Tony Bai

一个程序员的心路历程

Golang Channel用法简编

- 九月 29, 2014
- 0 条评论

在进入正式内容前，我这里先顺便转发一则消息，那就是Golang 1.3.2已经正式发布了。国内的golangtc已经镜像了golang.org的安装包下载页面，国内go程序员与爱好者们可以到"Golang中 国"，即golangtc.com去下载go 1.3.2版本。

Go这门语言也许你还不甚了解，甚至是完全不知道，这也有情可原，毕竟Go在TIOBE编程语言排行榜上位列30开外。但近期使用Golang 实现的一杀手级应用 Docker你却不该不知道。docker目前火得是一塌糊涂啊。你去国内外各大技术站点用眼轻瞥一下，如 果没有涉及到“docker”字样新闻的站点建 议你以后就不要再去访问了^_^。Docker是啥、怎么用以及基础实践可以参加国内一位仁兄的经验之作：《 Docker - 从入门到实践》。

据我了解，目前国内试水Go语言开发后台系统的大公司与初创公司日益增多，比如七牛、京东、小米，盛大，金山，东软，搜狗等，在这里我们可以看到一些公司的Go语言应用列表，并且目前这个列表似乎依旧在丰富中。国内Go语言的推广与布道也再稳步推进中，不过目前来看多以Go入 门与基础为主题，Go idioms、tips或Best Practice的Share并不多见，想必国内的先行者、布道师们还在韬光养晦，积攒经验，等到时机来临再厚积薄发。另外国内似乎还没有一个针对Go的 布道平台，比如Golang技术大会之类的的平台。

在国外，虽然Go也刚刚起步，但在Golang share的广度和深度方面显然更进一步。Go的国际会议目前还不多，除了Golang老东家Google在自己的各种大会上留给Golang展示自己的 机会外，由 Gopher Academy 发起的GopherCon 会议也于今年第一次举行，并放出诸多高质量资料，在这里可以下载。欧洲的Go语言大会.dotgo也即将开幕，估计后续这两个大会将撑起Golang技术分享 的旗帜。

言归正传，这里要写的东西并非原创，自己的Go仅仅算是入门级别，工程经验、Best Practice等还谈不上有多少，因此这里主要是针对GopherCon2014上的“舶来品”的学习心得。来自CloudFlare的工程师John Graham-Cumming谈了关于 Channel的实践经验，这里针对其分享的内容，记录一些学习体会和理解，并结合一些外延知识，也可以算是一种学习笔记吧，仅供参考。

一、Golang并发基础理论

Golang在并发设计方面参考了C.A.R Hoare的CSP，即Communicating Sequential Processes并发模型理论。但就像John Graham-Cumming所说的那样，多数Golang程序员或爱好者仅仅停留在“知道”这一层次，理解CSP理论的并不多，毕竟多数程序员是搞工程 的。不过要想系统学习CSP的人可以从这里下载到CSP论文的最新版本。

维基百科中概要罗列了CSP模型与另外一种并发模型Actor模型的区别：

Actor模型广义上讲与CSP模型很相似。但两种模型就提供的原语而言，又有一些根本上的不同之处：

- CSP模型处理过程是匿名的，而Actor模型中的Actor则具有身份标识。
- CSP模型的消息传递在收发消息进程间包含了一个交会点，即发送方只能在接收方准备

好接收消息时才能发送消息。相反，actor模型中的消息传递是异步 的，即消息的发送和接收无需在同一时间进行，发送方可以在接收方准备好接收消息前将消息发送出去。这两种方案可以认为是彼此对偶的。在某种意义下，基于交 会点的系统可以通过构造带缓冲的通信的方式来模拟异步消息系统。而异步系统可以通过构造带消息/应答协议的方式来同步发送方和接收方来模拟交会点似的通信 方式。

- CSP使用显式的Channel用于消息传递，而Actor模型则将消息发送给命名的目的Actor。这两种方法可以被认为是对偶的。某种意义下，进程可 以从一个实际上拥有身份标识的channel接收消息，而通过将actors构造成类Channel的行为模式也可以打破actors之间的名字耦合。

