Advanced Go Concurrency Patterns

Sameer Ajmani
Google

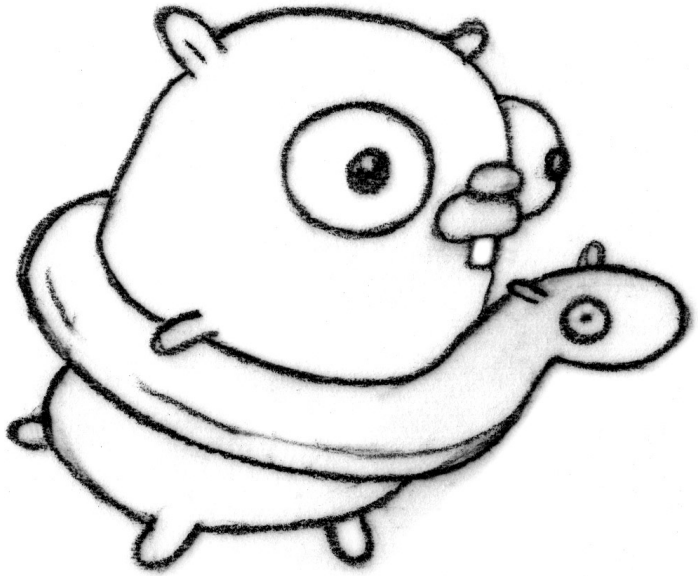
Video

This talk was presented at Google I/O in May 2013.

Watch the talk on YouTube (https://www.youtube.com/watch?v=QDDwwePbDtw)

2

Get ready



3

Go supports concurrency

In the language and runtime, not a library.

This changes how you structure your programs.

Goroutines and Channels

Goroutines are independently executing functions in the same address space.

```
go f()
go g(1, 2)
```

Channels are typed values that allow goroutines to synchronize and exchange information.

```
c := make(chan int)
go func() { c <- 3 }()
n := <-c
```

For more on the basics, watch Go Concurrency Patterns (Pike, 2012) (http://talks.golang.org/2012/concurrency.slide#1) 5.

Example: ping-pong

```
type Ball struct{ hits int }

func main() {
    table := make(chan *Ball)
    go player("ping", table)
    go player("pong", table)

    table <- new(Ball) // game on; toss the ball
    time.Sleep(1 * time.Second)
    <-table // game over; grab the ball
}

func player(name string, table chan *Ball) {
    for {
        ball := <-table
        ball.hits++
        fmt.Println(name, ball.hits)
        time.Sleep(100 * time.Millisecond)
        table <- ball
    }
}
```

Run

6

Deadlock detection

```
type Ball struct{ hits int }

func main() {
    table := make(chan *Ball)
    go player("ping", table)
    go player("pong", table)

    // table <- new(Ball) // game on; toss the ball
    time.Sleep(1 * time.Second)
    <-table // game over; grab the ball
}

func player(name string, table chan *Ball) {
    for {
        ball := <-table
        ball.hits++
        fmt.Println(name, ball.hits)
        time.Sleep(100 * time.Millisecond)
        table <- ball
    }
}
```

Run

7

Panic dumps the stacks

```
type Ball struct{ hits int }

func main() {
    table := make(chan *Ball)
    go player("ping", table)
    go player("pong", table)

    table <- new(Ball) // game on; toss the ball
    time.Sleep(1 * time.Second)
    <-table // game over; grab the ball

    panic("show me the stacks")
}

func player(name string, table chan *Ball) {
    for {
        ball := <-table
        ball.hits++
        fmt.Println(name, ball.hits)
        time.Sleep(100 * time.Millisecond)
        table <- ball
    }
}
```

Run

8

It's easy to go, but how to stop?

Long-lived programs need to clean up.

Let's look at how to write programs that handle communication, periodic events, and cancellation.

The core is Go's `select` statement: like a `switch`, but the decision is made based on the ability to communicate.

```
select {
case xc <- x:
    // sent x on xc
case y := <-yc:
    // received y from yc
}
```

9

Example: feed reader

My favorite feed reader disappeared. I need a new one.

Why not write one?

Where do we start?

