**GO并发编程**

这篇文章是参考了国外的大牛写的博客。

Go语言从基本面上就支持了并发，不像其他语言需要一个额外的库来支持，所以GO生而为并发而来。 某种程度上说，GO更加类似于Python中的协程。当然这里我们用线程来理解更符合大家的胃口。 需要强调的是**并行!=并发**, 两者是不同的。

**Go语言的goroutines、信道和死锁**

**goroutine**

Go语言中有个概念叫做goroutine, 这类似我们熟知的线程，但是更轻。

以下的程序，我们串行地去执行两次loop函数:

func loop() {

for i := 0; i < ; i++ {

fmt.Printf("%d ", i)

}

}

func main() {

loop()

loop()

}

输出会是：

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9

GO里面，我们用关键字go来启动一个goroutine,下面我们把一个loop放在一个goroutine里跑:

func main() {

go loop() // 启动一个goroutine

loop()

}

这次的输出变为:

0 1 2 3 4 5 6 7 8 9

奇怪的是，为什么只输出了一次呢？主线程一次，子routine一次，应该是两次才是啊。

原来，在goroutine还没准备跑的时候，主函数已经退出了。

main函数退出地太快了，我们要想办法阻止它过早地退出，一个办法是让main等待一下:

func main() {

go loop()

loop()

time.Sleep(time.Second) // 停顿一秒

}

这次确实输出了两次，目的达到了。

可是采用等待的办法并不好，首先我们并不知道go routine要跑多久。其次这里有硬编码。这里，在GO里面有标准的方式之一是waitgroup方法。

**waitgroup**

func main(){

var wg sync.waitgroup

for i := 0; i < ; i++ {

fmt.Printf("%d ", i)

}

wg.Add(1)

go func() {

for i := 0; i < ; i++ {

fmt.Printf("%d ", i)

}

defer wg.Done()

}

wg.Wait()

}

除此以外，还有一种通用的方法，利用信道来阻塞。

**信道**

信道是什么？简单说，是goroutine之间互相通讯的东西。类似我们Unix上的管道（可以在进程间传递消息）， 用来goroutine之间发消息和接收消息。其实，就是在做goroutine之间的内存共享。

利用关键字make来建立一个信道:

var channel chan int = make(chan int)

// 或

channel := make(chan int)

那如何向信道存消息和取消息呢？ 一个例子:

func main() {

var messages chan string = make(chan string)

go func(message string) {

messages <- message // 存消息

}("Ping!")

fmt.Println(<-messages) // 取消息

}

默认的，信道的存消息和取消息都是**阻塞的** (叫做无缓冲的信道，不过缓冲这个概念稍后了解，先说阻塞的问题)。

也就是说, 无缓冲的信道在取消息和存消息的时候都会挂起当前的goroutine，除非另一端已经准备好。

比如以下的main函数和foo函数:

var ch chan int = make(chan int)

func foo() {

ch <- 0 // 向ch中加数据，如果没有其他goroutine来取走这个数据，那么挂起foo, 直到main函数把0这个数据拿走

}

func main() {

go foo()

<- ch // 从ch取数据，如果ch中还没放数据，那就挂起main线，直到foo函数中放数据为止

}

那既然信道可以阻塞当前的goroutine, 那么回到上一部分「goroutine」所遇到的问题「如何让goroutine告诉主线我执行完毕了」 的问题来, 使用一个信道来告诉主线即可:

var complete chan int = make(chan int)

func loop() {

for i := 0; i < 10; i++ {

fmt.Printf("%d ", i)

}

complete <- 0 // 执行完毕了，发个消息

}

func main() {

go loop()

<- complete // 直到线程跑完, 取到消息. main在此阻塞住

}

如果不用信道来阻塞主线的话，主线就会过早跑完，loop线都没有机会执行、、、

其实，**无缓冲的信道永远不会存储数据，只负责数据的流通**，为什么这么讲呢？

* 从无缓冲信道取数据，必须要有数据流进来才可以，否则当前线阻塞
* 数据流入无缓冲信道, 如果没有其他goroutine来拿走这个数据，那么当前线阻塞

所以，你可以测试下，无论如何，我们测试到的无缓冲信道的大小都是0 (len(channel))

如果信道正有数据在流动，我们还要加入数据，或者信道枯竭，我们一直向无数据流入的空信道取数据呢？ 就会引起**死锁**

**死锁**

一个死锁的例子:

func main() {

ch := make(chan int)

<- ch // 阻塞main goroutine, 信道c被锁

}

执行这个程序你会看到Go报这样的错误:

fatal error: all goroutines are asleep - deadlock!

何谓死锁? 操作系统有讲过的，所有的线程或进程都在等待资源的释放。如上的程序中, 只有一个goroutine, 所以当你向里面加数据或者存数据的话，都会锁死信道， 并且阻塞当前 goroutine, 也就是所有的goroutine(其实就main线一个)都在等待信道的开放(没人拿走数据信道是不会开放的)，也就是死锁咯。

我发现死锁是一个很有意思的话题，这里有几个死锁的例子:

1. **只在单一的goroutine里操作无缓冲信道，一定死锁**。比如你只在main函数里操作信道:
2. func main() {
3. ch := make(chan int)
4. ch <- 1 // 1流入信道，堵塞当前线, 没人取走数据信道不会打开
5. fmt.Println("This line code wont run") //在此行执行之前Go就会报死锁
6. }
7. 如下也是一个死锁的例子:
8. var ch1 chan int = make(chan int)
9. var ch2 chan int = make(chan int)
10. func say(s string) {
11. fmt.Println(s)
12. ch1 <- <- ch2 // ch1 等待 ch2流出的数据
13. }
14. func main() {
15. go say("hello")
16. <- ch1 // 堵塞主线
17. }

