Go语言之三驾马车

唐郑望

2017/10/27 827 0 腾讯WeTest go语言

WeTest 导读

Go语言的三个核心设计: interface | goroutine | channel

> less is more —— **Wikipedia (https://en.wikipedia.org/wiki/Go_%28programming_language%29)**

从Python到Go
远离舒适区
保持饥饿感

interface

Go是一门面向接口编程的语言，interface的设计自然是重中之重。Go中对于interface设计的巧妙之处就在于空的interface可以被当作“Duck”类型使用，它使得Go这样的静态语言拥有了一定的动态性，却又不损失静态语言在类型安全方面拥有的编译时检查的优势。

source code

从底层实现来看，interface实际上是一个结构体，包含两个成员。其中一个成员指针指向了包含类型信息的区域，可以理解为虚表指针，而另一个则指向具体数据，也就是该interface实际引用的数据。

```
type iface struct {
    tab  *itab
    data unsafe.Pointer
}
```

itab的结构如下：

```
type itab struct {
    inter  *interfacetype
    _type  *_type
    link   *itab
    bad    int32
    inhash int32
    fun    [1]uintptr
}
```

其中 interfacetype 包含了一些关于interface本身的信息，_type表示具体实现类型，在下文eface中会有详细描述，bad 是一个状态变量，fun是一个长度为1的指针数组，在 fun[0] 的地址后面依次保存method对应的函数指针。go runtime 包里面有一个hash表，通过这个hash表可以取得 itab，link跟inhash则是为了保存hash表中对应的位置并设置标识。主要代码如下：

专题

项目开发（28） (/lab/tagId=50)
Android开发（27） (/lab/tag/?tagId=78)
服务器性能测试（25） (/lab/tag/?tagId=97)
APPStore（4） (/lab/tagId=173)
手游测试（2） (/lab/tagId=24)

最新文章

Go语言之三驾马车 (/lab/view/346.html)
Web前端性能优化——如提升静态文件的加载速度 (/lab/view/345.html)
微信高级运营分享（下）如何做运营的 (/lab/view/344.html)
腾讯首款战争策略手游者」的兼容测试之路 (/lab/view/343.html)
微信高级运营分享（上）是什么 (/lab/view/342

```
func additab(m *itab, locked, canfail bool) {
    ...
    h := itabhash(inter, typ)
    m.link = hash[h]
    m.inhash = 1
    // 存到itab的hash表缓存
    atomicstorep(unsafe.Pointer(&hash[h]), unsafe.Pointer(m))
}
```

空接口的实现略有不同。Go中任何对象都可以表示为interface{}，类似于C中的 void*，而且interface{}中存有类型信息。

```
type eface struct {
    _type *_type
    data  unsafe.Pointer
}
```

Type的结构如下：

```
type _type struct {
    size       uintptr // type size
    ptrdata    uintptr // size of memory prefix holding all pointers
    hash       uint32  // hash of type; avoids computation in hash tables
    tflag      tflag   // extra type information flags
    align      uint8   // alignment of variable with this type
    fieldalign uint8   // alignment of struct field with this type
    kind       uint8   // enumeration for C
    alg        *typeAlg  // algorithm table
    // gcdata stores the GC type data for the garbage collector.
    // If the KindGCProg bit is set in kind, gcdata is a GC program.
    // Otherwise it is a ptrmask bitmap. See mbitmap.go for details.
    gcdata     *byte    // garbage collection data
    str        nameOff  // string form
    ptrToThis  typeOff  // type for pointer to this type, may be zero
}
```

> **提示：**关于interface的更多信息，可以参考：https://research.swtch.com/interfaces

i_example

关于interface的应用，下面举个简单的例子，是关于Go与Mysql数据库交互的。

首先在mysql test库中创建一张任务信息表：

```
CREATE TABLE IF NOT EXISTS `test`.`task` (
    `id` INT(11) NOT NULL AUTO_INCREMENT COMMENT 'ID',
    `name` VARCHAR(45) NOT NULL COMMENT '任务名',
    `status` smallint(6) NOT NULL DEFAULT 0 COMMENT '状态码',
    `timeout` INT(11) NOT NULL DEFAULT 0 COMMENT '超时时间',
    PRIMARY KEY (`id`))
    ENGINE = InnoDB
    AUTO_INCREMENT = 1
    DEFAULT CHARACTER SET = utf8
    COMMENT = '任务信息表'
```

