golang channel 使用总结

文档来自: https://zhuanlan.zhihu.com/p/408598288 公众号: Golang发烧友

本文介绍了使用 golang channel 的诸多特性和技巧，已经熟悉了 go 语言特性的小伙伴也可以看看，很有启发。

不同于传统的多线程并发模型使用共享内存来实现线程间通信的方式，golang 的哲学是通过 channel 进行协程 (goroutine) 之间的通信来实现数据共享：

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

这种方式的优点是通过提供原子的通信原语，避免了竞态情形 (race condition) 下复杂的锁机制。channel 可以看成一个 FIFO 队列，对 FIFO 队列的读写都是原子的操作，不需要加锁。对 channel 的操作行为结果总结如下：

操作	nil channel	closed channel	not-closed non-nil channel
close	panic	panic	成功 close
写 ch <-	一直阻塞	panic	阻塞或成功写入数据
读 <- ch	一直阻塞	读取对应类型零值	阻塞或成功读取数据

读取一个已关闭的 channel 时，总是能读取到对应类型的零值，为了和读取非空未关闭 channel 的行为区别，可以使用两个接收值：

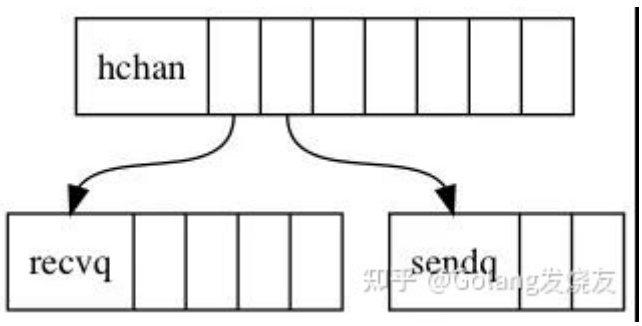
```
// ok is false when ch is closed
v, ok := <-ch
```

golang 中大部分类型都是值类型（只有 slice / channel / map 是引用类型），读/写类型是值类型的 channel 时，如果元素 size 比较大时，应该使用指针代替，避免频繁的内存拷贝开销。

内部实现

如图所示，在 channel 的内部实现中（具体定义在 `$GOROOT/src/runtime/chan.go` 里），维护了 3 个队列：

- 读等待协程队列 recvq，维护了阻塞在读此 channel 的协程列表
- 写等待协程队列 sendq，维护了阻塞在写此 channel 的协程列表
- 缓冲数据队列 buf，用环形队列实现，不带缓冲的 channel 此队列 size 则为 0



当协程尝试从未关闭的 channel 中读取数据时，内部的操作如下：

1. 当 buf 非空时，此时 recvq 必为空，buf 弹出一个元素给读协程，读协程获得数据后继续执行，此时若 sendq 非空，则从 sendq 中弹出一个写协程转入 running 状态，待写数据入队列 buf ，此时读取操作 <- ch 未阻塞；
2. 当 buf 为空但 sendq 非空时（不带缓冲的 channel），则从 sendq 中弹出一个写协程转入 running 状态，待写数据直接传递给读协程，读协程继续执行，此时读取操作

<- ch 未阻塞；

3. 当 buf 为空并且 sendq 也为空时，读协程入队列 recvq 并转入 blocking 状态，当后续有其他协程往 channel 写数据时，读协程才会重新转入 running 状态，此时读取操作

<- ch 阻塞。

类似的，当协程尝试往未关闭的 channel 中写入数据时，内部的操作如下：

1. 当队列 recvq 非空时，此时队列 buf 必为空，从 recvq 弹出一个读协程接收待写数据，此读协程此时结束阻塞并转入 running 状态，写协程继续执行，此时写入操作

ch <- 未阻塞；

2. 当队列 recvq 为空但 buf 未满时，此时 sendq 必为空，写协程的待写数据入 buf 然后继续执行，此时写入操作

ch <- 未阻塞；

3. 当队列 recvq 为空并且 buf 为满时，此时写协程入队列 sendq 并转入 blokcing 状态，当后续有其他协程从 channel 中读数据时，写协程才会重新转入 running 状态，此时写入操作

ch <- 阻塞。

当关闭 non-nil channel 时，内部的操作如下：

1. 当队列 recvq 非空时，此时 buf 必为空，recvq 中的所有协程都将收到对应类型的零值然后结束阻塞状态；
2. 当队列 sendq 非空时，此时 buf 必为满，sendq 中的所有协程都会产生 panic ，在 buf 中数据仍然会保留直到被其他协程读取。

