap 链表。同时还尝试合并相邻闲置 span 空间，减少碎片。

mheap.go

```
type mheap struct {
        free      [_MaxMHeapList]mspan   // 链表数组：页数 127 以内的闲置 span。
        freelarge mspan                  // 链表：页数大于 127 的闲置 span。
```

```
}

func mHeap_AllocSpanLocked(h *mheap, npage uintptr) *mspan {
    // 先尝试获取指定页数的 span，不行就试页数更多的。
    for i := int(npage); i < len(h.free); i++ {
        // 从链表取 span。
        if !mSpanList_IsEmpty(&h.free[i]) {
            s = h.free[i].next
            goto HaveSpan
        }
    }

    // 再不行，就试页数超过 127 的超大 span。
    s = mHeap_AllocLarge(h, npage)

    // 还没有，就得从操作系统申请新的了。
    if s == nil {
        if !mHeap_Grow(h, npage) {
            return nil
        }

        // 因为每次申请最小 1MB/128Pages，所以被放到 freelarge 链表，再试。
        s = mHeap_AllocLarge(h, npage)
        if s == nil {
            return nil
        }
    }

HaveSpan:
    // 从 free 链表移除。
    mSpanList_Remove(s)

    // 如果该 span 曾被释放物理内存，重新映射补回 ...

    // 如果该 span 页数多于预期 ...
    if s.npages > npage {
        // 创建新 span 用来管理多余的内存。
        t := (*mspan)(fixAlloc_Alloc(&h.spanalloc))
        mSpan_Init(t, s.start+pageID(npage), s.npages-npage)

        // 调整切割后的页数。
        s.npages = npage

        // 将新建 span 放回 heap。
        mHeap_FreeSpanLocked(h, t, false, false, s.unusedsince)
    }

    // 在 spans 填充全部指针。
    p := uintptr(s.start)
    p -= (uintptr(unsafe.Pointer(h.arena_start)) >> _PageShift)
    for n := uintptr(0); n < npage; n++ {
        h_spans[p+n] = s
    }
```

```
    return s
}
```

因为 freelarge 只是一个简单链表，没有页数做索引，也不曾按大小排序，所以只能遍历整个链表，然后选出最小，地址最靠前的块。

mheap.go

```
func mHeap_AllocLarge(h *mheap, npage uintptr) *mspan {
    return bestFit(&h.freelarge, npage, nil)
}

func bestFit(list *mspan, npage uintptr, best *mspan) *mspan {
    for s := list.next; s != list; s = s.next {
           if s.npages < npage {
                  continue
           }
           if best == nil || s.npages < best.npages ||
              (s.npages == best.npages && s.start < best.start) {
                  best = s
           }
    }
    return best
}
```

至于将 span 放回 heap 的 mHeap_FreeSpanLocked 操作，将在内存回收章节再做详述。内存分配阶段，也只剩如何向操作系统申请新内存块。

mheap.go

```
func mHeap_Grow(h *mheap, npage uintptr) bool {
    // 大小总是 64KB 的倍数，最少 1MB。
    npage = round(npage, (64<<10)/_PageSize)
    ask := npage << _PageShift
    if ask < _HeapAllocChunk {
           ask = _HeapAllocChunk
    }

    // 向操作系统申请内存。
    v := mHeap_SysAlloc(h, ask)

    // 创建 span 用来管理刚申请的内存。
    s := (*mspan)(fixAlloc_Alloc(&h.spanalloc))
    mSpan_Init(s, pageID(uintptr(v)>>_PageShift), ask>>_PageShift)

    // 填充在 spans 区域的信息。
    p := uintptr(s.start)
    p -= (uintptr(unsafe.Pointer(h.arena_start)) >> _PageShift)
    for i := p; i < p+s.npages; i++ {
```

```
            h_spans[i] = s
    }

    // 放到 heap 相关链表中。
    mHeap_FreeSpanLocked(h, s, false, true, 0)

    return true
}
```

依然是用 mmap 从指定位置申请内存。最重要的是同步扩张 bitmap 和 spans 区域，以及调整 arena_used 这个位置指示器。

malloc.go

```
func mHeap_SysAlloc(h *mheap, n uintptr) unsafe.Pointer {
    // 不能超出 arena 大小限制。
    if n <= uintptr(h.arena_end)-uintptr(h.arena_used) {
            // 从指定位置申请内存。
            p := h.arena_used
            sysMap((unsafe.Pointer)(p), n, h.arena_reserved, &memstats.heap_sys)

            // 同步扩张 bitmap 和 spans 内存。
            mHeap_MapBits(h, p+n)
            mHeap_MapSpans(h, p+n)

            // 调整下一次申请地址。
            h.arena_used = p + n

            return (unsafe.Pointer)(p)
    }
}
```

mem_linux.go

```
func sysMap(v unsafe.Pointer, n uintptr, reserved bool, sysStat *uint64) {
    if !reserved {
            p := mmap_fixed(v, n, _PROT_READ|_PROT_WRITE, _MAP_ANON|_MAP_PRIVATE, -1, 0)
            return
    }

    p := mmap(v, n, _PROT_READ|_PROT_WRITE, _MAP_ANON|_MAP_FIXED|_MAP_PRIVATE, -1, 0)
}
```

至此，内存分配操作流程正式结束。

4. 回收

内存回收的源头是垃圾清理操作。

之所以说回收而非释放，是因为整个内存分配器的核心是内存复用，不再使用的内存会被放回合适位置，等下次分配时再次使用。只有当空闲内存资源过多时，才会考虑释放。

基于效率考虑，回收操作自然不会直接盯着单个对象，而是以 span 为基本单位。通过比对 bitmap 里的扫描标记，逐步将 object 收归原 span，最终上交 central 或 heap 复用。

清理函数 sweepone 调用 mSpan_Sweep 来引发内存分配器回收流程。

mgcsweep.go

```
func sweepone() uintptr {
    if !mSpan_Sweep(s, false) {
            ...
    }
}

func mSpan_Sweep(s *mspan, preserve bool) bool {
    var head, end gclinkptr

    // 为 span 空闲 object 设置标记，无需再次扫描。
    for link := s.freelist; link.ptr() != nil; link = link.ptr().next {
            heapBitsForAddr(uintptr(link)).setMarkedNonAtomic()
    }

    // 遍历 span，收集未标记的不可达 object（不包括 freelist，它们已被标记）。
    heapBitsSweepSpan(s.base(), size, n, func(p uintptr) {
            if cl == 0 {
                    // 大对象：重置 bitmap，更新 sweepgen。
                    heapBitsForSpan(p).initSpan(s.layout())
                    atomicstore(&s.sweepgen, sweepgen)
                    freeToHeap = true
            } else {
                    // 使用 head、end 构建链表，收集不可达 object。
                    if head.ptr() == nil {
                            head = gclinkptr(p)
                    } else {
                            end.ptr().next = gclinkptr(p)
                    }
                    end = gclinkptr(p)
                    end.ptr().next = gclinkptr(0x0bade5)

                    // 收集计数。
                    nfree++
            }
    })
```

```
    // 回收内存。
    // 小对象：如果没有可回收 object，那么维持原状态，根本无需处理。
    // 大对象：整个 span 就是一个 object，直接交还 heap。
    if nfree > 0 {
            mCentral_FreeSpan(&mheap_.central[cl].mcentral, s, int32(nfree), head, end, ...)
    } else if freeToHeap {
            mHeap_Free(&mheap_, s, 1)
    }
}
```

遍历 span，将收集到的不可达 object 合并到 freelist 链表。如该 span 已收回全部 object，那么就将这块完全自由的内存还给 heap，以便后续复用。

mcentral.go

```
func mCentral_FreeSpan(c *mcentral, s *mspan, n int32, start, end gclinkptr ...) bool {
    // 判断 span 是否为空（没有空闲 object）。
    wasempty := s.freelist.ptr() == nil

    // 将收集到链表合并到 freelist。
    end.ptr().next = s.freelist
    s.freelist = start
    s.ref -= uint16(n)

    // 阻止进一步回收。
    if preserve {
            atomicstore(&s.sweepgen, mheap_.sweepgen)
            return false
    }

    // 将原本为空的 span 转移到 central.nonempty 链表。
    if wasempty {
            mSpanList_Remove(s)
            mSpanList_Insert(&c.nonempty, s)
    }

    // 如果还有 object 被使用，那么终止。
    if s.ref != 0 {
            return false
    }

    // 如果收回全部 object，就从 central 交还给 heap。
    mSpanList_Remove(s)
    heapBitsForSpan(s.base()).initSpan(s.layout())
    mHeap_Free(&mheap_, s, 0)

    return true
}
```

无论是向操作系统申请内存，还是清理回收内存，只要往 heap 里放 span，都会尝试合并左右相邻的闲置 span，以构成更大的自由块。

mheap.go

```
func mHeap_Free(h *mheap, s *mspan, acct int32) {
    systemstack(func() {
        mHeap_FreeSpanLocked(h, s, true, true, 0)
    })
}

func mHeap_FreeSpanLocked(h *mheap, s *mspan, acctinuse, acctidle bool, unusedsince int64) {
    // 从现有链表移除。
    mSpanList_Remove(s)

    // 计算偏移量。
    p := uintptr(s.start)
    p -= uintptr(unsafe.Pointer(h.arena_start)) >> _PageShift

    if p > 0 {
        // 通过 spans 数组访问左侧相邻 span。
        t := h_spans[p-1]

        // 检查合并条件。
        if t != nil && t.state != _MSpanInUse && t.state != _MSpanStack {
            // 合并，更新属性。
            s.start = t.start
            s.npages += t.npages

            // 更新 spans 里的信息。
            p -= t.npages
            h_spans[p] = s

            // 释放原左侧 span 对象。
            mSpanList_Remove(t)
            fixAlloc_Free(&h.spanalloc, (unsafe.Pointer)(t))
        }
    }

    // 检查右侧 span。
    if (p+s.npages)*ptrSize < h.spans_mapped {
        t := h_spans[p+s.npages]
        if t != nil && t.state != _MSpanInUse && t.state != _MSpanStack {
            // 合并右侧 span，更新属性。
            s.npages += t.npages

            // 更新 spans 信息。
            h_spans[p+s.npages-1] = s

            // 释放原右侧 span 对象。
            mSpanList_Remove(t)
            fixAlloc_Free(&h.spanalloc, (unsafe.Pointer)(t))
```

```
            }
    }

    // 根据页数插入 free/freelarge 链表。
    if s.npages < uintptr(len(h.free)) {
            mSpanList_Insert(&h.free[s.npages], s)
    } else {
            mSpanList_Insert(&h.freelarge, s)
    }
}
```

回收操作至此结束。这些被收回的 span 并不会被释放，而是等待复用。

5. 释放

在运行时入口函数 main.main 里，会专门启动一个监控任务 sysmon，它每隔一段时间就会检查 heap 里的闲置内存块。

proc.go

```
func sysmon() {
    scavengelimit := int64(5 * 60 * 1e9)

    for {
            usleep(delay)

            if lastscavenge+scavengelimit/2 < now {
                    mHeap_Scavenge(int32(nscavenge), uint64(now), uint64(scavengelimit))
                    lastscavenge = now
            }
    }
}
```

遍历 free、freelarge 里的所有 span，如闲置时间超过阈值，则释放其关联的物理内存。

mheap.go

```
func mHeap_Scavenge(k int32, now, limit uint64) {
    h := &mheap_

    // 遍历 free 数组里的所有链表。
    for i := 0; i < len(h.free); i++ {
            sumreleased += scavengelist(&h.free[i], now, limit)
    }

    // 遍历 freelarge 链表。
```

```
    sumreleased += scavengelist(&h.freelarge, now, limit)
}

func scavengelist(list *mspan, now, limit uint64) uintptr {
    var sumreleased uintptr

    // 遍历链表。
    for s := list.next; s != list; s = s.next {
        // 检查闲置时间是否超出限制，而且内存没有全部被释放过。
        // 因为存在 span 合并的情况，所以有局部释放很正常。
        if (now-uint64(s.unusedsince)) > limit && s.npreleased != s.npages {
            // 更新释放计数属性。
            released := (s.npages - s.npreleased) << _PageShift
            sumreleased += released
            s.npreleased = s.npages

            // 释放内存。
            sysUnused((unsafe.Pointer)(s.start<<_PageShift), s.npages<<_PageShift)
        }
    }
    return sumreleased
}
```

所谓物理内存释放，另有玄虚。

mem_linux.go

```
func sysUnused(v unsafe.Pointer, n uintptr) {
    madvise(v, n, _MADV_DONTNEED)
}
```

系统调用 madvise 告知操作系统某段内存暂不使用，建议内核收回对应物理内存。当然，这只是一个建议，是否回收由内核决定。如物理内存资源充足，该建议可能会被忽略，以避免无谓的损耗。而当再次使用该内存块时，会引发缺页异常，内核会自动重新关联物理内存页。

分配器面对的是虚拟内存，所以在地址空间充足的情况下，根本无需放弃这段虚拟内存，无需收回 mspan 等管理对象，这也是 arena 能线性扩张的根本原因。

Microsoft Windows 并不支持类似 madvise 机制，须在获取 span 时主动补上被 VirtualFree 掉的内存。

mem_windows.go

```
func sysUnused(v unsafe.Pointer, n uintptr) {
```

```
    r := stdcall3(_VirtualFree, uintptr(v), n, _MEM_DECOMMIT)
}

func sysUsed(v unsafe.Pointer, n uintptr) {
    r := stdcall4(_VirtualAlloc, uintptr(v), n, _MEM_COMMIT, _PAGE_READWRITE)
}
```

mheap.go

```
func mHeap_AllocSpanLocked(h *mheap, npage uintptr) *mspan {
    ...

HaveSpan:
    // 如果被释放过物理内存，重新补上。
    if s.npreleased > 0 {
            sysUsed((unsafe.Pointer)(s.start<<_PageShift), s.npages<<_PageShift)
            s.npreleased = 0
    }

    return s
}
```

多数 Unix-Like 系统都支持 madvise，所以它们的 sysUsed 函数大多什么都不做。
除周期性自动处理外，也可以调用 runtime/debug.FreeOSMemory 函数主动释放。

---

6. 其他

从运行时的角度，整个进程内的对象可分为两类。其一，自然是从 arena 区域分配的用户对象；另一种，则是运行时自身运行和管理所需，比如管理 arena 内存片段的 mspan，提供无锁分配的 mcache 等等。

管理对象的生命周期并不像用户对象那样复杂，且类型和长度都相对固定，所以算法策略显然不用那么复杂。还有，它们相对较长的生命周期也不适合占用 arena 区域，会导致更多碎片化。为此，运行时专门设计了 FixAlloc 固定分配器来为管理对象分配内存。

固定分配器使用相同的算法框架，只相关参数不同。

mfixalloc.go

```
type fixalloc struct {
    size   uintptr            // 固定分配长度。
    first  unsafe.Pointer     // 关联函数。
```

```
    arg    unsafe.Pointer     // 关联函数调用参数。
    list   *mlink             // 复用链表。
    chunk  *byte              // 内存块指针。
    nchunk uint32             // 内存块长度。
    inuse  uintptr            // 内存块已用长度。
}
```

在运行时在初始化 heap 时，一共构建了 4 种固定分配器。

mheap.go

```
func mHeap_Init(h *mheap, spans_size uintptr) {
    fixAlloc_Init(&h.spanalloc, unsafe.Sizeof(mspan{}), recordspan, ...)
    fixAlloc_Init(&h.cachealloc, unsafe.Sizeof(mcache{}), nil, nil, ...)
    fixAlloc_Init(&h.specialfinalizeralloc, unsafe.Sizeof(specialfinalizer{}), nil, ...)
    fixAlloc_Init(&h.specialprofilealloc, unsafe.Sizeof(specialprofile{}), nil, ...)
}
```

mfixalloc.go

```
func fixAlloc_Init(f *fixalloc, size uintptr, first ..., arg unsafe.Pointer, stat *uint64) {
    f.size = size
    f.first = *(*unsafe.Pointer)(unsafe.Pointer(&first))
    f.arg = arg
    f.list = nil
    f.chunk = nil
    f.nchunk = 0
    f.inuse = 0
    f.stat = stat
}
```

分配算法优先从复用链表获取内存，只在获取失败，或剩余空间不足时才获取新内存块。

mfixalloc.go

```
func fixAlloc_Alloc(f *fixalloc) unsafe.Pointer {
    // 尝试从可用链表提取。
    if f.list != nil {
            v := unsafe.Pointer(f.list)
            f.list = f.list.next
            f.inuse += f.size
            return v
    }

    // 如果剩余内存块已不足分配，则获取新内存块（16KB）。
    if uintptr(f.nchunk) < f.size {
            f.chunk = (*uint8)(persistentalloc(_FixAllocChunk, 0, f.stat))
```

```
            f.nchunk = _FixAllocChunk
    }

    // 获取新内存块时执行关联函数（通常用作初始化和拷贝数据）。
    v := (unsafe.Pointer)(f.chunk)
    if f.first != nil {
            fn := *(*func(unsafe.Pointer, unsafe.Pointer))(unsafe.Pointer(&f.first))
            fn(f.arg, v)
    }

    // 更新属性。
    f.chunk = (*byte)(add(unsafe.Pointer(f.chunk), f.size))
    f.nchunk -= uint32(f.size)
    f.inuse += f.size

    return v
}
```

固定分配器持有的这个 16KB 内存块分自 persistent 区域。该区域在很多地方为运行时提供后备内存，目的同样是为了减少并发锁，减少内存申请系统调用。

malloc.go

```
type persistentAlloc struct {
    base unsafe.Pointer
    off  uintptr
}

var globalAlloc struct {
    persistentAlloc
}

func persistentalloc(size, align uintptr, sysStat *uint64) unsafe.Pointer {
    systemstack(func() {
            p = persistentalloc1(size, align, sysStat)
    })
    return p
}

func persistentalloc1(size, align uintptr, sysStat *uint64) unsafe.Pointer {
    const (
            chunk    = 256 << 10
            maxBlock = 64 << 10   // VM reservation granularity is 64K on windows
    )

    // 直接分配大于 64KB 的内存块。
    if size >= maxBlock {
            return sysAlloc(size, sysStat)
    }

    // 后备内存块存放位置（本地或全局）。
    var persistent *persistentAlloc
```

```
    if mp != nil && mp.p != 0 {
            persistent = &mp.p.ptr().palloc
    } else {
            persistent = &globalAlloc.persistentAlloc
    }

    // 偏移位置对齐。
    persistent.off = round(persistent.off, align)

    // 如果后备块空间不足，则重新申请。
    if persistent.off+size > chunk || persistent.base == nil {
            // 申请新 256KB 后备内存。
            persistent.base = sysAlloc(chunk, &memstats.other_sys)
            persistent.off = 0
    }

    // 截取所需内存块。
    p := add(persistent.base, persistent.off)
    persistent.off += size
    return p
}
```

至于释放过程，只简单地放回复用链表。

mfixalloc.go

```
func fixAlloc_Free(f *fixalloc, p unsafe.Pointer) {
    f.inuse -= f.size
    v := (*mlink)(p)
    v.next = f.list
    f.list = v
}
```

recordspan

四个 FixAlloc，只有 mspan 指定了关联函数 recordspan，其作用是按需扩张 h_allspans 存储空间。h_allspans 保存了所有 span 对象指针，供垃圾回收时遍历。

> 内存分配器 spans 区域虽然保存了 page/span 映射关系，但有很多重复，基于效率考虑，并不适合用来作为遍历对象。

mheap.go

```
var h_allspans []*mspan
```

```
func mHeap_Init(h *mheap, spans_size uintptr) {
    fixAlloc_Init(&h.spanalloc, unsafe.Sizeof(mspan{}), recordspan, ...)
}

func recordspan(vh unsafe.Pointer, p unsafe.Pointer) {
    h := (*mheap)(vh)
    s := (*mspan)(p)

    // 如果空间已满 ...
    if len(h_allspans) >= cap(h_allspans) {
            // 计算新容量。
            n := 64 * 1024 / ptrSize
            if n < cap(h_allspans)*3/2 {
                    n = cap(h_allspans) * 3 / 2
            }

            // 申请新内存空间（直接用指针写 slice 内部属性）。
            var new []*mspan
            sp := (*slice)(unsafe.Pointer(&new))
            sp.array = sysAlloc(uintptr(n)*ptrSize, &memstats.other_sys)
            sp.len = len(h_allspans)
            sp.cap = n

            // 如果原空间有数据，则复制后释放。
            if len(h_allspans) > 0 {
                    // 拷贝数据。
                    copy(new, h_allspans)

                    // 释放旧内存块。
                    // 或由 gcSweep -> gcCopySpans 释放。
                    if h.allspans != mheap_.gcspans {
                            sysFree(unsafe.Pointer(h.allspans), ...)
                    }
            }

            // 指向新空间。
            h_allspans = new
            h.allspans = (**mspan)(unsafe.Pointer(sp.array))
    }

    // 注意：
    //      上面的扩张直接用 mmap 在 arena 以外申请空间。
    //      而 append 引发的扩张是在 arena 区域。
    //      基于管理目的的 h_allspans 不适合用 arena 区域。

    h_allspans = append(h_allspans, s)
    h.nspan = uint32(len(h_allspans))
}
```

五. 垃圾回收

垃圾回收器一直是被诟病最多，也是整个运行时中改进最努力的部分。所有变化都是为了缩短 STW 时间，提高程序实时性。

大事记：

- 2014/06, 1.3: 并发清理。
- 2015/08, 1.5: 三色并发标记。

注意：此处所说并发，是指垃圾回收和用户逻辑并发执行。

---

1. 概述

按官方说法，Golang GC 的基本特征是“非分代、非紧缩、写屏障、并发标记清理”。

mgc.go

```
The GC runs concurrently with mutator threads, is type accurate (aka precise), allows multiple
GC thread to run in parallel. It is a concurrent mark and sweep that uses a write barrier. It is
non-generational and non-compacting. Allocation is done using size segregated per P allocation
areas to minimize fragmentation while eliminating locks in the common case.

The algorithm decomposes into several steps.
```

该文件头部有 GC 的详细说明，只有个别地方和源码有些出入，但不影响对算法和过程的理解。

与之前版本在 STW 状态下完成标记不同，并发标记和用户代码同时执行让一切都处于不稳定状态。用户代码随时可能修改已经被扫描过的区域，在标记过程中还会不断分配新对象，这让垃圾回收变得很麻烦。

究竟什么时候启动垃圾回收？过早会严重浪费 CPU 资源，影响用户代码执行性能。而太晚，会导致堆内存恶性膨胀。如何正确平衡这些问题就是个巨大的挑战。

所有问题的核心：抑制堆增长，充分利用 CPU 资源。为此，引入一系列举措：

三色标记和写屏障

这是让标记和用户代码并发的基本保障，基本原理：

- 起初所有对象都是白色。
- 扫描找出所有可达对象，标记为灰色，放入待处理队列。
- 从队列提取灰色对象，将其引用对象标记为灰色放入队列，自身标记为黑色。
- 写屏障监视对象内存修改，重新标色或放回队列。

当完成全部扫描和标记工作后，剩余不是白色就是黑色，分别代表要待回收和活跃对象，清理操作只需将白色对象内存收回即可。

控制器

控制器全程参与并发回收任务，记录相关状态数据，动态调整运行策略，影响并发标记单元的工作模式和数量，平衡 CPU 资源占用。当回收结束时，参与 next_gc 回收阈值设置，调整垃圾回收触发频率。

mgc.go

```
gcController implements the GC pacing controller that determines when to trigger concurrent
garbage collection and how much marking work to do in mutator assists and background marking.

It uses a feedback control algorithm to adjust the memstats.next_gc trigger based on the heap
growth and GC CPU utilization each cycle.

This algorithm optimizes for heap growth to match GOGC and for CPU utilization between assist
and background marking to be 25% of GOMAXPROCS.

The high-level design of this algorithm is documented at https://golang.org/s/go15gcpacing.
```

辅助回收

某些时候，对象分配速度可能远快于后台标记。这会引发一系列恶果，比如堆恶性扩张，甚至让垃圾回收永远无法完成。

此时，让用户代码线程参与后台回收标记就非常有必要。在为对象分配堆内存时，通过相关策略去执行一定限度的回收操作，平衡分配和回收操作，让进程处于良性状态。

2. 初始化

初始化过程非常简单，重点是设置 gcpercent 和 next_gc 阈值。

mgc.go

```
func gcinit() {
     // 并发执行器。
     work.markfor = parforalloc(_MaxGcproc)

     // 设置 GOGC。
     _ = setGCPercent(readgogc())

     // 初始启动阈值（4MB）。
     memstats.next_gc = heapminimum
}

func readgogc() int32 {
     p := gogetenv("GOGC")
     if p == "" {
             return 100
     }
     if p == "off" {
             return -1
     }
     return int32(atoi(p))
}

func setGCPercent(in int32) (out int32) {
     out = gcpercent
     if in < 0 {
             in = -1
     }
     gcpercent = in
     heapminimum = defaultHeapMinimum * uint64(gcpercent) / 100
     return out
}
```

3. 启动

在为对象分配堆内存后，mallocgc 函数会检查垃圾回收触发条件，并依照相关状态启动或参与辅助回收。

malloc.go

```
func mallocgc(size uintptr, typ *_type, flags uint32) unsafe.Pointer {
     ...
```

```
    // 直接分配黑色对象。
    if gcphase == _GCmarktermination || gcBlackenPromptly {
            systemstack(func() {
                    gcmarknewobject_m(uintptr(x), size)
            })
    }

    // 检查垃圾回收触发条件。
    if shouldhelpgc && shouldtriggergc() {
            // 启动并发垃圾回收。
            startGC(gcBackgroundMode, false)
    } else if gcBlackenEnabled != 0 {
            // 辅助参与回收任务。
            gcAssistAlloc(size, shouldhelpgc)
    } else if shouldhelpgc && bggc.working != 0 {
            // 让出资源。
            gp := getg()
            if gp != gp.m.g0 && gp.m.locks == 0 && gp.m.preemptoff == "" {
                    Gosched()
            }
    }
}

func shouldtriggergc() bool {
    return memstats.heap_live >= memstats.next_gc && atomicloaduint(&bggc.working) == 0
}
```

---

heap_live 是活跃对象总量，不包括那些尚未被清理的白色对象。

垃圾回收默认以全并发模式运行，但可用环境变量或参数禁用并发标记和并发清理。GC goroutine 一直循环，直到符合触发条件时被唤醒。

mgc.go

```
func startGC(mode int, forceTrigger bool) {
    // 判断 GODEBUG 环境变量。
    // 1: 禁用并发标记。
    // 2: 禁用并发标记和并发清理。
    if debug.gcstoptheworld == 1 {
            mode = gcForceMode
    } else if debug.gcstoptheworld == 2 {
            mode = gcForceBlockMode
    }

    // 同步阻塞模式。
    if mode != gcBackgroundMode {
            gc(mode)
            return
```

```
    }

    // 检查触发条件。
    if !(forceTrigger || shouldtriggergc()) {
            return
    }

    // 全局变量 bggc 保存 GC 状态。
    // 创建或唤醒 GC goroutine。
    if !bggc.started {
            bggc.working = 1
            bggc.started = true
            go backgroundgc()
    } else if bggc.working == 0 {
            bggc.working = 1

            // 唤醒。
            ready(bggc.g, 0)
    }
}

var bggc struct {
    g       *g     // GC goroutine
    working uint   // 是否正处于工作状态。
    started bool   // 是否已创建。
}

func backgroundgc() {
    bggc.g = getg()
    for {
            gc(gcBackgroundMode)
            bggc.working = 0

            // 休眠，等待再次被唤醒。
            goparkunlock(&bggc.lock, "Concurrent GC wait", traceEvGoBlock, 1)
    }
}
```

经过种种手段的优化调整，整个回收周期，STW 被缩短到有限的几个片段，这让程序实时响应有了很大改善。

新 GC 的表现虽说不上惊艳，但让人相当惊喜。它代表了 Golang 不断进化，以及开发团队追求更好的精神，这让我们对其前景更为看好。不过，当前版本有很多过渡痕迹，甚至代码和文档有对不上的地方。这种情形曾出现在 1.3 里，或许下个版本才是最佳选择。

是否完全去掉 STW？是否能优化写屏障的性能问题？还有很多问题尚待解决。

并发模式（Background Mode）垃圾回收过程示意图：

```
          Phase      STW          Work
 ---------------------+-----------------------------------------------------------
                      |
                  OFF + --------> 准备 MarkWorker/P，使其休眠待命。
                      |
                     stop
                      |
      [WB:1 BE:1] SCAN +
                      |
                     start
                      |
                      + --------> 并发扫描，将灰色对象放入队列。
                      |           对白色对象的引用修改被写屏障捕获。
                      |           Malloc 分配白色对象。
                      |           MarkWorker 被唤醒，开始标记任务。
                      |
                 MARK +
                      |
                      + --------> 等待第一轮标记结束。
                      |           第一轮处理的是并发扫描捕获的灰色对象，不包括新分配白色对象。
                      |
                      + --------> 重新扫描 DATA、BSS 区域。
                      |           扫描新分配白色对象。
                      |
                      + --------> 等待第二轮标记结束。
                      |
                     stop
                      |
                [BE:0] +
                      |
    MARK TERMINATION +
                      |
                      + --------> STW 冻结，完成最终标记。
                      |
           [WB:0] OFF +
                      |
                      + --------> 并发清理。
                      |
                     start
                      |
                                                          STW: StopTheWorld
                                                           WB: WriteBarrierEnabled
                                                           BE: BlackenEnabled
```

