Go: code that grows with grace

Andrew Gerrand
Google Sydney

Video

A video of this talk was recorded at Øredev in Malmö, Sweden in November 2012.

Watch the talk on Vimeo (http://vimeo.com/53221560)

2

Go

You may have heard of Go.

It's my favorite language. I think you'll like it, too.

What is Go?

An open source (BSD licensed) project:

- Language specification,
- Small runtime (garbage collector, scheduler, etc),
- Two compilers (gc and gccgo),
- 'Batteries included' standard library,
- Tools (build, fetch, test, document, profile, format),
- Documentation.

As of September 2012 we have more than 300 contributors.4

Go is about composition

Go is Object Oriented, but not in the usual way.

- no classes (methods may be declared on any type)
- no subtype inheritance
- interfaces are satisfied implicitly (structural typing)

The result: simple pieces connected by small interfaces. 5

Go is about concurrency

Go provides CSP-like concurrency primitives.

- lightweight threads (goroutines)
- typed thread-safe communication and synchronization (channels)

The result: comprehensible concurrent code.

Go is about gophers

Core values

Go is about composition, concurrency, and gophers.

Keep that in mind.

8

Hello, go

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, go")
}
```

Run

9

Hello, net

```
package main

import (
    "fmt"
    "log"
    "net"
)

const listenAddr = "localhost:4000"

func main() {
    l, err := net.Listen("tcp", listenAddr)
    if err != nil {
        log.Fatal(err)
    }
    for {
        c, err := l.Accept()
        if err != nil {
            log.Fatal(err)
        }
        fmt.Fprintln(c, "Hello!")
        c.Close()
    }
}
```

Run

10

Interfaces

Hey neato! We just used `Fprintln` to write to a net connection.

That's because a `Fprintln` writes to an `io.Writer`, and `net.Conn` is an `io.Writer`.

```
        fmt.Fprintln(c, "Hello!")
```

```
func Fprintln(w io.Writer, a ...interface{}) (n int, err error)
```

```
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

```
type Conn interface {
    Read(b []byte) (n int, err error)
    Write(b []byte) (n int, err error)
    Close() error
    // ... some additional methods omitted ...
}
```

An echo server

```
package main

import (
    "io"
    "log"
    "net"
)

const listenAddr = "localhost:4000"

func main() {
    l, err := net.Listen("tcp", listenAddr)
    if err != nil {
        log.Fatal(err)
    }
    for {
        c, err := l.Accept()
        if err != nil {
            log.Fatal(err)
        }
        io.Copy(c, c)
    }
}
```

Run

12

A closer look at io.Copy

```
        io.Copy(c, c)
```

```
// Copy copies from src to dst until either EOF is reached
// on src or an error occurs.  It returns the number of bytes
// copied and the first error encountered while copying, if any.
func Copy(dst Writer, src Reader) (written int64, err error)
```

```
type Conn interface {
    Read(b []byte) (n int, err error)
    Write(b []byte) (n int, err error)
    Close() error
    // ... some additional methods omitted ...
}
```

```
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

Goroutines

Goroutines are lightweight threads that are managed by the Go runtime. To run a function in a new goroutine, just put "go" before the function call.

```
package main

import (
    "fmt"
    "time"
)

func main() {
    go say("let's go!", 3)
    go say("ho!", 2)
    go say("hey!", 1)
    time.Sleep(4 * time.Second)
}

func say(text string, secs int) {
    time.Sleep(time.Duration(secs) * time.Second)
    fmt.Println(text)
}
```

Run

14

A concurrent echo server

```
package main

import (
    "io"
    "log"
    "net"
)

const listenAddr = "localhost:4000"

