深入了解 Go 语言与并发编程

郭剑池 腾讯技术工程 2022-04-11 03:00

作者：fitchguo，腾讯 IEG 后台开发工程师

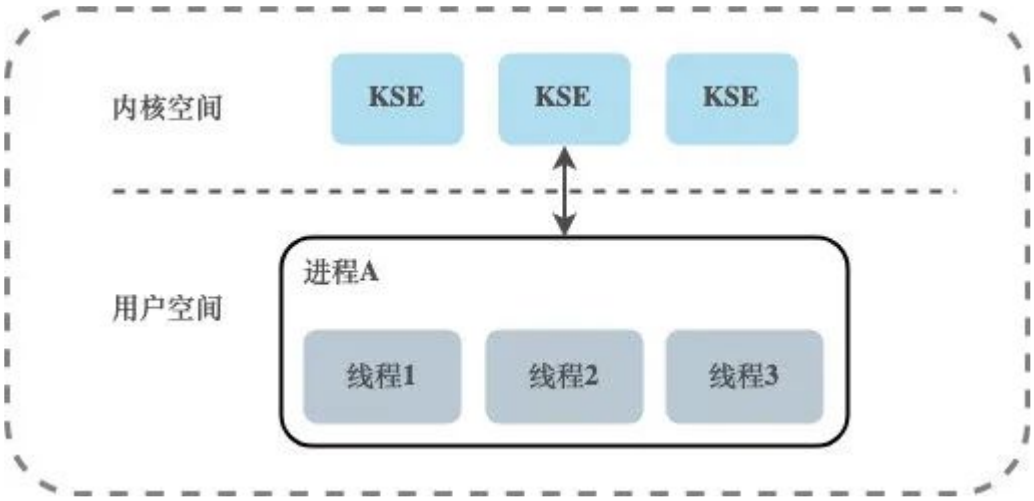
并发编程，可以说一直都是开发者们关注最多的主题之一。而 Golang 作为一个出道就自带“高并发”光环的编程语言，其并发编程的实现原理肯定是值得我们深入探究的。

Go 并发编程模型在底层是由操作系统所提供的线程库支撑的，这里先简要介绍一下线程实现模型的相关概念。

线程的实现模型

线程的实现模型主要有 3 个，分别是：用户级线程模型、内核级线程模型和两级线程模型。它们之间最大的差异在于用户线程与内核调度实体（KSE）之间的对应关系上。内核调度实体就是可以被操作系统内核调度器调度的对象，也称为内核级线程，是操作系统内核的最小调度单元。

用户级线程模型

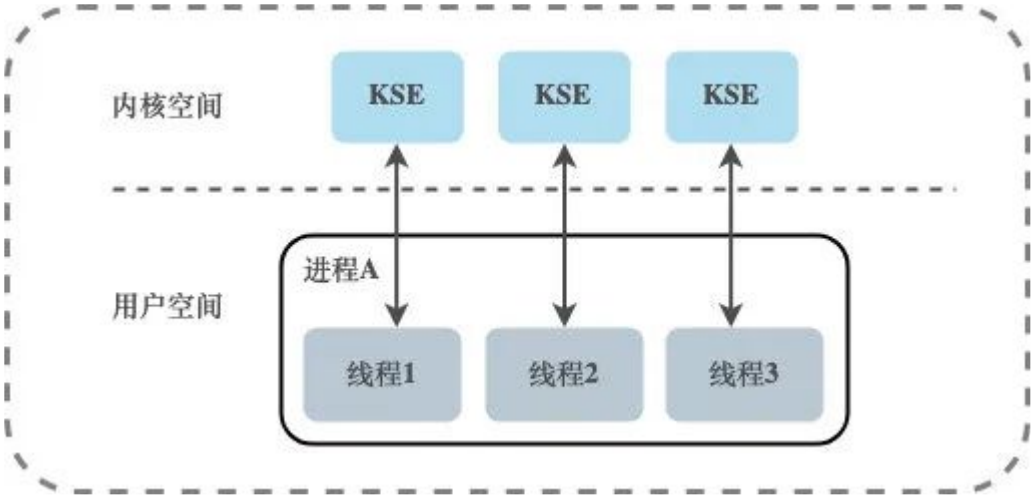


用户线程与 KSE 为多对一（N:1）的映射关系。此模型下的线程由用户级别的线程库全权管理，线程库存储在进程的用户空间之中，这些线程的存在对于内核来说是无法感知的，所以

这些线程也不是内核调度器调度的对象。一个进程中所有创建的线程都只和同一个 KSE 在运行时动态绑定，内核的所有调度都是基于用户进程的。

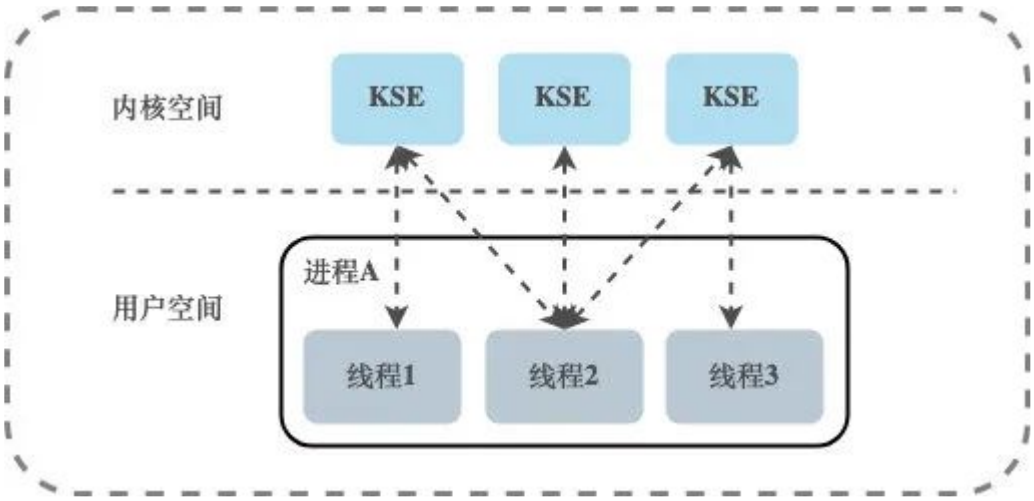
对于线程的调度则是在用户层面完成的，相较于内核调度不需要让 CPU 在用户态和内核态之间切换，这种实现方式相比内核级线程模型可以做的很轻量级，对系统资源的消耗会小很多，上下文切换所花费的代价也会小得多。许多语言实现的协程库基本上都属于这种方式。但是，此模型下的多线程并不能真正的并发运行。例如，如果某个线程在 I/O 操作过程中被阻塞，那么其所属进程内的所有线程都被阻塞，整个进程将被挂起。

