Go语言 Channel 通道

Channel介绍

单纯地将函数并发执行是没有意义的。函数与函数间需要交换数据才能体现并发执行函数的意义。

虽然可以使用共享内存进行数据交换，但是共享内存在不同的goroutine中容易发生竞态问题。为了保证数据交换的正确性，必须使用互斥量对内存进行加锁，这种做法势必造成性能问题。

Go语言的并发模型是CSP（Communicating Sequential Processes），提倡通过通信共享内存而不是通过共享内存而实现通信。

如果说goroutine是Go程序并发的执行体，channel就是它们之间的连接。channel是可以让一个goroutine发送特定值到另一个goroutine的通信机制。

Go 语言中的通道（channel）是一种特殊的类型。通道像一个传送带或者队列，总是遵循**先入先出**（First In First Out）的规则，保证收发数据的顺序。每一个通道都是一个具体类型的导管，也就是声明channel的时候需要为其指定元素类型。

channel 是Go语言在语言级别提供的 goroutine 间的通信方式。我们可以使用 channel 在两个或多个 goroutine 之间传递消息。当您作为 goroutine 执行并发活动时，需要在 goroutine 之间共享资源或数据，通道充当 goroutine 之间的管道（管道）并提供一种机制来保证同步交换。

channel 是进程内的通信方式，因此通过 channel 传递对象的过程和调用函数时的参数传递行为比较一致，比如也可以传递指针等。如果需要跨进程通信，我们建议用分布式系统的方法来解决，比如使用 Socket 或者 HTTP 等通信协议。Go语言对于网络方面也有非常完善的支持。

channel 是类型相关的，也就是说，一个 channel 只能传递一种类型的值，这个类型需要在声明 channel 时指定。可以将其认为是一种类型安全的管道。

Channel通道的特性

Go语言中的通道（channel）是一种特殊的类型。在任何时候，同时只能有一个 goroutine 访问通道进行发送和获取数据。goroutine 间通过通道就可以通信。

通道像一个传送带或者队列，总是遵循先入先出（First In First Out）的规则，保证收发数据的顺序。

根据数据交换的行为，有两种类型的通道：**无缓冲通道**和**缓冲通道**。无缓冲通道用于执行goroutine之间的**同步通信**，而缓冲通道用于执行**异步通信**。无缓冲通道保证在发送和接收发生的瞬间两个goroutine之间的交换。缓冲通道没有这样的保证。

Channel类型

通道本身需要一个类型进行修饰，就像切片类型需要标识元素类型。通道的元素类型就是在其内部传输的数据类型，channel是一种类型，一种引用类型。声明如下：

```
var 通道变量 chan 通道类型
```

- 通道类型：通道内的数据类型。
- 通道变量：保存通道的变量。

chan 类型的空值是 nil，声明后需要配合 make 后才能使用。

```
var ch1 chan int   // 声明一个传递整型的通道
var ch2 chan bool  // 声明一个传递布尔型的通道
var ch3 chan []int // 声明一个传递int切片的通道
```

创建Channel通道

通道是引用类型，通道类型的空值是nil。声明的通道后需要使用make函数初始化之后才能使用。格式如下：

```
通道实例 := make(chan 数据类型,[缓冲大小])
```

- 数据类型：通道内传输的元素类型。
- 通道实例：通过make创建的通道句柄。
- channel的缓冲大小是可选的。

```
ch4 := make(chan int)
ch5 := make(chan bool)
ch6 := make(chan []int)
```

Channel操作

通道有发送（send）、接收(receive）和关闭（close）三种操作。

发送和接收都使用`<-`符号。

使用通道发送数据

通道发送数据的格式

通道的发送使用特殊的操作符`<-`，将数据通过通道发送的格式为：

```
通道变量 <- 值
```

- 通道变量：通过make创建好的通道实例。
- 值：可以是变量、常量、表达式或者函数返回值等。值的类型必须与ch通道的元素类型一致。

示例：

```
ch := make(chan int)
ch <- 10 // 把10发送到ch中
```

把数据往通道中发送时，如果接收方一直都没有接收，那么发送操作将持续阻塞。Go 程序运行时能智能地发现一些永远无法发送成功的语句并做出提示。

使用通道接收数据

通道接收数据的格式

`<-`运算符附加到通道变量（ch）的左侧，以接收来自通道的值。

通道接收同样使用`<-`操作符，通道接收有如下特性：

1. 通道的收发操作在不同的两个 goroutine 间进行。由于通道的数据在没有接收方处理时，数据发送方会持续阻塞，因此通道的接收必定在另外一个 goroutine 中进行。
2. 接收将持续阻塞直到发送方发送数据。如果接收方接收时，通道中没有发送方发送数据，接收方也会发生阻塞，直到发送方发送数据为止。
3. 每次接收一个元素。通道一次只能接收一个数据元素。

