channel/select 源码分析

欧长坤 | Ou Changkun | changkun.de
University of Munich, Ph.D. Candidate

Go 夜读 | 第 56 期
August 22, 2019

主要内容

- 概述
- channel/select 回顾
- channel 的结构设计
- channel 的初始化行为
- channel 的发送与接收过程及其性能优化
- channel 的回收
- select 的本质及其相关编译器优化

从并发编程模型说起

- 从内存的角度看，并行计算只有两种：共享内存、消息通信（拷贝内存）
- 目的是为了解决多线程数据的一致性
- 基于共享内存的并发模型通常提供互斥锁同步原语
- Tony Hoare 1977 年基于消息通信使用 channel 原语提出并发的 Communication Sequential Processes (CSP) 数学理论
- Go 将其实现为**显式**的 channel 同步原语
- Go 同时提供 sync.*、atomic.* 的基于共享内存的同步原语

channel 101: buffered 和 unbuffered

buffered channel:

```
ch := make(chan interface{}, 1)

// write
ch <- v

// read
v <- ch

close(ch)
```

unbuffered channel:

```
ch := make(chan interface{})

// write
ch <- v

// read
v <- ch

close(ch)
```

Go 内存模型：***happens before*** 偏序 (≤)

在 channel 中体现为：

```
-   buffered: ch <- v   ≤  v <- ch
- unbuffered: v  <- ch  ≤ ch <- v
```

好像不太符合直觉?

- buffered channel *生产* 数据并存入 buffer, 然后 reader 从 buffer 中进行*消费*;
- unbuffered channel 会阻塞到 reader 从 channel 中读取数据

现在似乎符合直觉了

select 101: zero-case, uni-case 和 multi-case select

```
// zero-case
select {}
```

- 为什么 zero-case 会发生永久阻塞？

```
// uni-case
select {
case ch <- v:
    (...)
}
```

- uni-case 情况中，不同类型的 channel 会得到何种不同的结果，会发生什么？

```
// multi-case
select {
case v1 <- ch1:
    (...)
case v2 <- ch2:
    (...)
default:
    (...)
}
```

- multi-case 情况中， ch1 和 ch2 的数据如果同时有效，v1 与 v2 之间读取数据 channel 的顺序？

… 等等更多这样的问题

「源码之前，了无秘密」

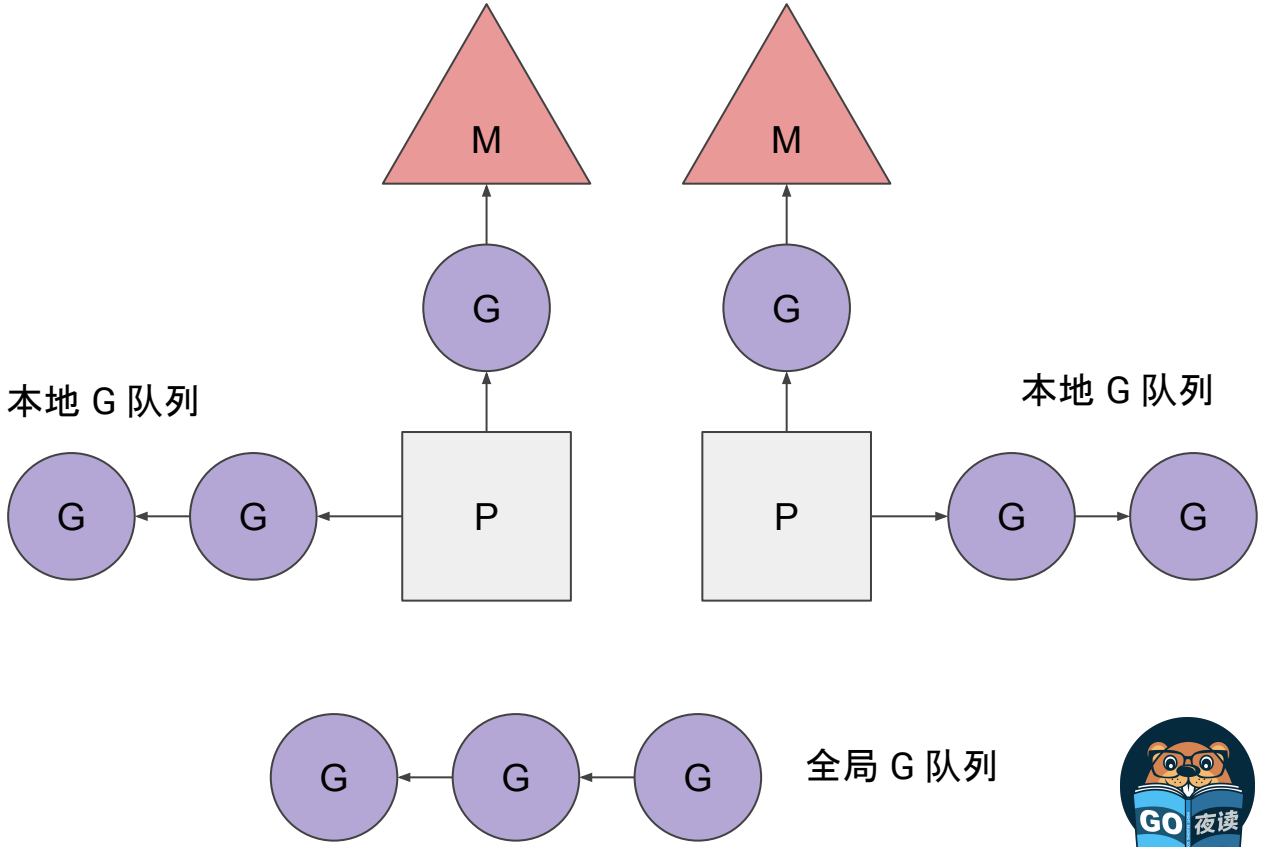
Go Scheduler in 5 Minutes

G: Goroutine
被调度的实体，即用户代码，在本地队列中不断的切换执行

M: Machine
传统线程实体，即系统线程，负责代码的执行

P: Processor
处理器抽象，目的是实现串联 G 的本地队列，当 M 持有 P 时，访问的 G 不会出现数据竞争

WSS：工作窃取调度
1. M 切换出 G 后，将其插入本地队列尾部
2. 本地队列为空时从其他队列队尾偷取 G
3. 都没有则去偷全局队列
4. 没有工作则休眠



channel 相关的翻译工作

src/runtime/chan.go:

```
1. make(chan interface{}, size) ⇒ runtime.makechan(interface{}, size)
   make(chan interface{})       ⇒ runtime.makechan(interface{}, 0)
2. ch <- v                      ⇒ runtime.chansend1(ch, &v)
3. v <- ch                      ⇒ runtime.chanrecv1(ch, &v)
   v, ok <- ch                  ⇒ ok := runtime.chanrecv2(ch, &v)
4. close(ch)                    ⇒ runtime.closechan(ch)
```

我们一个一个的来看

hchan 结构

```go
type hchan struct {
    qcount   uint           // 队列中的所有数据数
    dataqsiz uint           // 环形队列的大小
    buf      unsafe.Pointer // 指向大小为 dataqsiz 的数组
    elemsize uint16         // 元素大小
    closed   uint32         // 是否关闭
    elemtype *_type         // 元素类型
    sendx    uint           // 发送索引
    recvx    uint           // 接收索引
    recvq    waitq          // recv 等待列表，即 ( <-ch )
    sendq    waitq          // send 等待列表，即 ( ch<- )
    lock mutex
}
type waitq struct { // 等待队列 sudog 双向队列
    first *sudog
    last  *sudog
}
```

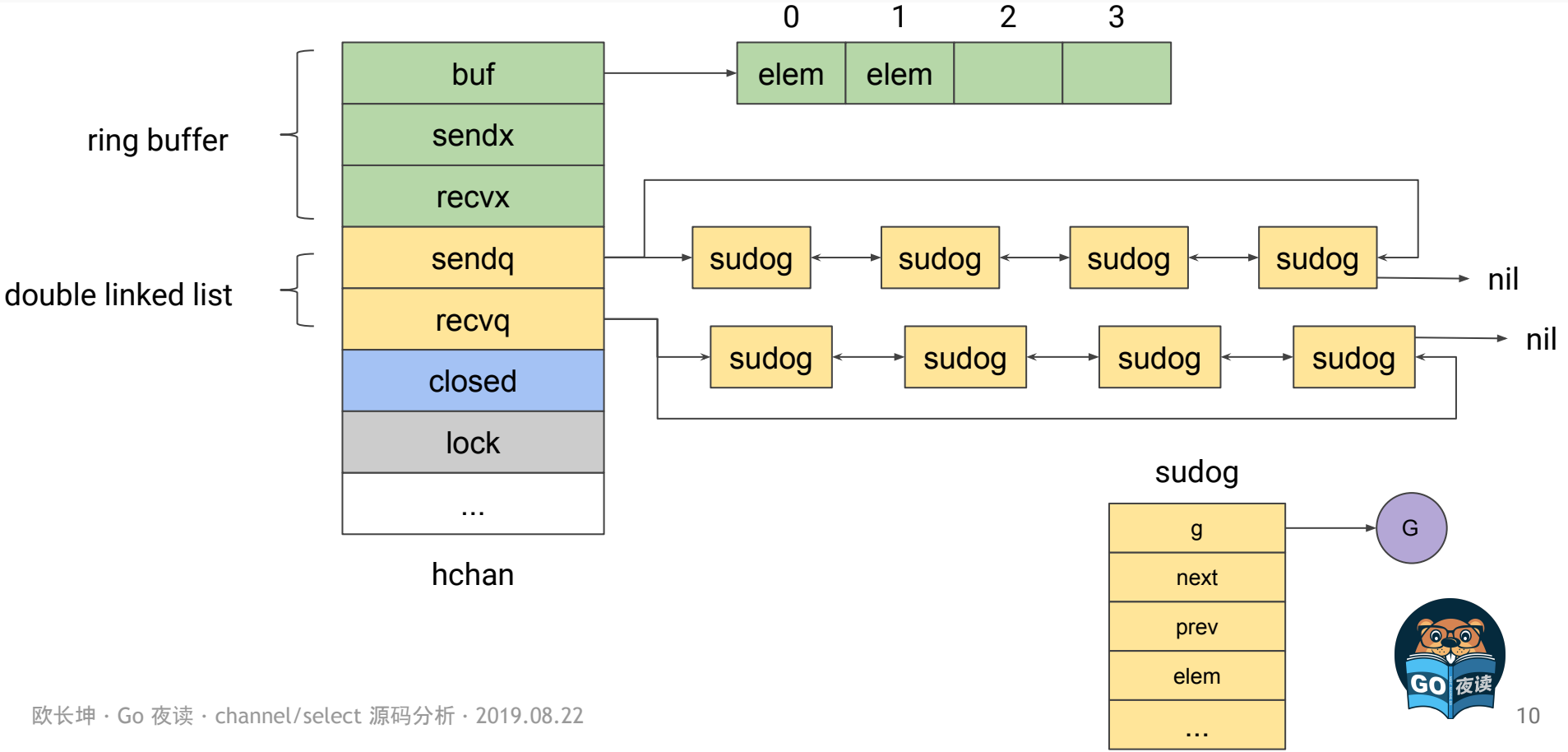
```go
type sudog struct {
    // 下面的字段由这个 sudog 阻塞的通道的 hchan.lock 进行保护。
    // shrinkstack 依赖于它服务于 sudog 相关的 channel 操作。

    g *g

    // isSelect 表示 g 正在参与一个 select
    isSelect bool
    next     *sudog
    prev     *sudog
    elem     unsafe.Pointer // 数据元素（可能指向栈）

    ( ... )
    c           *hchan // channel
}
```

hchan 结构 (可视化)

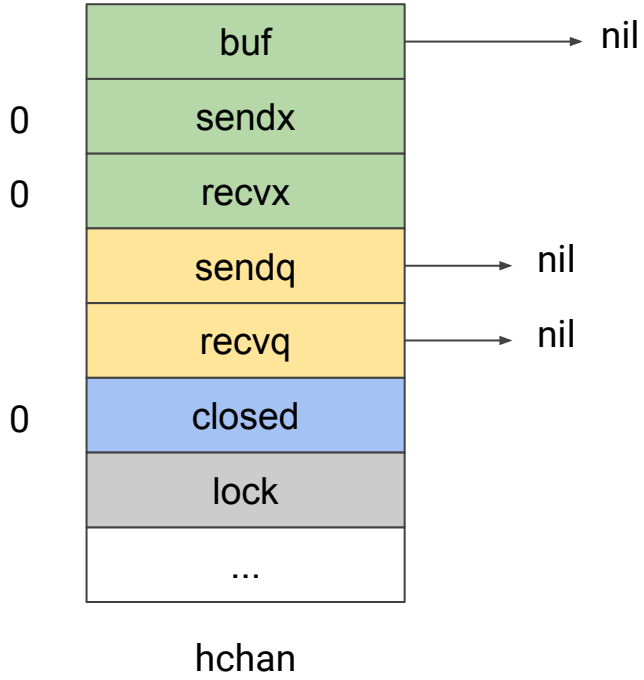

0
1
2
3
buf
elem
elem
ring buffer
sendx
recvx
sendq
sudog
sudog
sudog
sudog
nil
double linked list
recvq
sudog
sudog
sudog
sudog
nil
closed
lock
...
hchan
sudog
g
G
next
prev
elem
...

