Kostenloses eBook

LERNEN Go

Free unaffiliated eBook created from
Stack Overflow contributors.

#go

Inhaltsverzeichnis

Über

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.

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

Kapitel 1: Erste Schritte mit Go

Bemerkungen

Go ist eine kompilierte, statisch typisierte Open-Source-Sprache in der Tradition von Algol und C. Sie verfügt über Funktionen wie Speicherbereinigung, begrenzte strukturelle Typisierung, Speichersicherheitsfunktionen und benutzerfreundliche CSP- Stil-Programmierung.

Versionen

Die neueste Hauptversion ist unten in Fettdruck dargestellt. Den vollständigen Versionsverlauf finden Sie hier .

Ausführung	Veröffentlichungsdatum
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

Hallo Welt!

`hello.go` den folgenden Code in einen Dateinamen `hello.go` :

```
package main

import "fmt"

func main() {
```

```
    fmt.Println("Hello, □□")
}
```

Spielplatz

Wenn Go korrekt installiert ist , kann dieses Programm folgendermaßen kompiliert und ausgeführt werden:

```
go run hello.go
```

Ausgabe:

```
Hello, □□
```

Wenn Sie mit dem Code zufrieden sind, können Sie ihn zu einer ausführbaren Datei kompilieren, indem Sie Folgendes ausführen:

```
go build hello.go
```

Dadurch wird im aktuellen Verzeichnis eine für Ihr Betriebssystem geeignete ausführbare Datei erstellt, die Sie dann mit dem folgenden Befehl ausführen können:

Linux, OSX und andere Unix-ähnliche Systeme

```
./hello
```

Windows

```
hello.exe
```

***Hinweis** : Die chinesischen Zeichen sind wichtig, da sie zeigen, dass Go-Zeichenfolgen als schreibgeschützte Byte-Segmente gespeichert werden.*

FizzBuzz

Ein anderes Beispiel für "Hello World" -Stilprogramme ist FizzBuzz . Dies ist ein Beispiel für eine FizzBuzz-Implementierung. Sehr idiomatisch Gehen Sie hier ins Spiel.

```
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)
        }
    }
}
```

Spielplatz

Go-Umgebungsvariablen auflisten

Umgebungsvariablen, die das `go` Tool beeinflussen, können mit dem Befehl `go env [var ...]` angezeigt werden:

```
$ 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"
```

Standardmäßig wird die Liste als Shell-Skript gedruckt. Wenn jedoch ein oder mehrere Variablennamen als Argumente angegeben werden, wird der Wert jeder benannten Variablen ausgegeben.

```
$go env GOOS GOPATH
linux
/home/yourname
```

Umgebung einrichten

Wenn Go in Ihrem System nicht vorinstalliert ist, können Sie unter https://golang.org/dl/ die Plattform auswählen, in der Go heruntergeladen und installiert werden soll.

Um eine grundlegende Go-Entwicklungsumgebung einzurichten, müssen nur einige der vielen Umgebungsvariablen festgelegt werden, die das Verhalten des `go` Tools beeinflussen (siehe: Auflisten der Go-Umgebungsvariablen für eine vollständige Liste) (im Allgemeinen in `~/.profile`

Ihrer Shell) Datei oder gleichwertig auf Unix-ähnlichen Betriebssystemen).

GOPATH

Wie auch die Umgebungsvariable `PATH` des Systems ist Go path eine `:` (`;` unter Windows) getrennte Liste von Verzeichnissen, in denen Go nach Paketen sucht. Das `go get` Tool lädt auch Pakete in das erste Verzeichnis in dieser Liste.

In `GOPATH` Go die zugehörigen Ordner `bin` , `pkg` und `src` für den Arbeitsbereich eingerichtet:

- `src` - Speicherort der Quelldateien: `.go` , `.c` , `.g` , `.s`
- `pkg` - hat `.a` Dateien kompiliert
- `bin` - enthält ausführbare Dateien, die von Go erstellt wurden

Ab Go 1.8 hat die Umgebungsvariable `GOPATH` einen Standardwert, wenn sie nicht festgelegt ist. Der Standardwert ist $ HOME / go unter Unix / Linux und% USERPROFILE% / go unter Windows.

Einige Tools setzen voraus, dass `GOPATH` ein einzelnes Verzeichnis enthält.

GOBIN

Das Verzeichnis ist , wo `go install` und `go get` werden Binärdateien legen nach dem Aufbau `main` Paketen. Im Allgemeinen ist dies auf irgendwo auf dem Systempfad `PATH` sodass installierte Binärdateien problemlos ausgeführt und erkannt werden können.

GOROOT

Dies ist der Ort Ihrer Go-Installation. Es wird verwendet, um die Standardbibliotheken zu finden. Es ist sehr selten, dass Sie diese Variable festlegen müssen, da Go den Build-Pfad in die Toolchain einbettet. Die Einstellung von `GOROOT` ist erforderlich, wenn sich das Installationsverzeichnis vom Build-Verzeichnis (oder dem beim Erstellen festgelegten Wert) unterscheidet.

Offline-Zugriff auf die Dokumentation

Führen Sie für die vollständige Dokumentation den Befehl aus:

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

Für eine Tour von Go (sehr empfehlenswert für Anfänger in der Sprache):

```
go tool tour
```

Mit den beiden obigen Befehlen werden Webserver mit einer Dokumentation gestartet, die der hier und hier jeweils im Internet verfügbaren ähnelt.

Für eine schnelle Referenzprüfung in der Befehlszeile, z. B. für fmt.Print:

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

Allgemeine Hilfe ist auch über die Befehlszeile verfügbar:

```
go help [command]
```

Laufen Online gehen

Der Spielplatz

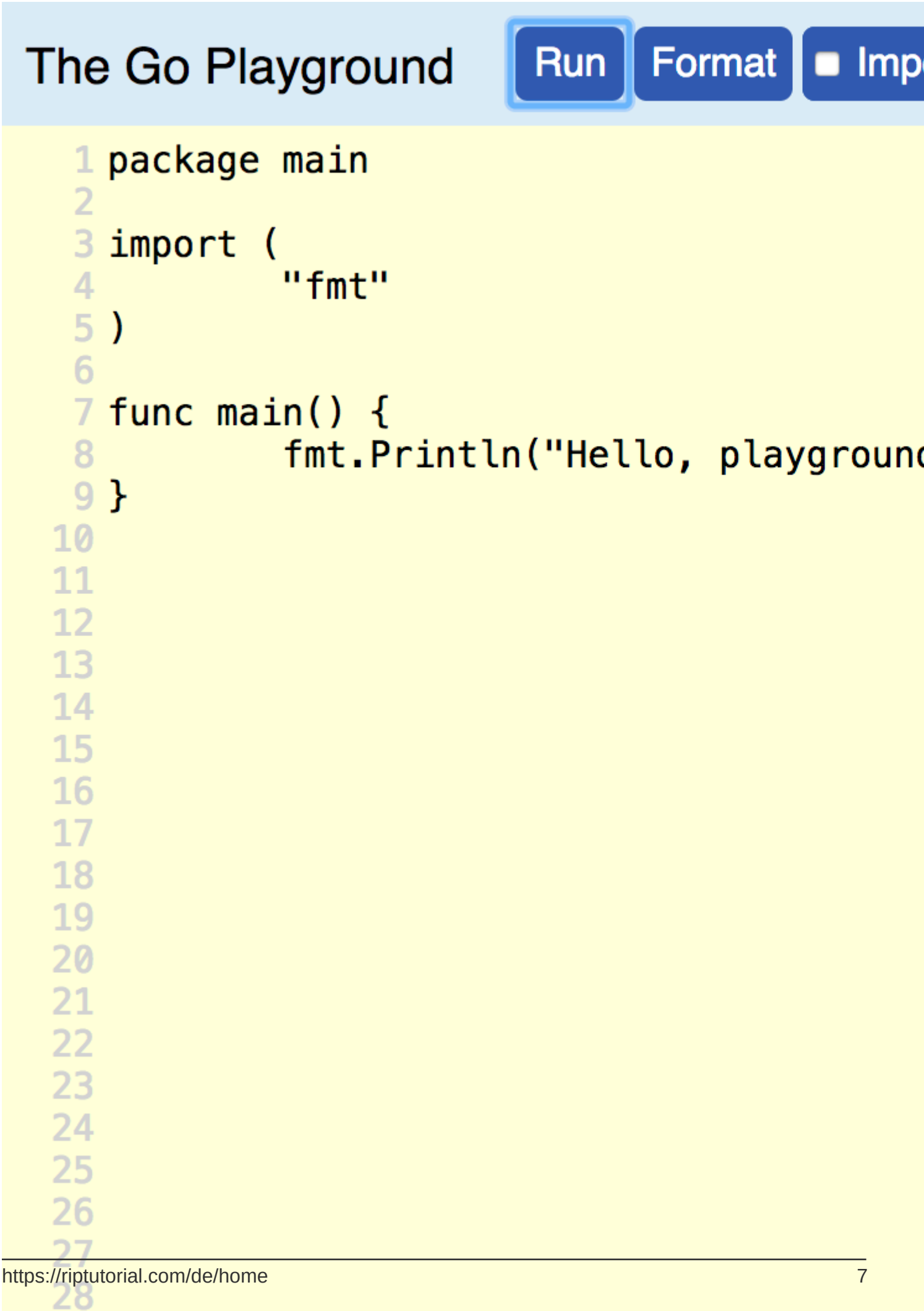
Ein wenig bekanntes Go-Tool ist **The Go Playground** . Wenn Sie mit Go experimentieren möchten, ohne es herunterzuladen, können Sie dies einfach mit tun. . .

1. Besuch des Spielplatzes in ihrem Webbrowser
2. Code eingeben
3. Klicken Sie auf "Ausführen".

Teilen Sie Ihren Code

Auf dem Go Playground stehen auch Tools zum Teilen zur Verfügung. Wenn ein Benutzer auf die Schaltfläche "Teilen" klickt, wird ein Link (wie dieser) generiert, der an andere Personen zum Testen und Bearbeiten gesendet wird.

In Aktion


The Go Playground
Run
Format
Imp
1 package main
2
3 import (
4 "fmt"
5)
6
7 func main() {
8 fmt.Println("Hello, playgroun
9 }
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28

Kapitel 2: Analysieren von Befehlszeilenargumenten und Flags

Examples

Kommandozeilenargumente

Das Analysieren von Befehlszeilenargumenten ist Go ist anderen Sprachen sehr ähnlich. In Ihrem Code greifen Sie einfach auf die Argumente zu, wobei das erste Argument der Name des Programms selbst ist.

Schnelles Beispiel:

```
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("")
}
```

Und Ausgabe wäre:

```
$ ./cmd test_arg1 test_arg2
Here we have program './cmd' launched with following flags: test_arg1 test_arg2
```

Jedes Argument ist nur eine Zeichenfolge. Im `os` Paket sieht das so aus: `var Args []string`

Flaggen

Go - Standardbibliothek bietet Paket - `flag` , die mit Parsing - Flags übergeben Programm hilft.

Beachten Sie, dass das `flag` Paket keine üblichen Flags im GNU-Stil enthält. Das bedeutet , dass mehrbuchstabigen Fahnen müssen mit Bindestrich wie diese gestartet werden: `-exampleflag` , das nicht: `--exampleflag` . Flaggen im GNU-Stil können mit einem Drittanbieter-Paket erstellt werden.

```
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` hilft uns Nachricht:

```
$ ./flags -h
Usage of ./flags:
  -int.flag int
        usage of intFlag (default 1)
  -string.flag string
        here comes usage (default "default value")
```

Mit allen Flaggen anrufen:

```
$ ./flags -string.flag test -int.flag 24
Value of stringFlag is: test
Value of intFlag is: 24
```

Anruf mit fehlenden Flags:

```
$ ./flags
Value of stringFlag is: default value
Value of intFlag is: 1
```

Analysieren von Befehlszeilenargumenten und Flags online lesen:
https://riptutorial.com/de/go/topic/4023/analysieren-von-befehlszeilenargumenten-und-flags

Kapitel 3: Analysieren von CSV-Dateien

Syntax

- csvReader: = csv.NewReader (r)
- data, err: = csvReader.Read ()

Examples

Einfaches CSV-Parsing

Betrachten Sie diese CSV-Daten:

```
#id,title,text
1,hello world,"This is a ""blog""."
2,second time,"My
second
entry."
```

Diese Daten können mit folgendem Code gelesen werden:

```
// 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])
}
```

Und produzieren:

```
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
```

Spielplatz: https://play.golang.org/p/d2E6-CGGIe .

Analysieren von CSV-Dateien online lesen: https://riptutorial.com/de/go/topic/5818/analysieren-von-csv-dateien

Kapitel 4: Arbeiterpools

Examples

Einfacher Arbeiterpool

Eine einfache Implementierung des Worker-Pools:

```
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)
    }
}
```

Auftragswarteschlange mit Arbeitspool

Eine Auftragswarteschlange, die einen Arbeitspool verwaltet, der für die Hintergrundverarbeitung auf Webservern nützlich ist

```
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 sispatcher 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)})
  }
}
```

Arbeiterpools online lesen: https://riptutorial.com/de/go/topic/4182/arbeiterpools

Kapitel 5: Arrays

Einführung

Arrays sind bestimmte Datentypen, die eine geordnete Sammlung von Elementen eines anderen Typs darstellen.

In Go können Arrays einfach (manchmal "Listen" genannt) oder mehrdimensional sein (wie zum Beispiel 2-Dimentions-Arrays eine geordnete Sammlung von Arrays darstellen, die Elemente enthält)

Syntax

- var variableName [5] ArrayType // Deklarieren eines Arrays der Größe 5.
- var variableName [2] [3] ArrayType = {{Value1, Value2, Value3}, {Value4, Value5, Value6}} // Deklaration eines mehrdimensionalen Arrays
- variableName: = [...] ArrayType {Value1, Value2, Value3} // Deklarieren Sie ein Array der Größe 3 (Der Compiler zählt die Array-Elemente, um die Größe zu bestimmen)
- arrayName [2] // Den Wert anhand des Index abrufen.
- arrayName [5] = 0 // Wert am Index setzen.
- arrayName [0] // Erster Wert des Arrays
- arrayName [len (arrayName) -1] // Letzter Wert des Arrays

Examples

Arrays erstellen

Ein Array in go ist eine geordnete Sammlung gleichartiger Elemente.
Die grundlegende Notation zur Darstellung von Arrays ist die Verwendung von `[]` mit dem Variablennamen.

Das Erstellen eines neuen Arrays sieht wie `var array = [size]Type` , wobei `size` durch eine Zahl ersetzt wird (zum Beispiel `42` um anzugeben, dass es eine Liste von 42 Elementen ist) und `Type` durch den Typ der Elemente ersetzt, die das Array enthalten kann (z Beispiel `int` oder `string`)

Direkt darunter sehen Sie ein Codebeispiel, das zeigt, wie ein Array in Go anders erstellt wird.

```
// 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("zeroes:", 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
```

spiele es auf dem Spielplatz

Mehrdimensionales Array

Mehrdimensionale Arrays sind im Grunde Arrays, die andere Arrays als Elemente enthalten.
Es wird als `[sizeDim1][sizeDim2]..[sizeLastDim]type` , wobei `sizeDim` durch Zahlen ersetzt wird, die der Länge der Dimension entsprechen, und `type` durch den Datentyp im mehrdimensionalen Array.

Zum Beispiel repräsentiert `[2][3]int` ein Array, das aus **2 Unterarrays** von **3 int typisierten Elementen besteht** .
Es kann grundsätzlich die Darstellung einer Matrix aus **2 Zeilen** und **3 Spalten sein** .

Wir können also ein Array mit großen Dimensionswerten wie `var values := [2017][12][31][24][60]int` wenn Sie beispielsweise seit Jahr 0 eine Zahl für jede Minute speichern müssen.

Um auf diese Art von Array zuzugreifen, suchen Sie für das letzte Beispiel bei der Suche nach dem Wert von 2016-01-31 um 19:42 auf die `values[2016][0][30][19][42]` (weil **Array indiziert beginnt um 0** und nicht um 1 wie Tage und Monate)

Einige Beispiele folgen:

```
// 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-Indizes

Auf Array-Werte sollte mit einer Nummer zugegriffen werden, die die Position des gewünschten Werts im Array angibt. Diese Nummer wird als Index bezeichnet.

Der Index beginnt bei **0** und endet bei der **Arraylänge -1** .

Um auf einen Wert zuzugreifen, müssen Sie `arrayName[index]` tun: `arrayName[index]` . Ersetzen Sie "index" durch die Zahl, die dem Rang des Werts in Ihrem Array entspricht.

Zum Beispiel:

```
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)
```

Um einen Wert im Array festzulegen oder zu ändern, ist der Weg derselbe.
Beispiel:

```
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]
```

Arrays online lesen: https://riptutorial.com/de/go/topic/390/arrays

Kapitel 6: Base64-Kodierung

Syntax

- func (enc * base64.Encoding) Encode (dst, src [] Byte)
- func (enc * base64.Encoding) Decodierung (dst, src [] byte) (n int, err Fehler)
- func (enc * base64.Encoding) EncodeToString (src [] Byte) Zeichenfolge
- func (enc * base64.Encoding) DecodeString (s Zeichenfolge) ([] Byte, Fehler)

Bemerkungen

Das `encoding/base64` Paket enthält mehrere eingebaute Encoder . Die meisten Beispiele in diesem Dokument verwenden `base64.StdEncoding` kann jedoch auch ein beliebiger Encoder (`URLEncoding` , `RawStdEncodign` , eigener benutzerdefinierter Encoder usw.) verwendet werden.

Examples

Codierung

```
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==
```

Spielplatz

Kodierung in einen String

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

Spielplatz

Dekodierung

```
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`
```

Spielplatz

Einen String dekodieren

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

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

Spielplatz

Base64-Kodierung online lesen: https://riptutorial.com/de/go/topic/4492/base64-kodierung

Kapitel 7: Befehle ausführen

Examples

Zeitüberschreitung mit Interrupt und dann 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")
}
```

Einfache Befehlsausführung

```
// Execute a command a 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 standerr output
out, err := exec.Command("echo", "foo").Output()
if err != nil {
    log.Fatal(err)
}
```

Einen Befehl ausführen, dann fortfahren und warten

```
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)
```

Einen Befehl zweimal ausführen

> Ein Cmd kann nach dem Aufruf seiner Run-, Output- oder CombinedOutput-Methoden nicht wiederverwendet werden

Einen Befehl zweimal auszuführen, *wird **nicht** funktionieren* :

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

> Fehler: exec: bereits gestartet

`exec.Command` muss man **zwei separate** `exec.Command` . Möglicherweise benötigen Sie auch eine gewisse Verzögerung zwischen den Befehlen.

```
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
```

Befehle ausführen online lesen: https://riptutorial.com/de/go/topic/1097/befehle-ausfuhren

Kapitel 8: Best Practices zur Projektstruktur

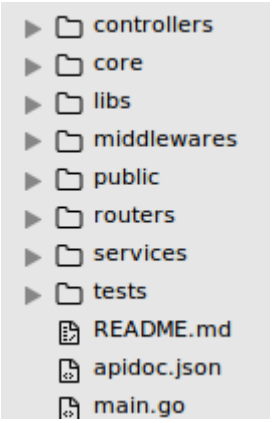
Examples

Restfull Projects API mit Gin

> Gin ist ein in Golang geschriebenes Web-Framework. Es verfügt über eine martiniähnliche API, die bis zu 40-mal schneller ist. Wenn Sie Leistung und gute Produktivität benötigen, werden Sie Gin lieben.

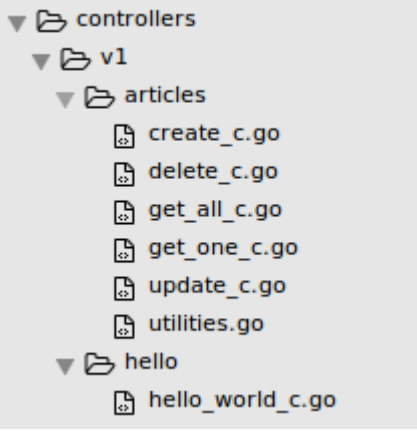
Es wird 8 Pakete + main.go geben

1. Steuerungen
2. Ader
3. Libs
4. Middlewares
5. Öffentlichkeit
6. Router
7. Dienstleistungen
8. Tests
9. main.go



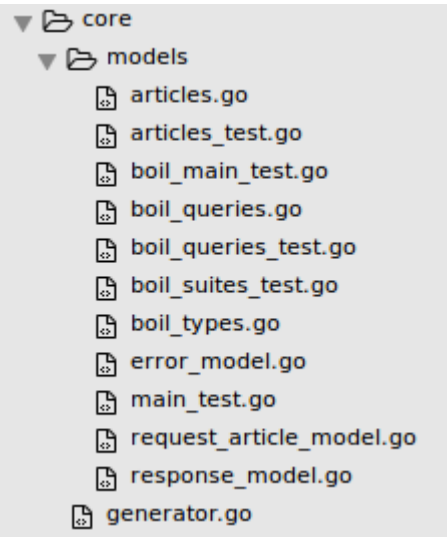
Steuerungen

Das Controller-Paket speichert die gesamte API-Logik. Was auch immer Ihre API ist, Ihre Logik wird hier geschehen



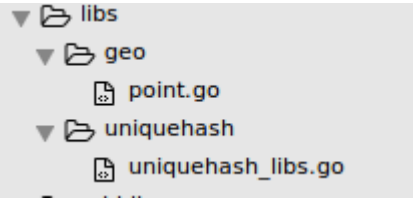
Ader

Im Kernpaket werden alle erstellten Modelle, ORM usw. gespeichert



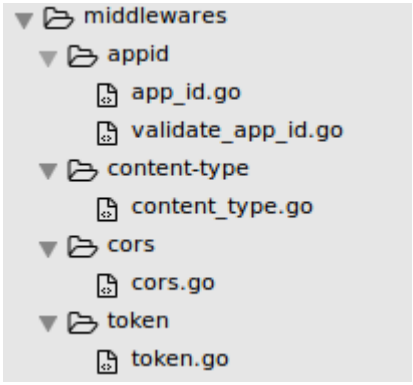
Libs

In diesem Paket wird jede Bibliothek gespeichert, die in Projekten verwendet wird. Aber nur für manuell erstellte / importierte Bibliotheken, die nicht verfügbar sind, wenn die Befehle `go get package_name` werden. Könnte Ihr eigener Hash-Algorithmus, Graph, Baum usw. sein



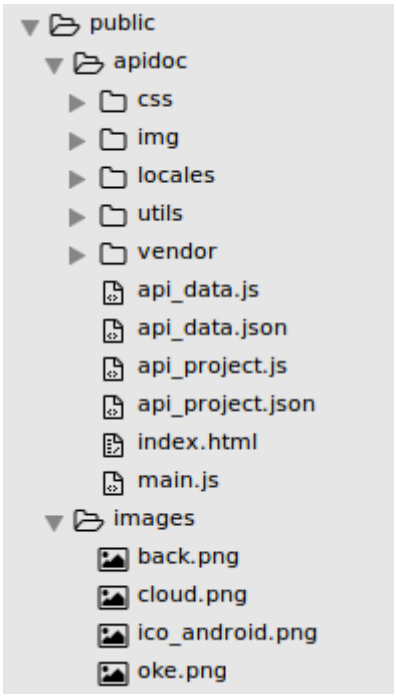
Middlewares

Dieses Paket speichert jede Middleware, die in einem Projekt verwendet wird. Dies kann die Erstellung / Validierung von Kors, die Geräte-ID, die Authentifizierung usw.



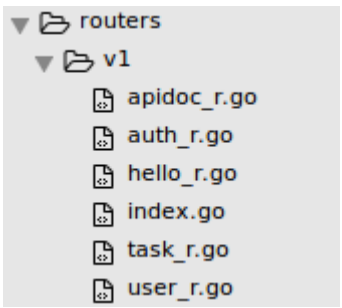
Öffentlichkeit

In diesem Paket werden alle öffentlichen und statischen Dateien gespeichert. Dies können HTML, CSS, Javascript, Bilder usw. sein



Router

Dieses Paket speichert alle Routen in Ihrer REST-API.



Siehe Beispielcode zum Zuweisen der Routen.

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.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRyc

 *            }
 *      }
 * @apiErrorExample {json} List error
 *    HTTP/1.1 500 Internal Server Error
 */

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

Wenn Sie sehen, ist der Grund, warum ich den Handler trenne, einfach, die einzelnen Router zu verwalten. So kann ich Kommentare zur API erstellen, die mit apidoc in strukturierter Dokumentation generiert werden. Dann werde ich die Funktion in index.go im aktuellen Paket aufrufen

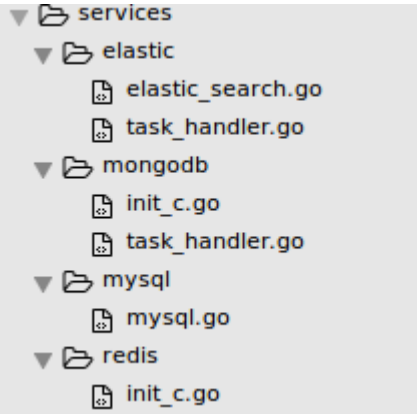
index.go

```
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)
}
```

Dienstleistungen

In diesem Paket werden alle in einem Projekt verwendeten Konfigurationen und Einstellungen gespeichert, z. B. Mongodb, Redis, MySQL, Elasticsearch usw.



---

main.go

Der Haupteingang der API. Alle Einstellungen zu den Einstellungen der Dev-Umgebung, zu Systemen, zum Port usw. werden hier konfiguriert.

Beispiel:
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: Jeder Code in diesem Beispiel stammt aus verschiedenen Projekten

---

Siehe Beispielprojekte auf Github

Best Practices zur Projektstruktur online lesen: https://riptutorial.com/de/go/topic/9463/best-practices-zur-projektstruktur

Kapitel 9: Bilder

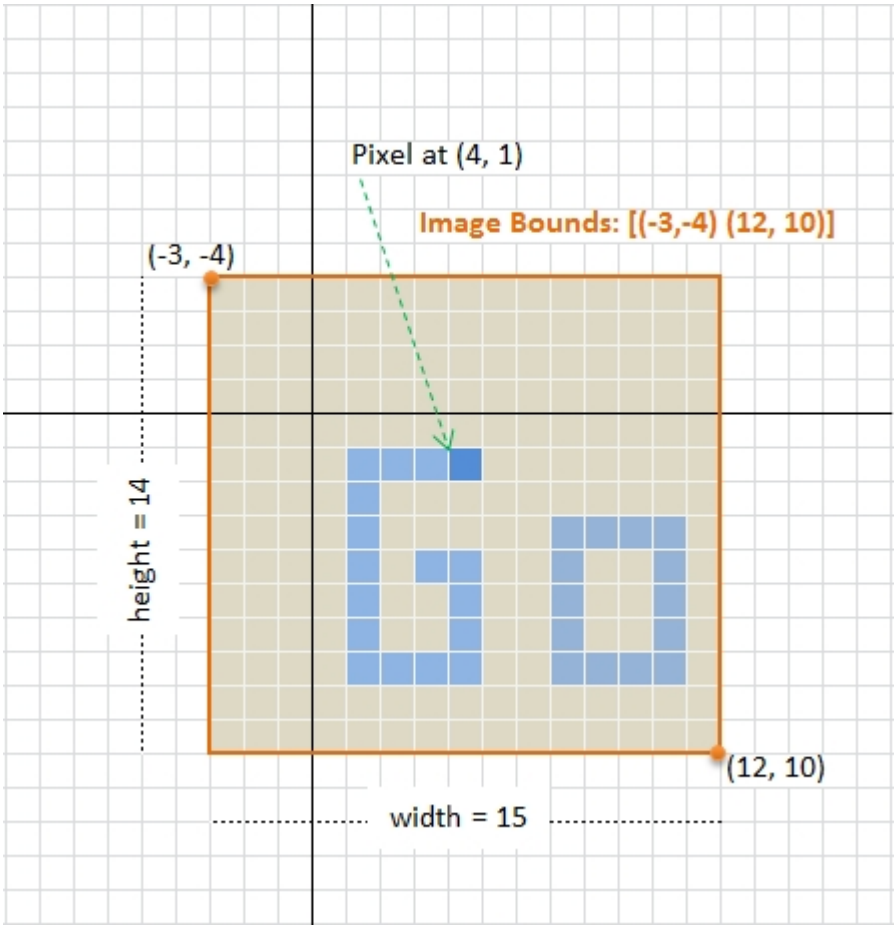
Einführung

Das Image- Paket bietet grundlegende Funktionen für die Arbeit mit 2-D-Images. In diesem Thema werden einige grundlegende Vorgänge beim Arbeiten mit Bildern beschrieben, z. B. Lesen und Schreiben eines bestimmten Bildformats, Zuschneiden, Zugreifen auf und Ändern von *Pixeln* , Farbkonvertierung, Größenänderung und grundlegende Bildfilterung.

Examples

Grundlegendes Konzept

Ein Bild repräsentiert ein rechteckiges Gitter von Bildelementen (*Pixel*). In dem Bildpaket, wird das Pixel als eine der Farbe dargestellt in definierten Bild / Farb Paket. Die 2-D-Geometrie des Bildes wird als `image.Rectangle` , während `image.Point` eine Position im Raster `image.Point` .



Die obige Abbildung veranschaulicht die grundlegenden Konzepte eines Bildes im Paket. Ein Bild der Größe 15x14 Pixel hat rechteckige *Begrenzungen,* die an der *oberen linken* Ecke beginnen (z. B. Koordinate (-3, -4) in der obigen Abbildung), und seine Achsen steigen nach rechts und nach unten in die *untere rechte* Ecke (z. B. Koordinate (12, 10) in der Figur). Beachten Sie, dass die Grenzen **nicht unbedingt an Punkt (0,0) beginnen oder diesen enthalten** .

Bildbezogener *Typ*

In `Go` implementiert ein Bild immer die folgende `image.Image` Schnittstelle

```
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 dem die `color.Color` Schnittstelle definiert ist als

```
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)
}
```

und `color.Model` ist eine als deklarierte Schnittstelle

```
type Model interface {
    Convert(c Color) Color
}
```

Zugriff auf Bildgröße und Pixel

Angenommen, wir haben ein Bild als Variable `img` gespeichert, dann können wir die Dimension und das Bildpixel erhalten durch:

```
// 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
    }
}
```

Beachten Sie, dass der Wert der einzelnen Komponenten `R,G,B,A` im Paket zwischen `0-65535` (`0x0000 - 0xffff`) liegt und **nicht zwischen** **`0-255`** .

Bild laden und speichern

Im Speicher kann ein Bild als Matrix aus Pixeln (Farbe) betrachtet werden. Wenn ein Bild in einem permanenten Speicher gespeichert wird, kann es jedoch so gespeichert werden (RAW-Format), Bitmap oder andere Bildformate mit einem bestimmten Kompressionsalgorithmus zum Speichern von Speicherplatz, z Bei einem bestimmten Format muss das Bild in `image.Image` mit dem entsprechenden Algorithmus *dekodiert* werden. Eine `image.Decode` deklarierte `image.Decode` Funktion

```
func Decode(r io.Reader) (Image, string, error)
```

wird für diese bestimmte Verwendung bereitgestellt. Um verschiedene Bildformate verarbeiten zu können, muss der Decoder vor dem Aufrufen der `image.Decode` Funktion über die als definierte `image.RegisterFormat` Funktion registriert werden

```
func RegisterFormat(name, magic string,
    decode func(io.Reader) (Image, error), decodeConfig func(io.Reader) (Config, error))
```

Derzeit unterstützt das Image-Paket drei Dateiformate: JPEG , GIF und PNG . Um einen Decoder zu registrieren, fügen Sie Folgendes hinzu

```
import _ "image/jpeg"    //register JPEG decoder
```

auf die Anwendung `main` Paket. Irgendwo in Ihrem Code (nicht notwendig in `main` Paket), ein JPEG - Bild zu laden, die folgenden Schnipsel verwenden:

```
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` ...
```

Speichern Sie in PNG

Um ein Bild in einem bestimmten Format zu speichern, muss der entsprechende *Encoder* explizit importiert werden

```
import "image/png"    //needed to use `png` encoder
```

Dann kann ein Bild mit den folgenden Ausschnitten gespeichert werden:

```
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
}
```

Wenn Sie eine andere Komprimierungsstufe als die `DefaultCompression` angeben `DefaultCompression` , erstellen Sie einen *Encoder* , z

```
enc := png.Encoder{
    CompressionLevel: png.BestSpeed,
}
err := enc.Encode(f, img)
```

Speichern Sie in JPEG

Verwenden Sie zum Speichern im `jpeg` Format Folgendes:

```
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
}
```

In GIF speichern

Verwenden Sie die folgenden Ausschnitte, um das Bild in einer GIF-Datei zu speichern.

```
import "image/gif"

// Samewhere 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
}
```

Bild zuschneiden

Die meisten Bildtypen in Bildpaket mit `SubImage(r Rectangle) Image` Methode, außer `image.Uniform` . Basierend auf dieser Tatsache können Wir eine Funktion implementieren, um ein beliebiges Bild wie folgt zu beschneiden

```
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
}
```

Beachten Sie, dass das zugeschnittene Bild möglicherweise die darunter liegenden Pixel mit dem Originalbild teilt. Wenn dies der Fall ist, wirken sich Änderungen am zugeschnittenen Bild auf das Originalbild aus.

Konvertieren Sie ein Farbbild in Graustufen

Bei einigen digitalen Bildverarbeitungsalgorithmen wie der Kantenerkennung ist die durch die Bildintensität (dh den Graustufenwert) übertragene Information ausreichend. Die Verwendung von Farbinformationen (`R, G, B` Kanal) kann ein etwas besseres Ergebnis liefern, die Algorithmuskomplexität wird jedoch erhöht. Daher müssen wir in diesem Fall das Farbbild vor der Anwendung eines solchen Algorithmus in ein Graustufenbild konvertieren.