数据库交互最基本的四个操作：增删改查， 这里以查询为例：

Go来实现查询这张表里面的所有数据

```
package main

import (
    "database/sql"
    "fmt"
    _ "github.com/go-sql-driver/mysql"
)

func query() {
    db, err := sql.Open("mysql", "user:password@tcp(localhost:3306)/test?
charset=utf8")
    checkErr(err)
    defer db.Close()
    rows, err := db.Query("SELECT * FROM task")
    checkErr(err)
    defer rows.Close()
    var (
        id       int
        name     string
        status   int
        timeout  int
    )
    for rows.Next() {
        err = rows.Scan(&id, &name, &status, &timeout)
        checkErr(err)
        fmt.Println(id)
        fmt.Println(name)
        fmt.Println(status)
        fmt.Println(timeout)
    }
```

其中

```
func checkErr(err error) {
    if err != nil {
        panic(err)
    }
}
```

这段代码可以实现查表这个简单的逻辑，但是有一个小小的问题就是，我们这张表结构比较简单只有4个字段，如果换一张有20+个字段甚至更多的表来查询的话，这段代码就显得太过于低效，这个时候我们便可以引入interface{}来进行优化。

优化后的代码如下：

```
func query() {
    db, err := sql.Open("mysql", "user:password@tcp(localhost:3306)/test?
charset=utf8")
    checkErr(err)
    defer db.Close()
    rows, err := db.Query("SELECT * FROM task")
    checkErr(err)
    defer rows.Close()

    columns, _ := rows.Columns()
    args := make([]interface{}, len(columns))
    values := make([]interface{}, len(columns))
    for i := range values {
        args[i] = &values[i]
    }

    record := make(map[string]string)
    for rows.Next() {
        err = rows.Scan(args...)
        for i, col := range values {
            if col != nil {
                record[columns[i]] = string(col.([]uint8))
            }
        }
        fmt.Println(record)
    }
}
```

由于interface{}可以保存任何类型的数据，所以通过构造args、values两个数组，其中args的每个值指向values相应值的地址，来对数据进行批量的读取及后续操作，值得注意的是Go是一门强类型的语言，而且不同的interface{}是存有不同的类型信息的，在进行赋值等相关操作时需要进行类型转换。

Go对于Mysql事务处理也提供了比较好的支持。一般的操作使用的是db对象的方法，事务则是使用sql.Tx对象。使用db的Begin方法可以创建tx对象。tx对象也有数据库交互的Query,Exec和Prepare方法，与db的操作类似。查询或修改的操作完毕之后，需要调用tx对象的Commit()提交或者Rollback()回滚。

例如，现在需要利用事务对之前创建的user表进行update操作，代码如下

```
func update(timeout int, status int, id int) {
    db, err := sql.Open("mysql", "user:password@tcp(localhost:3306)/test?
charset=utf8")
    checkErr(err)
    defer db.Close()
    tx, err := db.Begin()
    checkErr(err)
    defer func(){
        err = tx.Rollback()
        checkErr(err)
    }()
    stmt, err := tx.Prepare(`UPDATE task SET timeout=?, status=? WHERE id
=?`)
    checkErr(err)
    defer stmt.Close()
    res, err := stmt.Exec(timeout, status, id)
    checkErr(err)
    num, err := res.RowsAffected()
    checkErr(err)
    fmt.Println(num)
    err = tx.Commit();
}
```

> **注意：** “ :=“ 跟“ =“两个操作符不要弄混淆

如果不需要进行事务处理的话，update对应的代码如下

```
func update(timeout int, status int, id int) {
    db, err := sql.Open("mysql", "user:password@tcp(localhost:3306)/test?
charset=utf8")
    checkErr(err)
    defer db.Close()
    stmt, err := db.Prepare(`UPDATE task SET timeout=?, status=? WHERE id
=?`)
    checkErr(err)
    defer stmt.Close()
    res, err := stmt.Exec(timeout, status, id)
    checkErr(err)
    num, err := res.RowsAffected()
    checkErr(err)
    fmt.Println(num)
}
```