二、Go Channel基本操作语法

Go Channel的基本操作语法如下：

```
c := make(chan bool) //创建一个无缓冲的bool型Channel
c <- x          //向一个Channel发送一个值
<- c            //从一个Channel中接收一个值
x = <- c        //从Channel c接收一个值并将其存储到x中
x, ok = <- c    //从Channel接收一个值，如果channel关闭了或没有数据，那么ok将被置为false
```

不带缓冲的Channel兼具通信和同步两种特性，颇受青睐。

三、Channel用作信号(Signal)的场景

1、等待一个事件(Event)

等待一个事件，有时候通过close一个Channel就足够了。例如：

```
//testwaitevent1.go
package main

import "fmt"

func main() {
        fmt.Println("Begin doing something!")
        c := make(chan bool)
        go func() {
                fmt.Println("Doing something…")
                close(c)
        }()
        <-c
        fmt.Println("Done!")
}
```

这里main goroutine通过"<-c"来等待sub goroutine中的“完成事件”，sub goroutine通过close channel促发这一事件。当然也可以通过向Channel写入一个bool值的方式来作为事件通知。main goroutine在channel c上没有任何数据可读的情况下会阻塞等待。

关于输出结果：

根据《Go memory model》中关于close channel与recv from channel的order的定义：*The closing of a channel happens before a receive that returns a zero*

value because the channel is closed.

我们可以很容易判断出上面程序的输出结果：

```
Begin doing something!
Doing something…
Done!
```

如果将close(c)换成c<-true，则根据《Go memory model》中的定义：*A receive from an unbuffered channel happens before the send on that channel completes.*
"<-c"要先于"c<-true"完成，但也不影响日志的输出顺序，输出结果仍为上面三行。

2、协同多个Goroutines

同上，close channel还可以用于协同多个Goroutines，比如下面这个例子，我们创建了100个Worker Goroutine，这些Goroutine在被创建出来后都阻塞在"<-start"上，直到我们在main goroutine中给出开工的信号："close(start)"，这些goroutines才开始真正的并发运行起来。

```
//testwaitevent2.go
package main

import "fmt"

func worker(start chan bool, index int) {
        <-start
        fmt.Println("This is Worker:", index)
}

func main() {
        start := make(chan bool)
        for i := 1; i <= 100; i++ {
                go worker(start, i)
        }
        close(start)
        select {} //deadlock we expected
}
```

3、Select

【select的基本操作】
select是Go语言特有的操作，使用select我们可以同时在多个channel上进行发送/接收操作。下面是select的基本操作。

```
select {
case x := <- somechan:
    // … 使用x进行一些操作

case y, ok := <- someOtherchan:
    // … 使用y进行一些操作，
    // 检查ok值判断someOtherchan是否已经关闭
```

```
case outputChan <- z:
    // … z值被成功发送到Channel上时

default:
    // … 上面case均无法通信时，执行此分支
}
```

【惯用法：for/select】

我们在使用select时很少只是对其进行一次evaluation，我们常常将其与for {}结合在一起使用，并选择适当时机从for{}中退出。

```
for {
        select {
        case x := <- somechan:
            // … 使用x进行一些操作

        case y, ok := <- someOtherchan:
            // … 使用y进行一些操作，
            // 检查ok值判断someOtherchan是否已经关闭

        case outputChan <- z:
            // … z值被成功发送到Channel上时

        default:
            // … 上面case均无法通信时，执行此分支
        }
}
```

【终结workers】

下面是一个常见的终结sub worker goroutines的方法，每个worker goroutine通过select监视一个die channel来及时获取main goroutine的退出通知。

```
//testterminateworker1.go
package main

import (
    "fmt"
    "time"
)

func worker(die chan bool, index int) {
    fmt.Println("Begin: This is Worker:", index)
    for {
        select {
        //case xx:
            //做事的分支
        case <-die:
            fmt.Println("Done: This is Worker:", index)
            return
        }
```

```
    }
}

func main() {
    die := make(chan bool)

    for i := 1; i <= 100; i++ {
        go worker(die, i)
    }

    time.Sleep(time.Second * 5)
    close(die)
    select {} //deadlock we expected
}
```

