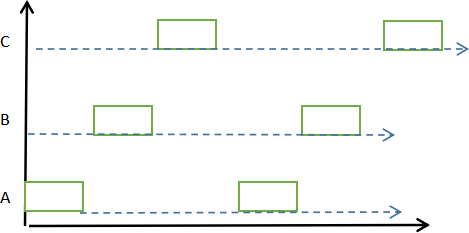
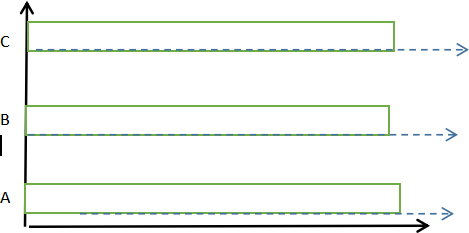
channel通信详解

在go语言中最好的特性即是多并发，这可以让我们的效率得到很大的提升，本文主要是对channel进行深入探讨。

**并发**:并发是在同一个时刻只有一条指令执行，但是多个指令被轮换的切换，这样然我们错觉的认为这些指令是同时进行运 行的，但是微观上而言，时间被分为若干段，使多个进程交替执行，打个比方，我们吃饭的时候我们先使用筷子夹菜吃饭， 随后我们再用汤勺喝汤，此后我们再进行夹菜吃饭，它们仍会有一个先后顺序之分的。即是在同一时间的不同时间段分别执 行。



**并行**：而并行可以在同一时间段使多个进程同时进行运行，但基于此肯定是需要多核的，单核是完成不了此种特性的。比方 说，我们有两台洗衣机，我们都从3点开始运行，4点停止，此时我们两台洗衣机都同时完成了工作，这就是并行。



而go中的goroutine是go中的并发体的重要思想（本文不重点探讨），而chan则是他们的媒介，当我们其中一个goroutine需要另一个goroutine的信息反馈时，此时我们就需要chan作为中间媒介进行通信，防止出现死锁。

type hchan struct {

qcount uint dataqsiz uint

// total data in the queue队列中有效元素个数

// size of the circular queue环形队列的大小

buf unsafe.Pointer // points to an array of dataqsiz elements指向缓冲区的指针

elemsize uint16//指明元素的大小closed uint32//判断chan是否已关闭

elemtype \*\_type // element type 元素类型

sendx uint // send index记录buf循环链表中发送的的下标recvx uint // receive index记录buf循环链表中接收的的下标

recvq waitq // list of recv waiters接收方（<-channel）抽象出来的的队列，是个双向链表

sendq waitq // list of send waiters同理，发送方（channel<-xxx）抽象出来的队列，也是个双向链表

// 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

//互斥资源的保护锁，在持有本互斥锁的时候，绝对不要修改goroutine的状态不然很有可能在栈扩缩容的时候出现死锁

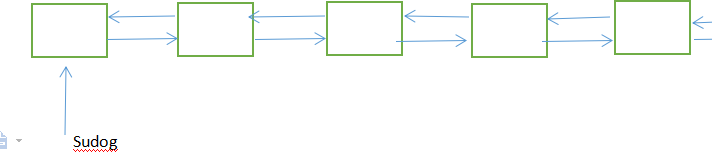
}

**waitq**

type waitq struct { first \*sudog last \*sudog

}

其中first为sudog的首指针，end表示链表的最后一个



// sudog represents a g in a wait list, such as for sending/receiving

// on a channel.

//

// sudog is necessary because the g synchronization object relation

// is many-to-many. A g can be on many wait lists, so there may be

// many sudogs for one g; and many gs may be waiting on the same

// synchronization object, so there may be many sudogs for one object.

//

// sudogs are allocated from a special pool. Use acquireSudog and

// releaseSudog to allocate and free them. type sudog struct {

// The following fields are protected by the hchan.lock of the

// channel this sudog is blocking on. shrinkstack depends on

// this for sudogs involved in channel ops.

g \*g //代表当前goroutine的地址，

next \*sudog //指向下个sudog，因为我们是个双向链表prev \*sudog //同理，指向上一个sudog

elem unsafe.Pointer // data element (may point to stack)元素指针

// The following fields are never accessed concurrently.

// For channels, waitlink is only accessed by g.