Der folgende Code ist ein Beispiel für das Konvertieren eines beliebigen Bildes in ein 8-Bit-Graustufenbild. Das Abbild wird mit einem `net/http` Paket vom Remote-Standort abgerufen, in Graustufen konvertiert und schließlich als PNG-Abbild gespeichert.

```
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)
    }
}
```

Die `Set(x, y int, c color.Color)` erfolgt bei der Zuweisung von Pixeln über `Set(x, y int, c color.Color)` das in `image.go` as implementiert ist

```
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
}
```

wobei `color.GrayModel` in `color.go` als definiert ist

```
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)}
}
```

Basierend auf den obigen Tatsachen wird die Intensität `Y` mit der folgenden Formel berechnet:

> Leuchtdichte: Y = 0,299 **R** + 0,587 **G** + 0,114 **B**

Wenn wir verschiedene Formeln / Algorithmen anwenden möchten, um eine Farbe in eine Intensität zu konvertieren, z

Mittelwert: Y = (**R** + **G** + **B**) / 3
Luma: Y = 0,2126 **R** + 0,7152 **G** + 0,0722 **B**
Glanz: Y = (min (**R** , **G** , **B**) + max (**R** , **G** , **B**)) / 2

Dann können die folgenden Ausschnitte verwendet werden.

```
// 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)
    }
}
```

Die obige Berechnung erfolgt durch Gleitkommamultiplikation und ist sicherlich nicht die effizienteste, reicht aber aus, um die Idee zu demonstrieren. Der andere Punkt ist, dass beim Aufruf von `Set(x, y int, c color.Color)` mit `color.Gray` als drittem Argument das Farbmodell keine `color.Gray` durchführt, wie in der vorherigen `grayModel` Funktion zu sehen ist.

Bilder online lesen: https://riptutorial.com/de/go/topic/10557/bilder

Kapitel 10: cgo

Examples

Cgo: Erste Schritte Tutorial

Einige Beispiele, um den Arbeitsablauf bei der Verwendung von Go-C-Bindungen zu verstehen

Was

In Go können Sie mit cgo C-Programme und Funktionen aufrufen . Auf diese Weise können Sie problemlos C-Bindungen für andere Anwendungen oder Bibliotheken erstellen, die C-API bereitstellen.

Wie

Alles, was Sie tun müssen, ist, ein `import "C"` am Anfang Ihres Go-Programms hinzuzufügen, **unmittelbar** nachdem Sie Ihr C-Programm hinzugefügt haben:

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

Mit dem vorherigen Beispiel können Sie das `stdio` Paket in Go verwenden.

Wenn Sie eine App verwenden müssen, die sich in demselben Ordner befindet, verwenden Sie dieselbe Syntax wie in C (mit `"` anstelle von `<>`)".

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

> **WICHTIG** : Lassen Sie **keine Newline zwischen den Anweisungen `include` und `import "C"` Andernfalls** erhalten Sie diese Art von Fehlern beim Build:

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

Das Beispiel

In diesem Ordner finden Sie ein Beispiel für C-Bindungen. Wir haben zwei sehr einfache "Bibliotheken" namens `hello.c` :

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

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

Das druckt einfach "Hallo Welt" in der Konsole und in `sum.c`

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

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

... das nimmt 2 Argumente und gibt die Summe zurück (nicht drucken).

Wir haben ein `main.go` Programm, das diese beiden Dateien verwenden wird. Zuerst importieren wir sie wie zuvor erwähnt:

```
//main.go
package main

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

Hallo Welt!

Jetzt können wir die C-Programme in unserer Go-App verwenden. Versuchen wir zuerst das Hello-Programm:

```
//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
}
```

Führen Sie nun das main.go-Programm mit dem `go run main.go` , um den Ausdruck des C-Programms zu erhalten: "Hallo Welt!". Gut gemacht!

Summe der Beträge

Lassen Sie uns etwas komplexer machen, indem Sie eine Funktion hinzufügen, die die beiden Argumente summiert.

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

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

Und wir rufen es von unserer vorherigen Go-App an.

```
//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
}
```

Schauen Sie sich die "makeSum" -Funktion an. Es empfängt zwei `int` Parameter, die zuvor mithilfe der `C.int` Funktion in C `int` `C.int` müssen. Die Rückgabe des Anrufs gibt uns auch ein C `int` und einen Fehler, falls etwas schief gelaufen ist. Wir müssen eine C-Antwort mit `int()` auf ein int von Go setzen.

Versuchen Sie, unsere Go-App mit `go run main.go`

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

Eine Binärdatei erzeugen

Wenn Sie einen go-Build versuchen, könnten Sie mehrere Definitionsfehler erhalten.

```
$ 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
```

Der Trick besteht darin, direkt auf die Hauptdatei zu verweisen, wenn Sie `go build` :

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

> Denken Sie daran, dass Sie der Binärdatei einen Namen geben können, indem Sie das Flag `-o` verwenden. `go build -o my_c_binding main.go`

Ich hoffe, dir hat dieses Tutorial gefallen.

cgo online lesen: https://riptutorial.com/de/go/topic/6125/cgo

Kapitel 11: cgo

Examples

Aufruf der C-Funktion von unterwegs

Mit Cgo können Sie Go-Pakete erstellen, die C-Code aufrufen.
Um `cgo` write normalen Go-Code zu verwenden, der ein Pseudo-Paket "C" importiert. Der Go-Code kann sich dann auf Typen wie `C.int` oder Funktionen wie `C.Add` .
Dem Import von "C" wird unmittelbar ein Kommentar vorangestellt, der als Präambel bezeichnet wird und beim Kompilieren der C-Teile des Pakets als Header verwendet wird.
Beachten Sie, dass sich zwischen dem `cgo` Kommentar und der import-Anweisung keine Leerzeilen befinden dürfen.
Beachten Sie, dass der `import "C"` nicht mit anderen Importen in einer in Klammern stehenden "Factored" -Import-Anweisung zusammengefasst werden kann. Sie müssen mehrere Importanweisungen schreiben, z.

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

Und es ist ein guter Stil, die factored import -Anweisung für andere Importe zu verwenden, wie:

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

Einfaches Beispiel mit `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
}
```

Dann `go build` und führen Sie es aus:

```
30
```

`cgo` zum Erstellen von `cgo` Paketen wie gewohnt `go build` oder `go install` . Das `go tool` erkennt den speziellen `"C"` -Import und verwendet automatisch `cgo` für diese Dateien.

C und C-Code in alle Richtungen verdrahten

Aufruf des C-Codes von Go

```
package main

/*
// Everything in comments above the import "C" is C code and will be compiles 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))
}
```

Go-Code von C aufrufen

```
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
}
```

Umgang mit Funktionszeigern

```
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
}
```

Konvertieren Sie Typen, Zugriffsstrukturen und Zeigerarithmetik

Aus der offiziellen Go-Dokumentation:

```
// 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
```

Wie man es benutzt:

```
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
}
```

cgo online lesen: https://riptutorial.com/de/go/topic/6455/cgo

Kapitel 12: Channels

Einführung

Ein Kanal enthält Werte eines bestimmten Typs. Werte können in einen Kanal geschrieben und von diesem gelesen werden, und sie zirkulieren innerhalb des Kanals in der Reihenfolge des ersten Durchlaufs. Es wird unterschieden zwischen gepufferten Kanälen, die mehrere Meldungen enthalten können, und ungepufferten Kanälen, die dies nicht können. Kanäle werden normalerweise zur Kommunikation zwischen Goroutinen verwendet, sind aber auch in anderen Situationen nützlich.

Syntax

- make (chan int) // einen ungepufferten Kanal erstellen
- make (chan int, 5) // einen gepufferten Kanal mit einer Kapazität von 5 erstellen
- close (ch) // schließt einen Kanal "ch"
- ch <- 1 // Schreibe den Wert von 1 in einen Kanal "ch"
- val: = <-ch // lese einen Wert aus Kanal "ch"
- val, ok: = <-ch // alternative Syntax; ok ist ein Bool, der anzeigt, ob der Kanal geschlossen ist

Bemerkungen

Ein Kanal, der die leere struct `make(chan struct{})` ist eine klare Nachricht an den Benutzer, dass keine Informationen über den Kanal übertragen werden und dass diese ausschließlich zur Synchronisation verwendet werden.

Bei ungepufferten Kanälen wird ein Kanalschreiben blockiert, bis ein entsprechender Lesevorgang von einer anderen Goroutine erfolgt. Dasselbe gilt für das Blockieren eines Kanals, während auf einen Brenner gewartet wird.

Examples

Bereich verwenden

Beim Lesen mehrerer Werte aus einem Kanal ist die Verwendung eines `range` ein übliches Muster:

```
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")
}
```

Spielplatz

Ausgabe

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

Timeouts

Kanäle werden häufig zur Implementierung von Timeouts verwendet.

```
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.
    }
}
```

Koordinierende Goroutinen

Stellen Sie sich eine Goroutine mit einem zweistufigen Prozess vor, bei der der Haupt-Thread zwischen jedem Schritt etwas Arbeit ausführen muss:

```
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
}
```

Gepuffert vs ungepuffert

```
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")
    }
}
```

Spielplatz gehen

Sperren und Entsperren von Kanälen

Standardmäßig erfolgt die Kommunikation über die Kanäle synchron. Wenn Sie einen Wert senden, muss es einen Empfänger geben. Andernfalls erhalten Sie einen `fatal error: all goroutines are asleep - deadlock!` wie folgt:

```
package main

import "fmt"

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

Es gibt jedoch eine Lösungsverwendung: Verwenden Sie gepufferte Kanäle:

```
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)
}
```

Warten, bis die Arbeit abgeschlossen ist

Eine gebräuchliche Technik für die Verwendung von Kanälen besteht darin, eine Anzahl von Arbeitern (oder Verbrauchern) zum Lesen aus dem Kanal zu erstellen. Die Verwendung einer sync.WaitGroup ist eine einfache Möglichkeit, auf die Ausführung dieser Worker zu warten.

```
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")
}
```

Channels online lesen: https://riptutorial.com/de/go/topic/1263/channels

Kapitel 13: Constraints erstellen

Syntax

- // + Tags erstellen

Bemerkungen

Build-Tags werden verwendet, um bestimmte Dateien in Ihrem Code bedingt zu erstellen. Build-Tags ignorieren möglicherweise Dateien, die nicht erstellt werden sollen, es sei denn, sie werden explizit einbezogen, oder es werden einige vordefinierte Build-Tags verwendet, damit eine Datei nur auf einer bestimmten Architektur oder einem bestimmten Betriebssystem erstellt wird.

Build-Tags können in jeder Art von Quelldatei (nicht nur in Go) angezeigt werden, sie müssen jedoch am oberen Rand der Datei angezeigt werden und dürfen nur Leerzeilen und andere Zeilenkommentare enthalten. Diese Regeln bedeuten, dass in Go-Dateien vor der Paketklausel eine Buildeinschränkung angezeigt werden muss.

Auf eine Reihe von Build-Tags muss eine Leerzeile folgen.

Examples

Separate Integrationstests

Build-Einschränkungen werden normalerweise verwendet, um normale Unit-Tests von Integrationstests zu trennen, für die externe Ressourcen erforderlich sind, beispielsweise eine Datenbank oder ein Netzwerkzugriff. Fügen Sie dazu eine benutzerdefinierte Buildeinschränkung oben in der Testdatei hinzu:

```
// +build integration

package main

import (
    "testing"
)

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

Die Testdatei wird nicht in die ausführbare Builddatei kompiliert, wenn der folgende Aufruf von `go test` verwendet wird:

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

Ergebnisse:

```
$ 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
```

Optimieren Sie Implementierungen basierend auf Architektur

Wir können eine einfache xor-Funktion nur für Architekturen optimieren, die nicht ausgerichtetes Lesen / Schreiben unterstützen, indem Sie zwei Dateien erstellen, die die Funktion definieren und ihnen eine Build-Einschränkung voranstellen (ein Beispiel für den hier außerhalb des Gültigkeitsbereichs befindlichen xor-Code finden Sie unter `crypto/cipher/xor.go` in der Standardbibliothek):

```
// +build 386 amd64 s390x

package cipher

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

und für andere Architekturen:

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

package cipher

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

Constraints erstellen online lesen: https://riptutorial.com/de/go/topic/2595/constraints-erstellen

Kapitel 14: Datei I / O

Syntax

- file, err: = os.Open (*name*) // Öffnet eine Datei im schreibgeschützten Modus. Ein Nicht-Null-Fehler wird zurückgegeben, wenn die Datei nicht geöffnet werden konnte.
- file, err: = os.Create (*name*) // Erzeugt oder öffnet eine Datei, wenn sie im schreibgeschützten Modus bereits vorhanden ist. Die Datei wird überschrieben, wenn sie bereits existiert. Ein Nicht-Null-Fehler wird zurückgegeben, wenn die Datei nicht geöffnet werden konnte.
- file, err: = os.OpenFile (*Name* , *Flags* , *Perm*) // Öffnet eine Datei in dem durch die Flags angegebenen Modus. Ein Nicht-Null-Fehler wird zurückgegeben, wenn die Datei nicht geöffnet werden konnte.
- data, err: = ioutil.ReadFile (*name*) // Liest die gesamte Datei und gibt sie zurück. Ein Nicht-Null-Fehler wird zurückgegeben, wenn die gesamte Datei nicht gelesen werden konnte.
- err: = ioutil.WriteFile (*Name* , *Daten* , *Perm*) // Erzeugt oder überschreibt eine Datei mit den bereitgestellten Daten und den UNIX-Berechtigungsbits. Ein Nicht-Null-Fehler wird zurückgegeben, wenn nicht in die Datei geschrieben werden konnte.
- err: = os.Remove (*name*) // Löscht eine Datei. Ein Nicht-Null-Fehler wird zurückgegeben, wenn die Datei nicht gelöscht werden konnte.
- err: = os.RemoveAll (*name*) // Löscht eine Datei oder eine ganze Verzeichnishierarchie. Ein Nicht-Null-Fehler wird zurückgegeben, wenn die Datei oder das Verzeichnis nicht gelöscht werden konnte.
- err: = os.Rename (*oldName* , *newName*) // Benennt eine Datei um oder verschiebt sie (kann über mehrere Verzeichnisse hinweg sein). Ein Nicht-Null-Fehler wird zurückgegeben, wenn die Datei nicht verschoben werden konnte.

Parameter

Parameter	Einzelheiten
Name	Ein Dateiname oder ein Pfad des Typs string. Zum Beispiel: `"hello.txt"` .
Irr	Ein `error` Wenn nicht gleich `nil` , handelt es sich um einen Fehler, der beim Aufruf der Funktion aufgetreten ist.
Datei	Ein `*os.File` vom Typ `*os.File` von den mit der `os` Paketdatei verbundenen Funktionen zurückgegeben wird. Es implementiert einen `io.ReadWriter` , das heißt, Sie können `Read(data)` und `Write(data)` darauf aufrufen. Beachten Sie, dass diese Funktionen möglicherweise nicht in Abhängigkeit von den offenen Flags der Datei aufgerufen werden können.
Daten	Ein Byte-Byte (`[]byte`), das die Rohdaten einer Datei darstellt.
Dauerwelle	Die UNIX-Berechtigungsbits, die zum Öffnen einer Datei mit dem Typ

Parameter	Einzelheiten
	`os.FileMode` . Für die Verwendung von Berechtigungsbits stehen mehrere Konstanten zur Verfügung.
Flagge	Dateiöffnungs-Flags, die die Methoden bestimmen, die für den File-Handler des Typs `int` aufgerufen werden können. Für die Verwendung von Flags stehen mehrere Konstanten zur Verfügung. Dies sind: `os.O_RDONLY` , `os.O_WRONLY` , `os.O_RDWR` , `os.O_APPEND` , `os.O_CREATE` , `os.O_EXCL` , `os.O_SYNC` und `os.O_TRUNC` .

Examples

Lesen und Schreiben in eine Datei mit ioutil

Ein einfaches Programm, das "Hallo, Welt!" Schreibt. `test.txt` liest die Daten zurück und druckt sie aus. Veranschaulicht einfache Datei-E / A-Vorgänge.

```
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))
}
```

Auflisten aller Dateien und Ordner im aktuellen Verzeichnis

```
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())
    }
}
```

Alle Ordner im aktuellen Verzeichnis auflisten

```
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())
        }
    }
}
```

Datei I / O online lesen: https://riptutorial.com/de/go/topic/1033/datei-i---o

Kapitel 15: Der Go-Befehl

Einführung

Der Befehl `go` ist ein Befehlszeilenprogramm, mit dem die Go-Entwicklung verwaltet werden kann. Es ermöglicht das Erstellen, Ausführen und Testen von Code sowie eine Vielzahl anderer Aufgaben im Zusammenhang mit Go.

Examples

Geh Rennen

`go run` wird ein Programm ausgeführt, ohne eine ausführbare Datei zu erstellen. Meist nützlich für die Entwicklung. `run` führt nur Pakete aus, deren *Paketname* **main ist** .

Um dies zu demonstrieren, verwenden wir ein einfaches Hello World-Beispiel `main.go` :

```
package main

import fmt

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

Ausführen ohne zu einer Datei zu kompilieren:

```
go run main.go
```

Ausgabe:

```
Hello, World!
```

Führen Sie mehrere Dateien im Paket aus

Wenn das Paket **main ist** und in mehrere Dateien aufgeteilt ist, müssen die anderen Dateien in den `run` :

```
go run main.go assets.go
```

Bauen Sie auf

`go build` kompiliert ein Programm in eine ausführbare Datei.

Um dies zu demonstrieren, verwenden wir ein einfaches Hello World-Beispiel main.go:

```
package main

import fmt

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

Kompilieren Sie das Programm:

```
go build main.go
```

`build` erstellt ein ausführbares Programm, in diesem Fall: `main` oder `main.exe` . Sie können diese Datei dann ausführen, um die Ausgabe zu sehen. `Hello, World!` . Sie können es auch auf ein ähnliches System kopieren, auf dem Go nicht installiert ist *, es ausführbar machen* und dort ausführen.

Geben Sie das Betriebssystem oder die Architektur im Build an:

Sie können angeben, welches System oder welche Architektur erstellt werden soll, indem Sie die `env` vor dem `build env` :

```
env GOOS=linux go build main.go # builds for Linux
env GOARCH=arm go build main.go # builds for ARM architecture
```

Erstellen Sie mehrere Dateien

Wenn Ihr Paket in mehrere Dateien aufgeteilt ist **und** der Paketname **main** lautet (*dh es handelt sich nicht um ein importierbares Paket*), müssen Sie alle zu erstellenden Dateien angeben:

```
go build main.go assets.go # outputs an executable: main
```

Paket erstellen

Um ein Paket namens `main` erstellen, können Sie einfach Folgendes verwenden:

```
go build . # outputs an executable with name as the name of enclosing folder
```

Gehen Sie sauber

`go clean` bereinigt alle temporären Dateien, die beim Aufruf von `go build` auf einem Programm erstellt wurden. Es werden auch Dateien gelöscht, die von Makefiles übrig sind.

Go Fmt

`go fmt` formatiert den Quellcode eines Programms auf eine übersichtliche, idiomatische Weise, die leicht zu lesen und zu verstehen ist. Es wird empfohlen, dass Sie `go fmt` für jede Quelle verwenden, bevor Sie es zur öffentlichen Anzeige einreichen oder in ein Versionskontrollsystem einbinden.

So formatieren Sie eine Datei:

```
go fmt main.go
```

Oder alle Dateien in einem Verzeichnis:

```
go fmt myProject
```

Sie können auch `gofmt -s` (**nicht** `go fmt`) verwenden, um zu versuchen, den möglichen Code zu vereinfachen.

`gofmt` (**not** `go fmt`) kann auch zum Refactor-Code verwendet werden. Es versteht Go und ist daher mächtiger als das einfache Suchen und Ersetzen. Zum Beispiel bei diesem Programm (`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")
}
```

Sie können die Methode `Original` durch `Refactor` durch `gofmt` :

```
gofmt -r 'Original -> Refactor' -d main.go
```

Welches wird den Unterschied erzeugen:

```
-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")
 }
```

Geh holen

`go get` die mit den Importpfaden benannten Pakete zusammen mit ihren Abhängigkeiten heruntergeladen. Dann werden die benannten Pakete wie "go install" installiert. Get akzeptiert auch Build-Flags, um die Installation zu steuern.

> Gehen Sie auf github.com/maknahar/phonecountry

Beim Auschecken eines neuen Pakets erstellt get das Zielverzeichnis `$GOPATH/src/<import-path>` . Wenn der GOPATH mehrere Einträge enthält, verwendet get den ersten. Ebenso werden kompilierte Binärdateien in `$GOPATH/bin` installiert.

Beim Auschecken oder Aktualisieren eines Pakets sollten Sie nach einem Zweig oder Tag suchen, der der lokal installierten Version von Go entspricht. Die wichtigste Regel ist, dass bei einer lokalen Installation der Version "go1" nach einem Zweig oder Tag mit dem Namen "go1" gesucht wird. Wenn keine solche Version vorhanden ist, wird die neueste Version des Pakets abgerufen.

Wenn Sie `go get` , bewirkt das Flag `-d` dass das angegebene Paket heruntergeladen, jedoch nicht installiert wird. Mit der `-u` das Paket und seine Abhängigkeiten aktualisiert werden.

Holen Sie sich nie auschecken oder aktualisieren Sie Code, der in Anbieterverzeichnissen gespeichert ist.

Geh env

`go env [var ...]` druckt go-Umgebungsinformationen.

> Standardmäßig werden alle Informationen gedruckt.

`$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"
```

> Wenn ein oder mehrere Variablennamen als Argumente angegeben sind, wird der Wert jeder benannten Variablen in einer eigenen Zeile ausgegeben.

```
$go env GOOS GOPATH
```

```
darwin
/Users/vikashkv/work
```

Der Go-Befehl online lesen: https://riptutorial.com/de/go/topic/4828/der-go-befehl

Kapitel 16: E-Mails senden / empfangen

Syntax

- func PlainAuth (Identität, Benutzername, Passwort, Host-String) Auth
- func SendMail (addr string, a Auth, von string zu [] string, msg [] byte) fehler

Examples

Senden von E-Mails mit smtp.SendMail ()

Das Versenden von E-Mails ist in Go ziemlich einfach. Es hilft, den RFC 822 zu verstehen, der den Stil einer E-Mail angibt. Der folgende Code sendet eine RFC 822-kompatible E-Mail.

```
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!")
}
```

Das Obige wird eine Nachricht wie die folgende senden:

```
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!
.
```

E-Mails senden / empfangen online lesen: https://riptutorial.com/de/go/topic/5912/e-mails-senden--empfangen

Kapitel 17: Entwickeln für mehrere Plattformen mit bedingtem Kompilieren

Einführung

Das plattformbasierte bedingte Kompilieren gibt es in Go in zwei Formen, eines mit Dateiendungen und das andere mit Build-Tags.

Syntax

- Nach " `// +build` " kann eine einzelne Plattform oder eine Liste folgen
- Plattform kann zurückgestellt werden, indem Sie mit `!` Zeichen
- Liste der durch Leerzeichen getrennten Plattformen werden ODER-verknüpft

Bemerkungen

Vorsichtsmaßnahmen für Build-Tags:

- Die `// +build` Einschränkung muss vor der Paketklausel oben in der Datei stehen.
- Es muss eine Leerzeile folgen, um sich von den Paketkommentaren zu trennen.

Liste der gültigen Plattformen für Build-Tags und Dateisuffixe
Android
Darwin
Libelle
Freebsd
Linux
netbsd
openbsd
plan9
Solaris
Fenster

Die aktuellste Plattformliste finden Sie in der `$GOOS` Liste unter https://golang.org/doc/install/source#environment .

Examples

Tags erstellen

```
// +build linux

package lib

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

Negiere eine Plattform durch Platzieren `!` bevor:

```
// +build !windows

package lib

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

Die Liste der Plattformen kann angegeben werden, indem sie durch Leerzeichen getrennt werden

```
// +build linux darwin plan9

package lib

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

Dateiendung

Wenn Sie Ihre Datei `lib_linux.go` , wird der gesamte Inhalt dieser Datei nur in Linux-Umgebungen kompiliert:

```
package lib

var OnlyCompiledInLinux string
```

Definieren separater Verhaltensweisen auf verschiedenen Plattformen

Verschiedene Plattformen können separate Implementierungen derselben Methode haben. Dieses Beispiel zeigt auch, wie Build-Tags und Dateisuffixe zusammen verwendet werden können.

Datei `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")
}
```

Entwickeln für mehrere Plattformen mit bedingtem Kompilieren online lesen: https://riptutorial.com/de/go/topic/8599/entwickeln-fur-mehrere-plattformen-mit-bedingtem-kompilieren

Kapitel 18: Erste Schritte mit Go Atom verwenden

Einführung

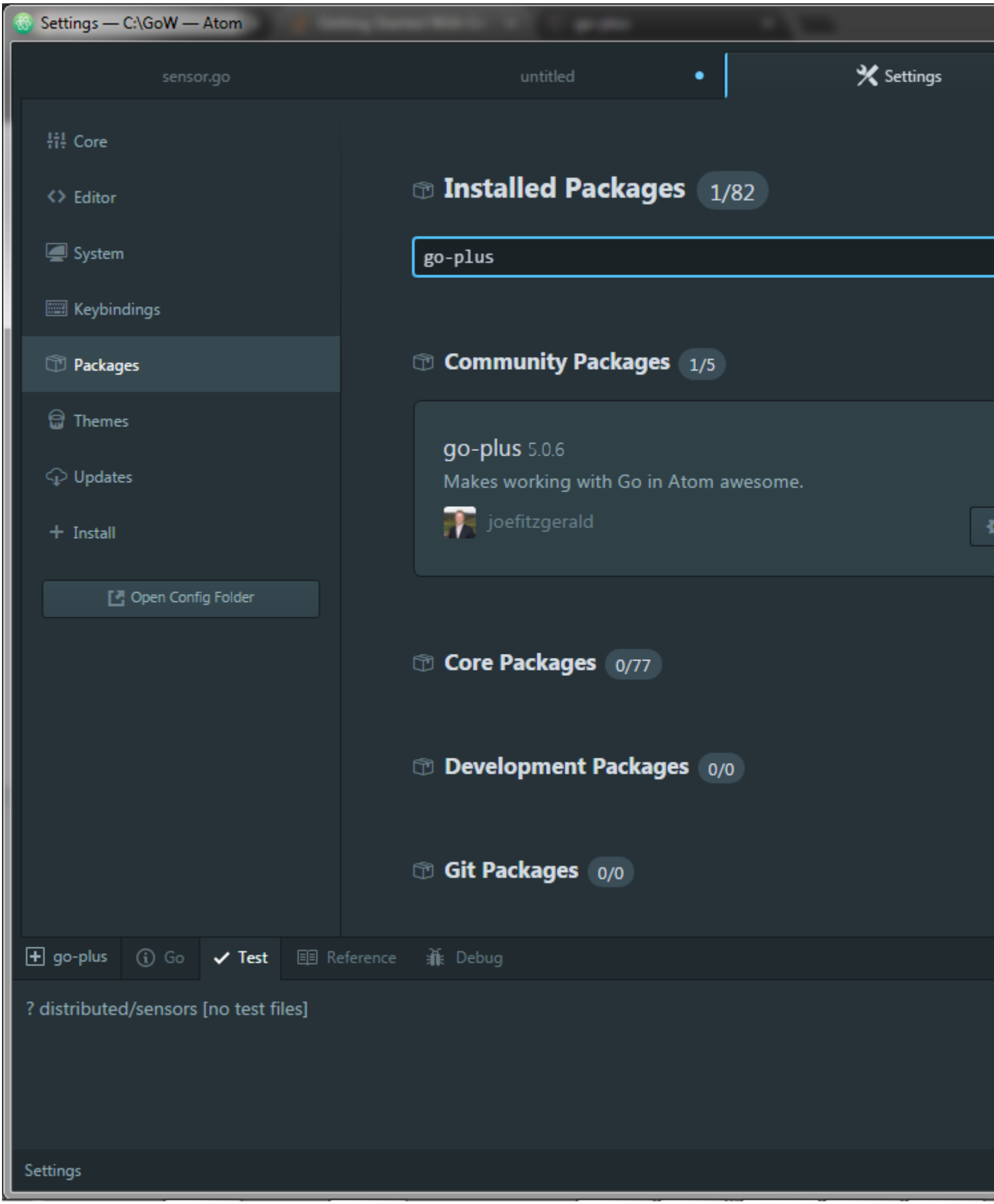
Nach der Installation von go (http://www.riptutorial.com/go/topic/198/getting-started-with-go) benötigen Sie eine Umgebung. Eine effiziente und kostenlose Möglichkeit, um loszulegen, ist der Atom-Texteditor (https://atom.io) und gulp. Eine Frage, die Ihnen vielleicht in den Sinn kam, ist, *warum Sie Schluck benutzen?* Wir brauchen Schluck für die automatische Vervollständigung. Lass uns anfangen!

Examples

Holen, installieren und installieren Sie Atom & Gulp

1. Installieren Sie Atom. Sie können Atom von hier bekommen
2. Gehen Sie zu den Atom-Einstellungen (Strg +,). Pakete -> Go-Plus-Paket installieren (Go-Plus)

Nach der Installation von go-plus in Atom:



3. Holen Sie sich diese Abhängigkeiten mit go get oder einem anderen Abhängigkeitsmanager: (Öffnen Sie eine Konsole und führen Sie diese Befehle aus.)

   Gehen Sie auf -u golang.org/x/tools/cmd/goimports

Gehen Sie auf -u golang.org/x/tools/cmd/gorename

Gehen Sie auf -u github.com/sqs/goreturns

Gehen Sie auf -u github.com/nsf/gocode

Gehen Sie auf -u github.com/alecthomas/gometalinter

Gehen Sie zu -u github.com/zmb3/gogetdoc

Gehen Sie auf -u github.com/rogpeppe/godef

Gehen Sie auf -u golang.org/x/tools/cmd/guru

4. Installieren Sie Gulp (Gulpjs) mit npm oder einem anderen Paketmanager (gulp-getting-started-doc):

   $ npm install --global gulp

Erstellen Sie $ GO_PATH / gulpfile.js

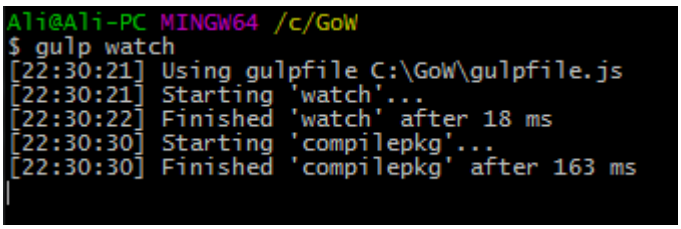
```
var gulp = require('gulp');
var path = require('path');
var shell = require('gulp-shell');