内核级线程模型



用户线程与 KSE 为一对一（1:1)的映射关系。此模型下的线程由内核负责管理，应用程序对线程的创建、终止和同步都必须通过内核提供的系统调用来完成，内核可以分别对每一个线程进行调度。所以，一对一线程模型可以真正的实现线程的并发运行，大部分语言实现的线程库基本上都属于这种方式。但是，此模型下线程的创建、切换和同步都需要花费更多的内核资源和时间，如果一个进程包含了大量的线程，那么它会给内核的调度器造成非常大的负担，甚至会影响到操作系统的整体性能。

两级线程模型



用户线程与 KSE 为多对多（N:M）的映射关系。两级线程模型吸收前两种线程模型的优点并且尽量规避了它们的缺点，区别于用户级线程模型，两级线程模型中的进程可以与多个内核线程 KSE 关联，也就是说一个进程内的多个线程可以分别绑定一个自己的 KSE，这点和内核级线程模型相似；其次，又区别于内核级线程模型，它的进程里的线程并不与 KSE 唯一绑定，而是可以多个用户线程映射到同一个 KSE，当某个 KSE 因为其绑定的线程的阻塞操作被内核调度出 CPU 时，其关联的进程中其余用户线程可以重新与其他 KSE 绑定运行。所以，两级线程模型既不是用户级线程模型那种完全靠自己调度的也不是内核级线程模型完全靠操作系统调度的，而是一种自身调度与系统调度协同工作的中间态，**即用户调度器实现用户线程到 KSE 的调度，内核调度器实现 KSE 到 CPU 上的调度。**

Go 的并发机制

在 Go 的并发编程模型中，不受操作系统内核管理的独立控制流不叫用户线程或线程，而称为 Goroutine。Goroutine 通常被认为是协程的 Go 实现，实际上 Goroutine 并不是传统意义上的协程，传统的协程库属于用户级线程模型，而 Goroutine 结合 Go 调度器的底层实现上属于两级线程模型。

Go 搭建了一个特有的两级线程模型。由 Go 调度器实现 Goroutine 到 KSE 的调度，由内核调度器实现 KSE 到 CPU 上的调度。Go 的调度器使用 G、M、P 三个结构体来实现 Goroutine 的调度，也称之为**GMP 模型。**

GMP 模型

G：表示 Goroutine。每个 Goroutine 对应一个 G 结构体，G 存储 Goroutine 的运行堆栈、状态以及任务函数，可重用。当 Goroutine 被调离 CPU 时，调度器代码负责把 CPU 寄存器的

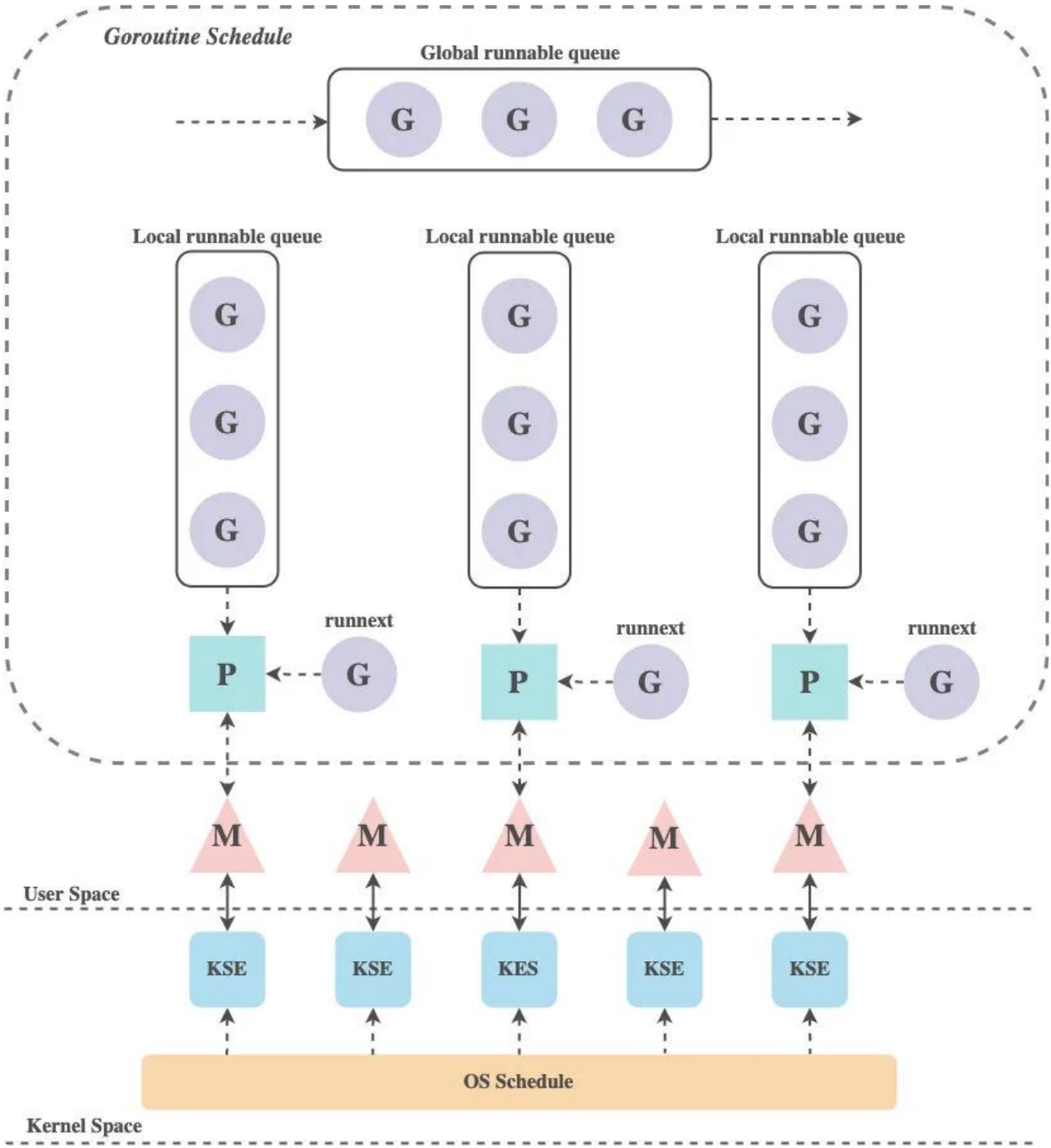
值保存在 G 对象的成员变量之中，当 Goroutine 被调度起来运行时，调度器代码又负责把 G 对象的成员变量所保存的寄存器的值恢复到 CPU 的寄存器