其中主线等ch1中的数据流出，ch1等ch2的数据流出，但是ch2等待数据流入，两个goroutine都在等，也就是死锁。

1. 其实，总结来看，为什么会死锁？非缓冲信道上如果发生了流入无流出，或者流出无流入，也就导致了死锁。或者这样理解 **Go启动的所有goroutine里的非缓冲信道一定要一个线里存数据，一个线里取数据，要成对才行** 。所以下面的示例一定死锁:
2. c, quit := make(chan int), make(chan int)
3. go func() {
4. c <- 1 // c通道的数据没有被其他goroutine读取走，堵塞当前goroutine
5. quit <- 0 // quit始终没有办法写入数据
6. }()
7. <- quit // quit 等待数据的写

仔细分析的话，是由于：主线等待quit信道的数据流出，quit等待数据写入，而func被c通道堵塞，所有goroutine都在等，所以死锁。

简单来看的话，一共两个线，func线中流入c通道的数据并没有在main线中流出，肯定死锁。

但是，是否果真 **所有不成对向信道存取数据的情况都是死锁?**

如下是个反例:

func main() {

c := make(chan int)

go func() {

c <- 1

}()

}

程序正常退出了，很简单，并不是我们那个总结不起作用了，还是因为一个让人很囧的原因，main又没等待其它goroutine，自己先跑完了， 所以没有数据流入c信道，一共执行了一个goroutine, 并且没有发生阻塞，所以没有死锁错误。

那么死锁的解决办法呢？

最简单的，把没取走的数据取走，没放入的数据放入， 因为无缓冲信道不能承载数据，那么就赶紧拿走！

具体来讲，就死锁例子3中的情况，可以这么避免死锁:

c, quit := make(chan int), make(chan int)

go func() {

c <- 1

quit <- 0

}()

<- c // 取走c的数据！

<-quit

另一个解决办法是缓冲信道, 即设置c有一个数据的缓冲大小:

c := make(chan int, 1)

这样的话，c可以缓存一个数据。也就是说，放入一个数据，c并不会挂起当前线, 再放一个才会挂起当前线直到第一个数据被其他goroutine取走, 也就是只阻塞在容量一定的时候，不达容量不阻塞。

这十分类似我们Python中的队列Queue不是吗？

**无缓冲信道的数据进出顺序**

我们已经知道，无缓冲信道从不存储数据，流入的数据必须要流出才可以。

观察以下的程序:

var ch chan int = make(chan int)

func foo(id int) { //id: 这个routine的标号

ch <- id

}

func main() {

// 开启5个routine

for i := 0; i < 5; i++ {

go foo(i)

}

// 取出信道中的数据

for i := 0; i < 5; i++ {

fmt.Print(<- ch)

}

}

我们开了5个goroutine，然后又依次取数据。其实整个的执行过程细分的话，5个线的数据 依次流过信道ch, main打印之, 而宏观上我们看到的即 **无缓冲信道的数据是先到先出，但是 无缓冲信道并不存储数据，只负责数据的流通**

**缓冲信道**

终于到了这个话题了, 其实缓存信道用英文来讲更为达意: buffered channel.

缓冲这个词意思是，缓冲信道不仅可以流通数据，还可以缓存数据。它是有容量的，存入一个数据的话 , 可以先放在信道里，不必阻塞当前线而等待该数据取走。

当缓冲信道达到满的状态的时候，就会表现出阻塞了，因为这时再也不能承载更多的数据了，「你们必须把 数据拿走，才可以流入数据」。

在声明一个信道的时候，我们给make以第二个参数来指明它的容量(默认为0，即无缓冲):

var ch chan int = make(chan int, 2) // 写入2个元素都不会阻塞当前goroutine, 存储个数达到2的时候会阻塞

如下的例子，缓冲信道ch可以无缓冲的流入3个元素:

func main() {

ch := make(chan int, 3)

ch <- 1

ch <- 2

ch <- 3

}

如果你再试图流入一个数据的话，信道ch会阻塞main线, 报死锁。

也就是说，缓冲信道会在满容量的时候加锁。

其实，缓冲信道是先进先出的，我们可以把**缓冲信道看作为一个线程安全的队列**：

func main() {

ch := make(chan int, 3)

ch <- 1

ch <- 2

ch <- 3

fmt.Println(<-ch) // 1

fmt.Println(<-ch) // 2

fmt.Println(<-ch) // 3

}

**信道数据读取和信道关闭**

你也许发现，上面的代码一个一个地去读取信道简直太费事了，Go语言允许我们使用range来读取信道:

func main() {

ch := make(chan int, 3)

ch <- 1

ch <- 2

ch <- 3

for v := range ch {

fmt.Println(v)

}

}

如果你执行了上面的代码，会报死锁错误的，原因是range不等到信道关闭是不会结束读取的。也就是如果 缓冲信道干涸了，那么range就会阻塞当前goroutine, 所以死锁咯。

那么，我们试着避免这种情况，比较容易想到的是读到信道为空的时候就结束读取:

ch := make(chan int, 3)

ch <- 1

ch <- 2

ch <- 3

for v := range ch {

fmt.Println(v)

if len(ch) <= 0 { // 如果现有数据量为0，跳出循环

break

}

}

以上的方法是可以正常输出的，但是注意检查信道大小的方法不能在信道存取都在发生的时候用于取出所有数据，这个例子 是因为我们只在ch中存了数据，现在一个一个往外取，信道大小是递减的。

