Introduction

Concurrency features in Go

People seemed fascinated by the concurrency features of Go when the language was first announced.

Questions:

- Why is concurrency supported?
- What is concurrency, anyway?
- Where does the idea come from?
- What is it good for?
- How do I use it?

Why?

Look around you. What do you see?

Do you see a single-stepping world doing one thing at a time?

Or do you see a complex world of interacting, independently behaving pieces?

That's why. Sequential processing on its own does not model the world's behavior.

What is concurrency?

Concurrency is the composition of independently executing computations.

Concurrency is a way to structure software, particularly as a way to write clean code that interacts well with the real world.

It is not parallelism.

Concurrency is not parallelism

Concurrency is not parallelism, although it enables parallelism.

If you have only one processor, your program can still be concurrent but it cannot be parallel.

On the other hand, a well-written concurrent program might run efficiently in parallel on a multiprocessor. That property could be important...

For more on that distinction, see the link below. Too much to discuss here.

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

Basic Examples

A boring function

We need an example to show the interesting properties of the concurrency primitives.

To avoid distraction, we make it a boring example.

```
func boring(msg string) {
    for i := 0; ; i++ {
        fmt.Println(msg, i)
        time.Sleep(time.Second)
    }
}
```

Slightly less boring

Make the intervals between messages unpredictable (still under a second).

```
func boring(msg string) {
    for i := 0; ; i++ {
        fmt.Println(msg, i)
        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
}
```

Running it

The boring function runs on forever, like a boring party guest.

```
func main() {
    boring("boring!")
}

func boring(msg string) {
    for i := 0; ; i++ {
        fmt.Println(msg, i)
        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
}
```

Ignoring it

The go statement runs the function as usual, but doesn't make the caller wait.

It launches a goroutine.

The functionality is analogous to the & on the end of a shell command.

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

func main() {
    go boring("boring!")
}
```

Ignoring it a little less

When main returns, the program exits and takes the boring function down with it.

We can hang around a little, and on the way show that both main and the launched goroutine are running.

```
func main() {
    go boring("boring!")
    fmt.Println("I'm listening.")
    time.Sleep(2 * time.Second)
    fmt.Println("You're boring; I'm leaving.")
}
```

Goroutines

What is a goroutine? It's an independently executing function, launched by a go statement.

It has its own call stack, which grows and shrinks as required.

It's very cheap. It's practical to have thousands, even hundreds of thousands of goroutines.

It's not a thread.

There might be only one thread in a program with thousands of goroutines.

Instead, goroutines are multiplexed dynamically onto threads as needed to keep all the goroutines running.

But if you think of it as a very cheap thread, you won't be far off.

Communication

Our boring examples cheated: the main function couldn't see the output from the other goroutine.

It was just printed to the screen, where we pretended we saw a conversation.

Real conversations require communication.

Channels

A channel in Go provides a connection between two goroutines, allowing them to communicate.

```
    // Declaring and initializing.
    var c chan int
    c = make(chan int)
    // or
    c := make(chan int)
```

```
    // Sending on a channel.
    c <- 1
```

```
    // Receiving from a channel.
    // The "arrow" indicates the direction of data flow.
    value = <-c
```

Using channels

A channel connects the main and boring goroutines so they can communicate.

```
func main() {
    c := make(chan string)
    go boring("boring!", c)
    for i := 0; i < 5; i++ {
        fmt.Printf("You say: %q\n", <-c) // Receive expression is just a value.
    }
    fmt.Println("You're boring; I'm leaving.")
}
```

```
func boring(msg string, c chan string) {
    for i := 0; ; i++ {
        c <- fmt.Sprintf("%s %d", msg, i) // Expression to be sent can be any suitable value.
        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
}
```

Synchronization

When the main function executes <–c, it will wait for a value to be sent.

Similarly, when the boring function executes c <– value, it waits for a receiver to be ready.

A sender and receiver must both be ready to play their part in the communication.
Otherwise we wait until they are.

Thus channels both communicate and synchronize.

An aside about buffered channels

Note for experts: Go channels can also be created with a buffer.

Buffering removes synchronization.

Buffering makes them more like Erlang's mailboxes.

Buffered channels can be important for some problems but they are more subtle to reason about.

We won't need them today.

The Go approach

Don't communicate by sharing memory, share memory by communicating.

"Patterns"

Generator: function that returns a channel

Channels are first-class values, just like strings or integers.