使用场景

除了常规的用来在协程之间传递数据外，本节列出了一些特殊的使用 channel 的场景。

futures / promises

golang 虽然没有直接提供 futrue / promise 模型的操作原语，但通过 goroutine 和 channel 可以实现类似的功能：

```
1  package main
2
3  import (
4      "io/ioutil"
5      "log"
6      "net/http"
7  )
8
9  // RequestFuture, http request promise.
10 func RequestFuture(url string) <-chan []byte {
11     c := make(chan []byte, 1)
```

```
    go func() {
        var body []byte
        defer func() {
            c <- body
        }()

        res, err := http.Get(url)
        if err != nil {
            return
        }
        defer res.Body.Close()

        body, _ = ioutil.ReadAll(res.Body)
    }()

    return c
}

func main() {
    future := RequestFuture("
https://api.github.com/users/octocat/orgs
")
    body := <-future
    log.Printf("reponse length: %d", len(body))
}
```

条件变量 (condition variable)

类型于 POSIX 接口中线程通知其他线程某个事件发生的条件变量，channel 的特性也可以用来当成协程之间同步的条件变量。因为 channel 只是用来通知，所以 channel 中具体的数据类型和值并不重要，这种场景一般用 `strct {}` 作为 channel 的类型。

一对一通知

类似 `pthread_cond_signal()` 的功能，用来在一个协程中通知另个某一个协程事件发生：

```
package main

import (
    "fmt"
```

```
    "time"
)

func main() {
    ch := make(chan struct{})
    nums := make([]int, 100)

    go func() {
        time.Sleep(time.Second)
        for i := 0; i < len(nums); i++ {
            nums[i] = i
        }
        // send a finish signal
        ch <- struct{}{}
    }()

    // wait for finish signal
    <-ch
    fmt.Println(nums)
}
```

广播通知

类似 `pthread_cond_broadcast()` 的功能。利用从已关闭的 channel 读取数据时总是非阻塞的特性，可以实现在一个协程中向其他多个协程广播某个事件发生的通知：

```
package main

import (
    "fmt"
    "time"
)

func main() {
    N := 10
    exit := make(chan struct{})
    done := make(chan struct{}, N)

    // start N worker goroutines
    for i := 0; i < N; i++ {
```

```
        go func(n int) {
            for {
                select {
                // wait for exit signal
                case <-exit:
                    fmt.Printf("worker goroutine #%d exit\n", n)
                    done <- struct{}{}
                    return
                case <-time.After(time.Second):
                    fmt.Printf("worker goroutine #%d is working...\n", n)
                }
            }
        }(i)
    }

    time.Sleep(3 * time.Second)
    // broadcast exit signal
    close(exit)
    // wait for all worker goroutines exit
    for i := 0; i < N; i++ {
        <-done
    }
    fmt.Println("main goroutine exit")
}
```

信号量

channel 的读/写相当于信号量的 P / V 操作，下面的示例程序中 channel 相当于信号量：

```
package main

import (
    "log"
    "math/rand"
    "time"
)

type Seat int
type Bar chan Seat

