



FREE eBook

LEARNING Go

Free unaffiliated eBook created from
Stack Overflow contributors.

#go

www.dbooks.org

Table of Contents

About.....	1
Chapter 1: Getting started with Go	2
Remarks.....	2
Versions.....	2
The latest major version release is in bold below. Full release history can be found here.....	2
Examples.....	2
Hello, World!.....	2
Output:.....	3
FizzBuzz.....	3
Listing Go Environment Variables.....	4
Setting up the environment.....	4
GOPATH.....	4
GOBIN.....	5
GOROOT.....	5
Accessing Documentation Offline.....	5
Running Go online.....	6
The Go Playground.....	6
Sharing your code.....	6
In action.....	6
Chapter 2: Arrays.....	8
Introduction.....	8
Syntax.....	8
Examples.....	8
Creating arrays.....	8
Multidimensional Array.....	9
Array Indexes.....	10
Chapter 3: Base64 Encoding.....	12
Syntax.....	12
Remarks.....	12
Examples.....	12

Encoding.....	12
Encoding to a String.....	12
Decoding.....	12
Decoding a String.....	13
Chapter 4: Best practices on project structure.....	14
Examples.....	14
Restfull Projects API with Gin.....	14
controllers.....	14
core.....	15
libs.....	15
middlewares.....	15
public.....	16
h21.....	16
routers.....	16
services.....	18
main.go.....	18
Chapter 5: Branching.....	20
Examples.....	20
Switch Statements.....	20
If Statements.....	21
Type Switch Statements.....	22
Goto statements.....	23
Break-continue statements.....	23
Chapter 6: Build Constraints.....	25
Syntax.....	25
Remarks.....	25
Examples.....	25
Separate integration tests.....	25
Optimize implementations based on architecture.....	26
Chapter 7: cgo.....	27
Examples.....	27
Cgo: First steps tutorial.....	27

What.....	27
How.....	27
The example.....	27
Hello World!.....	28
Sum of ints.....	29
Generating a binary.....	30
Chapter 8: cgo.....	32
Examples.....	32
Calling C Function From Go.....	32
Wire C and Go code in all directions.....	33
Chapter 9: Channels.....	36
Introduction.....	36
Syntax.....	36
Remarks.....	36
Examples.....	36
Using range.....	36
Timeouts.....	37
Coordinating goroutines.....	37
Buffered vs unbuffered.....	38
Blocking & unblocking of channels.....	39
Waiting for work to finish.....	39
Chapter 10: Closures.....	41
Examples.....	41
Closure Basics.....	41
Chapter 11: Concurrency.....	43
Introduction.....	43
Syntax.....	43
Remarks.....	43
Examples.....	43
Creating goroutines.....	43
Hello World Goroutine.....	43
Waiting for goroutines.....	44

Using closures with goroutines in a loop.....	45
Stopping goroutines.....	45
Ping pong with two goroutines.....	46
Chapter 12: Console I/O.....	48
Examples.....	48
Read input from console.....	48
Chapter 13: Constants.....	50
Remarks.....	50
Examples.....	50
Declaring a constant.....	50
Multiple constants declaration.....	51
Typed vs. Untyped Constants.....	51
Chapter 14: Context.....	53
Syntax.....	53
Remarks.....	53
Further Reading.....	53
Examples.....	53
Context tree represented as a directed graph.....	53
Using a context to cancel work.....	54
Chapter 15: Cross Compilation.....	56
Introduction.....	56
Syntax.....	56
Remarks.....	56
Examples.....	57
Compile all architectures using a Makefile.....	57
Simple cross compilation with go build.....	58
Cross compilation by using gox.....	59
Installation.....	59
Usage.....	59
Simple Example: Compile helloworld.go for arm architecture on Linux machine.....	59
Chapter 16: Cryptography.....	60

Introduction.....	60
Examples.....	60
Encryption and decryption.....	60
Foreword.....	60
Encryption.....	60
Introduction and data.....	60
Step 1.....	61
Step 2.....	61
Step 3.....	61
Step 4.....	61
Step 5.....	62
Step 6.....	62
Step 7.....	62
Step 8.....	62
Step 9.....	62
Step 10.....	63
Decryption.....	63
Introduction and data.....	63
Step 1.....	63
Step 2.....	63
Step 3.....	63
Step 4.....	63
Step 5.....	64
Step 6.....	64
Step 7.....	64
Step 8.....	64
Step 9.....	64
Step 10.....	64
Chapter 17: Defer.....	66
Introduction.....	66
Syntax.....	66

Remarks.....	66
Examples.....	66
Defer Basics.....	66
Deferred Function Calls.....	68
Chapter 18: Developing for Multiple Platforms with Conditional Compiling.....	70
Introduction.....	70
Syntax.....	70
Remarks.....	70
Examples.....	71
Build tags.....	71
File suffix.....	71
Defining separate behaviours in different platforms.....	71
Chapter 19: Error Handling.....	73
Introduction.....	73
Remarks.....	73
Examples.....	73
Creating an error value.....	73
Creating a custom error type.....	74
Returning an error.....	75
Handling an error.....	76
Recovering from panic.....	77
Chapter 20: Executing Commands.....	79
Examples.....	79
Timing Out with Interrupt and then Kill.....	79
Simple Command Execution.....	79
Executing a Command then Continue and Wait.....	79
Running a Command twice.....	80
Chapter 21: File I/O.....	81
Syntax.....	81
Parameters.....	81
Examples.....	82
Reading and writing to a file using ioutil.....	82

Listing all the files and folders in the current directory	82
Listing all folders in the current directory	83
Chapter 22: Fmt	84
Examples	84
Stringer	84
Basic fmt	84
Format Functions	84
Print	85
Sprint	85
Fprint	85
Scan	85
Stringer Interface	85
Chapter 23: Functions	86
Introduction	86
Syntax	86
Examples	86
Basic Declaration	86
Parameters	86
Return Values	86
Named Return Values	87
Literal functions & closures	87
Variadic functions	89
Chapter 24: Getting Started With Go Using Atom	90
Introduction	90
Examples	90
Get, Install And Setup Atom & Gulp	90
Create \$GO_PATH/gulpfile.js	92
Create \$GO_PATH/mypackage/source.go	93
Creating \$GO_PATH/main.go	93
Chapter 25: gob	97
Introduction	97
Examples	97

How to encode data and write to file with gob?	97
How to read data from file and decode with go?	97
How to encode an interface with gob?	98
How to decode an interface with gob?	99
Chapter 26: Goroutines	101
Introduction	101
Examples	101
Goroutines Basic Program	101
Chapter 27: HTTP Client	103
Syntax	103
Parameters	103
Remarks	103
Examples	103
Basic GET	103
GET with URL parameters and a JSON response	104
Time out request with a context	105
1.7+	105
Before 1.7	105
Further Reading	106
PUT request of JSON object	106
Chapter 28: HTTP Server	108
Remarks	108
Examples	108
HTTP Hello World with custom server and mux	108
Hello World	108
Using a handler function	109
Create a HTTPS Server	111
Generate a certificate	111
The necessary Go code	112
Responding to an HTTP Request using Templates	112
Serving content using ServeMux	113
Handling http method, accessing query strings & request body	114

Chapter 29: Images	116
Introduction	116
Examples	116
Basic concepts	116
Image related type	117
Accessing image dimension and pixel	117
Loading and saving image	118
Save to PNG	119
Save to JPEG	119
Save to GIF	120
Cropping image	120
Convert color image to grayscale	121
Chapter 30: Inline Expansion	124
Remarks	124
Examples	124
Disabling inline expansion	124
Chapter 31: Installation	127
Examples	127
Install in Linux or Ubuntu	127
Chapter 32: Installation	128
Remarks	128
Downloading Go	128
Extracting the download files	128
Mac and Windows	128
Linux	128
Setting Environment Variables	129
Windows	129
Mac	129
Linux	129
Finished!	130
Examples	130

Example .profile or .bash_profile.....	130
Chapter 33: Interfaces.....	131
Remarks.....	131
Examples.....	131
Simple interface.....	131
Determining underlying type from interface.....	133
Compile-time check if a type satisfies an interface.....	133
Type switch.....	134
Type Assertion.....	134
Go Interfaces from a Mathematical Aspect.....	135
Chapter 34: Iota.....	137
Introduction.....	137
Remarks.....	137
Examples.....	137
Simple use of iota.....	137
Using iota in an expression.....	137
Skipping values.....	138
Use of iota in an expression list.....	138
Use of iota in a bitmask.....	138
Use of iota in const.....	139
Chapter 35: JSON.....	140
Syntax.....	140
Remarks.....	140
Examples.....	140
Basic JSON Encoding.....	140
Basic JSON decoding.....	141
Decoding JSON data from a file.....	142
Using anonymous structs for decoding.....	143
Configuring JSON struct fields.....	144
Hide/Skip Certain Fields.....	145
Ignore Empty Fields.....	145
Marshaling structs with private fields.....	145

Encoding/Decoding using Go structs.....	146
Encoding.....	146
Decoding.....	147
Chapter 36: JWT Authorization in Go.....	148
Introduction.....	148
Remarks.....	148
Examples.....	148
Parsing and validating a token using the HMAC signing method.....	148
Creating a token using a custom claims type.....	149
Creating, signing, and encoding a JWT token using the HMAC signing method.....	149
Using the StandardClaims type by itself to parse a token.....	150
Parsing the error types using bitfield checks.....	150
Getting token from HTTP Authorization header.....	151
Chapter 37: Logging.....	152
Examples.....	152
Basic Printing.....	152
Logging to file.....	152
Logging to syslog.....	153
Chapter 38: Loops.....	154
Introduction.....	154
Examples.....	154
Basic Loop.....	154
Break and Continue.....	154
Conditional loop.....	155
Different Forms of For Loop.....	155
Timed loop.....	158
Chapter 39: Maps.....	160
Introduction.....	160
Syntax.....	160
Remarks.....	160
Examples.....	160
Declaring and initializing a map.....	160

Creating a map.....	162
Zero value of a map.....	163
Iterating the elements of a map.....	163
Iterating the keys of a map.....	164
Deleting a map element.....	164
Counting map elements.....	165
Concurrent Access of Maps.....	165
Creating maps with slices as values.....	166
Check for element in a map.....	166
Iterating the values of a map.....	167
Copy a Map.....	167
Using a map as a set.....	168
Chapter 40: Memory pooling.....	169
Introduction.....	169
Examples.....	169
sync.Pool.....	169
Chapter 41: Methods.....	171
Syntax.....	171
Examples.....	171
Basic methods.....	171
Chaining methods.....	172
Increment-Decrement operators as arguments in Methods.....	172
Chapter 42: mgo.....	174
Introduction.....	174
Remarks.....	174
Examples.....	174
Example.....	174
Chapter 43: Middleware.....	176
Introduction.....	176
Remarks.....	176
Examples.....	176
Normal Handler Function.....	176

Middleware Calculate time required for handlerFunc to execute.....	176
CORS Middleware.....	177
Auth Middleware.....	177
Recovery Handler to prevent server from crashing.....	177
Chapter 44: Mutex.....	178
Examples.....	178
Mutex Locking.....	178
Chapter 45: Object Oriented Programming.....	179
Remarks.....	179
Examples.....	179
Structs.....	179
Embedded structs.....	179
Methods.....	180
Pointer Vs Value receiver.....	181
Interface & Polymorphism.....	182
Chapter 46: OS Signals.....	184
Syntax.....	184
Parameters.....	184
Examples.....	184
Assigning signals to a channel.....	184
Chapter 47: Packages.....	186
Examples.....	186
Package initialization.....	186
Managing package dependencies.....	186
Using different package and folder name.....	186
What's the use of this?.....	187
Importing packages.....	187
Chapter 48: Panic and Recover.....	190
Remarks.....	190
Examples.....	190
Panic.....	190
Recover.....	190

Chapter 49: Parsing Command Line Arguments And Flags	192
Examples	192
Command line arguments	192
Flags	192
Chapter 50: Parsing CSV files	194
Syntax	194
Examples	194
Simple CSV parsing	194
Chapter 51: Plugin	195
Introduction	195
Examples	195
Defining and using a plugin	195
Chapter 52: Pointers	196
Syntax	196
Examples	196
Basic Pointers	196
Pointer v. Value Methods	197
Pointer Methods	197
Value Methods	197
Dereferencing Pointers	199
Slices are Pointers to Array Segments	199
Simple Pointers	200
Chapter 53: Profiling using go tool pprof	201
Remarks	201
Examples	201
Basic cpu and memory profiling	201
Basic memory Profiling	201
Set CPU/Block profile rate	202
Using Benchmarks to Create Profile	202
Accessing Profile File	202
Chapter 54: Protobuf in Go	204

Introduction.....	204
Remarks.....	204
Examples.....	204
Using Protobuf with Go.....	204
Chapter 55: Readers.....	206
Examples.....	206
Using bytes.Reader to read from a string.....	206
Chapter 56: Reflection.....	207
Remarks.....	207
Examples.....	207
Basic reflect.Value Usage.....	207
Structs.....	207
Slices.....	208
reflect.Value.Elem().....	208
Type of value - package "reflect".....	208
Chapter 57: Select and Channels.....	210
Introduction.....	210
Syntax.....	210
Examples.....	210
Simple Select Working with Channels.....	210
Using select with timeouts.....	211
Chapter 58: Send/receive emails.....	213
Syntax.....	213
Examples.....	213
Sending Email with smtp.SendMail().....	213
Chapter 59: Slices.....	215
Introduction.....	215
Syntax.....	215
Examples.....	215
Appending to slice.....	215
Adding Two slices together.....	215
Removing elements / "Slicing" slices.....	215

Length and Capacity	217
Copying contents from one slice to another slice	218
Creating Slices	218
Filtering a slice	219
Zero value of slice	219
Chapter 60: SQL	221
Remarks	221
Examples	221
Querying	221
MySQL	221
Opening a database	222
MongoDB: connect & insert & remove & update & query	222
Chapter 61: String	225
Introduction	225
Syntax	225
Examples	225
String type	225
Formatting text	226
strings package	227
Chapter 62: Structs	229
Introduction	229
Examples	229
Basic Declaration	229
Exported vs. Unexported Fields (Private vs Public)	229
Composition and Embedding	230
Embedding	230
Methods	231
Anonymous struct	232
Tags	233
Making struct copies	233
Struct Literals	234
Empty struct	235

Chapter 63: Templates	237
Syntax	237
Remarks	237
Examples	237
Output values of struct variable to Standard Output using a text template	237
Defining functions for calling from template	238
Chapter 64: Testing	239
Introduction	239
Examples	239
Basic Test	239
Benchmark tests	240
Table-driven unit tests	241
Example tests (self documenting tests)	242
Testing HTTP requests	244
Set/Reset Mock Function In Tests	244
Testing using setUp and tearDown function	244
View code coverage in HTML format	246
Chapter 65: Text + HTML Templating	247
Examples	247
Single item template	247
Multiple item template	247
Templates with custom logic	248
Templates with structs	249
HTML templates	250
How HTML templates prevent malicious code injection	251
Chapter 66: The Go Command	254
Introduction	254
Examples	254
Go Run	254
Run multiple files in package	254
Go Build	254
Specify OS or Architecture in build:	255

Build multiple files.....	255
Building a package.....	255
Go Clean.....	255
Go Fmt.....	255
Go Get.....	256
Go env.....	257
Chapter 67: Time	258
Introduction.....	258
Syntax.....	258
Examples.....	258
Return time.Time Zero Value when function has an Error.....	258
Time parsing.....	258
Comparing Time.....	259
Chapter 68: Type conversions.....	261
Examples.....	261
Basic Type Conversion.....	261
Testing Interface Implementation.....	261
Implement a Unit System with Types.....	261
Chapter 69: Variables.....	263
Syntax.....	263
Examples.....	263
Basic Variable Declaration.....	263
Multiple Variable Assignment.....	263
Blank Identifier.....	264
Checking a variable's type.....	264
Chapter 70: Vendoring.....	266
Remarks.....	266
Examples.....	266
Use govendor to add external packages.....	266
Using trash to manage ./vendor.....	267
Use golang/dep.....	268
Usage.....	268

vendor.json using Govendor tool.....	268
Chapter 71: Worker Pools.....	270
Examples.....	270
Simple worker pool.....	270
Job Queue with Worker Pool.....	271
Chapter 72: XML.....	274
Remarks.....	274
Examples.....	274
Basic decoding / unmarshalling of nested elements with data.....	274
Chapter 73: YAML.....	276
Examples.....	276
Creating a config file in YAML format.....	276
Chapter 74: Zero values.....	277
Remarks.....	277
Examples.....	277
Basic Zero Values.....	277
More Complex Zero Values.....	277
Struct Zero Values.....	278
Array Zero Values.....	278
Chapter 75: Zero values.....	279
Examples.....	279
Explanation.....	279
Credits.....	281

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: [go](#)

It is an unofficial and free Go ebook created for educational purposes. All the content is extracted from [Stack Overflow Documentation](#), which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Go.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with Go

Remarks

Go is an open-source, compiled, statically typed language in the tradition of Algol and C. It boasts features such as garbage collection, limited structural typing, memory safety features, and easy-to-use CSP-style concurrent programming.

Versions

The latest major version release is in bold below. Full release history can be found [here](#).

Version	Release Date
1.8.3	2017-05-24
1.8.0	2017-02-16
1.7.0	2016-08-15
1.6.0	2016-02-17
1.5.0	2015-08-19
1.4.0	2014-12-04
1.3.0	2014-06-18
1.2.0	2013-12-01
1.1.0	2013-05-13
1.0.0	2012-03-28

Examples

Hello, World!

Place the following code into a file name `hello.go`:

```
package main

import "fmt"

func main() {
```

```
fmt.Println("Hello, 世界")
}
```

Playground

When Go is [installed correctly](#) this program can be compiled and run like this:

```
go run hello.go
```

Output:

```
Hello, 世界
```

Once you are happy with the code it can be compiled to an executable by running:

```
go build hello.go
```

This will create an executable file appropriate for your operating system in the current directory, which you can then run with the following command:

Linux, OSX, and other Unix-like systems

```
./hello
```

Windows

```
hello.exe
```

Note: The Chinese characters are important because they demonstrate that Go strings are stored as read-only slices of bytes.

FizzBuzz

Another example of "Hello World" style programs is [FizzBuzz](#). This is one example of a FizzBuzz implementation. Very idiomatic Go in play here.

```
package main

// Simple fizzbuzz implementation

import "fmt"

func main() {
    for i := 1; i <= 100; i++ {
        s := ""
        if i % 3 == 0 {
            s += "Fizz"
        }
    }
}
```

```

        if i % 5 == 0 {
            s += "Buzz"
        }
        if s != "" {
            fmt.Println(s)
        } else {
            fmt.Println(i)
        }
    }
}

```

Playground

Listing Go Environment Variables

Environment variables that affect the `go` tool can be viewed via the `go env [var ...]` command:

```

$ go env
GOARCH="amd64"
GOBIN="/home/yourname/bin"
GOEXE=""
GOHOSTARCH="amd64"
GOHOSTOS="linux"
GOOS="linux"
GOPATH="/home/yourname"
GORACE=""
GOROOT="/usr/lib/go"
GOTOOLDIR="/usr/lib/go/pkg/tool/linux_amd64"
CC="gcc"
GOGCCFLAGS="-fPIC -m64 -pthread -fmessage-length=0 -fdebug-prefix-map=/tmp/go-build059426571=/tmp/go-build -gno-record-gcc-switches"
CXX="g++"
CGO_ENABLED="1"

```

By default it prints the list as a shell script; however, if one or more variable names are given as arguments, it prints the value of each named variable.

```

$go env GOOS GOPATH
linux
/home/yourname

```

Setting up the environment

If Go is not pre-installed in your system you can go to <https://golang.org/dl/> and choose your platform to download and install Go.

To set up a basic Go development environment, only a few of the many environment variables that affect the behavior of the `go` tool (See: [Listing Go Environment Variables](#) for a full list) need to be set (generally in your shell's `~/.profile` file, or equivalent on Unix-like OSs).

GOPATH

Like the system `PATH` environment variable, Go path is a `:` (or `;` on Windows) delimited list of

directories where Go will look for packages. The `go get` tool will also download packages to the first directory in this list.

The `GOPATH` is where Go will setup associated `bin`, `pkg`, and `src` folders needed for the workspace:

- `src` — location of source files: `.go`, `.c`, `.g`, `.s`
- `pkg` — has compiled `.a` files
- `bin` — contains executable files built by Go

From Go 1.8 onwards, the `GOPATH` environment variable will have a [default value](#) if it is unset. It defaults to `$HOME/go` on Unix/Linux and `%USERPROFILE%/go` on Windows.

Some tools assume that `GOPATH` will contain a single directory.

`GOBIN`

The `bin` directory where `go install` and `go get` will place binaries after building `main` packages. Generally this is set to somewhere on the system `PATH` so that installed binaries can be run and discovered easily.

`GOROOT`

This is the location of your Go installation. It is used to find the standard libraries. It is very rare to have to set this variable as Go embeds the build path into the toolchain. Setting `GOROOT` is needed if the installation directory differs from the build directory (or the value set when building).

Accessing Documentation Offline

For full documentation, run the command:

```
godoc -http=:<port-number>
```

For a tour of Go (highly recommended for beginners in the language):

```
go tool tour
```

The two commands above will start web-servers with documentation similar to what is found online [here](#) and [here](#) respectively.

For quick reference check from command-line, eg for `fmt.Print`:

```
godoc cmd/fmt Print
# or
go doc fmt Print
```

General help is also available from command-line:

```
go help [command]
```

The Go Playground

One little known Go tool is [The Go Playground](#). If one wants to experiment with Go without downloading it, they can easily do so simply by . . .

1. Visiting the [Playground](#) in their web browser
2. Entering their code
3. Clicking “Run”

Sharing your code

The Go Playground also has tools for sharing; if a user presses the “Share” button, a link (like [this one](#)) will be generated that can be sent to other people to test and edit.

In action

The Go Playground

[Run](#)[Format](#)[Import](#)

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8     fmt.Println("Hello, playground")
9 }
```

10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28

Chapter 2: Arrays

Introduction

Arrays are specific data type, representing an ordered collection of elements of another type.

In Go, Arrays can be simple (sometime called "lists") or multi-dimensional (like for example a 2-dimensions arrays is representing a ordered collection of arrays, that contains elements)

Syntax

- `var variableName [5]ArrayType` // Declaring an array of size 5.
- `var variableName [2][3]ArrayType = { {Value1, Value2, Value3}, {Value4, Value5, Value6} }` // Declaring a multidimensional array
- `variableName := [...]ArrayType {Value1, Value2, Value3}` // Declare an array of size 3 (The compiler will count the array elements to define the size)
- `arrayName[2]` // Getting the value by index.
- `arrayName[5] = 0` // Setting the value at index.
- `arrayName[0]` // First value of the Array
- `arrayName[len(arrayName)-1]` // Last value of the Array

Examples

Creating arrays

An array in go is an ordered collection of same types elements.

The basic notation to represent arrays is to use `[]` with the variable name.

Creating a new array looks like `var array = [size]Type`, replacing `size` by a number (for example 42 to specify it will be a list of 42 elements), and replacing `Type` by the type of the elements the array can contains (for example `int` or `string`)

Just below it's a code example showing the different way to create an array in Go.

```
// Creating arrays of 6 elements of type int,  
// and put elements 1, 2, 3, 4, 5 and 6 inside it, in this exact order:  
var array1 [6]int = [6]int {1, 2, 3, 4, 5, 6} // classical way  
var array2 = [6]int {1, 2, 3, 4, 5, 6} // a less verbose way  
var array3 = [...]int {1, 2, 3, 4, 5, 6} // the compiler will count the array elements by  
itself  
  
fmt.Println("array1:", array1) // > [1 2 3 4 5 6]  
fmt.Println("array2:", array2) // > [1 2 3 4 5 6]  
fmt.Println("array3:", array3) // > [1 2 3 4 5 6]  
  
// Creating arrays with default values inside:  
zeros := [8]int{} // Create a list of 8 int filled with 0
```

```

ptrs := [8]*int{}           // a list of int pointers, filled with 8 nil references (
<nil> )
emptystr := [8]string{}     // a list of string filled with 8 times ""

fmt.Println("zeros:", zeros)    // > [0 0 0 0 0 0 0 0]
fmt.Println("ptrs:", ptrs)      // > [<nil> <nil> <nil> <nil> <nil> <nil> <nil> <nil>]
fmt.Println("emptystr:", emptystr) // > [          ]
// values are empty strings, separated by spaces,
// so we can just see separating spaces

// Arrays are also working with a personalized type
type Data struct {
    Number int
    Text    string
}

// Creating an array with 8 'Data' elements
// All the 8 elements will be like {0, ""} (Number = 0, Text = "")
structs := [8]Data{}

fmt.Println("structs:", structs) // > [{0 } {0 } {0 } {0 } {0 } {0 } {0 } {0 }]
// prints {0 } because Number are 0 and Text are empty; separated by a space

```

[play it on playground](#)

Multidimensional Array

Multidimensional arrays are basically arrays containing others arrays as elements.

It is represented like `[sizeDim1][sizeDim2]..[sizeLastDim]type`, replacing `sizeDim` by numbers corresponding to the length of the dimension, and `type` by the type of data in the multidimensional array.

For example, `[2][3]int` is representing an array composed of **2 sub arrays** of **3 int typed elements**.

It can basically be the representation of a matrix of **2 lines** and **3 columns**.

So we can make huge dimensions number array like `var values := [2017][12][31][24][60]int` for example if you need to store a number for each minutes since Year 0.

To access this kind of array, for the last example, searching for the value of 2016-01-31 at 19:42, you will access `values[2016][0][30][19][42]` (because **array indexes starts at 0** and not at 1 like days and months)

Some examples following:

```

// Defining a 2d Array to represent a matrix like
// 1 2 3      So with 2 lines and 3 columns;
// 4 5 6
var multiDimArray := [2/*lines*/][3/*columns*/]int{ [3]int{1, 2, 3}, [3]int{4, 5, 6} }

// That can be simplified like this:
var simplified := [2][3]int{{1, 2, 3}, {4, 5, 6}}

// What does it looks like ?

```

```

fmt.Println(multiDimArray)
// > [[1 2 3] [4 5 6]]

fmt.Println(multiDimArray[0])
// > [1 2 3]      (first line of the array)

fmt.Println(multiDimArray[0][1])
// > 2            (cell of line 0 (the first one), column 1 (the 2nd one))

```

```

// We can also define array with as much dimensions as we need
// here, initialized with all zeros
var multiDimArray := [2][4][3][2]string{}

fmt.Println(multiDimArray);
// Yeah, many dimensions stores many data
// > [[["" ""] ["" ""]] ["" ""] ["" ""]] ["" ""] ["" ""]]
//      [[["" ""] ["" ""]] ["" ""] ["" ""]] ["" ""] ["" ""]]
//      [[["" ""] ["" ""]] ["" ""] ["" ""]] ["" ""] ["" ""]]
//      [[["" ""] ["" ""]] ["" ""] ["" ""]] ["" ""] ["" ""]]
//      [[["" ""] ["" ""]] ["" ""] ["" ""]] ["" ""] ["" ""]]
//      [[["" ""] ["" ""]] ["" ""] ["" ""]] ["" ""] ["" ""]]
//      [[["" ""] ["" ""]] ["" ""] ["" ""]] ["" ""] ["" ""]]
//      [[["" ""] ["" ""]] ["" ""] ["" ""]] ["" ""] ["" ""]]

```

```

// We can set some values in the array's cells
multiDimArray[0][0][0][0] := "All zero indexes" // Setting the first value
multiDimArray[1][3][2][1] := "All indexes to max" // Setting the value at extreme location

fmt.Println(multiDimArray);
// If we could see in 4 dimensions, maybe we could see the result as a simple format

// > [[["All zero indexes" ""] ["" ""]] ["" ""] ["" ""]] ["" ""] ["" ""]]
//      [[["" ""] ["" ""]] ["" ""] ["" ""]] ["" ""] ["" ""]]
//      [[["" ""] ["" ""]] ["" ""] ["" ""]] ["" ""] ["" ""]]
//      [[["" ""] ["" ""]] ["" ""] ["" ""]] ["" ""] ["" ""]]
//      [[["" ""] ["" ""]] ["" ""] ["" ""]] ["" ""] ["" ""]]
//      [[["" ""] ["" ""]] ["" ""] ["" ""]] ["" ""] ["" ""]]
//      [[["" ""] ["" ""]] ["" ""] ["" ""]] ["" ""] ["" ""]]
//      [[["" ""] ["" ""]] ["" ""] ["" ""]] ["" "All indexes to max"]]]

```

Array Indexes

Arrays values should be accessed using a number specifying the location of the desired value in the array. This number is called Index.

Indexes starts at **0** and finish at **array length -1**.

To access a value, you have to do something like this: `arrayName[index]`, replacing "index" by the number corresponding to the rank of the value in your array.

For example:

```

var array = [6]int {1, 2, 3, 4, 5, 6}

fmt.Println(array[-42]) // invalid array index -1 (index must be non-negative)
fmt.Println(array[-1]) // invalid array index -1 (index must be non-negative)

```

```
fmt.Println(array[0]) // > 1
fmt.Println(array[1]) // > 2
fmt.Println(array[2]) // > 3
fmt.Println(array[3]) // > 4
fmt.Println(array[4]) // > 5
fmt.Println(array[5]) // > 6
fmt.Println(array[6]) // invalid array index 6 (out of bounds for 6-element array)
fmt.Println(array[42]) // invalid array index 42 (out of bounds for 6-element array)
```

To set or modify a value in the array, the way is the same.

Example:

```
var array = [6]int {1, 2, 3, 4, 5, 6}

fmt.Println(array) // > [1 2 3 4 5 6]

array[0] := 6
fmt.Println(array) // > [6 2 3 4 5 6]

array[1] := 5
fmt.Println(array) // > [6 5 3 4 5 6]

array[2] := 4
fmt.Println(array) // > [6 5 4 4 5 6]

array[3] := 3
fmt.Println(array) // > [6 5 4 3 5 6]

array[4] := 2
fmt.Println(array) // > [6 5 4 3 2 6]

array[5] := 1
fmt.Println(array) // > [6 5 4 3 2 1]
```

Read Arrays online: <https://riptutorial.com/go/topic/390/arrays>

Chapter 3: Base64 Encoding

Syntax

- `func (enc *base64.Encoding) Encode(dst, src []byte)`
- `func (enc *base64.Encoding) Decode(dst, src []byte) (n int, err error)`
- `func (enc *base64.Encoding) EncodeToString(src []byte) string`
- `func (enc *base64.Encoding) DecodeString(s string) ([]byte, error)`

Remarks

The [encoding/base64](#) package contains several [built in encoders](#). Most of the examples in this document will use `base64.StdEncoding`, but any encoder (`URLEncoding`, `RawStdEncoding`, your own custom encoder, etc.) may be substituted.

Examples

Encoding

```
const foobar = `foo bar`
encoding := base64.StdEncoding
encodedFooBar := make([]byte, encoding.EncodedLen(len(foobar)))
encoding.Encode(encodedFooBar, []byte(foobar))
fmt.Printf("%s", encodedFooBar)
// Output: Zm9vIGJhcg==
```

Playground

Encoding to a String

```
str := base64.StdEncoding.EncodeToString([]byte(`foo bar`))
fmt.Println(str)
// Output: Zm9vIGJhcg==
```

Playground

Decoding

```
encoding := base64.StdEncoding
data := []byte(`Zm9vIGJhcg==`)
decoded := make([]byte, encoding.DecodedLen(len(data)))
n, err := encoding.Decode(decoded, data)
if err != nil {
    log.Fatal(err)
}

// Because we don't know the length of the data that is encoded
```



```
// (only the max length), we need to trim the buffer to whatever
// the actual length of the decoded data was.
decoded = decoded[:n]

fmt.Printf("`%s`", decoded)
// Output: `foo bar`
```

[Playground](#)

Decoding a String

```
decoded, err := base64.StdEncoding.DecodeString(`biws`)
if err != nil {
    log.Fatal(err)
}

fmt.Printf("%s", decoded)
// Output: n,,
```

[Playground](#)

Read Base64 Encoding online: <https://riptutorial.com/go/topic/4492/base64-encoding>

Chapter 4: Best practices on project structure

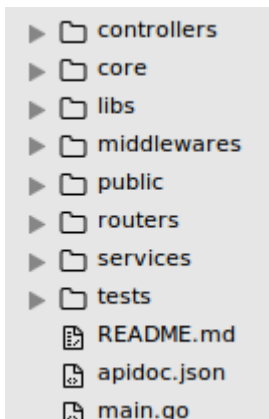
Examples

Restfull Projects API with Gin

Gin is a web framework written in Golang. It features a martini-like API with much better performance, up to 40 times faster. If you need performance and good productivity, you will love Gin.

There will be 8 packages + main.go

1. controllers
2. core
3. libs
4. middlewares
5. public
6. routers
7. services
8. tests
9. main.go



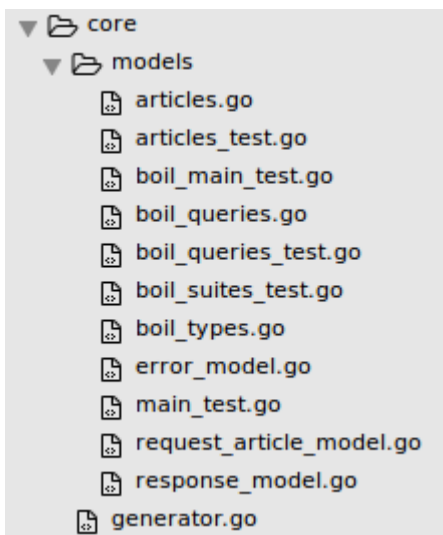
controllers

Controllers package will store all the API logic. Whatever your API, your logic will happen here



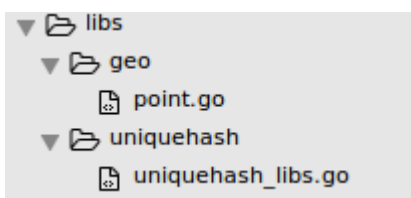
core

Core package will store all your created models, ORM, etc



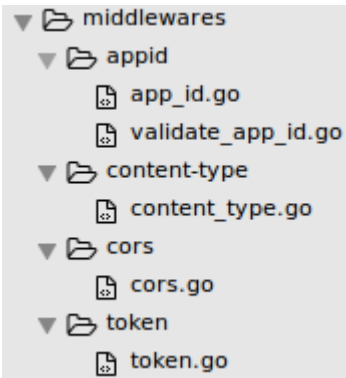
libs

This package will store any library that used in projects. But only for manually created/imported library, that not available when using `go get package_name` commands. Could be your own hashing algorithm, graph, tree etc.



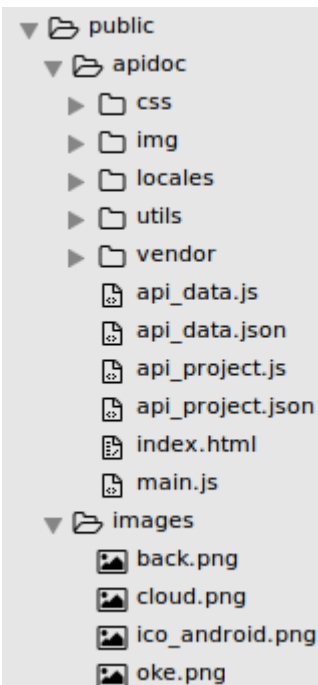
middlewares

This package store every middleware that used in project, could be creation/validation of cors,device-id , auth etc



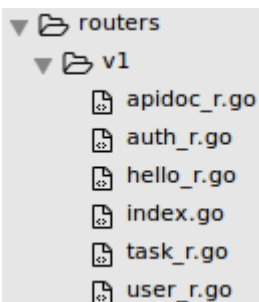
public

This package will store every public and static files, could be html, css, javascript ,images, etc



routers

This package will store every routes in your REST API.



See sample code how to assign the routes.

auth_r.go

```
import (
    auth "simple-api/controllers/v1/auth"
    "gopkg.in/gin-gonic/gin.v1"
)

func SetAuthRoutes(router *gin.RouterGroup) {

/**
 * @api {post} /v1/auth/login Login
 * @apiGroup Users
 * @apiHeader {application/json} Content-Type Accept application/json
 * @apiParam {String} username User username
 * @apiParam {String} password User Password
 * @apiParamExample {json} Input
 *     {
 *         "username": "your username",
 *         "password"      : "your password"
 *     }
 * @apiSuccess {Object} authenticate Response
 * @apiSuccess {Boolean} authenticate.success Status
 * @apiSuccess {Integer} authenticate.statuscode Status Code
 * @apiSuccess {String} authenticate.message Authenticate Message
 * @apiSuccess {String} authenticate.token Your JSON Token
 * @apiSuccessExample {json} Success
 *     {
 *         "authenticate": {
 *             "statuscode": 200,
 *             "success": true,
 *             "message": "Login Successfully",
 *             "token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0Ij0iOnRy
 *         }
 *     }
 * @apiErrorExample {json} List error
 *     HTTP/1.1 500 Internal Server Error
 */

    router.POST("/auth/login" , auth.Login)
}
```

If you see, the reason I separate the handler is, to easy us to manage each routers. So I can create comments about the API , that with apidoc will generate this into structured documentation. Then I will call the function in index.go in current package

index.go

```
package v1

import (
    "gopkg.in/gin-gonic/gin.v1"
    token "simple-api/middlewares/token"
    appid "simple-api/middlewares/appid"
)

func InitRoutes(g *gin.RouterGroup) {
```

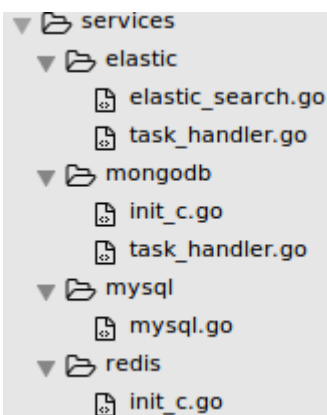
```

g.Use(appid.AppIDMiddleWare())
SetHelloRoutes(g)
SetAuthRoutes(g) // SetAuthRoutes invoked
g.Use(token.TokenAuthMiddleWare()) //secure the API From this line to bottom with JSON
Auth
g.Use(appid.ValidateAppIDMiddleWare())
SetTaskRoutes(g)
SetUserRoutes(g)
}

```

services

This package will store any configuration and setting to used in project from any used service, could be mongodb,redis,mysql, elasticsearch, etc.



main.go

The main entrance of the API. Any configuration about the dev environment settings, systems,port, etc will configured here.

Example:

main.go

```

package main
import (
    "fmt"
    "net/http"
    "gopkg.in/gin-gonic/gin.v1"
    "articles/services/mysql"
    "articles/routers/v1"
    "articles/core/models"
)

var router *gin.Engine;

func init() {
    mysql.CheckDB()
    router = gin.New();
    router.NoRoute(noRouteHandler())
    version1:=router.Group("/v1")
}

```

```

    v1.InitRoutes(version1)

}

func main() {
    fmt.Println("Server Running on Port: ", 9090)
    http.ListenAndServe(":9090",router)
}

func noRouteHandler() gin.HandlerFunc{
    return func(c *gin.Context) {
        var statuscode      int
        var message          string          = "Not Found"
        var data             interface{} = nil
        var listError [] models.ErrorModel = nil
        var endpoint         string = c.Request.URL.String()
        var method           string = c.Request.Method

        var tempEr models.ErrorModel
        tempEr.ErrorCode      = 4041
        tempEr.Hints          = "Not Found !! \n Routes In Valid. You enter on invalid
Page/Endpoint"
        tempEr.Info           = "visit http://localhost:9090/v1/docs to see the available routes"
        listError             = append(listError,tempEr)
        statuscode            = 404
        responseModel := &models.ResponseModel{
            statuscode,
            message,
            data,
            listError,
            endpoint,
            method,
        }
        var content gin.H = responseModel.NewResponse();
        c.JSON(statuscode,content)
    }
}

```

ps: Every code in this example, come from different projects

see sample [projects on github](#)

Read Best practices on project structure online: <https://riptutorial.com/go/topic/9463/best-practices-on-project-structure>

Chapter 5: Branching

Examples

Switch Statements

A simple `switch` statement:

```
switch a + b {
case c:
    // do something
case d:
    // do something else
default:
    // do something entirely different
}
```

The above example is equivalent to:

```
if a + b == c {
    // do something
} else if a + b == d {
    // do something else
} else {
    // do something entirely different
}
```

The `default` clause is optional and will be executed if and only if none of the cases compare true, even if it does not appear last, which is acceptable. The following is semantically the same as the first example:

```
switch a + b {
default:
    // do something entirely different
case c:
    // do something
case d:
    // do something else
}
```

This could be useful if you intend to use the `fallthrough` statement in the `default` clause, which must be the last statement in a case and causes program execution to proceed to the next case:

```
switch a + b {
default:
    // do something entirely different, but then also do something
    fallthrough
case c:
    // do something
case d:
```



```
// do something else
}
```

An empty switch expression is implicitly `true`:

```
switch {
case a + b == c:
    // do something
case a + b == d:
    // do something else
}
```

Switch statements support a simple statement similar to `if` statements:

```
switch n := getNumber(); n {
case 1:
    // do something
case 2:
    // do something else
}
```

Cases can be combined in a comma-separated list if they share the same logic:

```
switch a + b {
case c, d:
    // do something
default:
    // do something entirely different
}
```

If Statements

A simple `if` statement:

```
if a == b {
    // do something
}
```

Note that there are no parentheses surrounding the condition and that the opening curly brace `{` must be on the same line. The following will *not* compile:

```
if a == b
{
    // do something
}
```

An `if` statement making use of `else`:

```
if a == b {
```

```
    // do something
} else if a == c {
    // do something else
} else {
    // do something entirely different
}
```

Per [golang.org's documentation](https://golang.org/doc/faq#block-scoping), "The expression may be preceded by a simple statement, which executes before the expression is evaluated." Variables declared in this simple statement are scoped to the `if` statement and cannot be accessed outside it:

```
if err := attemptSomething(); err != nil {
    // attemptSomething() was successful!
} else {
    // attemptSomething() returned an error; handle it
}
fmt.Println(err) // compiler error, 'undefined: err'
```

Type Switch Statements

A simple type switch:

```
// assuming x is an expression of type interface{}
switch t := x.(type) {
case nil:
    // x is nil
    // t will be type interface{}
case int:
    // underlying type of x is int
    // t will be int in this case as well
case string:
    // underlying type of x is string
    // t will be string in this case as well
case float, bool:
    // underlying type of x is either float or bool
    // since we don't know which, t is of type interface{} in this case
default:
    // underlying type of x was not any of the types tested for
    // t is interface{} in this type
}
```

You can test for any type, including `error`, user-defined types, interface types, and function types:

```
switch t := x.(type) {
case error:
    log.Fatal(t)
case myType:
    fmt.Println(myType.message)
case myInterface:
    t.MyInterfaceMethod()
case func(string) bool:
    if t("Hello world?") {
        fmt.Println("Hello world!")
    }
}
```

```
}
```

Goto statements

A `goto` statement transfers control to the statement with the corresponding label within the same function. Executing the `goto` statement must not cause any variables to come into scope that were not already in scope at the point of the `goto`.

for example see the standard library source code: <https://golang.org/src/math/gamma.go> :

```
for x < 0 {
    if x > -1e-09 {
        goto small
    }
    z = z / x
    x = x + 1
}
for x < 2 {
    if x < 1e-09 {
        goto small
    }
    z = z / x
    x = x + 1
}

if x == 2 {
    return z
}

x = x - 2
p = (((((x*_gamP[0]+_gamP[1])*x+_gamP[2])*x+_gamP[3])*x+_gamP[4])*x+_gamP[5])*x + _gamP[6]
q =
((((((x*_gamQ[0]+_gamQ[1])*x+_gamQ[2])*x+_gamQ[3])*x+_gamQ[4])*x+_gamQ[5])*x+_gamQ[6])*x +
_gamQ[7]
    return z * p / q

small:
    if x == 0 {
        return Inf(1)
    }
    return z / ((1 + Euler*x) * x)
```

Break-continue statements

The `break` statement, on execution makes the current loop to force exit

package main

```
import "fmt"

func main() {
    i:=0
    for true {
        if i>2 {
            break
        }
    }
}
```

```

    }
    fmt.Println("Iteration : ",i)
    i++
  }
}

```

The continue statement, on execution moves the control to the start of the loop

```

import "fmt"

func main() {
  j:=100
  for j<110 {
    j++
    if j%2==0 {
      continue
    }
    fmt.Println("Var : ",j)
  }
}

```

Break/continue loop inside switch

```

import "fmt"

func main() {
  j := 100

loop:
  for j < 110 {
    j++

    switch j % 3 {
    case 0:
      continue loop
    case 1:
      break loop
    }

    fmt.Println("Var : ", j)
  }
}

```

Read Branching online: <https://riptutorial.com/go/topic/1342/branching>

Chapter 6: Build Constraints

Syntax

- `// +build tags`

Remarks

Build tags are used for conditionally building certain files in your code. Build tags may ignore files that you don't want build unless explicitly included, or some predefined build tags may be used to have a file only be built on a particular architecture or operating system.

Build tags may appear in any kind of source file (not just Go), but they must appear near the top of the file, preceded only by blank lines and other line comments. These rules mean that in Go files a build constraint must appear before the package clause.

A series of build tags must be followed by a blank line.

Examples

Separate integration tests

Build constraints are commonly used to separate normal unit tests from integration tests that require external resources, like a database or network access. To do this, add a custom build constraint to the top of the test file:

```
// +build integration

package main

import (
    "testing"
)

func TestThatRequiresNetworkAccess(t *testing.T) {
    t.Fatal("It failed!")
}
```

The test file will not compile into the build executable unless the following invocation of `go test` is used:

```
go test -tags "integration"
```

Results:

```
$ go test
?      bitbucket.org/yourname/yourproject    [no test files]
```

```
$ go test -tags "integration"
--- FAIL: TestThatRequiresNetworkAccess (0.00s)
    main_test.go:10: It failed!
FAIL
exit status 1
FAIL    bitbucket.org/yourname/yourproject    0.003s
```

Optimize implementations based on architecture

We can optimize a simple xor function for only architectures that support unaligned reads/writes by creating two files that define the function and prefixing them with a build constraint (for an actual example of the xor code which is out of scope here, see `crypto/cipher/xor.go` in the standard library):

```
// +build 386 amd64 s390x

package cipher

func xorBytes(dst, a, b []byte) int { /* This function uses unaligned reads / writes to
optimize the operation */ }
```

and for other architectures:

```
// +build !386,!amd64,!s390x

package cipher

func xorBytes(dst, a, b []byte) int { /* This version of the function just loops and xors */ }
```

Read Build Constraints online: <https://riptutorial.com/go/topic/2595/build-constraints>

Chapter 7: cgo

Examples

Cgo: First steps tutorial

Some examples to understand the workflow of using Go C Bindings

What

In Go you can call C programs and functions using [cgo](#). This way you can easily create C bindings to other applications or libraries that provides C API.

How

All you need to do is to add a `import "C"` at the beginning of your Go program **just** after including your C program:

```
//#include <stdio.h>
import "C"
```

With the previous example you can use the `stdio` package in Go.

If you need to use an app that is on your same folder, you use the same syntax than in C (with the `"` instead of `<>`)

```
//#include "hello.c"
import "C"
```

IMPORTANT: Do not leave a newline between the `include` and the `import "C"` statements or you will get this type of errors on build:

```
# command-line-arguments
could not determine kind of name for C.Hello
could not determine kind of name for C.sum
```

The example

On this folder you can find an example of C bindings. We have two very simple C "libraries" called `hello.c`:

```
//hello.c
#include <stdio.h>

void Hello(){
```

```
    printf("Hello world\n");
}
```

That simply prints "hello world" in the console and `sum.c`

```
//sum.c
#include <stdio.h>

int sum(int a, int b) {
    return a + b;
}
```

...that takes 2 arguments and returns its sum (do not print it).

We have a `main.go` program that will make use of this two files. First we import them as we mentioned before:

```
//main.go
package main

/*
    #include "hello.c"
    #include "sum.c"
*/
import "C"
```

Hello World!

Now we are ready to use the C programs in our Go app. Let's first try the Hello program:

```
//main.go
package main

/*
    #include "hello.c"
    #include "sum.c"
*/
import "C"

func main() {
    //Call to void function without params
    err := Hello()
    if err != nil {
        log.Fatal(err)
    }
}

//Hello is a C binding to the Hello World "C" program. As a Go user you could
//use now the Hello function transparently without knowing that it is calling
//a C function
func Hello() error {
    _, err := C.Hello()    //We ignore first result as it is a void function
    if err != nil {
        return errors.New("error calling Hello function: " + err.Error())
    }
}
```



```
    return nil
}
```

Now run the main.go program using the `go run main.go` to get print of the C program: "Hello world!". Well done!

Sum of ints

Let's make it a bit more complex by adding a function that sums its two arguments.