【终结验证】

有时候终结一个worker后，main goroutine想确认worker routine是否真正退出了，可采用下面这种方法：

```
//testterminateworker2.go
package main

import (
    "fmt"
    //"time"
)

func worker(die chan bool) {
    fmt.Println("Begin: This is Worker")
    for {
        select {
        //case xx:
        //做事的分支
        case <-die:
            fmt.Println("Done: This is Worker")
            die <- true
            return
        }
    }
}

func main() {
    die := make(chan bool)

    go worker(die)

    die <- true
    <-die
    fmt.Println("Worker goroutine has been terminated")
}
```

【关闭的Channel永远不会阻塞】

下面演示在一个已经关闭了的channel上读写的结果：

```
//testoperateonclosedchannel.go
package main

import "fmt"

func main() {
        cb := make(chan bool)
        close(cb)
        x := <-cb
        fmt.Printf("%#v\n", x)

        x, ok := <-cb
        fmt.Printf("%#v %#v\n", x, ok)

        ci := make(chan int)
        close(ci)
        y := <-ci
        fmt.Printf("%#v\n", y)

        cb <- true
}

$go run testoperateonclosedchannel.go
false
false false
0
panic: runtime error: send on closed channel
```

可以看到在一个已经close的unbuffered channel上执行读操作，回返回channel对应类型的零值，比如bool型channel返回false，int型channel返回0。但向close的channel写则会触发panic。不过无论读写都不会导致阻塞。

【关闭带缓存的channel】

将unbuffered channel换成buffered channel会怎样？我们看下面例子：

```
//testclosedbufferedchannel.go
package main

import "fmt"

func main() {
        c := make(chan int, 3)
        c <- 15
        c <- 34
        c <- 65
        close(c)
        fmt.Printf("%d\n", <-c)
```

```
        fmt.Printf("%d\n", <-c)
        fmt.Printf("%d\n", <-c)
        fmt.Printf("%d\n", <-c)

        c <- 1
}

$go run testclosedbufferedchannel.go
15
34
65
0
panic: runtime error: send on closed channel
```

可以看出带缓冲的channel略有不同。尽管已经close了，但我们依旧可以从中读出关闭前写入的3个值。第四次读取时，则会返回该channel类型的零值。向这类channel写入操作也会触发panic。

【range】

Golang中的range常常和channel并肩作战，它被用来从channel中读取所有值。下面是一个简单的实例：

```
//testrange.go
package main

import "fmt"

func generator(strings chan string) {
        strings <- "Five hour's New York jet lag"
        strings <- "and Cayce Pollard wakes in Camden Town"
        strings <- "to the dire and ever-decreasing circles"
        strings <- "of disrupted circadian rhythm."
        close(strings)
}

func main() {
        strings := make(chan string)
        go generator(strings)
        for s := range strings {
                fmt.Printf("%s\n", s)
        }
        fmt.Printf("\n")
}
```

四、隐藏状态

下面通过一个例子来演示一下channel如何用来隐藏状态：

1、例子：唯一的ID服务

```
//testuniqueid.go
```

```
package main

import "fmt"

func newUniqueIDService() <-chan string {
        id := make(chan string)
        go func() {
                var counter int64 = 0
                for {
                        id <- fmt.Sprintf("%x", counter)
                        counter += 1
                }
        }()
        return id
}
func main() {
        id := newUniqueIDService()
        for i := 0; i < 10; i++ {
                fmt.Println(<-id)
        }
}
```

```
$ go run testuniqueid.go
0
1
2
3
4
5
6
7
8
9
```

newUniqueIDService通过一个channel与main goroutine关联，main goroutine无需知道uniqueid实现的细节以及当前状态，只需通过channel获得最新id即可。

五、默认情况

我想这里John Graham-Cumming主要是想告诉我们select的default分支的实践用法。

1、select for non-blocking receive

```
idle:= make(chan []byte, 5) //用一个带缓冲的channel构造一个简单的队列

select {
case b = <-idle: //尝试从idle队列中读取
    …
default:  //队列空，分配一个新的buffer
        makes += 1
        b = make([]byte, size)
```

```
}
```

