go channel使用及其实现原理

迈莫coding 关注 IP属地: 北京
2021.01.15 15:03:44 字数 1,684 阅读 3,104

目录

channel背景

channel是Go的核心类型，是Go语言内置的类型，你无需引包，就能使用它。你可以把它看作一个管道，在Go语言中流传着一句话，"执行业务处理的goroutine不要通过共享内存通信，要通过channel管道进行共享数据"。

channel和Go的另一种特性goroutine一起为并发编程提供了优雅的，便利的方案，来应对并发场景。

channel基本用法

channel的基本用法非常简单，它提供了三种类型，分别为**只能接收**，**只能发送**，**既能接收也能发送**这三种类型。因此它的语法为：

```
chan<- struct{} // 只能发送struct
<-chan struct{} // 只能从chan里接收struct
chan string // 既能接收也能发送
```

我们把既能发送也能接收的chan被称为双向chan，把只能接收或者只能发送的chan称为单向chan。其中，"<-"表示单向chan，如果你记不住，我告诉你一个简便的方法：**这个箭头总是射向左边的，元素类型总在最右边。如果箭头指向 chan，就表示可以往 chan 中塞数据；如果箭头远离 chan，就表示 chan 会往外吐数据。**

通过`make`关键字，我们可以初始化一个`chan`，未初始化的chan的零值为`nil`。你可以设置他的容量，第二个参数为缓冲池的容量大小，也可以理解为即使`chan`未消费完，也可以存储数据。

```
make(chan int, 8)
```

如果chan中还有数据，那么从这个chan中接收数据就不会阻塞，如果chan中数据未达到队列容量，那么向该chan中存储数据也不会阻塞，反之会阻塞。

还有一个知识点要记住：**nil 是 chan 的零值，是一种特殊的 chan，对值是 nil 的 chan 的发送接收调用者总是会阻塞。**

接下来，我们用代码来学习一下chan的三种类型

- 只能接收数据的chan

> 代码示例

```go
package main

import "fmt"

// a 表示只能接收数据的chan
func goChanA(a <-chan int) {
  b := <-a
  fmt.Println("只能接收数据的channal[a]接收到的数据值为", b)
}

func main() {
  ch := make(chan int, 2)
  go goChanA(ch)
  // 往ch中写入数据值
  ch <- 2
  time.Sleep(time.Second)
}
```

结果

只能接收数据的channal[a]接收到的数据值为 2

- 只能发送数据的chan

> 代码示例

```go
package main

import "fmt

func main() {
  ch := make(chan<- int, 2)
  ch <- 200
}
```

往 `chan` 中发送一个数据使用“`ch<-`”。

这里的 `ch` 是 `chan int` 类型或者是 `chan <-int`。

channel应用场景

- 数据交流：当作并发的 buffer 或者 queue，解决生产者 - 消费者问题。多个 goroutine 可以并发当作生产者（Producer）和消费者（Consumer）。
- 数据传递：一个goroutine将数据交给另一个goroutine，相当于把数据的拥有权托付出去。
- 信号通知：一个goroutine可以将信号(closing，closed，data ready等)传递给另一个或者另一组goroutine。
- 任务编排：可以让一组goroutine按照一定的顺序并发或者串行的执行，这就是编排功能。
- 锁机制：利用channel实现互斥机制。

channel实现原理

channel数据结构

channel一个类型管道，通过它可以在goroutine之间发送消息和接收消息。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 Processes)的并发模型，通过 Channel实现这种同步模式。

channel结构体

```go
//path:src/runtime/chan.go
type hchan struct {
  qcount uint           // 当前队列列中剩余元素个数
  dataqsiz uint         // 环形队列长度，即可以存放的元素个数
  buf unsafe.Pointer    // 环形队列列指针
  elemsize uint16       // 每个元素的大小
```