通道的数据接收一共有以下 4 种写法。

(1) 阻塞接收数据

阻塞模式接收数据时，将接收变量作为`<-`操作符的左值，格式如下：

```
data := <-ch
```

执行该语句时将会阻塞，直到接收到数据并赋值给 data 变量。

(2) 非阻塞接收数据

使用非阻塞方式从通道接收数据时，语句不会发生阻塞，格式如下：

```
data, ok := <-ch
```

- data：表示接收到的数据。未接收到数据时，data 为通道类型的零值。
- ok：表示是否接收到数据。

非阻塞的通道接收方法可能造成高的 CPU 占用，因此使用非常少。如果需要实现接收超时检测，可以配合 select 和计时器 channel 进行，可以参见后面的内容。

(3) 接收任意数据，忽略接收的数据

阻塞接收数据后，忽略从通道返回的数据，格式如下：

```
<-ch
```

执行该语句时将会发生阻塞，直到接收到数据，但接收到的数据会被忽略。这个方式实际上只是通过通道在 goroutine 间阻塞收发实现并发同步。

```
package main

import (
    "fmt"
)

func main() {
    // 构建一个通道
    ch := make(chan int)

    // 开启一个并发匿名函数
    go func() {
        fmt.Println("start goroutine")
        // 通过通道通知main的goroutine
        ch <- 0 // 匿名 goroutine 即将结束时，通过通道通知 main 的 goroutine，这一句会一直阻塞直到 main 的 goroutine 接收为止。
        fmt.Println("exit goroutine")
```

```
    }()
    fmt.Println("wait goroutine")

    // 等待匿名goroutine
    <-ch  // 开启 goroutine 后，马上通过通道等待匿名 goroutine 结束
    fmt.Println("all done")
}
#结果
wait goroutine
start goroutine
exit goroutine
all done
```

(4) 循环接收

通道的数据接收可以借用 for range 语句进行多个元素的接收操作，格式如下：

```
for data := range ch {

}
```

通道 ch 是可以进行遍历的，遍历的结果就是接收到的数据。数据类型就是通道的数据类型。通过 for 遍历获得的变量只有一个，即上面例子中的 data。

```
package main

import (
    "fmt"
    "time"
)

func main() {
    // 构建一个整型元素的通道
    ch := make(chan int)

    // 开启一个并发匿名函数，将匿名函数并发执行。
    go func() {
        // 从3循环到0
        for i := 3; i >= 0; i-- {
            // 将 3 到 0 之间的数值依次发送到通道 ch 中
            ch <- i
            // 每次发送完时等待
            time.Sleep(time.Second)
        }
    }()

    // 遍历接收通道数据
    for data := range ch {
        // 打印通道数据
        fmt.Println(data)
        // 当遇到数据0时，退出接收循环
        if data == 0 {
            break
        }
```

```
    }
}
#结果
3
2
1
0
```

关闭通道

我们通过调用内置的close函数来关闭通道

```
close(ch)
```

关于关闭通道需要注意的事情是，只有在通知接收方goroutine所有的数据都发送完毕的时候才需要关闭通道。通道是可以被垃圾回收机制回收的，它和关闭文件是不一样的，在结束操作之后关闭文件是必须要做的，但**关闭通道不是必须的。**

关闭后的通道有以下特点：

1. 对一个关闭的通道再发送值就会导致panic。
2. 对一个关闭的通道进行接收会一直获取值直到通道为空。
3. 对一个关闭的并且没有值的通道执行接收操作会得到对应类型的零值。
4. 关闭一个已经关闭的通道会导致panic。