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']);
});
```

Im obigen Code haben wir eine *compliepkg*- Task definiert, die jedes Mal ausgelöst wird, wenn sich eine go-Datei in goPath (src / mypackage /) oder in Unterverzeichnissen ändert. Die Task führt den Shellbefehl go install changed_file.go aus

Nachdem Sie die gulp-Datei in go path erstellt und die Aufgabe definiert haben, öffnen Sie eine Befehlszeile und führen Sie Folgendes aus:

schluck zu sehen

Sie werden bei jeder Änderung der Datei so etwas sehen:

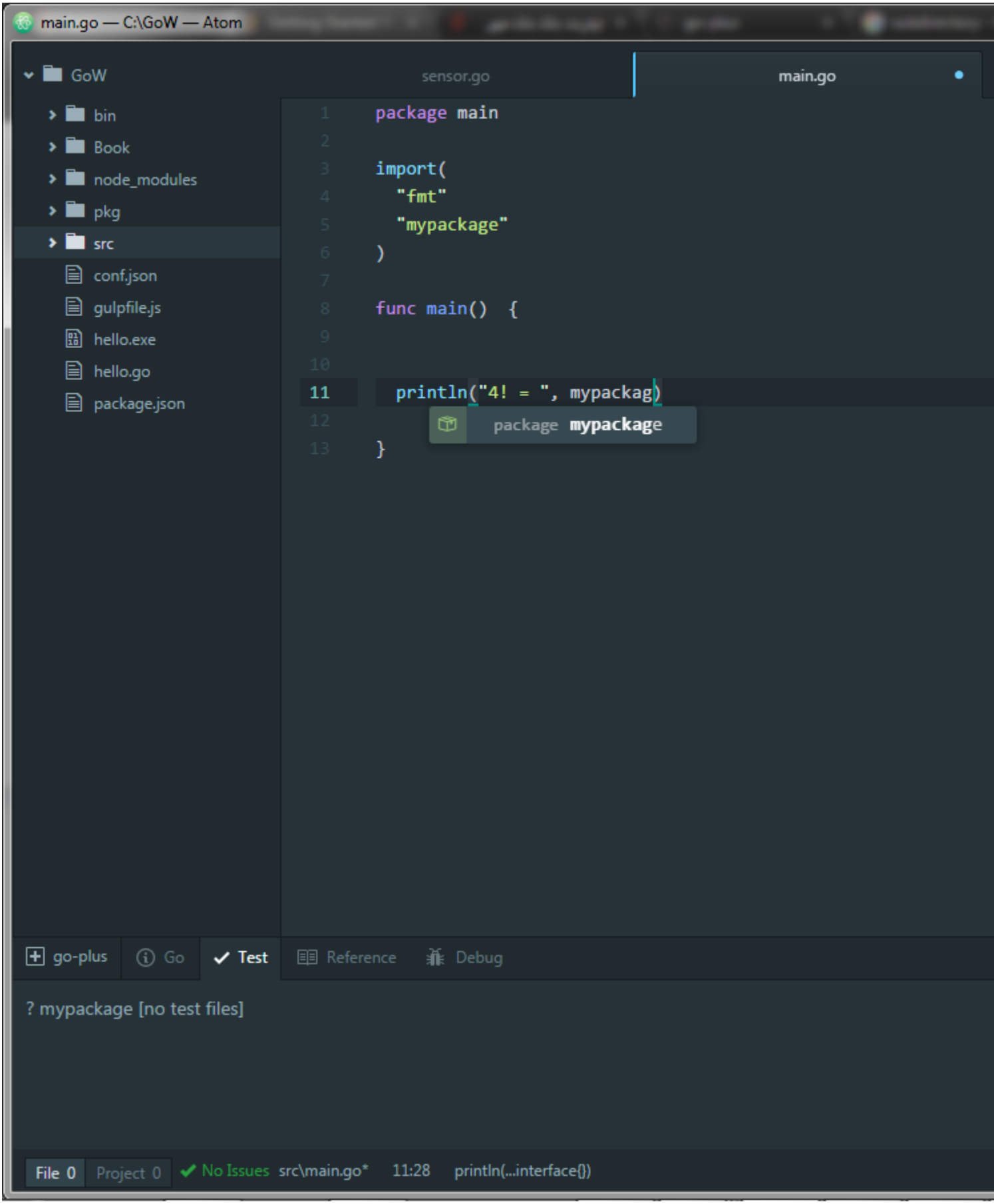


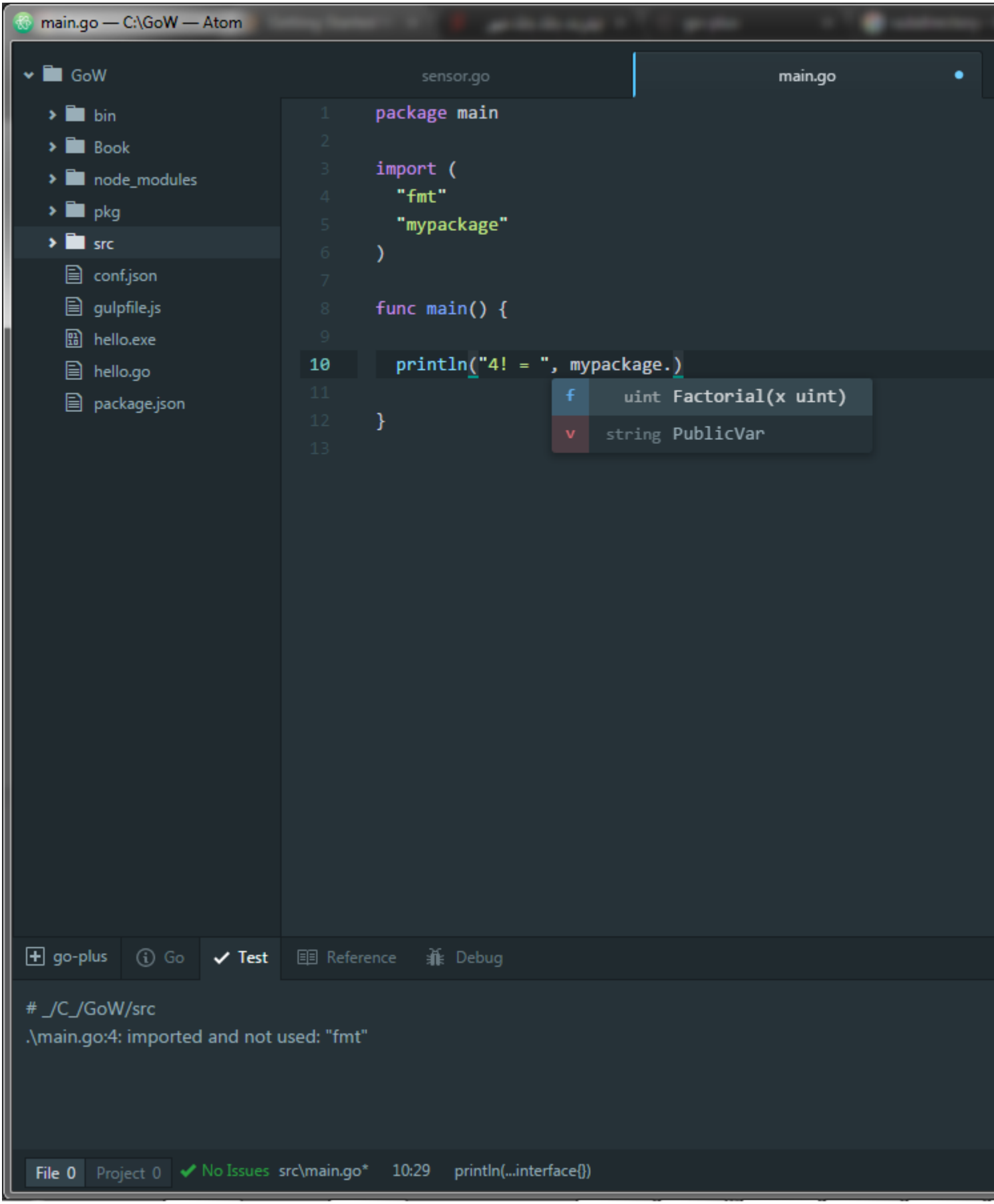
Erstellen Sie $ GO_PATH / mypackage / source.go

```
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)
}
```

$ GO_PATH / main.go erstellen

Jetzt können Sie mit Atom und Gulp Ihren eigenen Go-Code mit automatischer Vervollständigung schreiben:


main.go — C:\GoW — Atom
GoW
bin
Book
node_modules
pkg
src
conf.json
gulpfile.js
hello.exe
hello.go
package.json
sensor.go
main.go
1 package main
2
3 import(
4 "fmt"
5 "mypackage"
6)
7
8 func main() {
9
10
11 println("4! = ", mypackag)
12
13 }
package mypackage
go-plus
Go
Test
Reference
Debug
? mypackage [no test files]
File 0
Project 0
No Issues
src\main.go*
11:28
println(...interface{})




```
package main

import (
    "fmt"
```

```
    "mypackage"
)

func main() {

    println("4! = ", mypackage.Factorial(4))

}
```

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

Erste Schritte mit Go Atom verwenden online lesen: https://riptutorial.com/de/go/topic/8592/erste-schritte-mit-go-atom-verwenden

Kapitel 19: Fehlerbehandlung

Einführung

In Go werden unerwartete Situationen mit **Fehlern** und nicht mit Ausnahmen behandelt. Dieser Ansatz ist dem von C mit errno ähnlicher als der von Java oder anderen objektorientierten Sprachen mit ihren try / catch-Blöcken. Ein Fehler ist jedoch keine ganze Zahl, sondern eine Schnittstelle.

Eine Funktion, die möglicherweise fehlschlägt, gibt normalerweise einen **Fehler** als letzten Rückgabewert zurück. Wenn dieser Fehler nicht gleich **Null ist** , ist ein Fehler **aufgetreten** , und der Aufrufer der Funktion sollte entsprechend handeln.

Bemerkungen

Beachten Sie, wie in Go Sie keinen Fehler *machen.* Stattdessen *kehren* Sie einen Fehler im Fehlerfall.

Wenn eine Funktion fehlschlagen kann, ist der zuletzt zurückgegebene Wert im Allgemeinen ein `error` .

```
// 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

Fehlerwert erstellen

Die einfachste Methode zum Erstellen eines Fehlers ist die Verwendung des `errors` .

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

Wenn Sie einem Fehler zusätzliche Informationen hinzufügen möchten, bietet das `fmt` Paket auch eine nützliche Fehlererstellungsmethode:

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

Hier ist ein vollständiges Beispiel, bei dem der Fehler von einer Funktion zurückgegeben wird:

```
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")
}
```

Im Spielplatz öffnen

Erstellen eines benutzerdefinierten Fehlertyps

In Go wird ein Fehler durch einen beliebigen Wert dargestellt, der sich selbst als Zeichenfolge beschreiben kann. Jeder Typ, der die integrierte `error` implementiert, ist ein Fehler.

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

Das folgende Beispiel zeigt, wie Sie einen neuen Fehlertyp mithilfe eines zusammengesetzten Stringliteral definieren.

```
// 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)
}
```

Ich kann jetzt meinen benutzerdefinierten Fehlertyp als Fehler verwenden:

```
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
}
```

Fehler zurückgeben

In Go *erheben* Sie keinen Fehler. Stattdessen *kehren* Sie einen `error` im Fehlerfall.

```
// 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
}
```

Wenn die Methode mehrere Werte zurückgibt (und die Ausführung fehlschlagen kann), lautet die Standardkonvention, den Fehler als letztes Argument zurückzugeben.

```
// 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)
}
```

Fehler behandeln

In Go können Fehler von einem Funktionsaufruf zurückgegeben werden. Die Konvention ist, dass, wenn eine Methode fehlschlagen kann, das letzte zurückgegebene Argument ein `error` .

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

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

Sie verwenden mehrere Variablenzuweisungen, um zu überprüfen, ob die Methode fehlgeschlagen ist.

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

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

Wenn Sie nicht an dem Fehler interessiert sind, können Sie ihn einfach ignorieren, indem Sie ihn _ zuweisen.

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

Das Ignorieren eines Fehlers kann natürlich schwerwiegende Folgen haben. Daher wird dies generell nicht empfohlen.

Wenn Sie über mehrere Methodenaufrufe verfügen und eine oder mehrere Methoden in der Kette möglicherweise einen Fehler zurückgeben, sollten Sie den Fehler an die erste Ebene weiterleiten,

die ihn behandeln kann.

```
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
}
```

Sich von der Panik erholen

Ein häufiger Fehler ist das Deklarieren eines Slice und das Anfordern von Indizes ohne Initialisierung, was zu einer Panik "Index außerhalb des Bereichs" führt. Im folgenden Code wird erläutert, wie Sie die Panik beheben können, ohne das Programm zu beenden. Dies ist das normale Verhalten bei einer Panik. In den meisten Fällen ist es nur für Entwicklungs- oder Testzwecke sinnvoll, einen Fehler auf diese Weise zurückzugeben, anstatt das Programm aus Panikgründen zu beenden.

```
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)
        }
    }
}
```

Die Verwendung einer separaten Funktion (und nicht der Schließung) ermöglicht die Wiederverwendung derselben Funktion in anderen Funktionen, die zu Panik neigen.

Fehlerbehandlung online lesen: https://riptutorial.com/de/go/topic/785/fehlerbehandlung

Kapitel 20: Fmt

Examples

Stringer

Die `fmt.Stringer` Schnittstelle erfordert eine einzige Methode, `String() string` , um erfüllt zu werden. Die String-Methode definiert das "native" String-Format für diesen Wert und ist die Standarddarstellung, wenn der Wert für eine der Formatierungs- oder Druckroutinen des `fmt` Pakets bereitgestellt wird.

```
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

Paket fmt Geräte formatiert I / O mit Format *Verben:*

```
%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
```

Formatierungsfunktionen

Es gibt **4** Hauptfunktionstypen in `fmt` und verschiedene Varianten.

Drucken

```
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` kann innerhalb von `http` `Fprint` verwendet werden:

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

Scan

Scan scannt Text, der von der Standardeingabe gelesen wurde.

```
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-Schnittstelle

Jeder Wert, der über eine `String()` Methode verfügt, implementiert den `fmt` **inteface-** `Stringer`

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

Fmt online lesen: https://riptutorial.com/de/go/topic/2938/fmt

Kapitel 21: Funktionen

Einführung

Funktionen in Go stellen organisierten, wiederverwendbaren Code bereit, um eine Reihe von Aktionen auszuführen. Funktionen vereinfachen den Codierungsprozess, verhindern redundante Logik und machen Code leichter nachvollziehbar. In diesem Thema werden die Deklaration und Verwendung von Funktionen, Argumenten, Parametern, Rückgabeanweisungen und Gültigkeitsbereichen in Go beschrieben.

Syntax

- func () // Funktionstyp ohne Argumente und ohne Rückgabewert
- func (x int) int // akzeptiert eine ganze Zahl und gibt eine ganze Zahl zurück
- func (a, b int, z float32) bool // akzeptiert 2 ganze Zahlen, einen float und gibt einen booleschen Wert zurück
- func (Präfixzeichenfolge, Werte ... int) // "variadic" -Funktion, die eine Zeichenfolge und eine oder mehrere ganze Zahlen akzeptiert
- func () (int, bool) // Funktion, die zwei Werte zurückgibt
- func (a, b int, z float64, opt ... interface {}) (success bool) // akzeptiert 2 ganze Zahlen, einen float und eine oder mehrere Schnittstellen und gibt den genannten booleschen Wert zurück (der innerhalb der Funktion bereits initialisiert ist)

Examples

Grundlegende Erklärung

Eine einfache Funktion, die keine Parameter akzeptiert und keine Werte zurückgibt:

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

Parameter

Eine Funktion kann optional einen Parametersatz deklarieren:

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

Beachten Sie, dass der Typ für `firstName` wird, da er mit `lastName` identisch `lastName` .

Rückgabewerte

Eine Funktion kann einen oder mehrere Werte an den Aufrufer zurückgeben:

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

Der zweite Rückgabewert kann auch der Fehler var sein:

```
import errors

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

Zwei wichtige Dinge müssen beachtet werden:

- Die Klammer kann für einen einzelnen Rückgabewert weggelassen werden.
- Jede `return` Anweisung muss einen Wert für **alle** deklarierten Rückgabewerte bereitstellen.

Benannte Rückgabewerte

Rückgabewerte können einer lokalen Variablen zugewiesen werden. Eine leere `return` Anweisung kann dann verwendet werden, um ihre aktuellen Werte zurückzugeben. Dies wird als *"nackte"* Rückkehr bezeichnet. Naked return-Anweisungen sollten nur in kurzen Funktionen verwendet werden, da sie die Lesbarkeit in längeren Funktionen beeinträchtigen:

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

spiele es auf dem Spielplatz

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

spiele es auf dem Spielplatz

Zwei wichtige Dinge müssen beachtet werden:

- Die Klammern um die Rückgabewerte sind **obligatorisch** .
- Es muss immer eine leere `return` Anweisung angegeben werden.

Buchstäbliche Funktionen und Schließungen

Eine einfache wörtliche Funktion, die `Hello!` zu stdout:

```
package main

import "fmt"

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

spiele es auf dem Spielplatz

Eine literale Funktion, die das Argument `str` nach stdout druckt:

```
package main

import "fmt"

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

spiele es auf dem Spielplatz

Eine literale Funktion, die die Variable `str` :

```
package main

import "fmt"

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

spiele es auf dem Spielplatz

Es ist möglich, einer Variablen eine Literalfunktion zuzuweisen:

```
package main

import (
    "fmt"
```

```
)

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

spiele es auf dem Spielplatz

Variadische Funktionen

Eine variadische Funktion kann mit einer beliebigen Anzahl von **nachlaufenden** Argumenten aufgerufen werden. Diese Elemente werden in einem Slice gespeichert.

```
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")
}
```

spiele es auf dem Spielplatz

Sie können auch eine Variadic-Funktion mit einem Slice versehen, mit . . . :

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

     variadic(strs...)
}
```

spiele es auf dem Spielplatz

Funktionen online lesen: https://riptutorial.com/de/go/topic/373/funktionen

Kapitel 22: gob

Einführung

Gob ist eine Go-spezifische Serialisierungsmethode. Es unterstützt alle Go-Datentypen mit Ausnahme von Kanälen und Funktionen. Gob verschlüsselt die Typinformationen auch in die serialisierte Form. Anders als bei XML ist es, dass es wesentlich effizienter ist.

Die Einbeziehung von Typinformationen macht das Kodieren und Dekodieren gegenüber den Unterschieden zwischen Kodierer und Dekodierer ziemlich robust.

Examples

Wie kodiere ich Daten und schreibe mit Gob in eine Datei

```
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)

}
```

Wie kann ich Daten aus einer Datei lesen und mit go decodieren?

```
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)

}
```

Wie kodiere ich eine Schnittstelle mit 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)
    }

}
```

Wie dekodiere ich eine Schnittstelle mit 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

}
```

gob online lesen: https://riptutorial.com/de/go/topic/8820/gob

Kapitel 23: Goroutinen

Einführung

Eine Goroutine ist ein einfacher Thread, der von der Go-Laufzeitumgebung verwaltet wird.

gehe f (x, y, z)

startet eine neue Goroutine

f (x, y, z)

Die Bewertung von f, x, y und z erfolgt in der aktuellen Goroutine und die Ausführung von f in der neuen Goroutine.

Goroutinen laufen in demselben Adressraum, daher muss der Zugriff auf den gemeinsam genutzten Speicher synchronisiert werden. Das Sync-Paket bietet nützliche Grundelemente, obwohl Sie sie in Go nicht viel benötigen, da es andere Grundelemente gibt.

Referenz: https://tour.golang.org/concurrency/1

Examples

Goroutinen-Grundprogramm

```
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")
}
```

Eine Goroutine ist eine Funktion, die gleichzeitig mit anderen Funktionen ausgeführt werden kann. Um eine Goroutine zu erstellen, verwenden wir das Schlüsselwort `go` gefolgt von einem Funktionsaufruf:

```
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)
}
```

Im Allgemeinen führt der Funktionsaufruf alle Anweisungen innerhalb des Funktionskörpers aus und kehrt zur nächsten Zeile zurück. Bei Goroutines kehren wir jedoch sofort zur nächsten Zeile zurück, da sie nicht auf die Ausführung der Funktion warten. So ist ein Aufruf einer `Scanln` Funktion enthalten, ansonsten wurde das Programm ohne Drucken der Zahlen beendet.

Goroutinen online lesen: https://riptutorial.com/de/go/topic/9776/goroutinen

Kapitel 24: HTTP-Client

Syntax

- resp. err: = http.Get (url) // Erstellt eine HTTP-GET-Anforderung mit dem Standard-HTTP-Client. Wenn die Anforderung fehlschlägt, wird ein Nicht-Null-Fehler zurückgegeben.
- resp. err: = http.Post (URL, bodyType, body) // Erstellt eine HTTP-POST-Anforderung mit dem Standard-HTTP-Client. Wenn die Anforderung fehlschlägt, wird ein Nicht-Null-Fehler zurückgegeben.
- resp. err: = http.PostForm (URL, Werte) // Erstellt eine HTTP-Formular-POST-Anforderung mit dem Standard-HTTP-Client. Wenn die Anforderung fehlschlägt, wird ein Nicht-Null-Fehler zurückgegeben.

Parameter

Parameter	Einzelheiten
bzw	Eine Antwort vom Typ `*http.Response` auf eine HTTP-Anfrage
Irr	Ein `error` Wenn nicht gleich null, stellt dies einen Fehler dar, der beim Aufruf der Funktion aufgetreten ist.
URL	Eine URL vom Typ `string` an die eine HTTP-Anforderung gesendet werden soll.
Körpertyp	Der MIME-Typ der `string` der Body-Payload einer POST-Anforderung.
Karosserie	Ein `io.Reader` (implementiert `Read()`), aus dem gelesen wird, bis ein Fehler erreicht ist, der als Body-Payload einer POST-Anforderung übermittelt wird.
Werte	Eine Schlüsselwertzuordnung des Typs `url.Values` . Der zugrunde liegende Typ ist eine `map[string][]string` .

Bemerkungen

Es ist wichtig, `defer resp.Body.Close()` nach jeder HTTP-Anforderung zu `defer resp.Body.Close()` , die keinen Nicht-Null-Fehler `defer resp.Body.Close()` Ressourcen verloren.

Examples

Basic GET

Führen Sie eine grundlegende GET-Anforderung aus und drucken Sie den Inhalt einer 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 mit URL-Parametern und einer JSON-Antwort

Eine Anforderung für die Top 10 der zuletzt aktiven StackOverflow-Posts unter Verwendung der 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)
    }
}
```

Timeout-Anforderung mit Kontext

1,7+

Das Zeitlimit für eine HTTP-Anforderung mit einem Kontext kann nur mit der Standardbibliothek (nicht mit den untergeordneten Ordnern) in 1.7+ ausgeführt werden:

```
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()
```

Vor 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()
```

Lesen Sie weiter

Weitere Informationen zum `context` Sie unter Kontext .

PUT-Anforderung eines JSON-Objekts

Folgendes aktualisiert ein Benutzerobjekt über eine PUT-Anforderung und gibt den Statuscode der Anforderung aus:

```
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)
}
```

HTTP-Client online lesen: https://riptutorial.com/de/go/topic/1422/http-client

Kapitel 25: HTTP-Server

Bemerkungen

`http.ServeMux` bietet einen Multiplexer, der Handler für HTTP-Anforderungen aufruft.

Alternativen zum Standard-Bibliotheksmultiplexer sind:

- Gorilla Mux

Examples

HTTP Hello World mit benutzerdefiniertem Server und 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())
}
```

Drücken Sie `Strg` + `C` , um den Vorgang zu stoppen.

Hallo Welt

Das Schreiben von Webservern in Golang beginnt in der Regel mit dem Standardmodul `net/http` .

Es gibt auch eine Anleitung dazu hier .

Der folgende Code verwendet es auch. Hier ist die einfachste mögliche HTTP-Server-Implementierung. Es antwortet `"Hello World"` auf jede HTTP-Anfrage.

Speichern Sie den folgenden Code in einer `server.go` Datei in Ihren Arbeitsbereichen.

```
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))
}
```

Sie können den Server folgendermaßen ausführen:

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

Oder Sie können kompilieren und ausführen.

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

Der Server überwacht den angegebenen Port (`:8080`). Sie können es mit jedem HTTP-Client testen. Hier ist ein Beispiel mit `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!
```

Drücken Sie `Strg` + `C` , um den Vorgang zu stoppen.

Verwenden einer Handlerfunktion

`HandleFunc` registriert die Handler-Funktion für das angegebene Muster im Server-Multiplexer (Router).

Sie können eine anonyme Funktion definieren, wie wir es im einfachen *Hello World*- Beispiel gesehen haben:

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

Wir können aber auch einen `HandlerFunc` Typ übergeben. Mit anderen Worten, wir können jede Funktion übergeben, die die folgende Signatur berücksichtigt:

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

Wir können das vorherige Beispiel neu schreiben, indem wir die Referenz an eine zuvor definierte `HandlerFunc` . Hier ist das vollständige Beispiel:

```
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))
}
```

Natürlich können Sie mehrere Funktionshandler für verschiedene Pfade definieren.

```
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))
}
```

Hier ist die Ausgabe mit `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
```

Erstellen Sie einen HTTPS-Server

Generieren Sie ein Zertifikat

Um einen HTTPS-Server betreiben zu können, ist ein Zertifikat erforderlich. Das Generieren eines selbstsignierten Zertifikats mit `openssl` erfolgt durch Ausführen dieses Befehls:

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

Die Parameter sind:

- `req` Verwenden Sie das Zertifikatanforderungstool
- `x509` Erstellt ein selbstsigniertes Zertifikat
- `newkey rsa:4096` Erstellt einen neuen Schlüssel und ein neues Zertifikat unter Verwendung der RSA-Algorithmen mit einer Schlüssellänge von `4096` Bit
- `sha256` die SHA256-Hash-Algorithmen, die von großen Browsern als sicher eingestuft werden (Stand: 2017).
- `nodes` Deaktiviert den Kennwortschutz für den privaten Schlüssel. Ohne diesen Parameter musste Ihr Server Sie bei jedem Start nach dem Kennwort fragen.
- `keyout` die Datei an, wo der Schlüssel geschrieben werden soll
- `out` Benennt die Datei, in die das Zertifikat geschrieben werden soll
- `subj` Definiert den Domainnamen, für den dieses Zertifikat gültig ist
- `days` Wie viele Tage sollte dieses Zertifikat gültig sein? `3650` sind ca. 10 Jahre.

Hinweis: Ein selbstsigniertes Zertifikat kann z. B. für interne Projekte, für das Debuggen, Testen usw. verwendet werden. In jedem Browser wird angegeben, dass dieses Zertifikat nicht sicher ist. Um dies zu vermeiden, muss das Zertifikat von einer Zertifizierungsstelle unterzeichnet werden.

Meist ist dies nicht kostenlos verfügbar. Eine Ausnahme ist die Bewegung "Let's Encrypt": https://letsencrypt.org

Der notwendige Go-Code

Sie können die Konfiguration von TLS für den Server mit dem folgenden Code ausführen. `cert.pem` und `key.pem` sind Ihr SSL-Zertifikat und der Schlüssel, der mit dem obigen Befehl generiert wurde.

```
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))
}
```

Antworten auf eine HTTP-Anforderung mithilfe von Vorlagen

Antworten können mithilfe von Vorlagen in Go in einen `http.ResponseWriter` werden. Dies ist ein praktisches Werkzeug, wenn Sie dynamische Seiten erstellen möchten.

(Informationen zur Funktionsweise von Vorlagen in Go finden Sie auf der Seite zur Dokumentation von Go-Vorlagen .)

Fahren Sie mit einem einfachen Beispiel fort, um die `html/template` zu verwenden, um auf eine HTTP-Anforderung zu antworten:

```
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)
    }
}
```

Wo, der Inhalt von

1. `welcomeform.html` sind:

```
<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` sind:

```
<head>
    <title> Greetings, {{.Name}} </title>
</head>
<body>
    Greetings, {{.Name}}.<br>
    We know you are a {{.nationality}}!
</body>
```

Hinweis:

1. Stellen Sie sicher, dass sich die `.html` Dateien im richtigen Verzeichnis befinden.
2. Wenn `http://localhost:8080/` nach dem Start des Servers `http://localhost:8080/` kann.
3. Wie man nach dem Absenden des Formulars sehen kann, konnte das nicht *exportierte* Feld "Nationalität" der Struktur nicht wie erwartet vom *Musterpaket* analysiert werden.

Inhalte mit ServeMux bereitstellen

Ein einfacher statischer Dateiserver würde folgendermaßen aussehen:

```
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)
}
```

Behandlung der http-Methode, Zugriff auf Abfragezeichenfolgen und Anfragetext

Im Folgenden finden Sie ein einfaches Beispiel für einige allgemeine Aufgaben, die sich auf die Entwicklung einer API beziehen, wobei zwischen der HTTP-Methode der Anforderung, dem Zugriff auf Abfragezeichenfolgen und dem Zugriff auf den Anfragetext unterschieden wird.

Ressourcen

- http.Handler-Schnittstelle
- http.ResponseWriter
- http.Anfrage
- Verfügbare Methoden- und Statuskonstanten

```
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))
}
```

Muster-Curl-Ausgabe:

```
$ 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
```

HTTP-Server online lesen: https://riptutorial.com/de/go/topic/756/http-server

Kapitel 26: Inline-Erweiterung

Bemerkungen

Inline-Erweiterung ist eine häufige Optimierung in kompiliertem Code, bei der die Leistung der binären Größe vorrangig zugeordnet wurde. Der Compiler kann einen Funktionsaufruf durch den eigentlichen Rumpf der Funktion ersetzen. effektiv Code kopieren / einfügen von einer Stelle an eine andere zur Kompilierzeit. Da die Aufrufstelle so erweitert ist, dass sie nur die Maschinenbefehle enthält, die der Compiler für die Funktion generiert hat, müssen wir kein CALL oder PUSH (das x86-Äquivalent einer GOTO-Anweisung oder eines Stack-Frame-Pushs) oder das entsprechende Äquivalent auf einem anderen ausführen Architekturen.

Der Inliner entscheidet, ob eine Funktion basierend auf einer Reihe von Heuristiken inline gesetzt werden soll oder nicht, im Allgemeinen jedoch standardmäßig Inline. Da der Inliner Funktionsaufrufe loswird, kann er effektiv entscheiden, wo der Scheduler eine Goroutine ablehnen darf.

Funktionsaufrufe werden nicht eingebettet, wenn eine der folgenden Bedingungen zutrifft (es gibt viele andere Gründe, diese Liste ist unvollständig):

- Funktionen sind variadisch (zB haben sie `...` args)
- Funktionen haben eine "max hairyness", die über dem Budget liegt (sie rezyklieren zu stark oder können aus einem anderen Grund nicht analysiert werden)
- Sie enthalten `panic` , `recover` sich oder `defer`

Examples

Deaktivieren der Inline-Erweiterung

Die Inline-Erweiterung kann mit dem `go:noinline` Pragma `go:noinline` . Wenn wir beispielsweise das folgende einfache Programm erstellen:

```
package main

func printhello() {
    println("Hello")
}

func main() {
    printhello()
}
```

wir erhalten eine Ausgabe, die so aussieht (auf Lesbarkeit getrimmt):

```
$ 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:0xfffffff8, 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.printnl(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
…
```

Beachten Sie, dass es keinen `CALL printhello` an `printhello` . Wenn wir dann das Programm mit dem Pragma erstellen, dann:

```
package main

//go:noinline
func printhello() {
    println("Hello")
}

func main() {
    printhello()
}
```

Die Ausgabe enthält die printhello-Funktion und einen `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:0xfffffff8, 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:0xfffffff8, 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
…
```

Inline-Erweiterung online lesen: https://riptutorial.com/de/go/topic/2718/inline-erweiterung

Kapitel 27: Installation

Examples

Installieren Sie unter Linux oder Ubuntu

```
$ sudo apt-get update
$ sudo apt-get install -y build-essential git curl wget
$ wget https://storage.googleapis.com/golang/go<versions>.gz
```

Die Versionslisten finden Sie hier .

```
# 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
```

`$GOPATH`

```
$ mkdir $HOME/go
```

Fügen Sie am Ende der Datei ~ / .bashrc die folgenden beiden Zeilen ein

```
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
```

Jetzt können Sie loslegen. Testen Sie Ihre Go-Version mit:

```
$ go version
go version go<version> linux/amd64
```

Installation online lesen: https://riptutorial.com/de/go/topic/5776/installation

Kapitel 28: Installation

Bemerkungen

Go herunterladen

Besuchen Sie die Download-Liste und finden Sie das richtige Archiv für Ihr Betriebssystem. Die Namen dieser Downloads können für neue Benutzer etwas kryptisch sein.

Die Namen haben das Format go [Version]. [Betriebssystem] - [Architektur]. [Archiv].

Für die Version möchten Sie die neueste verfügbare Version auswählen. Dies sollten die ersten Optionen sein, die Sie sehen.

Für das Betriebssystem ist dies ziemlich selbsterklärend, außer für Mac-Benutzer, bei denen das Betriebssystem "Darwin" genannt wird. Dies ist nach dem Open-Source-Teil des Betriebssystems benannt, das von Mac-Computern verwendet wird .

Wenn Sie einen 64-Bit-Computer ausführen (der bei modernen Computern am häufigsten vorkommt), sollte der "Architektur" -Teil des Dateinamens "amd64" sein. Für 32-Bit-Maschinen wird es "386" sein. Wenn Sie sich auf einem ARM-Gerät wie einem Raspberry Pi befinden, benötigen Sie "armv6l".

Für den "Archiv" -Teil haben Mac- und Windows-Benutzer zwei Optionen, da Go Installationsprogramme für diese Plattformen bereitstellt. Für Mac möchten Sie wahrscheinlich "pkg". Für Windows möchten Sie wahrscheinlich "msi".

Wenn ich beispielsweise auf einem 64-Bit-Windows-Computer bin und Go 1.6.3 herunterladen möchte, wird der gewünschte Download genannt:

```
go1.6.3.windows-amd64.msi
```

Downloaden Sie die Dateien

Nachdem wir ein Go-Archiv heruntergeladen haben, müssen wir es irgendwo extrahieren.

Mac und Windows

Da Installer für diese Plattformen bereitgestellt werden, ist die Installation einfach. Führen Sie einfach das Installationsprogramm aus und akzeptieren Sie die Standardeinstellungen.

Linux

Es gibt kein Installationsprogramm für Linux, daher ist noch etwas Arbeit erforderlich. Sie sollten

eine Datei mit dem Suffix ".tar.gz" heruntergeladen haben. Dies ist eine Archivdatei, ähnlich einer ".zip" -Datei. Wir müssen es extrahieren. Wir extrahieren die Go-Dateien nach `/usr/local` da dies der empfohlene Speicherort ist.

Öffnen Sie ein Terminal und wechseln Sie in das Verzeichnis, in das Sie das Archiv heruntergeladen haben. Dies ist wahrscheinlich in `Downloads` . Wenn nicht, ersetzen Sie das Verzeichnis im folgenden Befehl entsprechend.

```
cd Downloads
```

Führen Sie nun das folgende aus, um das Archiv in `/usr/local` zu extrahieren, wobei `[filename]` durch den Namen der heruntergeladenen Datei ersetzt wird.

```
tar -C /usr/local -xzf [filename].tar.gz
```

Umgebungsvariablen einstellen

Es ist noch ein weiterer Schritt, bevor Sie mit der Entwicklung beginnen können. Wir müssen Umgebungsvariablen festlegen, dh Informationen, die Benutzer ändern können, um den Programmen eine bessere Vorstellung von den Einstellungen des Benutzers zu vermitteln.

Windows

Sie müssen den `GOPATH` , den Ordner, in dem Sie Go work `GOPATH` .

Sie können Umgebungsvariablen über die Schaltfläche "Umgebungsvariablen" auf der Registerkarte "Erweitert" der Systemsteuerung "System" einstellen. Einige Windows-Versionen bieten diese Systemsteuerung über die Option "Erweiterte Systemeinstellungen" in der Systemsteuerung "System" an.

Der Name Ihrer neuen Umgebungsvariablen sollte "GOPATH" sein. Der Wert sollte der vollständige Pfad zu einem Verzeichnis sein, in dem Sie Go-Code entwickeln. Ein Ordner mit dem Namen "go" in Ihrem Benutzerverzeichnis ist eine gute Wahl.

Mac

Sie müssen den `GOPATH` , den Ordner, in dem Sie Go work `GOPATH` .

Bearbeiten Sie eine Textdatei mit dem Namen ".bash_profile", die sich in Ihrem Benutzerverzeichnis befinden sollte, und fügen Sie am Ende die folgende neue Zeile ein. Ersetzen Sie `[work area]` durch einen vollständigen Pfad zu einem Verzeichnis, in dem Sie arbeiten möchten. Wenn ".bash_profile" ist nicht vorhanden, erstellen Sie es. Ein Ordner mit dem Namen "go" in Ihrem Benutzerverzeichnis ist eine gute Wahl.

```
export GOPATH=[work area]
```

Linux

Da Linux keinen Installer hat, ist etwas mehr Arbeit erforderlich. Wir müssen dem Terminal zeigen, wo sich der Go-Compiler und andere Tools befinden, und wir müssen den `GOPATH` , einen Ordner, in dem Sie Go arbeiten.