整个过程被封装在有些庞大的 gc 函数里。

mgc.go

```
func gc(mode int) {
    // 清理掉意外遗留的 span。
    for gosweepone() != ^uintptr(0) {
        sweep.nbgsweep++
    }

    // 创建 MarkWorker（休眠状态）。
    if mode == gcBackgroundMode {
        gcBgMarkStartWorkers()
    }
```

```
// STW : STOP
systemstack(stopTheWorldWithSema)

// 确保在进入扫描状态前，环境清理干净。
systemstack(finishsweep_m)

// 处理 sync.Pool。
clearpools()

// 重置全局状态变量 work。
gcResetMarkState()

// --- OFF (STW:STOP) ------------------------------------------------

// 并发标记模式。
if mode == gcBackgroundMode {
        // 控制器。
        gcController.startCycle()

        systemstack(func() {
                // 启用写屏障。
                setGCPhase(_GCscan)

                // 初始化相关状态和信号。
                gcBgMarkPrepare()

                // 允许黑色对象标记。
                atomicstore(&gcBlackenEnabled, 1)

                // STW : START
                startTheWorldWithSema()

                // --- SCAN (STW:START) --------------------------------

                // 并发扫描。
                gcscan_m()

                setGCPhase(_GCmark)
        })

        // --- MARK (STW:START) ---------------------------------------

        // 等待 MarkWorker 发回第一轮任务结束信号。
        work.bgMark1.clear()
        work.bgMark1.wait()

        // 第二轮扫描，目标新增白色对象和剩余区段。
        systemstack(func() {
                // DATA、BSS 保存全局变量。
                markroot(nil, _RootData)
                markroot(nil, _RootBss)

                gcBlackenPromptly = true
```

```
                        forEachP(func(_p_ *p) {
                                _p_.gcw.dispose()
                        })
                })

                // 等待 MarkWorker 发回第二轮任务结束信号。
                work.bgMark2.clear()
                work.bgMark2.wait()

                // STW : STOP
                systemstack(stopTheWorldWithSema)

                // 将所有 P.gcw 上交全局队列。
                gcFlushGCWork()

                gcController.endCycle()
        } else {
                // 阻塞模式（mode != gcBackgroundMode）
                gcResetGState()
        }

        // --- MARK TERMINATION (STW:STOP) ------------------------------------

        // 禁用黑色标记操作（MarkWorker 停止工作）。
        atomicstore(&gcBlackenEnabled, 0)
        gcBlackenPromptly = false
        setGCPhase(_GCmarktermination)

        // 完成最终标记工作。
        // 如果是阻塞模式，因为没有前期的扫描和标记操作，那么此处完成全部标记。
        systemstack(func() {
                gcMark(startTime)
        })

        // --- OFF (STW:STOP) -------------------------------------------------

        systemstack(func() {
                // 关闭写屏障。
                setGCPhase(_GCoff)

                // 开启清理操作（并发或阻塞）。
                gcSweep(mode)
        })

        // 全部工作完成。
        // STW : START
        systemstack(startTheWorldWithSema)
}
```

4. 标记

并发标记分为两个步骤：

1. 扫描：遍历相关内存区域，依照指针标记找出灰色可达对象，加入队列。
2. 标记：将灰色对象从队列取出，将其引用对象标记为灰色，自身标记黑色。

扫描

扫描函数 gcscan_m 启动时，用户代码和 MarkWorker 都在运行。

mgcmark.go

```
func gcscan_m() {
    // 重置扫描标志，返回所有 goroutine 数量。
    local_allglen := gcResetGState()

    // 并发执行扫描任务，不过此处仅使用当前线程执行。（避免抢占用户代码和 MarkWorker 资源？）
    // 任务单元包括所有 Root 和 goroutine stack。
    useOneP := uint32(1)
    parforsetup(work.markfor, useOneP, uint32(_RootCount+local_allglen), false, markroot)
    parfordo(work.markfor)
}

const (
    _RootData        = 0
    _RootBss         = 1
    _RootFinalizers  = 2
    _RootSpans       = 3
    _RootFlushCaches = 4
    _RootCount       = 5
)

func gcResetGState() (numgs int) {
    // 初始化所有 goroutine 相关标志。
    // 这些标志对于避免重复扫描很重要。
    for _, gp := range allgs {
            gp.gcscandone = false    // set to true in gcphasework
            gp.gcscanvalid = false   // stack has not been scanned
            gp.gcalloc = 0
            gp.gcscanwork = 0
    }
    numgs = len(allgs)
    return
}
```

parfor 是一个并行任务框架（详见本章第 7 节），其功能就是将任务平分，让多个线程各领一份并发执行。为保证整个任务组能尽快完成，它允许从执行较慢的线程偷取任务。

不过扫描函数仅使用了当前线程，并未启用并发方式执行，似乎后续版本另有变化。扫描目标包括多个 ROOT 区域，还有全部 goroutine 栈。

mgcmark.go

```
func markroot(desc *parfor, i uint32) {
    var gcw gcWork

    switch i {
    case _RootData:
            ...
    case _RootBss:
            ...
    case _RootFinalizers:
            ...
    case _RootSpans:
            ...
    case _RootFlushCaches:
            if gcphase != _GCscan {
                    // 将正在被 cache 使用的所有 span 全部上交 central。
                    // 将缓存在 cache 的 stack 归还给所属 span.freelist。
                    flushallmcaches()
            }

    default:
            // parfor 按顺序为每个任务提供一个 Id，所以访问 allgs 数组时需要去掉 Root。
            gp := allgs[i-_RootCount]

            // 收缩栈空间（此时不能执行用户代码，必须 STW）。
            if gcphase == _GCmarktermination {
                    shrinkstack(gp)
            }

            // 调用 scanstack -> scanblock。
            // scanstack 会设置和检查 gcscanvalid 标志，避免重复扫描。
            scang(gp)
    }

    // 将当前队列上交给全局队列。
    gcw.dispose()
}
```

所有这些扫描过程，最终通过 scanblock 比对 bitmap 区域信息找出合法指针，将其目标当做灰色可达对象添加到待处理队列。

mgcmark.go

```
func scanblock(b0, n0 uintptr, ptrmask *uint8, gcw *gcWork) {
    // 遍历。
    for i := uintptr(0); i < n; {
        bits := uint32(*addb(ptrmask, i/(ptrSize*8)))

        // 没有标记，跳过。
        if bits == 0 {
            i += ptrSize * 8
            continue
        }

        for j := 0; j < 8 && i < n; j++ {
            // 有 bitPointer 标记。
            if bits&1 != 0 {
                // 读取指针内容，目标对象地址。
                obj := *(*uintptr)(unsafe.Pointer(b + i))

                // 确认指针合法。
                if obj != 0 && arena_start <= obj && obj < arena_used {
                    if obj, hbits, span := heapBitsForObject(obj); obj != 0 {
                        // 标记为灰色对象。
                        greyobject(obj, b, i, hbits, span, gcw)
                    }
                }
            }
            bits >>= 1
            i += ptrSize
        }
    }
}

// 将尚未标记的对象标记为灰色，并放入队列。
func greyobject(obj, base, off uintptr, hbits heapBits, span *mspan, gcw *gcWork) {
    if hbits.isMarked() {
        return
    }

    hbits.setMarked()
    gcw.put(obj)
}
```

此处的 gcWork 是专门设计的高性能队列，它允许局部队列和全局队列 work.full/partial 协同工作，平衡任务分配（详见本章第 7 节）。

mgc.go

```
var work struct {
    full    uint64     // lock-free list of full blocks workbuf
    empty   uint64     // lock-free list of empty blocks workbuf
```

```
    partial uint64     // lock-free list of partially filled blocks workbuf
}
```

在 markroot 最后，所有扫描到的灰色对象都被提交给了 work.full 全局队列。

标记

并发标记由多个 MarkWorker goroutine 共同完成，它们在回收任务开始前被绑定到 P，然后进入休眠状态，直到被调度器唤醒。

mgc.go

```
func gcBgMarkStartWorkers() {
    // 为每个 P 绑定一个 Worker。
    for _, p := range &allp {
            if p.gcBgMarkWorker == nil {
                    go gcBgMarkWorker(p)

                    // 暂停，确保该 Worker 绑定到 P 后再继续。
                    notetsleepg(&work.bgMarkReady, -1)
                    noteclear(&work.bgMarkReady)
            }
    }
}
```

调度函数 schedule 从控制器 gcController 获取 MarkWorker goroutine 并执行。

proc1.go

```
func schedule() {
    if gp == nil && gcBlackenEnabled != 0 {
            gp = gcController.findRunnableGCWorker(_g_.m.p.ptr())
    }

    execute(gp, inheritTime)
}
```

控制器方法 findRunnableGCWorker 在返回当前 P 所绑定的 MarkWorker 时，会依据当前运行状态和相关策略设置工作模式，最后还负责将其唤醒。

MarkWorker 工作模式：

- gcMarkWorkerDedicatedMode：全力运行，直到并发标记任务结束。
- gcMarkWorkerFractionalMode：参与标记任务，但可被抢占和调度。
- gcMarkWorkerIdleMode：仅在空闲时参与标记任务。

在了解基本运作流程后，我们去看看标记工作的具体内容。

mgc.go

```
func gcBgMarkWorker(p *p) {
    // 将当前 goroutine 绑定到 P。
    gp := getg()
    p.gcBgMarkWorker = gp

    // 唤醒外层创建循环。
    notewakeup(&work.bgMarkReady)

    for {
            // 休眠，直到被 gcContoller.findRunnable 唤醒。
            gopark(..., "mark worker (idle)", ..., 0)

            // 只能在进入黑化阶段才能运行。
            if gcBlackenEnabled == 0 {
                    throw("gcBgMarkWorker: blackening not enabled")
            }

            decnwait := xadd(&work.nwait, -1)
            done := false

            // 工作模式。
            switch p.gcMarkWorkerMode {
            case gcMarkWorkerDedicatedMode:
                    // 全力工作，直到全部任务结束。
                    gcDrain(&p.gcw, gcBgCreditSlack)

                    done = true
                    if !p.gcw.empty() {
                            throw("gcDrain returned with buffer")
                    }
            case gcMarkWorkerFractionalMode, gcMarkWorkerIdleMode:
                    // 在抢占或无法获取任务时退出。
                    gcDrainUntilPreempt(&p.gcw, gcBgCreditSlack)

                    // 立即上交剩余缓存队列。
                    if gcBlackenPromptly {
                            p.gcw.dispose()
                    }

                    incnwait := xadd(&work.nwait, +1)
                    done = incnwait == work.nproc && work.full == 0 && work.partial == 0
            }
```

```
            // 如果标记任务全部完成，则发送信号。
            if done {
                    // 该标志在接获 bgMark1 后才被设置，确保 bgMark2 在 bgMark1 之后发送。
                    if gcBlackenPromptly {
                            if work.bgMark1.done == 0 {
                                    throw("completing mark 2, but bgMark1.done == 0")
                            }
                            work.bgMark2.complete()
                    } else {
                            work.bgMark1.complete()
                    }
            }
    }
}
```

不同模式的 MarkWorker 对待工作态度完全不同。

mgcmark.go

```
func gcDrain(gcw *gcWork, flushScanCredit int64) {
    for {
            // 如果全局队列已空，且有等待的 Worker，那么分出一部分任务。
            if work.nwait > 0 && work.full == 0 {
                    gcw.balance()
            }

            // 反复尝试从本地或全局队列获取任务，直到所有 Worker 完成任务。
            b := gcw.get()
            if b == 0 {
                    break
            }

            scanobject(b, gcw)
    }
}

func gcDrainUntilPreempt(gcw *gcWork, flushScanCredit int64) {
    gp := getg()

    // 检查抢占标志。
    for !gp.preempt {
            // 只要全局队列为空，就立即分出一部分任务，不关心是否有 Worker 进入等待状态。
            if work.full == 0 && work.partial == 0 {
                    gcw.balance()
            }

            // 尝试从本地或全局获取任务，失败则放弃。不关心其他 Worker 是否完成任务。
            b := gcw.tryGet()
            if b == 0 {
                    break
            }
```

```
            scanobject(b, gcw)
        }
    }
```

处理灰色对象时，无需知道其真实大小，只当做内存分配器提供的 object 块即可。按指针类型长度对齐，配合 bitmap 标记进行遍历，就可找出所有引用成员，将其作为灰色对象压入队列。当然，当前对象自然成为黑色对象，从队列移除。

mgcmark.go

```
func scanobject(b uintptr, gcw *gcWork) {
    hbits := heapBitsForAddr(b)
    s := spanOfUnchecked(b)
    n := s.elemsize

    for i = 0; i < n; i += ptrSize {
            bits := hbits.bits()

            // 标记位检查。
            if i >= 2*ptrSize && bits&bitMarked == 0 {
                    break   // no more pointers in this object
            }
            if bits&bitPointer == 0 {
                    continue   // not a pointer
            }

            // 读取指针内容，成员所引用对象地址。
            obj := *(*uintptr)(unsafe.Pointer(b + i))

            // 确认指针合法。
            if obj != 0 && arena_start <= obj && obj < arena_used && obj-b >= n {
                    // 将引用对象标记为灰色。
                    if obj, hbits, span := heapBitsForObject(obj); obj != 0 {
                            greyobject(obj, b, i, hbits, span, gcw)
                    }
            }
    }
}
```

在 STW 启动后，承担最终收尾工作的 gcMark 有点特殊。如果并发标记被禁用，那么它就需要完成全部的标记任务，回退到 1.4 的阻塞工作模式。

mgc.go

```
func gcMark(start_time int64) {
    // 确保所有任务都上交到全局队列。
    gcFlushGCWork()
```

```
	work.nproc = uint32(gcprocs())

	// 并发执行扫描任务（这次不是单个线程了）。
	// 因为已经 STW，所以这次需要做 flushallmcaches、shrinkstack 操作。
	parforsetup(work.markfor, work.nproc, uint32(_RootCount+allglen), false, markroot)
	if work.nproc > 1 {
		// 重置休眠标志。
		noteclear(&work.alldone)

		// parfor 并发执行的关键。
		helpgc(int32(work.nproc))
	}

	// 当前线程一起参加 mark+drain 任务。
	gchelperstart()
	parfordo(work.markfor)
	var gcw gcWork
	gcDrain(&gcw, -1)
	gcw.dispose()

	// 休眠，等待 gchelper 任务结束后被唤醒。
	if work.nproc > 1 {
		notesleep(&work.alldone)
	}

	// 释放不再使用的 stack 缓存对象。
	freeStackSpans()

	// 更新 cache 状态（被 markroot 处理过）。
	cachestats()

	// 计算下次回收阈值。
	memstats.next_gc = ...(memstats.heap_reachable) * (1 + gcController.triggerRatio))

	// 不能小于最低阈值 4MB。
	if memstats.next_gc < heapminimum {
		memstats.next_gc = heapminimum
	}

	minNextGC := memstats.heap_live + sweepMinHeapDistance*uint64(gcpercent)/100
	if memstats.next_gc < minNextGC {
		memstats.next_gc = minNextGC
	}
}
```

因为有 gcController 决策算法的参与，垃圾回收阈值 next_gc 变得更加灵活。

相比 gcscan_m + MarkWorker，gcMark 显然简单得多，关键问题就是 gchelper 如何执行。

1. 函数 helpgc 唤醒足够数量的线程 M 用于执行 parfordo 任务。
2. 被唤醒 M 检查 helpgc 标志，执行 gchelper 函数完成 mark + drain 任务。

有关 M 执行方式，请参考本书后续《并发调度》相关内容。

---

proc1.go

```
func helpgc(nproc int32) {
	pos := 0

	// 从 1 开始，因为当前线程 (M) 也参加并发任务。
	for n := int32(1); n < nproc; n++ {
		// 跳过当前 M 正在使用的 P。
		if allp[pos].mcache == _g_.m.mcache {
			pos++
		}

		// 获取并设置 M 参数。
		mp := mget()
		mp.helpgc = n       // 关键。
		mp.p.set(allp[pos])
		mp.mcache = allp[pos].mcache

		pos++

		// 唤醒 M 去执行任务。
		notewakeup(&mp.park)
	}
}
```

proc1.go

```
func stopm() {
	_g_ := getg()

retry:
	mput(_g_.m)

	// 休眠。
	notesleep(&_g_.m.park)
	noteclear(&_g_.m.park)

	// 被唤醒后，检查 helpgc 标志。
	if _g_.m.helpgc != 0 {
		// 执行 gchelper 函数。
		gchelper()
	}
}
```

mgc.go

```
func gchelper() {
    _g_ := getg()
    gchelperstart()

    // 执行 mark + drain 任务。
    parfordo(work.markfor)
    if gcphase != _GCscan {
            var gcw gcWork
            gcDrain(&gcw, -1) // blocks in getfull
            gcw.dispose()
    }

    nproc := work.nproc

    // 如果全部任务（注意 -1）完成，那么唤醒 GC 线程。
    if xadd(&work.ndone, +1) == nproc-1 {
            notewakeup(&work.alldone)
    }
}
```

5. 清理

与复杂的标记过程不同，清理操作要简单得多。此时，所有未被标记的白色对象都不再被引用，可简单地将其内存回收。

mgc.go

```
func gcSweep(mode int) {
    // 设置 work.spans = h_allspans。
    // 还记得 FixAlloc 里面的 recordspan 么?
    gcCopySpans()

    // 更新代龄。
    mheap_.sweepgen += 2
    mheap_.sweepdone = 0
    sweep.spanidx = 0

    // 阻塞模式。
    if !_ConcurrentSweep || mode == gcForceBlockMode {
            for sweepone() != ^uintptr(0) {
                    sweep.npausesweep++
            }

            return
    }
```

```
    // 并发模式。
    if sweep.parked {
            sweep.parked = false
            ready(sweep.g, 0)
    }
}
```

并发清理同样由一个专门的 goroutine 完成，它在 runtime.main 调用 gcenable 时被创建。

mgc.go

```
func gcenable() {
    c := make(chan int, 1)
    go bgsweep(c)
    <-c
    memstats.enablegc = true  // now that runtime is initialized, GC is okay
}
```

并发清理本质上就是一个死循环，被唤醒后开始执行清理任务。通过遍历所有 span 对象，触发内存分配器的回收操作。任务完成后，再次休眠，等待下次任务。

mgcsweep.go

```
var sweep sweepdata

// 并发清理状态。
type sweepdata struct {
    g       *g
    parked  bool
}

func bgsweep(c chan int) {
    // 当前 goroutine。
    sweep.g = getg()
    sweep.parked = true

    // 让 gcenable 退出。
    c <- 1

    // 休眠，等待 gcSweep 唤醒。
    goparkunlock(&sweep.lock, "GC sweep wait", traceEvGoBlock, 1)

    for {
            // 循环清理所有 span。
            for gosweepone() != ^uintptr(0) {
                    // 并发调度，避免长时间占用 CPU。
                    Gosched()
            }
```

```
            if !gosweepdone() {
                    continue
            }

            // 清理结束，休眠直到再次被唤醒。
            sweep.parked = true
            goparkunlock(&sweep.lock, "GC sweep wait", traceEvGoBlock, 1)
    }
}

func sweepone() uintptr {
    _g_ := getg()
    sg := mheap_.sweepgen

    for {
            // 从 0 开始的 work.spans (h_allspans) 索引号。
            idx := xadd(&sweep.spanidx, 1) - 1

            // 全部完成。
            if idx >= uint32(len(work.spans)) {
                    mheap_.sweepdone = 1
                    return ^uintptr(0)
            }

            s := work.spans[idx]

            // 跳过闲置的 span，直接更新代龄。
            if s.state != mSpanInUse {
                    s.sweepgen = sg
                    continue
            }

            // 跳过已经或者正在被清理的 span。
            if s.sweepgen != sg-2 || !cas(&s.sweepgen, sg-2, sg-1) {
                    continue
            }

            // 调用内存分配器回收方法。
            npages := s.npages
            if !mSpan_Sweep(s, false) {
                    npages = 0
            }

            return npages
    }
}
```

内存回收操作 mSpan_Sweep，请参考前文《内存分配》相关章节。

6. 监控

尽管有控制器、三色标记等一系列措施，但垃圾回收器依然有问题需要解决。

模拟场景：服务重启，海量客户端重新接入，瞬间分配大量对象，这会将垃圾回收的触发条件 next_gc 推到一个很大值。而当服务正常后，因活跃对象远小于该阈值，造成垃圾回收久久无法触发，服务进程内就会有大量白色对象无法被回收，造成隐性内存泄露。同样情形也可能是因为某个算法在短期内大量使用临时对象造成的。

用示例来模拟一下：

test.go

```
package main

import (
    "fmt"
    "runtime"
    "time"
)

func test() {
    type M [1 << 10]byte
    data := make([]*M, 1024*20)

    // 申请 20MB 内存分配。超出初始阈值，将 next_gc 推高。
    for i := range data {
        data[i] = new(M)
    }

    // 解除引用，预防内联导致 data 生命周期变长。
    for i := range data {
        data[i] = nil
    }
}

func main() {
    test()

    for {
        var ms runtime.MemStats
        runtime.ReadMemStats(&ms)
        fmt.Printf("%s %d MB\n", time.Now().Format("15:04:05"), ms.NextGC>>20)

        time.Sleep(time.Second * 30)
    }
}
```

编译执行：

```
$ go build -gcflags "-l" -o test test.go

$ GODEBUG="gctrace=1" ./test

gc 1 @0.005s 15%: ..., 6->6->6 MB, 4 MB goal, 2 P
gc 2 @0.016s  8%: ..., 8->8->8 MB, 13 MB goal, 2 P
gc 3 @0.022s  9%: ..., 14->14->14 MB, 17 MB goal, 2 P
09:36:01 26 MB
09:36:31 26 MB
09:37:01 26 MB
09:37:31 26 MB
09:38:01 26 MB
GC forced
gc 4 @120.037s 0%: ..., 20->20->0 MB, 29 MB goal, 2 P
scvg0: inuse: 0, idle: 20, sys: 21, released: 0, consumed: 21 (MB)
09:38:31 4 MB
09:39:01 4 MB
```

我们用 test 函数来模拟短期内大量分配对象行为。输出结果表明，在其结束后的相当长时间内都没有触发垃圾回收。直到 forcegc 介入，才将 next_gc 恢复正常。

这就是垃圾回收器最后的一道保险措施。监控服务 sysmon 每隔 2 分钟就会检查一次垃圾回收状态，如超出 2 分钟未曾触发，那就强制执行。

proc1.go

```
func sysmon() {
    // 如果超过 2 分钟未曾做垃圾回收，那么强制执行。
    forcegcperiod := int64(2 * 60 * 1e9)

    for {
            ...

            // 最后一次回收时间。
            lastgc := int64(atomicload64(&memstats.last_gc))

            if lastgc != 0 && unixnow-lastgc > forcegcperiod &&
               atomicload(&forcegc.idle) != 0 && atomicloaduint(&bggc.working) == 0 {
                    // 将 forcegc goroutine 放到待运行队列。
                    injectglist(forcegc.g)
            }
    }
}
```

和前文 bgsweep goroutine 一样，forcegc goroutine 也是死循环、休眠、等待唤醒模式。

proc.go

```
func init() {
    go forcegchelper()
}

func forcegchelper() {
    forcegc.g = getg()
    for {
        atomicstore(&forcegc.idle, 1)

        // 休眠待唤醒。
        goparkunlock(&forcegc.lock, "force gc (idle)", traceEvGoBlock, 1)

        // 参数 forceTrigger = true，让 gc 不检查 next_gc 值，直接执行。
        startGC(gcBackgroundMode, true)
    }
}
```

7. 其他

垃圾回收过程中使用的几种辅助结构。

并行任务框架

parfor 关注的是任务分配和调度，自身不具备执行能力。它将多个任务分组交给多个执行线程，然后在执行过程中重新平衡线程的任务分配，确保整个任务在最短时间内完成。

设置函数 parforsetup 用相关参数初始化 desc 状态，并完成任务分组。

parfor.go

```
func parforsetup(desc *parfor, nthr, n uint32, wait bool, body func(*parfor, uint32)) {
    desc.body = body       // 任务函数。
    desc.nthr = nthr       // 任务线程数量。
    desc.cnt = n           // 任务数量。
    ...

    // 任务分组。
    for i := range desc.thr {
        begin := uint32(uint64(n) * uint64(i) / uint64(nthr))
        end := uint32(uint64(n) * uint64(i+1) / uint64(nthr))
        desc.thr[i].pos = uint64(begin) | uint64(end)<<32
```

```
	}
}
```

最后的循环语句将 n 个任务编号平分成 nthr 份，并将开始和结束位置保存到 pos 的高低位。以 10 任务 5 线程为例，thr[0] 分到的任务单元就是 [0, 2)。

线程须主动调用 parfordo 来获取任务组，执行 body 任务函数。

parfor.go

```
func parfordo(desc *parfor) {
	// 为每个线程分配一个唯一序号。
	tid := xadd(&desc.thrseq, 1) - 1

	// 任务函数。
	body := desc.body

	// 如果只有单个线程，直接按序执行。
	if desc.nthr == 1 {
		for i := uint32(0); i < desc.cnt; i++ {
			body(desc, i)
		}
		return
	}

	// 用线程序号提取任务组。
	me := &desc.thr[tid]
	mypos := &me.pos
	for {
		// 先完成自身任务。
		for {
			// 任务进度：直接累加 pos 低位的起始位置。
			pos := xadd64(mypos, 1)

			// 未超出任务组边界，执行。
			begin := uint32(pos) - 1
			end := uint32(pos >> 32)
			if begin < end {
				body(desc, begin)
				continue
			}
			break
		}

		// 提前完成工作，尝试从其他线程偷取任务。
		idle := false
		for try := uint32(0); ; try++ {
			// 如多次行窃未果，那么准备打卡下班。
			if try > desc.nthr*4 && !idle {
				idle = true
```

```
        xadd(&desc.done, 1)
}

// 如果其他线程都已完成工作，结束。
extra := uint32(0)
if !idle {
        extra = 1
}
if desc.done+extra == desc.nthr {
        if !idle {
                xadd(&desc.done, 1)
        }
        goto exit
}

// 随机挑选一个线程。
var begin, end uint32
victim := fastrand1() % (desc.nthr - 1)
if victim >= tid {
        victim++
}
victimpos := &desc.thr[victim].pos
for {
        // 检查目标线程的当前任务进度。
        pos := atomicload64(victimpos)
        begin = uint32(pos)
        end = uint32(pos >> 32)
        if begin+1 >= end {
                end = 0
                begin = end
                break
        }

        // 有任务可偷，要忙起来了。
        if idle {
                xadd(&desc.done, -1)
                idle = false
        }

        // 将剩余任务偷一半（后半截）。
        begin2 := begin + (end-begin)/2

        // 记得修改原主的任务结束值。
        newpos := uint64(begin) | uint64(begin2)<<32
        if cas64(victimpos, pos, newpos) {
                begin = begin2
                break
        }
}

// 将偷来的任务编号保存到自己 pos 里。
if begin < end {
        atomicstore64(mypos, uint64(begin)|uint64(end)<<32)
        me.nsteal++
        me.nstealcnt += uint64(end) - uint64(begin)
```

```
                        // 跳出偷窃循环，进入外层循环重新执行自己的任务（尽管是偷来的）。
                        break
                }

                // 没偷到任务，就暂停或退出 ...
            }
        }
exit: ...
}
```

缓存队列

gcWork 被设计用来保存灰色对象，必须在保证并发安全的前提下，拥有足够高的性能。

mgcwork.go

```
type gcWork struct {
    wbuf  wbufptr
}
```

该结构的真正核心是 workbuf，gcWork 不过是外层包装。workbuf 作为无锁栈节点，其自身就是一个缓存容器（数组成员）。

mgcwork.go

```
type workbufhdr struct {
    node  lfnode
    nobj  int
}

type workbuf struct {
    workbufhdr
    obj [(_WorkbufSize - unsafe.Sizeof(workbufhdr{})) / ptrSize]uintptr
}
```