另一个方式是显式地关闭信道:

ch := make(chan int, 3)

ch <- 1

ch <- 2

ch <- 3

// 显式地关闭信道

close(ch)

for v := range ch {

fmt.Println(v)

}

被关闭的信道会禁止数据流入, 是只读的。我们仍然可以从关闭的信道中取出数据，但是不能再写入数据了。

**等待多gorountine的方案**

那好，我们回到最初的一个问题，**使用信道堵塞主线，等待开出去的所有goroutine跑完**。

这是一个模型，开出很多小goroutine, 它们各自跑各自的，最后跑完了向主线报告。

我们讨论如下2个版本的方案:

1. 只使用单个无缓冲信道阻塞主线
2. 使用容量为goroutines数量的缓冲信道

对于方案1, 示例的代码大概会是这个样子:

var quit chan int // 只开一个信道

func foo(id int) {

fmt.Println(id)

quit <- 0 // ok, finished

}

func main() {

count := 1000

quit = make(chan int) // 无缓冲

for i := 0; i < count; i++ {

go foo(i)

}

for i := 0; i < count; i++ {

<- quit

}

}

对于方案2, 把信道换成缓冲1000的:

quit = make(chan int, count) // 容量1000

其实区别仅仅在于一个是缓冲的，一个是非缓冲的。

对于这个场景而言，两者都能完成任务, 都是可以的。

* 无缓冲的信道是一批数据一个一个的「流进流出」
* 缓冲信道则是一个一个存储，然后一起流出去

关于Go语言的并发，还有很多东西值得研究。

**Go语言的并发和并行**

不知道你有没有注意到一个现象，还是这段代码，如果我跑在两个goroutines里面的话:

var quit chan int = make(chan int)

func loop() {

for i := 0; i < 10; i++ {

fmt.Printf("%d ", i)

}

quit <- 0

}

func main() {

// 开两个goroutine跑函数loop, loop函数负责打印10个数

go loop()

go loop()

for i := 0; i < 2; i++ {

<- quit

}

}

我们观察下输出:

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9

这是不是有什么问题？?

以前我们用线程去做类似任务的时候，系统的线程会抢占式地输出， 表现出来的是乱序地输出。而goroutine为什么是这样输出的呢？

**goroutine是在并行吗？**

我们找个例子测试下:

package main

import "fmt"

import "time"

var quit chan int

func foo(id int) {

fmt.Println(id)

time.Sleep(time.Second) // 停顿一秒

quit <- 0 // 发消息：我执行完啦！

}

func main() {

count := 1000

quit = make(chan int, count) // 缓冲1000个数据

for i := 0; i < count; i++ { //开1000个goroutine

go foo(i)

}

for i :=0 ; i < count; i++ { // 等待所有完成消息发送完毕。

<- quit

}

}

让我们跑一下这个程序(之所以先编译再运行，是为了让程序跑的尽量快,测试结果更好):

go build test.go

time ./test

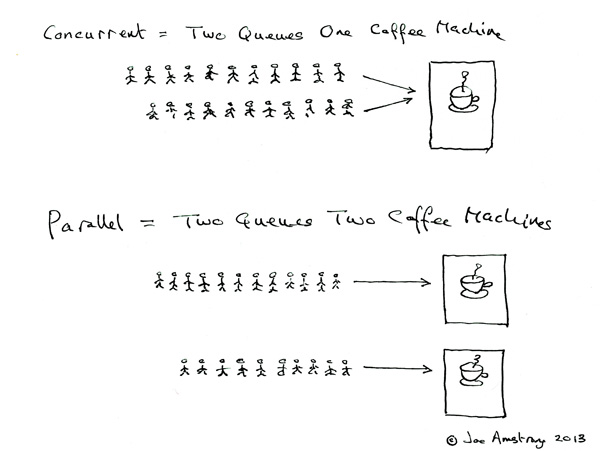
./test 0.01s user 0.01s system 1% cpu 1.016 total

我们看到，总计用时接近一秒。 貌似并行了！

我们需要首先考虑下什么是并发, 什么是并行

**并行和并发**

从概念上讲，并发和并行是不同的, 简单来说看这个图片(原图来自[这里](http://joearms.github.io/2013/04/05/concurrent-and-parallel-programming.html))



* 两个队列，一个Coffee机器，那是并发
* 两个队列，两个Coffee机器，那是并行

更多的资料： [并发不是并行](http://www.aqee.net/docs/Concurrency-is-not-Parallelism/), 当然Google上有更多关于并行和并发的区别。

那么回到一开始的疑问上，从上面的两个例子执行后的表现来看，多个goroutine跑loop函数会挨个goroutine去进行，而sleep则是一起执行的。

这是为什么？

**默认地， Go所有的goroutines只能在一个线程里跑** 。

也就是说， 以上两个代码都不是并行的，但是都是是并发的。

**如果当前goroutine不发生阻塞，它是不会让出CPU给其他goroutine的**, 所以例子一中的输出会是一个一个goroutine进行的，而sleep函数则阻塞掉了 当前goroutine, 当前goroutine主动让其他goroutine执行, 所以形成了逻辑上的并行, 也就是并发。

**真正的并行**

为了达到真正的并行，我们需要告诉Go我们允许同时最多使用多个核。

回到起初的例子，我们设置最大开2个原生线程, 我们需要用到runtime包(runtime包是goroutine的调度器):

import (

"fmt"

"runtime"

)

var quit chan int = make(chan int)

func loop() {

for i := 0; i < 100; i++ { //为了观察，跑多些

fmt.Printf("%d ", i)