Bearbeiten Sie eine Textdatei mit dem Namen ".profile", die sich in Ihrem Benutzerverzeichnis befinden sollte, und fügen Sie am Ende die folgende Zeile ein. Ersetzen Sie `[work area]` durch einen vollständigen Pfad zu einem Verzeichnis, in dem Sie arbeiten möchten. Wenn " .profile "existiert nicht, erstellen Sie es. Ein Ordner mit dem Namen "go" in Ihrem Benutzerverzeichnis ist eine gute Wahl.

Fügen Sie dann in einer anderen neuen Zeile der Datei ".profile" Folgendes hinzu.

```
export PATH=$PATH:/usr/local/go/bin
```

Fertig!

Wenn die Go-Tools im Terminal immer noch nicht verfügbar sind, schließen Sie das Fenster und öffnen Sie ein neues Terminalfenster.

Examples

Beispiel .profile oder .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
```

Installation online lesen: https://riptutorial.com/de/go/topic/6213/installation

Kapitel 29: Jota

Einführung

Iota bietet eine Möglichkeit, numerische Konstanten aus einem Startwert zu deklarieren, der monoton wächst. Iota kann verwendet werden, um Bitmasken zu deklarieren, die häufig in der System- und Netzwerkprogrammierung verwendet werden, sowie andere Listen von Konstanten mit zugehörigen Werten.

Bemerkungen

Der `iota` Bezeichner wird verwendet, um Listen von Konstanten Werte zuzuweisen. Wenn iota in einer Liste verwendet wird, beginnt es mit einem Wert von Null und wird für jeden Wert in der Liste der Konstanten um eins erhöht und für jedes `const` Schlüsselwort zurückgesetzt. Im Gegensatz zu den Aufzählungen anderer Sprachen kann iota in Ausdrücken (z. B. `iota + 1`) verwendet werden, was eine größere Flexibilität ermöglicht.

Examples

Einfache Verwendung von Iota

Um eine Liste von Konstanten zu erstellen, weisen Sie jedem Element einen `iota` Wert zu:

```
const (
  a = iota // a = 0
  b = iota // b = 1
  c = iota // c = 2
)
```

Um eine Liste von Konstanten auf eine verkürzte Weise zu erstellen, weisen Sie dem ersten Element einen `iota` Wert zu:

```
const (
  a = iota // a = 0
  b        // b = 1
  c        // c = 2
)
```

Verwenden von Iota in einem Ausdruck

`iota` kann in Ausdrücken verwendet werden. `iota` kann es auch verwendet werden, um andere Werte als einfache inkrementierende Ganzzahlen ab Null zuzuweisen. Um Konstanten für SI-Einheiten zu erstellen, verwenden Sie dieses Beispiel aus 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
)
```

Werte überspringen

Der Wert von `iota` wird weiterhin für jeden Eintrag in einer konstanten Liste erhöht, auch wenn Iota nicht verwendet wird:

```
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
)
```

es wird auch dann inkrementiert, wenn überhaupt keine Konstante erstellt wird, dh der leere Bezeichner kann zum vollständigen Überspringen von Werten verwendet werden

```
const (
  a = iota // a = 0
  _        // iota is incremented
  b        // b = 2
)
```

Der erste Codeblock wurde der Go-Spezifikation (CC-BY 3.0) entnommen.

Verwendung von Iota in einer Ausdrucksliste

Da `iota` nach jeder `ConstSpec` inkrementiert `ConstSpec` , haben Werte in derselben Ausdrucksliste denselben Wert für `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
)
```

Dieses Beispiel wurde der Go-Spezifikation (CC-BY 3.0) entnommen.

Verwendung von Iota in einer Bitmaske

Iota kann beim Erstellen einer Bitmaske sehr nützlich sein. Um beispielsweise den Status einer

Netzwerkverbindung darzustellen, die sicher, authentifiziert und / oder betriebsbereit sein kann, erstellen wir möglicherweise eine Bitmaske wie die folgende:

```
const (
    Secure = 1 << iota // 0b001
    Authn              // 0b010
    Ready              // 0b100
)

ConnState := Secure|Authn // 0b011: Connection is secure and authenticated, but not yet Ready
```

Verwendung von Iota in const

Dies ist eine Aufzählung für die Erstellung von const. Der Go-Compiler startet Iota von 0 und erhöht sich für jede folgende Konstante um eins. Der Wert wird zur Kompilierungszeit und nicht zur Laufzeit bestimmt. Aus diesem Grund können wir Iota nicht auf Ausdrücke anwenden, die zur Laufzeit ausgewertet werden.

Programm zur Verwendung von 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)
}
```

Versuchen Sie es in Go Playground

Jota online lesen: https://riptutorial.com/de/go/topic/2865/jota

Kapitel 30: JSON

Syntax

- func Marshal (v interface {}) ([] Byte, Fehler)
- func Unmarshal (data [] byte, v interface {}) fehler

Bemerkungen

Das Paket `"encoding/json"` Package json implementiert die Kodierung und Dekodierung von JSON-Objekten in `Go` .

Typen in JSON mit ihren entsprechenden konkreten Typen in Go sind:

JSON-Typ	Gehen Sie konkrete Art
boolean	bool
Zahlen	float64 oder int
Schnur	Schnur
Null	Null

Examples

Grundlegende JSON-Kodierung

`json.Marshal` aus dem Paket `"encoding/json"` kodiert einen Wert in JSON.

Der Parameter ist der zu codierende Wert. Die zurückgegebenen Werte sind ein Byte-Array, das die JSON-codierte Eingabe (bei Erfolg) und einen Fehler (bei Fehler) darstellt.

```
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"]"
```

Spielplatz

Hier einige grundlegende Beispiele für die Kodierung integrierter Datentypen:

```
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}
```

Spielplatz

Die Kodierung einfacher Variablen ist hilfreich, um zu verstehen, wie die JSON-Kodierung in Go funktioniert. In der realen Welt werden Sie jedoch wahrscheinlich komplexere Daten verschlüsseln, die in Strukturen gespeichert sind .

Grundlegende JSON-Dekodierung

`json.Unmarshal` aus dem Paket `"encoding/json"` dekodiert einen JSON-Wert in den Wert, der von der angegebenen Variablen `json.Unmarshal` wird.

Die Parameter sind der zu decodierende Wert in `[]bytes` und eine Variable, die als Speicher für den deserialisierten Wert verwendet werden soll. Der zurückgegebene Wert ist ein Fehler (bei einem Fehler).

```
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]
```

Spielplatz

Beachten Sie, dass wir im obigen Beispiel sowohl den Typ des Schlüssels als auch den Wert im Voraus kannten. Dies ist jedoch nicht immer der Fall. In den meisten Fällen enthält JSON gemischte Werttypen.

```
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"
```

Spielplatz

Im letzten Beispiel oben haben wir eine generische Karte verwendet, um den dekodierten Wert zu speichern. Wir müssen eine `map[string]interface{}` da wir wissen, dass es sich bei den Schlüsseln um Strings handelt, aber wir kennen den Typ ihrer Werte nicht im Voraus.

Dies ist ein sehr einfacher Ansatz, der aber auch äußerst begrenzt ist. In der realen Welt würden Sie im Allgemeinen einen JSON in einen benutzerdefinierten `struct` dekodieren .

JSON-Daten aus einer Datei decodieren

JSON-Daten können auch aus Dateien gelesen werden.

Nehmen wir an, wir haben eine Datei namens `data.json` mit folgendem Inhalt:

```
[
    {
      "Name" : "John Doe",
      "Standard" : 4
    },
    {
      "Name" : "Peter Parker",
      "Standard" : 11
    },
    {
      "Name" : "Bilbo Baggins",
      "Standard" : 150
    }
]
```

Das folgende Beispiel liest die Datei und dekodiert den Inhalt:

```
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)
    }
}
```

Die Datei `data.json` muss sich im selben Verzeichnis wie das ausführbare Programm von Go befinden. Weitere Informationen zur Arbeit mit Dateien in Go finden Sie in der Dokumentation zu Go File I / O.

Verwenden anonymer Strukturen zur Dekodierung

Das Ziel bei der Verwendung anonymer Strukturen ist es, nur die Informationen zu entschlüsseln, die uns wichtig sind, ohne unsere App mit Typen zu verschwenden, die nur in einer einzigen Funktion verwendet werden.

```
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": "abcdef0000000000000000000000000000000000",
        "_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)
```

Ausgabe: `{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}}]}`

Spielplatz

Für den allgemeinen Fall siehe auch:
http://stackoverflow.com/documentation/go/994/json/4111/encoding-decoding-go-structs

JSON-Strukturfelder konfigurieren

Betrachten Sie das folgende Beispiel:

```
type Company struct {
    Name     string
    Location string
}
```

Bestimmte Felder ausblenden / überspringen

Um `Revenue` und `Sales` zu exportieren, sie jedoch vor dem Kodieren / Dekodieren zu verbergen, verwenden Sie `json:"-"` oder benennen Sie die Variable um, um mit einem Kleinbuchstaben zu beginnen. Beachten Sie, dass dies verhindert, dass die Variable außerhalb des Pakets sichtbar ist.

```
type Company struct {
    Name     string `json:"name"`
    Location string `json:"location"`
    Revenue  int    `json:"-"`
    sales    int
}
```

Leere Felder ignorieren

Um zu verhindern , `Location` von im JSON enthalten ist , wenn es um seinen Wert Null gesetzt wird, fügen `,omitempty` zum `json` - Tag.

```
type Company struct {
    Name     string `json:"name"`
    Location string `json:"location,omitempty"`
}
```

Beispiel im Spielplatz

Marshaling-Strukturen mit privaten Feldern

Als guter Entwickler haben Sie folgende Struktur mit exportierten und nicht exportierten Feldern erstellt:

```
type MyStruct struct {
    uuid string
    Name string
}
```

Beispiel im Spielplatz: https://play.golang.org/p/Zk94Il2ANZ

Jetzt wollen Sie `Marshal()` diese Struktur in gültige JSON für die Speicherung in so etwas wie ETCD. Da jedoch `uuid` in nicht exportiert, die `json.Marshal()` überspringt er. Was ist zu tun? Verwenden Sie eine anonyme Struktur und die Schnittstelle `json.MarshalJSON()` ! Hier ist ein Beispiel:

```
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
}
```

Beispiel im Spielplatz: https://play.golang.org/p/Bv2k9GgbzE

Kodierung / Dekodierung mit Go-Strukturen

Nehmen wir an , wir haben die folgende `struct` , die eine definiert `City` Typ:

```
type City struct {
    Name string
    Temperature int
}
```

Wir können City-Werte mit dem `encoding/json` Paket codieren / decodieren.

Als Erstes müssen wir mithilfe der Go-Metadaten dem Encoder die Entsprechung zwischen den Strukturfeldern und den JSON-Schlüsseln mitteilen.

```
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
}
```

Um dieses Beispiel einfach zu halten, erklären wir eine explizite Entsprechung zwischen den Feldern und den Schlüsseln. Sie können jedoch mehrere Varianten der `json:` -Metadaten verwenden, wie in den Dokumenten erläutert .

WICHTIG: Nur exportierte Felder (Felder mit Großnamen) werden serialisiert / deserialisiert. Wenn Sie beispielsweise das Feld ***t** perperature benennen* , wird es ignoriert, auch wenn Sie die `json` Metadaten festlegen.

Codierung

Um eine `City` Struktur zu codieren, verwenden Sie `json.Marshal` wie im grundlegenden Beispiel:

```
// 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}
```

Spielplatz

Dekodierung

Um eine `City` Struktur zu decodieren, verwenden Sie `json.Unmarshal` wie im grundlegenden Beispiel:

```
// 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}
```

Spielplatz

JSON online lesen: https://riptutorial.com/de/go/topic/994/json

Kapitel 31: JWT-Autorisierung in Go

Einführung

JSON-Web-Token (JWTs) sind eine beliebte Methode, um Forderungen zwischen zwei Parteien sicher darzustellen. Bei der Entwicklung von Webanwendungen oder Anwendungsprogrammierschnittstellen ist es wichtig zu verstehen, wie mit ihnen gearbeitet wird.

Bemerkungen

context.Context und HTTP-Middleware sind außerhalb des Themas dieses Themas, aber dennoch sollten neugierige, wandernde Seelen https://github.com/goware/jwtauth , https://github.com/auth0/go-jwt- Middleware und https://github.com/dgrijalva/jwt-go .

Ein großes Lob an Dave Grijalva für seine erstaunliche Arbeit an go-jwt.

Examples

Analysieren und Validieren eines Tokens mithilfe der HMAC-Signaturmethode

```
// sample token string taken from the New example
tokenString :=
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmb28iOiJiYXIiLCJuYmYiOjE0NDQ0Nzg0MDB9.u1riaD1rW97opCoAuRCTy4w5
Zk-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)
}
```

Ausgabe:

```
bar 1.4444784e+09
```

(Aus der Dokumentation , mit freundlicher Genehmigung von Dave Grijalva.)

Token mit einem benutzerdefinierten Anspruchstyp erstellen

Der `StandardClaim` ist in den benutzerdefinierten Typ eingebettet, um die einfache Kodierung, Analyse und Validierung von Standardansprüchen zu ermöglichen.

```
tokenString :=
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmb28iOiJiYXIiLCJleHAiOjE1MDAwLCJpc3MiOiJ0ZXN0In0.HE7fK0xOQwFEr


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)
    }
})
```

Ausgabe:

```
bar 15000
```

(Aus der Dokumentation , mit freundlicher Genehmigung von Dave Grijalva.)

Erstellen, Signieren und Codieren eines JWT-Tokens mithilfe der HMAC-Signaturmethode

```
// 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)
```

Ausgabe:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmb28iOiJiYXIiLCJuYmYiOjE0NDQ0Nzg0MDB9.u1riaD1rW97opCoAuRCTy4w58
```

```
Zk-bh7vLiRIsrpU <nil>
```

(Aus der Dokumentation , mit freundlicher Genehmigung von Dave Grijalva.)

Verwenden Sie den StandardClaims-Typ, um ein Token zu analysieren

Der `StandardClaims` Typ kann in Ihre benutzerdefinierten Typen eingebettet werden, um Standardvalidierungsfunktionen bereitzustellen. Sie können es alleine verwenden, aber es ist nicht möglich, andere Felder nach der Analyse abzurufen. Siehe das Beispiel für benutzerdefinierte Ansprüche zur beabsichtigten Verwendung.

```
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)
```

Ausgabe:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1MDAwLCJpc3MiOiJ0ZXN0In0.QsODzZu3lUZMVdhbO76u3Jv02iYCvE
<nil>
```

(Aus der Dokumentation , mit freundlicher Genehmigung von Dave Grijalva.)

Analysieren der Fehlertypen mit Bitfeldprüfungen

```
// Token from another example.  This token is expired
var tokenString =
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmb28iOiJiYXIiLCJleHAiOjE1MDAwLCJpc3MiOiJ0ZXN0In0.HE7fK0xOQwFEr


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)
```

```
}
```

Ausgabe:

```
Timing is everything
```

(Aus der Dokumentation , mit freundlicher Genehmigung von Dave Grijalva.)

Token aus dem HTTP-Autorisierungsheader abrufen

```
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 context.WithValue(ctx, JWTTokenContextKey, token)
    }
}
```

(Von Go-Kit / Kit , mit freundlicher Genehmigung von Peter Bourgon)

JWT-Autorisierung in Go online lesen: https://riptutorial.com/de/go/topic/10161/jwt-autorisierung-in-go

Kapitel 32: Karten

Einführung

Maps sind Datentypen, die zum Speichern von ungeordneten Schlüsselwertpaaren verwendet werden, so dass das Nachschlagen des mit einem bestimmten Schlüssel verknüpften Werts sehr effizient ist. Schlüssel sind einzigartig. Die zugrunde liegende Datenstruktur wächst nach Bedarf, um neue Elemente aufzunehmen, sodass sich der Programmierer nicht um die Speicherverwaltung kümmern muss. Sie ähneln dem, was andere Sprachen Hashtabellen, Wörterbücher oder assoziative Arrays nennen.

Syntax

- var mapName map [KeyType] ValueType // eine Map deklarieren
- var mapName = map [KeyType] ValueType {} // Eine leere Map deklarieren und zuweisen
- var mapName = map [KeyType] ValueType {key1: val1, key2: val2} // Deklarieren und Zuordnen einer Map
- mapName: = make (map [KeyType] ValueType) // deklarieren und initialisieren Sie eine Standardgröße
- mapName: = machen (map [KeyType] Valuetype, Länge) // deklarieren und *Länge* Größe Karte initialisieren
- mapName: = map [KeyType] ValueType {} // Automatische Deklaration und Zuordnung einer leeren Map mit: =
- mapName: = map [KeyType] ValueType {key1: value1, key2: value2} // Automatische Deklaration und Zuordnung einer Map mit: =
- value: = mapName [key] // Wert nach Schlüssel abrufen
- value, hasKey: = mapName [key] // Wert nach Schlüssel abrufen, 'hasKey' ist 'true', wenn der Schlüssel in der Map vorhanden ist
- mapName [Schlüssel] = Wert // Wert nach Schlüssel setzen

Bemerkungen

Go bietet einen integrierten `map` , der eine *Hashtabelle implementiert* . Maps sind der integrierte assoziative Datentyp von Go (in anderen Sprachen auch *Hashes* oder *Wörterbücher* genannt).

Examples

Eine Karte deklarieren und initialisieren

Sie definieren eine Map mit der Keyword- `map` , gefolgt von den Schlüsseltypen und deren Werten:

```
// 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
```

Karten sind Referenztypen. Sobald sie definiert sind, haben sie den *Wert* `nil` . Bei Schreibvorgängen auf Nullkarten gerät Panik in Betracht, und Lesevorgänge geben immer den Nullwert zurück.

Um eine Karte zu initialisieren, verwenden Sie die `make` Funktion:

```
m := make(map[string]int)
```

Mit der Zwei-Parameter-Form von `make` ist es möglich, eine anfängliche Eingabekapazität für die Karte anzugeben, wodurch die Standardkapazität überschrieben wird:

```
m := make(map[string]int, 30)
```

Alternativ können Sie eine Map deklarieren, sie auf ihren Nullwert initialisieren und ihr später einen literalen Wert zuweisen. Dies ist hilfreich, wenn Sie die Struktur in json marshallieren und bei der Rückkehr eine leere Map erzeugen.

```
m := make(map[string]int, 0)
```

Sie können auch eine Karte erstellen und ihren ursprünglichen Wert mit geschweiften Klammern (`{}`) festlegen.

```
var m map[string]int = map[string]int{"Foo": 20, "Bar": 30}

fmt.Println(m["Foo"]) // outputs 20
```

Alle folgenden Anweisungen führen dazu, dass die Variable an denselben Wert gebunden ist.

```
// 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{}
```

Wir können auch ein *Kartenliteral verwenden* , um eine neue Karte mit einigen anfänglichen Schlüssel / Wert-Paaren zu erstellen .

Der Schlüsseltyp kann ein beliebiger vergleichbarer Typ sein. Dies gilt insbesondere für Funktionen, Karten und Slices . Der Werttyp kann ein beliebiger Typ sein, einschließlich benutzerdefinierter Typen oder `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{}
```

Eine Karte erstellen

Mit einem *zusammengesetzten Literal* kann eine Map in einer einzelnen Anweisung deklariert und initialisiert werden.

Verwendung des automatischen Typs Kurze Variablendeklaration:

```
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"}
```

Gleicher Code, jedoch mit Variablentypen:

```
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"}
```

Sie können auch eigene Strukturen in eine Karte einfügen:

Sie können benutzerdefinierte Typen als Wert verwenden:

```
// 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"}}
```

Ihre Struktur kann auch der *Schlüssel* zur Karte sein:

```
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}
```

Sie können eine leere Map erstellen, indem Sie einfach keinen Wert in die Klammern `{}` .

```
mapIntInt := map[int]int{}
mapIntString := map[int]string{}
mapStringInt := map[string]int{}
mapStringString := map[string]string{}
mapStringPerson := map[string]Person{}
```

Sie können eine Map direkt erstellen und verwenden, ohne sie einer Variablen zuordnen zu müssen. Sie müssen jedoch sowohl die Deklaration als auch den Inhalt angeben.

```
// 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)
```

Nullwert einer Karte

Der Nullwert einer `map` ist `nil` und hat eine Länge von `0` .

```
var m map[string]string
fmt.Println(m == nil) // true
fmt.Println(len(m) ==0) // true
```

Eine `nil` Karte hat weder Schlüssel noch können Schlüssel hinzugefügt werden. Eine `nil` Map verhält sich wie eine leere Map, wenn gelesen wird, verursacht jedoch eine Laufzeitpanik, wenn geschrieben wird.

```
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
```

Sie sollten nicht versuchen, aus einer Nullwertzuordnung zu lesen oder in diese zu schreiben. Initialisieren Sie stattdessen die Karte (mit `make` oder Zuweisung), bevor Sie sie verwenden.

```
var m map[string]string
m = make(map[string]string) // OR m = map[string]string{}
m["foo"] = "bar"
```

Iteration der Elemente einer Karte

```
import fmt

people := map[string]int{
  "john": 30,
  "jane": 29,
  "mark": 11,
}

for key, value := range people {
  fmt.Println("Name:", key, "Age:", value)
}
```

Beachten Sie, dass beim Durchlaufen einer Karte mit einer Bereichsschleife die Iterationsreihenfolge nicht angegeben wird und nicht von einer Iteration zur nächsten garantiert ist.

Sie können auch die Schlüssel oder die Werte der Karte verwerfen, wenn Sie nur nach Schlüsseln oder Werten suchen.

Iteration der Schlüssel einer Karte

```
people := map[string]int{
  "john": 30,
  "jane": 29,
  "mark": 11,
}

for key, _ := range people {
  fmt.Println("Name:", key)
}
```

Wenn Sie nur nach den Schlüsseln suchen, da sie der erste Wert sind, können Sie den Unterstrich einfach fallen lassen:

```
for key := range people {
  fmt.Println("Name:", key)
}
```

Beachten Sie, dass beim Durchlaufen einer Karte mit einer Bereichsschleife die Iterationsreihenfolge nicht angegeben wird und nicht von einer Iteration zur nächsten garantiert ist.

Kartenelement löschen

Die integrierte `delete` entfernt das Element mit dem angegebenen Schlüssel aus einer Karte.

```
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]
```

Wenn die `map` `nil` oder kein solches Element vorhanden ist, hat das `delete` keine Auswirkung.

```
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
```

Kartenelemente zählen

Die eingebaute Funktion `len` gibt die Anzahl der Elemente in einer `map`

```
m := map[string]int{}
len(m) // 0

m["foo"] = 1
len(m) // 1
```

Wenn eine Variable auf eine `nil` Karte zeigt, gibt `len` 0 zurück.

```
var m map[string]int
len(m) // 0
```

Gleichzeitiger Zugriff auf Karten

Karten in go sind für Parallelität nicht sicher. Sie müssen eine Sperre zum Lesen und Schreiben auf sie nehmen, wenn Sie gleichzeitig darauf zugreifen. Normalerweise ist die beste Option die Verwendung von `sync.RWMutex` da Sie Lese- und Schreibsperren haben können. Es kann jedoch auch ein `sync.Mutex` verwendet werden.

```
type RWMap struct {
    sync.RWMutex
    m map[string]int
}

// Get is a wrapper for getting the value from the underlying map
func (r RWMap) 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 RWMap) Set(key string, val int) {
    r.Lock()
    defer r.Unlock()
    r.m[key] = val
}

// Inc increases the value in the RWMap for a key.
//   This is more pleasant than r.Set(key, r.Get(key)++)
func (r RWMap) Inc(key string) {
    r.Lock()
    defer r.Unlock()
    r.m[key]++
}

func main() {

    // Init
    counter := RWMap{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")
}
```

Der Kompromiss zwischen den Wrapper-Funktionen besteht zwischen dem öffentlichen Zugriff auf die zugrunde liegende Karte und der korrekten Verwendung der entsprechenden Sperren.

Erstellen von Karten mit Slices als Werten

```
m := make(map[string][]int)
```

Wenn Sie auf einen nicht vorhandenen Schlüssel zugreifen, wird ein Nullwert als Wert zurückgegeben. Da keine Slices wie Slices mit einer Länge von null funktionieren, müssen sie normalerweise nicht mit `append` oder anderen integrierten Funktionen verwendet werden. `append` müssen Sie normalerweise nicht prüfen, ob ein Schlüssel vorhanden ist:

```
// m["key1"] == nil && len(m["key1"]) == 0
m["key1"] = append(m["key1"], 1)
// len(m["key1"]) == 1
```

Durch das Löschen eines Schlüssels aus der Karte wird der Schlüssel auf ein Nullteil zurückgesetzt:

```
delete(m, "key1")
// m["key1"] == nil
```

Überprüfen Sie das Element in einer Karte

Um einen Wert aus der Karte zu erhalten, müssen Sie nur Folgendes tun: 00

```
value := mapName[ key ]
```

Wenn die Karte den Schlüssel enthält, wird der entsprechende Wert zurückgegeben.
Wenn nicht, wird ein Nullwert des Wertetyps der Map zurückgegeben (" `0` `""` wenn `""` `int` -Werte vorhanden sind, `""` wenn `""` `string` `""`).

```
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
```

Um zwischen leeren Werten und nicht vorhandenen Schlüsseln zu unterscheiden, können Sie den zweiten zurückgegebenen Wert des `value, hasKey := map["key"]` (mit `value, hasKey := map["key"]`).

Dieser zweite Wert ist vom `boolean` und lautet:

- `true` wenn sich der Wert in der Karte befindet,
- `false` wenn die Karte den angegebenen Schlüssel nicht enthält.

Sehen Sie sich folgendes Beispiel an:

```
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
}
```

Die Werte einer Karte wiederholen

```
people := map[string]int{
  "john": 30,
  "jane": 29,
  "mark": 11,
}

for _, value := range people {
```

```
  fmt.Println("Age:", value)
}
```

Beachten Sie, dass beim Durchlaufen einer Karte mit einer Bereichsschleife die Iterationsreihenfolge nicht angegeben wird und nicht von einer Iteration zur nächsten garantiert ist.

Kopieren Sie eine Karte

Karten enthalten wie Slices **Verweise** auf eine zugrunde liegende Datenstruktur. Wenn Sie also den Wert einer anderen Variablen zuweisen, wird nur die Referenz übergeben. Um die Karte zu kopieren, müssen Sie eine andere Karte erstellen und jeden Wert kopieren:

```
// 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
}
```

Eine Karte als Set verwenden

Einige Sprachen haben eine native Struktur für Mengen. Um einen Satz in Go zu erstellen, empfiehlt es sich, eine Zuordnung vom Werttyp des Satzes zu einer leeren Struktur (`map[Type]struct{}`) zu verwenden.

Zum Beispiel mit 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")
}
```

Karten online lesen: https://riptutorial.com/de/go/topic/732/karten

Kapitel 33: Konsolen-E / A

Examples

Lesen Sie die Eingabe von der Konsole

`scanf`

> Scanf scannt aus der Standardeingabe gelesenen Text und speichert aufeinanderfolgende, durch Leerzeichen getrennte Werte in aufeinanderfolgenden Argumenten, wie durch das Format bestimmt. Die Anzahl der erfolgreich gescannten Elemente wird zurückgegeben. Wenn dies weniger als die Anzahl der Argumente ist, meldet err warum. Zeilenumbrüche in der Eingabe müssen mit den Zeilenumbrüchen im Format übereinstimmen. Die einzige Ausnahme: Das Verb% c scannt immer die nächste Rune in der Eingabe, auch wenn es sich um ein Leerzeichen (oder einen Tabulator usw.) oder um eine Zeilenschaltung handelt.

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

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

`scan`

> Scan scannt Text, der aus der Standardeingabe gelesen wurde, und speichert aufeinanderfolgende, durch Leerzeichen getrennte Werte in aufeinanderfolgenden Argumenten. Zeilenumbrüche gelten als Leerzeichen. Die Anzahl der erfolgreich gescannten Elemente wird zurückgegeben. Wenn dies weniger als die Anzahl der Argumente ist, meldet err warum.

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

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

Mit `scanln`

> Sscanln ähnelt Sscan, stoppt jedoch das Scannen bei einem Zeilenumbruch und nach dem letzten Eintrag muss ein Zeilenumbruch oder EOF vorhanden sein.

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

`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())
}
```

Konsolen-E / A online lesen: https://riptutorial.com/de/go/topic/9741/konsolen-e---a

Kapitel 34: Konstanten

Bemerkungen

Go unterstützt Konstanten von Zeichen-, Zeichenfolgen-, booleschen und numerischen Werten.

Examples

Konstante deklarieren

Konstanten werden wie Variablen deklariert, verwenden jedoch das Schlüsselwort `const` :

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

Wie bei Variablen werden Namen, die mit einem Großbuchstaben beginnen, exportiert (*öffentlich*), Namen, die mit Kleinbuchstaben beginnen, sind nicht zulässig.

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

Konstanten können wie jede andere Variable verwendet werden, mit der Ausnahme, dass der Wert nicht geändert werden kann. Hier ist ein Beispiel:

```
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
```

```
}
```

Spielplatz

Deklaration mehrerer Konstanten

Sie können mehrere Konstanten innerhalb desselben `const` Blocks deklarieren:

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

Und erhöht automatisch Konstanten mit dem Schlüsselwort `iota` :

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

Weitere Beispiele für die Verwendung von `iota` zum Deklarieren von Konstanten finden Sie unter Iota .

Sie können auch mehrere Konstanten mithilfe der Mehrfachzuweisung deklarieren. Diese Syntax kann jedoch schwieriger zu lesen sein und wird im Allgemeinen vermieden.

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

Typisierte und nicht typisierte Konstanten

Konstanten in Go können typisiert oder nicht typisiert sein. Zum Beispiel das folgende String-Literal:

```
"bar"
```

Man könnte sagen, dass der Typ des Literal `string` ist, dies ist jedoch nicht semantisch korrekt. Literale sind *stattdessen Zeichenkettenkonstanten ohne Typ* . Es ist eine Zeichenfolge (genauer gesagt, der *Standardtyp* ist `string`), aber es ist kein Go- **Wert** und hat daher keinen Typ, bis er zugewiesen oder in einem Kontext verwendet wird, der eingegeben wird. Dies ist eine subtile Unterscheidung, die jedoch verständlich ist.

Ähnlich, wenn wir das Literal einer Konstanten zuweisen:

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

Es bleibt untypisiert, da Konstanten standardmäßig nicht typisiert sind. Es ist auch möglich, es als *typisierte String-Konstante* zu deklarieren:

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

Der Unterschied kommt zum Tragen, wenn wir versuchen, diese Konstanten in einem Kontext mit Typ zuzuweisen. Betrachten Sie zum Beispiel Folgendes:

```
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
```

Konstanten online lesen: https://riptutorial.com/de/go/topic/1047/konstanten

Kapitel 35: Kontext

Syntax

- type CancelFunc func ()
- func Hintergrund () Kontext
- func TODO () Kontext
- func WithCancel (übergeordneter Context) (ctx Context, CancelFunc abbrechen)
- func WithDeadline (übergeordneter Kontext, Deadline-Zeit.Zeitpunkt) (Kontext, CancelFunc)
- func WithTimeout (übergeordneter Kontext, Timeout-Zeit.Dauer) (Kontext, CancelFunc)
- func WithValue (übergeordneter Kontext, Schlüsselschnittstelle {}, Wertschnittstelle {})

Bemerkungen

Das `context` (in Go 1.7) oder das Paket `golang.org/x/net/context` (Pre 1.7) ist eine Schnittstelle zum Erstellen von Kontexten, mit der Werte und Fristen für Anforderungsbereiche über API-Grenzen und zwischen Diensten übertragen werden können als einfache Implementierung der Schnittstelle.

beiseite: das wort "context" wird lose verwendet, um sich auf den gesamten baum oder auf einzelne blätter im baum zu beziehen, z. die tatsächlichen `context.Context` Werte.

Auf hoher Ebene ist ein Kontext ein Baum. Neue Blätter werden der `context.Context` hinzugefügt, wenn sie erstellt werden (ein `context.Context` mit einem übergeordneten Wert), und Blätter werden niemals aus der `context.Context` entfernt. Jeder Kontext hat Zugriff auf alle darüber liegenden Werte (der Datenzugriff fließt nur aufwärts), und wenn ein Kontext gelöscht wird, werden auch seine untergeordneten Elemente gelöscht (Abbruchsignale werden nach unten abgesetzt). Das Löschsignal wird mittels einer Funktion implementiert, die einen Kanal zurückgibt, der geschlossen wird (lesbar), wenn der Kontext gelöscht wird. Dies macht Kontexte zu einem sehr effizienten Weg, um das Parallelitätsmuster der Pipeline und des Abbruchs oder Timeouts zu implementieren.

`ctx context.Context` haben Funktionen, die einen Kontext enthalten, das erste Argument `ctx context.Context` . Dies ist zwar nur eine Konvention, sollte aber befolgt werden, da viele statische Analysewerkzeuge speziell nach diesem Argument suchen. Da es sich bei Context um eine Schnittstelle handelt, ist es auch möglich, vorhandene kontextähnliche Daten (Werte, die in einer Request-Call-Chain weitergegeben werden) in einen normalen Go-Kontext umzuwandeln und diese durch die Implementierung einiger Methoden rückwärtskompatibel zu verwenden. Darüber hinaus sind Kontexte für den gleichzeitigen Zugriff sicher, sodass Sie sie von vielen Goroutinen (unabhängig davon, ob sie auf parallelen Threads oder als gleichzeitige Coroutinen ausgeführt werden) ohne Angst verwenden können.

Lesen Sie weiter

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

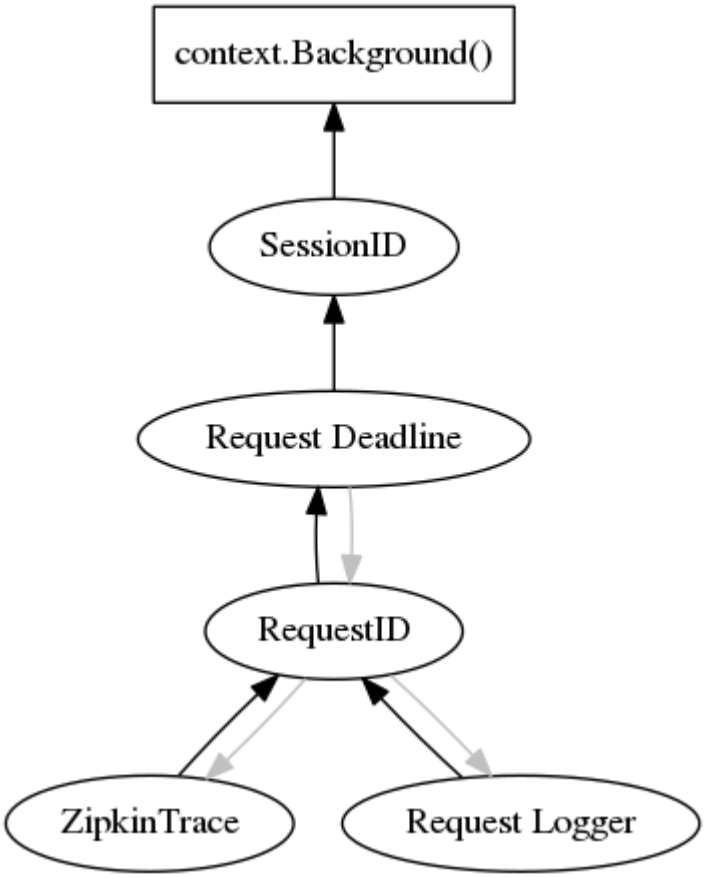
Examples

Kontextbaum als gerichteter Graph dargestellt

Eine einfache Kontextbaumstruktur (die einige gängige Werte enthält, die möglicherweise einen Anforderungsbereich haben und in einen Kontext eingeschlossen sind), die aus Go-Code wie folgt aufgebaut ist:

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

Ist ein Baum, der als gerichteter Graph dargestellt werden kann, der folgendermaßen aussieht:



Jeder untergeordnete Kontext hat Zugriff auf die Werte seiner übergeordneten Kontexte, sodass der Datenzugriff im Baum nach oben fließt (durch schwarze Kanten dargestellt). Aufhebungssignale dagegen ziehen den Baum hinunter (wenn ein Kontext abgebrochen wird, werden auch alle seine untergeordneten Elemente gelöscht). Der Löschsignalfluss wird durch die

grauen Kanten dargestellt.

Verwendung eines Kontexts zum Abbrechen der Arbeit

Durch Übergeben eines Kontexts mit einer Zeitüberschreitung (oder mit einer Abbruchfunktion) an eine Funktion mit langer Laufzeit können Sie die Funktionsabbrüche abbrechen:

```
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!
    }
}
```

Kontext online lesen: https://riptutorial.com/de/go/topic/2743/kontext

Kapitel 36: Kreuzzusammenstellung

Einführung

Der Go-Compiler kann Binärdateien für viele Plattformen, dh Prozessoren und Systeme, erstellen. Im Gegensatz zu den meisten anderen Compilern gibt es keine besonderen Anforderungen für das Cross-Compilieren. Es ist so einfach zu verwenden wie das normale Compilieren.

Syntax

- GOOS = Linux GOARCH = Amd64 go build

Bemerkungen

Unterstützte Zielsystemkombinationen für Betriebssystem und Architektur (Quelle)

$ GOOS	$ GOARCH
Android	Arm
Darwin	386
Darwin	amd64
Darwin	Arm
Darwin	arm64
Libelle	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
Fenster	386
Fenster	amd64

Examples

Kompilieren Sie alle Architekturen mit einem Makefile

Dieses Makefile wird ausführbare Dateien für Windows, Mac und Linux (ARM und x86) zusammenstellen und komprimieren.