透过 gcWork 相关方法，我们可以观察 workbuf 是如何工作的。

mgc.go

```
var work struct {
    full    uint64    // lock-free list of full blocks workbuf
    empty   uint64    // lock-free list of empty blocks workbuf
    partial uint64    // lock-free list of partially filled blocks workbuf
```

```
}
```

mgcwork.go

```
func (ww *gcWork) put(obj uintptr) {
    w := (*gcWork)(noescape(unsafe.Pointer(ww)))

    // 从 work.empty 获取一个 workbuf 复用。
    wbuf := w.wbuf.ptr()
    if wbuf == nil {
        wbuf = getpartialorempty(42)
        w.wbuf = wbufptrOf(wbuf)
    }

    // 直接将 obj 保存在 workbuf.obj 数组。
    wbuf.obj[wbuf.nobj] = obj
    wbuf.nobj++

    // 如果数组填满，则将该数组移交给 work.full。
    // 本地 obj = nil，下次 put 时获取一个复用对象填充。
    if wbuf.nobj == len(wbuf.obj) {
        putfull(wbuf, 50)
        w.wbuf = 0
    }
}

func putfull(b *workbuf, entry int) {
    lfstackpush(&work.full, &b.node)
}
```

这种做法有点像内存分配器的 cache，优先操作本地缓存，直到满足某个阈值再与全局交换。如此，可以保证性能，避免直接操作全局队列。另一方面，从全局获取任务时，总是能一次性拿到一组。

就算是无锁数据结构，使用原子操作也会有性能损耗，尤其是在多核环境下。

这段代码，包括 work 全局变量有很多 C 的影子，看上去有些别扭，完全不是 Golang 的风格。如果是自动代码转换，那么下个版本是不是要对 runtime 里面很多违和的地方清理一下。

消费完毕的 workbuf 对象会被放回 work.empty，以供复用。

mgcwork.go

```
func (ww *gcWork) get() uintptr {
    w := (*gcWork)(noescape(unsafe.Pointer(ww)))
```

```
	// 从 work.full 获取一个 workbuf 对象。
	wbuf := w.wbuf.ptr()
	if wbuf == nil {
		wbuf = getfull(103)
		if wbuf == nil {
			return 0
		}
		w.wbuf = wbufptrOf(wbuf)
	}

	// 直接从本地 workbuf 提取。
	wbuf.nobj--
	obj := wbuf.obj[wbuf.nobj]

	// 本地 workbuf 已空，将其放回 work.empty 供复用。
	if wbuf.nobj == 0 {
		putempty(wbuf, 115)
		w.wbuf = 0
	}

	return obj
}

func putempty(b *workbuf, entry int) {
	lfstackpush(&work.empty, &b.node)
}
```

至于 Free-Lock Stack 的实现也很简单，利用 CAS（Compare & Swap）指令来实现原子替换操作。这里用 Node Pointer + Node.PushCount 实现了 Double-CAS。

lfstack.go

```
func lfstackpush(head *uint64, node *lfnode) {
	// 累加计数器。
	node.pushcnt++

	// 利用 pointer + pushcnt 获得唯一流水号。
	new := lfstackPack(node, node.pushcnt)

	// 逆向展开流水号，进行错误检查。
	if node1, _ := lfstackUnpack(new); node1 != node {
		throw("lfstackpush")
	}

	// 类似自旋，重试直到成功。
	for {
		// 原子读取原 head node 流水号。(多核)
		old := atomicload64(head)

		// 将当前 node 作为 head。
		// 未成功前，这个操作并不影响原 stack。
```

```
            node.next = old

            // 利用 CAS 指令替换原 head。
            // 如替换失败，则循环重试。
            if cas64(head, old, new) {
                    break
            }
    }
}

func lfstackpop(head *uint64) unsafe.Pointer {
    for {
            // 原子读取 stack head。
            old := atomicload64(head)
            if old == 0 {
                    return nil
            }

            // 展开流水号，获取 pointer。
            node, _ := lfstackUnpack(old)

            // 利用 CAS 指令修改 stack head。
            next := atomicload64(&node.next)
            if cas64(head, old, next) {
                    return unsafe.Pointer(node)
            }
    }
}
```

如果 CAS 指令判断的仅是 old 指针地址，而该地址又被意外重用，那就会造成错误结果，这就是所谓 ABA 问题。利用 "指针地址+计数器" 生成唯一流水号，实现 Double-CAS，就能避开。

---

lfstack_amd64.go

```
func lfstackPack(node *lfnode, cnt uintptr) uint64 {
    return uint64(uintptr(unsafe.Pointer(node)))<<16 | uint64(cnt&(1<<19-1))
}
```

内存状态统计

除用 GODEBUG="gctrace=1" 输出垃圾回收状态信息外，某些时候我们还需要自行获取内存相关统计数据。

与之相关的数据结构，分别是运行时内部使用的 mstats 和面向用户的 MemStats。两者大部分结构相同，只是在输出结果上有细微调整。

mstats.go

```
type mstats struct {
    alloc           uint64        // 当前分配的 object 内存（含未回收的白色对象）。
    total_alloc     uint64        // 历史累计分配内存（当前正在使用和历次回收释放）。
    sys             uint64        // 当前从操作系统获取的内存（所有分配总和，不包括已释放）。
    nmalloc         uint64        // 分配次数累计。
    nfree           uint64        // 释放次数累计。

    heap_alloc      uint64        // 同 alloc。
    heap_sys        uint64        // 从操作系统获取的内存（不包括已释放）。
    heap_idle       uint64        // 闲置 span 内存。
    heap_inuse      uint64        // 正在使用 span 内存（从 heap 提取，包括 stack）。
    heap_released   uint64        // 当前已归还操作系统的内存。
    heap_objects    uint64        // 正在使用 object 数量（不含闲置链表）。

    stacks_inuse    uint64        // 正在使用 stack 内存（含 stackpool）。
    mspan_inuse     uint64        // 正在使用 mspan 内存。
    mcache_inuse    uint64        // 正在使用 mcache 内存。

    next_gc         uint64        // 下次垃圾回收阈值。
    last_gc         uint64        // 上次垃圾回收结束时间（UnixNano，不包括并发清理）。
    pause_total_ns  uint64        // 累计 STW 暂停时间。
    pause_ns        [256]uint64   // 最近垃圾回收周期里 STW 暂停时间（循环缓冲区）。
    pause_end       [256]uint64   // 最近垃圾回收周期里 STW 暂停结束时间（UnixNano）。
    numgc           uint32        // 垃圾回收次数。
    gc_cpu_fraction float64       // GC 所耗 CPU 时间比例（f*100 %）。

    heap_live       uint64        // 自上次回收后堆使用内存（黑色+新分配，不包括白色对象）。
}
```

---

object 特指 cache 分配的小块内存，以及 large object，而非实际用户对象。

用户通过 runtime.ReadMemStats 函数来获取统计数据。

mstats.go

```
func ReadMemStats(m *MemStats) {
     stopTheWorld("read mem stats")

     systemstack(func() {
            readmemstats_m(m)
     })

     startTheWorld()
}
```

```
func readmemstats_m(stats *MemStats) {
    updatememstats(nil)

    // 前部分数据结构相同，直接拷贝。
    memmove(unsafe.Pointer(stats), unsafe.Pointer(&memstats), sizeof_C_MStats)

    // 将栈内存从统计数据剔除，仅显示用户逻辑消耗。
    stats.StackSys += stats.StackInuse
    stats.HeapInuse -= stats.StackInuse
    stats.HeapSys -= stats.StackInuse
}
```

注意：ReadMemStats 会进行 STW 操作，应控制调用时间和次数。

监控输出示例。

```
                   heap_idle           heap_released
                   |                   |
scvg0: inuse: 3, idle: 1, sys: 5, released: 0, consumed: 5 (MB)
         |                   |                        |
         heap_inuse          heap_sys                 heap_sys - heap_released
```

六. 并发调度

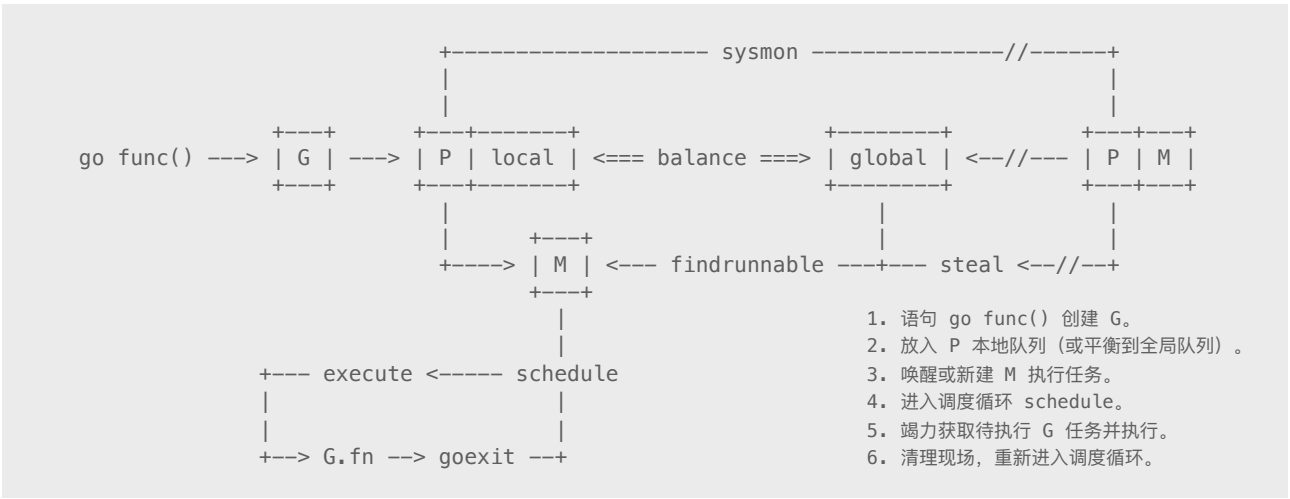
因为 Goroutine，才让 Golang 与众不同。

你所看到的和产出的一切都在以并发方式运行，垃圾回收、系统监控、网络通讯、文件读写，还有用户并发任务等等，所有这些都需要一个高效且聪明的调度器来指挥协调。

1. 概述

内置运行时，在进程和线程的基础上做更高层次的抽象是现代语言最流行的做法。虽然算不上激进，但 Golang 也设计了全新架构模型，将一切都基于并发体系之上，以适应多核时代。刻意模糊线程或协程概念，通过三种基本对象相互协作，来实现在用户空间管理和调度并发任务。

基本关系示意图：



首先是 Processor（简称 P），其作用类似 CPU 核，用来控制可同时并发执行的任务数量。每个工作线程都必须绑定一个有效 P 才被允许执行任务，否则只能休眠，直到有空闲 P 时被唤醒。P 还为线程提供执行资源，比如对象分配内存、本地任务队列等。线程独享所绑定的 P 资源，可在无锁状态下执行高效操作。

基本上，进程内的一切都在以 goroutine（简称 G）方式运行，包括运行时相关服务，以及 main.main 入口函数。需要指出，G 并非执行体，它仅仅保存并发任务状态，为任务执行提供所需栈内存空间。G 任务创建后被放置在 P 本地队列或全局队列，等待工作线程调度执行。

实际执行体是系统线程（简称 M），它和 P 绑定，以调度循环方式不停执行 G 并发任务。M 通过修改寄存器，将执行栈指向 G 自带栈内存，并在此空间内分配堆栈帧，执行任务函数。当需要中途切换时，只要将相关寄存器值保存回 G 空间即可维持状态，任何 M 都可据此恢复执行。线程仅负责执行，不再持有状态，这是并发任务跨线程调度，实现多路复用的根本所在。

尽管 P/M 构成执行组合体，但两者数量并非一一对应。通常情况下，P 数量相对恒定，默认与 CPU 核数量相同，但也可能更多或更少，而 M 则是调度器按需创建。举例来说，当 M 因陷入系统调用而长时间阻塞时，P 就会被监控线程抢回，去新建（或唤醒）一个 M 执行其他任务，如此 M 的数量就会增长。

因为 G 初始栈仅有 2KB，且创建操作只是在用户空间简单的对象分配，远比进入内核态分配线程要简单得多。调度器让多个 M 进入调度循环，不停获取并执行任务，所以我们才能创建成千上万个并发任务。

2. 初始化

调度器初始化函数 schedinit 在前文已多次提及，除去内存分配、垃圾回收等操作外，针对自身的初始化无非是 MaxMcount、GOMAXPROCS。

proc1.go

```
func schedinit() {
    // 设置最大 M 数量。
    sched.maxmcount = 10000

    // 初始化栈空间复用管理链表。
    stackinit()

    // 初始化当前 M。
    mcommoninit(_g_.m)

    // 默认值总算从 1 调整为 CPU Core 数量了。
    procs := int(ncpu)
    if n := atoi(gogetenv("GOMAXPROCS")); n > 0 {
        if n > _MaxGomaxprocs {
            n = _MaxGomaxprocs
        }
        procs = n
    }

    // 调整 P 数量。
    // 注意：此刻所有 P 都是新建，所以不可能返回有本地任务的 P。
    if procresize(int32(procs)) != nil {
```

```
            throw("unknown runnable goroutine during bootstrap")
        }
    }
```

GOMAXPROCS 默认值总算从 1 改为 CPU Cores 了。

因为 P 的数量有最大限制，所以用一个足够大的数组存储才是最正常的做法。虽然浪费点空间，但省去很多内存增减的麻烦。

runtime2.go

```
var allp [_MaxGomaxprocs + 1]*p

type schedt struct {
    pidle      puintptr    // 空闲 P 链表。
    npidle     uint32      // 空闲 P 数量。
}
```

调整 P 数量并不意味着全部分配新对象，仅仅做去余补缺。

proc1.go

```
func procresize(nprocs int32) *p {
    old := gomaxprocs

    // 新增。
    for i := int32(0); i < nprocs; i++ {
            pp := allp[i]

            // 申请新 P 对象。
            if pp == nil {
                    pp = new(p)
                    pp.id = i
                    pp.status = _Pgcstop

                    // 保存到 allp。
                    atomicstorep(unsafe.Pointer(&allp[i]), unsafe.Pointer(pp))
            }

            // 为 P 分配 cache 对象。
            if pp.mcache == nil {
                    if old == 0 && i == 0 {
                            // bootstrap
                            pp.mcache = getg().m.mcache
                    } else {
                            // 创建 cache。
                            pp.mcache = allocmcache()
```

```
                }
        }
}

// 释放多余的 P。
for i := nprocs; i < old; i++ {
        p := allp[i]

        // 将本地任务转移到全局队列。
        for p.runqhead != p.runqtail {
                p.runqtail--
                gp := p.runq[p.runqtail%uint32(len(p.runq))]
                globrunqputhead(gp)
        }
        if p.runnext != 0 {
                globrunqputhead(p.runnext.ptr())
                p.runnext = 0
        }

        // 释放当前 P 绑定的 cache。
        freemcache(p.mcache)
        p.mcache = nil

        // 将当前 P 的 G 复用链转移到全局。
        gfpurge(p)

        // 似乎就丢那不管了，反正也没剩下啥。
        p.status = _Pdead
        // can't free P itself because it can be referenced by an M in syscall
}

_g_ := getg()

// 如果当前正在用的 P 属于被释放的那拨，那就换成 allp[0]。
// 调度器初始化阶段，根本没有 P，那就绑定 allp[0]。
if _g_.m.p != 0 && _g_.m.p.ptr().id < nprocs {
        // 继续使用当前 P。
        _g_.m.p.ptr().status = _Prunning
} else {
        // 释放当前 P，因为它已经失效。
        if _g_.m.p != 0 {
                _g_.m.p.ptr().m = 0
        }
        _g_.m.p = 0
        _g_.m.mcache = nil

        // 换成 allp[0]。
        p := allp[0]
        p.m = 0
        p.status = _Pidle
        acquirep(p)
}

// 将没有本地任务的 P 放到空闲链表。
```

```
	var runnablePs *p
	for i := nprocs - 1; i >= 0; i-- {
		p := allp[i]

		// 确保不是当前正在用的 P。
		if _g_.m.p.ptr() == p {
			continue
		}

		p.status = _Pidle
		if runqempty(p) {
			// 放入空闲链表。
			pidleput(p)
		} else {
			// 有本地任务，构建链表。
			p.m.set(mget())
			p.link.set(runnablePs)
			runnablePs = p
		}
	}

	// 返回有本地任务的 P（链表）。
	return runnablePs
}

// 将 P 放入空闲链表。
func pidleput(_p_ *p) {
	_p_.link = sched.pidle
	sched.pidle.set(_p_)
	xadd(&sched.npidle, 1)
}
```

默认只有 schedinit 和 startTheWorld 会调用 procresize 函数。在调度器初始化阶段，所有 P 对象都是新建。除分配给当前主线程的外，其他都被放入空闲链表。而 startTheWorld 会激活全部有本地任务的 P 对象（详见后文）。

在完成调度器初始化后，引导过程才创建并运行 main goroutine。

asm_amd64.s

```
TEXT runtime·rt0_go(SB),NOSPLIT,$0
	// save m->g0 = g0
	MOVQ	CX, m_g0(AX)
	// save m0 to g0->m
	MOVQ	AX, g_m(CX)

	CALL	runtime·schedinit(SB)

	// 创建 main goroutine，并将其放入当前 P 本地队列。
	MOVQ	$runtime·mainPC(SB), AX
```

```
    PUSHQ   AX
    PUSHQ   $0
    CALL    runtime·newproc(SB)
    POPQ    AX
    POPQ    AX

    // 让当前 M0 进入调度，执行 main goroutine。
    CALL    runtime·mstart(SB)

    // M0 永远不会执行这条崩溃测试指令。
    MOVL    $0xf1, 0xf1  // crash
    RET
```

虽然可在运行期用 runtime.GOMAXPROCS 函数修改 P 数量，但需付出极大代价。

debug.go

```
func GOMAXPROCS(n int) int {
    if n > _MaxGomaxprocs {
            n = _MaxGomaxprocs
    }

    // 返回当前值（这个才是最常用的做法）。
    ret := int(gomaxprocs)
    if n <= 0 || n == ret {
            return ret
    }

    // STW !!!
    stopTheWorld("GOMAXPROCS")

    newprocs = int32(n)

    // 调用 procresize，并激活有任务的 P。
    startTheWorld()

    return ret
}
```

3. 任务

我们已经知道编译器会将“go func(...)”语句翻译成 newproc 调用，但这中间究竟有什么不为人知的秘密？

test.go

```
package main
```

```
import ()

func add(x, y int) int {
    z := x + y
    return z
}

func main() {
    x := 0x100
    y := 0x200
    go add(x, y)
}
```

尽管这个示例有些简陋，但这不重要，重点是编译器要做什么。

```
$ go build -o test test.go

$ go tool objdump -s "main\.main" test

TEXT main.main(SB) test.go
    test.go:10  SUBQ $0x28, SP
    test.go:11  MOVQ $0x100, CX
    test.go:12  MOVQ $0x200, AX
    test.go:13  MOVQ CX, 0x10(SP)         // 实参 x 入栈。
    test.go:13  MOVQ AX, 0x18(SP)         // 实参 y 入栈。
    test.go:13  MOVL $0x18, 0(SP)         // 参数长度入栈。
    test.go:13  LEAQ 0x879ff(IP), AX      // 将函数 add 地址存入 AX 寄存器。
    test.go:13  MOVQ AX, 0x8(SP)          // 地址入栈。
    test.go:13  CALL runtime.newproc(SB)
    test.go:14  ADDQ $0x28, SP
    test.go:14  RET
```

从反汇编代码可以看出，Golang 采用了类似 C/cdecl 调用约定。由调用方负责提供参数空间，并从右往左入栈。

proc1.go

```
func newproc(siz int32, fn *funcval) {
     // 获取第一参数地址。
     argp := add(unsafe.Pointer(&fn), ptrSize)

     // 获取调用方 PC/IP 寄存器值。
     pc := getcallerpc(unsafe.Pointer(&siz))

     // 用 g0 栈创建 G/goroutine 对象。
     systemstack(func() {
            newproc1(fn, (*uint8)(argp), siz, 0, pc)
```

```
    })
}
```

目标函数 newproc 只有两个参数，但 main 却向栈压入了四个值。按照顺序，后三个值应该会被合并成 funcval。还有，add 返回值被忽略。

runtime2.go

```
type funcval struct {
    fn uintptr
    // variable-size, fn-specific data here
}
```

果然是变长结构类型（目标函数参数不定），此处其补全状态应该是：

```
type struct {
    fn uintptr
    x  int
    y  int
}
```

如此一来，关于“go 语句会复制参数值”的规则就很好理解了。站在 newproc 角度，我们可以画出执行栈的状态示意图。

```
  lower         SP +-------------+
                   |             |
    ^              +-------------+  <----  newproc frame
    .              | pc/ip       |
    .              +-------------+
    .              | siz         |
 address           +-------------+  ---+
    .              | add         |     |
    .              +-------------+     |
    .              | x           |      >  fn
    .              +-------------+     |
                   | y           |     |
  higher           +-------------+  ---+
```

用“fn + ptrsize”跳过 add 获得第一个参数 x 的地址，getcallerpc 用“siz - 8”读取 CALL 指令压入的 main PC/IP 寄存器值，这就是 newproc 为 newproc1 准备的相关参数值。

asm_amd64.s

```
TEXT runtime·getcallerpc(SB),NOSPLIT,$8-16
    MOVQ    argp+0(FP),AX                       // addr of first arg
```

```
    MOVQ    -8(AX),AX                       // get calling pc
    CMPQ    AX, runtime·stackBarrierPC(SB)
    JNE     nobar
    CALL    runtime·nextBarrierPC(SB)       // Get original return PC.
    MOVQ    0(SP), AX
nobar:
    MOVQ    AX, ret+8(FP)
    RET
```

至此，我们大概知道 go 语句编译后的真实模样。接下来，就转到 newproc1 看看如何创建并发任务单元 G。

runtime2.go

```
type g struct {
    stack   stack      // 执行栈
    sched   gobuf      // 用于保存执行现场。
    goid    int64      // 唯一序号。
    gopc    uintptr    // 调用者 PC/IP。
    startpc uintptr    // 任务函数。
}
```

proc1.go

```
func newproc1(fn *funcval, argp *uint8, narg int32, nret int32, callerpc uintptr) *g {
    _g_ := getg()

    // “参数 + 返回值” 所需空间（对齐）。
    siz := narg + nret
    siz = (siz + 7) &^ 7

    // 从当前 P 复用链表获取空闲 G 对象。
    _p_ := _g_.m.p.ptr()
    newg := gfget(_p_)

    // 获取失败，新建。
    if newg == nil {
            newg = malg(_StackMin)
            casgstatus(newg, _Gidle, _Gdead)
            allgadd(newg)
    }

    // 测试 G stack。
    if newg.stack.hi == 0 {
            throw("newproc1: newg missing stack")
    }

    // 测试 G status。
    if readgstatus(newg) != _Gdead {
```

```
                throw("newproc1: new g is not Gdead")
        }

        // 计算所需空间大小，并对齐。
        totalSize := 4*regSize + uintptr(siz)
        totalSize += -totalSize & (spAlign - 1)

        // 确定 SP 和参数入栈位置。
        sp := newg.stack.hi - totalSize
        spArg := sp

        // 将执行参数拷贝入栈。
        memmove(unsafe.Pointer(spArg), unsafe.Pointer(argp), uintptr(narg))

        // 初始化用于保存执行现场的区域。
        memclr(unsafe.Pointer(&newg.sched), unsafe.Sizeof(newg.sched))
        newg.sched.sp = sp
        newg.sched.pc = funcPC(goexit) + _PCQuantum
        newg.sched.g = guintptr(unsafe.Pointer(newg))
        gostartcallfn(&newg.sched, fn)

        // 初始化基本状态。
        newg.gopc = callerpc
        newg.startpc = fn.fn
        casgstatus(newg, _Gdead, _Grunnable)

        // 设置唯一 id。
        if _p_.goidcache == _p_.goidcacheend {
                // sched.goidgen 是一个全局计数器。
                // 每次取回一段有效区间，然后在该区间分配，避免频繁去全局操作。
                // [sched.goidgen+1, sched.goidgen+GoidCacheBatch]
                _p_.goidcache = xadd64(&sched.goidgen, _GoidCacheBatch)
                _p_.goidcache -= _GoidCacheBatch - 1
                _p_.goidcacheend = _p_.goidcache + _GoidCacheBatch
        }
        newg.goid = int64(_p_.goidcache)
        _p_.goidcache++

        // 将 G 放入待运行队列。
        runqput(_p_, newg, true)

        // 如果有其他空闲 P，则尝试唤醒某个 M 出来执行任务。
        // 如果有 M 处于自旋等待 P 或 G 状态，放弃。
        // 如果当前创建的是 main goroutine (runtime.main)，那么还没有其他任务需要执行，放弃。
        if atomicload(&sched.npidle) != 0 &&
           atomicload(&sched.nmspinning) == 0 &&
           unsafe.Pointer(fn.fn) != unsafe.Pointer(funcPC(main)) {
                wakep()
        }

        return newg
}
```

整个创建过程中，有一系列问题需要分开详说。

首先，G 对象默认会复用，这看上去有点像 cache/object 做法。除 P 本地的复用链表外，还有全局链表在多个 P 之间共享。

runtime2.go

```
type p struct {
    gfree    *g
    gfreecnt int32
}

type schedt struct {
    gfree  *g
    ngfree int32
}
```

proc1.go

```
func gfget(_p_ *p) *g {
retry:
    // 从 P 本地队列提取复用对象。
    gp := _p_.gfree

    // 如果提取失败，尝试从全局链表转移一批到 P 本地。
    if gp == nil && sched.gfree != nil {
            // 最多转移 32 个。
            for _p_.gfreecnt < 32 && sched.gfree != nil {
                    _p_.gfreecnt++
                    gp = sched.gfree
                    sched.gfree = gp.schedlink.ptr()
                    sched.ngfree--
                    gp.schedlink.set(_p_.gfree)
                    _p_.gfree = gp
            }

            // 再试。
            goto retry
    }

    // 如果成功获取复用对象。
    if gp != nil {
            // 调整 P 复用链表。
            _p_.gfree = gp.schedlink.ptr()
            _p_.gfreecnt--

            // 检查 G stack。
            if gp.stack.lo == 0 {
                    // 分配新栈。
```

```
                        systemstack(func() {
                                gp.stack, gp.stkbar = stackalloc(_FixedStack)
                        })
                        gp.stackguard0 = gp.stack.lo + _StackGuard
                        gp.stackAlloc = _FixedStack
                } else {
                }
        }

        return gp
}
```

而当 goroutine 执行完毕，调度器相关函数会将 G 对象放回 P 复用链表。

proc1.go

```
func gfput(_p_ *p, gp *g) {
        // 如果栈发生过扩张，则释放。
        stksize := gp.stackAlloc
        if stksize != _FixedStack {
                // non-standard stack size - free it.
                stackfree(gp.stack, gp.stackAlloc)
                gp.stack.lo = 0
                gp.stack.hi = 0
                gp.stackguard0 = 0
                gp.stkbar = nil
                gp.stkbarPos = 0
        } else {
                // Reset stack barriers.
                gp.stkbar = gp.stkbar[:0]
                gp.stkbarPos = 0
        }

        // 放回 P 本地复用链表。
        gp.schedlink.set(_p_.gfree)
        _p_.gfree = gp
        _p_.gfreecnt++

        // 如果本地复用对象过多，则转移一批到全局链表。
        if _p_.gfreecnt >= 64 {
                // 本地仅保留 32 个。
                for _p_.gfreecnt >= 32 {
                        _p_.gfreecnt--
                        gp = _p_.gfree
                        _p_.gfree = gp.schedlink.ptr()
                        gp.schedlink.set(sched.gfree)
                        sched.gfree = gp
                        sched.ngfree++
                }
        }
}
```

最初，G 对象都是由 malg 创建。

stack2.go

```
_StackMin = 2048
```

proc1.go

```
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
}
```

默认采用 2KB 栈空间，并且都被 allg 引用。这是垃圾回收遍历扫描需要，以便获取指针引用，收缩栈空间。

proc1.go

```
var (
    allg    **g
    allglen  uintptr
    allgs    []*g
)

func allgadd(gp *g) {
    allgs = append(allgs, gp)
    allg = &allgs[0]
    allglen = uintptr(len(allgs))
}
```

现在我们知道 G 的由来，以及复用方式。只是有个小问题，G 似乎从来不被释放，会不会有存留过多的问题？不过好在垃圾回收会调用 shrinkstack 将其栈空间回收。有关栈的相关细节，留待后文再说。

在获取 G 对象后，newproc1 会进行一系列初始化操作，毕竟不管新建还是复用，这些参数都必须正确设置。同时，相关执行参数会被拷贝到 G 的栈空间，因为它和当前任务不再有任何关系，各自使用独立的栈空间。毕竟，“go func(...)”语句仅仅创建并发任务，当前流程会继续自己的逻辑。

创建完毕的 G 任务被优先放入 P 本地队列等待执行，这属于无锁操作。

proc1.go

```
func runqput(_p_ *p, gp *g, next bool) {
    if randomizeScheduler && next && fastrand1()%2 == 0 {
        next = false
    }

    // 如果可能，将 G 直接保存在 P.runnext，作为下一个优先执行任务。
    if next {
    retryNext:
        oldnext := _p_.runnext
        if !_p_.runnext.cas(oldnext, guintptr(unsafe.Pointer(gp))) {
            goto retryNext
        }
        if oldnext == 0 {
            return
        }

        // 原本的 next G 会被放回本地队列。
        gp = oldnext.ptr()
    }

retry:
    // runqhead 是一个数组实现的循环队列。
    // head、tail 累加，通过取模即可获得索引位置，很典型算法。
    h := atomicload(&_p_.runqhead)
    t := _p_.runqtail

    // 如果本地队列未满，直接放到尾部。
    if t-h < uint32(len(_p_.runq)) {
        _p_.runq[t%uint32(len(_p_.runq))] = gp
        atomicstore(&_p_.runqtail, t+1)
        return
    }

    // 放入全局队列。
    // 因为需要加锁，所以 slow。
    if runqputslow(_p_, gp, h, t) {
        return
    }

    goto retry
}
```