可以与上面增加事务操作的代码进行对比，因为操作比较简单所以也就增加了几行代码，以及将db对象换成了tx对象。

> **提示：**关于Go对sql的更多支持，可以参考**官方文档：**https://golang.org/pkg/database/sql/

goroutine

并发：同一时间内处理(dealing with)不同的事情
并行：同一时间内做(doing)不同的事情

Go从语言层面就支持了并行，而goroutine则是Go并行设计的核心。本质上，goroutine就是协程，拥有独立的可以自行管理的调用栈，可以把goroutine理解为轻量级的thread。但是thread是操作系统调度的，抢占式的。goroutine是通过自己的调度器来调度的。

scheduler

Go的调度器实现了G-P-M调度模型，其中有三个重要的结构：M，P，G

M：Machine (OS thread)

P：Context (Go Scheduler)

G：Goroutine

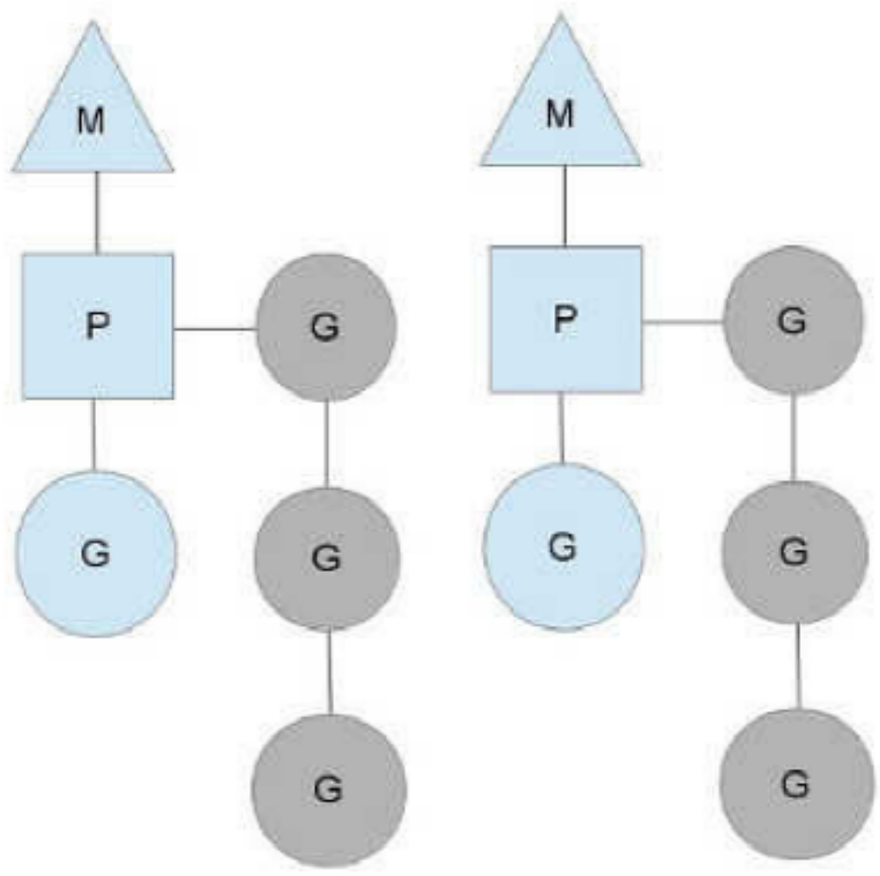
底层的数据结构长这样：

```
type g struct {
    stack       stack   // offset known to runtime/cgo
    stackguard0 uintptr // offset known to liblink
    stackguard1 uintptr // offset known to liblink
    _panic      *_panic // innermost panic - offset known to liblink
    _defer      *_defer // innermost defer
    m           *m      // current m; offset known to arm liblink
    sched       gobuf
    ...
}
```