func main() {
    l, err := net.Listen("tcp", listenAddr)
    if err != nil {
        log.Fatal(err)
    }
    for {
        c, err := l.Accept()
        if err != nil {
            log.Fatal(err)
        }
        go io.Copy(c, c)
    }
}
```

Run

15

"Chat roulette"

In this talk we'll look at a simple program, based on the popular "chat roulette" site.

In short:

- a user connects,
- another user connects,
- everything one user types is sent to the other.

16

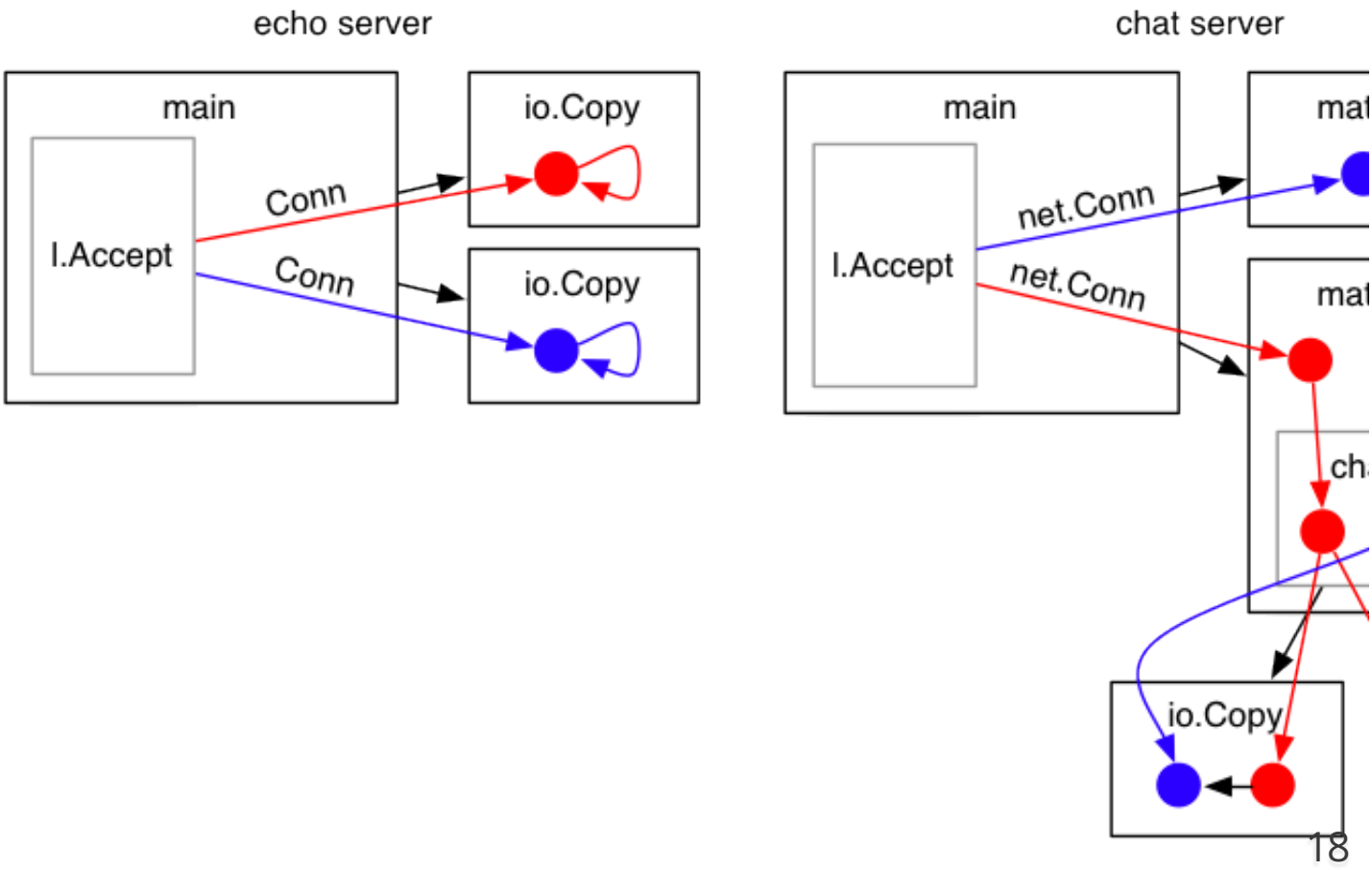
Design

The chat program is similar to the echo program. With echo, we copy a connection's incoming data back to the same connection.

For chat, we must copy the incoming data from one user's connection to another's.

Copying the data is easy. As in real life, the hard part is matching one partner with another.

Design diagram



Channels

Goroutines communicate via channels. A channel is a typed conduit that may be synchronous (unbuffered) or asynchronous (buffered).

```
package main

import "fmt"

func main() {
    ch := make(chan int)
    go fibs(ch)
    for i := 0; i < 20; i++ {
        fmt.Println(<-ch)
    }
}

func fibs(ch chan int) {
    i, j := 0, 1
    for {
        ch <- j
        i, j = j, i+j
    }
}
```

Run

19

Select

A select statement is like a switch, but it selects over channel operations (and chooses exactly one of them).

```
package main

import (
    "fmt"
    "time"
)

func main() {
    ticker := time.NewTicker(time.Millisecond * 250)
    boom := time.After(time.Second * 1)
    for {
        select {
        case <-ticker.C:
            fmt.Println("tick")
        case <-boom:
            fmt.Println("boom!")
            return
        }
    }
}
```

Run

20

Modifying echo to create chat

In the accept loop, we replace the call to `io.Copy`:

```
    for {
        c, err := l.Accept()
        if err != nil {
            log.Fatal(err)
        }
        go io.Copy(c, c)
    }
```

with a call to a new function, `match`:

```
    for {
        c, err := l.Accept()
        if err != nil {
            log.Fatal(err)
        }
        go match(c)
    }
```

The matcher

The `match` function simultaneously tries to send and receive a connection on a channel.

- If the send succeeds, the connection has been handed off to another goroutine, so the function exits and the goroutine shuts down.
- If the receive succeeds, a connection has been received from another goroutine. The current goroutine then has two connections, so it starts a chat session between them.