10

Find an RSS client

Searching godoc.org (http://godoc.org) for **"rss"** turns up several hits, including one that provides:

```
// Fetch fetches Items for uri and returns the time when the next
// fetch should be attempted.  On failure, Fetch returns an error.
func Fetch(uri string) (items []Item, next time.Time, err error)

type Item struct{
    Title, Channel, GUID string // a subset of RSS fields
}
```

But I want a stream:

```
<-chan Item
```

And I want multiple subscriptions.

Here's what we have

```
type Fetcher interface {
    Fetch() (items []Item, next time.Time, err error)
}

func Fetch(domain string) Fetcher {...} // fetches Items from domain
```

12

Here's what we want

```
type Subscription interface {
    Updates() <-chan Item // stream of Items
    Close() error         // shuts down the stream
}

func Subscribe(fetcher Fetcher) Subscription {...} // converts Fetches to a stream

func Merge(subs ...Subscription) Subscription {...} // merges several streams
```

13

Example

```
func main() {
    // Subscribe to some feeds, and create a merged update stream.
    merged := Merge(
        Subscribe(Fetch("blog.golang.org")),
        Subscribe(Fetch("googleblog.blogspot.com")),
        Subscribe(Fetch("googledevelopers.blogspot.com")))

    // Close the subscriptions after some time.
    time.AfterFunc(3*time.Second, func() {
        fmt.Println("closed:", merged.Close())
    })

    // Print the stream.
    for it := range merged.Updates() {
        fmt.Println(it.Channel, it.Title)
    }

    panic("show me the stacks")
}
```

Run

Subscribe

`Subscribe` creates a new `Subscription` that repeatedly fetches items until `Close` is called.

```
func Subscribe(fetcher Fetcher) Subscription {
    s := &sub{
        fetcher: fetcher,
        updates: make(chan Item), // for Updates
    }
    go s.loop()
    return s
}

// sub implements the Subscription interface.
type sub struct {
    fetcher Fetcher   // fetches items
    updates chan Item // delivers items to the user
}

// loop fetches items using s.fetcher and sends them
// on s.updates.  loop exits when s.Close is called.
func (s *sub) loop() {...}
```

15

Implementing Subscription

To implement the `Subscription` interface, define `Updates` and `Close`.

```
func (s *sub) Updates() <-chan Item {
    return s.updates
}
```

```
func (s *sub) Close() error {
    // TODO: make loop exit
    // TODO: find out about any error
    return err
}
```

16

What does loop do?

- periodically call `Fetch`
- send fetched items on the `Updates` channel
- exit when `Close` is called, reporting any error

Naive Implementation

```
    for {
        if s.closed {
            close(s.updates)
            return
        }
        items, next, err := s.fetcher.Fetch()
        if err != nil {
            s.err = err
            time.Sleep(10 * time.Second)
            continue
        }
        for _, item := range items {
            s.updates <- item
        }
        if now := time.Now(); next.After(now) {
            time.Sleep(next.Sub(now))
        }
    }
```

Run

```
func (s *naiveSub) Close() error {
    s.closed = true
    return s.err
}
```

18

Bug 1: unsynchronized access to s.closed/s.err

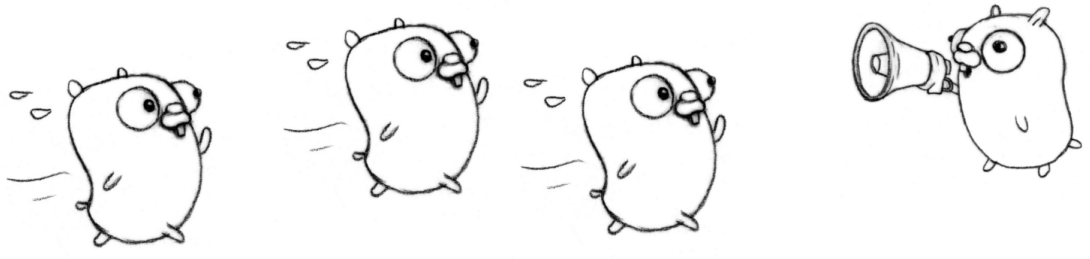
```
    for {
        if s.closed {
            close(s.updates)
            return
        }
        items, next, err := s.fetcher.Fetch()
        if err != nil {
            s.err = err
            time.Sleep(10 * time.Second)
            continue
        }
        for _, item := range items {
            s.updates <- item
        }
        if now := time.Now(); next.After(now) {
            time.Sleep(next.Sub(now))
        }
    }
```

```
func (s *naiveSub) Close() error {
    s.closed = true
    return s.err
}
```