如何判断一个 channel 是否已经被关闭？我们可以在读取的时候使用多重返回值的方式：

```
x, ok := <-ch
```

这个用法与 map 中的按键获取 value 的过程比较类似，只需要看第二个 bool 返回值即可，如果返回值是 false 则表示 ch 已经被关闭。

> 如果你的管道不往里存值或者取值的时候一定记得关闭管道

单向通道

Go语言的类型系统提供了单方向的 channel 类型，顾名思义，单向 channel 就是只能用于写入或者只能用于读取数据。当然 channel 本身必然是同时支持读写的，否则根本没法用。

假如一个 channel 真的只能读取数据，那么它肯定只会是空的，因为你没机会往里面写数据。同理，如果一个 channel 只允许写入数据，即使写进去了，也没有丝毫意义，因为没有办法读取到里面的数据。所谓的单向 channel 概念，其实只是**对 channel 的一种使用限制。**

单向通道的声明格式

我们在将一个 channel 变量传递到一个函数时，可以通过将其指定为单向 channel 变量，从而限制该函数中可以对此 channel 的操作，比如只能往这个 channel 中写入数据，或者只能从这个 channel 读取数据。

单向 channel 变量的声明非常简单，只能写入数据的通道类型为`chan<-`，只能读取数据的通道类型为`<-chan`，格式如下：

```
var 通道实例 chan<- 元素类型     // 只能写入数据的通道
var 通道实例 <-chan 元素类型     // 只能读取数据的通道
```

- 元素类型：通道包含的元素类型。
- 通道实例：声明的通道变量。

示例：

```
ch := make(chan int)
// 声明一个只能写入数据的通道类型，并赋值为ch
var chSendOnly chan<- int = ch
//声明一个只能读取数据的通道类型，并赋值为ch
var chRecvOnly <-chan int = ch
```

上面的例子中，chSendOnly 只能写入数据，如果尝试读取数据，将会出现报错。

同理，chRecvOnly 也是不能写入数据的。

当然，使用 make 创建通道时，也可以创建一个只写入或只读取的通道：

```
ch := make(<-chan int)
var chReadOnly <-chan int = ch
<-chReadOnly
```

上面代码编译正常，运行也是正确的。但是，一个不能写入数据只能读取的通道是毫无意义的。

time包中的单向通道

time 包中的计时器会返回一个 timer 实例，代码如下：

```
timer := time.NewTimer(time.Second)
```

timer的Timer类型定义如下：

```
type Timer struct {
    C <-chan Time
    r runtimeTimer
}
```

C 通道的类型就是一种只能读取的单向通道。如果此处不进行通道方向约束，一旦外部向通道写入数据，将会造成其他使用到计时器的地方逻辑产生混乱。

因此，单向通道有**利于代码接口的严谨性。**

无缓冲的通道

Go语言中无缓冲的通道（unbuffered channel）是指在接收前没有能力保存任何值的通道。这种类型的通道要求发送 goroutine 和接收 goroutine 同时准备好，才能完成发送和接收操作。

如果两个 goroutine 没有同时准备好，通道会导致先执行发送或接收操作的 goroutine 阻塞等待。这种对通道进行发送和接收的交互行为本身就是同步的。其中任意一个操作都无法离开另一个操作单独存在。

