Channels - Part 2. The most Useful package in Go: sync

Session 16

Golang course by Exadel

05 Dec 2022

Sergio Kovtunenko
Lead backend developer, Exadel

Agenda

- ▶️ Send (to channel) statement details
- ▶️ Receive (from channel) operator
- ▶️ for-range loop with
- ▶️ Goroutine gotchas
- ▶️ Select statement
- ▶️ Blocking vs. non-blocking flows
- ▶️ The most useful package: sync
- ▶️ Advanced concurrency patterns
- ▶️ Next time...

Goroutines and channels - Part 2

Send (to channel) statement

- ✓ A send statement *sends a value on a channel*.
- ✓ The channel expression must be:
  - of channel type **AND**
  - the channel direction must permit send operations **AND**
  - the type of the value to be sent must be assignable to the channel's element type.
- ✓ Both the *channel* and the value expression are evaluated before communication begins.
- ✓ Communication blocks until the send can proceed.
- ✓ Potential issue with **mutable** data (pointers to data) send into channel.

- ✓ Restrictions:
  - A send on an unbuffered channel can proceed **if a receiver is ready.**
  - A send on a buffered channel can proceed **if there is room in the buffer.**
  - A send on a closed channel proceeds by **causing a run-time panic.**
  - A send on a nil channel **blocks forever.**

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

Receive (from channel) operator: recap from the last session

- ✓ For an operand **ch** of *channel type*, the value of the receive operation **<-ch** is the value *received from the channel* ***ch***.
- ✓ The **channel direction** must:
  - permit receive operations
  - and the type of the receive operation is the element type of the channel.
- ✓ The expression blocks until a value is available.
- ✓ Receiving from a **nil** channel **blocks forever**.
- ✓ A **receive operation** on a **closed channel** can always proceed immediately
  - yielding the element type's zero value after any previously sent values have been received.

```
v1 := <-ch
v2 = <-ch
f(<-ch)
<-strobe  // wait until clock pulse and discard received value
```

Receive (from channel) operator: comma-ok check

- ✓ A **receive expression** used in an assignment or initialization of the special form
  - yields an additional untyped boolean result reporting whether the communication succeeded.
  - The value of **ok** is **true** if the value received was delivered by a successful send operation to the channel
    - or **false** if it is a zero value generated because the channel is closed and empty.
  - Example:

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

For-range loop over channel

- ✓ For *channels*, the *iteration values* produced are the successive values sent on the channel until the channel is closed.
- ✓ If the channel is nil, the range expression **blocks forever.**

```
package main
import "fmt"
func FibonacciProducer(ch chan int, count int) {
     n2, n1 := 0, 1
     for count >= 0 {
          ch <- n2
          count--
          n2, n1 = n1, n2+n1
     }
     close(ch)
}
func main() {
     ch := make(chan int)
     go FibonacciProducer(ch, 10)
     idx := 0
     // To break such iteration channel needs to be closed explicitly.
     // Otherwise range would block forever in the same way as for nil channel.
     for num := range ch {
          fmt.Printf("F(%d): \t%d\n", idx, num)
          idx++
     }
}
```

For-range loop over channel - example

▶️ Example: `code/chanexample/01_chan_iteration_test.go`

Blocked Goroutines and Resource Leaks: problem statement

- ✓ Blocked Goroutines and Resource Leaks:
  - Fetching the first result from a number of targets is one of them (goroutine leak in this example):
    - To avoid the leaks you need to make sure all goroutines exit.

```
func First(query string, replicas ...Search) Result {
    c := make(chan Result) // Unbuffered (synchronous) channel!

    searchReplica := func(i int) { c <- replicas[i](query) }
    for i := range replicas {
        go searchReplica(i)
    }

    return <-c // This means that only the first goroutine returns.
    // All other goroutines are stuck trying to send their results.
    // This means if you have more than one replica each call will leak resources.
}
```

Source: "50 Shades of Go: Traps, Gotchas, and Common Mistakes for New Golang Devs"

(http://devs.cloudimmunity.com/gotchas-and-common-mistakes-in-go-golang/index.html#blocked_goroutines)

Blocked Goroutines and Resource Leaks: solution 1

▶️ Solution 1:

✓ **Solution 1:** Fix using a buffered result channel big enough to hold all results:

```
func First(query string, replicas ...Search) Result {
    // One potential solution is to use a buffered result channel big enough to hold all results:
    c := make(chan Result, len(replicas))

    searchReplica := func(i int) { c <- replicas[i](query) }
    for i := range replicas {
        go searchReplica(i)
    }
    return <-c
}
```

Blocked Goroutines and Resource Leaks: solution 2

▶️ Solution 2:

✓ **Solution 2**: Fix using a select statement with a `default case` and a buffered result channel that can hold one value.