M：OS 底层线程的抽象，它本身就与一个内核线程进行绑定，每个工作线程都有唯一的一个 M 结构体的实例对象与之对应，它代表着真正执行计算的资源，由操作系统的调度器调度和管理。M 结构体对象除了记录着工作线程的诸如栈的起止位置、当前正在执行的 Goroutine 以及是否空闲等等状态信息之外，还通过指针维持着与 P 结构体的实例对象之间的绑定关系

P：表示逻辑处理器。对 G 来说，P 相当于 CPU 核，G 只有绑定到 P(在 P 的 local runq 中)才能被调度。对 M 来说，P 提供了相关的执行环境(Context)，如内存分配状态(mcache)，任务队列(G)等。它维护一个局部 Goroutine 可运行 G 队列，工作线程优先使用自己的局部运行队列，只有必要时才会去访问全局运行队列，这可以大大减少锁冲突，提高工作线程的并发性，并且可以良好的运用程序的局部性原理

一个 G 的执行需要 P 和 M 的支持。一个 M 在与一个 P 关联之后，就形成了一个有效的 G 运行环境（内核线程+上下文）。每个 P 都包含一个可运行的 G 的队列（runq）。该队列中的 G 会被依次传递给与本地 P 关联的 M，并获得运行时机。

M 与 KSE 之间总是一一对应的关系，一个 M 仅能代表一个内核线程。M 与 KSE 之间的关联非常稳固，一个 M 在其生命周期内，会且仅会与一个 KSE 产生关联，而 M 与 P、P 与 G 之间的关联都是可变的，M 与 P 也是一对一的关系，P 与 G 则是一对多的关系。



G

运行时，G 在调度器中的地位与线程在操作系统中差不多，但是它占用了更小的内存空间，也降低了上下文切换的开销。它是 Go 语言在用户态提供的线程，作为一种粒度更细的资源调度单元，使用得当，能够在高并发的场景下更高效地利用机器的 CPU。

g 结构体部分源码（src/runtime/runtime2.go）：

```
type g struct {
    stack    stack  // Goroutine的栈内存范围[stack.lo, stack.hi)
    stackguard0   uintptr // 用于调度器抢占式调度
    m     *m  // Goroutine占用的线程
    sched    gobuf  // Goroutine的调度相关数据
    atomicstatus  uint32 // Goroutine的状态
    ...
}

type gobuf struct {
    sp  uintptr  // 栈指针
    pc  uintptr  // 程序计数器
    g  guintptr  // gobuf对应的Goroutine
    ret  sys.Uintewg // 系统调用的返回值
    ...
}
```

gobuf 中保存的内容会在调度器保存或恢复上下文时使用，其中栈指针和程序计数器会用来存储或恢复寄存器中的值，改变程序即将执行的代码。

atomicstatus 字段存储了当前 Goroutine 的状态，Goroutine 主要可能处于以下几种状态：

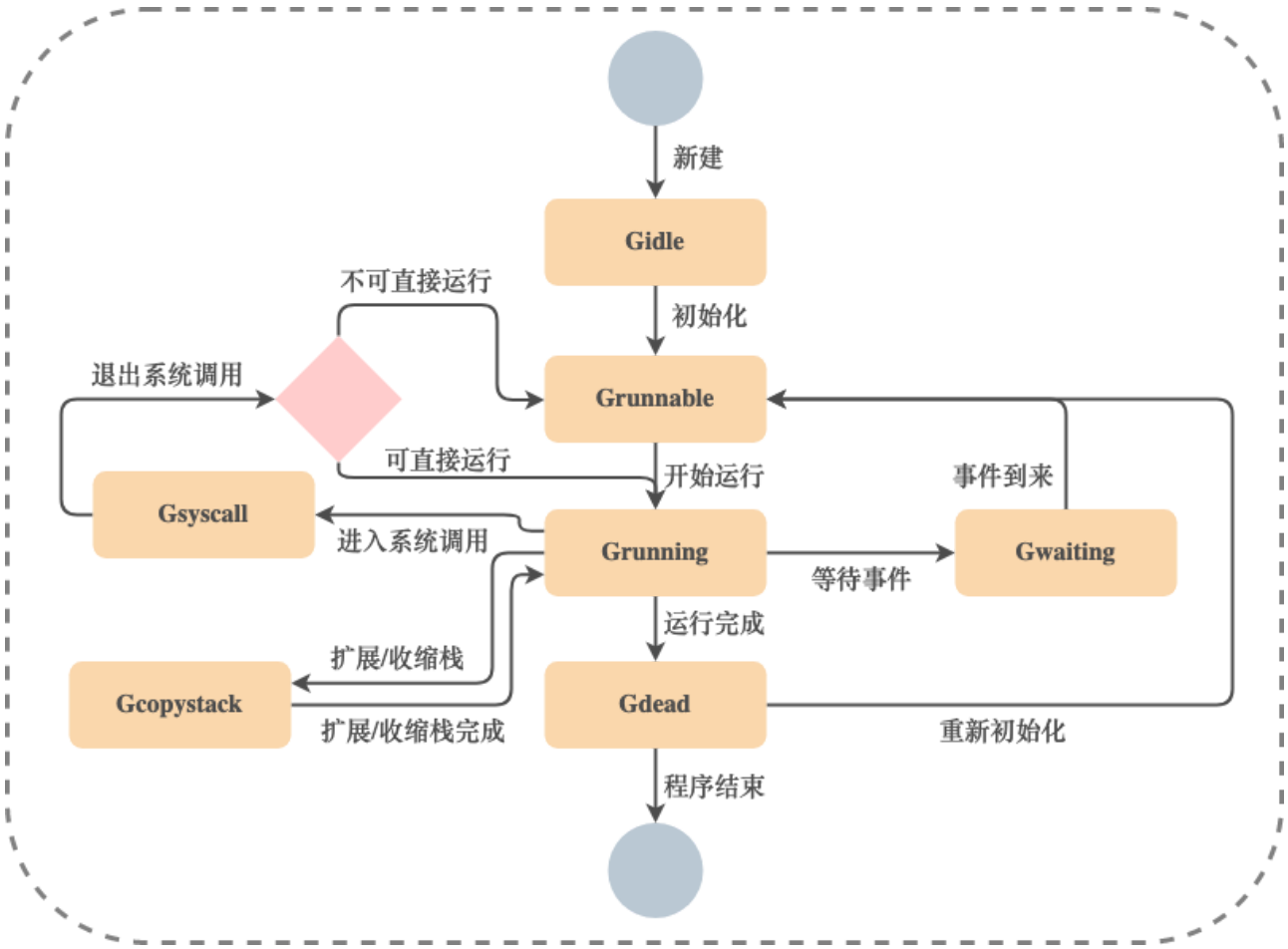
状态	描述
_Gidle	刚刚被分配并且还没有被初始化
_Grunnable	没有执行代码，没有栈的所有权，存储在运行队列中
_Grunning	可以执行代码，拥有栈的所有权，被赋予了内核线程 M 和处理器 P
_Gsyscall	正在执行系统调用，拥有栈的所有权，没有执行用户代码，被赋予了内核线程 M 但是不在运行队列上
_Gwaiting	由于运行时而被阻塞，没有执行用户代码并且不在运行队列上，但是可能存在于 Channel 的等待队列上
_Gdead	没有被使用，没有执行代码，可能有分配的栈
_Gcopystack	栈正在被拷贝，没有执行代码，不在运行队列上
_Gpreempted	由于抢占而被阻塞，没有执行用户代码并且不在运行队列上，等待唤醒
_Gscan	GC 正在扫描栈空间，没有执行代码，可以与其他状态同时存在

