Konichiwa Golang chan

Golang Warsaw #31
27 Nov 2019

Tomasz Grodzki
Co-Founder &
Developer,
AlphaSOC
tomasz@alphasoc.com

November 28, 2019

Channel type

[[https://golang.org/ref/spec#Channel_types][golang.org/ref/spec]] says:
A channel provides a mechanism for concurrently executing functions to communicate by sending and receiving values of a specified element type. Channels act as first-in-first-out queues (FIFO) – the values are received in the order sent.

Channel type

Channels can be directional or bidirectional:

```
chan T          // can be used to send and receive values of type T
chan<- float64  // can only be used to send float64s
<-chan int      // can only be used to receive ints
```

A (bidirectional) channel may be constrained only to send or only to receive:

```
c := make(chan int)
var r <-chan int = c
var s chan<- int = c
```

Channel type

A new, initialized channel value can be made using the built-in function make.
Arguments: channel type and capacity (optional).

```
make(chan int) // same as make(chan int, 0)
make(chan int, 100)
```

The capacity, in number of elements, sets the size of the buffer in the channel.
If the capacity is zero (or absent), the channel is unbuffered.
Otherwise, the channel is buffered.

Channel type

For **unbuffered** channel communication succeeds only when both a sender and receiver are ready.
For **buffered** channel communication succeeds without blocking if the buffer is not full (sends) or not empty (receives).

Channel type

A single channel may be used by any number of goroutines without further synchronization in:

- send statements
- receive operations
- calls to the built-in functions: `cap` and `len`

(Note that `close` is not on the list.)

send

```
ch <- 3  // send value 3 to channel ch
```

- expressions are evaluated before communication begins
- blocks until send can proceed
- can proceed if there is room in the buffer or a receiver is ready (unbuffered)
- send on closed channel == panic
- send on nil channel blocks forever

receive

```
v := <-ch
v, ok := <-ch // ok == true if channel not closed
```

- blocks until a value is available
- receiving from a nil channel blocks forever
- receiving from closed channel returns type's zero value and false

cap/len

```go
package main

import "fmt"

func main() {
        c := make(chan string, 10)
        c <- "konichiwa"

        fmt.Println("cap:", cap(c)) // 10
        fmt.Println("len:", len(c)) // 1
}
```

close

Close indicates no more values will be sent on the channel.
Closing an already closed channel causes a panic.

```go
c := make(chan int, 2)
c <- 10
c <- 20
close(c)

for n := 0; n < 5; n++ {
        v, ok := <-c
        fmt.Println(v, ok)
}

close(c) // panic!
```

for-range

A ”for” statement with a ”range” clause iterates through all entries of an array, slice, string or map, or **values received on a channel.**

```
c := make(chan time.Time, 10)

// Fill the channel
for len(c) < cap(c) {
        c <- time.Now()
}
close(c) // no more items

// Read values from the channel
for v := range c {
        fmt.Println(v.Format(time.RFC3339Nano))
}
```

Give me some REAL code!

Prime (number) counter

Prime counter

Our program will count prime numbers <= N.
One routine sending possible candidates.
Multiple routines checking if candidates are prime.
One routine collecting results.
Two channels:

- incoming jobs (possible candidates)
- results from workers

```
#                                    / isPrime? \
#                    candidates ===> - isPrime? - ===> results
#                                    \ isPrime? /
```

Prime counter

```
package main

import (
        "fmt"
        "math"
        "time"
)

func isPrime(n int) bool {
        for m := 2; m <= int(math.Sqrt(float64(n))); m++ {
                if n%m == 0 {
                        return false
                }
        }
        return true
}

// ...
```

Prime counter

Send candidates:

```
func main() {
        start := time.Now()

        check := make(chan int, 100)
        results := make(chan int)

        // Send candidates (all odd numbers in range)
        go func() {
                for n := 3; n < 2e6; n += 2 {
                        check <- n // HL
                }
                close(check) // HL
        }()

        // ...
```