- The `default case` ensures that the goroutines don't get stuck even when the result channel can't receive messages:

```
func First(query string, replicas ...Search) Result {
    c := make(chan Result, 1) // Buffered channel with size 1
    searchReplica := func(i int) {
        select {
        case c <- replicas[i](query):
        default:
            // Will be executed for the rest of goroutines blocked on sending values!
        }
    }
    for i := range replicas {
        go searchReplica(i)
    }
    return <-c
}
```

Blocked Goroutines and Resource Leaks: solution 3

▶️ Solution 3:

✓ **Solution 3**: Fix using a special cancellation channel to interrupt the workers:

```
func First(query string, replicas ...Search) Result {
    c := make(chan Result)

    done := make(chan struct{}) // Special cancellation channel
    defer close(done)

    searchReplica := func(i int) {
        select {
        case c <- replicas[i](query):
        case <- done:
            // Will be executed for the rest of goroutines blocked on sending values!
        }
    }
    for i := range replicas {
        go searchReplica(i)
    }

    return <-c
}
```

`select` statement: general information

- ✓ A "select" statement chooses which of a set of possible *send* or *receive* operations will proceed.
  - It looks similar to a "switch" statement but with the cases *all referring to communication operations*.

- ✓ Since communication on nil channels can never proceed:
  - select with only nil channels and no default case blocks forever.
  - but using nil chans will disable a select case.

`select` statement: execution flow details

✓ Execution of a "select" statement proceeds in several steps:

- For all the cases in the statement, the channel operands of **receive operations and the channel** and **right-hand-side expressions of send statements** are evaluated exactly once, in source order, upon entering the "select" statement.
  - The result is a set of channels to **receive from** or **send to**, and the corresponding values to send.
  - Any side effects in that evaluation will occur irrespective of which (if any) communication operation is selected to proceed.
    - Expressions on the left-hand side of a *"Receive Statement"* with a short *variable declaration* or *assignment* are not yet evaluated.
- If ***one or more of the communications can proceed***, a single one that can proceed is chosen via a uniform pseudo-random selection.
  - Otherwise, if there is a **default case**, that case is chosen.
    - If there is no **default case**, the **"select" statement blocks** until at least one of the communications can proceed.
- Unless the selected case is the **default case**, the respective communication operation is executed.
- If the selected case is a *"Receive Statement"* (with a short variable declaration or an assignment), the left-hand side expressions are evaluated and the received value (or values) are assigned.
- The statement list of the selected case is executed.

`select` statement: example

✓ Example:

```
func demo() {
    var a []int
    var c, c1, c2, c3, c4 chan int
    var i1, i2 int
    select {
    case i1 = <-c1:
        print("received ", i1, " from c1\n")
    case c2 <- i2:
        print("sent ", i2, " to c2\n")
    case i3, ok := (<-c3):  // same as: i3, ok := <-c3
        if ok {
            print("received ", i3, " from c3\n")
        } else {
            print("c3 is closed\n")
        }
    case a[f()] = <-c4:
    // same as:
    // case t := <-c4
    //   a[f()] = t
    default:
        print("no communication\n")
    }

    for {  // send random sequence of bits to c
        select {
        case c <- 0:  // note: no statement, no fallthrough, no folding of cases
        case c <- 1:
        }
    }

    select {}  // block forever
}
```

`sync` package

Overview of `sync` package

▶️ sync package documentation (https://pkg.go.dev/sync)

▶️ Package **sync** provides basic synchronization primitives such as mutual exclusion locks.

▶️ Other than the **Once** and **WaitGroup** types, *most are intended for use by low-level library routines*.

- *Higher-level synchronization is better done via channels and communication*.
- Values containing the types defined in this package **MUST not be copied**.

