Google Go: an introduction to the language

8 Feb 2013

Video

These slides are based on a talk by Andrew Gerrand.

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

Watch the talk on Vimeo (http://vimeo.com/53221558)

Background

Why a new language?

Motivated by Google's needs:

- Efficiency
- Safety
- Concurrency
- Scalability
- Fast development cycle
- No surprises

History

- Project starts at Google in 2007 (by Griesemer, Pike, Thompson)
- Open source release in November 2009
- More than 250 contributors join the project
- Version 1.0 release in March 2012

Design

"Consensus drove the design. Nothing went into the language until [Ken Thompson, Robert Griesemer, and myself] all agreed that it was right. Some features didn't get resolved until after a year or more of discussion." - Rob Pike

Go is:

- Lightweight, avoids unnecessary repetition
- Object Oriented, but not in the usual way
- Concurrent, in a way that keeps you sane
- Designed for working programmers

The web is the best up-to-date reference

web site:

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

- it out even if the web design is a bit "geeky"
- have a look at the
- browse through the to get a feel
- Go Nuts
- Go

Go 1

Released in March 2012

A specification of the language and libraries that will be supported for years.

The guarantee: code written for Go 1.0 will build and run with Go 1.x.

"Best thing we ever did."

The gopher

Go is a trade-off between dynamically and statically-typed languages

The "Java-like" side

- compiled
- runtime and garbage collector

but:

- no runtime installation or byte-code (not running in the background)
- no classes
- no inheritance
- no generics

The "functional" side

- functions are first-class objects (closures etc.)
- interfaces are inferred by the compiler and never explicit
- simple and intuitive APIs
- compilation units and fast parsing

but:

- much faster
- the compiler makes things safer

Getting started

Install Go

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

- Install from binary distributions or build from source
- 32- and 64-bit x86 and ARM processors
- Windows, Mac OS X, Linux, and FreeBSD
- Other platforms may be supported by gccgo

The go tool

The go tool is the standard tool for building, testing, and installing Go programs.

Compile and run `hello.go`:

```
$ go run hello.go
```

Run `zip` tests:

```
$ go test archive/zip
```

Build and format the files in the current directory:

```
$ go build
$ go fmt
```

Fetch and install `websocket`:

```
$ go get code.google.com/p/go.net/websocket
```

Workspaces

The go tool derives build instructions from Go source code.

There's no need to write and maintain build scripts.

For this to work, some prescribed directory structure, known as a workspace, is required.

```
workspace/
  bin # executable binaries
  pkg # compiled object files
  src # source code
```

Create a workspace

Create your workspace now.

```
$ mkdir -p $HOME/gocode/src
```

(The `bin` and `pkg` sub-directories will be created by the `go` tool.)

Tell the `go` tool where your workspace is by setting the `GOPATH` environment variable:

```
export GOPATH=$HOME/gocode
```

You may also want to add the `bin` sub-directory of your workspace to your `PATH`:

```
export PATH=$PATH:$GOPATH/bin
```

This lets you run your Go programs without specifying their full path.

(You may want to put these `export` commands in the `.bash_profile` file in your home directory.)

Choose a namespace

Choose a special place for your Go code.

I use "`github.com/mezzato`", the root of my GitHub account (useful with `go get`).

```
$ mkdir -p $GOPATH/src/github.com/mezzato/go.talks/hello
```

Create a `hello` directory in your namespace and copy `hello.go` there:

```
$ mkdir $GOPATH/src/github.com/mezzato/go.talks/hello
$ cp hello.go $GOPATH/src/github.com/mezzato/go.talks/hello
```

Now you can build install the hello program with the go tool:

```
$ go install github.com/mezzato/go.talks/hello
```

This builds an executable named `hello`, and installs it to the `bin` directory of your workspace.

```
$ $GOPATH/bin/hello
Hello World
```

Before coding an IDE to make life easier

There are many:

geekmonkey.org/articles/20-comparison-of-ides-for-google-go (http://geekmonkey.org/articles/20-comparison-of-ides-for-google-go)

What I like:

- goclipse for Eclipse

  code.google.com/p/goclipse/ (http://code.google.com/p/goclipse/)

- liteide for debugging

  code.google.com/p/liteide/ (http://code.google.com/p/liteide/)

An example

An example: the command line version

Hello World

```
package main

import "fmt"

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

Run

On these slides just in a window; edit if you like and click on **Run**

An example: the web version

A command-line program that starts up a web site and writes "Hello World" when a GET request comes in.

The program will:

- start up a web site
- reply with the HTTP body "Hello World"

To get started, create directory inside your namespace called `hello-web`:

```
$ mkdir $GOPATH/src/github.com/mezzato/go.talks/hello-web
```

This is where you will put your Go source files.

Hello, go on the web

```
package main

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

type Greeting string

func (g Greeting) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    fmt.Fprint(w, g)
}

func main() {
    err := http.ListenAndServe("localhost:4000", Greeting("Hello World"))
    if err != nil {
        log.Fatal(err)
    }
}
```

Run

Hello World on the web: package statement

All Go code belongs to a package.