Prime counter

Check candidates:

```
// Run workers
numOfWorkers := 4

for n := 0; n < numOfWorkers; n++ {
        go func() {
                found := 0
                for n := range check { // HL
                        if isPrime(n) {
                                found++
                        }
                }
                results <- found // HL
        }()
}

// ...
```

Prime counter

Finish up:

```
        // Collect the results
        found := 0
        for n := 0; n < numOfWorkers; n++ {
                found += <-results // HL
        }

        fmt.Printf("found %d prime numbers in %s\n", found, time.Since(
}
```

Showing primes

Showing primes

Unknown number of results, different closing strategy.

```go
var wg sync.WaitGroup

for n := 0; n < numOfWorkers; n++ {
        wg.Add(1) // HL
        go func() {
                defer wg.Done() // HL
                for n := range check {
                        if isPrime(n) {
                                primes <- n // HL
                        }
                }
        }()
}

go func() {
        wg.Wait()     // HL
        close(primes) // HL
}()
```

Showing primes

Now we can loop over the results:

```
for n := range primes {
        fmt.Printf("prime: %d\n", n)
}
```

Semaphore

Semaphore

```
semaphore := make(chan struct{}, 2) // HL

task := func() {
        semaphore <- struct{}{}        // acquire // HL
        defer func() { <-semaphore }() // release // HL

        // some work here
        time.Sleep(time.Second)
}

for n := 0; n < 10; n++ {
        go func(id int) {
                log.Printf("[%d]: starting...\n", id)
                for {
                        task()
                        log.Printf("[%d]: task done!\n", id)
                }
        }(n)
}
```

Semaphore (non-blocking)

```go
semaphore := make(chan struct{}, 2)

task := func() bool {
        // non-blocking acquire // HL
        select { // HL
        case semaphore <- struct{}{}: // HL
                defer func() { <-semaphore }() // HL
        default: // HL
                // channel full // HL
                return false // HL
        } // HL

        // some work here
        time.Sleep(time.Second)
        return true
}
```

Semaphore (non-blocking)

```
for n := 0; n < 10; n++ {
        go func(id int) {
                for {
                        if task() {
                                log.Printf("[%d]: task done!\n"
                        } else {
                                log.Printf("[%d]: would block,
                                time.Sleep(time.Second / 2)
                        }
                }
        }(n)
}
```

Real semaphore

godoc.org/golang.org/x/sync/semaphore

Scheduler

Scheduler

Send tasks into the channel.
Receive tasks from the channel according to some priority.
Priority queue instead of simple FIFO.
Two channels: incoming and outgoing.

```
#
#                       tasks ===> scheduler ===> tasks (reordered)
#
```

Scheduler

```go
func main() {
        in := make(chan int)  // incoming tasks
        out := make(chan int) // outgoing tasks (reordered)

        go func() {
                defer close(in)
                for n := 0; n < 10; n++ {
                        in <- rand.Intn(1000)
                }
        }()

        go func() {
                defer close(out)
                scheduler(in, out)
        }()

        for n := range out {
                fmt.Println(n)
                time.Sleep(time.Millisecond) // some work
        }
}
```

Scheduler

```
type queue []int

// Add number to the queue, keep in low-to-high priority order
func (q *queue) Add(n int) {
        *q = append(*q, n)
        sort.Ints(*q)
}

// Top returns item with the highest priority
func (q queue) Top() int {
        if len(q) > 0 {
                return q[len(q)-1]
        }
        return 0
}

// Pop removes item with the highest priority
func (q *queue) Pop() {
        *q = (*q)[:len(*q)-1]
}
```

Scheduler

```go
func scheduler(in <-chan int, out chan<- int) {
        var q queue

        for len(q) > 0 || in != nil {
                var xout chan<- int // = out if something to send
                if len(q) > 0 {
                        xout = out
                }

                select {
                case xout <- q.Top():
                        q.Pop()
                case n, ok := <-in:
                        if ok {
                                q.Add(n)
                        } else {
                                // channel closed, no more items
                                in = nil
                        }
                }
        }
}
```

Homework

- broadcasting (one-to-many)
- asynchronous function calls