Goroutine 的状态迁移是一个十分复杂的过程，触发状态迁移的方法也很多。这里主要介绍一下比较常见的五种状态_Grunnable、_Grunning、_Gsyscall、_Gwaiting 和_Gpreempted。

可以将这些不同的状态聚合成三种：等待中、可运行、运行中，运行期间会在这三种状态来回切换：

- 等待中：Goroutine 正在等待某些条件满足，例如：系统调用结束等，包括_Gwaiting、_Gsyscall 和_Gpreempted 几个状态
- 可运行：Goroutine 已经准备就绪，可以在线程运行，如果当前程序中有非常多的 Goroutine，每个 Goroutine 就可能会等待更多的时间，即_Grunnable
- 运行中：Goroutine 正在某个线程上运行，即_Grunning

G 常见的状态转换图：



G状态转换

进入死亡状态的 G 可以重新初始化并使用。

M

Go 语言并发模型中的 M 是操作系统线程。调度器最多可以创建 10000 个线程，但是最多只会有 GOMAXPROCS（P 的数量）个活跃线程能够正常运行。在默认情况下，运行时会将 GOMAXPROCS 设置成当前机器的核数，我们也可以在程序中使用 runtime.GOMAXPROCS 来改变最大的活跃线程数。

例如，对于一个四核的机器，runtime 会创建四个活跃的操作系统线程，每一个线程都对应一个运行时中的 runtime.m 结构体。在大多数情况下，我们都会使用 Go 的默认设置，也就是线程数等于 CPU 数，默认的设置不会频繁触发操作系统的线程调度和上下文切换，所有的调度都会发生在用户态，由 Go 语言调度器触发，能够减少很多额外开销。

m 结构体源码（部分）：

```
type m struct {
    g0   *g   // 一个特殊的goroutine，执行一些运行时任务
    gsignal  *g   // 处理signal的G
    curg  *g   // 当前M正在运行的G的指针
    p   puintptr // 正在与当前M关联的P
    nextp  puintptr // 与当前M潜在关联的P
    oldp  puintptr // 执行系统调用之前使用线程的P
    spinning bool  // 当前M是否正在寻找可运行的G
    lockedg  *g   // 与当前M锁定的G
}
```

g0 表示一个特殊的 Goroutine，由 Go 运行时系统在启动之处创建，它会深度参与运行时的调度过程，包括 Goroutine 的创建、大内存分配和 CGO 函数的执行。curg 是在当前线程上运行的用户 Goroutine。

P

调度器中的处理器 P 是线程和 Goroutine 的中间层，它能提供线程需要的上下文环境，也会负责调度线程上的等待队列，通过处理器 P 的调度，每一个内核线程都能够执行多个 Goroutine，它能在 Goroutine 进行一些 I/O 操作时及时让出计算资源，提高线程的利用率。

P 的数量等于 GOMAXPROCS，设置 GOMAXPROCS 的值只能限制 P 的最大数量，对 M 和 G 的数量没有任何约束。当 M 上运行的 G 进入系统调用导致 M 被阻塞时，运行时系统会把该

M 和与之关联的 P 分离开来，这时，如果该 P 的可运行 G 队列上还有未被运行的 G，那么运行时系统就会找一个空闲的 M，或者新建一个 M 与该 P 关联，满足这些 G 的运行需要。因此，M 的数量很多时候都会比 P 多。

p 结构体源码（部分）：

```
type p struct {
 // p 的状态
 status   uint32
 // 对应关联的 M
 m        muintptr
 // 可运行的Goroutine队列，可无锁访问
 runqhead uint32
 runqtail uint32
 runq     [256]guintptr
 // 缓存可立即执行的G
 runnext  guintptr
 // 可用的G列表，G状态等于Gdead
 gFree struct {
  gList
  n int32
 }
 ...
}
```

P 可能处于的状态如下：

状态	描述
_Pidle	P 已初始化，但未与任何 M 关联
_Prunning	P 正与某个 M 关联并运行 G
_Psyscall	当前运行的 G 正在执行系统调用
_pgcstop	运行时系统需要停止调度，如垃圾回收前的准备
_Pdead	P 已经不被使用

调度器

两级线程模型中的一部分调度任务会由操作系统之外的程序承担。在 Go 语言中，调度器就负责这一部分调度任务。调度的主要对象就是 G、M 和 P 的实例。每个 M（即每个内核线程）在运行过程中都会执行一些调度任务，他们共同实现了 Go 调度器的调度功能。

g0 和 m0

运行时系统中的每个 M 都会拥有一个特殊的 G，一般称为 M 的 g0。M 的 g0 不是由 Go 程序中的代码间接生成的，而是由 Go 运行时系统在初始化 M 时创建并分配给该 M 的。M 的 g0 一般用于执行调度、垃圾回收、栈管理等方面的任务。M 还会拥有一个专用于处理信号的 G，称为 gsignal。

除了 g0 和 gsignal 之外，其他由 M 运行的 G 都可以视为用户级别的 G，简称用户 G，g0 和 gsignal 可称为系统 G。Go 运行时系统会进行切换，以使每个 M 都可以交替运行用户 G 和它的 g0。这就是前面所说的“每个 M 都会运行调度程序”的原因。