2、select for non-blocking send

```
idle:= make(chan []byte, 5) //用一个带缓冲的channel构造一个简单的队列

select {
case idle <- b: //尝试向队列中插入一个buffer
        //…
default: //队列满?

}
```

六、Nil Channels

1、nil channels阻塞

对一个没有初始化的channel进行读写操作都将发生阻塞，例子如下：

```
package main

func main() {
        var c chan int
        <-c
}

$go run testnilchannel.go
fatal error: all goroutines are asleep - deadlock!

package main

func main() {
        var c chan int
        c <- 1
}

$go run testnilchannel.go
fatal error: all goroutines are asleep - deadlock!
```

2、nil channel在select中很有用

看下面这个例子：

```
//testnilchannel_bad.go
package main

import "fmt"
import "time"

func main() {
        var c1, c2 chan int = make(chan int), make(chan int)
        go func() {
                time.Sleep(time.Second * 5)
```

```
                c1 <- 5
                close(c1)
        }()

        go func() {
                time.Sleep(time.Second * 7)
                c2 <- 7
                close(c2)
        }()

        for {
                select {
                case x := <-c1:
                        fmt.Println(x)
                case x := <-c2:
                        fmt.Println(x)
                }
        }
        fmt.Println("over")
}
```

我们原本期望程序交替输出5和7两个数字，但实际的输出结果却是：

```
5
0
0
0
… … 0死循环
```

再仔细分析代码，原来select每次按case顺序evaluate：

- 前5s，select一直阻塞；
- 第5s，c1返回一个5后被close了，“case x := <-c1”这个分支返回，select输出5，并重新select
- 下一轮select又从“case x := <-c1”这个分支开始evaluate，由于c1被close，按照前面的知识，close的channel不会阻塞，我们会读出这个 channel对应类型的零值，这里就是0；select再次输出0；这时即便c2有值返回，程序也不会走到c2这个分支
- 依次类推，程序无限循环的输出0

我们利用nil channel来改进这个程序，以实现我们的意图，代码如下：

```
//testnilchannel.go
package main

import "fmt"
import "time"

func main() {
        var c1, c2 chan int = make(chan int), make(chan int)
        go func() {
                time.Sleep(time.Second * 5)
                c1 <- 5
```

```
                close(c1)
        }()

        go func() {
                time.Sleep(time.Second * 7)
                c2 <- 7
                close(c2)
        }()

        for {
                select {
                case x, ok := <-c1:
                        if !ok {
                                c1 = nil
                        } else {
                                fmt.Println(x)
                        }
                case x, ok := <-c2:
                        if !ok {
                                c2 = nil
                        } else {
                                fmt.Println(x)
                        }
                }
                if c1 == nil && c2 == nil {
                        break
                }
        }
        fmt.Println("over")
}

$go run testnilchannel.go
5
7
over
```

可以看出：通过将已经关闭的channel置为nil，下次select将会阻塞在该channel上，使得select继续下面的分支evaluation。

七、Timers

1、超时机制Timeout

带超时机制的select是常规的tip，下面是示例代码，实现30s的超时select：

```
func worker(start chan bool) {
        timeout := time.After(30 * time.Second)
        for {
                select {
                     // … do some stuff
                case <- timeout:
```

```
                        return
                }
        }
}
```

2、心跳HeartBeart

与timeout实现类似，下面是一个简单的心跳select实现：

```
func worker(start chan bool) {
        heartbeat := time.Tick(30 * time.Second)
        for {
                select {
                     // … do some stuff
                case <- heartbeat:
                    //… do heartbeat stuff
                }
        }
}
```


Related posts:

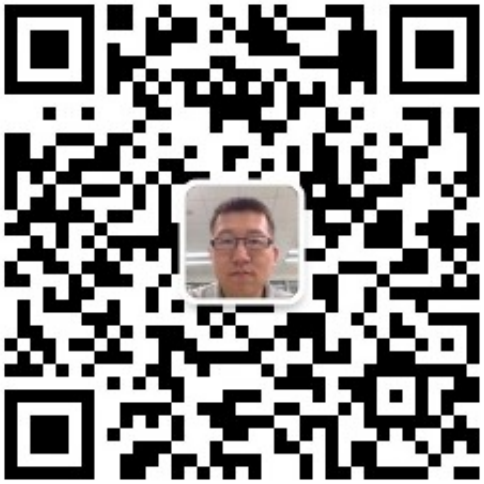
1. Go程序设计语言(三)
2. Go中的系统Signal处理
3. Go程序设计语言(一)
4. Go与C语言的互操作
5. Go程序设计语言(二)

