

Concurrency made easy

GopherCon Singapore



Good afternoon! It's my pleasure to be here with you at the inaugural Gophercon Singapore.

This talk is "Concurrency made Easy".

@davecheney

Go programmer from Sydney, Australia



My name is David. I'm a software programmer and hardware enthusiast from Sydney, Australia.

If i'm honest, this slide is just here so keynote's timer starts correctly.

**“Wow, I really went overboard
with channels.”**

For my talk, I want to continue the theme of concurrency that the previous speakers have touched upon.

As someone who has the privilege of talking with programmers who are considering adopting Go, I find that a common motivation amongst many is they want to take advantage of the parallelism inherent in modern hardware.

“I went crazy with goroutines, it was impossible to understand what my program did.”

Similarly, in my self appointed role of developer advocate, when I meet people who've learnt Go, one of the more common thing they say after they have been writing Go for a year or so are things like this:

Do either of these quotes resonate with you? Maybe a little bit?

If you learn Go formally from a book or training course, you might have noticed that the concurrency section is always one of the last you'll cover. I know that this is true for my own training material.

There's a disconnect between the concurrency primitives that Go, and the expectations of those who try it.

Clearly there is a disconnect between the concurrency primitives that the language offers, and the expectations of many who come to Go for exactly those features.

There is a dichotomy here; Go's headline feature is our simple, lightweight concurrency model. As a product, our language almost sells itself on this one feature alone.

But on the other hand there is a narrative that concurrency isn't actually that easy to use, otherwise authors wouldn't make it the last chapter in their book and we wouldn't look back on our formative efforts with regret.

I believe that Go's concurrency model is one of the simplest available, and the requirement to embrace parallelism has never been more acute.

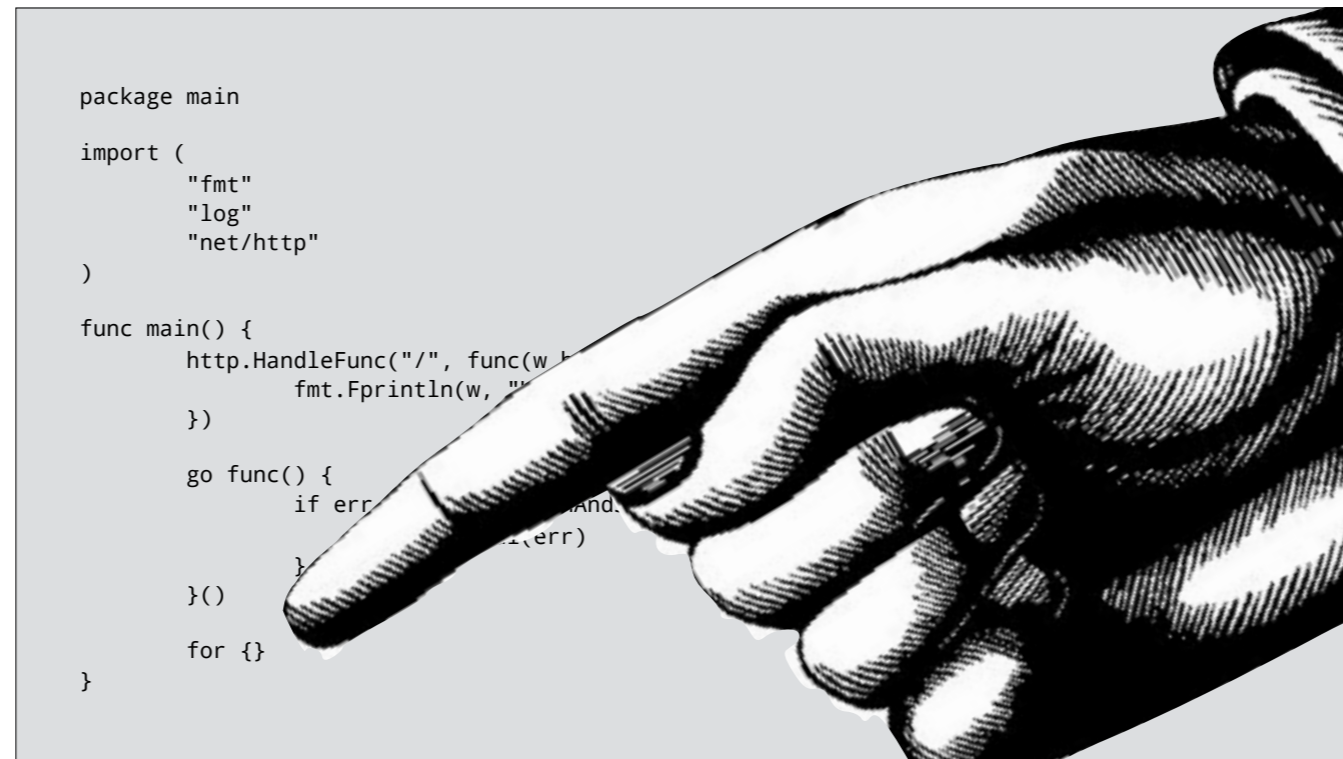
And this is a shame, because I believe that not only is Go's concurrency model one of the simplest available