除了每个 M 都拥有属于它自己的 g0 外，还存在一个 runtime.g0。runtime.g0 用于执行引导程序，它运行在 Go 程序拥有的第一个内核线程之中，这个线程也称为 runtime.m0，runtime.m0 的 g0 就是 runtime.g0。

核心元素的容器

上面讲了 Go 的线程实现模型中的 3 个核心元素——G、M 和 P，下面看看承载这些元素实例的容器：

名称	结构	作用域	说明
全局 M 列表	runtime.allm	运行时系统	存放所有 M 的单向链表
全局 P 列表	runtime.allp	运行时系统	存放所有 P 的数组
全局 G 列表	runtime.allgs	运行时系统	存放所有 G 的切片
调度器的空闲 M 列表	sched.midle	调度器	存放空闲 M 的单向链表
调度器的空闲 P 列表	sched.pidle	调度器	存放空闲 P 的单向链表
调度器的可运行 G 列表	sched.runqhead	调度器	存放可运行 G 的队列
调度器的自由 G 列表	sched.gfreeStack	调度器	存放自由 G 的单向链表
P 的可运行 G 列表	p.runq	本地 P	存放当前 P 中可运行 G 的队列
P 的自由 G 列表	p.gfree	本地 P	存放当前 P 中自由 G 的链表

和 G 相关的四个容器值得我们特别注意，任何 G 都会存在于全局 G 列表中，其余四个容器只会存放当前作用域内的、具有某个状态的 G。两个可运行的 G 列表中的 G 都拥有几乎平等的运行机会，只不过不同时机的调度会把 G 放在不同的地方，例如，从 Gsyscall 状态转移出来的 G 都会被放入调度器的可运行 G 队列，而刚刚被初始化的 G 都会被放入本地 P 的可运行 G 队列。此外，这两个可运行 G 队列之间也会互相转移 G，例如，本地 P 的可运行 G 队列已满时，其中一半的 G 会被转移到调度器的可运行 G 队列中。

调度器的空闲 M 列表和空闲 P 列表用于存放暂时不被使用的元素实例。运行时系统需要时，会从中获取相应元素的实例并重新启用它。

调度循环

调用 runtime.schedule 进入调度循环：

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

top:
 var gp *g
 var inheritTime bool

 if gp == nil {
        // 为了公平，每调用schedule函数61次就要从全局可运行G队列中获取
  if _g_.m.p.ptr().schedtick%61 == 0 && sched.runqsize > 0 {
   lock(&sched.lock)
   gp = globrunqget(_g_.m.p.ptr(), 1)
   unlock(&sched.lock)
  }
 }
    // 从P本地获取G任务
 if gp == nil {
  gp, inheritTime = runqget(_g_.m.p.ptr())
 }
    // 运行到这里表示从本地运行队列和全局运行队列都没有找到需要运行的G
 if gp == nil {
        // 阻塞地查找可用G
  gp, inheritTime = findrunnable()
 }
    // 执行G任务函数
```

```
 execute(gp, inheritTime)
}
```

runtime.schedule 函数会从下面几个地方查找待执行的 Goroutine：

- 为了保证公平，当全局运行队列中有待执行的 Goroutine 时，通过 schedtick 保证有一定几率会从全局的运行队列中查找对应的 Goroutine
- 从处理器本地的运行队列中查找待执行的 Goroutine
- 如果前两种方法都没有找到 G，会通过 findrunnable 函数去其他 P 里面去“偷”一些 G 来执行，如果“偷”不到，就阻塞查找直到有可运行的 G

接下来由 runtime.execute 执行获取的 Goroutine：

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

 // 将G绑定到当前M上
 _g_.m.curg = gp
 gp.m = _g_.m
 // 将g正式切换为_Grunning状态
 casgstatus(gp, _Grunnable, _Grunning)
 gp.waitsince = 0
 // 抢占信号
 gp.preempt = false
 gp.stackguard0 = gp.stack.lo + _StackGuard
 if !inheritTime {
  // 调度器调度次数增加1
  _g_.m.p.ptr().schedtick++
 }
 ...
    // gogo完成从g0到gp的切换
 gogo(&gp.sched)
}
```

当开始执行 execute 后，G 会被切换到_Grunning 状态，并将 M 和 G 进行绑定，最终调用 runtime.gogo 将 Goroutine 调度到当前线程上。runtime.gogo 会从 runtime.gobuf 中取出 runtime.goexit 的程序计数器和待执行函数的程序计数器，并将：

- runtime.goexit 的程序计数器被放到栈 SP 上
- 待执行函数的程序计数器被放到了寄存器 BX 上

```
MOVL gobuf_sp(BX), SP  // 将runtime.goexit函数的PC恢复到SP中
MOVL gobuf_pc(BX), BX  // 获取待执行函数的程序计数器
JMP  BX                // 开始执行
```

当 Goroutine 中运行的函数返回时，程序会跳转到 runtime.goexit 所在位置，最终在当前线程的 g0 的栈上调用 runtime.goexit0 函数，该函数会将 Goroutine 转换为_Gdead 状态、清理其中的字段、移除 Goroutine 和线程的关联并调用 runtime.gfput 将 G 重新加入处理器的 Goroutine 空闲列表 gFree 中：

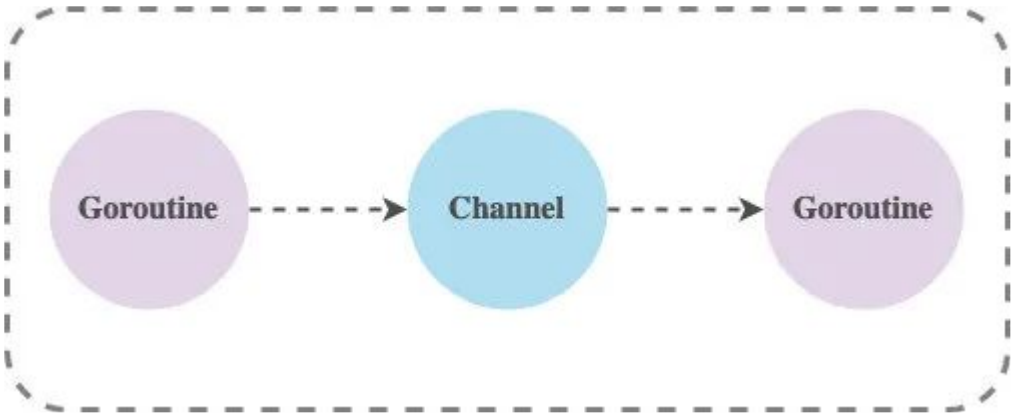
```
func goexit0(gp *g) {
 _g_ := getg()
 // 设置当前G状态为_Gdead
 casgstatus(gp, _Grunning, _Gdead)
 // 清理G
 gp.m = nil
 ...
 gp.writebuf = nil
 gp.waitreason = 0
 gp.param = nil
 gp.labels = nil
 gp.timer = nil

 // 解绑M和G
 dropg()
 ...
 // 将G扔进gfree链表中等待复用
 gfput(_g_.m.p.ptr(), gp)
 // 再次进行调度
 schedule()
}
```