输入关键字搜索 搜索

这里是Tony Bai的个人Blog，欢迎访问、订阅和留言！订阅Feed请点击上面图片。

如果您觉得这里的文章对您有帮助，请扫描上方二维码进行捐赠，加油后的Tony Bai将会为您呈现更多精彩的文章，谢谢！

如果您喜欢通过微信App浏览本站内容，可以扫描下方二维码，订阅本站官方微信订阅号“iamtonybai”；点击二维码，可直达本人官方微博主页^_^：



本站Powered by Digital Ocean VPS。

选择Digital Ocean VPS主机，即可获得10美元现金充值，可免费使用两个月哟！

著名主机提供商Linode 10$优惠码：linode10，在这里注册即可免费获得。

阿里云推荐码：1WFZOV，立享9折！

文章

- Go包导入与Java的差别
- vim-go更新小记
- 智慧城市到底满足的是谁的诉求
- Go 1.7中值得关注的几个变化
- 闲话智慧城市
- 理解Unikernels
- 部署devstack
- GopherChina2016后记
- Rancher使用入门
- 使用Filebeat输送Docker容器的日志

评论

- LV 在 Rancher使用入门
  很有用的文档，谢谢作者，收藏了。
-

ngrok 在 搭建自己的ngrok服务
自己搭建的ngrok1.7有严重内存泄露问题，直接使用natapp 吧。内存泄露已解决

- love02xp 在 使用Hugo搭建静态站点
感谢，试用中。。。
- 语无伦次 在 godep的一个“坑”
就看见出错了！成功了！想一想！具体怎么做的语无伦次！
- 
gtt 在 智慧城市到底满足的是谁的诉求
好文章，作为一个码农也能毫无压力的看懂，确实写的好！
- 武昌鱼股票 在 使用Hugo搭建静态站点
谢谢，用到了~~~~~~~~~~~~~~~~~~~~~
- Wish商户平台 在 使用Golang开发微信公众平台-接入验证
默默的学习怎么做。
- 奇虎分享网 在 Go 1.6中值得关注的几个变化
今天才发现你的博客，连着看了几篇呢
- Serval 在 使用astyle美化代码
加载svn client hook里是一个好主意！原先怎么没想到呢。
- 镧爨 在 Go 1.7中值得关注的几个变化
赞一个
- 下一页 »

分类

- 光影汇 (7)
- 影音坊 (36)
- 思考控 (63)
- 技术志 (478)
- 杂货铺 (75)
- 生活簿 (152)
- 职场录 (14)
- 读书吧 (14)
- 运动迷 (76)
- 驴友秀 (40)

标签

Blog Blogger C Cpp English GCC github GNU Go Golang Google Java Kernel Linux

Messi Opensource Programmer Python Soccer Solaris Subversion Ubuntu Unix Windows 世界杯 博客 女儿 学习 工作 巴萨 开源 思考 感悟 摄影 旅游 果果 梅西 球迷 生活 电影 程序员 编译器 翻译 足球 驴友

归档

- 2016 年九月 (2)
- 2016 年八月 (1)
- 2016 年六月 (2)
- 2016 年五月 (2)
- 2016 年四月 (2)
- 2016 年三月 (2)
- 2016 年二月 (3)
- 2016 年一月 (2)
- 2015 年十二月 (1)
- 2015 年十一月 (1)
- 2015 年十月 (1)
- 2015 年九月 (3)
- 2015 年八月 (5)
- 2015 年七月 (6)
- 2015 年六月 (4)
- 2015 年五月 (1)
- 2015 年四月 (2)
- 2015 年三月 (2)
- 2015 年一月 (2)
- 2014 年十二月 (5)
- 2014 年十一月 (8)
- 2014 年十月 (9)
- 2014 年九月 (2)
- 2014 年八月 (1)
- 2014 年七月 (1)
- 2014 年五月 (2)
- 2014 年四月 (5)
- 2014 年三月 (4)
- 2014 年二月 (1)
- 2014 年一月 (1)
- 2013 年十二月 (3)
- 2013 年十一月 (5)
- 2013 年十月 (6)
- 2013 年九月 (4)
- 2013 年八月 (5)
- 2013 年七月 (6)
- 2013 年六月 (2)
- 2013 年五月 (6)
- 2013 年四月 (3)
- 2013 年三月 (7)
- 2013 年二月 (4)
- 2013 年一月 (6)
- 2012 年十二月 (8)
- 2012 年十一月 (10)
- 2012 年十月 (5)