任务队列分为三级，按优先级从高到低分别是 P.runnext、P.runq、Sched.runq，很有些 CPU 多级缓存的意思。

runtime2.go

```
type schedt struct {
     runqhead guintptr
     runqtail guintptr
     runqsize int32
}

type p struct {
     runqhead uint32
     runqtail uint32
     runq     [256]*g     // 本地队列，访问无需加锁。
     runnext  guintptr    // 优先执行。
}

type g struct {
     schedlink guintptr   // 链表。
}
```

往全局队列添加任务，显然需要加锁，只是专门取名为 runqputslow 就很有说法了。去看看到底怎么个慢法。

proc1.go

```
func runqputslow(_p_ *p, gp *g, h, t uint32) bool {
     // 这意思显然是要从 P 本地转移一半任务到全局队列。
     // "+1" 是别忘了当前这个 gp。
     var batch [len(_p_.runq)/2 + 1]*g

     // 计算一半的实际数量。
     n := t - h
     n = n / 2

     // 从队列头部提取。
     for i := uint32(0); i < n; i++ {
             batch[i] = _p_.runq[(h+i)%uint32(len(_p_.runq))]
     }

     // 调整 P 队列头部位置。
     if !cas(&_p_.runqhead, h, h+n) {
             return false
     }

     // 加上当前 gp 这家伙。
     batch[n] = gp
```

```
    // 对顺序进行洗牌。
    if randomizeScheduler {
        for i := uint32(1); i <= n; i++ {
            j := fastrand1() % (i + 1)
            batch[i], batch[j] = batch[j], batch[i]
        }
    }

    // 串成链表。
    for i := uint32(0); i < n; i++ {
        batch[i].schedlink.set(batch[i+1])
    }

    // 添加到全局队列尾部。
    globrunqputbatch(batch[0], batch[n], int32(n+1))
    return true
}

func globrunqputbatch(ghead *g, gtail *g, n int32) {
    gtail.schedlink = 0
    if sched.runqtail != 0 {
        sched.runqtail.ptr().schedlink.set(ghead)
    } else {
        sched.runqhead.set(ghead)
    }
    sched.runqtail.set(gtail)
    sched.runqsize += n
}
```

如本地队列已满，一次性转移半数到全局队列。这个好理解，因为其他 P 可能正饿着呢。这也正好解释了 newproc1 最后尝试用 wakep 唤醒其他 M/P 去执行任务的意图，毕竟充分发挥多核优势才是正途。

最后标记一下 G 的状态切换过程。

```
-- gfree -----+
              |
--> IDLE --> DEAD ------> RUNNABLE ---> RUNNING ---> DEAD --- ... --> gfree -->
    新建      初始化前         初始化后         调度执行       执行完毕
```

4. 线程

当 newproc1 成功创建 G 任务后，会尝试用 wakep 唤醒 M 执行任务。

proc1.go

```
func wakep() {
     // 被唤醒的线程需要绑定 P，累加自旋计数，避免 newproc1 唤醒过多线程。
     if !cas(&sched.nmspinning, 0, 1) {
            return
     }
     startm(nil, true)
}

func startm(_p_ *p, spinning bool) {
     // 如果没有指定 P，尝试获取空闲 P。
     if _p_ == nil {
            _p_ = pidleget()

            // 获取失败，终止。
            if _p_ == nil {
                   // 递减自旋计数。
                   if spinning {
                          xadd(&sched.nmspinning, -1)
                   }
                   return
            }
     }

     // 获取休眠的闲置 M。
     mp := mget()

     // 如没有闲置 M，新建。
     if mp == nil {
            // 默认启动函数。
            // 主要是判断 M.nextp 是否有暂存的 P，以此调整自旋计数。
            var fn func()
            if spinning {
                   fn = mspinning
            }
            newm(fn, _p_)
            return
     }

     // 设置自旋状态和暂存 P。
     mp.spinning = spinning
     mp.nextp.set(_p_)

     // 唤醒 M。
     notewakeup(&mp.park)
}
```

---

notewakeup/notesleep 实现细节参见后文。

和前文 G 对象复用类似，这个过程同样有闲置获取和新建两种方式。先不去理会闲置列表，看看 M 究竟如何创建，如何包装系统线程。

runtime2.go

```
type m struct {
    g0            *g              // 提供系统栈空间。
    mstartfn      func()          // 启动函数。
    curg          *g              // 当前运行 G。
    p             puintptr        // 绑定 P。
    nextp         puintptr        // 临时存放 P。
    spinning      bool            // 自旋状态。
    park          note            // 休眠锁。
    schedlink     muintptr        // 链表。
}
```

proc1.go

```
func newm(fn func(), _p_ *p) {
    // 创建 M 对象。
    mp := allocm(_p_, fn)

    // 暂存 P。
    mp.nextp.set(_p_)

    // 创建系统线程。
    newosproc(mp, unsafe.Pointer(mp.g0.stack.hi))
}

func allocm(_p_ *p, fn func()) *m {
    mp := new(m)
    mp.mstartfn = fn  // 启动函数。
    mcommoninit(mp)   // 初始化。

    // 创建 g0。
    // In case of cgo or Solaris, pthread_create will make us a stack.
    // Windows and Plan 9 will layout sched stack on OS stack.
    if iscgo || GOOS == "solaris" || GOOS == "windows" || GOOS == "plan9" {
            mp.g0 = malg(-1)
    } else {
            mp.g0 = malg(8192 * stackGuardMultiplier)
    }
    mp.g0.m = mp

    return mp
}
```

M 最特别的就是自带一个名为 g0，默认 8KB 栈内存的 G 对象属性。它的栈内存地址被传给 newosproc 函数，作为系统线程默认堆栈空间（并非所有系统都支持）。

os1_linux.go

```
const cloneFlags = _CLONE_VM |           /* share memory */
                   _CLONE_FS |           /* share cwd, etc */
                   _CLONE_FILES |        /* share fd table */
                   _CLONE_SIGHAND |      /* share sig handler table */
                   _CLONE_THREAD         /* revisit - okay for now */

func newosproc(mp *m, stk unsafe.Pointer) {
    ret := clone(cloneFlags, stk, unsafe.Pointer(mp), unsafe.Pointer(mp.g0),
                 unsafe.Pointer(funcPC(mstart)))
}
```

系统调用 clone 更多信息，请参考 man 2 手册。

---

os1_windows.go

```
func newosproc(mp *m, stk unsafe.Pointer) {
    const _STACK_SIZE_PARAM_IS_A_RESERVATION = 0x00010000
    thandle := stdcall6(_CreateThread, 0, 0x20000,
            funcPC(tstart_stdcall), uintptr(unsafe.Pointer(mp)),
            _STACK_SIZE_PARAM_IS_A_RESERVATION, 0)
    if thandle == 0 {
            print("runtime: failed to create new OS thread (have ",
                    mcount(), " already; errno=", getlasterror(), ")\n")
            throw("runtime.newosproc")
    }
}
```

Windows API CreateThread 不支持自定义线程堆栈。

在进程执行过程中，有两类代码需要运行。其一自然是用户逻辑，直接使用 G 栈内存；另一种是运行时管理指令，它并不方便直接在用户栈上执行，因为这需要处理与用户逻辑现场有关的一大堆事务。

举例来说，G 任务可在中途暂停，放回队列后由其他 M 获取执行。如不更改执行栈，那可能会造成多个线程共享内存，从而引发混乱。另外，在执行垃圾回收操作时，如何收缩依旧被线程持有的 G 栈空间？为此，当需要执行管理指令时，会将线程栈临时切换到 g0，与用户逻辑彻底隔离。

其实，在前文就经常看到 systemstack 这种执行方式，它就是切换到 g0 栈后再执行运行时相关管理操作。

proc1.go

```
func newproc(siz int32, fn *funcval) {
    systemstack(func() {
           newproc1(fn, (*uint8)(argp), siz, 0, pc)
    })
}
```

asm_amd64.s

```
TEXT runtime·systemstack(SB), NOSPLIT, $0-8
    MOVQ    fn+0(FP), DI                  // DI = fn
    MOVQ    g(CX), AX                     // AX = g
    MOVQ    g_m(AX), BX                   // BX = m

    MOVQ    m_g0(BX), DX                  // DX = g0
    CMPQ    AX, DX                        // 如果当前 g 已经是 g0，那么无需切换。
    JEQ     noswitch

    MOVQ    m_curg(BX), R8                // 当前 g。
    CMPQ    AX, R8                        // 如果是用户逻辑 g，切换。
    JEQ     switch

    // Bad: g is not gsignal, not g0, not curg. What is it?
    MOVQ    $runtime·badsystemstack(SB), AX
    CALL    AX

switch:
    // 将 G 状态保存到 sched。
    MOVQ    $runtime·systemstack_switch(SB), SI
    MOVQ    SI, (g_sched+gobuf_pc)(AX)
    MOVQ    SP, (g_sched+gobuf_sp)(AX)
    MOVQ    AX, (g_sched+gobuf_g)(AX)
    MOVQ    BP, (g_sched+gobuf_bp)(AX)

    // 切换到 g0.stack。
    MOVQ    DX, g(CX)                     // DX = g0
    MOVQ    (g_sched+gobuf_sp)(DX), BX    // 从 g0.sched 获取 SP。
    SUBQ    $8, BX                        // 调整 SP。
    MOVQ    $runtime·mstart(SB), DX
    MOVQ    DX, 0(BX)
    MOVQ    BX, SP                        // 通过调整 SP 寄存器值来切换栈内存。

    // 执行系统管理函数。
    MOVQ    DI, DX                        // DI = fn
    MOVQ    0(DI), DI
    CALL    DI
```

```
    // 切换回 G，恢复执行现场。
    MOVQ    g(CX), AX
    MOVQ    g_m(AX), BX
    MOVQ    m_curg(BX), AX
    MOVQ    AX, g(CX)
    MOVQ    (g_sched+gobuf_sp)(AX), SP
    MOVQ    $0, (g_sched+gobuf_sp)(AX)
    RET

noswitch:
    // already on m stack, just call directly
    MOVQ    DI, DX
    MOVQ    0(DI), DI
    CALL    DI
    RET
```

从这段代码，我们可以看出 g0 为什么同样是个 G 对象，而不是直接用 stack 的原因。

M 初始化操作会检查已有数量，如超出最大限制（默认 10000）会导致进程崩溃。所有 M 被添加到 allm 链表，且不被释放。

runtime2.go

```
var allm *m
```

proc1.go

```
func mcommoninit(mp *m) {
    mp.id = sched.mcount
    sched.mcount++
    checkmcount()
    mpreinit(mp)

    mp.alllink = allm
    atomicstorep(unsafe.Pointer(&allm), unsafe.Pointer(mp))
}

func checkmcount() {
    if sched.mcount > sched.maxmcount {
            throw("thread exhaustion")
    }
}
```

可用 runtime/debug.SetMaxThreads 修改最大线程数量限制，但仅建议在测试阶段通过设置较小值作为错误触发条件。

回到 wakep/startm 流程，默认优先选用闲置 M，只是这个闲置从何而来？

runtime2.go

```
type schedt struct {
    midle        muintptr   // 闲置 M 链表。
    nmidle       int32      // 闲置 M 数量。
    mcount       int32      // 已创建 M 总数。
    maxmcount    int32      // M 最大闲置。
}
```

proc1.go

```
// 从空闲链表获取 M。
func mget() *m {
    mp := sched.midle.ptr()
    if mp != nil {
        sched.midle = mp.schedlink
        sched.nmidle--
    }
    return mp
}
```

被唤醒进入工作状态的 M，会陷入调度循环，从各种可能场所获取并执行 G 任务。只有当彻底找不到可执行任务，或因任务用时过长、系统调用阻塞等原因被剥夺 P 时，才会进入休眠状态。

proc1.go

```
// 停止 M，使其休眠。
func stopm() {
    _g_ := getg()

    // 取消自旋状态。
    if _g_.m.spinning {
        _g_.m.spinning = false
        xadd(&sched.nmspinning, -1)
    }

retry:
    // 放回闲置队列。
    mput(_g_.m)

    // 休眠，等待被唤醒。
    notesleep(&_g_.m.park)
    noteclear(&_g_.m.park)
```

```
    // 绑定 P。
    acquirep(_g_.m.nextp.ptr())
    _g_.m.nextp = 0
}

// 将 M 放入闲置链表。
func mput(mp *m) {
    mp.schedlink = sched.midle
    sched.midle.set(mp)
    sched.nmidle++
}
```

我们允许进程里有成千上万的并发任务 G，但最好不要有太多的 M。且不说通过系统调用创建线程本身就有很大的性能损耗，大量闲置且不被回收的线程、M 对象、g0 栈空间都是资源浪费。好在这种情形极少出现，不过还是建议在生产部署前做严格测试。

下面是利用 cgo 调用 sleep syscall 来生成大量 M 的示例。

test.go

```
package main

import (
    "sync"
    "time"
)

// #include <unistd.h>
import "C"

func main() {
    var wg sync.WaitGroup
    wg.Add(1000)

    for i := 0; i < 1000; i++ {
        go func() {
            C.sleep(1)
            wg.Done()
        }()
    }

    wg.Wait()
    println("done!")
    time.Sleep(time.Second * 5)
}
```

利用 GODEBUG 输出调度器状态，你会看到大量闲置线程。

```
$ go build -o test test

$ GODEBUG="schedtrace=1000" ./test

SCHED    0ms: gomaxprocs=2 idleprocs=1 threads=3   spinningthreads=0 idlethreads=0   runqueue=0 [0 0]
SCHED 1006ms: gomaxprocs=2 idleprocs=0 threads=728 spinningthreads=0 idlethreads=0   runqueue=125 [113 33]
SCHED 2009ms: gomaxprocs=2 idleprocs=2 threads=858 spinningthreads=0 idlethreads=590 runqueue=0 [0 0]
done!
SCHED 3019ms: gomaxprocs=2 idleprocs=2 threads=858 spinningthreads=0 idlethreads=855 runqueue=0 [0 0]
SCHED 4029ms: gomaxprocs=2 idleprocs=2 threads=858 spinningthreads=0 idlethreads=855 runqueue=0 [0 0]
SCHED 5038ms: gomaxprocs=2 idleprocs=2 threads=858 spinningthreads=0 idlethreads=855 runqueue=0 [0 0]
SCHED 6048ms: gomaxprocs=2 idleprocs=2 threads=858 spinningthreads=0 idlethreads=855 runqueue=0 [0 0]
```

---

runqueue 输出全局队列，以及 P 本地队列 G 任务数量。

可将 done 后的等待时间修改得更长（比如 10 分钟），用来观察垃圾回收和系统监控等机制是否会影响 idlethreads 数量。

```
$ GODEBUG="gctrace=1,schedtrace=1000" ./test
```

除线程数量外，程序执行时间（user, sys）也有很大差别，可以简单对比一下。

```
func main() {
    var wg sync.WaitGroup
    wg.Add(1000)

    for i := 0; i < 1000; i++ {
        go func() {
            C.sleep(1) // 测试 1
            // time.Sleep(time.Second) // 测试 2

            wg.Done()
        }()
    }

    wg.Wait()
}
```

```
$ go build -o test1 test.go && time ./test1

real    0m1.159s
user    0m0.056s
```

```
sys     0m0.105s

$ go build -o test2 test.go && time ./test2

real    0m1.022s
user    0m0.006s
sys     0m0.006s
```

> 输出结果中 user 和 sys 分别表示用户态和内核态执行时间，多核累加。

标准库封装的 time.Sleep 针对 goroutine 进行了改进，并未使用 syscall。当然，这个示例和测试结果也仅用于演示，具体问题具体对待。

5. 执行

M 执行 G 并发任务有两个起点：线程启动函数 mstart，还有就是 stopm 休眠唤醒后再度恢复调度循环。

让我们从头开始。

proc1.go

```
func mstart() {
    _g_ := getg()

    // 确定栈边界。
    if _g_.stack.lo == 0 {
            // 对于无法使用 g0 stack 的系统，直接在系统堆栈上划出所需空间。
            size := _g_.stack.hi
            if size == 0 {
                    size = 8192 * stackGuardMultiplier
            }
            // 通过取 size 变量指针来确定高位地址。
            _g_.stack.hi = uintptr(noescape(unsafe.Pointer(&size)))
            _g_.stack.lo = _g_.stack.hi - size + 1024
    }
    _g_.stackguard0 = _g_.stack.lo + _StackGuard
    _g_.stackguard1 = _g_.stackguard0

    mstart1()
}

func mstart1() {
    _g_ := getg()
```

```
    if _g_ != _g_.m.g0 {
        throw("bad runtime·mstart")
    }

    // 初始化 g0 执行现场。
    gosave(&_g_.m.g0.sched)
    _g_.m.g0.sched.pc = ^uintptr(0) // make sure it is never used

    // 执行启动函数。
    if fn := _g_.m.mstartfn; fn != nil {
        fn()
    }

    // 在 GC startTheWorld 时，会检查闲置 M 是否少于并发标记需求（needaddgcproc）。
    // 新建 M，设置 m.helpgc = -1，加入闲置队列等待唤醒。
    if _g_.m.helpgc != 0 {
        _g_.m.helpgc = 0
        stopm()
    } else if _g_.m != &m0 {
        // 绑定 P。
        acquirep(_g_.m.nextp.ptr())
        _g_.m.nextp = 0
    }

    // 进入任务调度循环（不再返回）。
    schedule()
}
```

准备进入工作状态的 M 必须绑定一个有效 P，nextp 临时持有待绑定 P 对象。因为在未正式执行前，并不适合直接设置相关属性。P 为 M 提供 cache，以便为执行绪提供对象内存分配。

proc1.go

```
func acquirep(_p_ *p) {
    acquirep1(_p_)

    // 绑定 mcache。
    _g_ := getg()
    _g_.m.mcache = _p_.mcache
}

func acquirep1(_p_ *p) {
    _g_ := getg()
    _g_.m.p.set(_p_)
    _p_.m.set(_g_.m)
    _p_.status = _Prunning
}
```

一切就绪后，M 进入核心调度循环，这是一个由 schedule、execute、goroutine fn、goexit 函数构成的逻辑循环。就算 M 在休眠唤醒后，也只是从“断点”恢复。

proc1.go

```
func schedule() {
    _g_ := getg()

top:
    // 准备进入 GC STW，休眠。
    if sched.gcwaiting != 0 {
            gcstopm()
            goto top
    }

    var gp *g

    // 当从 P.next 提取 G 时，inheritTime = true。
    // 不累加 P.schedtick 计数，使得它延长本地队列处理时间。
    var inheritTime bool

    // 进入 GC MarkWorker 工作模式。
    if gp == nil && gcBlackenEnabled != 0 {
            gp = gcController.findRunnableGCWorker(_g_.m.p.ptr())
            if gp != nil {
                    resetspinning()
            }
    }

    // 每处理 n 个任务后就去全局队列获取 G 任务，以确保公平。
    if gp == nil {
            if _g_.m.p.ptr().schedtick%61 == 0 && sched.runqsize > 0 {
                    lock(&sched.lock)
                    gp = globrunqget(_g_.m.p.ptr(), 1)
                    unlock(&sched.lock)
                    if gp != nil {
                            resetspinning()
                    }
            }
    }

    // 从 P 本地队列获取 G 任务。
    if gp == nil {
            gp, inheritTime = runqget(_g_.m.p.ptr())
            if gp != nil && _g_.m.spinning {
                    throw("schedule: spinning with local work")
            }
    }

    // 从其他可能的地方获取 G 任务。
    // 如果获取失败，会让 M 进入休眠状态，被唤醒后重试。
    if gp == nil {
            gp, inheritTime = findrunnable() // blocks until work is available
```

```
            resetspinning()
        }

        // 执行 goroutine 任务函数。
        execute(gp, inheritTime)
    }
```

> 有关 lockedg 细节，参见后文。

调度函数获取可用的 G 后，交由 execute 去执行。同时，还检查环境开关来决定是否参与垃圾回收。

把相关细节放下，先走完整个调度循环再说。

proc1.go

```
func execute(gp *g, inheritTime bool) {
    _g_ := getg()

    casgstatus(gp, _Grunnable, _Grunning)
    gp.waitsince = 0
    gp.preempt = false
    gp.stackguard0 = gp.stack.lo + _StackGuard

    _g_.m.curg = gp
    gp.m = _g_.m

    gogo(&gp.sched)
}
```

真正关键的就是汇编实现的 gogo 函数。它从 g0 栈切换到 G 栈，然后用一个 JMP 指令进入 G 任务函数代码。

asm_amd64.s

```
TEXT runtime·gogo(SB), NOSPLIT, $0-8
    MOVQ    buf+0(FP), BX          // gobuf
    MOVQ    gobuf_g(BX), DX        // G
    MOVQ    0(DX), CX              // make sure g != nil
    get_tls(CX)
    MOVQ    DX, g(CX)              // g = G
    MOVQ    gobuf_sp(BX), SP       // 通过恢复 SP 寄存器值切换到 G 栈。
    MOVQ    gobuf_ret(BX), AX
    MOVQ    gobuf_ctxt(BX), DX
    MOVQ    gobuf_bp(BX), BP
```

```
    MOVQ    $0, gobuf_sp(BX)      // clear to help garbage collector
    MOVQ    $0, gobuf_ret(BX)
    MOVQ    $0, gobuf_ctxt(BX)
    MOVQ    $0, gobuf_bp(BX)
    MOVQ    gobuf_pc(BX), BX      // 获取 G 任务函数地址。
    JMP     BX                    // 执行。
```

这里有个细节，JMP 并不是 CALL，也就是说不会将 PC/IP 入栈，那么执行完任务函数后，RET 指令恢复的 PC/IP 值是什么？我们在 schedule、execute 里也没看到 goexit 调用，究竟如何再次进入调度循环呢？

在 newproc1 创建 G 任务时，我们曾忽略了一个细节。

proc1.go

```
func newproc1(fn *funcval, argp *uint8, narg int32, nret int32, callerpc uintptr) *g {
    newg.sched.sp = sp

    // 此处保存的是 goexit 地址。
    newg.sched.pc = funcPC(goexit) + _PCQuantum
    newg.sched.g = guintptr(unsafe.Pointer(newg))

    // 此处调用是关键所在。
    gostartcallfn(&newg.sched, fn)

    newg.gopc = callerpc
    newg.startpc = fn.fn
}
```

在初始化 G.sched 时，pc 保存的是 goexit 而非 fn。关键秘密就是随后调用的 gostartcallfn 函数。

stack1.go

```
func gostartcallfn(gobuf *gobuf, fv *funcval) {
    gostartcall(gobuf, fn, (unsafe.Pointer)(fv))
}
```

sys_x86.go

```
func gostartcall(buf *gobuf, fn, ctxt unsafe.Pointer) {
    // 调整 sp。
    sp := buf.sp
    if regSize > ptrSize {
            sp -= ptrSize
```

```
            *(*uintptr)(unsafe.Pointer(sp)) = 0
    }
    sp -= ptrSize

    // 将 buf.pc 也就是 goexit 入栈。
    *(*uintptr)(unsafe.Pointer(sp)) = buf.pc

    // 然后再次设置 sp 和 pc，此时 pc 才是 G 任务函数。
    buf.sp = sp
    buf.pc = uintptr(fn)
    buf.ctxt = ctxt
}
```

ARM 使用 LR 寄存器存储 PC 值，而非保存在栈上。

很显然，在初始化完成后，G 栈顶端被压入了 goexit 地址。汇编函数 gogo JMP 跳转执行 G 任务，那么函数尾部的 RET 指令必然是将 goexit 地址恢复到 PC/IP，从而实现任务结束清理操作和再次进入调度循环。

asm_amd64.s

```
TEXT runtime·goexit(SB),NOSPLIT,$0-0
    CALL    runtime·goexit1(SB)    // does not return
```

proc1.go

```
func goexit1() {
    // 切换到 g0 执行 goexit0。
    mcall(goexit0)
}

// goexit continuation on g0.
func goexit0(gp *g) {
    _g_ := getg()

    // 清理 G 状态。
    casgstatus(gp, _Grunning, _Gdead)
    gp.m = nil
    gp.lockedm = nil
    _g_.m.lockedg = nil
    gp.paniconfault = false
    gp._defer = nil
    gp._panic = nil
    gp.writebuf = nil
    gp.waitreason = ""
    gp.param = nil

    dropg()
```

```
    _g_.m.locked = 0

    // 将 G 放回复用链表。
    gfput(_g_.m.p.ptr(), gp)

    // 重新进入调度循环。
    schedule()
}
```

无论是 mcall、systemstack，还是 gogo 都不会更新 g0.sched 栈现场。需要切换到 g0 栈时，直接从“g_sched+gobuf_sp”读取地址恢复 SP。所以调用 goexit0/schedule 时，g0 栈又从头开始，原调用堆栈全部失效，就算不返回也无所谓。

在 mstart1 里调用 gosave 初始化了 g0.sched.sp 等数据，

proc1.go

```
func mstart1() {
    // Record top of stack for use by mcall.
    // Once we call schedule we're never coming back,
    // so other calls can reuse this stack space.
    gosave(&_g_.m.g0.sched)
    _g_.m.g0.sched.pc = ^uintptr(0) // make sure it is never used
}
```

asm_amd64.s

```
// save state in Gobuf; setjmp
TEXT runtime·gosave(SB), NOSPLIT, $0-8
    MOVQ    buf+0(FP), AX         // gobuf
    LEAQ    buf+0(FP), BX         // caller's SP
    MOVQ    BX, gobuf_sp(AX)
    MOVQ    0(SP), BX             // caller's PC
    MOVQ    BX, gobuf_pc(AX)
    MOVQ    $0, gobuf_ret(AX)
    MOVQ    $0, gobuf_ctxt(AX)
    MOVQ    BP, gobuf_bp(AX)
    MOVQ    g(CX), BX
    MOVQ    BX, gobuf_g(AX)
    RET
```

至此，单次任务完整结束，又回到查找待运行 G 任务状态，循环往复。

findrunnable

为了找到可以运行的 G 任务，findrunnable 可谓费尽心机。本地队列、全局队列、网络任务（netpoll），甚至是从其他 P 任务队列偷窃。所有的目的都是为了尽快完成所有任务，充分发挥多核并行能力。

proc1.go

```
func findrunnable() (gp *g, inheritTime bool) {
    _g_ := getg()

top:
    // 垃圾回收。
    if sched.gcwaiting != 0 {
            gcstopm()
            goto top
    }

    // fing 是用来执行 finalizer 的 goroutine。
    if fingwait && fingwake {
            if gp := wakefing(); gp != nil {
                    ready(gp, 0)
            }
    }

    // 从本地队列获取。
    if gp, inheritTime := runqget(_g_.m.p.ptr()); gp != nil {
            return gp, inheritTime
    }

    // 从全局队列获取。
    if sched.runqsize != 0 {
            gp := globrunqget(_g_.m.p.ptr(), 0)
            if gp != nil {
                    return gp, false
            }
    }

    // 检查 netpoll 任务。
    if netpollinited() && sched.lastpoll != 0 {
            if gp := netpoll(false); gp != nil { // non-blocking
                    // 返回的是多任务链表，将其他任务放回全局队列。
                    // gp.schedlink 链表结构。
                    injectglist(gp.schedlink.ptr())
                    casgstatus(gp, _Gwaiting, _Grunnable)
                    return gp, false
            }
    }

    // 随机挑一个 P，偷些任务。
    for i := 0; i < int(4*gomaxprocs); i++ {
            if sched.gcwaiting != 0 {
```

```
                        goto top
                }
                // 随机数取模确定目标 P。
                _p_ := allp[fastrand1()%uint32(gomaxprocs)]
                var gp *g
                if _p_ == _g_.m.p.ptr() {
                        // 本地队列。
                        gp, _ = runqget(_p_)
                } else {
                        // 如果尝试次数太多，连目标 P.runnext 都偷，这是饿得狠了。
                        stealRunNextG := i > 2*int(gomaxprocs)
                        gp = runqsteal(_g_.m.p.ptr(), _p_, stealRunNextG)
                }
                if gp != nil {
                        return gp, false
                }
        }

stop:

        // 检查 GC MarkWorker。
        if _p_ := _g_.m.p.ptr(); gcBlackenEnabled != 0 && _p_.gcBgMarkWorker != nil &&
                gcMarkWorkAvailable(_p_) {
                _p_.gcMarkWorkerMode = gcMarkWorkerIdleMode
                gp := _p_.gcBgMarkWorker
                casgstatus(gp, _Gwaiting, _Grunnable)
                return gp, false
        }

        // 再次检查垃圾回收状态。
        if sched.gcwaiting != 0 || _g_.m.p.ptr().runSafePointFn != 0 {
                goto top
        }

        // 再次尝试全局队列。
        if sched.runqsize != 0 {
                gp := globrunqget(_g_.m.p.ptr(), 0)
                return gp, false
        }

        // 释放当前 P，取消自旋状态。
        _p_ := releasep()
        pidleput(_p_)
        if _g_.m.spinning {
                _g_.m.spinning = false
                xadd(&sched.nmspinning, -1)
        }

        // 再次检查所有 P 任务队列。
        for i := 0; i < int(gomaxprocs); i++ {
                _p_ := allp[i]
                if _p_ != nil && !runqempty(_p_) {
                        // 绑定一个空闲 P，回到头部尝试偷取任务。
                        _p_ = pidleget()
                        if _p_ != nil {
```

```
                        acquirep(_p_)
                        goto top
                }
                break
        }
    }

    // 再次检查 netpoll。
    if netpollinited() && xchg64(&sched.lastpoll, 0) != 0 {
        gp := netpoll(true) // block until new work is available
        atomicstore64(&sched.lastpoll, uint64(nanotime()))
        if gp != nil {
            _p_ = pidleget()
            if _p_ != nil {
                acquirep(_p_)
                injectglist(gp.schedlink.ptr())
                casgstatus(gp, _Gwaiting, _Grunnable)
                return gp, false
            }
            injectglist(gp)
        }
    }

    // 一无所得，休眠。
    stopm()
    goto top
}
```