```
    c := boring("boring!") // Function returning a channel.
    for i := 0; i < 5; i++ {
        fmt.Printf("You say: %q\n", <-c)
    }
    fmt.Println("You're boring; I'm leaving.")
```

```
func boring(msg string) <-chan string { // Returns receive-only channel of strings.
    c := make(chan string)
    go func() { // We launch the goroutine from inside the function.
        for i := 0; ; i++ {
            c <- fmt.Sprintf("%s %d", msg, i)
            time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
        }
    }()
    return c // Return the channel to the caller.
}
```

Channels as a handle on a service

Our boring function returns a channel that lets us communicate with the boring service it provides.

We can have more instances of the service.

```
func main() {
    joe := boring("Joe")
    anna := boring("Anna")
    for i := 0; i < 5; i++ {
        fmt.Println(<-joe)
        fmt.Println(<-anna)
    }
    fmt.Println("You're both boring; I'm leaving.")
}
```

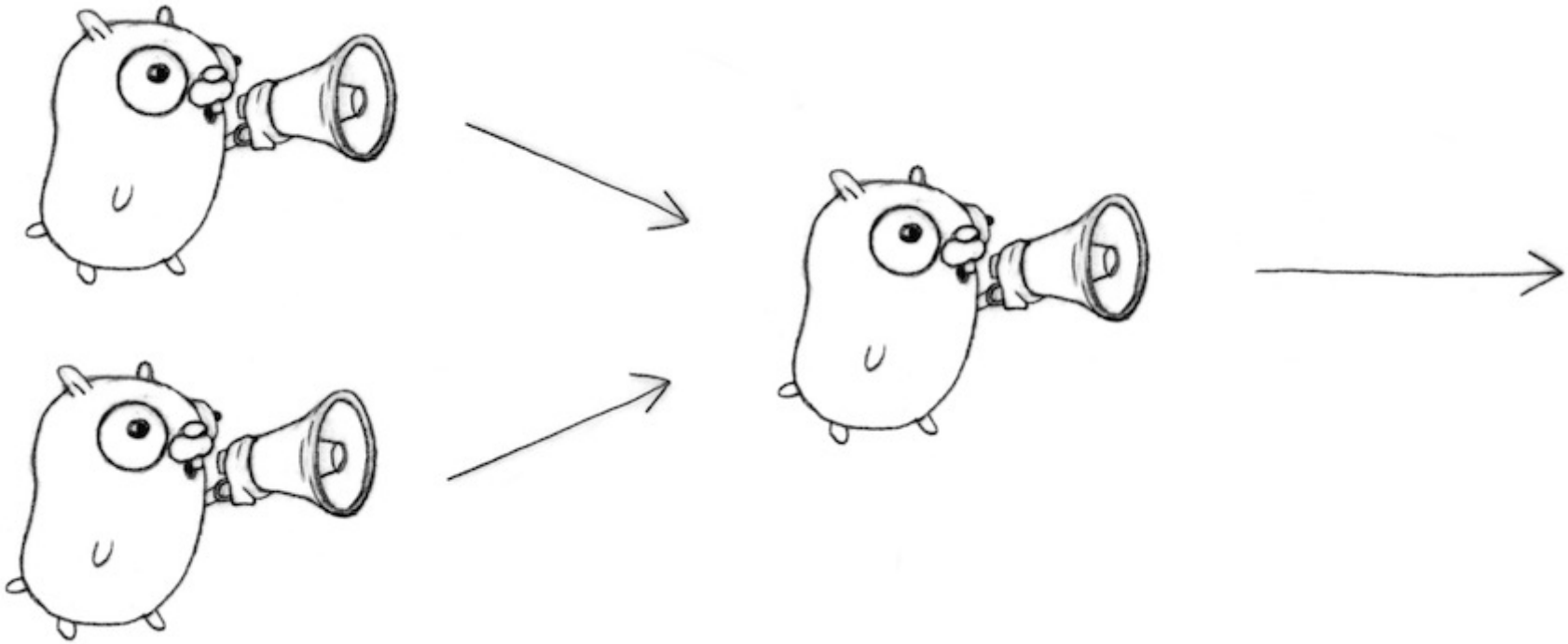
Multiplexing

These programs make Joe and Anna count in lockstep.
We can instead use a fan-in function to let whosoever is ready talk.