but the requirement for programmers to embrace the parallelism inherent in their hardware has never been more acute.

So with this as a background, I'd like to spend my time today talking some ideas to make concurrency easier to use, not just for newcomers, but for all of us.

Let's start with the go keyword.

Let's start, at the obvious place, with talking about the go keyword.



Here's a simple Web 2.0 Hello World program. Can anyone tell me what's wrong with it?

(click)



```
for {}
```

That's right, `for{}` is an infinite loop.

`for{}` is going to block the main goroutine because it doesn't do any IO, wait on a lock, send or receive on a channel, or otherwise communicate with the scheduler.

As the runtime is mostly cooperatively scheduled, this program is going to spin fruitlessly on a single CPU, and may eventually end up live-locked.

How could we fix this? Here's one suggestion.

```
package main

import (
    "fmt"
    "log"
    "net/http"
    "runtime"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Hello, GopherCon SG")
    })

    go func() {
        if err := http.ListenAndServe(":8080", nil); err != nil {
            log.Fatal(err)
        }
    }()

    for {
        runtime.Gosched()
    }
}
```

Don't laugh, this is a common solution I see. It's symptomatic of not understanding the underlying problem.

Now, if you're a little more experienced with go, you might instead write something like this.

```
package main

import (
    "fmt"
    "log"
    "net/http"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Hello, GopherCon SG")
    })

    go func() {
        if err := http.ListenAndServe(":8080", nil); err != nil {
            log.Fatal(err)
        }
    }()

    select {}
}
```

An empty select statement will block forever. This is a useful property because now we're not spinning a whole CPU just to call `runtime.Gosched()`.

However, as I said before, we're only treating the symptom, not the cause.

I want to present to you another solution, one which has hopefully already occurred to you.

```
package main

import (
    "fmt"
    "log"
    "net/http"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Hello, GopherCon SG")
    })

    if err := http.ListenAndServe(":8080", nil); err != nil {
        log.Fatal(err)
    }
}
```

Rather than run ListenAndServe in a goroutine, leaving us with the problem of what to do with the main goroutine

Simply run ListenAndServe on the main gorouting itself

If you have to wait for the result of an operation, it's easier to do it yourself.

So this is my first suggestion:

If your goroutine cannot make progress until it gets the result from another, oftentimes it is simpler to just do the work yourself rather than to delegate it.

This often coincides with eliminating a lot of state tracking and channel manipulation required to plumb a result back from a goroutine to its initiator.

I've chosen to make this my first example, because compared to my next, it's not only the shorter, but perhaps the more profound. Many Go programmers overuse goroutines, especially when they are starting out.

**This is Go, you get no points for
doing one thing at a time.**

The previous example showed using a goroutine when one wasn't really necessary.