```
//sum.c
#include <stdio.h>

int sum(int a, int b) {
    return a + b;
}
```

And we'll call it from our previous Go app.

```
//main.go
package main

/*
#include "hello.c"
#include "sum.c"
*/
import "C"

import (
    "errors"
    "fmt"
    "log"
)

func main() {
    //Call to void function without params
    err := Hello()
    if err != nil {
        log.Fatal(err)
    }

    //Call to int function with two params
    res, err := makeSum(5, 4)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("Sum of 5 + 4 is %d\n", res)
}

//Hello is a C binding to the Hello World "C" program. As a Go user you could
//use now the Hello function transparently without knowing that is calling a C
//function
func Hello() error {
    _, err := C.Hello() //We ignore first result as it is a void function
    if err != nil {
```

```

        return errors.New("error calling Hello function: " + err.Error())
    }

    return nil
}

//makeSum also is a C binding to make a sum. As before it returns a result and
//an error. Look that we had to pass the Int values to C.int values before using
//the function and cast the result back to a Go int value
func makeSum(a, b int) (int, error) {
    //Convert Go ints to C ints
    aC := C.int(a)
    bC := C.int(b)

    sum, err := C.sum(aC, bC)
    if err != nil {
        return 0, errors.New("error calling Sum function: " + err.Error())
    }

    //Convert C.int result to Go int
    res := int(sum)

    return res, nil
}

```

Take a look at the "makeSum" function. It receives two `int` parameters that need to be converted to `C int` before by using the `C.int` function. Also, the return of the call will give us a `C int` and an error in case something went wrong. We need to cast C response to a Go's int using `int()`.

Try running our go app by using `go run main.go`

```

$ go run main.go
Hello world!
Sum of 5 + 4 is 9

```

Generating a binary

If you try a go build you could get multiple definition errors.

```

$ go build
# github.com/sayden/c-bindings
/tmp/go-build329491076/github.com/sayden/c-bindings/_obj/hello.o: In function `Hello':
../../../../go/src/github.com/sayden/c-bindings/hello.c:5: multiple definition of `Hello'
/tmp/go-build329491076/github.com/sayden/c-
bindings/_obj/main.cgo2.o:/home/mariocaster/go/src/github.com/sayden/c-bindings/hello.c:5:
first defined here
/tmp/go-build329491076/github.com/sayden/c-bindings/_obj/sum.o: In function `sum':
../../../../go/src/github.com/sayden/c-bindings/sum.c:5: multiple definition of `sum'
/tmp/go-build329491076/github.com/sayden/c-
bindings/_obj/main.cgo2.o:/home/mariocaster/go/src/github.com/sayden/c-bindings/sum.c:5: first
defined here
collect2: error: ld returned 1 exit status

```

The trick is to refer to the main file directly when using `go build`:

```
$ go build main.go
$ ./main
Hello world!
Sum of 5 + 4 is 9
```

Remember that you can provide a name to the binary file by using `-o` flag `go build -o my_c_binding main.go`

I hope you enjoyed this tutorial.

Read cgo online: <https://riptutorial.com/go/topic/6125/cgo>

Chapter 8: cgo

Examples

Calling C Function From Go

Cgo enables the creation of Go packages that call C code.

To use `cgo` write normal Go code that imports a pseudo-package "C". The Go code can then refer to types such as `C.int`, or functions such as `C.Add`.

The import of "C" is immediately preceded by a comment, that comment, called the preamble, is used as a header when compiling the C parts of the package.

Note that there must be no blank lines in between the `cgo` comment and the import statement.

Note that `import "C"` can not grouped with other imports into a parenthesized, "factored" import statement. You must write multiple import statements, like:

```
import "C"
import "fmt"
```

And it is good style to use the factored import statement, for other imports, like:

```
import "C"
import (
    "fmt"
    "math"
)
```

Simple example using `cgo`:

```
package main

//int Add(int a, int b){
//    return a+b;
//}
import "C"
import "fmt"

func main() {
    a := C.int(10)
    b := C.int(20)
    c := C.Add(a, b)
    fmt.Println(c) // 30
}
```

Then `go build`, and run it, output:

```
30
```

To build `cgo` packages, just use `go build` or `go install` as usual. The `go tool` recognizes the special "C" import and automatically uses `cgo` for those files.

Wire C and Go code in all directions

Calling C code from Go

```
package main

/*
// Everything in comments above the import "C" is C code and will be compiled with the GCC.
// Make sure you have a GCC installed.

int addInC(int a, int b) {
    return a + b;
}
*/
import "C"
import "fmt"

func main() {
    a := 3
    b := 5

    c := C.addInC(C.int(a), C.int(b))

    fmt.Println("Add in C:", a, "+", b, "=", int(c))
}
```

Calling Go code from C

```
package main

/*
static inline int multiplyInGo(int a, int b) {
    return go_multiply(a, b);
}
*/
import "C"
import (
    "fmt"
)

func main() {
    a := 3
    b := 5

    c := C.multiplyInGo(C.int(a), C.int(b))

    fmt.Println("multiplyInGo:", a, "*", b, "=", int(c))
}

//export go_multiply
func go_multiply(a C.int, b C.int) C.int {
    return a * b
}
```

Dealing with Function pointers

```
package main
```

```

/*
int go_multiply(int a, int b);

typedef int (*multiply_f)(int a, int b);
multiply_f multiply;

static inline init() {
    multiply = go_multiply;
}

static inline int multiplyWithFp(int a, int b) {
    return multiply(a, b);
}
*/
import "C"
import (
    "fmt"
)

func main() {
    a := 3
    b := 5
    C.init(); // OR:
    C.multiply = C.multiply_f(go_multiply);

    c := C.multiplyWithFp(C.int(a), C.int(b))

    fmt.Println("multiplyInGo:", a, "+", b, "=", int(c))
}

//export go_multiply
func go_multiply(a C.int, b C.int) C.int {
    return a * b
}

```

Convert Types, Access Structs and Pointer Arithmetic

From the official Go documentation:

```

// Go string to C string
// The C string is allocated in the C heap using malloc.
// It is the caller's responsibility to arrange for it to be
// freed, such as by calling C.free (be sure to include stdlib.h
// if C.free is needed).
func C.CString(string) *C.char

// Go []byte slice to C array
// The C array is allocated in the C heap using malloc.
// It is the caller's responsibility to arrange for it to be
// freed, such as by calling C.free (be sure to include stdlib.h
// if C.free is needed).
func C.CBytes([]byte) unsafe.Pointer

// C string to Go string
func C.GoString(*C.char) string

// C data with explicit length to Go string
func C.GoStringN(*C.char, C.int) string

```

```
// C data with explicit length to Go []byte
func C.GoBytes(unsafe.Pointer, C.int) []byte
```

How to use it:

```
func go_handleData(data *C.uint8_t, length C.uint8_t) []byte {
    return C.GoBytes(unsafe.Pointer(data), C.int(length))
}

// ...

goByteSlice := []byte {1, 2, 3}
goUnsafePointer := C.CBytes(goByteSlice)
cPointer := (*C.uint8_t)(goUnsafePointer)

// ...

func getPayload(packet *C.packet_t) []byte {
    dataPtr := unsafe.Pointer(packet.data)
    // Lets assume a 2 byte header before the payload.
    payload := C.GoBytes(unsafe.Pointer(uintptr(dataPtr)+2), C.int(packet.dataLength-2))
    return payload
}
```

Read cgo online: <https://riptutorial.com/go/topic/6455/cgo>

Chapter 9: Channels

Introduction

A channel contains values of a given type. Values can be written to a channel and read from it, and they circulate inside the channel in first-in-first-out order. There is a distinction between buffered channels, which can contain several messages, and unbuffered channels, which cannot. Channels are typically used to communicate between goroutines, but are also useful in other circumstances.

Syntax

- `make(chan int)` // create an unbuffered channel
- `make(chan int, 5)` // create a buffered channel with a capacity of 5
- `close(ch)` // closes a channel "ch"
- `ch <- 1` // write the value of 1 to a channel "ch"
- `val := <-ch` // read a value from channel "ch"
- `val, ok := <-ch` // alternate syntax; ok is a bool indicating if the channel is closed

Remarks

A channel holding the empty struct `make(chan struct{})` is a clear message to the user that no information is transmitted over the channel and that it's purely used for synchronization.

Regarding unbuffered channels, a channel write will block until a corresponding read occurs from another goroutine. The same is true for a channel read blocking while waiting for a writer.

Examples

Using range

When reading multiple values from a channel, using `range` is a common pattern:

```
func foo() chan int {
    ch := make(chan int)

    go func() {
        ch <- 1
        ch <- 2
        ch <- 3
        close(ch)
    }()

    return ch
}
```



```
func main() {
    for n := range foo() {
        fmt.Println(n)
    }

    fmt.Println("channel is now closed")
}
```

Playground

Output

```
1
2
3
channel is now closed
```

Timeouts

Channels are often used to implement timeouts.

```
func main() {
    // Create a buffered channel to prevent a goroutine leak. The buffer
    // ensures that the goroutine below can eventually terminate, even if
    // the timeout is met. Without the buffer, the send on the channel
    // blocks forever, waiting for a read that will never happen, and the
    // goroutine is leaked.
    ch := make(chan struct{}, 1)

    go func() {
        time.Sleep(10 * time.Second)
        ch <- struct{}{}
    }()

    select {
    case <-ch:
        // Work completed before timeout.
    case <-time.After(1 * time.Second):
        // Work was not completed after 1 second.
    }
}
```

Coordinating goroutines

Imagine a goroutine with a two step process, where the main thread needs to do some work between each step:

```
func main() {
    ch := make(chan struct{})
    go func() {
        // Wait for main thread's signal to begin step one
        <-ch

        // Perform work
        time.Sleep(1 * time.Second)
    }
```

```

    // Signal to main thread that step one has completed
    ch <- struct{}{}

    // Wait for main thread's signal to begin step two
    <-ch

    // Perform work
    time.Sleep(1 * time.Second)

    // Signal to main thread that work has completed
    ch <- struct{}{}
}()

// Notify goroutine that step one can begin
ch <- struct{}{}

// Wait for notification from goroutine that step one has completed
<-ch

// Perform some work before we notify
// the goroutine that step two can begin
time.Sleep(1 * time.Second)

// Notify goroutine that step two can begin
ch <- struct{}{}

// Wait for notification from goroutine that step two has completed
<-ch
}

```

Buffered vs unbuffered

```

func bufferedUnbufferedExample(buffered bool) {
    // We'll declare the channel, and we'll make it buffered or
    // unbuffered depending on the parameter `buffered` passed
    // to this function.
    var ch chan int
    if buffered {
        ch = make(chan int, 3)
    } else {
        ch = make(chan int)
    }

    // We'll start a goroutine, which will emulate a webserver
    // receiving tasks to do every 25ms.
    go func() {
        for i := 0; i < 7; i++ {
            // If the channel is buffered, then while there's an empty
            // "slot" in the channel, sending to it will not be a
            // blocking operation. If the channel is full, however, we'll
            // have to wait until a "slot" frees up.
            // If the channel is unbuffered, sending will block until
            // there's a receiver ready to take the value. This is great
            // for goroutine synchronization, not so much for queueing
            // tasks for instance in a webserver, as the request will
            // hang until the worker is ready to take our task.
            fmt.Println(">", "Sending", i, "...")
            ch <- i
        }
    }()
}

```

```

        fmt.Println(">", i, "sent!")
        time.Sleep(25 * time.Millisecond)
    }
    // We'll close the channel, so that the range over channel
    // below can terminate.
    close(ch)
}()

for i := range ch {
    // For each task sent on the channel, we would perform some
    // task. In this case, we will assume the job is to
    // "sleep 100ms".
    fmt.Println("<", i, "received, performing 100ms job")
    time.Sleep(100 * time.Millisecond)
    fmt.Println("<", i, "job done")
}
}

```

[go playground](#)

Blocking & unblocking of channels

By default communication over the channels is sync; when you send some value there must be a receiver. Otherwise you will get fatal error: all goroutines are asleep - deadlock! as follows:

```

package main

import "fmt"

func main() {
    msg := make(chan string)
    msg <- "Hey There"
    go func() {
        fmt.Println(<-msg)
    }()
}

```

But there is a solution use: use buffered channels :

```

package main

import "fmt"
import "time"

func main() {
    msg := make(chan string, 1)
    msg <- "Hey There!"
    go func() {
        fmt.Println(<-msg)
    }()
    time.Sleep(time.Second * 1)
}

```

Waiting for work to finish

A common technique for using channels is to create some number of workers (or consumers) to

read from the channel. Using a `sync.WaitGroup` is an easy way to wait for those workers to finish running.

```
package main

import (
    "fmt"
    "sync"
    "time"
)

func main() {
    numPiecesOfWork := 20
    numWorkers := 5

    workCh := make(chan int)
    wg := &sync.WaitGroup{}

    // Start workers
    wg.Add(numWorkers)
    for i := 0; i < numWorkers; i++ {
        go worker(workCh, wg)
    }

    // Send work
    for i := 0; i < numPiecesOfWork; i++ {
        work := i % 10 // invent some work
        workCh <- work
    }

    // Tell workers that no more work is coming
    close(workCh)

    // Wait for workers to finish
    wg.Wait()

    fmt.Println("done")
}

func worker(workCh <-chan int, wg *sync.WaitGroup) {
    defer wg.Done() // will call wg.Done() right before returning

    for work := range workCh { // will wait for work until workCh is closed
        doWork(work)
    }
}

func doWork(work int) {
    time.Sleep(time.Duration(work) * time.Millisecond)
    fmt.Println("slept for", work, "milliseconds")
}
```

Read Channels online: <https://riptutorial.com/go/topic/1263/channels>

Chapter 10: Closures

Examples

Closure Basics

A *Closure* is a function taken together with an environment. The function is typically an anonymous function defined inside another function. The environment is the lexical scope of the enclosing function (very basic idea of a lexical scope of a function would be the scope that exists between the function's braces.)

```
func g() {  
    i := 0  
    f := func() { // anonymous function  
        fmt.Println("f called")  
    }  
}
```

Within the body of an anonymous function (say f) defined within another function (say g), variables present in scopes of both f and g are accessible. However, it is the scope of g that forms the environment part of the closure (function part is f) and as a result, changes made to the variables in g 's scope retain their values (i.e. the environment persists between calls to f).

Consider the below function:

```
func NaturalNumbers() func() int {  
    i := 0  
    f := func() int { // f is the function part of closure  
        i++  
        return i  
    }  
    return f  
}
```

In above definition, `NaturalNumbers` has an inner function `f` which `NaturalNumbers` returns. Inside `f`, variable `i` defined within the scope of `NaturalNumbers` is being accessed.

We get a new function from `NaturalNumbers` like so:

```
n := NaturalNumbers()
```

Now `n` is a closure. It is a function (defined by `f`) which also has an associated environment (scope of `NaturalNumbers`).

In case of `n`, the environment part only contains one variable: `i`

Since `n` is a function, it can be called:

```

fmt.Println(n()) // 1
fmt.Println(n()) // 2
fmt.Println(n()) // 3

```

As evident from above output, each time `n` is called, it increments `i`. `i` starts at 0, and each call to `n` executes `i++`.

The value of `i` is retained between calls. That is, the environment, being a part of closure, persists.

Calling `NaturalNumbers` again would create and return a new function. This would initialize a new `i` within `NaturalNumbers`. Which means that the newly returned function forms another closure having the same part for function (still `f`) but a brand new environment (a newly initialized `i`).

```

o := NaturalNumbers()

fmt.Println(n()) // 4
fmt.Println(o()) // 1
fmt.Println(o()) // 2
fmt.Println(n()) // 5

```

Both `n` and `o` are closures containing same function part (which gives them the same behavior), but different environments. Thus, use of closures allow functions to have access to a persistent environment that can be used to retain information between calls.

Another example:

```

func multiples(i int) func() int {
    var x int = 0
    return func() int {
        x++
        // parameter to multiples (here it is i) also forms
        // a part of the environment, and is retained
        return x * i
    }
}

two := multiples(2)
fmt.Println(two(), two(), two()) // 2 4 6

fortyTwo := multiples(42)
fmt.Println(fortyTwo(), fortyTwo(), fortyTwo()) // 42 84 126

```

Read Closures online: <https://riptutorial.com/go/topic/2741/closures>

Chapter 11: Concurrency

Introduction

In Go, concurrency is achieved through the use of goroutines, and communication between goroutines is usually done with channels. However, other means of synchronization, like mutexes and wait groups, are available, and should be used whenever they are more convenient than channels.

Syntax

- `go doWork()` // run the function `doWork` as a goroutine
- `ch := make(chan int)` // declare new channel of type `int`
- `ch <- 1` // sending on a channel
- `value = <-ch` // receiving from a channel

Remarks

Goroutines in Go are similar to threads in other languages in terms of usage. Internally, Go creates a number of threads (specified by `GOMAXPROCS`) and then schedules the goroutines to run on the threads. Because of this design, Go's concurrency mechanisms are much more efficient than threads in terms of memory usage and initialization time.

Examples

Creating goroutines

Any function can be invoked as a goroutine by prefixing its invocation with the keyword `go`:

```
func DoMultiply(x,y int) {
    // Simulate some hard work
    time.Sleep(time.Second * 1)
    fmt.Printf("Result: %d\n", x * y)
}

go DoMultiply(1,2) // first execution, non-blocking
go DoMultiply(3,4) // second execution, also non-blocking

// Results are printed after a single second only,
// not 2 seconds because they execute concurrently:
// Result: 2
// Result: 12
```

Note that the return value of the function is ignored.

Hello World Goroutine

single channel, single goroutine, one write, one read.

```
package main

import "fmt"
import "time"

func main() {
    // create new channel of type string
    ch := make(chan string)

    // start new anonymous goroutine
    go func() {
        time.Sleep(time.Second)
        // send "Hello World" to channel
        ch <- "Hello World"
    }()
    // read from channel
    msg, ok := <-ch
    fmt.Printf("msg='%s', ok='%v'\n", msg, ok)
}
```

[Run it on playground](#)

The channel `ch` is an **unbuffered or synchronous channel**.

The `time.Sleep` is here to illustrate `main()` function will **wait** on the `ch` channel, which means the **function literal** executed as a goroutine has the time to send a value through that channel: the **receive operator** `<-ch` will block the execution of `main()`. If it didn't, the goroutine would be killed when `main()` exits, and would not have time to send its value.

Waiting for goroutines

Go programs end when the `main` function ends, therefore it is common practice to wait for all goroutines to finish. A common solution for this is to use a `sync.WaitGroup` object.

```
package main

import (
    "fmt"
    "sync"
)

var wg sync.WaitGroup // 1

func routine(i int) {
    defer wg.Done() // 3
    fmt.Printf("routine %v finished\n", i)
}

func main() {
    wg.Add(10) // 2
    for i := 0; i < 10; i++ {
        go routine(i) // *
    }
    wg.Wait() // 4
}
```



```
    fmt.Println("main finished")
}
```

Run the example in the playground

WaitGroup usage in order of execution:

1. Declaration of global variable. Making it global is the easiest way to make it visible to all functions and methods.
2. Increasing the counter. This must be done in the main goroutine because there is no guarantee that a newly started goroutine will execute before 4 due to memory model [guarantees](#).
3. Decreasing the counter. This must be done at the exit of a goroutine. By using a deferred call, we make sure that it [will be called whenever function ends](#), no matter how it ends.
4. Waiting for the counter to reach 0. This must be done in the main goroutine to prevent the program from exiting before all goroutines have finished.

* Parameters are [evaluated before starting a new goroutine](#). Thus it is necessary to define their values explicitly before `wg.Add(10)` so that possibly-panicking code will not increase the counter. Adding 10 items to the WaitGroup, so it will wait for 10 items before `wg.Wait` returns the control back to `main()` goroutine. Here, the value of `i` is defined in the for loop.

Using closures with goroutines in a loop

When in a loop, the loop variable (`val`) in the following example is a single variable that changes value as it goes over the loop. Therefore one must do the following to actually pass each `val` of values to the goroutine:

```
for val := range values {
    go func(val interface{}) {
        fmt.Println(val)
    }(val)
}
```

If you were to do just `go func(val interface{}) { ... }()` without passing `val`, then the value of `val` will be whatever `val` is when the goroutines actually runs.

Another way to get the same effect is:

```
for val := range values {
    val := val
    go func() {
        fmt.Println(val)
    }()
}
```

The strange-looking `val := val` creates a new variable in each iteration, which is then accessed by the goroutine.

Stopping goroutines

```

package main

import (
    "log"
    "sync"
    "time"
)

func main() {
    // The WaitGroup lets the main goroutine wait for all other goroutines
    // to terminate. However, this is no implicit in Go. The WaitGroup must
    // be explicitly incremented prior to the execution of any goroutine
    // (i.e. before the `go` keyword) and it must be decremented by calling
    // wg.Done() at the end of every goroutine (typically via the `defer` keyword).
    wg := sync.WaitGroup{}

    // The stop channel is an unbuffered channel that is closed when the main
    // thread wants all other goroutines to terminate (there is no way to
    // interrupt another goroutine in Go). Each goroutine must multiplex its
    // work with the stop channel to guarantee liveness.
    stopCh := make(chan struct{})

    for i := 0; i < 5; i++ {
        // It is important that the WaitGroup is incremented before we start
        // the goroutine (and not within the goroutine) because the scheduler
        // makes no guarantee that the goroutine starts execution prior to
        // the main goroutine calling wg.Wait().
        wg.Add(1)
        go func(i int, stopCh <-chan struct{}) {
            // The defer keyword guarantees that the WaitGroup count is
            // decremented when the goroutine exits.
            defer wg.Done()

            log.Printf("started goroutine %d", i)

            select {
                // Since we never send empty structs on this channel we can
                // take the return of a receive on the channel to mean that the
                // channel has been closed (recall that receive never blocks on
                // closed channels).
                case <-stopCh:
                    log.Printf("stopped goroutine %d", i)
            }
        }(i, stopCh)
    }

    time.Sleep(time.Second * 5)
    close(stopCh)
    log.Printf("stopping goroutines")
    wg.Wait()
    log.Printf("all goroutines stopped")
}

```

Ping pong with two goroutines

```

package main

import (

```

```

    "fmt"
    "time"
)

// The pinger prints a ping and waits for a pong
func pinger(pinger <-chan int, ponger chan<- int) {
    for {
        <-pinger
        fmt.Println("ping")
        time.Sleep(time.Second)
        ponger <- 1
    }
}

// The ponger prints a pong and waits for a ping
func ponger(pinger chan<- int, ponger <-chan int) {
    for {
        <-ponger
        fmt.Println("pong")
        time.Sleep(time.Second)
        pinger <- 1
    }
}

func main() {
    ping := make(chan int)
    pong := make(chan int)

    go pinger(ping, pong)
    go ponger(ping, pong)

    // The main goroutine starts the ping/pong by sending into the ping channel
    ping <- 1

    for {
        // Block the main thread until an interrupt
        time.Sleep(time.Second)
    }
}

```

Run a slightly modified version of this code in [Go Playground](#)

Read Concurrency online: <https://riptutorial.com/go/topic/376/concurrency>

Chapter 12: Console I/O

Examples

Read input from console

Using `scanf`

`Scanf` scans text read from standard input, storing successive space-separated values into successive arguments as determined by the format. It returns the number of items successfully scanned. If that is less than the number of arguments, `err` will report why. Newlines in the input must match newlines in the format. The one exception: the verb `%c` always scans the next rune in the input, even if it is a space (or tab etc.) or newline.

```
# Read integer
var i int
fmt.Scanf("%d", &i)

# Read string
var str string
fmt.Scanf("%s", &str)
```

Using `scan`

`Scan` scans text read from standard input, storing successive space-separated values into successive arguments. Newlines count as space. It returns the number of items successfully scanned. If that is less than the number of arguments, `err` will report why.

```
# Read integer
var i int
fmt.Scan(&i)

# Read string
var str string
fmt.Scan(&str)
```

Using `scanln`

`Sscanln` is similar to `Sscan`, but stops scanning at a newline and after the final item there must be a newline or EOF.

```
# Read string
var input string
fmt.Scanln(&input)
```

Using `bufio`

```
# Read using Reader
reader := bufio.NewReader(os.Stdin)
```

```
text, err := reader.ReadString('\n')

# Read using Scanner
scanner := bufio.NewScanner(os.Stdin)
for scanner.Scan() {
    fmt.Println(scanner.Text())
}
```

Read Console I/O online: <https://riptutorial.com/go/topic/9741/console-i-o>

Chapter 13: Constants

Remarks

Go supports constants of character, string, boolean, and numeric values.

Examples

Declaring a constant

Constants are declared like variables, but using the `const` keyword:

```
const Greeting string = "Hello World"
const Years int = 10
const Truth bool = true
```

Like for variables, names starting with an upper case letter are exported (*public*), names starting with lower case are not.

```
// not exported
const alpha string = "Alpha"
// exported
const Beta string = "Beta"
```

Constants can be used like any other variable, except for the fact that the value cannot be changed. Here's an example:

```
package main

import (
    "fmt"
    "math"
)

const s string = "constant"

func main() {
    fmt.Println(s) // constant

    // A `const` statement can appear anywhere a `var` statement can.
    const n = 10
    fmt.Println(n) // 10
    fmt.Printf("n=%d is of type %T\n", n, n) // n=10 is of type int

    const m float64 = 4.3
    fmt.Println(m) // 4.3

    // An untyped constant takes the type needed by its context.
    // For example, here `math.Sin` expects a `float64`.
    const x = 10
    fmt.Println(math.Sin(x)) // -0.5440211108893699
```

```
}
```

Playground

Multiple constants declaration

You can declare multiple constants within the same `const` block:

```
const (  
    Alpha = "alpha"  
    Beta  = "beta"  
    Gamma = "gamma"  
)
```

And automatically increment constants with the `iota` keyword:

```
const (  
    Zero = iota // Zero == 0  
    One   // One  == 1  
    Two   // Two  == 2  
)
```

For more examples of using `iota` to declare constants, see [iota](#).

You can also declare multiple constants using the multiple assignment. However, this syntax may be harder to read and it is generally avoided.

```
const Foo, Bar = "foo", "bar"
```

Typed vs. Untyped Constants

Constants in Go may be typed or untyped. For instance, given the following string literal:

```
"bar"
```

one might say that the type of the literal is `string`, however, this is not semantically correct. Instead, literals are *Untyped string constants*. It is a string (more correctly, its *default type* is `string`), but it is not a Go **value** and therefore has no type until it is assigned or used in a context that is typed. This is a subtle distinction, but a useful one to understand.

Similarly, if we assign the literal to a constant:

```
const foo = "bar"
```

It remains untyped since, by default, constants are untyped. It is possible to declare it as a *typed string constant* as well:

```
const typedFoo string = "bar"
```

The difference comes into play when we attempt to assign these constants in a context that does have type. For instance, consider the following:

```
var s string
s = foo          // This works just fine
s = typedFoo    // As does this

type MyString string
var mys MyString
mys = foo        // This works just fine
mys = typedFoo  // cannot use typedFoo (type string) as type MyString in assignment
```

Read Constants online: <https://riptutorial.com/go/topic/1047/constants>

Chapter 14: Context

Syntax

- type CancelFunc func()
- func Background() Context
- func TODO() Context
- func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
- func WithDeadline(parent Context, deadline time.Time) (Context, CancelFunc)
- func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
- func WithValue(parent Context, key interface{}, val interface{})

Remarks

The `context` package (in Go 1.7) or the `golang.org/x/net/context` package (Pre 1.7) is an interface for creating contexts that can be used to carry request scoped values and deadlines across API boundaries and between services, as well as a simple implementation of said interface.

aside: the word "context" is loosely used to refer to the entire tree, or to individual leaves in the tree, eg. the actual `context.Context` values.

At a high level, a context is a tree. New leaves are added to the tree when they are constructed (a `context.Context` with a parent value), and leaves are never removed from the tree. Any context has access to all of the values above it (data access only flows upwards), and if any context is canceled its children are also canceled (cancellation signals propagate downwards). The cancel signal is implemented by means of a function that returns a channel which will be closed (readable) when the context is canceled; this makes contexts a very efficient way to implement the [pipeline and cancellation concurrency pattern](#), or timeouts.

By convention, functions that take a context have the first argument `ctx context.Context`. While this is just a convention, it's one that should be followed since many static analysis tools specifically look for this argument. Since Context is an interface, it's also possible to turn existing context-like data (values that are passed around throughout a request call chain) into a normal Go context and use them in a backwards compatible way just by implementing a few methods. Furthermore, contexts are safe for concurrent access so you can use them from many goroutines (whether they're running on parallel threads or as concurrent coroutines) without fear.

Further Reading

- <https://blog.golang.org/context>

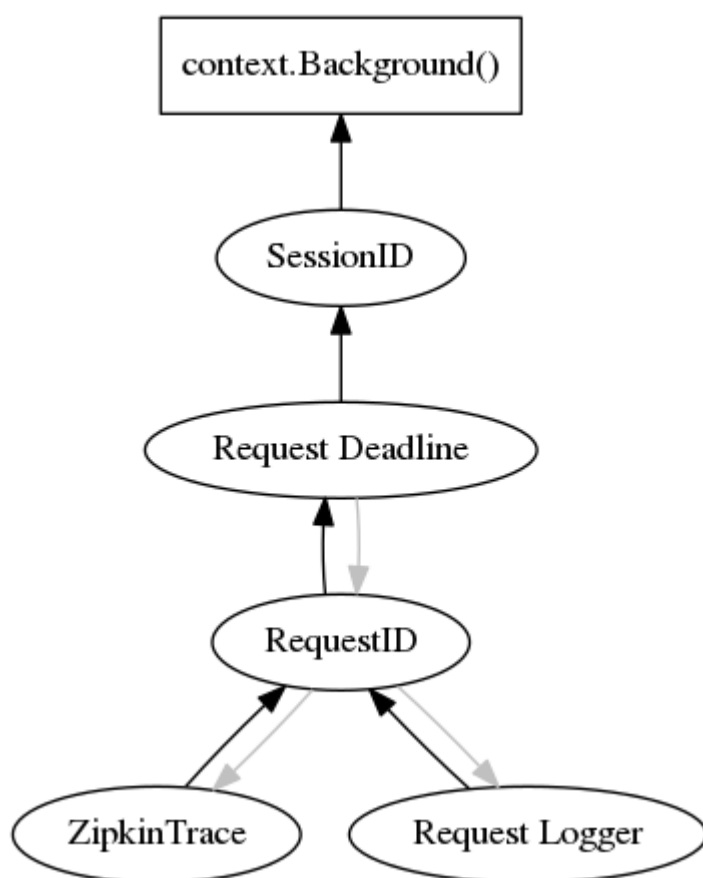
Examples

Context tree represented as a directed graph

A simple context tree (containing some common values that might be request scoped and included in a context) built from Go code like the following:

```
// Pseudo-Go
ctx := context.WithValue(
    context.WithDeadline(
        context.Background(), sidKey, sid),
        time.Now().Add(30 * time.Minute),
    ),
    ridKey, rid,
)
trCtx := trace.NewContext(ctx, tr)
logCtx := myRequestLogging.NewContext(ctx, myRequestLogging.NewLogger())
```

Is a tree that can be represented as a directed graph that looks like this:



Each child context has access to the values of its parent contexts, so the data access flows upwards in the tree (represented by black edges). Cancellation signals on the other hand travel down the tree (if a context is canceled, all of its children are also canceled). The cancellation signal flow is represented by the grey edges.

Using a context to cancel work

Passing a context with a timeout (or with a cancel function) to a long running function can be used to cancel that functions work:

```
ctx, _ := context.WithTimeout(context.Background(), 200*time.Millisecond)
```

```
for {
    select {
    case <-ctx.Done():
        return ctx.Err()
    default:
        // Do an iteration of some long running work here!
    }
}
```

Read Context online: <https://riptutorial.com/go/topic/2743/context>

Chapter 15: Cross Compilation

Introduction

The Go compiler can produce binaries for many platforms, i.e. processors and systems. Unlike with most other compilers, there is no specific requirement to cross-compiling, it is as easy to use as regular compiling.

Syntax

- `GOOS=linux GOARCH=amd64 go build`

Remarks

Supported Operating System and Architecture target combinations ([source](#))

\$GOOS	\$GOARCH
android	arm
darwin	386
darwin	amd64
darwin	arm
darwin	arm64
dragonfly	amd64
freebsd	386
freebsd	amd64
freebsd	arm
linux	386
linux	amd64
linux	arm
linux	arm64
linux	ppc64
linux	ppc64le

\$GOOS	\$GOARCH
linux	mips64
linux	mips64le
netbsd	386
netbsd	amd64
netbsd	arm
openbsd	386
openbsd	amd64
openbsd	arm
plan9	386
plan9	amd64
solaris	amd64
windows	386
windows	amd64

Examples

Compile all architectures using a Makefile

This Makefile will cross compile and zip up executables for Windows, Mac and Linux (ARM and x86).

```
# Replace demo with your desired executable name
appname := demo

sources := $(wildcard *.go)

build = GOOS=$(1) GOARCH=$(2) go build -o build/$(appname)$(3)
tar = cd build && tar -cvzf $(1)_$(2).tar.gz $(appname)$(3) && rm $(appname)$(3)
zip = cd build && zip $(1)_$(2).zip $(appname)$(3) && rm $(appname)$(3)

.PHONY: all windows darwin linux clean

all: windows darwin linux

clean:
    rm -rf build/

##### LINUX BUILDS #####
linux: build/linux_arm.tar.gz build/linux_arm64.tar.gz build/linux_386.tar.gz
```

```

build/linux_amd64.tar.gz

build/linux_386.tar.gz: $(sources)
    $(call build,linux,386,)
    $(call tar,linux,386)

build/linux_amd64.tar.gz: $(sources)
    $(call build,linux,amd64,)
    $(call tar,linux,amd64)

build/linux_arm.tar.gz: $(sources)
    $(call build,linux,arm,)
    $(call tar,linux,arm)

build/linux_arm64.tar.gz: $(sources)
    $(call build,linux,arm64,)
    $(call tar,linux,arm64)

##### DARWIN (MAC) BUILDS #####
darwin: build/darwin_amd64.tar.gz

build/darwin_amd64.tar.gz: $(sources)
    $(call build,darwin,amd64,)
    $(call tar,darwin,amd64)

##### WINDOWS BUILDS #####
windows: build/windows_386.zip build/windows_amd64.zip

build/windows_386.zip: $(sources)
    $(call build,windows,386,.exe)
    $(call zip,windows,386,.exe)

build/windows_amd64.zip: $(sources)
    $(call build,windows,amd64,.exe)
    $(call zip,windows,amd64,.exe)

```

(be cautious that [Makefile's need hard tabs not spaces](#))

Simple cross compilation with go build

From your project directory, run the `go build` command and specify the operating system and architecture target with the `GOOS` and `GOARCH` environment variables:

Compiling for Mac (64-bit):

```
GOOS=darwin GOARCH=amd64 go build
```

Compiling for Windows x86 processor:

```
GOOS=windows GOARCH=386 go build
```

You might also want to set the filename of the output executable manually to keep track of the architecture:

```
GOOS=windows GOARCH=386 go build -o appname_win_x86.exe
```

From version 1.7 and onwards you can get a list of all possible GOOS and GOARCH combinations with:

```
go tool dist list
```

(or for easier machine consumption `go tool dist list -json`)

Cross compilation by using gox

Another convenient solution for cross compilation is the usage of `gox`:

<https://github.com/mitchellh/gox>

Installation

The installation is done very easily by executing `go get github.com/mitchellh/gox`. The resulting executable gets placed at Go's binary directory, e.g. `/golang/bin` or `~/golang/bin`. Ensure that this folder is part of your path in order to use the `gox` command from an arbitrary location.

Usage

From within a Go project's root folder (where you perform e.g. `go build`), execute `gox` in order to build all possible binaries for any architecture (e.g. x86, ARM) and operating system (e.g. Linux, macOS, Windows) which is available.

In order to build for a certain operating system, use e.g. `gox -os="linux"` instead. Also the architecture option could be defined: `gox -osarch="linux/amd64"`.

Simple Example: Compile helloworld.go for arm architecture on Linux machine

Prepare `helloworld.go` (find below)

```
package main

import "fmt"

func main(){
    fmt.Println("hello world")
}
```

Run `GOOS=linux GOARCH=arm go build helloworld.go`

Copy generated `helloworld` (arm executable) file to your target machine.

Read Cross Compilation online: <https://riptutorial.com/go/topic/1020/cross-compilation>

Chapter 16: Cryptography

Introduction

Find out how to encrypt and decrypt data with Go. Keep in mind that this is not a course about cryptography but rather how to achieve it with Go.

Examples

Encryption and decryption

Foreword

This is a detailed example about how to encrypt and decrypt data with Go. The used code is shortened, e.g. the error handling is not mentioned. The full working project with error handling and user interface could be found on Github [here](#).

Encryption

Introduction and data

This example describes a full working encryption and decryption in Go. In order to do so, we need a data. In this example, we use our own data structure `secret`:

```
type secret struct {
    DisplayName      string
    Notes            string
    Username         string
    EMail            string
    CopyMethod       string
    Password         string
    CustomField01Name string
    CustomField01Data string
    CustomField02Name string
    CustomField02Data string
    CustomField03Name string
    CustomField03Data string
    CustomField04Name string
    CustomField04Data string
    CustomField05Name string
    CustomField05Data string
    CustomField06Name string
    CustomField06Data string
}
```


Next, we want to encrypt such a `secret`. The full working example could be found [here \(link to Github\)](#). Now, the step-by-step process:

Step 1

First of all, we need a kind of master password to protect the secret: `masterPassword := "PASS"`

Step 2

All the crypto methods working with bytes instead of strings. Thus, we construct a byte array with the data from our secret.