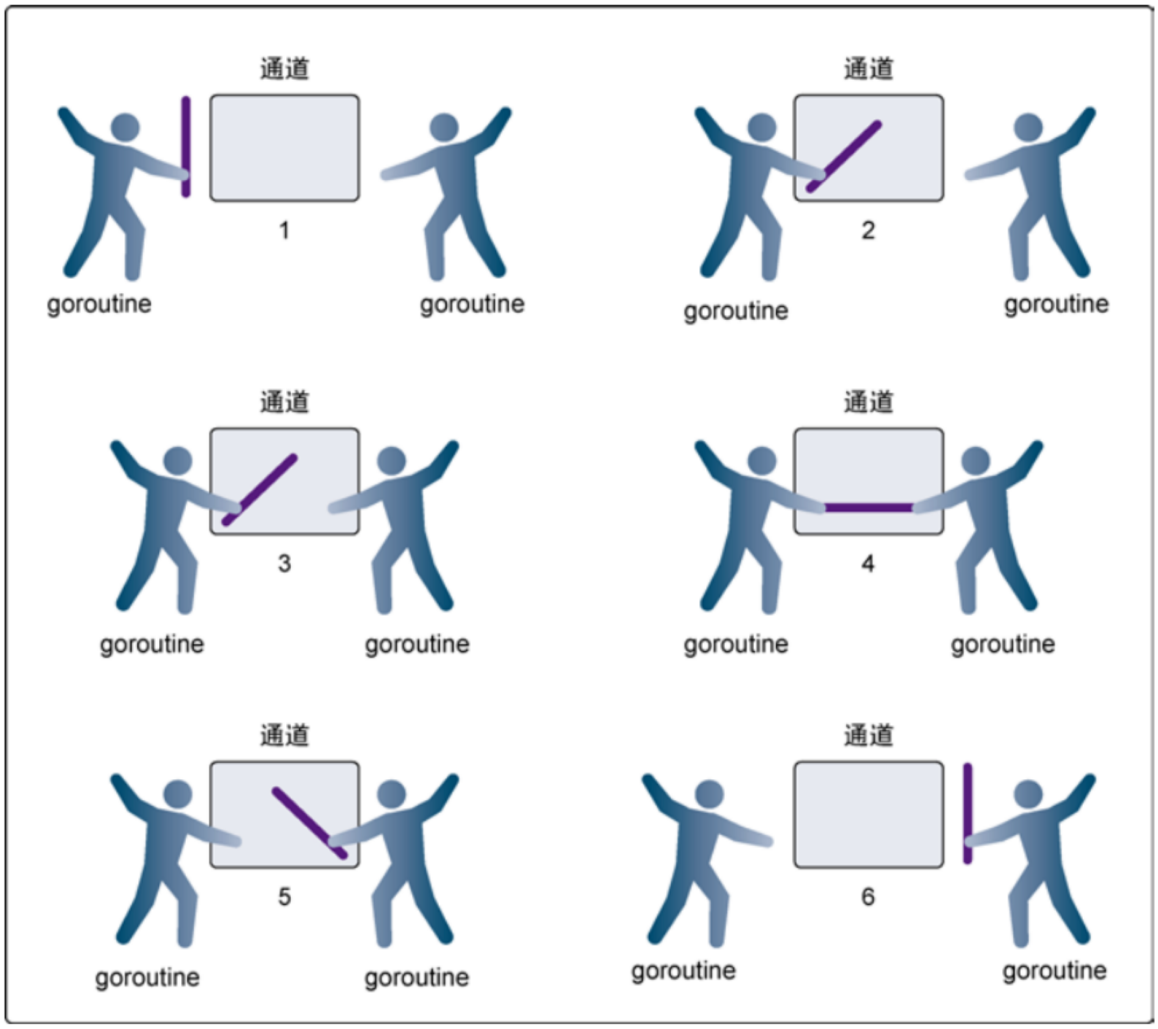
阻塞指的是由于某种原因数据没有到达，当前协程（线程）持续处于等待状态，直到条件满足才解除阻塞。

同步指的是在两个或多个协程（线程）之间，保持数据内容一致性的机制。

简单来说**无缓冲的通道必须有接收才能发送。**

使用无缓冲通道进行通信将导致发送和接收的goroutine同步化。因此，无缓冲通道也被称为**同步通道。**

下图展示两个 goroutine 如何利用无缓冲的通道来共享一个值。



使用无缓冲的通道在 goroutine 之间同步

示例:

```
package main

import (
    "fmt"
)

// 接收通道
func recv(c chan int) {
    ret := <-c
    fmt.Println("接收成功", ret)
}

func main() {
    ch := make(chan int)
    go recv(ch) // 启用goroutine从通道接收值
    ch <- 10    // 发送10值
    fmt.Println("发送成功")
}
```

```
#结果
发送成功
接收成功 10
```

