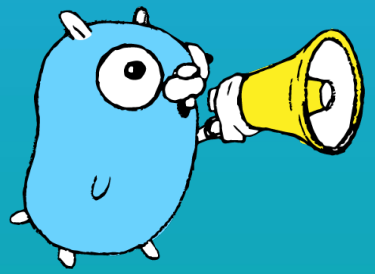
Seoul, May 27 2021

Deep Dive into Channel

방일섭

당근마켓

Speaker



About Channel

Introduction 01

Channel 02

Send to Channel 03

Receive from Channel 04

Close Channel 05

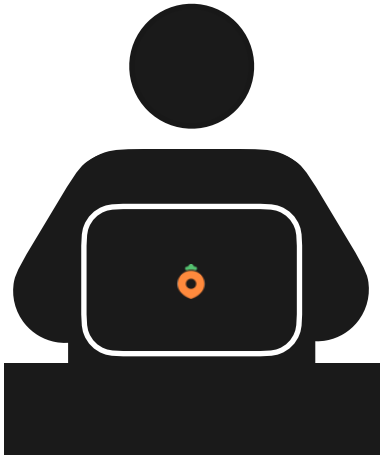
Select Case 06

SECTION ONE

Introduction

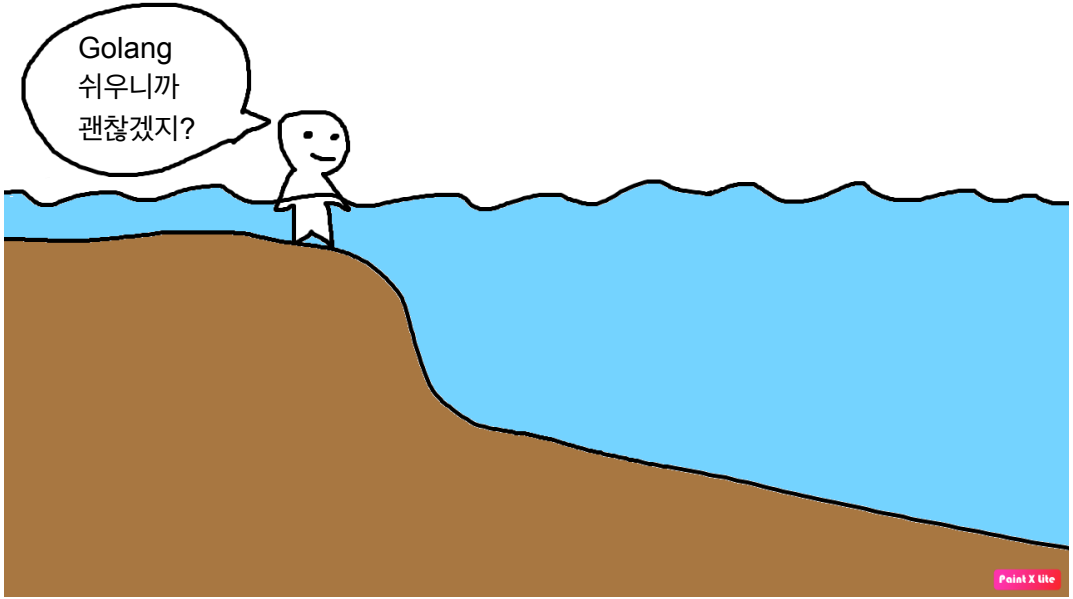
Introduction

와 채널 진짜 편하다



어떻게 만들었는지
궁금한걸?