```
secretBytesDecrypted :=
[]byte(fmt.Sprintf("%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n",
    artifact.DisplayName,
    strings.Replace(artifact.Notes, "\n", string(65000), -1),
    artifact.Username,
    artifact.Email,
    artifact.CopyMethod,
    artifact.Password,
    artifact.CustomField01Name,
    artifact.CustomField01Data,
    artifact.CustomField02Name,
    artifact.CustomField02Data,
    artifact.CustomField03Name,
    artifact.CustomField03Data,
    artifact.CustomField04Name,
    artifact.CustomField04Data,
    artifact.CustomField05Name,
    artifact.CustomField05Data,
    artifact.CustomField06Name,
    artifact.CustomField06Data,
))
```

Step 3

We create some salt in order to prevent rainbow table attacks, cf. [Wikipedia](#): `saltBytes := uuid.NewV4().Bytes()`. Here, we use an UUID v4 which is not predictable.

Step 4

Now, we are able to derive a key and a vector out of the master password and the random salt, regarding RFC 2898:

```
keyLength := 256
rfc2898Iterations := 6

keyVectorData := pbkdf2.Key(masterPassword, saltBytes, rfc2898Iterations,
    (keyLength/8)+aes.BlockSize, sha1.New)
keyBytes := keyVectorData[:keyLength/8]
vectorBytes := keyVectorData[keyLength/8:]
```

Step 5

The desired CBC mode works with whole blocks. Thus, we have to check if our data is aligned to a full block. If not, we have to pad it:

```
if len(secretBytesDecrypted)%aes.BlockSize != 0 {
    numberNecessaryBlocks := int(math.Ceil(float64(len(secretBytesDecrypted)) /
float64(aes.BlockSize)))
    enhanced := make([]byte, numberNecessaryBlocks*aes.BlockSize)
    copy(enhanced, secretBytesDecrypted)
    secretBytesDecrypted = enhanced
}
```

Step 6

Now we create an AES cipher: `aesBlockEncrypter, aesErr := aes.NewCipher(keyBytes)`

Step 7

We reserve the necessary memory for the encrypted data: `encryptedData := make([]byte, len(secretBytesDecrypted))`. In case of AES-CBC, the encrypted data had the same length as the unencrypted data.

Step 8

Now, we should create the encrypter and encrypt the data:

```
aesEncrypter := cipher.NewCBCEncrypter(aesBlockEncrypter, vectorBytes)
aesEncrypter.CryptBlocks(encryptedData, secretBytesDecrypted)
```

Now, the encrypted data is inside the `encryptedData` variable.

Step 9

The encrypted data must be stored. But not only the data: Without the salt, the encrypted data could not be decrypted. Thus, we must use some kind of file format to manage this. Here, we encode the encrypted data as base64, cf. [Wikipedia](#):

```
encodedBytes := make([]byte, base64.StdEncoding.EncodedLen(len(encryptedData)))
base64.StdEncoding.Encode(encodedBytes, encryptedData)
```

Next, we define our file content and our own file format. The format looks like this: `salt[0x10]base64 content`. First, we store the salt. In order to mark the beginning of the base64 content, we store the byte `10`. This works, because base64 does not use this value. Therefore, we could find the start of base64 by search the first occurrence of `10` from the end to the beginning of the file.

```
fileContent := make([]byte, len(saltBytes))
copy(fileContent, saltBytes)
fileContent = append(fileContent, 10)
fileContent = append(fileContent, encodedBytes...)
```

Step 10

Finally, we could write our file: `writeErr := ioutil.WriteFile("my secret.data", fileContent, 0644).`

Decryption

Introduction and data

As for encryption, we need some data to work with. Thus, we assume we have an encrypted file and the mentioned structure `secret`. The goal is to read the encrypted data from the file, decrypt it, and create an instance of the structure.

Step 1

The first step is identical to the encryption: We need a kind of master password to decrypt the `secret`: `masterPassword := "PASS".`

Step 2

Now, we read the encrypted data from file: `encryptedFileData, bytesErr := ioutil.ReadFile(filename).`

Step 3

As mentioned before, we could split salt and encrypted data by the delimiter byte `10`, searched backwards from the end to the beginning:

```
for n := len(encryptedFileData) - 1; n > 0; n-- {
    if encryptedFileData[n] == 10 {
        saltBytes = encryptedFileData[:n]
        encryptedBytesBase64 = encryptedFileData[n+1:]
        break
    }
}
```

Step 4

Next, we must decode the base64 encoded bytes:

```
decodedBytes := make([]byte, len(encryptedBytesBase64))
countDecoded, decodedErr := base64.StdEncoding.Decode(decodedBytes, encryptedBytesBase64)
encryptedBytes = decodedBytes[:countDecoded]
```

Step 5

Now, we are able to derive a key and a vector out of the master password and the random salt, regarding RFC 2898:

```
keyLength := 256
rfc2898Iterations := 6

keyVectorData := pbkdf2.Key(masterPassword, saltBytes, rfc2898Iterations,
(keyLength/8)+aes.BlockSize, sha1.New)
keyBytes := keyVectorData[:keyLength/8]
vectorBytes := keyVectorData[keyLength/8:]
```

Step 6

Create an AES cipher: `aesBlockDecrypter, aesErr := aes.NewCipher(keyBytes).`

Step 7

Reserve the necessary memory for the decrypted data: `decryptedData := make([]byte, len(encryptedBytes)).` By definition, it has the same length as the encrypted data.

Step 8

Now, create the decrypter and decrypt the data:

```
aesDecrypter := cipher.NewCBCDecrypter(aesBlockDecrypter, vectorBytes)
aesDecrypter.CryptBlocks(decryptedData, encryptedBytes)
```

Step 9

Convert the read bytes to string: `decryptedString := string(decryptedData).` Because we need lines, split the string: `lines := strings.Split(decryptedString, "\n").`

Step 10

Construct a `secret` out of the lines:

```
artifact := secret{}
artifact.DisplayName = lines[0]
artifact.Notes = lines[1]
artifact.Username = lines[2]
```

```
artifact.Email = lines[3]
artifact.CopyMethod = lines[4]
artifact.Password = lines[5]
artifact.CustomField01Name = lines[6]
artifact.CustomField01Data = lines[7]
artifact.CustomField02Name = lines[8]
artifact.CustomField02Data = lines[9]
artifact.CustomField03Name = lines[10]
artifact.CustomField03Data = lines[11]
artifact.CustomField04Name = lines[12]
artifact.CustomField04Data = lines[13]
artifact.CustomField05Name = lines[14]
artifact.CustomField05Data = lines[15]
artifact.CustomField06Name = lines[16]
artifact.CustomField06Data = lines[17]
```

Finally, re-create the line breaks within the notes field: `artifact.Notes = strings.Replace(artifact.Notes, string(65000), "\n", -1).`

Read Cryptography online: <https://riptutorial.com/go/topic/10065/cryptography>

Chapter 17: Defer

Introduction

A `defer` statement pushes a function call onto a list. The list of saved calls is executed after the surrounding function returns. Defer is commonly used to simplify functions that perform various clean-up actions.

Syntax

- `defer someFunc(args)`
- `defer func(){ //code goes here }()`

Remarks

Defer works by injecting a new stack frame (the called function after the `defer` keyword) into the call stack below the currently executing function. This means that defer is guaranteed to run as long as the stack will be unwound (if your program crashes or gets a `SIGKILL`, defer will not execute).

Examples

Defer Basics

A *defer statement* in Go is simply a function call marked to be executed at a later time. Defer statement is an ordinary function call prefixed by the keyword `defer`.

```
defer someFunction()
```

A deferred function is executed once the function that contains the `defer` statement returns. Actual call to the deferred function occurs when the enclosing function:

- executes a return statement
- falls off the end
- panics

Example:

```
func main() {
    fmt.Println("First main statement")
    defer logExit("main") // position of defer statement here does not matter
    fmt.Println("Last main statement")
}

func logExit(name string) {
    fmt.Printf("Function %s returned\n", name)
```

```
}
```

Output:

```
First main statement
Last main statement
Function main returned
```

If a function has multiple deferred statements, they form a stack. The last `defer` is the first one to execute after the enclosing function returns, followed by subsequent calls to preceding `defers` in order (below example returns by causing a panic):

```
func main() {
    defer logNum(1)
    fmt.Println("First main statement")
    defer logNum(2)
    defer logNum(3)
    panic("panic occurred")
    fmt.Println("Last main statement") // not printed
    defer logNum(3) // not deferred since execution flow never reaches this line
}

func logNum(i int) {
    fmt.Printf("Num %d\n", i)
}
```

Output:

```
First main statement
Num 3
Num 2
Num 1
panic: panic occurred

goroutine 1 [running]:
....
```

Note that deferred functions have their arguments evaluated at the time `defer` executes:

```
func main() {
    i := 1
    defer logNum(i) // deferred function call: logNum(1)
    fmt.Println("First main statement")
    i++
    defer logNum(i) // deferred function call: logNum(2)
    defer logNum(i*i) // deferred function call: logNum(4)
    return // explicit return
}

func logNum(i int) {
    fmt.Printf("Num %d\n", i)
}
```

Output:

```
First main statement
Num 4
Num 2
Num 1
```

If a function has named return values, a deferred anonymous function within that function can access and update the returned value even after the function has returned:

```
func main() {
    fmt.Println(plusOne(1)) // 2
    return
}

func plusOne(i int) (result int) { // overkill! only for demonstration
    defer func() {result += 1}() // anonymous function must be called by adding ()

    // i is returned as result, which is updated by deferred function above
    // after execution of below return
    return i
}
```

Finally, a `defer` statement is generally used operations that often occur together. For example:

- open and close a file
- connect and disconnect
- lock and unlock a mutex
- mark a waitgroup as done (`defer wg.Done()`)

This use ensures proper release of system resources irrespective of the flow of execution.

```
resp, err := http.Get(url)
if err != nil {
    return err
}
defer resp.Body.Close() // Body will always get closed
```

Deferred Function Calls

Deferred function calls serve a similar purpose to things like `finally` blocks in languages like Java: they ensure that some function will be executed when the outer function returns, regardless of if an error occurred or which return statement was hit in cases with multiple returns. This is useful for cleaning up resources that must be closed like network connections or file pointers. The `defer` keyword indicates a deferred function call, similarly to the `go` keyword initiating a new goroutine. Like a `go` call, function arguments are evaluated immediately, but unlike a `go` call, deferred functions are not executed concurrently.

```
func MyFunc() {
    conn := GetConnection() // Some kind of connection that must be closed.
    defer conn.Close()      // Will be executed when MyFunc returns, regardless of how.
    // Do some things...
    if someCondition {
        return // conn.Close() will be called
    }
}
```



```
}  
// Do more things  
} // Implicit return - conn.Close() will still be called
```

Note the use of `conn.Close()` instead of `conn.Close` - you're not just passing in a function, you're deferring a full function *call*, including its arguments. Multiple function calls can be deferred in the same outer function, and each will be executed once in reverse order. You can also defer closures - just don't forget the parens!

```
defer func(){  
    // Do some cleanup  
}()
```

Read Defer online: <https://riptutorial.com/go/topic/2795/defer>

Chapter 18: Developing for Multiple Platforms with Conditional Compiling

Introduction

Platform based conditional compiling comes in two forms in Go, one is with file suffixes and the other is with build tags.

Syntax

- After "`// +build`", a single platform or a list can follow
- Platform can be reverted by preceding it by `!` sign
- List of space separated platforms are ORed together

Remarks

Caveats for build tags:

- The `// +build` constraint must be placed at the top of the file, even before package clause.
- It must be followed by one blank line to separate from package comments.

List of valid platforms for both build tags and file suffixes

android

darwin

dragonfly

freebsd

linux

netbsd

openbsd

plan9

solaris

windows

Refer to `$GOOS` list in <https://golang.org/doc/install/source#environment> for the most up-to-date platform list.

Examples

Build tags

```
// +build linux

package lib

var OnlyAccessibleInLinux int // Will only be compiled in Linux
```

Negate a platform by placing `!` before it:

```
// +build !windows

package lib

var NotWindows int // Will be compiled in all platforms but not Windows
```

List of platforms can be specified by separating them with spaces

```
// +build linux darwin plan9

package lib

var SomeUnix int // Will be compiled in linux, darwin and plan9 but not on others
```

File suffix

If you name your file `lib_linux.go`, all the content in that file will only be compiled in linux environments:

```
package lib

var OnlyCompiledInLinux string
```

Defining separate behaviours in different platforms

Different platforms can have separate implementations of the same method. This example also illustrates how build tags and file suffixes can be used together.

File `main.go`:

```
package main

import "fmt"

func main() {
    fmt.Println("Hello World from Conditional Compilation Doc!")
    printDetails()
}
```

details.go:

```
// +build !windows

package main

import "fmt"

func printDetails() {
    fmt.Println("Some specific details that cannot be found on Windows")
}
```

details_windows.go:

```
package main

import "fmt"

func printDetails() {
    fmt.Println("Windows specific details")
}
```

Read [Developing for Multiple Platforms with Conditional Compiling](https://riptutorial.com/go/topic/8599/developing-for-multiple-platforms-with-conditional-compiling) online:

<https://riptutorial.com/go/topic/8599/developing-for-multiple-platforms-with-conditional-compiling>

Chapter 19: Error Handling

Introduction

In Go, unexpected situations are handled using **errors**, not exceptions. This approach is more similar to that of C, using `errno`, than to that of Java or other object-oriented languages, with their `try/catch` blocks. However, an error is not an integer but an interface.

A function that may fail typically returns an **error** as its last return value. If this error is not **nil**, something went wrong, and the caller of the function should take action accordingly.

Remarks

Note how in Go you don't *raise* an error. Instead, you *return* an error in case of failure.

If a function can fail, the last returned value is generally an `error` type.

```
// This method doesn't fail
func DoSomethingSafe() {
}

// This method can fail
func DoSomething() (error) {
}

// This method can fail and, when it succeeds,
// it returns a string.
func DoAndReturnSomething() (string, error) {
}
```

Examples

Creating an error value

The simplest way to create an error is by using the `errors` package.

```
errors.New("this is an error")
```

If you want to add additional information to an error, the `fmt` package also provides a useful error creation method:

```
var f float64
fmt.Errorf("error with some additional information: %g", f)
```

Here's a full example, where the error is returned from a function:

```
package main
```

```

import (
    "errors"
    "fmt"
)

var ErrThreeNotFound = errors.New("error 3 is not found")

func main() {
    fmt.Println(DoSomething(1)) // succeeds! returns nil
    fmt.Println(DoSomething(2)) // returns a specific error message
    fmt.Println(DoSomething(3)) // returns an error variable
    fmt.Println(DoSomething(4)) // returns a simple error message
}

func DoSomething(someID int) error {
    switch someID {
    case 3:
        return ErrThreeNotFound
    case 2:
        return fmt.Errorf("this is an error with extra info: %d", someID)
    case 1:
        return nil
    }

    return errors.New("this is an error")
}

```

[Open in Playground](#)

Creating a custom error type

In Go, an error is represented by any value that can describe itself as string. Any type that implement the built-in `error` interface is an error.

```

// The error interface is represented by a single
// Error() method, that returns a string representation of the error
type error interface {
    Error() string
}

```

The following example shows how to define a new error type using a string composite literal.

```

// Define AuthorizationError as composite literal
type AuthorizationError string

// Implement the error interface
// In this case, I simply return the underlying string
func (e AuthorizationError) Error() string {
    return string(e)
}

```

I can now use my custom error type as error:

```

package main

```

```

import (
    "fmt"
)

// Define AuthorizationError as composite literal
type AuthorizationError string

// Implement the error interface
// In this case, I simply return the underlying string
func (e AuthorizationError) Error() string {
    return string(e)
}

func main() {
    fmt.Println(DoSomething(1)) // succeeds! returns nil
    fmt.Println(DoSomething(2)) // returns an error message
}

func DoSomething(someID int) error {
    if someID != 1 {
        return AuthorizationError("Action not allowed!")
    }

    // do something here

    // return a nil error if the execution succeeded
    return nil
}

```

Returning an error

In Go you don't *raise* an error. Instead, you *return* an `error` in case of failure.

```

// This method can fail
func DoSomething() error {
    // functionThatReportsOK is a side-effecting function that reports its
    // state as a boolean. NOTE: this is not a good practice, so this example
    // turns the boolean value into an error. Normally, you'd rewrite this
    // function if it is under your control.
    if ok := functionThatReportsOK(); !ok {
        return errors.New("functionThatReportsSuccess returned a non-ok state")
    }

    // The method succeeded. You still have to return an error
    // to properly obey to the method signature.
    // But in this case you return a nil error.
    return nil
}

```

If the method returns multiple values (and the execution can fail), then the standard convention is to return the error as the last argument.

```

// This method can fail and, when it succeeds,
// it returns a string.
func DoAndReturnSomething() (string, error) {
    if os.Getenv("ERROR") == "1" {
        return "", errors.New("The method failed")
    }
}

```

```

    }

    s := "Success!"

    // The method succeeded.
    return s, nil
}

result, err := DoAndReturnSomething()
if err != nil {
    panic(err)
}

```

Handling an error

In Go errors can be returned from a function call. The convention is that if a method can fail, the last returned argument is an `error`.

```

func DoAndReturnSomething() (string, error) {
    if os.Getenv("ERROR") == "1" {
        return "", errors.New("The method failed")
    }

    // The method succeeded.
    return "Success!", nil
}

```

You use multiple variable assignments to check if the method failed.

```

result, err := DoAndReturnSomething()
if err != nil {
    panic(err)
}

// This is executed only if the method didn't return an error
fmt.Println(result)

```

If you are not interested in the error, you can simply ignore it by assigning it to `_`.

```

result, _ := DoAndReturnSomething()
fmt.Println(result)

```

Of course, ignoring an error can have serious implications. Therefore, this is generally not recommended.

If you have multiple method calls, and one or more methods in the chain may return an error, you should propagate the error to the first level that can handle it.

```

func Foo() error {
    return errors.New("I failed!")
}

func Bar() (string, error) {
    err := Foo()

```



```

    if err != nil {
        return "", err
    }

    return "I succeeded", nil
}

func Baz() (string, string, error) {
    res, err := Bar()
    if err != nil {
        return "", "", err
    }

    return "Foo", "Bar", nil
}

```

Recovering from panic

A common mistake is to declare a slice and start requesting indexes from it without initializing it, which leads to an "index out of range" panic. The following code explains how to recover from the panic without exiting the program, which is the normal behavior for a panic. In most situations, returning an error in this fashion rather than exiting the program on a panic is only useful for development or testing purposes.

```

type Foo struct {
    Is []int
}

func main() {
    fp := &Foo{}
    if err := fp.Panic(); err != nil {
        fmt.Printf("Error: %v", err)
    }
    fmt.Println("ok")
}

func (fp *Foo) Panic() (err error) {
    defer PanicRecovery(&err)
    fp.Is[0] = 5
    return nil
}

func PanicRecovery(err *error) {

    if r := recover(); r != nil {
        if _, ok := r.(runtime.Error); ok {
            //fmt.Println("Panicing")
            //panic(r)
            *err = r.(error)
        } else {
            *err = r.(error)
        }
    }
}

```

The use of a separate function (rather than closure) allows re-use of the same function in other functions prone to panic.

Read Error Handling online: <https://riptutorial.com/go/topic/785/error-handling>

Chapter 20: Executing Commands

Examples

Timing Out with Interrupt and then Kill

```
c := exec.Command(name, arg...)
b := &bytes.Buffer{}
c.Stdout = b
c.Stdin = stdin
if err := c.Start(); err != nil {
    return nil, err
}
timedOut := false
intTimer := time.AfterFunc(timeout, func() {
    log.Printf("Process taking too long. Interrupting: %s %s", name, strings.Join(arg, " "))
    c.Process.Signal(os.Interrupt)
    timedOut = true
})
killTimer := time.AfterFunc(timeout*2, func() {
    log.Printf("Process taking too long. Killing: %s %s", name, strings.Join(arg, " "))
    c.Process.Signal(os.Kill)
    timedOut = true
})
err := c.Wait()
intTimer.Stop()
killTimer.Stop()
if timedOut {
    log.Print("the process timed out\n")
}
```

Simple Command Execution

```
// Execute a command and capture standard out. exec.Command creates the command
// and then the chained Output method gets standard out. Use CombinedOutput()
// if you want both standard out and stderr output
out, err := exec.Command("echo", "foo").Output()
if err != nil {
    log.Fatal(err)
}
```

Executing a Command then Continue and Wait

```
cmd := exec.Command("sleep", "5")

// Does not wait for command to complete before returning
err := cmd.Start()
if err != nil {
    log.Fatal(err)
}

// Wait for cmd to Return
err = cmd.Wait()
```

```
log.Printf("Command finished with error: %v", err)
```

Running a Command twice

A Cmd cannot be reused after calling its Run, Output or CombinedOutput methods

Running a command twice *will **not** work*:

```
cmd := exec.Command("xte", "key XF86AudioPlay")
_ := cmd.Run() // Play audio key press
// .. do something else
err := cmd.Run() // Pause audio key press, fails
```

Error: exec: already started

Rather, one must use **two separate** `exec.Command`. You might also need some delay between commands.

```
cmd := exec.Command("xte", "key XF86AudioPlay")
_ := cmd.Run() // Play audio key press
// .. wait a moment
cmd := exec.Command("xte", "key XF86AudioPlay")
_ := cmd.Run() // Pause audio key press
```

Read Executing Commands online: <https://riptutorial.com/go/topic/1097/executing-commands>

Chapter 21: File I/O

Syntax

- `file, err := os.Open(name)` // Opens a file in read-only mode. A non-nil error is returned if the file could not be opened.
- `file, err := os.Create(name)` // Creates or opens a file if it already exists in write-only mode. The file is overwritten to if it already exists. A non-nil error is returned if the file could not be opened.
- `file, err := os.OpenFile(name, flags, perm)` // Opens a file in the mode specified by the flags. A non-nil error is returned if the file could not be opened.
- `data, err := ioutil.ReadFile(name)` // Reads the entire file and returns it. A non-nil error is returned if the entire file could not be read.
- `err := ioutil.WriteFile(name, data, perm)` // Creates or overwrites a file with the provided data and UNIX permission bits. A non-nil error is returned if the file failed to be written to.
- `err := os.Remove(name)` // Deletes a file. A non-nil error is returned if the file could not be deleted.
- `err := os.RemoveAll(name)` // Deletes a file or whole directory hierarchy. A non-nil error is returned if the file or directory could not be deleted.
- `err := os.Rename(oldName, newName)` // Renames or moves a file (can be across directories). A non-nil error is returned if the file could not be moved.

Parameters

Parameter	Details
name	A filename or path of type string. For example: "hello.txt".
err	An <code>error</code> . If not <code>nil</code> , it represents an error that occurred when the function was called.
file	A file handler of type <code>*os.File</code> returned by the <code>os</code> package file related functions. It implements an <code>io.ReadWriter</code> , meaning you can call <code>Read(data)</code> and <code>Write(data)</code> on it. Note that these functions may not be able to be called depending on the open flags of the file.
data	A slice of bytes (<code>[]byte</code>) representing the raw data of a file.
perm	The UNIX permission bits used to open a file with of type <code>os.FileMode</code> . Several constants are available to help with the use of permission bits.
flag	File open flags that determine the methods that can be called on the file handler of type <code>int</code> . Several constants are available to help with the use of flags. They are: <code>os.O_RDONLY</code> , <code>os.O_WRONLY</code> , <code>os.O_RDWR</code> , <code>os.O_APPEND</code> , <code>os.O_CREATE</code> , <code>os.O_EXCL</code> , <code>os.O_SYNC</code> , and <code>os.O_TRUNC</code> .

Examples

Reading and writing to a file using ioutil

A simple program that writes "Hello, world!" to `test.txt`, reads back the data, and prints it out. Demonstrates simple file I/O operations.

```
package main

import (
    "fmt"
    "io/ioutil"
)

func main() {
    hello := []byte("Hello, world!")

    // Write `Hello, world!` to test.txt that can read/written by user and read by others
    err := ioutil.WriteFile("test.txt", hello, 0644)
    if err != nil {
        panic(err)
    }

    // Read test.txt
    data, err := ioutil.ReadFile("test.txt")
    if err != nil {
        panic(err)
    }

    // Should output: `The file contains: Hello, world!`
    fmt.Println("The file contains: " + string(data))
}
```

Listing all the files and folders in the current directory

```
package main

import (
    "fmt"
    "io/ioutil"
)

func main() {
    files, err := ioutil.ReadDir(".")
    if err != nil {
        panic(err)
    }

    fmt.Println("Files and folders in the current directory:")

    for _, fileInfo := range files {
        fmt.Println(fileInfo.Name())
    }
}
```

Listing all folders in the current directory

```
package main

import (
    "fmt"
    "io/ioutil"
)

func main() {
    files, err := ioutil.ReadDir(".")
    if err != nil {
        panic(err)
    }

    fmt.Println("Folders in the current directory:")

    for _, fileInfo := range files {
        if fileInfo.IsDir() {
            fmt.Println(fileInfo.Name())
        }
    }
}
```

Read File I/O online: <https://riptutorial.com/go/topic/1033/file-i-o>

Chapter 22: Fmt

Examples

Stringer

The `fmt.Stringer` interface requires a single method, `String() string` to be satisfied. The `String` method defines the "native" string format for that value, and is the default representation if the value is provided to any of the `fmt` packages formatting or printing routines.

```
package main

import (
    "fmt"
)

type User struct {
    Name  string
    Email string
}

// String satisfies the fmt.Stringer interface for the User type
func (u User) String() string {
    return fmt.Sprintf("%s <%s>", u.Name, u.Email)
}

func main() {
    u := User{
        Name:  "John Doe",
        Email: "johndoe@example.com",
    }

    fmt.Println(u)
    // output: John Doe <johndoe@example.com>
}
```

[Playground](#)

Basic fmt

Package `fmt` implements formatted I/O using format *verbs*:

```
%v    // the value in a default format
%T    // a Go-syntax representation of the type of the value
%s    // the uninterpreted bytes of the string or slice
```

Format Functions

There are **4** main function types in `fmt` and several variations within.

Print

```
fmt.Print("Hello World")           // prints: Hello World
fmt.Println("Hello World")         // prints: Hello World\n
fmt.Printf("Hello %s", "World")    // prints: Hello World
```

Sprint

```
formattedString := fmt.Sprintf("%v %s", 2, "words") // returns string "2 words"
```

Fprint

```
byteCount, err := fmt.Fprint(w, "Hello World") // writes to io.Writer w
```

Fprint can be used, inside http handlers:

```
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello %s!", "Browser")
} // Writes: "Hello Browser!" onto http response
```

Scan

Scan scans text read from standard input.

```
var s string
fmt.Scanln(&s) // pass pointer to buffer
// Scanln is similar to fmt.Scan(), but it stops scanning at new line.
fmt.Println(s) // whatever was inputted
```

Stringer Interface

Any value which has a `String()` method implements the `fmt` **interface** `Stringer`

```
type Stringer interface {
    String() string
}
```

Read Fmt online: <https://riptutorial.com/go/topic/2938/fmt>

Chapter 23: Functions

Introduction

Functions in Go provide organized, reusable code to perform a set of actions. Functions simplify the coding process, prevent redundant logic, and make code easier to follow. This topic describes the declaration and utilization of functions, arguments, parameters, return statements and scopes in Go.

Syntax

- `func()` // function type with no arguments and no return value
- `func(x int) int` // accepts integer and returns an integer
- `func(a, b int, z float32) bool` // accepts 2 integers, one float and returns a boolean
- `func(prefix string, values ...int)` // "variadic" function which accepts one string and one or more number of integers
- `func() (int, bool)` // function returning two values
- `func(a, b int, z float64, opt ...interface{}) (success bool)` // accepts 2 integers, one float and one or more number of interfaces and returns named boolean value (which is already initialized inside of function)

Examples

Basic Declaration

A simple function that does not accept any parameters and does not return any values:

```
func SayHello() {  
    fmt.Println("Hello!")  
}
```

Parameters

A function can optionally declare a set of parameters:

```
func SayHelloToMe(firstName, lastName string, age int) {  
    fmt.Printf("Hello, %s %s!\n", firstName, lastName)  
    fmt.Printf("You are %d", age)  
}
```

Notice that the type for `firstName` is omitted because it is identical to `lastName`.

Return Values

A function can return one or more values to the caller:

```
func AddAndMultiply(a, b int) (int, int) {
    return a+b, a*b
}
```

The second return value can also be the error var :

```
import errors

func Divide(dividend, divisor int) (int, error) {
    if divisor == 0 {
        return 0, errors.New("Division by zero forbidden")
    }
    return dividend / divisor, nil
}
```

Two important things must be noted:

- The parenthesis may be omitted for a single return value.
- Each `return` statement must provide a value for **all** declared return values.

Named Return Values

Return values can be assigned to a local variable. An empty `return` statement can then be used to return their current values. This is known as "*naked*" return. Naked return statements should be used only in short functions as they harm readability in longer functions:

```
func Inverse(v float32) (reciprocal float32) {
    if v == 0 {
        return
    }
    reciprocal = 1 / v
    return
}
```

[play it on playground](#)

```
//A function can also return multiple values
func split(sum int) (x, y int) {
    x = sum * 4 / 9
    y = sum - x
    return
}
```

[play it on playground](#)

Two important things must be noted:

- The parenthesis around the return values are **mandatory**.
- An empty `return` statement must always be provided.

Literal functions & closures

A simple literal function, printing `Hello!` to stdout:

```
package main

import "fmt"

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

[play it on playground](#)

A literal function, printing the `str` argument to stdout:

```
package main

import "fmt"

func main() {
    func(str string) {
        fmt.Println(str)
    }("Hello!")
}
```

[play it on playground](#)

A literal function, closing over the variable `str`:

```
package main

import "fmt"

func main() {
    str := "Hello!"
    func() {
        fmt.Println(str)
    }()
}
```

[play it on playground](#)

It is possible to assign a literal function to a variable:

```
package main

import (
    "fmt"
)

func main() {
```

```
str := "Hello!"
anon := func() {
    fmt.Println(str)
}
anon()
```

[play it on playground](#)

Variadic functions

A variadic function can be called with any number of **trailing** arguments. Those elements are stored in a slice.

```
package main

import "fmt"

func variadic(strs ...string) {
    // strs is a slice of string
    for i, str := range strs {
        fmt.Printf("%d: %s\n", i, str)
    }
}

func main() {
    variadic("Hello", "Goodbye")
    variadic("Str1", "Str2", "Str3")
}
```

[play it on playground](#)

You can also give a slice to a variadic function, with ...:

```
func main() {
    strs := []string {"Str1", "Str2", "Str3"}

    variadic(strs...)
}
```

[play it on playground](#)

Read Functions online: <https://riptutorial.com/go/topic/373/functions>

Chapter 24: Getting Started With Go Using Atom

Introduction

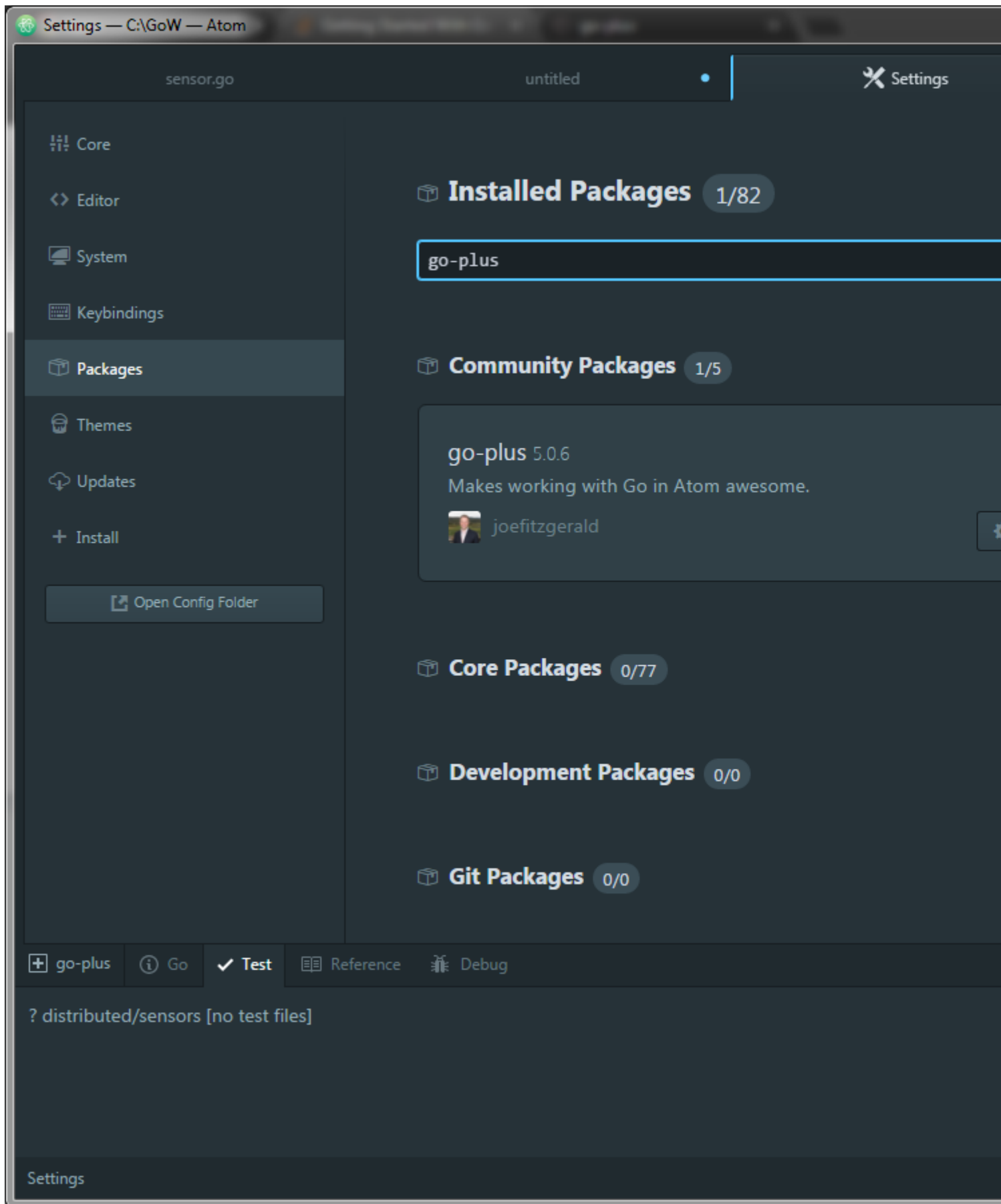
After installing go (<http://www.riptutorial.com/go/topic/198/getting-started-with-go>) you'll need an environment. An efficient and free way to get you started is using Atom text editor (<https://atom.io>) and gulp. A question that maybe crossed your mind is *why use gulp?*. We need gulp for auto-completion. Let's get Started!

Examples

Get, Install And Setup Atom & Gulp

1. Install Atom. You can get atom from [here](#)
2. Go to Atom settings (ctrl+,). Packages -> Install go-plus package ([go-plus](#))

After Installing go-plus in Atom:



3. Get these dependencies using go get or another dependency manager: (open a console and run these commands)

```
go get -u golang.org/x/tools/cmd/goimports
```

```
go get -u golang.org/x/tools/cmd/gorename

go get -u github.com/sqs/goreturns

go get -u github.com/nsf/gocode

go get -u github.com/alecthomas/gometalinter

go get -u github.com/zmb3/gogetdoc

go get -u github.com/rogppe/godef

go get -u golang.org/x/tools/cmd/guru
```

4. Install Gulp ([Gulpjs](#)) using npm or any other package manager ([gulp-getting-started-doc](#)):

```
$ npm install --global gulp
```

Create \$GO_PATH/gulpfile.js

```
var gulp = require('gulp');
var path = require('path');
var shell = require('gulp-shell');

var goPath = 'src/mypackage/**/*.go';

gulp.task('compilepkg', function() {
  return gulp.src(goPath, {read: false})
    .pipe(shell(['go install <%= stripPath(file.path) %>'],
      {
        templateData: {
          stripPath: function(filePath) {
            var subPath = filePath.substring(process.cwd().length + 5);
            var pkg = subPath.substring(0, subPath.lastIndexOf(path.sep));
            return pkg;
          }
        }
      }
    ))
  );
});

gulp.task('watch', function() {
  gulp.watch(goPath, ['compilepkg']);
});
```

In the code above we defined a *compilepkg* task that will be triggered every time any go file in goPath (src/mypackage/) or subdirectories changes. the task will run the shell command go install changed_file.go

After creating the gulp file in go path and define the task open a command line and run:

```
gulp watch
```

You'll see something like this everytime any file changes:


```
Ali@Ali-PC MINGW64 /c/GoW
$ gulp watch
[22:30:21] Using gulpfile C:\GoW\gulpfile.js
[22:30:21] Starting 'watch'...
[22:30:22] Finished 'watch' after 18 ms
[22:30:30] Starting 'compilepkg'...
[22:30:30] Finished 'compilepkg' after 163 ms
|
```

Create \$GO_PATH/mypackage/source.go

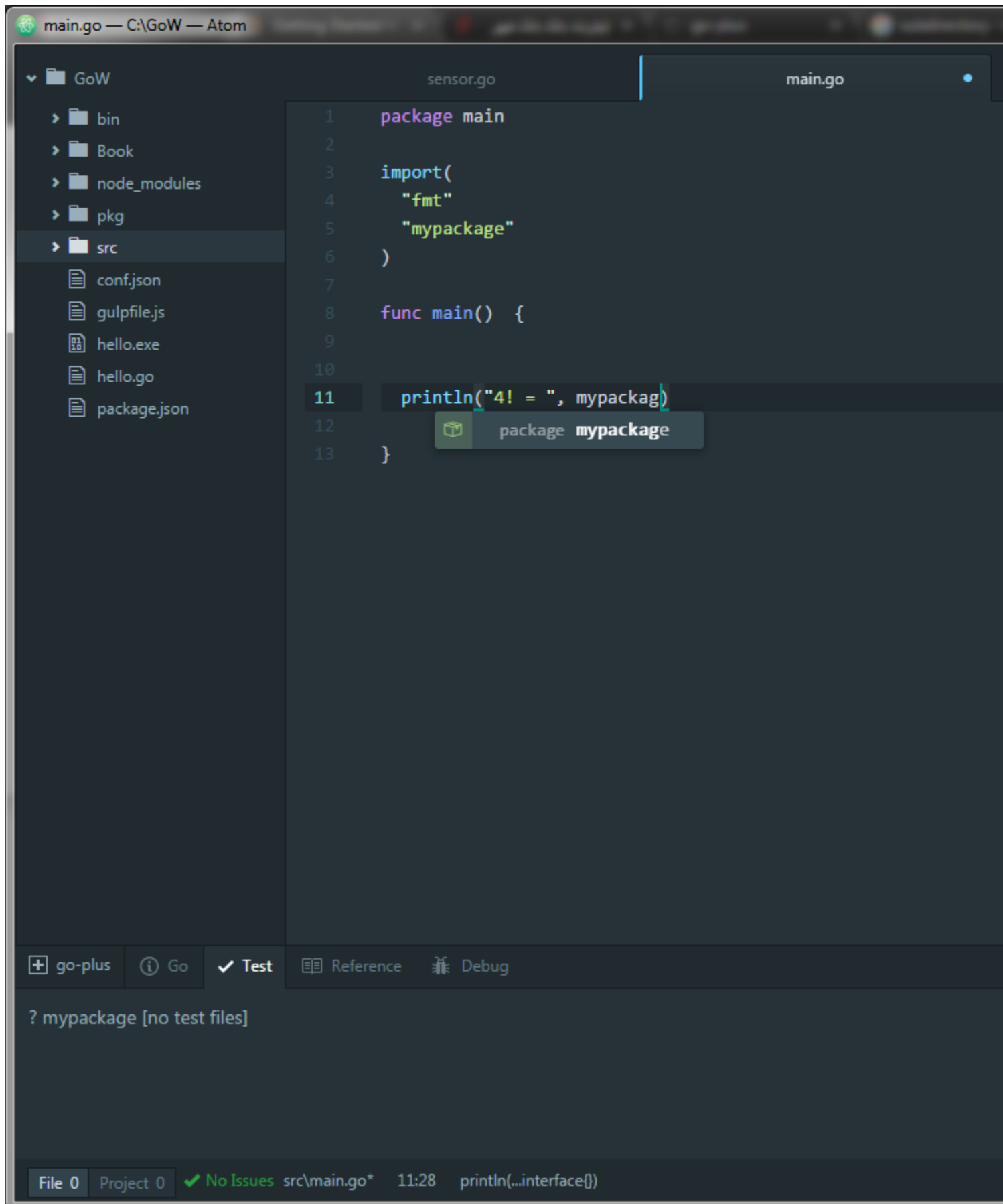
```
package mypackage

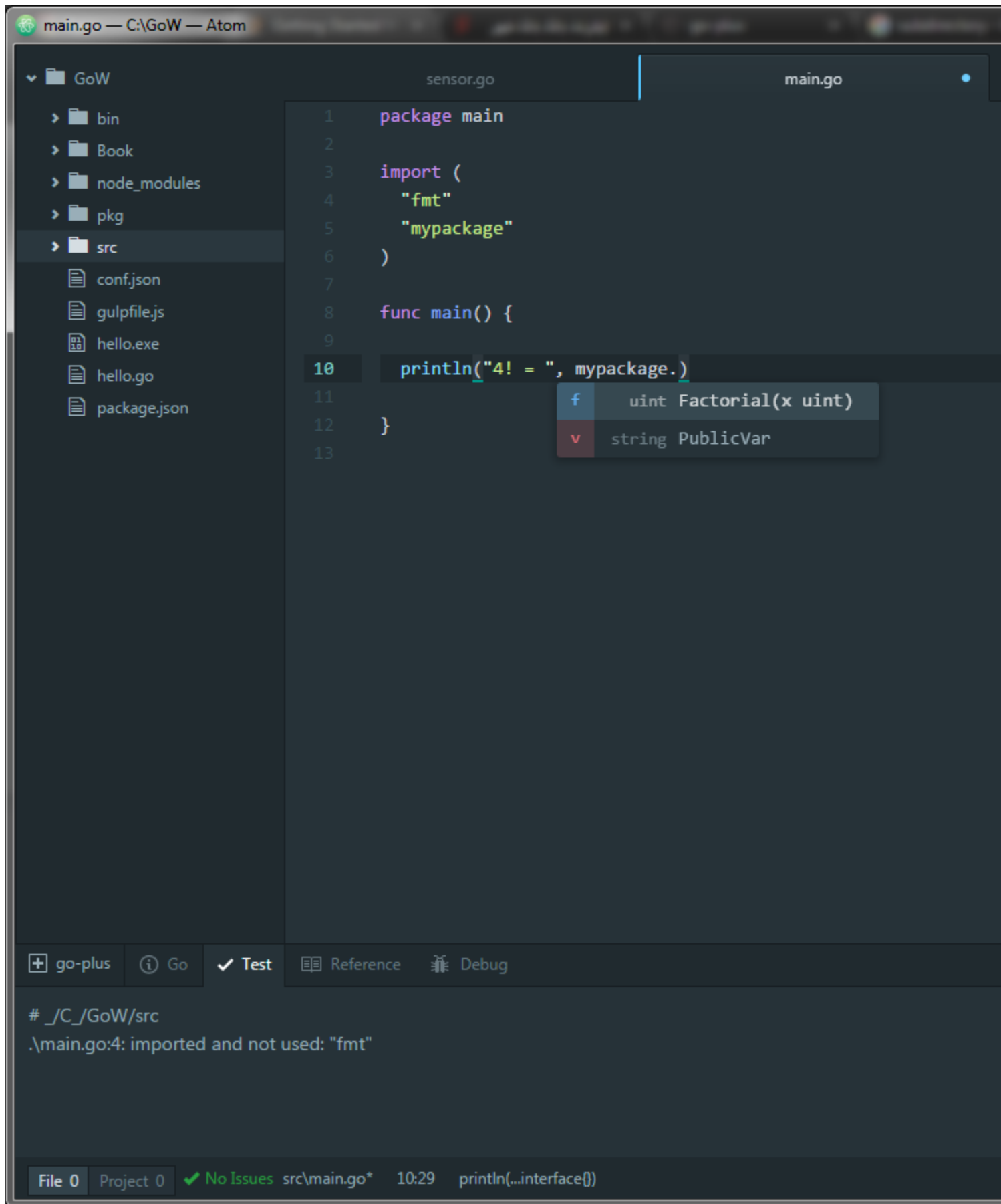
var PublicVar string = "Hello, dear reader!"

//Calculates the factorial of given number recursively!
func Factorial(x uint) uint {
    if x == 0 {
        return 1
    }
    return x * Factorial(x-1)
}
```

Creating \$GO_PATH/main.go

Now you can start writing your own go code with auto-completion using Atom and Gulp:

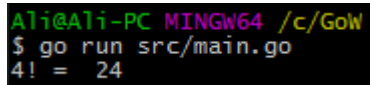




```
package main
```

```
import (  
    "fmt"
```

```
    "mypackage"  
)  
  
func main() {  
    println("4! = ", mypackage.Factorial(4))  
}
```

A terminal window with a black background and green text. The prompt is 'Ali@Ali-PC MINGW64 /c/GoW'. The command '\$ go run src/main.go' has been executed, resulting in the output '4! = 24'.

```
Ali@Ali-PC MINGW64 /c/GoW  
$ go run src/main.go  
4! = 24
```

Read Getting Started With Go Using Atom online: <https://riptutorial.com/go/topic/8592/getting-started-with-go-using-atom>

Chapter 25: gob

Introduction

Gob is a Go specific serialisation method. it has support for all Go data types except for channels and functions. Gob also encodes the type information into the serialised form, what makes it different from say XML is that it is much more efficient.