1. 创建 buffered channel (1)

```
make(chan interface{}, 4)
```

1.1 分配 hchan

从堆中分配，所有字段均为零值

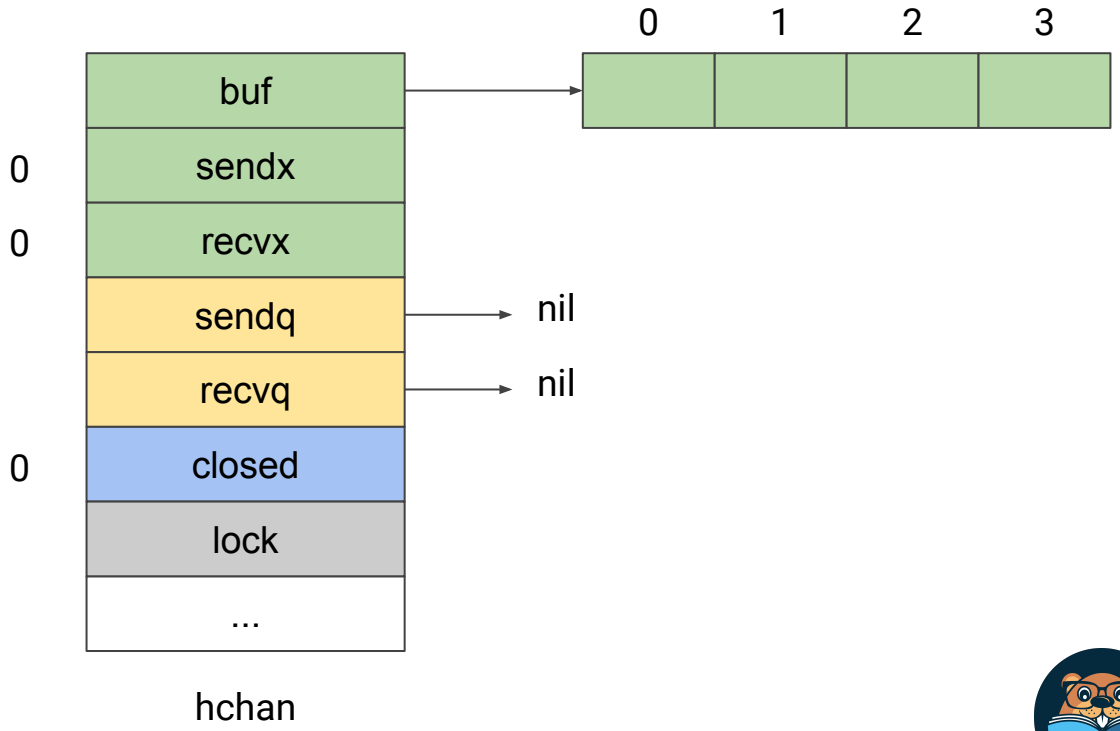


1. 创建 buffered channel (2)

```
make(chan interface{}, 4)
```

1.1 分配 hchan
从堆中分配，所有字段均为零值

1.2 分配 ring buffer
从堆中分配，根据给定大小创建



1. 创建 unbuffered channel

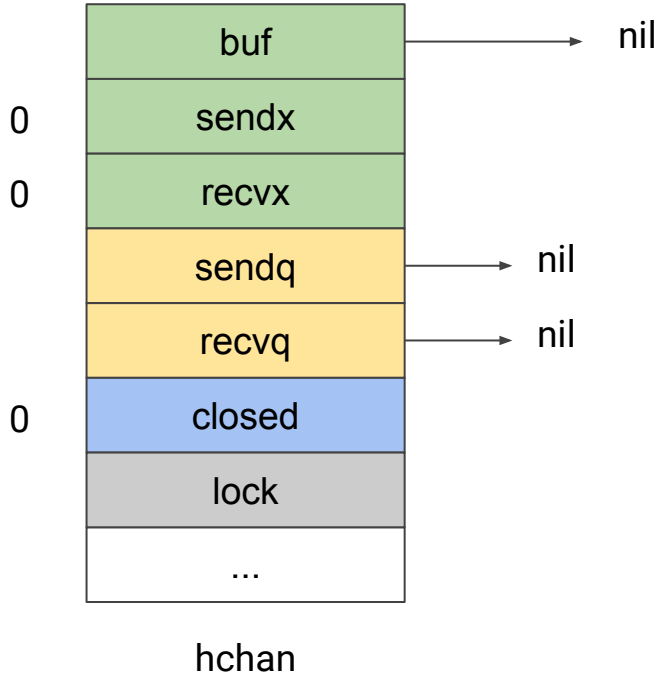
```
make(chan interface{})
```

1.1 分配 hchan
从堆中分配，所有字段均为零值

1.2 不分配 ring buffer

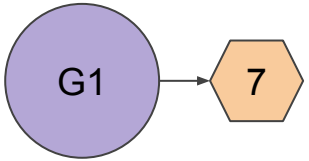
> 到目前为止未发现 buffered channel 和 unbuffered channel 的存在任何差异

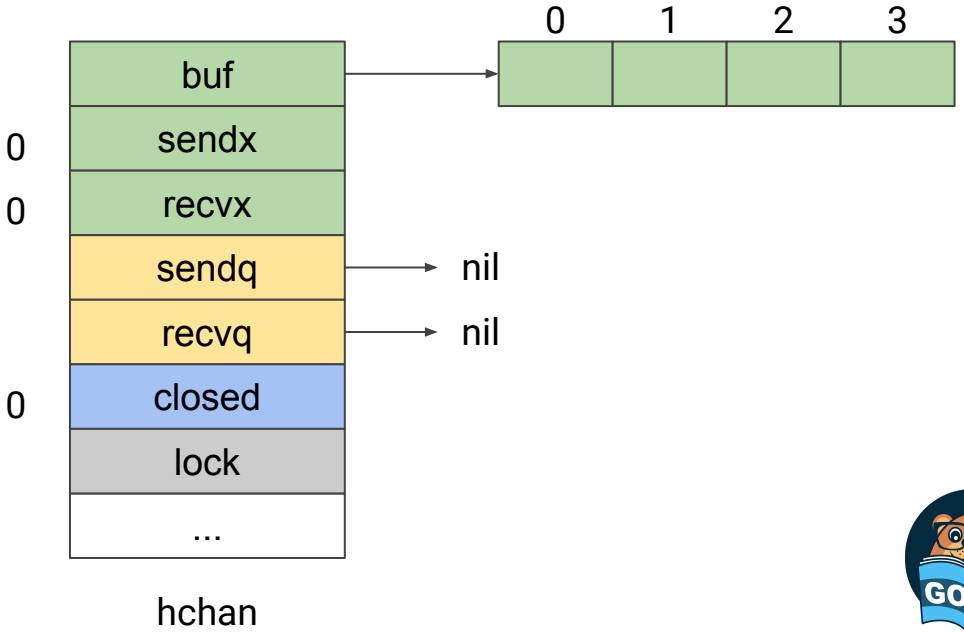
> 实际实现中，分配过程是一次性完成的，hchan 及其 ring buffer 是一段连续的内存



2. 向 buffered channel 发送数据 (1)

```
ch <- v
```

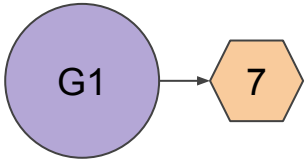


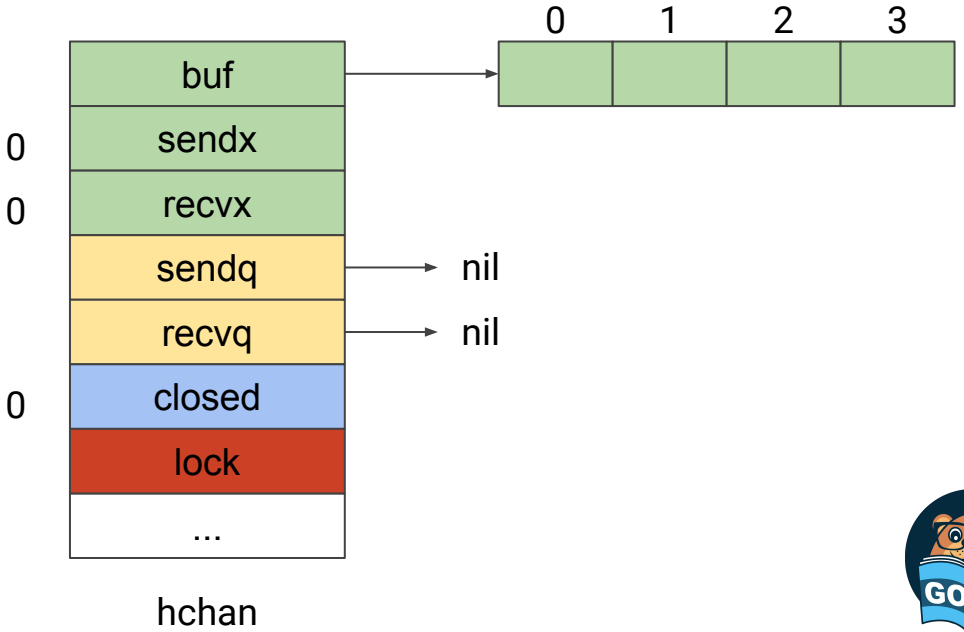


2. 向 buffered channel 发送数据 (2)

```
ch <- v
```

2.1 对 channel 上锁





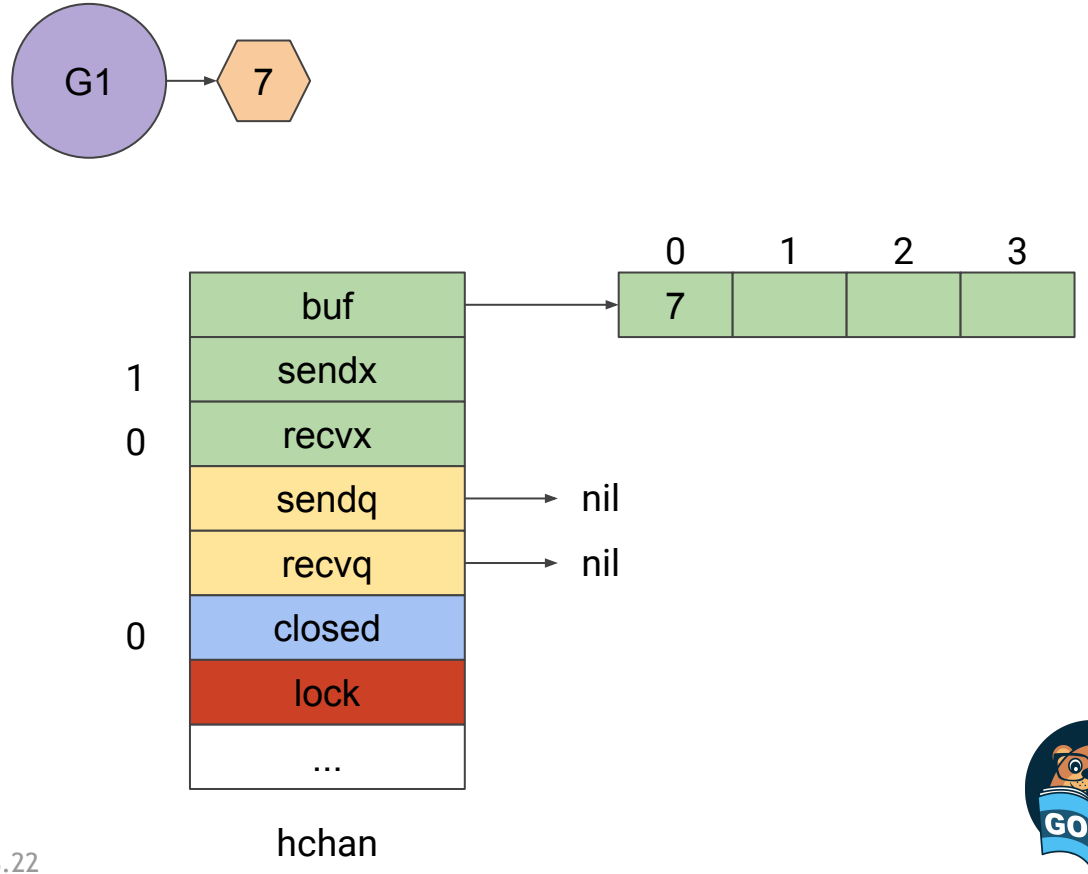
2. 向 buffered channel 发送数据 (3)

ch <- v

2.1 对 channel 上锁

2.2 存入 buf

在 buf 中拷贝要发送的数据



2. 向 buffered channel 发送数据 (4)