```
# 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)
```

(Seien Sie vorsichtig, dass Makefile harte Tabs und keine Leerzeichen benötigt.)

Einfache Cross-Kompilierung mit Go Build

Führen Sie in Ihrem Projektverzeichnis den Befehl `go build` und geben Sie das Betriebssystem und das Architekturziel mit den Umgebungsvariablen `GOOS` und `GOARCH` :

Kompilieren für Mac (64-Bit):

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

Kompilieren für Windows x86-Prozessor:

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

Möglicherweise möchten Sie auch den Dateinamen der ausführbaren Ausgabedatei manuell festlegen, um die Architektur zu verfolgen:

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

Ab Version 1.7 erhalten Sie eine Liste aller möglichen Kombinationen von GOOS und GOARCH mit:

```
go tool dist list
```

(oder für einen einfacheren Maschinenverbrauch `go tool dist list -json`)

Kreuzzusammenstellung mit gox

Eine andere praktische Lösung für die Cross-Kompilierung ist die Verwendung von `gox` : https://github.com/mitchellh/gox

Installation

Die Installation ist sehr einfach, indem Sie `go get github.com/mitchellh/gox` ausführen. Die resultierende ausführbare Datei wird in Gos Binärverzeichnis `/golang/bin` , z. B. `/golang/bin` oder `~/golang/bin` . Stellen Sie sicher, dass dieser Ordner Teil Ihres Pfads ist, um den Befehl `gox` von einem beliebigen Ort aus zu verwenden.

Verwendungszweck

`gox` Sie `gox` dem Root-Ordner eines Go-Projekts (wo Sie z. B. `go build` ausführen) aus, um alle möglichen Binaries für jede verfügbare Architektur (z. B. x86, ARM) und Betriebssystem (z. B. Linux, macOS, Windows) zu erstellen.

Um für ein bestimmtes Betriebssystem zu bauen, verwenden `gox -os="linux"` stattdessen zB `gox -os="linux"` . Auch die Architekturoption könnte definiert werden: `gox -osarch="linux/amd64"` .

Einfaches Beispiel: Kompilieren Sie helloworld.go für die Architektur des Arms auf einem Linux-Computer

Bereite helloworld.go vor (siehe unten)

```
package main

import "fmt"

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

Führen Sie `GOOS=linux GOARCH=arm go build helloworld.go`

Kopieren Sie die generierte `helloworld` Datei (Arm- `helloworld`) auf Ihren Zielcomputer.

Kreuzzusammenstellung online lesen:

https://riptutorial.com/de/go/topic/1020/kreuzzusammenstellung

Kapitel 37: Kryptographie

Einführung

Finden Sie heraus, wie Sie mit Go Daten verschlüsseln und entschlüsseln können. Beachten Sie, dass dies kein Kurs über Kryptographie ist, sondern wie Sie ihn mit Go erreichen.

Examples

Verschlüsselung und Entschlüsselung

Vorwort

Dies ist ein detailliertes Beispiel, wie Daten mit Go verschlüsselt und entschlüsselt werden. Der Anwendungscode ist kurz, dh die Fehlerbehandlung wird nicht erwähnt. Der vollständige Arbeitsprojekt mit der Fehlerbehandlung und Benutzeroberfläche könnte auf Github finden hier .

Verschlüsselung

Einleitung und Daten

Dieses Beispiel beschreibt eine vollständige Verschlüsselung und Entschlüsselung in Go. Dazu benötigen wir Daten. In diesem Beispiel verwenden wir unsere eigenen Datenstruktur `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
}
```

Als nächstes wollen wir ein solches `secret` verschlüsseln. Das vollständige Arbeitsbeispiel finden Sie hier (Link zu Github) . Nun der Schritt für Schritt:

Schritt 1

Zunächst benötigen wir eine Art Master-Passwort, um das Geheimnis zu schützen: `masterPassword := "PASS"`

Schritt 2

Alle Krypto-Methoden, die mit Bytes statt mit Strings arbeiten. Daher bauen wir ein Byte-Array mit den Daten unseres Geheimnisses auf.

```
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%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,
))
```

Schritt 3

Wir erzeugen etwas Salz, um Regenbogenangriffe zu verhindern, vgl. Wikipedia : `saltBytes := uuid.NewV4().Bytes()` . Hier verwenden wir eine UUID v4, die nicht vorhersehbar ist.

Schritt 4

Nun können wir einen Schlüssel und einen Vektor aus dem Master-Passwort und dem Zufalls-Salt für RFC 2898 ableiten:

```
keyLength := 256
rfc2898Iterations := 6

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

```
vectorBytes := keyVectorData[keyLength/8:]
```

Schritt 5

Der gewünschte CBC-Modus funktioniert mit ganzen Blöcken. Daher müssen wir prüfen, ob unsere Daten auf einen vollständigen Block ausgerichtet sind. Wenn nicht, müssen wir es auffüllen:

```
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
}
```

Schritt 6

Jetzt erstellen wir eine AES-Chiffre: `aesBlockEncrypter, aesErr := aes.NewCipher(keyBytes)`

Schritt 7

Wir reservieren den erforderlichen Speicher für die verschlüsselten Daten: `encryptedData := make([]byte, len(secretBytesDecrypted))` . Im Falle von AES-CBC hatten die verschlüsselten Daten die gleiche Länge wie die unverschlüsselten Daten.

Schritt 8

Nun sollten wir den Verschlüsseler erstellen und die Daten verschlüsseln:

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

Die verschlüsselten Daten befinden sich jetzt in der Variablen `encryptedData` .

Schritt 9

Die verschlüsselten Daten müssen gespeichert werden. Aber nicht nur die Daten: Ohne das Salz könnten die verschlüsselten Daten nicht entschlüsselt werden. Daher müssen wir ein Dateiformat verwenden, um dies zu verwalten. Hier verschlüsseln wir die verschlüsselten Daten als base64, vgl. Wikipedia :

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

Als Nächstes definieren wir unseren Dateiinhalt und unser eigenes Dateiformat. Das Format sieht

folgendermaßen aus: `salt[0x10]base64 content` . Zuerst lagern wir das Salz. Um den Anfang des base64-Inhalts zu markieren, speichern wir das Byte `10` . Dies funktioniert, weil base64 diesen Wert nicht verwendet. Daher konnten wir den Anfang von base64 finden, indem wir das erste Vorkommen von `10` vom Ende bis zum Anfang der Datei suchen.

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

Schritt 10

Schließlich könnten wir unsere Datei schreiben: `writeErr := ioutil.WriteFile("my secret.data", fileContent, 0644)` .

Entschlüsselung

Einleitung und Daten

Für die Verschlüsselung benötigen wir einige Daten, um damit arbeiten zu können. Wir gehen also davon aus, dass wir eine verschlüsselte Datei und die erwähnte Struktur `secret` . Ziel ist es, die verschlüsselten Daten aus der Datei zu lesen, zu entschlüsseln und eine Instanz der Struktur zu erstellen.

Schritt 1

Der erste Schritt ist identisch mit der Verschlüsselung: Wir benötigen eine Art Master-Passwort, um das Geheimnis zu entschlüsseln: `masterPassword := "PASS"` .

Schritt 2

Nun lesen wir die verschlüsselten Daten aus der Datei: `encryptedFileData, bytesErr := ioutil.ReadFile(filename)` .

Schritt 3

Wie bereits erwähnt, konnten wir Salt- und verschlüsselte Daten nach dem Trennzeichen-Byte `10` aufteilen, das vom Ende bis zum Anfang gesucht wurde:

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

```
    }
}
```

Schritt 4

Als Nächstes müssen wir die base64-codierten Bytes dekodieren:

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

Schritt 5

Nun können wir einen Schlüssel und einen Vektor aus dem Master-Passwort und dem Zufalls-Salt für RFC 2898 ableiten:

```
keyLength := 256
rfc2898Iterations := 6

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

Schritt 6

Erstellen Sie eine AES-Verschlüsselung: `aesBlockDecrypter, aesErr := aes.NewCipher(keyBytes)` .

Schritt 7

Reservieren Sie den erforderlichen Speicher für die entschlüsselten Daten: `decryptedData := make([]byte, len(encryptedBytes))` . Definitionsgemäß hat es dieselbe Länge wie die verschlüsselten Daten.

Schritt 8

Erstellen Sie nun den Entschlüsseler und entschlüsseln Sie die Daten:

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

Schritt 9

Konvertieren Sie die gelesenen Bytes in string: `decryptedString := string(decryptedData)` . Da wir Zeilen benötigen, teilen Sie die Zeichenfolge: `lines := strings.Split(decryptedString, "\n")` .

Schritt 10

Konstruiere ein `secret` aus den Linien:

```
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]
```

Erstellen Sie schließlich die Zeilenumbrüche innerhalb des Notizfeldes neu: `artifact.Notes = strings.Replace(artifact.Notes, string(65000), "\n", -1)`

Kryptographie online lesen: https://riptutorial.com/de/go/topic/10065/kryptographie

Kapitel 38: Leser

Examples

Verwenden von bytes.Reader zum Lesen aus einer Zeichenfolge

Eine Implementierung der `io.Reader` Schnittstelle befindet sich im `bytes` Paket. Damit kann ein Byte-Slice als Quelle für einen Reader verwendet werden. In diesem Beispiel wird das Byte-Slice aus einer Zeichenfolge übernommen, es wurde jedoch wahrscheinlicher aus einer Datei oder Netzwerkverbindung gelesen.

```
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)
```

Spielplatz gehen

Leser online lesen: https://riptutorial.com/de/go/topic/7000/leser

Kapitel 39: Methoden

Syntax

- func (t T) exampleOne (i int) (n int) {return i} // diese Funktion erhält eine Kopie von struct
- func (t * T) exampleTwo (i int) (n int) {return i} // Diese Methode empfängt einen Zeiger auf struct und kann ihn ändern

Examples

Grundlegende Methoden

Methoden in Go sind wie Funktionen, außer dass sie einen *Empfänger haben* .

Empfänger ist normalerweise eine Art Struktur oder Typ.

```
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)

}
```

Ausgabe:

```
Bobs rank now is:  1
Lets promote Bob!
Now Bobs rank is:  2
```

Verkettungsmethoden

Mit Methoden in golang können Sie die Methode "ketten" verwenden, indem Sie den Zeiger auf die Methode übergeben und den Zeiger auf dieselbe Struktur wie folgt zurückgeben:

```
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)

}
```

Ausgabe:

```
Set name of new Employee to Bob
Promoting Bob
Here we have Bob with rank 2
```

Increment-Decrement-Operatoren als Argumente in Methoden

Obwohl Go die Operatoren ++ und - unterstützt und das Verhalten fast mit c / c ++ vergleichbar ist, können Variablen mit solchen Operatoren nicht als Argument an function übergeben werden.

```
    package main

    import (
        "fmt"
```

```
)

func abcd(a int, b int) {
 fmt.Println(a," ",b)
}
func main() {
    a:=5
    abcd(a++,++a)
}
```

Ausgabe: Syntaxfehler: unerwartetes ++ erwartet Komma oder)

Methoden online lesen: https://riptutorial.com/de/go/topic/3890/methoden

Kapitel 40: mgo

Einführung

mgo (ausgesprochen als mango) ist ein MongoDB-Treiber für die Go-Sprache, der eine umfassende und gut getestete Auswahl an Funktionen unter einer sehr einfachen API implementiert, die den Standard-Go-Idiomen folgt.

Bemerkungen

API-Dokumentation

[https://gopkg.in/mgo.v2]

Examples

Beispiel

```
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)
}
```

mgo online lesen: https://riptutorial.com/de/go/topic/8898/mgo

Kapitel 41: Middleware

Einführung

In Go Middleware kann Code vor und nach der Handler-Funktion ausgeführt werden. Es nutzt die Leistung von Single Function Interfaces. Kann jederzeit ohne Auswirkungen auf die andere Middleware eingeführt werden. Beispiel: Authentifizierungsprotokollierung kann in späteren Entwicklungsphasen hinzugefügt werden, ohne den vorhandenen Code zu stören.

Bemerkungen

Die **Signatur der Middleware** sollte (http.ResponseWriter, * http.Request) sein, dh vom Typ **http.handlerFunc** .

Examples

Normale Handler-Funktion

```
func loginHandler(w http.ResponseWriter, r *http.Request) {
            // Steps to login
}


func main() {
    http.HandleFunc("/login", loginHandler)
    http.ListenAndServe(":8080", nil)
}
```

Middleware Berechnet die für die Ausführung von handlerFunc erforderliche Zeit

```
// logger middlerware 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
        }
    })
}
```

Wiederherstellungshandler, um zu verhindern, dass der Server abstürzt

```
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)
    })
}
```

Middleware online lesen: https://riptutorial.com/de/go/topic/9343/middleware

Kapitel 42: Mutex

Examples

Mutex-Verriegelung

Mutex Locking in Go ermöglicht Ihnen sicherzustellen, dass immer nur eine Goroutine gesperrt ist:

```
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()
}
```

Durch die Verwendung eines `Mutex` können Sie Race-Bedingungen, gleichzeitige Änderungen und andere Probleme im Zusammenhang mit mehreren gleichzeitigen Routinen vermeiden, die auf denselben Ressourcen ausgeführt werden. Beachten Sie, dass `Mutex.Unlock()` von jeder Routine ausgeführt werden kann, nicht nur von der Routine, die die Sperre erhalten hat. Beachten Sie auch, dass der Aufruf von `Mutex.Lock()` nicht fehlschlägt, wenn eine andere Routine die Sperre hält. es wird blockiert, bis die Sperre freigegeben wird.

Tipp: Wenn Sie eine Mutex-Variable an eine Funktion übergeben, übergeben Sie sie immer als Zeiger. Andernfalls wird eine Kopie Ihrer Variablen erstellt, die den Zweck des Mutex vereitelt. Wenn Sie eine ältere Go-Version (<1.7) verwenden, warnt Sie der Compiler nicht vor diesem Fehler!

Mutex online lesen: https://riptutorial.com/de/go/topic/2607/mutex

Kapitel 43: Nullwerte

Bemerkungen

Eine Sache zu beachten - Typen, die einen Nullwert ungleich Null haben, wie Strings, Ints, Floats, Bools und Structs, können nicht auf Null gesetzt werden.

Examples

Grundlegende Nullwerte

Variablen in Go werden immer initialisiert, unabhängig davon, ob Sie ihnen einen Startwert geben oder nicht. Jeder Typ, einschließlich benutzerdefinierter Typen, hat einen Nullwert, auf den sie festgelegt werden, wenn kein Wert angegeben wird.

```
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
```

Dies gilt auch für Karten, Slices, Channels und Funktionstypen. Diese Typen werden mit Null initialisiert. In Arrays wird jedes Element mit dem Nullwert seines jeweiligen Typs initialisiert.

Komplexere Nullwerte

In Slices ist der Nullwert ein leeres Slice.

```
var myIntSlice []int    // [] - an empty slice
```

Verwenden Sie `make` , um ein mit Werten gefülltes Slice zu erstellen. Alle im Slice erstellten Werte werden auf den Wert Null des Slice-Typs gesetzt. Zum Beispiel:

```
myIntSlice := make([]int, 5)    // [0, 0, 0, 0, 0] - a slice with 5 zeroes
fmt.Println(myIntSlice[3])
// Prints 0
```

In diesem Beispiel ist `myIntSlice` ein `int myIntSlice` , das 5 Elemente enthält, die alle 0 sind, da dies der Nullwert für den Typ `int` .

Sie können auch ein Slice mit `new` erstellen. Dadurch wird ein Zeiger auf ein Slice erstellt.

```
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
```

Hinweis: Scheibe Zeiger nicht Indizierung unterstützen , so dass Sie nicht die Werte zugreifen können `myIntSlice[3]` , anstatt was Sie tun müssen , um es wie `(*myIntSlice)[3]` .

Struktur Nullwerte

Wenn Sie eine Struktur erstellen, ohne sie zu initialisieren, wird jedes Feld der Struktur mit ihrem jeweiligen Nullwert initialisiert.

```
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-Nullwerte

Wie im Go-Blog :

> Arrays müssen nicht explizit initialisiert werden. Der Nullwert eines Arrays ist ein gebrauchsfertiges Array, dessen Elemente selbst auf Null gesetzt werden

Zum Beispiel wird `myIntArray` mit dem Nullwert von `int` initialisiert, der 0 ist:

```
var myIntArray [5]int     // an array of five 0's: [0, 0, 0, 0, 0]
```

Nullwerte online lesen: https://riptutorial.com/de/go/topic/6069/nullwerte

Kapitel 44: Nullwerte

Examples

Erläuterung

Nullwerte oder Nullinitialisierung sind einfach zu implementieren. Bei Sprachen wie Java kann es kompliziert erscheinen, dass einige Werte gleich `nil` können, andere dagegen nicht. Zusammenfassend aus Zero Value: Die Spezifikation der Go-Programmiersprache :

> Nur Zeiger, Funktionen, Schnittstellen, Schnitte, Kanäle und Karten können null sein. Der Rest wird basierend auf den jeweiligen Typen mit falschen, null oder leeren Zeichenfolgen initialisiert.

Wenn eine Funktion eine Bedingung überprüft, können Probleme auftreten:

```
func isAlive() bool {
    //Not implemented yet
    return false
}
```

Der Nullwert ist bereits vor der Implementierung falsch. Unit-Tests, die von der Rückgabe dieser Funktion abhängen, können falsche Positive / Negative ergeben.

Eine typische Problemumgehung besteht darin, auch einen Fehler zurückzugeben, der in Go idiomatisch ist:

```
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())
    }

}
```

spiele es auf dem Spielplatz

Wenn Sie sowohl eine Struktur als auch einen Fehler zurückgeben, benötigen Sie eine Rückgabestruktur, die nicht sehr elegant ist. Es gibt zwei Gegenoptionen:

- Mit Schnittstellen arbeiten: Geben Sie Null zurück, indem Sie eine Schnittstelle zurückgeben.

- Mit Zeigern arbeiten: Ein Zeiger **kann** `nil`

Der folgende Code gibt beispielsweise einen Zeiger zurück:

```
func(d *DB) GetUser(id uint64) (*User, error) {
    //Some error ocurred
    return nil, err
}
```

Nullwerte online lesen: https://riptutorial.com/de/go/topic/6379/nullwerte

Kapitel 45: Objekt orientierte Programmierung

Bemerkungen

Die Schnittstelle kann nicht mit Zeigerempfängern implementiert werden, da `*User` kein `User`

Examples

Structs

Go unterstützt benutzerdefinierte Typen in Form von Strukturen und Typ-Aliasnamen. structs sind zusammengesetzte Typen, die Komponententeile, aus denen der Strukturtyp besteht, werden *Felder genannt* . Ein Feld hat einen Typ und einen Namen, die einmalig sein müssen.

```
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}
}
```

Dies ist auch eine gesetzliche Syntax zur Definition von Strukturen

```
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"
```

Eingebettete Strukturen

Da eine Struktur auch ein Datentyp ist, kann sie als anonymes Feld verwendet werden. Die

äußere Struktur kann direkt auf die Felder der eingebetteten Struktur zugreifen, selbst wenn die Struktur aus einem anderen Paket stammt. Dieses Verhalten bietet eine Möglichkeit, einige oder alle Ihrer Implementierung von einem anderen Typ oder einer Gruppe von Typen abzuleiten.

```
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
}
```

Methoden

In Go ist eine Methode

> eine Funktion, die auf eine Variable eines bestimmten Typs wirkt, der als Empfänger bezeichnet wird

Der Empfänger kann alles sein, nicht nur `structs` sondern sogar eine `function` . Alias-Typen für integrierte Typen wie `int` , `string` und `bool` können eine Methode haben. Eine Ausnahme von dieser Regel besteht darin, dass `interfaces` (`structs`) keine Methoden haben können, da eine Schnittstelle ist eine abstrakte Definition und eine Methode ist eine Implementierung, bei der versucht wird, einen Kompilierungsfehler zu generieren.

Durch das Kombinieren von `structs` und `methods` Sie eine genaue Entsprechung einer `class` in der objektorientierten Programmierung.

Eine Methode in Go hat die folgende Signatur

```
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 Wertempfänger

Der Empfänger einer Methode ist aus Leistungsgründen normalerweise ein Zeiger, da wir keine Kopie der Instanz erstellen würden, wie dies beim Wertempfänger der Fall wäre. Dies gilt insbesondere, wenn der Empfängertyp eine Struktur ist. Ein anderer Grund, aus dem Empfängertyp einen Zeiger zu machen, wäre so, dass wir die Daten ändern könnten, auf die der Empfänger zeigt.

Ein Wertempfänger wird verwendet, um eine Änderung der Daten zu vermeiden, die der Empfänger enthält. Ein vaule-Empfänger kann einen Performance-Treffer verursachen, wenn der Empfänger eine große Struktur hat.

```
package main

type User struct {
    ID uint64
    FullName, Email string
```

```
}

// We do no 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 ponter
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
}
```

Schnittstelle & Polymorphismus

Schnittstellen bieten eine Möglichkeit, das Verhalten eines Objekts anzugeben. Wenn dies möglich ist, kann es hier verwendet werden. Eine Schnittstelle definiert eine Reihe von Methoden. Diese Methoden enthalten jedoch keinen Code, da sie abstrakt sind oder die Implementierung dem Benutzer der Schnittstelle überlassen wird. Im Gegensatz zu den meisten objektorientierten Sprachen können Schnittstellen in Go Variablen enthalten.

Polymorphismus ist die Essenz der objektorientierten Programmierung: Die Fähigkeit, Objekte unterschiedlichen Typs einheitlich zu behandeln, solange sie an derselben Schnittstelle haften. Go-Schnittstellen bieten diese Funktion auf eine direkte und intuitive Weise

```
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 fullfils 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)

}
```

Objekt orientierte Programmierung online lesen: https://riptutorial.com/de/go/topic/8801/objekt-orientierte-programmierung

Kapitel 46: OS-Signale

Syntax

- func Notify (c chan <- os.Signal, sig ... os.Signal)

Parameter

Parameter	Einzelheiten
c chan <- os.Signal	`channel` speziell vom Typ `os.Signal` ; einfach mit `sigChan := make(chan os.Signal)`
sig ... os.Signal	Liste der `os.Signal` Typen, die diesen `channel` abfangen und `os.Signal` . Weitere Optionen finden Sie unter https://golang.org/pkg/syscall/#pkg-constants .

Examples

Signale einem Kanal zuordnen

In vielen Fällen haben Sie Grund zu erfahren, wann Ihr Programm aufgefordert wird, beim Betriebssystem anzuhalten und Maßnahmen zu ergreifen, um den Status zu erhalten oder Ihre Anwendung zu bereinigen. Dazu können Sie das `os/signal` Paket aus der Standardbibliothek verwenden. Nachfolgend finden Sie ein einfaches Beispiel, um alle Signale des Systems einem Kanal zuzuweisen und anschließend auf diese Signale zu reagieren.

```
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)
```

```
    }
}
```

Wenn Sie das obige Skript ausführen, wird ein Kanal erstellt und dann blockiert, bis dieser Kanal ein Signal empfängt.

```
$ go run signals.go
^CReceived signal from OS: interrupt
```

Das `^C` oben ist der Tastaturbefehl `CTRL+C` der das `SIGINT` Signal sendet.

OS-Signale online lesen: https://riptutorial.com/de/go/topic/4497/os-signale

Kapitel 47: Pakete

Examples

Paketinitalisierung

Paket kann `init` Methoden haben, die **nur einmal** vor main ausgeführt werden.

```
package usefull

func init() {
    // init code
}
```

Wenn Sie nur die Paketinitialisierung ausführen möchten, ohne etwas davon zu referenzieren, verwenden Sie den folgenden Importausdruck.

```
import _ "usefull"
```

Paketabhängigkeiten verwalten

Go-Abhängigkeiten werden häufig mithilfe des Befehls `go get <package>` heruntergeladen. Das Paket wird im globalen / shared- `$GOPATH/src` . Dies bedeutet, dass eine einzelne Version jedes Pakets mit jedem Projekt verknüpft wird, das es als Abhängigkeit enthält. Dies bedeutet auch , dass , wenn ein neuer Entwickler Projekt setzt, werden sie `go get` die neueste Version von jeder Abhängigkeit.

Sie können jedoch die Erstellungsumgebung konsistent halten, indem Sie alle Abhängigkeiten eines Projekts in das Verzeichnis `vendor/` . Wenn Sie die Abhängigkeiten zusammen mit dem Repository Ihres Projekts beibehalten, können Sie die Versionsverwaltung für jedes Projekt durchführen und eine konsistente Umgebung für Ihren Build bereitstellen.

So sieht Ihre Projektstruktur aus:

```
$GOPATH/src/
├── github.com/username/project/
|   ├── main.go
|   ├── vendor/
|   |   ├── github.com/pkg/errors
|   |   ├── github.com/gorilla/mux
```

Verwenden eines anderen Paket- und Ordnernamens

Es ist vollkommen in Ordnung, einen anderen Paketnamen als den Ordnernamen zu verwenden. Wenn wir dies tun, müssen wir das Paket immer noch basierend auf der Verzeichnisstruktur importieren, aber nach dem Import müssen wir mit dem Namen, den wir in der Paketklausel verwenden, darauf verweisen.

Wenn Sie beispielsweise einen Ordner `$GOPATH/src/mypck` , haben wir eine Datei `a.go` :

```
package apple

const Pi = 3.14
```

Verwendung dieses Pakets:

```
package main

import (
    "mypck"
    "fmt"
)

func main() {
    fmt.Println(apple.Pi)
}
```

Obwohl dies funktioniert, sollten Sie einen guten Grund haben, den Paketnamen vom Ordnernamen abzuweichen (oder es kann zu Missverständnissen und Verwirrung führen).

Was nützt das?

Einfach. Ein Paketname ist ein Go- Identifikator :

```
identifier = letter { letter | unicode_digit } .
```

Damit können Unicode-Buchstaben in Bezeichnern verwendet werden, z. B. ist `αβ` ein gültiger Bezeichner in Go. Ordner- und Dateinamen werden nicht von Go, sondern vom Betriebssystem verarbeitet, und für verschiedene Dateisysteme gelten andere Einschränkungen. Es gibt tatsächlich viele Dateisysteme, die nicht alle gültigen Go-Bezeichner als Ordnernamen zulassen. Daher können Sie Ihre Pakete nicht so benennen, wie es die Sprachspezifikation sonst zulässt.

Da Sie die Option haben, andere Paketnamen als ihre Ordner zu verwenden, haben Sie die Möglichkeit, Ihre Pakete so zu benennen, wie es die Sprachspezifikation zulässt, unabhängig vom zugrunde liegenden Betriebssystem und Dateisystem.

Pakete importieren

Sie können ein einzelnes Paket mit der Anweisung importieren:

```
import "path/to/package"
```

oder mehrere Importe zusammenfassen:

```
import (
    "path/to/package1"
    "path/to/package2"
)
```

Dies wird in den entsprechenden aussehen `import` innerhalb der Pfade `$GOPATH` für `.go` Dateien und lässt Sie exportierten Namen durch Zugriff auf `packagename.AnyExportedName` .

Sie können auch auf lokale Pakete im aktuellen Ordner zugreifen, indem Sie Paketen mit `./` voranstellen. In einem Projekt mit einer Struktur wie folgt:

```
project
├── src
│   ├── package1
│   │   └── file1.go
│   └── package2
│       └── file2.go
└── main.go
```

Sie können dies in `main.go` , um den Code in `file1.go` und `file2.go` zu importieren:

```
import (
    "./src/package1"
    "./src/package2"
)
```

Da Paketnamen in verschiedenen Bibliotheken kollidieren können, möchten Sie möglicherweise ein Paket mit einem neuen Namen versehen. Sie können dies tun, indem Sie Ihrer import-Anweisung den Namen voranstellen, den Sie verwenden möchten.

```
import (
    "fmt" //fmt from the standardlibrary
    tfmt "some/thirdparty/fmt" //fmt from some other library
)
```

Auf diese Weise können Sie mit `fmt.*` das frühere `fmt` Paket und mit `tfmt.*` das letzte `fmt` Paket `tfmt.*` .

Sie können das Paket auch in den eigenen Namespace importieren, sodass Sie ohne das `package.` auf die exportierten Namen `package.` Präfix mit einem einzelnen Punkt als Alias:

```
import (
    . "fmt"
)
```

Das obige Beispiel importiert `fmt` in den globalen Namespace und ermöglicht beispielsweise den direkten Aufruf von `Printf` : Playground

Wenn Sie ein Paket importieren, aber keinen der exportierten Namen verwenden, gibt der Go-Compiler eine Fehlermeldung aus. Um dies zu umgehen, können Sie den Alias auf den Unterstrich setzen:

```
import (
    _ "fmt"
)
```

Dies kann nützlich sein, wenn Sie nicht direkt auf dieses Paket zugreifen, aber dessen `init` Funktionen ausführen müssen.

Hinweis:

Da die Paketnamen auf der Ordnerstruktur basieren, führen alle Änderungen an den Ordnernamen und Importreferenzen (einschließlich Berücksichtigung der Groß- und Kleinschreibung) zu einem Fehler bei der Kompilierzeit "Bei der Importkollision ohne Berücksichtigung der Groß- und Kleinschreibung" in Linux & OS-X, der schwer zu verfolgen ist und fix (die Fehlermeldung ist für bloße Sterbliche irgendwie kryptisch, da versucht wird, das Gegenteil zu vermitteln - der Vergleich ist aufgrund der Groß- / Kleinschreibung fehlgeschlagen).

Beispiel: "Pfad / zu / Paket1" vs "Pfad / zu / Paket1"

Live-Beispiel: https://github.com/akamai-open/AkamaiOPEN-edgegrid-golang/issues/2

Pakete online lesen: https://riptutorial.com/de/go/topic/401/pakete

Kapitel 48: Panik und Genesung

Bemerkungen

Dieser Artikel setzt Kenntnisse über Defer-Grundlagen voraus

Lesen Sie für die normale Fehlerbehandlung das Thema zur Fehlerbehandlung

Examples

Panik

Eine Panik stoppt den normalen Ausführungsablauf und verlässt die aktuelle Funktion. Alle zurückgestellten Aufrufe werden dann ausgeführt, bevor die Steuerung an die nächst höhere Funktion des Stapels übergeben wird. Die Funktion jedes Stacks wird beendet und verzögerte Aufrufe ausgeführt, bis die Panik mit einem verzögerten `recover()` wird oder bis die Panik `main()` und das Programm beendet. In diesem Fall werden das zur Panik bereitgestellte Argument und eine Stapelablaufverfolgung an `stderr` .