The inclusion of type information makes encoding and decoding fairly robust to differences between encoder and decoder.

Examples

How to encode data and write to file with gob?

```
package main

import (
    "encoding/gob"
    "os"
)

type User struct {
    Username string
    Password string
}

func main() {

    user := User{
        "zola",
        "supersecretpassword",
    }

    file, _ := os.Create("user.gob")

    defer file.Close()

    encoder := gob.NewEncoder(file)

    encoder.Encode(user)

}
```

How to read data from file and decode with go?

```
package main

import (
    "encoding/gob"
    "fmt"
    "os"
)
```

```

)

type User struct {
    Username string
    Password string
}

func main() {

    user := User{}

    file, _ := os.Open("user.gob")

    defer file.Close()

    decoder := gob.NewDecoder(file)

    decoder.Decode(&user)

    fmt.Println(user)

}

```

How to encode an interface with gob?

```

package main

import (
    "encoding/gob"
    "fmt"
    "os"
)

type User struct {
    Username string
    Password string
}

type Admin struct {
    Username string
    Password string
    IsAdmin bool
}

type Deleter interface {
    Delete()
}

func (u User) Delete() {
    fmt.Println("User ==> Delete() ")
}

func (a Admin) Delete() {
    fmt.Println("Admin ==> Delete() ")
}

func main() {

    user := User{

```

```

        "zola",
        "supersecretpassword",
    }

    admin := Admin{
        "john",
        "supersecretpassword",
        true,
    }

    file, _ := os.Create("user.gob")

    adminFile, _ := os.Create("admin.gob")

    defer file.Close()

    defer adminFile.Close()

    gob.Register(User{}) // registering the type allows us to encode it

    gob.Register(Admin{}) // registering the type allows us to encode it

    encoder := gob.NewEncoder(file)

    adminEncoder := gob.NewEncoder(adminFile)

    InterfaceEncode(encoder, user)

    InterfaceEncode(adminEncoder, admin)
}

func InterfaceEncode(encoder *gob.Encoder, d Deleter) {

    if err := encoder.Encode(&d); err != nil {
        fmt.Println(err)
    }

}

```

How to decode an interface with gob?

```

package main

import (
    "encoding/gob"
    "fmt"
    "log"
    "os"
)

type User struct {
    Username string
    Password string
}

type Admin struct {
    Username string
    Password string
}

```

```

    IsAdmin    bool
}

type Deleter interface {
    Delete()
}

func (u User) Delete() {
    fmt.Println("User ==> Delete()")
}

func (a Admin) Delete() {
    fmt.Println("Admin ==> Delete()")
}

func main() {

    file, _ := os.Open("user.gob")

    adminFile, _ := os.Open("admin.gob")

    defer file.Close()

    defer adminFile.Close()

    gob.Register(User{}) // registering the type allows us to encode it

    gob.Register(Admin{}) // registering the type allows us to encode it

    var admin Deleter

    var user Deleter

    userDecoder := gob.NewDecoder(file)

    adminDecoder := gob.NewDecoder(adminFile)

    user = InterfaceDecode(userDecoder)

    admin = InterfaceDecode(adminDecoder)

    fmt.Println(user)

    fmt.Println(admin)

}

func InterfaceDecode(decoder *gob.Decoder) Deleter {

    var d Deleter

    if err := decoder.Decode(&d); err != nil {
        log.Fatal(err)
    }

    return d

}

```

Read gob online: <https://riptutorial.com/go/topic/8820/gob>

Chapter 26: Goroutines

Introduction

A goroutine is a lightweight thread managed by the Go runtime.

```
go f(x, y, z)
```

starts a new goroutine running

```
f(x, y, z)
```

The evaluation of `f`, `x`, `y`, and `z` happens in the current goroutine and the execution of `f` happens in the new goroutine.

Goroutines run in the same address space, so access to shared memory must be synchronized. The `sync` package provides useful primitives, although you won't need them much in Go as there are other primitives.

Reference: <https://tour.golang.org/concurrency/1>

Examples

Goroutines Basic Program

```
package main

import (
    "fmt"
    "time"
)

func say(s string) {
    for i := 0; i < 5; i++ {
        time.Sleep(100 * time.Millisecond)
        fmt.Println(s)
    }
}

func main() {
    go say("world")
    say("hello")
}
```

A goroutine is a function that is capable of running concurrently with other functions. To create a goroutine we use the keyword `go` followed by a function invocation:

```
package main

import "fmt"
```

```
func f(n int) {  
    for i := 0; i < 10; i++ {  
        fmt.Println(n, ":", i)  
    }  
}  
  
func main() {  
    go f(0)  
    var input string  
    fmt.Scanln(&input)  
}
```

Generally, function call executes all the statements inside the function body and return to the next line. But, with goroutines we return immediately to the next line as it don't wait for the function to complete. So, a call to a `Scanln` function included, otherwise the program has been exited without printing the numbers.

Read Goroutines online: <https://riptutorial.com/go/topic/9776/goroutines>

Chapter 27: HTTP Client

Syntax

- `resp, err := http.Get(url)` // Makes a HTTP GET request with the default HTTP client. A non-nil error is returned if the request fails.
- `resp, err := http.Post(url, bodyType, body)` // Makes a HTTP POST request with the default HTTP client. A non-nil error is returned if the request fails.
- `resp, err := http.PostForm(url, values)` // Makes a HTTP form POST request with the default HTTP client. A non-nil error is returned if the request fails.

Parameters

Parameter	Details
<code>resp</code>	A response of type <code>*http.Response</code> to an HTTP request
<code>err</code>	An <code>error</code> . If not nil, it represents an error that occurred when the function was called.
<code>url</code>	A URL of type <code>string</code> to make a HTTP request to.
<code>bodyType</code>	The MIME type of type <code>string</code> of the body payload of a POST request.
<code>body</code>	An <code>io.Reader</code> (implements <code>Read()</code>) which will be read from until an error is reached to be submitted as the body payload of a POST request.
<code>values</code>	A key-value map of type <code>url.Values</code> . The underlying type is a <code>map[string][]string</code> .

Remarks

It is important to `defer resp.Body.Close()` after every HTTP request that does not return a non-nil error, else resources will be leaked.

Examples

Basic GET

Perform a basic GET request and prints the contents of a site (HTML).

```
package main

import (
```

```

    "fmt"
    "io/ioutil"
    "net/http"
)

func main() {
    resp, err := http.Get("https://example.com/")
    if err != nil {
        panic(err)
    }

    // It is important to defer resp.Body.Close(), else resource leaks will occur.
    defer resp.Body.Close()

    data, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        panic(err)
    }

    // Will print site contents (HTML) to output
    fmt.Println(string(data))
}

```

GET with URL parameters and a JSON response

A request for the top 10 most recently active StackOverflow posts using the Stack Exchange API.

```

package main

import (
    "encoding/json"
    "fmt"
    "net/http"
    "net/url"
)

const apiURL = "https://api.stackexchange.com/2.2/posts?"

// Structs for JSON decoding
type postItem struct {
    Score int    `json:"score"`
    Link   string    `json:"link"`
}

type postsType struct {
    Items []postItem `json:"items"`
}

func main() {
    // Set URL parameters on declaration
    values := url.Values{
        "order": []string{"desc"},
        "sort":  []string{"activity"},
        "site":  []string{"stackoverflow"},
    }

    // URL parameters can also be programmatically set
    values.Set("page", "1")
    values.Set("pagesize", "10")
}

```

```

resp, err := http.Get(apiURL + values.Encode())
if err != nil {
    panic(err)
}

defer resp.Body.Close()

// To compare status codes, you should always use the status constants
// provided by the http package.
if resp.StatusCode != http.StatusOK {
    panic("Request was not OK: " + resp.Status)
}

// Example of JSON decoding on a reader.
dec := json.NewDecoder(resp.Body)
var p postsType
err = dec.Decode(&p)
if err != nil {
    panic(err)
}

fmt.Println("Top 10 most recently active StackOverflow posts:")
fmt.Println("Score", "Link")
for _, post := range p.Items {
    fmt.Println(post.Score, post.Link)
}
}

```

Time out request with a context

1.7+

Timing out an HTTP request with a context can be accomplished with only the standard library (not the subrepos) in 1.7+:

```

import (
    "context"
    "net/http"
    "time"
)

req, err := http.NewRequest("GET", `https://example.net`, nil)
ctx, _ := context.WithTimeout(context.TODO(), 200 * time.Milliseconds)
resp, err := http.DefaultClient.Do(req.WithContext(ctx))
// Be sure to handle errors.
defer resp.Body.Close()

```

Before 1.7

```

import (
    "net/http"
    "time"

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

```

```

    "golang.org/x/net/context/ctxhttp"
)

ctx, err := context.WithTimeout(context.TODO(), 200 * time.Milliseconds)
resp, err := ctxhttp.Get(ctx, http.DefaultClient, "https://www.example.net")
// Be sure to handle errors.
defer resp.Body.Close()

```

Further Reading

For more information on the `context` package see [Context](#).

PUT request of JSON object

The following updates a User object via a PUT request and prints the status code of the request:

```

package main

import (
    "bytes"
    "encoding/json"
    "fmt"
    "net/http"
)

type User struct {
    Name  string
    Email string
}

func main() {
    user := User{
        Name:  "John Doe",
        Email: "johndoe@example.com",
    }

    // initialize http client
    client := &http.Client{}

    // marshal User to json
    json, err := json.Marshal(user)
    if err != nil {
        panic(err)
    }

    // set the HTTP method, url, and request body
    req, err := http.NewRequest(http.MethodPut, "http://api.example.com/v1/user",
bytes.NewBuffer(json))
    if err != nil {
        panic(err)
    }

    // set the request header Content-Type for json
    req.Header.Set("Content-Type", "application/json; charset=utf-8")
    resp, err := client.Do(req)
    if err != nil {
        panic(err)
    }
}

```

```
}  
  
fmt.Println(resp.StatusCode)  
}
```

Read HTTP Client online: <https://riptutorial.com/go/topic/1422/http-client>

Chapter 28: HTTP Server

Remarks

[http.ServeMux](#) provides a multiplexer which calls handlers for HTTP requests.

Alternatives to the standard library multiplexer include:

- [Gorilla Mux](#)

Examples

HTTP Hello World with custom server and mux

```
package main

import (
    "log"
    "net/http"
)

func main() {

    // Create a mux for routing incoming requests
    m := http.NewServeMux()

    // All URLs will be handled by this function
    m.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("Hello, world!"))
    })

    // Create a server listening on port 8000
    s := &http.Server{
        Addr:    ":8000",
        Handler: m,
    }

    // Continue to process new requests until an error occurs
    log.Fatal(s.ListenAndServe())
}
```

Press **Ctrl+C** to stop the process.

Hello World

The typical way to begin writing web servers in golang is to use the standard library `net/http` module.

There is also a tutorial for it [here](#).

The following code also uses it. Here is the simplest possible HTTP server implementation. It

responds "Hello World" to any HTTP request.

Save the following code in a `server.go` file in your workspaces.

```
package main

import (
    "log"
    "net/http"
)

func main() {
    // All URLs will be handled by this function
    // http.HandleFunc uses the DefaultServeMux
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("Hello, world!"))
    })

    // Continue to process new requests until an error occurs
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

You can run the server using:

```
$ go run server.go
```

Or you can compile and run.

```
$ go build server.go
$ ./server
```

The server will listen to the specified port (`:8080`). You can test it with any HTTP client. Here's an example with `cURL`:

```
curl -i http://localhost:8080/
HTTP/1.1 200 OK
Date: Wed, 20 Jul 2016 18:04:46 GMT
Content-Length: 13
Content-Type: text/plain; charset=utf-8

Hello, world!
```

Press `Ctrl+C` to stop the process.

Using a handler function

`HandleFunc` registers the handler function for the given pattern in the server mux (router).

You can pass define an anonymous function, as we have seen in the basic *Hello World* example:

```
http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Hello, world!")
})
```

But we can also pass a `HandlerFunc` type. In other words, we can pass any function that respects the following signature:

```
func FunctionName(w http.ResponseWriter, req *http.Request)
```

We can rewrite the previous example passing the reference to a previously defined `HandlerFunc`. Here's the full example:

```
package main

import (
    "fmt"
    "net/http"
)

// A HandlerFunc function
// Notice the signature of the function
func RootHandler(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintln(w, "Hello, world!")
}

func main() {
    // Here we pass the reference to the `RootHandler` handler function
    http.HandleFunc("/", RootHandler)
    panic(http.ListenAndServe(":8080", nil))
}
```

Of course, you can define several function handlers for different paths.

```
package main

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

func FooHandler(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintln(w, "Hello from foo!")
}

func BarHandler(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintln(w, "Hello from bar!")
}

func main() {
    http.HandleFunc("/foo", FooHandler)
    http.HandleFunc("/bar", BarHandler)

    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Here's the output using `cURL`:

```
➔ ~ curl -i localhost:8080/foo
HTTP/1.1 200 OK
```

```
Date: Wed, 20 Jul 2016 18:23:08 GMT
Content-Length: 16
Content-Type: text/plain; charset=utf-8
```

Hello from foo!

```
→ ~ curl -i localhost:8080/bar
HTTP/1.1 200 OK
Date: Wed, 20 Jul 2016 18:23:10 GMT
Content-Length: 16
Content-Type: text/plain; charset=utf-8
```

Hello from bar!

```
→ ~ curl -i localhost:8080/
HTTP/1.1 404 Not Found
Content-Type: text/plain; charset=utf-8
X-Content-Type-Options: nosniff
Date: Wed, 20 Jul 2016 18:23:13 GMT
Content-Length: 19
```

404 page not found

Create a HTTPS Server

Generate a certificate

In order to run a HTTPS server, a certificate is necessary. Generating a self-signed certificate with `openssl` is done by executing this command:

```
openssl req -x509 -newkey rsa:4096 -sha256 -nodes -keyout key.pem -out cert.pem -subj
"/CN=example.com" -days 3650`
```

The parameters are:

- `req` Use the certificate request tool
- `x509` Creates a self-signed certificate
- `newkey rsa:4096` Creates a new key and certificate by using the RSA algorithms with 4096 bit key length
- `sha256` Forces the SHA256 hashing algorithms which major browsers consider as secure (at the year 2017)
- `nodes` Disables the password protection for the private key. Without this parameter, your server had to ask you for the password each time its starts.
- `keyout` Names the file where to write the key
- `out` Names the file where to write the certificate
- `subj` Defines the domain name for which this certificate is valid
- `days` Fow how many days should this certificate valid? 3650 are approx. 10 years.

Note: A self-signed certificate could be used e.g. for internal projects, debugging, testing, etc. Any browser out there will mention, that this certificate is not safe. In order to avoid this, the certificate must signed by a certification authority. Mostly, this is not available for free. One exception is the

"Let's Encrypt" movement: <https://letsencrypt.org>

The necessary Go code

You can handle configure TLS for the server with the following code. `cert.pem` and `key.pem` are your SSL certificate and key, which were generated with the above command.

```
package main

import (
    "log"
    "net/http"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("Hello, world!"))
    })

    log.Fatal(http.ListenAndServeTLS(":443", "cert.pem", "key.pem", nil))
}
```

Responding to an HTTP Request using Templates

Responses can be written to a `http.ResponseWriter` using templates in Go. This proves as a handy tool if you wish to create dynamic pages.

(To learn how Templates work in Go, please visit the [Go Templates Documentation](#) page.)

Continuing with a simple example to utilise the `html/template` to respond to an HTTP Request:

```
package main

import (
    "html/template"
    "net/http"
    "log"
)

func main(){
    http.HandleFunc("/", WelcomeHandler)
    http.ListenAndServe(":8080", nil)
}

type User struct{
    Name string
    nationality string //unexported field.
}

func check(err error){
    if err != nil{
        log.Fatal(err)
    }
}
```

```
func WelcomeHandler(w http.ResponseWriter, r *http.Request){
    if r.Method == "GET"{
        t,err := template.ParseFiles("welcomeform.html")
        check(err)
        t.Execute(w,nil)
    }else{
        r.ParseForm()
        myUser := User{}
        myUser.Name = r.Form.Get("entered_name")
        myUser.nationality = r.Form.Get("entered_nationality")
        t, err := template.ParseFiles("welcomeresponse.html")
        check(err)
        t.Execute(w,myUser)
    }
}
```

Where, the contents of

1. welcomeform.html are:

```
<head>
    <title> Help us greet you </title>
</head>
<body>
    <form method="POST" action="/">
        Enter Name: <input type="text" name="entered_name">
        Enter Nationality: <input type="text" name="entered_nationality">
        <input type="submit" value="Greet me!">
    </form>
</body>
```

1. welcomeresponse.html are:

```
<head>
    <title> Greetings, {{.Name}} </title>
</head>
<body>
    Greetings, {{.Name}}.<br>
    We know you are a {{.nationality}}!
</body>
```

Note:

1. Make sure that the .html files are in the correct directory.
2. When `http://localhost:8080/` can be visited after starting the server.
3. As it can be seen after submitting the form, the *unexported* nationality field of the struct could not be parsed by the template package, as expected.

Serving content using ServeMux

A simple static file server would look like this:

```

package main

import (
    "net/http"
)

func main() {
    muxer := http.NewServeMux()
    fileServerCss := http.FileServer(http.Dir("src/css"))
    fileServerJs := http.FileServer(http.Dir("src/js"))
    fileServerHtml := http.FileServer(http.Dir("content"))
    muxer.Handle("/", fileServerHtml)
    muxer.Handle("/css", fileServerCss)
    muxer.Handle("/js", fileServerJs)
    http.ListenAndServe(":8080", muxer)
}

```

Handling http method, accessing query strings & request body

Here are a simple example of some common tasks related to developing an API, differentiating between the HTTP Method of the request, accessing query string values and accessing the request body.

Resources

- [http.Handler interface](#)
- [http.ResponseWriter](#)
- [http.Request](#)
- [Available Method and Status constants](#)

```

package main

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

type customHandler struct{}

// ServeHTTP implements the http.Handler interface in the net/http package
func (h customHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {

    // ParseForm will parse query string values and make r.Form available
    r.ParseForm()

    // r.Form is map of query string parameters
    // its' type is url.Values, which in turn is a map[string][]string
    queryMap := r.Form

    switch r.Method {
    case http.MethodGet:
        // Handle GET requests
        w.WriteHeader(http.StatusOK)
        w.Write([]byte(fmt.Sprintf("Query string values: %s", queryMap)))
        return
    }
}

```

```

case http.MethodPost:
    // Handle POST requests
    body, err := ioutil.ReadAll(r.Body)
    if err != nil {
        // Error occurred while parsing request body
        w.WriteHeader(http.StatusBadRequest)
        return
    }
    w.WriteHeader(http.StatusOK)
    w.Write([]byte(fmt.Sprintf("Query string values: %s\nBody posted: %s", queryMap,
body)))
    return
}

// Other HTTP methods (eg PUT, PATCH, etc) are not handled by the above
// so inform the client with appropriate status code
w.WriteHeader(http.StatusMethodNotAllowed)
}

func main() {
    // All URLs will be handled by this function
    // http.Handle, similarly to http.HandleFunc
    // uses the DefaultServeMux
    http.Handle("/", customHandler{})

    // Continue to process new requests until an error occurs
    log.Fatal(http.ListenAndServe(":8080", nil))
}

```

Sample curl output:

```

$ curl -i 'localhost:8080?city=Seattle&state=WA' -H 'Content-Type: text/plain' -X GET
HTTP/1.1 200 OK
Date: Fri, 02 Sep 2016 16:36:24 GMT
Content-Length: 51
Content-Type: text/plain; charset=utf-8

Query string values: map[city:[Seattle] state:[WA]]%

$ curl -i 'localhost:8080?city=Seattle&state=WA' -H 'Content-Type: text/plain' -X POST -d
"some post data"
HTTP/1.1 200 OK
Date: Fri, 02 Sep 2016 16:36:35 GMT
Content-Length: 79
Content-Type: text/plain; charset=utf-8

Query string values: map[city:[Seattle] state:[WA]]
Body posted: some post data%

$ curl -i 'localhost:8080?city=Seattle&state=WA' -H 'Content-Type: text/plain' -X PUT
HTTP/1.1 405 Method Not Allowed
Date: Fri, 02 Sep 2016 16:36:41 GMT
Content-Length: 0
Content-Type: text/plain; charset=utf-8

```

Read HTTP Server online: <https://riptutorial.com/go/topic/756/http-server>

Chapter 29: Images

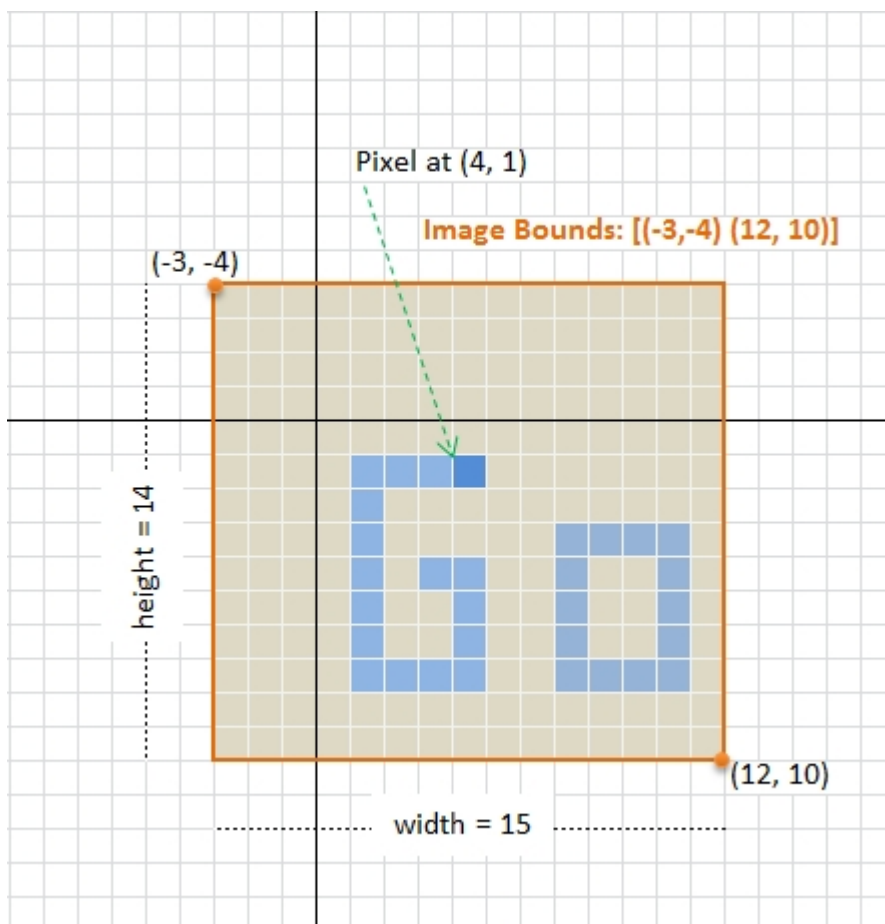
Introduction

The `image` package provides basic functionalities for working with 2-D image. This topic describes several basic operations when working with image such as reading and writing a particular image format, cropping, accessing and modifying *pixel*, color conversion, resizing and basic image filtering.

Examples

Basic concepts

An image represents a rectangular grid of picture elements (*pixel*). In the `image` package, the pixel is represented as one of the color defined in `image/color` package. The 2-D geometry of the image is represented as `image.Rectangle`, while `image.Point` denotes a position on the grid.



The above figure illustrates the basic concepts of an image in the package. An image of size 15x14 pixels has a rectangular *bounds* started at *upper left* corner (e.g. co-ordinate (-3, -4) in the above figure), and its axes increase right and down to *lower right* corner (e.g. co-ordinate (12, 10) in the figure). Note that the bounds **do not necessarily start from or contain point (0,0)**.

Image related *type*

In Go, an image always implement the following `image.Image` interface

```
type Image interface {
    // ColorModel returns the Image's color model.
    ColorModel() color.Model
    // Bounds returns the domain for which At can return non-zero color.
    // The bounds do not necessarily contain the point (0, 0).
    Bounds() Rectangle
    // At returns the color of the pixel at (x, y).
    // At(Bounds().Min.X, Bounds().Min.Y) returns the upper-left pixel of the grid.
    // At(Bounds().Max.X-1, Bounds().Max.Y-1) returns the lower-right one.
    At(x, y int) color.Color
}
```

in which the `color.Color` interface is defined as

```
type Color interface {
    // RGBA returns the alpha-premultiplied red, green, blue and alpha values
    // for the color. Each value ranges within [0, 0xffff], but is represented
    // by a uint32 so that multiplying by a blend factor up to 0xffff will not
    // overflow.
    //
    // An alpha-premultiplied color component c has been scaled by alpha (a),
    // so has valid values 0 <= c <= a.
    RGBA() (r, g, b, a uint32)
}
```

and `color.Model` is an interface declared as

```
type Model interface {
    Convert(c Color) Color
}
```

Accessing image dimension and pixel

Suppose we have an image stored as variable `img`, then we can obtain the dimension and image pixel by:

```
// Image bounds and dimension
b := img.Bounds()
width, height := b.Dx(), b.Dy()
// do something with dimension ...

// Corner co-ordinates
top := b.Min.Y
left := b.Min.X
bottom := b.Max.Y
right := b.Max.X

// Accessing pixel. The (x,y) position must be
// started from (left, top) position not (0,0)
```

```

for y := top; y < bottom; y++ {
    for x := left; x < right; x++ {
        cl := img.At(x, y)
        r, g, b, a := cl.RGBA()
        // do something with r,g,b,a color component
    }
}

```

Note that in the package, the value of each `R,G,B,A` component is between 0-65535 (`0x0000 - 0xffff`), **not** 0-255.

Loading and saving image

In memory, an image can be seen as a matrix of pixel (color). However, when an image being stored in a permanent storage, it may be stored as is (RAW format), [Bitmap](#) or other image formats with particular compression algorithm for saving storage space, e.g. PNG, JPEG, GIF, etc. When loading an image with particular format, the image must be *decoded* to `image.Image` with corresponding algorithm. An [image.Decode](#) function declared as

```

func Decode(r io.Reader) (Image, string, error)

```

is provided for this particular usage. In order to be able to handle various image formats, prior to calling the `image.Decode` function, the decoder must be registered through [image.RegisterFormat](#) function defined as

```

func RegisterFormat(name, magic string,
    decode func(io.Reader) (Image, error), decodeConfig func(io.Reader) (Config, error))

```

Currently, the image package supports three file formats: [JPEG](#), [GIF](#) and [PNG](#). To register a decoder, add the following

```

import _ "image/jpeg" //register JPEG decoder

```

to the application's `main` package. Somewhere in your code (not necessary in `main` package), to load a JPEG image, use the following snippets:

```

f, err := os.Open("inputimage.jpg")
if err != nil {
    // Handle error
}
defer f.Close()

img, fmtName, err := image.Decode(f)
if err != nil {
    // Handle error
}

// `fmtName` contains the name used during format registration
// Work with `img` ...

```

Save to PNG

To save an image into particular format, the corresponding *encoder* must be imported explicitly, i.e.

```
import "image/png"    //needed to use `png` encoder
```

then an image can be saved with the following snippets:

```
f, err := os.Create("outimage.png")
if err != nil {
    // Handle error
}
defer f.Close()

// Encode to `PNG` with `DefaultCompression` level
// then save to file
err = png.Encode(f, img)
if err != nil {
    // Handle error
}
```

If you want to specify compression level other than `DefaultCompression` level, create an *encoder*, e.g.

```
enc := png.Encoder{
    CompressionLevel: png.BestSpeed,
}
err := enc.Encode(f, img)
```

Save to JPEG

To save to `jpeg` format, use the following:

```
import "image/jpeg"

// Somewhere in the same package
f, err := os.Create("outimage.jpg")
if err != nil {
    // Handle error
}
defer f.Close()

// Specify the quality, between 0-100
// Higher is better
opt := jpeg.Options{
    Quality: 90,
}
err = jpeg.Encode(f, img, &opt)
if err != nil {
    // Handle error
}
```

Save to GIF

To save the image to GIF file, use the following snippets.

```
import "image/gif"

// Somewhere in the same package
f, err := os.Create("outimage.gif")
if err != nil {
    // Handle error
}
defer f.Close()

opt := gif.Options {
    NumColors: 256,
    // Add more parameters as needed
}

err = gif.Encode(f, img, &opt)
if err != nil {
    // Handle error
}
```

Cropping image

Most of image type in [image](#) package having `SubImage(r Rectangle) Image` method, except `image.Uniform`. Based on this fact, We can implement a function to crop an arbitrary image as follows

```
func CropImage(img image.Image, cropRect image.Rectangle) (cropImg image.Image, newImg bool) {
    //Interface for asserting whether `img`
    //implements SubImage or not.
    //This can be defined globally.
    type CropableImage interface {
        image.Image
        SubImage(r image.Rectangle) image.Image
    }

    if p, ok := img.(CropableImage); ok {
        // Call SubImage. This should be fast,
        // since SubImage (usually) shares underlying pixel.
        cropImg = p.SubImage(cropRect)
    } else if cropRect = cropRect.Intersect(img.Bounds()); !cropRect.Empty() {
        // If `img` does not implement `SubImage`,
        // copy (and silently convert) the image portion to RGBA image.
        rgbaImg := image.NewRGBA(cropRect)
        for y := cropRect.Min.Y; y < cropRect.Max.Y; y++ {
            for x := cropRect.Min.X; x < cropRect.Max.X; x++ {
                rgbaImg.Set(x, y, img.At(x, y))
            }
        }
        cropImg = rgbaImg
        newImg = true
    } else {
        // Return an empty RGBA image
        cropImg = &image.RGBA{}
    }
}
```

```

        newImg = true
    }

    return cropImg, newImg
}

```

Note that the cropped image may shared its underlying pixels with the original image. If this is the case, any modification to the cropped image will affect the original image.

Convert color image to grayscale

Some digital image processing algorithm such as edge detection, information carried by the image intensity (i.e. grayscale value) is sufficient. Using color information (R, G, B channel) may provides slightly better result, but the algorithm complexity will be increased. Thus, in this case, we need to convert the color image to grayscale image prior to applying such algorithm.

The following code is an example of converting arbitrary image to 8-bit grayscale image. The image is retrieved from remote location using `net/http` package, converted to grayscale, and finally saved as PNG image.

```

package main

import (
    "image"
    "log"
    "net/http"
    "os"

    _ "image/jpeg"
    "image/png"
)

func main() {
    // Load image from remote through http
    // The Go gopher was designed by Renee French. (http://reneefrench.blogspot.com/)
    // Images are available under the Creative Commons 3.0 Attributions license.
    resp, err := http.Get("http://golang.org/doc/gopher/fiveyears.jpg")
    if err != nil {
        // handle error
        log.Fatal(err)
    }
    defer resp.Body.Close()

    // Decode image to JPEG
    img, _, err := image.Decode(resp.Body)
    if err != nil {
        // handle error
        log.Fatal(err)
    }
    log.Printf("Image type: %T", img)

    // Converting image to grayscale
    grayImg := image.NewGray(img.Bounds())
    for y := img.Bounds().Min.Y; y < img.Bounds().Max.Y; y++ {
        for x := img.Bounds().Min.X; x < img.Bounds().Max.X; x++ {
            grayImg.Set(x, y, img.At(x, y))
        }
    }
}

```

```

    }
}

// Working with grayscale image, e.g. convert to png
f, err := os.Create("fiveyears_gray.png")
if err != nil {
    // handle error
    log.Fatal(err)
}
defer f.Close()

if err := png.Encode(f, grayImg); err != nil {
    log.Fatal(err)
}
}

```

Color conversion occurs when assigning pixel through `Set(x, y int, c color.Color)` which is implemented in [image.go](#) as

```

func (p *Gray) Set(x, y int, c color.Color) {
    if !(Point{x, y}.In(p.Rect)) {
        return
    }

    i := p.PixOffset(x, y)
    p.Pix[i] = color.GrayModel.Convert(c).(color.Gray).Y
}

```

in which, `color.GrayModel` is defined in [color.go](#) as

```

func grayModel(c Color) Color {
    if _, ok := c.(Gray); ok {
        return c
    }
    r, g, b, _ := c.RGBA()

    // These coefficients (the fractions 0.299, 0.587 and 0.114) are the same
    // as those given by the JFIF specification and used by func RGBToYCbCr in
    // ycbcr.go.
    //
    // Note that 19595 + 38470 + 7471 equals 65536.
    //
    // The 24 is 16 + 8. The 16 is the same as used in RGBToYCbCr. The 8 is
    // because the return value is 8 bit color, not 16 bit color.
    y := (19595*r + 38470*g + 7471*b + 1<<15) >> 24

    return Gray{uint8(y)}
}

```

Based on the above facts, the intensity Y is calculated with the following formula:

$$\text{Luminance: } Y = 0.299R + 0.587G + 0.114B$$

If we want to apply different [formula/algorithms](#) to convert a color into an intensity, e.g.

$$\text{Mean: } Y = (R + G + B) / 3$$

Luma: $Y = 0.2126R + 0.7152G + 0.0722B$

Luster: $Y = (\min(R, G, B) + \max(R, G, B))/2$

then, the following snippets can be used.

```
// Converting image to grayscale
grayImg := image.NewGray(img.Bounds())
for y := img.Bounds().Min.Y; y < img.Bounds().Max.Y; y++ {
    for x := img.Bounds().Min.X; x < img.Bounds().Max.X; x++ {
        R, G, B, _ := img.At(x, y).RGBA()
        //Luma: Y = 0.2126*R + 0.7152*G + 0.0722*B
        Y := (0.2126*float64(R) + 0.7152*float64(G) + 0.0722*float64(B)) * (255.0 / 65535)
        grayPix := color.Gray{uint8(Y)}
        grayImg.Set(x, y, grayPix)
    }
}
```

The above calculation is done by floating point multiplication, and certainly is not the most efficient one, but it's enough for demonstrating the idea. The other point is, when calling `Set(x, y int, c color.Color)` with `color.Gray` as third argument, the color model will not perform color conversion as can be seen in the previous `grayModel` function.

Read Images online: <https://riptutorial.com/go/topic/10557/images>

Chapter 30: Inline Expansion

Remarks

Inline expansion is a common optimization in compiled code that prioritized performance over binary size. It lets the compiler replace a function call with the actual body of the function; effectively copy/pasting code from one place to another at compile time. Since the call site is expanded to just contain the machine instructions that the compiler generated for the function, we don't have to perform a CALL or PUSH (the x86 equivalent of a GOTO statement or a stack frame push) or their equivalent on other architectures.

The inliner makes decisions about whether or not to inline a function based on a number of heuristics, but in general Go inlines by default. Because the inliner gets rid of function calls, it effectively gets to decide where the scheduler is allowed to preempt a goroutine.

Function calls will not be inlined if any of the following are true (there are many other reasons too, this list is incomplete):

- Functions are variadic (eg. they have `... args`)
- Functions have a "max hairyness" greater than the budget (they recurse too much or can't be analyzed for some other reason)
- They contain `panic`, `recover`, or `defer`

Examples

Disabling inline expansion

Inline expansion can be disabled with the `go:noinline` pragma. For example, if we build the following simple program:

```
package main

func printhello() {
    println("Hello")
}

func main() {
    printhello()
}
```

we get output that looks like this (trimmed for readability):

```
$ go version
go version go1.6.2 linux/amd64
$ go build main.go
$ ./main
Hello
$ go tool objdump main
```



```

TEXT main.main(SB) /home/sam/main.go
    main.go:7      0x401000      64488b0c25f8ffffff      FS MOVQ FS:0xffffffff8, CX
    main.go:7      0x401009      483b6110      CMPQ 0x10(CX), SP
    main.go:7      0x40100d      7631      JBE 0x401040
    main.go:7      0x40100f      4883ec10      SUBQ $0x10, SP
    main.go:8      0x401013      e8281f0200      CALL runtime.printlock(SB)
    main.go:8      0x401018      488d1d01130700      LEAQ 0x71301(IP), BX
    main.go:8      0x40101f      48891c24      MOVQ BX, 0(SP)
    main.go:8      0x401023      48c744240805000000      MOVQ $0x5, 0x8(SP)
    main.go:8      0x40102c      e81f290200      CALL runtime.printstring(SB)
    main.go:8      0x401031      e89a210200      CALL runtime.println(SB)
    main.go:8      0x401036      e8851f0200      CALL runtime.printunlock(SB)
    main.go:9      0x40103b      4883c410      ADDQ $0x10, SP
    main.go:9      0x40103f      c3      RET
    main.go:7      0x401040      e87b9f0400      CALL
runtime.morestack_noctxt(SB)
    main.go:7      0x401045      ebb9      JMP main.main(SB)
    main.go:7      0x401047      cc      INT $0x3
    main.go:7      0x401048      cc      INT $0x3
    main.go:7      0x401049      cc      INT $0x3
    main.go:7      0x40104a      cc      INT $0x3
    main.go:7      0x40104b      cc      INT $0x3
    main.go:7      0x40104c      cc      INT $0x3
    main.go:7      0x40104d      cc      INT $0x3
    main.go:7      0x40104e      cc      INT $0x3
    main.go:7      0x40104f      cc      INT $0x3
...

```

note that there is no `CALL` to `printhello`. However, if we then build the program with the pragma in place:

```

package main

//go:noinline
func printhello() {
    println("Hello")
}

func main() {
    printhello()
}

```

The output contains the `printhello` function and a `CALL main.printhello`:

```

$ go version
go version go1.6.2 linux/amd64
$ go build main.go
$ ./main
Hello
$ go tool objdump main
TEXT main.printhello(SB) /home/sam/main.go
    main.go:4      0x401000      64488b0c25f8ffffff      FS MOVQ FS:0xffffffff8, CX
    main.go:4      0x401009      483b6110      CMPQ 0x10(CX), SP
    main.go:4      0x40100d      7631      JBE 0x401040
    main.go:4      0x40100f      4883ec10      SUBQ $0x10, SP
    main.go:5      0x401013      e8481f0200      CALL runtime.printlock(SB)
    main.go:5      0x401018      488d1d01130700      LEAQ 0x71301(IP), BX
    main.go:5      0x40101f      48891c24      MOVQ BX, 0(SP)

```

```

main.go:5      0x401023      48c744240805000000      MOVQ $0x5, 0x8(SP)
main.go:5      0x40102c      e83f290200      CALL runtime.printstring(SB)
main.go:5      0x401031      e8ba210200      CALL runtime.printnl(SB)
main.go:5      0x401036      e8a51f0200      CALL runtime.printunlock(SB)
main.go:6      0x40103b      4883c410      ADDQ $0x10, SP
main.go:6      0x40103f      c3      RET
main.go:4      0x401040      e89b9f0400      CALL
runtime.morestack_noctxt(SB)
main.go:4      0x401045      ebb9      JMP main.printhello(SB)
main.go:4      0x401047      cc      INT $0x3
main.go:4      0x401048      cc      INT $0x3
main.go:4      0x401049      cc      INT $0x3
main.go:4      0x40104a      cc      INT $0x3
main.go:4      0x40104b      cc      INT $0x3
main.go:4      0x40104c      cc      INT $0x3
main.go:4      0x40104d      cc      INT $0x3
main.go:4      0x40104e      cc      INT $0x3
main.go:4      0x40104f      cc      INT $0x3

TEXT main.main(SB) /home/sam/main.go
main.go:8      0x401050      64488b0c25f8ffffff      FS MOVQ FS:0xffffffff8, CX
main.go:8      0x401059      483b6110      CMPQ 0x10(CX), SP
main.go:8      0x40105d      7606      JBE 0x401065
main.go:9      0x40105f      e89cffffff      CALL main.printhello(SB)
main.go:10     0x401064      c3      RET
main.go:8      0x401065      e8769f0400      CALL
runtime.morestack_noctxt(SB)
main.go:8      0x40106a      ebe4      JMP main.main(SB)
main.go:8      0x40106c      cc      INT $0x3
main.go:8      0x40106d      cc      INT $0x3
main.go:8      0x40106e      cc      INT $0x3
main.go:8      0x40106f      cc      INT $0x3

```

...

Read Inline Expansion online: <https://riptutorial.com/go/topic/2718/inline-expansion>

Chapter 31: Installation

Examples

Install in Linux or Ubuntu

```
$ sudo apt-get update
$ sudo apt-get install -y build-essential git curl wget
$ wget https://storage.googleapis.com/golang/go<versions>.gz
```

You can find the version lists [here](#).

```
# To install go1.7 use
$ wget https://storage.googleapis.com/golang/go1.7.linux-amd64.tar.gz

# Untar the file
$ sudo tar -C /usr/local -xzf go1.7.linux-amd64.tar.gz
$ sudo chown -R $USER:$USER /usr/local/go
$ rm go1.5.4.linux-amd64.tar.gz
```

Update \$GOPATH

```
$ mkdir $HOME/go
```

Add following two lines at the end of the ~/.bashrc file

```
export GOPATH=$HOME/go
export PATH=$GOPATH/bin:/usr/local/go/bin:$PATH
```

```
$ nano ~/.bashrc
export GOPATH=$HOME/go
export PATH=$GOPATH/bin:/usr/local/go/bin:$PATH

$ source ~/.bashrc
```

Now are set to go, test your go version using:

```
$ go version
go version go<version> linux/amd64
```

Read Installation online: <https://riptutorial.com/go/topic/5776/installation>

Chapter 32: Installation

Remarks

Downloading Go

Visit the [Downloads List](#) and find the right archive for your operating system. The names of these downloads can be a bit cryptic to new users.

The names are in the format `go[version].[operating system]-[architecture].[archive]`

For the version, you want to choose the newest available. These should be the first options you see.

For the operating system, this is fairly self-explanatory except for Mac users, where the operating system is named "darwin". This is named after the [open-source part of the operating system used by Mac computers](#).

If you are running a 64-bit machine (which is the most common in modern computers), the "architecture" part of the file name should be "amd64". For 32-bit machines, it will be "386". If you're on an ARM device like a Raspberry Pi, you'll want "armv6l".

For the "archive" part, Mac and Windows users have two options because Go provides installers for those platforms. For Mac, you probably want "pkg". For Windows, you probably want "msi".

So, for instance, if I'm on a 64-bit Windows machine and I want to download Go 1.6.3, the download I want will be named:

```
go1.6.3.windows-amd64.msi
```

Extracting the download files

Now that we have a Go archive downloaded, we need to extract it somewhere.

Mac and Windows

Since installers are provided for these platforms, installation is easy. Just run the installer and accept the defaults.

Linux

There is no installer for Linux, so some more work is required. You should have downloaded a file with the suffix ".tar.gz". This is an archive file, similar to a ".zip" file. We need to extract it. We will be extracting the Go files to `/usr/local` because it is the recommended location.

Open up a terminal and change directories to the place where you downloaded the archive. This is probably in `Downloads`. If not, replace the directory in the following command appropriately.

```
cd Downloads
```

Now, run the following to extract the archive into `/usr/local`, replacing `[filename]` with the name of the file you downloaded.

```
tar -C /usr/local -xzf [filename].tar.gz
```

Setting Environment Variables

There's one more step to go before you're ready to start developing. We need to set environment variables, which is information that users can change to give programs a better idea of the user's setup.

Windows

You need to set the `GOPATH`, which is the folder that you will be doing Go work in.

You can set environment variables through the "Environment Variables" button on the "Advanced" tab of the "System" control panel. Some versions of Windows provide this control panel through the "Advanced System Settings" option inside the "System" control panel.