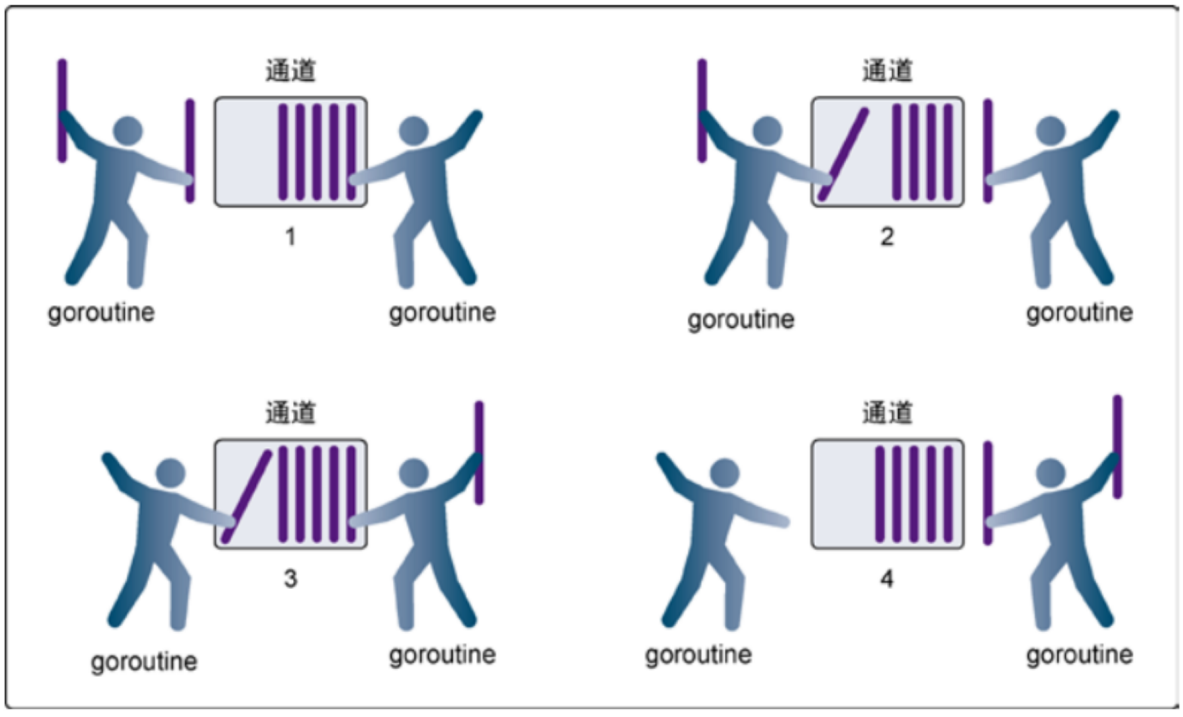
带缓冲的通道

Go语言中有缓冲的通道（buffered channel）是一种在被接收前能存储一个或者多个值的通道。这种类型的通道并不强制要求 goroutine 之间必须同时完成发送和接收。通道会阻塞发送和接收动作的条件也会不同。只有在通道中没有要接收的值时，接收动作才会阻塞。只有在通道没有可用缓冲区容纳被发送的值时，发送动作才会阻塞。

这导致有缓冲的通道和无缓冲的通道之间的一个很大的不同：无缓冲的通道保证进行发送和接收的 goroutine 会在同一时间进行数据交换；有缓冲的通道没有这种保证。

在无缓冲通道的基础上，为通道增加一个有限大小的存储空间形成带缓冲通道。带缓冲通道在发送时无需等待接收方接收即可完成发送过程，并且不会发生阻塞，只有当存储空间满时才会发生阻塞。同理，如果缓冲通道中有数据，接收时将不会发生阻塞，直到通道中没有数据可读时，通道将会再度阻塞。

无缓冲通道保证收发过程同步。无缓冲收发过程类似于快递员给你电话让你下楼取快递，整个递交快递的过程是同步发生的，你和快递员不见不散。但这样做快递员就必须等待所有人下楼完成操作后才能完成所有投递工作。如果快递员将快递放入快递柜中，并通知用户来取，快递员和用户就成了异步收发过程，效率可以有明显的提升。带缓冲的通道就是这样的一个“快递柜”。



使用有缓冲的通道在 goroutine 之间同步数据

创建带缓冲通道

格式：

```
通道实例 := make(chan 通道类型, 缓冲大小)
```

- 通道类型：和无缓冲通道用法一致，影响通道发送和接收的数据类型。
- 缓冲大小：决定通道最多可以保存的元素数量。
- 通道实例：被创建出的通道实例。

示例：

```
package main

import "fmt"

func main() {
    // 创建一个3个元素缓冲大小的整型通道
    ch := make(chan int, 3)
    // 查看当前通道的大小
    fmt.Println(len(ch)) // 带缓冲的通道在创建完成时，内部的元素是空的，因此使用 len() 获取到的返回值为 0
    // 发送3个整型元素到通道，因为使用了缓冲通道。即便没有 goroutine 接收，发送者也不会发生阻塞。
    ch <- 1
    ch <- 2
    ch <- 3
    // 查看当前通道的大小
    fmt.Println(len(ch)) // 填充了 3 个通道，此时的通道长度变为 3
}
#结果
0
3
```