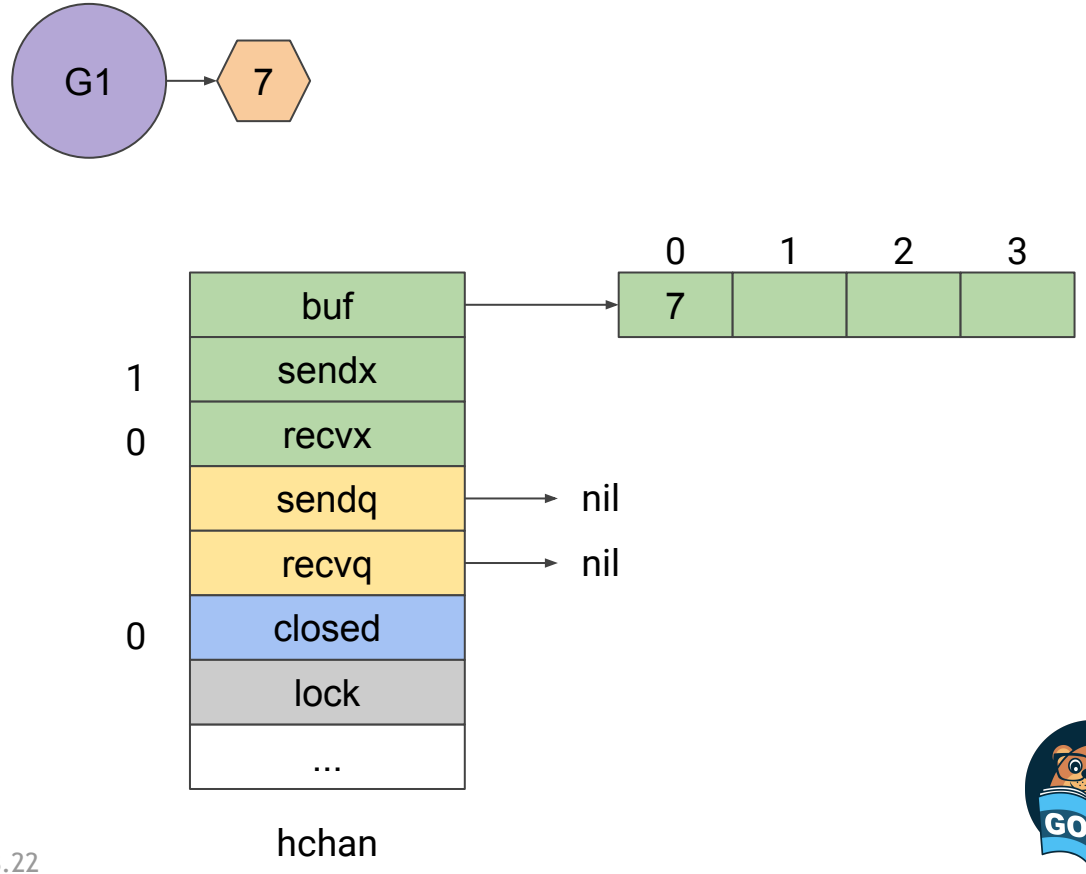
ch <- v

2.1 对 channel 上锁

2.2 存入 buf

在 buf 中拷贝要发送的数据

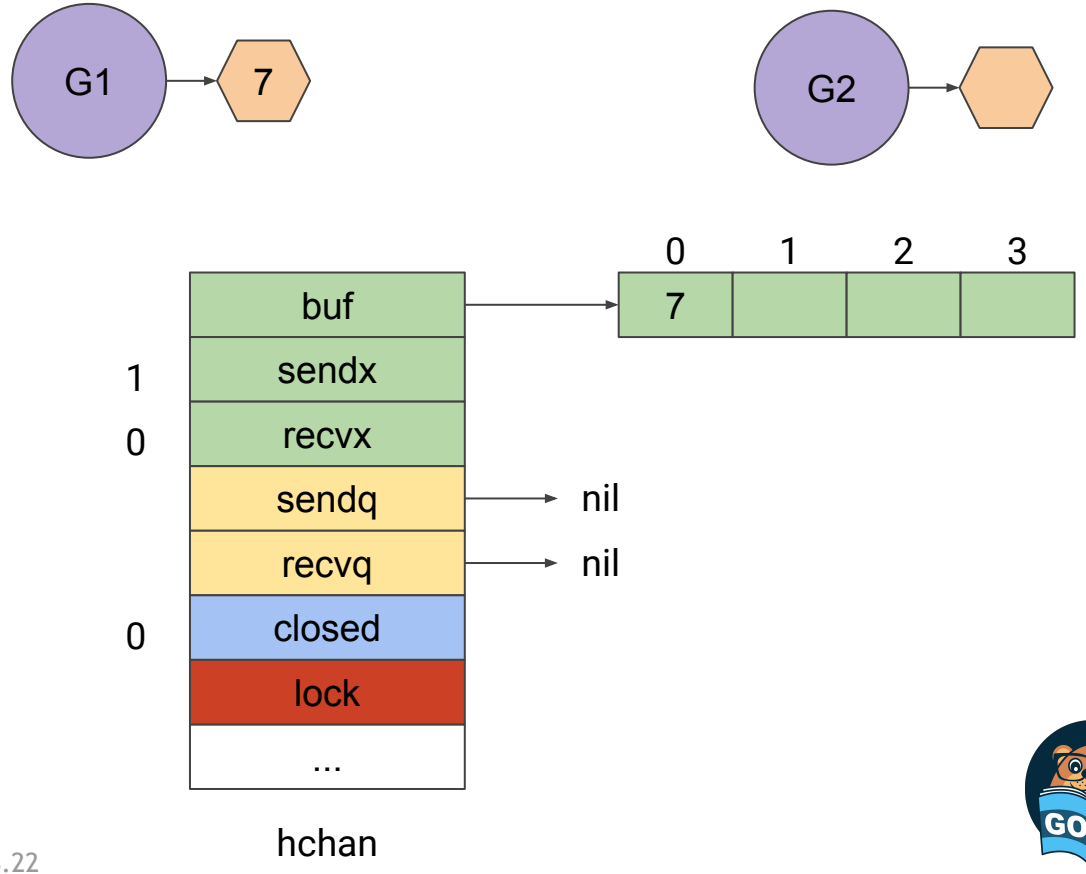
2.3 对 channel 解锁



3. 从 buffered channel 接受数据 (1)

`v <- ch`

3.1 对 channel 上锁

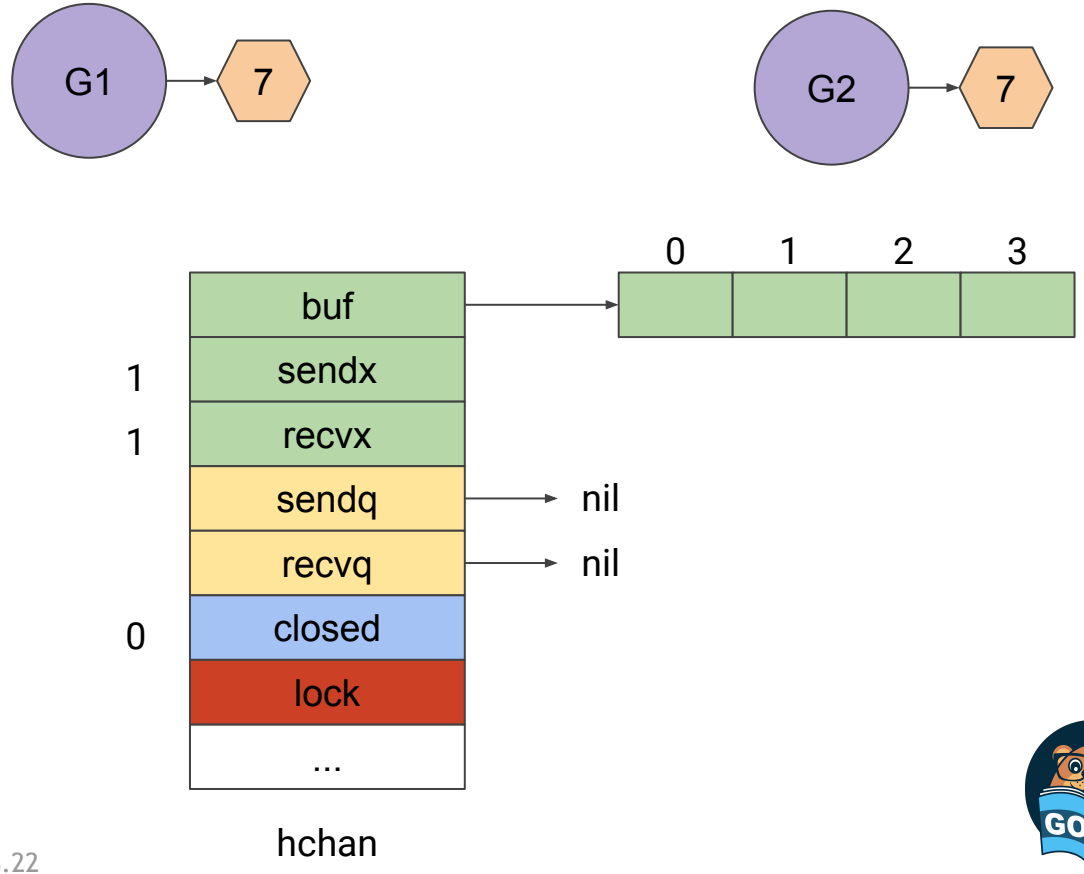


3. 从 buffered channel 接受数据 (2)

v <- ch

3.1 对 channel 上锁

3.2 从 buf 取出
从 buf 中拷出要发送的数据



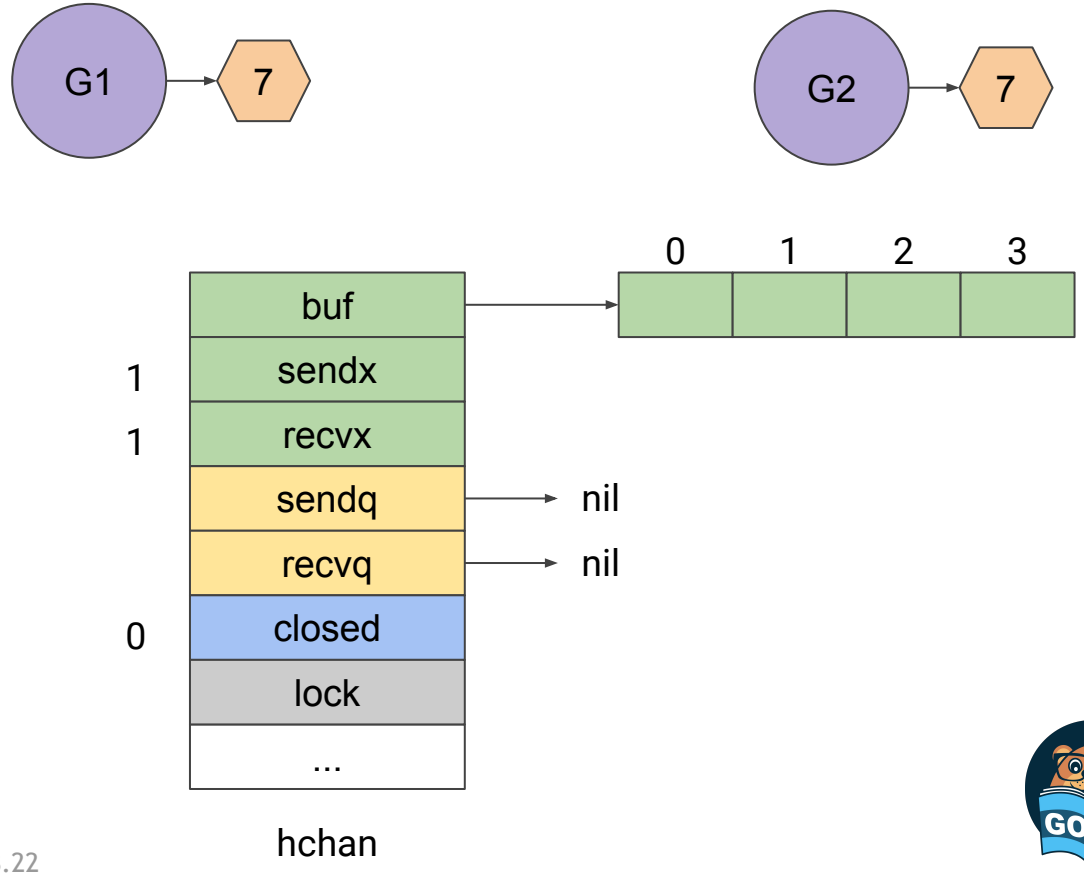
3. 从 buffered channel 接受数据 (3)

v <- ch

3.1 对 channel 上锁

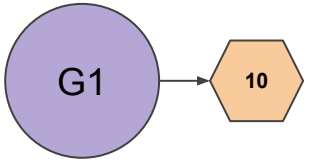
3.2 从 buf 取出
从 buf 中拷出要发送的数据

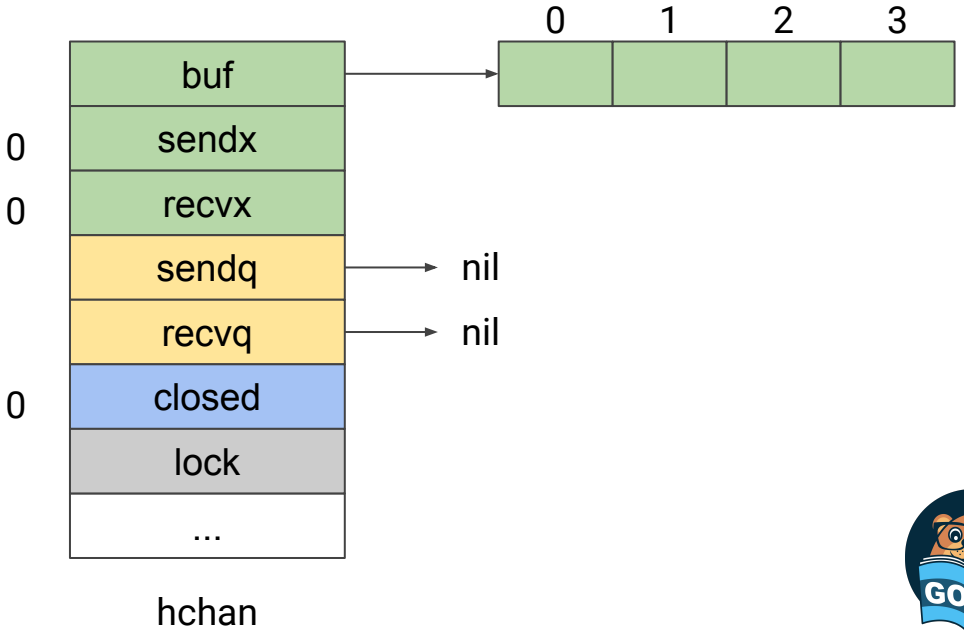
3.3 对 channel 解锁



如果发送时候 buf 已满呢？(1)