```
package main

import "fmt"

func foo() {
    defer fmt.Println("Exiting foo")
    panic("bar")
}

func main() {
    defer fmt.Println("Exiting main")
    foo()
}
```

Ausgabe:

```
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
```

Es ist wichtig zu wissen, dass die `panic` jeden Typ als Parameter akzeptiert.

Genesen

Wiederherstellen, wie der Name schon sagt, können versuchen, sich von einer `panic` zu erholen. Die Wiederherstellung *muss* in einer aufgeschobenen Anweisung versucht werden, da der normale Ausführungsablauf angehalten wurde. Die `recover` muss *direkt* im Gehäuse der verzögerten Funktion angezeigt werden. Anweisungen zum Wiederherstellen von Funktionen, die von zurückgestellten Funktionsaufrufen aufgerufen werden, werden nicht berücksichtigt. Der Aufruf von `recover()` gibt das angegebene Argument an die ursprüngliche Panik zurück, wenn das Programm gerade in Panik gerät. Wenn das Programm momentan nicht in Panik gerät, wird " `recover() 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")
}
```

Ausgabe:

```
Entering main
Recovered with message bar
Exiting main the normal way
```

Panik und Genesung online lesen: https://riptutorial.com/de/go/topic/4350/panik-und-genesung

Kapitel 49: Parallelität

Einführung

In Go wird Parallelität durch die Verwendung von Goroutinen erreicht, und die Kommunikation zwischen den Goroutinen erfolgt normalerweise über Kanäle. Es gibt jedoch auch andere Synchronisationsmittel wie Mutexe und Wartegruppen, die immer dann verwendet werden sollten, wenn sie bequemer als Kanäle sind.

Syntax

- go doWork () // führe die Funktion doWork als Goroutine aus
- ch: = make (chan int) // Neuen Kanal vom Typ int deklarieren
- ch <- 1 // Senden eines Kanals
- value = <-ch // Empfangen von einem Kanal

Bemerkungen

Goroutines in Go ähneln Threads in anderen Sprachen. Intern erstellt Go eine Anzahl von Threads (von `GOMAXPROCS`) und plant die Ausführung der Goroutines auf den Threads. Aufgrund dieses Designs sind die Parallelitätsmechanismen von Go hinsichtlich Speicherauslastung und Initialisierungszeit viel effizienter als Threads.

Examples

Erstellen von Goroutinen

Jede Funktion kann als Goroutine aufgerufen werden, indem dem Aufruf `go` das Schlüsselwort `go` vorangestellt wird:

```
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
```

Beachten Sie, dass der Rückgabewert der Funktion ignoriert wird.

Hallo Welt Goroutine

Einzelkanal, Einzel-Goroutine, einmal schreiben, einmal lesen.

```
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)
}
```

Lass es auf dem Spielplatz laufen

Der Kanal `ch` ist ein **ungepufferter oder synchroner Kanal** .

Die `time.Sleep` dient hier zur Veranschaulichung der Funktion `main()` , die auf den Kanal `ch` **wartet** , was bedeutet, dass das als Goroutine ausgeführte Funktionsliteral Zeit hat, einen Wert über diesen Kanal zu senden: Der Empfangsoperator `<-ch` blockiert die Ausführung von `main()` . Wenn dies nicht der Fall ist, wird die Goroutine beim Beenden von `main()` und hat keine Zeit, ihren Wert zu senden.

Warten auf Goroutinen

Go - Programme zu beenden , wenn die `main` endet , deshalb ist es üblich, für alle goroutines zu warten zu beenden. Eine gängige Lösung hierfür ist die Verwendung eines sync.WaitGroup-Objekts.

```
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")
}
```

Führen Sie das Beispiel auf dem Spielplatz aus

WaitGroup-Verwendung in der Reihenfolge der Ausführung:

1. Deklaration der globalen Variablen. Global zu machen ist der einfachste Weg, um es für alle Funktionen und Methoden sichtbar zu machen.
2. Erhöhen Sie den Zähler. Dies muss in der Haupt goroutine erfolgen, da es keine Garantie gibt , dass ein neu gestartet goroutine vor 4 aufgrund Speichermodell ausgeführt wird garantiert .
3. Zähler verringern. Dies muss am Ausgang einer Goroutine erfolgen. Durch die Verwendung eines zurückgestellten Anrufs stellen wir sicher, dass er immer dann aufgerufen wird, wenn die Funktion endet , unabhängig davon, wie sie endet.
4. Warten, bis der Zähler 0 erreicht. Dies muss in der Haupt-Goroutine erfolgen, um zu verhindern, dass das Programm beendet wird, bevor alle Goroutinen abgeschlossen sind.

* Parameter werden ausgewertet, bevor eine neue Goroutine gestartet wird . Daher ist es notwendig, ihre Werte explizit vor `wg.Add(10)` zu definieren, damit möglicherweise in Panik `wg.Add(10)` Code den Zähler nicht erhöht. Hinzufügen von 10 Elementen zur WaitGroup, sodass auf 10 Elemente `wg.Wait` wird, bevor `wg.Wait` das Steuerelement wieder an `main()` goroutine zurückgibt. Hier wird der Wert von i in der for-Schleife definiert.

Verwenden von Verschlüssen mit Goroutinen in einer Schleife

In einer Schleife ist die Schleifenvariable (val) im folgenden Beispiel eine einzelne Variable, deren Wert sich ändert, wenn sie über die Schleife geht. Daher muss man Folgendes tun, um jeden Wertwert tatsächlich an die Goroutine zu übergeben:

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

Wenn Sie einfach go `func(val interface{}) { ... }()` ausführen möchten, ohne val zu übergeben, ist der Wert von `val` der Wert von `val` , wenn die Goroutinen tatsächlich ausgeführt werden.

Ein anderer Weg, um den gleichen Effekt zu erzielen, ist:

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

```
}
```

Das seltsam aussehende `val := val` erstellt in jeder Iteration eine neue Variable, auf die die Goroutine zugreift.

Anhalten von Goroutinen

```
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 explicitely 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 mit zwei Goroutinen

```
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)
    }
}
```

Führen Sie eine geringfügig modifizierte Version dieses Codes in Go Playground aus

Parallelität online lesen: https://riptutorial.com/de/go/topic/376/parallelitat

Kapitel 50: Plugin

Einführung

Go bietet einen Plugin-Mechanismus, mit dem anderer Go-Code zur Laufzeit dynamisch verknüpft werden kann.

Ab Go 1.8 ist es nur unter Linux verwendbar.

Examples

Ein Plugin definieren und verwenden

```
package main

import "fmt"

var V int

func F() { fmt.Printf("Hello, number %d\n", V) }
```

Dies kann gebaut werden mit:

```
go build -buildmode=plugin
```

Und dann aus Ihrer Anwendung geladen und verwendet:

```
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"
```

Beispiel aus *The State of Go 2017* .

Plugin online lesen: https://riptutorial.com/de/go/topic/9150/plugin

Kapitel 51: Profilieren mit go tool pprof

Bemerkungen

Weitere Informationen zum Erstellen von Profilen finden Sie im Go-Blog .

Examples

Grundlegendes CPU- und Speicher-Profiling

Fügen Sie den folgenden Code in Ihr Hauptprogramm ein.

```
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()
    }
}
```

`go build main.go` **bauen Sie** das go-Programm, wenn es in main `go build main.go` Hauptprogramm mit Flags `main.exe -cpuprofile cpu.prof -memprof mem.prof` die im Code `main.exe -cpuprofile cpu.prof -memprof mem.prof` . Wenn die Profilerstellung für Testfälle durchgeführt wird, führen Sie die Tests mit denselben Flags aus `go test -cpuprofile cpu.prof -memprofile mem.prof`

Grundlegendes Speicherprofil

```
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
```

CPU / Block-Profilrate einstellen

```
// 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)
```

Verwenden von Benchmarks zum Erstellen eines Profils

Schreiben Sie für ein Nicht-Main-Paket sowie für Main **anstelle von Flags im Code Benchmarks** in das Testpaket. Beispiel:

```
func BenchmarkHello(b *testing.B) {
    for i := 0; i < b.N; i++ {
        fmt.Sprintf("hello")
    }
}
```

Führen Sie dann den Test mit der Profilmarkierung aus

> go test -cpuprofile cpu.prof -bench =.

Die Benchmarks werden ausgeführt und erstellen eine Prof-Datei mit dem Dateinamen cpu.prof (im obigen Beispiel).

Zugriff auf die Profildatei

Sobald eine Prof-Datei erstellt wurde, kann mit **den Tools** von **go** auf die Prof-Datei **zugegriffen werden** :

```
go tool pprof cpu.prof
```

Dadurch wird eine Befehlszeilenschnittstelle zum Erkunden des `profile`

Allgemeine Befehle umfassen:

```
(pprof) top
```

listet die wichtigsten Prozesse im Speicher auf

```
(pprof) peek
```

Listet alle Prozesse auf. Verwenden Sie *Regex,* um die Suche *einzugrenzen* .

```
(pprof) web
```

Öffnet ein Diagramm (im SVG-Format) des Prozesses.

Ein Beispiel für den `top` Befehl:

```
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%
    12.44s 12.34% 12.34%     27.87s 27.64%  runtime.mapaccess1
    10.94s 10.85% 23.19%     10.94s 10.85%  runtime.duffcopy
     9.45s  9.37% 32.56%     54.61s 54.16%  github.com/tester/test.(*Circle).Draw
     8.88s  8.81% 41.36%      8.88s  8.81%  runtime.aeshashbody
     7.90s  7.83% 49.20%     11.04s 10.95%  runtime.mapaccess1_fast64
     5.86s  5.81% 55.01%      9.59s  9.51%  github.com/tester/test.(*Circle).isCircle
     5.03s  4.99% 60.00%      8.89s  8.82%  github.com/tester/test.(*Circle).openCircle
     3.14s  3.11% 63.11%      3.14s  3.11%  runtime.aeshash64
     3.08s  3.05% 66.16%      7.85s  7.78%  runtime.mallocgc
     2.57s  2.55% 68.71%     12.03s 11.93%  runtime.memhash
```

Profilieren mit go tool pprof online lesen: https://riptutorial.com/de/go/topic/7748/profilieren-mit-go-tool-pprof

Kapitel 52: Protobuf in Go

Einführung

Protobuf oder Protocol Buffer codiert und decodiert Daten, so dass verschiedene Anwendungen oder Module, die in anderen Sprachen geschrieben sind, die große Anzahl von Nachrichten schnell und zuverlässig austauschen können, ohne den Kommunikationskanal zu überlasten. Mit protobuf ist die Leistung direkt proportional zur Anzahl der Nachrichten, die Sie normalerweise senden. Es komprimiert die Nachricht, um sie in einem serialisierten Binärformat zu senden, indem sie die Tools bereitstellt, um die Nachricht an der Quelle zu codieren und sie am Ziel zu decodieren.

Bemerkungen

Es gibt zwei Schritte zur Verwendung von **Protobuf** .

1. Zuerst müssen Sie die Protokollpufferdefinitionen kompilieren
2. Importieren Sie die obigen Definitionen mit der Support-Bibliothek in Ihr Programm.

gRPC-Unterstützung

Wenn eine Protodatei RPC-Dienste angibt, kann protoc-gen-go angewiesen werden, mit gRPC (http://www.grpc.io/) kompatiblen Code zu erzeugen. Übergeben Sie dazu den Parameter `plugins` an protoc-gen-go; Der übliche Weg ist, ihn in das Argument --go_out des Protokolls einzufügen:

```
protoc --go_out=plugins=grpc:. *.proto
```

Examples

Protobuf mit Go verwenden

Die Nachricht, die Sie serialisieren und senden möchten, die Sie in eine Datei **test.proto einfügen können** , die **Folgendes** enthält:

```
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;
  }
}
```

Um die Protokollpufferdefinition zu kompilieren, führen Sie protoc mit dem Parameter --go_out auf das Verzeichnis aus, in das Sie den Go-Code ausgeben möchten.

```
protoc --go_out=. *.proto
```

So erstellen und spielen Sie ein Testobjekt aus dem Beispielpaket

```
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.
}
```

Um zusätzliche Parameter an das Plugin zu übergeben, verwenden Sie eine durch Kommas getrennte Parameterliste, die durch einen Doppelpunkt vom Ausgabeverzeichnis getrennt ist:

```
protoc --go_out=plugins=grpc,import_path=mypackage:. *.proto
```

Protobuf in Go online lesen: https://riptutorial.com/de/go/topic/9729/protobuf-in-go

Kapitel 53: Protokollierung

Examples

Grundlegendes Drucken

Go verfügt über eine integrierte Protokollierungsbibliothek, die als `log` mit der häufig verwendeten Methode `Print` und seinen Varianten bezeichnet wird. Sie können die Bibliothek importieren und dann einige grundlegende Druckvorgänge durchführen:

```
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
}
```

Protokollierung in Datei

Es ist möglich, das Protokollziel mit etwas anzugeben, das die io.Writer-Schnittstelle anzeigt. Damit können wir uns in eine Datei einloggen:

```
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.Fatalln(err)
    }
    defer logfile.Close()

    log.SetOutput(logfile)
    log.Println("Log entry")
}
```

Ausgabe:

```
$ cat test.log
2016/07/26 07:29:05 Log entry
```

Protokollierung in Syslog

Es ist auch möglich, sich mit `log/syslog` wie folgt bei syslog `log/syslog` :

```
package main

import (
    "log"
    "log/syslog"
)

func main() {
    syslogger, err := syslog.New(syslog.LOG_INFO, "syslog_example")
    if err != nil {
        log.Fatalln(err)
    }

    log.SetOutput(syslogger)
    log.Println("Log entry")
}
```

Nach dem Ausführen können wir diese Zeile in syslog sehen:

```
Jul 26 07:35:21 localhost syslog_example[18358]: 2016/07/26 07:35:21 Log entry
```

Protokollierung online lesen: https://riptutorial.com/de/go/topic/3724/protokollierung

Kapitel 54: Reflexion

Bemerkungen

Die `reflect` Dokumente sind eine gute Referenz. In der allgemeinen Computerprogrammierung ist die **Reflexion** die Fähigkeit eines Programms **,** die Struktur und das Verhalten von **sich** zur **`runtime`** **.**

Basierend auf seinem strengen `static type` Typensystem hat Go lang einige Regeln (Gesetze der Reflexion)

Examples

Grundlegende 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"
```

Scheiben

```
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
```

Art des Wertes - Paket "reflektieren"

reflect.TypeOf kann verwendet werden, um den Variablentyp beim Vergleichen zu überprüfen

```
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))

}
```

Ausgabe :
Schnur
Hauptinformationen

Reflexion online lesen: https://riptutorial.com/de/go/topic/1854/reflexion

Kapitel 55: Scheiben

Einführung

Ein Slice ist eine Datenstruktur, die ein Array einkapselt, sodass der Programmierer beliebig viele Elemente hinzufügen kann, ohne sich um die Speicherverwaltung kümmern zu müssen. Slices können sehr effizient in Sub-Slices geschnitten werden, da die resultierenden Slices alle auf dasselbe interne Array zeigen. Go-Programmierer nutzen dies häufig, um das Kopieren von Arrays zu vermeiden, was normalerweise in vielen anderen Programmiersprachen der Fall wäre.

Syntax

- slice: = make ([] type, len, cap) // ein neues Slice erstellen
- Slice = Anhängen (Slice, Element) // Element an ein Slice anhängen
- Slice = Anhängen (Slice, Elemente ...) // Slice-Elemente an ein Slice anhängen
- len: = len (Slice) // Liefert die Länge eines Slice
- cap: = cap (slice) // Liefert die Kapazität eines Slice
- elNum: = copy (dst, slice) // kopiere den Inhalt eines Slice in ein anderes Slice

Examples

Anhängen an Scheibe

```
slice = append(slice, "hello", "world")
```

Zwei Scheiben zusammen hinzufügen

```
slice1 := []string{"!"}
slice2 := []string{"Hello", "world"}
slice  := append(slice1, slice2...)
```

Laufen Sie auf dem Go Playground

Elemente entfernen / "Scheiben schneiden"

Wenn Sie ein oder mehrere Elemente aus einem Slice entfernen müssen oder wenn Sie mit einem Sub-Slice eines anderen vorhandenen arbeiten müssen. Sie können die folgende Methode verwenden.

> In den folgenden Beispielen wird Slice of Int verwendet, dies funktioniert jedoch mit allen Slice-Typen.

Dafür brauchen wir ein Stück, von dem wir einige Elemente entfernen werden:

```
slice := []int{1, 2, 3, 4, 5, 6}
// > [1 2 3 4 5 6]
```

Wir brauchen auch die Indexe der Elemente, die entfernt werden sollen:

```
// 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
```

Und so können wir das Slice "schneiden" und unerwünschte Elemente entfernen:

```
// 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]
```

Um nur ein Element zu entfernen, müssen Sie einfach den Index dieses Elements als erstes UND als den letzten zu entfernenden Index angeben:

```
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]
```

Sie können auch Elemente vom Anfang des Slice entfernen:

```
newSlice6 := append(slice[:0], slice[last+1:]...)
// > [6]

// That can be simplified into
newSlice6 := slice[last+1:]
// > [6]
```

Sie können auch einige Elemente vom Ende des Slice entfernen:

```
newSlice7 := append(slice[:first], slice[first+1:len(slice)-1]...)
```

```
// > [1 2]

// That can be simplified into
newSlice7 := slice[:first]
// > [1 2]
```

> Wenn das neue Segment genau dieselben Elemente wie das erste enthalten muss, können Sie dasselbe verwenden, jedoch mit `last := first-1` .
> (Dies kann nützlich sein, wenn Ihre Indizes zuvor berechnet wurden.)

Länge und Kapazität

Scheiben haben sowohl Länge als auch Kapazität. Die Länge eines Slices ist die Anzahl der Elemente, die sich *aktuell* im Slice befinden, während die Kapazität der Anzahl der Elemente entspricht, die das Slice aufnehmen *kann,* bevor es neu zugewiesen werden muss.

Beim Erstellen eines Slice mithilfe der integrierten Funktion `make()` können Sie seine Länge und optional die Kapazität angeben. Wenn die Kapazität nicht explizit angegeben ist, hat dies die angegebene Länge.

```
var s = make([]int, 3, 5) // length 3, capacity 5
```

Sie können die Länge eines Slices mit der eingebauten Funktion `len()` überprüfen:

```
var n = len(s) // n == 3
```

Sie können die Kapazität mit der integrierten Funktion `cap()` überprüfen:

```
var c = cap(s) // c == 5
```

Von `make()` erzeugte Elemente werden auf den Nullwert für den Elementtyp des Slice gesetzt:

```
for idx, val := range s {
    fmt.Println(idx, val)
}
// output:
// 0 0
// 1 0
// 2 0
```

Führen Sie es auf play.golang.org aus

Sie können nicht auf Elemente zugreifen, die über die Länge eines Slice hinausgehen, auch wenn der Index nicht ausgelastet ist

```
var x = s[3] // panic: runtime error: index out of range
```

Solange die Kapazität die Länge überschreitet, können Sie neue Elemente anfügen, ohne sie neu zuzuordnen:

```
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
```

Wenn Sie an ein Slice anhängen, dem die Kapazität zur Aufnahme der neuen Elemente fehlt, wird das zugrunde liegende Array mit ausreichender Kapazität für Sie neu zugewiesen:

```
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
```

Es empfiehlt sich daher, beim Erstellen eines Slice ausreichend Kapazität zuzuweisen, wenn Sie wissen, wie viel Speicherplatz Sie benötigen, um unnötige Neuzuordnungen zu vermeiden.

Inhalt von einem Slice in ein anderes Slice kopieren

Wenn Sie den Inhalt eines Slice in ein anfangs leeres Slice kopieren möchten, können Sie folgende Schritte ausführen, um dies zu erreichen:

1. Erstellen Sie das Quell-Slice:

```
var sourceSlice []interface{} = []interface{}{"Hello",5.10,"World",true}
```

2. Erstellen Sie die Zielscheibe mit:

- Länge = Länge von sourceSlice

```
var destinationSlice []interface{} = make([]interface{},len(sourceSlice))
```

3. Da das zugrunde liegende Array des Ziel-Slice nun groß genug ist, um alle Elemente des Quell-Slice aufzunehmen, können wir die Elemente mithilfe der eingebauten `copy` :

```
copy(destinationSlice,sourceSlice)
```

Slices erstellen

Slices sind der typische Weg, mit dem Programmierer Datenlisten speichern.

Um eine Slice-Variable zu deklarieren, verwenden Sie die Syntax `[]Type` .

```
var a []int
```

Um eine Slice-Variable in einer Zeile zu deklarieren und zu initialisieren, verwenden Sie die Syntax `[]Type{values}` .

```
var a []int = []int{3, 1, 4, 1, 5, 9}
```

Eine andere Möglichkeit, ein Slice zu initialisieren, ist die Funktion `make` . Es gibt drei Argumente: den `Type` des Slice (oder der Karte), die `length` und die `capacity` .

```
a := make([]int, 0, 5)
```

Sie können mit `append` Elemente zu Ihrem neuen Slice hinzufügen.

```
a = append(a, 5)
```

Überprüfen Sie mit `len` die Anzahl der Elemente in Ihrem Slice.

```
length := len(a)
```

Überprüfen Sie die Kapazität Ihrer Scheibe mit einer `cap` . Die Kapazität ist die Anzahl der Elemente, die momentan für den Slice im Speicher vorhanden sind. Sie können bei Kapazität immer an ein Slice anhängen, da Go automatisch ein größeres Slice für Sie erstellt.

```
capacity := cap(a)
```

Sie können mit einer typischen Indizierungssyntax auf Elemente in einem Slice zugreifen.

```
a[0]  // Gets the first member of `a`
```

Sie können auch eine `for` Schleife über Scheiben mit `range` . Die erste Variable ist der Index im angegebenen Array und die zweite Variable ist der Wert für den Index.

```
for index, value := range a {
    fmt.Println("Index: " + index + " Value: " + value)  // Prints "Index: 0 Value: 5" (and
continues until end of slice)
}
```

Spielplatz gehen

Filtern eines Slice

So filtern Sie ein Slice, ohne ein neues zugrunde liegendes Array zuzuordnen:

```
// 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]
```

Nullwert des Slice

Der Nullwert von Slice ist `nil` , was die Länge und Kapazität `0` . Ein Slice ohne `nil` hat kein darunter liegendes Array. Es gibt aber auch Nicht-Null-Scheiben der Länge und Kapazität `0` , wie `[]int{}` oder `make([]int, 5)[5:]` .

Jeder Typ, der nil-Werte hat, kann in `nil` Slice umgewandelt werden:

```
s = []int(nil)
```

Um zu testen, ob ein Slice leer ist, verwenden Sie:

```
if len(s) == 0 {
    fmt.Ptintf("s is empty.")
}
```

Scheiben online lesen: https://riptutorial.com/de/go/topic/733/scheiben

Kapitel 56: Schleifen

Einführung

Als eine der grundlegendsten Funktionen beim Programmieren sind Schleifen in fast jeder Programmiersprache ein wichtiges Element. Mithilfe von Schleifen können Entwickler bestimmte Teile ihres Codes so einstellen, dass sie durch eine Reihe von Schleifen wiederholt werden, die als Iterationen bezeichnet werden. In diesem Thema werden verschiedene Arten von Schleifen und Anwendungen von Schleifen in Go behandelt.

Examples

Grundschleife

`for` ist die einzige Schleifenanweisung in go, daher könnte eine grundlegende Schleifenimplementierung folgendermaßen aussehen:

```
// 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
```

Pause und fortfahren

Das Ausbrechen der Schleife und das Fortfahren mit der nächsten Iteration wird wie in vielen anderen Sprachen auch in Go unterstützt:

```
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
```

Die `break` und `continue` Anweisungen akzeptieren zusätzlich eine optionale Beschriftung, mit der

äußere Schleifen für das Targeting mit der Anweisung identifiziert werden:

```
OuterLoop:
for {
    for {
        if allDone() {
            break OuterLoop
        }
        if innerDone() {
            continue OuterLoop
        }
        // do something
    }
}
```

Bedingte Schleife

Das Schlüsselwort `for` wird auch für bedingte Schleifen verwendet, üblicherweise `while` Schleifen in anderen Programmiersprachen.

```
package main

import (
    "fmt"
)

func main() {
    i := 0
    for i < 3 { // Will repeat if condition is true
        i++
        fmt.Println(i)
    }
}
```

spiele es auf dem Spielplatz

Wird ausgegeben:

```
1
2
3
```

Endlosschleife:

```
for {
    // This will run until a return or break.
}
```

Verschiedene Formen von for-Schleife

Einfaches Formular mit einer Variablen:

```
for i := 0; i < 10; i++ {
```

```
    fmt.Print(i, " ")
}
```

Verwendung von zwei Variablen (oder mehr):

```
for i, j := 0, 0; i < 5 && j < 10; i, j = i+1, j+2 {
    fmt.Println(i, j)
}
```

Ohne Initialisierungsanweisung:

```
i := 0
for ; i < 10; i++ {
    fmt.Print(i, " ")
}
```

Ohne einen Testausdruck:

```
for i := 1; ; i++ {
    if i&1 == 1 {
        continue
    }
    if i == 22 {
        break
    }
    fmt.Print(i, " ")
}
```

Ohne Zuwachsausdruck:

```
for i := 0; i < 10; {
    fmt.Print(i, " ")
    i++
}
```

Wenn alle drei Initialisierungs-, Test- und Inkrementierungsausdrücke entfernt werden, wird die Schleife unendlich:

```
i := 0
for {
    fmt.Print(i, " ")
    i++
    if i == 10 {
        break
    }
}
```

Dies ist ein Beispiel für eine Endlosschleife mit einem mit Null initialisierten Zähler:

```
for i := 0; ; {
    fmt.Print(i, " ")
    if i == 9 {
        break
```

```
    }
    i++
}
```

Wenn nur der Testausdruck verwendet wird (verhält sich wie eine typische while-Schleife):

```
i := 0
for i < 10 {
    fmt.Print(i, " ")
    i++
}
```

Verwenden Sie nur einen inkrementierten Ausdruck:

```
i := 0
for ; ; i++ {
    fmt.Print(i, " ")
    if i == 9 {
        break
    }
}
```

Über einen Bereich von Werten mit Index und Wert iterieren:

```
ary := [5]int{0, 1, 2, 3, 4}
for index, value := range ary {
    fmt.Println("ary[", index, "] =", value)
}
```

Durchlaufen Sie einen Bereich nur mit dem Index:

```
for index := range ary {
    fmt.Println("ary[", index, "] =", ary[index])
}
```

Durchlaufen Sie einen Bereich nur mit dem Index:

```
for index, _ := range ary {
    fmt.Println("ary[", index, "] =", ary[index])
}
```

Über einen Bereich iterieren, indem nur der Wert verwendet wird:

```
for _, value := range ary {
    fmt.Print(value, " ")
}
```

Durchlaufen eines Bereichs mithilfe des Schlüssels und des Werts für die Karte (möglicherweise nicht in der richtigen Reihenfolge):

```
mp := map[string]int{"One": 1, "Two": 2, "Three": 3}
for key, value := range mp {
```

```
    fmt.Println("map[", key, "] =", value)
}
```

Durchlaufen Sie einen Bereich nur mit der Taste für die Karte (möglicherweise nicht in der richtigen Reihenfolge):

```
for key := range mp {
    fmt.Print(key, " ") //One Two Three
}
```

Durchlaufen Sie einen Bereich nur mit der Taste für die Karte (möglicherweise nicht in der richtigen Reihenfolge):

```
for key, _ := range mp {
    fmt.Print(key, " ") //One Two Three
}
```

Durchlaufen Sie einen Bereich nur mit dem Wert für map (möglicherweise nicht in der richtigen Reihenfolge):

```
for _, value := range mp {
    fmt.Print(value, " ") //2 3 1
}
```

Durchlaufe einen Bereich für Kanäle (wird beendet, wenn der Kanal geschlossen ist):

```
ch := make(chan int, 10)
for i := 0; i < 10; i++ {
    ch <- i
}
close(ch)

for i := range ch {
    fmt.Print(i, " ")
}
```

Durchlaufen Sie einen Bereich für string (gibt Unicode-Codepunkte an):

```
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
```

Wie Sie sehen, hat `utf8str` 6 Runen (Unicode-Codepunkte) und 7 Bytes.

Zeitgesteuerte Schleife

```
package main

import(
    "fmt"
    "time"
)

func main() {
    for _ = range time.Tick(time.Second * 3) {
        fmt.Println("Ticking every 3 seconds")
    }
}
```

Schleifen online lesen: https://riptutorial.com/de/go/topic/975/schleifen

Kapitel 57: Schnittstellen

Bemerkungen

Schnittstellen in Go sind nur feste Methodensätze. Ein Typ implementiert *implizit* eine Schnittstelle, wenn deren Methodensatz eine Obermenge der Schnittstelle ist. *Es gibt keine Absichtserklärung.*

Examples

Einfache Schnittstelle

In Go ist eine Schnittstelle nur eine Reihe von Methoden. Wir verwenden eine Schnittstelle, um ein Verhalten eines bestimmten Objekts anzugeben.

```
type Painter interface {
    Paint()
}
```

Der implementierende Typ **muss nicht** angeben, dass er die Schnittstelle implementiert. Es reicht aus, Methoden mit derselben Signatur zu definieren.

```
type Rembrandt struct{}

func (r Rembrandt) Paint() {
    // use a lot of canvas here
}
```

Jetzt können wir die Struktur als Schnittstelle verwenden.

```
var p Painter
p = Rembrandt{}
```

Eine Schnittstelle kann von einer beliebigen Anzahl von Typen erfüllt (oder implementiert) werden. Ein Typ kann auch eine beliebige Anzahl von Schnittstellen implementieren.

```
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")
}
```

Die `Human` Struktur `OnlySinger` sowohl der `Singer` als auch der `Writer` Schnittstelle, die `OnlySinger` Struktur jedoch nur der `Singer` Schnittstelle.

Leere Schnittstelle

Es gibt einen leeren Schnittstellentyp, der keine Methoden enthält. Wir erklären es als `interface{}` . Das enthält keine Methoden, so dass jeder `type` erfüllt. Daher kann die leere Schnittstelle einen beliebigen Typwert enthalten.

```
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"}
```

Der häufigste Anwendungsfall für Schnittstellen besteht darin, sicherzustellen, dass eine Variable ein oder mehrere Verhalten unterstützt. Im Gegensatz dazu besteht der Hauptanwendungsfall für die leere Schnittstelle darin, eine Variable zu definieren, die unabhängig von ihrem konkreten Typ einen beliebigen Wert enthalten kann.

Um diese Werte wieder als ihre ursprünglichen Typen zu erhalten, müssen wir nur noch tun

```
i = a.(int)
s = a.(string)
m := a.(*StructType)
```

oder

```
i, ok := a.(int)
s, ok := a.(string)
m, ok := a.(*StructType)
```

`ok` zeigt an, ob die `interface a` in einen gegebenen Typ konvertierbar ist. Wenn dies nicht möglich ist, ist `ok false` .

Schnittstellenwerte

Wenn Sie eine Variable einer Schnittstelle deklarieren, kann sie jeden Werttyp speichern, der die von der Schnittstelle deklarierten Methoden implementiert.

Wenn wir `h` der `interface Singer` deklarieren, kann ein Wert vom Typ `Human` oder `OnlySinger`. Dies liegt daran, dass sie alle Methoden implementieren, die von der `Singer` Schnittstelle angegeben werden.

```
var h Singer
h = &human{}

h.Sing()
```

Festlegen des zugrunde liegenden Typs über die Schnittstelle

Unterwegs kann es manchmal nützlich sein, zu wissen, welcher Basistyp Sie übergeben haben. Dies kann mit einem Typwechsel erfolgen. Dies setzt voraus, dass wir zwei Strukturen haben:

```
type Rembrandt struct{}

func (r Rembrandt) Paint() {}

type Picasso struct{}

func (r Picasso) Paint() {}
```

Implementieren Sie die Painter-Schnittstelle:

```
type Painter interface {
    Paint()
}
```

Dann können wir diesen Schalter verwenden, um den zugrunde liegenden Typ zu bestimmen:

```
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")
    }
}
```

Überprüfung der Kompilierzeit, ob ein Typ eine Schnittstelle erfüllt

Schnittstellen und Implementierungen (Typen, die eine Schnittstelle implementieren) werden "getrennt". Es ist also eine berechtigte Frage, wie man zur Kompilierzeit prüfen kann, ob ein Typ eine Schnittstelle implementiert.

Eine Möglichkeit, den Compiler zu fragen, ob der Typ `T` die Schnittstelle `I` implementiert, besteht darin, eine Zuweisung unter Verwendung des Nullwerts für `T` oder des Zeigers auf `T` versuchen. Und wir können dem leeren Bezeichner zuweisen, um unnötigen Müll zu vermeiden:

```
type T struct{}

var _ I = T{}       // Verify that T implements I.
var _ I = (*T)(nil) // Verify that *T implements I.
```

Wenn `T` oder `*T` kein `I` implementieren, liegt ein Fehler bei der Kompilierung vor.

Diese Frage erscheint auch in der offiziellen FAQ: Wie kann ich garantieren, dass mein Typ eine Schnittstelle erfüllt?