19

Race Detector

```
go run -race naivemain.go
```



```
    for {
        if s.closed {
            close(s.updates)
            return
        }
        items, next, err := s.fetcher.Fetch()
        if err != nil {
            s.err = err
```

Run

```
func (s *naiveSub) Close() error {
    s.closed = true
    return s.err
}
```

20

Bug 2: time.Sleep may keep loop running

```
for {
    if s.closed {
        close(s.updates)
        return
    }
    items, next, err := s.fetcher.Fetch()
    if err != nil {
        s.err = err
        time.Sleep(10 * time.Second)
        continue
    }
    for _, item := range items {
        s.updates <- item
    }
    if now := time.Now(); next.After(now) {
        time.Sleep(next.Sub(now))
    }
}
```

21

Bug 3: loop may block forever on s.updates

```
    for {
        if s.closed {
            close(s.updates)
            return
        }
        items, next, err := s.fetcher.Fetch()
        if err != nil {
            s.err = err
            time.Sleep(10 * time.Second)
            continue
        }
        for _, item := range items {
            s.updates <- item
        }
        if now := time.Now(); next.After(now) {
            time.Sleep(next.Sub(now))
        }
    }
```

22

Solution

Change the body of `loop` to a `select` with three cases:

- `Close` was called
- it's time to call `Fetch`
- send an item on `s.updates`

Structure: for-select loop

`loop` runs in its own goroutine.

`select` lets `loop` avoid blocking indefinitely in any one state.

```
func (s *sub) loop() {
    ... declare mutable state ...
    for {
        ... set up channels for cases ...
        select {
        case <-c1:
            ... read/write state ...
        case c2 <- x:
            ... read/write state ...
        case y := <-c3:
            ... read/write state ...
        }
    }
}
```

The cases interact via local state in `loop`.

Case 1: Close

`Close` communicates with `loop` via `s.closing`.

```
type sub struct {
    closing chan chan error
}
```

The service (`loop`) listens for requests on its channel (`s.closing`).

The client (`Close`) sends a request on `s.closing`: *exit and reply with the error*

In this case, the only thing in the request is the *reply channel*.

Case 1: Close

`Close` asks loop to exit and waits for a response.

```
func (s *sub) Close() error {
    errc := make(chan error)
    s.closing <- errc
    return <-errc
}
```

`loop` handles `Close` by replying with the `Fetch` error and exiting.

```
    var err error // set when Fetch fails
    for {
        select {
        case errc := <-s.closing:
            errc <- err
            close(s.updates) // tells receiver we're done
            return
        }
    }
```

Case 2: Fetch

Schedule the next `Fetch` after some delay.

```
var pending []Item // appended by fetch; consumed by send
var next time.Time // initially January 1, year 0
var err error
for {
    var fetchDelay time.Duration // initally 0 (no delay)
    if now := time.Now(); next.After(now) {
        fetchDelay = next.Sub(now)
    }
    startFetch := time.After(fetchDelay)

    select {
    case <-startFetch:
        var fetched []Item
        fetched, next, err = s.fetcher.Fetch()
        if err != nil {
            next = time.Now().Add(10 * time.Second)
            break
        }
        pending = append(pending, fetched...)
    }
}
```

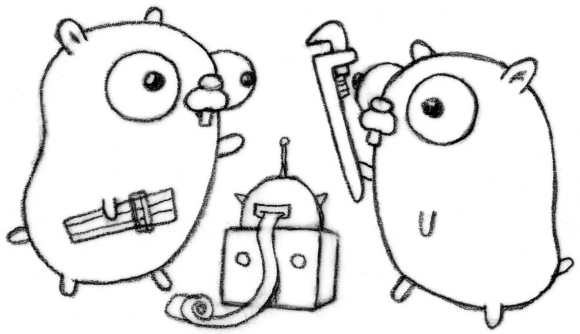
27

Case 3: Send

Send the fetched items, one at a time.

```
var pending []Item // appended by fetch; consumed by send
for {
    select {
    case s.updates <- pending[0]:
        pending = pending[1:]
    }
}
```

Whoops. This crashes.