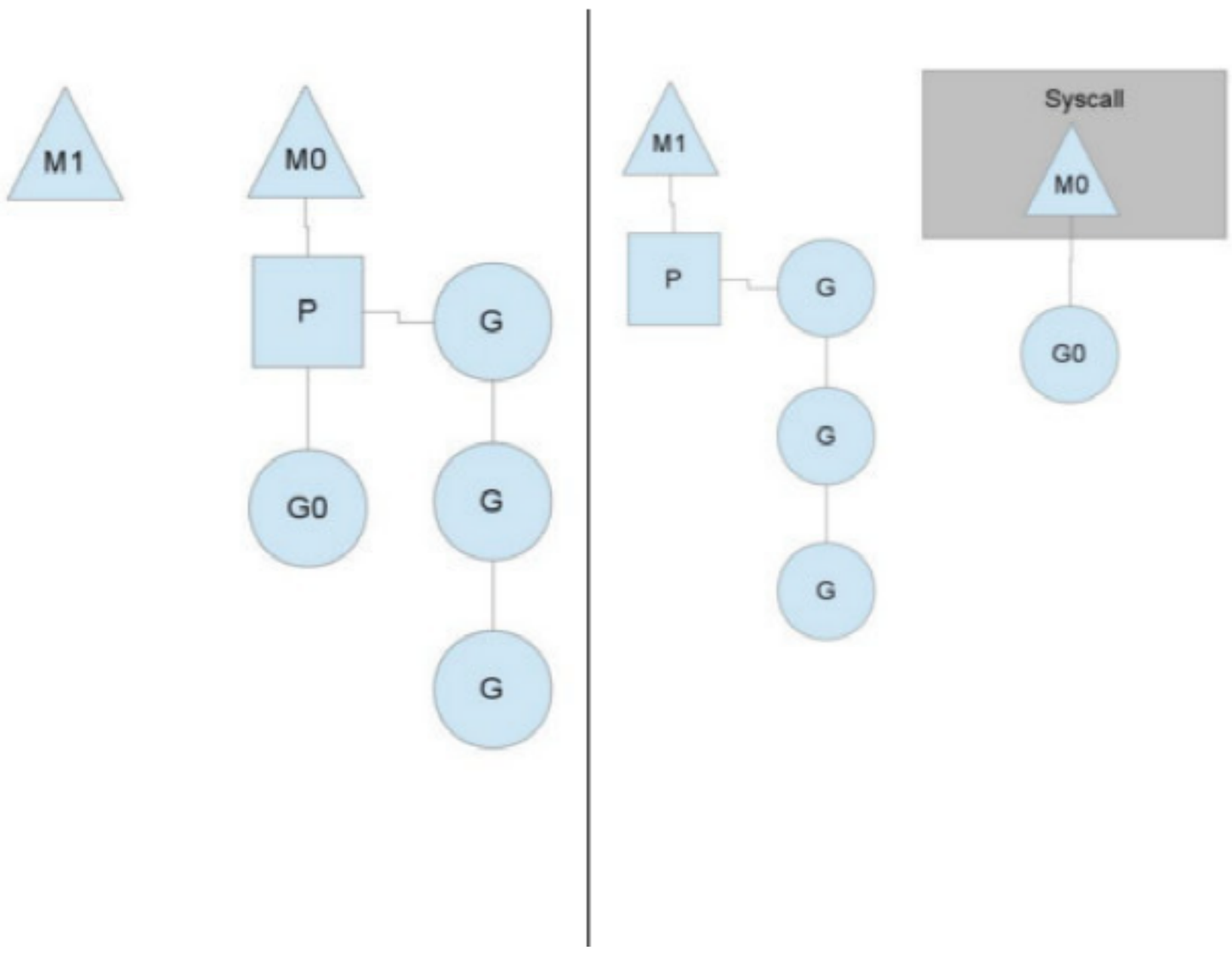
```
type m struct {
  g0          *g     // goroutine with scheduling stack
  morebuf     gobuf  // gobuf arg to morestack
  divmod      uint32 // div/mod denominator for arm - known to liblink
  // Fields not known to debuggers.
  procid      uint64     // for debuggers, but offset not hard-coded
  gsignal     *g         // signal-handling g
  sigmask     sigset     // storage for saved signal mask
  ...
}
```

```
type p struct {
  lock        mutex
  id          int32
  status      uint32 // one of pidle/prunning/...
  link        puintptr
  schedtick   uint32     // incremented on every scheduler call
  syscalltick uint32     // incremented on every system call
  ...
}
```