The name of your new environment variable should be "GOPATH". The value should be the full path to a directory you'll be developing Go code in. A folder called "go" in your user directory is a good choice.

Mac

You need to set the `GOPATH`, which is the folder that you will be doing Go work in.

Edit a text file named `".bash_profile"`, which should be in your user directory, and add the following new line to the end, replacing `[work area]` with a full path to a directory you would like to do Go work in. If `".bash_profile"` does not exist, create it. A folder called "go" in your user directory is a good choice.

```
export GOPATH=[work area]
```

Linux

Because Linux doesn't have an installer, it requires a bit more work. We need to show the terminal where the Go compiler and other tools are, and we need to set the `GOPATH`, which is a folder that you will be doing Go work in.

Edit a text file named `".profile"`, which should be in your user directory, and add the following line to the end, replacing `[work area]` with a full path to a directory you would like to do Go work in. If

".profile" does not exist, create it. A folder called "go" in your user directory is a good choice.

Then, on another new line, add the following to your ".profile" file.

```
export PATH=$PATH:/usr/local/go/bin
```

Finished!

If the Go tools are still not available to you in the terminal, try closing that window and opening a fresh terminal window.

Examples

Example .profile or .bash_profile

```
# This is an example of a .profile or .bash_profile for Linux and Mac systems
export GOPATH=/home/user/go
export PATH=$PATH:/usr/local/go/bin
```

Read Installation online: <https://riptutorial.com/go/topic/6213/installation>

Chapter 33: Interfaces

Remarks

Interfaces in Go are just fixed method sets. A type *implicitly* implements an interface if its method set is a superset of the interface. *There is no declaration of intent.*

Examples

Simple interface

In Go, an interface is just a set of methods. We use an interface to specify a behavior of a given object.

```
type Painter interface {  
    Paint()  
}
```

The implementing type **need not** declare that it is implementing the interface. It is enough to define methods of the same signature.

```
type Rembrandt struct{}  
  
func (r Rembrandt) Paint() {  
    // use a lot of canvas here  
}
```

Now we can use the structure as an interface.

```
var p Painter  
p = Rembrandt{}
```

An interface can be satisfied (or implemented) by an arbitrary number of types. Also a type can implement an arbitrary number of interfaces.

```
type Singer interface {  
    Sing()  
}  
  
type Writer interface {  
    Write()  
}  
  
type Human struct{}  
  
func (h *Human) Sing() {  
    fmt.Println("singing")  
}
```

```
func (h *Human) Write() {
    fmt.Println("writing")
}

type OnlySinger struct{}
func (o *OnlySinger) Sing() {
    fmt.Println("singing")
}
```

Here, The `Human` struct satisfy both the `Singer` and `Writer` interface, but the `OnlySinger` struct only satisfy `Singer` interface.

Empty Interface

There is an empty interface type, that contains no methods. We declare it as `interface{}`. This contains no methods so every `type` satisfies it. Hence empty interface can contain any type value.

```
var a interface{}
var i int = 5
s := "Hello world"

type StructType struct {
    i, j int
    k string
}

// all are valid statements
a = i
a = s
a = &StructType{1, 2, "hello"}
```

The most common use case for interfaces is to ensure that a variable supports one or more behaviours. By contrast, the primary use case for the empty interface is to define a variable which can hold any value, regardless of its concrete type.

To get these values back as their original types we just need to do

```
i = a.(int)
s = a.(string)
m := a.(*StructType)
```

or

```
i, ok := a.(int)
s, ok := a.(string)
m, ok := a.(*StructType)
```

`ok` indicates if the `interface a` is convertible to given type. If it is not possible to cast `ok` will be `false`.

Interface Values

If you declare a variable of an interface, it may store any value type that implements the methods declared by the interface!

If we declare `h` of interface `Singer`, it may store a value of type `Human` or `OnlySinger`. This is because of the fact that they all implement methods specified by the `Singer` interface.

```
var h Singer
h = &human{}

h.Sing()
```

Determining underlying type from interface

In go it can sometimes be useful to know which underlying type you have been passed. This can be done with a type switch. This assumes we have two structs:

```
type Rembrandt struct{}

func (r Rembrandt) Paint() {}

type Picasso struct{}

func (r Picasso) Paint() {}
```

That implement the `Painter` interface:

```
type Painter interface {
    Paint()
}
```

Then we can use this switch to determine the underlying type:

```
func WhichPainter(painter Painter) {
    switch painter.(type) {
    case Rembrandt:
        fmt.Println("The underlying type is Rembrandt")
    case Picasso:
        fmt.Println("The underlying type is Picasso")
    default:
        fmt.Println("Unknown type")
    }
}
```

Compile-time check if a type satisfies an interface

Interfaces and implementations (types that implement an interface) are "detached". So it is a rightful question how to check at compile-time if a type implements an interface.

One way to ask the compiler to check that the type `T` implements the interface `I` is by attempting an assignment using the zero value for `T` or pointer to `T`, as appropriate. And we may choose to

assign to the [blank identifier](#) to avoid unnecessary garbage:

```
type T struct{}

var _ I = T{}           // Verify that T implements I.
var _ I = (*T)(nil)    // Verify that *T implements I.
```

If `T` or `*T` does not implement `I`, it will be a compile time error.

This question also appears in the official FAQ: [How can I guarantee my type satisfies an interface?](#)

Type switch

Type switches can also be used to get a variable that matches the type of the case:

```
func convint(v interface{}) (int,error) {
    switch u := v.(type) {
    case int:
        return u, nil
    case float64:
        return int(u), nil
    case string:
        return strconv(u)
    default:
        return 0, errors.New("Unsupported type")
    }
}
```

Type Assertion

You can access the real data type of interface with Type Assertion.

```
interfaceVariable.(DataType)
```

Example of struct `MyType` which implement interface `Subber`:

```
package main

import (
    "fmt"
)

type Subber interface {
    Sub(a, b int) int
}

type MyType struct {
    Msg string
}

//Implement method Sub(a,b int) int
func (m *MyType) Sub(a, b int) int {
    m.Msg = "SUB!!!"
}
```

```

    return a - b;
}

func main() {
    var interfaceVar Subber = &MyType{}
    fmt.Println(interfaceVar.Sub(6,5))
    fmt.Println(interfaceVar.(*MyType).Msg)
}

```

Without `.(*MyType)` we wouldn't be able to access `Msg` Field. If we try `interfaceVar.Msg` it will show compile error:

```
interfaceVar.Msg undefined (type Subber has no field or method Msg)
```

Go Interfaces from a Mathematical Aspect

In mathematics, especially *Set Theory*, we have a collection of things which is called *set* and we name those things as *elements*. We show a set with its name like A, B, C, ... or explicitly with putting its member on brace notation: {a, b, c, d, e}. Suppose we have an arbitrary element x and a set Z, The key question is: "How we can understand that x is member of Z or not?".

Mathematician answer to this question with a concept: **Characteristic Property** of a set.

Characteristic Property of a set is an expression which describe set completely. For example we have set of *Natural Numbers* which is {0, 1, 2, 3, 4, 5, ...}. We can describe this set with this

expression: $\{a_n \mid a_0 = 0, a_n = a_{n-1} + 1\}$. In last expression $a_0 = 0$, $a_n = a_{n-1} + 1$ is the characteristic property of set of natural numbers. **If we have this expression, we can build this set**

completely. Let describe the set of *even numbers* in this manner. We know that this set is made by this numbers: {0, 2, 4, 6, 8, 10, ...}. With a glance we understand that all of this numbers are also a *natural number*, in other words *if we add some extra conditions to characteristic property of natural numbers, we can build a new expression which describe this set*. So we can describe with this expression: $\{n \mid n \text{ is a member of natural numbers and the remainder of } n \text{ on } 2 \text{ is zero}\}$. Now we can create a filter which get the characteristic property of a set and filter some desired elements to return elements of our set. For example if we have a natural number filter, both of natural numbers and even numbers can pass this filter, but if we have a even number filter, then some elements like 3 and 137871 can't pass the filter.

Definition of interface in Go is like defining the characteristic property and mechanism of using interface as an argument of a function is like a filter which detect the element is a member of our desired set or not. Lets describe this aspect with code:

```

type Number interface {
    IsNumber() bool // the implementation filter "meysam" from 3.14, 2 and 3
}

type NaturalNumber interface {
    Number
    IsNaturalNumber() bool // the implementation filter 3.14 from 2 and 3
}

type EvenNumber interface {
    NaturalNumber
}

```

```
IsEvenNumber() bool // the implementation filter 3 from 2
}
```

The characteristic property of `Number` is all structures that have `IsNumber` method, for `NaturalNumber` is all ones that have `IsNumber` and `IsNaturalNumber` methods and finally for `EvenNumber` is all types which have `IsNumber`, `IsNaturalNumber` and `IsEvenNumber` methods. Thanks to this interpretation of interface, easily we can understand that since `interface{}` doesn't have any characteristic property, accept all types (because it doesn't have any filter for distinguishing between values).

Read Interfaces online: <https://riptutorial.com/go/topic/1221/interfaces>

Chapter 34: Iota

Introduction

Iota provides a way of declaring numeric constants from a starting value that grows monotonically. Iota can be used to declare bitmasks which are often used in system and network programming and other lists of constants with related values.

Remarks

The `iota` identifier is used to assign values to lists of constants. When `iota` is used in a list it starts with a value of zero, and increments by one for each value in the list of constants and is reset on each `const` keyword. Unlike the enumerations of other languages, `iota` can be used in expressions (eg. `iota + 1`) which allows for greater flexibility.

Examples

Simple use of iota

To create a list of constants - assign `iota` value to each element:

```
const (  
    a = iota // a = 0  
    b = iota // b = 1  
    c = iota // c = 2  
)
```

To create a list of constants in a shortened way - assign `iota` value to the first element:

```
const (  
    a = iota // a = 0  
    b          // b = 1  
    c          // c = 2  
)
```

Using iota in an expression

`iota` can be used in expressions, so it can also be used to assign values other than simple incrementing integers starting from zero. To create constants for SI units, use this example from [Effective Go](#):

```
type ByteSize float64  
  
const (  
    _ = iota // ignore first value by assigning to blank identifier  
    KB ByteSize = 1 << (10 * iota)  
    MB
```

```

    GB
    TB
    PB
    EB
    ZB
    YB
)

```

Skipping values

The value of `iota` is still incremented for every entry in a constant list even if `iota` is not used:

```

const ( // iota is reset to 0
    a = 1 << iota // a == 1
    b = 1 << iota // b == 2
    c = 3          // c == 3 (iota is not used but still incremented)
    d = 1 << iota // d == 8
)

```

it will also be incremented even if no constant is created at all, meaning the empty identifier can be used to skip values entirely:

```

const (
    a = iota // a = 0
    _        // iota is incremented
    b        // b = 2
)

```

The first code block was taken from the [Go Spec](#) (CC-BY 3.0).

Use of `iota` in an expression list

Because `iota` is incremented after each `ConstSpec`, values within the same expression list will have the same value for `iota`:

```

const (
    bit0, mask0 = 1 << iota, 1<<iota - 1 // bit0 == 1, mask0 == 0
    bit1, mask1          // bit1 == 2, mask1 == 1
    _ , _                // skips iota == 2
    bit3, mask3          // bit3 == 8, mask3 == 7
)

```

This example was taken from the [Go Spec](#) (CC-BY 3.0).

Use of `iota` in a bitmask

`iota` can be very useful when creating a bitmask. For instance, to represent the state of a network connection which may be secure, authenticated, and/or ready, we might create a bitmask like the following:

```

const (

```

```
Secure = 1 << iota // 0b001
Authn   // 0b010
Ready   // 0b100
)

ConnState := Secure|Authn // 0b011: Connection is secure and authenticated, but not yet Ready
```

Use of iota in const

This is an enumeration for const creation. Go compiler starts iota from 0 and increments by one for each following constant. The value is determined at compile time rather than run time. Because of this we can't apply iota to expressions which are evaluated at run time.

Program to use iota in const

```
package main

import "fmt"

const (
    Low = 5 * iota
    Medium
    High
)

func main() {
    // Use our iota constants.
    fmt.Println(Low)
    fmt.Println(Medium)
    fmt.Println(High)
}
```

Try it in [Go Playground](#)

Read iota online: <https://riptutorial.com/go/topic/2865/iota>

Chapter 35: JSON

Syntax

- `func Marshal(v interface{}) ([]byte, error)`
- `func Unmarshal(data []byte, v interface{}) error`

Remarks

The package `"encoding/json"` Package `json` implements encoding and decoding of JSON objects in Go.

Types in JSON along with their corresponding concrete types in Go are:

JSON Type	Go Concrete Type
boolean	bool
numbers	float64 or int
string	string
null	nil

Examples

Basic JSON Encoding

`json.Marshal` from the package `"encoding/json"` encodes a value to JSON.

The parameter is the value to encode. The returned values are an array of bytes representing the JSON-encoded input (on success), and an error (on failure).

```
decodedValue := []string{"foo", "bar"}

// encode the value
data, err := json.Marshal(decodedValue)

// check if the encoding is successful
if err != nil {
    panic(err)
}

// print out the JSON-encoded string
// remember that data is a []byte
fmt.Println(string(data))
// "["foo","bar"]"
```


Playground

Here's some basic examples of encoding for built-in data types:

```
var data []byte

data, _ = json.Marshal(1)
fmt.Println(string(data))
// 1

data, _ = json.Marshal("1")
fmt.Println(string(data))
// "1"

data, _ = json.Marshal(true)
fmt.Println(string(data))
// true

data, _ = json.Marshal(map[string]int{"London": 18, "Rome": 30})
fmt.Println(string(data))
// {"London":18,"Rome":30}
```

Playground

Encoding simple variables is helpful to understand how the JSON encoding works in Go. However, in the real world, you'll likely [encode more complex data stored in structs](#).

Basic JSON decoding

`json.Unmarshal` from the package `"encoding/json"` decodes a JSON value into the value pointed by the given variable.

The parameters are the value to decode in `[]bytes` and a variable to use as a storage for the deserialized value. The returned value is an error (on failure).

```
encodedValue := []byte(`{"London":18,"Rome":30}`)

// generic storage for the decoded JSON
var data map[string]interface{}

// decode the value into data
// notice that we must pass the pointer to data using &data
err := json.Unmarshal(encodedValue, &data)

// check if the decoding is successful
if err != nil {
    panic(err)
}

fmt.Println(data)
map[London:18 Rome:30]
```

Playground

Notice how in the example above we knew in advance both the type of the key and the value. But

this is not always the case. In fact, in most cases the JSON contains mixed value types.

```
encodedValue := []byte(`{"city":"Rome","temperature":30}`)

// generic storage for the decoded JSON
var data map[string]interface{}

// decode the value into data
if err := json.Unmarshal(encodedValue, &data); err != nil {
    panic(err)
}

// if you want to use a specific value type, we need to cast it
temp := data["temperature"].(float64)
fmt.Println(temp) // 30
city := data["city"].(string)
fmt.Println(city) // "Rome"
```

Playground

In the last example above we used a generic map to store the decoded value. We must use a `map[string]interface{}` because we know that the keys are strings, but we don't know the type of their values in advance.

This is a very simple approach, but it's also extremely limited. In the real world, you would generally [decode a JSON into a custom-defined struct type](#).

Decoding JSON data from a file

JSON data can also be read from files.

Let's assume we have a file called `data.json` with the following content:

```
[
  {
    "Name" : "John Doe",
    "Standard" : 4
  },
  {
    "Name" : "Peter Parker",
    "Standard" : 11
  },
  {
    "Name" : "Bilbo Baggins",
    "Standard" : 150
  }
]
```

The following example reads the file and decodes the content:

```
package main

import (
    "encoding/json"
    "fmt"
)
```

```

    "log"
    "os"
)

type Student struct {
    Name      string
    Standard  int `json:"Standard"`
}

func main() {
    // open the file pointer
    studentFile, err := os.Open("data.json")
    if err != nil {
        log.Fatal(err)
    }
    defer studentFile.Close()

    // create a new decoder
    var studentDecoder *json.Decoder = json.NewDecoder(studentFile)
    if err != nil {
        log.Fatal(err)
    }

    // initialize the storage for the decoded data
    var studentList []Student

    // decode the data
    err = studentDecoder.Decode(&studentList)
    if err != nil {
        log.Fatal(err)
    }

    for i, student := range studentList {
        fmt.Println("Student", i+1)
        fmt.Println("Student name:", student.Name)
        fmt.Println("Student standard:", student.Standard)
    }
}

```

The file `data.json` must be in the same directory of the Go executable program. Read [Go File I/O documentation](#) for more information on how to work with files in Go.

Using anonymous structs for decoding

The goal with using anonymous structs is to decode only the information we care about without littering our app with types that are used only in a single function.

```

jsonBlob := []byte(`
{
    "_total": 1,
    "_links": {
        "self":
"https://api.twitch.tv/kraken/channels/foo/subscriptions?direction=ASC&limit=25&offset=0",
        "next":
"https://api.twitch.tv/kraken/channels/foo/subscriptions?direction=ASC&limit=25&offset=25"
    },
    "subscriptions": [
        {

```

```

        "created_at": "2011-11-23T02:53:17Z",
        "_id": "abcdef000000000000000000000000000000000000000000000000000000000000",
        "_links": {
            "self": "https://api.twitch.tv/kraken/channels/foo/subscriptions/bar"
        },
        "user": {
            "display_name": "bar",
            "_id": 123456,
            "name": "bar",
            "staff": false,
            "created_at": "2011-06-16T18:23:11Z",
            "updated_at": "2014-10-23T02:20:51Z",
            "logo": null,
            "_links": {
                "self": "https://api.twitch.tv/kraken/users/bar"
            }
        }
    }
}
` )

var js struct {
    Total int `json:"_total"`
    Links struct {
        Next string `json:"next"`
    } `json:"_links"`
    Subs []struct {
        Created string `json:"created_at"`
        User struct {
            Name string `json:"name"`
            ID int `json:"_id"`
        } `json:"user"`
    } `json:"subscriptions"`
}

err := json.Unmarshal(jsonBlob, &js)
if err != nil {
    fmt.Println("error:", err)
}
fmt.Printf("%+v", js)

```

Output: {Total:1

Links:{Next:https://api.twitch.tv/kraken/channels/foo/subscriptions?direction=ASC&limit=25&offset=25}
 Subs:[{Created:2011-11-23T02:53:17Z User:{Name:bar ID:123456}}]}

Playground

For the general case see also:

<http://stackoverflow.com/documentation/go/994/json/4111/encoding-decoding-go-structs>

Configuring JSON struct fields

Consider the following example:

```

type Company struct {
    Name      string
    Location  string
}

```

```
}
```

Hide/Skip Certain Fields

To export `Revenue` and `Sales`, but hide them from encoding/decoding, use `json:"-"` or rename the variable to begin with a lowercase letter. Note that this prevents the variable from being visible outside the package.

```
type Company struct {
    Name      string `json:"name"`
    Location  string `json:"location"`
    Revenue   int    `json:"-"`
    sales     int
}
```

Ignore Empty Fields

To prevent `Location` from being included in the JSON when it is set to its zero value, add `,omitempty` to the `json` tag.

```
type Company struct {
    Name      string `json:"name"`
    Location  string `json:"location,omitempty"`
}
```

[Example in Playground](#)

Marshaling structs with private fields

As a good developer you have created following struct with both exported and unexported fields:

```
type MyStruct struct {
    uuid string
    Name string
}
```

Example in Playground: <https://play.golang.org/p/Zk94II2ANZ>

Now you want to `Marshal()` this struct into valid JSON for storage in something like etcd. However, since `uuid` is not exported, the `json.Marshal()` skips it. What to do? Use an anonymous struct and the `json.MarshalJSON()` interface! Here's an example:

```
type MyStruct struct {
    uuid string
    Name string
}

func (m MyStruct) MarshalJSON() ([]byte, error) {
    j, err := json.Marshal(struct {
        Uuid string
        Name string
    })
```

```

    }{
        Uuid: m.uuid,
        Name: m.Name,
    })
    if err != nil {
        return nil, err
    }
    return j, nil
}

```

Example in Playground: <https://play.golang.org/p/Bv2k9GgbzE>

Encoding/Decoding using Go structs

Let's assume we have the following `struct` that defines a `City` type:

```

type City struct {
    Name string
    Temperature int
}

```

We can encode/decode `City` values using the `encoding/json` package.

First of all, we need to use the Go metadata to tell the encoder the correspondence between the struct fields and the JSON keys.

```

type City struct {
    Name string `json:"name"`
    Temperature int `json:"temp"`
    // IMPORTANT: only exported fields will be encoded/decoded
    // Any field starting with a lower letter will be ignored
}

```

To keep this example simple, we'll declare an explicit correspondence between the fields and the keys. However, you can use several variants of the `json:` metadata [as explained in the docs](#).

IMPORTANT: Only `exported fields` (fields with capital name) will be serialized/deserialized. For example, if you name the field `temperature` it will be ignored even if you set the `json` metadata.

Encoding

To encode a `City` struct, use `json.Marshal` as in the basic example:

```

// data to encode
city := City{Name: "Rome", Temperature: 30}

// encode the data
bytes, err := json.Marshal(city)
if err != nil {
    panic(err)
}

fmt.Println(string(bytes))

```

```
// {"name":"Rome","temp":30}
```

[Playground](#)

Decoding

To decode a `City` struct, use `json.Unmarshal` as in the basic example:

```
// data to decode
bytes := []byte(`{"name":"Rome","temp":30}`)

// initialize the container for the decoded data
var city City

// decode the data
// notice the use of &city to pass the pointer to city
if err := json.Unmarshal(bytes, &city); err != nil {
    panic(err)
}

fmt.Println(city)
// {Rome 30}
```

[Playground](#)

Read JSON online: <https://riptutorial.com/go/topic/994/json>

Chapter 36: JWT Authorization in Go

Introduction

JSON Web Tokens (JWTs) are a popular method for representing claims securely between two parties. Understanding how to work with them is important when developing web applications or application programming interfaces.

Remarks

context.Context and HTTP middleware are outside the scope of this topic, but nonetheless those curious, wandering souls should check out <https://github.com/goware/jwtauth>, <https://github.com/auth0/go-jwt-middleware>, and <https://github.com/dgrijalva/jwt-go>.

Huge kudos to Dave Grijalva for his amazing work on go-jwt.

Examples

Parsing and validating a token using the HMAC signing method

```
// sample token string taken from the New example
tokenString :=
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmb28iOiJiYXIIiLCJuYmYiOiJlE0NDQ0Nzg0MDB9.ulriaD1rW97opCoAuRCTy4wZk-bh7vLiRIsrpU"

// Parse takes the token string and a function for looking up the key. The latter is
// especially
// useful if you use multiple keys for your application. The standard is to use 'kid' in the
// head of the token to identify which key to use, but the parsed token (head and claims) is
// provided
// to the callback, providing flexibility.
token, err := jwt.Parse(tokenString, func(token *jwt.Token) (interface{}, error) {
    // Don't forget to validate the alg is what you expect:
    if _, ok := token.Method.(*jwt.SigningMethodHMAC); !ok {
        return nil, fmt.Errorf("Unexpected signing method: %v", token.Header["alg"])
    }

    // hmacSampleSecret is a []byte containing your secret, e.g. []byte("my_secret_key")
    return hmacSampleSecret, nil
})

if claims, ok := token.Claims.(jwt.MapClaims); ok && token.Valid {
    fmt.Println(claims["foo"], claims["nbf"])
} else {
    fmt.Println(err)
}
```

Output:

```
bar 1.4444784e+09
```


(From the [documentation](#), courtesy of Dave Grijalva.)

Creating a token using a custom claims type

The `StandardClaim` is embedded in the custom type to allow for easy encoding, parsing and validation of standard claims.

```
tokenString :=
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmb28iOiJiYXIIiLCJleHAiOjE1MDAwLCJpc3MiOiJ0ZXN0In0.HE7fK0xOQwFE1

type MyCustomClaims struct {
    Foo string `json:"foo"`
    jwt.StandardClaims
}

// sample token is expired.  override time so it parses as valid
at(time.Unix(0, 0), func() {
    token, err := jwt.ParseWithClaims(tokenString, &MyCustomClaims{}, func(token *jwt.Token)
(interface{}, error) {
        return []byte("AllYourBase"), nil
    })

    if claims, ok := token.Claims.(*MyCustomClaims); ok && token.Valid {
        fmt.Printf("%v %v", claims.Foo, claims.StandardClaims.ExpiresAt)
    } else {
        fmt.Println(err)
    }
})
```

Output:

```
bar 15000
```

(From the [documentation](#), courtesy of Dave Grijalva.)

Creating, signing, and encoding a JWT token using the HMAC signing method

```
// Create a new token object, specifying signing method and the claims
// you would like it to contain.
token := jwt.NewWithClaims(jwt.SigningMethodHS256, jwt.MapClaims{
    "foo": "bar",
    "nbf": time.Date(2015, 10, 10, 12, 0, 0, 0, time.UTC).Unix(),
})

// Sign and get the complete encoded token as a string using the secret
tokenString, err := token.SignedString(hmacSampleSecret)

fmt.Println(tokenString, err)
```

Output:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmb28iOiJiYXIIiLCJleHAiOjE0NDQ0Nzg0MDBB9.ulriaD1rW97opCoAuRCTy4w5Zk-bh7vLiRIsrpU <nil>
```

(From the [documentation](#), courtesy of Dave Grijalva.)

Using the StandardClaims type by itself to parse a token

The `StandardClaims` type is designed to be embedded into your custom types to provide standard validation features. You can use it alone, but there's no way to retrieve other fields after parsing. See the custom claims example for intended usage.

```
mySigningKey := []byte("AllYourBase")

// Create the Claims
claims := &jwt.StandardClaims{
    ExpiresAt: 15000,
    Issuer:    "test",
}

token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
ss, err := token.SignedString(mySigningKey)
fmt.Printf("%v %v", ss, err)
```

Output:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1MDAwLCJpc3MiOiJ0ZXN0In0.QsODzZu3lUZMVdhbO76u3Jv02iYCVI
<nil>
```

(From the [documentation](#), courtesy of Dave Grijalva.)

Parsing the error types using bitfield checks

```
// Token from another example. This token is expired
var tokenString =
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmb28iOiJiYXIIiLCJleHAiOjE1MDAwLCJpc3MiOiJ0ZXN0In0.HE7fK0xOQwFE

token, err := jwt.Parse(tokenString, func(token *jwt.Token) (interface{}, error) {
    return []byte("AllYourBase"), nil
})

if token.Valid {
    fmt.Println("You look nice today")
} else if ve, ok := err.(*jwt.ValidationError); ok {
    if ve.Errors&jwt.ValidationErrorMalformed != 0 {
        fmt.Println("That's not even a token")
    } else if ve.Errors&(jwt.ValidationErrorExpired|jwt.ValidationErrorNotValidYet) != 0 {
        // Token is either expired or not active yet
        fmt.Println("Timing is everything")
    } else {
        fmt.Println("Couldn't handle this token:", err)
    }
} else {
    fmt.Println("Couldn't handle this token:", err)
}
```

Output:

(From the [documentation](#), courtesy of Dave Grijalva.)

Getting token from HTTP Authorization header

```
type contextKey string

const (
    // JWTTokenContextKey holds the key used to store a JWT Token in the
    // context.
    JWTTokenContextKey contextKey = "JWTToken"

    // JWTClaimsContextKey holds the key used to store the JWT Claims in the
    // context.
    JWTClaimsContextKey contextKey = "JWTClaims"
)

// ToHTTPContext moves JWT token from request header to context.
func ToHTTPContext() http.RequestFunc {
    return func(ctx context.Context, r *stdhttp.Request) context.Context {
        token, ok := extractTokenFromAuthHeader(r.Header.Get("Authorization"))
        if !ok {
            return ctx
        }

        return ctx.WithValue(ctx, JWTTokenContextKey, token)
    }
}
```

(From [go-kit/kit](#), courtesy of Peter Bourgon)

Read JWT Authorization in Go online: <https://riptutorial.com/go/topic/10161/jwt-authorization-in-go>

Chapter 37: Logging

Examples

Basic Printing

Go has a built-in logging library known as `log` with a commonly use method `Print` and its variants. You can import the library then do some basic printing:

```
package main

import "log"

func main() {

    log.Println("Hello, world!")
    // Prints 'Hello, world!' on a single line

    log.Print("Hello, again! \n")
    // Prints 'Hello, again!' but doesn't break at the end without \n

    hello := "Hello, Stackers!"
    log.Printf("The type of hello is: %T \n", hello)
    // Allows you to use standard string formatting. Prints the type 'string' for %T
    // 'The type of hello is: string'
}
```

Logging to file

It is possible to specify log destination with something that statisfies `io.Writer` interface. With that we can log to file:

```
package main

import (
    "log"
    "os"
)

func main() {
    logfile, err := os.OpenFile("test.log", os.O_RDWR|os.O_CREATE|os.O_APPEND, 0666)
    if err != nil {
        log.Fatalf(err)
    }
    defer logfile.Close()

    log.SetOutput(logfile)
    log.Println("Log entry")
}
```

Output:

```
$ cat test.log
2016/07/26 07:29:05 Log entry
```

Logging to syslog

It is also possible to log to syslog with `log/syslog` like this:

```
package main

import (
    "log"
    "log/syslog"
)

func main() {
    syslogger, err := syslog.New(syslog.LOG_INFO, "syslog_example")
    if err != nil {
        log.Fatalf(err)
    }

    log.SetOutput(syslogger)
    log.Println("Log entry")
}
```

After running we will be able to see that line in syslog:

```
Jul 26 07:35:21 localhost syslog_example[18358]: 2016/07/26 07:35:21 Log entry
```

Read Logging online: <https://riptutorial.com/go/topic/3724/logging>

Chapter 38: Loops

Introduction

As one of the most basic functions in programming, loops are an important piece to nearly every programming language. Loops enable developers to set certain portions of their code to repeat through a number of loops which are referred to as iterations. This topic covers using multiple types of loops and applications of loops in Go.

Examples

Basic Loop

`for` is the only loop statement in go, so a basic loop implementation could look like this:

```
// like if, for doesn't use parens either.
// variables declared in for and if are local to their scope.
for x := 0; x < 3; x++ { // ++ is a statement.
    fmt.Println("iteration", x)
}

// would print:
// iteration 0
// iteration 1
// iteration 2
```

Break and Continue

Breaking out of the loop and continuing to the next iteration is also supported in Go, like in many other languages:

```
for x := 0; x < 10; x++ { // loop through 0 to 9
    if x < 3 { // skips all the numbers before 3
        continue
    }
    if x > 5 { // breaks out of the loop once x == 6
        break
    }
    fmt.Println("iteration", x)
}

// would print:
// iteration 3
// iteration 4
// iteration 5
```

The `break` and `continue` statements additionally accept an optional label, used to identify outer loops to target with the statement:

```

OuterLoop:
for {
    for {
        if allDone() {
            break OuterLoop
        }
        if innerDone() {
            continue OuterLoop
        }
        // do something
    }
}

```

Conditional loop

The `for` keyword is also used for conditional loops, traditionally `while` loops in other programming languages.

```

package main

import (
    "fmt"
)

func main() {
    i := 0
    for i < 3 { // Will repeat if condition is true
        i++
        fmt.Println(i)
    }
}

```

[play it on playground](#)

Will output:

```

1
2
3

```

infinite loop:

```

for {
    // This will run until a return or break.
}

```

Different Forms of For Loop

Simple form using one variable:

```

for i := 0; i < 10; i++ {
    fmt.Print(i, " ")
}

```

Using two variables (or more):

```
for i, j := 0, 0; i < 5 && j < 10; i, j = i+1, j+2 {  
    fmt.Println(i, j)  
}
```

Without using initialization statement:

```
i := 0  
for ; i < 10; i++ {  
    fmt.Print(i, " ")  
}
```

Without a test expression:

```
for i := 1; ; i++ {  
    if i&1 == 1 {  
        continue  
    }  
    if i == 22 {  
        break  
    }  
    fmt.Print(i, " ")  
}
```

Without increment expression:

```
for i := 0; i < 10; {  
    fmt.Print(i, " ")  
    i++  
}
```

When all three initialization, test and increment expressions are removed, the loop becomes infinite:

```
i := 0  
for {  
    fmt.Print(i, " ")  
    i++  
    if i == 10 {  
        break  
    }  
}
```

This is an example of infinite loop with counter initialized with zero:

```
for i := 0; ; {  
    fmt.Print(i, " ")  
    if i == 9 {  
        break  
    }  
    i++  
}
```


When just the test expression is used (acts like a typical while loop):

```
i := 0
for i < 10 {
    fmt.Print(i, " ")
    i++
}
```

Using just increment expression:

```
i := 0
for ; ; i++ {
    fmt.Print(i, " ")
    if i == 9 {
        break
    }
}
```

Iterate over a range of values using index and value:

```
ary := [5]int{0, 1, 2, 3, 4}
for index, value := range ary {
    fmt.Println("ary[", index, "] =", value)
}
```

Iterate over a range using just index:

```
for index := range ary {
    fmt.Println("ary[", index, "] =", ary[index])
}
```

Iterate over a range using just index:

```
for index, _ := range ary {
    fmt.Println("ary[", index, "] =", ary[index])
}
```

Iterate over a range using just value:

```
for _, value := range ary {
    fmt.Print(value, " ")
}
```

Iterate over a range using key and value for map (may not be in order):

```
mp := map[string]int{"One": 1, "Two": 2, "Three": 3}
for key, value := range mp {
    fmt.Println("map[", key, "] =", value)
}
```

Iterate over a range using just key for map (may be not in order):

```
for key := range mp {
    fmt.Print(key, " ") //One Two Three
}
```

Iterate over a range using just key for map (may be not in order):

```
for key, _ := range mp {
    fmt.Print(key, " ") //One Two Three
}
```

Iterate over a range using just value for map (may be not in order):

```
for _, value := range mp {
    fmt.Print(value, " ") //2 3 1
}
```

Iterate over a range for channels (exits if the channel is closed):

```
ch := make(chan int, 10)
for i := 0; i < 10; i++ {
    ch <- i
}
close(ch)

for i := range ch {
    fmt.Print(i, " ")
}
```

Iterate over a range for string (gives Unicode code points):

```
utf8str := "B = \u00b5H" //B = µH
for _, r := range utf8str {
    fmt.Print(r, " ") //66 32 61 32 181 72
}
fmt.Println()
for _, v := range []byte(utf8str) {
    fmt.Print(v, " ") //66 32 61 32 194 181 72
}
fmt.Println(len(utf8str)) //7
```

as you see `utf8str` has 6 runes (Unicode code points) and 7 bytes.

Timed loop

```
package main

import (
    "fmt"
    "time"
)

func main() {
    for _ = range time.Tick(time.Second * 3) {
        fmt.Println("Ticking every 3 seconds")
    }
}
```

```
}  
}
```

Read Loops online: <https://riptutorial.com/go/topic/975/loops>

Chapter 39: Maps

Introduction

Maps are data types used for storing unordered key-value pairs, so that looking up the value associated to a given key is very efficient. Keys are unique. The underlying data structure grows as needed to accommodate new elements, so the programmer does not need to worry about memory management. They are similar to what other languages call hash tables, dictionaries, or associative arrays.

Syntax

- `var mapName map[KeyType]ValueType` // declare a Map
- `var mapName = map[KeyType]ValueType{}` // declare and assign an empty Map
- `var mapName = map[KeyType]ValueType{key1: val1, key2: val2}` // declare and assign a Map
- `mapName := make(map[KeyType]ValueType)` // declare and initialize default size map
- `mapName := make(map[KeyType]ValueType, length)` // declare and initialize *length* size map
- `mapName := map[KeyType]ValueType{}` // auto-declare and assign an empty Map with `:=`
- `mapName := map[KeyType]ValueType{key1: value1, key2: value2}` // auto-declare and assign a Map with `:=`
- `value := mapName[key]` // Get value by key
- `value, hasKey := mapName[key]` // Get value by key, 'hasKey' is 'true' if key exists in map
- `mapName[key] = value` // Set value by key

Remarks

Go provides a built-in `map` type that implements a *hash table*. Maps are Go's built-in associative data type (also called *hashes* or *dictionaries* in other languages).

Examples

Declaring and initializing a map

You define a map using the keyword `map`, followed by the types of its keys and its values:

```
// Keys are ints, values are ints.
var m1 map[int]int // initialized to nil

// Keys are strings, values are ints.
var m2 map[string]int // initialized to nil
```

Maps are reference types, and once defined they have a *zero value* of `nil`. Writes to nil maps will

[panic](#) and reads will always return the zero value.

To initialize a map, use the [make](#) function:

```
m := make(map[string]int)
```

With the two-parameter form of `make`, it's possible to specify an initial entry capacity for the map, overriding the default capacity:

```
m := make(map[string]int, 30)
```

Alternatively, you can declare a map, initializing it to its zero value, and then assign a literal value to it later, which helps if you marshal the struct into json thereby producing an empty map on return.

```
m := make(map[string]int, 0)
```

You can also make a map and set its initial value with curly brackets (`{}`).

```
var m map[string]int = map[string]int{"Foo": 20, "Bar": 30}

fmt.Println(m["Foo"]) // outputs 20
```

All the following statements result in the variable being bound to the same value.

```
// Declare, initializing to zero value, then assign a literal value.
var m map[string]int
m = map[string]int{}

// Declare and initialize via literal value.
var m = map[string]int{}

// Declare via short variable declaration and initialize with a literal value.
m := map[string]int{}
```

We can also use a *map literal* to [create a new map with some initial key/value pairs](#).

The key type can be any [comparable](#) type; notably, [this excludes functions, maps, and slices](#). The value type can be any type, including custom types or `interface{}`.

```
type Person struct {
    FirstName string
    LastName  string
}

// Declare via short variable declaration and initialize with make.
m := make(map[string]Person)

// Declare, initializing to zero value, then assign a literal value.
var m map[string]Person
m = map[string]Person{}
```

```
// Declare and initialize via literal value.
var m = map[string]Person{}

// Declare via short variable declaration and initialize with a literal value.
m := map[string]Person{}
```

Creating a map

One can declare and initialize a map in a single statement using a [composite literal](#).

Using automatic type Short variable declaration:

```
mapIntInt := map[int]int{10: 100, 20: 100, 30: 1000}
mapIntString := map[int]string{10: "foo", 20: "bar", 30: "baz"}
mapStringInt := map[string]int{"foo": 10, "bar": 20, "baz": 30}
mapStringString := map[string]string{"foo": "one", "bar": "two", "baz": "three"}
```

The same code, but with Variable types:

```
var mapIntInt = map[int]int{10: 100, 20: 100, 30: 1000}
var mapIntString = map[int]string{10: "foo", 20: "bar", 30: "baz"}
var mapStringInt = map[string]int{"foo": 10, "bar": 20, "baz": 30}
var mapStringString = map[string]string{"foo": "one", "bar": "two", "baz": "three"}
```

You can also include your own structs in a map:

You can use custom types as value:

```
// Custom struct types
type Person struct {
    FirstName, LastName string
}

var mapStringPerson = map[string]Person{
    "john": Person{"John", "Doe"},
    "jane": Person{"Jane", "Doe"}}
mapStringPerson := map[string]Person{
    "john": Person{"John", "Doe"},
    "jane": Person{"Jane", "Doe"}}
```

Your struct can also be the *key* to the map:

```
type RouteHit struct {
    Domain string
    Route string
}

var hitMap = map[RouteHit]int{
    RouteHit{"example.com", "/home"}: 1,
    RouteHit{"example.com", "/help"}: 2}
hitMap := map[RouteHit]int{
    RouteHit{"example.com", "/home"}: 1,
    RouteHit{"example.com", "/help"}: 2}
```

You can create an empty map simply by not entering any value within the brackets `{}`.

```
mapIntInt := map[int]int{}
mapIntString := map[int]string{}
mapStringInt := map[string]int{}
mapStringString := map[string]string{}
mapStringPerson := map[string]Person{}
```

You can create and use a map directly, without the need to assign it to a variable. However, you will have to specify both the declaration and the content.

```
// using a map as argument for fmt.Println()
fmt.Println(map[string]string{
    "FirstName": "John",
    "LastName": "Doe",
    "Age": "30"})

// equivalent to
data := map[string]string{
    "FirstName": "John",
    "LastName": "Doe",
    "Age": "30"}
fmt.Println(data)
```

Zero value of a map

The zero value of a `map` is `nil` and has a length of 0.

```
var m map[string]string
fmt.Println(m == nil) // true
fmt.Println(len(m) == 0) // true
```

A `nil` map has no keys nor can keys be added. A `nil` map behaves like an empty map if read from but causes a runtime panic if written to.

```
var m map[string]string

// reading
m["foo"] == "" // true. Remember "" is the zero value for a string
_, ok = m["foo"] // ok == false

// writing
m["foo"] = "bar" // panic: assignment to entry in nil map
```

You should not try to read from or write to a zero value map. Instead, initialize the map (with `make` or assignment) before using it.

```
var m map[string]string
m = make(map[string]string) // OR m = map[string]string{}
m["foo"] = "bar"
```

Iterating the elements of a map

```
import fmt

people := map[string]int{
    "john": 30,
    "jane": 29,
    "mark": 11,
}

for key, value := range people {
    fmt.Println("Name:", key, "Age:", value)
}
```

Note that when iterating over a map with a range loop, [the iteration order is not specified](#) and is not guaranteed to be the same from one iteration to the next.

You can also discard either the keys or the values of the map, if you are looking to just [grab keys](#) or just grab values.

Iterating the keys of a map

```
people := map[string]int{
    "john": 30,
    "jane": 29,
    "mark": 11,
}

for key, _ := range people {
    fmt.Println("Name:", key)
}
```

If you are just looking for the keys, since they are the first value, you can simply drop the underscore:

```
for key := range people {
    fmt.Println("Name:", key)
}
```

Note that when iterating over a map with a range loop, [the iteration order is not specified](#) and is not guaranteed to be the same from one iteration to the next.

Deleting a map element

The [delete](#) built-in function removes the element with the specified key from a map.

```
people := map[string]int{"john": 30, "jane": 29}
fmt.Println(people) // map[john:30 jane:29]

delete(people, "john")
fmt.Println(people) // map[jane:29]
```

If the `map` is `nil` or there is no such element, `delete` has no effect.


```
people := map[string]int{"john": 30, "jane": 29}
fmt.Println(people) // map[john:30 jane:29]

delete(people, "notfound")
fmt.Println(people) // map[john:30 jane:29]

var something map[string]int
delete(something, "notfound") // no-op
```

Counting map elements

The built-in function `len` returns the number of elements in a `map`

```
m := map[string]int{}
len(m) // 0

m["foo"] = 1
len(m) // 1
```

If a variable points to a `nil` map, then `len` returns 0.

```
var m map[string]int
len(m) // 0
```

Concurrent Access of Maps

Maps in go are not safe for concurrency. You must take a lock to read and write on them if you will be accessing them concurrently. Usually the best option is to use `sync.RWMutex` because you can have read and write locks. However, a `sync.Mutex` could also be used.

```
type RWMutex struct {
    sync.RWMutex
    m map[string]int
}

// Get is a wrapper for getting the value from the underlying map
func (r RWMutex) Get(key string) int {
    r.RLock()
    defer r.RUnlock()
    return r.m[key]
}

// Set is a wrapper for setting the value of a key in the underlying map
func (r RWMutex) Set(key string, val int) {
    r.Lock()
    defer r.Unlock()
    r.m[key] = val
}

// Inc increases the value in the RWMutex for a key.
// This is more pleasant than r.Set(key, r.Get(key)++)
func (r RWMutex) Inc(key string) {
    r.Lock()
    defer r.Unlock()
    r.m[key]++
}
```

```

}

func main() {

    // Init
    counter := RWMutex{m: make(map[string]int)}

    // Get a Read Lock
    counter.RLock()
    _ = counter["Key"]
    counter.RUnlock()

    // the above could be replaced with
    _ = counter.Get("Key")

    // Get a write Lock
    counter.Lock()
    counter.m["some_key"]++
    counter.Unlock()

    // above would need to be written as
    counter.Inc("some_key")
}

```

The trade-off of the wrapper functions is between the public access of the underlying map and using the appropriate locks correctly.