Typ wechseln

Typenschalter können auch verwendet werden, um eine Variable abzurufen, die dem Typ des Falls entspricht:

```
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")
    }
}
```

Typ Assertion

Sie können mit Type Assertion auf den realen Datentyp der Schnittstelle zugreifen.

```
interfaceVariable.(DataType)
```

Beispiel für struct `MyType` das die Schnittstelle `Subber` implementiert:

```
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)
}
```

Ohne `.(*MyType)` wir keinen Zugriff auf `Msg` Field. Wenn wir `interfaceVar.Msg` versuchen, wird ein Kompilierungsfehler angezeigt:

```
interfaceVar.Msg undefined (type Subber has no field or method Msg)
```

Gehen Sie auf Schnittstellen aus einem mathematischen Aspekt

In der Mathematik, insbesondere der *Satztheorie* , haben wir eine Sammlung von Dingen, die als *Set bezeichnet wird,* und wir nennen diese Dinge als *Elemente* . Wir zeigen ein Set mit seinem Namen wie A, B, C, ... oder explizit, indem das Element in eine geschweifte Klammer geschrieben wird: {a, b, c, d, e}. Angenommen, wir haben ein beliebiges Element x und eine Menge Z. Die Schlüsselfrage lautet: "Wie können wir verstehen, dass x ein Mitglied von Z ist oder nicht?". Die Antwort auf diese Frage gibt der Mathematiker mit einem Konzept: **Charakteristikum** eines Satzes. *Charakteristische Eigenschaft* einer Menge ist ein Ausdruck, der eine Menge vollständig beschreibt. Zum Beispiel haben wir *natürliche Zahlen gesetzt,* die {0, 1, 2, 3, 4, 5, ...} sind. Wir können diese Menge mit diesem Ausdruck beschreiben: {a_n | $a_0 = 0 = a_{n-1} a_n + 1$}. In letzten Ausdruck $a_0 = 0 = a_{n-1} a_n + 1$ die charakteristische Eigenschaft der Menge der natürlichen Zahlen ist. **Wenn wir diesen Ausdruck haben, können wir diesen Satz vollständig erstellen** . Beschreiben Sie die *Anzahl* der *geraden Zahlen* auf diese Weise. Wir wissen, dass diese Menge aus folgenden Zahlen besteht: {0, 2, 4, 6, 8, 10, ...}. Mit einem Blick verstehen wir, dass alle diese Zahlen auch eine *natürliche Zahl* sind. Mit anderen Worten, *wenn wir der charakteristischen Eigenschaft der natürlichen Zahlen zusätzliche Bedingungen hinzufügen, können wir einen neuen Ausdruck erstellen, der diese Menge beschreibt* . Wir können also mit diesem Ausdruck beschreiben: {n | n ist ein Mitglied von natürlichen Zahlen *und* die Erinnerung an n auf 2 ist null}. Jetzt können wir einen Filter erstellen, der die charakteristische Eigenschaft eines Sets erhält, und einige gewünschte Elemente filtern, um Elemente unseres Sets zurückzugeben. Wenn wir beispielsweise einen Filter für natürliche Zahlen haben, können sowohl natürliche als auch gerade Zahlen diesen Filter passieren, aber wenn wir einen Filter für gerade Zahlen haben, können einige Elemente wie 3 und 137871 den Filter nicht passieren.

Die Definition der Schnittstelle in Go ist wie das Definieren der charakteristischen Eigenschaft und

der Mechanismus der Verwendung der Schnittstelle als Argument einer Funktion ist wie ein Filter, der erkennt, dass das Element ein Mitglied unserer gewünschten Menge ist oder nicht. Beschreiben wir diesen Aspekt mit 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
}
```

Die charakteristische Eigenschaft von `Number` ist alle Strukturen, die die `IsNumber` Methode haben, für `NaturalNumber` sind alle diejenigen, die die `IsNumber` und `IsNaturalNumber` Methode haben, und schließlich für `EvenNumber` sind alle Typen, die die `IsNumber` , `IsNaturalNumber` und `IsEvenNumber` Methode haben. Dank dieser Interpretation des Interfaces können wir leicht verstehen, dass das `interface{}` keine charakteristische Eigenschaft besitzt und alle Typen akzeptiert (da es keinen Filter zur Unterscheidung von Werten gibt).

Schnittstellen online lesen: https://riptutorial.com/de/go/topic/1221/schnittstellen

Kapitel 58: Speicher-Pooling

Einführung

sync.Pool speichert einen Cache mit zugewiesenen, aber nicht verwendeten Elementen für die spätere Verwendung. Dadurch werden Speicheränderungen bei häufig geänderten Sammlungen vermieden und eine effiziente, Thread-sichere Wiederverwendung des Speichers ermöglicht. Es ist nützlich, eine Gruppe temporärer Elemente zu verwalten, die von gleichzeitigen Clients eines Pakets gemeinsam genutzt werden, z. B. eine Liste von Datenbankverbindungen oder eine Liste von Ausgabepuffern.

Examples

sync.Pool

Mit der `sync.Pool` Struktur können wir Objekte zusammenfassen und wiederverwenden.

```
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)
}
```

Speicher-Pooling online lesen: https://riptutorial.com/de/go/topic/4647/speicher-pooling

Kapitel 59: SQL

Bemerkungen

Eine Liste der SQL-Datenbanktreiber finden Sie im offiziellen Go-Wiki-Artikel *SQLDrivers* .

Die SQL-Treiber werden importiert und mit _ , sodass sie *nur* für den Treiber verfügbar sind.

Examples

Abfragen

In diesem Beispiel wird gezeigt, wie eine Datenbank mit `database/sql` abgefragt wird. Beispiel: MySql-Datenbank.

```
package main

import (
    "log"
    "fmt"
    "database/sql"
    _ "github.com/go-sql-driver/mysql"
)

func main() {
    dsn := "mysql_username:CHANGEME@tcp(localhost:3306)/dbname"

    db, err := sql.Open("mysql", dsn)
    if err != nil {
        log.Fatal(err)
    }
    defer sql.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

Um MySQL zu aktivieren, ist ein Datenbanktreiber erforderlich. Zum Beispiel github.com/go-sql-

driver/mysql .

```
import (
    "database/sql"
    _ "github.com/go-sql-driver/mysql"
)
```

Datenbank öffnen

Das Öffnen einer Datenbank ist datenbankspezifisch. Hier finden Sie Beispiele für einige Datenbanken.

Quadrat 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: Verbinden und Einfügen sowie Entfernen und Aktualisieren und Abfragen

```
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
}
```

SQL online lesen: https://riptutorial.com/de/go/topic/1273/sql

Kapitel 60: String

Einführung

Eine Zeichenfolge ist in der Tat eine schreibgeschützte Teilmenge von Bytes. In Go enthält ein String-Literal immer eine gültige UTF-8-Darstellung seines Inhalts.

Syntax

- variableName: = "Hello World" // deklariere eine Zeichenfolge
- variableName: = "Hello World" // deklariert eine unformatierte Zeichenfolge
- variableName: = "Hallo" + "Welt" // verkettet Zeichenketten
- substring: = "Hello World" [0: 4] // Einen Teil der Zeichenfolge erhalten
- letter: = "Hello World" [6] // Holen Sie sich ein Zeichen der Zeichenfolge
- fmt.Sprintf ("% s", "Hello World") // formatiert eine Zeichenfolge

Examples

String-Typ

Mit dem `string` können Sie Text speichern, der aus einer Reihe von Zeichen besteht. Es gibt mehrere Möglichkeiten, Strings zu erstellen. Eine Literalzeichenfolge wird erstellt, indem der Text zwischen Anführungszeichen gesetzt wird.

```
text := "Hello World"
```

Da Go-Zeichenfolgen UTF-8 unterstützen, ist das vorige Beispiel vollkommen gültig. Zeichenfolgen enthalten beliebige Bytes, was nicht unbedingt bedeutet, dass jede Zeichenfolge gültige UTF-8-Werte enthält, während Zeichenfolgenliterale immer gültige UTF-8-Sequenzen enthalten.

Der Nullwert von Zeichenfolgen ist eine leere Zeichenfolge `""` .

Zeichenfolgen können mit dem Operator `+` verkettet werden.

```
text := "Hello " + "World"
```

Strings können auch mit den Backticks `` definiert werden. Dadurch wird ein Roh-String-Literal erstellt, das bedeutet, dass Zeichen nicht mit Escapezeichen versehen werden.

```
text1 := "Hello\nWorld"
text2 := `Hello
World`
```

Im vorherigen Beispiel entgeht `text1` dem `\n` Zeichen, das eine neue Zeile darstellt, während `text2`

das neue `text2` direkt enthält. Wenn Sie `text1 == text2` vergleichen, ist das Ergebnis `true` .

Allerdings würde `text2 := `Hello\nWorld`` das Zeichen `\n` nicht entgehen, was bedeutet, dass die Zeichenfolge den Text `Hello\nWorld` ohne eine neue Zeile enthält. `text1 := "Hello\\nWorld"` wäre gleichbedeutend mit der Eingabe von `text1 := "Hello\\nWorld"` .

Text formatieren

Package `fmt` implementiert Funktionen und Format Text drucken Format *Verben* verwenden. Verben werden mit einem Prozentzeichen dargestellt.

Allgemeine Verben:

```
%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
```

Ganze Zahl:

```
%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"
```

Fließkomma und komplexe Bestandteile:

```
%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
```

Zeichenfolge und Segment von Bytes (gleichwertig mit diesen Verben behandelt):

```
%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
```

Zeiger:

```
%p    // base 16 notation, with leading 0x
```

Mit den Verben können Sie Zeichenfolgen erstellen, die mehrere Typen verketten:

```
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)
```

Die Funktion `Sprintf` formatiert die Zeichenfolge im ersten Parameter, ersetzt die Verben durch den Wert der Werte in den nächsten Parametern und gibt das Ergebnis zurück. Wie `Sprintf` die Funktion `Printf` Formate auch , sondern das Ergebnis der Rücksendung druckt die Zeichenfolge.

String-Paket

- 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, baz
  ```

- strings.Replace

  ```
  fmt.Println(strings.Replace("foobar", "bar", "baz", 1)) // foobaz
  ```

- strings.Split

  ```
  s := "foo, bar, bar"
  fmt.Println(strings.Split(s, ", ")) // [foo bar baz]
  ```

- strings.ToLower

  ```
  fmt.Println(strings.ToLower("FOOBAR")) // foobar
  ```

- strings.ToUpper

```
fmt.Println(strings.ToUpper("foobar")) // FOOBAR
```

- `strings.TrimSpace`

```
fmt.Println(strings.TrimSpace("  foobar  ")) // foobar
```

Mehr: https://golang.org/pkg/strings/ .

String online lesen: https://riptutorial.com/de/go/topic/9666/string

Kapitel 61: Structs

Einführung

Strukturen sind Mengen verschiedener Variablen, die zusammengepackt sind. Die struct selbst ist nur ein *Paket,* das Variablen enthält und leicht zugänglich ist.

Anders als in C können Gos Strukturen mit Methoden verbunden werden. Es erlaubt ihnen auch, Schnittstellen zu implementieren. Dadurch ähneln die Strukturen von Go Objekten, aber sie fehlen (wahrscheinlich absichtlich) einige wichtige Funktionen, die in objektorientierten Sprachen bekannt sind, wie Vererbung.

Examples

Grundlegende Erklärung

Eine grundlegende Struktur wird wie folgt deklariert:

```
type User struct {
    FirstName, LastName string
    Email               string
    Age                 int
}
```

Jeder Wert wird als Feld bezeichnet. Die Felder werden normalerweise einzeln geschrieben, wobei der Name des Felds vor seinem Typ steht. Aufeinanderfolgende Felder desselben Typs können im obigen Beispiel als `FirstName` und `LastName` kombiniert werden.

Exportierte vs. nicht exportierte Felder (privat und öffentlich)

Strukturfelder, deren Namen mit einem Großbuchstaben beginnen, werden exportiert. Alle anderen Namen sind nicht exportiert.

```
type Account struct {
    UserID      int    // exported
    accessToken string // unexported
}
```

Auf nicht enthaltene Felder kann nur durch Code innerhalb desselben Pakets zugegriffen werden. Wenn Sie also jemals auf ein Feld aus einem *anderen* Paket zugreifen, muss dessen Name mit einem Großbuchstaben beginnen.

```
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
    }
}
```

Doch innerhalb der `bank` Paket können Sie sowohl Benutzer - ID und accessToken ohne Probleme zugreifen.

Das Paket `bank` könnte wie folgt realisiert werden:

```
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
}
```

Komposition und Einbettung

Die Zusammensetzung bietet eine Alternative zur Vererbung. Eine Struktur kann in der Deklaration einen anderen Typ nach Namen enthalten:

```
type Request struct {
    Resource string
}

type AuthenticatedRequest struct {
    Request
    Username, Password string
}
```

Im obigen Beispiel enthält `AuthenticatedRequest` vier öffentliche Mitglieder: `Resource` , `Request` , `Username` und `Password` .

Zusammengesetzte Strukturen können instanziiert und auf dieselbe Weise wie normale Strukturen verwendet werden:

```
func main() {
    ar := new(AuthenticatedRequest)
    ar.Resource = "example.com/request"
    ar.Username = "bob"
    ar.Password = "P@ssw0rd"
    fmt.Printf("%#v", ar)
}
```

spiele es auf dem Spielplatz

Einbetten

Im vorherigen Beispiel ist `Request` ein eingebettetes Feld. Die Zusammensetzung kann auch durch Einbetten eines anderen Typs erreicht werden. Dies ist z. B. nützlich, um eine Struktur mit mehr Funktionen zu dekorieren. Wenn Sie beispielsweise mit dem Beispiel "Ressource" fortfahren, möchten wir eine Funktion, die den Inhalt des Felds "Resource" so formatiert, dass `http://` oder `https://` vorangestellt wird. Wir haben zwei Möglichkeiten: Erstellen Sie die neuen Methoden in AuthenticatedRequest oder **binden Sie** sie aus einer anderen Struktur ein:

```
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
}
```

Und jetzt könnte die Hauptfunktion Folgendes tun:

```
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)
}
```

Sehen Sie sich das `AuthenticatedRequest` mit einer eingebetteten `ResourceFormatter` Struktur an.

Aber der Nachteil davon ist , dass Sie keine Objekte außerhalb Ihrer Komposition zugreifen können. `ResourceFormatter` kann also nicht über `AuthenticatedRequest` auf Mitglieder zugreifen.

spiele es auf dem Spielplatz

Methoden

Strukturmethoden sind Funktionen sehr ähnlich:

```
type User struct {
    name string
```

```
}

func (u User) Name() string {
    return u.name
}

func (u *User) SetName(newName string) {
    u.name = newName
}
```

Der einzige Unterschied besteht in der Hinzufügung des Methodenempfängers. Sie kann entweder als Instanz des Typs oder als Zeiger auf eine Instanz des Typs deklariert werden. Da `SetName()` die Instanz verändert, muss der Empfänger ein Zeiger sein, um eine permanente Änderung in der Instanz zu bewirken.

Zum Beispiel:

```
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())
}
```

Spielplatz gehen

Anonyme Struktur

Es ist möglich, eine anonyme Struktur zu erstellen:

```
data := struct {
    Number int
    Text   string
} {
    42,
    "Hello world!",
}
```

Vollständiges Beispiel:

```
package main

import (
    "fmt"
)

func main() {
    data := struct {Number int; Text string}{42, "Hello world!"} // anonymous struct
    fmt.Printf("%+v\n", data)
}
```

spiele es auf dem Spielplatz

Stichworte

Strukturfeldern können Tags zugeordnet sein. Diese Tags können vom `reflect` Paket gelesen werden, um vom Entwickler benutzerdefinierte Informationen zu einem Feld zu erhalten.

```
struct Account {
    Username      string `json:"username"`
    DisplayName   string `json:"display_name"`
    FavoriteColor string `json:"favorite_color,omitempty"`
}
```

Im obigen Beispiel werden die Tags verwendet, um die Schlüsselnamen zu ändern, die vom `encoding/json` Paket verwendet werden, wenn JSON gemarshallt oder das Marshallen aufgehoben wird.

Das Tag kann ein beliebiger Zeichenfolgenwert sein. Es wird jedoch empfohlen, durch Leerzeichen getrennte `key:"value"` -Paare:

```
struct StructName {
    FieldName int `package1:"customdata,moredata" package2:"info"`
}
```

Die struct-Tags, die mit dem Paket `encoding/xml` und `encoding/json` werden, werden in der Standard-Bibliothek verwendet.

Strukturkopien erstellen.

Eine Struktur kann einfach durch Zuweisung kopiert werden.

```
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 obigem Fall sind `'t'` und 'u' jetzt separate Objekte (Strukturwerte).

Da `T` keine Referenztypen (Slices, Map, Channels) als Felder enthält, können `t` und `u` oben geändert werden, ohne sich gegenseitig zu beeinflussen.

```
fmt.Printf("t.I = %d, u.I = %d\n", t.I, u.I) // t.I = 100, u.I = 1
```

Wenn `T` einen Referenztyp enthält, beispielsweise:

```
type T struct {
    I  int
    S  string
    xs []int // a slice is a reference type
}
```

Dann würde eine einfache Kopie durch Zuweisung den Wert des Slice-Type-Felds in das neue Objekt kopieren. Dies würde dazu führen, dass zwei verschiedene Objekte auf dasselbe Slice-Objekt verweisen.

```
// 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
```

Da sich u und t über ihr Feld auf dasselbe Segment beziehen, würde das Aktualisieren eines Werts im Slice eines Objekts die Änderung im anderen Objekt widerspiegeln.

```
// 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]
```

Daher muss besonders darauf geachtet werden, dass diese Referenztyp-Eigenschaft nicht zu unbeabsichtigtem Verhalten führt.

Um beispielsweise Objekte zu kopieren, kann eine explizite Kopie des Slice-Felds durchgeführt werden:

```
// 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

Ein Wert eines Strukturtyps kann mit einem *Strukturliteral* geschrieben werden, das Werte für seine Felder angibt.

```
type Point struct { X, Y int }
p := Point{1, 2}
```

Das obige Beispiel gibt jedes Feld in der richtigen Reihenfolge an. Was nicht sinnvoll ist, da sich Programmierer die genauen Felder in der richtigen Reihenfolge merken müssen. Häufiger kann eine Struktur durch Auflisten einiger oder aller Feldnamen und ihrer entsprechenden Werte initialisiert werden.

```
anim := gif.GIF{LoopCount: nframes}
```

Ausgelassene Felder werden für ihren Typ auf Null gesetzt.

Hinweis: **Die beiden Formulare können nicht in demselben Literal gemischt werden.**

Leere Struktur

Eine Struktur ist eine Folge benannter Elemente, genannt Felder, von denen jedes einen Namen und einen Typ hat. Leere Struktur hat keine Felder wie diese anonyme leere Struktur:

```
var s struct{}
```

Oder wie dieser benannte leere Strukturtyp:

```
type T struct{}
```

Das Interessante an der leeren Struktur ist, dass sie null ist (versuchen Sie den Go Playground):

```
fmt.Println(unsafe.Sizeof(s))
```

Dies gibt `0` , sodass die leere Struktur selbst keinen Speicherplatz benötigt. Es ist also eine gute Option für das Beenden eines Kanals, wie (versuchen Sie es mit 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.")
}
```

Structs online lesen: https://riptutorial.com/de/go/topic/374/structs

Kapitel 62: Testen

Einführung

Go verfügt über eigene Testeinrichtungen, die alles bieten, um Tests und Benchmarks durchzuführen. Im Gegensatz zu den meisten anderen Programmiersprachen ist ein separates Test-Framework oft nicht erforderlich, obwohl einige vorhanden sind.

Examples

Grundtest

`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)
    }
}
```

Um den Test auszuführen, verwenden Sie einfach den Befehl `go test` :

```
$ go test
ok      test_app    0.005s
```

Verwenden Sie das Flag `-v` , um die Ergebnisse jedes Tests `-v` :

```
$ go test -v
=== RUN   TestSum
--- PASS: TestSum (0.00s)
PASS
ok      _/tmp    0.000s
```

Verwenden Sie den Pfad `./...` , um Unterverzeichnisse rekursiv zu testen:

```
$ 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
```

Einen bestimmten Test ausführen:
Wenn mehrere Tests vorhanden sind und Sie einen bestimmten Test ausführen möchten, können Sie dies folgendermaßen tun:

```
go test -v -run=<TestName> // will execute only test with this name
```

Beispiel:

```
go test -v run=TestSum
```

Benchmark-Tests

Wenn Sie Benchmarks messen möchten, fügen Sie eine Testmethode wie folgt hinzu:

`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)
    }
}
```

Dann, um einen einfachen Benchmark auszuführen:

```
$ go test -bench=.
BenchmarkSum-8    2000000000             0.49 ns/op
ok      so/sum    1.027s
```

Tischgesteuerte Unit-Tests

Diese Art des Testens ist eine beliebte Technik zum Testen mit vordefinierten Eingabe- und Ausgabewerten.

Erstellen Sie eine Datei namens `main.go` mit Inhalt:

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println(Sum(4, 5))
}

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

Nach dem Ausführen sehen Sie, dass die Ausgabe `9` . Obwohl die `Sum` Funktion recht einfach aussieht, sollten Sie Ihren Code testen. Dazu erstellen wir eine weitere Datei namens `main_test.go` im selben Ordner wie `main.go` mit folgendem 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)
        }
    }
```

```
}
```

Wie Sie sehen, wird ein Teil der anonymen Strukturen erstellt, die jeweils eine Reihe von Eingaben und das erwartete Ergebnis enthalten. Dies ermöglicht die Erstellung einer großen Anzahl von Testfällen an einem Ort, die dann in einer Schleife ausgeführt werden, wodurch die Codewiederholung reduziert und die Übersichtlichkeit verbessert wird.

Beispieltests (Selbstdokumentationstests)

Diese Art von Tests stellt sicher, dass Ihr Code ordnungsgemäß kompiliert wird und in der generierten Dokumentation für Ihr Projekt angezeigt wird. Darüber hinaus können die Beispieltests behaupten, dass Ihr Test eine korrekte Ausgabe erzeugt.

`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
}
```

Um Ihren Test ausführen ausführen `go test` - `sum go test ./sum go test` in den Ordner, die Dateien oder die zwei Dateien in einem Unterordner namens setzen `sum` und dann aus den übergeordneten Ordner ausführen `go test ./sum` . In beiden Fällen erhalten Sie eine Ausgabe ähnlich der folgenden:

```
ok      so/sum    0.005s
```

Wenn Sie sich fragen, wie dies Ihren Code testet, finden Sie hier eine weitere Beispielfunktion, die den Test tatsächlich nicht besteht:

```
func ExampleSum_fail() {
```

```
    x := Sum(1, 2)
    fmt.Println(x)

    // Output:
    // 5
}
```

Wenn Sie `go test` ausführen, erhalten Sie folgende Ausgabe:

```
$ go test
--- FAIL: ExampleSum_fail (0.00s)
got:
3
want:
5
FAIL
exit status 1
FAIL    so/sum    0.006s
```

Wenn Sie die Dokumentation für Ihr `sum` anzeigen möchten, führen Sie einfach Folgendes aus:

```
go doc -http=:6060
```

und navigieren Sie zu http: // localhost: 6060 / pkg / FOLDER / sum / , wobei *FOLDER* der Ordner ist, der das `sum` (in diesem Beispiel `so`). Die Dokumentation für die Summenmethode sieht folgendermaßen aus:

Package sum

```
import "so/sum"
```


Overview ▾

Package sum is a sample package for test purpos

Index ▾

Examples

Package files

`defer` und dann mit der `defer` Anweisung den übergebenen Funktionsaufruf wieder seiner ursprünglichen Funktion zuweisen.

```
var validate = validateDTD

// ParseXML parses b for XML elements and values, and returns them as a map of
// string key/value pairs.
func ParseXML(b []byte) (map[string]string, error) {
    // we don't care about validating against DTD in our unit test
    if err := validate(b); err != nil {
        return err
    }

    // code to parse b etc.
}

func validateDTD(b []byte) error {
    // get the DTD from some external storage, use it to validate b etc.
}
```

In unserem Unit-Test

```
func TestParseXML(t *testing.T) {
    // assign the original validate function to a variable.
    originalValidate = validate
    // use the mockValidate function in this test.
    validate = mockValidate
    // defer the re-assignment back to the original validate function.
    defer func() {
       validate = originalValidate
    }()

    var input []byte
    actual, err := ParseXML(input)
    // assertion etc.
}

func mockValidate(b []byte) error {
    return nil // always return nil since we don't care
}
```

Testen mit der Funktion setUp und tearDown

Sie können eine SetUp- und TearDown-Funktion einstellen.

- Eine SetUp-Funktion bereitet Ihre Umgebung für Tests vor.
- Eine TearDown-Funktion führt ein Rollback durch.

Dies ist eine gute Option, wenn Sie Ihre Datenbank nicht ändern können und ein Objekt erstellen müssen, das ein von der Datenbank mitgebrachtes Objekt simuliert oder bei jedem Test eine Konfiguration initiiert.

Ein dummes Beispiel wäre:

```
// 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)
    }
}
```

Ein anderes Beispiel wäre das Vorbereiten der Datenbank zum Testen und Ausführen eines Rollbacks

```
 // 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)
    }
}
```

Anzeigen der Codeabdeckung im HTML-Format

Führen Sie den `go test` wie üblich aus, jedoch mit dem `coverprofile` Flag. Verwenden Sie `go tool` um die Ergebnisse als HTML anzuzeigen.

```
    go test -coverprofile=c.out
    go tool cover -html=c.out
```

Testen online lesen: https://riptutorial.com/de/go/topic/1234/testen

Kapitel 63: Text + HTML-Vorlage

Examples

Einzelartikel-Vorlage

Beachten Sie die Verwendung von `{{.}}` , Um das Element innerhalb der Vorlage auszugeben.

```
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")
}
```

Ergebnisse in:

```
Dear Professor Jones, How are you?
```

Vorlage für mehrere Elemente

Beachten Sie die Verwendung von `{{range .}}` Und `{{end}}` , um die Sammlung zu durchlaufen.

```
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())
```

```
    }

    tmpl.Execute(os.Stdout, []string{"Harry", "Jane", "Lisa", "George"})
}
```

Ergebnisse in:

```
Dear Harry, Jane, Lisa, George,  How are you?
```

Vorlagen mit eigener Logik

In diesem Beispiel wird eine Funktionszuordnung mit dem Namen `funcMap` über die Methode `Funcs()` an die Vorlage `Funcs()` und dann innerhalb der Vorlage aufgerufen. Hier wird die Funktion `increment()` verwendet, um das Fehlen einer weniger oder gleichen Funktion in der Schablonensprache zu umgehen. Beachten Sie in der Ausgabe, wie der letzte Artikel in der Sammlung behandelt wird.

Ein `-` am Anfang `{{-` oder Ende `-}}` wird zum Abschneiden von Leerzeichen verwendet und kann dazu verwendet werden, die Vorlage lesbarer zu machen.

```
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?`
    )

    tmpl, err := template.New("letter").Funcs(funcMap).Parse(letter)
    if err != nil {
        fmt.Println(err.Error())
    }

    tmpl.Execute(os.Stdout, []string{"Harry", "Jane", "Lisa", "George"})
}
```

Ergebnisse in:

```
Dear Harry, Jane, Lisa, and George; How are you?
```

Vorlagen mit Strukturen

Beachten Sie, wie Feldwerte mit `{{.FieldName}}` abgerufen werden.

```
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})
}
```

Ergebnisse 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-Vorlagen

Beachten Sie den unterschiedlichen Paketimport.

```
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><img src="{{.AvatarUrl}}"></td>
<td>{{.FirstName}} {{.LastName}}</td>
<td>{{.Street}}, {{.City}}, {{.State}} {{.Zip}}</td>
</tr>
{{end}}
</table></body></html>`
```

```
    )

    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",
        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})
}
```

Ergebnisse in:

```
<html><body><table>
<tr><th></th><th>Name</th><th>Address</th></tr>

<tr>
<td><img src="harry.png"></td>
<td>Harry Jones</td>
<td>1234 Main St., Springfield, IL 12345-6789</td>
</tr>

<tr>
<td><img src="jane.png"></td>
<td>Jane Sherman</td>
<td>8511 1st Ave., Dayton, OH 18515-6261</td>
</tr>

</table></body></html>
```

Wie HTML-Vorlagen die Injektion von schädlichem Code verhindern

Zunächst einmal, was passieren kann, wenn `text/template` für HTML verwendet wird. Beachten Sie Harrys `FirstName` Eigenschaft).

```
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><img src="{{.AvatarUrl}}"></td>
<td>{{.FirstName}} {{.LastName}}</td>
<td>{{.Street}}, {{.City}}, {{.State}} {{.Zip}}</td>
</tr>
{{end}}
</table></body></html>`
    )

    tmpl, 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",
    }

    tmpl.Execute(os.Stdout, []Person{harry, jane})
}
```

Ergebnisse in:

```
<html><body><table>
```

```
<tr><th></th><th>Name</th><th>Address</th></tr>

<tr>
<td><img src="harry.png"></td>
<td>Harry<script>alert("You've been hacked!")</script> Jones</td>
<td>1234 Main St., Springfield, IL 12345-6789</td>
</tr>

<tr>
<td><img src="jane.png"></td>
<td>Jane Sherman</td>
<td>8511 1st Ave., Dayton, OH 18515-6261</td>
</tr>

</table></body></html>
```

Wenn das obige Beispiel von einem Browser aus aufgerufen wird, würde das Skript eine Warnung erzeugen. Wenn statt `html/template` statt `text/template` importiert würde, wäre das Skript sicher bereinigt:

```
<html><body><table>
<tr><th></th><th>Name</th><th>Address</th></tr>

<tr>
<td><img src="harry.png"></td>
<td>Harry&lt;script&gt;alert(&#34;You&#39;ve been hacked!&#34;)&lt;/script&gt; Jones</td>
<td>1234 Main St., Springfield, IL 12345-6789</td>
</tr>

<tr>
<td><img src="jane.png"></td>
<td>Jane Sherman</td>
<td>8511 1st Ave., Dayton, OH 18515-6261</td>
</tr>

</table></body></html>
```

Das zweite Ergebnis würde beim Laden in einem Browser verstümmelt aussehen, würde jedoch nicht dazu führen, dass ein potenziell schädliches Skript ausgeführt wird.

Text + HTML-Vorlage online lesen: https://riptutorial.com/de/go/topic/3888/text-plus-html-vorlage

Kapitel 64: Typumwandlungen

Examples

Grundtypumwandlung

Es gibt zwei grundlegende Arten der Typkonvertierung 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.
```

Testen der Schnittstellenimplementierung

Da Go die implizite Schnittstellenimplementierung verwendet, wird kein Fehler bei der Kompilierung angezeigt, wenn Ihre Struktur keine Schnittstelle implementiert, die Sie implementieren wollten. Sie können die Implementierung mithilfe von Typumwandlung explizit testen: type MyInterface-Schnittstelle {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
```

Implementieren Sie ein Einheitensystem mit Typen

Dieses Beispiel zeigt, wie das Typsystem von Go zum Implementieren eines Einheitensystems verwendet werden kann.

```
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("%vmps = %vkmh = %vmps\n", mps, kmh, mps2)
}
```

Im Spielplatz öffnen

Typumwandlungen online lesen: https://riptutorial.com/de/go/topic/2851/typumwandlungen

Kapitel 65: Variablen

Syntax

- var x int // deklariere die Variable x mit dem Typ int
- var s string // deklariere die Variable s mit dem Typ string
- x = 4 // x Wert definieren
- s = "foo" // s-Wert definieren
- y: = 5 // deklarieren und definieren y, wenn dessen Typ auf int gesetzt wird
- f: = 4.5 // deklarieren und definieren, wenn f den Typ auf float64 bezieht
- b: = "bar" // deklarieren und definieren b, wenn dessen Typ auf string verweist

Examples

Grundlegende Variablendeklaration

Go ist eine statisch typisierte Sprache, was bedeutet, dass Sie im Allgemeinen den Typ der verwendeten Variablen angeben müssen.

```
// 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
```

Mehrfachzuweisung von Variablen

In Go können Sie mehrere Variablen gleichzeitig deklarieren.

```
// 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`
```

Wenn eine Funktion mehrere Werte zurückgibt, können Sie den Werten basierend auf den

Rückgabewerten der Funktion auch Werte zuweisen.

```
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
```

Leere Kennung

Go gibt einen Fehler aus, wenn eine Variable nicht verwendet wird, um Sie zu ermutigen, besseren Code zu schreiben. Es gibt jedoch Situationen, in denen Sie wirklich keinen in einer Variablen gespeicherten Wert verwenden müssen. In diesen Fällen verwenden Sie einen "leeren Bezeichner" _ , um den zugewiesenen Wert zuzuweisen und zu verwerfen.

Einem leeren Bezeichner kann ein Wert eines beliebigen Typs zugewiesen werden. Er wird am häufigsten in Funktionen verwendet, die mehrere Werte zurückgeben.