}

quit <- 0

}

func main() {

runtime.GOMAXPROCS(2) // 最多使用2个核

go loop()

go loop()

for i := 0; i < 2; i++ {

<- quit

}

}

这下会看到两个goroutine会抢占式地输出数据了。

我们还可以这样显式地让出CPU时间：

func loop() {

for i := 0; i < 10; i++ {

runtime.Gosched() // 显式地让出CPU时间给其他goroutine

fmt.Printf("%d ", i)

}

quit <- 0

}

func main() {

go loop()

go loop()

for i := 0; i < 2; i++ {

<- quit

}

}

观察下结果会看到这样有规律的输出:

0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9

其实，这种主动让出CPU时间的方式仍然是在单核里跑。但手工地切换goroutine导致了看上去的“并行”。

其实作为一个Python程序员，goroutine让我更多地想到的是gevent的协程，而不是原生线程。

关于runtime包对goroutine的调度，在stackoverflow上有一个不错的答案: <http://stackoverflow.com/questions/13107958/what-exactly-does-runtime-gosched-do>

在GO1.5的版本已经调整了，默认的使用核数已经变为最大值了，设定的默认GOMAXPROCS不再是1.

**runtime调度器**

runtime调度器是个很神奇的东西，但是我**真是但愿它不存在，我希望显式调度能更为自然些，多核处理默认开启**。

关于runtime包几个函数:

* Gosched 让出cpu
* NumCPU 返回当前系统的CPU核数量
* GOMAXPROCS 设置最大的**可同时使用**的CPU核数
* Goexit 退出当前goroutine(但是defer语句会照常执行)

**总结**

我们从例子中可以看到，默认的, 所有goroutine会在一个原生线程里跑，也就是只使用了一个CPU核。

**在同一个原生线程里，如果当前goroutine不发生阻塞，它是不会让出CPU时间给其他同线程的goroutines的**，这是Go运行时对goroutine的调度，我们也可以使用runtime包来手工调度。

本文开头的两个例子都是限制在单核CPU里执行的，所有的goroutines跑在一个线程里面，分析如下:

* 对于代码例子一（loop函数的那个），每个goroutine没有发生堵塞(直到quit流入数据), 所以在quit之前每个goroutine不会主动让出CPU，也就发生了串行打印
* 对于代码例子二（time的那个），每个goroutine在sleep被调用的时候会阻塞，让出CPU, 所以例子二并发执行。

那么关于我们开启多核的时候呢？Go语言对goroutine的调度行为又是怎么样的？

我们可以在Golang官方网站的[这里](http://golang.org/doc/faq#goroutines) 找到一句话:

When a coroutine blocks, such as by calling a blocking system call, the run-time automatically moves other coroutines on the same operating system thread to a different, runnable thread so they won't be blocked.

也就是说:

**当一个goroutine发生阻塞，Go会自动地把与该goroutine处于同一系统线程的其他goroutines转移到另一个系统线程上去，以使这些goroutines不阻塞**

**开启多核的实验**

仍然需要做一个实验，来测试下多核支持下goroutines的对原生线程的分配, 也验证下我们所得到的结论“goroutine不阻塞不放开CPU”。

实验代码如下:

package main

import (

"fmt"

"runtime"

)

var quit chan int = make(chan int)

func loop(id int) { // id: 该goroutine的标号

for i := 0; i < 10; i++ { //打印10次该goroutine的标号

fmt.Printf("%d ", id)

}

quit <- 0

}

func main() {

runtime.GOMAXPROCS(2) // 最多同时使用2个核

for i := 0; i < 3; i++ { //开三个goroutine

go loop(i)

}

for i := 0; i < 3; i++ {

<- quit

}

}

多跑几次会看到类似这些输出(不同机器环境不一样):

0 0 0 0 0 1 1 0 0 1 0 0 1 0 1 2 1 2 1 2 1 2 1 2 1 2 2 2 2 2

0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2

0 0 0 0 0 0 0 1 1 1 1 1 0 1 0 1 0 1 2 1 2 1 2 2 2 2 2 2 2 2

0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 2 0 2 0 2 2 2 2 2 2 2 2

0 0 0 0 0 0 0 1 0 0 1 0 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 2 2

执行它我们会发现以下现象:

* 有时会发生抢占式输出(说明Go开了不止一个原生线程，达到了真正的并行)
* 有时会顺序输出, 打印完0再打印1, 再打印2(说明Go开一个原生线程，单线程上的goroutine不阻塞不松开CPU)

那么，我们还会观察到一个现象，无论是抢占地输出还是顺序的输出，都会有那么两个数字表现出这样的现象:

* 一个数字的所有输出都会在另一个数字的所有输出之前

原因是， 3个goroutine分配到至多2个线程上，就会至少两个goroutine分配到同一个线程里，单线程里的goroutine 不阻塞不放开CPU, 也就发生了顺序输出。

**Go语言并发的设计模式和应用场景**

以下设计模式和应用场景来自Google IO上的关于Goroutine的PPT: <https://talks.golang.org/2012/concurrency.slide>

本文的示例代码在: <https://github.com/hit9/Go-patterns-with-channel>

**生成器**

在Python中我们可以使用yield关键字来让一个函数成为生成器，在Go中我们可以使用信道来制造生成器(一种lazy load类似的东西)。

当然我们的信道并不是简单的做阻塞主线的功能来使用的哦。

下面是一个制作自增整数生成器的例子，直到主线向信道索要数据，我们才添加数据到信道

func xrange() chan int{ // xrange用来生成自增的整数

var ch chan int = make(chan int)

go func() { // 开出一个goroutine

for i := 0; ; i++ {

ch <- i // 直到信道索要数据，才把i添加进信道

}

}()

return ch

}

func main() {

generator := xrange()

for i:=0; i < 1000; i++ { // 我们生成1000个自增的整数！

fmt.Println(<-generator)

}

}

这不禁叫我想起了Python中可爱的xrange, 所以给了生成器这个名字！

**服务化**

比如我们加载一个网站的时候，例如我们登入新浪微博，我们的消息数据应该来自一个独立的服务，这个服务只负责 返回某个用户的新的消息提醒。

如下是一个使用示例:

func get\_notification(user string) chan string{

/\*

\* 此处可以查询数据库获取新消息等等..