```

```go
func (bar Bar) ServeConsumer(customerId int) {
    log.Print("-> consumer#", customerId, " enters the bar")
    seat := <-bar // need a seat to drink
    log.Print("consumer#", customerId, " drinks at seat#", seat)
    time.Sleep(time.Second * time.Duration(2+rand.Intn(6)))
    log.Print("<- consumer#", customerId, " frees seat#", seat)
    bar <- seat // free the seat and leave the bar
}

func main() {
    rand.Seed(time.Now().UnixNano())

    bar24x7 := make(Bar, 10) // the bar has 10 seats
    // Place seats in an bar.
    for seatId := 0; seatId < cap(bar24x7); seatId++ {
        bar24x7 <- Seat(seatId) // none of the sends will block
    }

    // a new consumer try to enter the bar for each second
    for customerId := 0; ; customerId++ {
        time.Sleep(time.Second)
        go bar24x7.ServeConsumer(customerId)
    }
}
```

互斥量

互斥量相当于二元信号里，所以 cap 为 1 的 channel 可以当成互斥量使用：

```go
package main

import "fmt"

func main() {
    mutex := make(chan struct{}, 1) // the capacity must be one

    counter := 0
    increase := func() {
        mutex <- struct{}{} // lock
        counter++
```

```
        <-mutex // unlock
    }

    increase1000 := func(done chan<- struct{}) {
        for i := 0; i < 1000; i++ {
            increase()
        }
        done <- struct{}{}
    }

    done := make(chan struct{})
    go increase1000(done)
    <-done; <-done
    fmt.Println(counter) // 2000
}
```

关闭 channel

关闭不再需要使用的 channel 并不是必须的。跟其他资源比如打开的文件、socket 连接不一样，这类资源使用完后不关闭后会造成句柄泄露，channel 使用完后不关闭也没有关系，channel 没有被任何协程用到后最终会被 GC 回收。关闭 channel 一般是用来通知其他协程某个任务已经完成了。golang 也没有直接提供判断 channel 是否已经关闭的接口，虽然可以用其他不太优雅的方式自己实现一个：

```
func isClosed(ch chan int) bool {
    select {
    case <-ch:
        return true
    default:
    }
    return false
}
```

不过实现一个这样的接口也没什么必要。因为就算通过 `isClosed()` 得到当前 channel 当前还未关闭，如果试图往 channel 里写数据，仍然可能会发生 panic ，因为在调用 `isClosed()` 后，其他协程可能已经把 channel 关闭了。关闭 channel 时应该注意以下准则：

- 不要在读取端关闭 channel ，因为写入端无法知道 channel 是否已经关闭，往已关闭的 channel 写数据会 panic ；
- 有多个写入端时，不要再写入端关闭 channle ，因为其他写入端无法知道 channel 是否已经关闭，关闭已经关闭的 channel 会发生 panic ；
- 如果只有一个写入端，可以在这个写入端放心关闭 channel 。

关闭 channel 粗暴一点的做法是随意关闭，如果产生了 panic 就用 recover 避免进程挂掉。稍好一点的方案是使用标准库的 `sync` 包来做关闭 channel 时的协程同步，不过使用起来也稍微复杂些。下面介绍一种优雅些的做法。

一写多读

这种场景下这个唯一的写入端可以关闭 channel 用来通知读取端所有数据都已经写入完成了。读取端只需要用 `for range` 把 channel 中数据遍历完就可以了，当 channel 关闭时，`for range` 仍然会将 channel 缓冲中的数据全部遍历完然后再退出循环：

```go
package main

import (
    "fmt"
    "sync"
)

func main() {
    wg := &sync.WaitGroup{}
    ch := make(chan int, 100)

    send := func() {
        for i := 0; i < 100; i++ {
            ch <- i
        }
        // signal sending finish
        close(ch)
    }

    recv := func(id int) {
        defer wg.Done()
        for i := range ch {
            fmt.Printf("receiver #%d get %d\n", id, i)
        }
        fmt.Printf("receiver #%d exit\n", id)
    }

    wg.Add(3)
    go recv(0)
    go recv(1)
```

```
    go recv(2)
    send()

    wg.Wait()
}
```

多写一读

这种场景下虽然可以用 `sync.Once` 来解决多个写入端重复关闭 channel 的问题，但更优雅的办法设置一个额外的 channel，由读取端通过关闭来通知写入端任务完成不要再继续再写入数据了：