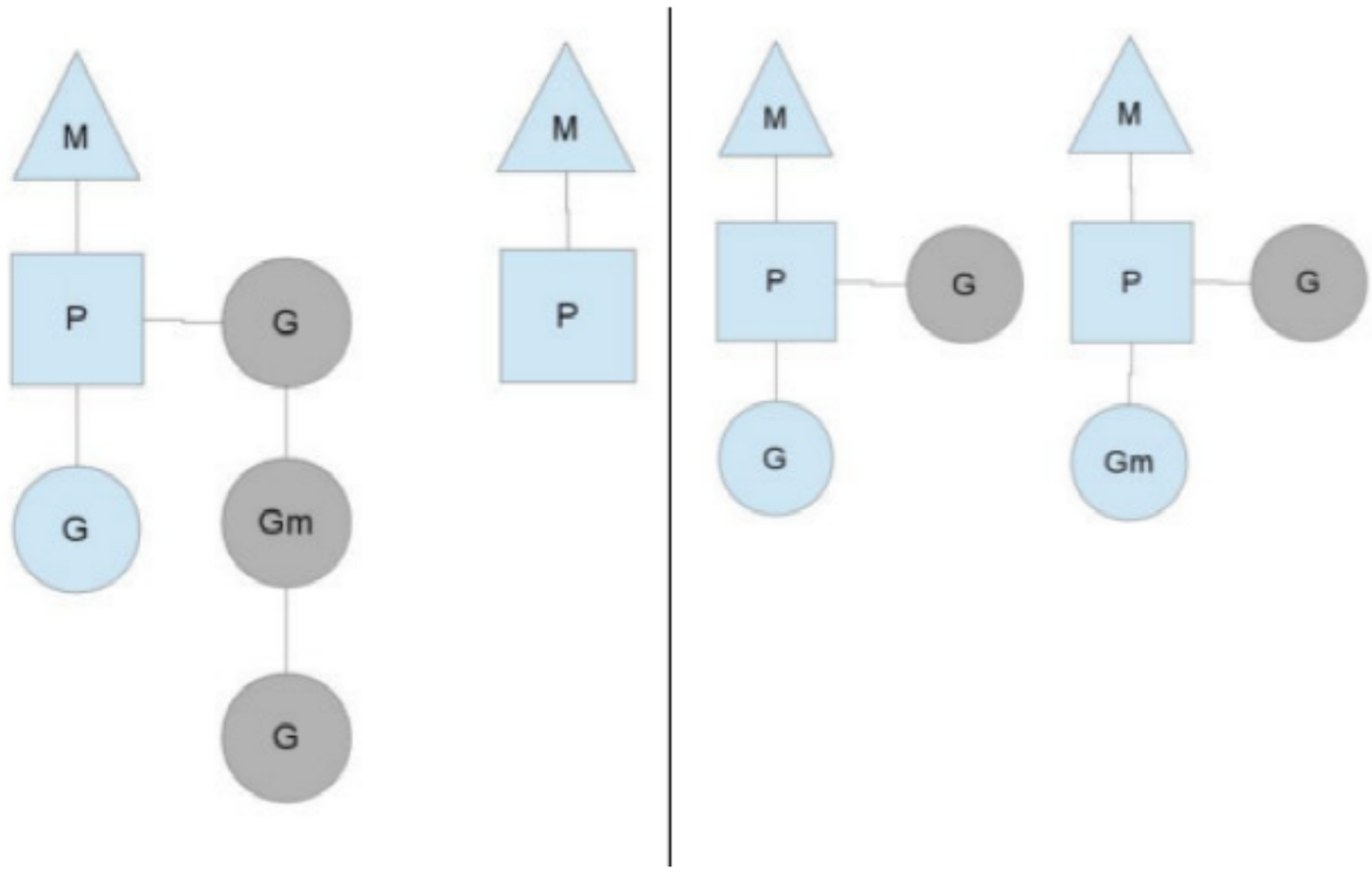
M、P 和 G 之间的交互可以通过下面这几张来自**go runtime scheduler**（http://morsmachine.dk/go-scheduler）的图来展现



上图中看，有2个物理线程M，每一个M都拥有一个上下文P，也都有一个正在运行的goroutine G。图中灰色的那些G并没有运行，而是出于ready的就绪态，正在等待被调度。由P来维护着这个runqueue队列。



图中的M1可能是被新建出来的，也可能是从线程缓存中取出来的。当M0返回时，它必须尝试获取P来运行G，通常情况下，它会尝试从其他的thread那里”steal”一个P过来，失败的话，它就把G放在一个global runqueue里，然后自己会被放入线程缓存里。所有的P会周期性的检查global runqueue，否则global runqueue上的G永远无法执行。