```
package main
```

Go programs begin with function `main` inside package `main`.

Hello World on the web: import statement

The import declaration specifies the file's dependencies.

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

Each string is an import path. It tells the Go tools where to find the package.

These packages are all from the Go standard library.

Hello World on the web: the main function

```
func main() {
    err := http.ListenAndServe("localhost:4000", Greeting("Hello World"))
    if err != nil {
        log.Fatal(err)
    }
}
```

The main function takes no arguments and has no return values.

Hello World on the web: a custom type

```
type Greeting string
```

This is a type declaration. The type is the same as the string type but is renamed to be extended with methods (called receivers).

Hello World on the web: extending a type with methods and implementing interfaces for free

```
type Greeting string

func (g Greeting) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    fmt.Fprint(w, g)
}
```

- The method is attached to the type `Greeting`
- This is why we had to call the new string type `Greeting` other than `string`
- The compiler recognizes that the type implements the interface **http.Handler**:

which exposes just one method:

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

Hello World on the web: error handling

```
func main() {
    err := http.ListenAndServe("localhost:4000", Greeting("Hello World"))
    if err != nil {
        log.Fatal(err)
    }
}
```

Compare `err` against `nil`, the zero-value for the built-in `error` type.

The `err` variable will be nil if the request was successful.

If not, call the `log.Fatal` function to print the error message and exit the program.

Hello World on the web: a struct now

```
package main

import (
    "fmt"
    "log"
    "net/http"
)
type Greeting struct {
    Message string
    Count   int
}
func (g *Greeting) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    g.Count++
    m := fmt.Sprintf("%s. Fired %d times.", g.Message, g.Count) // or var m string = ...
    //fmt.Printf("Full url: %s\n", r.URL)
    fmt.Fprint(w, m)
}
func main() {
    var g *Greeting = &Greeting{Message: "Hello World"} // or new(Greeting)...
    err := http.ListenAndServe("localhost:4000", g)
    if err != nil {
        log.Fatal(err)
    }
}
```

Run

Hello World on the web: extensible types but no classes

- **Struct** are value types, they get copied when passed along
- These **struct-types** are **created but not initialized** with the keyword. Everything is zeroed
- There are pointers for this reason

There are other types

There are **other types** which are **automatically "referenced"**, memory is allocated with
in this case.
The variable is initialized as well in this case.

They are the types:
slice (~ ArrayList)
map (~ HashMap)
channel (this is new)

Abount channels... the modern (or at least recycled) side

- cross-platform
- modern functionality (networking, websockets, JSON e.g.)
- orthogonality, one way of doing things
- **concurrent**

Rob Pike, Go Concurrency Patterns

Video

This talk was presented at Google I/O in June 2012.

Watch the talk on YouTube (http://www.youtube.com/watch?v=f6kdp27TYZs)

The following slides are an excerpt of the beginning.

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)

A model for software construction

Easy to understand.

Easy to use.

Easy to reason about.

You don't need to be an expert!

(Much nicer than dealing with the minutiae of parallelism (threads, semaphores, locks, barriers, etc.))

History

To many, the concurrency features of Go seemed new.

But they are rooted in a long history, reaching back to Hoare's CSP in 1978 and even Dijkstra's guarded commands (1975).

Languages with similar features:

- Occam (May, 1983)
- Erlang (Armstrong, 1986)
- Newsqueak (Pike, 1988)
- Concurrent ML (Reppy, 1993)
- Alef (Winterbottom, 1995)
- Limbo (Dorward, Pike, Winterbottom, 1996).

Distinction

Go is the latest on the Newsqueak-Alef-Limbo branch, distinguished by first-class channels.

Erlang is closer to the original CSP, where you communicate to a process by name rather than over a channel.

The models are equivalent but express things differently.

Rough analogy: writing to a file by name (process, Erlang) vs. writing to a file descriptor (channel, Go).

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)
    }
}
```

Run

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)
    }
}
```

Run

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)
    }
}
```

Run

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!")
}
```

Run

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.")
}
```

Run

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.")
}
```

Run

```
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 valu
e.
        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.