每次看到这里，我都想吐槽一句：这代码就不能改改？

按查找流程，我们依次查看不同优先级的获取方式。首先是本地队列，其中 P.runnext 优先级最高。

proc1.go

```
func runqget(_p_ *p) (gp *g, inheritTime bool) {
    // 优先从 runnext 获取。
    // 循环尝试 cas。为什么用同步操作？因为有可能其他 P 从本地队列偷任务。
    for {
        next := _p_.runnext
        if next == 0 {
            break
        }
        if _p_.runnext.cas(next, 0) {
            return next.ptr(), true
        }
    }

    // 本地队列。
    for {
```

```
            h := atomicload(&_p_.runqhead)
            t := _p_.runqtail
            if t == h {
                    return nil, false
            }

            // 从头部提取。
            gp := _p_.runq[h%uint32(len(_p_.runq))]
            if cas(&_p_.runqhead, h, h+1) { // cas-release, commits consume
                    return gp, false
            }
    }
}
```

---

runnext 不会影响 schedtick 计数，也就是说让 schedule 执行更多的任务才会去检查全局队列，所以才会有 inheritTime = true 的说法。

在检查全局队列时，除返回一个可用 G 外，还会批量转移一批到 P 本地队列，毕竟不能每次加锁去操作全局队列。

proc1.go

```
func globrunqget(_p_ *p, max int32) *g {
    if sched.runqsize == 0 {
            return nil
    }

    // 将全局队列任务等分，计算最多能批量获取的任务数量。
    n := sched.runqsize/gomaxprocs + 1
    if n > sched.runqsize {
            n = sched.runqsize
    }
    if max > 0 && n > max {
            n = max
    }

    // 不能超过 runq 数组长度的一半（128）。
    if n > int32(len(_p_.runq))/2 {
            n = int32(len(_p_.runq)) / 2
    }

    // 调整计数。
    sched.runqsize -= n
    if sched.runqsize == 0 {
            sched.runqtail = 0
    }

    // 返回第一个 G 任务，随后的才是要批量转移到本地的任务。
    gp := sched.runqhead.ptr()
    sched.runqhead = gp.schedlink
```

```
    n--
    for ; n > 0; n-- {
        gp1 := sched.runqhead.ptr()
        sched.runqhead = gp1.schedlink
        runqput(_p_, gp1, false)
    }

    return gp
}
```

只有当本地和全局队列都为空时，才会考虑去检查其他 P 任务队列。这个优先级最低，因为会影响目标 P 的执行（必须使用原子操作）。

proc1.go

```
func runqsteal(_p_, p2 *p, stealRunNextG bool) *g {
    t := _p_.runqtail

    // 尝试从 p2 偷取一半任务存入 p 本地队列。
    n := runqgrab(p2, &_p_.runq, t, stealRunNextG)
    if n == 0 {
        return nil
    }

    // 返回尾部的 G 任务。
    n--
    gp := _p_.runq[(t+n)%uint32(len(_p_.runq))]
    if n == 0 {
        return gp
    }

    // 调整目标队列尾部状态。
    atomicstore(&_p_.runqtail, t+n)

    return gp
}

func runqgrab(_p_ *p, batch *[256]*g, batchHead uint32, stealRunNextG bool) uint32 {
    for {
        // 计算批量转移任务数量。
        h := atomicload(&_p_.runqhead)
        t := atomicload(&_p_.runqtail)
        n := t - h
        n = n - n/2

        // 如果没有，那就尝试偷 runnext 吧。
        if n == 0 {
            if stealRunNextG {
                if next := _p_.runnext; next != 0 {
                    usleep(100)
                    if !_p_.runnext.cas(next, 0) {
                        continue
```

```
                                }
                                batch[batchHead%uint32(len(batch))] = next.ptr()
                                return 1
                        }
                }
                return 0
        }

        // 数据异常，不可能超过一半值。重试。
        if n > uint32(len(_p_.runq)/2) { // read inconsistent h and t
                continue
        }

        // 转移任务。
        for i := uint32(0); i < n; i++ {
                g := _p_.runq[(h+i)%uint32(len(_p_.runq))]
                batch[(batchHead+i)%uint32(len(batch))] = g
        }

        // 修改源 P 队列状态。
        // 失败重试。因为没有修改源和目标队列位置状态，所以没有影响。
        if cas(&_p_.runqhead, h, h+n) { // cas-release, commits consume
                return n
        }
    }
}
```

这就是某份官方文档里提及的 Work-Stealing 算法。

lockedg

在执行 cgo 调用时，会用 lockOSThread 将 G 锁定在当前线程。

cgocall.go

```
func cgocall(fn, arg unsafe.Pointer) int32 {
    /*
     * Lock g to m to ensure we stay on the same stack if we do a
     * cgo callback. Add entry to defer stack in case of panic.
     */
    lockOSThread()
    mp := getg().m
    mp.ncgocall++
    mp.ncgo++
    defer endcgo(mp)
}

func endcgo(mp *m) {
    mp.ncgo--
```

```
    unlockOSThread() // invalidates mp
}
```

锁定操作很简单，只需设置 G.lockedm 和 M.lockedg 即可。

proc.go

```
func lockOSThread() {
    getg().m.locked += _LockInternal
    dolockOSThread()
}

func dolockOSThread() {
    _g_ := getg()
    _g_.m.lockedg = _g_
    _g_.lockedm = _g_.m
}
```

当调度函数 schedule 检查到 locked 属性时，会适时移交，让正确的 M 去完成任务。

简单点说，就是 lockedm 会休眠，直到某人将 lockedg 交给它。而不幸拿到 lockedg 的 M，则要将 lockedg 连同 P 一起传递给 lockedm，还负责将其唤醒。至于自己，则因失去 P 被迫休眠，直到 wakep 带着新的 P 唤醒它。

proc1.go

```
func schedule() {
    _g_ := getg()

    // 如果当前 M 是 lockedm，那么休眠。
    // 没有立即 execute(lockedg)，是因为该 lockedg 此时可能被其他 M 获取。
    //   兴许是中途用 gosched 暂时让出 P，进入待运行队列。
    if _g_.m.lockedg != nil {
            stoplockedm()
            execute(_g_.m.lockedg, false) // Never returns.
    }

top:
    ...

    // 如果获取到的 G 是 lockedg，那么将其连同 P 交给 lockedm 去执行。
    // 休眠，等待唤醒后重新获取可用 G。
    if gp.lockedm != nil {
            startlockedm(gp)
            goto top
    }
```

```
    // 执行 goroutine 任务函数。
    execute(gp, inheritTime)
}

func startlockedm(gp *g) {
    _g_ := getg()
    mp := gp.lockedm

    // 移交 P，并唤醒 lockedm。
    _p_ := releasep()
    mp.nextp.set(_p_)
    notewakeup(&mp.park)

    // 当前 M 休眠。
    stopm()
}
```

从中可以看出，除 lockedg 只能由 lockedm 执行外，lockedm 在完成任务或主动解除锁定前也不会执行其他任务。这也是在前面章节我们用 cgo 生成大量 M 实例的原因。

proc1.go

```
func goexit0(gp *g) {
    _g_ := getg()

    // 解除锁定设置。
    gp.m = nil
    gp.lockedm = nil
    _g_.m.lockedg = nil
}
```

可调用 UnlockOSThread 主动解除锁定，以便允许其他 M 完成当前任务。

proc1.go

```
func unlockOSThread() {
    _g_ := getg()
    if _g_.m.locked < _LockInternal {
            systemstack(badunlockosthread)
    }
    _g_.m.locked -= _LockInternal
    dounlockOSThread()
}

func dounlockOSThread() {
    _g_ := getg()
    if _g_.m.locked != 0 {
            return
    }
```

```
    _g_.m.lockedg = nil
    _g_.lockedm = nil
}
```

6. 连续栈

历经 Golang 1.3、1.4 两个版本的过渡，连续栈（Contiguous Stack）的地位已经稳固。而且 1.5 和 1.4 比起来，似乎也没太多的变化，这是个好现象。

连续栈将调用堆栈（call stack）所有栈帧分配在一个连续内存空间。当空间不足时，另分配 2x 内存块，并拷贝当前栈全部数据，以避免分段栈（Segmented Stack）链表结构在函数调用频繁时可能引发的切分热点（hot split）问题。

结构示意图：

```
lo                stackguard0                                                 hi
+-----------------+-------------------------------------------------------------+
| StackGuard      |                                                             |
+-----------------+-------------------------------------------------------------+
                                                                    <-- SP
```

runtime2.go

```
type stack struct {
    lo uintptr
    hi uintptr
}

type g struct {
    // Stack parameters.
    // stack describes the actual stack memory: [stack.lo, stack.hi).
    // stackguard0 is the stack pointer compared in the Go stack growth prologue.
    // It is stack.lo+StackGuard normally, but can be StackPreempt to trigger a preemption.
    stack       stack
    stackguard0 uintptr
}
```

其中 stackguard0 是个非常重要的指针。在函数头部，编译器会插入一段指令，用它和 SP 寄存器进行比较，从而决定是否需要对栈空间扩容。另外，它还被用作抢占调度标志。

栈空间的初始分配发生在 newproc1 创建新 G 对象时。

stack2.go

```
// 操作系统需要保留的区域，比如用来处理信号等等。
_StackSystem = goos_windows*512*ptrSize + goos_plan9*512 + goos_darwin*goarch_arm*1024

// 默认栈大小。
_StackMin = 2048

// StackGuard 是一个警戒指针，用来判断栈容量是否需要扩张。
_StackGuard = 640*stackGuardMultiplier + _StackSystem
```

几个相关常量值，以 Linux 系统为例，_StackSystem = 0，_StackGuard = 640。

---

proc1.go

```
func newproc1(fn *funcval, argp *uint8, narg int32, nret int32, callerpc uintptr) *g {
     newg := gfget(_p_)
     if newg == nil {
            newg = malg(_StackMin)
     }
}

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
}
```

在获取栈空间后，会立即设置 stackguard0 指针。

stackcache

因栈空间使用频繁，所以采取了和 cache/object 类似的做法，就是按大小分成几个等级进行缓存复用，当然也包括回收过多的闲置块。

以 Linux 为例，_FixedStack 大小和 _StackMin 相同，_NumStackOrders 等于 4。

stack2.go

```
// The minimum stack size to allocate.
// The hackery here rounds FixedStack0 up to a power of 2.
_FixedStack0 = _StackMin + _StackSystem
_FixedStack1 = _FixedStack0 - 1
_FixedStack2 = _FixedStack1 | (_FixedStack1 >> 1)
_FixedStack3 = _FixedStack2 | (_FixedStack2 >> 2)
_FixedStack4 = _FixedStack3 | (_FixedStack3 >> 4)
_FixedStack5 = _FixedStack4 | (_FixedStack4 >> 8)
_FixedStack6 = _FixedStack5 | (_FixedStack5 >> 16)
_FixedStack  = _FixedStack6 + 1
```

malloc.go

```
// Number of orders that get caching.  Order 0 is FixedStack
// and each successive order is twice as large.
// We want to cache 2KB, 4KB, 8KB, and 16KB stacks.  Larger stacks
// will be allocated directly.
// Since FixedStack is different on different systems, we
// must vary NumStackOrders to keep the same maximum cached size.
//   OS               | FixedStack | NumStackOrders
//   -----------------+------------+---------------
//   linux/darwin/bsd | 2KB        | 4
//   windows/32       | 4KB        | 3
//   windows/64       | 8KB        | 2
//   plan9            | 4KB        | 3
_NumStackOrders = 4 - ptrSize/4*goos_windows - 1*goos_plan9
```

基于同样的性能考虑（无锁分配），栈空间被缓存在 Cache.stackcache 数组，且使用方法和 object 基本相同。

mcache.go

```
type mcache struct {
    stackcache [_NumStackOrders]stackfreelist
}

type stackfreelist struct {
    list gclinkptr   // linked list of free stacks
    size uintptr     // total size of stacks in list
}
```

在获取栈空间时，优先检查缓存链表。大空间直接从 heap 分配。

malloc.go

```
// Per-P, per order stack segment cache size.
_StackCacheSize = 32 * 1024
```

stack1.go

```
func stackalloc(n uint32) (stack, []stkbar) {
    var v unsafe.Pointer

    // 检查是否从缓存分配。
    if stackCache != 0 && n < _FixedStack<<_NumStackOrders && n < _StackCacheSize {
            // 计算 order 等级。
            order := uint8(0)
            n2 := n
            for n2 > _FixedStack {
                    order++
                    n2 >>= 1
            }

            var x gclinkptr
            c := thisg.m.mcache

            // 从对应链表提取复用空间。
            x = c.stackcache[order].list

            // 提取失败，扩容后重试。
            if x.ptr() == nil {
                    stackcacherefill(c, order)
                    x = c.stackcache[order].list
            }

            // 调整缓存链表。
            c.stackcache[order].list = x.ptr().next
            c.stackcache[order].size -= uintptr(n)

            v = (unsafe.Pointer)(x)
    } else {
            // 大空间直接从 heap 分配。
            s := mHeap_AllocStack(&mheap_, round(uintptr(n), _PageSize)>>_PageShift)
            v = (unsafe.Pointer)(s.start << _PageShift)
    }

    top := uintptr(n) - nstkbar
    stkbarSlice := slice{add(v, top), 0, maxstkbar}
    return stack{uintptr(v), uintptr(v) + top}, *(*[]stkbar)(unsafe.Pointer(&stkbarSlice))
}
```

这个函数代码删除较多，主要是为了不影响阅读。stackpoolalloc 下面一样会有介绍。

和前文内存分配器的做法很像不是吗？我们继续看看如何扩容。

stack1.go

```
func stackcacherefill(c *mcache, order uint8) {
    var list gclinkptr
    var size uintptr

    // 提取一批复用空间。
    for size < _StackCacheSize/2 {
        // 每次提取一个。
        x := stackpoolalloc(order)
        x.ptr().next = list
        list = x
        size += _FixedStack << order
    }

    // 保存到 cache.stackcache 数组。
    c.stackcache[order].list = list
    c.stackcache[order].size = size
}
```

有个全局缓存 stackpool 似乎在充当 central 的角色。

stack1.go

```
var stackpool [_NumStackOrders]mspan

func stackpoolalloc(order uint8) gclinkptr {
    // 尝试从全局缓存获取。
    list := &stackpool[order]
    s := list.next

    // 重新从 heap 获取 span 切分。
    if s == list {
        s = mHeap_AllocStack(&mheap_, _StackCacheSize>>_PageShift)
        for i := uintptr(0); i < _StackCacheSize; i += _FixedStack << order {
            x := gclinkptr(uintptr(s.start)<<_PageShift + i)
            x.ptr().next = s.freelist
            s.freelist = x
        }
        mSpanList_Insert(list, s)
    }

    // 从链表返回一个空间。
    x := s.freelist
    s.freelist = x.ptr().next
    s.ref++
```

```
    // 如果当前链表已空，则移除 span。
    if s.freelist.ptr() == nil {
        // all stacks in s are allocated.
        mSpanList_Remove(s)
    }

    return x
}
```

从 heap 获取 span 的过程没有任何惊喜。

mheap.go

```
func mHeap_AllocStack(h *mheap, npage uintptr) *mspan {
    s := mHeap_AllocSpanLocked(h, npage)
    if s != nil {
        s.state = _MSpanStack
        s.freelist = 0
        s.ref = 0
    }
    return s
}
```

简单总结一下：栈内存一样从 arena 区域分配，使用和对象分配相同的策略和算法。只是我有些不明白，这东西是不是可以做到内存分配器里面？还是说为了以后修改方便才独立出来的？

morestack

执行函数前，需要为其准备好所需栈帧空间，此时是检查连续栈是否需要扩容的最佳时机。为此，编译器会在函数头部插入几条特殊指令，通过比较 stackguard0 和 SP 来决定是否进行扩容操作。

test.go

```
package main

func test() {
   println("hello")
}

func main() {
    test()
}
```

编译（禁用内联），反汇编。

```
$ go build -gcflags "-l" -o test test.go

$ go tool objdump -s "main\.test" test

TEXT main.test(SB) test.go
    test.go:3   0x2040  GS MOVQ GS:0x8a0, CX         // 当前 G。
    test.go:3   0x2049  CMPQ 0x10(CX), SP            // G+0x10 指向 g.stackguard0，和 SP 比较。
    test.go:3   0x204d  JBE 0x2080                   // 如果 SP <= stackguard0，则跳转到 0x2080。
    test.go:3   0x204f  SUBQ $0x10, SP               // 预留当前栈帧空间。
    test.go:4   0x2053  CALL runtime.printlock(SB)
    test.go:4   0x2058  LEAQ 0x6b4f9(IP), BX
    test.go:4   0x205f  MOVQ BX, 0(SP)
    test.go:4   0x2063  MOVQ $0x5, 0x8(SP)
    test.go:4   0x206c  CALL runtime.printstring(SB)
    test.go:4   0x2071  CALL runtime.printnl(SB)
    test.go:4   0x2076  CALL runtime.printunlock(SB)
    test.go:5   0x207b  ADDQ $0x10, SP
    test.go:5   0x207f  RET
    test.go:3   0x2080  CALL runtime.morestack_noctxt(SB)   // 执行 morestack 扩容。
    test.go:3   0x2085  JMP main.test(SB)                   // 扩容结束后，重新执行当前函数。
```

这几条指令很简单。如果 SP 指针地址小于 stackguard0（栈从高位地址向低位分配），那么显然已经溢出，这就需要扩容，否则当前和后续函数就无从分配栈帧内存。

细心一点，你会发现 CMP 指令并没将当前栈帧所需空间算上。假如 SP 大于 stackguard0，但相差又小于当前栈帧大小呢？这显然不会跳转执行扩容操作，但又不能满足当前函数需求，只能眼看着堆栈溢出？

我们知道在 stack.lo 和 stackguard0 之间尚有部分保留空间，所以适当“溢出”是允许的。

stack2.go

```
// After a stack split check the SP is allowed to be this many bytes below the stack guard.
// This saves an instruction in the checking sequence for tiny frames.
_StackSmall = 128
```

修改一下测试代码，看看效果。

test.go

```
package main
```

```
func test1() {
    var x [128]byte
    x[1] = 1
}

func test2() {
    var x [129]byte
    x[1] = 1
}

func main() {
    test1()
    test2()
}
```

```
$ go build -gcflags "-l" -o test test.go

$ go tool objdump -s "main\.test" test

TEXT main.test1(SB) test.go
    test.go:3   0x2040  GS MOVQ GS:0x8a0, CX
    test.go:3   0x2049  CMPQ 0x10(CX), SP     // 当前栈帧 0x80 正好是 128。
    test.go:3   0x204d  JBE 0x206e
    test.go:3   0x204f  SUBQ $0x80, SP

TEXT main.test2(SB) test.go
    test.go:8   0x2080  GS MOVQ GS:0x8a0, CX
    test.go:8   0x2089  LEAQ -0x8(SP), AX     // 当前栈帧 0x88 - 128 = 0x8，适当调整 SP 后再比较。
    test.go:8   0x208e  CMPQ 0x10(CX), AX
    test.go:8   0x2092  JBE 0x20b5
    test.go:8   0x2094  SUBQ $0x88, SP
```

很显然，如果当前栈帧是 SmallStack（0x80），那么就允许在 [lo, stackguard0] 之间分配。

对栈扩容并不是件容易的事情，其中涉及很多内容。不过呢，在这里我们只需了解其基本过程和算法意图，无须深入到所有细节。

asm_amd64.s

```
TEXT runtime·morestack_noctxt(SB),NOSPLIT,$0
     MOVL    $0, DX
     JMP     runtime·morestack(SB)

TEXT runtime·morestack(SB),NOSPLIT,$0-0
     // Call newstack on m->g0's stack.
     MOVQ    m_g0(BX), BX
     MOVQ    BX, g(CX)
     MOVQ    (g_sched+gobuf_sp)(BX), SP
```

```
    CALL    runtime·newstack(SB)
    MOVQ    $0, 0x1003     // crash if newstack returns
    RET
```

基本过程就是分配一个 2x 大小的新栈，然后将数据拷贝过去，替换掉旧栈。当然，这期间需要对指针等内容做些调整。

stack1.go

```
func newstack() {
    thisg := getg()
    gp := thisg.m.curg

    // 调整执行现场记录。
    rewindmorestack(&gp.sched)

    casgstatus(gp, _Grunning, _Gwaiting)
    gp.waitreason = "stack growth"

    sp := gp.sched.sp

    // 扩张 2 倍。
    oldsize := int(gp.stackAlloc)
    newsize := oldsize * 2

    casgstatus(gp, _Gwaiting, _Gcopystack)

    // 拷贝栈数据后切换到新栈。
    copystack(gp, uintptr(newsize))

    // 恢复执行。
    casgstatus(gp, _Gcopystack, _Grunning)
    gogo(&gp.sched)
}

func copystack(gp *g, newsize uintptr) {
    old := gp.stack
    used := old.hi - gp.sched.sp

    // 从缓存或堆分配新栈空间。
    new, newstkbar := stackalloc(uint32(newsize))

    // 清零。
    if stackPoisonCopy != 0 {
        fillstack(new, 0xfd)
    }

    // 调整指针等操作 ...

    // 拷贝数据到新栈空间。
    memmove(unsafe.Pointer(new.hi-used), unsafe.Pointer(old.hi-used), used)
```

```
    // 切换到新栈。
    gp.stack = new
    gp.stackguard0 = new.lo + _StackGuard
    gp.sched.sp = new.hi - used
    oldsize := gp.stackAlloc
    gp.stackAlloc = newsize
    gp.stkbar = newstkbar

    // 将旧栈清零后释放。
    if stackPoisonCopy != 0 {
            fillstack(old, 0xfc)
    }
    stackfree(old, oldsize)
}
```

stackfree

释放栈空间的操作，依旧与回收 object 类似。

stack1.go

```
func stackfree(stk stack, n uintptr) {
    gp := getg()
    v := (unsafe.Pointer)(stk.lo)

    // 放回缓存链表。
    if stackCache != 0 && n < _FixedStack<<_NumStackOrders && n < _StackCacheSize {
            // 计算 order 等级。
            order := uint8(0)
            n2 := n
            for n2 > _FixedStack {
                    order++
                    n2 >>= 1
            }
            x := gclinkptr(v)
            c := gp.m.mcache

            // 如果缓存大小超出限制，则释放一些。
            if c.stackcache[order].size >= _StackCacheSize {
                    stackcacherelease(c, order)
            }

            // 放回缓存链表。
            x.ptr().next = c.stackcache[order].list
            c.stackcache[order].list = x
            c.stackcache[order].size += n
    } else {
            s := mHeap_Lookup(&mheap_, v)
            if gcphase == _GCoff {
```

```
                        // 归还给 heap。
                        mHeap_FreeStack(&mheap_, s)
                } else {
                        // 如果正在垃圾回收期间，那么放到一个待处理队列，由垃圾回收器处理。
                        mSpanList_Insert(&stackFreeQueue, s)
                }
        }
}
```

回收的栈空间被放回对应复用链表。如缓存过多，则转移一批到全局链表，或直接将自由的 span 归还给 heap。

```
func stackcacherelease(c *mcache, order uint8) {
        x := c.stackcache[order].list
        size := c.stackcache[order].size

        // 如果当前链表过大，则释放一半。
        for size > _StackCacheSize/2 {
                y := x.ptr().next

                // 每次释放一个，它们可能属于不同的 span。
                stackpoolfree(x, order)

                x = y
                size -= _FixedStack << order
        }

        c.stackcache[order].list = x
        c.stackcache[order].size = size
}

func stackpoolfree(x gclinkptr, order uint8) {
        // 找到所属 span。
        s := mHeap_Lookup(&mheap_, (unsafe.Pointer)(x))
        if s.freelist.ptr() == nil {
                mSpanList_Insert(&stackpool[order], s)
        }

        // 添加到 span.freelist。
        x.ptr().next = s.freelist
        s.freelist = x
        s.ref--

        // 如果该 span 已收回全部空间，那么将其归还给 heap。
        if gcphase == _GCoff && s.ref == 0 {
                mSpanList_Remove(s)
                s.freelist = 0
                mHeap_FreeStack(&mheap_, s)
        }
}
```

除 morestack 调用导致 stackfree 操作外，另一原因就是垃圾回收对栈空间的收缩处理。

mgcmark.go

```
func markroot(desc *parfor, i uint32) {
    switch i {
    case _RootFlushCaches:
            if gcphase != _GCscan {
                    flushallmcaches()
            }
    default:
            if gcphase == _GCmarktermination {
                    shrinkstack(gp)
            }
    }
}
```

mstats.go

```
func flushallmcaches() {
    for i := 0; ; i++ {
            p := allp[i]
            c := p.mcache
            mCache_ReleaseAll(c)
            stackcache_clear(c)
    }
}
```

mgc.go

```
func gcMark(start_time int64) {
    freeStackSpans()
}
```

因垃圾回收需要，stackcache_clear 会将所有 cache 缓存的栈空间归还给全局或 heap。

stack1.go

```
func stackcache_clear(c *mcache) {
    for order := uint8(0); order < _NumStackOrders; order++ {
            x := c.stackcache[order].list
            for x.ptr() != nil {
                    y := x.ptr().next
                    stackpoolfree(x, order)
                    x = y
            }
```

```
                c.stackcache[order].list = 0
                c.stackcache[order].size = 0
        }
}
```

而 shrinkstack 主要目的是收缩那些曾经扩容的栈空间，以节约内存。

```
func shrinkstack(gp *g) {
        if readgstatus(gp) == _Gdead {
                if gp.stack.lo != 0 {
                        // 回收 G 的栈空间，重新使用前会为其补上。
                        stackfree(gp.stack, gp.stackAlloc)
                        gp.stack.lo = 0
                        gp.stack.hi = 0
                        gp.stkbar = nil
                        gp.stkbarPos = 0
                }
                return
        }

        // 收缩目标是一半大小。
        oldsize := gp.stackAlloc
        newsize := oldsize / 2
        if newsize < _FixedStack {
                return
        }

        // 如果使用空间超过 1/4，则不收缩。
        avail := gp.stack.hi - gp.stack.lo
        if used := gp.stack.hi - gp.sched.sp + _StackLimit; used >= avail/4 {
                return
        }

        // 用较小的栈替换。
        oldstatus := casgcopystack(gp)
        copystack(gp, newsize)
        casgstatus(gp, _Gcopystack, oldstatus)
}
```

最后就是 freeStackSpans，它扫描全局队列 stackpool 和暂存队列 stackFreeQueue，将那些空间已完全收回的 span 交还给 heap。

stack1.go

```
func freeStackSpans() {
        for order := range stackpool {
                list := &stackpool[order]
                for s := list.next; s != list; {
                        next := s.next
```

```
                    if s.ref == 0 {
                            mSpanList_Remove(s)
                            s.freelist = 0
                            mHeap_FreeStack(&mheap_, s)
                    }
                    s = next
            }
    }

    for stackFreeQueue.next != &stackFreeQueue {
            s := stackFreeQueue.next
            mSpanList_Remove(s)
            mHeap_FreeStack(&mheap_, s)
    }
}
```

另外，调整 P 数量的 procresize，将任务完成的 G 对象放回复用链表的 gfput 同样会引发栈空间释放操作。只是流程和上述基本类似，不再赘述。

> 运行时三大核心组件之间，相互纠缠太多太细，已无从划分边界，我个人觉得这并不是什么好主意。诚然为了性能，很多地方直接植入代码，而非通过消息或接口隔离等方式封装，但随着各部件复杂度和规模的提升，其可维护性也必然降低。不知道开发团队对此有什么具体的想法。