Creating maps with slices as values

```
m := make(map[string][]int)
```

Accessing a non-existent key will return a nil slice as a value. Since nil slices act like zero length slices when used with `append` or other built-in functions you do not normally need to check to see if a key exists:

```

// m["key1"] == nil && len(m["key1"]) == 0
m["key1"] = append(m["key1"], 1)
// len(m["key1"]) == 1

```

Deleting a key from map sets the key back to a nil slice:

```

delete(m, "key1")
// m["key1"] == nil

```

Check for element in a map

To get a value from the map, you just have to do something like:

```
value := mapName[ key ]
```

If the map contains the key, it returns the corresponding value.

If not, it returns zero-value of the map's value type (0 if map of `int` values, "" if map of `string`

values...)

```
m := map[string]string{"foo": "foo_value", "bar": ""}
k := m["foo"] // returns "foo_value" since that is the value stored in the map
k2 := m["bar"] // returns "" since that is the value stored in the map
k3 := m["nop"] // returns "" since the key does not exist, and "" is the string type's zero value
```

To differentiate between empty values and non-existent keys, you can use the second returned value of the map access (using like `value, hasKey := map["key"]`).

This second value is `boolean` typed, and will be:

- `true` when the value is in the map,
- `false` when the map does not contains the given key.

Look at the following example:

```
value, hasKey = m[ key ]
if hasKey {
    // the map contains the given key, so we can safely use the value
    // If value is zero-value, it's because the zero-value was pushed to the map
} else {
    // The map does not have the given key
    // the value will be the zero-value of the map's type
}
```

Iterating the values of a map

```
people := map[string]int{
    "john": 30,
    "jane": 29,
    "mark": 11,
}

for _, value := range people {
    fmt.Println("Age:", value)
}
```

Note that when iterating over a map with a range loop, [the iteration order is not specified](#) and is not guaranteed to be the same from one iteration to the next.

Copy a Map

Like slices, maps hold **references** to an underlying data structure. So by assigning its value to another variable, only the reference will be passed. To copy the map, it is necessary to create another map and copy each value:

```
// Create the original map
originalMap := make(map[string]int)
originalMap["one"] = 1
originalMap["two"] = 2
```

```
// Create the target map
targetMap := make(map[string]int)

// Copy from the original map to the target map
for key, value := range originalMap {
    targetMap[key] = value
}
```

Using a map as a set

Some languages have a native structure for sets. To make a set in Go, it's best practice to use a map from the value type of the set to an empty struct (`map[Type]struct{}`).

For example, with strings:

```
// To initialize a set of strings:
greetings := map[string]struct{}{
    "hi":    {},
    "hello": {},
}

// To delete a value:
delete(greetings, "hi")

// To add a value:
greetings["hey"] = struct{}{}

// To check if a value is in the set:
if _, ok := greetings["hey"]; ok {
    fmt.Println("hey is in greetings")
}
```

Read Maps online: <https://riptutorial.com/go/topic/732/maps>

Chapter 40: Memory pooling

Introduction

`sync.Pool` stores a cache of allocated but unused items for future use, avoiding memory churn for frequently changed collections, and allowing efficient, thread-safe re-use of memory. It is useful to manage a group of temporary items shared between concurrent clients of a package, for example a list of database connections or a list of output buffers.

Examples

`sync.Pool`

Using `sync.Pool` structure we can pool objects and reuse them.

```
package main

import (
    "bytes"
    "fmt"
    "sync"
)

var pool = sync.Pool{
    // New creates an object when the pool has nothing available to return.
    // New must return an interface{} to make it flexible. You have to cast
    // your type after getting it.
    New: func() interface{} {
        // Pools often contain things like *bytes.Buffer, which are
        // temporary and re-usable.
        return &bytes.Buffer{}
    },
}

func main() {
    // When getting from a Pool, you need to cast
    s := pool.Get().(*bytes.Buffer)
    // We write to the object
    s.Write([]byte("dirty"))
    // Then put it back
    pool.Put(s)

    // Pools can return dirty results

    // Get 'another' buffer
    s = pool.Get().(*bytes.Buffer)
    // Write to it
    s.Write([]bytes("append"))
    // At this point, if GC ran, this buffer *might* exist already, in
    // which case it will contain the bytes of the string "dirtyappend"
    fmt.Println(s)
    // So use pools wisely, and clean up after yourself
    s.Reset()
}
```

```
pool.Put(s)

// When you clean up, your buffer should be empty
s = pool.Get().(*bytes.Buffer)
// Defer your Puts to make sure you don't leak!
defer pool.Put(s)
s.Write([]byte("reset!"))
// This prints "reset!", and not "dirtyappendreset!"
fmt.Println(s)
}
```

Read Memory pooling online: <https://riptutorial.com/go/topic/4647/memory-pooling>

Chapter 41: Methods

Syntax

- `func (t T) exampleOne(i int) (n int) { return i }` // this function will receive copy of struct
- `func (t *T) exampleTwo(i int) (n int) { return i }` // this method will receive pointer to struct and will be able to modify it

Examples

Basic methods

Methods in Go are just like functions, except they have *receiver*.

Usually receiver is some kind of struct or type.

```
package main

import (
    "fmt"
)

type Employee struct {
    Name string
    Age  int
    Rank int
}

func (empl *Employee) Promote() {
    empl.Rank++
}

func main() {

    Bob := new(Employee)

    Bob.Rank = 1
    fmt.Println("Bobs rank now is: ", Bob.Rank)
    fmt.Println("Lets promote Bob!")

    Bob.Promote()

    fmt.Println("Now Bobs rank is: ", Bob.Rank)
}
```

Output:

```
Bobs rank now is:  1
Lets promote Bob!
Now Bobs rank is:  2
```

Chaining methods

With methods in go lang you can do method "chaining" passing pointer to method and returning pointer to the same struct like this:

```
package main

import (
    "fmt"
)

type Employee struct {
    Name string
    Age  int
    Rank int
}

func (empl *Employee) Promote() *Employee {
    fmt.Printf("Promoting %s\n", empl.Name)
    empl.Rank++
    return empl
}

func (empl *Employee) SetName(name string) *Employee {
    fmt.Printf("Set name of new Employee to %s\n", name)
    empl.Name = name
    return empl
}

func main() {

    worker := new(Employee)

    worker.Rank = 1

    worker.SetName("Bob").Promote()

    fmt.Printf("Here we have %s with rank %d\n", worker.Name, worker.Rank)

}
```

Output:

```
Set name of new Employee to Bob
Promoting Bob
Here we have Bob with rank 2
```

Increment-Decrement operators as arguments in Methods

Though Go supports ++ and -- operators and the behaviour is found to be almost similar to c/c++, variables with such operators cannot be passed as argument to function.

```
package main

import (
    "fmt"
)
```



```
)  
  
func abcd(a int, b int) {  
    fmt.Println(a, " ",b)  
}  
func main() {  
    a:=5  
    abcd(a++,++a)  
}
```

Output: syntax error: unexpected ++, expecting comma or)

Read Methods online: <https://riptutorial.com/go/topic/3890/methods>

Chapter 42: mgo

Introduction

mgo (pronounced as mango) is a MongoDB driver for the Go language that implements a rich and well tested selection of features under a very simple API following standard Go idioms.

Remarks

API Documentation

[\[https://gopkg.in/mgo.v2\]](https://gopkg.in/mgo.v2)^[1]

Examples

Example

```
package main

import (
    "fmt"
    "log"
    "gopkg.in/mgo.v2"
    "gopkg.in/mgo.v2/bson"
)

type Person struct {
    Name string
    Phone string
}

func main() {
    session, err := mgo.Dial("server1.example.com,server2.example.com")
    if err != nil {
        panic(err)
    }
    defer session.Close()

    // Optional. Switch the session to a monotonic behavior.
    session.SetMode(mgo.Monotonic, true)

    c := session.DB("test").C("people")
    err = c.Insert(&Person{"Ale", "+55 53 8116 9639"},
        &Person{"Cla", "+55 53 8402 8510"})
    if err != nil {
        log.Fatal(err)
    }

    result := Person{}
    err = c.Find(bson.M{"name": "Ale"}).One(&result)
    if err != nil {
        log.Fatal(err)
    }
}
```

```
    }  
  
    fmt.Println("Phone:", result.Phone)  
}
```

Read mgo online: <https://riptutorial.com/go/topic/8898/mgo>

Chapter 43: Middleware

Introduction

In Go Middleware can be used to execute code before and after handler function. It uses the power of Single Function Interfaces. Can be introduced at any time without affecting the other middleware. For Ex: Authentication logging can be added in later stages of development without disturbing the existing code.

Remarks

The **Signature of middleware** should be (http.ResponseWriter, *http.Request) i.e. of **http.HandlerFunc** type.

Examples

Normal Handler Function

```
func loginHandler(w http.ResponseWriter, r *http.Request) {
    // Steps to login
}

func main() {
    http.HandleFunc("/login", loginHandler)
    http.ListenAndServe(":8080", nil)
}
```

Middleware Calculate time required for handlerFunc to execute

```
// logger middleware that logs time taken to process each request
func Logger(h http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        startTime := time.Now()
        h.ServeHTTP(w, r)
        endTime := time.Since(startTime)
        log.Printf("%s %d %v", r.URL, r.Method, endTime)
    })
}

func loginHandler(w http.ResponseWriter, r *http.Request) {
    // Steps to login
}

func main() {
    http.HandleFunc("/login", Logger(loginHandler))
    http.ListenAndServe(":8080", nil)
}
```

CORS Middleware

```
func CORS(h http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        origin := r.Header.Get("Origin")
        w.Header().Set("Access-Control-Allow-Origin", origin)
        if r.Method == "OPTIONS" {
            w.Header().Set("Access-Control-Allow-Credentials", "true")
            w.Header().Set("Access-Control-Allow-Methods", "GET,POST")

            w.RespWriter.Header().Set("Access-Control-Allow-Headers", "Content-Type, X-CSRF-Token, Authorization")
            return
        } else {
            h.ServeHTTP(w, r)
        }
    })
}

func main() {
    http.HandleFunc("/login", Logger(CORS(loginHandler)))
    http.ListenAndServe(":8080", nil)
}
```

Auth Middleware

```
func Authenticate(h http.Handler) http.Handler {
    return CustomHandlerFunc(func(w *http.ResponseWriter, r *http.Request) {
        // extract params from req
        // post params | headers etc
        if CheckAuth(params) {
            log.Println("Auth Pass")
            // pass control to next middleware in chain or handler func
            h.ServeHTTP(w, r)
        } else {
            log.Println("Auth Fail")
            // Responsd Auth Fail
        }
    })
}
```

Recovery Handler to prevent server from crashing

```
func Recovery(h http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        defer func() {
            if err := recover(); err != nil {
                // respondInternalServerError
            }
        }()
        h.ServeHTTP(w, r)
    })
}
```

Read Middleware online: <https://riptutorial.com/go/topic/9343/middleware>

Chapter 44: Mutex

Examples

Mutex Locking

Mutex locking in Go allows you to ensure that only one goroutine at a time has a lock:

```
import "sync"

func mutexTest() {
    lock := sync.Mutex{}
    go func(m *sync.Mutex) {
        m.Lock()
        defer m.Unlock() // Automatically unlock when this function returns
        // Do some things
    }(&lock)

    lock.Lock()
    // Do some other things
    lock.Unlock()
}
```

Using a `Mutex` allows you to avoid race conditions, concurrent modifications, and other issues associated with multiple concurrent routines operating on the same resources. Note that `Mutex.Unlock()` can be executed by any routine, not just the routine that got the lock. Also note that the call to `Mutex.Lock()` will not fail if another routine holds the lock; it will block until the lock is released.

Tip: Whenever you're passing a `Mutex` variable to a function, always pass it as a pointer. Otherwise a copy is made of your variable, which defeats the purpose of the `Mutex`. If you're using an older Go version (< 1.7), the compiler will not warn you about this mistake!

Read `Mutex` online: <https://riptutorial.com/go/topic/2607/mutex>

Chapter 45: Object Oriented Programming

Remarks

Interface can't be implemented with pointer receivers because `*User` is not `User`

Examples

Structs

Go supports user defined types in the form of structs and type aliases. structs are composite types, the component pieces of data that constitute the struct type are called *fields*. a field has a type and a name which must be unique.

```
package main

type User struct {
    ID uint64
    FullName string
    Email    string
}

func main() {
    user := User{
        1,
        "Zelalem Mekonen",
        "zola.mk.27@gmail.com",
    }

    fmt.Println(user) // {1 Zelalem Mekonen zola.mk.27@gmail.com}
}
```

this is also a legal syntax for defining structs

```
type User struct {
    ID uint64
    FullName, Email string
}

user := new(User)

user.ID = 1
user.FullName = "Zelalem Mekonen"
user.Email = "zola.mk.27@gmail.com"
```

Embedded structs

because a struct is also a data type, it can be used as an anonymous field, the outer struct can directly access the fields of the embedded struct even if the struct came from a different package. this behaviour provides a way to derive some or all of your implementation from another type or a

set of types.

```
package main

type Admin struct {
    Username, Password string
}

type User struct {
    ID uint64
    FullName, Email string
    Admin // embedded struct
}

func main() {
    admin := Admin{
        "zola",
        "supersecretpassword",
    }

    user := User{
        1,
        "Zelalem Mekonen",
        "zola.mk.27@gmail.com",
        admin,
    }

    fmt.Println(admin) // {zola supersecretpassword}

    fmt.Println(user) // {1 Zelalem Mekonen zola.mk.27@gmail.com {zola supersecretpassword}}

    fmt.Println(user.Username) // zola

    fmt.Println(user.Password) // supersecretpassword
}
```

Methods

In Go a method is

a function that acts on a variable of a certain type, called the receiver

the receiver can be anything, not only `structs` but even a `function`, alias types for built in types such as `int`, `string`, `bool` can have a method, an exception to this rule is that `interfaces` (discussed later) cannot have methods, since an interface is an abstract definition and a method is an implementation, trying it generate a compile error.

combining `structs` and `methods` you can get a close equivalent of a `class` in Object Oriented programming.

a method in Go has the following signature

```
func (name receiverType) methodName(paramterList) (returnList) {}
```

```
package main
```



```

type Admin struct {
    Username, Password string
}

func (admin Admin) Delete() {
    fmt.Println("Admin Deleted")
}

type User struct {
    ID uint64
    FullName, Email string
    Admin
}

func (user User) SendEmail(email string) {
    fmt.Printf("Email sent to: %s\n", user.Email)
}

func main() {
    admin := Admin{
        "zola",
        "supersecretpassword",
    }

    user := User{
        1,
        "Zelalem Mekonen",
        "zola.mk.27@gmail.com",
        admin,
    }

    user.SendEmail("Hello") // Email sent to: zola.mk.27@gmail.com

    admin.Delete() // Admin Deleted
}

```

Pointer Vs Value receiver

the receiver of a method is usually a pointer for performance reason because we wouldn't make a copy of the instance, as it would be the case in value receiver, this is especially true if the receiver type is a struct. another reason to make the receiver type a pointer would be so we could modify the data the receiver points to.

a value receiver is used to avoid modification of the data the receiver contains, a value receiver may cause a performance hit if the receiver is a large struct.

```

package main

type User struct {
    ID uint64
    FullName, Email string
}

// We do not require any special syntax to access field because receiver is a pointer
func (user *User) SendEmail(email string) {
    fmt.Printf("Sent email to: %s\n", user.Email)
}

```

```
// ChangeMail will modify the users email because the receiver type is a pointer
func (user *User) ChangeEmail(email string) {
    user.Email = email;
}

func main() {
    user := User{
        1,
        "Zelalem Mekonen",
        "zola.mk.27@gmail.com",
    }

    user.SendEmail("Hello") // Sent email to: zola.mk.27@gmail.com

    user.ChangeEmail("zola@gmail.com")

    fmt.Println(user.Email) // zola@gmail.com
}
```

Interface & Polymorphism

Interfaces provide a way to specify the behaviour of an object, if something can do this then it can be used here. an interface defines a set of methods, but these methods do not contain code as they are abstract or the implementation is left to the user of the interface. unlike most Object Oriented languages interfaces can contain variables in Go.

Polymorphism is the essence of object-oriented programming: the ability to treat objects of different types uniformly as long as they adhere to the same interface. Go interfaces provide this capability in a very direct and intuitive way

```
package main

type Runner interface {
    Run()
}

type Admin struct {
    Username, Password string
}

func (admin Admin) Run() {
    fmt.Println("Admin ==> Run()");
}

type User struct {
    ID uint64
    FullName, Email string
}

func (user User) Run() {
    fmt.Println("User ==> Run()")
}

// RunnerExample takes any type that fulfills the Runner interface
func RunnerExample(r Runner) {
    r.Run()
}
```

```
}

func main() {
    admin := Admin{
        "zola",
        "supersecretpassword",
    }

    user := User{
        1,
        "Zelalem Mekonen",
        "zola.mk.27@gmail.com",
    }

    RunnerExample(admin)

    RunnerExample(user)
}
```

Read Object Oriented Programming online: <https://riptutorial.com/go/topic/8801/object-oriented-programming>

Chapter 46: OS Signals

Syntax

- `func Notify(c chan<- os.Signal, sig ...os.Signal)`

Parameters

Parameter	Details
<code>c chan<- os.Signal</code>	Receiving <code>channel</code> specifically of type <code>os.Signal</code> ; easily created with <code>sigChan := make(chan os.Signal)</code>
<code>sig ...os.Signal</code>	List of <code>os.Signal</code> types to catch and send down this <code>channel</code> . See https://golang.org/pkg/syscall/#pkg-constants for more options.

Examples

Assigning signals to a channel

Often times you will have reason to catch when your program is being told to stop by the OS and take some actions to preserve the state, or clean up your application. To accomplish this you can use the `os/signal` package from the standard library. Below is a simple example of assigning all signals from the system to a channel, and then how to react to those signals.

```
package main

import (
    "fmt"
    "os"
    "os/signal"
)

func main() {
    // create a channel for os.Signal
    sigChan := make(chan os.Signal)

    // assign all signal notifications to the channel
    signal.Notify(sigChan)

    // blocks until you get a signal from the OS
    select {
    // when a signal is received
    case sig := <-sigChan:
        // print this line telling us which signal was seen
        fmt.Println("Received signal from OS:", sig)
    }
}
```

When you run the above script it will create a channel, and then block until that channel receives a signal.

```
$ go run signals.go
^CReceived signal from OS: interrupt
```

The `^C` above is the keyboard command `CTRL+C` which sends the `SIGINT` signal.

Read OS Signals online: <https://riptutorial.com/go/topic/4497/os-signals>

Chapter 47: Packages

Examples

Package initialization

Package can have `init` methods which are run **only once** before `main`.

```
package usefull

func init() {
    // init code
}
```

If you just want to run the package initialization without referencing anything from it use the following import expression.

```
import _ "usefull"
```

Managing package dependencies

A common way to download Go dependencies is by using the `go get <package>` command, which will save the package into the global/shared `$GOPATH/src` directory. This means that a single version of each package will be linked into each project that includes it as a dependency. This also means that when a new developers deploys your project, they will `go get` the latest version of each dependency.

However you can keep the build environment consistent, by attaching all the dependencies of a project into the `vendor/` directory. Keeping vendored dependencies committed along with your project's repository allows you to do per-project dependency versioning, and provide a consistent environment for your build.

This is what your project's structure will look like:

```
$GOPATH/src/
├─ github.com/username/project/
│   ├─ main.go
│   └─ vendor/
│       ├─ github.com/pkg/errors
│       └─ github.com/gorilla/mux
```

Using different package and folder name

It is perfectly fine to use a package name other than the folder name. If we do so, we still have to import the package based on the directory structure, but after the import we have to refer to it by the name we used in the package clause.

For example, if you have a folder `$GOPATH/src/mypck`, and in it we have a file `a.go`:

```
package apple

const Pi = 3.14
```

Using this package:

```
package main

import (
    "mypck"
    "fmt"
)

func main() {
    fmt.Println(apple.Pi)
}
```

Even though this works, you should have a good reason to deviate package name from the folder name (or it may become source of misunderstanding and confusion).

What's the use of this?

Simple. A package name is a Go [identifier](#):

```
identifier = letter { letter | unicode_digit } .
```

Which allows unicode letters to be used in identifiers, e.g. `αβ` is a valid identifier in Go. Folder and file names are not handled by Go but by the Operating System, and different file systems have different restrictions. There are actually many file systems which would not allow all valid Go identifiers as folder names, so you would not be able to name your packages what otherwise the language spec would allow.

Having the option to use different package names than their containing folders, you have the option to really name your packages what the language spec allows, regardless of the underlying operating and file system.

Importing packages

You can import a single package with the statement:

```
import "path/to/package"
```

or group multiple imports together:

```
import (
    "path/to/package1"
    "path/to/package2"
)
```

This will look in the corresponding `import` paths inside of the `$GOPATH` for `.go` files and lets you access exported names through `packagename.AnyExportedName`.

You can also access local packages inside of the current folder by prefacing packages with `./`. In a project with a structure like this:

```
project
├── src
│   ├── package1
│   │   └── file1.go
│   └── package2
│       └── file2.go
└── main.go
```

You could call this in `main.go` in order to import the code in `file1.go` and `file2.go`:

```
import (
    "./src/package1"
    "./src/package2"
)
```

Since package-names can collide in different libraries you may want to alias one package to a new name. You can do this by prefixing your import-statement with the name you want to use.

```
import (
    "fmt" //fmt from the standardlibrary
    tfmt "some/thirdparty/fmt" //fmt from some other library
)
```

This allows you to access the former `fmt` package using `fmt.*` and the latter `fmt` package using `tfmt.*`.

You can also import the package into the own namespace, so that you can refer to the exported names without the `package.` prefix using a single dot as alias:

```
import (
    . "fmt"
)
```

Above example imports `fmt` into the global namespace and lets you call, for example, `Printf` directly: [Playground](#)

If you import a package but don't use any of it's exported names, the Go compiler will print an error-message. To circumvent this, you can set the alias to the underscore:

```
import (
    _ "fmt"
)
```

This can be useful if you don't access this package directly but need it's `init` functions to run.

Note:

As the package names are based on the folder structure, any changes in the folder names & import references (including case sensitivity) will cause a compile time error "case-insensitive import collision" in Linux & OS-X, which is difficult to trace and fix (the error message is kinda cryptic for mere mortals as it tries to convey the opposite - that, the comparison failed due to case sensitivity).

ex: "path/to/Package1" vs "path/to/package1"

Live example: <https://github.com/akamai-open/AkamaiOPEN-edgegrid-golang/issues/2>

Read Packages online: <https://riptutorial.com/go/topic/401/packages>

Chapter 48: Panic and Recover

Remarks

This article assumes knowledge of [Defer Basics](#)

For ordinary error handling, read the [topic on error handling](#)

Examples

Panic

A panic halts normal execution flow and exits the current function. Any deferred calls will then be executed before control is passed to the next higher function on the stack. Each stack's function will exit and run deferred calls until the panic is handled using a deferred `recover()`, or until the panic reaches `main()` and terminates the program. If this occurs, the argument provided to `panic` and a stack trace will be printed to `stderr`.

```
package main

import "fmt"

func foo() {
    defer fmt.Println("Exiting foo")
    panic("bar")
}

func main() {
    defer fmt.Println("Exiting main")
    foo()
}
```

Output:

```
Exiting foo
Exiting main
panic: bar

goroutine 1 [running]:
panic(0x128360, 0x1040a130)
    /usr/local/go/src/runtime/panic.go:481 +0x700
main.foo()
    /tmp/sandbox550159908/main.go:7 +0x160
main.main()
    /tmp/sandbox550159908/main.go:12 +0x120
```

It is important to note that `panic` will accept any type as its parameter.

Recover

Recover as the name implies, can attempt to recover from a `panic`. The `recover` *must* be attempted in a deferred statement as normal execution flow has been halted. The `recover` statement must appear *directly* within the deferred function enclosure. Recover statements in functions called by deferred function calls will not be honored. The `recover()` call will return the argument provided to the initial panic, if the program is currently panicking. If the program is not currently panicking, `recover()` will return `nil`.

```
package main

import "fmt"

func foo() {
    panic("bar")
}

func bar() {
    defer func() {
        if msg := recover(); msg != nil {
            fmt.Printf("Recovered with message %s\n", msg)
        }
    }()
    foo()
    fmt.Println("Never gets executed")
}

func main() {
    fmt.Println("Entering main")
    bar()
    fmt.Println("Exiting main the normal way")
}
```

Output:

```
Entering main
Recovered with message bar
Exiting main the normal way
```

Read Panic and Recover online: <https://riptutorial.com/go/topic/4350/panic-and-recover>

Chapter 49: Parsing Command Line Arguments And Flags

Examples

Command line arguments

Command line arguments parsing in Go is very similar to other languages. In your code you just access slice of arguments where first argument will be the name of program itself.

Quick example:

```
package main

import (
    "fmt"
    "os"
)

func main() {

    progName := os.Args[0]
    arguments := os.Args[1:]

    fmt.Printf("Here we have program '%s' launched with following flags: ", progName)

    for _, arg := range arguments {
        fmt.Printf("%s ", arg)
    }

    fmt.Println("")
}
```

And output would be:

```
$ ./cmd test_arg1 test_arg2
Here we have program './cmd' launched with following flags: test_arg1 test_arg2
```

Each argument is just a string. In `os` package it looks like: `var Args []string`

Flags

Go standard library provides package `flag` that helps with parsing flags passed to program.

Note that `flag` package doesn't provide usual GNU-style flags. That means that multi-letter flags must be started with single hyphen like this: `-exampleflag`, not this: `--exampleflag`. GNU-style flags can be done with some 3-rd party package.

```
package main
```

```

import (
    "flag"
    "fmt"
)

func main() {

    // basic flag can be defined like this:
    stringFlag := flag.String("string.flag", "default value", "here comes usage")
    // after that stringFlag variable will become a pointer to flag value

    // if you need to store value in variable, not pointer, than you can
    // do it like:
    var intFlag int
    flag.IntVar(&intFlag, "int.flag", 1, "usage of intFlag")

    // after all flag definitions you must call
    flag.Parse()

    // then we can access our values
    fmt.Printf("Value of stringFlag is: %s\n", *stringFlag)
    fmt.Printf("Value of intFlag is: %d\n", intFlag)

}

```

`flag` does help message for us:

```

$ ./flags -h
Usage of ./flags:
-int.flag int
    usage of intFlag (default 1)
-string.flag string
    here comes usage (default "default value")

```

Call with all flags:

```

$ ./flags -string.flag test -int.flag 24
Value of stringFlag is: test
Value of intFlag is: 24

```

Call with missing flags:

```

$ ./flags
Value of stringFlag is: default value
Value of intFlag is: 1

```

Read Parsing Command Line Arguments And Flags online:

<https://riptutorial.com/go/topic/4023/parsing-command-line-arguments-and-flags>

Chapter 50: Parsing CSV files

Syntax

- `csvReader := csv.NewReader(r)`
- `data, err := csvReader.Read()`

Examples

Simple CSV parsing

Consider this CSV data:

```
#id,title,text
1,hello world,"This is a \"blog\"."
2,second time,"My
second
entry."
```

This data can be read with the following code:

```
// r can be any io.Reader, including a file.
csvReader := csv.NewReader(r)
// Set comment character to '#'.
csvReader.Comment = '#'
for {
    post, err := csvReader.Read()
    if err != nil {
        log.Println(err)
        // Will break on EOF.
        break
    }
    fmt.Printf("post with id %s is titled %q: %q\n", post[0], post[1], post[2])
}
```

And produce:

```
post with id 1 is titled "hello world": "This is a \"blog\"."
post with id 2 is titled "second time": "My\nsecond\nentry."
2009/11/10 23:00:00 EOF
```

Playground: <https://play.golang.org/p/d2E6-CGGIe>.

Read Parsing CSV files online: <https://riptutorial.com/go/topic/5818/parsing-csv-files>

Chapter 51: Plugin

Introduction

Go provides a plugin mechanism that can be used to dynamically link other Go code at runtime.

As of Go 1.8, it is only usable on Linux.

Examples

Defining and using a plugin

```
package main

import "fmt"

var V int

func F() { fmt.Printf("Hello, number %d\n", V) }
```

This can be built with:

```
go build -buildmode=plugin
```

And then loaded and used from your application:

```
p, err := plugin.Open("plugin_name.so")
if err != nil {
    panic(err)
}

v, err := p.Lookup("V")
if err != nil {
    panic(err)
}

f, err := p.Lookup("F")
if err != nil {
    panic(err)
}

*v.(*int) = 7
f.(func())() // prints "Hello, number 7"
```

Example from *The State of Go 2017*.

Read Plugin online: <https://riptutorial.com/go/topic/9150/plugin>

Chapter 52: Pointers

Syntax

- `pointer := &variable` // get pointer from variable
- `variable := *pointer` // get variable from pointer
- `*pointer = value` // set value from variable through the pointer
- `pointer := new(Struct)` // get pointer of new struct

Examples

Basic Pointers

Go supports [pointers](#), allowing you to pass references to values and records within your program.

```
package main

import "fmt"

// We'll show how pointers work in contrast to values with
// 2 functions: `zeroval` and `zeroptr`. `zeroval` has an
// `int` parameter, so arguments will be passed to it by
// value. `zeroval` will get a copy of `ival` distinct
// from the one in the calling function.
func zeroval(ival int) {
    ival = 0
}

// `zeroptr` in contrast has an `*int` parameter, meaning
// that it takes an `int` pointer. The `*iptr` code in the
// function body then dereferences the pointer from its
// memory address to the current value at that address.
// Assigning a value to a dereferenced pointer changes the
// value at the referenced address.
func zeroptr(iptr *int) {
    *iptr = 0
}
```

Once these functions are defined, you can do the following:

```
func main() {
    i := 1
    fmt.Println("initial:", i) // initial: 1

    zeroval(i)
    fmt.Println("zeroval:", i) // zeroval: 1
    // `i` is still equal to 1 because `zeroval` edited
    // a "copy" of `i`, not the original.

    // The `&i` syntax gives the memory address of `i`,
    // i.e. a pointer to `i`. When calling `zeroptr`,
    // it will edit the "original" `i`.
```



```

zeroptr(&i)
fmt.Println("zeroptr:", i) // zeroptr: 0

// Pointers can be printed too.
fmt.Println("pointer:", &i) // pointer: 0x10434114
}

```

[Try this code](#)

Pointer v. Value Methods

Pointer Methods

Pointer methods can be called even if the variable is itself not a pointer.

According to the [Go Spec](#),

... a reference to a non-interface method with a pointer receiver using an addressable value will automatically take the address of that value: `t.Mp` is equivalent to `(&t).Mp`.

You can see this in this example:

```

package main

import "fmt"

type Foo struct {
    Bar int
}

func (f *Foo) Increment() {
    f.Bar += 1
}

func main() {
    var f Foo

    // Calling `f.Increment` is automatically changed to `(&f).Increment` by the compiler.
    f = Foo{}
    fmt.Printf("f.Bar is %d\n", f.Bar)
    f.Increment()
    fmt.Printf("f.Bar is %d\n", f.Bar)

    // As you can see, calling `(&f).Increment` directly does the same thing.
    f = Foo{}
    fmt.Printf("f.Bar is %d\n", f.Bar)
    (&f).Increment()
    fmt.Printf("f.Bar is %d\n", f.Bar)
}

```

[Play it](#)

Value Methods

Similarly to pointer methods, value methods can be called even if the variable is itself not a value.

According to the [Go Spec](#),

... a reference to a non-interface method with a value receiver using a pointer will automatically dereference that pointer: `pt.Mv` is equivalent to `(*pt).Mv`.

You can see this in this example:

```
package main

import "fmt"

type Foo struct {
    Bar int
}

func (f Foo) Increment() {
    f.Bar += 1
}

func main() {
    var p *Foo

    // Calling `p.Increment` is automatically changed to `(*p).Increment` by the compiler.
    // (Note that `*p` is going to remain at 0 because a copy of `*p`, and not the original
    // `*p` are being edited)
    p = &Foo{}
    fmt.Printf("(p).Bar is %d\n", (p).Bar)
    p.Increment()
    fmt.Printf("(p).Bar is %d\n", (p).Bar)

    // As you can see, calling `(*p).Increment` directly does the same thing.
    p = &Foo{}
    fmt.Printf("(p).Bar is %d\n", (p).Bar)
    (*p).Increment()
    fmt.Printf("(p).Bar is %d\n", (p).Bar)
}
```

Play it

To learn more about pointer and value methods, visit the [Go Spec section on Method Values](#), or see the [Effective Go section about Pointers v. Values](#).

*Note 1: The parenthesis `()` around `*p` and `&f` before selectors like `.Bar` are there for grouping purposes, and must be kept.*

Note 2: Although pointers can be converted to values (and vice-versa) when they are the receivers for a method, they are not automatically converted to each other when they are arguments inside of a function.

Dereferencing Pointers

Pointers can be **dereferenced** by adding an asterisk `*` before a pointer.

```
package main

import (
    "fmt"
)

type Person struct {
    Name string
}

func main() {
    c := new(Person) // returns pointer
    c.Name = "Catherine"
    fmt.Println(c.Name) // prints: Catherine
    d := c
    d.Name = "Daniel"
    fmt.Println(c.Name) // prints: Daniel
    // Adding an Asterix before a pointer dereferences the pointer
    i := *d
    i.Name = "Ines"
    fmt.Println(c.Name) // prints: Daniel
    fmt.Println(d.Name) // prints: Daniel
    fmt.Println(i.Name) // prints: Ines
}
```

Slices are Pointers to Array Segments

Slices are **pointers** to arrays, with the length of the segment, and its capacity. They behave as pointers, and assigning their value to another slice, will assign the memory address. To **copy** a slice value to another, use the built-in **copy** function: `func copy(dst, src []Type) int` (returns the amount of items copied).

```
package main

import (
    "fmt"
)

func main() {
    x := []byte{'a', 'b', 'c'}
    fmt.Printf("%s", x) // prints: abc
    y := x
    y[0], y[1], y[2] = 'x', 'y', 'z'
    fmt.Printf("%s", x) // prints: xyz
    z := make([]byte, len(x))
    // To copy the value to another slice, but
    // but not the memory address use copy:
    _ = copy(z, x) // returns count of items copied
    fmt.Printf("%s", z) // prints: xyz
    z[0], z[1], z[2] = 'a', 'b', 'c'
    fmt.Printf("%s", x) // prints: xyz
    fmt.Printf("%s", z) // prints: abc
}
```

Simple Pointers

```
func swap(x, y *int) {  
    *x, *y = *y, *x  
}  
  
func main() {  
    x := int(1)  
    y := int(2)  
    // variable addresses  
    swap(&x, &y)  
    fmt.Println(x, y)  
}
```

Read Pointers online: <https://riptutorial.com/go/topic/1239/pointers>

Chapter 53: Profiling using go tool pprof

Remarks

For more in profiling go programs visit the [go blog](#).

Examples

Basic cpu and memory profiling

Add the following code in you main program.

```
var cpuprofile = flag.String("cpuprofile", "", "write cpu profile `file`")
var memprofile = flag.String("memprofile", "", "write memory profile to `file`")

func main() {
    flag.Parse()
    if *cpuprofile != "" {
        f, err := os.Create(*cpuprofile)
        if err != nil {
            log.Fatal("could not create CPU profile: ", err)
        }
        if err := pprof.StartCPUProfile(f); err != nil {
            log.Fatal("could not start CPU profile: ", err)
        }
        defer pprof.StopCPUProfile()
    }
    ...
    if *memprofile != "" {
        f, err := os.Create(*memprofile)
        if err != nil {
            log.Fatal("could not create memory profile: ", err)
        }
        runtime.GC() // get up-to-date statistics
        if err := pprof.WriteHeapProfile(f); err != nil {
            log.Fatal("could not write memory profile: ", err)
        }
        f.Close()
    }
}
```

after that **build** the go program if added in main `go build main.go`. Run main program with flags defined in code `main.exe -cpuprofile cpu.prof -memprof mem.prof`. If the profiling is done for test cases run the tests with same flags `go test -cpuprofile cpu.prof -memprofile mem.prof`

Basic memory Profiling

```
var memprofile = flag.String("memprofile", "", "write memory profile to `file`")

func main() {
    flag.Parse()
    if *memprofile != "" {
```

```

    f, err := os.Create(*memprofile)
    if err != nil {
        log.Fatal("could not create memory profile: ", err)
    }
    runtime.GC() // get up-to-date statistics
    if err := pprof.WriteHeapProfile(f); err != nil {
        log.Fatal("could not write memory profile: ", err)
    }
    f.Close()
}
}

```

```

go build main.go
main.exe -memprofile mem.prof
go tool pprof main.exe mem.prof

```

Set CPU/Block profile rate

```

// Sets the CPU profiling rate to hz samples per second
// If hz <= 0, SetCPUProfileRate turns off profiling
runtime.SetCPUProfileRate(hz)

// Controls the fraction of goroutine blocking events that are reported in the blocking
// profile
// Rate = 1 includes every blocking event in the profile
// Rate <= 0 turns off profiling
runtime.SetBlockProfileRate(rate)

```

Using Benchmarks to Create Profile

For a non-main packages as well as main, **instead of adding flags inside the code**, write **benchmarks** in the test package , for example:

```

func BenchmarkHello(b *testing.B) {
    for i := 0; i < b.N; i++ {
        fmt.Sprintf("hello")
    }
}

```

Then run the test with the profile flag

```
go test -cpuprofile cpu.prof -bench=.
```

And the benchmarks will be run and create a prof file with filename cpu.prof (in the above example).

Accessing Profile File

once a prof file has been generated, one can access the prof file using **go tools**:

```
go tool pprof cpu.prof
```

This will enter into a command line interface for exploring the `profile`

Common commands include:

```
(pprof) top
```

lists top processes in memory

```
(pprof) peek
```

Lists all processes, use *regex* to narrow search.

```
(pprof) web
```

Opens an graph (in svg format) of the process.

An example of the `top` command:

```
69.29s of 100.84s total (68.71%)
Dropped 176 nodes (cum <= 0.50s)
Showing top 10 nodes out of 73 (cum >= 12.03s)
   flat  flat%   sum%        cum   cum%   runtime.mapaccess1
 12.44s 12.34% 12.34%    27.87s 27.64% runtime.duffcopy
 10.94s 10.85% 23.19%    10.94s 10.85% github.com/tester/test.(*Circle).Draw
   9.45s   9.37% 32.56%    54.61s 54.16% runtime.aeshashbody
   8.88s   8.81% 41.36%     8.88s   8.81% runtime.mapaccess1_fast64
   7.90s   7.83% 49.20%    11.04s 10.95% github.com/tester/test.(*Circle).isCircle
   5.86s   5.81% 55.01%     9.59s   9.51% github.com/tester/test.(*Circle).openCircle
   5.03s   4.99% 60.00%     8.89s   8.82% runtime.aeshash64
   3.14s   3.11% 63.11%     3.14s   3.11% runtime.mallocgc
   3.08s   3.05% 66.16%     7.85s   7.78% runtime.memhash
   2.57s   2.55% 68.71%    12.03s 11.93%
```

Read Profiling using go tool pprof online: <https://riptutorial.com/go/topic/7748/profiling-using-go-tool-pprof>

Chapter 54: Protobuf in Go

Introduction

Protobuf or Protocol Buffer encodes and decodes data so that different applications or modules written in unlike languages can exchange the large number of messages quickly and reliably without overloading the communication channel. With protobuf, the performance is directly proportional to the number of message you tend to send. It compress the message to send in a serialized binary format by providing your the tools to encode the message at source and decode it at the destination.

Remarks

There are two steps of using **protobuf**.

1. First you must compile the protocol buffer definitions
2. Import the above definitions, with the support library into your program.

gRPC Support

If a proto file specifies RPC services, protoc-gen-go can be instructed to generate code compatible with gRPC (<http://www.grpc.io/>). To do this, pass the `plugins` parameter to protoc-gen-go; the usual way is to insert it into the `--go_out` argument to protoc:

```
protoc --go_out=plugins=grpc:. *.proto
```

Examples

Using Protobuf with Go

The message you want to serialize and send that you can include into a file **test.proto**, containing

```
package example;

enum FOO { X = 17; };

message Test {
  required string label = 1;
  optional int32 type = 2 [default=77];
  repeated int64 reps = 3;
  optional group OptionalGroup = 4 {
    required string RequiredField = 5;
  }
}
```

To compile the protocol buffer definition, run protoc with the `--go_out` parameter set to the directory you want to output the Go code to.


```
protoc --go_out=. *.proto
```

To create and play with a Test object from the example package,

```
package main

import (
    "log"

    "github.com/golang/protobuf/proto"
    "path/to/example"
)

func main() {
    test := &example.Test {
        Label: proto.String("hello"),
        Type:  proto.Int32(17),
        Reps:  []int64{1, 2, 3},
        Optionalgroup: &example.Test_OptionalGroup {
            RequiredField: proto.String("good bye"),
        },
    }
    data, err := proto.Marshal(test)
    if err != nil {
        log.Fatal("marshaling error: ", err)
    }
    newTest := &example.Test{}
    err = proto.Unmarshal(data, newTest)
    if err != nil {
        log.Fatal("unmarshaling error: ", err)
    }
    // Now test and newTest contain the same data.
    if test.GetLabel() != newTest.GetLabel() {
        log.Fatalf("data mismatch %q != %q", test.GetLabel(), newTest.GetLabel())
    }
    // etc.
}
```

To pass extra parameters to the plugin, use a comma-separated parameter list separated from the output directory by a colon:

```
protoc --go_out=plugins=grpc,import_path=mypackage:. *.proto
```

Read Protobuf in Go online: <https://riptutorial.com/go/topic/9729/protobuf-in-go>

Chapter 55: Readers

Examples

Using bytes.Reader to read from a string

One implementation of the `io.Reader` interface can be found in the `bytes` package. It allows a byte slice to be used as the source for a Reader. In this example the byte slice is taken from a string, but is more likely to have been read from a file or network connection.

```
message := []byte("Hello, playground")

reader := bytes.NewReader(message)

bs := make([]byte, 5)
n, err := reader.Read(bs)
if err != nil {
    log.Fatal(err)
}

fmt.Printf("Read %d bytes: %s", n, bs)
```

[Go Playground](#)

Read Readers online: <https://riptutorial.com/go/topic/7000/readers>

Chapter 56: Reflection

Remarks

The [reflect docs](#) are a great reference. In general computer programming, **reflection** is ability of a program to **examine** the structure and behavior of **itself** at **runtime**.