最后 runtime.goexit0 会重新调用 runtime.schedule 触发新一轮的 Goroutine 调度，调度器从 runtime.schedule 开始，最终又回到 runtime.schedule，这就是 Go 语言的调度循环。

Channel

Go 中经常被人提及的一个设计模式：不要通过共享内存的方式进行通信，而是应该通过通信的方式共享内存。Goroutine 之间会通过 channel 传递数据，作为 Go 语言的核心数据结构和 Goroutine 之间的通信方式，channel 是支撑 Go 语言高性能并发编程模型的重要结构。



channel 在运行时的内部表示是 runtime.hchan，该结构体中包含了用于保护成员变量的互斥锁，从某种程度上说，channel 是一个用于同步和通信的有锁队列。hchan 结构体源码：

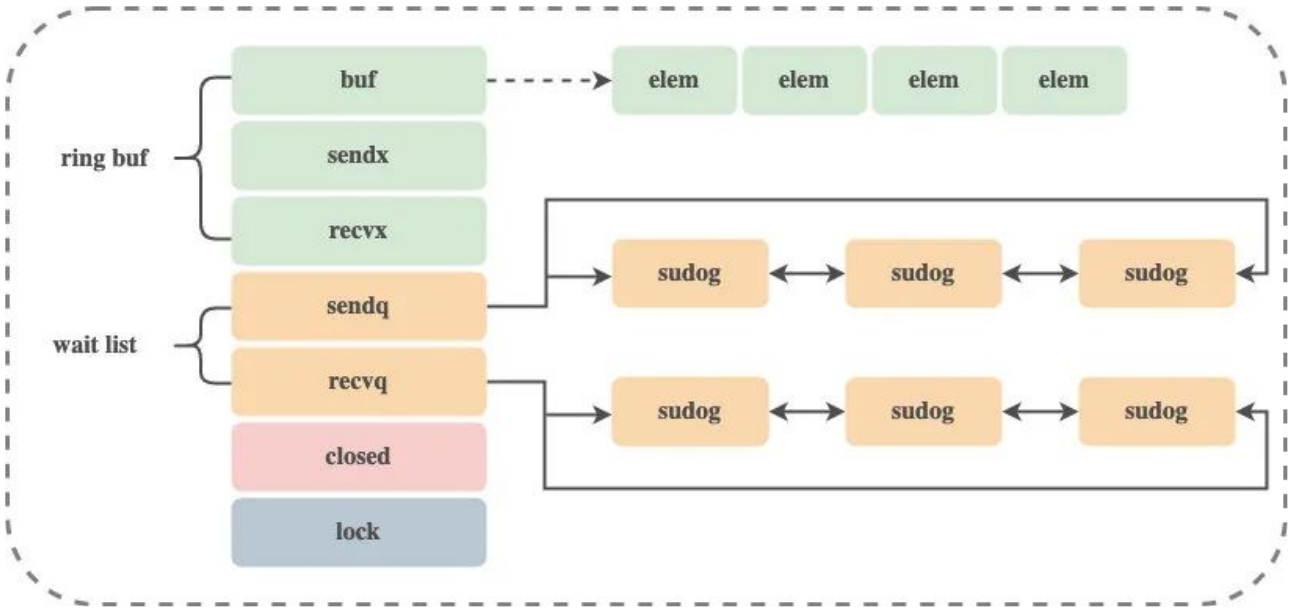
```
type hchan struct {
    qcount   uint     // 循环列表元素个数
    dataqsiz  uint     // 循环队列的大小
    buf     unsafe.Pointer // 循环队列的指针
    elemsize  uint16    // chan中元素的大小
    closed   uint32    // 是否已close
    elemtype  *_type    // chan中元素类型
    sendx    uint     // chan的发送操作处理到的位置
    recvx    uint     // chan的接收操作处理到的位置
    recvq    waitq     // 等待接收数据的Goroutine列表
    sendq    waitq     // 等待发送数据的Goroutine列表

    lock    mutex     // 互斥锁
}

type waitq struct {     // 双向链表
    first *sudog
```

```
    last *sudog
}
```

waitq 中连接的是一个 sudog 双向链表，保存的是等待中的 Goroutine。



创建 chan

使用 make 关键字来创建管道，make(chan int， 3)会调用到 runtime.makechan 函数中：

```
const (
 maxAlign  = 8
 hchanSize = unsafe.Sizeof(hchan{}) + uintptr(-int(unsafe.Sizeof(hchan{}))&(maxAlign-1))
)

func makechan(t *chantype, size int) *hchan {
 elem := t.elem

    // 计算需要分配的buf空间大小
 mem, overflow := math.MulUintptr(elem.size, uintptr(size))
 if overflow || mem > maxAlloc-hchanSize || size < 0 {
  panic(plainError("makechan: size out of range"))
 }

 var c *hchan
 switch {
 case mem == 0:
```

```
        // chan的大小或者elem的大小为0，不需要创建buf
        c = (*hchan)(mallocgc(hchanSize, nil, true))
        // Race detector uses this location for synchronization.
        c.buf = c.raceaddr()
    case elem.ptrdata == 0:
        // elem不含指针，分配一块连续的内存给hchan数据结构和buf
        c = (*hchan)(mallocgc(hchanSize+mem, nil, true))
        c.buf = add(unsafe.Pointer(c), hchanSize)
    default:
        // elem包含指针，单独分配buf
        c = new(hchan)
        c.buf = mallocgc(mem, elem, true)
    }

    // 更新hchan的elemsize、elemtype、dataqsiz字段
    c.elemsize = uint16(elem.size)
    c.elemtype = elem
    c.dataqsiz = uint(size)

    return c
}
```