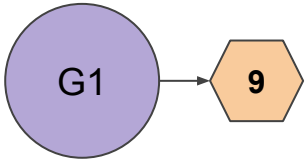
```
ch <- v
```

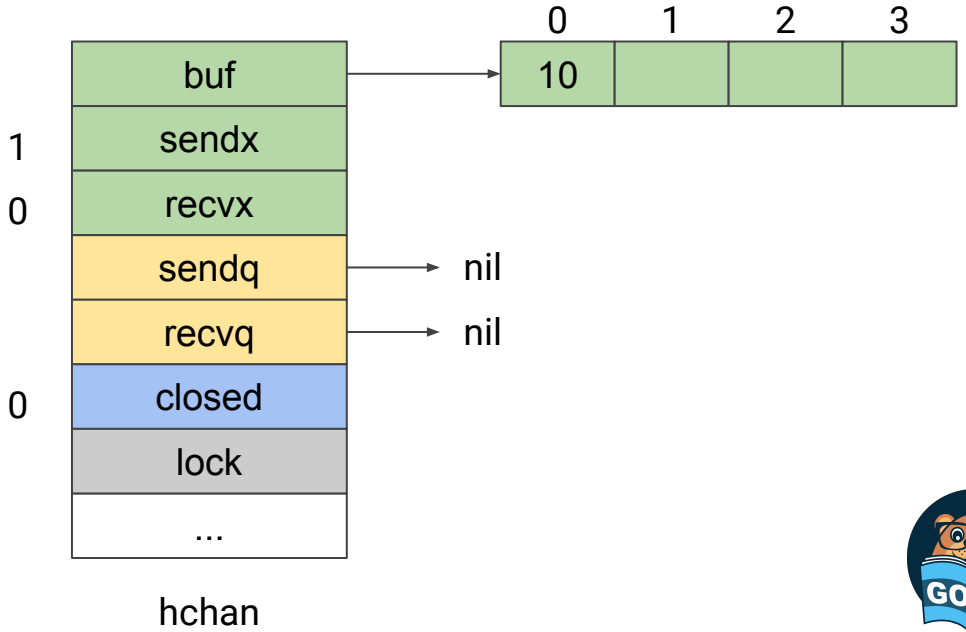




如果发送时候 buf 已满呢？(2)

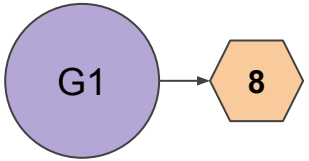
```
ch <- v
```

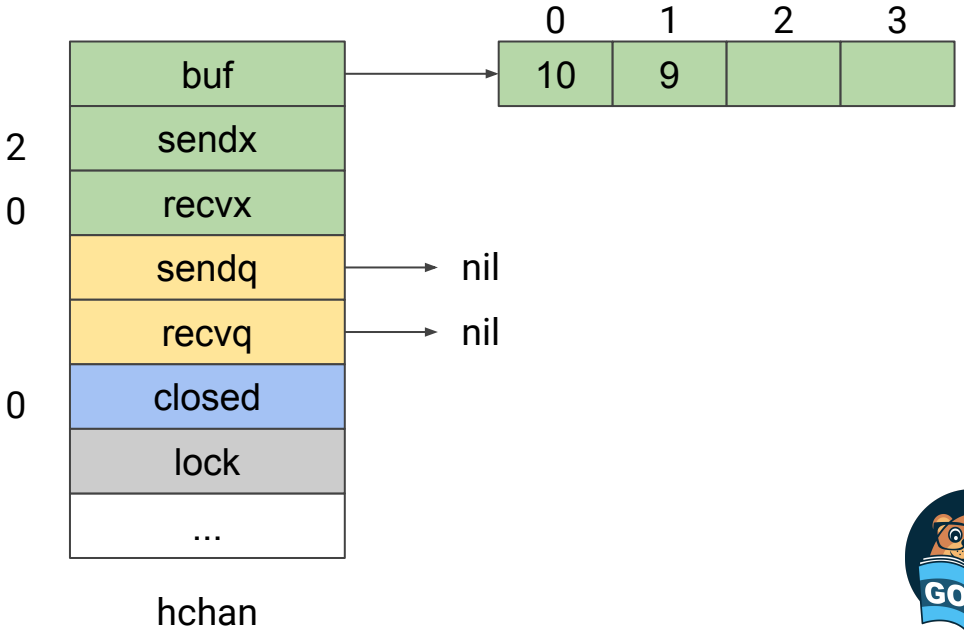




如果发送时候 buf 已满呢？(3)

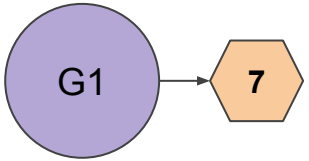
```
ch <- v
```

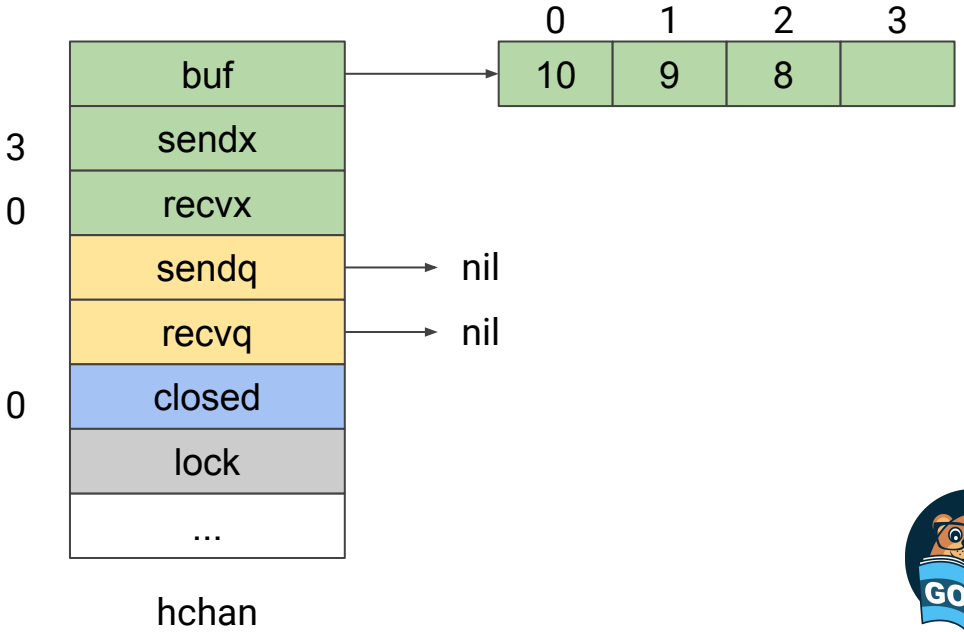




如果发送时候 buf 已满呢？(4)

```
ch <- v
```





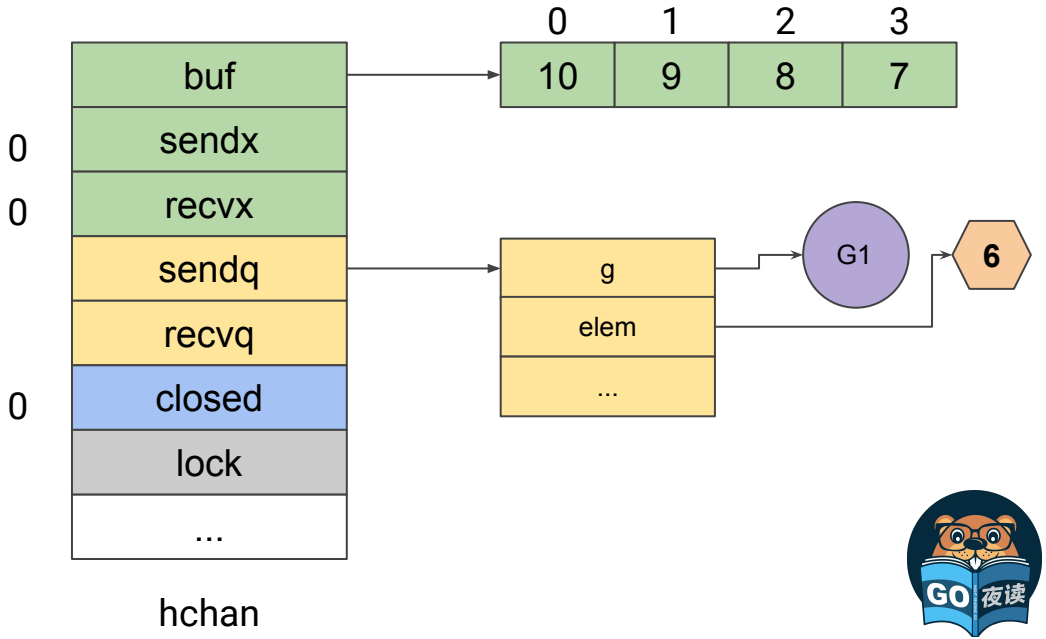
如果发送时候 buf 已满呢？(5)

```
ch <- v
```

1. 使用 sudog 包裹 g 和 要发送的数据

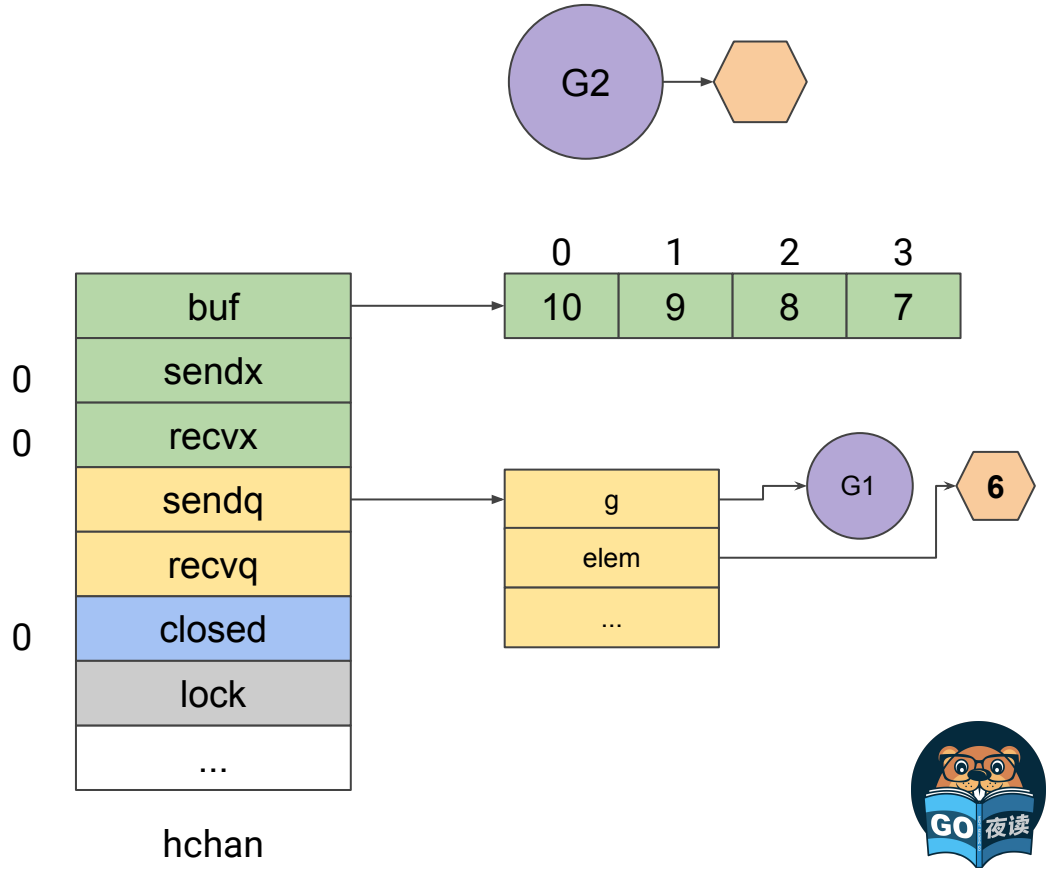
2. 入队 sendq

3. gopark
   + m 解除当前的 g
   + m 重新进入调度循环
   + 这个时候的 g 没有进入调度队列



如果发送时候 buf 已满呢？(6)