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    wg := &sync.WaitGroup{}
    ch := make(chan int, 100)
    done := make(chan struct{})

    send := func(id int) {
        defer wg.Done()
        for i := 0; ; i++ {
            select {
            case <-done:
                // get exit signal
                fmt.Printf("sender #%d exit\n", id)
                return
            case ch <- id*1000 + i:
            }
        }
    }

    recv := func() {
        count := 0
        for i := range ch {
            fmt.Printf("receiver get %d\n", i)
```

```
            count++
            if count >= 1000 {
                // signal recving finish
                close(done)
                return
            }
        }
    }

    wg.Add(3)
    go send(0)
    go send(1)
    go send(2)
    recv()

    wg.Wait()
}
```

多写多读

这种场景稍微复杂，和上面的例子一样，也需要设置一个额外 channel 用来通知多个写入端和读取端。另外需要起一个额外的协程来通过关闭这个 channel 来广播通知：

```
package main

import (
    "fmt"
    "sync"
    "time"
)

func main() {
    wg := &sync.WaitGroup{}
    ch := make(chan int, 100)
    done := make(chan struct{})

    send := func(id int) {
        defer wg.Done()
        for i := 0; ; i++ {
            select {
```

```
            case <-done:
                // get exit signal
                fmt.Printf("sender #%d exit\n", id)
                return
            case ch <- id*1000 + i:
            }
        }
    }

    recv := func(id int) {
        defer wg.Done()
        for {
            select {
            case <-done:
                // get exit signal
                fmt.Printf("receiver #%d exit\n", id)
                return
            case i := <-ch:
                fmt.Printf("receiver #%d get %d\n", id, i)
                time.Sleep(time.Millisecond)
            }
        }
    }

    wg.Add(6)
    go send(0)
    go send(1)
    go send(2)
    go recv(0)
    go recv(1)
    go recv(2)

    time.Sleep(time.Second)
    // signal finish
    close(done)
    // wait all sender and receiver exit
    wg.Wait()
}
```

总结

channle 作为 golang 最重要的特性，用起来还是比较爽的。传统的 C 里要实现类型的功能的话，一般需要用到 socket 或者 FIFO 来实现，另外还要考虑数据包的完整性与并发冲突的问题，channel 则屏蔽了这些底层细节，使用者只需要考虑读写就可以了。channel 是引用类型，了解一下 channel 底层的机制对更好的使用 channel 还是很用必要的。虽然操作原语简单，但涉及到阻塞的问题，使用不当可能会造成死锁或者无限制的协程创建最终导致进程挂掉。channel 除在可以用来在协程之间通信外，其阻塞和唤醒协程的特性也可以用作协程之间的同步机制，文中也用示例简单介绍了这种场景下的用法。关闭 channel 并不是必须的，只要没有协程没用引用 channel ，最终会被 GC 清理。所以使用的时候要特别注意，不要让协程阻塞在 channel 上，这种情况很难检测到，而且会造成 channel 和阻塞在 channel 的协程占有的资源无法被 GC 清理最终导致内存泄露。channle 方便 golang 程序使用 CSP 的编程范形，但是 golang 是一种多范形的编程语言，golang 也支持传统的通过共享内存来通信的编程方式。终极的原则是根据场景选择合适的编程范型，不要因为 channel 好用而滥用 CSP 。

> 转自：ExplorerManhttp://cnblogs.com/ExMan/p/11710017.html文章转载：Go开发大全（版权归原作者所有，侵删）

推荐阅读

Golang channel 使用指南mjlzz**golang 系列：channel 全面解析**Linco...发表于后端开发**Go channel 实现原理分析**前言channel一个类型管道，通过它可以在goroutine之间发送和接收消息。它是Golang在语言层面提供的goroutine间的通信方式。 众所周知，Go依赖于称为CSP（Communicating Sequential Processe...孤烟**Ready to Go - channel深入解读**900Mo...发表于Afank...