But of course, this is Go, and one receives no points for only doing one thing at a time

Indeed there are many instances where you want to exploit the parallelism available in your hardware. And to do so, you must use goroutines

```

func restore(repos []string) error {
    errChan := make(chan error, 1)
    sem := make(chan int, 4) // four jobs at once
    var wg sync.WaitGroup
    wg.Add(len(repos))
    for _, repo := range repos {
        sem <- 1
        go func() {
            defer func() {
                wg.Done()
                <-sem
            }()
            if err := fetch(repo); err != nil {
                errChan <- err
            }
        }()
    }
    wg.Wait()
    close(sem)
    close(errChan)
    return <-errChan
}

```

In this example, simplified from a prior version of gb-vendor, we're attempting to fetch a set of dependencies from their remote repositories, in parallel.

I'll give you a minute to look at this code. Does it look reasonable to you? Can anyone spot any problems with the restore function?

It turns out that there are several problems with this piece of code.

```

func restore(repos []string) error {
    errChan := make(chan error, 1)
    sem := make(chan int, 4) // four jobs at once
    var wg sync.WaitGroup
    wg.Add(len(repos))
    for _, repo := range repos {
        sem <- 1
        go func() {
            defer func() {
                wg.Done()
                <-sem
            }()
            if err := fetch(repo); err != nil {
                errChan <- err
            }
        }()
    }
    wg.Wait()
    close(sem)
    close(errChan)
    return <-errChan
}

```

As a code reviewer my first point of concern is the interaction between this section:

```

defer func() {
    wg.Done()
    <-sem
}()

```

and this section:

```

wg.Wait()
close(sem)

```

My question for you is, `close(sem)` happens after `wg.Wait()` therefore it also happens after `wg.Done()`, but not necessarily after `<-sem` — the close could occur before.

Could this cause a panic?

**<- sem happens before close(sem),
which drains a value from sem, then sem
is marked closed
or
close(sem) occurs first, and <- sem
returns the zero value for the channel
without blocking**

As it happens, no it cannot cause a panic.

Either <-sem happens before close(sem), in which case it drains a value from sem and then sem is marked closed, or close(sem) occurs first,

What does receiving from a closed channel return? the zero value

And when does it return? immediately

But you had to think about it to be sure. The logic is unnecessarily confusing.

If we simplify the defer statement and reorder the operations to get

```
func restore(repos []string) error {
    errChan := make(chan error, 1)
    sem := make(chan int, 4) // four jobs at once
    var wg sync.WaitGroup
    wg.Add(len(repos))
    for _, repo := range repos {
        sem <- 1
        go func() {
            defer wg.Done()
            if err := fetch(repo); err != nil {
                errChan <- err
            }
            <-sem
        }()
    }
    wg.Wait()
    close(sem)
    close(errChan)
    return <-errChan
}
```

Now there is no question in which order the operations will occur.

In this program we add to the wait group, then push a value onto the sem channel, raising the level of the semaphore.

When each goroutine is done, the reverse occurs, we remove a value from the sem channel, lowering the semaphore, and then defer calls wg.Done as the final operation, to indicate that the goroutine is done.

So my suggestion to you is

Release locks and semaphores in the reverse order you acquired them.

Always release locks and semaphores in the reverse order to which you acquired them.

At best, mixing the order of acquire and release generates confusing code which is difficult to reason about.

At worst, mixing acquire and release leads to lock inversion and deadlocks.

Why `close(sem)` ?

Now that we've rearranged the program a little, we can ask another question. What is the reason for closing `sem`?

If a restaurant closes, it does not remove anyone seated at that time, it's just an indication that the restaurant is not taking additional patrons.

Similarly, channels are not resources like files or network sockets; the `close` signal does not free a channel, it just marks that channel as no longer accepting new values.

```
func restore(repos []string) error {
    errChan := make(chan error, 1)
    sem := make(chan int, 4) // four jobs at once
    var wg sync.WaitGroup
    wg.Add(len(repos))
    for _, repo := range repos {
        sem <- 1
        go func() {
            defer wg.Done()
            if err := fetch(repo); err != nil {
                errChan <- err
            }
            <-sem
        }()
    }
    wg.Wait()
    close(sem)
    close(errChan)
    return <-errChan
}
```

In our example nothing is waiting in a select or range loop for a close signal on sem, so we can remove the close(sem) call.