Mehrere Rückgabewerte

```
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
}
```

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)
    }

}
```

Typ einer Variable prüfen

Es gibt Situationen, in denen Sie nicht sicher sind, welcher Typ eine Variable ist, wenn sie von einer Funktion zurückgegeben wird. Sie können den Variablentyp immer mit `var.(type)` überprüfen, wenn Sie nicht sicher sind, um welchen Typ es sich handelt:

```
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
}
```

Variablen online lesen: https://riptutorial.com/de/go/topic/674/variablen

Kapitel 66: Vendoring

Bemerkungen

Vendoring ist eine Methode, um sicherzustellen, dass alle in Ihrem Go-Projekt verwendeten Pakete von Drittanbietern für alle, die für Ihre Anwendung entwickeln, konsistent sind.

Wenn Ihr Go-Paket ein anderes Paket importiert, überprüft der Compiler normalerweise `$GOPATH/src/` auf den Pfad des importierten Projekts. Wenn Ihr Paket jedoch einen Ordner mit dem Namen `vendor` enthält, `vendor` der Compiler diesen Ordner *zuerst ein* . Dies bedeutet, dass Sie Pakete anderer Parteien in Ihrem eigenen Code-Repository importieren können, ohne ihren Code ändern zu müssen.

Vendoring ist eine Standardfunktion in Go 1.6 und höher. In Go 1.5 müssen Sie die Umgebungsvariable von `GO15VENDOREXPERIMENT=1` , um das Vendoring zu aktivieren.

Examples

Verwenden Sie govendor, um externe Pakete hinzuzufügen

Govendor ist ein Tool, mit dem Pakete von Drittanbietern auf eine Weise in Ihr Code-Repository importiert werden können, die mit dem Verkauf von golang kompatibel ist.

Sagen Sie zum Beispiel, dass Sie ein Drittanbieter-Paket `bosun.org/slog` :

```
package main

import "bosun.org/slog"

func main() {
    slog.Infof("Hello World")
}
```

Ihre Verzeichnisstruktur könnte folgendermaßen aussehen:

```
$GOPATH/src/
├── github.com/me/helloworld/
|   ├── hello.go
├── bosun.org/slog/
|   ├── ... (slog files)
```

Wer jedoch `github.com/me/helloworld` `$GOPATH/src/bosun.org/slog/` möglicherweise keinen Ordner `$GOPATH/src/bosun.org/slog/` , wodurch *der* Build aufgrund fehlender Pakete fehlschlägt.

Wenn Sie den folgenden Befehl an der Eingabeaufforderung ausführen, werden alle externen Pakete aus Ihrem Go-Paket abgerufen und die erforderlichen Bits in einen Herstellerordner gepackt:

```
govendor add +e
```

Dadurch wird govendor angewiesen, alle externen Pakete zu Ihrem aktuellen Repository hinzuzufügen.

Die Verzeichnisstruktur Ihrer Anwendung würde jetzt so aussehen:

```
$GOPATH/src/
├── github.com/me/helloworld/
|   ├── vendor/
|   |   ├── bosun.org/slog/
|   |   |   ├── ... (slog files)
|   ├── hello.go
```

und diejenigen, die Ihr Repository klonen, erhalten auch die erforderlichen Pakete von Drittanbietern.

Verwalten von ./vendor mithilfe von Papierkorb

`trash` ist ein minimalistisches Verkaufstool, das Sie mit der Datei `vendor.conf` konfigurieren. Dieses Beispiel ist für den `trash` selbst:

```
# 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
```

Die erste Zeile ohne Kommentar ist das Paket, für das wir ./vendor verwalten (Hinweis: Dies kann buchstäblich jedes Paket in Ihrem Projekt sein, nicht nur das Stammpaket).

Kommentierte Zeilen beginnen mit `#` .

Jede nicht leere und nicht kommentierte Zeile enthält eine Abhängigkeit. Es muss nur das "root" - Paket der Abhängigkeit aufgeführt werden.

Nach dem Paketnamen wird die Version (Festschreiben, Tag oder Zweig) und optional die URL des Paket-Repositorys angegeben (standardmäßig wird sie aus dem Paketnamen abgeleitet).

Um Ihr ./vendor-Verzeichnis aufzufüllen, müssen Sie die `vendor.conf` Datei im aktuellen `vendor.conf` haben. `vendor.conf` Sie einfach `vendor.conf` aus:

```
$ trash
```

Der Papierkorb klappt die verkauften Bibliotheken in `~/.trash-cache` (standardmäßig), checkt die

angeforderten Versionen aus, kopiert die Dateien in das `./vendor` und entfernt **nicht importierte Pakete und Testdateien** . Dieser letzte Schritt sorgt dafür, dass Ihr ./vendor lean und mean bleibt und Platz in Ihrem Projekt-Repo einspart.

Hinweis: Ab v0.2.5 steht Papierkorb für Linux und MacOS zur Verfügung und unterstützt nur git zum Abrufen von Paketen, da gits das beliebteste ist. Wir arbeiten jedoch daran, alle anderen `go get` unterstützt werden.

Verwenden Sie golang / dep

golang / dep ist ein Werkzeug zum Management von Prototypabhängigkeiten. Bald ein offizielles Versionierungswerkzeug. Aktueller Status **Alpha** .

Verwendungszweck

Holen Sie sich das Tool über

```
$ go get -u github.com/golang/dep/...
```

Eine typische Verwendung für ein neues Repo könnte sein

```
$ dep init
$ dep ensure -update
```

Um eine Abhängigkeit auf eine neue Version zu aktualisieren, können Sie sie ausführen

```
$ dep ensure github.com/pkg/errors@^0.8.0
```

Beachten Sie, dass die Dateiformate für Manifeste und Sperren **jetzt abgeschlossen wurden** . Diese bleiben auch bei einem Werkzeugwechsel kompatibel.

vendor.json mit dem 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 depenencies in vendor.json
govendor sync
```

Beispiel vendor.json

```
{

"comment": "",
"ignore": "test",
```

```
"package": [
    {
        "checksumSHA1": "kBeNcaKk56FguvPSUCEaH6AxpRc=",
        "path": "github.com/golang/protobuf/proto",
        "revision": "2bba0603135d7d7f5cb73b2125beeda19c09f4ef",
        "revisionTime": "2017-03-31T03:19:02Z"
    },
    {
        "checksumSHA1": "1DRAxdlWzS4U0xKN/yQ/fdNN7f0=",
        "path": "github.com/syndtr/goleveldb/leveldb/errors",
        "revision": "8c81ea47d4c41a385645e133e15510fc6a2a74b4",
        "revisionTime": "2017-04-09T01:48:31Z"
    }
],
"rootPath": "github.com/sample"

}
```

Vendoring online lesen: https://riptutorial.com/de/go/topic/978/vendoring

Kapitel 67: Verschieben

Einführung

Eine `defer` Anweisung drückt einen Funktionsaufruf in eine Liste. Die Liste der gespeicherten Anrufe wird ausgeführt, nachdem die umgebende Funktion zurückgegeben wurde. Defer wird häufig verwendet, um Funktionen zu vereinfachen, die verschiedene Bereinigungsaktionen ausführen.

Syntax

- someFunc (args) verschieben
- defer func () {// code geht hier hin} ()

Bemerkungen

Defer funktioniert, indem ein neuer Stack-Frame (die nach dem Schlüsselwort `defer` aufgerufene Funktion) in den Aufruf-Stack unterhalb der aktuell ausgeführten Funktion eingefügt wird. Dies bedeutet, dass die Verzögerung verzögert wird, solange der Stapel abgewickelt wird (wenn Ihr Programm abstürzt oder einen `SIGKILL` erhält, wird die Verzögerung nicht ausgeführt).

Examples

Defer Grundlagen

Eine *verzögerte Anweisung* in Go ist einfach ein Funktionsaufruf, der zu einem späteren Zeitpunkt ausgeführt wird. Defer-Anweisung ist ein gewöhnlicher Funktionsaufruf, dem das Schlüsselwort `defer` vorangestellt ist.

```
defer someFunction()
```

Eine zurückgestellte Funktion wird ausgeführt, sobald die Funktion, die `defer` Anweisung enthält, zurückgegeben wird. Der tatsächliche Aufruf der verzögerten Funktion erfolgt, wenn die einschließende Funktion:

- führt eine return-Anweisung aus
- fällt vom Ende ab
- Panik

Beispiel:

```
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)
}
```

Ausgabe:

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

Wenn eine Funktion mehrere zurückgestellte Anweisungen hat, bilden sie einen Stapel. Der letzte `defer` ist der erste, der ausgeführt wird, nachdem die einschließende Funktion zurückgegeben wurde, gefolgt von nachfolgenden Aufrufen der vorhergehenden `defer` in der Reihenfolge (das folgende Beispiel führt zu einer Panik):

```
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)
}
```

Ausgabe:

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

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

Beachten Sie, dass die Argumente für verzögerte Funktionen zum Zeitpunkt der Ausführung der `defer` ausgewertet werden:

```
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)
}
```

Ausgabe:

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

Wenn eine Funktion Rückgabewerte benannt hat, kann eine zurückgestellte anonyme Funktion innerhalb dieser Funktion auf den zurückgegebenen Wert zugreifen und ihn aktualisieren, auch nachdem die Funktion zurückgegeben wurde:

```
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
}
```

Eine `defer` ist in der Regel eine Operation, die häufig zusammen vorkommt. Zum Beispiel:

- öffnen und schließen Sie eine Datei
- verbinden und trennen
- sperren und entsperren Sie einen Mutex
- Eine Wartegruppe als erledigt markieren (`defer wg.Done()`)

Diese Verwendung gewährleistet die ordnungsgemäße Freigabe von Systemressourcen unabhängig vom Ausführungsfluss.

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

Aufgeschobene Funktionsaufrufe

Latente Funktionsaufrufe dienen einem ähnlichen Zweck , um Dinge wie `finally` Blöcke in Sprachen wie Java: Sie sorgen dafür , dass eine bestimmte Funktion ausgeführt wird, wenn die äußeren Funktion zurückkehrt, unabhängig davon , ob ein Fehler aufgetreten ist oder welche Anweisung Rückkehr wurde in Fällen mit mehreren Rückkehr getroffen. Dies ist nützlich, um Ressourcen zu bereinigen, die wie Netzwerkverbindungen oder Dateizeiger geschlossen werden

müssen. Das `defer` Schlüsselwort gibt einen latenten Funktionsaufruf, ähnlich wie bei dem `go` Begriff neu goroutine initiieren. Funktionsargumente werden wie ein `go` Aufruf sofort ausgewertet, aber im Gegensatz zu einem `go` Aufruf werden aufgeschobene Funktionen nicht gleichzeitig ausgeführt.

```
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
```

Beachten Sie die Verwendung von `conn.Close()` statt `conn.Close` - Sie sind nicht nur auf der Durch in einer Funktion, werden Sie eine volle *Funktionsaufruf* aufzuschieben, einschließlich seiner Argumente. Mehrere Funktionsaufrufe können in derselben äußeren Funktion zurückgestellt werden und werden jeweils einmal in umgekehrter Reihenfolge ausgeführt. Sie können Schließungen auch verschieben - vergessen Sie nicht die Parens!

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

Verschieben online lesen: https://riptutorial.com/de/go/topic/2795/verschieben

Kapitel 68: Verschlüsse

Examples

Schließungsgrundlagen

Ein *Abschluss* ist eine Funktion, die zusammen mit einer Umgebung genommen wird. Die Funktion ist normalerweise eine anonyme Funktion, die in einer anderen Funktion definiert ist. Die Umgebung ist der lexikalische Geltungsbereich der einschließenden Funktion (eine grundlegende Idee eines lexikalischen Gültigkeitsbereichs einer Funktion wäre der Gültigkeitsbereich, der zwischen den geschweiften Klammern der Funktion besteht.)

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

Innerhalb des Körpers einer anonymen Funktion (etwa `f`), die in einer anderen Funktion (beispielsweise `g`) definiert ist, sind Variablen zugänglich, die in Bereichen von sowohl `f` als auch `g` sind. Es ist jedoch der Bereich von `g` , der den Umgebungsteil der Schließung bildet (Funktionsteil ist `f`). Als Folge davon behalten Änderungen an den Variablen im Bereich von `g` ihre Werte bei (dh die Umgebung bleibt zwischen Aufrufen von `f`) .

Betrachten Sie die folgende Funktion:

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

In der obigen Definition hat `NaturalNumbers` eine innere Funktion `f` die `NaturalNumbers` zurückgibt. Innerhalb von `f` wird auf die im Rahmen von `NaturalNumbers` definierte Variable `i` zugegriffen.

Wir bekommen eine neue Funktion von `NaturalNumbers` wie `NaturalNumbers` :

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

Nun ist `n` eine Schließung. Es ist eine Funktion (definiert durch `f`), der auch eine zugehörige Umgebung (Umfang von `NaturalNumbers`) zugeordnet ist.

Im Fall von `n` enthält der Umgebungsabschnitt nur eine Variable: `i`

Da `n` eine Funktion ist, kann es aufgerufen werden:

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

Wie aus der obigen Ausgabe hervorgeht, erhöht sich jedes Mal, wenn `n` aufgerufen wird, `i` . `i` beginnt bei 0 und jeder Aufruf an `n` führt `i++` .

Der Wert von `i` wird zwischen Aufrufen beibehalten. Das heißt, die Umwelt als Teil der Schließung bleibt bestehen.

`NaturalNumbers` Sie `NaturalNumbers` erneut `NaturalNumbers` , wird eine neue Funktion erstellt und zurückgegeben. Dies würde ein neues `i` in `NaturalNumbers` initialisieren. Das bedeutet, dass die neu zurückgegebene Funktion einen weiteren Abschluss bildet, der den gleichen Teil für die Funktion (noch `f`) aufweist, aber eine völlig neue Umgebung (ein neu initialisiertes `i`).

```
o := NaturalNumbers()

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

Sowohl `n` als auch `o` sind Verschlüsse, die denselben Funktionsteil enthalten (was dasselbe Verhalten bewirkt), aber unterschiedliche Umgebungen. Durch die Verwendung von Verschlüssen können Funktionen auf eine permanente Umgebung zugreifen, in der Informationen zwischen Aufrufen gespeichert werden können.

Ein anderes Beispiel:

```
func multiples(i int) func() int {
    var x int = 0
    return func() int {
        x++
        // paramenter 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
```

Verschlüsse online lesen: https://riptutorial.com/de/go/topic/2741/verschlusse

Kapitel 69: Verzweigung

Examples

Wechseln Sie die Anweisungen

Eine einfache `switch` Anweisung:

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

Das obige Beispiel ist äquivalent zu:

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

Die `default` ist optional und wird nur dann ausgeführt, wenn keine der Fälle wahr ist, auch wenn sie nicht zuletzt `default` wird. Dies ist akzeptabel. Das folgende ist semantisch dasselbe wie beim ersten Beispiel:

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

Dies kann nützlich sein, wenn Sie die `fallthrough` Anweisung in der `default` verwenden `fallthrough` , die die letzte Anweisung in einem Fall sein muss und dazu führt, dass die Programmausführung zum nächsten Fall `fallthrough` :

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

```
case d:
    // do something else
}
```

Ein leerer Schalterausdruck ist implizit `true` :

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

Switch-Anweisungen unterstützen eine einfache Anweisung ähnlich der `if` Anweisung:

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

Fälle können in einer durch Kommas getrennten Liste zusammengefasst werden, wenn sie dieselbe Logik verwenden:

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

Wenn Aussagen

Eine einfache `if` Anweisung:

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

Beachten Sie, dass der Zustand nicht in Klammern steht und dass sich die öffnende geschweifte Klammer `{` in derselben Zeile befinden muss. Folgendes wird *nicht* kompiliert:

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

Eine `if` Anweisung, die `else` :

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

Laut golang.org-Dokumentation "Dem Ausdruck kann eine einfache Anweisung vorangestellt werden, die ausgeführt wird, bevor der Ausdruck ausgewertet wird." In dieser einfachen Anweisung deklarierte Variablen beziehen sich auf die `if` Anweisung und können nicht außerhalb der Anweisung aufgerufen werden:

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

Typwechselanweisungen

Ein einfacher Schalter:

```
// 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
}
```

Sie können für jeden Typ testen, einschließlich `error` , benutzerdefinierte Typen, Schnittstellentypen und Funktionsarten:

```
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!")
    }
}
```

Springen Sie Anweisungen

Eine `goto` Anweisung überträgt die Kontrolle an die Anweisung mit dem entsprechenden Label innerhalb derselben Funktion. Die Ausführung der `goto` Anweisung darf nicht dazu führen, dass Variablen in den Gültigkeitsbereich fallen, die zum Zeitpunkt des `goto` noch nicht im Gültigkeitsbereich waren.

Siehe zum Beispiel den Standard-Quellcode der Bibliothek: 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-Anweisungen

Die break-Anweisung bewirkt bei Ausführung, dass die aktuelle Schleife das Beenden erzwingt

Paket Haupt

```
import "fmt"
```

```
func main() {
    i:=0
    for true {
      if i>2 {
        break
        }
    fmt.Println("Iteration : ",i)
    i++
    }
}
```

Die continue-Anweisung bewegt das Steuerelement bei der Ausführung an den Anfang der Schleife

```
import "fmt"

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

Schleife im Schalter unterbrechen / fortsetzen

```
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)
    }
}
```

Verzweigung online lesen: https://riptutorial.com/de/go/topic/1342/verzweigung

Kapitel 70: Vorlagen

Syntax

- t, err: = template.Parse (`{{.MyName .MyAge}}`)
- t.Execute (os.Stdout, struct {MyValue, MyAge-Zeichenfolge} {"John Doe", "40.1"})

Bemerkungen

Golang bietet Pakete wie:

1. `text/template`
2. `html/template`

datengesteuerte Vorlagen zum Generieren von Text- und HTML-Ausgaben zu implementieren.

Examples

Ausgabe von Werten der Strukturvariablen an die Standardausgabe mithilfe einer Textvorlage

```
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)
    }
}
```

Spielplatz

Definieren von Funktionen zum Aufrufen aus Vorlage

```
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)
    }
}
```

Hier haben wir unsere definierten Funktion `status_code` zu Statuscode der Web - Seite direkt aus einer Vorlage zu erhalten.

Ausgabe:

```
http://godoc.org 200

http://stackoverflow.com 200

http://linux.org 200
```

Vorlagen online lesen: https://riptutorial.com/de/go/topic/1402/vorlagen

Kapitel 71: Wählen Sie und Channels

Einführung

Das Schlüsselwort `select` bietet eine einfache Methode, um mit Kanälen zu arbeiten und erweiterte Aufgaben auszuführen. Es wird häufig für verschiedene Zwecke verwendet: - Timeouts behandeln. - Wenn es mehrere Kanäle gibt, aus denen gelesen werden kann, wird die Auswahl zufällig von einem Kanal gelesen, der Daten enthält. - Eine einfache Möglichkeit zu definieren, was passiert, wenn auf einem Kanal keine Daten verfügbar sind.

Syntax

- wählen {}
- Wählen Sie {case true:}
- select {case incomingData: = <-someChannel:}
- Wählen Sie {default:}

Examples

Einfach auswählen Mit Kanälen arbeiten

In diesem Beispiel erstellen wir eine Goroutine (eine Funktion, die in einem separaten Thread ausgeführt wird), die einen `chan` Parameter akzeptiert und einfach Schleifen durchführt, wobei jedes Mal Informationen in den Kanal gesendet werden.

In der `main` wir eine `for` Schleife und eine `select` . Die `select` blockiert die Verarbeitung, bis eine der `case` Anweisungen wahr ist. Hier haben wir zwei Fälle erklärt; Der erste ist, wenn Informationen durch den Kanal kommen, und der andere ist, wenn kein anderer Fall auftritt, was als `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)
        }
    }
}
```

Versuchen Sie es auf dem Go Playground!

Select mit Timeouts verwenden

Hier habe ich die `for` Schleifen entfernt und ein **Timeout vorgenommen,** indem der `select` einen zweiten `case` hinzugefügt hat, der nach 3 Sekunden zurückkehrt. Da die `select` nur wartet, bis JEDER Fall erfüllt ist, wird der zweite `case` ausgelöst, und dann endet unser Skript, und `chatter()` erhält niemals eine Chance, es zu beenden.

```
// 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!")
    }
}
```

Wählen Sie und Channels online lesen: https://riptutorial.com/de/go/topic/3539/wahlen-sie-und-channels

Kapitel 72: XML

Bemerkungen

Während viele Verwendungen des `encoding/xml` Pakets das Marshallen und das Aufheben des Marshalls für eine Go- `struct` , ist zu beachten, dass dies keine direkte Zuordnung ist. In der Paketdokumentation heißt es:

> Die Zuordnung zwischen XML-Elementen und Datenstrukturen ist inhärent fehlerhaft: Ein XML-Element ist eine auftragsabhängige Sammlung anonymer Werte, während eine Datenstruktur eine auftragsunabhängige Sammlung von benannten Werten ist.

Bei einfachen, ungeordneten Schlüsselwertpaaren ist die Verwendung einer anderen Kodierung wie Gob's oder JSON möglicherweise besser geeignet. Für geordnete Daten- oder Ereignis- / Rückruf-basierte Datenströme ist XML die beste Wahl.

Examples

Grundlegende Dekodierung / Aufhebung der Verschachtelung verschachtelter Elemente mit Daten

XML-Elemente verschachteln sich häufig, haben Daten in Attributen und / oder als Zeichendaten. Die Erfassung dieser Daten erfolgt durch Verwendung von `,attr` bzw. `,chardata` für diese Fälle.

```
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

XML online lesen: https://riptutorial.com/de/go/topic/1846/xml

Kapitel 73: YAML

Examples

Erstellen einer Konfigurationsdatei im 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
```

YAML online lesen: https://riptutorial.com/de/go/topic/2503/yaml

Kapitel 74: Zeiger

Syntax

- Zeiger: = & variable // Zeiger aus Variable holen
- variable: = * pointer // Variable vom Zeiger abrufen
- * pointer = value // Wert von Variable durch den Zeiger setzen
- pointer: = new (Struct) // Zeiger der neuen Struktur abrufen

Examples

Grundlegende Zeiger

Go unterstützt Zeiger , mit denen Sie Verweise auf Werte und Datensätze in Ihrem Programm übergeben können.

```
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
}
```

Sobald diese Funktionen definiert sind, können Sie Folgendes tun:

```
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
}
```

Versuchen Sie diesen Code

Zeiger v. Wertmethoden

Zeiger-Methoden

Zeigermethoden können auch aufgerufen werden, wenn die Variable selbst kein Zeiger ist.

Nach der Go-Spezifikation

> . . . Ein Verweis auf eine Methode ohne Schnittstelle, bei der ein `t.Mp` einen adressierbaren Wert verwendet, übernimmt automatisch die Adresse dieses Werts: `t.Mp` entspricht `(&t).Mp` .

Sie können dies in diesem Beispiel sehen:

```
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)
}
```

Spiel es

Wertmethoden

Ähnlich wie bei Pointer-Methoden können Value-Methoden aufgerufen werden, auch wenn die Variable selbst kein Wert ist.

Nach der Go-Spezifikation

> . . . Ein Verweis auf eine Methode ohne Schnittstelle, bei der ein `pt.Mv` einen Zeiger verwendet, dereferenziert diesen Zeiger automatisch: `pt.Mv` entspricht `(*pt).Mv` .

Sie können dies in diesem Beispiel sehen:

```
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)
}
```

Spiel es

Weitere Informationen zu Pointer- und Value-Methoden finden Sie im Abschnitt "Go-Spezifikation" unter "Methodenwerte" oder im Abschnitt "Effective Go" über "Pointer vs. Values" .

Hinweis 1: Die Klammern (`()`) um `*p` und `&f` vor Selektoren wie `.Bar` dienen zur Gruppierung und müssen beibehalten werden.

Hinweis 2: Obwohl Zeiger in Werte umgewandelt werden können (und umgekehrt), wenn sie die Empfänger einer Methode sind, werden sie nicht *automatisch in einander konvertiert, wenn sie*

Argumente innerhalb einer Funktion sind.

Dereferenzierungszeiger

Zeiger können durch Hinzufügen eines Sterns * vor einem Zeiger **dereferenziert** werden.

```
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 sind Zeiger auf Arraysegmente

Slices sind **Zeiger** auf Arrays mit der Länge des Segments und seiner Kapazität. Sie verhalten sich als Zeiger und weisen ihren Wert einem anderen Slice zu. Sie weisen die Speicheradresse zu. Um ein Stück Wert auf einen anderen **zu kopieren,** verwenden Sie die integrierten **Kopierfunktion:** `func copy(dst, src []Type) int` (liefert die Menge der kopierten Elemente).

```
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
}
```

Einfache Zeiger

```
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)
}
```

Zeiger online lesen: https://riptutorial.com/de/go/topic/1239/zeiger

Kapitel 75: Zeit

Einführung

Das Go- `time` Paket bietet Funktionen zum Messen und Anzeigen von Zeit.

Dieses Paket enthält eine Struktur `time.Time` , die das Speichern und Berechnen von Datums- und Zeitangaben ermöglicht.

Syntax

- time.Date (2016, time.Dezember 31, 23, 59, 59, 999, time.UTC) // initialisieren
- date1 == date2 // gibt `true` zurück `true` wenn die 2 den gleichen Moment haben
- date1! = date2 // gibt `true` zurück `true` wenn die 2 unterschiedliche Momente sind
- date1.Before (date2) // gibt `true` zurück `true` wenn das erste streng vor dem zweiten ist
- date1.Nach (date2) // gibt `true` zurück `true` wenn der erste streng nach dem zweiten ist

Examples

Return time.Time Zero Wert, wenn die Funktion einen Fehler aufweist

```
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
}
```

Zeitanalyse

Wenn Sie ein Datum als Zeichenfolge gespeichert haben, müssen Sie es analysieren. Verwenden Sie `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
```

Der erste Parameter ist das Layout, in dem die Zeichenfolge das Datum speichert, und der zweite Parameter ist die Zeichenfolge, die das Datum enthält. `01/02/2006` ist das gleiche als zu sagen, dass das Format `MM/DD/YYYY` .

Das Layout definiert das Format, indem es zeigt, wie die Referenzzeit (`Mon Jan 2 15:04:05 -0700 MST 2006` interpretiert würde, wenn dies der Wert wäre. Es dient als Beispiel für das Eingabeformat. Die gleiche Interpretation wird dann für die Eingabezeichenfolge vorgenommen.

Sie können die im Zeitpaket definierten Konstanten sehen, um zu wissen, wie die Layoutzeichenfolge geschrieben wird. Beachten Sie jedoch, dass die Konstanten nicht exportiert werden und nicht außerhalb des Zeitpakets verwendet werden können.

```
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"
)
```

Zeit vergleichen

Manchmal müssen Sie mit 2 Datumsobjekten wissen, ob es sich um dasselbe Datum handelt oder welches Datum hinter dem anderen liegt.

In **Go** gibt es vier Möglichkeiten, Daten zu vergleichen:

- `date1 == date2` gibt `true` zurück `true` wenn die 2 den gleichen Moment haben
- `date1 != date2` , gibt `true` zurück `true` wenn die 2 unterschiedliche Momente sind

- `date1.Before(date2)` gibt `true` zurück `true` wenn das erste streng vor dem zweiten ist
- `date1.After(date2)` gibt `true` zurück `true` wenn der erste streng nach dem zweiten ist

  WARNUNG: Wenn die 2 Zeit zum Vergleichen gleich sind (oder dem gleichen Datum entsprechen), werden die Funktionen `After` und `Before` als `false` , da ein Datum weder vor noch hinter sich liegt

  - `date1 == date1` gibt `true`
  - `date1 != date1` , gibt `false`
  - `date1.After(date1)` gibt `false`
  - `date1.Before(date1)` gibt `false`

  TIPPS: Wenn Sie wissen müssen, ob ein Datum vor oder einem anderen Datum liegt, müssen Sie nur die 4 Operatoren kombinieren

  - `date1 == date2 && date1.After(date2)` wird `true` wenn Datum1 nach oder gleich Datum2 ist
    oder mit `! (date1.Before(date2))`
  - `date1 == date2 && date1.Before(date2)` , gibt `true` zurück `true` wenn date1 vor oder gleich date2 oder mit `!(date1.After(date2))`

Einige Beispiele zur Verwendung:

```
// 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
```

Zeit online lesen: https://riptutorial.com/de/go/topic/8860/zeit

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit 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	Analysieren von Befehlszeilenargumenten und Flags	Ingve, Pavel Kazhevets, Sam Whited
3	Analysieren von CSV-Dateien	Ainar-G
4	Arbeiterpools	burfl, photoionized, seriousdev
5	Arrays	NatNgs, nouney, Noval Agung Prayogo, Sam Whited
6	Base64-Kodierung	Nathan Osman, RamenChef, Sam Whited
7	Befehle ausführen	Krzysztof Kowalczyk, Kyle Brandt, Nevermore
8	Best Practices zur Projektstruktur	Iman Tumorang
9	Bilder	putu
10	cgo	MaC, Vojtech Kane
11	Channels	Chris Lucas, Howl, Jeremy, Kwarrtz, metmirr, RamenChef, Rodolfo Carvalho, Zoyd
12	Constraints erstellen	4444, RamenChef, Sam Whited, seriousdev
13	Datei I / O	1lann, Andres Kütt, greatwolf, Grzegorz Żur, koblas, noisewaterphd, Quentin Skousen, Sam Whited
14	Der Go-Befehl	ganesh kumar, Harshal Sheth, Ingve, Lanzafame, Mayank

		Patel, Nevermore, Quentin Skousen, Sam Whited, theflametrooper, Vikash
15	E-Mails senden / empfangen	Utahcon
16	Entwickeln für mehrere Plattformen mit bedingtem Kompilieren	ecem
17	Erste Schritte mit Go Atom verwenden	Ali M, Danny Chen, Katamaritaco
18	Fehlerbehandlung	bowsersenior, elevine, Elijah Sarver, Florian Hämmerle, groob, Ingve, Joe, Kin, Paul Hankin, Quentin Skousen, Sam Whited, Simone Carletti, Sridhar, Surreal Dreams, Vervious, Zoyd
19	Fmt	Lanzafame, Nevermore, Sam Whited
20	Funktionen	Boris Le Méec, Dmytro Sadovnychyi, Grzegorz Żur, jayantS , LeoTao, Nathan Osman, nouney, palestamp, RamenChef, Right leg, Thomas Gerot
21	gob	zola
22	Goroutinen	mohan08p
23	HTTP-Client	1lann, dmportella, Lanzafame, Sam Whited, SommerEngineering
24	HTTP-Server	Chief, frigo americain, Jon Erickson, Kin, Nathan Osman, rogerdpack, Sam Whited, Sascha, seriousdev, Simone Carletti, SommerEngineering, Tanmay Garg, Zhinkk
25	Inline-Erweiterung	Sam Whited
26	Installation	sadlil
27	Jota	4444, Florian Hämmerle, Ingve, mohan08p, Sam Whited, Wojciech Kazior, Zoyd
28	JSON	Dmitry Udod, Joe, Jon Chan, Kyle Brandt, Nathan Osman, RamenChef, Sam Whited, shayan, Simone Carletti, sztanpet, Tanmay Garg, Utahcon
29	JWT-Autorisierung in Go	AniSkywalker
30	Karten	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
31	Konsolen-E / A	Abhilekh Singh
32	Konstanten	Pavel Nikolov, RamenChef, Sam Whited, Simone Carletti
33	Kontext	Ingaz, Sam Whited
34	Kreuzzusammenstellung	Jordan, Katamaritaco, mbb, mohan08p, RamenChef, Riley Guerin, SH', Siu Ching Pong -Asuka Kenji-, SommerEngineering, sztanpet, Zoyd
35	Kryptographie	SommerEngineering
36	Leser	Mike Houston
37	Methoden	ganesh kumar, Pavel Kazhevets
38	mgo	Florian Hämmerle, Sourabh
39	Middleware	Ankit Deshpande
40	Mutex	Adrian, Prutswonder
41	Nullwerte	Harshal Sheth, raulsntos, Surreal Dreams
42	Objekt orientierte Programmierung	Davyd Dzhahaiev, Sam Whited, zola
43	OS-Signale	Community, Sam Whited, Utahcon
44	Pakete	dmportella, Grzegorz Żur, icza, Michael, Nathan Osman, RadicalFish, RamenChef, skunkmb, tkausl
45	Panik und Genesung	JunLe Meng, Kaedys, Kristoffer Sall-Storgaard, Sam Whited
46	Parallelität	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
47	Plugin	Sam Whited
48	Profilieren mit go tool pprof	mbb, Nevermore, radbrawler
49	Protobuf in Go	mohan08p

50	Protokollierung	Grzegorz Żur, Jon Chan, Nathan Osman, Pavel Kazhevets, Sam Whited
51	Reflexion	ganesh kumar, mammothbane, radbrawler
52	Scheiben	1lann, Benjamin Kadish, burfl, cizixs, Grzegorz Żur, Guillaume, Jared Hooper, Joost, Jukurrpa, Kyle Brandt, Mark Henderson, NatNgs, RamenChef, Simone Carletti, skunkmb, Tanmay Garg, Zoyd
53	Schleifen	1lann, burfl, Community, ivan73, jayantS, Jon Chan, mgh, MohamedAlaa, RamenChef, Sam Whited, Steven Maude, Thomas Gerot
54	Schnittstellen	Cody Roseborough, dotctor, Francis Norton, Grzegorz Żur, icza, Ingve, meysam, Mike, ptman, sadlil, Sam Whited, Wendy Adi
55	Speicher-Pooling	Elijah Sarver, Grzegorz Żur, Kenny Grant
56	SQL	Adrian, artamonovdev, bernardn, Francesco Pasa, Nevermore, Sam Whited, Sascha, Tanmay Garg, wrfly
57	String	Ainar-G, NatNgs, raulsntos
58	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
59	Testen	Adrian, Ankit Deshpande, Harshal Sheth, ivan.sim, Jared Ririe, Nathan Osman, Omid, Pavel Nikolov, Rodolfo Carvalho, seriousdev, Toni Villena, Zoyd
60	Text + HTML-Vorlage	Stephen Rudolph
61	Typumwandlungen	Adrian, Florian Hämmerle
62	Variablen	Community, FredMaggiowski, Jon Chan, Simone Carletti
63	Vendoring	Abhilekh Singh, Boris Le Méec, burfl, Dmitri Goldring, Ivan Mikushin, Mark Henderson, Martin Campbell, Michael, Sam Whited, Vardius
64	Verschieben	abhink, Adrian, Sam Whited, Vikash
65	Verschlüsse	abhink

66	Verzweigung	burfl, Community, ganesh kumar, Ingve, nk2ge5k
67	Vorlagen	Pavel Kazhevets, RamenChef, Tanmay Garg
68	Wählen Sie und Channels	Harshal Sheth, Kaedys, RamenChef, Sam Whited, Utahcon
69	XML	ivarg, Sam Whited
70	YAML	Nathan Osman, Orr, Sam Whited
71	Zeiger	David Hoelzer, Jon Chan, Joost, Mal Curtis, metmirr, Nevermore, skunkmb
72	Zeit	Lanzafame, NatNgs, raulsntos