// For semaphores, all fields (including the ones above)

// are only accessed when holding a semaRoot lock.

acquiretime int64//分配sudog releasetime int64//释放sudog ticket uint32// 随 机 数

// isSelect indicates g is participating in a select, so

// g.selectDone must be CAS'd to win the wake-up race. isSelect bool//代表当前g是否被选择

// success indicates whether communication over channel c

// succeeded. It is true if the goroutine was awoken because a

// value was delivered over channel c, and false if awoken

// because c was closed.

success bool//表示goroutine是否通信成功，若g被唤醒是应为值被传送到channel中就为true，如果chan被关闭并且被唤醒则会是FAL

parent \*sudog // semaRoot binary tree semaroot形成一个树堆

waitlink \*sudog // g.waiting list or semaRoot goroutine等待队列或semaroot（此处不讨论semaroot） waittail \*sudog // semaRoot

c \*hchan // channel

}

此处的每个sudog对应着一个g，sudog是肯定需要的，因为我们的g可以在多个等待队列中，因此一个g就会对应多个sudog，当然很多g可能会等待同一个同步对象（感觉可以理解为内存访问资源），因此进而一个同步对象可以拥有多个sudog。此处的意思是我们chan底层首先会对消息进行接收并放入缓冲区中，如果是发信息则会对senx进行++操作，如果 是接收信息的话我们就会对recvx进行++操作。当我们的缓冲区为空或者缓冲区满了，我们就会使用等待队列即waitq，即 在chan中放置一个接收队列和一个发送队列，当我们的goroutine被阻塞时，我们就会将g的引用指针和相关数据放入这个recvq或sendq中，而每一个recvq或者sendq又会被抽象成一个含有n个sudog的struct的链表，每个sudog用来存放上面 所说的g的引用指针和相关数据等，直到此g再次被唤醒，我们才会将sudog从waitq中拿出来交给chan。

上述为channel实现的底层结构体，channel实现的机制也是由队列（先入先出）进行实现。而channel具体实现就是一个 环形缓冲区，但是channel拥有读写指针，读指针仅仅用作读取数据，写指针仅仅用于写数据，而lock并发锁的保护也是因 为但我们有多人读或写时我们需要对其进行分配，来实现互斥的进行访问资源（操作系统中的知识）

接下来我们会介绍相关底层方法的操作

func makechan(t \*chantype, size int) \*hchan { elem := t.elem

// compiler checks this but be safe. if elem.size >= 1<<16 {

throw("makechan: invalid channel element type")

}

if hchanSize%maxAlign != 0 || elem.align > maxAlign { throw("makechan: bad alignment")

}

mem, overflow := math.MulUintptr(elem.size, uintptr(size)) if overflow || mem > maxAlloc-hchanSize || size < 0 {

panic(plainError("makechan: size out of range"))

}

// Hchan does not contain pointers interesting for GC when elements stored in buf do not contain pointers.

// buf points into the same allocation, elemtype is persistent.

// SudoG's are referenced from their owning thread so they can't be collected.

// TODO(dvyukov,rlh): Rethink when collector can move allocated objects.

//如果缓冲区类型中未包含指针或者不需要进行缓冲，chan和缓冲区则是相同的一片区域。 var c \*hchan

switch {

case mem == 0:

// Queue or element size is zero.

c = (\*hchan)(mallocgc(hchanSize, nil, true))

// Race detector uses this location for synchronization. c.buf = c.raceaddr()

case elem.ptrdata == 0:

// Elements do not contain pointers.

// Allocate hchan and buf in one call.

c = (\*hchan)(mallocgc(hchanSize+mem, nil, true)) c.buf = add(unsafe.Pointer(c), hchanSize)

default:

// Elements contain pointers. c = new(hchan)

c.buf = mallocgc(mem, elem, true)

}

c.elemsize = uint16(elem.size) c.elemtype = elem

c.dataqsiz = uint(size) lockInit(&c.lock, lockRankHchan)

if debugChan {

print("makechan: chan=", c, "; elemsize=", elem.size, "; dataqsiz=", size, "\n")