7. 系统调用

为支持并发调度，专门对 syscall、cgo 进行了包装，以便在长时间阻塞时能切换执行其他任务。标准库 syscall 包里，将相关系统调用函数分为 Syscall 和 RawSyscall 两类。

src/syscall/zsyscall_linux_amd64.s

```
func Getcwd(buf []byte) (n int, err error) {
    r0, _, e1 := Syscall(SYS_GETCWD, uintptr(_p0), uintptr(len(buf)), 0)
}

func EpollCreate(size int) (fd int, err error) {
    r0, _, e1 := RawSyscall(SYS_EPOLL_CREATE, uintptr(size), 0, 0)
}
```

让我们看看这两者有什么区别。

src/syscall/asm_linux_amd64.s

```
TEXT ·Syscall(SB),NOSPLIT,$0-56
```

```
    CALL    runtime·entersyscall(SB)
    MOVQ    trap+0(FP), AX                // syscall entry
    SYSCALL
    JLS     ok
    CALL    runtime·exitsyscall(SB)
    RET
ok:
    CALL    runtime·exitsyscall(SB)
    RET


TEXT ·RawSyscall(SB),NOSPLIT,$0-56
    MOVQ    trap+0(FP), AX                // syscall entry
    SYSCALL
    JLS     ok1
    RET
ok1:
    RET
```

最大的不同在于 Syscall 增加了 entrysyscall/exitsyscall，这就是允许调度的关键所在。

proc1.go

```
func entersyscall(dummy int32) {
    reentersyscall(getcallerpc(unsafe.Pointer(&dummy)), getcallersp(unsafe.Pointer(&dummy)))
}

func reentersyscall(pc, sp uintptr) {
    _g_ := getg()

    // 保存执行现场。
    save(pc, sp)

    _g_.syscallsp = sp
    _g_.syscallpc = pc
    casgstatus(_g_, _Grunning, _Gsyscall)

    // 确保 sysmon 运行。
    if atomicload(&sched.sysmonwait) != 0 {
            systemstack(entersyscall_sysmon)
            save(pc, sp)
    }

    // 设置相关状态。
    _g_.m.syscalltick = _g_.m.p.ptr().syscalltick
    _g_.sysblocktraced = true
    _g_.m.mcache = nil
    _g_.m.p.ptr().m = 0
    atomicstore(&_g_.m.p.ptr().status, _Psyscall)
}
```

监控线程 sysmon 对 syscall 非常重要，因为它负责将因系统调用而长时间阻塞的 P 抢回，用于执行其他任务。否则，整体性能会严重下降，甚至整个进程被冻结。

proc1.go

```
func entersyscall_sysmon() {
    if atomicload(&sched.sysmonwait) != 0 {
            atomicstore(&sched.sysmonwait, 0)
            notewakeup(&sched.sysmonnote)
    }
}
```

某些系统调用本身就可以确定长时间阻塞（比如锁），那么它会选择执行 entersyscallblock 主动交出所关联的 P。

proc1.go

```
func entersyscallblock(dummy int32) {
    casgstatus(_g_, _Grunning, _Gsyscall)
    systemstack(entersyscallblock_handoff)
}

func entersyscallblock_handoff() {
    // 释放 P，让它去执行其他任务。
    handoffp(releasep())
}

func handoffp(_p_ *p) {
    // 如果 P 本地或全局有任务，直接唤醒某个 M 开始工作。
    if !runqempty(_p_) || sched.runqsize != 0 {
            startm(_p_, false)
            return
    }

    ...

    // 没有任务就放回空闲队列。
    pidleput(_p_)
}
```

从系统调用返回时，必须检查 P 是否依然可用，因为可能已被 sysmon 抢走。

proc1.go

```
func exitsyscall(dummy int32) {
    _g_ := getg()
    oldp := _g_.m.p.ptr()
```

```
    if exitsyscallfast() {
            casgstatus(_g_, _Gsyscall, _Grunning)
            return
    }

    mcall(exitsyscall0)
}
```

快速退出 exitsyscallfast 是指能重新绑定原有或空闲的 P，以继续当前 G 任务执行。

proc1.go

```
func exitsyscallfast() bool {
    _g_ := getg()

    // STW 状态，就不要继续了。
    if sched.stopwait == freezeStopWait {
            _g_.m.mcache = nil
            _g_.m.p = 0
            return false
    }

    // 尝试关联原本的 P。
    if _g_.m.p != 0 && _g_.m.p.ptr().status == _Psyscall &&
       cas(&_g_.m.p.ptr().status, _Psyscall, _Prunning) {
            _g_.m.mcache = _g_.m.p.ptr().mcache
            _g_.m.p.ptr().m.set(_g_.m)
            return true
    }

    // 获取其他空闲 P。
    oldp := _g_.m.p.ptr()
    _g_.m.mcache = nil
    _g_.m.p = 0
    if sched.pidle != 0 {
            var ok bool
            systemstack(func() {
                    ok = exitsyscallfast_pidle()
            })
            if ok {
                    return true
            }
    }
    return false
}

func exitsyscallfast_pidle() bool {
    _p_ := pidleget()

    // 唤醒 sysmon。
    if _p_ != nil && atomicload(&sched.sysmonwait) != 0 {
```

```
                atomicstore(&sched.sysmonwait, 0)
                notewakeup(&sched.sysmonnote)
        }

        // 重新关联。
        if _p_ != nil {
                acquirep(_p_)
                return true
        }
        return false
}
```

如果多次尝试绑定 P 失败，那么只能将当前任务放入待运行队列。

proc1.go

```
func exitsyscall0(gp *g) {
        _g_ := getg()

        // 修改状态，解除和 M 的关联。
        casgstatus(gp, _Gsyscall, _Grunnable)
        dropg()

        // 再次获取空闲 P。
        _p_ := pidleget()
        if _p_ == nil {
                // 获取失败，放回全局任务队列。
                globrunqput(gp)
        } else if atomicload(&sched.sysmonwait) != 0 {
                atomicstore(&sched.sysmonwait, 0)
                notewakeup(&sched.sysmonnote)
        }

        // 再次检查 P，以便执行当前任务。
        if _p_ != nil {
                acquirep(_p_)
                execute(gp, false) // Never returns.
        }

        // 关联 P 失败，休眠当前 M。
        stopm()
        schedule() // Never returns.
}
```

需要注意，cgo 使用了相同的封装方式，因为它同样不受调度器管理。

cgocall.go

```
func cgocall(fn, arg unsafe.Pointer) int32 {
```

```
    /*
     * Announce we are entering a system call
     * so that the scheduler knows to create another M to run goroutines while we are in
     * the foreign code.
     *
     * The call to asmcgocall is guaranteed not to split the stack and does not allocate
     * memory, so it is safe to call while "in a system call", outside the $GOMAXPROCS
     * accounting.
     */
    entersyscall(0)
    errno := asmcgocall(fn, arg)
    exitsyscall(0)
}
```

8. 监控

系统监控线程我们在前面已经介绍过好几回了，现在对它做个总结。

- 释放闲置超过 5 分钟的 span 物理内存。
- 如果超过 2 分钟没有垃圾回收，强制执行。
- 将长时间未处理的 netpoll 结果添加到任务队列。
- 向长时间运行的 G 任务发出抢占调度。
- 收回因 syscall 长时间阻塞的 P。

在进入垃圾回收状态时，sysmon 会自动进入休眠，所以我们才会在 syscall 里看到很多唤醒指令。另外，startTheWorld 也会做唤醒处理。保证监控线程正常运行，对内存分配、垃圾回收和并发调度都非常重要。

proc1.go

```
func startTheWorldWithSema() {
    sched.gcwaiting = 0
    if sched.sysmonwait != 0 {
            sched.sysmonwait = 0
            notewakeup(&sched.sysmonnote)
    }
}
```

现在，让我们忽略其他任务，看看对 syscall 和 preempt 的处理。

proc1.go

```
func sysmon() {
```

```
    for {
            usleep(delay)

            // STW 时休眠 sysmon。
            if debug.schedtrace <= 0 &&
              (sched.gcwaiting != 0 || atomicload(&sched.npidle) == uint32(gomaxprocs)) {
                  if atomicload(&sched.gcwaiting) != 0 ||
                     atomicload(&sched.npidle) == uint32(gomaxprocs) {
                          // 设置休眠标志，休眠（有个超时，苏醒保障）。
                          atomicstore(&sched.sysmonwait, 1)
                          notetsleep(&sched.sysmonnote, maxsleep)

                          // 唤醒后重置状态标志，继续执行。
                          atomicstore(&sched.sysmonwait, 0)
                          noteclear(&sched.sysmonnote)
                  }
            }

            lastpoll := int64(atomicload64(&sched.lastpoll))
            now := nanotime()
            unixnow := unixnanotime()

            // 获取超过 10ms 的 netpoll 结果。
            if lastpoll != 0 && lastpoll+10*1000*1000 < now {
                    cas64(&sched.lastpoll, uint64(lastpoll), uint64(now))
                    gp := netpoll(false) // non-blocking - returns list of goroutines
                    if gp != nil {
                            injectglist(gp)
                    }
            }

            // 抢夺 syscall 长时间阻塞的 P。
            // 向长时间运行的 G 发出抢占调度。
            if retake(now) != 0 {
                    idle = 0
            } else {
                    idle++
            }
    }
}
```

专门有个 pdesc 的全局变量用于保存 sysmon 运行统计信息，据此来判断 syscall 和 G 是否超时。

proc1.go

```
var pdesc [_MaxGomaxprocs]struct {
    schedtick   uint32
    schedwhen   int64
    syscalltick uint32
    syscallwhen int64
}
```

```
const forcePreemptNS = 10 * 1000 * 1000   // 10ms

func retake(now int64) uint32 {
    // 遍历 P。
    for i := int32(0); i < gomaxprocs; i++ {
            _p_ := allp[i]
            pd := &pdesc[i]
            s := _p_.status

            // P 处于 syscall 模式。
            if s == _Psyscall {
                    // 更新 syscall 统计信息。
                    t := int64(_p_.syscalltick)
                    if int64(pd.syscalltick) != t {
                            pd.syscalltick = uint32(t)
                            pd.syscallwhen = now
                            continue
                    }

                    // 检查是否有其他任务需要 P，是否超出时间限制（2 tick, 20us），是否有必要抢夺 P。
                    if runqempty(_p_) &&
                       atomicload(&sched.nmspinning)+atomicload(&sched.npidle) > 0 &&
                       pd.syscallwhen+10*1000*1000 > now {
                            continue
                    }

                    // 抢夺 P。
                    if cas(&_p_.status, s, _Pidle) {
                            _p_.syscalltick++
                            handoffp(_p_)
                    }
            } else if s == _Prunning {
                    // 更新 G 运行统计信息。
                    t := int64(_p_.schedtick)
                    if int64(pd.schedtick) != t {
                            pd.schedtick = uint32(t)
                            pd.schedwhen = now
                            continue
                    }

                    // 如果没超过 10ms，则忽略。
                    if pd.schedwhen+forcePreemptNS > now {
                            continue
                    }

                    // 发出抢占调度。
                    preemptone(_p_)
            }
    }
}
```

抢占调度

所谓抢占调度要比你想象的简单许多，远不是你以为的“抢占式多任务操作系统”那种样子。因为 Golang 调度器并没有真正意义上的时间片概念，只是在目标 G 上设置一个抢占标志，当该任务调用某个函数时，被编译器安插的指令就会检查这个标志，从而决定是否暂停当前任务。

proc1.go

```
// Tell the goroutine running on processor P to stop.
// This function is purely best-effort.  It can incorrectly fail to inform the
// goroutine.  It can send inform the wrong goroutine.  Even if it informs the
// correct goroutine, that goroutine might ignore the request if it is
// simultaneously executing newstack.
// No lock needs to be held.
// Returns true if preemption request was issued.
// The actual preemption will happen at some point in the future
// and will be indicated by the gp->status no longer being
// Grunning
func preemptone(_p_ *p) bool {
    mp := _p_.m.ptr()
    gp := mp.curg
    gp.preempt = true

    // Every call in a go routine checks for stack overflow by
    // comparing the current stack pointer to gp->stackguard0.
    // Setting gp->stackguard0 to StackPreempt folds
    // preemption into the normal stack overflow check.
    gp.stackguard0 = stackPreempt
    return true
}
```

保留这段代码里的注释，只是告诉你，preempt 真的有些不靠谱。

有两个标志，实际起作用的是 G.stackguard0。G.preempt 只是后备，以便在 stackguard0 做回溢出检查标志时，依然可用 preempt 恢复抢占状态。

编译器插入的指令？没错，就是那个 morestack。当它调用 newstack 扩容时会检查抢占标志，并决定是否暂停当前任务，当然这发生在实际扩容之前。

stack1.go

```
func newstack() {
    preempt := atomicloaduintptr(&gp.stackguard0) == stackPreempt

    if preempt {
```

```
            // 如果 M 持有锁，或者正在进行内存分配、垃圾回收等操作，不抢占，留待下次。
            if thisg.m.locks != 0 || thisg.m.mallocing != 0 ||
               thisg.m.preemptoff != "" || thisg.m.p.ptr().status != _Prunning {
                    // stackguard0 恢复溢出检查用途，下次用 G.preempt 恢复。
                    gp.stackguard0 = gp.stack.lo + _StackGuard
                    gogo(&gp.sched) // never return
            }
    }

    if preempt {
            // 垃圾回收本身也算一次抢占，忽略本次抢占调度。
            if gp.preemptscan {
                    for !castogscanstatus(gp, _Gwaiting, _Gscanwaiting) {
                            // Likely to be racing with the GC as
                            // it sees a _Gwaiting and does the
                            // stack scan. If so, gcworkdone will
                            // be set and gcphasework will simply
                            // return.
                    }
                    if !gp.gcscandone {
                            scanstack(gp)
                            gp.gcscandone = true
                    }
                    gp.preemptscan = false
                    gp.preempt = false
                    casfrom_Gscanstatus(gp, _Gscanwaiting, _Gwaiting)
                    casgstatus(gp, _Gwaiting, _Grunning)
                    gp.stackguard0 = gp.stack.lo + _StackGuard
                    gogo(&gp.sched) // never return
            }

            // 开始抢占调度，将当前 G 放回队列，让 M 执行其他任务。
            casgstatus(gp, _Gwaiting, _Grunning)
            gopreempt_m(gp) // never return
    }

    // Allocate a bigger segment and move the stack.
    copystack(gp, uintptr(newsize))
    gogo(&gp.sched)
}
```

proc1.go

```
func gopreempt_m(gp *g) {
    goschedImpl(gp)
}

func goschedImpl(gp *g) {
    status := readgstatus(gp)
    casgstatus(gp, _Grunning, _Grunnable)
    dropg()
    globrunqput(gp)
```

```
    schedule()
}
```

> 这个抢占调度机制给我的感觉是越来越弱，毕竟垃圾回收和栈扩容这个时机都不是很“确定”和“实时”。更何况还有函数内联和纯算法循环等造成 morestack 不会执行因素。不知道对此后续版本会有何改进。

9. 其他

与任务执行有关的几种暂停操作。

Gosched

可被用户调用的 runtime.Gosched 将当前 G 任务暂停，重新放回全局队列，让出当前 M 去执行其他任务。我们无需对 G 做唤醒操作，因为总归会被某个 M 重新拿到，并从“断点”恢复。

proc.go

```
func Gosched() {
    mcall(gosched_m)
}
```

proc1.go

```
func gosched_m(gp *g) {
    goschedImpl(gp)
}

func goschedImpl(gp *g) {
    // 重置属性。
    casgstatus(gp, _Grunning, _Grunnable)
    dropg()

    // 将当前 G 放回全局队列。
    globrunqput(gp)

    // 重新调度执行其他任务。
    schedule()
}
```

```
func dropg() {
    _g_ := getg()

    if _g_.m.lockedg == nil {
            _g_.m.curg.m = nil
            _g_.m.curg = nil
    }
}
```

实现“断点恢复”的关键由 mcall 实现，它将当前执行状态，包括 SP、PC 寄存器等值保存到 G.sched 区域。

asm_amd64.s

```
TEXT runtime·mcall(SB), NOSPLIT, $0-8
    MOVQ    fn+0(FP), DI

    get_tls(CX)
    MOVQ    g(CX), AX                   // save state in g->sched
    MOVQ    0(SP), BX                   // caller's PC
    MOVQ    BX, (g_sched+gobuf_pc)(AX)
    LEAQ    fn+0(FP), BX                // caller's SP
    MOVQ    BX, (g_sched+gobuf_sp)(AX)
    MOVQ    AX, (g_sched+gobuf_g)(AX)
    MOVQ    BP, (g_sched+gobuf_bp)(AX)

    // switch to m->g0 & its stack, call fn
    ...
```

当 execute/gogo 再次执行该任务时，自然可从中恢复状态。反正执行栈是 G 自带的，不用担心执行数据丢失。

gopark

与 Gosched 最大的区别在于，gopark 并没将 G 放回待运行队列。也就是说，必须主动恢复，否则该任务会遗失。

proc.go

```
func gopark(unlockf func(*g, unsafe.Pointer) bool, lock unsafe.Pointer, reason string, ...) {
    mp := acquirem()
    gp := mp.curg

    mp.waitlock = lock
    mp.waitunlockf = *(*unsafe.Pointer)(unsafe.Pointer(&unlockf))
    gp.waitreason = reason
```

```
    mp.waittraceev = traceEv
    mp.waittraceskip = traceskip
    releasem(mp)

    mcall(park_m)
}
```

同样是由 mcall 保存执行状态，还有个 unlockf 作为暂停判断条件。

proc1.go

```
func park_m(gp *g) {
    _g_ := getg()

    // 重置属性。
    casgstatus(gp, _Grunning, _Gwaiting)
    dropg()

    // 执行解锁函数。如果返回 false，则恢复执行。
    if _g_.m.waitunlockf != nil {
        fn := *(*func(*g, unsafe.Pointer) bool)(unsafe.Pointer(&_g_.m.waitunlockf))
        ok := fn(gp, _g_.m.waitlock)
        _g_.m.waitunlockf = nil
        _g_.m.waitlock = nil
        if !ok {
            casgstatus(gp, _Gwaiting, _Grunnable)
            execute(gp, true) // Schedule it back, never returns.
        }
    }

    // 调度执行其他任务。
    schedule()
}
```

与之配套，goready 用于恢复执行，G 被放回优先级最高的 P.runnext。

proc.go

```
func goready(gp *g, traceskip int) {
    systemstack(func() {
        ready(gp, traceskip)
    })
}
```

proc1.go

```
func ready(gp *g, traceskip int) {
    // 修正状态，重新放回本地 runnext。
    casgstatus(gp, _Gwaiting, _Grunnable)
    runqput(_g_.m.p.ptr(), gp, true)
}
```

notesleep

相比 gosched、gopark，反应更敏捷的 notesleep 既不让出 M，也就不会让 G 重回任务队列。它直接让线程休眠直到被唤醒，更适合 stopm、gcMark 这类近似自旋的场景。

在 linux、dragonfly、freebsd 平台，notesleep 是基于 futex 的高性能实现。

> Futex 通常称作“快速用户区互斥”，是一种在用户空间实现的锁（互斥）机制。多执行单位（进程或线程）通过共享同一快内存（整数）来实现等待和唤醒操作。因为 Futex 只在操作结果不一致时才进入内核仲裁，所以有非常高的执行效率。
>
> 更多内容请参考 man 2 futex。

runtime2.go

```
type m struct {
    park note
}

type note struct {
    // Futex-based impl treats it as uint32 key, while sema-based impl as M* waitm.
    key uintptr
}
```

围绕 note.key 值来处理休眠和唤醒操作。

lock_futex.go

```
func notesleep(n *note) {
    gp := getg()

    for atomicload(key32(&n.key)) == 0 {
            gp.m.blocked = true
            futexsleep(key32(&n.key), 0, -1)      // 检查 n.key == 0，休眠。
            gp.m.blocked = false                  // 唤醒后 n.key == 1。
    }
}
```

```
func notewakeup(n *note) {
    // 如果 old != 0，表示已经执行过唤醒操作。
    old := xchg(key32(&n.key), 1)
    if old != 0 {
        throw("notewakeup - double wakeup")
    }

    // 唤醒后 n.key == 1。
    futexwakeup(key32(&n.key), 1)
}

// 重置休眠条件。
func noteclear(n *note) {
    n.key = 0
}
```

os1_linux.go

```
func futexsleep(addr *uint32, val uint32, ns int64) {
    var ts timespec

    // 不超时。
    if ns < 0 {
        futex(unsafe.Pointer(addr), _FUTEX_WAIT, val, nil, nil, 0)
        return
    }

    ts.set_sec(ns / 1000000000)
    ts.set_nsec(int32(ns % 1000000000))

    // 如果 futex_value == val，则进入休眠等待状态，直到 FUTEX_WAKE 或超时。
    futex(unsafe.Pointer(addr), _FUTEX_WAIT, val, unsafe.Pointer(&ts), nil, 0)
}

func futexwakeup(addr *uint32, cnt uint32) {
    // 唤醒 cnt 个等待单位，这会设置 futex_value = 1。
    ret := futex(unsafe.Pointer(addr), _FUTEX_WAKE, cnt, nil, nil, 0)
}
```

其他不支持 futex 的 darwin、windows 等平台，可参阅 lock_sema.go 基于 semaphore 的实现。

Goexit

用户可调用 runtime.Goexit 立即终止 G 任务，不管当前处于调用堆栈的哪个层次。在终止前，它确保所有 G.defer 被执行。

panic.go

```
func Goexit() {
     gp := getg()
     for {
            d := gp._defer
            ...
            freedefer(d)
     }
     goexit1()
}
```

比较有趣的是在 main goroutine 里执行 Goexit，它会等待其他 goroutine 结束后才会崩溃。

test.go

```
package main

import (
    "fmt"
    "runtime"
    "time"
)

func main() {
    for i := 0; i < 3; i++ {
        go func(n int) {
            time.Sleep(time.Second * time.Duration(n+1))
            fmt.Printf("G%d end.\n", n)
        }(i)
    }

    println("Goexit.")
    runtime.Goexit()
    println("never execute.")
}
```

```
$ go build -o test test.go && ./test

Goexit.
G0 end.
G1 end.
G2 end.

fatal error: no goroutines (main called runtime.Goexit) - deadlock!

runtime stack:
runtime.throw(0x52cdc0, 0x36)
    /usr/local/go/src/runtime/panic.go:527 +0x90
```

```
runtime.checkdead()
    /usr/local/go/src/runtime/proc1.go:2933 +0x1fb
runtime.mput(0xc82002a900)
    /usr/local/go/src/runtime/proc1.go:3268 +0x46
runtime.stopm()
    /usr/local/go/src/runtime/proc1.go:1126 +0xdd
runtime.findrunnable(0xc82001c000, 0x0)
    /usr/local/go/src/runtime/proc1.go:1530 +0x69e
runtime.schedule()
    /usr/local/go/src/runtime/proc1.go:1639 +0x267
runtime.goexit0(0xc820001380)
    /usr/local/go/src/runtime/proc1.go:1765 +0x1a2
runtime.mcall(0x0)
    /usr/local/go/src/runtime/asm_amd64.s:204 +0x5b
```

stopTheWorld

本章的最后，我们看看导致整个进程用户逻辑停止的 STW 是如何实现的。

用户逻辑暂停必须是在一个安全点上，否则会引发很多意外问题。因此，stopTheWorld 同样是通过“通知”机制，让 G 主动停止。比如，设置“gcwaiting = 1”让调度函数 schedule 主动休眠 M；向所有正在运行的 G 任务发出抢占调度，使其暂停。

proc1.go

```
func stopTheWorld(reason string) {
     semacquire(&worldsema, false)
     getg().m.preemptoff = reason
     systemstack(stopTheWorldWithSema)
}

func stopTheWorldWithSema() {
     _g_ := getg()
     sched.stopwait = gomaxprocs

     // 设置停止标志，让 schedule 之类的调用主动休眠 M。
     atomicstore(&sched.gcwaiting, 1)

     // 向所有正在运行的 G 发出抢占调度。
     preemptall()

     // 暂停当前 P。
     _g_.m.p.ptr().status = _Pgcstop
     sched.stopwait--

     // 尝试暂停所有 syscall 状态的 P。
     for i := 0; i < int(gomaxprocs); i++ {
            p := allp[i]
```

```
                s := p.status
                if s == _Psyscall && cas(&p.status, s, _Pgcstop) {
                        p.syscalltick++
                        sched.stopwait--
                }
        }

        // 处理空闲 P。
        for {
                p := pidleget()
                if p == nil {
                        break
                }
                p.status = _Pgcstop
                sched.stopwait--
        }

        wait := sched.stopwait > 0

        // 等待。
        if wait {
                for {
                        // 暂停 100us 后，重新发出抢占调度。
                        // handoffp、gcstopm、entersyscall_gcwait 等操作都会 sched.stopwait--，
                        //   如果 stopwait == 0 则尝试唤醒 stopnote。
                        //   唤醒成功，跳出循环。失败，则重新发出抢占调度，再次等待。
                        if notetsleep(&sched.stopnote, 100*1000) {
                                noteclear(&sched.stopnote)
                                break
                        }
                        preemptall()
                }
        }

        // 检查所有 P 状态。
        for i := 0; i < int(gomaxprocs); i++ {
                p := allp[i]
                if p.status != _Pgcstop {
                        throw("stopTheWorld: not stopped")
                }
        }
}

// 向所有 P 发出抢占调度。
func preemptall() bool {
        res := false
        for i := int32(0); i < gomaxprocs; i++ {
                _p_ := allp[i]
                if _p_ == nil || _p_.status != _Prunning {
                        continue
                }
                if preemptone(_p_) {
                        res = true
                }
        }
```

```
    return res
}
```

总体上看，stopTheWorld 还是很平和的一种手段，会循环等待目标任务进入一个安全点后主动暂停。而 startTheWorld 就更简单，毕竟是从冻结状态开始，无非是唤醒相关 P/M 继续执行任务。

proc1.go

```
func startTheWorld() {
    systemstack(startTheWorldWithSema)
    semrelease(&worldsema)
    getg().m.preemptoff = ""
}

func startTheWorldWithSema() {
    _g_ := getg()

    // 检查是否需要 procresize。
    p1 := procresize(procs)

    // 解除停止状态。
    sched.gcwaiting = 0

    // 唤醒 sysmon。
    if sched.sysmonwait != 0 {
        sched.sysmonwait = 0
        notewakeup(&sched.sysmonnote)
    }

    // 循环有任务的 P 链表，让它们继续工作。
    for p1 != nil {
        p := p1
        p1 = p1.link.ptr()
        if p.m != 0 {
            mp := p.m.ptr()
            p.m = 0
            mp.nextp.set(p)
            notewakeup(&mp.park)
        } else {
            // Start M to run P.  Do not start another M below.
            newm(nil, p)
            add = false
        }
    }

    // 让闲置的家伙都起来工作!
    if atomicload(&sched.npidle) != 0 && atomicload(&sched.nmspinning) == 0 {
        wakep()
    }
```

```
    // 重置抢占标志。
    if _g_.m.locks == 0 && _g_.preempt {
        _g_.stackguard0 = stackPreempt
    }
}
```

七. 通道

通道（channel）是 Golang 实现 CSP 并发模型的关键，鼓励用通讯来实现数据共享。可以说，缺了 channel，goroutine 会黯然失色。

> Don't communicate by sharing memory, share memory by communicating.
> CSP: Communicating Sequential Process.

