

channel

基本实现

环形队列元数据	qcount int	循环队列中现存元素数量
	datasiz uint	循环队列大小, 或者说容量
	buf unsafe.Pointer	指向循环队列
	elemsize uint16	channel 中元素大小
环形队列相关索引	sendx uint	send 在 buf 中的索引
	recvx uint	recv 在 buf 中的索引
等待队列 (FIFO)	recvq waitq	接受者等待队列
	sendq waitq	发送者等待队列
	lock mutex	互斥锁, 保护 hchan 中字段的并发访问

channel 本质上就是一个环形队列, 使用环形队列而不是双向链表实现的原因在于, channel 的大小是固定的, 初始化时即确定大小, 环形队列无需频繁的申请和释放队列节点内存

另外需要注意的是 `recvq` 和 `sendq` 这两个等待队列是严格 FIFO 的, 当多个 Goroutine 等待同一个 channel 时, 会按先后顺序进行排队

我第一眼看到 channel 的时候就觉得它是一个阻塞队列, 认为和使用 `pthread_cond_wait()`/`pthread_cond_signal()` 所实现的阻塞队列没什么太大的区别。从 `hchan` 的实现来看, 原理也确实差不多一样, 不过 `hchan` 的等待队列是在用户空间实现的, 并且 channel 支持容量为 0 的 unbuffered channel

channel 的类型通常可以从两个维度上进行划分, 从功能上可分为只能接收、只能发送和既可以接收也可以发送这三种类型; 从容量上又可以分为 buffered channel 和 unbuffered channel

unbuffered channel 只有在读、写都准备好的时候才不会发生阻塞。有任意一方未准备好时, 对 unbuffered channel 进行任何操作都会发生阻塞

创建 通常会使用 `make()` 方法初始化一个 channel, 可以选择性的指定容量

- `ch := make(chan int)` — unbuffered channel 通常作为信号通知和任务编排使用
- `ch := make(chan int, 10)` — buffered channel 一般作为数据交流和数据传递, 典型应用就是多生产者-多消费者

基本操作

简单接收

```
x := <-ch // 把接收的一条数据赋值给变量 x
y, ok := <-ch // ok 是一个 bool 值, 作用在下方详述
foo(<-ch) // 把接收的一个数据作为参数传给函数
<-ch // 丢弃接收的一条数据
```

循环接收

```
for v := range ch {
    fmt.Println(v)
}
```

当我们 close 了一个 buffered channel, 如果其队列中仍有数据的话, 接收者可以将数据全部取出, 并且此时 `ok` 的值为 `true`。当数据全部从队列中取出以后再向 channel 接收数据将返回零值, 并且, 此时的 `ok` 值转变为 `false`

往 channel 发送一条数据, 使用 "`ch<-`"
从 channel 取出一条数据, 使用 "`<-ch`"

发送消息

`ch<- 1024` // 将 1024 发送给 ch

如果说我们的 channel 只是作为 Goroutine 之间消息通知的话, 可以使用空结构体来作为“信件”, 因为空结构体并不占用内存

```
ch := make(chan struct{})
// goroutine 1
ch<- struct{}{}
// goroutine 2
<- ch
```

关闭

channel 如果不再使用了, 可使用 `Close(ch)` 进行关闭, 这并不是必需的。channel 和文件描述符还是不一样的, GC 可根据实际情况进行回收。

最后, channel 的零值为 `nil`, 是一种很特殊的 channel, 对值为 `nil` 的 channel 不管是进行接收还是发送操作都会永久阻塞, 救不回来的那种。并且, 对 `nil` channel 执行 `Close()` 会导致 panic

应用

任务编排

并发编程面试中有一道非常经典的题目: 交替打印 1、2、3、4 — 对于 C、C++ 或者是 Python 来讲, 可以使用条件变量来完成

Golang 的话就可以使用 unbuffered chan 实现了

```
type Token struct{}
// 第三种方式, 其实就是把第一种方式抽象了出来
func AlternatelyPrint3(total int) {
    // 初始化一堆 channel, 用于多个 Goroutine 间通信
    channels := make([]chan Token, total)
    for i := 0; i < total; i++ {
        channels[i] = make(chan Token)
    }
    for i := 0; i < total; i++ {
        go func(index int, current chan Token, nextChan chan Token) {
            for {
                <-current
                fmt.Printf("Goroutine %d \n", index)
                time.Sleep(time.Second)
                nextChan <- Token{}
            }
        }(i+1, channels[i], channels[(i+1)%total])
    }
    channels[0] <- Token{}
    select {}
}
```

实现互斥锁

使用 channel 也可以实现互斥锁, 只需要一个容量为 1 的 buffered channel, 哪个 Goroutine 获得了 channel 中的元素, 就代表获取到了锁。解锁时再把元素放回至 channel 中

但是, 尽量不要使用 channel 在生产环境中实现互斥锁, 如果需要互斥锁, 使用 `Mutex` 或者是 `RWMutex`。channel 实现的互斥锁就是拿来写 demo 的

使用 channel 的易错点

panic

- 1 close 一个值为 `nil` 的 channel — 我们只需要确保每次都使用 `make()` 方法初始化即可
 - 2 向已经 close 的 channel 发送数据 — 实际上, 如果接收方调用了 `Close()` 方法关闭了 channel, 发送方是没有办法来判断它到底是不是已经被关闭了。那么此时发送方再发送数据的话, 就会引发 panic
 - 3 close 已经 close 了的 chan — 尽管我们可以使用 `unsafe.Pointer` 去获取 `hchan` 中的 `closed` 字段, 从而判断 chan 是否被关闭, 但是这和 runtime 耦合, 是一个不稳定因素
- 因此, 官方给出的建议就是只在 sender 那一端执行 close, 然后让 receiver 去判断 channel 是否被关闭。https://pkg.go.dev/builtin#close

Goroutine 泄漏

```
func TestGoroutineLeak(t *testing.T) {
    ch := make(chan struct{})
    go func() {
        // 模拟耗时任务
        time.Sleep(5 * time.Second)
        // 任务执行完后向外发出通知
        ch<- struct{}{}
    }()
    select {
    case <- ch:
        fmt.Println(a...: "Task Done")
    case <-time.After(2 * time.Second):
        fmt.Println(a...: "Time Limit Exceeded")
    }
}
```

这段代码的目的很简单, 给一个任务加上一个超时时间

这段代码的问题就在于, 使用了一个 unbuffered channel, unbuffered channel 只有在 receiver 和 sender 都准备好的时候才不会发生阻塞。当任务执行时间超过 2s 时, 定时器到期后直接退出 `select {}` 多路复用, 我们就再也不可能接收到 ch 中的数据了。也就是说, 执行耗时任务的那个 goroutine 永远都不会退出

久而久之, 就会有非常多的 goroutine 堆积在内存无法得到释放, 最终可能会造成内存泄漏。一个解决办法就是将 ch 的容量设置为 1

这里留一个有趣的问题, 当我们把 ch 的容量设置为 1 之后, 再运行上面的代码, ch 会被 GC 回收吗?

死锁

常常出现在 unbuffered channel 中