```
func fanIn(input1, input2 <-chan string) <-chan string {
    c := make(chan string)
    go func() { for { c <- <-input1 } }()
    go func() { for { c <- <-input2 } }()
    return c
}
```

```
func main() {
    c := fanIn(boring("Joe"), boring("Anna"))
    for i := 0; i < 10; i++ {
        fmt.Println(<-c)
    }
    fmt.Println("You're both boring; I'm leaving.")
}
```

Fan-in



Restoring sequencing

Send a channel on a channel, making goroutine wait its turn.

Receive all messages, then enable them again by sending on a private channel.

First we define a message type that contains a channel for the reply.

```
type Message struct {
    str string
    wait chan bool
}
```

Restoring sequencing.

Each speaker must wait for a go-ahead.

```
    for i := 0; i < 5; i++ {
        msg1 := <-c; fmt.Println(msg1.str)
        msg2 := <-c; fmt.Println(msg2.str)
        msg1.wait <- true
        msg2.wait <- true
    }
```

```
    waitForIt := make(chan bool) // Shared between all messages.
```

```
            c <- Message{ fmt.Sprintf("%s: %d", msg, i), waitForIt }
            time.Sleep(time.Duration(rand.Intn(2e3)) * time.Millisecond)
            <-waitForIt
```

Select

A control structure unique to concurrency.

The reason channels and goroutines are built into the language.

Select

The select statement provides another way to handle multiple channels.
It's like a switch, but each case is a communication:
- All channels are evaluated.
- Selection blocks until one communication can proceed, which then does.
- If multiple can proceed, select chooses pseudo-randomly.
- A default clause, if present, executes immediately if no channel is ready.

```
select {
case v1 := <-c1:
    fmt.Printf("received %v from c1\n", v1)
case v2 := <-c2:
    fmt.Printf("received %v from c2\n", v1)
case c3 <- 23:
    fmt.Printf("sent %v to c3\n", 23)
default:
    fmt.Printf("no one was ready to communicate\n")
}
```

Fan-in again

Rewrite our original fanIn function. Only one goroutine is needed. Old:

```
func fanIn(input1, input2 <-chan string) <-chan string {
    c := make(chan string)
    go func() { for { c <- <-input1 } }()
    go func() { for { c <- <-input2 } }()
    return c
}
```

Fan-in using select

Rewrite our original fanIn function. Only one goroutine is needed. New:

```
func fanIn(input1, input2 <-chan string) <-chan string {
    c := make(chan string)
    go func() {
        for {
            select {
            case s := <-input1:  c <- s
            case s := <-input2:  c <- s
            }
        }
    }()
    return c
}
```

Timeout using select

The time.After function returns a channel that blocks for the specified duration.
After the interval, the channel delivers the current time, once.

```
func main() {
    c := boring("Joe")
    for {
        select {
        case s := <-c:
            fmt.Println(s)
        case <-time.After(1 * time.Second):
            fmt.Println("You're too slow.")
            return
        }
    }
}
```

Timeout for whole conversation using select

Create the timer once, outside the loop, to time out the entire conversation.
(In the previous program, we had a timeout for each message.)

```
func main() {
    c := boring("Joe")
    timeout := time.After(5 * time.Second)
    for {
        select {
        case s := <-c:
            fmt.Println(s)
        case <-timeout:
            fmt.Println("You talk too much.")
            return
        }
    }
}
```

Quit channel

We can turn this around and tell Joe to stop when we're tired of listening to him.