TO RECAP: When use Go? Still missing

- When should I consider using Go?

networking,
concurrency,
systems language,
speed,
multi-platform

- It would be nice to see:

a nice web framework
a cross-platform desktop framework

"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.")
```

Run

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

Run

Multiplexing

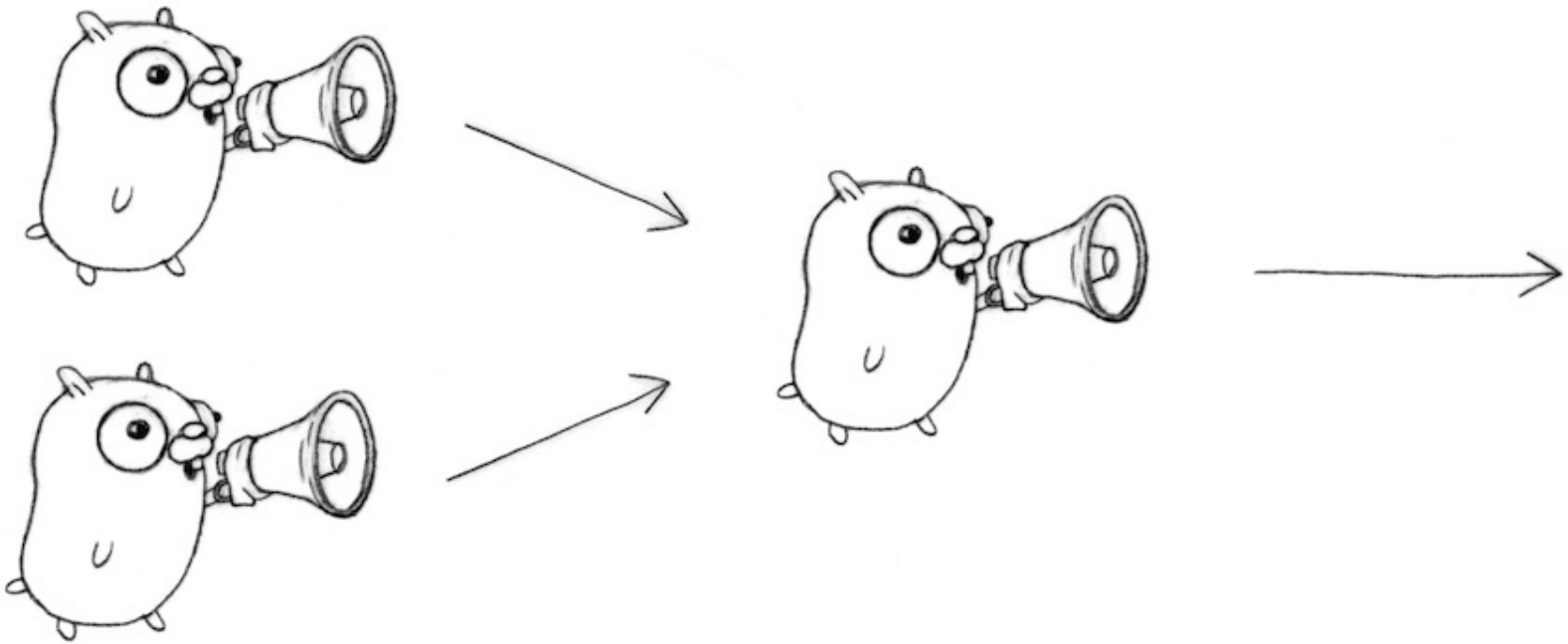
These programs make Joe and Ann count in lockstep.
We can instead use a fan-in function to let whosoever is ready talk.

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

Run

Fan-in



Standard library

Packages

Go code lives in packages.

Packages contain type, function, variable, and constant declarations.

Packages can be very small (package `errors` has just one declaration) or very large (package `net/http` has >100 declarations). Most are somewhere in between.

Case determines visibility: `Foo` is exported, `foo` is not

Tools

The go tool

The go tool is the de facto standard for building and installing Go code.

Compile and run a single-file program:

```
$ go run hello.go
```

Build and install the package in the current directory (and its dependencies):

```
$ go install
```

Build and install the `fmt` package (and its dependencies):