1. 创建

同步和异步的区别，在于是否有缓冲槽。

chan.go

```
type hchan struct {
    dataqsiz uint                 // 缓冲槽大小（可存储数据项数量）。
    buf      unsafe.Pointer       // 缓冲槽指针。
    elemsize uint16               // 数据项大小。
    elemtype *_type               // 数据项类型。
}
```

chan.go

```
func makechan(t *chantype, size int64) *hchan {
    elem := t.elem

    // 数据项不能超过 64KB（这时候用指针更合适一些）。
    if elem.size >= 1<<16 {
        throw("makechan: invalid channel element type")
    }

    // 缓冲槽大小检查。
    if size < 0 || int64(uintptr(size)) != size ||
      (elem.size > 0 && uintptr(size) > (_MaxMem-hchanSize)/uintptr(elem.size)) {
        panic("makechan: size out of range")
    }

    var c *hchan

    // 受垃圾回收器限制，指针类型缓冲槽须单独分配内存。
    if elem.kind&kindNoPointers != 0 || size == 0 {
        // 因为缓冲槽大小固定，所以可一次性分配内存。
        c = (*hchan)(mallocgc(hchanSize+uintptr(size)*uintptr(elem.size), nil, flagNoScan))
        if size > 0 && elem.size != 0 {
```

```
                        // 调整缓冲槽起始指针。
                        c.buf = add(unsafe.Pointer(c), hchanSize)
                } else {
                        c.buf = unsafe.Pointer(c)
                }
        } else {
                c = new(hchan)
                c.buf = newarray(elem, uintptr(size))
        }

        // 设置属性。
        c.elemsize = uint16(elem.size)
        c.elemtype = elem
        c.dataqsiz = uint(size)

        return c
}
```

2. 收发

必须对 channel 收发双方（G）进行包装，因为要携带数据项，存储相关状态。

runtime2.go

```
type g struct {
    param unsafe.Pointer   // 传递唤醒参数。
}

type sudog struct {
    g      *g
    elem   unsafe.Pointer   // 数据存储空间指针。
}
```

另外，channel 还得维护发送和接收者等待队列，以及异步缓冲槽环状队列索引位置。

chan.go

```
type hchan struct {
    qcount   uint                 // 缓冲槽有效数据项数量。
    closed   uint32               // 是否关闭。
    sendx    uint                 // 缓冲槽发送位置索引。
    recvx    uint                 // 缓冲槽接收位置索引。
    recvq    waitq                // 接收者等待队列。
    sendq    waitq                // 发送者等待队列。
}
```

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

和以往一样，sudog 也实现了二级缓存复用体系。

runtime2.go

```
type p struct {
    sudogcache []*sudog       // 在 procresize new(p) 时指向 sudogbuf。
    sudogbuf   [128]*sudog
}

type schedt struct {
    sudogcache *sudog
}
```

proc.go

```
func acquireSudog() *sudog {
    pp := mp.p.ptr()

    // 如果本地缓存为空。
    if len(pp.sudogcache) == 0 {
            // 从全局缓存转移一批到本地。
            for len(pp.sudogcache) < cap(pp.sudogcache)/2 && sched.sudogcache != nil {
                    s := sched.sudogcache
                    sched.sudogcache = s.next
                    s.next = nil
                    pp.sudogcache = append(pp.sudogcache, s)
            }

            // 如果失败，则新建。
            if len(pp.sudogcache) == 0 {
                    pp.sudogcache = append(pp.sudogcache, new(sudog))
            }
    }

    // 从尾部提取，并调整本地缓存。
    n := len(pp.sudogcache)
    s := pp.sudogcache[n-1]
    pp.sudogcache[n-1] = nil
    pp.sudogcache = pp.sudogcache[:n-1]

    return s
}

func releaseSudog(s *sudog) {
    pp := mp.p.ptr()
```

```
    // 如果本地缓存已满。
    if len(pp.sudogcache) == cap(pp.sudogcache) {
            // 转移一半到全局。
            var first, last *sudog
            for len(pp.sudogcache) > cap(pp.sudogcache)/2 {
                    n := len(pp.sudogcache)
                    p := pp.sudogcache[n-1]
                    pp.sudogcache[n-1] = nil
                    pp.sudogcache = pp.sudogcache[:n-1]
                    if first == nil {
                            first = p
                    } else {
                            last.next = p
                    }
                    last = p
            }

            // 将提取的链表挂到全局。
            last.next = sched.sudogcache
            sched.sudogcache = first
    }

    pp.sudogcache = append(pp.sudogcache, s)
}
```

---

sched.sudogcache 缓存会在垃圾回收执行 clearpools 时被清理，但 P 本地缓存会被保留。

同步和异步收发算法有很大差异，但不知作者为什么非要将它们塞到一起。这些看起来有些巨大的函数，让人看着很不舒服。为便于分析，我们将其拆解开来。

同步

同步模式的关键是找到匹配的接收或发送方，找到则直接拷贝数据，找不到就将自身打包后放入等待队列，由另一方复制数据并唤醒。

在同步模式下，channel 的作用仅是维护发送和接收者队列，数据复制与 channel 无关。另外在唤醒后，需要验证唤醒者身份，以此决定是否有实际的数据传递。

chan.go

```
func chansend1(t *chantype, c *hchan, elem unsafe.Pointer) {
    chansend(t, c, elem, true, getcallerpc(unsafe.Pointer(&t)))
}
```

```
// 参数 eq 是数据项指针。
func chansend(t *chantype, c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
    // 同步模式
    if c.dataqsiz == 0 {
            // 从等待队列获取接收者。
            sg := c.recvq.dequeue()
            if sg != nil {
                    recvg := sg.g

                    // 直接用 memmove 将数据项复制给接收者。
                    if sg.elem != nil {
                            syncsend(c, sg, ep)
                    }

                    // 唤醒检查标志，表明是由发送者唤醒。
                    // closechan 一样会唤醒接收者，但 param = nil。
                    recvg.param = unsafe.Pointer(sg)

                    // 唤醒接收者。
                    goready(recvg, 3)
                    return true
            }

            // 如果没有接收者，则打包成 sudog。
            gp := getg()
            mysg := acquireSudog()  // 新建，或从缓存获取复用 sudog 对象。
            mysg.elem = ep
            mysg.g = gp
            gp.param = nil

            // 将发送 sudog 放入等待队列，休眠，等待被接收者唤醒。
            c.sendq.enqueue(mysg)
            goparkunlock(&c.lock, "chan send", traceEvGoBlockSend, 3)

            // 被唤醒，检查是否被 closechan 唤醒。
            // 此时数据已被接收者复制，无需再做处理。
            gp.waiting = nil
            if gp.param == nil {
                    if c.closed == 0 {
                            throw("chansend: spurious wakeup")
                    }
                    panic("send on closed channel")
            }
            gp.param = nil

            // 将 sudog 放回复用缓存。
            releaseSudog(mysg)
            return true
    }
}
```

接收代码和发送几乎一致，差别在于谁先进入等待队列，谁负责唤醒。编译器会将不同语法翻译成不同的函数调用。

chan.go

```
// <- chan
func chanrecv1(t *chantype, c *hchan, elem unsafe.Pointer) {
    chanrecv(t, c, elem, true)
}

// x, ok := <- chan
// for x := range chan
func chanrecv2(t *chantype, c *hchan, elem unsafe.Pointer) (received bool) {
    _, received = chanrecv(t, c, elem, true)
    return
}
```

chan.go

```
func chanrecv(t *chantype, c *hchan, ep unsafe.Pointer, block bool) (selected, received bool) {
    // 同步模式。
    if c.dataqsiz == 0 {
            // 从等待队列获取发送者。
            sg := c.sendq.dequeue()
            if sg != nil {
                    // 从发送者复制数据。
                    if ep != nil {
                            typedmemmove(c.elemtype, ep, sg.elem)
                    }
                    sg.elem = nil
                    gp := sg.g

                    // 设置唤醒检查标志。
                    gp.param = unsafe.Pointer(sg)

                    // 唤醒发送者，解除其阻塞。
                    goready(gp, 3)

                    selected = true
                    received = true
                    return
            }

            // 如果没有发送者，打包成 sudog。
            gp := getg()
            mysg := acquireSudog()
            mysg.elem = ep
            mysg.g = gp
            gp.param = nil

            // 放入等待队列，休眠，等待被发送者唤醒。
```

```
            c.recvq.enqueue(mysg)
            goparkunlock(&c.lock, "chan receive", traceEvGoBlockRecv, 3)

            // 被唤醒。
            // 数据已被发送者复制过来。
            gp.waiting = nil

            // 通过检查唤醒标志来决定是否有数据被复制。
            haveData := gp.param != nil
            gp.param = nil

            // 将 sudog 放回复用缓存。
            releaseSudog(mysg)

            if haveData {
                    selected = true
                    received = true
                    return
            }

            return recvclosed(c, ep)
    }
}
```

异步

异步模式围绕缓冲槽进行。当有空位时，发送者向槽中复制数据；有数据后，接收者从槽中获取数据。双方都有唤醒排队另一方继续工作的责任。

chan.go

```
func chansend(t *chantype, c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
    // 异步模式。

    // 如果缓冲槽没有空位。
    for futile := byte(0); c.qcount >= c.dataqsiz; futile = traceFutileWakeup {
            // 打包成 sudog。
            gp := getg()
            mysg := acquireSudog()
            mysg.g = gp
            mysg.elem = nil

            // 放入发送者等待队列，休眠。等待有空位时被唤醒。
            c.sendq.enqueue(mysg)
            goparkunlock(&c.lock, "chan send", traceEvGoBlockSend|futile, 3)

            // 唤醒后，如果 qcount < dataqsiz 表示有空位，跳出循环。

            // 将 sudog 放回复用缓存。
```

```
            releaseSudog(mysg)
    }

    // 将数据复制到缓冲槽。
    typedmemmove(c.elemtype, chanbuf(c, c.sendx), ep)

    // 调整缓冲槽队列索引和数据项计数。
    c.sendx++
    if c.sendx == c.dataqsiz {
            c.sendx = 0
    }
    c.qcount++

    // 现在缓冲槽不为空，唤醒某个排队的接收者从槽中获取数据。
    sg := c.recvq.dequeue()
    if sg != nil {
            recvg := sg.g
            goready(recvg, 3)
    }

    return true
}
```

发送须有缓冲槽空位，而接收则须槽中有可用数据项。

chan.go

```
func chanrecv(t *chantype, c *hchan, ep unsafe.Pointer, block bool) (selected, received bool) {
    // 异步模式。

    // 如果缓冲槽中没有数据项。
    for futile := byte(0); c.qcount <= 0; futile = traceFutileWakeup {
            // 打包成 sudog。
            gp := getg()
            mysg := acquireSudog()
            mysg.elem = nil
            mysg.g = gp

            // 放入接收等待队列，休眠。等待有数据项时被唤醒。
            c.recvq.enqueue(mysg)
            goparkunlock(&c.lock, "chan receive", traceEvGoBlockRecv|futile, 3)

            // 唤醒后，qcount > 0，跳出循环。

            // 将 sudog 返回复用缓存。
            releaseSudog(mysg)
    }

    // 从缓冲槽复制数据项。
    if ep != nil {
            typedmemmove(c.elemtype, ep, chanbuf(c, c.recvx))
    }
```

```
    // 清零。调整缓冲槽队列索引及计数。
    memclr(chanbuf(c, c.recvx), uintptr(c.elemsize))
    c.recvx++
    if c.recvx == c.dataqsiz {
        c.recvx = 0
    }
    c.qcount--

    // 现在有空位了，唤醒某个排队的发送者向槽中发送数据。
    sg := c.sendq.dequeue()
    if sg != nil {
        gp := sg.g
        goready(gp, 3)
    }

    selected = true
    received = true
    return
}
```

关闭

关闭操作将所有排队者唤醒，并通过 chan.closed、g.param 参数告知由 close 发出。

- 向 closed channel 发送数据，触发 panic。
- 从 closed channel 读取数据，返回零值。
- 无论收发，nil channel 都会阻塞。

chan.go

```
func closechan(c *hchan) {
    // 不能重复关闭。
    if c.closed != 0 {
        panic("close of closed channel")
    }

    // 设置关闭标志。
    c.closed = 1

    // 释放所有接收者。
    for {
        sg := c.recvq.dequeue()
        if sg == nil {
            break
        }
        gp := sg.g
        sg.elem = nil
```

```
            // 这个参数表明唤醒者是 closechan。
            gp.param = nil

            // 唤醒接收者。
            goready(gp, 3)
    }

    // 释放所有发送者。
    for {
            sg := c.sendq.dequeue()
            if sg == nil {
                    break
            }
            gp := sg.g
            sg.elem = nil

            // closechan 唤醒。
            gp.param = nil
            goready(gp, 3)
    }
}
```

3. 选择

选择模式（select）是从多个 channel 里随机选出可用的那个，编译器会将相关语句翻译成具体的函数调用。

test.go

```
package main

import ()

func main() {
    c1, c2 := make(chan int), make(chan int, 2)

    select {
    case c1 <- 1:
        println(0x11)
    case <-c2:
        println(0x22)
    default:
        println(0xff)
    }
}
```

反汇编。

```
$ go build -o test test.go

$ go tool objdump -s "main\.main" test

TEXT main.main(SB) test.go
    test.go:6   0x2073  CALL runtime.makechan(SB)        // make(c1)
    test.go:6   0x2096  CALL runtime.makechan(SB)        // make(c2)
    test.go:8   0x20e2  CALL runtime.newselect(SB)       // newselect
    test.go:9   0x2104  CALL runtime.selectsend(SB)      // case c1
    test.go:11  0x2153  CALL runtime.selectrecv(SB)      // case c2
    test.go:13  0x2189  CALL runtime.selectdefault(SB)   // case default
    test.go:8   0x21c2  CALL runtime.selectgo(SB)        // selectgo
```

完整的 select 对象由“header + [n]scase”组成，完全是 C 不定长结构体的风格。

select.go

```
type hselect struct {
    tcase     uint16      // ncase 总数。
    ncase     uint16      // ncase 初始化顺序。
    pollorder *uint16     // 乱序后的 scase 序号。
    lockorder **hchan     // 按 scase channel 地址排序。
    scase     [1]scase    // scase 数组。
}

type scase struct {
    elem        unsafe.Pointer    // data element
    c           *hchan            // chan
    pc          uintptr           // return pc
    kind        uint16
    so          uint16            // vararg of selected bool
    receivedp   *bool             // pointer to received bool (recv2)
    releasetime int64
}
```

初始化函数 newselect 除设置相关初始属性外，还将一次性分配的内存切分给相关字段。

select.go

```
func newselect(sel *hselect, selsize int64, size int32) {
    if selsize != int64(selectsize(uintptr(size))) {
            throw("bad select size")
    }
    sel.tcase = uint16(size)
    sel.ncase = 0
    sel.lockorder = (**hchan)(add(unsafe.Pointer(&sel.scase),
```

```
                                    uintptr(size)*unsafe.Sizeof(hselect{}.scase[0])))
    sel.pollorder = (*uint16)(add(unsafe.Pointer(sel.lockorder),
                                    uintptr(size)*unsafe.Sizeof(*hselect{}.lockorder)))
}

func selectsize(size uintptr) uintptr {
    selsize := unsafe.Sizeof(hselect{}) +
           (size-1)*unsafe.Sizeof(hselect{}.scase[0]) +
           size*unsafe.Sizeof(*hselect{}.lockorder) +
           size*unsafe.Sizeof(*hselect{}.pollorder)
    return round(selsize, _Int64Align)
}
```

在处理好 select 对象后，还需初始化 scase。过程并不复杂，依 ncase 确定位置，设置相关参数。依照 case channel 操作方式，可分为 send、recv、default 三种。

select.go

```
func selectsendImpl(sel *hselect, c *hchan, pc uintptr, elem unsafe.Pointer, so uintptr) {
    // 确定位置。
    i := sel.ncase
    sel.ncase = i + 1

    // 获取 scase，初始化。
    cas := (*scase)(add(unsafe.Pointer(&sel.scase), uintptr(i)*unsafe.Sizeof(sel.scase[0])))

    cas.pc = pc
    cas.c = c
    cas.so = uint16(so)
    cas.kind = caseSend
    cas.elem = elem
}

func selectrecvImpl(sel *hselect, c *hchan, pc uintptr, elem unsafe.Pointer, ...) {
    i := sel.ncase
    sel.ncase = i + 1
    cas := (*scase)(add(unsafe.Pointer(&sel.scase), uintptr(i)*unsafe.Sizeof(sel.scase[0])))
    cas.pc = pc
    cas.c = c
    cas.so = uint16(so)
    cas.kind = caseRecv
    cas.elem = elem
    cas.receivedp = received
}

func selectdefaultImpl(sel *hselect, callerpc uintptr, so uintptr) {
    i := sel.ncase
    sel.ncase = i + 1
    cas := (*scase)(add(unsafe.Pointer(&sel.scase), uintptr(i)*unsafe.Sizeof(sel.scase[0])))
    cas.pc = callerpc
    cas.c = nil              // 注意，这个会影响后面 lockorder 排序。
    cas.so = uint16(so)
```

```
    cas.kind = caseDefault
}
```

选择算法又是一个充斥 goto 跳转的超长大杂烩，精简掉无关代码后，耐心点慢慢看。

select.go

```
func selectgo(sel *hselect) {
    pc, offset := selectgoImpl(sel)
    *(*bool)(add(unsafe.Pointer(&sel), uintptr(offset))) = true
    setcallerpc(unsafe.Pointer(&sel), pc)
}

func selectgoImpl(sel *hselect) (uintptr, uint16) {
    // 为访问方便，将 scase 封装成 slice。
    scaseslice := slice{unsafe.Pointer(&sel.scase), int(sel.ncase), int(sel.ncase)}
    scases := *(*[]scase)(unsafe.Pointer(&scaseslice))

    // pollorder: 对 scases 序号洗牌，乱序。
    // lockorder: 按 channel 地址顺序排序。

    // 锁定全部 channel。
    sellock(sel)

loop:

    // ------------------------
    // 1: 查找已准备好的 case。
    // ------------------------

    for i := 0; i < int(sel.ncase); i++ {
            // 从乱序的 pollorder 中获取，这就是 select 随机选择的关键。
            cas = &scases[pollorder[i]]
            c = cas.c

            switch cas.kind {
            case caseRecv:
                    if c.dataqsiz > 0 {             // 异步。
                            if c.qcount > 0 {       // 缓冲槽有数据。
                                    goto asyncrecv
                            }
                    } else {                        // 同步。
                            sg = c.sendq.dequeue()
                            if sg != nil {          // 有发送者。
                                    goto syncrecv
                            }
                    }
                    if c.closed != 0 {              // 关闭。
                            goto rclose
                    }
```

```
                case caseSend:
                        if c.closed != 0 {
                                goto sclose
                        }
                        if c.dataqsiz > 0 {
                                if c.qcount < c.dataqsiz {
                                        goto asyncsend
                                }
                        } else {
                                sg = c.recvq.dequeue()
                                if sg != nil {
                                        goto syncsend
                                }
                        }

                case caseDefault:
                        dfl = cas
                }
        }

        // 如果没有准备好的 case，尝试执行 default。
        if dfl != nil {
                selunlock(sel)
                cas = dfl
                goto retc
        }

        // ------------------------------------------------------
        // 2: 如果没有任何准备好的 case，将当前 select G 打包成 sudog，
        //    放到所有 channel 排队列表，等待唤醒。
        // ------------------------------------------------------

        gp = getg()
        done = 0
        for i := 0; i < int(sel.ncase); i++ {
                cas = &scases[pollorder[i]]
                c = cas.c

                // 打包成 sudog。
                // 每个 case 的 sudog 都不同。
                sg := acquireSudog()
                sg.g = gp
                sg.selectdone = (*uint32)(noescape(unsafe.Pointer(&done)))
                sg.elem = cas.elem

                // 全部 sudog 被放入 gp.waiting 链表。
                // 此链表顺序同 pollorder，后面以此识别是哪个 case 唤醒。
                sg.waitlink = gp.waiting
                gp.waiting = sg

                // 根据 case 类型，决定放入发送或接收者排队列表。
                switch cas.kind {
                case caseRecv:
                        c.recvq.enqueue(sg)
                case caseSend:
```

```
                c.sendq.enqueue(sg)
        }
}

// 休眠 select G，直到某个 case channel 活动后，从排队列表将其提取并唤醒。
gp.param = nil
gopark(selparkcommit, unsafe.Pointer(sel), "select", traceEvGoBlockSelect|futile, 2)

// 被唤醒。
sellock(sel)
sg = (*sudog)(gp.param)  // 注意唤醒参数，就是待查 sudog。
gp.param = nil

// ---------------------------------
// 3: 找出是哪个 case 唤醒 select G。
// ---------------------------------

// 使用第二步准备好的 gp.waiting 链表。
sglist = gp.waiting
gp.waiting = nil

for i := int(sel.ncase) - 1; i >= 0; i-- {
        // 同样使用 pollorder，所以和 gp.waiting 顺序一致。
        k = &scases[pollorder[i]]

        if sg == sglist {
                // 匹配。
                cas = k
        } else {
                // 不匹配，将 sudog 从 channel 排队列表移除。
                c = k.c
                if k.kind == caseSend {
                        c.sendq.dequeueSudoG(sglist)
                } else {
                        c.recvq.dequeueSudoG(sglist)
                }
        }

        // 利用循环清理掉所有排队 sudog。
        sgnext = sglist.waitlink
        sglist.waitlink = nil
        releaseSudog(sglist)
        sglist = sgnext
}

// 没找到匹配，可能被意外唤醒，重新开始。
if cas == nil {
        goto loop
}

// 找到目标，解锁，退出。
selunlock(sel)
goto retc
```

```
// 这些前面已分析过，略过。
asyncrecv:
asyncsend:
syncrecv:
rclose:
syncsend:

retc:
    return cas.pc, cas.so

sclose:
    selunlock(sel)
    panic("send on closed channel")
}
```

对这种代码风格，真是无语了。就算原本是 C 也没必要写成这样吧，什么时候才能整理干净?

简化后的流程看上去就清爽多了。

1. 用 pollorder“随机”遍历，找出准备好的 case。
2. 如没有可用 case，则尝试 default case。
3. 如都不可用，则将 select G 打包放入所有 channel 的排队列表。
4. 直到 select G 被某个 channel 唤醒，遍历 ncase 查找目标 case。

每次操作，都需要对全部 channel 加锁，这粒度似乎太大了些。

select.go

```
func sellock(sel *hselect) {
    lockslice := slice{unsafe.Pointer(sel.lockorder), int(sel.ncase), int(sel.ncase)}
    lockorder := *(*[]*hchan)(unsafe.Pointer(&lockslice))

    var c *hchan
    for _, c0 := range lockorder {
            // 如果和前一 channel 地址不同，则加锁。
            // lockorder 的作用就是避免对同一 channel 重复加锁。
            if c0 != nil && c0 != c {
                    c = c0
                    lock(&c.lock)
            }
    }
}

func selunlock(sel *hselect) {
    n := int(sel.ncase)
    r := 0
    lockslice := slice{unsafe.Pointer(sel.lockorder), n, n}
```

```
    lockorder := *(*[]*hchan)(unsafe.Pointer(&lockslice))

    // 因为 default case 的 channel = nil，所以总是排在 lockorder[0]，跳过。
    if n > 0 && lockorder[0] == nil {
            r = 1
    }

    for i := n - 1; i >= r; i-- {
            c := lockorder[i]

            // 避免重复解锁。
            if i > 0 && c == lockorder[i-1] {
                    continue
            }
            unlock(&c.lock)
    }
}
```

官方确定要改进 select lock，只是 release 时间未定。

八. 延迟

延迟调用（defer） 最大优势是，即便函数执行出错，依然能保证回收资源等操作得以执行。但如果对性能有要求，且错误能被控制，那么还是直接执行比较好。

1. 定义

我们用一个简单的示例来揭开 defer 的秘密。

test.go

```
package main

import ()

func main() {
    defer println(0x11)
}
```

反编译：

```
$ go build -o test test.go

$ go tool objdump -s "main\.main" test

TEXT main.main(SB) test.go
    test.go:5   0x204f  SUBQ $0x18, SP
    test.go:6   0x2053  MOVQ $0x11, 0x10(SP)     // arg 0x11
    test.go:6   0x205c  MOVL $0x8, 0(SP)         // arg size
    test.go:6   0x2063  LEAQ 0x8379e(IP), AX     // 0x8379e(0x206a) = 0x85808 print function
    test.go:6   0x206a  MOVQ AX, 0x8(SP)         //           +--- IP 指向下一条指令。
    test.go:6   0x206f  CALL runtime.deferproc(SB)
    test.go:6   0x2074  CMPL $0x0, AX
    test.go:6   0x2077  JNE 0x2084
    test.go:7   0x2079  NOPL
    test.go:7   0x207a  CALL runtime.deferreturn(SB)
    test.go:7   0x207f  ADDQ $0x18, SP
    test.go:7   0x2083  RET

$ nm test | grep "85808"

0000000000085808 s main.print.1.f
```

编译器将 defer 处理成两个函数调用，deferproc 定义一个延迟调用对象，然后在函数结束前通过 deferreturn 完成最终调用。

和前面一样，对于这类参数不确定的都是用 funcval 处理，siz 是目标函数参数长度。

runtime2.go

```
type _defer struct {
    siz     int32
    started bool
    sp      uintptr   // 调用 deferproc 时的 SP。
    pc      uintptr   // 调用 deferproc 时的 IP。
    fn      *funcval
    _panic  *_panic   // panic that is running defer
    link    *_defer
}
```

panic.go

```
func deferproc(siz int32, fn *funcval) { // arguments of fn follow fn
    sp := getcallersp(unsafe.Pointer(&siz))
    argp := uintptr(unsafe.Pointer(&fn)) + unsafe.Sizeof(fn)
    callerpc := getcallerpc(unsafe.Pointer(&siz))

    systemstack(func() {
            d := newdefer(siz)
            d.fn = fn
            d.pc = callerpc
            d.sp = sp
            memmove(add(unsafe.Pointer(d), unsafe.Sizeof(*d)),
                    unsafe.Pointer(argp), uintptr(siz))
    })

    // deferproc returns 0 normally.
    // a deferred func that stops a panic makes the deferproc return 1.
    // the code the compiler generates always checks the return value and jumps to the
    // end of the function if deferproc returns != 0.
    return0()
}
```

这个函数粗看没什么复杂的地方，但有两个问题：第一，参数被复制到 defer 对象后面的内存空间；第二，匿名函数中创建的 d 保存在哪?

panic.go

```
func newdefer(siz int32) *_defer {
    var d *_defer
```

```
    // 参数长度对齐后，获取缓存等级。
    sc := deferclass(uintptr(siz))

    mp := acquirem()

    // 未超出缓存大小。
    if sc < uintptr(len(p{}.deferpool)) {
            pp := mp.p.ptr()

            // 如果 P 本地缓存已空，从全局提取一批到本地。
            if len(pp.deferpool[sc]) == 0 && sched.deferpool[sc] != nil {
                    for len(pp.deferpool[sc]) < cap(pp.deferpool[sc])/2 &&
                        sched.deferpool[sc] != nil {
                            d := sched.deferpool[sc]
                            sched.deferpool[sc] = d.link
                            d.link = nil
                            pp.deferpool[sc] = append(pp.deferpool[sc], d)
                    }
            }

            // 从本地缓存尾部提取。
            if n := len(pp.deferpool[sc]); n > 0 {
                    d = pp.deferpool[sc][n-1]
                    pp.deferpool[sc][n-1] = nil
                    pp.deferpool[sc] = pp.deferpool[sc][:n-1]
            }
    }

    // 新建。很显然分配的空间大小除 _defer 外，还有参数。
    if d == nil {
            // Allocate new defer+args.
            total := roundupsize(totaldefersize(uintptr(siz)))
            d = (*_defer)(mallocgc(total, deferType, 0))
    }

    d.siz = siz

    // 将 d 保存到 G._defer 链表。
    gp := mp.curg
    d.link = gp._defer
    gp._defer = d

    releasem(mp)
    return d
}
```

runtime2.go

```
type p struct {
    deferpool [5][]*_defer
}

type g struct {
```

```
    _defer *_defer
}
```

defer 同样使用了二级缓存，这个没兴趣深究。newdefer 函数解释了前面的两个问题：一次性为 defer 和参数分配空间，其次 d 被挂到 G._defer 链表。

那么，退出前 deferreturn 自然是从 G._defer 获取并执行延迟函数了。

panic.go

```
func deferreturn(arg0 uintptr) {
    gp := getg()

    // 提取 defer 延迟对象。
    d := gp._defer
    if d == nil {
        return
    }

    // 对比 SP，避免调用其他栈帧的延迟函数。（arg0 也就是 deferproc siz 参数）
    sp := getcallersp(unsafe.Pointer(&arg0))
    if d.sp != sp {
        return
    }

    mp := acquirem()

    // 将延迟函数的参数复制到堆栈（这会覆盖掉 siz、fn，不过没有影响）。
    memmove(unsafe.Pointer(&arg0), deferArgs(d), uintptr(d.siz))
    fn := d.fn
    d.fn = nil

    // 调整 G._defer 链表。
    gp._defer = d.link

    // 释放 _defer 对象，放回缓存。
    systemstack(func() {
        freedefer(d)
    })

    releasem(mp)

    // 执行延迟函数。
    jmpdefer(fn, uintptr(unsafe.Pointer(&arg0)))
}
```

---

freedefer 将 _defer 放回 P.deferpool 缓存，当数量超出时会转移部分到 sched.deferpool。垃圾回收时，clearpools 会清理掉 sched.deferpool 缓存。

汇编实现的 jmpdefer 函数很有意思。

首先通过 arg0 参数，也就是调用 deferproc 时压入的第一参数 siz 获取 main.main SP。当 main 调用 deferreturn 时，用 SP-8 就可以获取当时保存的 main IP 值。因为 IP 保存了下一条指令地址，那么用该地址减去 CALL 指令长度，自然又回到了 main 调用 deferreturn 函数的位置。将这个计算得来的地址入栈，加上 jmpdefer 没有保存现场，那么延迟函数 fn RET 自然回到 CALL deferreturn，如此就实现了多个 defer 延迟调用循环。

asm_amd64.s

```
TEXT runtime·jmpdefer(SB), NOSPLIT, $0-16
    MOVQ    fv+0(FP), DX           // 延迟函数 fn 地址。
    MOVQ    argp+8(FP), BX         // argp+8 是 arg0 地址，也就是 main 的 SP。
    LEAQ    -8(BX), SP             // 将 SP-8 获取的其实是 call deferreturn 是压入的 main IP。
    SUBQ    $5, (SP)               // CALL 指令长度 5，-5 返回的就是 call deferreturn 指令地址。
    MOVQ    0(DX), BX              // 执行 fn 函数。
    JMP     BX
```

费得好大力气，真有必要这么做么?