Based on its strict `static type` system [Go](#) lang has some rules ([laws of reflection](#))

Examples

Basic reflect.Value Usage

```
import "reflect"

value := reflect.ValueOf(4)

// Interface returns an interface{}-typed value, which can be type-asserted
value.Interface().(int) // 4

// Type gets the reflect.Type, which contains runtime type information about
// this value
value.Type().Name() // int

value.SetInt(5) // panics -- non-pointer/slice/array types are not addressable

x := 4
reflect.ValueOf(&x).Elem().SetInt(5) // works
```

Structs

```
import "reflect"

type S struct {
    A int
    b string
}

func (s *S) String() { return s.b }

s := &S{
    A: 5,
    b: "example",
}

indirect := reflect.ValueOf(s) // effectively a pointer to an S
value := indirect.Elem()       // this is addressable, since we've derefed a pointer

value.FieldByName("A").Interface() // 5
value.Field(2).Interface()          // "example"
```

```

value.NumMethod()    // 0, since String takes a pointer receiver
indirect.NumMethod() // 1

indirect.Method(0).Call([]reflect.Value{}) // "example"
indirect.MethodByName("String").Call([]reflect.Value{}) // "example"

```

Slices

```

import "reflect"

s := []int{1, 2, 3}

value := reflect.ValueOf(s)

value.Len()           // 3
value.Index(0).Interface() // 1
value.Type().Kind()   // reflect.Slice
value.Type().Elem().Name() // int

value.Index(1).CanAddr() // true -- slice elements are addressable
value.Index(1).CanSet()  // true -- and settable
value.Index(1).Set(5)

typ := reflect.SliceOf(reflect.TypeOf("example"))
newS := reflect.MakeSlice(typ, 0, 10) // an empty []string{} with capacity 10

```

reflect.Value.Elem()

```

import "reflect"

// this is effectively a pointer dereference

x := 5
ptr := reflect.ValueOf(&x)
ptr.Type().Name() // *int
ptr.Type().Kind() // reflect.Ptr
ptr.Interface()   // [pointer to x]
ptr.Set(4)        // panic

value := ptr.Elem() // this is a deref
value.Type().Name() // int
value.Type().Kind() // reflect.Int
value.Set(4)        // this works
value.Interface()   // 4

```

Type of value - package "reflect"

reflect.TypeOf can be used to check the type of variables when comparing

```

package main

import (

```

```
    "fmt"  
    "reflect"  
)  
type Data struct {  
    a int  
}  
func main() {  
    s:="hey dude"  
    fmt.Println(reflect.TypeOf(s))  
  
    D := Data{a:5}  
    fmt.Println(reflect.TypeOf(D))  
  
}
```

Output :

string

main.Data

Read Reflection online: <https://riptutorial.com/go/topic/1854/reflection>

Chapter 57: Select and Channels

Introduction

The `select` keyword provides an easy method to work with channels and perform more advanced tasks. It is frequently used for a number of purposes: - Handling timeouts. - When there are multiple channels to read from, the select will randomly read from one channel which has data. - Providing an easy way to define what happens if no data is available on a channel.

Syntax

- `select {}`
- `select { case true: }`
- `select { case incomingData := <-someChannel: }`
- `select { default: }`

Examples

Simple Select Working with Channels

In this example we create a goroutine (a function running in a separate thread) that accepts a `chan` parameter, and simply loops, sending information into the channel each time.

In the `main` we have a `for` loop and a `select`. The `select` will block processing until one of the `case` statements becomes true. Here we have declared two cases; the first is when information comes through the channel, and the other is if no other case occurs, which is known as `default`.

```
// Use of the select statement with channels (no timeouts)
package main

import (
    "fmt"
    "time"
)

// Function that is "chatty"
// Takes a single parameter a channel to send messages down
func chatter(chatChannel chan<- string) {
    // Clean up our channel when we are done.
    // The channel writer should always be the one to close a channel.
    defer close(chatChannel)

    // loop five times and die
    for i := 1; i <= 5; i++ {
        time.Sleep(2 * time.Second) // sleep for 2 seconds
        chatChannel <- fmt.Sprintf("This is pass number %d of chatter", i)
    }
}
```

```
// Our main function
func main() {
    // Create the channel
    chatChannel := make(chan string, 1)

    // start a go routine with chatter (separate, non blocking)
    go chatter(chatChannel)

    // This for loop keeps things going while the chatter is sleeping
    for {
        // select statement will block this thread until one of the two conditions below is
        met
        // because we have a default, we will hit default any time the chatter isn't chatting
        select {
            // anytime the chatter chats, we'll catch it and output it
            case spam, ok := <-chatChannel:
                // Print the string from the channel, unless the channel is closed
                // and we're out of data, in which case exit.
                if ok {
                    fmt.Println(spam)
                } else {
                    fmt.Println("Channel closed, exiting!")
                    return
                }
            default:
                // print a line, then sleep for 1 second.
                fmt.Println("Nothing happened this second.")
                time.Sleep(1 * time.Second)
        }
    }
}
```

[Try it on the Go Playground!](#)

Using select with timeouts

So here, I have removed the `for` loops, and made a **timeout** by adding a second `case` to the `select` that returns after 3 seconds. Because the `select` just waits until ANY case is true, the second `case` fires, and then our script ends, and `chatter()` never even gets a chance to finish.

```
// Use of the select statement with channels, for timeouts, etc.
package main

import (
    "fmt"
    "time"
)

// Function that is "chatty"
//Takes a single parameter a channel to send messages down
func chatter(chatChannel chan<- string) {
    // loop ten times and die
    time.Sleep(5 * time.Second) // sleep for 5 seconds
    chatChannel<- fmt.Sprintf("This is pass number %d of chatter", 1)
}

// out main function
func main() {
```

```

    // Create the channel, it will be taking only strings, no need for a buffer on this
project
chatChannel := make(chan string)
// Clean up our channel when we are done
defer close(chatChannel)

// start a go routine with chatter (separate, no blocking)
go chatter(chatChannel)

// select statement will block this thread until one of the two conditions below is met
// because we have a default, we will hit default any time the chatter isn't chatting
select {
// anytime the chatter chats, we'll catch it and output it
case spam := <-chatChannel:
    fmt.Println(spam)
// if the chatter takes more than 3 seconds to chat, stop waiting
case <-time.After(3 * time.Second):
    fmt.Println("Ain't no time for that!")
}
}

```

Read Select and Channels online: <https://riptutorial.com/go/topic/3539/select-and-channels>

Chapter 58: Send/receive emails

Syntax

- func PlainAuth(identity, username, password, host string) Auth
- func SendMail(addr string, a Auth, from string, to []string, msg []byte) error

Examples

Sending Email with smtp.SendMail()

Sending email is pretty simple in Go. It helps to understand the RFC 822, which specifies the style an email need to be in, the code below sends a RFC 822 compliant email.

```
package main

import (
    "fmt"
    "net/smtp"
)

func main() {
    // user we are authorizing as
    from := "someuser@example.com"

    // use we are sending email to
    to := "otheruser@example.com"

    // server we are authorized to send email through
    host := "mail.example.com"

    // Create the authentication for the SendMail()
    // using PlainText, but other authentication methods are encouraged
    auth := smtp.PlainAuth("", from, "password", host)

    // NOTE: Using the backtick here ` works like a heredoc, which is why all the
    // rest of the lines are forced to the beginning of the line, otherwise the
    // formatting is wrong for the RFC 822 style
    message := `To: "Some User" <someuser@example.com>
From: "Other User" <otheruser@example.com>
Subject: Testing Email From Go!!

This is the message we are sending. That's it!
`

    if err := smtp.SendMail(host+":25", auth, from, []string{to}, []byte(message)); err != nil {
        fmt.Println("Error SendMail: ", err)
        os.Exit(1)
    }
    fmt.Println("Email Sent!")
}
```

The above will send a message like the following:

```
To: "Other User" <otheruser@example.com>  
From: "Some User" <someuser@example.com>  
Subject: Testing Email From Go!!
```

```
This is the message we are sending. That's it!  
.
```

Read Send/receive emails online: <https://riptutorial.com/go/topic/5912/send-receive-emails>

Chapter 59: Slices

Introduction

A slice is a data structure that encapsulates an array so that the programmer can add as many elements as needed without having to worry about memory management. Slices can be cut into sub-slices very efficiently, since the resulting slices all point to the same internal array. Go programmers often take advantage of this to avoid copying arrays, which would typically be done in many other programming languages.

Syntax

- `slice := make([]type, len, cap)` // create a new slice
- `slice = append(slice, item)` // append a item to a slice
- `slice = append(slice, items...)` // append slice of items to a slice
- `len := len(slice)` // get the length of a slice
- `cap := cap(slice)` // get the capacity of a slice
- `elNum := copy(dst, slice)` // copy a the contents of a slice to an other slice

Examples

Appending to slice

```
slice = append(slice, "hello", "world")
```

Adding Two slices together

```
slice1 := []string{"!"}  
slice2 := []string{"Hello", "world"}  
slice := append(slice1, slice2...)
```

[Run in the Go Playground](#)

Removing elements / "Slicing" slices

If you need to remove one or more elements from a slice, or if you need to work with a sub slice of another existing one; you can use the following method.

Following examples uses slice of int, but that works with all type of slice.

So for that, we need a slice, from witch we will remove some elements:

```
slice := []int{1, 2, 3, 4, 5, 6}  
// > [1 2 3 4 5 6]
```

We need also the indexes of elements to remove:

```
// index of first element to remove (corresponding to the '3' in the slice)
var first = 2

// index of last element to remove (corresponding to the '5' in the slice)
var last = 4
```

And so we can "slice" the slice, removing undesired elements:

```
// keeping elements from start to 'first element to remove' (not keeping first to remove),
// removing elements from 'first element to remove' to 'last element to remove'
// and keeping all others elements to the end of the slice
newSlice1 := append(slice[:first], slice[last+1:]...)
// > [1 2 6]

// you can do using directly numbers instead of variables
newSlice2 := append(slice[:2], slice[5:]...)
// > [1 2 6]

// Another way to do the same
newSlice3 := slice[:first + copy(slice[first:], slice[last+1:])]
// > [1 2 6]

// same that newSlice3 with hard coded indexes (without use of variables)
newSlice4 := slice[:2 + copy(slice[2:], slice[5:])]
// > [1 2 6]
```

To remove only one element, just have to put the index of this element as the first AND as the last index to remove, just like that:

```
var indexToRemove = 3
newSlice5 := append(slice[:indexToRemove], slice[indexToRemove+1:]...)
// > [1 2 3 5 6]

// hard-coded version:
newSlice5 := append(slice[:3], slice[4:]...)
// > [1 2 3 5 6]
```

And you can also remove elements from the beginning of the slice:

```
newSlice6 := append(slice[:0], slice[last+1:]...)
// > [6]

// That can be simplified into
newSlice6 := slice[last+1:]
// > [6]
```

You can also removing some elements from the end of the slice:

```
newSlice7 := append(slice[:first], slice[first+1:len(slice)-1]...)
// > [1 2]

// That can be simplified into
newSlice7 := slice[:first]
```

```
// > [1 2]
```

If the new slice have to contains exactly the same elements than the first one, you can use the same thing but with `last := first-1`.
(This can be useful in case of your indexes are previously computed)

Length and Capacity

Slices have both length and capacity. The length of a slice is the number of elements *currently* in the slice, while the capacity is the number of elements the slice *can hold* before needing to be reallocated.

When creating a slice using the built-in `make()` function, you can specify its length, and optionally its capacity. If the capacity is not explicitly specified, it will be the specified length.

```
var s = make([]int, 3, 5) // length 3, capacity 5
```

You can check the length of a slice with the built-in `len()` function:

```
var n = len(s) // n == 3
```

You can check the capacity with the built-in `cap()` function:

```
var c = cap(s) // c == 5
```

Elements created by `make()` are set to the zero value for the element type of the slice:

```
for idx, val := range s {
    fmt.Println(idx, val)
}
// output:
// 0 0
// 1 0
// 2 0
```

[Run it on play.golang.org](https://play.golang.org)

You cannot access elements beyond the length of a slice, even if the index is within capacity:

```
var x = s[3] // panic: runtime error: index out of range
```

However, as long as the capacity exceeds the length, you can append new elements without reallocating:

```
var t = []int{3, 4}
s = append(s, t) // s is now []int{0, 0, 0, 3, 4}
n = len(s) // n == 5
c = cap(s) // c == 5
```

If you append to a slice which lacks the capacity to accept the new elements, the underlying array will be reallocated for you with sufficient capacity:

```
var u = []int{5, 6}
s = append(s, u) // s is now []int{0, 0, 0, 3, 4, 5, 6}
n = len(s) // n == 7
c = cap(s) // c > 5
```

It is, therefore, generally good practice to allocate sufficient capacity when first creating a slice, if you know how much space you'll need, to avoid unnecessary reallocations.

Copying contents from one slice to another slice

If you wish to copy the contents of a slice into an initially empty slice, following steps can be taken to accomplish it-

1. Create the source slice:

```
var sourceSlice []interface{} = []interface{}{"Hello", 5.10, "World", true}
```

2. Create the destination slice, with:

- Length = Length of sourceSlice

```
var destinationSlice []interface{} = make([]interface{}, len(sourceSlice))
```

3. Now that the destination slice's underlying array is big enough to accomodate all the elements of the source slice, we can proceed to copy the elements using the builtin `copy`:

```
copy(destinationSlice, sourceSlice)
```

Creating Slices

Slices are the typical way go programmers store lists of data.

To declare a slice variable use the `[]Type` syntax.

```
var a []int
```

To declare and initialize a slice variable in one line use the `[]Type{values}` syntax.

```
var a []int = []int{3, 1, 4, 1, 5, 9}
```

Another way to initialize a slice is with the `make` function. It three arguments: the `Type` of the slice (or `map`), the `length`, and the `capacity`.

```
a := make([]int, 0, 5)
```

You can add elements to your new slice using `append`.

```
a = append(a, 5)
```

Check the number of elements in your slice using `len`.

```
length := len(a)
```

Check the capacity of your slice using `cap`. The capacity is the number of elements currently allocated to be in memory for the slice. You can always append to a slice at capacity as Go will automatically create a bigger slice for you.

```
capacity := cap(a)
```

You can access elements in a slice using typical indexing syntax.

```
a[0] // Gets the first member of `a`
```

You can also use a `for` loop over slices with `range`. The first variable is the index in the specified array, and the second variable is the value for the index.

```
for index, value := range a {
    fmt.Println("Index: " + index + " Value: " + value) // Prints "Index: 0 Value: 5" (and
    continues until end of slice)
}
```

[Go Playground](#)

Filtering a slice

To filter a slice without allocating a new underlying array:

```
// Our base slice
slice := []int{ 1, 2, 3, 4 }
// Create a zero-length slice with the same underlying array
tmp := slice[:0]

for _, v := range slice {
    if v % 2 == 0 {
        // Append desired values to slice
        tmp = append(tmp, v)
    }
}

// (Optional) Reassign the slice
slice = tmp // [2, 4]
```

Zero value of slice

The zero value of slice is `nil`, which has the length and capacity 0. A `nil` slice has no underlying

array. But there are also non-nil slices of length and capacity 0, like `[]int{}` or `make([]int, 5)[5:]`.

Any type that have nil values can be converted to `nil` slice:

```
s = []int(nil)
```

To test whether a slice is empty, use:

```
if len(s) == 0 {  
    fmt.Printf("s is empty.")  
}
```

Read Slices online: <https://riptutorial.com/go/topic/733/slices>

Chapter 60: SQL

Remarks

For a list of SQL database drivers see the official Go wiki article [SQLDrivers](#).

The SQL drivers are imported and prefixed by `_`, so that they are *only* available to driver.

Examples

Querying

This example is showing how to query a database with `database/sql`, taking as example a MySQL database.

```
package main

import (
    "log"
    "fmt"
    "database/sql"
    _ "github.com/go-sql-driver/mysql"
)

func main() {
    dsn := "mysql_username:CHANGE@tcp(localhost:3306)/dbname"

    db, err := sql.Open("mysql", dsn)
    if err != nil {
        log.Fatal(err)
    }
    defer db.Close()

    rows, err := db.Query("select id, first_name from user limit 10")
    if err != nil {
        log.Fatal(err)
    }
    defer rows.Close()

    for rows.Next() {
        var id int
        var username string
        if err := rows.Scan(&id, &username); err != nil {
            log.Fatal(err)
        }
        fmt.Printf("%d-%s\n", id, username)
    }
}
```

MySQL

To enable MySQL, a database driver is needed. For example github.com/go-sql-driver/mysql.

```
import (
    "database/sql"
    _ "github.com/go-sql-driver/mysql"
)
```

Opening a database

Opening a database is database specific, here there are examples for some databases.

Sqlite 3

```
file := "path/to/file"
db_, err := sql.Open("sqlite3", file)
if err != nil {
    panic(err)
}
```

MySQL

```
dsn := "mysql_username:CHANGEME@tcp(localhost:3306)/dbname"
db, err := sql.Open("mysql", dsn)
if err != nil {
    panic(err)
}
```

MongoDB: connect & insert & remove & update & query

```
package main

import (
    "fmt"
    "time"

    log "github.com/Sirupsen/logrus"
    mgo "gopkg.in/mgo.v2"
    "gopkg.in/mgo.v2/bson"
)

var mongoConn *mgo.Session

type MongoDB_Conn struct {
    Host string `json:"Host"`
    Port string `json:"Port"`
    User string `json:"User"`
    Pass string `json:"Pass"`
    DB    string `json:"DB"`
}

func MongoConn(mdb MongoDB_Conn) (*mgo.Session, string, error) {
    if mongoConn != nil {
        if mongoConn.Ping() == nil {
            return mongoConn, nil
        }
    }
    user := mdb.User
```

```

pass := mdb.Pass
host := mdb.Host
port := mdb.Port
db := mdb.DB
if host == "" || port == "" || db == "" {
    log.Fatal("Host or port or db is nil")
}
url := fmt.Sprintf("mongodb://%s:%s@%s:%s/%s", user, pass, host, port, db)
if user == "" {
    url = fmt.Sprintf("mongodb://%s:%s/%s", host, port, db)
}
mongo, err := mgo.DialWithTimeout(url, 3*time.Second)
if err != nil {
    log.Errorf("Mongo Conn Error: [%v], Mongo ConnUrl: [%v]",
        err, url)
    errTextReturn := fmt.Sprintf("Mongo Conn Error: [%v]", err)
    return &mgo.Session{}, errors.New(errTextReturn)
}
mongoConn = mongo
return mongoConn, nil
}

func MongoInsert(dbName, C string, data interface{}) error {
    mongo, err := MongoConn()
    if err != nil {
        log.Error(err)
        return err
    }
    db := mongo.DB(dbName)
    collection := db.C(C)
    err = collection.Insert(data)
    if err != nil {
        return err
    }
    return nil
}

func MongoRemove(dbName, C string, selector bson.M) error {
    mongo, err := MongoConn()
    if err != nil {
        log.Error(err)
        return err
    }
    db := mongo.DB(dbName)
    collection := db.C(C)
    err = collection.Remove(selector)
    if err != nil {
        return err
    }
    return nil
}

func MongoFind(dbName, C string, query, selector bson.M) ([]interface{}, error) {
    mongo, err := MongoConn()
    if err != nil {
        return nil, err
    }
    db := mongo.DB(dbName)
    collection := db.C(C)
    result := make([]interface{}, 0)
    err = collection.Find(query).Select(selector).All(&result)

```

```
    return result, err
}

func MongoUpdate(dbName, C string, selector bson.M, update interface{}) error {
    mongo, err := MongoConn()
    if err != nil {
        log.Error(err)
        return err
    }
    db := mongo.DB(dbName)
    collection := db.C(C)
    err = collection.Update(selector, update)
    if err != nil {
        return err
    }
    return nil
}
```

Read SQL online: <https://riptutorial.com/go/topic/1273/sql>

Chapter 61: String

Introduction

A string is in effect a read-only slice of bytes. In Go a string literal will always contain a valid UTF-8 representation of its content.

Syntax

- `variableName := "Hello World" // declare a string`
- `variableName := `Hello World` // declare a raw literal string`
- `variableName := "Hello " + "World" // concatenates strings`
- `substring := "Hello World"[0:4] // get a part of the string`
- `letter := "Hello World"[6] // get a character of the string`
- `fmt.Sprintf("%s", "Hello World") // formats a string`

Examples

String type

The `string` type allows you to store text, which is a series of characters. There are multiple ways to create strings. A literal string is created by writing the text between double quotes.

```
text := "Hello World"
```

Because Go strings support UTF-8, the previous example is perfectly valid. Strings hold arbitrary bytes which does not necessarily mean every string will contain valid UTF-8 but string literals will always hold valid UTF-8 sequences.

The zero value of strings is an empty string `""`.

Strings can be concatenated using the `+` operator.

```
text := "Hello " + "World"
```

Strings can also be defined using backticks ```. This creates a raw string literal which means characters won't be escaped.

```
text1 := "Hello\nWorld"  
text2 := `Hello  
World`
```

In the previous example, `text1` escapes the `\n` character which represents a new line while `text2` contains the new line character directly. If you compare `text1 == text2` the result will be `true`.

However, `text2 := `Hello\nWorld`` would not escape the `\n` character which means the string contains the text `Hello\nWorld` without a new line. It would be the equivalent of typing `text1 := "Hello\\nWorld"`.

Formatting text

Package `fmt` implements functions to print and format text using format *verbs*. Verbs are represented with a percent sign.

General verbs:

```
%v    // the value in a default format
      // when printing structs, the plus flag (%+v) adds field names
%#v   // a Go-syntax representation of the value
%T    // a Go-syntax representation of the type of the value
%%    // a literal percent sign; consumes no value
```

Boolean:

```
%t    // the word true or false
```

Integer:

```
%b    // base 2
%c    // the character represented by the corresponding Unicode code point
%d    // base 10
%o    // base 8
%q    // a single-quoted character literal safely escaped with Go syntax.
%x    // base 16, with lower-case letters for a-f
%X    // base 16, with upper-case letters for A-F
%U    // Unicode format: U+1234; same as "U+%04X"
```

Floating-point and complex constituents:

```
%b    // decimalless scientific notation with exponent a power of two,
      // in the manner of strconv.FormatFloat with the 'b' format,
      // e.g. -123456p-78
%e    // scientific notation, e.g. -1.234456e+78
%E    // scientific notation, e.g. -1.234456E+78
%f    // decimal point but no exponent, e.g. 123.456
%F    // synonym for %f
%g    // %e for large exponents, %f otherwise
%G    // %E for large exponents, %F otherwise
```

String and slice of bytes (treated equivalently with these verbs):

```
%s    // the uninterpreted bytes of the string or slice
%q    // a double-quoted string safely escaped with Go syntax
%x    // base 16, lower-case, two characters per byte
%X    // base 16, upper-case, two characters per byte
```

Pointer:

```
%p // base 16 notation, with leading 0x
```

Using the verbs, you can create strings concatenating multiple types:

```
text1 := fmt.Sprintf("Hello %s", "World")
text2 := fmt.Sprintf("%d + %d = %d", 2, 3, 5)
text3 := fmt.Sprintf("%s, %s (Age: %d)", "Obama", "Barack", 55)
```

The function `Sprintf` formats the string in the first parameter replacing the verbs with the value of the values in the next parameters and returns the result. Like `Sprintf`, the function `Printf` also formats but instead of returning the result it prints the string.

strings package

- `strings.Contains`

```
fmt.Println(strings.Contains("foobar", "foo")) // true
fmt.Println(strings.Contains("foobar", "baz")) // false
```

- `strings.HasPrefix`

```
fmt.Println(strings.HasPrefix("foobar", "foo")) // true
fmt.Println(strings.HasPrefix("foobar", "baz")) // false
```

- `strings.HasSuffix`

```
fmt.Println(strings.HasSuffix("foobar", "bar")) // true
fmt.Println(strings.HasSuffix("foobar", "baz")) // false
```

- `strings.Join`

```
ss := []string{"foo", "bar", "bar"}
fmt.Println(strings.Join(ss, ", ")) // foo, bar, bar
```

- `strings.Replace`

```
fmt.Println(strings.Replace("foobar", "bar", "baz", 1)) // foobaz
```

- `strings.Split`

```
s := "foo, bar, bar"
fmt.Println(strings.Split(s, ", ")) // [foo bar bar]
```

- `strings.ToLower`

```
fmt.Println(strings.ToLower("FOOBAR")) // foobar
```

- `strings.ToUpper`

```
fmt.Println(strings.ToUpper("foobar")) // FOOBAR
```

- `strings.TrimSpace`

```
fmt.Println(strings.TrimSpace("  foobar  ")) // foobar
```

More: <https://golang.org/pkg/strings/>.

Read String online: <https://riptutorial.com/go/topic/9666/string>

Chapter 62: Structs

Introduction

Structs are sets of various variables packed together. The struct itself is only a *package* containing variables and making them easily accessible.

Unlike in C, Go's structs can have methods attached to them. It also allows them to implement interfaces. That makes Go's structs similar to objects, but they are (probably intentionally) missing some major features known in object oriented languages like inheritance.

Examples

Basic Declaration

A basic struct is declared as follows:

```
type User struct {
    FirstName, LastName string
    Email                string
    Age                  int
}
```

Each value is called a field. Fields are usually written one per line, with the field's name preceding its type. Consecutive fields of the same type may be combined, as `FirstName` and `LastName` in the above example.

Exported vs. Unexported Fields (Private vs Public)

Struct fields whose names begin with an uppercase letter are exported. All other names are unexported.

```
type Account struct {
    UserID      int    // exported
    accessToken string // unexported
}
```

Unexported fields can only be accessed by code within the same package. As such, if you are ever accessing a field from a *different* package, its name needs to start with an uppercase letter.

```
package main

import "bank"

func main() {
    var x = &bank.Account{
        UserID: 1,           // this works fine
        accessToken: "one", // this does not work, since accessToken is unexported
    }
}
```

```
}  
}
```

However, from within the `bank` package, you can access both `UserId` and `accessToken` without issue.

The package `bank` could be implemented like this:

```
package bank  
  
type Account struct {  
    UserID int  
    accessToken string  
}  
  
func ProcessUser(u *Account) {  
    u.accessToken = doSomething(u) // ProcessUser() can access u.accessToken because  
                                   // it's defined in the same package  
}
```

Composition and Embedding

Composition provides an alternative to inheritance. A struct may include another type by name in its declaration:

```
type Request struct {  
    Resource string  
}  
  
type AuthenticatedRequest struct {  
    Request  
    Username, Password string  
}
```

In the example above, `AuthenticatedRequest` will contain four public members: `Resource`, `Request`, `Username`, and `Password`.

Composite structs can be instantiated and used the same way as normal structs:

```
func main() {  
    ar := new(AuthenticatedRequest)  
    ar.Resource = "example.com/request"  
    ar.Username = "bob"  
    ar.Password = "P@ssw0rd"  
    fmt.Printf("%#v", ar)  
}
```

[play it on playground](#)

Embedding

In the previous example, `Request` is an embedded field. Composition can also be achieved by embedding a different type. This is useful, for example, to decorate a Struct with more functionality. For example, continuing with the `Resource` example, we want a function that formats the content of the `Resource` field to prefix it with `http://` or `https://`. We have two options: create the new methods on `AuthenticatedRequest` or **embed** it from a different struct:

```
type ResourceFormatter struct {}

func(r *ResourceFormatter) FormatHTTP(resource string) string {
    return fmt.Sprintf("http://%s", resource)
}
func(r *ResourceFormatter) FormatHTTPS(resource string) string {
    return fmt.Sprintf("https://%s", resource)
}

type AuthenticatedRequest struct {
    Request
    Username, Password string
    ResourceFormatter
}
```

And now the main function could do the following:

```
func main() {
    ar := new(AuthenticatedRequest)
    ar.Resource = "www.example.com/request"
    ar.Username = "bob"
    ar.Password = "P@ssw0rd"

    println(ar.FormatHTTP(ar.Resource))
    println(ar.FormatHTTPS(ar.Resource))

    fmt.Printf("%#v", ar)
}
```

Look that the `AuthenticatedRequest` that has a `ResourceFormatter` embedded struct.

But the downside of it is that you cannot access objects outside of your composition. So `ResourceFormatter` cannot access members from `AuthenticatedRequest`.

[play it on playground](#)

Methods

Struct methods are very similar to functions:

```
type User struct {
    name string
}

func (u User) Name() string {
    return u.name
}
```

```
func (u *User) SetName(newName string) {
    u.name = newName
}
```

The only difference is the addition of the method receiver. It may be declared either as an instance of the type or a pointer to an instance of the type. Since `SetName()` mutates the instance, the receiver must be a pointer in order to effect a permanent change in the instance.

For example:

```
package main

import "fmt"

type User struct {
    name string
}

func (u User) Name() string {
    return u.name
}

func (u *User) SetName(newName string) {
    u.name = newName
}

func main() {
    var me User

    me.SetName("Slim Shady")
    fmt.Println("My name is", me.Name())
}
```

[Go Playground](#)

Anonymous struct

It is possible to create an anonymous struct:

```
data := struct {
    Number int
    Text   string
} {
    42,
    "Hello world!",
}
```

Full example:

```
package main

import (
    "fmt"
)
```

```
func main() {
    data := struct {Number int; Text string}{42, "Hello world!"} // anonymous struct
    fmt.Printf("%+v\n", data)
}
```

[play it on playground](#)

Tags

Struct fields can have tags associated with them. These tags can be read by the `reflect` package to get custom information specified about a field by the developer.

```
struct Account {
    Username      string `json:"username"`
    DisplayName    string `json:"display_name"`
    FavoriteColor string `json:"favorite_color,omitempty"`
}
```

In the above example, the tags are used to change the key names used by the `encoding/json` package when marshaling or unmarshaling JSON.

While the tag can be any string value, it's considered best practice to use space separated `key:"value"` pairs:

```
struct StructName {
    FieldName int `package1:"customdata,moredata" package2:"info"`
}
```

The struct tags used with the `encoding/xml` and `encoding/json` package are used throughout the standard library.

Making struct copies.

A struct can simply be copied using assignment.

```
type T struct {
    I int
    S string
}

// initialize a struct
t := T{1, "one"}

// make struct copy
u := t // u has its field values equal to t

if u == t { // true
    fmt.Println("u and t are equal") // Prints: "u and t are equal"
}
```

In above case, `'t'` and `'u'` are now separate objects (struct values).

Since `T` does not contain any reference types (slices, map, channels) as its fields, `t` and `u` above can be modified without affecting each other.

```
fmt.Printf("t.I = %d, u.I = %d\n", t.I, u.I) // t.I = 100, u.I = 1
```

However, if `T` contains a reference type, for example:

```
type T struct {
    I    int
    S    string
    xs []int // a slice is a reference type
}
```

Then a simple copy by assignment would copy the value of slice type field as well to the new object. This would result in two different objects referring to the same slice object.

```
// initialize a struct
t := T{I: 1, S: "one", xs: []int{1, 2, 3}}

// make struct copy
u := t // u has its field values equal to t
```

Since both `u` and `t` refer to the same slice through their field `xs` updating a value in the slice of one object would reflect the change in the other.

```
// update a slice field in u
u.xs[1] = 500

fmt.Printf("t.xs = %d, u.xs = %d\n", t.xs, u.xs)
// t.xs = [1 500 3], u.xs = [1 500 3]
```

Hence, extra care must be taken to ensure this reference type property does not produce unintended behavior.

To copy above objects for example, an explicit copy of the slice field could be performed:

```
// explicitly initialize u's slice field
u.xs = make([]int, len(t.xs))
// copy the slice values over from t
copy(u.xs, t.xs)

// updating slice value in u will not affect t
u.xs[1] = 500

fmt.Printf("t.xs = %d, u.xs = %d\n", t.xs, u.xs)
// t.xs = [1 2 3], u.xs = [1 500 3]
```

Struct Literals

A value of a struct type can be written using a *struct literal* that specifies values for its fields.

```
type Point struct { X, Y int }
p := Point{1, 2}
```

The above example specifies every field in the right order. Which is not useful, because programmers have to remember the exact fields in order. More often, a struct can be initialized by listing some or all of the field names and their corresponding values.

```
anim := gif.GIF{LoopCount: nframes}
```

Omitted fields are set to the zero value for its type.

Note: The two forms cannot be mixed in the same literal.

Empty struct

A struct is a sequence of named elements, called fields, each of which has a name and a type. Empty struct has no fields, like this anonymous empty struct:

```
var s struct{}
```

Or like this named empty struct type:

```
type T struct{}
```

The interesting thing about the empty struct is that, its size is zero (try [The Go Playground](#)):

```
fmt.Println(unsafe.Sizeof(s))
```

This prints 0, so the empty struct itself takes no memory. so it is good option for quit channel, like (try [The Go Playground](#)):

```
package main

import (
    "fmt"
    "time"
)

func main() {
    done := make(chan struct{})
    go func() {
        time.Sleep(1 * time.Second)
        close(done)
    }()

    fmt.Println("Wait...")
    <-done
    fmt.Println("done.")
}
```

Read Structs online: <https://riptutorial.com/go/topic/374/structs>

Chapter 63: Templates

Syntax

- `t, err := template.Parse({{.MyName .MyAge}})`
- `t.Execute(os.Stdout,struct{MyValue,MyAge string}{"John Doe","40.1"})`

Remarks

Golang provides packages like:

1. `text/template`
2. `html/template`

to implement data-driven templates for generating textual and HTML outputs.

Examples

Output values of struct variable to Standard Output using a text template

```
package main

import (
    "log"
    "text/template"
    "os"
)

type Person struct{
    MyName string
    MyAge int
}

var myTempContents string= `
This person's name is : {{.MyName}}
And he is {{.MyAge}} years old.
`

func main() {
    t,err := template.New("myTemp").Parse(myTempContents)
    if err != nil{
        log.Fatal(err)
    }
    myPersonSlice := []Person{ {"John Doe",41},{"Peter Parker",17} }
    for _,myPerson := range myPersonSlice{
        t.Execute(os.Stdout,myPerson)
    }
}
```

[Playground](#)

Defining functions for calling from template

```
package main

import (
    "fmt"
    "net/http"
    "os"
    "text/template"
)

var requestTemplate string = `
{{range $i, $url := .URLs}}
{{ $url }} {{(status_code $url)}}
{{ end }}`

type Requests struct {
    URLs []string
}

func main() {
    var fns = template.FuncMap{
        "status_code": func(x string) int {
            resp, err := http.Head(x)
            if err != nil {
                return -1
            }
            return resp.StatusCode
        },
    }

    req := new(Requests)
    req.URLs = []string{"http://godoc.org", "http://stackoverflow.com", "http://linux.org"}

    tmpl := template.Must(template.New("getBatch").Funcs(fns).Parse(requestTemplate))
    err := tmpl.Execute(os.Stdout, req)
    if err != nil {
        fmt.Println(err)
    }
}
```

Here we use our defined function `status_code` to get status code of web page right from template.

Output:

```
http://godoc.org 200
http://stackoverflow.com 200
http://linux.org 200
```

Read Templates online: <https://riptutorial.com/go/topic/1402/templates>

Chapter 64: Testing

Introduction

Go comes with its own testing facilities that has everything needed to run tests and benchmarks. Unlike in most other programming languages, there is often no need for a separate testing framework, although some exist.

Examples

Basic Test

main.go:

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println(Sum(4,5))
}

func Sum(a, b int) int {
    return a + b
}
```

main_test.go:

```
package main

import (
    "testing"
)

// Test methods start with `Test`
func TestSum(t *testing.T) {
    got := Sum(1, 2)
    want := 3
    if got != want {
        t.Errorf("Sum(1, 2) == %d, want %d", got, want)
    }
}
```

To run the test just use the `go test` command:

```
$ go test
ok      test_app    0.005s
```

Use the `-v` flag to see the results of each test:

```
$ go test -v
=== RUN    TestSum
--- PASS: TestSum (0.00s)
PASS
ok      _/tmp      0.000s
```

Use the path `./...` to test subdirectories recursively:

```
$ go test -v ./...
ok      github.com/me/project/dir1    0.008s
=== RUN    TestSum
--- PASS: TestSum (0.00s)
PASS
ok      github.com/me/project/dir2    0.008s
=== RUN    TestDiff
--- PASS: TestDiff (0.00s)
PASS
```

Run a Particular Test:

If there are multiple tests and you want to run a specific test, it can be done like this:

```
go test -v -run=<TestName> // will execute only test with this name
```

Example:

```
go test -v run=TestSum
```

Benchmark tests

If you want to measure benchmarks add a testing method like this:

sum.go:

```
package sum

// Sum calculates the sum of two integers
func Sum(a, b int) int {
    return a + b
}
```

sum_test.go:

```
package sum

import "testing"

func BenchmarkSum(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = Sum(2, 3)
    }
}
```

Then in order to run a simple benchmark:

```
$ go test -bench=.
BenchmarkSum-8      2000000000      0.49 ns/op
ok      so/sum      1.027s
```

Table-driven unit tests

This type of testing is popular technique for testing with predefined input and output values.

Create a file called `main.go` with content:

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println(Sum(4, 5))
}

func Sum(a, b int) int {
    return a + b
}
```

After you run it with, you will see that the output is `9`. Although the `Sum` function looks pretty simple, it is a good idea to test your code. In order to do this, we create another file named `main_test.go` in the same folder as `main.go`, containing the following code:

```
package main

import (
    "testing"
)

// Test methods start with Test
func TestSum(t *testing.T) {
    // Note that the data variable is of type array of anonymous struct,
    // which is very handy for writing table-driven unit tests.
    data := []struct {
        a, b, res int
    }{
        {1, 2, 3},
        {0, 0, 0},
        {1, -1, 0},
        {2, 3, 5},
        {1000, 234, 1234},
    }

    for _, d := range data {
        if got := Sum(d.a, d.b); got != d.res {
            t.Errorf("Sum(%d, %d) == %d, want %d", d.a, d.b, got, d.res)
        }
    }
}
```

As you can see, a slice of anonymous structs is created, each with a set of inputs and the expected result. This allows a large number of test cases to be created all together in one place, then executed in a loop, reducing code repetition and improving clarity.

Example tests (self documenting tests)

This type of tests make sure that your code compiles properly and will appear in the generated documentation for your project. In addition to that, the example tests can assert that your test produces proper output.

sum.go:

```
package sum

// Sum calculates the sum of two integers
func Sum(a, b int) int {
    return a + b
}
```

sum_test.go:

```
package sum

import "fmt"

func ExampleSum() {
    x := Sum(1, 2)
    fmt.Println(x)
    fmt.Println(Sum(-1, -1))
    fmt.Println(Sum(0, 0))

    // Output:
    // 3
    // -2
    // 0
}
```

To execute your test, run `go test` in the folder containing those files OR put those two files in a sub-folder named `sum` and then from the parent folder execute `go test ./sum`. In both cases you will get an output similar to this:

```
ok      so/sum    0.005s
```

If you are wondering how this is testing your code, here is another example function, which actually fails the test:

```
func ExampleSum_fail() {
    x := Sum(1, 2)
    fmt.Println(x)

    // Output:
    // 5
}
```

When you run `go test`, you get the following output:

```
$ go test
--- FAIL: ExampleSum_fail (0.00s)
got:
3
want:
5
FAIL
exit status 1
FAIL    so/sum    0.006s
```

If you want to see the documentation for your `sum` package – just run:

```
go doc -http=:6060
```

and navigate to <http://localhost:6060/pkg/FOLDER/sum/>, where *FOLDER* is the folder containing the `sum` package (in this example `so`). The documentation for the `sum` method looks like this:

Package sum

```
import "so/sum"
```

[Overview](#)

[Index](#)

[Examples](#)

Overview ▼

Package sum is a sample package for test purposes.

Index ▼

```
func Sum(a, b int) int
```

Examples

Sum

Package files

[sum.go](#)

- A `tearDown` function does a rollback.

This is a good option when you can't modify your database and you need to create an object that simulate an object brought of database or need to init a configuration in each test.

A stupid example would be:

```
// Standard numbers map
var numbers map[string]int = map[string]int{"zero": 0, "three": 3}

// TestMain will exec each test, one by one
func TestMain(m *testing.M) {
    // exec setUp function
    setUp("one", 1)
    // exec test and this returns an exit code to pass to os
    retCode := m.Run()
    // exec tearDown function
    tearDown("one")
    // If exit code is distinct of zero,
    // the test will be failed (red)
    os.Exit(retCode)
}

// setUp function, add a number to numbers slice
func setUp(key string, value int) {
    numbers[key] = value
}

// tearDown function, delete a number to numbers slice
func tearDown(key string) {
    delete(numbers, key)
}

// First test
func TestOnePlusOne(t *testing.T) {
    numbers["one"] = numbers["one"] + 1

    if numbers["one"] != 2 {
        t.Error("1 plus 1 = 2, not %v", value)
    }
}

// Second test
func TestOnePlusTwo(t *testing.T) {
    numbers["one"] = numbers["one"] + 2

    if numbers["one"] != 3 {
        t.Error("1 plus 2 = 3, not %v", value)
    }
}
```

Other example would be to prepare database to test and to do rollback

```
// ID of Person will be saved in database
personID := 12345
// Name of Person will be saved in database
personName := "Toni"
```

```

func TestMain(m *testing.M) {
    // You create an Person and you save in database
    setUp(&Person{
        ID:    personID,
        Name:  personName,
        Age:   19,
    })
    retCode := m.Run()
    // When you have executed the test, the Person is deleted from database
    tearDown(personID)
    os.Exit(retCode)
}

func setUp(P *Person) {
    // ...
    db.add(P)
    // ...
}

func tearDown(id int) {
    // ...
    db.delete(id)
    // ...
}

func getPerson(t *testing.T) {
    P := Get(personID)

    if P.Name != personName {
        t.Error("P.Name is %s and it must be Toni", P.Name)
    }
}

```

View code coverage in HTML format

Run `go test` as normal, yet with the `coverprofile` flag. Then use `go tool` to view the results as HTML.

```

go test -coverprofile=c.out
go tool cover -html=c.out

```

Read Testing online: <https://riptutorial.com/go/topic/1234/testing>

Chapter 65: Text + HTML Templating

Examples

Single item template

Note the use of `{{.}}` to output the item within the template.

```
package main

import (
    "fmt"
    "os"
    "text/template"
)

func main() {
    const (
        letter = `Dear {{.}}, How are you?`
    )

    tmpl, err := template.New("letter").Parse(letter)
    if err != nil {
        fmt.Println(err.Error())
    }

    tmpl.Execute(os.Stdout, "Professor Jones")
}
```

Results in:

```
Dear Professor Jones, How are you?
```

Multiple item template

Note the use of `{{range .}}` and `{{end}}` to cycle over the collection.

```
package main

import (
    "fmt"
    "os"
    "text/template"
)

func main() {
    const (
        letter = `Dear {{range .}}{{.}}, {{end}} How are you?`
    )

    tmpl, err := template.New("letter").Parse(letter)
    if err != nil {
        fmt.Println(err.Error())
    }
}
```

```

    }

    tpl.Execute(os.Stdout, []string{"Harry", "Jane", "Lisa", "George"})
}

```

Results in:

```
Dear Harry, Jane, Lisa, George,  How are you?
```

Templates with custom logic

In this example, a function map named `funcMap` is supplied to the template via the `Funcs()` method and then invoked inside the template. Here, the function `increment()` is used to get around the lack of a less than or equal function in the templating language. Note in the output how the final item in the collection is handled.