}

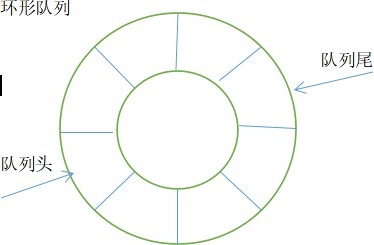
return c

}

我们传入的是chantype来进行chan的相关描述，用数据结构来表示T类型；上面所述代码首先是对数据类型的判断是否合 理以及chan对象8字节对齐后是否符合，即我们不能超过八字节，随后我们进而对chan申请的缓冲区大小判断，缓冲区大 小肯定是不能超过系统总内存的，并且长度肯定也不可能<0，不然直接panic。最后我们就会向系统申请内存空间，如果我 们没有设置缓冲区，我们直接创建一个chan，此时的chan和缓冲区相当于使用同一片内存，如果我们设立了缓冲区，我们 则还需要创建缓冲区大小，chan和缓冲区是两片不同的内存。

此外的send和receive操作我们不展示全部代码，若读者有兴趣可以自行查看go/src/runtime/chan.go进行查看，此处我们 对send和receive简单解释（后续有时间会详细进行解释）

**send** 我们的send操作首先会对操作进行判断，判断chan是否为空，以及是否被阻塞，随后我们开始发送数据时会首先进行加锁操作，如果我们的接收者还有并且被阻塞着，我们则直接从waitq中取出接收者来进行接收数据，如果我们没有接收者 我们就会先向buf中填入数据，并对sendx进行++和qcount++，一旦我们的缓冲区没有或为或者缓冲区满则会放入waitq 进行等待，直到再次被唤醒，下文会对其再次的解释，当我们的g进入等待时我们就会释放锁 **receive** 接收操作首先也会对chan进行相关分析，但是有个很怪的点就是当我们的chan已经关闭后，但是buf仍然有数据的话我们仍然可以从其中读取 数据，按理说这样是不行的，但是golang有着自己的垃圾回收机制，所以不用担心。随后我们会判断是否存在被阻塞的发 送者，如果有我们直接从发送者那里接收数据即可，这时我们是不用对其进行上锁操作的，下文同样会给出，随后我们会对 缓冲区的数据进行接收读取，对recvx++操作，并对qcount–操作。



其中chan队列分为无缓冲区和有缓冲区队列**无缓冲区chan**

func main(){ ch:=make(chan int,0) go func(){

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

fmt.Println("向channel发送数据",i)

}

}()

for i:=0;i<=2;i++{ num:=<-ch

fmt.Println("接收chan中传过来的数据",num)