上述代码根据 channel 中收发元素的类型和缓冲区的大小初始化 runtime.hchan 和缓冲区：

- 若缓冲区所需大小为 0，就只会为 hchan 分配一段内存
- 若缓冲区所需大小不为 0 且 elem 不包含指针，会为 hchan 和 buf 分配一块连续的内存
- 若缓冲区所需大小不为 0 且 elem 包含指针，会单独为 hchan 和 buf 分配内存

发送数据到 chan

发送数据到 channel，ch <- i 会调用到 runtime.chansend 函数中，该函数包含了发送数据的全部逻辑：

```
func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
    if c == nil {
        // 对于非阻塞的发送，直接返回
```

```
  if !block {
   return false
  }
  // 对于阻塞的通道，将goroutine挂起
  gopark(nil, nil, waitReasonChanSendNilChan, traceEvGoStop, 2)
  throw("unreachable")
 }
    // 加锁
 lock(&c.lock)
    // channel已关闭，panic
 if c.closed != 0 {
  unlock(&c.lock)
  panic(plainError("send on closed channel"))
 }
    ...
}
```

block 表示当前的发送操作是否是阻塞调用。如果 channel 为空，对于非阻塞的发送，直接返回 false，对于阻塞的发送，将 goroutine 挂起，并且永远不会返回。对 channel 加锁，防止多个线程并发修改数据，如果 channel 已关闭，报错并中止程序。

runtime.chansend 函数的执行过程可以分为以下三个部分：

- 当存在等待的接收者时，通过 runtime.send 直接将数据发送给阻塞的接收者
- 当缓冲区存在空余空间时，将发送的数据写入缓冲区
- 当不存在缓冲区或缓冲区已满时，等待其他 Goroutine 从 channel 接收数据

直接发送

如果目标 channel 没有被关闭且 recvq 队列中已经有处于读等待的 Goroutine，那么 runtime.chansend 会从接收队列 recvq 中取出最先陷入等待的 Goroutine 并直接向它发送数据，注意，由于有接收者在等待，所以如果有缓冲区，那么缓冲区一定是空的：

```
func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
    ...
    // 从recvq中取出一个接收者
 if sg := c.recvq.dequeue(); sg != nil {
  // 如果接收者存在，直接向该接收者发送数据，绕过buf
```

```
  send(c, sg, ep, func() { unlock(&c.lock) }, 3)
  return true
 }
    ...
}
```

直接发送会调用 runtime.send 函数：

```
func send(c *hchan, sg *sudog, ep unsafe.Pointer, unlockf func(), skip int) {
 ...
 if sg.elem != nil {
  // 直接把要发送的数据copy到接收者的栈空间
  sendDirect(c.elemtype, sg, ep)
  sg.elem = nil
 }
 gp := sg.g
 unlockf()
 gp.param = unsafe.Pointer(sg)
 if sg.releasetime != 0 {
  sg.releasetime = cputicks()
 }
 // 设置对应的goroutine为可运行状态
 goready(gp, skip+1)
}
```

sendDirect 方法调用 memmove 进行数据的内存拷贝。goready 方法将等待接收数据的 Goroutine 标记成可运行状态（Grunnable）并把该 Goroutine 发到发送方所在的处理器的 runnext 上等待执行，该处理器在下一次调度时会立刻唤醒数据的接收方。注意，只是放到了 runnext 中，并没有立刻执行该 Goroutine。

发送到缓冲区

如果缓冲区未满，则将数据写入缓冲区：

```
func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
 ...
 // 如果缓冲区没有满，直接将要发送的数据复制到缓冲区
```

```
 if c.qcount < c.dataqsiz {
  // 找到buf要填充数据的索引位置
  qp := chanbuf(c, c.sendx)
  ...
  // 将数据拷贝到buf中
  typedmemmove(c.elemtype, qp, ep)
  // 数据索引前移，如果到了末尾，又从0开始
  c.sendx++
  if c.sendx == c.dataqsiz {
   c.sendx = 0
  }
  // 元素个数加1，释放锁并返回
  c.qcount++
  unlock(&c.lock)
  return true
 }
 ...
}
```

找到缓冲区要填充数据的索引位置，调用 typedmemmove 方法将数据拷贝到缓冲区中，然后重新设值 sendx 偏移量。

阻塞发送

当 channel 没有接收者能够处理数据时，向 channel 发送数据会被下游阻塞，使用 select 关键字可以向 channel 非阻塞地发送消息：

```
func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
 ...
 // 缓冲区没有空间了，对于非阻塞调用直接返回
 if !block {
  unlock(&c.lock)
  return false
 }
 // 创建sudog对象
 gp := getg()
 mysg := acquireSudog()
 mysg.releasetime = 0
 if t0 != 0 {
```

```
    mysg.releasetime = -1
  }
  mysg.elem = ep
  mysg.waitlink = nil
  mysg.g = gp
  mysg.isSelect = false
  mysg.c = c
  gp.waiting = mysg
  gp.param = nil
  // 将sudog对象入队
  c.sendq.enqueue(mysg)
  // 进入等待状态
  gopark(chanparkcommit, unsafe.Pointer(&c.lock), waitReasonChanSend, traceEvGoBlockSend, 2)
  ...
}
```

对于非阻塞的调用会直接返回，对于阻塞的调用会创建 sudog 对象并将 sudog 对象加入发送等待队列。调用 gopark 将当前 Goroutine 转入 waiting 状态。调用 gopark 之后，在使用者看来向该 channel 发送数据的代码语句会被阻塞。

发送数据整个流程大致如下：

注意，发送数据的过程中包含几个会触发 Goroutine 调度的时机：

- 发送数据时发现从 channel 上存在等待接收数据的 Goroutine，立刻设置处理器的 runnext 属性，但是并不会立刻触发调度

- 发送数据时并没有找到接收方并且缓冲区已经满了，这时会将自己加入 channel 的 sendq 队列并调用 gopark 触发 Goroutine 的调度让出处理器的使用权

从 chan 接收数据

从 channel 获取数据最终调用到 runtime.chanrecv 函数：

```
func chanrecv(c *hchan, ep unsafe.Pointer, block bool) (selected, received bool) {
 if c == nil {
        // 如果c为空且是非阻塞调用，直接返回
  if !block {
   return
  }
        // 阻塞调用直接等待
  gopark(nil, nil, waitReasonChanReceiveNilChan, traceEvGoStop, 2)
  throw("unreachable")
 }
 ...
 lock(&c.lock)
    // 如果c已经关闭，并且c中没有数据，返回
 if c.closed != 0 && c.qcount == 0 {
  unlock(&c.lock)
  if ep != nil {
   typedmemclr(c.elemtype, ep)
  }
  return true, false
 }
    ...
}
```