Channels aren't resources like files or sockets, you don't need to close them to free them.

Closing a channel is a signal to its receivers that it is no longer accepting new data; nothing more, nothing less.

Closing a channel is not necessary to "free" a channel. You don't need to close a channel to "clean up" its resources.

**Speaking of semaphores, let's
look at how sem is used.**

Speaking of semaphores, let's look a bit closer at how sem is used.

```

func restore(repos []string) error {
    errChan := make(chan error, 1)
    sem := make(chan int, 4) // four jobs at once
    var wg sync.WaitGroup
    wg.Add(len(repos))
    for _, repo := range repos {
        sem <- 1
        go func() {
            defer wg.Done()
            if err := fetch(repo); err != nil {
                errChan <- err
            }
            <-sem
        }()
    }
    wg.Wait()
    close(errChan)
    return <-errChan
}

```

The role of sem is to make sure that at any one time, there is a cap on the number of fetch operations running. In this example, sem has a capacity of four.

But if you look closely, sem isn't guaranteeing there are no more than four fetch operations running, it's guaranteeing that there are no more than four goroutines running.


```
for _, repo := range repos {
    sem <- 1
    go func() {
        defer wg.Done()
        if err := fetch(repo); err != nil {
            errChan <- err
        }
    }()
    <-sem
}
}
```

Assuming there are enough values in repos, each time through the loop we try to push the number 1 onto the sem channel, then we fire off a fetch goroutine.

What happens when it's the fifth time through the loop? Most likely we'll have four fetch goroutines running, or possibly those four goroutines won't even have been scheduled to run yet—remember that the scheduler doesn't give any guarantees if it will run a goroutine immediately, or schedule for later.

On the fifth iteration the main loop is going to block trying to push a value onto sem. Rather than spawning len(repos) goroutines which coordinate amongst themselves for a semaphore, this loop will proceed at the rate that fetch invocations finish.

While it doesn't matter in this example--restore blocks until all repos have been fetched—there are many situations where the calling code may expect the function scheduling its work to complete quickly and return, while the work occurs in the background.

```

func restore(repos []string) error {
    errChan := make(chan error, 1)
    sem := make(chan int, 4) // four jobs at once
    var wg sync.WaitGroup
    wg.Add(len(repos))
    for _, repo := range repos {
        go func() {
            defer wg.Done()
            sem <- 1
            if err := fetch(repo); err != nil {
                errChan <- err
            }
            <-sem
        }()
    }
    wg.Wait()
    close(errChan)
    return <-errChan
}

```

The solution to this problem is to move `sem <- 1` inside the goroutine.

Now all the fetch goroutines will be created immediately, and will negotiate a semaphore when they get scheduled by the runtime.

And this leads to my next recommendation

Acquire semaphores when you're ready to use them.

Although goroutines are cheap to create and schedule, the resources they operate on, files, sockets, bandwidth, and so on, are often scarcer. The pattern of using a channel as a semaphore to limit work in progress is quite common.

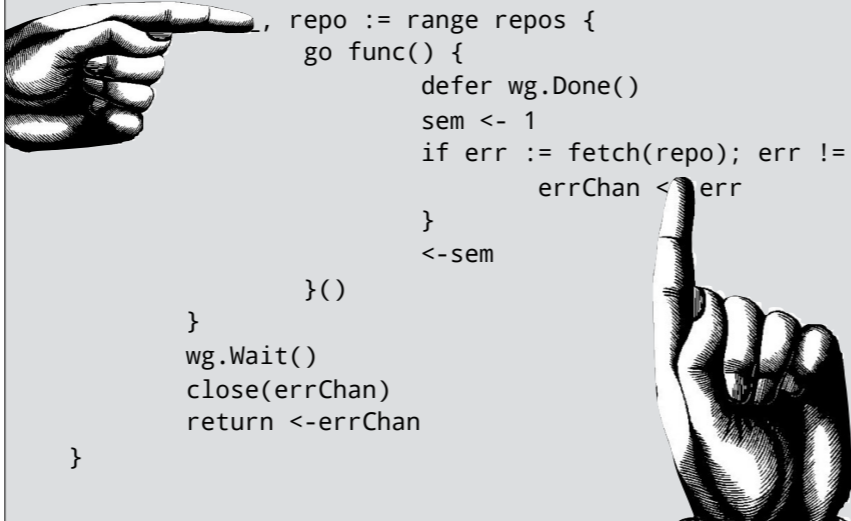
However, to make sure that you don't unduly block the code offloading work to a goroutine, acquire a semaphore when you're ready to use them, not when you expect to use them.