}

}

/\*

向channel发送数据 0

接收channel中传过来的数据 0

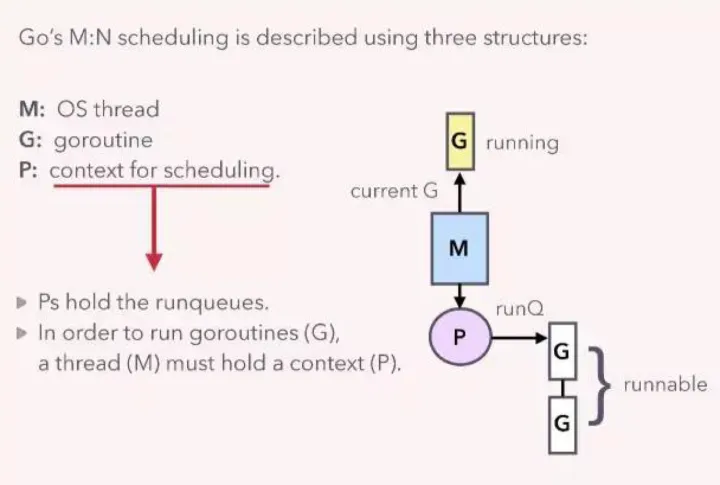
接收channel中传过来的数据 1

向channel发送数据 1

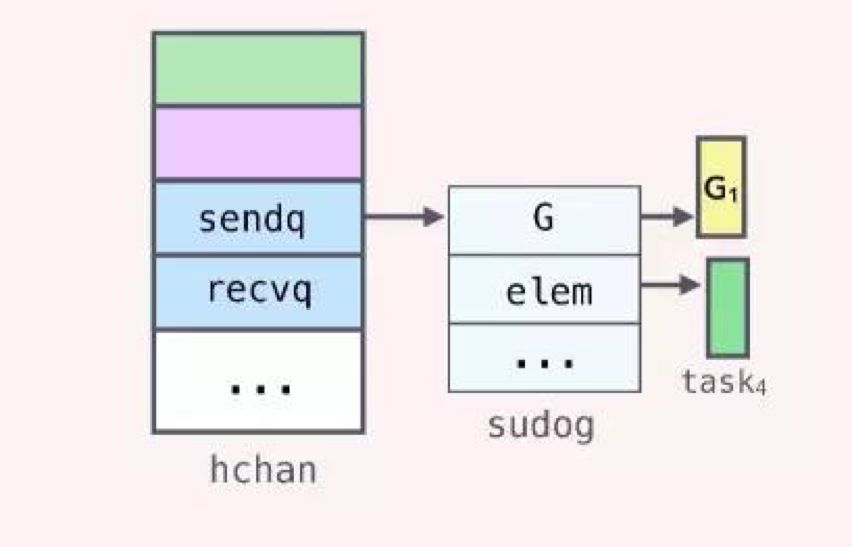
向channel发送数据 2

接收channel中传过来的数据 2 此时我们便可以看出我们的子协程还未执行完，就被阻塞，开始执行主协程了

\*/

****

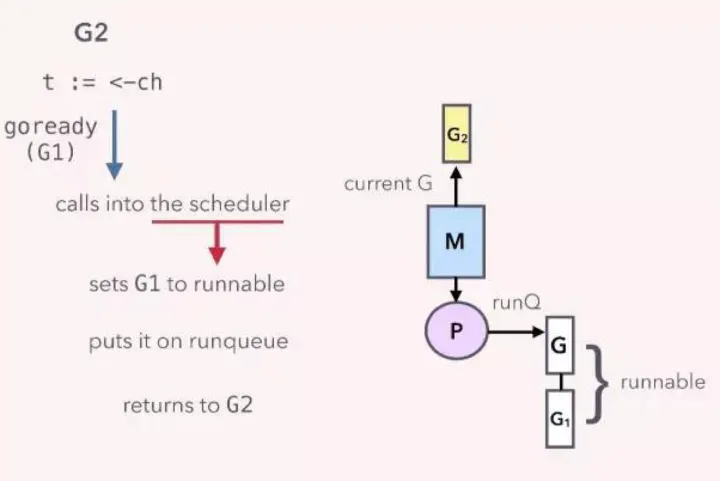
我们的g是goroutine，M是线程，p是与调度相关的context，每一个线程都拥有一个p，每个p维护了一个可以运行的goroutine队列



g1是发送者，g2是接收者

# 先发送在接收

当我们执行send操作时，此时我们向chan发送数据时，由于是无缓冲区，此时我们会包装成studgo结构体，此结构体含有 g1协程的指针和元素的值的引用等，将此类sudog包装完成后会推入waitq队列中，如果是发送者会推送到sendq等待队列 中，然后我们的runtime会进行调度将g1进行等待阻塞



随后我们的runtime会调用g2执行，g2会从chan中接收数据后，（其实这里接收数据的过程其实是个拷贝的过程），会通 知任务调度器，将g1状态设置为runnable，然后加入p的执行队列中，等待线程执行。

# 向接收后发送

当我们g2先执行时，g1后执行时，我们还是首先调度g2，让其在channel中接收数据，因为是无缓冲区，所以也会包装成 一个sudog结构体，同样包含着goroutine2的指针和空接收域等，进而将此sudog放入recvq中，并将g2进行阻塞，在此后 当有个协程g1向channel中发消息时，**此时我们并不会锁住g1，我们直接将g1的数据进行copy到g2即可**，这样直接提高了 效率。

**有缓冲区的chan**： 有缓冲区的chan则是我们设立缓冲区，如果缓冲区有空闲，我们就会直接把数据放入缓冲区，当我们的缓冲区满了之后我们也会变成无缓冲区模式进行操作。

**总结** 其实chan十分好理解，chan即是一个通道，但我们有人发生数据时我们就会判断是否有缓冲区，或缓冲区是否已满， 当然缓冲区满了或者为空我们就会把他放入一个等待队列中，直到有人来取数据我们才再次唤醒他，同理接收者也是如此， 但接收者会判断当前是否有发送着被阻塞，有的话我们直接去读取数据即可。最后我们要知道我们的取数据就是个元素拷贝 过程。

相关论文参考：

[【golang源码分析】chan底层原理——附带读写用户队列的环形缓冲区](https://blog.csdn.net/idwtwt/article/details/104502821)

[深入理解Golang Channel](https://zhuanlan.zhihu.com/p/27917262)

[深入浅出golang的chan](https://blog.csdn.net/weixin_42663840/article/details/81285886)

[Golang channel 源码分析](https://blog.csdn.net/weixin_38418951/article/details/127330082)