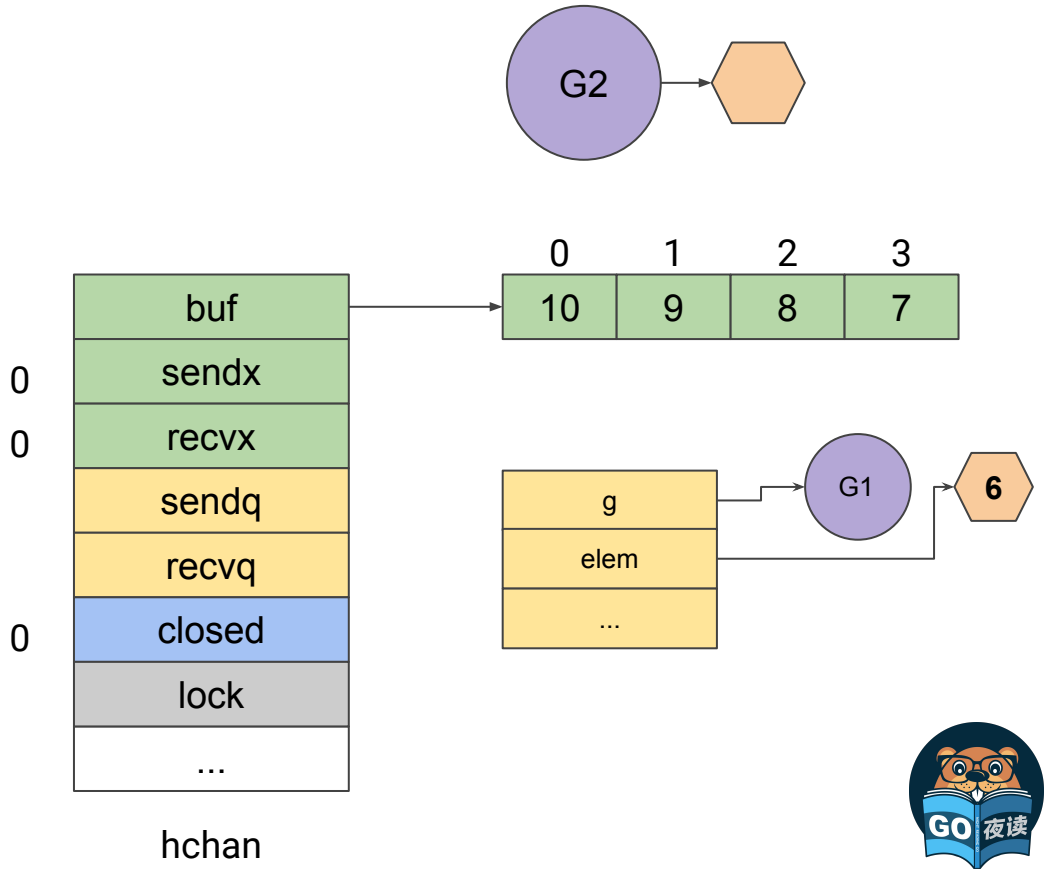
v <- ch
假设出现了一个新的接收方



如果发送时候 buf 已满呢？(7)

v <- ch
假设出现了一个新的接收方

1. sendq 出队

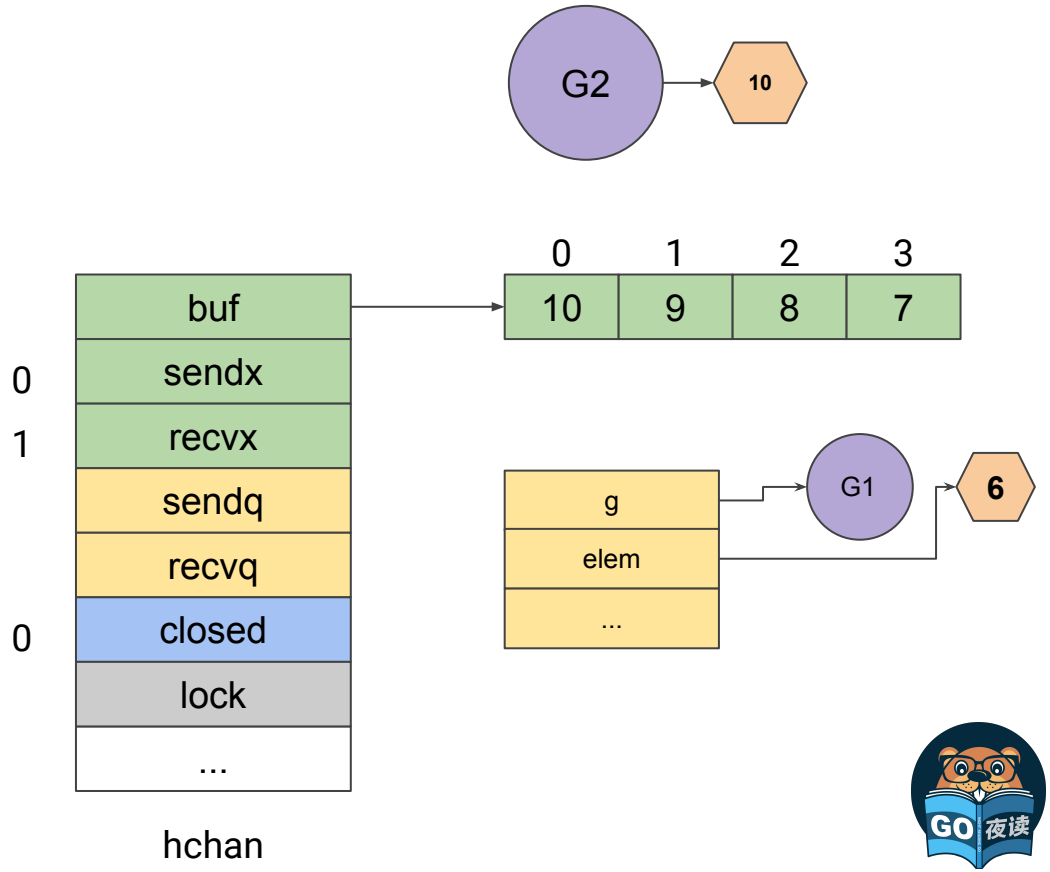


如果发送时候 buf 已满呢？(8)

v <- ch
假设出现了一个新的接收方

1. sendq 出队

2. 从 buf 拷贝队头

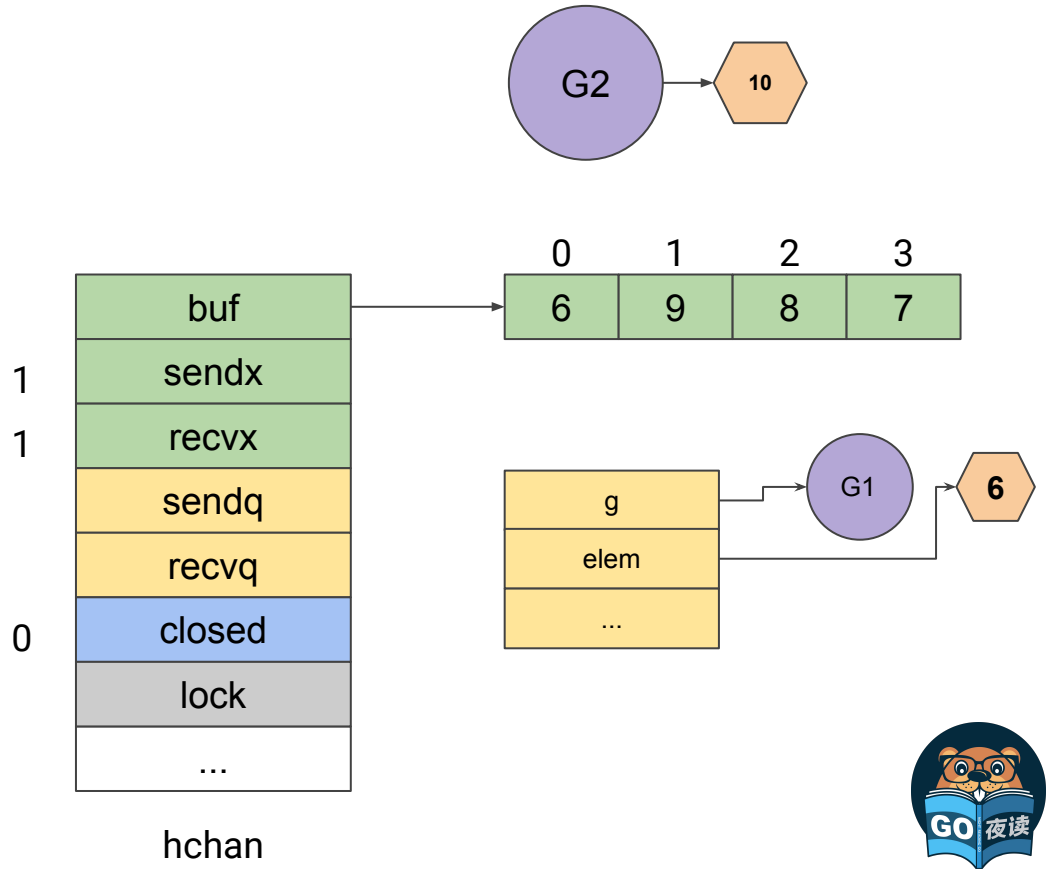


如果发送时候 buf 已满呢？(9)

v <- ch

假设出现了一个新的接收方

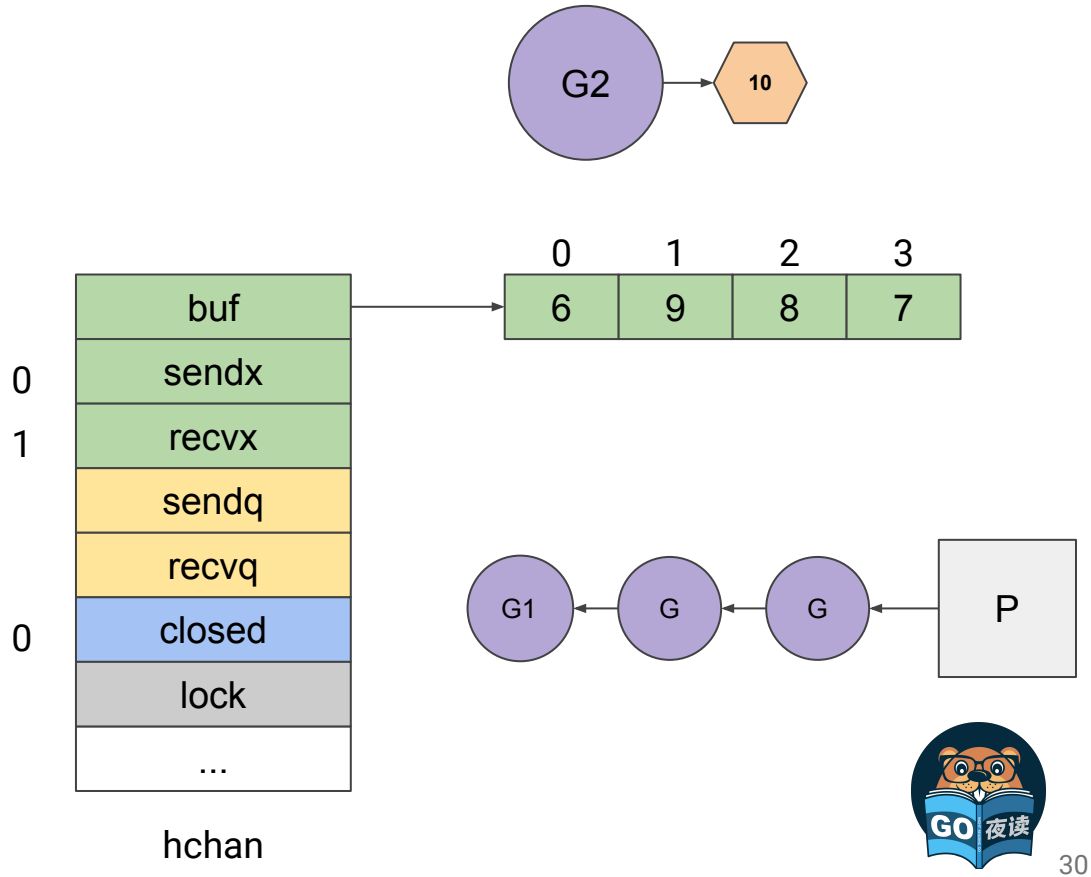
1. sendq 出队
2. 从 buf 拷贝队头
3. 从 sender 拷贝到队尾