```
quit := make(chan bool)
c := boring("Joe", quit)
for i := rand.Intn(10); i >= 0; i-- { fmt.Println(<-c) }
quit <- true
```

```
        select {
        case c <- fmt.Sprintf("%s: %d", msg, i):
            // do nothing
        case <-quit:
            return
        }
```

Receive on quit channel

How do we know it's finished? Wait for it to tell us it's done: receive on the quit channel

```
quit := make(chan string)
c := boring("Joe", quit)
for i := rand.Intn(10); i >= 0; i-- { fmt.Println(<-c) }
quit <- "Bye!"
fmt.Printf("Joe says: %q\n", <-quit)
```

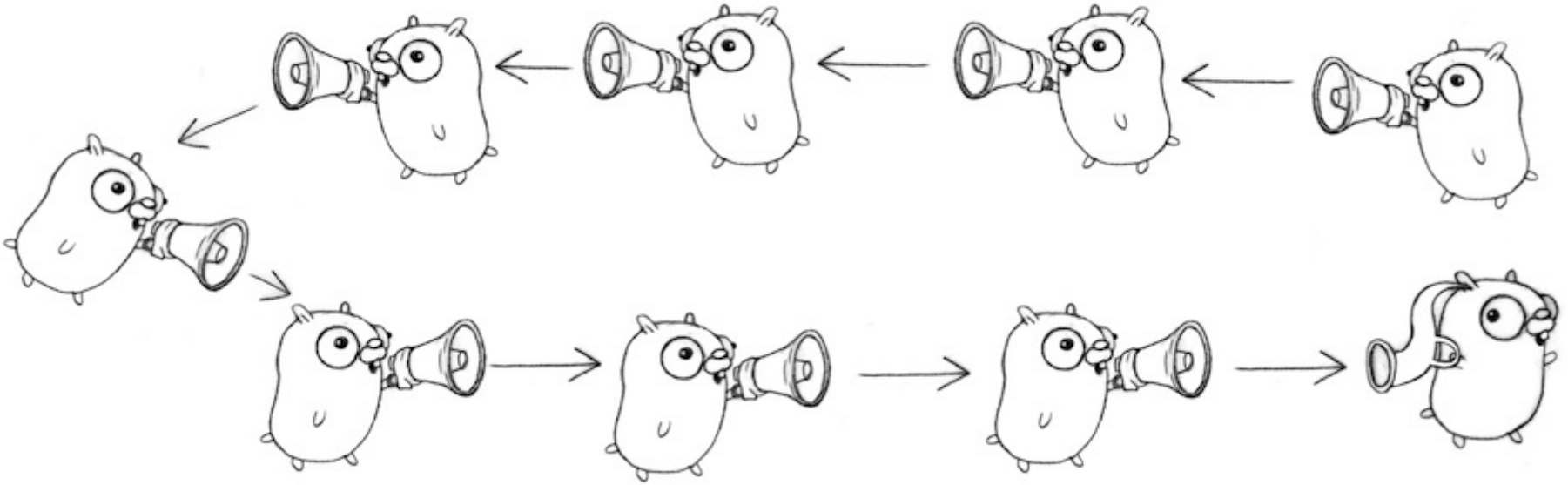
```
        select {
        case c <- fmt.Sprintf("%s: %d", msg, i):
            // do nothing
        case <-quit:
            cleanup()
            quit <- "See you!"
            return
        }
```

Daisy-chain

```
func f(left, right chan int) {
    left <- 1 + <-right
}

func main() {
    const n = 10000
    leftmost := make(chan int)
    right := leftmost
    left := leftmost
    for i := 0; i < n; i++ {
        right = make(chan int)
        go f(left, right)
        left = right
    }
    go func(c chan int) { c <- 1 }(right)
    fmt.Println(<-leftmost)
}
```

Chinese whispers, gopher style



Systems software

Go was designed for writing systems software.
Let's see how the concurrency features come into play.

Example: Google Search

Q: What does Google search do?

A: Given a query, return a page of search results (and some ads).

Q: How do we get the search results?

A: Send the query to Web search, Image search, YouTube, Maps, News,etc., then mix the results.

How do we implement this?

Google Search: A fake framework

We can simulate the search function, much as we simulated conversation before.