```
var partner = make(chan io.ReadWriteCloser)

func match(c io.ReadWriteCloser) {
    fmt.Fprint(c, "Waiting for a partner...")
    select {
    case partner <- c:
        // now handled by the other goroutine
    case p := <-partner:
        chat(p, c)
    }
}
```

22

The conversation

The chat function sends a greeting to each connection and then copies data from one to the other, and vice versa.

Notice that it launches another goroutine so that the copy operations may happen concurrently.

```
func chat(a, b io.ReadWriteCloser) {
    fmt.Fprintln(a, "Found one! Say hi.")
    fmt.Fprintln(b, "Found one! Say hi.")
    go io.Copy(a, b)
    io.Copy(b, a)
}
```

23

Demo

24

Error handling

It's important to clean up when the conversation is over. To do this we send the error value from each `io.Copy` call to a channel, log any non-nil errors, and close both connections.

```
func chat(a, b io.ReadWriteCloser) {
    fmt.Fprintln(a, "Found one! Say hi.")
    fmt.Fprintln(b, "Found one! Say hi.")
    errc := make(chan error, 1)
    go cp(a, b, errc)
    go cp(b, a, errc)
    if err := <-errc; err != nil {
        log.Println(err)
    }
    a.Close()
    b.Close()
}
```

```
func cp(w io.Writer, r io.Reader, errc chan<- error) {
    _, err := io.Copy(w, r)
    errc <- err
}
```

25

Demo

26

Taking it to the web

"Cute program," you say, "But who wants to chat over a raw TCP connection?"

Good point. Let's modernize it by turning it a web application.

Instead of TCP sockets, we'll use websockets.

We'll serve the user interface with Go's standard `net/http` package, and websocket support is provided by the `websocket` package from the `go.net` sub-repository,

Hello, web

```
package main

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

const listenAddr = "localhost:4000"

func main() {
    http.HandleFunc("/", handler)
    err := http.ListenAndServe(listenAddr, nil)
    if err != nil {
        log.Fatal(err)
    }
}

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprint(w, "Hello, web")
}
```

Run

28

Hello, WebSocket

```
var sock = new WebSocket("ws://localhost:4000/");
sock.onmessage = function(m) { console.log("Received:", m.data); }
sock.send("Hello!\n")
```

```
package main

import (
    "fmt"
    "golang.org/x/net/websocket"
    "net/http"
)

func main() {
    http.Handle("/", websocket.Handler(handler))
    http.ListenAndServe("localhost:4000", nil)
}

func handler(c *websocket.Conn) {
    var s string
    fmt.Fscan(c, &s)
    fmt.Println("Received:", s)
    fmt.Fprint(c, "How do you do?")
}
```

Run

29

Using the http and websocket packages

```
package main

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

    "golang.org/x/net/websocket"
)

const listenAddr = "localhost:4000"

func main() {
    http.HandleFunc("/", rootHandler)
    http.Handle("/socket", websocket.Handler(socketHandler))
    err := http.ListenAndServe(listenAddr, nil)
    if err != nil {
        log.Fatal(err)
    }
}
```

30

Serving the HTML and JavaScript

```
import "html/template"
```

```
func rootHandler(w http.ResponseWriter, r *http.Request) {
    rootTemplate.Execute(w, listenAddr)
}

var rootTemplate = template.Must(template.New("root").Parse(`
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<script>
```

```
    websocket = new WebSocket("ws://{{.}}/socket");
    websocket.onmessage = onMessage;
    websocket.onclose = onClose;
```

```
</html>
`))
```

31

Adding a socket type

We can't just use a `websocket.Conn` instead of the `net.Conn`, because a `websocket.Conn` is held open by its handler function. Here we use a channel to keep the handler running until the socket's `Close` method is called.

```
type socket struct {
    conn *websocket.Conn
    done chan bool
}

func (s socket) Read(b []byte) (int, error)  { return s.conn.Read(b)
func (s socket) Write(b []byte) (int, error) { return s.conn.Write(b

func (s socket) Close() error {
    s.done <- true
    return nil
}

func socketHandler(ws *websocket.Conn) {
    s := socket{conn: ws, done: make(chan bool)}
    go match(s)
    <-s.done
}
```

32

Struct embedding

Go supports a kind of "mix-in" functionality with a feature known as "struct embedding". The embedding struct delegates calls to the embedded type's methods.