Introduction



SECTION TWO

Channel

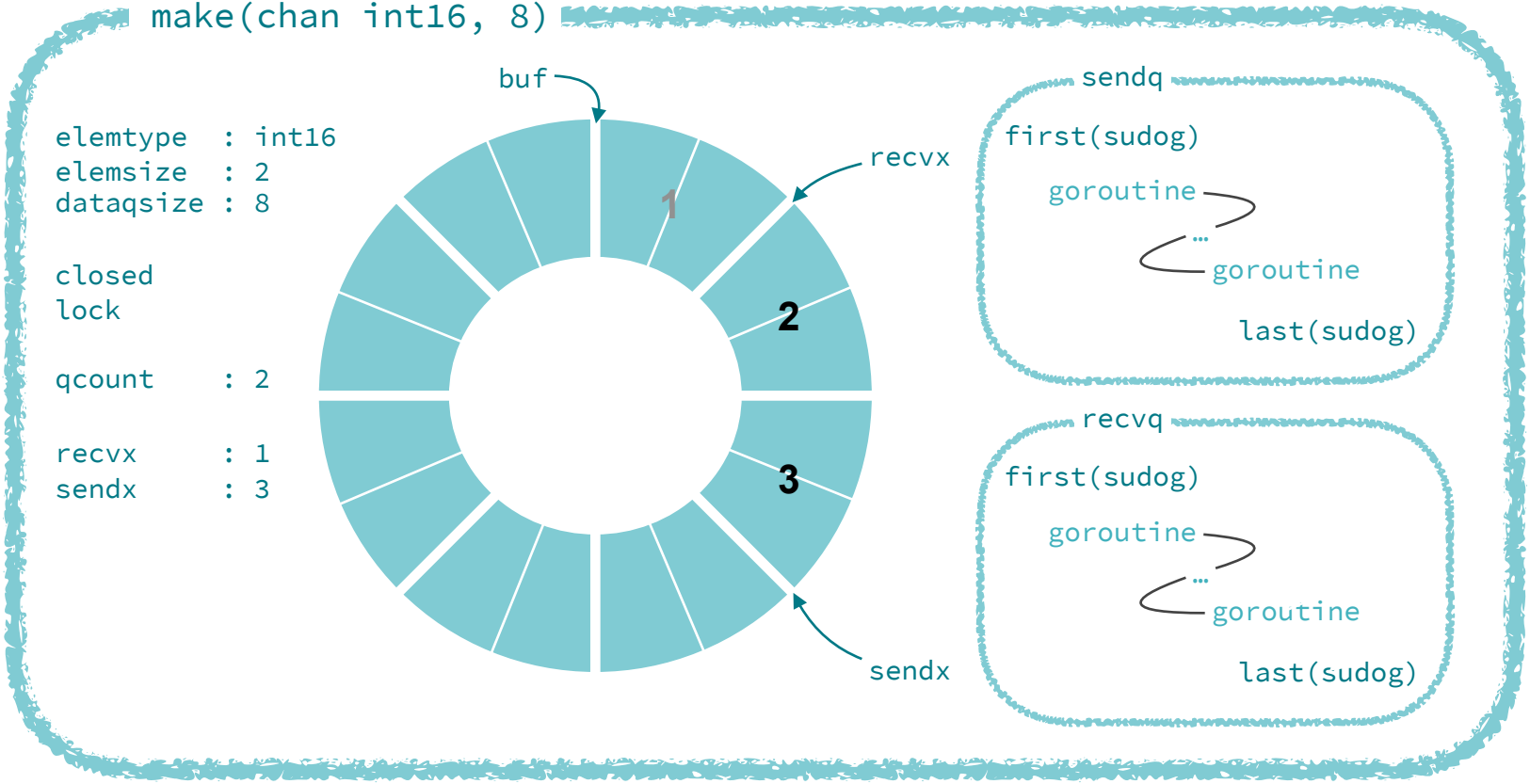
Channel

```go
type hchan struct {
    qcount   uint           // total data in the queue
    dataqsiz uint           // size of the circular queue
    buf      unsafe.Pointer // points to an array of dataqsiz elements
    elemsize uint16
    closed   uint32
    elemtype *_type // element type
    sendx    uint   // send index
    recvx    uint   // receive index
    recvq    waitq  // list of recv waiters
    sendq    waitq  // list of send waiters

    // 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
}
```