另一种情况是P所分配的任务G很快就执行完了（因为分配不均），这就导致了某些P处于空闲状态而系统却依然在运行态。但如果global runqueue没有任务G了，那么P就不得不从其他的P那里拿一些G来执行。通常情况下，如果P从其他的P那里要偷一个任务的话，一般就‘steal’ runqueue的一半，这就确保了每个thread都能充分的使用。

P如何从其他P维护的队列中”steal”到G呢？这就涉及到work-stealing算法，关于该算法的更多信息可以参考：https://rakyll.org/scheduler/

g_example

举个简单的例子来演示下goroutine是如何运行的

```
package main
import (
    "fmt"
    "time"
)

func say(s string) {
    for i := 0; i < 3; i++ {
        time.Sleep(10 * time.Millisecond)
        fmt.Println(s)
    }
}

func main() {
    go say("world!!!") // new goroutine
    say("hello")    // main goroutine
}
```

这段代码非常简单，两个不同的goroutine异步运行
运行结果如下：

```
world!!!
hello
hello
world!!!
world!!!
hello
```

然后做个小小的改动，只是将main()中的两个函数的位置互换，其余代码变：

```
func main() {
    say("hello")
    go say("world!!!")
}
```

会出现一件有意思的事情：

```
hello
hello
hello
```

原因也很简单，因为main()返回时， 并不会等待其他goroutine(非主goroutine)结束。对上面的例子， 主函数执行完第一个say()后，创建了一个新的goroutine没来得及执行程序就结束了，所以会出现上面的运行结果。

channel

goroutine在相同的地址空间中运行，因此必须同步对共享内存的访问。Go语言提供了一个很好的通信机制channel，来满足goroutine之间数据的通信。channel与Unix shell 中的双向管道有些类似：可以通过它发送或者接收值。

source code

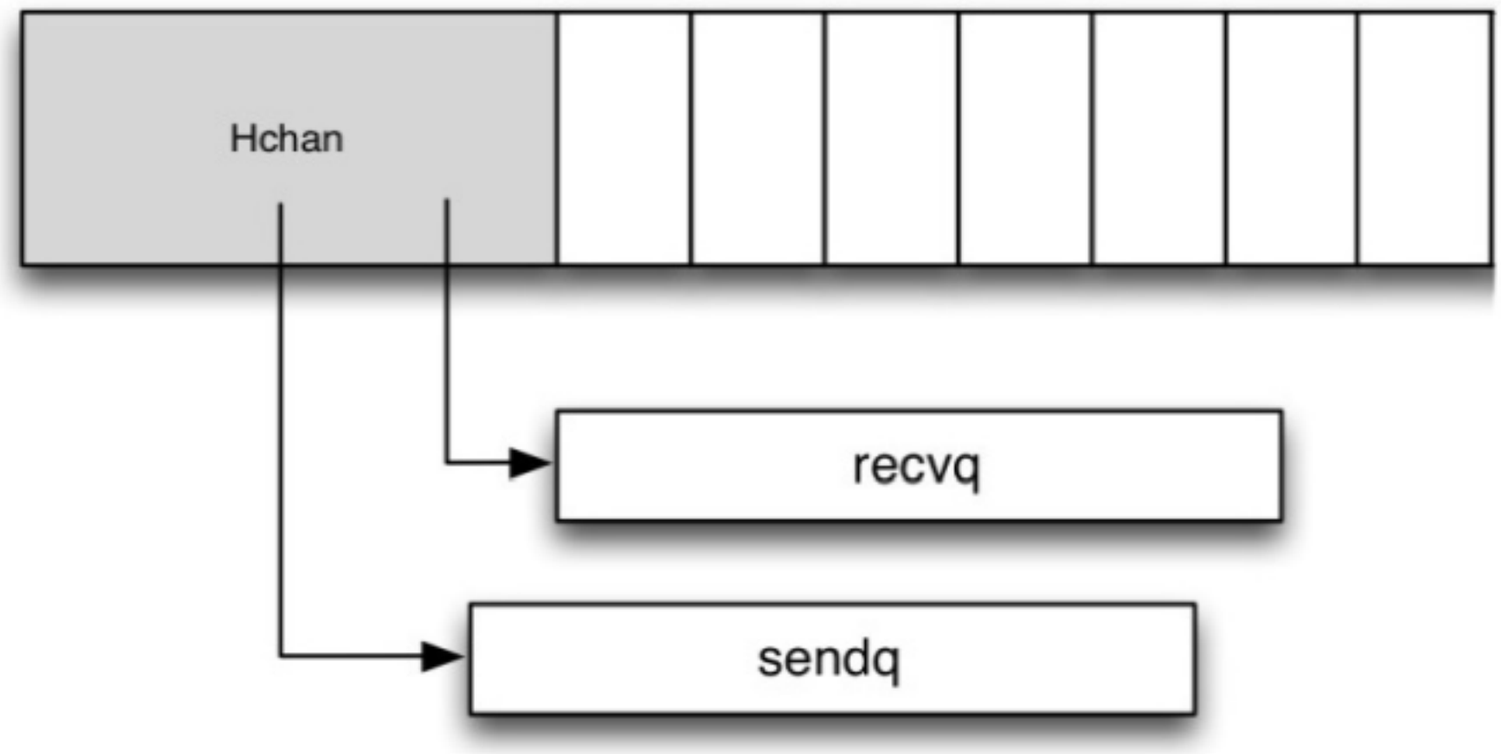
```
type hchan struct {
    qcount   uint           // total data in the queue
    dataqsiz uint           // size of the circular queue
    buf      unsafe.Pointer // points to an array of dataqsiz elements
    elemsize uint16
    closed   uint32
    elemtype *_type // element type
    sendx    uint   // send index
    recvx    uint   // receive index
    recvq    waitq  // list of recv waiters
    sendq    waitq  // list of send waiters

    // lock protects all fields in hchan, as well as several
    // fields in sudogs blocked on this channel.
    //
    // Do not change another G's status while holding this lock
    // (in particular, do not ready a G), as this can deadlock
    // with stack shrinking.
    lock     mutex
}
```