虽然整个调用堆栈的 defer 都挂在 G._defer 链表，但在 deferreturn 里面通过 sp 值的比对，可避免调用其他栈帧的延迟函数。

如中途用 Goexit 终止，它会负责处理整个调用堆栈的延迟函数。

panic.go

```
func Goexit() {
    gp := getg()
    for {
            d := gp._defer
            if d == nil {
                    break
            }
            if d.started {
                    if d._panic != nil {
                            d._panic.aborted = true
                            d._panic = nil
                    }
                    d.fn = nil
                    gp._defer = d.link
                    freedefer(d)
                    continue
```

```
            }
            d.started = true
            reflectcall(nil, unsafe.Pointer(d.fn), deferArgs(d), uint32(d.siz), uint32(d.siz))
            if gp._defer != d {
                    throw("bad defer entry in Goexit")
            }
            d._panic = nil
            d.fn = nil
            gp._defer = d.link
            freedefer(d)
        }
        goexit1()
}
```

2. 性能

正如你所见，延迟调用远不是一个 CALL 指令那么简单，会涉及很多内容。诸如对象分配、缓存，以及多次函数调用。在某些性能要求比较高的场合，应该避免使用 defer。

test_test.go

```
package main

import (
    "sync"
    "testing"
)

var lock sync.Mutex

func test() {
    lock.Lock()
    lock.Unlock()
}

func testdefer() {
    lock.Lock()
    defer lock.Unlock()
}

func BenchmarkTest(b *testing.B) {
    for i := 0; i < b.N; i++ {
        test()
    }
}

func BenchmarkTest2(b *testing.B) {
    for i := 0; i < b.N; i++ {
        testdefer()
    }
```

```
}
```

性能测试：

```
$ go test -v -test.bench .

BenchmarkTest-4     100000000           22.0 ns/op
BenchmarkTest2-4    20000000            93.4 ns/op
```

> 相较以前版本，defer 性能有所改进，但还是有 4x 以上的差异。该结果仅供参考！

3. 错误

不知从何时起，panic 就成了一个禁忌话题，诸多教程里都有“Don't Panic!”这样的条例。这让我想起 Python __del__ 的话题，颇为类似。其实，对于不可恢复性的错误用 panic 并无不妥，见仁见智吧。

从源码看，panic/recover 的实现和 defer 息息相关，且过程算不上复杂。

runtime2.go

```
type _panic struct {
    argp      unsafe.Pointer   // pointer to arguments of deferred call run during panic
    arg       interface{}      // argument to panic
    link      *_panic          // link to earlier panic
    recovered bool             // whether this panic is over
    aborted   bool             // the panic was aborted
}

type _defer struct {
    _panic  *_panic
}

type g struct {
    _panic *_panic
}
```

编译器将 panic 翻译成 gopainc 函数调用。它会将错误信息打包成 _panic 对象，并挂到 G._panic 链表的头部。然后遍历执行 G._defer 链表，检查是否 recover。如被 recovered，

则终止遍历执行，跳转到正常的 deferreturn 环节。否则执行整个调用堆栈的延迟函数后，显示异常信息，终止进程。

panic.go

```
func gopanic(e interface{}) {
    gp := getg()

    // 新建 _painc，挂到 G._panic 链表头部。
    var p _panic
    p.arg = e
    p.link = gp._panic
    gp._panic = (*_panic)(noescape(unsafe.Pointer(&p)))

    // 遍历执行 G._defer（整个调用堆栈），直到某个 recover。
    for {
        d := gp._defer
        if d == nil {
            break
        }

        // 如果 defer 已经执行，继续下一个。
        if d.started {
            if d._panic != nil {
                d._panic.aborted = true
            }
            d._panic = nil
            d.fn = nil
            gp._defer = d.link
            freedefer(d)
            continue
        }

        // 不移除 defer，便于 traceback 输出所有调用堆栈信息。
        d.started = true

        // 将 _panic 保存到 defer._panic。
        d._panic = (*_panic)(noescape((unsafe.Pointer)(&p)))

        // 执行 defer 函数。
        // p.argp 地址很重要，defer 里的 recover 以此来判断是否直接在 defer 内执行。
        // reflectcall 会修改 p.argp。
        p.argp = unsafe.Pointer(getargp(0))
        reflectcall(nil, unsafe.Pointer(d.fn), deferArgs(d), uint32(d.siz), uint32(d.siz))
        p.argp = nil

        // 将已经执行的 defer 从 G._defer 链表移除。
        d._panic = nil
        d.fn = nil
        gp._defer = d.link

        pc := d.pc
        sp := unsafe.Pointer(d.sp)
```

```
            freedefer(d)

            // 如果该 defer 内执行了 recover，那么 recovered = true。
            if p.recovered {
                    // 移除当前 recovered panic。
                    gp._panic = p.link

                    // 移除 aborted panic。
                    for gp._panic != nil && gp._panic.aborted {
                            gp._panic = gp._panic.link
                    }

                    // recovery 会跳转会 defer.pc，也就是调用 deferproc 后。
                    // 编译器会调用 deferproc 后插入比较指令，通过标志判断，跳转
                    //   到 deferreturn 执行剩余 defer 函数。
                    gp.sigcode0 = uintptr(sp)
                    gp.sigcode1 = pc
                    mcall(recovery)
                    throw("recovery failed") // mcall should not return
            }

            // 如果没有 recovered，那么循环执行整个调用堆栈的延迟函数，
            // 要么被后续 recover，要么崩溃。
    }

    // 如果没有捕获，显示错误信息后终止（exit）进程。
    startpanic()
    printpanics(gp._panic)
    dopanic(0)              // should not return
    *(*int)(nil) = 0        // not reached
}
```

和 panic 相比，recover 函数除返回最后一个错误信息外，主要是设置 recovered 标志。注意，它会通过参数堆栈地址确认是否在延迟函数内被直接调用。

panic.go

```
func gorecover(argp uintptr) interface{} {
    gp := getg()
    p := gp._panic
    if p != nil && !p.recovered && argp == uintptr(p.argp) {
            p.recovered = true
            return p.arg
    }
    return nil
}
```

九. 析构

我也不确定怎样用中文表达 Finalizer 最合适。其主要用途是在对象被垃圾回收时执行一个关联函数，效果如同 OOP 里的析构方法（Destructor Method）。

使用示例：

test.go

```
package main

import (
    "runtime"
    "time"
)

func main() {
    x := 123
    runtime.SetFinalizer(&x, func(x *int) {
        println(x, *x, "finalizer.")
    })

    runtime.GC()
    time.Sleep(time.Minute)
}
```

1. 设置

首先得为目标对象关联一个析构函数。SetFinalizer 会通过接口内部的类型信息对目标对象和 finalizer 函数（参数数量、类型等）做出检查，确保符合要求。

mfinal.go

```
func SetFinalizer(obj interface{}, finalizer interface{}) {
    // 从接口获取类型和对象指针。
    e := (*eface)(unsafe.Pointer(&obj))
    etyp := e._type
    ot := (*ptrtype)(unsafe.Pointer(etyp))

    // 忽略 nil 对象。
    _, base, _ := findObject(e.data)
    if base == nil {
            // 0-length objects are okay.
            if e.data == unsafe.Pointer(&zerobase) {
                    return
```

```
                }
        }

        // 获取 finalizer 函数信息。
        f := (*eface)(unsafe.Pointer(&finalizer))
        ftyp := f._type

        // 如果 finalizer = nil，则移除析构函数。
        if ftyp == nil {
                systemstack(func() {
                        removefinalizer(e.data)
                })
                return
        }

        // 确保 finalizer 是函数。
        if ftyp.kind&kindMask != kindFunc {
                throw("runtime.SetFinalizer: second argument is " + *ftyp._string +
                      ", not a function")
        }

        // 检查 finalizer 参数数量及其类型。
        ft := (*functype)(unsafe.Pointer(ftyp))
        ins := *(*[]*_type)(unsafe.Pointer(&ft.in))
        if ft.dotdotdot || len(ins) != 1 {
                throw("runtime.SetFinalizer: cannot pass " + *etyp._string +
                      " to finalizer " + *ftyp._string)
        }
        fint := ins[0]
        switch {
        case fint == etyp:
                // ok - 相同类型。
                goto okarg
        case fint.kind&kindMask == kindPtr:
                goto okarg
        case fint.kind&kindMask == kindInterface:
                goto okarg
        }

        // 检查结果错误，抛出异常。
        throw("runtime.SetFinalizer: cannot pass " + *etyp._string +
              " to finalizer " + *ftyp._string)

okarg:
        // 计算返回参数大小。
        nret := uintptr(0)
        for _, t := range *(*[]*_type)(unsafe.Pointer(&ft.out)) {
                nret = round(nret, uintptr(t.align)) + uintptr(t.size)
        }
        nret = round(nret, ptrSize)

        // 确保 finalizer goroutine 运行。
        createfing()

        // 不能重复设置 finalizer 函数。
```

```
    systemstack(func() {
            if !addfinalizer(e.data, (*funcval)(f.data), nret, fint, ot) {
                    throw("runtime.SetFinalizer: finalizer already set")
            }
    })
}
```

析构函数会被打包成 specialfinalizer 对象。

mheap.go

```
type special struct {
    next   *special   // 链表。
    offset uint16     // 目标对象地址偏移量。
    kind   byte       // 类型。
}

type specialfinalizer struct {
    special special   // 匿名嵌入。
    fn      *funcval
    nret    uintptr
    fint    *_type
    ot      *ptrtype
}

func addfinalizer(p unsafe.Pointer, f *funcval, nret uintptr, fint *_type, ot *ptrtype) bool {
    // 从固定分配器创建 specialfinalizer。
    s := (*specialfinalizer)(fixAlloc_Alloc(&mheap_.specialfinalizeralloc))
    s.special.kind = _KindSpecialFinalizer
    s.fn = f
    s.nret = nret
    s.fint = fint
    s.ot = ot

    // 添加（注意，使用了匿名嵌入字段）。
    if addspecial(p, &s.special) {
            return true
    }

    // 已经有 finalizer，释放当前 specialfinalizer。
    fixAlloc_Free(&mheap_.specialfinalizeralloc, (unsafe.Pointer)(s))
    return false
}
```

最终 specialfinalizer 被保存到 span.specials 链表。这里的算法很有意思，利用目标对象在 span 的地址偏移量作为去重和排序条件。如此，单个循环就可以完成去重判断和有序添加操作。

mheap.go

```
type mspan struct {
    specials *special  // 按偏移量排序链表。
}

func addspecial(p unsafe.Pointer, s *special) bool {
    // 找到目标对象所属 span，计算地址偏移量。
    span := mHeap_LookupMaybe(&mheap_, p)
    offset := uintptr(p) - uintptr(span.start<<_PageShift)
    kind := s.kind

    // 遍历 span.specials 链表，
    // 通过偏移量和 _KindSpecialFinalizer 检查是否已设置 finalizer。
    t := &span.specials
    for {
        x := *t
        if x == nil {
            break
        }

        // 已设置。
        if offset == uintptr(x.offset) && kind == x.kind {
            return false // already exists
        }

        // 因为 span.specials 按 offset 排序，所以没必要超出范围检查。
        if offset < uintptr(x.offset) || (offset == uintptr(x.offset) && kind < x.kind) {
            break
        }

        t = &x.next
    }

    // 利用上面循环中断，将 special 插入链表合适地方，保持有序。
    s.offset = uint16(offset)
    s.next = *t
    *t = s

    return true
}
```

移除操作也只需用偏移量遍历 span.specials 链表即可完成。

mheap.go

```
func removespecial(p unsafe.Pointer, kind uint8) *special {
    // 查找所属 span，计算地址偏移量。
    span := mHeap_LookupMaybe(&mheap_, p)
    offset := uintptr(p) - uintptr(span.start<<_PageShift)
```

```
    // 遍历链表，移除 special。
    t := &span.specials
    for {
            s := *t
            if s == nil {
                    break
            }
            if offset == uintptr(s.offset) && kind == s.kind {
                    *t = s.next
                    return s
            }
            t = &s.next
    }
    return nil
}
```

2. 清理

垃圾清理操作在处理 span 时会检查 specials 链表，将不可达对象的 finalizer 函数添加到一个特定待执行队列。

mgcsweep.go

```
func mSpan_Sweep(s *mspan, preserve bool) bool {
    specialp := &s.specials
    special := *specialp
    for special != nil {
            // 利用偏移量计算出目标对象地址。
            p := uintptr(s.start<<_PageShift) + uintptr(special.offset)/size*size

            // 检查回收标记。
            // 如果没有标记，那么属于可回收对象，准备执行 finalizer。
            hbits := heapBitsForAddr(p)
            if !hbits.isMarked() {
                    p := uintptr(s.start<<_PageShift) + uintptr(special.offset)
                    y := special

                    // 调整 span.specials 链表。
                    special = special.next
                    *specialp = special

                    // 释放 special，将 finalizer 放入待执行队列。
                    // 下次回收该对象时，已经没有 finalizer 需要处理了。
                    if !freespecial(y, unsafe.Pointer(p), size, false) {
                            // 重新将目标对象标记，避免被清理。
                            // 为了让 finalizer 正确执行，必须延长目标对象生命周期。
                            hbits.setMarkedNonAtomic()
                    }
            } else {
```

```
                    // object is still live: keep special record
                    specialp = &special.next
                    special = *specialp
            }
    }
}
```

mheap.go

```
func freespecial(s *special, p unsafe.Pointer, size uintptr, freed bool) bool {
    switch s.kind {
    case _KindSpecialFinalizer:
            // addfinalizer 创建的原本就是 specialfinalizer，它匿名嵌入 special，
            // 此处不过是转换回原本样子而已。
            sf := (*specialfinalizer)(unsafe.Pointer(s))

            // 将函数放入待执行队列。
            queuefinalizer(p, sf.fn, sf.nret, sf.fint, sf.ot)

            // 释放 specialfinalizer 对象。
            fixAlloc_Free(&mheap_.specialfinalizeralloc, (unsafe.Pointer)(sf))
            return false // don't free p until finalizer is done
    case _KindSpecialProfile:
            ...
    default:
            throw("bad special kind")
            panic("not reached")
    }
}
```

从清理操作对持有 finalizer 不可达对象的态度可以看出，析构函数会延长对象的生命周期，直到下一次垃圾回收才会真正被清理。其根本理由就是，finalizer 函数执行时可能会访问目标对象，比如释放目标对象持有的相关资源等等。

另外，finalizer 的执行依赖于垃圾清理操作，我们无法确定其准确执行时间。且不保证在进程退出前，会一定得到执行。因此，不能用 finalizer 去执行类似 flush cache 操作。

3. 执行

在探究执行方式之前，先得搞清楚这个执行队列是怎么回事。

析构函数相关信息从 special 解包后，被重新打包成 finalizer，然后被存储到一个由数组封装而成的 finblock 容器（块）里。多个 finblock 串成链表，形成队列。看上去很像前文垃圾回收器里的 gcWork 高性能缓存队列的做法。

mfinal.go

```
type finalizer struct {
    fn   *funcval        // function to call
    arg  unsafe.Pointer  // ptr to object
    nret uintptr         // bytes of return values from fn
    fint *_type          // type of first argument of fn
    ot   *ptrtype        // type of ptr to object
}

type finblock struct {
    alllink *finblock
    next    *finblock
    cnt     int32
    _       int32
    fin     [(_FinBlockSize - 2*ptrSize - 2*4) / unsafe.Sizeof(finalizer{})]finalizer
}
```

另有几个全局变量用来管理 finblock。

mfinal.go

```
var finq *finblock     // 待执行 finalizer 队列（链表）。
var finc *finblock     // 提供 finblock 缓存复用对象。
var allfin *finblock   // 所有 finblock 列表。
```

向队列添加析构函数的过程，基本上就是对 finblock 的操作。每次都向待执行队列的第一个容器 finq 添加，直到装满后将 finq 换成新的 finblock 块。

mfinal.go

```
func queuefinalizer(p unsafe.Pointer, fn *funcval, nret uintptr, fint *_type, ot *ptrtype) {
    // finq 是链表的第一个 finblock，也是当前操作的目标。
    // 如果为空，或者内部数组已满，则重新获取 finblock 替换 finq。
    if finq == nil || finq.cnt == int32(len(finq.fin)) {
            // 如果复用缓存已空，申请新内存。
            if finc == nil {
                    // 有关 persistent，请参考内存分配相关章节。
                    finc = (*finblock)(persistentalloc(_FinBlockSize, 0, &memstats.gc_sys))

                    // 添加到 allfin 链表。
                    finc.alllink = allfin
                    allfin = finc
            }

            // 从复用缓存头部提取 finblock，并调整 finc 链表。
```

```
                block := finc
                finc = block.next

                // 将新 finblock 挂到 finq 待执行队列头部。
                block.next = finq
                finq = block
        }

        // finq.cnt 记录了 finblock 内部数组使用位置索引。
        f := &finq.fin[finq.cnt]
        finq.cnt++

        // 设置相关属性。
        f.fn = fn
        f.nret = nret
        f.fint = fint
        f.ot = ot
        f.arg = p

        // 设置 fing 唤醒标志。
        fingwake = true
}
```

准备好执行队列后，须由专门的 fing goroutine 负责执行。在 SetFinalizer 里我们就看到过 createfing 函数调用。

mfinal.go

```
func createfing() {
        // 确保仅执行一次。
        if fingCreate == 0 && cas(&fingCreate, 0, 1) {
                go runfinq()
        }
}
```

mfinal.go

```
var fing *g         // goroutine that runs finalizers
var fingwait bool   // 休眠标记。
var fingwake bool   // 唤醒标记。

func runfinq() {
        for {
                // 置换运行队列。
                // 因为是并发，所以在执行 runfinq 时不能影响新的添加操作。
                fb := finq
                finq = nil

                // 如果队列为空，则进入休眠。
```

```
            if fb == nil {
                    // 设置全局变量 fing。
                    gp := getg()
                    fing = gp

                    // 设置休眠标志，休眠。
                    fingwait = true
                    goparkunlock(&finlock, "finalizer wait", traceEvGoBlock, 1)

                    // 唤醒后重新检查队列。
                    continue
            }

            // 遍历 finq 链表。
            for fb != nil {
                    // 遍历 finblock 内部数组。
                    for i := fb.cnt; i > 0; i-- {
                            // 获取并执行 finalizer。
                            f := (*finalizer)(add(unsafe.Pointer(&fb.fin), ...))
                            reflectcall(nil, unsafe.Pointer(f.fn), frame, ...)

                            f.fn = nil
                            f.arg = nil
                            f.ot = nil
                            fb.cnt = i - 1
                    }

                    next := fb.next

                    // 将当前已完成任务的 finalizer 对象放回 finc 复用缓存。
                    fb.next = finc
                    finc = fb

                    fb = next
            }
    }
}
```

> 一路走来，已记不清 runtime 创建了多少类似 fing 这样在以死循环方式工作的 goroutine 了。好在像 fing 这样都是按需创建的。

循环遍历所有 finblock，执行其中的析构函数。要说有所不同，就是它们会在同一个 G 栈串行执行。剩余问题是，当 fing 执行完毕进入休眠后，谁来唤醒？要知道 queuefinalizer 仅仅设置了 fingwake 标志。

还记得调度循环里四处查找可用任务的 findrunnable 函数吗？没错，fing 也是它要寻找的目标之一。

proc1.go

```
func findrunnable() (gp *g, inheritTime bool) {
    // 如果 fing 正在休眠，且被设置了唤醒标志。
    if fingwait && fingwake {
            // 唤醒。
            if gp := wakefing(); gp != nil {
                    ready(gp, 0)
            }
    }
}
```

mfinal.go

```
func wakefing() *g {
    var res *g

    // 再次检查唤醒条件。
    if fingwait && fingwake {
            fingwait = false
            fingwake = false
            res = fing
    }

    return res
}
```

不管是 panic，还是 finalizer，都有特定的使用场景，因为它们有相应的设计制约。这种制约不应被看做缺陷，毕竟我们本就不该让它们去做无法保证的事情。保持有限度的谨慎和悲观不是坏事，但不能因此就无理由地去抵制和忽视。了解其原理，永远不要停留在文档的字里行间里。

十. 缓存池

设计对象缓存池，除避免内存分配操作开销外，更多的是为了避免分配大量临时对象对垃圾回收器造成负面影响。只是有一个问题需要解决，就是如何在多线程共享的情况下，解决同步锁带来的性能弊端，尤其是高并发情形下。

因 Golang goroutine 机制对线程的抽象，造成我们以往基于 LTS 的方案统统无法实施。就算 runtime 对我们开放线程访问接口也未必有用。因为 G 可能在中途被调度给其他线程，甚至你设置了 LTS 的线程会滚回闲置队列休眠。

为此，官方提供了一个深入 runtime 内核运作机制的 sync.Pool。其算法已被内存分配、垃圾回收和调度器所使用，算是得到验证的成熟高效体系。

1. 初始化

用于提供本地缓存对象分配的 poolLocal 类似内存分配器里的 cache，总是和 P 绑定，为当前工作线程提供快速无锁分配。而 Pool 则管理多个 P/poolLocal。

pool.go

```
type Pool struct {
    local      unsafe.Pointer  // [P]poolLocal 数组指针。
    localSize  uintptr         // 数组内 poolLocal 数量。
    New func() interface{}     // 新建对象函数。
}

type poolLocal struct {
    private interface{}    // 私有缓存区。
    shared  []interface{}  // 可共享缓存区。
    Mutex
    pad     [128]byte
}
```

Pool 用 local 和 localSize 维护一个动态 poolLocal 数组。无论是 Get，还是 Put 操作都会通过 pin 来返回与当前 P 绑定的 poolLocal 对象，这里面就有初始化的关键。

pool.go

```
func (p *Pool) pin() *poolLocal {
    // 返回当前 P.id。
    pid := runtime_procPin()
```

```
    s := atomic.LoadUintptr(&p.localSize)
    l := p.local

    // 如果 P.id 没有超出数组索引限制，则直接返回。
    // 这是考虑到 procresize/GOMAXPROCS 的影响。
    if uintptr(pid) < s {
            return indexLocal(l, pid)
    }

    // 没有结果时，会涉及到全局加锁操作。
    // 比如重新分配数组内存，添加到全局列表。
    return p.pinSlow()
}
```

pool.go

```
var (
    allPoolsMu Mutex
    allPools   []*Pool
)

func (p *Pool) pinSlow() *poolLocal {
    // M.lock--
    runtime_procUnpin()

    // 加锁。
    allPoolsMu.Lock()
    defer allPoolsMu.Unlock()

    pid := runtime_procPin()

    // 再次检查是否符合条件，可能中途已被其他线程调用。
    s := p.localSize
    l := p.local
    if uintptr(pid) < s {
            return indexLocal(l, pid)
    }

    // 如果数组为空，新建。
    // 将其添加到 allPools，垃圾回收器以此获取所有 Pool 实例。
    if p.local == nil {
            allPools = append(allPools, p)
    }

    // 根据 P 数量创建 slice。
    size := runtime.GOMAXPROCS(0)
    local := make([]poolLocal, size)

    // 将底层数组起始指针保存到 Pool.local，并设置 P.localSize。
    atomic.StorePointer((*unsafe.Pointer)(&p.local), unsafe.Pointer(&local[0]))
```

```
    atomic.StoreUintptr(&p.localSize, uintptr(size))

    // 返回本次所需的 poolLocal。
    return &local[pid]
}
```

至于 indexLocal 操作完全是“聪明且偷懒”的做法。

pool.go

```
func indexLocal(l unsafe.Pointer, i int) *poolLocal {
    // 不去考虑 Pool.local，也就是 l 参数实际数组长度，反正也不会超过 1000000。
    // 直接将其转换成大数组，然后按索引号返回 poolLocal 即可。
    return &(*[1000000]poolLocal)(l)[i]
}
```

不要觉得无厘头，这种做法在 C 里很常见，甚至你在某些操作系统的源码里也会看到类似的东西。这么做不用去考虑 P 数量变化，或者对 _MaxGomaxprocs 的修改，直接以性能优先。

---

2. 操作

和调度器对 P.runq 队列处理方式类似。每个 poolLocal 有两个缓存区域，其中 private 完全私有，无需任何锁操作，优先级最高。另一区域 share，允许被其他 poolLocal 访问，用来平衡调度缓存对象，需要加锁处理。不过调度并非时刻发生，这个锁多数时候仅面对当前线程，所以对性能影响并不大。

pool.go

```
func (p *Pool) Get() interface{} {
    // 返回 poolLocal。
    l := p.pin()

    // 优先从 private 选择。
    x := l.private
    l.private = nil
    if x != nil {
            return x
    }

    // 加锁，从 share 区域获取。
    l.Lock()

    // 从 shared 尾部提取缓存对象。
```

```
	last := len(l.shared) - 1
	if last >= 0 {
		x = l.shared[last]
		l.shared = l.shared[:last]
	}

	l.Unlock()
	if x != nil {
		return x
	}

	// 如果提取失败，则需要获取新的缓存对象。
	return p.getSlow()
}
```

如果从本地获取缓存失败，则考虑从其他 poolLocal 借调（就是惯偷）一个过来。实在不行，调用 New 函数新建（最终手段）。

pool.go

```
func (p *Pool) getSlow() (x interface{}) {
	size := atomic.LoadUintptr(&p.localSize)
	local := p.local

	// 当前 P.id。
	pid := runtime_procPin()

	// 从其他 poolLocal 偷取一个缓存对象。
	for i := 0; i < int(size); i++ {
		// 获取目标 poolLocal，且保证不是自身。
		l := indexLocal(local, (pid+i+1)%int(size))

		// 对目标 poolLocal 加锁，以便访问其 share 区域。
		l.Lock()

		// 偷取一个换换对象。
		last := len(l.shared) - 1
		if last >= 0 {
			x = l.shared[last]
			l.shared = l.shared[:last]

			l.Unlock()
			break
		}
		l.Unlock()
	}

	// 偷取失败，使用 New 函数新建。
	if x == nil && p.New != nil {
		x = p.New()
	}
```

```
    return x
}
```

> 注意：Get 操作后，缓存对象彻底与 Pool 失去引用关联，需要自行 Put 放回。

至于 Put 操作，就更简单了，不需考虑不同 poolLocal 之间的平衡调度。

pool.go

```
func (p *Pool) Put(x interface{}) {
    if x == nil {
        return
    }

    // 获取 poolLocal。
    l := p.pin()

    // 优先放入 private。
    if l.private == nil {
        l.private = x
        x = nil
    }
    if x == nil {
        return
    }

    // 放入 share。
    l.Lock()
    l.shared = append(l.shared, x)
    l.Unlock()
}
```

3. 清理

借助垃圾回收机制，我们无需考虑 Pool 收缩问题。只是官方的设计似乎有些粗暴。

mgc.go

```
func gc(mode int) {
    systemstack(stopTheWorldWithSema)
    clearpools()
}

var poolcleanup func()
```

```
func clearpools() {
    // clear sync.Pools
    if poolcleanup != nil {
        poolcleanup()
    }
}
```

这个 poolcleanup 函数需要额外注册。

mgc.go

```
//go:linkname sync_runtime_registerPoolCleanup sync.runtime_registerPoolCleanup
func sync_runtime_registerPoolCleanup(f func()) {
    poolcleanup = f
}
```

pool.go

```
func init() {
    runtime_registerPoolCleanup(poolCleanup)
}
```

真正的目标是 poolCleanup。此时正处于 STW 状态，所以无需加锁操作。

pool.go

```
func poolCleanup() {
    // 遍历所有 Pool 实例。
    for i, p := range allPools {
        // 解除引用。
        allPools[i] = nil

        // 遍历 Pool.poolLocal 数组。
        for i := 0; i < int(p.localSize); i++ {
            // 获取 poolLocal。
            l := indexLocal(p.local, i)

            // 清理 private 和 share 区域。
            l.private = nil
            for j := range l.shared {
                l.shared[j] = nil
            }
            l.shared = nil
        }

        // 设置 Pool.local  = nil，除解除所引用的数组空间外，
        // 还让 Pool.pinSlow 方法会将其重新添加到 allPools。
```

```
            p.local = nil
            p.localSize = 0
        }

        // 重置 allPools，需要所有 Pool.pinSlow 重新添加。
        allPools = []*Pool{}
    }
```

清理操作对已被 Get 的可达对象没有任何影响，因为两者之间并没有引用关联，留下的缓存对象都属于仅被 Pool 引用的可移除“白色对象”。

或许我们希望设置一个阈值，仅清理超出数量限制的缓存对象。如此，可避免在垃圾回收后频繁执行 New 操作。但考虑到此时还有一批可能的黑色缓存对象存在，所以需求也不是那么急切。只是，这个 Pool 的设计显然有进一步改进的余地。

（全书结束）