Select and nil channels

Sends and receives on nil channels block.

Select never selects a blocking case.

```
func main() {
    a, b := make(chan string), make(chan string)
    go func() { a <- "a" }()
    go func() { b <- "b" }()
    if rand.Intn(2) == 0 {
        a = nil
        fmt.Println("nil a")
    } else {
        b = nil
        fmt.Println("nil b")
    }
    select {
    case s := <-a:
        fmt.Println("got", s)
    case s := <-b:
        fmt.Println("got", s)
    }
}
```

Run

29

Case 3: Send (fixed)

Enable send only when pending is non-empty.

```
var pending []Item // appended by fetch; consumed by send
for {
    var first Item
    var updates chan Item
    if len(pending) > 0 {
        first = pending[0]
        updates = s.updates // enable send case
    }

    select {
    case updates <- first:
        pending = pending[1:]
    }
}
```

30

Select

Put the three cases together:

```
        select {
        case errc := <-s.closing:
            errc <- err
            close(s.updates)
            return
        case <-startFetch:
            var fetched []Item
            fetched, next, err = s.fetcher.Fetch()
            if err != nil {
                next = time.Now().Add(10 * time.Second)
                break
            }
            pending = append(pending, fetched...)
        case updates <- first:
            pending = pending[1:]
        }
```

The cases interact via `err`, `next`, and `pending`.

No locks, no condition variables, no callbacks.

Bugs fixed

- Bug 1: unsynchronized access to `s.closed` and `s.err`
- Bug 2: `time.Sleep` may keep loop running
- Bug 3: `loop` may block forever sending on `s.updates`

```
        select {
        case errc := <-s.closing:
            errc <- err
            close(s.updates)
            return
        case <-startFetch:
            var fetched []Item
            fetched, next, err = s.fetcher.Fetch()
            if err != nil {
                next = time.Now().Add(10 * time.Second)
                break
            }
            pending = append(pending, fetched...)
        case updates <- first:
            pending = pending[1:]
        }
```

32

We can improve loop further

33

Issue: Fetch may return duplicates

```
var pending []Item
var next time.Time
var err error
```

```
case <-startFetch:
    var fetched []Item
    fetched, next, err = s.fetcher.Fetch()
    if err != nil {
        next = time.Now().Add(10 * time.Second)
        break
    }
    pending = append(pending, fetched...)
```

Fix: Filter items before adding to pending

```
var pending []Item
var next time.Time
var err error
var seen = make(map[string]bool) // set of item.GUIDs
```

```
        case <-startFetch:
            var fetched []Item
            fetched, next, err = s.fetcher.Fetch()
            if err != nil {
                next = time.Now().Add(10 * time.Second)
                break
            }
            for _, item := range fetched {
                if !seen[item.GUID] {
                    pending = append(pending, item)
                    seen[item.GUID] = true
                }
            }
```

35

Issue: Pending queue grows without bound

```
        case <-startFetch:
            var fetched []Item
            fetched, next, err = s.fetcher.Fetch()
            if err != nil {
                next = time.Now().Add(10 * time.Second)
                break
            }
            for _, item := range fetched {
                if !seen[item.GUID] {
                    pending = append(pending, item)
                    seen[item.GUID] = true
                }
            }
```

Fix: Disable fetch case when too much pending

```
const maxPending = 10
```

```
        var fetchDelay time.Duration
        if now := time.Now(); next.After(now) {
            fetchDelay = next.Sub(now)
        }
        var startFetch <-chan time.Time
        if len(pending) < maxPending {
            startFetch = time.After(fetchDelay) // enable fetch case
        }
```

Could instead drop older items from the head of pending.