```
var (
    Web = fakeSearch("web")
    Image = fakeSearch("image")
    Video = fakeSearch("video")
)

type Search func(query string) Result

func fakeSearch(kind string) Search {
        return func(query string) Result {
              time.Sleep(time.Duration(rand.Intn(100)) * time.Millisecond)
              return Result(fmt.Sprintf("%s result for %q\n", kind, query))
        }
}
```

Google Search: Test the framework

```
func main() {
    rand.Seed(time.Now().UnixNano())
    start := time.Now()
    results := Google("golang")
    elapsed := time.Since(start)
    fmt.Println(results)
    fmt.Println(elapsed)
}
```

Google Search 1.0

The Google function takes a query and returns a slice of Results (which are just strings).

Google invokes Web, Image, and Video searches serially, appending them to the results slice.

```
func Google(query string) (results []Result) {
    results = append(results, Web(query))
    results = append(results, Image(query))
    results = append(results, Video(query))
    return
}
```

Google Search 2.0

Run the Web, Image, and Video searches concurrently, and wait for all results.

No locks. No condition variables. No callbacks.

```
func Google(query string) (results []Result) {
    c := make(chan Result)
    go func() { c <- Web(query) } ()
    go func() { c <- Image(query) } ()
    go func() { c <- Video(query) } ()

    for i := 0; i < 3; i++ {
        result := <-c
        results = append(results, result)
    }
    return
}
```

Google Search 2.1

Don't wait for slow servers. No locks. No condition variables. No callbacks.

```
    c := make(chan Result)
    go func() { c <- Web(query) } ()
    go func() { c <- Image(query) } ()
    go func() { c <- Video(query) } ()

    timeout := time.After(80 * time.Millisecond)
    for i := 0; i < 3; i++ {
        select {
        case result := <-c:
            results = append(results, result)
        case <-timeout:
            fmt.Println("timed out")
            return
        }
    }
    return
```

Avoid timeout

Q: How do we avoid discarding results from slow servers?

A: Replicate the servers. Send requests to multiple replicas, and use the first response.

```
func First(query string, replicas ...Search) Result {
    c := make(chan Result)
    searchReplica := func(i int) { c <- replicas[i](query) }
    for i := range replicas {
        go searchReplica(i)
    }
    return <-c
}
```

Using the First function

```
func main() {
    rand.Seed(time.Now().UnixNano())
    start := time.Now()
    result := First("golang",
        fakeSearch("replica 1"),
        fakeSearch("replica 2"))
    elapsed := time.Since(start)
    fmt.Println(result)
    fmt.Println(elapsed)
}
```

Google Search 3.0

Reduce tail latency using replicated search servers.

```
    c := make(chan Result)
    go func() { c <- First(query, Web1, Web2) } ()
    go func() { c <- First(query, Image1, Image2) } ()
    go func() { c <- First(query, Video1, Video2) } ()
    timeout := time.After(80 * time.Millisecond)
    for i := 0; i < 3; i++ {
        select {
        case result := <-c:
            results = append(results, result)
        case <-timeout:
            fmt.Println("timed out")
            return
        }
    }
    return
```

And still...

No locks. No condition variables. No callbacks.

Summary

In just a few simple transformations we used Go's concurrency primitives to convert a

- slow
- sequential
- failure-sensitive

program into one that is

- fast
- concurrent
- replicated
- robust.

Don't overdo it

They're fun to play with, but don't overuse these ideas.

Goroutines and channels are big ideas. They're tools for program construction.

But sometimes all you need is a reference counter.

Go has "sync" and "sync/atomic" packages that provide mutexes, condition variables, etc. They provide tools for smaller problems.

Often, these things will work together to solve a bigger problem.

Always use the right tool for the job.

Conclusions

Goroutines and channels make it easy to express complex operations dealing with

- multiple inputs
- multiple outputs
- timeouts
- failure

And they're fun to use.