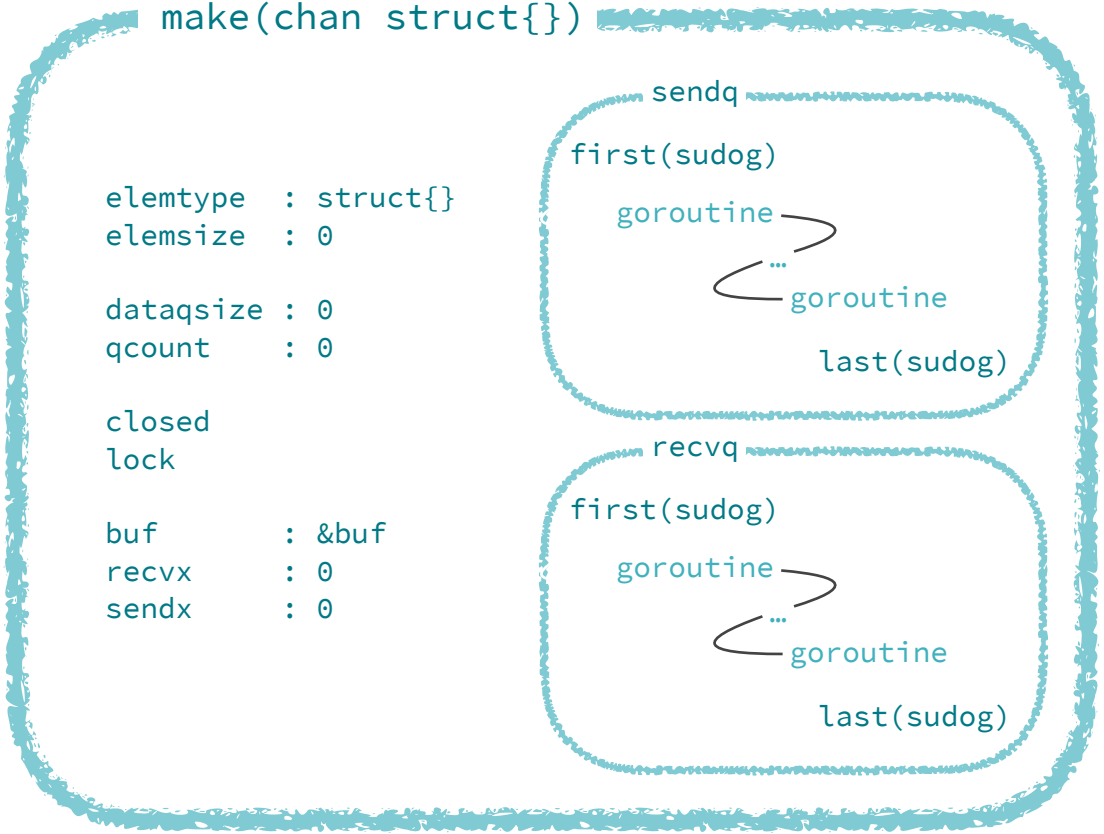
채널의 최대 capacity
= math.MaxUint

채널 타입의 최대 크기
= 1 << 16 - 1
= 65535

Channel


make(chan int16, 8)
elemtype : int16
elemsize : 2
dataqsize : 8
closed
lock
qcount : 2
recvx : 1
sendx : 3
buf
1
2
3
recvx
sendx
sendq
first(sudog)
goroutine
...
goroutine
last(sudog)
recvq
first(sudog)
goroutine
...
goroutine
last(sudog)

Channel


make(chan struct{})
elemtype : struct{}
elemsize : 0
dataqsize : 0
qcount : 0
closed
lock
buf : &buf
recvx : 0
sendx : 0
sendq
first(sudog)
goroutine
…
goroutine
last(sudog)
recvq
first(sudog)
goroutine
…
goroutine
last(sudog)

SECTION THREE

Send to Channel

Send to Channel

Success to send?

```
ch <- v
```

(blocking operation)

true

```
func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool
```

false

```
select {
case ch <- v:
default:
}
```

(non-blocking operation)

Send to Channel

01. Check if channel is available

```
if c == nil {
    if !block {
        return false
    }
    // ...
    throw("unreachable")
}
```

Send to Channel

02. Return if non-block operation, not closed, and channel is full

```
if !block && c.closed == 0 && full(c) {
    return false
}

// func full(c *hchan) bool {
//   if c.dataqsiz == 0 {
//       return c.recvq.first == nil
//   }
//   return c.qcount == c.dataqsiz
// }
```

Send to Channel

03. panic if channel is closed

```
lock(&c.lock)

if c.closed != 0 {
    unlock(&c.lock)
    panic(plainError("send on closed channel"))
}
```

Send to Channel

04. Dequeue receiver and send message if available

```
if sg := c.recvq.dequeue(); sg != nil {
    send(c, sg, ep, func() { unlock(&c.lock) }, 3)
    return true
}
```