Issue: Loop blocks on Fetch

```
        case <-startFetch:
            var fetched []Item
            fetched, next, err = s.fetcher.Fetch()
            if err != nil {
                next = time.Now().Add(10 * time.Second)
                break
            }
            for _, item := range fetched {
                if !seen[item.GUID] {
                    pending = append(pending, item)
                    seen[item.GUID] = true
                }
            }
```

Fix: Run Fetch asynchronously

Add a new `select` case for `fetchDone`.

```
type fetchResult struct{ fetched []Item; next time.Time; err error }
```

```
    var fetchDone chan fetchResult // if non-nil, Fetch is running
```

```
        var startFetch <-chan time.Time
        if fetchDone == nil && len(pending) < maxPending {
            startFetch = time.After(fetchDelay) // enable fetch case
        }
```

```
        select {
        case <-startFetch:
            fetchDone = make(chan fetchResult, 1)
            go func() {
                fetched, next, err := s.fetcher.Fetch()
                fetchDone <- fetchResult{fetched, next, err}
            }()
        case result := <-fetchDone:
            fetchDone = nil
            // Use result.fetched, result.next, result.err
```

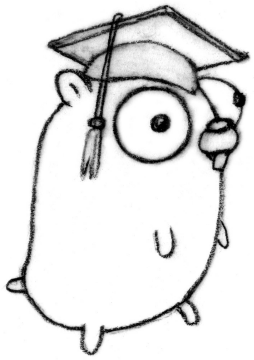
Implemented Subscribe

Responsive. Cleans up. Easy to read and change.

Three techniques:

- `for-select` loop
- service channel, reply channels (`chan chan error`)
- `nil` channels in `select` cases

More details online, including `Merge`.

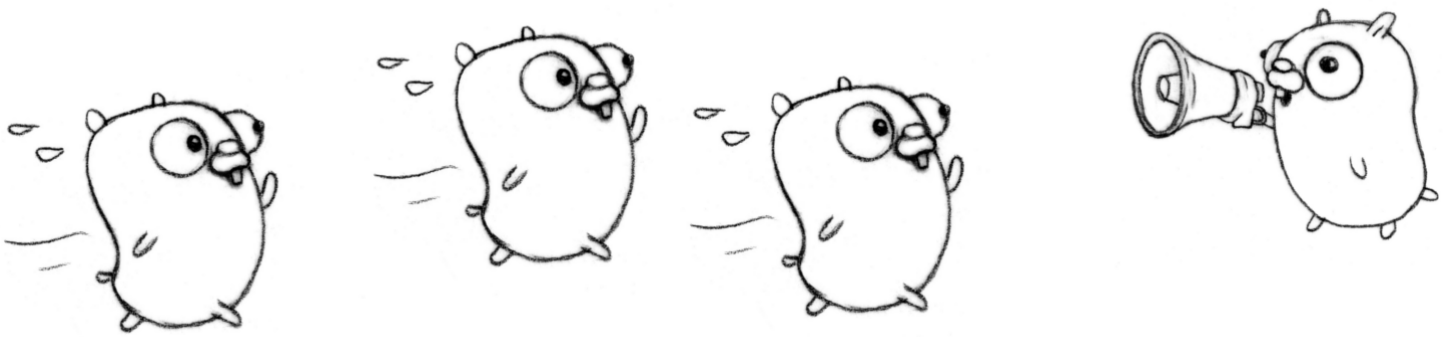


Conclusion

Concurrent programming can be tricky.

Go makes it easier:

- channels convey data, timer events, cancellation signals
- goroutines serialize access to local mutable state
- stack traces & deadlock detector
- race detector



Links

Go Concurrency Patterns (2012)

talks.golang.org/2012/concurrency.slide (http://talks.golang.org/2012/concurrency.slide)

Concurrency is not parallelism

golang.org/s/concurrency-is-not-parallelism (http://golang.org/s/concurrency-is-not-parallelism)

Share memory by communicating

golang.org/doc/codewalk/sharemem (http://golang.org/doc/codewalk/sharemem)

Go Tour (learn Go in your browser)

tour.golang.org (http://tour.golang.org)

Thank you

Sameer Ajmani
Google
http://profiles.google.com/ajmani (http://profiles.google.com/ajmani)
@Sajma (http://twitter.com/Sajma)
http://golang.org (http://golang.org)