`sync` package: sync.Mutex

▶️ Type: sync.Mutex (https://pkg.go.dev/sync#Mutex)

▶️ It allows a **mutual exclusion** on a shared resource (no simultaneous access):

```
mutex := &sync.Mutex{}

mutex.Lock()
mutex.Unlock()
```

▶️ A `sync.Mutex` **cannot be copied**!!!

`sync` package: sync.RWMutex

▶️ Type: sync.RWMutex (https://pkg.go.dev/sync#RWMutex).

▶️ API usage example:

```
mutex := &sync.RWMutex{}

mutex.Lock()
mutex.Unlock()

mutex.RLock()
mutex.RUnlock()
```

▶️ A `sync.RWMutex` allows either **at least** one reader or **exactly** one writer whereas a `sync.Mutex` allows **exactly one reader or writer**.

▶️ A `sync.RWMutex` should rather be used when we have **frequent reads** and **in**frequent writes.

▶️ A `sync.RWMutex` **cannot be copied**!!!

`sync` package: sync.WaitGroup

▶️ Type sync.WaitGroup (https://pkg.go.dev/sync#WaitGroup)

▶️ Provides an idiomatic way for a goroutine to wait for the **completion of a collection of goroutines**.

▶️ `sync.WaitGroup` holds an internal counter.

- If this counter is equal to 0, the Wait() method returns immediately.
- Otherwise, it is blocked until the counter is 0.

```
wg := &sync.WaitGroup{}
for i := 0; i < 8; i++ {
  wg.Add(1)
  go func() {
    // Do something...
    wg.Done()
  }()
}
wg.Wait() // Continue execution afterwards...
```

`sync` package: sync.Once

▶️ To execute a code only once: sync.Once (https://pkg.go.dev/sync#Once)

▶️ Example:

```
once := &sync.Once{}

for i := 0; i < 4; i++ {
    i := i

    go func() {
        once.Do(func() {
            fmt.Printf("first %d\n", i)
        })
    }()
}
```

`sync` package: rest of APIs

▶️ Concurrent pool, in charge to hold safely a set of objects: sync.Pool (https://pkg.go.dev/sync#Pool).

- `Get() interface{}` to retrieve an element;
- `Put(interface{})` to add an element;

▶️ Concurrent map implementation: sync.Map (https://pkg.go.dev/sync#Map)

▶️ The less frequently used primitive: sync.Cond (https://pkg.go.dev/sync#Cond)

- It is used to emit a signal (one-to-one) or broadcast a signal (one-to-many) to goroutine(s).

Advanced concurrency patterns

General info about advanced patterns: fan-out, fan-in

▶️ Fan-out:

- **Multiple functions can read from the same channel** until that channel is closed;
- This provides a way to distribute work amongst a group of workers to parallelize CPU use and I/O.

▶️ Fan-in:

- A function can read from multiple inputs and proceed until all are closed by multiplexing the input channels onto a single channel that's closed when all the inputs are closed.

Generators

▶️ Example code: `code/02_advancedexample/01_generate_data_test.go`

▶️ Remember this code:

```
func asChan(nums ...int) <-chan int {
    ch := make(chan int)
    go func() {
        defer close(ch)
        for _, val := range nums {
            ch <- val
        }
    }()
    return ch
}
```

Merging channels together

▶️ Merge two (or N known channels):

- Example code: `code/02_advancedexample/02_merge_two_channels_test.go`

▶️ Merge many (unknown) number of channels of same type:

- Example code: `code/02_advancedexample/03_merge_many_channels_test.go`

Homework:

- ▶️ Recap: Go Concurrency Patterns: Pipelines and cancellation (https://go.dev/blog/pipelines)

Next session...

▶️ Session17:

HTTP servers and routers

- standard package `context`
- Typical HTTP Server
- HTTP server settings
- Testing HTTP server

Thank you

Golang course by Exadel

05 Dec 2022

Sergio Kovtunenko
Lead backend developer, Exadel
skovtunenko@exadel.com (mailto:skovtunenko@exadel.com)
https://github.com/skovtunenko (https://github.com/skovtunenko)
@realSKovtunenko (http://twitter.com/realSKovtunenko)