Send to Channel

05. Copy to buf if available

```
if c.qcount < c.dataqsiz /* channel buffer space available */ {
    qp := chanbuf(c, c.sendx)
    if raceenabled { ... }
    typedmemmove(c.elemtype, qp, ep)
    c.sendx++
    if c.sendx == c.dataqsiz {
        c.sendx = 0
    }
    c.qcount++
    unlock(&c.lock)
    return true
}
```

Send to Channel

06. Return if non-block operation or parking goroutine

```
if !block {
    unlock(&c.lock)
    return false
}

mysg := /* make sudog */
c.sendq.enqueue(mysg)
// ...
gopark( ... )
```

Send to Channel

07. Return or panic after wake up

```
closed := !mysg.success
/* cleanup sudog */
if closed {
    if c.closed == 0 {
        throw("chansend: spurious wakeup")
    }
    panic(plainError("send on closed channel"))
}
return true
```

Send to Channel

1. Check if channel is valid (non-nil)
2. Check if channel is full
3. Check if channel is closed
4. Check if there is any goroutine waiting for recv data
5. Check if there is any space to save data
6. Enqueue to sendq and parking goroutine
7. Check if operation is succeed after wakeup

SECTION FOUR

Receive from Channel

Receive from Channel

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

```
func chanrecv(c *hchan, ep unsafe.Pointer, block bool) (selected, received bool)
```

```
select {
case <- ctx.Done():
default:
}
```

Receive from Channel

01. Check if channel is available

```
if c == nil {
    if !block {
        return false
    }
    // ...
    throw("unreachable")
}
```

Receive from Channel

02. Return if non-block and channel is empty

```
if !block && empty(c) {
    if atomic.Load(&c.closed) == 0 {
        return
    }
    if empty(c) {
        if raceenabled { ... }
        if ep != nil {
            typedmemclr(c.elemtype, ep)
        }
        return true, false
    }
}
```

Receive from Channel

03. Check if channel is closed

```
lock(&c.lock)
if c.closed != 0 && c.qcount == 0 {
    if raceenabled { ... }
    unlock(&c.lock)
    if ep != nil {
        typedmemclr(c.elemtype, ep)
    }
    return true, false
}
```

Receive from Channel

04. Dequeue sender and receive

```
if sg := c.sendq.dequeue(); sg != nil {
    recv(c, sg, ep, func() { unlock(&c.lock) }, 3)
    return true, true
}
```

Receive from Channel

05. Copy from buffer if available

```
if c.qcount > 0 {
    qp := chanbuf(c, c.recvx)
    if raceenabled { ... }
    if ep != nil {
        typedmemmove(c.elemtype, ep, qp)
    }
    typedmemclr(c.elemtype, qp)
    c.recvx++
    if c.recvx == c.dataqsiz {
        c.recvx = 0
    }
    c.qcount--
    unlock(&c.lock)
    return true, true
}
```

Copy last item to destination from channel ring buffer

Remove last item from channel ring buffer

Receive from Channel

06. Return if non-block operation or parking goroutine

```
if !block {
    unlock(&c.lock)
    return false, false
}

// no sender available: block on this channel.
mysg := /* make sudog */
c.recvq.enqueue(mysg)
// ...
gopark(chanparkcommit, unsafe.Pointer(&c.lock), waitReasonChanReceive, traceEvGoBlockRecv, 2)
```

Receive from Channel

07. Return after wakeup

```
success := mysg.success
/* clean up sudog */

return true, success
```

Receive from Channel

1. Check if channel is valid (non-nil)
2. Check if channel is empty
3. Check if channel is closed
4. Check if there is any goroutine waiting for send data
5. Check if there is any data from ring-buffer
6. Enqueue to recvq and parking goroutine
7. Check if operation is succeed after wakeup