其中waitq的结构如下

```
type waitq struct {
    first *sudog
    last  *sudog
  }
```

可以看到channel其实就是一个队列加一个锁。其中sendx和recvx可以看做生产者跟消费者队列，分别保存的是等待在channel上进行读操作的goroutine和等待在channel上进行写操作的goroutine，如下图所示。



写channel (ch <- x)的具体实现如下(只选取了核心代码)：
具体可以分为三种情况：

— 有goroutine阻塞在channel上，而且chanbuf为空，直接将数据发送给该goroutine上。

— chanbuf有空间可用：将数据放到chanbuf里面。

— chanbuf没有空间可用：阻塞当前goroutine。

```
func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr)
bool {
    if c == nil {
        if !block {
            return false
        }
        gopark(nil, nil, "chan send (nil chan)", traceEvGoStop, 2)
        throw("unreachable")
    }

    if debugChan {
        print("chansend: chan=", c, "\n")
    }

    if raceenabled {
        racereadpc(unsafe.Pointer(c), callerpc, funcPC(chansend))
    }

    if !block && c.closed == 0 && ((c.dataqsiz == 0 && c.recvq.first == n
il) ||
        (c.dataqsiz > 0 && c.qcount == c.dataqsiz)) {
        return false
    }

    var t0 int64
    if blockprofilerate > 0 {
        t0 = cputicks()
    }

    lock(&c.lock)

    if c.closed != 0 {
        unlock(&c.lock)
        panic(plainError("send on closed channel"))
```

```
    }
    // 第一种情况。从 channel 的队列中取出等待的 goroutine，然后调用 send。
    if sg := c.recvq.dequeue(); sg != nil {
        // Found a waiting receiver. We pass the value we want to send
        // directly to the receiver, bypassing the channel buffer (if an
y).
        send(c, sg, ep, func() { unlock(&c.lock) }, 3)
        return true
    }
    //第二种情况。如果缓冲区有可用空间，就将元素追加到缓冲区中.
    if c.qcount < c.dataqsiz {
        // Space is available in the channel buffer. Enqueue the element
to send.
        qp := chanbuf(c, c.sendx)
        if raceenabled {
            raceacquire(qp)
            racerelease(qp)
        }
        typedmemmove(c.elemtype, qp, ep)
        c.sendx++
        if c.sendx == c.dataqsiz {
            c.sendx = 0
        }
        c.qcount++
        unlock(&c.lock)
        return true
    }

    if !block {
        unlock(&c.lock)
        return false
    }

    // Block on the channel. Some receiver will complete our operation fo
r us.
    //第三种情况如下,阻塞当前goroutine。等写入的元素保存在当前goroutine结构里，然后
将goroutine放入sendq队列中并被挂起，等待后续有goroutine来读取元素后才会被唤醒。
    gp := getg()
    mysg := acquireSudog()
    mysg.releasetime = 0
    if t0 != 0 {
        mysg.releasetime = -1
    }
    // No stack splits between assigning elem and enqueuing mysg
    // on gp.waiting where copystack can find it.
    mysg.elem = ep
    mysg.waitlink = nil
    mysg.g = gp
    mysg.selectdone = nil
    mysg.c = c
    gp.waiting = mysg
    gp.param = nil
    c.sendq.enqueue(mysg)
    goparkunlock(&c.lock, "chan send", traceEvGoBlockSend, 3)
```

```
    // someone woke us up.
    if mysg != gp.waiting {
        throw("G waiting list is corrupted")
    }
    gp.waiting = nil
    if gp.param == nil {
        if c.closed == 0 {
            throw("chansend: spurious wakeup")
        }
        panic(plainError("send on closed channel"))
    }
    gp.param = nil
    if mysg.releasetime > 0 {
        blockevent(mysg.releasetime-t0, 2)
    }
    mysg.c = nil
    releaseSudog(mysg)
    return true
  }
}
```