A `-` at the beginning `{{- or end -}}` is used to trim whitespace and can be used to help make the template more legible.

```

package main

import (
    "fmt"
    "os"
    "text/template"
)

var funcMap = template.FuncMap{
    "increment": increment,
}

func increment(x int) int {
    return x + 1
}

func main() {
    const (
        letter = `Dear {{with $names := .}}
        {{- range $i, $val := $names}}
            {{- if lt (increment $i) (len $names)}}
                {{- $val}}, {{else -}} and {{$val}}{{end}}
            {{- end}}{{end}}; How are you?`
    )

    tpl, err := template.New("letter").Funcs(funcMap).Parse(letter)
    if err != nil {
        fmt.Println(err.Error())
    }

    tpl.Execute(os.Stdout, []string{"Harry", "Jane", "Lisa", "George"})
}

```

Results in:

```
Dear Harry, Jane, Lisa, and George; How are you?
```

Templates with structs

Note how field values are obtained using `{{.FieldName}}`.

```
package main

import (
    "fmt"
    "os"
    "text/template"
)

type Person struct {
    FirstName string
    LastName  string
    Street    string
    City      string
    State     string
    Zip       string
}

func main() {
    const (
        letter = `-----
{{range .}}{{.FirstName}} {{.LastName}}
{{.Street}}
{{.City}}, {{.State}} {{.Zip}}

Dear {{.FirstName}},
    How are you?

-----
{{end}}`
    )

    tmpl, err := template.New("letter").Parse(letter)
    if err != nil {
        fmt.Println(err.Error())
    }

    harry := Person{
        FirstName: "Harry",
        LastName:  "Jones",
        Street:    "1234 Main St.",
        City:      "Springfield",
        State:     "IL",
        Zip:       "12345-6789",
    }

    jane := Person{
        FirstName: "Jane",
        LastName:  "Sherman",
        Street:    "8511 1st Ave.",
        City:      "Dayton",
        State:     "OH",
        Zip:       "18515-6261",
    }

    tmpl.Execute(os.Stdout, []Person{harry, jane})
}
```

Results in:

```
-----  
Harry Jones  
1234 Main St.  
Springfield, IL 12345-6789  
  
Dear Harry,  
    How are you?  
  
-----  
Jane Sherman  
8511 1st Ave.  
Dayton, OH 18515-6261  
  
Dear Jane,  
    How are you?  
  
-----
```

HTML templates

Note the different package import.

```
package main  
  
import (  
    "fmt"  
    "html/template"  
    "os"  
)  
  
type Person struct {  
    FirstName string  
    LastName  string  
    Street    string  
    City      string  
    State     string  
    Zip       string  
    AvatarUrl string  
}  
  
func main() {  
    const (  
        letter = `  
        <html><body><table>  
        <tr><th></th><th>Name</th><th>Address</th></tr>  
        {{range .}}  
        <tr>  
        <td></td>  
        <td>{{.FirstName}} {{.LastName}}</td>  
        <td>{{.Street}}, {{.City}}, {{.State}} {{.Zip}}</td>  
        </tr>  
        {{end}}  
        </table></body></html>`  
    )  
  
    tpl, err := template.New("letter").Parse(letter)  
    if err != nil {
```

```

    fmt.Println(err.Error())
}

harry := Person{
    FirstName: "Harry",
    LastName:  "Jones",
    Street:    "1234 Main St.",
    City:      "Springfield",
    State:     "IL",
    Zip:       "12345-6789",
    AvatarUrl: "harry.png",
}

jane := Person{
    FirstName: "Jane",
    LastName:  "Sherman",
    Street:    "8511 1st Ave.",
    City:      "Dayton",
    State:     "OH",
    Zip:       "18515-6261",
    AvatarUrl: "jane.png",
}

tmpl.Execute(os.Stdout, []Person{harry, jane})
}

```

Results in:

```

<html><body><table>
<tr><th></th><th>Name</th><th>Address</th></tr>

<tr>
<td></td>
<td>Harry Jones</td>
<td>1234 Main St., Springfield, IL 12345-6789</td>
</tr>

<tr>
<td></td>
<td>Jane Sherman</td>
<td>8511 1st Ave., Dayton, OH 18515-6261</td>
</tr>

</table></body></html>

```

How HTML templates prevent malicious code injection

First, here's what can happen when `text/template` is used for HTML. Note Harry's `FirstName` property).

```

package main

import (
    "fmt"
    "html/template"
    "os"
)

```

```

type Person struct {
    FirstName string
    LastName  string
    Street    string
    City      string
    State     string
    Zip       string
    AvatarUrl string
}

func main() {
    const (
        letter = `<body><table>
<tr><th></th><th>Name</th><th>Address</th></tr>
{{range .}}
<tr>
<td></td>
<td>{{.FirstName}} {{.LastName}}</td>
<td>{{.Street}}, {{.City}}, {{.State}} {{.Zip}}</td>
</tr>
{{end}}
</table></body></html>`
    )

    tpl, err := template.New("letter").Parse(letter)
    if err != nil {
        fmt.Println(err.Error())
    }

    harry := Person{
        FirstName: `Harry<script>alert("You've been hacked!")</script>`,
        LastName:   "Jones",
        Street:    "1234 Main St.",
        City:      "Springfield",
        State:     "IL",
        Zip:       "12345-6789",
        AvatarUrl: "harry.png",
    }

    jane := Person{
        FirstName: "Jane",
        LastName:  "Sherman",
        Street:    "8511 1st Ave.",
        City:      "Dayton",
        State:     "OH",
        Zip:       "18515-6261",
        AvatarUrl: "jane.png",
    }

    tpl.Execute(os.Stdout, []Person{harry, jane})
}

```

Results in:

```

<html><body><table>
<tr><th></th><th>Name</th><th>Address</th></tr>

<tr>
<td></td>

```



```

<td>Harry<script>alert("You've been hacked!")</script> Jones</td>
<td>1234 Main St., Springfield, IL 12345-6789</td>
</tr>

<tr>
<td></td>
<td>Jane Sherman</td>
<td>8511 1st Ave., Dayton, OH 18515-6261</td>
</tr>

</table></body></html>

```

The above example, if accessed from a browser, would result in the script being executed and an alert being generated. If, instead, the `html/template` were imported instead of `text/template`, the script would be safely sanitized:

```

<html><body><table>
<tr><th></th><th>Name</th><th>Address</th></tr>

<tr>
<td></td>
<td>Harry<script>alert(&#34;You&#39;ve been hacked!&#34;)&lt;/script> Jones</td>
<td>1234 Main St., Springfield, IL 12345-6789</td>
</tr>

<tr>
<td></td>
<td>Jane Sherman</td>
<td>8511 1st Ave., Dayton, OH 18515-6261</td>
</tr>

</table></body></html>

```

The second result would look garbled when loaded in a browser, but would not result in a potentially malicious script executing.

Read Text + HTML Templating online: <https://riptutorial.com/go/topic/3888/text-plus-html-templating>

Chapter 66: The Go Command

Introduction

The `go` command is a command-line program that allows for the management of Go development. It enables building, running, and testing code, as well as a variety of other Go-related tasks.

Examples

Go Run

`go run` will run a program without creating an executable file. Mostly useful for development. `run` will only execute packages whose *package name* is **main**.

To demonstrate, we will use a simple Hello World example `main.go`:

```
package main

import fmt

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

Execute without compiling to a file:

```
go run main.go
```

Output:

```
Hello, World!
```

Run multiple files in package

If the package is **main** and split into multiple files, one must include the other files in the `run` command:

```
go run main.go assets.go
```

Go Build

`go build` will compile a program into an executable file.

To demonstrate, we will use a simple Hello World example `main.go`:

```
package main

import fmt

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

Compile the program:

```
go build main.go
```

`build` creates an executable program, in this case: `main` or `main.exe`. You can then run this file to see the output `Hello, World!`. You can also copy it to a similar system that doesn't have Go installed, *make it executable*, and run it there.

Specify OS or Architecture in build:

You can specify what system or architecture to build by modifying the `env` before `build`:

```
env GOOS=linux go build main.go # builds for Linux
env GOARCH=arm go build main.go # builds for ARM architecture
```

Build multiple files

If your package is split into multiple files **and** the package name is **main** (that is, *it is not an importable package*), you must specify all the files to build:

```
go build main.go assets.go # outputs an executable: main
```

Building a package

To build a package called `main`, you can simply use:

```
go build . # outputs an executable with name as the name of enclosing folder
```

Go Clean

`go clean` will clean up any temporary files created when invoking `go build` on a program. It will also clean files left over from Makefiles.

Go Fmt

`go fmt` will format a program's source code in a neat, idiomatic way that is easy to read and understand. It is recommended that you use `go fmt` on any source before you submit it for public viewing or committing into a version control system, to make reading it easier.

To format a file:

```
go fmt main.go
```

Or all files in a directory:

```
go fmt myProject
```

You can also use `gofmt -s` (**not** `go fmt`) to attempt to simplify any code that it can.

`gofmt` (**not** `go fmt`) can also be used to refactor code. It understands Go, so it is more powerful than using a simple search and replace. For example, given this program (`main.go`):

```
package main

type Example struct {
    Name string
}

func (e *Example) Original(name string) {
    e.Name = name
}

func main() {
    e := &Example{"Hello"}
    e.Original("Goodbye")
}
```

You can replace the method `Original` with `Refactor` with `gofmt`:

```
gofmt -r 'Original -> Refactor' -d main.go
```

Which will produce the diff:

```
-func (e *Example) Original(name string) {
+func (e *Example) Refactor(name string) {
    e.Name = name
}

func main() {
    e := &Example{"Hello"}
-    e.Original("Goodbye")
+    e.Refactor("Goodbye")
}
```

Go Get

`go get` downloads the packages named by the import paths, along with their dependencies. It then installs the named packages, like 'go install'. Get also accepts build flags to control the installation.

```
go get github.com/maknahar/phonecountry
```

When checking out a new package, `get` creates the target directory `$GOPATH/src/<import-path>`. If the `GOPATH` contains multiple entries, `get` uses the first one. Similarly, it will install compiled binaries in `$GOPATH/bin`.

When checking out or updating a package, `get` looks for a branch or tag that matches the locally installed version of Go. The most important rule is that if the local installation is running version "go1", `get` searches for a branch or tag named "go1". If no such version exists it retrieves the most recent version of the package.

When using `go get`, the `-d` flag causes it to download but not install the given package. The `-u` flag will allow it to update the package and its dependencies.

`Get` never checks out or updates code stored in vendor directories.

Go env

`go env [var ...]` prints go environment information.

By default it prints all the information.

```
$go env
```

```
GOARCH="amd64"
GOBIN=""
GOEXE=""
GOHOSTARCH="amd64"
GOHOSTOS="darwin"
GOOS="darwin"
GOPATH="/Users/vikashkv/work"
GORACE=""
GOROOT="/usr/local/Cellar/go/1.7.4_1/libexec"
GOTOOLDIR="/usr/local/Cellar/go/1.7.4_1/libexec/pkg/tool/darwin_amd64"
CC="clang"
GOGCCFLAGS="-fPIC -m64 -pthread -fno-caret-diagnostics -Qunused-arguments -fmessage-length=0 -fdebug-prefix-map=/var/folders/xf/t3j24fjd2b7bv8c9gdr_0mj80000gn/T/go-build785167995=/tmp/go-build -gno-record-gcc-switches -fno-common"
CXX="clang++"
CGO_ENABLED="1"
```

If one or more variable names is given as arguments, it prints the value of each named variable on its own line.

```
$go env GOOS GOPATH
```

```
darwin
/Users/vikashkv/work
```

Read The Go Command online: <https://riptutorial.com/go/topic/4828/the-go-command>

Chapter 67: Time

Introduction

The Go `time` package provides functionality for measuring and displaying time.

This package provide a structure `time.Time`, allowing to store and do computations on dates and time.

Syntax

- `time.Date(2016, time.December, 31, 23, 59, 59, 999, time.UTC)` // initialize
- `date1 == date2` // returns `true` when the 2 are the same moment
- `date1 != date2` // returns `true` when the 2 are different moment
- `date1.Before(date2)` // returns `true` when the first is strictly before the second
- `date1.After(date2)` // returns `true` when the first is strictly after the second

Examples

Return `time.Time` Zero Value when function has an Error

```
const timeFormat = "15 Monday January 2006"

func ParseDate(s string) (time.Time, error) {
    t, err := time.Parse(timeFormat, s)
    if err != nil {
        // time.Time{} returns January 1, year 1, 00:00:00.000000000 UTC
        // which according to the source code is the zero value for time.Time
        // https://golang.org/src/time/time.go#L23
        return time.Time{}, err
    }
    return t, nil
}
```

Time parsing

If you have a date stored as a string you will need to parse it. Use `time.Parse`.

```
//          time.Parse(    format    , date to parse)
date, err := time.Parse("01/02/2006", "04/08/2017")
if err != nil {
    panic(err)
}

fmt.Println(date)
// Prints 2017-04-08 00:00:00 +0000 UTC
```

The first parameter is the layout in which the string stores the date and the second parameter is

the string that contains the date. `01/02/2006` is the same than saying the format is `MM/DD/YYYY`.

The layout defines the format by showing how the reference time, defined to be `Mon Jan 2 15:04:05 -0700 MST 2006` would be interpreted if it were the value; it serves as an example of the input format. The same interpretation will then be made to the input string.

You can see the constants defined in the time package to know how to write the layout string, but note that the constants are not exported and can't be used outside the time package.

```
const (
    stdLongMonth      // "January"
    stdMonth          // "Jan"
    stdNumMonth       // "1"
    stdZeroMonth      // "01"
    stdLongWeekDay    // "Monday"
    stdWeekDay        // "Mon"
    stdDay            // "2"
    stdUnderDay       // "_2"
    stdZeroDay        // "02"
    stdHour           // "15"
    stdHour12         // "3"
    stdZeroHour12     // "03"
    stdMinute         // "4"
    stdZeroMinute     // "04"
    stdSecond         // "5"
    stdZeroSecond     // "05"
    stdLongYear       // "2006"
    stdYear           // "06"
    stdPM             // "PM"
    stdpm            // "pm"
    stdTZ             // "MST"
    stdISO8601TZ      // "Z0700" // prints Z for UTC
    stdISO8601SecondsTZ // "Z070000"
    stdISO8601ShortTZ // "Z07"
    stdISO8601ColonTZ // "Z07:00" // prints Z for UTC
    stdISO8601ColonSecondsTZ // "Z07:00:00"
    stdNumTZ          // "-0700" // always numeric
    stdNumSecondsTz   // "-070000"
    stdNumShortTZ     // "-07" // always numeric
    stdNumColonTZ     // "-07:00" // always numeric
    stdNumColonSecondsTZ // "-07:00:00"
)
```

Comparing Time

Sometime you will need to know, with 2 dates objects, if there are corresponding to the same date, or find which date is after the other.

In **Go**, there is 4 way to compare dates:

- `date1 == date2`, returns `true` when the 2 are the same moment
- `date1 != date2`, returns `true` when the 2 are different moment
- `date1.Before(date2)`, returns `true` when the first is strictly before the second
- `date1.After(date2)`, returns `true` when the first is strictly after the second

WARNING: When the 2 Time to compare are the same (or correspond to the exact same date), functions `After` and `Before` will return `false`, as a date is neither before nor after itself

- `date1 == date1`, returns `true`
- `date1 != date1`, returns `false`
- `date1.After(date1)`, returns `false`
- `date1.Before(date1)`, returns `false`

TIPS: If you need to know if a date is before or equal another one, just need to combine the 4 operators

- `date1 == date2 && date1.After(date2)`, returns `true` when `date1` is after or equal `date2`
or using `! (date1.Before(date2))`
- `date1 == date2 && date1.Before(date2)`, returns `true` when `date1` is before or equal `date2` or using `! (date1.After(date2))`

Some examples to see how to use:

```
// Init 2 dates for example
var date1 = time.Date(2009, time.November, 10, 23, 0, 0, 0, time.UTC)
var date2 = time.Date(2017, time.July, 25, 16, 22, 42, 123, time.UTC)
var date3 = time.Date(2017, time.July, 25, 16, 22, 42, 123, time.UTC)

bool1 := date1.Before(date2) // true, because date1 is before date2
bool2 := date1.After(date2)  // false, because date1 is not after date2

bool3 := date2.Before(date1) // false, because date2 is not before date1
bool4 := date2.After(date1)  // true, because date2 is after date1

bool5 := date1 == date2 // false, not the same moment
bool6 := date1 == date3 // true, different objects but representing the exact same time

bool7 := date1 != date2 // true, different moments
bool8 := date1 != date3 // false, not different moments

bool9 := date1.After(date3) // false, because date1 is not after date3 (that are the same)
bool10 := date1.Before(date3) // false, because date1 is not before date3 (that are the same)

bool11 := !(date1.Before(date3)) // true, because date1 is not before date3
bool12 := !(date1.After(date3))  // true, because date1 is not after date3
```

Read Time online: <https://riptutorial.com/go/topic/8860/time>

Chapter 68: Type conversions

Examples

Basic Type Conversion

There are two basic styles of type conversion in Go:

```
// Simple type conversion
var x := Foo{}    // x is of type Foo
var y := (Bar)Foo // y is of type Bar, unless Foo cannot be cast to Bar, then compile-time
error occurs.
// Extended type conversion
var z,ok := x.(Bar)    // z is of type Bar, ok is of type bool - if conversion succeeded, z
has the same value as x and ok is true. If it failed, z has the zero value of type Bar, and ok
is false.
```

Testing Interface Implementation

As Go uses implicit interface implementation, you will not get a compile-time error if your struct does not implement an interface you had intended to implement. You can test the implementation explicitly using type casting: `type MyInterface interface { Thing() }`

```
type MyImplementer struct {}

func (m MyImplementer) Thing() {
    fmt.Println("Huzzah!")
}

// Interface is implemented, no error. Variable name _ causes value to be ignored.
var _ MyInterface = (*MyImplementer)nil

type MyNonImplementer struct {}

// Compile-time error - cannot case because interface is not implemented.
var _ MyInterface = (*MyNonImplementer)nil
```

Implement a Unit System with Types

This example illustrates how Go's type system can be used to implement some unit system.

```
package main

import (
    "fmt"
)

type MetersPerSecond float64
type KilometersPerHour float64

func (mps MetersPerSecond) toKilometersPerHour() KilometersPerHour {
```

```
    return KilometersPerHour(mps * 3.6)
}

func (kmh KilometersPerHour) toMetersPerSecond() MetersPerSecond {
    return MetersPerSecond(kmh / 3.6)
}

func main() {
    var mps MetersPerSecond
    mps = 12.5
    kmh := mps.toKilometersPerHour()
    mps2 := kmh.toMetersPerSecond()
    fmt.Printf("%vmmps = %vkmh = %vmmps\n", mps, kmh, mps2)
}
```

[Open in Playground](#)

[Read Type conversions online: https://riptutorial.com/go/topic/2851/type-conversions](#)

Chapter 69: Variables

Syntax

- `var x int` // declare variable x with type int
- `var s string` // declare variable s with type string
- `x = 4` // define x value
- `s = "foo"` // define s value
- `y := 5` // declare and define y inferring its type to int
- `f := 4.5` // declare and define f inferring its type to float64
- `b := "bar"` // declare and define b inferring its type to string

Examples

Basic Variable Declaration

Go is a statically typed language, meaning you generally have to declare the type of the variables you are using.

```
// Basic variable declaration. Declares a variable of type specified on the right.
// The variable is initialized to the zero value of the respective type.
var x int
var s string
var p Person // Assuming type Person struct {}

// Assignment of a value to a variable
x = 3

// Short declaration using := infers the type
y := 4

u := int64(100) // declare variable of type int64 and init with 100
var u2 int64 = 100 // declare variable of type int64 and init with 100
```

Multiple Variable Assignment

In Go, you can declare multiple variables at the same time.

```
// You can declare multiple variables of the same type in one line
var a, b, c string

var d, e string = "Hello", "world!"

// You can also use short declaration to assign multiple variables
x, y, z := 1, 2, 3

foo, bar := 4, "stack" // `foo` is type `int`, `bar` is type `string`
```

If a function returns multiple values, you can also assign values to variables based on the

function's return values.

```
func multipleReturn() (int, int) {
    return 1, 2
}

x, y := multipleReturn() // x = 1, y = 2

func multipleReturn2() (a int, b int) {
    a = 3
    b = 4
    return
}

w, z := multipleReturn2() // w = 3, z = 4
```

Blank Identifier

Go will throw an error when there is a variable that is unused, in order to encourage you to write better code. However, there are some situations when you really don't need to use a value stored in a variable. In those cases, you use a "blank identifier" `_` to assign and discard the assigned value.

A blank identifier can be assigned a value of any type, and is most commonly used in functions that return multiple values.

Multiple Return Values

```
func SumProduct(a, b int) (int, int) {
    return a+b, a*b
}

func main() {
    // I only want the sum, but not the product
    sum, _ := SumProduct(1,2) // the product gets discarded
    fmt.Println(sum) // prints 3
}
```

Using `range`

```
func main() {

    pets := []string{"dog", "cat", "fish"}

    // Range returns both the current index and value
    // but sometimes you may only want to use the value
    for _, pet := range pets {
        fmt.Println(pet)
    }

}
```

Checking a variable's type

There are some situations where you won't be sure what type a variable is when it is returned from a function. You can always check a variable's type by using `var.(type)` if you are unsure what type it is:

```
x := someFunction() // Some value of an unknown type is stored in x now

switch x := x.(type) {
    case bool:
        fmt.Printf("boolean %t\n", x)           // x has type bool
    case int:
        fmt.Printf("integer %d\n", x)           // x has type int
    case string:
        fmt.Printf("pointer to boolean %s\n", x) // x has type string
    default:
        fmt.Printf("unexpected type %T\n", x)    // %T prints whatever type x is
}
```

Read Variables online: <https://riptutorial.com/go/topic/674/variables>

Chapter 70: Vendoring

Remarks

Vendoring is a method of ensuring that all of your 3rd party packages that you use in your Go project are consistent for everyone who develops for your application.

When your Go package imports another package, the compiler normally checks `$GOPATH/src/` for the path of the imported project. However if your package contains a folder named `vendor`, the compiler will check in that folder *first*. This means that you can import other parties packages inside your own code repository, without having to modify their code.

Vendoring is a standard feature in Go 1.6 and up. In Go 1.5, you need to set the environment variable of `GO15VENDOREXPERIMENT=1` to enable vendoring.

Examples

Use govendor to add external packages

[Govendor](#) is a tool that is used to import 3rd party packages into your code repository in a way that is compatible with golang's vendoring.

Say for example that you are using a 3rd party package `bosun.org/slog`:

```
package main

import "bosun.org/slog"

func main() {
    slog.Infof("Hello World")
}
```

Your directory structure might look like:

```
$GOPATH/src/
├─ github.com/me/helloworld/
│   └─ hello.go
├─ bosun.org/slog/
│   └─ ... (slog files)
```

However someone who clones `github.com/me/helloworld` may not have a `$GOPATH/src/bosun.org/slog/` folder, causing *their* build to fail due to missing packages.

Running the following command at your command prompt will grab all the external packages from your Go package and package the required bits into a vendor folder:

```
govendor add +e
```

This instructs govendor to add all of the external packages into your current repository.

Your application's directory structure would now look like:

```
$GOPATH/src/  
├─ github.com/me/helloworld/  
│   └─ vendor/  
│       └─ bosun.org/slog/  
│           └─ ... (slog files)  
└─ hello.go
```

and those who clone your repository will also grab the required 3rd party packages.

Using trash to manage ./vendor

`trash` is a minimalistic vendoring tool that you configure with `vendor.conf` file. This example is for `trash` itself:

```
# package  
github.com/rancher/trash  
  
github.com/Sirupsen/logrus          v0.10.0  
github.com/urfave/cli                v1.18.0  
github.com/cloudfoundry-incubator/candiedyaml 99c3df8  
https://github.com/imikushin/candiedyaml.git  
github.com/stretchr/testify          v1.1.3  
github.com/davecgh/go-spew           5215b55  
github.com/pmezard/go-difflib        792786c  
golang.org/x/sys                     a408501
```

The first non-comment line is the package we're managing `./vendor` for (note: this can be literally any package in your project, not just the root one).

Commented lines begin with `#`.

Each non-empty and non-comment line lists one dependency. Only the "root" package of the dependency needs to be listed.

After the package name goes the version (commit, tag or branch) and optionally the package repository URL (by default, it's inferred from the package name).

To populate your `./vendor` dir, you need to have `vendor.conf` file in the current dir and just run:

```
$ trash
```

Trash will clone the vendored libraries into `~/.trash-cache` (by default), checkout requested versions, copy the files into `./vendor` dir and **prune non-imported packages and test files**. This last step keeps your `./vendor` lean and mean and helps save space in your project repo.

Note: as of v0.2.5 `trash` is available for Linux and macOS, and only supports git to retrieve packages, as git's the most popular one, but we're working on adding all the others that `go get` supports.

Use golang/dep

[golang/dep](#) is a prototype dependency management tool. Soon to be an official versioning tool. Current status **Alpha**.

Usage

Get the tool via

```
$ go get -u github.com/golang/dep/...
```

Typical usage on a new repo might be

```
$ dep init
$ dep ensure -update
```

To update a dependency to a new version, you might run

```
$ dep ensure github.com/pkg/errors@^0.8.0
```

Note that the manifest and lock file formats **have now been finalized**. These will remain compatible even as the tool changes.

vendor.json using Govendor tool

```
# It creates vendor folder and vendor.json inside it
govendor init

# Add dependencies in vendor.json
govendor fetch <dependency>

# Usage on new repository
# fetch dependencies in vendor.json
govendor sync
```

Example vendor.json

```
{
  "comment": "",
  "ignore": "test",
  "package": [
    {
      "checksumSHA1": "kBeNcaKk56FguvPSUCEaH6AxpRc=",
      "path": "github.com/golang/protobuf/proto",
      "revision": "2bba0603135d7d7f5cb73b2125beeda19c09f4ef",
      "revisionTime": "2017-03-31T03:19:02Z"
    },
    {
      "checksumSHA1": "1DRAxd1WzS4U0xKN/yQ/fdNN7f0=",
      "path": "github.com/syndtr/goleveldb/leveldb/errors",

```



```
        "revision": "8c81ea47d4c41a385645e133e15510fc6a2a74b4",  
        "revisionTime": "2017-04-09T01:48:31Z"  
    },  
    ],  
    "rootPath": "github.com/sample"  
}
```

Read Vending online: <https://riptutorial.com/go/topic/978/vending>

Chapter 71: Worker Pools

Examples

Simple worker pool

A simple worker pool implementation:

```
package main

import (
    "fmt"
    "sync"
)

type job struct {
    // some fields for your job type
}

type result struct {
    // some fields for your result type
}

func worker(jobs <-chan job, results chan<- result) {
    for j := range jobs {
        var r result
        // do some work
        results <- r
    }
}

func main() {
    // make our channels for communicating work and results
    jobs := make(chan job, 100) // 100 was chosen arbitrarily
    results := make(chan result, 100)

    // spin up workers and use a sync.WaitGroup to indicate completion
    wg := sync.WaitGroup
    for i := 0; i < runtime.NumCPU; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            worker(jobs, results)
        }()
    }

    // wait on the workers to finish and close the result channel
    // to signal downstream that all work is done
    go func() {
        defer close(results)
        wg.Wait()
    }()

    // start sending jobs
    go func() {
        defer close(jobs)
```

```

    for {
        jobs <- getJob()    // I haven't defined getJob() and noMoreJobs()
        if noMoreJobs() {  // they are just for illustration
            break
        }
    }
}()

// read all the results
for r := range results {
    fmt.Println(r)
}
}

```

Job Queue with Worker Pool

A job queue that maintains a worker pool, useful for doing things like background processing in web servers:

```

package main

import (
    "fmt"
    "runtime"
    "strconv"
    "sync"
    "time"
)

// Job - interface for job processing
type Job interface {
    Process()
}

// Worker - the worker threads that actually process the jobs
type Worker struct {
    done          sync.WaitGroup
    readyPool     chan chan Job
    assignedJobQueue chan Job

    quit chan bool
}

// JobQueue - a queue for enqueueing jobs to be processed
type JobQueue struct {
    internalQueue     chan Job
    readyPool         chan chan Job
    workers           []*Worker
    dispatcherStopped sync.WaitGroup
    workersStopped    sync.WaitGroup
    quit              chan bool
}

// NewJobQueue - creates a new job queue
func NewJobQueue(maxWorkers int) *JobQueue {
    workersStopped := sync.WaitGroup{}
    readyPool := make(chan chan Job, maxWorkers)
    workers := make([]*Worker, maxWorkers, maxWorkers)
    for i := 0; i < maxWorkers; i++ {

```

```

    workers[i] = NewWorker(readyPool, workersStopped)
}
return &JobQueue{
    internalQueue:    make(chan Job),
    readyPool:        readyPool,
    workers:          workers,
    dispatcherStopped: sync.WaitGroup{},
    workersStopped:    workersStopped,
    quit:              make(chan bool),
}
}

// Start - starts the worker routines and dispatcher routine
func (q *JobQueue) Start() {
    for i := 0; i < len(q.workers); i++ {
        q.workers[i].Start()
    }
    go q.dispatch()
}

// Stop - stops the workers and dispatcher routine
func (q *JobQueue) Stop() {
    q.quit <- true
    q.dispatcherStopped.Wait()
}

func (q *JobQueue) dispatch() {
    q.dispatcherStopped.Add(1)
    for {
        select {
        case job := <-q.internalQueue: // We got something in on our queue
            workerChannel := <-q.readyPool // Check out an available worker
            workerChannel <- job           // Send the request to the channel
        case <-q.quit:
            for i := 0; i < len(q.workers); i++ {
                q.workers[i].Stop()
            }
            q.workersStopped.Wait()
            q.dispatcherStopped.Done()
            return
        }
    }
}

// Submit - adds a new job to be processed
func (q *JobQueue) Submit(job Job) {
    q.internalQueue <- job
}

// NewWorker - creates a new worker
func NewWorker(readyPool chan chan Job, done sync.WaitGroup) *Worker {
    return &Worker{
        done:        done,
        readyPool:    readyPool,
        assignedJobQueue: make(chan Job),
        quit:          make(chan bool),
    }
}

// Start - begins the job processing loop for the worker
func (w *Worker) Start() {

```

```

go func() {
    w.done.Add(1)
    for {
        w.readyPool <- w.assignedJobQueue // check the job queue in
        select {
            case job := <-w.assignedJobQueue: // see if anything has been assigned to the queue
                job.Process()
            case <-w.quit:
                w.done.Done()
                return
        }
    }
}()

// Stop - stops the worker
func (w *Worker) Stop() {
    w.quit <- true
}

////////// Example //////////

// TestJob - holds only an ID to show state
type TestJob struct {
    ID string
}

// Process - test process function
func (t *TestJob) Process() {
    fmt.Printf("Processing job '%s'\n", t.ID)
    time.Sleep(1 * time.Second)
}

func main() {
    queue := NewJobQueue(runtime.NumCPU())
    queue.Start()
    defer queue.Stop()

    for i := 0; i < 4*runtime.NumCPU(); i++ {
        queue.Submit(&TestJob{strconv.Itoa(i)})
    }
}

```

Read Worker Pools online: <https://riptutorial.com/go/topic/4182/worker-pools>

Chapter 72: XML

Remarks

While many uses of the `encoding/xml` package include marshaling and unmarshaling to a Go `struct`, it's worth noting that this is not a direct mapping. The package documentation states:

Mapping between XML elements and data structures is inherently flawed: an XML element is an order-dependent collection of anonymous values, while a data structure is an order-independent collection of named values.

For simple, unordered, key-value pairs, using a different encoding such as Gob's or `JSON` may be a better fit. For ordered data or event / callback based streams of data, XML may be the best choice.

Examples

Basic decoding / unmarshalling of nested elements with data

XML elements often nest, have data in attributes and/or as character data. The way to capture this data is by using `,attr` and `,chardata` respectively for those cases.

```
var doc = `  
<parent>  
  <child1 attr1="attribute one"/>  
  <child2>and some cdata</child2>  
</parent>  
`  
  
type parent struct {  
    Child1 child1 `xml:"child1"`  
    Child2 child2 `xml:"child2"`  
}  
  
type child1 struct {  
    Attr1 string `xml:"attr1,attr"`  
}  
  
type child2 struct {  
    Cdata1 string `xml:",cdata"`  
}  
  
func main() {  
    var obj parent  
    err := xml.Unmarshal([]byte(doc), &obj)  
    if err != nil {  
        log.Fatal(err)  
    }  
  
    fmt.Println(obj.Child2.Cdata1)  
}
```

Playground

Read XML online: <https://riptutorial.com/go/topic/1846/xml>

Chapter 73: YAML

Examples

Creating a config file in YAML format

```
import (
    "io/ioutil"
    "path/filepath"

    "gopkg.in/yaml.v2"
)

func main() {
    filename, _ := filepath.Abs("config/config.yml")
    yamlFile, err := ioutil.ReadFile(filename)
    var config Config
    err = yaml.Unmarshal(yamlFile, &config)
    if err != nil {
        panic(err)
    }
    //env can be accessed from config.Env
}

type Config struct {
    Env string `yaml:"env"`
}

//config.yml should be placed in config/config.yml for example, and needs to have the
following line for the above example:
//env: test
```

Read YAML online: <https://riptutorial.com/go/topic/2503/yaml>

Chapter 74: Zero values

Remarks

One thing to note - types that have a non-nil zero value like strings, ints, floats, bools and structs can't be set to nil.

Examples

Basic Zero Values

Variables in Go are always initialized whether you give them a starting value or not. Each type, including custom types, has a zero value they are set to if not given a value.

```
var myString string      // "" - an empty string
var myInt int64          // 0 - applies to all types of int and uint
var myFloat float64     // 0.0 - applies to all types of float and complex
var myBool bool         // false
var myPointer *string   // nil
var myInter interface{} // nil
```

This also applies to maps, slices, channels and function types. These types will initialize to nil. In arrays, each element is initialized to the zero value of its respective type.

More Complex Zero Values

In slices the zero value is an empty slice.

```
var myIntSlice []int    // [] - an empty slice
```

Use `make` to create a slice populated with values, any values created in the slice are set to the zero value of the type of the slice. For instance:

```
myIntSlice := make([]int, 5)    // [0, 0, 0, 0, 0] - a slice with 5 zeroes
fmt.Println(myIntSlice[3])
// Prints 0
```

In this example, `myIntSlice` is a `int` slice that contains 5 elements which are all 0 because that's the zero value for the type `int`.

You can also create a slice with `new`, this will create a pointer to a slice.

```
myIntSlice := new([]int)       // &[] - a pointer to an empty slice
*myIntSlice = make([]int, 5)   // [0, 0, 0, 0, 0] - a slice with 5 zeroes
fmt.Println((*myIntSlice)[3])
// Prints 0
```

Note: Slice pointers don't support indexing so you can't access the values using `myIntSlice[3]`, instead you need to do it like `(*myIntSlice)[3]`.

Struct Zero Values

When creating a struct without initializing it, each field of the struct is initialized to its respective zero value.

```
type ZeroStruct struct {
    myString string
    myInt     int64
    myBool    bool
}

func main() {
    var myZero = ZeroStruct{}
    fmt.Printf("Zero values are: %q, %d, %t\n", myZero.myString, myZero.myInt, myZero.myBool)
    // Prints "Zero values are: "", 0, false"
}
```

Array Zero Values

As per the [Go blog](#):

Arrays do not need to be initialized explicitly; the zero value of an array is a ready-to-use array whose elements are themselves zeroed

For example, `myIntArray` is initialized with the zero value of `int`, which is 0:

```
var myIntArray [5]int // an array of five 0's: [0, 0, 0, 0, 0]
```

Read Zero values online: <https://riptutorial.com/go/topic/6069/zero-values>

Chapter 75: Zero values

Examples

Explanation

Zero values or zero initialization are simple to implement. Coming from languages like Java it may seem complicated that some values can be `nil` while others are not. In summary from [Zero Value: The Go Programming Language Specification](#):

Pointers, functions, interfaces, slices, channels, and maps are the only types that can be `nil`. The rest are initialized to false, zero, or empty strings based on their respective types.

If a functions that checks some condition, problems may arise:

```
func isAlive() bool {  
    //Not implemented yet  
    return false  
}
```

The zero value will be false even before implementation. Unit tests dependant on the return of this function could be giving false positives/negatives.

A typical workaround is to also return an error, which is idiomatic in Go:

```
package main  
  
import "fmt"  
  
func isAlive() (bool, error) {  
    //Not implemented yet  
    return false, fmt.Errorf("Not implemented yet")  
}  
  
func main() {  
    _, err := isAlive()  
    if err != nil {  
        fmt.Printf("ERR: %s\n", err.Error())  
    }  
}
```

[play it on playground](#)

When returning both a struct and an error you need a User structure for return, which is not very elegant. There are two counter-options:

- Work with interfaces: Return `nil` by returning an interface.
- Work with pointers: A pointer **can** be `nil`

For example, the following code returns a pointer:

```
func(d *DB) GetUser(id uint64) (*User, error) {  
    //Some error occurred  
    return nil, err  
}
```

Read Zero values online: <https://riptutorial.com/go/topic/6379/zero-values>

Credits

S. No	Chapters	Contributors
1	Getting started with Go	4444 , alejosocorro , Alexander , Amitay Stern , Andrej Bencic , Andrii Abramov , burfl , Burhan Ali , cat , Cody Gustafson , Community , David G. , Dmitri Goldring , Feckmore , Florian Hämmerle , Franck Dernoncourt , Gerep , Greg Bray , hellyale , Hunter , James Taylor , Jared Hooper , Jon Chan , Katamaritaco , Mark Henderson , Matt , mbb , MegaTom , mmlb , mnoronha , mohan08p , Nir , nix , nouney , patterns , Pavel Nikolov , ProfNandaa , Quentin Skousen , Radouane ROUFID , Rahul Nair , RamenChef , raulsntos , Sam Whited , seriousdev , Simone Carletti , skunkmb , sztanpet , Tanmay Garg , Topo , Unapiedra , Vikash , Xavier Nicollet
2	Arrays	NatNGs , nouney , Noval Agung Prayogo , Sam Whited
3	Base64 Encoding	Nathan Osman , RamenChef , Sam Whited
4	Best practices on project structure	Iman Tumorang
5	Branching	burfl , Community , ganesh kumar , Ingve , nk2ge5k
6	Build Constraints	4444 , RamenChef , Sam Whited , seriousdev
7	cgo	MaC , Vojtech Kane
8	Channels	Chris Lucas , Howl , Jeremy , Kwartz , metmirr , RamenChef , Rodolfo Carvalho , Zoyd
9	Closures	abhink
10	Concurrency	Chris Lucas , Community , Florian Hämmerle , flyingfinger , Grzegorz Żur , Harshal Sheth , Ilya , Inanc Gumus , Kyle Brandt , Nathan Osman , Roland Illig , Ryan Kelln , Tim S. Van Haren , VonC , zianwar , Zoyd
11	Console I/O	Abhilekh Singh
12	Constants	Pavel Nikolov , RamenChef , Sam Whited , Simone Carletti
13	Context	Ingaz , Sam Whited
14	Cross Compilation	Jordan , Katamaritaco , mbb , mohan08p , RamenChef , Riley Guerin , SH' , Siu Ching Pong -Asuka Kenji- ,

		SommerEngineering, sztanpet, Zoyd
15	Cryptography	SommerEngineering
16	Defer	abhink, Adrian, Sam Whited, Vikash
17	Developing for Multiple Platforms with Conditional Compiling	ecem
18	Error Handling	browsersenior, elevine, Elijah Sarver, Florian Hämmerle, groob, Ingve, Joe, Kin, Paul Hankin, Quentin Skousen, Sam Whited, Simone Carletti, Sridhar, Surreal Dreams, Vervious, Zoyd
19	Executing Commands	Krzysztof Kowalczyk, Kyle Brandt, Nevermore
20	File I/O	1lann, Andres Kütt, greatwolf, Grzegorz Żur, koblas, noisewaterphd, Quentin Skousen, Sam Whited
21	Fmt	Lanzafame, Nevermore, Sam Whited
22	Functions	Boris Le Méec, Dmytro Sadovnychi, Grzegorz Żur, jayantS, LeoTao, Nathan Osman, nouney, palestamp, RamenChef, Right leg, Thomas Gerot
23	Getting Started With Go Using Atom	Ali M, Danny Chen, Katamaritaco
24	gob	zola
25	Goroutines	mohan08p
26	HTTP Client	1lann, dmportella, Lanzafame, Sam Whited, SommerEngineering
27	HTTP Server	Chief, frigo americain, Jon Erickson, Kin, Nathan Osman, rogerdpack, Sam Whited, Sascha, seriousdev, Simone Carletti, SommerEngineering, Tanmay Garg, Zhinkk
28	Images	putu
29	Inline Expansion	Sam Whited
30	Installation	sadlil
31	Interfaces	Cody Roseborough, dotctor, Francis Norton, Grzegorz Żur, icza, Ingve, meysam, Mike, ptman, sadlil, Sam Whited, Wendy Adi

32	Iota	4444 , Florian Hämmerle , Ingve , mohan08p , Sam Whited , Wojciech Kazior , Zoyd
33	JSON	Dmitry Udod , Joe , Jon Chan , Kyle Brandt , Nathan Osman , RamenChef , Sam Whited , shayan , Simone Carletti , sztanpet , Tanmay Garg , Utahcon
34	JWT Authorization in Go	AniSkywalker
35	Logging	Grzegorz Żur , Jon Chan , Nathan Osman , Pavel Kazhevets , Sam Whited
36	Loops	1lann , burfl , Community , ivan73 , jayantS , Jon Chan , mgh , MohamedAlaa , RamenChef , Sam Whited , Steven Maude , Thomas Gerot
37	Maps	Abhay , abhink , Amitay Stern , Brendan , burfl , chowey , Chris Lucas , cizixs , Community , creker , Dair , Dmitri Goldring , gbulmer , Hugo , James , JepZ , Joe , Kaedys , Kamil Kisiel , Kyle Brandt , Mark Henderson , matt.s , Milo Christiansen , NatNgs , Oleg Sklyar , radbrawler , RamenChef , Roland Illig , Sam Whited , seh , Simone Carletti , skunkmb , Surreal Dreams , Vojtech Kane , Zoyd , Zyerah
38	Memory pooling	Elijah Sarver , Grzegorz Żur , Kenny Grant
39	Methods	ganesh kumar , Pavel Kazhevets
40	mgo	Florian Hämmerle , Sourabh
41	Middleware	Ankit Deshpande
42	Mutex	Adrian , Prutswonder
43	Object Oriented Programming	Davyd Dzhahaiev , Sam Whited , zola
44	OS Signals	Community , Sam Whited , Utahcon
45	Packages	dimportella , Grzegorz Żur , icza , Michael , Nathan Osman , RadicalFish , RamenChef , skunkmb , tkausi
46	Panic and Recover	JunLe Meng , Kaedys , Kristoffer Sall-Storgaard , Sam Whited
47	Parsing Command Line Arguments And Flags	Ingve , Pavel Kazhevets , Sam Whited
48	Parsing CSV files	Ainar-G

49	Plugin	Sam Whited
50	Pointers	David Hoelzer , Jon Chan , Joost , Mal Curtis , metmirr , Nevermore , skunkmb
51	Profiling using go tool pprof	mbb , Nevermore , radbrawler
52	Protobuf in Go	mohan08p
53	Readers	Mike Houston
54	Reflection	ganesh kumar , mammothbane , radbrawler
55	Select and Channels	Harshal Sheth , Kaedys , RamenChef , Sam Whited , Utahcon
56	Send/receive emails	Utahcon
57	Slices	1lann , Benjamin Kadish , burfl , cizixs , Grzegorz Żur , Guillaume , Jared Hooper , Joost , Jukurpa , Kyle Brandt , Mark Henderson , NatNgs , RamenChef , Simone Carletti , skunkmb , Tanmay Garg , Zoyd
58	SQL	Adrian , artamonovdev , bernardn , Francesco Pasa , Nevermore , Sam Whited , Sascha , Tanmay Garg , wrfly
59	String	Ainar-G , NatNgs , raulsntos
60	Structs	abhink , Amitay Stern , Anthony Atkinson , Blixt , burfl , cizixs , Community , FredMaggiowski , Howl , Ingve , Kin , MaC , Mark Henderson , matt.s , mohan08p , Nathan Osman , nouney , Patrick , Quentin Skousen , radbrawler , RamenChef , Roland Illig , Simone Carletti , sunkuet02 , Vojtech Kane , Wojciech Kazior
61	Templates	Pavel Kazhevets , RamenChef , Tanmay Garg
62	Testing	Adrian , Ankit Deshpande , Harshal Sheth , ivan.sim , Jared Ririe , Nathan Osman , Omid , Pavel Nikolov , Rodolfo Carvalho , seriousdev , Toni Villena , Zoyd
63	Text + HTML Templating	Stephen Rudolph
64	The Go Command	ganesh kumar , Harshal Sheth , Ingve , Lanzafame , Mayank Patel , Nevermore , Quentin Skousen , Sam Whited , theflametrooper , Vikash
65	Time	Lanzafame , NatNgs , raulsntos
66	Type conversions	Adrian , Florian Hämmerle

67	Variables	Community , FredMaggiowski , Jon Chan , Simone Carletti
68	Vendoring	Abhilekh Singh , Boris Le Méec , burfl , Dmitri Goldring , Ivan Mikushin , Mark Henderson , Martin Campbell , Michael , Sam Whited , Vardius
69	Worker Pools	burfl , photoionized , seriousdev
70	XML	ivarg , Sam Whited
71	YAML	Nathan Osman , Orr , Sam Whited
72	Zero values	Harshal Sheth , raulsntos , Surreal Dreams