**Hopefully we've got all the bugs
out of this program ...**

Hopefully we've got all the bugs out of this program.

But there's one more obvious one that we haven't talked about yet. And it's a serious one that catches almost every Go programmer out at least once.

```
func restore(repos []string) error {
    errChan := make(chan error, 1)
    sem := make(chan int, 4) // four jobs at once
    var wg sync.WaitGroup
    wg.Add(len(repos))
    for _, repo := range repos {
        go func() {
            defer wg.Done()
            sem <- 1
            if err := fetch(repo); err != nil {
                errChan <- err
            }
            <-sem
        }()
    }
    wg.Wait()
    close(errChan)
    return <-errChan
}
```



The variable `repo` in the for loop is lexically captured by the anonymous function executed as a goroutine.

click / click

This is going to lead to two problems:

1. The value of `repo` inside the goroutine is going to be overwritten on each loop iteration; all the `fetch` calls will likely end up trying to fetch the last repository.
2. This is a data race because we have one goroutine assigning to `repo` concurrently with others are trying to read it.

```

func restore(repos []string) error {
    errChan := make(chan error, 1)
    sem := make(chan int, 4) // four jobs at once
    var wg sync.WaitGroup
    wg.Add(len(repos))
    for i := range repos {
        go func() {
            defer wg.Done()
            sem <- 1
            if err := fetch(repos[i]); err != nil {
                errChan <- err
            }
            <-sem
        }()
    }
    wg.Wait()
    close(errChan)
    return <-errChan
}

```

This is one of the classical Go paper cuts which all of us have to learn the hard way. You might think that rewriting the program to use an index variable would solve it

But in fact this is just a different version of the same problem; rather than capturing repo lexically, each goroutine captures the variable i.

In my opinion, the cause of this common error is the interaction between anonymous functions, lexical closures, and goroutines.

It's cute to be able to write an anonymous function and execute it inline with a goroutine, but the reader has to consider the effects of lexical closure, and we know that the ideas of nested scopes are already a pain point for many developers no matter their level of experience.

```

func restore(repos []string) error {
    errChan := make(chan error, 1)
    sem := make(chan int, 4) // four jobs at once
    var wg sync.WaitGroup
    wg.Add(len(repos))
    for i := range repos {
        go func(repo string) {
            defer wg.Done()
            sem <- 1
            if err := fetch(repo); err != nil {
                errChan <- err
            }
            <-sem
        }(repos[i])
    }
    wg.Wait()
    close(errChan)
    return <-errChan
}

```

Instead what we must do is ensure that unique values of repo (or i) are passed to each fetch goroutine as they are invoked.

There are a few ways to do this, but the most obvious is to explicitly pass the repo value into our goroutine.

***Avoid mixing anonymous functions
and goroutines.***

The recommendation I draw from this is

Avoid mixing anonymous functions and goroutines


```
func restore(repos []string) error {
    errChan := make(chan error, 1)
    sem := make(chan int, 4) // four jobs at once
    var wg sync.WaitGroup
    wg.Add(len(repos))
    for _, repo := range repos {
        go worker(repo, sem, &wg, errChan)
    }
    wg.Wait()
    close(errChan)
    return <-errChan
}

func worker(repo string, sem chan int, wg *sync.WaitGroup, errChan chan error) {
    defer wg.Done()
    sem <- 1
    if err := fetch(repo); err != nil {
        errChan <- err
    }
    <-sem
}
```

To apply my own advice, we replace the anonymous function with a named one, and pass all the variables necessary to it.