```
$ go install fmt
```

This tool also acts as an interface for most of the Go tools.

Import paths

The go tool is a "zero configuration" tool. No Makefiles or scripts. Just Go code.
Your build schema and code are always in sync; they are one and the same.

Package import paths mirror the code's location in the file system:

```
src/
  github.com/nf/
    gosynth/
      main.go
      note.go
      osc.go
    wav/
      writer.go
```

The gosynth program imports the wav package:

```
import "github.com/nf/wav"
```

Installing gosynth will automatically install the wav package:

```
$ go install github.com/nf/gosynth
```

Remote dependencies

The go tool also fetches Go code from remote repositories.

Import paths can be URLs:

```
import "code.google.com/p/go.net/websocket"
```

To fetch, build and install a package:

```
$ go get code.google.com/p/go.net/websocket
```

To fetch, build, and install gosynth and its dependencies:

```
$ go get github.com/nf/gosynth
```

This simple design leads to other cool tools:

go.pkgdoc.org (http://go.pkgdoc.org)

Godoc

Godoc extracts documentation from Go code and presents it in a variety of forms.

Comments need no special format, they just need to precede what they document.

```
// Split slices s into all substrings separated by sep and returns a slice of
// the substrings between those separators.
// If sep is empty, Split splits after each UTF-8 sequence.
// It is equivalent to SplitN with a count of -1.
func Split(s, sep string) []string {
```

func Split

```
func Split(s, sep string) []string
```

Split slices s into all substrings separated by sep and returns a slice of the substrings between those separators. If sep is empty, Split splits after each UTF-8 sequence. It is equivalent to SplitN with a count of -1.

Documentation that lives with code is easy to keep up-to-date.

Gofmt

The `gofmt` tool is a pretty-printer for Go source code.

All Go code in the core is gofmt'd, as is ~70% of open source Go code.

Ends boring formatting discussions.

Improves readability. Improves writability.

Saves a amount of time.

Tests: writing

The go tool and the `testing` package provide a lightweight test framework.

```
func TestIndex(t *testing.T) {
    var tests = []struct {
        s   string
        sep string
        out int
    }{
        {"", "", 0},
        {"", "a", -1},
        {"fo", "foo", -1},
        {"foo", "foo", 0},
        {"oofofoofooo", "f", 2},
        // etc
    }
    for _, test := range tests {
        actual := strings.Index(test.s, test.sep)
        if actual != test.out {
            t.Errorf("Index(%q,%q) = %v; want %v", test.s, test.sep, actual, test.out)
        }
    }
}
```

Tests: running

The go tool runs tests.

```
$ go test
PASS

$ go test -v
=== RUN TestIndex
--- PASS: TestIndex (0.00 seconds)
PASS
```

To run the tests for all my projects:

```
$ go test github.com/nf/...
```

Tests: benchmarks

The `testing` package also supports benchmarks.

A sample benchmark function:

```
func BenchmarkIndex(b *testing.B) {
    const s = "some_text=some☺value"
    for i := 0; i < b.N; i++ {
        strings.Index(s, "v")
    }
}
```

The benchmark package will vary `b.N` until the benchmark function lasts long enough to be timed reliably.

```
$ go test -test.bench=Index
PASS
BenchmarkIndex    50000000            37.3 ns/op
```

Tests: doc examples

The `testing` package also supports testable examples.

```
func ExampleIndex() {
    fmt.Println(strings.Index("chicken", "ken"))
    fmt.Println(strings.Index("chicken", "dmr"))
    // Output:
    // 4
    // -1
}
```

Examples and built and run as part of the normal test suite:

```
$ go test -v
=== RUN: ExampleIndex
--- PASS: ExampleIndex (0.00 seconds)
PASS
```

The example is displayed in godoc alongside the thing it demonstrates:

golang.org/pkg/strings/#Index (http://golang.org/pkg/strings/#Index)

And there's more

- `vet`: checks code for common programmer mistakes
- `pprof`: CPU and memory profiling
- `fix`: automatically migrate code as APIs change
- GDB support
- Editor support: Vim, Emacs, Eclipse, Sublime Text

Some package examples

io

The `io` package provides fundamental I/O interfaces that are used throughout most Go code.

The most ubiquitous are the `Reader` and `Writer` types, which describe streams of data.