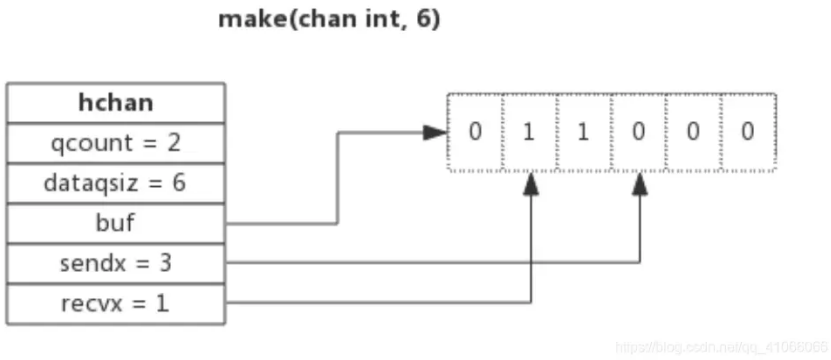
```
    closed uint32        // 标识关闭状态
    elemtype *_type      // 元素类型
    sendx uint           // 队列下标，指示元素写入入时存放到队列列中的位置 x
    recvx uint           // 队列下标，指示元素从队列列的该位置读出
    recvq waitq          // 等待读消息的goroutine队列
    sendq  waitq         // 等待写消息的goroutine队列
    lock mutex           // 互斥锁，chan不允许并发读写
}
```

从数据结构可以看出channel由队列、类型信息、goroutine等待队列组成。

channel实现方式

chan内部实现了一个缓冲队列作为缓冲区，队列的长度是创建chan时指定的。

下图展示了可缓存6个元素的channel示意图：



- dataqsiz：指向队列的长度为6，即可缓存6个元素
- buf：指向队列的内存，队列中还剩余两个元素
- qcount：当前队列中剩余的元素个数
- sendx：指后续写入元素的位置
- recvx：指从该位置读取数据

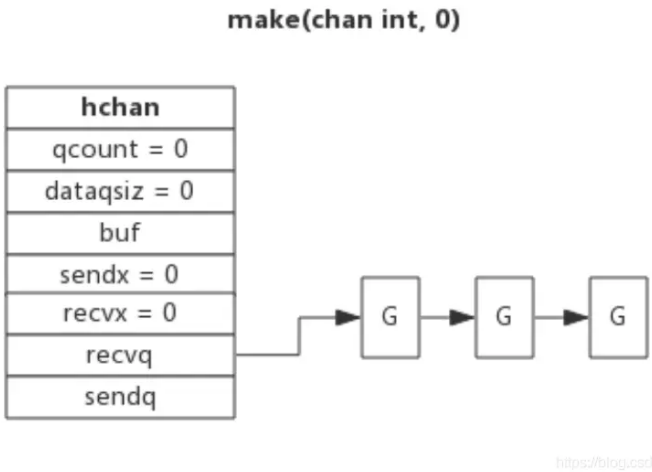
等待队列

从channel中读数据，如果channel缓冲区为空或者没有缓冲区，当前goroutine会被阻塞；向channel中写数据，如果channel缓冲区已满或者没有缓冲区，当前goroutine会被阻塞。

被阻塞的goroutine将会被挂在channel的等待队列中：

- 因读阻塞的goroutine会被向channel写入数据的goroutine唤醒
- 因写阻塞的goroutine会被从channel读数据的goroutine唤醒

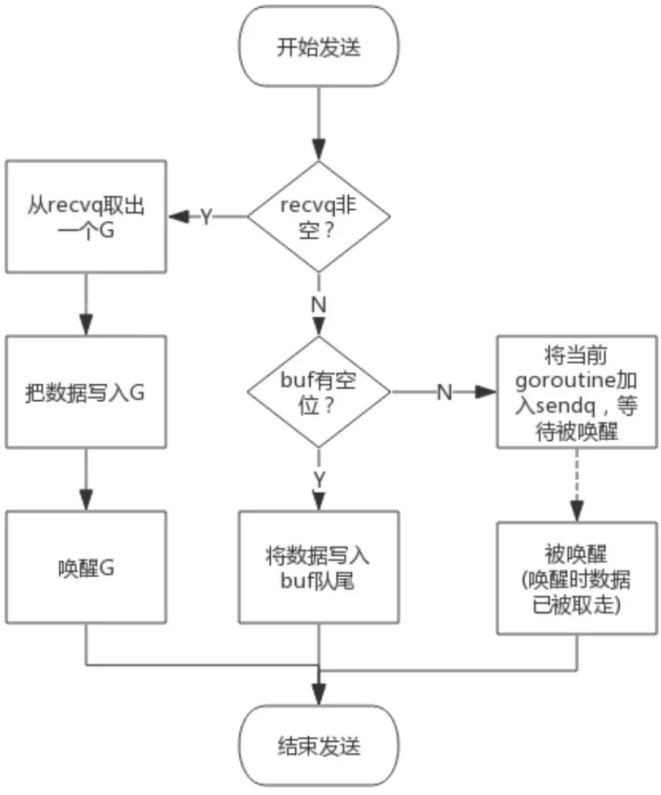
下面展示了一个没有缓冲区的channel，有几个goroutine阻塞等待数据：