This results in a less pithy example, but in return it eliminates the possibility of lexical capture, and the associated data race.

This is the last bug, right?

So now, after this review we've refactored the code to make a clean separation between producing work to be performed—our restore function—from the execution of that work—the worker function.

Have we fixed all the issues with this code? Not yet, there's still one left.

```

func restore(repos []string) error {
    errChan := make(chan error, 1)
    sem := make(chan int, 4) // four jobs at once
    var wg sync.WaitGroup
    wg.Add(len(repos))
    for _, repo := range repos {
        go worker(repo, sem, &wg, errChan)
    }
    wg.Wait()
    close(errChan)
    return <-errChan
}

func worker(repo string, sem chan int, wg *sync.WaitGroup, errChan chan err) {
    defer wg.Done()
    sem <- 1
    if err := fetch(repo); err != nil {
        errChan <- err
    }
    <-sem
}

```

Let's look at the core of the worker function, calling fetch and handling the error.

Can anyone see a problem with this? I'll give you a hint; what happens if more than one fetch fails, say if the network goes down and all the fetches fail at once. What would happen?

```
func restore(repos []string) error {
    errChan := make(chan error, 1)
    sem := make(chan int, 4) // four jobs at once
    var wg sync.WaitGroup
    wg.Add(len(repos))
    for _, repo := range repos {
        go worker(repo, sem, &wg, errChan)
    }
    wg.Wait()
    close(errChan)
    return <-errChan
}

func worker(repo string, sem chan int, wg *sync.WaitGroup, errChan chan error) {
    defer wg.Done()
    sem <- 1
    if err := fetch(repo); err != nil {
        errChan <- err
    }
    <-sem
}
```

Let's have a look at the declaration of errChan.

It's a buffered channel with a capacity of one. But our semaphore channel has a capacity of four, so potentially we could have up to four goroutines trying to write errors to errChan.

errChan won't be read from until wg.Wait returns, and wg.Wait will not return until wg.Done has been called in each goroutine. So we have a potential deadlock situation.

And to resolve this, I want to introduce my last piece of advice.

***Before you start a goroutine, always
know when, and how, it will stop.***

Before you start a goroutine, always know when and how it will stop.

How to we go about applying this advice? I like to work backwards and ask what are the ways this goroutine can exit. We know the goroutine starts at the worker function, so when worker exits the goroutine is done.

```
func worker(repo string, sem chan int, wg *sync.WorkGroup, errChan chan err) {  
    defer wg.Done()  
    sem <- 1  
    if err := fetch(repo); err != nil {  
        errChan <- err  
    }  
    <-sem  
}
```

Because this function contains a defer statement we consider this statement last. If there were multiple defer statements they are executed Last in, first out.

wg.Done is a signal the waitgroup—another kind of semaphore—that the task is done. wg.Done never blocks so this statement will not prevent worker returning.

```
func worker(repo string, sem chan int, wg *sync.WorkGroup, errChan chan err) {
    defer wg.Done()
    sem <- 1
    if err := fetch(repo); err != nil {
        errChan <- err
    }
    <-sem
}
```

The penultimate statement is the receive from `<-sem`, which cannot block because we could not have got to this point without placing a value into `sem`.

The value we get out may not be the one we placed there, but we are guaranteed to receive a value, so that statement won't block.

```
func worker(repo string, sem chan int, wg *sync.WorkGroup, errChan chan err) {
    defer wg.Done()
    sem <- 1
    if err := fetch(repo); err != nil {
        errChan <- err
    }
    <-sem
}
```

This just leaves the call to `fetch` which we assume has appropriate timeouts in place to handle badly behaved remote servers.

If there is no error from `fetch` the function will return, passing through `defer` as we saw earlier and the goroutine is done. However if there is an error, then we must first place it onto the `errChan` channel.