\*/

notifications := make(chan string)

go func() { // 悬挂一个信道出去

notifications <- fmt.Sprintf("Hi %s, welcome to weibo.com!", user)

}()

return notifications

}

func main() {

jack := get\_notification("jack") // 获取jack的消息

joe := get\_notification("joe") // 获取joe的消息

// 获取消息的返回

fmt.Println(<-jack)

fmt.Println(<-joe)

}

**多路复合**

上面的例子都使用一个信道作为返回值，可以把信道的数据合并到一个信道的。 不过这样的话，我们需要按顺序输出我们的返回值（先进先出）。

如下，我们假设要计算很复杂的一个运算 100-x , 分为三路计算， 最后统一在一个信道中取出结果:

func do\_stuff(x int) int { // 一个比较耗时的事情，比如计算

time.Sleep(time.Duration(rand.Intn(10)) \* time.Millisecond) //模拟计算

return 100 - x // 假如100-x是一个很费时的计算

}

func branch(x int) chan int{ // 每个分支开出一个goroutine做计算并把计算结果流入各自信道

ch := make(chan int)

go func() {

ch <- do\_stuff(x)

}()

return ch

}

func fanIn(chs... chan int) chan int {

ch := make(chan int)

for \_, c := range chs {

// 注意此处明确传值

go func(c chan int) {ch <- <- c}(c) // 复合

}

return ch

}

func main() {

result := fanIn(branch(1), branch(2), branch(3))

for i := 0; i < 3; i++ {

fmt.Println(<-result)

}

}

**select监听信道**

Go有一个语句叫做select，用于监测各个信道的数据流动。

如下的程序是select的一个使用例子，我们监视三个信道的数据流出并收集数据到一个信道中。

func foo(i int) chan int {

c := make(chan int)

go func () { c <- i }()

return c

}

func main() {

c1, c2, c3 := foo(1), foo(2), foo(3)

c := make(chan int)

go func() { // 开一个goroutine监视各个信道数据输出并收集数据到信道c

for {

select { // 监视c1, c2, c3的流出，并全部流入信道c

case v1 := <- c1: c <- v1

case v2 := <- c2: c <- v2

case v3 := <- c3: c <- v3

}

}

}()

// 阻塞主线，取出信道c的数据

for i := 0; i < 3; i++ {

fmt.Println(<-c) // 从打印来看我们的数据输出并不是严格的1,2,3顺序

}

}

有了select, 我们在 多路复合中的示例代码中的函数fanIn还可以这么来写(这样就不用开好几个goroutine来取数据了):

func fanIn(branches ... chan int) chan int {

c := make(chan int)

go func() {

for i := 0 ; i < len(branches); i++ { //select会尝试着依次取出各个信道的数据

select {

case v1 := <- branches[i]: c <- v1

}

}

}()

return c

}

使用select的时候，有时需要超时处理, 其中的timeout信道相当有趣:

timeout := time.After(1 \* time.Second) // timeout 是一个计时信道, 如果达到时间了，就会发一个信号出来

for is\_timeout := false; !is\_timeout; {

select { // 监视信道c1, c2, c3, timeout信道的数据流出

case v1 := <- c1: fmt.Printf("received %d from c1", v1)

case v2 := <- c2: fmt.Printf("received %d from c2", v2)

case v3 := <- c3: fmt.Printf("received %d from c3", v3)

case <- timeout: is\_timeout = true // 超时

}

}

**结束标志**

我们已经讲过信道的一个很重要也很平常的应用，就是使用无缓冲信道来阻塞主线，等待goroutine结束。

这样我们不必再使用timeout。

那么对上面的timeout来结束主线的方案作个更新：

func main() {

c, quit := make(chan int), make(chan int)

go func() {

c <- 2 // 添加数据

quit <- 1 // 发送完成信号

} ()

for is\_quit := false; !is\_quit; {

select { // 监视信道c的数据流出

case v := <-c: fmt.Printf("received %d from c", v)

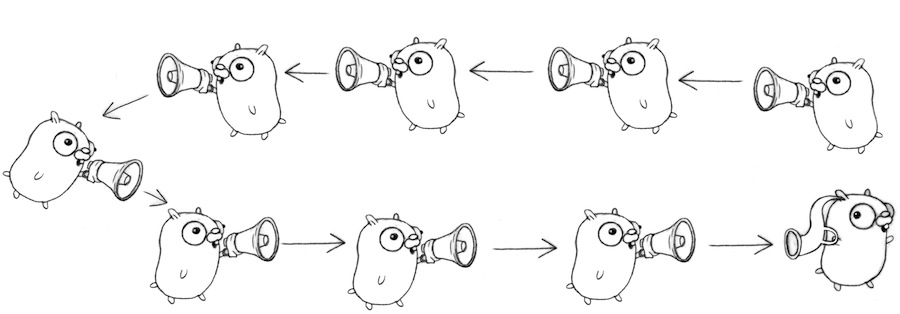
case <-quit: is\_quit = true // quit信道有输出，关闭for循环