如果发送时候 buf 已满呢？(10)

v <- ch
假设出现了一个新的接收方

1. sendq 出队
2. 从 buf 拷贝队头
3. 从 sender 拷贝到队尾
4. goready
   + 放入调度队列
   + 等待被调度



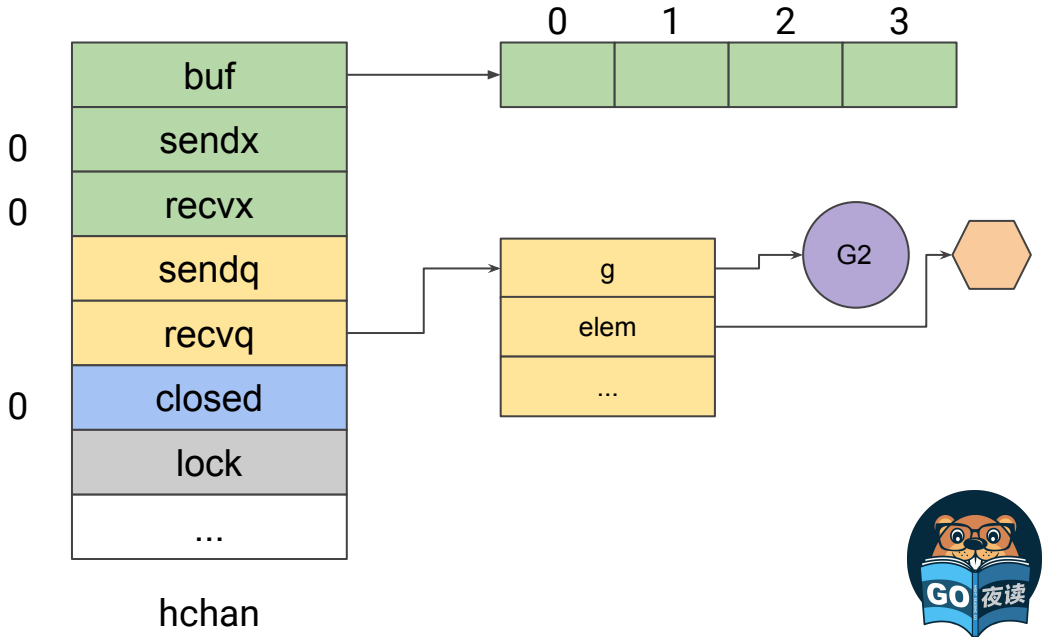
如果接收时候 buf 为空呢？(1)

v <- ch

1. 使用 sudog 包裹 g 和接收数据的位置

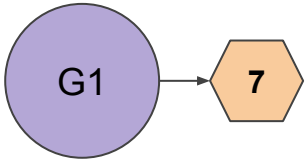
2. 入队 recvq

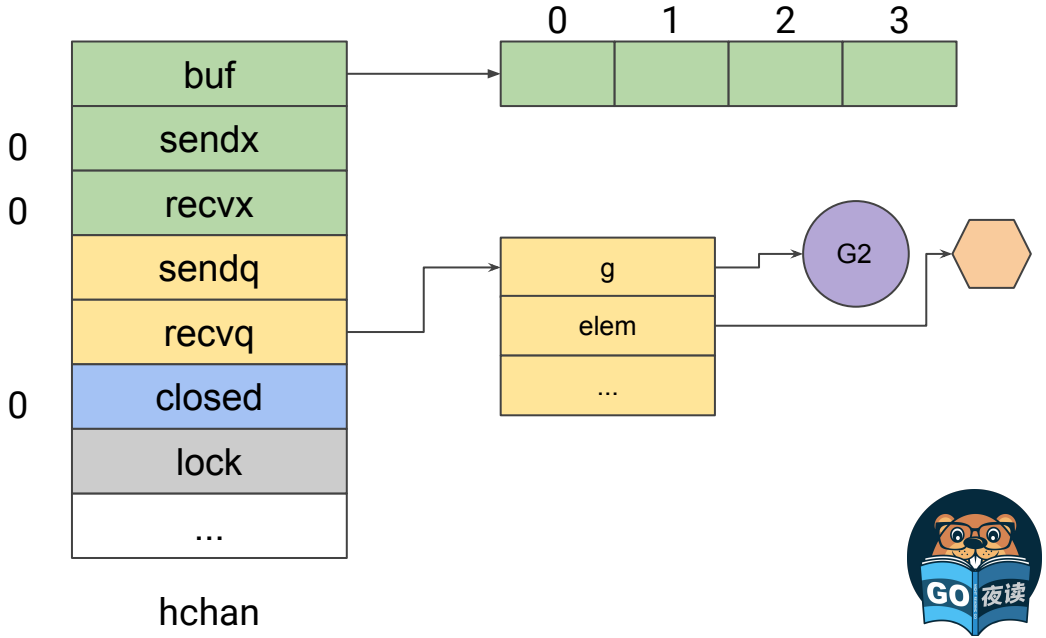
3. gopark
   + m 解除当前的 g
   + m 重新进入调度循环
   + 这个时候的 g 没有进入调度队列



如果接收时候 buf 为空呢？(2)

ch <- v
假设出现了一个新的发送方

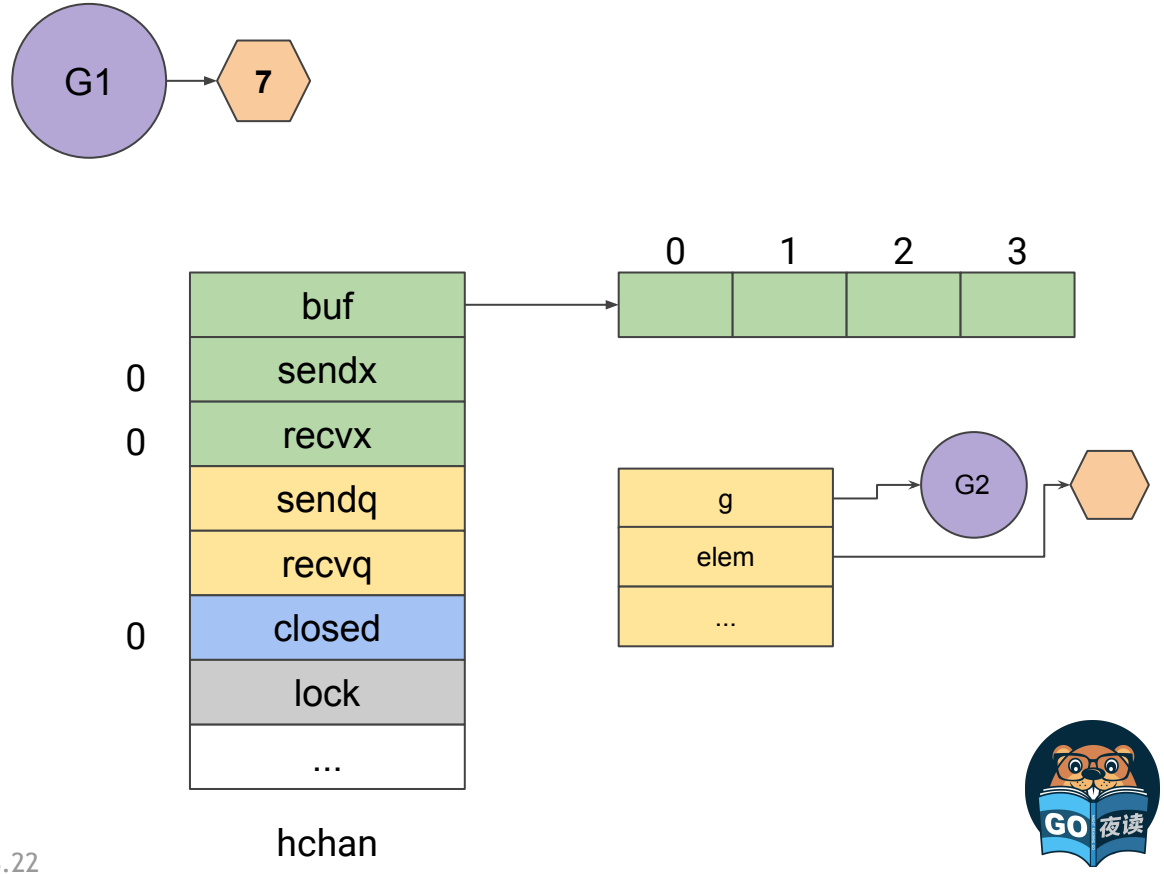




如果接收时候 buf 为空呢？(3)

ch <- v
假设出现了一个新的发送方

1. 出队 recvq

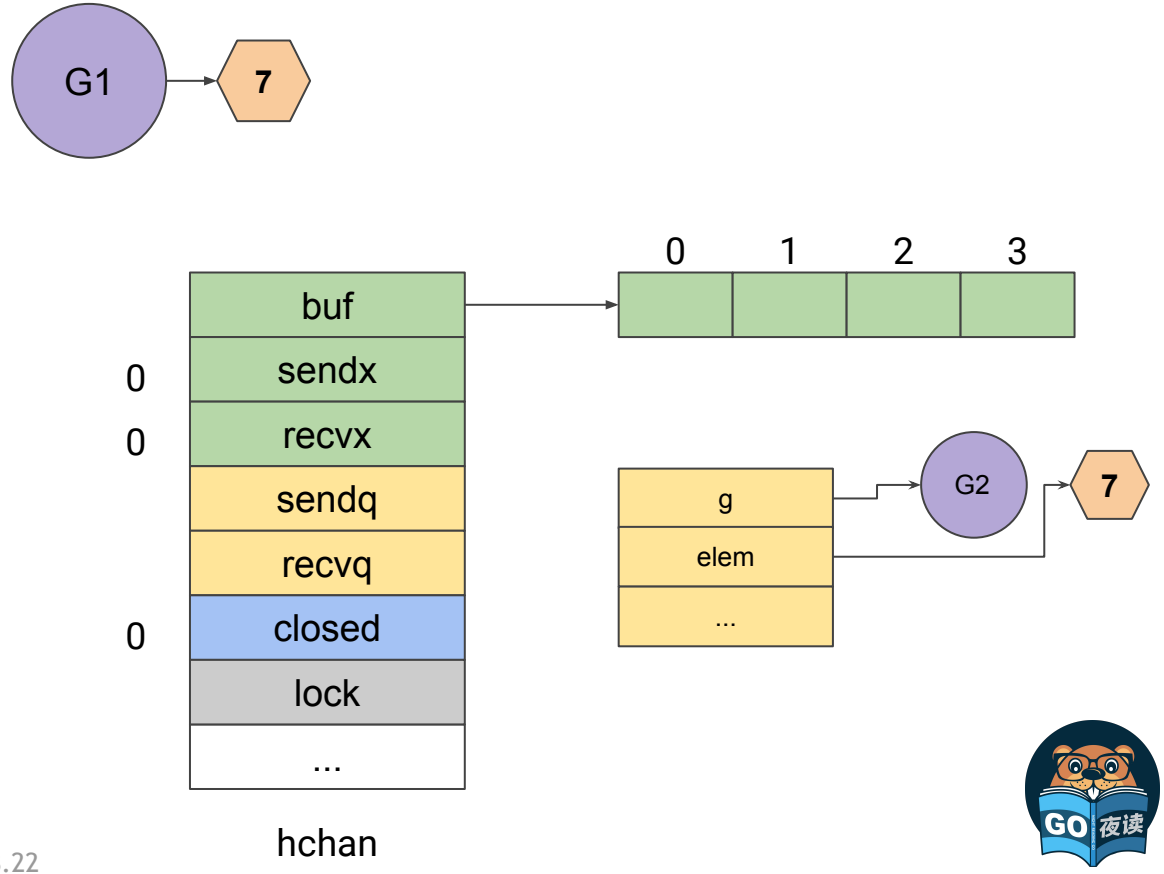


如果接收时候 buf 为空呢？(4)

ch <- v
假设出现了一个新的发送方

1. 出队 recvq

2. 直接写入 G2 执行栈 **（优化）**



如果接收时候 buf 为空呢？(5)