If want to make sure all possible writes to `errChan` do not block, what's the right capacity for `errChan`?


```

func restore(repos []string) error {
    errChan := make(chan error, len(repos))
    sem := make(chan int, 4) // four jobs at once
    var wg sync.WaitGroup
    wg.Add(len(repos))
    for _, repo := range repos {
        go worker(repo, sem, &wg, errChan)
    }
    wg.Wait()
    close(errChan)
    return <-errChan
}

func worker(repo string, sem chan int, wg *sync.WaitGroup, errChan chan error) {
    defer wg.Done()
    sem <- 1
    if err := fetch(repo); err != nil {
        errChan <- err
    }
    <-sem
}

```

One answer would be to set the capacity of errChan to the size of len(repos).

This guarantees that even if every fetch were to fail, there will be capacity to store each error without blocking the send.

However there is another option. restore only returns one error value, which it reads from errChan.

```

func restore(repos []string) error {
    errChan := make(chan error, 1)
    sem := make(chan int, 4) // four jobs at once
    var wg sync.WaitGroup
    wg.Add(len(repos))
    for _, repo := range repos {
        go worker(repo, sem, &wg, errChan)
    }
    wg.Wait()
    close(errChan)
    return <-errChan
}

func worker(repo string, sem chan int, wg *sync.WaitGroup, errChan chan error) {
    defer wg.Done()
    sem <- 1
    if err := fetch(repo); err != nil {
        select {
        case errChan <- err:
            // we're the first worker to fail
        default:
            // some other failure has already happened
        }
    }
    <-sem
}

```

Rather than creating space for all possible errors, we can use a non blocking send to place the error onto errChan if non already exists, otherwise the value is discarded.

```
func restore(repos []string) error {
    errChan := make(chan error, 1)
    sem := make(chan int, 4) // four jobs at once
    var wg sync.WaitGroup
    wg.Add(len(repos))
    for _, repo := range repos {
        go worker(repo, sem, &wg, errChan)
    }
    wg.Wait()
    close(errChan)
    return <-errChan
}
```

As a bonus question, which you can come and talk to me about afterwards. The read from errChan in restore is guaranteed to never block, even if all calls to fetch succeeded. How does this work?

Concurrency made easy

The name of this talk is "Concurrency made Easy", my nod to Rich Hickey's seminal paper "Simple made Easy", in which Hickey demonstrates that easy and simple are not synonyms. They are in fact, orthogonal.

Given that I've just spent the last 20 minutes discussing 6 bugs in a 20 line function, I cannot in good conscience say to you that concurrency is simple. A concurrent system is inherently more complex than a sequential system.

If you have to wait for the result of an operation, it's easier to do it yourself.

Release locks and semaphores in the reverse order you acquired them.

Channels aren't resources like files or sockets, you don't need to close them to free them.

Acquire semaphores when you're ready to use them.

Avoid mixing anonymous functions and goroutines

Before you start a goroutine, always know when, and how, it will stop.

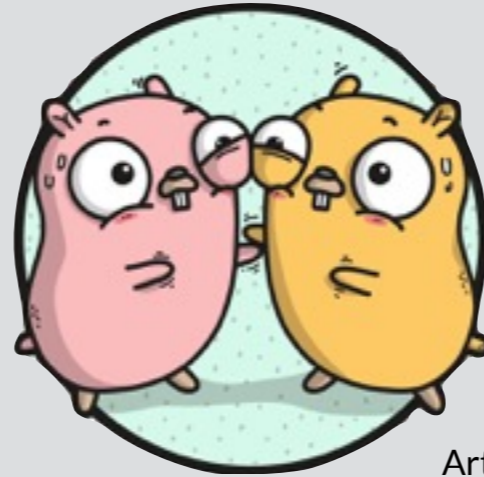
What I've pitched you today are not patterns to be applied in a specific situation.

Instead what I want you to take away from this presentation are the general suggestions I've illustrated—principals, if you will.

Principals, not rules or patterns. Principals, which give you a framework to answer your own questions as you encounter them while writing concurrent programs.

and hopefully by applying these principals, you can make your time writing concurrent Go code, if not easy, at least a little easier.

Thank you!



Artwork @ashleymacnamara
Gopher design @reneefrench