```
type A struct{}

func (A) Hello() {
    fmt.Println("Hello!")
}

type B struct {
    A
}

// func (b B) Hello() { b.A.Hello() } // (implicitly!)

func main() {
    var b B
    b.Hello()
}
```

Run

33

Embedding the websocket connection

By embedding the *websocket.Conn as an io.ReadWriter, we can drop the explicit socket Read and Write methods.

```
type socket struct {
    io.ReadWriter
    done          chan bool
}

func (s socket) Close() error {
    s.done <- true
    return nil
}

func socketHandler(ws *websocket.Conn) {
    s := socket{ws, make(chan bool)}
    go match(s)
    <-s.done
}
```

34

Demo

35

Relieving loneliness

What if you connect, but there's noone there?

Wouldn't it be nice if we could synthesize a chat partner?

Let's do it.

Generating text with markov chains

```
Source
"I am not a number! I am a free man!"

Prefix           Suffix
"" ""            "I"
"" "I"           "am"
"I" "am"         "a"
"I" "am"         "not"
"a" "free"       "man!"
"am" "a"         "free"
"am" "not"       "a"
"a" "number!"    "I"
"number!" "I"    "am"
"not" "a"        "number!"

Generated sentences beginning with the prefix "I am"
"I am a free man!"
"I am not a number! I am a free man!"
"I am not a number! I am not a number! I am a free man!"
"I am not a number! I am not a number! I am not a number! I am a free
```

37

Generating text with markov chains

Fortunately, the Go docs include a markov chain implementation:

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

We'll use a version that has been modified to be safe for concurrent use.

```
// Chain contains a map ("chain") of prefixes to a list of suffixes.
// A prefix is a string of prefixLen words joined with spaces.
// A suffix is a single word. A prefix can have multiple suffixes.
type Chain struct {
```

```
// Write parses the bytes into prefixes and suffixes that are stored
func (c *Chain) Write(b []byte) (int, error) {
```

```
// Generate returns a string of at most n words generated from Chain
func (c *Chain) Generate(n int) string {
```

38

Feeding the chain

We will use all text that enters the system to build the markov chains.
To do this we split the socket's `ReadWriter` into a `Reader` and a `Writer`,
and feed all incoming data to the `Chain` instance.

```
type socket struct {
    io.Reader
    io.Writer
    done chan bool
}
```

```
var chain = NewChain(2) // 2-word prefixes

func socketHandler(ws *websocket.Conn) {
    r, w := io.Pipe()
    go func() {
        _, err := io.Copy(io.MultiWriter(w, chain), ws)
        w.CloseWithError(err)
    }()
    s := socket{r, ws, make(chan bool)}
    go match(s)
    <-s.done
}
```

39

The markov bot

```
// Bot returns an io.ReadWriteCloser that responds to
// each incoming write with a generated sentence.
func Bot() io.ReadWriteCloser {
    r, out := io.Pipe() // for outgoing data
    return bot{r, out}
}
```

```
type bot struct {
    io.ReadCloser
    out io.Writer
}
```

```
func (b bot) Write(buf []byte) (int, error) {
    go b.speak()
    return len(buf), nil
}
```

```
func (b bot) speak() {
    time.Sleep(time.Second)
    msg := chain.Generate(10) // at most 10 words
    b.out.Write([]byte(msg))
}
```

40

Integrating the markov bot

The bot should jump in if a real partner doesn't join.
To do this, we add a case to the select that triggers after 5 seconds, starting a chat between the user's socket and a bot.

```
func match(c io.ReadWriteCloser) {
    fmt.Fprint(c, "Waiting for a partner...")
    select {
    case partner <- c:
        // now handled by the other goroutine
    case p := <-partner:
        chat(p, c)
    case <-time.After(5 * time.Second):
        chat(Bot(), c)
    }
}
```

The chat function remains untouched.

Demo

42

One more thing

43

TCP and HTTP at the same time

```
func main() {
    go netListen()
    http.HandleFunc("/", rootHandler)
    http.Handle("/socket", websocket.Handler(socketHandler))
    err := http.ListenAndServe(listenAddr, nil)
    if err != nil {
        log.Fatal(err)
    }
}
```

```
func netListen() {
    l, err := net.Listen("tcp", "localhost:4001")
    if err != nil {
        log.Fatal(err)
    }
    for {
        c, err := l.Accept()
        if err != nil {
            log.Fatal(err)
        }
        go match(c)
    }
}
```

44

Demo

45

Discussion

46

Further reading

All about Go:

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

The slides for this talk:

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

"Go Concurrency Patterns" by Rob Pike:

golang.org/s/concurrency-patterns (http://golang.org/s/concurrency-patterns) 47

Thank you

Andrew Gerrand
Google Sydney
http://andrewgerrand.com (http://andrewgerrand.com)
@enneff (http://twitter.com/enneff)
http://golang.org (http://golang.org)