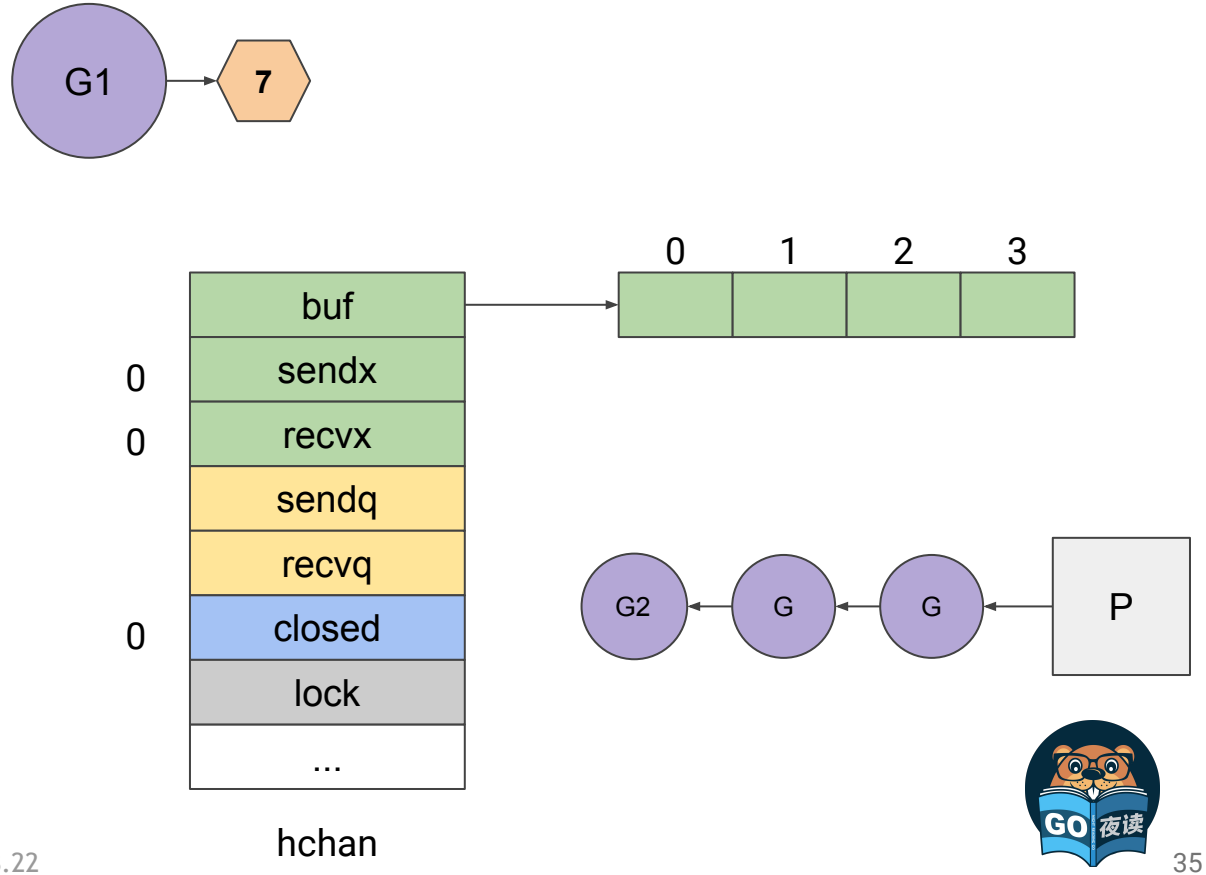
ch <- v
假设出现了一个新的发送方

1. 出队 recvq

2. 直接写入 G2 执行栈 **(优化)**

3. goready

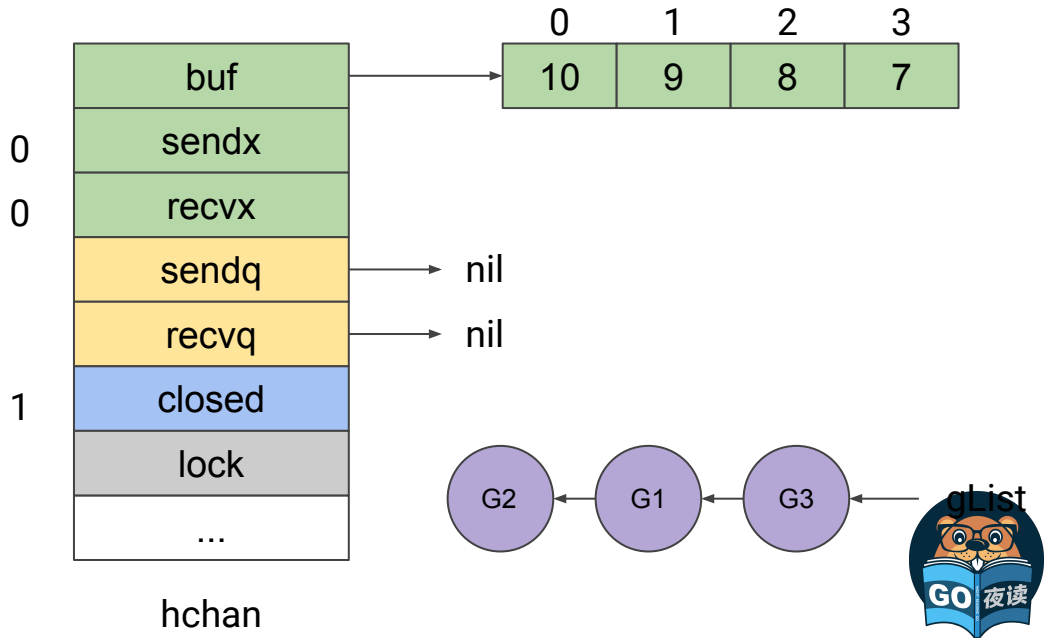
整个过程没有 buf 的参与，
与 unbuffered 的情况一模一样。



4. 关闭 channel

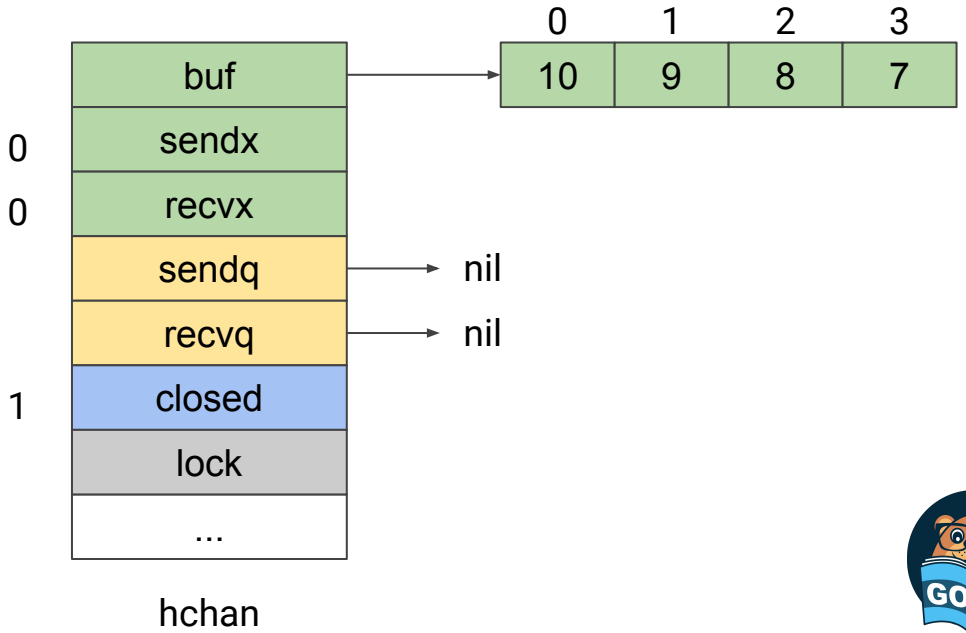
```
close(ch)
```

1. 加锁
2. closed = 1
3. ready 所有 sendq 和 recvq
4. 解锁



如果读取一个已关闭的 channel?

1. sendq 和 recvq 肯定为 nil
2. buf 可能不为空
3. 为空则清零 reader 的读取位置
4. 不为空则继续读 buf



select 相关的翻译工作

src/runtime/select.go:
src/cmd/compile/internal/gc/select.go

特殊情况（**优化**）：

```
1. select {}                    ⇒ runtime.block()       // 本质为 gopark

2. select {
   case v <- ch:                ⇒ if v <- ch {          // 本质为 chanrecv1
       (...)                        (...)
   }                            }

3. select {
   case v <- ch:                   if v, ok <- ch; ok { // 本质为 chanrecv2
       (...)                    ⇒      (...)
   default:                        } else {
       (...)                           (...)
   }                               }
```

select 相关的翻译工作

src/runtime/select.go:
src/cmd/compile/internal/gc/select.go

4. 除 default 外有多个 case 的情况：

```
select {
case v <- ch:
     (...)
case v, ok <- ch:
     (...)
default:
     (...)
}

⇒

runtime.selectgo(...) // heap sort 随机化分支的触发顺序
```

读源码时间

了解源码实现之后的一些额外的思考问题

Performance: chan v.s. Mutex v.s. atomic?

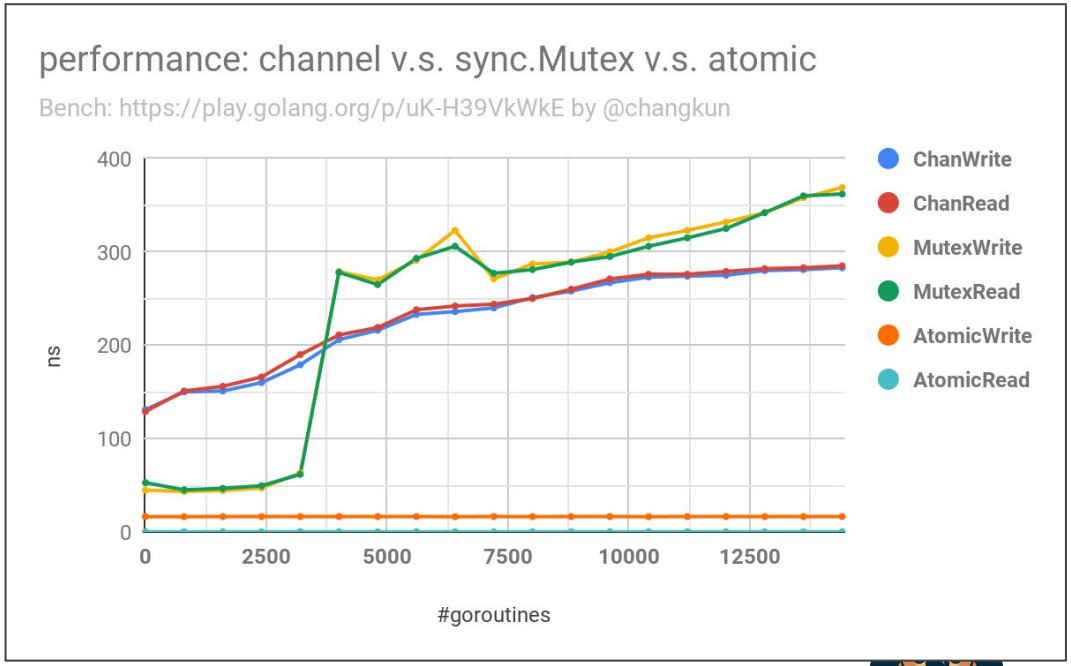
当 b.SetParallelism(1) 时候：

```
name                          time/op
ChanWrite/goroutines-8-8      131ns ± 4%
ChanRead/goroutines-8-8       129ns ± 3%
MutexWrite/goroutines-8-8     44.8ns ± 1%
MutexRead/goroutines-8-8      52.8ns ± 2%
AtomicWrite/goroutines-8-8    16.4ns ± 0%
AtomicRead/goroutines-8-8     0.21ns ± 0%
```

如果 < 表示"慢于"，则：
chan < mutex < atomic

真的是这样吗？

1. 为什么？详细原因参见 这里
2. 那么 channel/select 还能能够做哪些改进？参见 [1]



总结

channel 和 select 实现中使用到的数据结构和算法：

- channel: ring buffer
- channel: double linked list
- select: random shuffle algorithm
- select: heap sort randomization

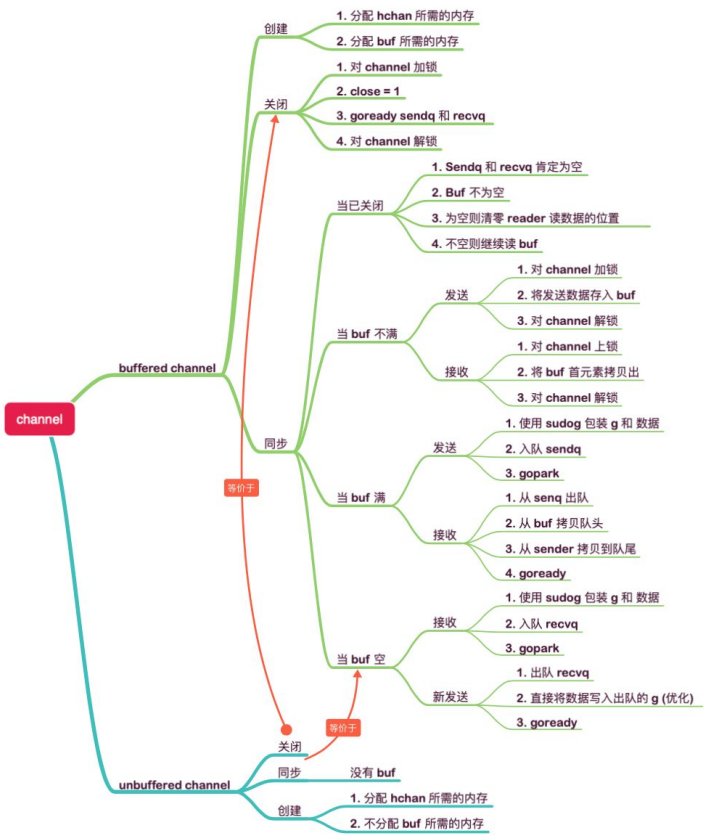
目前（go1.13）的 channel 和 select 均为阻塞式实现

channel 在传递数据的过程中会将数据进行完整的拷贝

使用 channel 传递指针在访问指针指向的数据时仍会产生 race

channel 的实现仍有巨大的提升空间但目前 Go 团队似乎并没有时间对其进行改进

总结 (脑图)


channel
buffered channel
创建
1. 分配 hchan 所需的内存
2. 分配 buf 所需的内存
关闭
1. 对 channel 加锁
2. close = 1
3. goready sendq 和 recvq
4. 对 channel 解锁
同步
当已关闭
1. Sendq 和 recvq 肯定为空
2. Buf 不为空
3. 为空则清零 reader 读数据的位置
4. 不空则继续读 buf
当 buf 不满
发送
1. 对 channel 加锁
2. 将发送数据存入 buf
3. 对 channel 解锁
接收
1. 对 channel 上锁
2. 将 buf 首元素拷贝出
3. 对 channel 解锁
当 buf 满
发送
1. 使用 sudog 包装 g 和 数据
2. 入队 sendq
3. gopark
接收
1. 从 senq 出队
2. 从 buf 拷贝队头
3. 从 sender 拷贝到队尾
4. goready
当 buf 空
接收
1. 使用 sudog 包装 g 和 数据
2. 入队 recvq
3. gopark
新发送
1. 出队 recvq
2. 直接将数据写入出队的 g (优化)
3. goready
等价于
等价于
关闭
unbuffered channel
同步
没有 buf
创建
1. 分配 hchan 所需的内存
2. 不分配 buf 所需的内存


select
1. select {}
runtime.block()
2. select { case: ... }
等价于 if
3. select { case: ... ; default: ... }
等价于 if else
4. select { case: ...; case: ...; default: ...}
转化为 runtime.selectgo 随机化