```
package io

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

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

`Reader` and `Writer` implementations include files, sockets, (de)compressors, image and JSON codecs, and many more.

net/http

The net/http package implements an HTTP server and client.

```
package main

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

type Greeting string

func (g Greeting) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    fmt.Fprint(w, g)
}

func main() {
    err := http.ListenAndServe("localhost:4000", Greeting("Hello World"))
    if err != nil {
        log.Fatal(err)
    }
}
```

Run

encoding/json

The encoding/json package converts JSON-encoded data to and from native Go data structures.

```
const blob = `[
    {"Title":"Øredev", "URL":"http://oredev.org"},
    {"Title":"Strange Loop", "URL":"http://thestrangeloop.com"}
]`

type Item struct {
    Title string
    URL   string
}

func main() {
    var items []*Item
    json.NewDecoder(strings.NewReader(blob)).Decode(&items)
    for _, item := range items {
        fmt.Printf("Title: %v URL: %v\n", item.Title, item.URL)
    }
}
```

Run

time

The `time` package provides a representation of time and duration, and other time-related functions.

```
if time.Now().Hour() < 12 {
    fmt.Println("Good morning.")
} else {
    fmt.Println("Good afternoon (or evening).")
}
```

Run

```
birthday, _ := time.Parse("Jan 2 2006", "Nov 10 2009") // time.Time
age := time.Since(birthday)                            // time.Duration
fmt.Printf("Go is %d days old\n", age/(time.Hour*24))
```

Run

`time.Time` values also contain a `time.Location` (for display only):

```
t := time.Now()
fmt.Println(t.In(time.UTC))
home, _ := time.LoadLocation("Australia/Sydney")
fmt.Println(t.In(home))
```

Run

flag

The `flag` package provides a simple API for parsing command-line flags.

```
package main

import (
    "flag"
    "fmt"
    "time"
)

var (
    message = flag.String("message", "Hello!", "what to say")
    delay   = flag.Duration("delay", 2*time.Second, "how long to wait")
)

func main() {
    flag.Parse()
    fmt.Println(*message)
    time.Sleep(*delay)
}
```

Run

```
$ flag -message 'Hold on...' -delay 5m
```

An example

Webfront

`Webfront` is an HTTP server and reverse proxy.

It reads a JSON-formatted rule file like this:

```
[
    {"Host": "example.com", "Serve": "/var/www"},
    {"Host": "example.org", "Forward": "localhost:8080"}
]
```

For all requests to the host `example.com` (or any name ending in `".example.com"`) it serves files from the `/var/www` directory.

For requests to `example.org`, it forwards the request to the HTTP server listening on localhost port 8080.

The Rule type

A `Rule` value specifies what to do for a request to a specific host.

```
// Rule represents a rule in a configuration file.
type Rule struct {
    Host    string // to match against request Host header
    Forward string // non-empty if reverse proxy
    Serve   string // non-empty if file server
}
```

It corresponds directly with the entries in the JSON configuration file.

```
[
    {"Host": "example.com", "Serve": "/var/www"},
    {"Host": "example.org", "Forward": "localhost:8080"}
]
```

Rule methods

```
// Match returns true if the Rule matches the given Request.
func (r *Rule) Match(req *http.Request) bool {
    return req.Host == r.Host || strings.HasSuffix(req.Host, "."+r.Host)
}
```

```
// Handler returns the appropriate Handler for the Rule.
func (r *Rule) Handler() http.Handler {
    if h := r.Forward; h != "" {
        return &httputil.ReverseProxy{
            Director: func(req *http.Request) {
                req.URL.Scheme = "http"
                req.URL.Host = h
            },
        }
    }
    if d := r.Serve; d != "" {
        return http.FileServer(http.Dir(d))
    }
    return nil
}
```

The Server type

The Server type is responsible for loading (and refreshing) the rules from the rule file and serving HTTP requests with the appropriate handler.

```
// Server implements an http.Handler that acts as either a reverse proxy or
// a simple file server, as determined by a rule set.
type Server struct {
    mu    sync.RWMutex // guards the fields below
    mtime time.Time    // when the rule file was last modified
    rules []*Rule
}
```

```
// ServeHTTP matches the Request with a Rule and, if found, serves the
// request with the Rule's handler.
func (s *Server) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    if h := s.handler(r); h != nil {
        h.ServeHTTP(w, r)
        return
    }
    http.Error(w, "Not found.", http.StatusNotFound)
}
```

The handler method

```
// handler returns the appropriate Handler for the given Request,
// or nil if none found.
func (s *Server) handler(req *http.Request) http.Handler {
    s.mu.RLock()
    defer s.mu.RUnlock()
    for _, r := range s.rules {
        if r.Match(req) {
            return r.Handler()
        }
    }
    return nil
}
```