}

}

}

**菊花链**



简单地来说，数据从一端流入，从另一端流出，看上去好像一个链表，不知道为什么要取这么个尴尬的名字。。

菊花链的英文名字叫做: Daisy-chain, 它的一个应用就是做过滤器，比如我们来筛下100以内的素数(你需要先知道什么是筛法)

程序有详细的注释，不再说明了。

/\*

\* 利用信道菊花链筛法求某一个整数范围的素数

\* 筛法求素数的基本思想是：把从1开始的、某一范围内的正整数从小到大顺序排列，

\* 1不是素数，首先把它筛掉。剩下的数中选择最小的数是素数，然后去掉它的倍数。

\* 依次类推，直到筛子为空时结束

\*/

package main

import "fmt"

func xrange() chan int{ // 从2开始自增的整数生成器

var ch chan int = make(chan int)

go func() { // 开出一个goroutine

for i := 2; ; i++ {

ch <- i // 直到信道索要数据，才把i添加进信道

}

}()

return ch

}

func filter(in chan int, number int) chan int {

// 输入一个整数队列，筛出是number倍数的, 不是number的倍数的放入输出队列

// in: 输入队列

out := make(chan int)

go func() {

for {

i := <- in // 从输入中取一个

if i % number != 0 {

out <- i // 放入输出信道

}

}

}()

return out

}

func main() {

const max = 100 // 找出100以内的所有素数

nums := xrange() // 初始化一个整数生成器

number := <-nums // 从生成器中抓一个整数(2), 作为初始化整数

for number <= max { // number作为筛子，当筛子超过max的时候结束筛选

fmt.Println(number) // 打印素数, 筛子即一个素数

nums = filter(nums, number) //筛掉number的倍数

number = <- nums // 更新筛子

}

}

**随机数生成器**

信道可以做生成器使用，作为一个特殊的例子，它还可以用作随机数生成器。如下是一个随机01生成器:

func rand01() chan int {

ch := make(chan int)

go func () {

for {

select { //select会尝试执行各个case, 如果都可以执行，那么随机选一个执行

case ch <- 0:

case ch <- 1:

}

}

}()

return ch

}

func main() {

generator := rand01() //初始化一个01随机生成器

//测试，打印10个随机01

for i := 0; i < 10; i++ {

fmt.Println(<-generator)

}

}

**定时器**

我们刚才其实已经接触了信道作为定时器, time包里的After会制作一个定时器。

看看我们的定时器吧！

/\*

\* 利用信道做定时器

\*/

package main

import (

"fmt"

"time"

)

func timer(duration time.Duration) chan bool {

ch := make(chan bool)

go func() {

time.Sleep(duration)

ch <- true // 到时间啦！

}()

return ch

}

func main() {

timeout := timer(time.Second) // 定时1s

for {

select {

case <- timeout:

fmt.Println("already 1s!") // 到时间

return //结束程序

}

}

}

**TODO**

Google的应用场景例子。

本篇主要总结了使用信道, goroutine的一些设计模式。

**信号**

sigRecv1:=make(chan os.Signal,1)

sigs1:=[]os.Signal{syscall.SIGINT,syscall.SIGQUIT}

signal.Notify(sigRecv1,sigs1...)

sigRecv2:=make(chan os.Signal,1)

sigs2:=[]os.Signal{syscall.SIGINT,syscall.SIGQUIT}

signal.Notify(sigRecv2,sigs2...)

// 接着我们用两个for循环来接收消息.考虑到主程序会退出，所以用waitgroup等待完成后退出.

var wg sync.WaitGroup

wg.Add(2)

go func(){

for sig:=range sigRecv1{

fmt.Printf("Received a signal from sigRecv1%v",sig)

}

fmt.Printf("End. [sigRecv1]\n")

}()

go func(){

for sig:=range sigRecv2{

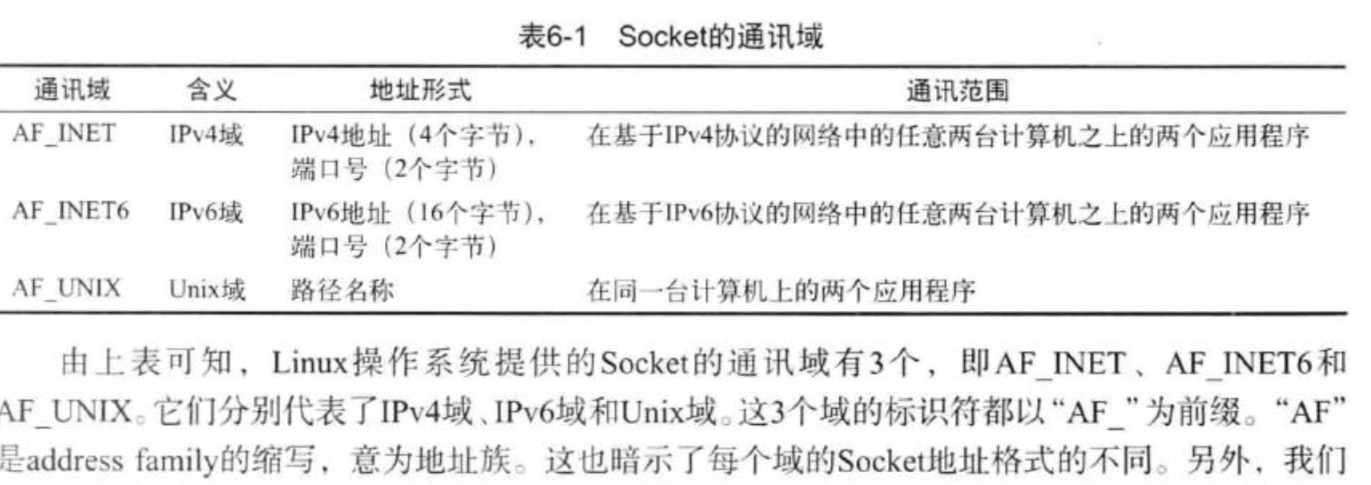
fmt.Printf("Received a signal from sigRecv1%v",sig)