- 2012 年九月 (3)
- 2012 年八月 (10)
- 2012 年七月 (4)
- 2012 年六月 (2)
- 2012 年五月 (4)
- 2012 年四月 (10)
- 2012 年三月 (8)
- 2012 年二月 (6)
- 2012 年一月 (6)
- 2011 年十二月 (4)
- 2011 年十一月 (4)
- 2011 年十月 (5)
- 2011 年九月 (8)
- 2011 年八月 (7)
- 2011 年七月 (6)
- 2011 年六月 (7)
- 2011 年五月 (8)
- 2011 年四月 (6)
- 2011 年三月 (10)
- 2011 年二月 (7)
- 2011 年一月 (10)
- 2010 年十二月 (7)
- 2010 年十一月 (6)
- 2010 年十月 (7)
- 2010 年九月 (12)
- 2010 年八月 (8)
- 2010 年七月 (3)
- 2010 年六月 (5)
- 2010 年五月 (4)
- 2010 年四月 (2)
- 2010 年三月 (6)
- 2010 年二月 (4)
- 2010 年一月 (6)
- 2009 年十二月 (6)
- 2009 年十一月 (6)
- 2009 年十月 (5)
- 2009 年九月 (8)
- 2009 年八月 (6)
- 2009 年七月 (7)
- 2009 年六月 (1)
- 2009 年五月 (3)
- 2009 年四月 (4)
- 2009 年三月 (6)
- 2009 年二月 (4)
- 2009 年一月 (8)
- 2008 年十二月 (9)
- 2008 年十一月 (5)
- 2008 年十月 (10)
- 2008 年九月 (9)
- 2008 年八月 (13)
- 2008 年七月 (3)
- 2008 年六月 (1)
- 2008 年五月 (7)
- 2008 年四月 (4)

- 2008 年三月 (9)
- 2008 年二月 (11)
- 2008 年一月 (15)
- 2007 年十二月 (11)
- 2007 年十一月 (14)
- 2007 年十月 (4)
- 2007 年九月 (5)
- 2007 年八月 (1)
- 2007 年七月 (10)
- 2007 年六月 (10)
- 2007 年五月 (10)
- 2007 年四月 (8)
- 2007 年三月 (15)
- 2007 年二月 (4)
- 2007 年一月 (17)
- 2006 年十二月 (18)
- 2006 年十一月 (9)
- 2006 年十月 (11)
- 2006 年九月 (6)
- 2006 年八月 (5)
- 2006 年七月 (22)
- 2006 年六月 (35)
- 2006 年五月 (24)
- 2006 年四月 (26)
- 2006 年三月 (25)
- 2006 年二月 (18)
- 2006 年一月 (15)
- 2005 年十二月 (10)
- 2005 年十一月 (10)
- 2005 年九月 (13)
- 2005 年八月 (11)
- 2005 年七月 (6)
- 2005 年六月 (2)
- 2005 年五月 (3)
- 2005 年四月 (6)
- 2005 年三月 (1)
- 2005 年一月 (15)
- 2004 年十二月 (9)
- 2004 年十一月 (14)
- 2004 年十月 (2)
- 2004 年九月 (2)

链接

- @douban
- @flickr
- @github
- @googlecode
- @picasa
- @slideshare
- @twitter
- @weibo
- Hoterran
- Lionel Messi

- Puras He
- 梦想风暴
- 过眼云烟

开源项目

- buildc
- cbehave
- lcut

翻译项目

- C语言编码风格和标准
- 《Programming in Haskell》中文翻译项目