当从一个空 channel 接收数据时，直接调用 gopark 让出处理器使用权。如果当前 channel 已被关闭且缓冲区中没有数据，直接返回。

runtime.chanrecv 函数的具体执行过程可以分为以下三个部分：

- 当存在等待的发送者时，通过 runtime.recv 从阻塞的发送者或者缓冲区中获取数据
- 当缓冲区存在数据时，从 channel 的缓冲区中接收数据
- 当缓冲区中不存在数据时，等待其他 Goroutine 向 channel 发送数据

直接接收

当 channel 的 sendq 队列中包含处于发送等待状态的 Goroutine 时，调用 runtime.recv 直接从这个发送者那里提取数据。注意，由于有发送者在等待，所以如果有缓冲区，那么缓冲区一定是满的。

```
func chanrecv(c *hchan, ep unsafe.Pointer, block bool) (selected, received bool) {
 ...
 // 从发送者队列获取数据
 if sg := c.sendq.dequeue(); sg != nil {
  // 发送者队列不为空，直接从发送者那里提取数据
  recv(c, sg, ep, func() { unlock(&c.lock) }, 3)
  return true, true
 }
 ...
}
```

主要看一下 runtime.recv 的实现：

```
func recv(c *hchan, sg *sudog, ep unsafe.Pointer, unlockf func(), skip int) {
    // 如果是无缓冲区chan
 if c.dataqsiz == 0 {
  if ep != nil {
            // 直接从发送者拷贝数据
   recvDirect(c.elemtype, sg, ep)
  }
    // 有缓冲区chan
 } else {
        // 获取buf的存放数据指针
  qp := chanbuf(c, c.recvx)
        // 直接从缓冲区拷贝数据给接收者
  if ep != nil {
   typedmemmove(c.elemtype, ep, qp)
  }
        // 从发送者拷贝数据到缓冲区
  typedmemmove(c.elemtype, qp, sg.elem)
  c.recvx++
  c.sendx = c.recvx // c.sendx = (c.sendx+1) % c.dataqsiz
 }
```

```
 gp := sg.g
 gp.param = unsafe.Pointer(sg)
    // 设置对应的goroutine为可运行状态
 goready(gp, skip+1)
}
```

该函数会根据缓冲区的大小分别处理不同的情况：

- 如果 channel 不存在缓冲区
  - 直接从发送者那里提取数据
- 如果 channel 存在缓冲区
  - 将缓冲区中的数据拷贝到接收方的内存地址
  - 将发送者数据拷贝到缓冲区，并唤醒发送者

无论发生哪种情况，运行时都会调用 goready 将等待发送数据的 Goroutine 标记成可运行状态（Grunnable）并将当前处理器的 runnext 设置成发送数据的 Goroutine，在调度器下一次调度时将阻塞的发送方唤醒。

从缓冲区接收

如果 channel 缓冲区中有数据且发送者队列中没有等待发送的 Goroutine 时，直接从缓冲区中 recvx 的索引位置取出数据：

```
func chanrecv(c *hchan, ep unsafe.Pointer, block bool) (selected, received bool) {
 ...
    // 如果缓冲区中有数据
 if c.qcount > 0 {
  qp := chanbuf(c, c.recvx)
        // 从缓冲区复制数据到ep
  if ep != nil {
   typedmemmove(c.elemtype, ep, qp)
  }
  typedmemclr(c.elemtype, qp)
        // 接收数据的指针前移
  c.recvx++
        // 环形队列，如果到了末尾，再从0开始
  if c.recvx == c.dataqsiz {
```

```
    c.recvx = 0
   }
         // 缓冲区中现存数据减一
   c.qcount--
         unlock(&c.lock)
   return true, true
  }
  ...
}
```

阻塞接收

当 channel 的发送队列中不存在等待的 Goroutine 并且缓冲区中也不存在任何数据时，从管道中接收数据的操作会被阻塞，使用 select 关键字可以非阻塞地接收消息：

```
func chanrecv(c *hchan, ep unsafe.Pointer, block bool) (selected, received bool) {
  ...
  // 非阻塞，直接返回
  if !block {
   unlock(&c.lock)
   return false, false
  }
  // 创建sudog
  gp := getg()
  mysg := acquireSudog()
  ...
  gp.waiting = mysg
  mysg.g = gp
  mysg.isSelect = false
  mysg.c = c
  gp.param = nil
  // 将sudog添加到等待接收队列中
  c.recvq.enqueue(mysg)
  // 阻塞Goroutine，等待被唤醒
  gopark(chanparkcommit, unsafe.Pointer(&c.lock), waitReasonChanReceive, traceEvGoBlockRecv, 2)
  ...
}
```

如果是非阻塞调用，直接返回。阻塞调用会将当前 Goroutine 封装成 sudog，然后将 sudog 添加到等待接收队列中，调用 gopark 让出处理器的使用权并等待调度器的调度。

注意，接收数据的过程中包含几个会触发 Goroutine 调度的时机：

- 当 channel 为空时
- 当 channel 的缓冲区中不存在数据并且 sendq 中也不存在等待的发送者时

关闭 chan

关闭通道会调用到 runtime.closechan 方法：

```
func closechan(c *hchan) {
    // 校验逻辑
    ...
    lock(&c.lock)
    // 设置chan已关闭
 c.closed = 1
 var glist gList
    // 获取所有接收者
 for {
  sg := c.recvq.dequeue()
  if sg == nil {
   break
  }
  if sg.elem != nil {
   typedmemclr(c.elemtype, sg.elem)
   sg.elem = nil
  }
  gp := sg.g
  gp.param = nil
  glist.push(gp)
 }
 // 获取所有发送者
 for {
  sg := c.sendq.dequeue()
  ...
 }
    unlock(&c.lock)
    // 唤醒所有glist中的goroutine
 for !glist.empty() {
```

```
    gp := glist.pop()
    gp.schedlink = 0
    goready(gp, 3)
  }
 }
```

将 recvq 和 sendq 两个队列中的 Goroutine 加入到 gList 中，并清除所有 sudog 上未被处理的元素。最后将所有 glist 中的 Goroutine 加入调度队列，等待被唤醒。注意，发送者在被唤醒之后会 panic。

总结一下发送/接收/关闭操作可能引发的结果：

Goroutine 和 channel 的实现共同支撑起了 Go 语言的并发机制。

近期好文：

Git 全功能介绍

MongoDB 全方位知识图谱

大牛书单 | Python方向的好书