}

fmt.Printf("End. [sigRecv2]\n")

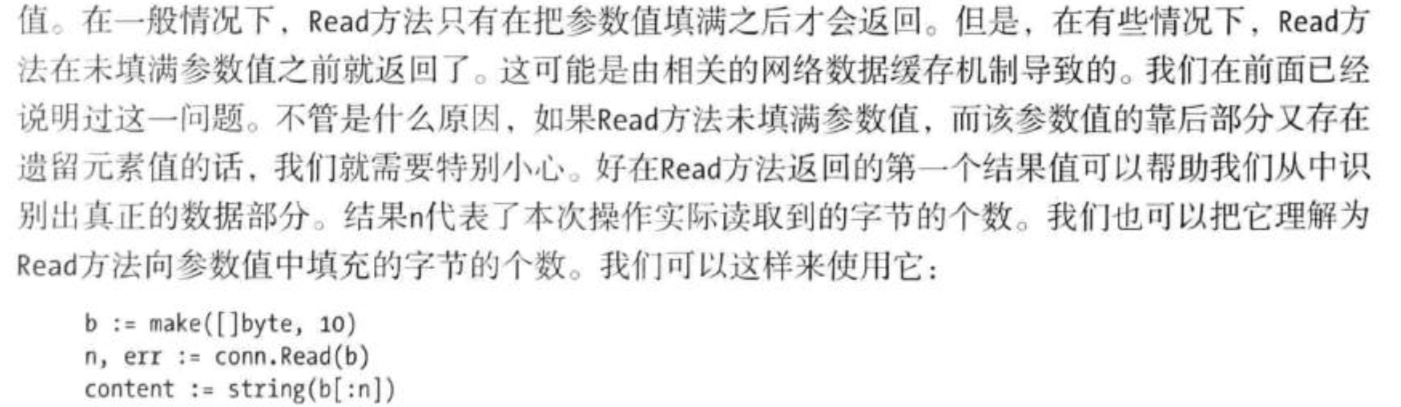
}()

wg.Wait()

int socket(int domain,int type,int protocol) socket type 类型 

**TCP UDP**

tcp 连结类型

可能会传回io.EOF,表明连接结束

var dataBuffer bytes.Buffer

b:=make([byte,10])

for {

n,err:=conn.Read(b)

if err!=nil{

if err==io.EOF{

fmt.Println("The connection is closed.")

conn.Close()

} else{

fmt.Printf("Read Error:%s\n",err)

}

break

}

dataBuffer.Write(b[:n])

}

bufio 是 Buffered I/O缩写.

由于net.Conn类型实现了接口类型io.Reader中的Read接口，所以该接口的类型的一个实现类型。因此，我们可以使用bufio.NewReader函数来包装变量conn,像这样： reader:=bufio.NewReader(conn) 可以调用reader变量的ReadBytes("\n")来接受一个byte类型的参数. 当然很多情况下并不是查找一个但直接字符那么简单。比如,http协议中规定,消息头部的信息的末尾是两个空行,即是字符串"\r\n\r\n", writer:=bufio.NewWriter(conn)来写入数据

**close()**

关闭连接

**LocalAddr & RemoteAddr 方法**

conn.RmeoteAddr().Network() .String()

**SetDeadline, SetReadDeadline,SetWrteDeadline**

b:=make([]byte,10)

conn.SetDeadline(time.Now().Add(2\*time.Second))

for {

n,err:=conn.Read(b)

}

我们通过调用time.Now()方法当前绝对时间加2s.2s后的那一刻.假设之后的第二次迭代时间超过2s.则第三次迭代尚未开始，就已经超时了，所以改为.

b:=make([]byte,10)

for {

conn.SetDeadline(time.Now().Add(2\*time.Second))

n,err:=conn.Read(b)

}

**一个server 的简单实现**



缓冲器中的缓存机制，在很多时候，它会读取比足够多更多一点的数据到其中的缓冲器。会产生提前读取的问题.

net/http在 net/tcp的基础上构建了非常好用的接口，除此以外，标准库，net/rcp中api为我们提供了两个go程序之间建立通讯和交换数据的另外一种方式。 远程过程调用（remote procedure call）也是基于TCP/IP协议栈的。 Unix系统中，POSIX标准中定义的线程及其属性和操作方法被广泛认可. Linux中称为NPTL。

**多线程**

线程一个id, TID. 编写高并发程序的建议：

1. 控制临界区的纯度。临界区中的仅仅包含操作共享数据的代码。
2. 控制临界区的粒度。
3. 减少临界区中代码的执行耗时。
4. 避免长时间的持有互斥变量。
5. 优先使用院子操作而不是互斥量。