注意，**一般情况下recvq和sendq至少有一个为空。只有一个例外，那就是同一个goroutine使用select语句向channel一边写数据一边读数据。**

向channel写数据

流程图：

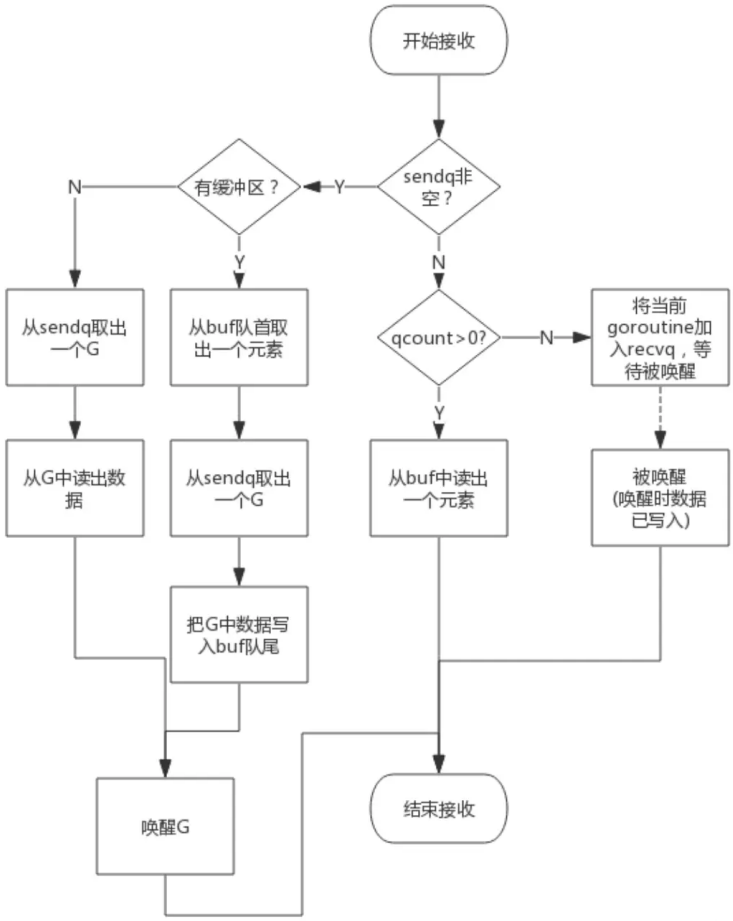


详细过程

- 如果recvq队列不为空，说明缓冲区没有数据或者没有缓冲区，此时直接从recvq等待队列中取出一个G，并把数据写入，最后把该G唤醒，结束发送过程；
- 如果缓冲区有空余位置，则把数据写入缓冲区中，结束发送过程；
- 如果缓冲区没有空余位置，则把数据写入G，将当前G写入sendq队列，进入休眠，等待被读goroutine唤醒；

从channel读数据

流程图



详细过程

- 如果等待发送队列sendq不为空，且没有缓冲区，直接从sendq队列中取出G，把G中数据读出，最后把G唤醒，结束读取过程；
- 如果等待发送队列sendq不为空，说明缓冲区已满，从缓冲队列中首部读取数据，从sendq等待发送队列中取出G，把G中的数据写入缓冲区尾部，结束读取过程；
- 如果缓冲区中有数据，则从缓冲区取出数据，结束读取过程；

channel注意事项

- 向已经关闭的channel中写入数据会发生Panic
- 关闭已经关闭的channel会发生Panic
- 关闭值为nil的channel会发生Panic