读channel(<-ch）和发送的操作类似，就不帖代码展示了。

c_example

关于goroutine跟channel进行通信的一个简单的例子,逻辑很简单:

```
package main

import "fmt"
import "time"

func worker(id int, jobs <-chan int, results chan<- int) {
    for j := range jobs {
        fmt.Println("worker", id, "processing job", j)
        time.Sleep(time.Second)
        results <- j * 2
    }
 }

func main() {
    jobs    := make(chan int, 10)
    results := make(chan int, 10)
    defer close(jobs)
    defer close(results)

    for w := 1; w <= 3; w++ {
        go worker(w, jobs, results)
    }

    for j := 1; j <= 9; j++ {
            jobs <- j
    }

    for a := 1; a <= 9; a++ {
        <-results
    }
}
```

这里我们定义了两个带缓存的channel jobs 和 results，如果把这两个channel都换成不带缓存的，就会报错，不过可以这样进行处理就可以了：

```
func main() {
    jobs    := make(chan int)
    results := make(chan int)

    defer close(jobs)
    defer close(results)

    for w := 1; w <= 3; w++ {
        go worker(w, jobs, results)
    }

    go func() {
        for j := 1; j <= 9; j++ {
            jobs <- j
        }
    }()

    for a := 1; a <= 9; a++ {
        <-results
    }
}
```

比较常见的channel操作还有select ， 存在多个channel的时候，可以通过select可以监听channel上的数据流动。

```go
package main
import "fmt"

func main() {
    ch1 := make(chan int, 1)
    ch2 := make(chan int, 1)

    select {
    case <-ch1:
        fmt.Println("ch1 pop one element")
    case <-ch2:
        fmt.Println("ch2 pop one element")
     default:
         fmt.Println("all channels are empty")


    }
}
```

因为 ch1 和 ch2 都为空，所以 case1 和 case2 都不会读取成功。 则 select 执行 default 语句。

这篇文章是对这段时间学习Go的一次小结，也算是抛砖引玉，文中如有理解不对或者描述错误的地方，也恳请大家批评指正，关于Go的学习，更希望能与大家多多交流，谢谢！

推酷 (http://www.tuicool.com/)　前端教程 (http://www.softwhy.com/)　互联网的一些事 (http://www.yixieshi.com/)　手游那点事 (http://www.sykong.com/)
火狐社区 (http://mozilla.com.cn/moz-portal.html)　51Testing (http://www.51testing.com/html/index.html)　慕课网 (http://www.imooc.com)　9秒社区 (http://www.9miao.com/forum.p
segmentfault (https://segmentfault.com)　安卓巴士 (http://www.apkbus.com)　W3school (http://www.w3school.com.cn/)