GO语言是操作系统提供的内核线程之上搭建了一个特有的两级线程模型。 Goroutine代表的正确的含义： 不要用共享内存的方式来通信。作为替代，应该用通信作为手段来共享内存。 把数据放在共享内存区域中供多个线程中的程序访问的这种方式虽然在基本思想上非常简单，但是却让并发访问控制变得异常复杂。只有做好了很多约束和限制，才有可能让这些简单的方法正确的实施。但是正确性的同时，也需要有可伸缩性。 Go语言不推荐用共享内存区的方式传递数据。作为替代，优先使用Channel。作为多个Goroutine之间的传递数据，并且还会保证其过程的同步。 GO语言线程3个必须知道的核心元素。 M：machine.一个M代表了一个内核线程。 P：processor.一个P代表了M所需的上下文Content. G:Goroutine.一个G代表了对一段需要被并发执行的GO语言代码的封装。

**GO并发编程.**

type IntChan chan int. 元素类型为int 的通道类型。 这样的通道是双向类型的。

如果向一个nil（被关闭的channel）发送消息，则这个进程会被永久阻塞。 需要注意的是：当我们向一个通道发送消息的时候，我们得到的是一个值的copy，当这个copy形成之后，之后对原来的值的修改都不会影响通道中的值。

**select例子：**

select语句和switch语句不同的是，跟在每个case 后面的只能是针对某个通道的发送语句或者接受语句。 针对每个case 都有初始化了一个通道。通道的类型不受任何约束。元素的类型和容量是任意的。

**分支选择规则**

重做到右，从上到下。 但是当系统发现第一个满足选择条件的case时，运行时系统就会执行该case所包含的语句。这也意味着其他case 会被忽略。如果同时有多个case满足条件，那么运行时，系统会通过一个伪随机算法决定那个case会被执行。例如下面的代码，发送5个范围在【1，3】的整数：

package main

import (

"fmt"

)

func main() {

chanCap := 5

ch7 := make(chan int, chanCap)

for i := 0; i < chanCap; i++ {

select {

case ch7 <- 1:

case ch7 <- 2:

case ch7 <- 3:

}

}

for i := 0; i < chanCap; i++ {

fmt.Printf("%v\n", <-ch7)

}

}

但是，如果所有的case 都不满足（并且没有default case），就会阻塞住。直到某个case 被触发。 所以通常情况下，default case 都是必要的.且default只能包含一个default case. 两个变量赋值， e,ok:=<-chan 第二个变量指的是通道是否被关闭。

package main

import (

"fmt"

)

func main() {

go func(){

for {

select{

case e,ok:=<-ch11:

if !ok{

fmt.Println("End.")

break

}else{

fmt.Printf("%d\n",e)ß

}

}

}

}

}

//修改方案如下，不然会死锁。

func main(){

go func(){

var e int

ok:=true //声明在外，方便外层使用

for {

select{

case e,ok=<-ch11:

if !ok{

fmt.Println("End.")

break

}else{

fmt.Printf("%d\n",e)

}

}

if !ok{

break

}

}

}

}

有的时候我们需要早点关闭流程。这里我们添加新的超时行为。

timeout:=make(chan bool,1)

go func(){

time.Sleep(time.Millisecond)

timeout<-false

}

在原来的基础上，添加

go func(){

var e int

ok := true

for {

select{

case e,ok=<-ch11:

...

case ok=<-timeout:

fmt.Println("Timeout.")

break

}

}

if !ok{

break

}

}

**非缓冲的Channel**

缓冲通道：由于元素的传递是异步的，所以发送操作在成功向通道发送元素值之后就会立即结束。 非缓冲通道：等待能够接受该元素值的那个接收操作。并且只有确保成功接收，才会真正的完成执行。

package main

import (

"fmt"

"time"

)

func main() {

unbufChan := make(chan int)

go func() {

fmt.Printf("Sleep a second ...\n")

time.Sleep(time.Second)

num := <-unbufChan

fmt.Printf("Received a integer %d.\n", num)

}()

num := 1

fmt.Printf("Send integer %d ...\n", num)

unbufChan <- num

fmt.Printf("Done")

}

**select 语句与非缓冲通道**

在使用select语句向某个非缓冲通道发送元素的时候，我们需要打起鸡血。因为，与操作缓冲通道的select语句相比，它被阻塞的概率非常之大。其基本原因依然是非缓冲通道会以同步的方式传递元素值。

**time包与Channel**

**定时器**

首先，结构体类型，time.Timer. new的两种方法：

time.NewTimer(Duration) & time.AfterFunc.

Duration=3\*time.Hour+36\*.time.Minute

Timer{

Reset()

Stop()

}

//到达时间，是通过字段C（chan time.Time）缓冲通道。 时间一道向自己发送一个time.Time类型元素。绝对到期时间。

重构之前的结构：

case <-time.NewTimer(time.Millisecond).C:

fmt.Println("Timeout")

ok=false

break

定时器是可以被复用的。所以在case中的接受雨具中初始化是浪费的。下面的使用方式更好：

go func(){

var timer \*time.Timer

for{

select{

case <-func()<-chan time.Time{

if timer==nil{

timer=time.NewTimer(time.Millisecond)

}else{

timer.Reset(time.Millisedcond)

}

return timer.C

}():

fmt.Println("Timeout")

ok=false

break

}

}

}

**断续器**

断续器与定时器的使用场景不同，当作超时触发器是不合适的。因为它对到期事件的传达虽然可能被放弃，当绝对不会被延误。断续器一旦被初始化，它所有的到期时间都是固定的了。 固定不变的到期时间恰恰使断续器非常适合被作为定时任务的触发器。 例如要求两次执行之间的最短间隔时间为10分钟。我们可以这样编写满足这一需求的代码：

**高并发负载均衡器**

QPS(Query Per Second,每秒查询量) & TPS(Transactions Per Second,每秒事物处理量)。

切片和数组都不是并发安全的。所以用一个chan来存储结果。