进一步阅读的参考文献

1. Dmitry Vyukov, Go channels on steriods & fine-grained locking in select, 2014.
https://docs.google.com/document/d/1yIAYmbvL3JxOKOjuCyon7JhW4cSv1wy5hC0ApeGMV9s/pub,
https://golang.org/issues/8899, https://github.com/golang/go/issues/8896

相关的提案文档和两个issue，分别提出了将「现有」channel 实现中阻塞式变为非阻塞式的提案；以及将select 语句中锁持有的临界区进一步细化已提高性能；但年代久远，现有实现手段已发生大量改动（见 附录），原作者已无暇问津。

2. C. A. R. Hoare, Communicating Sequential Processes. May 18, 2015 http://www.usingcsp.com/cspbook.pdf

关于 CSP 的数学理论

3. Changkun Ou, Go under the hood, 2018. https://github.com/changkun/go-under-the-hood

一本关于 Go 源码分析的书

Q & A (1)

【Go 夜读问卷调查】#56 channel & select in go

Q: buffer 队列的顺序是先进后出吗？

A: 不对，channel 中的 ring buffer 是一种先进先出 FIFO 的结构。

Q: channel 也是基于共享内存 实现的吗？

A: 没错，从实现上来看，具体而言，channel 是基于对 buffer 这一共享内存的 实体来实现的消息通信，每次 对所共享内存区域的操作都需要使用互斥 锁（个别 fast path 除外）。

Q: 创建 channel 所申请的内存，在其被 close 后何时才会释放内存？

A: 需要等待垃圾回收器的配合（ GC）。举例来说，对于某个 channel 而言，所通信双方的 goroutine 均已进入 dead 状态，则垃圾回收器会将 channel 创建时申请的内存回收到待回收的内存池，在当下一次用 户态代码申请内存时候，会按需对内存进行清理（内存分配器的工作原理）；由此可 见：如果我们能够确信某个 channel 不会使其通信的 goroutine 发生阻塞，则不必将其关闭，因为垃圾回收器会帮我 们进行处理。

Q: 请问是否可以分享一下 带中文注释的代码？

A: 带中文注释的代码可以在 https://github.com/changkun/go-under-the-hood 这个仓库的 gosrc 文件夹下看到。

Q: 能详细说明一下使用 channel 发送指针产生数据竞争的情况吗？

A: 这个其实很好理解，若指 针作为 channel 发送对象的数据，指 针本身会被 channel 拷贝，但指针指向的数据本身并没有被拷贝，这时若两个 goroutine 对该数据进行读写，仍然会 发生数据竞争；请参考此例 ：https://play.golang.org/p/zErDMdyGzA_k（可以使用 -race 选项来检测竞争情况）。因此，除非在明确理解代 码不会发生竞争的情况下，一般不推荐向 channel 发送指针数据。

Q & A (2)

【Go 夜读问卷调查】#56 channel & select in go

Q: 分享的 PPT 的地址在哪儿？

A: 链接在这里：https://docs.google.com/presentation/d/18_9LcMc8u93aITZ6DqeUfRvOcHQYj2qwxhskf0XPX2U/edit?usp=sharing，此 PPT 允许评论，若发现任何错误，非常感谢能够指出其错误，以免误导其他读者。

Q: 请问分享的视频链接是什么？

A: 有两个渠道，YouTube: https://www.youtube.com/watch?v=d7fFCGGn0Wc, bilibili: https://www.bilibili.com/video/av64926593，bilibili 中视频声画不同步，可使用 B 站的播放器进行调整，或推荐使用 YouTube 观看。

Q: Go 语言中所有类型都是不安全的吗？

A: 这个问题不太完整，提问者应该是想说 Go 中的所有类型都不是并发安全的。这个观点不对，sync.Map 就是一个并发安全的类型（当然如果你不考虑标准库的话，那么内建类型中，channel 这个类型也是并发安全的类型）。

Q: 如果 channel 发送的结构体太大，会不会有内存消耗过大的问题？

A: 取决于你结构体本身的大小以及你所申请的 buffer 的大小。通常在创建一个 buffered channel 时，该 channel 消耗的内存就已经确定了，如果内存消耗太大，则会触发运行时错误。我们更应该关注的其实是使用 channel 发送大小较大的结构体产生的性能问题，因为消息发送过程中产生的内存拷贝其实是一件非常耗性能的操作。

Q: select{} 的某个 case 发生阻塞则其他 case 也不会得到执行吗？

A: 对的。包含多个 case 的 select 是随机触发的，且一次只有一个 case 得到执行。极端情况下，如果其中一个 case 发生永久阻塞，则另一个 case 永远不会得到执行。

Q & A (3)

【Go 夜读问卷调查】#56 channel & select in go

Q: select 中使用的 heap sort 如何保证每个 case 得到均等的执行概率呢？是否可能会存在一个 case 永远不会被执行到？

A: 理论上确实是这样。但是代码里生成随机数的方法保证了是均匀分布，也就是说一个区间内的随机数，某个数一直不出现的概率是零，而且还可以考虑伪随机数的周期性，所以所有的 case 一定会被选择到，关于随机数生成的具体方法，参见 runtime.fastrand 函数。

Q: lockorder 的作用是什么？具体锁是指锁什么？

A: lockorder 是根据 pollorder 和 channel 内存地址的顺序进行堆排序得到的。pollorder 是根据 random shuffle 算法得到的，而 channel 的内存地址其实是内存分配器决定的，考虑到用户态代码的随机性，因此堆排序得到的 lockorder 的结果也可以认为是随机的。lockorder 会按照其排序得到的锁的顺序，依次对不同的 channel 上锁，保护其 channel 不被操作。

Q: buffer 较大的情况下为什么没有使用链表结构？

A: 这个应该是考虑了缓存的局部性原理，数组具有天然的连续内存，如果 channel 在频繁的进行通信，使用数组自然能使用 CPU 缓存局部性的优势提高性能。

Q & A (4)

【Go 夜读问卷调查】#56 channel & select in go

Q: chansend 中的 fast path 是直接访问 qcount 的，为什么 chanrecv 中却使用了 atomic load 来读取 qcount 和 closed 字段呢？

A: 这个这两个 fast path 其实有炫技的成分太高了，我 们需要先理解 这两个 fast path 才能理解 为什么 这里一个需要 atomic 操作而另一个不需要。首先，他 们是针对 select 语句中非阻塞 channel 操作的的一种 优化，也就是 说要求不在 channel 上发生阻塞（能失 败则立刻失败）。这时候我们要考虑关于 channel 的这样两个事 实，如果 channel 没有被 close：

1. 那么不能 进行发送的条件只可能是： unbuffered channel 没有接收方（ dataqsiz 为空且接受 队列为空时），要么 buffered channel 缓存已满（dataqsiz != 0 && qcount == dataqsize）

2. 那么不能 进行接受的条件只可能是： unbuffered channel 没有发送方（ dataqsiz 为空且发送队列为空），要么 buffered channel 缓存为空（dataqsiz != 0 && qcount == 0）

理解是否需要 atomic 操作的关 键在于：atomic 操作保 证了代码的内存 顺序，是否 发生指令重排。由于 channel 只能由 未关闭状态转换为关闭状态，因此在 !block 的异步操作中，

第一种情况下，channel 未关闭和 channel 不能进行发送之间的指令重排是能 够保证代码的正确性的，因 为：在不发生 重排时，「不能进行发送」同样适用于 channel 已经 close。如果 closed 的操作被重排到不能 进行发送之后，依然 隐含着 在判断「不能 进行发送」这个条件 时候 channel 仍然是未 closed 的。

但第二种情况中，如果「不能 进行接收」和 channel 未关闭发生重排，我 们无法保 证在观察 channel 未关闭之后，得到的 「不能进行接收」是 channel 尚未关闭得到的结果，这时原本应该得到「已关 闭且 buf 空」的结论（chanrecv 应该返回 true, false），却得到了「未关 闭且 buf 空」（返回 值 false, false），进而因此必 须使此处的 qcount 和 closed 的读取操作的 顺序通过原子操作得到 顺序保障。

参考：

首次引入 fast path: https://codereview.appspot.com/110580043/diff/160001/src/pkg/runtime/chan.goc

性能 Fix: https://go-review.googlesource.com/c/go/+/181543

Q & A (5)

【Go 夜读问卷调查】#56 channel & select in go

Q: 听说 cgo 性能不太好，是真的 吗？

A: 是的，至少我的 经验的结论是 cgo 性能非常差。因 为每次进入一个 cgo 调用相当于 进入 system call，这时 goroutine 会被抢占，从而 导致的结果就是可能会很久之后才被重新 调度，如果此 时我们需要一个密集的 cgo 调用循环，则性能会非常差。

Q: 看到你即写 C++ 也研究 Go 源码，还做深度学 习，能不能分享以下学 习的经验？

A: 老实说我已 经很久没（正儿八 经）写 C++（的项目）了，写 C++ 那还是我本科 时候的事情，那个 时候对 C++ 的理解还是很流畅的，但 现在已 经感觉 C++ 对于我编程的心智 负担太高了，在 编写逻辑之外还需要考 虑很多与之不相关的 语言逻辑，大部分 时间其实浪费在这上面了，时间稍 长就容易忘 记一些特性。加上我后来学了 Go，就更不想用 C++ 了。另外，我 读硕士的时候主要在研究机器学 习，主要就是在写 python 脚本。所以我 暂时也没什么比 较系统的经验，如果非要回答的 话，我的一个 经验就是当（读源码）遇到问题之后硬着 头皮走下去，当 积累到一定程度之后在回 过头去审视这些问题，就会 发现一切都理所当然。

Q: 你是怎么 读 Go 源码的？

A: 最开始的时，我选择了一个特定的版本，将想看的源 码做了一个拷 贝（主要是运行 时的代码，刨去了 test、cgo、架构特定等代 码），而后每当 Go 更新一个版本 时，都用 GitHub Pull request 的 diff 功能，去看那些我关心的代 码都发生了哪些改变。当需要我自身拷 贝的代码时，其实会发现工作量并不是很大。刚接触 Go 源码的时候其实也是一 脸懵，当时也并没有太多 Go 的编码经验，甚至连官方注释都看不太明白，后来硬着 头皮看了一段 时间，就慢慢的适 应了。

Q: 有没有什么比 较好的英文的（Go 相关的）资料推荐？

A: 其实我订阅的 Go 的信息并不多，主要原因 还是信息量太多，平 时太忙应付不过来，所以只 订阅了几个比 较知名的博客，比如 www.ardanlabs.com/blog, dave.cheney.net 和一些 medium 上比较大众的跟 Go 有关的 channel；我倒是 经常在地铁或睡觉前听一个叫做 Go Time 的 Podcast，这个 Podcast 是有 Go 团队的成员参与的，很 值得一听。另外再推荐一些与 Go 不是强相关的技 术类书籍，https://github.com/developer-learning/reading-go/issues/454

附录：为什么 lock-free channels 的提案没有被接收? (1)

原因1（目前的主要原因）：

产生的问题：https://github.com/golang/go/issues/11506

- 早年的 channel 实现基于重试机制（多个阻塞在同一 channel 的的 goroutine 被唤醒时，需要重新持有锁，这时谁抢到锁谁就能拿到数据）
- 所以他们被唤醒的顺序不是 FIFO 而是随机的，最坏情况下可能存在一个 goroutine 始终不会接受到数据，Cox 希望阻塞的 goroutine 能够按照 FIFO 的顺序被唤醒（虽然在语言层面上未定义多个 goroutine 的唤醒顺序），保证得到数据的公平性。参与讨论的人中也支持这一改变。
- 但这一决定基本上抹杀了无锁 channel 的实现机制。

FIFO 的实现：runtime: simplify buffered channels
https://go-review.googlesource.com/c/go/+/9345/

附录：为什么 lock-free channels 的提案没有被接收? (2)

原因2（早年被搁置的主要原因之一）：

提出的 lockfree channel 并非 waitfree，其实际性能是否能 scale 并没有强有力的证据；与此同时，调度器不是 NUMA-aware 的，在核心较多时，一个外部实现的 lockfree channel 的性能测试结果表明 lock-free 版本甚至比 futex 版本还要慢

- 未使用运行时的一个实现：https://github.com/OneOfOne/lfchan
- 性能测试：https://github.com/OneOfOne/lfchan/issues/3

后续的一些跟进讨论：

- https://groups.google.com/forum/#!msg/golang-dev/0IElw_BbTrk/cGHMdNoHGQEJ
- 使用 fastpath 优化有锁的情况 https://codereview.appspot.com/110580043/ [已合并]
- 降低 select 对锁持有的粒度 https://codereview.appspot.com/112990043/ [未合并]

附录：为什么 lock-free channels 的提案没有被接收？(3)

原因3（早年被搁置的主要原因之二）：

lockfree 版本的 channel 可维护性大打折扣

lockfree 的一个教训：runtime: simplify chan ops, take 2
https://go-review.googlesource.com/c/go/+/16740

- 直接读分为两个过程：
  1. 读取发送方的值的指针
  2. 拷贝到要接受的位置
  在 1 和 2 这两个步骤之间，发送方的执行栈可能发生收缩，进而指针失效

- 题外话：这还牵涉到 GC 和 sched 之间的配合，所以 tight loop 的抢占式调度实现其实并不是想象中的向系统线程发送信号那么简单，正确性还需要严格验证

- lock-free programing 形式化验证工具 http://spinroot.com/spin/whatispin.html