Parsing rules

The `parseRules` function uses the `encoding/json` package to read the rule file into a Go data structure.

```
// parseRules reads rule definitions from file returns the resultant Rules.
func parseRules(file string) ([]*Rule, error) {
    f, err := os.Open(file)
    if err != nil {
        return nil, err
    }
    defer f.Close()
    var rules []*Rule
    err = json.NewDecoder(f).Decode(&rules)
    if err != nil {
        return nil, err
    }
    return rules, nil
}
```

The loadRules method

```
// loadRules tests whether file has been modified
// and, if so, loads the rule set from file.
func (s *Server) loadRules(file string) error {
    fi, err := os.Stat(file)
    if err != nil {
        return err
    }
    mtime := fi.ModTime()
    if mtime.Before(s.mtime) && s.rules != nil {
        return nil // no change
    }
    rules, err := parseRules(file)
    if err != nil {
        return fmt.Errorf("parsing %s: %v", file, err)
    }
    s.mu.Lock()
    s.mtime = mtime
    s.rules = rules
    s.mu.Unlock()
    return nil
}
```

Constructing the server

```
// NewServer constructs a Server that reads rules from file with a period
// specified by poll.
func NewServer(file string, poll time.Duration) (*Server, error) {
    s := new(Server)
    if err := s.loadRules(file); err != nil {
        return nil, err
    }
    go s.refreshRules(file, poll)
    return s, nil
}
```

This constructor function launches a goroutine running the `refreshRules` method.

Refreshing the rules

```
// refreshRules polls file periodically and refreshes the Server's rule
// set if the file has been modified.
func (s *Server) refreshRules(file string, poll time.Duration) {
    for {
        if err := s.loadRules(file); err != nil {
            log.Println(err)
        }
        time.Sleep(poll)
    }
}
```

Bringing it all together

The main function parses command-line flags, constructs a `Server`, and launches an HTTP server that serves all requests with the `Server`.

```
var (
    httpAddr     = flag.String("http", ":80", "HTTP listen address")
    ruleFile     = flag.String("rules", "", "rule definition file")
    pollInterval = flag.Duration("poll", time.Second*10, "file poll interval")
)

func main() {
    flag.Parse()

    s, err := NewServer(*ruleFile, *pollInterval)
    if err != nil {
        log.Fatal(err)
    }

    err = http.ListenAndServe(*httpAddr, s)
    if err != nil {
        log.Fatal(err)
    }
}
```

Demo

Testing (1/3)

The `Server` integration test uses the `httptest` package to construct a dummy HTTP server, synthesizes a set of rules, and constructs a `Server` instance that uses those rules.

```
func testHandler(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("OK"))
}

func TestServer(t *testing.T) {
    dummy := httptest.NewServer(http.HandlerFunc(testHandler))
    defer dummy.Close()

    ruleFile := writeRules([]*Rule{
        {Host: "example.com", Forward: dummy.Listener.Addr().String()},
        {Host: "example.org", Serve: "testdata"},
    })
    defer os.Remove(ruleFile)

    s, err := NewServer(ruleFile, time.Hour)
    if err != nil {
        t.Fatal(err)
    }
    // continued next slide
```

Testing (2/3)

Each test case in the table specifies a request URL and the expected response code and body.

```
    // continued from previous slide

    var tests = []struct {
        url  string
        code int
        body string
    }{
        {"http://example.com/", 200, "OK"},
        {"http://foo.example.com/", 200, "OK"},
        {"http://example.org/", 200, "contents of index.html\n"},
        {"http://example.net/", 404, "Not found.\n"},
        {"http://fooexample.com/", 404, "Not found.\n"},
    }

    // continued next slide
```

Testing (3/3)

For each test case, construct an `http.Request` for the url and an `httptest.ResponseRecorder` to capture the response, and pass them to the `Server.ServeHTTP` method. Then check that the response matches the test case.

```
    // continued from previous slide

    for _, test := range tests {
        req, _ := http.NewRequest("GET", test.url, nil)
        rw := httptest.NewRecorder()
        rw.Body = new(bytes.Buffer)
        s.ServeHTTP(rw, req)
        if g, w := rw.Code, test.code; g != w {
            t.Errorf("%s: code = %d, want %d", test.url, g, w)
        }
        if g, w := rw.Body.String(), test.body; g != w {
            t.Errorf("%s: body = %q, want %q", test.url, g, w)
        }
    }
}
```

Demo

Conclusions

Further reading

All about Go:

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

Thank you

8 Feb 2013