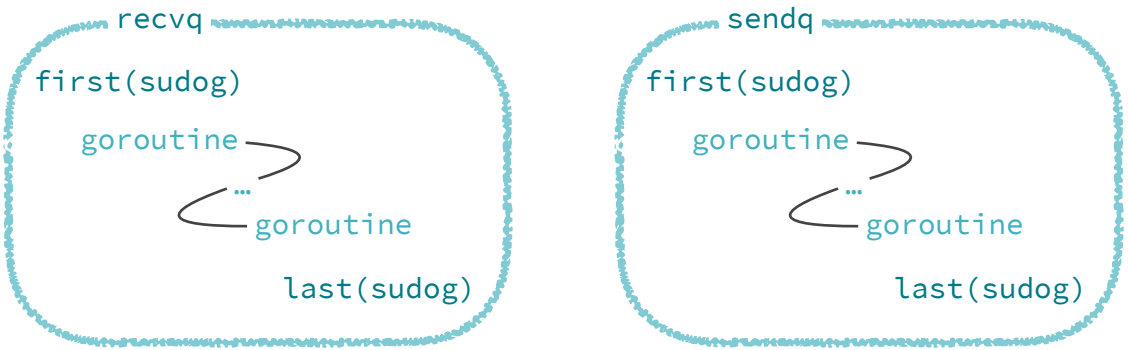
SECTION FIVE

Close Channel

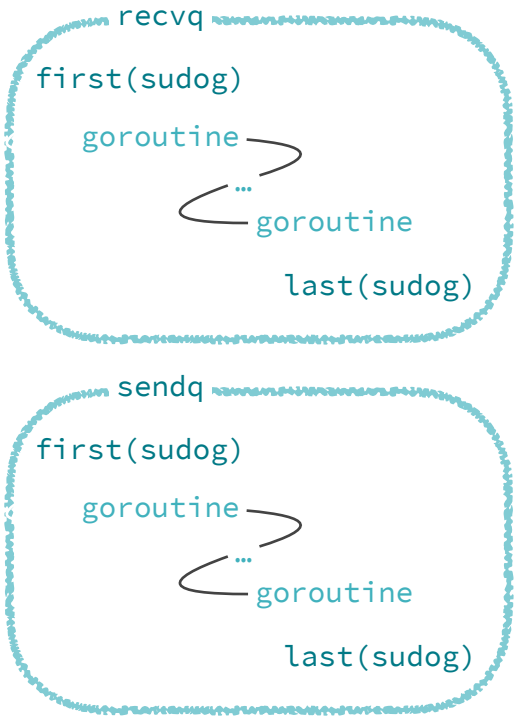
Close Channel

```go
if c == nil {
	panic(plainError("close of nil channel"))
}

lock(&c.lock)
if c.closed != 0 {
	unlock(&c.lock)
	panic(plainError("close of closed channel"))
}

if raceenabled { ... }
c.closed = 1
```