只要通道的容量大于零，那么该通道就是有缓冲的通道，通道的容量表示通道中能存放元素的数量。就像你小区的快递柜只有那么个多格子，格子满了就装不下了，就阻塞了，等到别人取走一个快递员就能往里面放一个。

我们可以使用内置的len函数获取通道内元素的数量，使用cap函数获取通道的容量，虽然我们很少会这么做。

阻塞条件

带缓冲通道在很多特性上和无缓冲通道是类似的。无缓冲通道可以看作是长度永远为 0 的带缓冲通道。因此根据这个特性，带缓冲通道在下面列举的情况下依然会发生阻塞：

- 带缓冲通道被填满时，尝试再次发送数据时发生阻塞。
- 带缓冲通道为空时，尝试接收数据时发生阻塞。

为什么Go语言对通道要限制长度而不提供无限长度的通道?

我们知道通道（channel）是在两个 goroutine 间通信的桥梁。使用 goroutine 的代码必然有一方提供数据，一方消费数据。当提供数据一方的数据供给速度大于消费方的数据处理速度时，如果通道不限制长度，那么内存将不断膨胀直到应用崩溃。因此，限制通道的长度有利于约束数据提供方的供给速度，供给数据量必须在消费方处理量+通道长度的范围内，才能正常地处理数据。

Channel超时机制（select）

Go语言没有提供直接的超时处理机制，所谓超时可以理解为当我们上网浏览一些网站时，如果一段时间之后不作操作，就需要重新登录。

那么我们应该如何实现这一功能呢，这时就可以使用 select 来设置超时。

虽然 select 机制不是专门为超时而设计的，却能很方便的解决超时问题，因为 select 的特点是只要其中有一个 case 已经完成，程序就会继续往下执行，而不会考虑其他 case 的情况。

超时机制本身虽然也会带来一些问题，比如在运行比较快的机器或者高速的网络上运行正常的程序，到了慢速的机器或者网络上运行就会出问题，从而出现结果不一致的现象，但从根本上来说，解决死锁问题的价值要远大于所带来的问题。

select 的用法与 switch 语言非常类似，由 select 开始一个新的选择块，每个选择条件由 case 语句来描述。

与 switch 语句相比，select 有比较多的限制，其中最大的一条限制就是每个 case 语句里必须是一个 IO 操作，大致的结构如下：

```
select {
    case <-chan1:
    // 如果chan1成功读到数据，则进行该case处理语句
    case chan2 <- 1:
    // 如果成功向chan2写入数据，则进行该case处理语句
    default:
    // 如果上面都没有成功，则进入default处理流程
}
```

在一个 select 语句中，Go语言会按顺序从头至尾评估每一个发送和接收的语句。

如果其中的任意一语句可以继续执行（即没有被阻塞），那么就从那些可以执行的语句中任意选择一条来使用。

如果没有任意一条语句可以执行（即所有的通道都被阻塞），那么有如下两种可能的情况：

- 如果给出了 default 语句，那么就会执行 default 语句，同时程序的执行会从 select 语句后的语句中恢复；
- 如果没有 default 语句，那么 select 语句将被阻塞，直到至少有一个通信可以进行下去。

示例：

```
package main

import (
    "fmt"
    "time"
)

func main() {
    ch := make(chan int)
    quit := make(chan bool)
    //新开一个协程
    go func() {
        for {
            select {
            case num := <-ch: //接收到ch通道值
                fmt.Println("num = ", num)
            case <-time.After(3 * time.Second): // 没有接收到ch值那么等待3秒超时
                fmt.Println("超时")
                quit <- true // 写入
            }
        }
    }() //别忘了() 匿名函数自己执行

    // 循环写入到ch通道，即：0、1、2、3、4
```

```
    for i := 0; i < 5; i++ {
        ch <- i // 写入
        time.Sleep(time.Second)
    }
    <-quit // 接收值为true 结束程序
    fmt.Println("程序结束")
}
#结果
num =  0
num =  1
num =  2
num =  3
num =  4
超时
程序结束
```