Close Channel

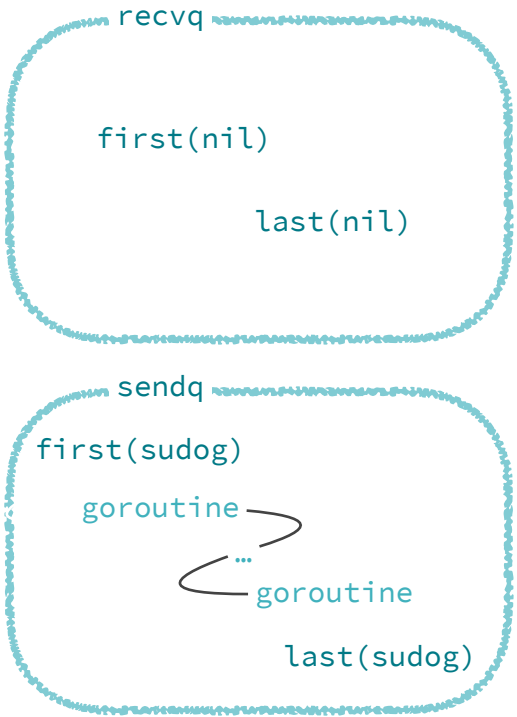


Close Channel



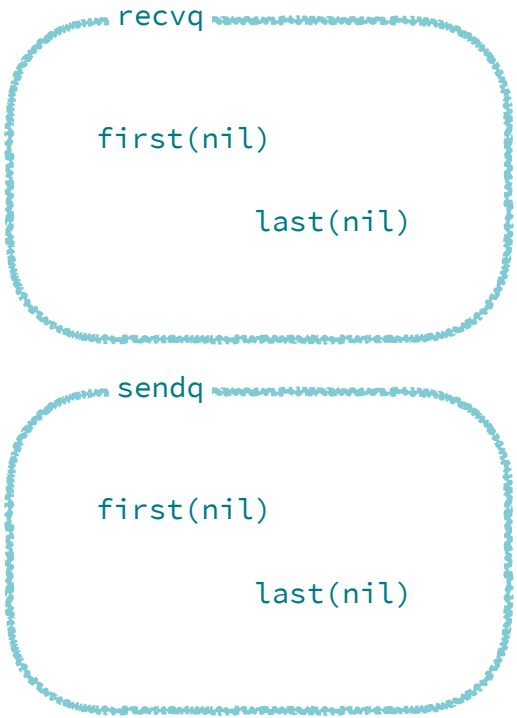

```
var glist gList
```

Close Channel




```go
var glist gList

for {
    sg := c.recvq.dequeue()
    if sg == nil {
        break
    }
    // ...
    gp := sg.g
    sg.success = false
    if raceenabled { ... }
    glist.push(gp)
}
```

Close Channel




```go
var glist gList

for {
    sg := c.sendq.dequeue()
    if sg == nil {
        break
    }
    // ...
    gp := sg.g
    sg.success = false
    if raceenabled { ... }
    glist.push(gp)
}
```

Close Channel

```
for !glist.empty() {
    gp := glist.pop()
    gp.schedlink = 0
    goready(gp, 3)
}
```

SECTION SIX

Select Case

Select Case

```go
func selectgo(cas0 *scase, order0 *uint16, pc0 *uintptr, nsends, nrecvs int, block bool) (int, bool)
```

Select Case

```go
select {
case ch1 <- v1:
    // ...
case v2, ok := <-ch2:
    // ...
}
```

```go
type scase struct {
    c    *hchan          // chan
    elem unsafe.Pointer // data element
}
```

Select Case

```
scases := []scase{
        { case ch1 <- v1 },
        { case v2, ok := <-ch2 },
}
```

Select Case

```
scases := []scase{
        { case ch1 <- v1 },
        { case v2, ok := <-ch2 },
}
```

Lock all hchans

Select Case

```
scases := []scase{
    { case ch1 <- v1 },
    { case v2, ok := <-ch2 },
}
```

Do as non-block

sendchan(…)

Select Case

```
scases := []scase{
    { case ch1 <- v1 },
    { case v2, ok := <-ch2 },
}
```

Do as non-block

recvchan(…)

Select Case

```
scases := []scase{
     { case ch1 <- v1 },
     { case v2, ok := <-ch2 },
}

mysg := /* make sudog */
```

Make sudog

Select Case

```
scases := []scase{
    { case ch1 <- v1 },
    { case v2, ok := <-ch2 },
}
```

```
ch1.sendq.enqueue(mysg)
```

```
mysg := /* make sudog */
```

Select Case

```
scases := []scase{
    { case ch1 <- v1 },
    { case v2, ok := <-ch2 },
}
```

ch2.recvq.enqueue(mysg)

```
mysg := /* make sudog */
```

Select Case

```
scases := []scase{
    { case ch1 <- v1 },
    { case v2, ok := <-ch2 },
}
```

```
mysg := /* make sudog */
```

```
gopark( ... )
```

gopark 함수에서 모든 channel 을 unlock 함

Select Case

```
scases := []scase{
    { case ch1 <- v1 },
    { case v2, ok := <-ch2 },
}
```

Wake up!

```
mysg := /* make sudog */
```

Select Case

```
scases := []scase{
        { case ch1 <- v1 },
        { case v2, ok := <-ch2 },
}

mysg := /* make sudog */
```

Lock all other hchans

Select Case

```
scases := []scase{
    { case ch1 <- v1 },
    { case v2, ok := <-ch2 },
}

mysg := /* make sudog */
```

ch1.sendq.dequeue(mysg)

QnA

Thank You!

시간 나시면 코드 읽어보세요! 읽기 쉬워요 ㅎㅎ ~~나만 당할 수 없지~~
https://github.com/golang/go/blob/master/src/runtime/chan.go
https://github.com/golang/go/blob/master/src/runtime/select.go